

第一章 程序基础

1.1 什么是 .Net Framework

学习 C# 之前,要先大体上知道什么是 .NET。 .NET 的全名是 “.NET Framework” (dot net framework [dɒt] [net] [ˈ freimwɜ:k] 网络架构), 大体上讲, 它是一种技术平台。微软的网站上这么描写的: .NET Framework 是一个集成在 Windows 中的组件,它支持生成和运行下一代应用程序与 XML Web Services。 .NET Framework 旨在实现下列目标:

- 提供一个一致的面向对象的编程环境,而无论对象代码是在本地存储和执行,还是在本地执行但在 Internet 上分布,或者是在远程执行。
- 提供一个将软件部署和版本控制冲突最小化的代码执行环境。
- 提供一个可提高代码(包括由未知的或不完全受信任的第三方创建的代码)执行安全性的代码执行环境。
- 提供一个可消除脚本环境或解释环境的性能问题的代码执行环境。
- 使开发人员的经验在面对类型大不相同的应用程序(如基于 Windows 的应用程序和基于 Web 的应用程序)时保持一致。
- 按照工业标准生成所有通信,以确保基于 .NET Framework 的代码可与任何其他代码集成。

既然 .NET 是一个环境,那么它包含什么呢? 它包含两个方面的内容,一是公共语言运行时(Common Language Runtime);二是 .NET Framework 类库。公共语言运行时(简写: CLR)是 .NET Framework 的基础,一切 .NET Framework 的程序都运行在以它为基础的平台。公共语言运行时有点类似于 JAVA 中的虚拟机,它提供内存管理、线程管理和远程处理等核心服务,并且还强制实施严格的类型安全以及可提高安全性和可靠性的其他形式的代码准确性。这对于初学者可能有点难以理解,比如说,在 C++ 中,类型检查、内存越界、内存释放等都需要程序员进行把握,以免出现错误。但是在 .NET Framework 中,这些都由公共语言运行时(CLR)来完成,大大简化了编程的复杂度。

.NET Framework 类库是一个综合性的面向对象的 reusable 类型集合,可以使用它开发多种应用程序。这就是面向对象语言的代码重用性特点。在类库中包括比如通用基础类、集合类、线程及同步类、XML 类等等。

.NET Framework 类库可开发的程序种类有传统的命令行或图形用户界面(GUI)应用程序,也包括基于 ASP.NET 所提供的最新创新的应用程序(如 Web 窗体和 XML Web Services)。

对于 .NET 架构的概念,初学者不用深究,随着编程学习的深入,就会慢慢了解。毕竟,写出可以用的程序才是第一重要的。

1.2 什么是 C# 语言

C# 读作 C Sharp([si:] [ʃɑ:p]),是微软公司在 2000 年 7 月发布的一种全新且简单、安全、面向对象的程序设计语言,是专门为 .NET 的应用而开发的语言。它吸收了 C++、Visual Basic、Delphi、Java 等语言的优点,C# 继承了 C 语言的语法风格,同时又继承了 C++ 的面向对象特性。不同的是,C# 的对象模型已经面向 Internet 进行了重新设计,使用的是 .NET 框架的类库;C# 不再提供对指针类型的支持,使得程序不能随便访问内存地址空间,从而更加健壮;C# 不再支持多重继承,避免了以往类层次结构中由于多重继承带来的可怕后果。.NET 框架为 C# 提供了一个强大的、易用的、逻辑结构一致的程序设计环境。同时,公共语言运行时(CLR)为 C# 程序语言提供了一个托管的运行环境,使程序比以往更加稳定、安全。

C# 对 C++ 进行了多处改进,主要区别如下:

- 编译目标: C++ 代码直接编译为本地可执行代码,而 C# 默认编译为中间语言(IL)代码,执行时再通过实时(Just-In-Time)编译器,将需要的模块临时编译成本地代码。
- 内存管理: C++ 需要显式地删除动态分配给堆的内存,而 C# 不需要这么做,C# 采用垃圾回收机制自动在合适的时机回收不再使用的内存。

- 指针：C++中大量地使用指针，而C#使用对类实例的引用，如果确实想在C#中使用指针，必须声明该内容是非安全的。不过，一般情况下C#中没有必要使用指针。

- 字符串处理：在C#中，字符串是作为一种基本数据类型来对待的，因此比C++中对字符串的处理要简单得多。

- 库：C++依赖于以继承和模板为基础的标准库，C#则依赖于.NET 基库。

- C++允许类的多继承，而C#只允许类的单继承，而通过接口实现多继承。

在后面的学习中会发现，C#与C++相比还有很多不同和改进之处，包括一些细节上的差别，这里就不一一列举了。

C#与Java 的比较，主要有以下特点：

- C#面向对象的程度比Java 高。
- C#中的基本类型都是面向对象的。
- C#具有比Java 更强大的功能。
- C#语言的执行速度比Java 快。

其实，C#只是.NET 平台上的其中一种语言，在.NET 平台上还有许多种语言可以用，比如VB.NET、F#、Powershell 等。不过，微软把主要的精力都放在了C#上，据说，微软的许多产品已经用C#语言进行了重写。使用C#就可以开发上面提到的种种类型的程序。所以C#确实很强大。

1.3 托管代码与非托管代码

托管与非托管这些名词也是由微软提出的，其实也很简单，比如说传统的C++程序，编译完成后，直接生成二进制文件，然后直接由操作系统调用，加载到内存中执行。这样的程序就是非托管程序，而托管程序就像C#一样，它编译后生成的不是直接的二进制文件，而是称作CIL（CIL，Common Intermediate Language 公共中间语言）的中间代码，CIL 代码由CLR 管理执行。这和JAVA 有点像，只不过JAVA 是解释执行的，而C#是先编译再执行，所以效率更高一些。.NET Framework 可由非托管组件承载，这些组件将公共语言运行时加载到它们的进程中并启动托管代码的执行，从而创建一个可以同时利用托管和非托管功能的软件环境。具体说来，

1. 托管程序所申请的资源统一由CLR 管理，非托管代码所申请的内存等资源则需要程序员主动去释放。

2. 非托管程序是二进制代码，运行速度快；托管程序非二进制代码，速度差的很多，资源占用很多。

3. 非托管程序一般都是在对操作系统进行直接或者间接的操作，托管程序是需要通过访问公共语言运行时（CLR）才能访问操作系统的资源，而非托管程序不用通过访问公共语言运行时（CLR），可以直接访问操作系统的资源。

1.4 C#与.NET Framework 的关系

前面提到过，C#是.NET Framework 平台上的其中一种开发语言，而.NET Framework 为它上面的各种开发语言，提供了一个统一的开发环境，换句话说，虽然各种语言的语法不同，但是实现一个具体功能调用的函数在底层是一个。这样一来，在托管程序和操作系统之间就有了一个中间层。这样做的好处是，程序员写出的托管程序是可以很容易移植的，托管程序的代码不用改变，只要在其上运行的操作系统具有.NET Framework 类库就可以了。这样，不管是Windows 还是Linux，或者是其他环境，只要支持了.NET Framework，那么就可以不用修改托管程序代码，而直接到其平台上运行。.NET Framework 的作用是屏蔽掉了不同操作系统之间底层的差异。

.NET Framework 不只是建立了操作系统平台的无关性，它还建立了在其上开发程序的语言之间的无关性，比如说，用两种语言写程序，这两个项目组之间虽然使用了不同的语言，但是他们之间的类是可以互相调用的。这就大大减少了开发成本，提高了开发程序的方便性。

1.5 程序结构

在C#语言中，对于语言的组织结构来说，有如下几个概念，分别是程序、命名空间、类型、成员和程序集。

首先，C#程序由若干个.cs 源文件组成，这是物理上的组织结构，在逻辑上，用命名空间进行组织，在命名空间中可以定义类型，类型包括类和接口。在类型中定义成员，成员包括字段、属性、方法以及事件等。最后，它们都被编译打包成物理上的程序集进行分发。程序集根据使用目的不同，有两种，应用程序和类库。扩展名分别是.exe 和.dll。

下面来阐述一下什么是程序集，程序集在官方的定义足够复杂，如果照本宣科，那么还是不能够让人明白。这里通过程序集的生成过程来讲解一下什么是程序集。Visual Studio 编译生成的.exe 或者.dll 文件是托管代码，它的运行必须依赖于 CLR（公共语言运行时）和.Net 基础类库。所以说，Visual Studio 编译生成的.exe 或者.dll 文件是和托管代码的.exe 或者.dll 文件不同的。比如用 C++ Builder 生成的.exe 程序包含的就是机器码，它不依赖于CLR 这样的虚拟机组件而能够直接在操作系统上运行。而Visual Studio 编译器在编译过程中将把源代码编译为CIL（公共中间语言）代码，为何C#写的代码能够跨平台，就是因为编译的 CIL 代码是和平台无关的。那么，和平台有关的是什么呢？就是 CLR，它的平台相关性决定了 CIL 代码的平台无关性。当最后生成.exe 或者.dll 文件时，它里面包含的就是 CIL 代码再加上元数据。这就是一个单文件程序集。

这里又涉及到了一个名词：元数据。官方的资料将程序集定义为自描述的，意思是什么呢？就是指的功能，元数据详尽描述了代码中每个类的特征，比如这个类继承自哪个类，实现了哪个接口，以及类的成员等。同时，程序集本身也会用元数据来描述，这类元数据也叫做“清单”，它描述了程序集的当前版本、文化信息（用于本地化字符串和图像资源）以及引用了哪些外部的程序集。元数据由编译器编译时自动生成。

程序集可以分为单文件程序集和多文件程序集，当生成一个.exe 或者.dll 文件时，可以认为这就是一个单文件程序集，它包含 CIL、元数据和清单。多文件程序集则由多个模块组成，其中一个模块为主模块，它包含程序集清单和 CIL 及类的元数据。其他模块则包含一个模块级别的程序集清单、CIL 和类的元数据。那么显而易见，当执行的时候，肯定是先加载主模块，然后根据清单调用其他模块。多文件程序集的模块以.netmodule 作为扩展名，它的最大的作用就是将程序集分拆为更小的单元，便于网络应用。

程序集在运行过程中由 JIT（just-in-time，即时编译）将 CIL 编译为二进制代码执行。但是在执行程序集的时候，并不是一次性将全部代码编译成二进制执行，而是将第一次必须执行的进行即时编译。然后根据需要，比如调用哪部分功能，再进行即时编译，但是直到程序运行完毕，已经编译过的代码不再进行编译。所以当重复调用的时候，就可以加快速度。因此在C#程序第一次启动的时候，会感觉比较慢。但是当重复调用的时候就会快很多。多文件程序集则根据代码调用的需要，实时从网络下载必要的模块。

因为程序集是一个自描述的功能单元，在它的里面包含了代码和元数据，因此C#不需要C++那样的include 指令和头文件，如果要在程序中使用某个程序集中的类和成员，只需要引用该程序集即可。

1.6 Hello World

本小节从一个最简单的在控制台屏幕上打印一行文字的程序开始介绍C#语法。代码如下：

```
01 using System;
02 namespace Hello_World
03 {
04     class Program
05     {
06         static void Main()
07         {
08             Console.WriteLine("Hello World");
09             Console.ReadKey();
10             return;
11         }
12     }
```

```
13 }
```

这个项目在 Visual Studio 中建立的时候需要选择控制台项目类型，因为它是在控制台窗口中进行显示输出。这段代码编辑完成后，是储存在一个.cs 源码文件中。

首先第一行代码是一个 using 指令，它的作用是引用 System 命名空间，这行语句以分号结束。为何要引用 System 命名空间呢？因为下面的第 8 行和第 9 行代码使用了 .Net Framework 类库中的类。如果不使用 using 指令，那么第 8 行和第 9 行就要使用完全限定名称来使用类库中的类，使用的语法是这样的：`System.Console.WriteLine("Hello World");`这里出现了一个命名空间的概念，在 C# 中，使用命名空间在逻辑上以分层的方式组织程序和库。当然，微软提供的 .Net Framework 类库也是以命名空间来进行逻辑上的组织。在命名空间中还可以包含其他命名空间和类。`System.Console.WriteLine("Hello World");`这行语句就是调用了 System 命名空间中的 Console 类的一个静态方法，方法名称是 WriteLine。正因为使用了 using 语句，所以才可以用非完全限定名称的方式调用 WriteLine 方法。因此，using 语句在这里的作用是减少了代码输入的敲击次数。

第 2 行代码使用 namespace 关键字定义了一个命名空间，这样，新建立的类就都位于这个命名空间中，新定义命名空间的作用就是防止定义的类和其他类重名。命名空间的名称后面以一对大括号开始和结束，不用分号结尾。在大括号中可以再定义命名空间和类。

第 4 行代码使用 class 关键字定义了一个类，类是对世界上实际事物的抽象，在 class 关键字后是类的名称，然后以一对大括号开始和结束，在大括号中是类体，类的定义也不以分号结束。

第 6 行代码定义了一个方法，这个方法的名称是 Main，注意第一个字母 M 需要大写，C# 是大小写敏感的语言，也就是说，大写的 M 和小写的 m 在 C# 中是不同的。Main 方法是一个应用程序的入口点方法，程序开始执行时，需要从这个方法开始。在方法名称前有两个关键字修饰，一个是 void，它紧挨着方法名，它表示这个方法没有返回值。另一个是 static，它表示这个方法是静态的方法，静态的方法意思是这个方法属于类，调用的时候不用定义一个类的对象来调用它，而是直接使用类名.方法名的方式来调用。因为在程序刚开始执行时，无法创建任何类的对象，所以需要静态 Main 方法来作为入口点方法。方法也以一对大括号开始和结束，不用分号结尾。在大括号中是方法体。

第 8 行和第 9 行调用了类库中的两个方法，WriteLine 方法向控制台窗口打印一行文字，文字是以参数的方式传递给 WriteLine 方法，ReadKey 方法是读取一个键盘按键，它的作用是避免当程序执行完毕时，控制台窗口马上消失。这样就不好观察程序的执行结果。调用方法时，每行语句都需要以分号结尾。

第 10 行是一个 return 语句，它的作用是向调用方返回方法的返回值，因为这个方法使用了 void 关键字修饰，是没有返回值的。所以 return 关键字后没有跟参数。

编译这段代码，执行结果如下：

```
Hello World
```

1.7 简单类型

简单类型是 C# 提供的预定义结构类型集，也就是说，简单类型本质上都是结构体。简单类型通过 C# 的关键字标识。例如 int、bool、byte 等。那么 C# 的关键字有哪些呢？C# 的所有关键字如表 1-1 所示。

表 1-1 C# 关键字

abstract	as	base	bool
break	byte	case	catch
char	checked	class	const
continue	decimal	default	delegate
do	double	else	enum
event	explicit	extern	False

finally	static	float	for
foreach	goto	if	implicit
in	in	int	interface
internal	is	lock	long
namespace	new	null	object
operator	out	out	override
params	private	protected	public
readonly	ref	return	sbyte
sealed	short	sizeof	stackalloc
static	string	struct	switch
this	throw	True	try
typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual
void	volatile	while	

在表格中包含了用于定义简单类型的关键字。除了关键字，在定义一个简单类型的时候，还需要使用名称标识符确定类型的名称，表格中列出的这些关键字是不允许用作名称标识符的。

1.7.1 标识符

什么是标识符呢？例如上一小节中的类的名称就是一个标识符。可以作为变量标识符开头第一位的只有大小写字母和下划线，其余字符都不可以打头。标识符中只可以有字母、数字和下划线，不可以包括其它字符。如果要将关键字用作标识符，可以使用@符号作为前缀，例如：@int。但通常不将关键字用作标识符。除了这些关键字以外，C#中还有一些上下文关键字，上下文关键字如表 1-2 所示。

表 1-2 上下文关键字

关键字	说明
add	定义一个自定义事件访问器，客户端代码订阅事件时将调用该访问器。
dynamic	定义一个引用类型，实现发生绕过编译时类型检查的操作。
get	为属性或索引器定义访问器方法。
global	指定未以其他方式命名的默认全局命名空间。
Partial	在整个同一编译单元内定义分部类、结构和接口。
remove	定义一个自定义事件访问器，客户端代码取消订阅事件时将调用该访问器。
set	为属性或索引器定义访问器方法。
value	用于设置访问器和添加或移除事件处理程序。

var	使编译器能够确定在方法作用域中声明的变量的类型。
where	将约束添加到泛型声明。
yield	在迭代器块中使用，用于向枚举数对象返回值或发信号结束迭代。

另外，在查询表达式中也有一些上下文关键字，这些关键字尽量都不要用作标识符。查询表达式中的上下文关键字如表 1-3 所示。

表 1-3 查询表达式中的上下文关键字

子句	说明
from	指定数据源和范围变量（类似于迭代变量）。
where	根据一个或多个由逻辑“与”和逻辑“或”运算符（&& 或 ）分隔的布尔表达式筛选源元素。
select	指定当执行查询时返回的序列中的元素将具有的类型和形式。
group	按照指定的键值对查询结果进行分组。
into	提供一个标识符，它可以充当对 join、group 或 select 子句的结果的引用。
orderby	基于元素类型的默认比较器按升序或降序对查询结果进行排序。
join	基于两个指定匹配条件之间的相等比较来联接两个数据源。
let	引入一个用于存储查询表达式中的子表达式结果的范围变量。
in	join 子句中的上下文关键字。
on	join 子句中的上下文关键字。
equals	join 子句中的上下文关键字。
by	group 子句中的上下文关键字。
ascending	orderby 子句中的上下文关键字。
descending	orderby 子句中的上下文关键字。

C#的简单类型是类库中预定义结构类型的别名，和 C++中的单一简单类型不同，C#的简单类型其实是一种结构。别名和结构类型名称的对应关系如表 1-4 所示。

表 1-4 别名及结构类型名称对应表

保留字	对应的类型
sbyte	System.SByte
byte	System.Byte
short	System.Int16
ushort	System.UInt16
int	System.Int32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64

char	System.Char
float	System.Single
double	System.Double
bool	System.Boolean
decimal	System.Decimal

表 1-4 中的简单类型根据表示数据的类型还可以进一步细分为下面几种：

● 整型

整型共有 9 种，sbyte、byte、short、ushort、int、uint、long、ulong 和 char。它们的大小和取值范围是：

1. sbyte（有符号字节型）类型表示有符号 8 位整数，其值介于 -128 和 127 之间。
2. byte（无符号字节型）类型表示无符号 8 位整数，其值介于 0 和 255 之间。
3. short（短整型）类型表示有符号 16 位整数，其值介于 -32768 和 32767 之间。
4. ushort（无符号短整型）类型表示无符号 16 位整数，其值介于 0 和 65535 之间。
5. int（整型）类型表示有符号 32 位整数，其值介于 -2147483648 和 2147483647 之间。
6. uint（无符号整型）类型表示无符号 32 位整数，其值介于 0 和 4294967295 之间。
7. long（长整型）类型表示有符号 64 位整数，其值介于 -9223372036854775808 和 9223372036854775807 之间。
8. ulong（无符号长整型）类型表示无符号 64 位整数，其值介于 0 和 18446744073709551615 之间。
9. char（字符型）类型表示无符号 16 位整数，其值介于 0 和 65535 之间。char 类型的可能值集与 Unicode 字符集相对应。虽然 char 的表示形式与 ushort 相同，但是可以对一种类型进行的所有计算并非都可以对另一种类型执行。

● 浮点型

C#中的浮点型分为 float（单精度浮点型）和 double（双精度浮点型）。它们分别用 32 位单精度和 64 位双精度 IEEE 754 格式来表示，这些格式提供以下几组值：

1. 正零和负零。大多数情况下，正零和负零的行为与简单的值零相同，但某些运算会区别对待此两种零。
2. 正无穷大和负无穷大。无穷大是由非零数字被零除这样的运算产生的。例如， $1.0 / 0.0$ 产生正无穷大，而 $-1.0 / 0.0$ 产生负无穷大。
3. 非数字(Not-a-Number)值，常缩写为 NaN。NaN 是由无效的浮点运算（如零被零除）产生的。
4. 以 $s \times m \times 2^e$ 形式表示的非零值的有限集，其中 s 为 1 或 -1，m 和 e 由特殊的浮点类型确定：
对于 float，为 $0 < m < 224$ 并且 $-149 \leq e \leq 104$ ；对于 double，为 $0 < m < 253$ 并且 $-1075 \leq e \leq 970$ 。非标准化的浮点数被视为有效非零值。

float 类型可表示精度为 7 位、在大约 1.5×10^{-45} 到 3.4×10^{38} 的范围内的值。double 类型可表示精度为 15-16 位、在大约 1.5×10^{-324} 到 1.7×10^{308} 的范围内的值。

● decimal 类型

decimal 类型是 128 位的数据类型，适合用于财务计算和货币计算。decimal 类型可以表示具有 28 或 29 个有效数字、从 1.0×10^{-28} 到大约 7.9×10^{28} 范围内的值。

● bool 类型

bool 类型表示布尔逻辑量。bool 类型的可能值为 true 和 false。

1.8 文本

文本是一个值的源代码表现形式。举个例子，定义一个整型的变量，`int a = 33;`，这里 a 是变量的标识符，等号是赋值运算符，它表示将 33 赋值给整型变量 a。这个语句中的 33 就是文本。

1.8.1 变量

介绍文本的时候就不能不涉及变量的概念。变量表示一个存储位置，C#是一种类型安全的语言，每种变量都有一个明确的类型。在使用变量之前，变量必须已经明确赋值，否则编译器会报错。这一点和 C++ 语言不同。变量可以通过运算符改变变量的值或为其赋值。在代码中定义变量的时候，变量有两种状态，初始未赋值或初始已赋值。下面看一段代码：

```
01 using System;
02 namespace Simple_Type
03 {
04     class Program
05     {
06         static void Main()
07         {
08             int a = 1;
09             int b;
10             a = a + 1;
11             Console.WriteLine(a);
12             Console.ReadKey();
13         }
14     }
15 }
```

在代码中的第 8 行定义了一个整型的变量 a，并且在定义的同时将 1 赋值给它。a 就是一个初始已赋值的变量。

第 9 行定义了一个整型变量 b，在定义的同时没有赋值操作，它就是一个初始未赋值的变量。

第 10 行使用算术运算符“+”将 a 和 1 相加，然后再次将结果赋值给了 a。

第 11 行打印 a 的值。

如果在代码中不打印 b 的值，那么编译器会给出一个警告，如下：

声明了变量“b”，但从未使用过

如果在代码中像打印 a 的值一样打印 b 的值，也就是未赋值就使用它，编译器就会报错。如下：

使用了未赋值的局部变量“b”

需要注意，这是一个错误，而不是一个警告了。变量的使用规则是，在从变量定义到使用它的路径上必须对它有赋值的操作。在这段代码中的第 10 行就是做的这一操作。

变量的类型有多种，下面介绍第一种变量类型，局部变量。局部变量定义在方法中。现在介绍方法体中的局部变量，局部变量可以位于方法体中、方法体中的代码块、方法体中的 for 语句、switch 语句、if 语句、以及 using 语句中。那么，什么是块呢？块由一对大括号构成。例如类的声明和方法体的声明，都是以大括号开始和结束。在块中可以写多条语句，如果在块中没有语句，则称块是空的。局部变量定义后是有生存期的，或称它有作用域。在块中定义局部变量时，这个局部变量的生存期就是位于这个块，代码的执行控制从左大括号开始，然后依次执行块中的每条语句，最后控制到达右大括号的结束点。当代码的执行控制到达块的结束点后，标志着块中的局部变量的生存期结束，块外的语句将不能访问这个局部变量。换句话说，局部变量的作用域就是包含它的块，在包含它的块外没有权限可以访问这个局部变量。下面看代码：

```
01 using System;
02 namespace Simple_Type
03 {
04     class Program
05     {
```



```

06      static void Main()
07      {
08          int a = 1;
09          {
10              int b = 1;
11              Console.WriteLine(a);
12              Console.WriteLine(b);
13          }
14          Console.WriteLine(a);
15          Console.WriteLine(b);
16          Console.ReadKey();
17      }
18  }
19 }

```

这段代码中的 Main 方法体就是一个代码块。在其中，第 9 行到第 13 行又定义了一个代码块，在其中包含了三条语句。程序的流程是这样的，首先程序控制进入 Main 方法的方法体代码块，第 10 行定义了一个局部变量 a，并且为其赋值为 1。接下来，控制进入第 9 行的代码块，定义一个局部变量 b，为其赋值为 1。第 11 行和第 12 行代码打印 a 和 b 的值。为什么能打印 a 的值呢？因为 Main 方法的方法体代码块从第 7 行开始，到第 17 行结束。第 9 行开始的代码块包含在 Main 方法的方法体代码块中。所以 a 的生存期到第 17 行结束。在第 9 行的代码块中可以访问到变量 a。这样，第 11 行和第 12 行的代码访问 a 和 b 都是允许的。下面看第 14 行和第 15 行代码，第 14 行代码再次打印 a 的值，这没有问题，因为没有超出 a 的作用域。但是，第 15 行代码打印 b 的值就会出错，因为 b 的生存期到第 13 行就结束了。在第 15 行将不再能访问到局部变量 b。这时编译器会报错，如下：

当前上下文中不存在名称“b”

1.8.2 整型的文本表示

介绍完局部变量后，下面再来介绍整型的文本表示。整型的文本表示有两种形式，十进制和十六进制。代码实例如下：

```

01 using System;
02 namespace Simple_Type
03 {
04     class Program
05     {
06         static void Main()
07         {
08             int a = 1;
09             int b = 0x1AFE;
10             int c = 0X1afe;
11             Console.WriteLine(a);
12             Console.WriteLine(b);
13             Console.WriteLine(c);
14             Console.ReadKey();
15         }
16     }
17 }

```

第 9 行和第 10 行代码定义了两个十六进制的代码，十六进制的文本表示是以 0x 开头，x 可以大写也可以小写，十六进制的文本同样不区分大小写。这可以通过运行结果来验证，代码执行结果如下：

```
1
6910
6910
```

因为使用了不带格式设置的 WriteLine 方法，所以输出都被转化为十进制。

在源代码中输入一个整型的文本表示时，这个输入文本的类型按如下顺序进行确定，int、uint、long、ulong。也就是说，按照文本表示数的大小范围，如果 int 类型能表示它，它就是 int 类型。如果 int 的类型无法表示它，而 uint 类型的大小范围可以表示它，那么它就是 uint 类型，以此类推。看下面的代码：

```
01 using System;
02 namespace Simple_Type
03 {
04     class Program
05     {
06         static void Main()
07         {
08             var a = 32766;
09             var b = 18446744073709551614;
10             Type t1 = a.GetType();
11             Type t2 = b.GetType();
12             Console.WriteLine(t1.ToString());
13             Console.WriteLine(t2.ToString());
14             Console.ReadKey();
15         }
16     }
17 }
```

在代码的第 8 行和第 9 行定义了两个 var 类型的变量，这个类型是推断类型。编译器会根据为其赋值的类型推断出这个变量的实际类型。第 10 行和第 11 行定义了两个 Type 类型的变量，目前先不用了解它们的意思，只知道这两行代码的目的就是要得到变量 a 和 b 的实际类型即可。在第 12 行和第 13 行打印 a 和 b 的类型，执行结果如下：

```
System.Int32
System.UInt64
```

可以看到，根据文本的取值范围，a 被推断为 int 类型，而 b 则被推断为 ulong 类型。

在进行文本表示时，还可以通过字母后缀进一步表示其类型，在 C# 中可以使用的字母后缀有 U、L 和 UL。U 和 L 是不区分大小写的，它和 u、l 是一样的。同样，UL 也可以写做 ul，甚至还可以写成 lu、LU。它们表示的意思都是一样的，但是为了和数字 1 进行区分，便于识别，通常将它们大写。

当使用 U 后缀时，它属于以下所列的类型中第一个能够表示其值的那个类型：uint、ulong。

当使用 L 后缀时，它属于以下所列的类型中第一个能够表示其值的那个类型：long、ulong。

当使用 UL 后缀时，就更明确了，它就属于 ulong 类型。

如果文本表示的值超出了 ulong 能表示的范围，将发生编译时错误。下面看代码实例：

```
01 using System;
02 namespace Simple_Type
03 {
04     class Program
```

```

05  {
06      static void Main()
07      {
08          var a = 18446744073709551614U;
09          var b = 9223372036854775806L;
10          var c = 18446744073709551616;
11          Type t1 = a.GetType();
12          Type t2 = b.GetType();
13          Console.WriteLine(t1.ToString());
14          Console.WriteLine(t2.ToString());
15          Console.ReadKey();
16      }
17  }
18 }

```

在这段代码中，变量 a 和 b 都使用了字母后缀，而变量 c 的取值超出了 ulong 类型能表示的范围，如果不定义 c，那么代码的输出结果如下：

```
System.UInt64
```

```
System.Int64
```

而当定义了变量 c 之后，编译器将会报错，如下：

```
整数常量太大
```

1.8.3 浮点型的文本表示

实数文本用来表示 float、double 和 decimal 类型的值。实数文本也有其字母后缀表示具体的类型。如果文本没有使用字母后缀，则其为 double 类型。代码如下：

```

01 using System;
02 namespace Simple_Type
03 {
04     class Program
05     {
06         static void Main()
07         {
08             var a = 3.14;
09             var b = 1.0/0.0;
10             var c = 0.0 / 0.0;
11             var d = 0.0 / 0.0 + 0.1;
12             Type t1 = a.GetType();
13             Console.WriteLine(t1.ToString());
14             Console.WriteLine(b);
15             Console.WriteLine(c);
16             Console.WriteLine(d);
17             Console.ReadKey();
18         }
19     }
20 }

```

第 8 行代码输入的文本没有使用字母后缀，它的类型默认将是 double 类型。第 9 行代码的除法运算将产生

正无穷大的结果。第 10 行代码的运算将产生非数字的结果。第 11 行代码的除法运算将产生非数字结果，将其与一个 double 类型的数字相加，结果还是非数字。代码执行结果如下：

```
System.Double
正无穷大
非数字
非数字
```

对于浮点数的运算不会产生异常，但是会产生 0、无穷大或非数字的结果。

- 如果浮点运算的结果对于目标格式太小，则运算结果变成正零或负零。
- 如果浮点运算的结果对于目标格式太大，则运算结果变成正无穷大或负无穷大。
- 如果浮点运算无效，则运算的结果变成非数字。
- 如果参与浮点运算的操作数有非数字，则运算的结果变成非数字。

实数文本用来表示 float 和 double 类型的值时，可以使用字母后缀。以字母 f 或 F 结尾的文本表示 float 类型；以字母 d 或 D 结尾的文本表示 double 类型。代码如下：

```
01 using System;
02 namespace Simple_Type
03 {
04     class Program
05     {
06         static void Main()
07         {
08             var a = 3.14F;
09             var b = 3.14D;
10             Type t1 = a.GetType();
11             Type t2 = b.GetType();
12             Console.WriteLine(t1.ToString());
13             Console.WriteLine(t2.ToString());
14             Console.ReadKey();
15         }
16     }
17 }
```

这段代码中的第 8 行和第 9 行分别使用了字母后缀确定文本的类型，然后打印 a 和 b 的类型，输出结果如下：

```
System.Single
System.Double
```

1.8.4 decimal 类型的文本表示

如果文本以 m 或 M 结尾，则这个文本表示一个 decimal 类型的值。代码实例如下：

```
01 using System;
02 namespace Simple_Type
03 {
04     class Program
05     {
06         static void Main()
07         {
08             var a = 3.14M;
```

```
09         Type t1 = a.GetType();
10         Console.WriteLine(t1.ToString());
11         Console.ReadKey();
12     }
13 }
14 }
```

这段代码比较简单，不做过多介绍，代码的执行结果如下：

```
System.Decimal
```

1.8.5 char 类型的文本表示

char 类型的文本由一个置于一对单引号中的字符组成，或者由转义序列组成。代码如下：

```
01 using System;
02 namespace Simple_Type
03 {
04     class Program
05     {
06         static void Main()
07         {
08             var a = 'a';
09             var b = '\x00cc';
10             var c = '\u0062';
11             var d = '\a';
12             Console.WriteLine(a);
13             Console.WriteLine(b);
14             Console.WriteLine(c);
15             Console.WriteLine(d);
16             Console.ReadKey();
17         }
18     }
19 }
```

这段代码的执行结果如下：

```
a
i
b
```

在控制台窗口上输出三个字符以后，计算机将发出“嘟”的一声警告。代码第 8 行是最为普通的字符型文本定义方法，由一个置于一对单引号中的字符组成。第 9 行是字符文本的十六进制转义序列表示方式，它以“\x”开始，后跟四个十六进制数。这里的 x 必须小写，否则会发生编译时错误。第 10 行是 Unicode 转义序列形式，它以“\u”后跟四位十六进制数的形式表示。这里的 u 也必须小写，十六进制转义序列和 Unicode 转义序列的取值范围都是 0000–FFFF。超出这个范围，编译器会报错。十六进制的形式既可以是大写也可以是小写。第 11 行是简单转义序列表示方式，它实际上也是表示的一个 Unicode 字符编码。简单转义序列如表 1-5 所示。

表 1-5 简单转义序列表

转义序列	字符	Unicode 十六进制编码
\'	单引号	0x0027
\"	双引号	0x0022

\\	反斜杠	0x005C
\0	null	0x0000
\a	响铃警告	0x0007
\b	Backspace	0x0008
\f	换页符	0x000C
\n	换行符	0x000A
\r	回车	0x000D
\t	水平制表符	0x0009
\v	垂直制表符	0x000B

还需要注意，以 Unicode 转义序列方式定义文本的时候，转义序列不进行多次转换。看下面的代码：

```

01 using System;
02 namespace Simple_Type
03 {
04     class Program
05     {
06         static void Main()
07         {
08             var a = "\u005cu005c";
09             Console.WriteLine(a);
10             Console.ReadKey();
11         }
12     }
13 }

```

第 8 行代码中的 \u005c 只执行一次转换，并且十六进制数要满足 4 位。其后的所有位数不再进行转换（在这里，文本的类型是 string 类型，这个类型是引用类型，后面会讲到）。因此执行结果如下：

```
\u005c
```

1.8.6 bool 类型的文本表示

表示 bool 类型的文本一共有两个，true 和 false。与 C++ 语言不同，在 C# 中，0 不能表示 false，非 0 值不能表示 true。代码如下：

```

01 using System;
02 namespace Simple_Type
03 {
04     class Program
05     {
06         static void Main()
07         {
08             var a = true;
09             var b = false;
10             Type t1 = a.GetType();
11             Type t2 = b.GetType();
12             Console.WriteLine(t1);
13             Console.WriteLine(t2);
14             Console.ReadKey();
15         }

```

```
16     }
17 }
```

这段代码的执行结果如下：

```
System.Boolean
System.Boolean
```

1.9 变量类型的确切指定

前面的代码中都是以文本来确定变量类型，如果要确切地控制变量的类型，可以在变量名称标识符前指定变量的类型。代码如下：

```
01 using System;
02 namespace Note
03 {
04     class Program
05     {
06         static void Main()
07         {
08             byte a = 12;
09             ushort b = 65534;
10             short c = 32766;
11             float d = 3.14F;
12             double e = 3.14D;
13             Console.WriteLine(a);
14             Console.WriteLine(b);
15             Console.WriteLine(c);
16             Console.WriteLine(d);
17             Console.WriteLine(e);
18             Console.ReadKey();
19         }
20     }
21 }
```

第 8 行至第 12 行代码以确切指定变量类型的方式定义了若干变量。使用这种方式应该了解各种变量的取值范围。因为 C# 是一种类型安全的语言，它的编译器会在编译时检查变量，如果超出了取值范围，编译器会给出提示，以便程序员进行处理或进行强制转换。在定义变量的时候也可以使用字母后缀，以第 9 行代码为例，当为 ushort 类型的变量 b 赋值 65534 时，因为文本没有超出 ushort 类型的范围，所以不会有问題。但是如果赋值方法如下：

```
ushort f = 65534U;
```

则编译器会给出一个错误提示，如下：

无法将类型“uint”隐式转换为“ushort”。存在一个显式转换(是否缺少强制转换?)

这是因为使用字母后缀 U 的时候，文本首先会按照 uint、ulong 的次序判断类型，因为 uint 符合范围，所以这个文本被认为是 uint 类型，将它赋值给一个 ushort 类型就会造成类型不匹配，编译器会提示进行类型转换。上面代码实例的执行结果如下：

```
12
65534
32766
3.14
```

3.14

确切类型的指定和 var 类型不同。var 类型是需要一个初始化式存在的，编译器依靠初始化式推断出变量的具体类型。具体类型推断出以后，它就遵循 C# 的类型安全规则，是一种类型安全的变量。在定义 var 类型变量时，也有一些规则需要遵守，后面会介绍到。

1.10 注释

注释的作用是对源代码的一种文字说明。但是它本身不参与编译，只对程序作者或程序员起到提示和代码解释的作用。尤其是在规模比较大的程序中，注释是必不可少的。C# 支持三种形式的注释，单行注释、多行注释和 XML 风格的注释。代码实例如下：

```

01  using System;
02  namespace Note
03  {
04      class Program
05      {
06          static void Main(string[] args)
07          {
08              /* 此处是一个多行注释，
09              下面开始定义若干变量 */
10              byte a = 12; //定义一个 byte 类型的变量
11              ushort b = 65534;
12              short c = 32766;
13              float d = 3.14F;
14              double e = 3.14D;
15              Console.WriteLine(a);
16              Console.WriteLine(b);
17              Console.WriteLine(c);
18              Console.WriteLine(d);
19              Console.WriteLine(e);
20              Show(3, 5);
21              Console.ReadKey();
22          }
23          /// <summary>
24          /// 打印两个整型值的和
25          /// </summary>
26          /// <param name="a">参与运算的第一个整型数</param>
27          /// <param name="b">参与运算的第二个整型数</param>
28          static void Show(int a, int b)
29          {
30              int c = a + b;
31              Console.WriteLine(c);
32          }
33      }
34  }

```

首先来看单行注释，第 10 行有一个单行注释，单行注释是以两个左斜线开始，注释的内容只能在一行内，不能换行。第 8 行开始是一个多行注释，多行注释以 “/*” 开始，以 “*/” 结束。多行注释可以跨行。在

第 28 行代码，定义了一个方法。这里先不必理会方法的定义方法，后面会逐步介绍到。这里只看 C# 的第三种 XML 风格的注释。XML 风格的注释用来生成代码的文档，可以使用第三方的文档生成器来生成。使用 XML 风格注释的一个好处是，在代码中调用带注释的成员时，开发环境会有相关的智能提示，方便代码的编写。XML 风格的注释，以三个左斜杠开始，当在开发环境中输入三个左斜杠时，开发环境会自动生成注释的格式。在这段代码中，需要在两个 summary 标记中间输入方法的说明，在 param 标记中间的是方法的参数说明。输入完毕后，如果在代码中调用这个方法，就会有相关的智能提示，如图 1-1 所示。

```
Console.WriteLine(e);  
Show(  
    void Program.Show(int a, int b)  
    }  
    /// 打印两个整型值的和  
    /// a: 参与运算的第一个整型数  
    /// </summary>  
    /// <param name="a">参与运算的第一个整型数</param>  
    /// <param name="b">参与运算的第二个整型数</param>  
    static void Show(int a, int b)  
    {  
        int c = a + b;  
        Console.WriteLine(c);  
    }  
}
```

图 1-1 智能提示

智能提示是很有作用的，尤其是当编写了多个类库的时候，它可以让程序员很方便的了解相关成员的作用。

第二章 基本运算符

2.1 算术运算符

算术运算符包括+、-、*、/和%。它们的作用分别是加、减、乘、除和求余数。下面看代码实例：

```
01 using System;
02 namespace Operator_Show
03 {
04     class Program
05     {
06         static void Main()
07         {
08             int a = 3 + 2;
09             int b = a - 1;
10             int c = b * 4;
11             int d = c / 8;
12             int e = 16 % 7;
13             Console.WriteLine(a);
14             Console.WriteLine(b);
15             Console.WriteLine(c);
16             Console.WriteLine(d);
17             Console.WriteLine(e);
18             Console.ReadKey();
19         }
20     }
21 }
```

代码的第8行到第12行演示了这五种算术运算符的使用方法。可以看到，这五种运算符都有两个操作数进行参与计算，这样的运算符叫做二元运算符。这段代码的执行结果如下：

```
5
4
16
2
2
```

2.2 移位运算符

移位运算符包括“<<”和“>>”运算符，它们分别是左移位运算符和右移位运算符，它们都是二元运算符。下面先看代码实例：

```
01 using System;
02 namespace Operator_Show
03 {
04     class Program
05     {
06         static void Main()
07         {
08             int a = 2147483647;
09             int b = 0x135b;
```

```

10         int c = -3976;
11         //调用类库中的 Convert 类的 ToString 方法输出转换后的二进制数
12         Console.WriteLine("初始 a 的二进制值: ");
13         Console.WriteLine(Convert.ToString(a, 2));
14         Console.WriteLine("初始 b 的二进制值: ");
15         Console.WriteLine(Convert.ToString(b, 2));
16         Console.WriteLine("初始 c 的二进制值: ");
17         Console.WriteLine(Convert.ToString(c, 2));
18         a = a << 3;
19         b = b >> 3;
20         c = c >> 3;
21         Console.WriteLine("移位后 a 的值: ");
22         Console.WriteLine(a);
23         Console.WriteLine("移位后 a 的二进制值: ");
24         Console.WriteLine(Convert.ToString(a, 2));
25         Console.WriteLine("移位后 b 的二进制值: ");
26         Console.WriteLine(Convert.ToString(b, 2));
27         Console.WriteLine("移位后 c 的二进制值: ");
28         Console.WriteLine(Convert.ToString(c, 2));
29         Console.ReadKey();
30     }
31 }
32 }

```

这段代码的执行结果如下:

```

初始 a 的二进制值:
11111111111111111111111111111111
初始 b 的二进制值:
1001101011011
初始 c 的二进制值:
111111111111111111111111000001111000
移位后 a 的值:
-8
移位后 a 的二进制值:
111111111111111111111111111111000
移位后 b 的二进制值:
1001101011
移位后 c 的二进制值:
11111111111111111111111111000001111

```

在这段代码中,第 8 行代码定义了一个最大值的 int 类型,它的二进制值是 31 位的二进制 1。前面介绍过, int 类型是 32 位的,为何它的最大值是 31 位的呢?这是因为它是有符号类型,它的最高位是符号位。如果是正值,则最高位是 0;如果是负值,最高位是 1。第 9 行代码定义了一个 int 类型的值,它是使用十六进制格式定义的。第 10 行定义了一个负数,当打印 c 的二进制数时,可以看到它的最高位是 1。

打印 int 类型的值的二进制数时,使用了 Convert 类的 ToString 方法。

第 18 行将 a 左移 3 位,然后又赋值给了 a。第 19 行将 b 右移 3 位,然后又赋值给了 b。第 20 行将 c

右移 3 位，然后又赋值给了 c。

接下来，第 22 行打印 a 的值，这时可以发现，a 的值变为了负数。这是为什么呢？当对一个数左移时，对于超出类型范围的高序位，将被放弃掉。而左移后，空出来的低序位，将用 0 填补。因为 int 类型的最大值，符号位始终占着一位。初始值是 31 位的，并不是 32 位的。所以初始值的第三位并没有丢失掉，它移动后变成了最高位。而它的前两位被丢弃了。因为第三位是 1，所以移动后的最高位符号位就是 1。它就变成了一个负数。需要注意的是，二进制值并不能直接转化成十进制值，int 类型的负数最小值，是最高位为 1，其余 31 位都为 0 的二进制数。

第 19 行将 b 右移了 3 位，当初始值为 int 或 long 类型时，放弃初始值的低序位，将剩余的位向右移。如果初始值非负值，则将高序位置设置为零。当 b 向右移动 3 位时，它的低序位 011 就被舍弃掉了，因为它是一个正数，所以最高位设置为 0。

第 20 行将 c 也右移了 3 位，因为 c 是一个负值，所以右移后，它的高序位被设置为 1。

当初始值为 uint 或 ulong 类型时，丢弃初始值右移位数的低序位后，将高序位位置设置为零。

2.3 比较运算符

比较运算符包括==、!=、<、>、<=、和>=，这 6 种比较运算符都会返回一个 bool 类型的值。代码实例如下：

```
01 using System;
02 namespace Operator_Show
03 {
04     class Program
05     {
06         static void Main()
07         {
08             bool a;
09             int b = 3;
10             int c = 2;
11             a = b == c;
12             Console.WriteLine(a);
13             a = b != c;
14             Console.WriteLine(a);
15             a = b < c;
16             Console.WriteLine(a);
17             a = b > c;
18             Console.WriteLine(a);
19             a = b <= c;
20             Console.WriteLine(a);
21             a = 15 >= 15;
22             Console.WriteLine(a);
23             Console.ReadKey();
24         }
25     }
26 }
```

这段代码首先定义了一个 bool 类型的变量 a 用来接收比较运算返回的结果。第 9 行代码和第 10 行代码定义了两个整型变量，并做了初始化。第 11 行代码比较 b 和 c 是否相等，因为比较运算符的优先级高于赋值运算符，所以代码先计算 b 和 c 是否相等，然后将返回值赋值给 a。第 12 行打印返回值结果，因为 b 和 c

不相等，所以它将返回 false。第 13 行判断 b 是否不等于 c，它将返回 true。第 15 行判断 b 是否小于 c，它将返回 false。第 17 行判断 b 是否大于 c，它将返回 true。第 19 行判断 b 是否小于或等于 c，它将返回 false。第 21 行判断整型文本 15 是否大于或等于 15，它将返回 true。这段代码的执行结果如下：

```
False
True
False
True
False
True
```

对于整型数来说，比较运算比较简单。但是对于浮点类型来说，比较运算就相对比较复杂一些。看下面的代码实例：

```
01  using System;
02  namespace Operator_Show
03  {
04      class Program
05      {
06          static void Main()
07          {
08              bool a;
09              float b = 3.14F;
10              float c = 3.15F;
11              a = b == c;
12              Console.WriteLine(a);
13              a = b != c;
14              Console.WriteLine(a);
15              a = b < c;
16              Console.WriteLine(a);
17              a = b > c;
18              Console.WriteLine(a);
19              a = b <= c;
20              Console.WriteLine(a);
21              a = -0.0F == 0.0F;
22              Console.WriteLine(a);
23              a = b > 0.0F / 0.0F;
24              Console.WriteLine(a);
25              a = b < 0.0F / 0.0F;
26              Console.WriteLine(a);
27              a = b != 0.0F / 0.0F;
28              Console.WriteLine(a);
29              a = 0.0F / 0.0F == 0.0F / 0.0F;
30              Console.WriteLine(a);
31              a = 0.0F / 0.0F != 0.0F / 0.0F;
32              Console.WriteLine(a);
33              a = c < 1.0F / 0.0F;
```

```
34         Console.WriteLine(a);
35         Console.ReadKey();
36     }
37 }
38 }
```

代码中的第 11 行到第 20 行的比较都比较好理解，第 21 行比较-0.0 和+0.0，它们是相等的。第 23 行定义了一个非数字。浮点类型和一个非数字相比较，除了“!=”外，都返回 false。第 27 行代码就是比较的浮点数和非数字是否不相等，它将返回 true。第 29 行比较两个非数字是否相等，它将返回 false。第 31 行比较两个非数字是否不相等，它将返回 true。第 33 行代码比较一个浮点数和正无穷大的大小，它将返回 true。在浮点类型的比较运算当中，负无穷大最小，正无穷大最大。而负无穷大等于其它负无穷大，正无穷大等于其它正无穷大。这段代码的执行结果如下：

```
False
True
True
False
True
True
False
False
True
False
True
True
```

虽然很少遇到，但是还是需要介绍一下 bool 类型的比较运算，首先看下面的代码：

```
01 using System;
02 namespace Operator_Show
03 {
04     class Program
05     {
06         static void Main()
07         {
08             bool a = true;
09             bool b = false;
10             bool c = true;
11             bool d = false;
12             Console.WriteLine(a == b);
13             Console.WriteLine(a != b);
14             Console.WriteLine(a == c);
15             Console.WriteLine(a != c);
16             Console.WriteLine(b == d);
17             Console.ReadKey();
18         }
19     }
20 }
```

对于 bool 类型来说，只能比较是否相等。代码第 8 行到第 11 行定义了四个 bool 类型的变量，代码第 12 行比较 a 和 b 是否相等，然后将返回值通过调用 WriteLine 方法打印出来。因为 a 和 b 一个为 true，另一个为 false，所以它将返回 false。第 13 行将返回 true。第 14 行比较两个 true 值是否相等，它将返回 true。第 15 行将返回 false。第 16 行比较两个 false 值是否相等，它将返回 true。代码的执行结果如下：

```
False
True
True
False
True
```

2.4 逻辑运算符

逻辑运算符包括&、|和^，它们的名称分别是与、或和异或。它们的使用看下面的代码：

```
01 using System;
02 namespace Operator_Show
03 {
04     class Program
05     {
06         static void Main()
07         {
08             int a = 3789;
09             Console.WriteLine("a 的二进制值是: ");
10             Console.WriteLine(Convert.ToString(a, 2));
11             int b = 237;
12             Console.WriteLine("b 的二进制值是: ");
13             Console.WriteLine(Convert.ToString(b, 2));
14             int c = 0;
15             c = a & b;
16             Console.WriteLine("c 的二进制值是: ");
17             Console.WriteLine(Convert.ToString(c, 2));
18             c = a | b;
19             Console.WriteLine("c 的二进制值是: ");
20             Console.WriteLine(Convert.ToString(c, 2));
21             c = a ^ b;
22             Console.WriteLine("c 的二进制值是: ");
23             Console.WriteLine(Convert.ToString(c, 2));
24             Console.ReadKey();
25         }
26     }
27 }
```

这三个逻辑运算符都是二元运算符，它们是对两个整型数进行相与、相或、相异或操作。代码的执行结果如下：

```
a 的二进制值是:
111011001101
b 的二进制值是:
11101101
```

c 的二进制值是:

11001101

c 的二进制值是:

111011101101

c 的二进制值是:

111000100000

Bool 类型的数也能进行&、|和^运算,代码如下:

```
01 using System;
02 namespace Operator_Show
03 {
04     class Program
05     {
06         static void Main()
07         {
08             bool a = true;
09             bool b = false;
10             bool c = true;
11             bool d = false;
12             bool e;
13             Console.WriteLine("a、b 之间的操作: ");
14             e = a & b;
15             Console.WriteLine(e);
16             e = a | b;
17             Console.WriteLine(e);
18             e = a ^ b;
19             Console.WriteLine(e);
20             Console.WriteLine("a、c 之间的操作: ");
21             e = a & c;
22             Console.WriteLine(e);
23             e = a | c;
24             Console.WriteLine(e);
25             e = a ^ c;
26             Console.WriteLine(e);
27             Console.WriteLine("b、d 之间的操作: ");
28             e = b & d;
29             Console.WriteLine(e);
30             e = b | d;
31             Console.WriteLine(e);
32             e = b ^ d;
33             Console.WriteLine(e);
34             Console.ReadKey();
35         }
36     }
37 }
```


首先看这段代码的执行结果：

a、b 之间的操作：

False

True

True

a、c 之间的操作：

True

True

False

b、d 之间的操作：

False

False

False

代码的第 8 行到第 12 行定义了五个 bool 类型的变量。第 14 行计算 $a \& b$ 的值，并将结果赋值给变量 e。因为 a 为 true，b 为 false，所以它们相与的结果是 false。第 16 行 $a | b$ 的结果为 true。第 18 行 $a \wedge b$ 的结果为 true。当两个参与逻辑运算的操作数是 bool 类型时，异或运算的结果与 $!=$ 比较运算符产生的结果是一样的。第 21 行是计算 $a \& c$ 的结果，因为 a 和 c 都为 true，所以 $a \& c$ 为 true。 $a | c$ 的结果为 true。 $a \wedge c$ 的结果为 false。第 28 行开始计算 $b \& d$ 的结果，因为 b 和 d 都为 false，所以 $b \& d$ 的结果是 false。 $b | d$ 的结果是 false。 $b \wedge d$ 的结果是 false。

2.5 条件逻辑运算符

条件逻辑运算符只有两个， $\&\&$ 和 $||$ 。可以将它们理解成并且和或者。代码实例如下：

```
01 using System;
02 namespace Operator_Show
03 {
04     class Program
05     {
06         static void Main()
07         {
08             bool a = true;
09             bool b = false;
10             bool c = true;
11             bool d = false;
12             bool e;
13             Console.WriteLine("a、b 之间的操作：");
14             e = a && b;
15             Console.WriteLine(e);
16             e = a || b;
17             Console.WriteLine(e);
18             Console.WriteLine("a、c 之间的操作：");
19             e = a && c;
20             Console.WriteLine(e);
21             e = a || c;
22             Console.WriteLine(e);
23             Console.WriteLine("b、d 之间的操作：");
```

```

24         e = b && d;
25         Console.WriteLine(e);
26         e = b || d;
27         Console.WriteLine(e);
28         Console.ReadKey();
29     }
30 }
31 }

```

这段代码的执行结果如下：

a、b 之间的操作：

False

True

a、c 之间的操作：

True

True

b、d 之间的操作：

False

False

和前面的实例一样，这段代码也先定义了五个 bool 类型的变量，当然实际使用的时候，前四个参与运算的变量那里可能是一个比较运算表达式。因为表达式语句还没介绍到，所以这里就用一个 bool 类型的变量代替表达式的结果。&&运算符表示参与运算的两个操作数同时为 true，则表达式返回 true，否则返回 false。因为 a 为 true，b 为 false，所以 a&&b 返回 false。||运算符表示，如果两个参与运算的操作数有一个为 true，则表达式返回 true。所以 a||b 的结果为 true。

因为 a 和 c 都为 true，所以 a&&c 为 true，a||c 为 true。

因为 b 和 d 都为 false，所以 b&&d 为 false，b||d 也为 false。

2.6 条件运算符

条件运算符也叫三元运算符，因为参与运算的操作数有三个，它的形式是 a?b:c。意思是如果 a 为 true，则返回 b，否则返回 c。这里的 a、b 和 c 也可以是单独的表达式。下面看代码实例：

```

01 using System;
02 namespace Operator_Show
03 {
04     class Program
05     {
06         static void Main()
07         {
08             //最基本的形式如下
09             bool a = true;
10             bool b = false;
11             bool c = true;
12             bool d;
13             d = a ? b : c;
14             Console.WriteLine(d);
15             a = false;
16             d = a ? b : c;

```

```

17         Console.WriteLine(d);
18         //稍复杂的，带有表达式的形式如下
19         int e = 0;
20         e = (3 > 2) ? (15 - 3) : (15 + 3);
21         Console.WriteLine(e);
22         //再复杂一些的表达式形式如下
23         e = (3 > 4) ? (15 - 2) : (b ? (4 + 3) : (4 - 3));
24         Console.WriteLine(e);
25         Console.ReadKey();
26     }
27 }
28 }

```

第 13 行代码是最基本的条件运算符表达式，首先判断 a 是否为 true，如果为 true，则返回 b 的值，否则返回 c 的值。因为代码中 a 为 true，所以返回 b 的值是 false。

代码的第 15 行将 a 设置为 false，再次进行判断，这时将返回 c 的值 true。

第 19 行开始是稍稍复杂一些的表达式形式了，首先定义一个整型变量 e，并且初始化为 0，它用来接收返回的值。第 20 行开始，首先判断 3 是否大于 2，如果为 true，则返回 15-3 的结果，否则返回 15+3 的结果。因为 3>2 为 true，所以返回 12。

第 23 行是更复杂一些的表达式，它在三元运算符表达式中又嵌套了一个三元运算符表达式。它的判断顺序是这样的，先判断 3>4 的结果，如果为 true，则返回 15-2 的结果，否则对 b ? (4 + 3) : (4 - 3) 表达式进行判断，先判断 b 是否为 true，如果为 true，则返回 4+3 的结果，否则返回 4-3 的结果，因为 b 为 false，所以返回 4-3 的结果 1。代码的执行结果如下：

```

False
True
12
1

```

2.7 赋值运算符

赋值运算符的作用就是为可赋值的元素赋新值，前面已经接触到了为变量赋值的赋值运算符“=”，它又被称为简单赋值运算符，它将参与运算的右操作数的值赋值给左操作数给出的可赋值元素。其它赋值运算符还包括+=、-=、*=、/=、%=、&=、|=、^=、<<=和>>=，它们又称为复合赋值运算符，因为在赋值之前它们都先经过了一步运算，然后再赋值。实例代码如下：

```

01 using System;
02 namespace Operator_Show
03 {
04     class Program
05     {
06         static void Main()
07         {
08             int a = 11;
09             a += 1;
10             Console.WriteLine(a);
11             a -= 1;
12             Console.WriteLine(a);
13             a *= 8;

```

```
14 Console.WriteLine(a);
15 a /= 2;
16 Console.WriteLine(a);
17 a %= 5;
18 Console.WriteLine(a);
19 Console.WriteLine("a 的二进制值是: ");
20 Console.WriteLine(Convert.ToString(a, 2));
21 a &= 3;
22 Console.WriteLine("a&3 后的二进制值是: ");
23 Console.WriteLine(Convert.ToString(a, 2));
24 a |= 5;
25 Console.WriteLine("a|5 后的二进制值是: ");
26 Console.WriteLine(Convert.ToString(a, 2));
27 a ^= 4;
28 Console.WriteLine("a^4 后的二进制值是: ");
29 Console.WriteLine(Convert.ToString(a, 2));
30 a <<= 2;
31 Console.WriteLine("a 左移 2 位后的二进制值是: ");
32 Console.WriteLine(Convert.ToString(a, 2));
33 a >>= 2;
34 Console.WriteLine("a 右移 2 位后的二进制值是: ");
35 Console.WriteLine(Convert.ToString(a, 2));
36 Console.ReadKey();
37     }
38 }
39 }
```

第 8 行代码是一个简单赋值运算符赋值运算，第 9 行代码开始是复合赋值运算符的使用方法。其实复合赋值运算符比较好理解，就是左操作数和右操作数先进行“=”运算符左侧运算符的运算，然后再将值重新赋给左操作数指定的变量。这段代码的执行结果如下：

```
12
11
88
44
4
a 的二进制值是:
100
a&3 后的二进制值是:
0
a|5 后的二进制值是:
101
a^4 后的二进制值是:
1
a 左移 2 位后的二进制值是:
100
```

a 右移 2 位后的二进制值是：

1

2.8 运算符的优先级

在一个表达式内，如果同时存在多个运算符，这些运算符的运算是有先后顺序的。它们的先后顺序取决于运算符的优先级。先看下面代码：

```
01 using System;
02 namespace Operator_Show
03 {
04     class Program
05     {
06         static void Main()
07         {
08             int a = 3 + 2 * 4 & 5 | 8;
09             Console.WriteLine(a);
10             Console.ReadKey();
11         }
12     }
13 }
```

这段代码的第 8 行是一个复合运算的表达式，如果不知道运算符的优先级，那么很难确定哪个运算符先参与运算。下表给出了 C# 中的所有运算符的优先级，其中也包括没有接触过的，这里先介绍一下，做个了解。

表 2-1 运算符优先级表

类别	运算符
基本运算符	., f(x) (方法)、a[x] (数组或索引)、x++、x--、new typeof、default、checked、unchecked、delegate
一元运算符	+, -, !, ~, ++x, --x, (T)x (强制转换)
乘、除、求余	*, /, %
加、减	+, -
移位	<<, >>
比较和类型检测	<, >, <=, >=, is, as
是否相等	==, !=
逻辑与	&
逻辑异或	^
逻辑或	
条件 AND (并且)	&&
条件 OR (或者)	
null 合并	??
三元运算符	?:
赋值和 Lamda 表达式	=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, =, =>

表中的运算符的优先级是按照从上到下，由高到低的顺序排列的。也就是说，基本运算符的优先级最高，赋值和 Lambda 表达式的优先级最低。

然后，根据表中的运算符优先级排列，再来看代码中的复合表达式。可以得出，先算乘法、然后计算加法、再计算逻辑与，最后计算逻辑或。代码的执行结果如下：

9

2.8.1 使用括号改变优先级的技巧

运算符的优先级并不需要死记硬背下来，如果在写代码的时候不能确定运算符的优先级顺序，可以通过小括号来确定优先级。代码实例如下：

```

01 using System;
02 namespace Operator_Show
03 {
04     class Program
05     {
06         static void Main()
07         {
08             int a = (3 + 2) * 4 / (4 + 1);
09             Console.WriteLine(a);
10             int b = (4 + 4) * ((3 * 12) - 4);
11             Console.WriteLine(b);
12             Console.ReadKey();
13         }
14     }
15 }

```

括号具有最高优先级。使用括号可以清楚的体现运算顺序，第 8 行代码是一个复合运算表达式，在这行代码中使用了括号。它的优先级顺序是这样的，第一步，表达式变成 $5*4/5$ ，这是因为处于同一级别的小括号具有相同优先级。第二步，表达式变成 $20/5$ 。第三步，计算出结果 4。第四步，赋值给 a。

第 10 行代码的括号中嵌套了括号，它的优先级是内层的括号先行计算，然后再计算外层。因此第一步，表达式变成 $8*(36-4)$ ；第二步，表达式为 $8*32$ ；第三步，计算出结果 256；第四步，将结果赋值给 b。

可以看到，使用括号后，程序的流程变得非常清楚了，有利于编程人员的理解和代码的编写。

2.9 一元运算符

前面介绍过的都是二元运算符，参加运算的是两个操作数。而一元运算符参加运算的只有一个操作数。基本的一元运算符包括+、-、!、~、++和--。它们分别是加、减、非、求补、自增和自减运算符。对于+运算符，比较好理解，它返回的就是操作数本身。代码如下：

```

01 using System;
02 namespace Operator_Show
03 {
04     class Program
05     {
06         static void Main()
07         {
08             int a = 16;
09             Console.WriteLine(+a);
10             short b = 32;
11             Console.WriteLine(+b);
12             long c = 3355;
13             Console.WriteLine(+c);
14             float d = 3.14F;
15             Console.WriteLine(+d);
16             decimal e = 3.14M;
17             Console.WriteLine(+e);

```

```

18         Console.ReadKey();
19     }
20 }
21 }

```

“+”运算符可以应用于整数、浮点数和 decimal 小数类型。它返回的就是操作数本身。

对于“-”运算符，首先来看应用于整数的情况。当对一个整型数应用-运算符时，它返回的结果是 0 减去这个整型数的结果。实例代码如下：

```

01 using System;
02 namespace Operator_Show
03 {
04     class Program
05     {
06         static void Main()
07         {
08             int a = 16;
09             a = -a;
10             Console.WriteLine(a);
11             int b = int.MinValue;
12             b = -b;
13             Console.WriteLine(b);
14             long c = long.MinValue;
15             c = -c;
16             Console.WriteLine(c);
17             uint d = uint.MaxValue;
18             var e = -d;
19             Console.WriteLine(e);
20             Type t1 = e.GetType();
21             Console.WriteLine(t1.ToString());
22             Console.ReadKey();
23         }
24     }
25 }

```

这段代码的执行结果如下：

```

-16
-2147483648
-9223372036854775808
-4294967295
System.Int64

```

第 8 行代码定义了一个整型数，第 9 行对这个整型数应用了“-”运算符，然后又重新赋值给了 a。这时，它的运算结果是-16。

第 11 行代码定义了一个整型数 b，然后用 int 类型的最小值对它赋值。语句中的 int.MinValue 是引用的 int 结构的属性。属性现在还没介绍到，目前只需要知道是赋的最小值就可以了。int 的最小值是个负数。但是第 12 行代码对它应用“-”运算符后，它并没有改变符号。第 13 行代码输出它的结果还是一个负值。同理，第 14 行代码对 long 类型的变量 c 赋值后，对它进行-运算符的操作后，结果也是这样。这是

为什么呢？这是因为 `int` 类型和 `long` 类型的最小值进行“-”运算符操作后，都变成了正数。而且这个正数超过了其类型的最大值，因此发生了溢出，导致结果不正确。

第 17 行定义了一个 `uint` 类型的变量，并为它赋了 `uint` 类型的最大值。然后对它执行“-”操作。这时它将变成一个负值。因为 `uint` 没有负值，编译器将会把它自动转换成一个 `long` 类型的值。但是，现在还没有介绍过隐式转换和强制转换，所以用一个推断类型来接收它的返回值。第 21 行打印它的类型，可以观察到，它就是一个 `long` 类型的值。

在这段代码执行并发生溢出时，并没有给出提示。这是因为默认情况下，编译器不检查数据的溢出。如果要编译器检查溢出，可以在项目的编译选项中更改。方法是单击开发环境的菜单栏中的“项目”，然后单击“属性”如图 2-1 所示。

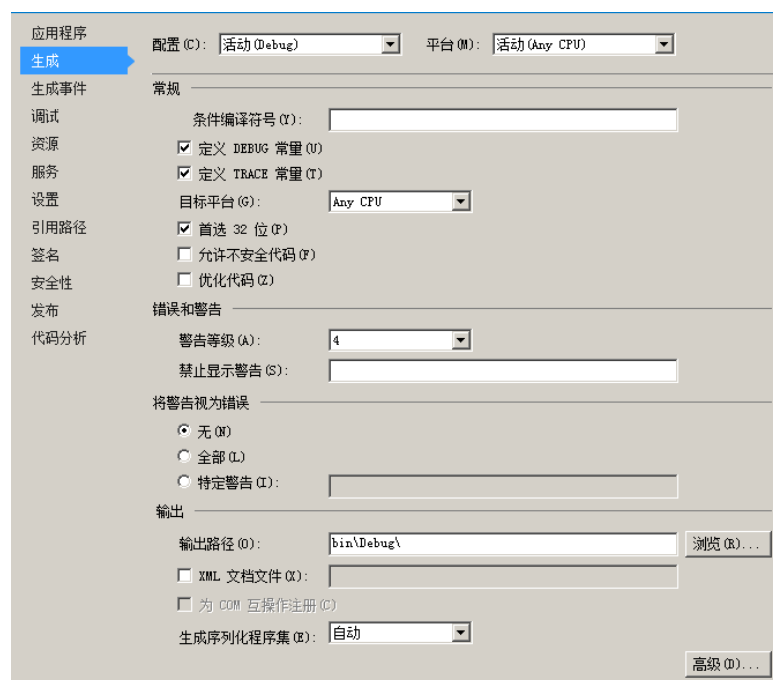


图 2-1 “属性”选项

接下来在“生成”项目中，单击“高级”按钮，如图 2-2 所示。

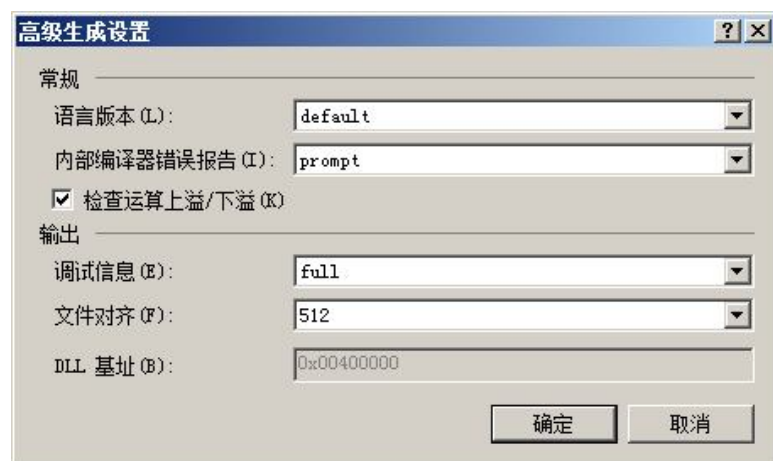


图 2-2 生成设置

在“高级生成设置”窗口中选中“检查运算上溢/下溢”选项，然后单击“确定”按钮。接下来，编译这段代码，然后执行，系统会报告错误。如下：

-16

未经处理的异常：System.OverflowException：算术运算导致溢出。

如果参与运算的操作数为 `ulong` 类型时，编译器将直接报告错误。代码如下：


```
01 using System;
02 namespace Operator_Show
03 {
04     class Program
05     {
06         static void Main()
07         {
08             ulong a = 5693;
09             var b = -a;
10             Console.WriteLine(b);
11             Console.ReadKey();
12         }
13     }
14 }
```

编写代码时，开发环境会直接报告错误，如下：

运算符“-”无法应用于“ulong”类型的操作数

对于 ulong 类型的数，不能对其使用-运算符。

介绍完整型数的-运算符后，再来看浮点数和 decimal 类型的小数的“-”运算符。代码如下：

```
01 using System;
02 namespace Operator_Show
03 {
04     class Program
05     {
06         static void Main()
07         {
08             float a = 3.14159F;
09             double b = 5.8765D;
10             a = -a;
11             b = -b;
12             Console.WriteLine(a);
13             Console.WriteLine(b);
14             float c = 0.0F / 0.0F;
15             c = -c;
16             Console.WriteLine(c);
17             decimal d = 2.457M;
18             d = -d;
19             Console.WriteLine(d);
20             Console.ReadKey();
21         }
22     }
23 }
```

对于 float 类型和 double 类型来说，对其应用-运算符时，就是将其原来符号进行反转。在代码的第 14 行定义了一个非数字的变量 c，如果对非数字应用-运算符，则结果仍然是非数字。代码的第 17 行定义了一个 decimal 类型的变量，对其应用“-”运算符时，结果是 0 减去其值的结果。这段代码的执行结果如下：

```
-3.14159
-5.8765
非数字
-2.457
```

下面再来看!运算符，!运算符应用于 bool 类型。实例代码如下：

```
01 using System;
02 namespace Operator_Show
03 {
04     class Program
05     {
06         static void Main()
07         {
08             bool a = true;
09             bool b = false;
10             Console.WriteLine("应用!运算符之前，a、b 的值是：");
11             Console.WriteLine(a);
12             Console.WriteLine(b);
13             a = !a;
14             b = !b;
15             Console.WriteLine("应用!运算符之后，a、b 的值是：");
16             Console.WriteLine(a);
17             Console.WriteLine(b);
18             Console.ReadKey();
19         }
20     }
21 }
```

第 8 行和第 9 行代码定义了两个 bool 类型的变量，在第 13 行代码和第 14 行分别对其应用了“!”运算符。“!”运算符就是对 bool 类型的值的否定。因此，第 16 行和第 17 行代码分别打印出了其否定值。执行结果如下：

```
应用!运算符之前，a、b 的值是：
True
False
应用!运算符之后，a、b 的值是：
False
True
```

求补运算符“~”应用于 int、uint、long、ulong 四种整型。对于每个类型执行的是按位求补运算。

代码如下：

```
01 using System;
02 namespace Operator_Show
03 {
04     class Program
05     {
06         static void Main()
07         {
```

```

08         int a = 3378;
09         Console.WriteLine("a 的原始二进制数是: ");
10         Console.WriteLine(Convert.ToString(a, 2));
11         a = ~a;
12         Console.WriteLine("对 a 求补后的二进制数是: ");
13         Console.WriteLine(Convert.ToString(a, 2));
14         Console.WriteLine(a);
15         Console.ReadKey();
16     }
17 }
18 }

```

这段代码的执行结果如下:

```

a 的原始二进制数是:
110100110010
对 a 求补后的二进制数是:
11111111111111111111001011001101
-3379

```

在打印 a 的原始二进制数的时候, 不要忘记 int 类型是 32 位的整数, 其最高位是符号位。因为 a 初始化为一个正数, 所以最高位为 0。当按位求补后, 最高位变为 1。所以结果是一个负数。

“++” 和 “--” 运算符有前缀和后缀之分, 在实际应用时, 得到的结果也完全不同。在 C# 中, 存在预定义 “++” 和 “--” 运算符的类型有 sbyte、byte、short、ushort、int、uint、long、ulong、char、float、double 和 decimal 类型。对这些类型应用 ++ 和 -- 运算符后的操作就是将这个操作数加 1 或减 1。实例代码如下:

```

01 using System;
02 namespace Operator_Show
03 {
04     class Program
05     {
06         static void Main()
07         {
08             int a = 11;
09             float b = 5.1F;
10             char c = 'a';
11             decimal d = 3.14M;
12             //定义同样类型的四个变量用来接收返回值
13             int e;
14             float f;
15             char g;
16             decimal h;
17             Console.WriteLine("应用前置++前, a、b、c、d 的值是: ");
18             Console.WriteLine(a);
19             Console.WriteLine(b);
20             Console.WriteLine(c);
21             Console.WriteLine(d);

```

```
22         e = ++a;
23         f = ++b;
24         g = ++c;
25         h = ++d;
26         Console.WriteLine("应用前置++后, e、f、g、h 的值是: ");
27         Console.WriteLine(e);
28         Console.WriteLine(f);
29         Console.WriteLine(g);
30         Console.WriteLine(h);
31         Console.WriteLine("应用前置++后, a、b、c、d 的值是: ");
32         Console.WriteLine(a);
33         Console.WriteLine(b);
34         Console.WriteLine(c);
35         Console.WriteLine(d);
36         e = a++;
37         f = b++;
38         g = c++;
39         h = d++;
40         Console.WriteLine("应用后置++后, e、f、g、h 的值是: ");
41         Console.WriteLine(e);
42         Console.WriteLine(f);
43         Console.WriteLine(g);
44         Console.WriteLine(h);
45         Console.WriteLine("赋值后, a、b、c、d 的值是: ");
46         Console.WriteLine(a);
47         Console.WriteLine(b);
48         Console.WriteLine(c);
49         Console.WriteLine(d);
50         Console.ReadKey();
51     }
52 }
53 }
```

这段代码的第 8 行到第 11 行定义了四个变量, 分别是 int、float、char 和 decimal 类型。第 13 行代码到第 16 行代码定义了同样类型的四个变量用来接收前四个变量应用运算符后的返回值。使用前置“++”后, 变量本身立刻进行了加 1 操作, 并返回加 1 操作后的值。可以看到第 27 行到第 30 行打印的结果和第 32 行到第 35 行打印的结果相同。但是使用后置“++”运算符就不同了, 使用后置“++”运算符后, 首先会返回操作数的原始值, 然后操作数再执行++操作。这样, 代码的第 41 行到第 44 行打印的结果和第 46 行到第 49 行打印的结果就会不同, 它比第 46 行到第 49 行打印的结果少 1。这段代码的执行结果如下:

应用前置++前, a、b、c、d 的值是:

```
11
5.1
a
3.14
```

应用前置++后, e、f、g、h 的值是:

```

12
6. 1
b
4. 14
应用前置++后, a、b、c、d 的值是:
12
6. 1
b
4. 14
应用后置++后, e、f、g、h 的值是:
12
6. 1
b
4. 14
赋值后, a、b、c、d 的值是:
13
7. 1
c
5. 14

```

2.10 checked 和 unchecked 运算符

前面在介绍“-”运算符时涉及到了数值的溢出问题。并且介绍了如何设置项目属性在程序编译时进行溢出检查或取消溢出检查。在 C#中, 与溢出检查配合使用的运算符就是 checked 和 unchecked, 它们分别是检查溢出和取消检查溢出运算符。下面看代码实例:

```

01 using System;
02 namespace Operator_Show
03 {
04     class Program
05     {
06         static void Main()
07         {
08             int a = 2147483647;
09             checked
10             {
11                 a++;
12                 Console.WriteLine(a);
13             }
14             Console.ReadKey();
15         }
16     }
17 }

```

这段代码首先定义了一个 int 类型的变量, 并取了 int 类型的变量的最大值。然后第 9 行代码使用了 checked 块将变量 a 的自增运算包在了语句块中。接下来在项目的属性中关闭编译器的溢出检测。然后编译代码并运行, 这时, 程序会抛出异常信息, 如下:

未经处理的异常: System.OverflowException: 算术运算导致溢出。

可见，checked 运算符的作用是进行溢出检查。

下面是 checked 运算符的另一种用法，代码如下：

```
01 using System;
02 namespace Operator_Show
03 {
04     class Program
05     {
06         static int Main()
07         {
08             int a = 2147483647;
09             int b;
10             b = checked(a + 1);
11             Console.ReadKey();
12             return checked(b = a + 1);
13         }
14     }
15 }
```

checked 运算符还可以用作检测后的值返回。在这段代码中的第 10 行和第 12 行就是这种用法。checked 后跟括号，括号内是要检测的数据。这段代码在运行时也会抛出溢出异常。

相反，unchecked 运算符就是取消检测的作用。看下面的代码：

```
01 using System;
02 namespace Operator_Show
03 {
04     class Program
05     {
06         static void Main()
07         {
08             int a = 2147483647;
09             int b;
10             unchecked
11             {
12                 b = a + 1;
13                 Console.WriteLine(b);
14             }
15             Console.ReadKey();
16         }
17     }
18 }
```

在编译这段代码之前，首先打开项目属性中的溢出检查项。这段代码中的第 10 行使用了 unchecked 关键字，取消了下面代码块中的对于 int 类型 b 的溢出检查。因此，运行不会抛出错误，代码的执行结果如下：

```
-2147483648
```

可以看到，执行结果已经溢出了。超出类型范围的结果的高序列位被放弃掉了。再看下面的代码：

```
01 using System;
02 namespace Operator_Show
```

```
03  {
04      class Program
05      {
06          static void Main()
07          {
08              int a = 2147483647 + 1;
09              Console.WriteLine(a);
10              Console.ReadKey();
11          }
12      }
13  }
```

在这段代码中，定义了一个常量表达式 $2147483647 + 1$ ，用它对 `a` 赋值。不论项目的属性中是否打开了溢出检查，开发环境都会进行常量表达式的溢出检查，除非使用 `unchecked` 关键字取消编译器的溢出检查。在这段代码编写时，开发环境会给出错误提示，如下：

在 `checked` 模式下，运算在编译时溢出

第三章 数组

3.1 值类型与引用类型

在 C# 中，数据类型可以分为三种。分别是值类型、引用类型和指针。其中指针用在不安全代码中。前面介绍过的基本类型都属于值类型。值类型的变量直接包含数据；而引用类型的变量就像 C++ 中的指针一样，它包含对托管堆中的对象的一个引用。因此，就有可能出现这样的情况，一个或多个变量同时引用着托管堆中的对象。如果对一个变量进行改变，那么，其实其他变量引用的内容也同时发生了改变。但是，值类型不会出现这样的情况，值类型与值类型之间是相对独立的个体，对一个值类型变量的改动不会影响其它值类型变量。

3.2 类类型

类类型是引用类型中的一种。类是一种抽象的概念，它代表一种有着公共特征的对象群体概念。举个例子来说，比如在地球上有着各种各样的人，从人种上说有黑色人种、白色人种、黄色人种，还有的人建议再分出棕色人种等；在性别上可分为男性和女性；在所做的工作上就分工更明确了，有工程师、科学家、医生等；每个具体的人也有着共同的特征，比如身高、年龄、血型等。在这个例子中，“人”如果在编程中就是抽象出来的一个类，而身高、年龄、工作等就是人的属性，每个人就是人这个类的一个具体对象。在程序设计中，类被定义为一种数据结构，它包含数据成员和方法成员以及嵌套类。同时，类支持继承机制，从一个类可以派生出一个子类来，子类对父类进行了扩展和专用化。类类型包含数据成员、函数成员和嵌套类型的数据结构。这其中数据成员又包括常量和字段，函数成员包括方法、属性、事件、索引器、运算符、实例构造方法、析构方法和静态构造方法。下面定义一个最简单的类，代码如下：

```
01 using System;
02 namespace Class_Base
03 {
04     class Program
05     {
06         public int a = 10;
07         static void Main()
08         {
09             Program p = new Program();
10             Console.WriteLine("Program 类中的字段 a 的值是：");
11             Console.WriteLine(p.a);
12             Console.ReadKey();
13         }
14     }
15 }
```

这段代码在前面介绍的只含有一个 Main 方法的类的代码基础上增加了几行代码。第 6 行代码定义了一个变量，它也叫字段，它属于实例变量。也就是说是属于类在托管堆中创建的类的实例（对象）的。它必须借由具体创建的实例才能来访问。它用 public 关键字修饰，说明它是公共访问权限的。使用“实例名.a”这样的形式就可以访问。代码第 7 行的 Main 方法前面用 static 关键字修饰，它说明这个方法是静态方法，它是属于类的。使用“类名.Main”这样的方式就可以访问。但是 Main 方法没用 public 修饰，它是 private（私有）权限的。不能从类外部直接访问，只能由操作系统来调用它。

在代码的第 9 行定义了一个 Program 类的实例，变量名为 p。创建类的实例要用 new 关键字在托管堆上分配内存，new 关键字后跟类名和一对小括号。

在代码的第 11 行使用 p.a 的形式访问了这个字段，打印了它的值。这段代码的执行结果如下：

Program 类中的字段 a 的值是：

10

这就是一个最简单的类，定义类用一对大括号，定义类的时候并不是创建一个实例，而是创建一个构架，也可以说是一个模板，当定义实例的时候，就用这个模板来在托管堆中创建实例。类的定义就好像一个印章一样，盖在纸上形成了一个实例对象。

3.3 object 类

在 C# 的类库中存在一个 object 类型，它是 System.Object 类型的别名。object 类型是所有其它类型的最终基类，C# 中的每种类型都是直接或间接从 object 类型派生，包括程序员自定义的类型。当一个类从另一个类派生的时候，被派生的类叫做基类或父类，派生出的新类叫做派生类或子类。当派生时，派生类除了基类的构造方法、析构方法以及静态构造方法外，拥有基类的其它所有成员。就像儿子继承爸爸的财产一样，只不过这些继承下来的成员本身有权限，存在子类能不能直接访问的问题。也就好像爸爸有一个保险箱，装了爸爸的私有物品，虽然继承给儿子了，但是儿子不知道密码，不能打开，只有通过爸爸的密码才能打开。爸爸的密码就好比基类的公有权限的方法，通过公有权限的方法才能访问基类的私有权限的成员。

因此，当一个类从 object 类继承时，它就自动拥有了 object 类的除构造方法、析构方法以及静态构造方法之外的所有成员，其中的公有成员可以直接访问，下面看代码实例：

```
01 using System;
02 namespace Class_Base
03 {
04     class Program
05     {
06         public int a = 11;
07         static void Main()
08         {
09             Program p1 = new Program();
10             Program p2 = new Program();
11             Console.WriteLine("p1 和 p2 是否相等: ");
12             bool b1 = p1.Equals(p2);
13             Console.WriteLine(b1);
14             int g1 = p1.GetHashCode();
15             int g2 = p2.GetHashCode();
16             Console.WriteLine(g1);
17             Console.WriteLine(g2);
18             Type t1 = p1.GetType();
19             Console.WriteLine(t1.ToString());
20             Program p3 = (Program)p1.MemberwiseClone();
21             Console.WriteLine("p3 的字段 a 的值是: ");
22             Console.WriteLine(p3.a);
23             Console.WriteLine(p1.ToString());
24             Console.ReadKey();
25         }
26     }
27 }
```

这段代码定义了一个名称为 Program 的类，在这个类中有一个 public 访问权限的字段 a，并且被初始化为 11。在代码的第 9 行和第 10 行各定义了一个类的实例。因为 C# 中所有的类都继承自 object 类，所以这个

类的实例也可以访问 object 基类的公有方法。当在开发环境中用实例名加“.”运算符时，就会出现它继承自 object 类的方法，共有五个。

第 12 行代码是访问的第一个可访问的方法，Equals 方法，它的作用是比较两个变量是否相等，如果被比较的对象是引用类型，则比较这两个引用类型是否指向同一个托管堆中的对象。bool 类型的变量 b1 的值为 false, 因为 p1 和 p2 不是同一个对象。

第 14 行和第 15 行代码是访问的 GetHashCode 方法，它的作用是返回对象的 Hash 码。

第 18 行代码调用了 GetType 方法，这个前面已经接触过了，它是返回对象的类型。

第 20 行调用了 MemberwiseClone 方法，它的作用是浅表复制，什么是浅表复制呢？如果对象的字段是一个值类型，则它会如实的再复制一个相同的值类型，如果对象的字段是一个引用类型，则它会复制出一个相同引用类型的字段，并且这个字段和原始的字段指向同一个托管堆中的对象，这就导致了改变复制出的对象的字段，则原始对象中字段指向的对象也会被同时改变。因为 MemberwiseClone 方法返回的是一个 object 类型，所以使用 Program 类型的变量接收它时，需要一个强制类型转换，强制类型转换后面会介绍。第 22 行打印复制出的对象的字段的值，可以看到，它和原始对象的字段值一样。

第 23 行调用了 ToString 方法，它是将对象转换成字符串。默认情况下，它会将对象的类名转换出来。这段代码的执行结果如下：

```
p1 和 p2 是否相等:
False
27226607
42549079
Class_Base.Program
p3 的字段 a 的值是:
11
Class_Base.Program
```

3.4 什么是数组

数组是一种包含若干元素的数据结构。包含的这些元素可以通过数组的索引来访问，并且这些元素具有相同的数据类型，元素的数据类型被称为数组的元素类型。数组是一种引用类型，定义的数组变量是指向托管堆中数组对象的引用。定义的数组变量位于栈中，因此，只要提到引用类型就要想到内存中关于区域的两个概念，栈是存储引用变量的，托管堆是存储实际对象或称为实例的，引用变量如同指针一样指向托管堆中的实例，或将引用变量想象成勺子的勺柄一样，拿着这个勺柄，就可以使用勺子。

3.5 一维数组

一维数组是一个线性排列的数据结构。它的定义实例代码如下：

```
01 using System;
02 namespace Array_Show
03 {
04     class Program
05     {
06         static void Main()
07         {
08             int[] a = { 0, 1, 2, 3, 4, 5 };
09             int[] b = new int[] { 0, 1, 2, 3, 4, 5 };
10             int[] c = new int[6];
11             foreach (int i in c)
12             {
13                 Console.WriteLine(i);
```

```

14      }
15      Console.WriteLine("-----");
16      c[0] = 0;
17      c[1] = 1;
18      c[2] = 2;
19      c[3] = 3;
20      c[4] = 4;
21      c[5] = 5;
22      foreach (int i in c)
23      {
24          Console.WriteLine(i);
25      }
26      Console.ReadKey();
27  }
28  }
29  }

```

第 8 行代码是最简单的一种一维数组的定义方式，数组的定义首先需要声明数组中实际存放的元素类型，然后跟一个方括号，再接数组的名称，形如：“int[] a”，就表示定义一个一维数组，名字为 a，数组中存放的数据类型为 int 类型。数组的维数也叫做数组的秩。它由数组定义中的方括号中的逗号个数加 1 来确定，一维数组的方括号中没有逗号，因此逗号个数为 0，再加 1，所以没有逗号的数组为一维数组。定义数组的名称后，用一个大括号括起来的初始化列表对数组进行了初始化，并用赋值运算符将它赋值给了 a。初始化列表中共有 6 个元素，它们在初始化的时候用逗号隔开。每个元素都可以通过一个索引来找到它。需要注意的是，索引是从 0 开始的，因此，第 6 个元素的索引是 5。

第 9 行代码使用了 new 运算符，同时应用了初始化列表，实际上，第 8 行代码就是第 9 行代码的简化形式。使用 new 运算符，就表明它在托管堆上分配了内存。一维数组的内存分配图如图 3-1 所示。

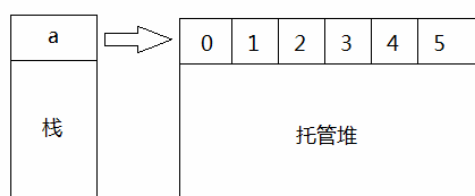


图 3-1 内存分配图

当定义一个一维数组后，内存中的分配就是像图 3-1 所表示的那样，数组类型的引用变量 a 被分配到了栈中，而实际数据被分配到了托管堆中，变量 a 指向了堆中的数据。其实，其它引用类型的内存分配方式也是这样。

第 10 行代码只是使用了 new 运算符，然后在方括号中定义数组中的元素个数。定义的个数也叫做数组的维数或长度。这样，在实际执行时，CLR 就会根据数组中元素的类型和方括号中定义的元素个数在堆中开辟空间。第 10 行代码执行完毕后，堆中的空间会开辟好，并且数组的每个元素都会自动初始化为 0。如果数组中的元素是引用类型的变量，则会初始化为空引用。

第 11 行代码使用了 foreach 语句循环打印了数组的元素，foreach 语句后面会介绍到。这里只看结果，每个数组元素可以观察到都为 0。

第 16 行代码开始使用索引的方式对每一个数组元素进行单独赋值，赋值的方法是使用数组名，然后跟一对方括号，方括号中给出数组实际元素的索引。不只是对数组元素写入的时候可以这样使用索引，当把单个元素赋值给别的元素的时候，也可以使用像“x = c[0];”这样的语句读出索引处的元素值。

第 22 行代码再次循环打印出了数组的所有值。这段代码的执行结果如下：

```
0
0
0
0
0
0
0
-----
0
1
2
3
4
5
```

3.6 多维数组

当数组的秩大于 1 时，这个数组被称为多维数组。下面看实例代码：

```
01 using System;
02 namespace Array_Show
03 {
04     class Program
05     {
06         static void Main()
07         {
08             int[,] a = { {0, 1}, {2, 3}, {4, 5} };
09             int[,] b = new int[3,2];
10             b[0, 0] = 0;
11             b[0, 1] = 1;
12             b[1, 0] = 2;
13             b[1, 1] = 3;
14             b[2, 0] = 4;
15             b[2, 1] = 5;
16             int[, ,] c = new int[2, 2, 2] { { { 0, 1 }, { 2, 3 } }, { { 4, 5 }, { 6, 7 } } };
17             foreach (int i in c)
18             {
19                 Console.WriteLine(i);
20             }
21             Console.ReadKey();
22         }
23     }
24 }
```

第 8 行代码定义了一个二维数组，二维数组以方括号中含有一个逗号来表示，含有一个逗号，它的秩就是 2。在初始化列表中，大括号里的第一层括号的个数就是第一维的维数，再往里是第二维的维数。

第 9 行代码使用 new 运算符定义了一个二维数组，用它和第 8 行代码的初始化式比较，可以看出，初始化式的大括号里的第一层括号的个数对应第 9 行中第一个逗号左面的 3，再往里用逗号隔开的数量对应

第 9 行代码中的逗号右边的 2。

第 10 行代码开始是根据每个值的索引对每个数组元素初始化。

第 16 行代码定义了一个三维数组，观察初始化式，大括号中的第一层括号的个数是第一维，也就是第一个逗号左边的维数，然后，第一维的每个括号里面又用逗号分隔开两对括号，它对应第二维的数量，再往里是分隔开的元素，它的数量是第三层的维数。

多维数组的元素个数可以用每个维数相乘的积来计算得出。第 17 行代码用 foreach 语句可以打印出数组中的每个值。由此可以看出，其实多维数组依然是线性排列元素的。

3.7 交错数组

前面讲过的一维数组和多维数组的元素类型都是前面给出的预定义类型，如果数组的元素类型也是数组的话，这个数组就叫做交错数组。实例代码如下：

```
01 using System;
02 namespace Array_Show
03 {
04     class Program
05     {
06         static void Main()
07         {
08             int[][] a = new int[3][];
09             a[0] = new int[2] { 0, 1 };
10             a[1] = new int[3] { 2, 3, 4 };
11             a[2] = new int[2] { 5, 6 };
12             Console.ReadKey();
13         }
14     }
15 }
```

第 8 行代码定义了一个交错数组，它代表定义了一个数组 a，数组的类型是整型数组类型。并且第一维的数量为 3。在定义交错数组时，第一个方括号代表数组的维数，如果是多维交错数组，那么维数在这个括号进行定义。

第 9 行代码对数组的第一个元素又进行了分配堆内存，它指向了一个新的整型数组，数组含有两个元素。

第 10 行代码对第二个元素进行定义，它指向一个新的维数为 3 的一维整型数组。

第 11 行代码对第三个元素进行定义，指向一个新的维数为 2 的整型数组。

交错数组的内存分配图如图 3-2 所示。

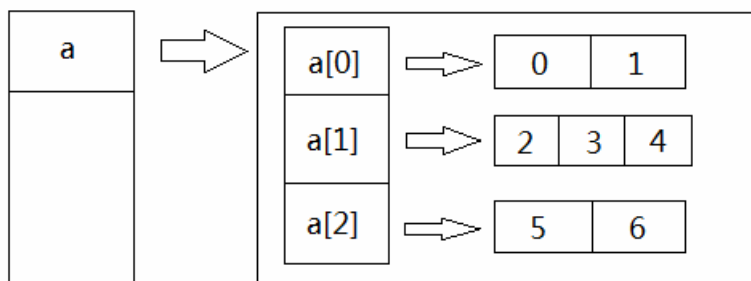


图 3-2 交错数组内存分配

可以看到，交错数组的第一维是一个线性数组。然后，每个元素又是一个引用，指向另一个新数组。

3.8 System.Array 类

前面介绍过，所有的类型都隐式地派生自 object 类，也就是说，object 类是所有类的基类。C#中的

数组都隐式派生自 `System.Array` 类，而 `System.Array` 类又派生自 `object` 类。`System.Array` 是一个抽象基类，抽象类不能直接创建实例，这个后面会介绍到。因此，当定义一个数组的时候，它能调用 `System.Array` 类的方法，当在实例名后用 “.” 运算符的时候，开发环境会给出智能提示，如图 3-3 所示。

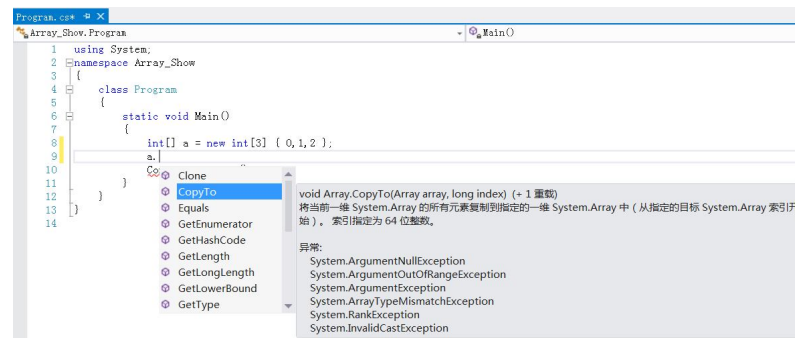


图 3-3 `Array` 类成员

`Array` 类是抽象类，数组都继承自它，但 `Array` 类不是一个数组类型，它是类类型。虽然它不能用来创建对象，但是它的成员，数组对象可以调用。

第四章 语句

4.1 表达式语句

程序的执行是通过一条条的语句来实现的。前面已经接触过运算符和表达式。下面的代码演示了表达式语句的用法。

```
01 using System;
02 namespace Statement_Show
03 {
04     class Program
05     {
06         static void Main()
07         {
08             int a = 3 + 2;
09             a++;
10             a += 12;
11             {
12                 Console.WriteLine("a 的值现在是: ");
13                 Console.WriteLine(a);
14             }
15         }
16     }
17 }
```

表达式语句的作用是对表达式进行求值。这段代码的第 8 行到第 10 行都是表达式语句。它们都以分号进行结尾。

第 11 行是一个块，前面已经接触过了。在块中可以含有一个语句列表，语句列表就是多条可以顺序执行的语句。块的作用就是包含语句，它也可以什么也不包含，那么它就是一个空块。当然，这样也没有什么意义。

第 12 行开始是方法调用语句，它调用了类库中的方法。

4.2 标记语句

标记语句的特点是在一个语句前加上一个标签作为这条语句的标记。代码实例如下：

```
01 using System;
02 namespace Statement_Show
03 {
04     class Program
05     {
06         static void Main()
07         {
08             int a = 11;
09             Label1:
10             {
11                 Console.WriteLine("这是 Label1 块");
12             }
13             Label2:
14             {
15                 Console.WriteLine("这是 Label2 块");
16             }
17         }
18     }
19 }
```

```

15         }
16     }
17     a: Console.WriteLine("这是 a 标记语句");
18     Console.ReadKey();
19 }
20 }
21 }

```

第 9 行代码定义了一个标记语句，它的标签是 Label1，标签要后跟一个冒号。Label1 的有效范围是它所在的块，在这段代码中就是 Main 方法的方法体。

第 12 行又定义了一个标签 Label2，它的范围就是 Label1 标记的块范围。

第 17 行定义了一个标签 a，需要注意的是，标签的名称和第 8 行代码定义的局部变量名称相同。这是允许的。

标记语句是为 goto 语句准备的，供 goto 语句进行程序跳转使用。

4.3 goto 语句

goto 语句是一种跳转语句，它能改变程序的流程。实例代码如下：

```

01 using System;
02 namespace Statement_Show
03 {
04     class Program
05     {
06         static void Main()
07         {
08             int a = 11;
09             goto Label1;
10         Label1:
11             {
12                 Console.WriteLine("这是 Label1 块");
13             Label2:
14                 {
15                     Console.WriteLine("这是 Label2 块");
16                 }
17             }
18             a: Console.WriteLine("这是 a 标记语句");
19             Console.WriteLine(a);
20             Console.ReadKey();
21         }
22     }
23 }

```

这段代码基于上一小节的代码，在定义局部变量之后，第 9 行添加了一个 goto 语句，goto 语句后跟要跳转到的标签 Label1。因为 Label1 标签就是 goto 语句的下一条语句，所以，后面的所有语句都得以顺序执行，而不管它们有无标签。代码的执行结果如下：

```

这是 Label1 块
这是 Label2 块
这是 a 标记语句

```



```
11
```

从结果可以观察到，所有代码路径都得到了执行，而没有遗漏。下面将第 9 行代码稍加改动，如下：

```
goto a;
```

这时的执行结果如下：

```
这是 a 标记语句
```

```
11
```

这时可以观察到，在上一执行结果的前两句都没有得到执行。也就是说，程序的流程进行了跳转。同时，开发环境也会给出 3 个警告，如下：

```
警告 1 这个标签尚未被引用 Program.cs 10
```

```
警告 2 检测到无法访问的代码 Program.cs 10
```

```
警告 3 这个标签尚未被引用 Program.cs 13
```

因为代码从第 9 行直接跳转到了第 18 行，所以第 10 行代码到第 17 行代码不能得到执行，编译器会检测到这些无法访问的代码，并且会给出提示是哪行。其余两个警告是因为标签 Label1 和标签 Label2 两个标签写入代码后却没有得到使用。

如果将第 9 行代码改为：

```
goto Label2;
```

这就会出现一条错误提示，如下：

```
错误 1 goto 语句范围内没有“Label2”这样的标签 Program.cs 9
```

这是因为 Label2 存在于 Label1 所在的块内，在 goto 语句所在的范围内，是无法看到 Label2 的。所以，goto 语句无法进入嵌套的标记语句。

如果某个语句无法访问，则称这个语句是程序执行流程不能到达的，如果程序执行能到达某个语句，则称这个语句是可到达的。

4.4 if 语句

if 语句通过对条件的判断实现流程的跳转。下面通过代码实例来进行说明，代码如下：

```
01 using System;
02 namespace Statement_Show
03 {
04     class Program
05     {
06         static void Main()
07         {
08             int a = 11;
09             Console.Write("请输入一个整型值进行判断: ");
10             int b = Convert.ToInt32(Console.ReadLine());
11             if (a > b)
12             {
13                 Console.WriteLine("您输入的值过小。");
14             }
15             if (a < b)
16             {
17                 Console.WriteLine("您输入的值过大。");
18             }
19             if (a == b)
20             {
```

```

21         Console.WriteLine("您输入的值正好。");
22     }
23     Console.ReadKey();
24 }
25 }
26 }

```

首先第 8 行代码定义了一个整型变量并且赋初始值为 11，它的作用是作为比较的基准。

第 10 行代码调用了 `Console.ReadLine` 方法读取一个用户输入的字符串，然后调用 `Convert.ToInt32` 方法将它转化为一个整型数，并用整型 `b` 来接收它。

从代码的第 11 行开始，就进入了 `if` 语句的判断流程，语法是 `if` 关键字后跟一对小括号，在小括号中写入判断语句，这个判断语句要求返回的是 `bool` 类型的值。然后后跟一个语句块，里面是符合判断时，要执行的语句列表。如果 `if` 语句后跟的代码块内只有一条语句，也可以不用大括号括起来。但是为了代码编写的规范，即使只有一条语句，也最好用大括号括起来。在这段代码中的 `if` 判断语句共写了三条。在小括号中可以使用的关系运算符如表 4-1 所示（其中定义两个 `int` 类型的变量做示例，`int a = 27, b = 10`）：

表 4-1 `if` 语句关系运算符表

操作符	使用方法
<code>==</code>	<code>if(a == 27)</code> ，返回 <code>true</code>
<code>!=</code>	<code>if(a != 27)</code> ，判断 <code>a</code> 是否不等于 27，返回 <code>false</code>
<code><</code>	<code>if(a < 30)</code> ，返回 <code>true</code>
<code>></code>	<code>if(b > 5)</code> ，返回 <code>true</code>
<code><=</code>	<code>if(a <= 27)</code> ，返回 <code>true</code>
<code>>=</code>	<code>if(b >=10)</code> ，返回 <code>true</code>
<code>&&</code>	<code>if(a < 28 && b >=10)</code> ，返回 <code>true</code> ，逻辑与操作，只有前后表达式均为 <code>true</code> ，才返回 <code>true</code> 。它可以理解为“并且”的意思
<code> </code>	<code>if(a < 28 b >10)</code> ，返回 <code>true</code> ，逻辑或操作，前后只要有一个为 <code>true</code> ，就返回 <code>true</code> ，它可以理解成“或者”的意思
<code>!</code>	<code>if(!(a<30))</code> ，返回 <code>false</code> ，逻辑非操作，如果!后面的表达式为 <code>false</code> ，则整个表达式为真

接下来开始执行代码，如果输入的值小于 11，那么第 11 行代码将会得到执行，执行语句块中的语句后，程序流程会到达第 23 行，而第 15 行到第 22 行将是不可达的。当输入符合 15 行或 19 行代码的条件时，那么相关的语句块中的内容也会得到执行，其它两个 `if` 语句就是不可达的。

下面执行代码并输入 11，执行结果如下：

```

请输入一个整型值进行判断: 11
您输入的值正好。

```

4.5 else 语句

`else` 语句是作为 `if` 语句的补充存在，代码实例如下：

```

01 using System;
02 namespace Statement_Show
03 {
04     class Program
05     {
06         static void Main()
07         {
08             int a = 11;
09             Console.Write("请输入一个整型值进行判断: ");

```

```
10         int b = Convert.ToInt32(Console.ReadLine());
11         if (a > b)
12         {
13             Console.WriteLine("您输入的值过小。");
14         }
15         else if (a < b)
16         {
17             Console.WriteLine("您输入的值过大。");
18         }
19         else
20         {
21             Console.WriteLine("您输入的值正好。");
22         }
23         Console.ReadKey();
24     }
25 }
26 }
```

这段代码将第 15 行和第 19 行代码进行了改写，添加了 else（否则）语句。修改后的代码逻辑并没有发生改变，和原来的执行流程都是一样的，但是 else 语句的存在让 if 语句的逻辑性更有条理，而且 else 可以与 if 联用，让条件判断更有层次感。这段代码的执行结果如下：

请输入一个整型值进行判断：13

您输入的值过大。

4.6 for 语句

for 循环语句是让代码按照一定的判断条件重复调用。实例代码如下：

```
01 using System;
02 namespace Statement_Show
03 {
04     class Program
05     {
06         static void Main()
07         {
08             //for 关键字后面的括号中是判断条件，由三部分语句组成
09             //执行顺序是 i 先初始化，然后判断条件，如果符合则执行代码块
10             //代码块执行完毕后，进行判断语句的最后一部分自加
11             //然后再回到语句的第二部分进行判断，如符合则再次执行代码块
12             for (int i = 0; i < 3; i++)
13             {
14                 if (i == 0)
15                 {
16                     Console.WriteLine("本小姐第一次变化");
17                     Console.WriteLine("被唐僧的大徒弟孙悟空打死了");
18                 }
19                 if (i == 1)
20                 {
```

```

21         Console.WriteLine("本小姐第二次变化");
22         Console.WriteLine("被那 500 年前大闹天宫的弼马温又打死了");
23     }
24     if (i == 2)
25     {
26         Console.WriteLine("本小姐第三次变化");
27         Console.WriteLine("被那该死的泼猴再次打死了");
28     }
29 }
30 Console.WriteLine("这就是三打白骨精的由来");
31 Console.ReadKey();
32 }
33 }
34 }

```

for 语句是一种循环语句，它的语法是，for 关键字后跟一对小括号，在小括号中首先定义一个整型变量，并对变量进行初始化，并以分号结尾。然后进行逻辑判断，如果符合条件，则执行语句块中的内容。代码块中的语句执行完毕后，执行小括号中的最后一个语句对变量进行操作。比如这段代码中的 i++。

这段代码的执行流程是这样的：当代码的执行流程到达第 12 行的时候，定义一个整型变量 i，并将它初始化为 0。然后判断它是否小于 3，结果是符合，代码将返回 true。这时程序流程就会到达语句块中，在语句块中又会进行 if 语句的判断。

结果是第 14 行的语句得到执行，因此执行它后面的代码块。因为其它 if 语句不符合条件，所以这些语句不可到达。接下来对变量 i 进行自增，然后再进行判断是否小于 3，结果仍然为 true。这时代码流程又会到达语句块，再进行 if 判断。

直到 for 语句后面括号中的判断语句返回 false 了，程序流程将会跳出 for 循环，到达代码的第 30 行。然后依次往下执行。这段代码的执行结果如下：

```

本小姐第一次变化
被唐僧的大徒弟孙悟空打死了
本小姐第二次变化
被那 500 年前大闹天宫的弼马温又打死了
本小姐第三次变化
被那该死的泼猴再次打死了
这就是三打白骨精的由来

```

for 语句中对变量 i 的操作部分，就是这段代码中的 i++ 部分，也可以是其他语句，如：i+=2, i-- 等。对 i 操作的那一部分也可以放在后面的块中，执行效果是一样的，代码如下：

```

using System;
namespace Statement_Show
{
    class Program
    {
        static void Main()
        {
            for (int i = 0; i < 3;)
            {
                if (i == 0)

```

```

        {
            Console.WriteLine("本小姐第一次变化");
            Console.WriteLine("被唐僧的大徒弟孙悟空打死了");
        }
        if (i == 1)
        {
            Console.WriteLine("本小姐第二次变化");
            Console.WriteLine("被那 500 年前大闹天宫的弼马温又打死了");
        }
        if (i == 2)
        {
            Console.WriteLine("本小姐第三次变化");
            Console.WriteLine("被那该死的泼猴再次打死了");
        }
        i++;
    }
    Console.WriteLine("这就是三打白骨精的由来");
    Console.ReadKey();
}
}
}

```

甚至用来判断的变量 `i` 也可以是浮点型或 `decimal` 小数类型，但是通常在代码中没有这样使用的。

4.7 空语句和 `break` 语句

`for` 循环的一种特殊变化是无限循环，它永远不跳出循环。想跳出循环就得程序员自己想办法。实例代码如下：

```

01 using System;
02 namespace Statement_Show
03 {
04     class Program
05     {
06         static void Main()
07         {
08             //无限循环的特点是使用空语句
09             for (; ; )
10             {
11                 string myText = @"从前有座山
12                             山上有座庙
13                             庙里有个老和尚和一个小和尚
14                             老和尚给小和尚讲故事";
15                 Console.WriteLine(myText);
16                 string i = Console.ReadLine();
17                 if (i == "q")
18                 {
19                     //使用 break 语句可以打断循环

```

```

20             break;
21         }
22     }
23     Console.WriteLine("唠唠叨叨的故事终于讲完了");
24     Console.ReadKey();
25 }
26
27 }
28 }

```

首先介绍一下什么是空语句，空语句只含有一个语句结束的分号，它什么也不做。但是什么也不做，并不代表它没有用处。

第 9 行代码在 for 循环中使用了空语句。可以看到，在 for 关键字后的括号中只有两个分号。因为不判断退出的条件，所以这个循环将无限进行下去。也就是说，它是个无限循环。

第 11 行代码定义了 string 类型的变量，并且用@开始的字符串文本初始化这个变量。string 类型的变量后面会介绍到。这个变量也可以定义在 for 循环的外面，这样就可以不必每次循环都重新分配内存。

第 15 行打印这个定义的字符串。

第 16 行代码定义了一个 string 类型的变量接收用户输入的字符串。

第 17 行代码判断这个输入的字符串是否为“q”，如果为“q”，则第 20 行代码使用 break 语句退出循环。如果不为“q”，则循环将继续下去，也继续打印字符串 myText。

如果退出了循环，则继续从第 23 行代码执行下去。

代码中使用的 break 语句就是打断循环的作用，即使是无限循环，遇到这个关键字，也会立即退出循环。这段代码的执行结果如下：

```

从前有座山
山上有座庙
庙里有个老和尚和一个小和尚
老和尚给小和尚讲故事
t
从前有座山
山上有座庙
庙里有个老和尚和一个小和尚
老和尚给小和尚讲故事
q
唠唠叨叨的故事终于讲完了

```

4.8 while 语句

while 语句也是一个循环语句。和 for 循环语句不同的是，它没有定义局部变量，只对一个条件进行判断。如果为 true，则进行循环；如果为 false，则跳出循环。实例代码如下：

```

01 using System;
02 namespace Statement_Show
03 {
04     class Program
05     {
06         static void Main()
07         {
08             Console.WriteLine("—————金角大王的宝葫芦—————");

```

```

09          //定义一个字符串并初始化为空
10          string name = string.Empty;
11          //while 关键字之后的括号内是判断语句，满足条件则持续循环
12          while (name != "孙行者")
13          {
14              Console.WriteLine("金角大王：我叫你，你敢答应吗？——孙行者");
15              //当输入的字符串为“孙行者”时，条件判断为 false,跳出循环
16              name = Console.ReadLine();
17          }
18          Console.WriteLine("嗖的一声....金角大王：哈哈，得手了！");
19          Console.ReadKey();
20      }
21  }
22 }

```

while 语句的使用方法是首先使用 while 关键字，然后后跟一对小括号。在括号中是判断是否循环的条件。如果为 true，则进行循环，并执行后面的语句块内的语句；如果为 false，则跳出循环。

本段代码在第 10 行定义了一个 string 类型的变量并初始化为空字符串，这里使用了 string 类型的 Empty 属性。

在第 12 行开始 while 语句并进行条件判断，如果符合则执行语句块内的语句。

第 16 行代码接受用户输入的字符串，并且为第 10 行代码定义的局部变量赋值，以便下次判断条件时使用。

如果条件符合，即 name 变量的内容为“孙行者”时，条件判断表达式返回 false，这时跳出循环，执行第 18 行开始的语句。这段代码的执行结果如下：

```

—————金角大王的宝葫芦—————
金角大王：我叫你，你敢答应吗？——孙行者
者行孙
金角大王：我叫你，你敢答应吗？——孙行者
行者孙
金角大王：我叫你，你敢答应吗？——孙行者
孙行者
嗖的一声....金角大王：哈哈，得手了！

```

4.9 do...while 语句

和 while 语句密切相关的是 do...while 语句。它与 while 的不同点是，while 语句先判断条件，然后进行循环；而 do...while 语句如同莽张飞一样，先做一遍，然后再判断。实例代码如下：

```

01 using System;
02 namespace Statement_Show
03 {
04     class Program
05     {
06         static void Main()
07         {
08             Console.WriteLine("—————金角大王的宝葫芦—————");
09             //定义一个字符串并初始化为空
10             string name = string.Empty;

```

```

11         do
12         {
13             Console.WriteLine("金角大王：我叫你，你敢答应吗？——孙行者");
14             //当输入的字符串为“孙行者”时，条件判断为 false,跳出循环
15             name = Console.ReadLine();
16             //此处别忘记用分号结尾
17         } while (name != "孙行者");
18         Console.WriteLine("嗖的一声... 金角大王：哈哈，得手了！");
19         Console.ReadKey();
20     }
21 }
22 }

```

do...while 语句是将 do 语句放在前面，以表示它是首先要执行一遍的内容，然后用 while 语句进行判断条件。在 do 关键字后紧跟着一个语句块，在其中放入首先要执行一遍的内容，然后在语句块之后接 while 语句，但是有一点不要忘记，while 语句后是需要分号结尾的，这和单纯的 while 语句不同。

这段代码改编自上一小节的代码，在第 11 行定义一个 do 语句，然后将原来 while 语句块内的内容置入其中。

在代码的第 17 行是 do 语句块的结尾分号，然后后接一个 while 语句进行判断。这段代码的执行结果是和上一小节一样的。

4.10 switch 语句

switch 语句是一种选择语句，它是根据给定的条件去匹配各个选项，以达到控制程序流程的目的。代码实例如下：

```

01 using System;
02 namespace Statement_Show
03 {
04     class Program
05     {
06         static void Main()
07         {
08             Console.WriteLine("—————葵花宝典—————");
09             //定义一个无限循环，可以让用户重复选择
10             for (; ; )
11             {
12                 Console.WriteLine("1. 第一章 2. 第二章 3. 第三章");
13                 //变量使用 int 类型，因此需要转换类型
14                 int myInt;
15                 myInt = Convert.ToInt32(Console.ReadLine());
16                 //与以下数个 case 标签进行判断
17                 switch (myInt)
18                 {
19                     //在标签后的语句执行后，必须使用 break 语句跳出
20                     case 1:
21                         Console.WriteLine("欲练神功，必先自宫");
22                         break;

```



```

23             case 2:
24                 Console.WriteLine("即使自宫，未必成功");
25                 break;
26             case 3:
27                 Console.WriteLine("早知如此，何必自宫");
28                 break;
29             //可以将多个 case 标签放到一起，实现多选
30             case 4:
31             case 5:
32             default:
33                 Console.WriteLine("本宝典一共就三章, 请从头练起");
34                 break;
35         }
36         if (myInt == 9)
37         {
38             Console.WriteLine("程序即将退出");
39             break;
40         }
41     }
42     Console.ReadKey();
43 }
44 }
45 }

```

第 10 行代码使用了一个无限循环的 for 语句，它的作用是让章节的选择不断的重复出现，直到符合条件的时候才跳出。条件的判断在第 36 行，如果输入的字符串是 9，则使用 break 语句跳出循环。

第 17 行代码是 switch 语句，switch 关键字后跟一个括号，括号中是要判断的变量。

第 18 行开始是语句块，在语句块中，使用 case 关键字后跟要匹配的值，然后再接冒号。冒号后就是要执行的语句列表，需要注意的是，每个语句列表都要使用一个 break 语句跳出 switch 语句。如果没有 break 语句，则会产生语句的贯穿现象。

如果要执行的语句列表匹配多个值，那么就可以像第 30 行代码那样，多个 case 语句联用。但最终都要使用 break 语句。

对于其它可能的值，如果要使用一个默认的语句列表进行匹配执行，就可以使用 default 语句。如同第 32 行那样使用。这样，凡是不在 case 语句列表中匹配的值，都执行第 32 行后面给出的语句列表。

如果任何一个 case 语句忘记使用了 break 语句，出现了贯穿，则编译器会给出提示。如下：

控制不能从一个 case 标签 ("case 1:") 贯穿到另一个 case 标签 Program.cs 20

出现贯穿结果会如何呢？在 C++ 中会出现上个 case 中的语句列表执行完毕后，程序控制继续跳转到下一个 case 语句块执行的情况。但是，C# 就不同了，它是安全的代码，在编译时就可以检查到这个错误，并给出错误提示。这时，编译不会执行下去。

最后需要注意的是，case 语句后面用来匹配的值不能是变量，而只能是一个文本值。否则，编译器会给出下面的错误提示：

应输入常量值 Program.cs 21

这段代码的执行结果如下：

-----葵花宝典-----

1. 第一章 2. 第二章 3. 第三章

```

1
欲练神功，必先自宫
1. 第一章 2. 第二章 3. 第三章
2
即使自宫，未必成功
1. 第一章 2. 第二章 3. 第三章
3
早知如此，何必自宫
1. 第一章 2. 第二章 3. 第三章
4
本宝典一共就三章, 请从头练起
1. 第一章 2. 第二章 3. 第三章
9
本宝典一共就三章, 请从头练起
程序即将退出

```

4.11 foreach 循环语句

foreach 语句用于遍历数组和集合对象中的所有项，而无需关心数组成员的个数问题。并且可以对数组或集合中的每个元素执行一次语句块中的语句。下面通过一个代码实例来说明用法：

```

01 using System;
02 namespace Statement_Show
03 {
04     class Program
05     {
06         static void Main()
07         {
08             //定义两个数组
09             char[] myStringArray = new char[8] { '刀', '枪', '剑', '戟', '斧', '钺',
            '勾', '叉' };
10             int[] myAgeArray = new int[8] { 10, 20, 30, 40, 50, 60, 70, 80 };
11             //定义 foreach 循环遍历输出数组的每个项
12             Console.WriteLine("花果山的兵器库有这些种类的兵器：");
13             foreach (char c in myStringArray)
14             {
15                 Console.Write(c + " ");
16             }
17             Console.WriteLine();
18             Console.WriteLine("花果山的猴子的年龄有这样的分布：");
19             foreach (int age in myAgeArray)
20             {
21                 Console.Write(age.ToString() + " ");
22             }
23             Console.ReadKey();
24         }
25     }

```

```
26 }
```

第 9 行代码定义了一个 char 类型的数组并进行了初始化。

第 10 行代码定义了一个整型数组并进行了初始化。

为了遍历数组中的元素，第 13 行代码定义了一个 foreach 语句，foreach 语句的用法是 foreach 关键字后跟一对括号，在括号中首先定义一个和集合中的元素相同的变量，然后接 in 关键字，再接数组或集合的变量名。在后面要执行的语句块中写入对定义的变量的操作。但是要注意，每个遍历出的变量都是只读的，不能对它执行写入的操作。

第 19 行定义了一个同样的 foreach 语句遍历整型数组的每个项。这段代码的执行结果如下：

花果山的兵器库有这些种类的兵器：

刀 枪 剑 戟 斧 钺 勾 叉

花果山的猴子的年龄有这样的分布：

10 20 30 40 50 60 70 80

在 foreach 语句块中也可以不对变量进行操作，语句块中是否对变量操作，和对数组或集合进行循环遍历没有影响。如果将第 21 行代码改成如下形式：

```
Console.WriteLine("20");
```

这时，代码依旧可以执行，执行结果如下：

花果山的兵器库有这些种类的兵器：

刀 枪 剑 戟 斧 钺 勾 叉

花果山的猴子的年龄有这样的分布：

2020202020202020

如果在实际编写代码过程中，无法确定数组或集合中的元素类型，则可以使用推断类型作为遍历的变量，将前面的 foreach 语句改写如下：

```
foreach (var c in myStringArray)
{
    Console.WriteLine(c + " ");
}
```

这时的执行结果也和原来一样。如果在语句块中对遍历出的变量执行了写入操作，则会引发编译时错误，修改代码如下：

```
01 using System;
02 namespace Statement_Show
03 {
04     class Program
05     {
06         static void Main()
07         {
08             //定义两个数组
09             char[] myStringArray = new char[8] { '刀', '枪', '剑', '戟', '斧', '钺',
10             '勾', '叉' };
11             int[] myAgeArray = new int[8] { 10, 20, 30, 40, 50, 60, 70, 80 };
12             //定义 foreach 循环修改数组的每个项
13             foreach (char c in myStringArray)
14             {
15                 c = c + 1;
16             }
```

```

17         foreach (int age in myAgeArray)
18         {
19             age += 1;
20         }
21         Console.ReadKey();
22
23     }
24 }
25 }

```

在代码的第 15 行和第 19 行对变量进行了写入的操作，这时编译器会给出错误提示如下：

“c” 是一个 “foreach 迭代变量”，无法为它赋值
“age” 是一个 “foreach 迭代变量”，无法为它赋值

4.12 continue 语句

continue 语句的作用是开始封闭它的 for、while、do...while 以及 foreach 语句的下一循环。在 continue 所在的循环体内，如果 continue 语句下面还有语句，则不再执行，执行新一轮的循环。实例代码如下：

```

01 using System;
02 namespace Statement_Show
03 {
04     class Program
05     {
06         static void Main()
07         {
08             Console.WriteLine("—————迭代输出孙悟空的百宝囊—————");
09             string[] myString = new string[9] { "刀", "枪", "剑", "戟", "金箍棒", "斧",
10             "钺", "勾", "叉" };
11             foreach (string s in myString)
12             {
13                 //检测到符合条件，则继续下一次迭代，不再继续执行下面的代码
14                 if (s == "金箍棒")
15                 {
16                     Console.WriteLine("金箍棒在耳朵里，不在百宝囊中!");
17                     continue;
18                 }
19                 Console.WriteLine(s);
20             }
21             Console.ReadKey();
22         }
23     }

```

第 9 行代码定义了一个 string 类型的数组并进行了初始化。不用担心，马上就会介绍到 string 类型。

第 10 行代码开始了一个 foreach 循环，在语句块中，对遍历到的每个元素由第 13 行代码的 if 语句进行判断，如果条件符合则执行 if 语句块中的内容。

第 16 行代码是当 if 语句的判断为 true 时，执行 continue 语句开始 foreach 语句的下一循环。这

时，第 18 行代码被跳过，不再执行了。这段代码的执行结果如下：

```
-----迭代输出孙悟空的百宝囊-----
刀
枪
剑
戟
金箍棒在耳朵里，不在百宝囊中！
斧
钺
勾
叉
```

4.12.1 使用 goto 语句穿越控制

continue 语句继续执行的是包裹它的最近一层的循环，要想穿越多个循环体转移代码的执行控制，只能使用 goto 语句。再看下面的代码：

```
01  using System;
02  namespace Statement_Show
03  {
04      class Program
05      {
06          static void Main()
07          {
08              for (int i = 0; i < 5; i++)
09              {
10                  for (int j = 0; j < 8; j++)
11                  {
12                      if (j == 2)
13                      {
14                          Console.WriteLine("现在 j 的值是：");
15                          Console.WriteLine(j);
16                          Console.WriteLine("继续下一轮的循环");
17                          continue;
18                      }
19                      Console.Write("内层数值：");
20                      Console.Write(j);
21                      Console.Write('\n');
22                      if (j == 3)
23                      {
24                          Console.WriteLine("跳转到外层");
25                          goto a;
26                      }
27                  }
28              a: Console.Write("当前 i 的值是： " + i + '\n');
29                  Console.WriteLine("外层开始新一次循环");
30          }
```

```
31         Console.ReadKey();  
32     }  
33 }  
34 }
```

这段代码中定义了两个嵌套的 for 语句，第 12 行代码所在的 if 语句对内层的 for 语句的变量 j 进行判断，如果 j 为 2，则继续下一次循环。这时第 19 行代码到第 26 行代码都不会得到执行，它们都是不可达的。但是，因为这个 continue 语句是位于内层的 for 语句中，所以它起作用的范围只是包裹它的那层 for 循环。它不会对外层的 for 语句产生作用。

要想穿越控制，到达外层 for 语句，只有使用 goto 语句。因此第 22 行对 j 进行判断，如果符合条件，则跳转到标签为 a 的标记语句，也就是第 28 行代码，这时，代码的控制已经穿越到外层 for 循环中了。虽然内层循环中，j 是小于 8 的数，但是 j 的值绝不会大于 3，这是因为 goto 语句让内层代码没有执行完，最后一次的 j++ 并没有得到执行。这段代码的执行结果如下：

```
内层数值: 0  
内层数值: 1  
现在 j 的值是:  
2  
继续下一轮的循环  
内层数值: 3  
跳转到外层  
当前 i 的值是: 0  
外层开始新一次循环  
内层数值: 0  
内层数值: 1  
现在 j 的值是:  
2  
继续下一轮的循环  
内层数值: 3  
跳转到外层  
当前 i 的值是: 1  
外层开始新一次循环  
内层数值: 0  
内层数值: 1  
现在 j 的值是:  
2  
继续下一轮的循环  
内层数值: 3  
跳转到外层  
当前 i 的值是: 2  
外层开始新一次循环  
内层数值: 0  
内层数值: 1  
现在 j 的值是:  
2  
继续下一轮的循环
```

```

内层数值: 3
跳转到外层
当前 i 的值是: 3
外层开始新一次循环
内层数值: 0
内层数值: 1
现在 j 的值是:
2
继续下一轮的循环
内层数值: 3
跳转到外层
当前 i 的值是: 4
外层开始新一次循环

```

4.13 return 语句

return 语句的作用是将控制返回到当前方法的调用方。例如前面讲过的 Main 方法，如果 Main 方法使用关键字 void 修饰，则表明它不需要返回值，这时可以使用 return 语句，也可以不使用 return 语句。但是前面如果有一个返回值，则需要使用 return 语句指明返回值。return 语句有两个重要的特点：

1. 只能用在方法中
2. 可以返回值，也可以不返回值

因为目前只接触过 Main 方法，所以这里再次论述 Main 方法的四种形式，从而阐明 return 语句在方法中的用法。代码如下：

```

01  using System;
02  namespace Statement_Show
03  {
04      class Program
05      {
06          static int Main()
07          {
08              int a = 33;
09              int b = 34;
10              Console.ReadKey();
11              return a + b;
12          }
13      }
14  }

```

这段代码的第 6 行为 Main 方法定义了必须有返回值，返回值的类型为 int。第 8 行和第 9 行代码定义了两个整型变量，第 11 行使用 return 语句返回了 a 和 b 的和。如果 return 语句后跟了运算表达式，则表示它将返回一个值。return 语句还有一个重要作用就是转移代码的控制流程，只要遇到 return 语句，代码的控制就转移了。所以，如果第 10 行代码的语句位于 return 语句的后面，它将得不到执行。

从 Main 方法的返回值来区别的话，它有两种形式：

```

static int Main()
static void Main()

```

当 Main 方法的返回值类型是 void 时，表示 Main 方法无返回值，这时可以使用 return 语句，它的形式是：

```

return;

```

return 语句后不跟任何表达式，程序的控制直接就返回给调用方了。也可以不使用 return 语句。Main 方法到最后一条语句之后就直接结束了。

当 Main 方法有返回值时，必须使用 return 语句返回一个 int 类型的值。return 语句可以跟一个 int 类型的文本值，也可以跟一个表达式，如下：

```
return 33;
return 33 + 1;
return a + b; //后跟运算表达式
```

但是，Main 方法的返回值必须是 int 类型，不可以是其它类型。比如，如果代码如下：

```
01 using System;
02 namespace Statement_Show
03 {
04     class Program
05     {
06         static float Main()
07         {
08             Console.ReadKey();
09             return 33.33F;
10         }
11     }
12 }
```

这段代码的第 6 行将 Main 方法的返回值类型定义成 float 类型。第 9 行返回一个 float 类型的文本值。这时，编译器会报告错误，如下：

“Statement_Show.Program.Main()” 的签名错误，不能作为入口点
程序 Statement_Show.exe 不包含适合于入口点的静态 “Main” 方法

4.14 string 类型

为何要在语句这一章中介绍 string 类型呢？因为 string 类型和要介绍的 Main 方法息息相关。前面的代码实例已经使用过 string 类型，string 类型是一个引用类型，它从 object 类型直接继承。string 类型的对象是一个 Unicode 字符串，string 关键字是 .Net 类库中的 System.String 类的别名。string 类型对象的定义也不是必须用 new 运算符。它用一个字符串文本直接赋值即可。字符串文本有两种形式，一种是常规字符串，另一种是原义字符串。两种字符串文本对字符串变量的定义方式如下：

```
01 using System;
02 namespace Statement_Show
03 {
04     class Program
05     {
06         static void Main()
07         {
08             string a = "寸寸微云，丝丝残照，有无明灭难消。";
09             string b = "\u0045";
10             string c = "\x0050";
11             string d = "\"看小小双脚，袅袅无聊。\"";
12             string e = @"          念奴娇·长千里
13                 逶迤曲巷，在春城斜角，绿杨荫里。
14                 赭白青黄墙砌石，门映碧溪流水。
```



```

15             细雨钗箫，斜阳牧笛，一径穿桃李。
16             风吹花落，落花风又吹起。";
17         Console.WriteLine(a);
18         Console.WriteLine(b);
19         Console.WriteLine(c);
20         Console.WriteLine(d);
21         Console.WriteLine(e);
22         Console.ReadKey();
23     }
24 }
25 }

```

string 类型常规字符串由一对双引号中的 0 个或多个字符组成。如同这段代码的第 8 行。

string 类型也和 char 类型一样，可以使用 Unicode 转义序列、十六进制转义序列和简单转义序列。第 9 行到第 11 行代码是这三种定义形式。和 char 类型不同的是，它们都需要双引号进行包裹。

第 12 行代码开始是一个原义字符串定义，它由一个@符号开始，在@符号之后的文本将保留原有格式。并且不处理任何 Unicode 转义序列、十六进制转义序列和简单转义序列，还可跨越多行。这段代码的执行结果如下：

```

寸寸微云，丝丝残照，有无明灭难消。
E
P
"看小小双卿，袅袅无聊。"
        念奴娇·长干里
        逶迤曲巷，在春城斜角，绿杨荫里。
        赭白青黄墙砌石，门映碧溪流水。
        细雨钗箫，斜阳牧笛，一径穿桃李。
        风吹花落，落花风又吹起。

```

string 类型有一个特点，当在一个程序中的两个变量使用了相同的字符串后，它们可能引用的是同一个实例，代码如下：

```

01 using System;
02 namespace Statement_Show
03 {
04     class Program
05     {
06         static void Main()
07         {
08             string a = "寸寸微云，丝丝残照，有无明灭难消。";
09             string b = "寸寸微云，丝丝残照，有无明灭难消。";
10             Console.WriteLine(a == b);
11             Console.WriteLine(a.GetHashCode());
12             Console.WriteLine(b.GetHashCode());
13             Console.ReadKey();
14         }
15     }
16 }

```

在这段代码中，第 8 行和第 9 行都定义了一个 string 类型的变量，并且用相同的文本初始化了这两个变量。但是在实际内存分配上，并没有在托管堆中分配两块内存空间。而是这两个变量指向了同一块内存。判断两个变量是不是指向同一块内存，string 类型可以使用 == 运算符来判断。如果返回 true，则为同一个字符串。因为是同一个字符串，所以它们的 GetHashCode 也相同。这是因为 string 类型的 GetHashCode 方法为相同的字符串值返回相同的 GetHashCode。这段代码的执行结果如下：

```
True
1076658156
1076658156
```

string 类型还有一个特点是 string 类型的不可变性。代码实例如下：

```
01 using System;
02 namespace Statement_Show
03 {
04     class Program
05     {
06         static void Main()
07         {
08             string myString1 = "寒雨连江夜入吴，平民送客楚山孤。洛阳亲友如相问，";
09             //实际生成了一个新的字符串，但因没有变量引用它，所以将被回收
10             myString1.Insert(myString1.Length, "一片冰心在玉壶。");
11             //原来的字符串并未改变
12             Console.WriteLine(myString1);
13             //使用一个变量引用这个新的字符串
14             string myString2 = myString1.Insert(myString1.Length, "一片冰心在玉壶。");
15             Console.WriteLine(myString1);
16             Console.WriteLine(myString2);
17             Console.ReadKey();
18         }
19     }
20 }
```

第 8 行代码定义了一个字符串，然后第 10 行代码调用了 string 类型的 Insert 方法，将一个字符串插在了原来字符串的结尾。Insert 方法的作用是返回一个变化后的字符串。但是第 10 行代码没有使用一个 string 类型的变量来接收它，所以这个新生成的实例随后将被垃圾回收器回收内存。

第 12 行打印原来的字符串变量 myString1，这时可以发现原来的字符串没有发生变化。第 14 行定义了一个 string 类型的变量，然后再次调用 Insert 方法。第 15 行和第 16 行打印 myString1 和 myString2 的内容，可以发现它们是不同的字符串。这段代码的执行结果如下：

```
寒雨连江夜入吴，平民送客楚山孤。洛阳亲友如相问，
寒雨连江夜入吴，平民送客楚山孤。洛阳亲友如相问，一片冰心在玉壶。
寒雨连江夜入吴，平民送客楚山孤。洛阳亲友如相问，一片冰心在玉壶。
```

这个过程用图来说明，如图 4-1 所示。

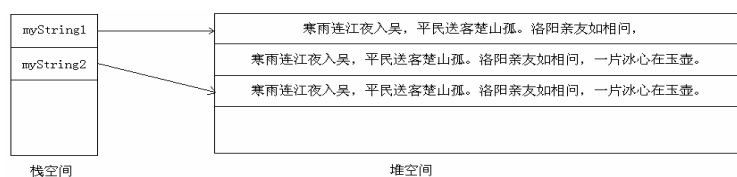


图 4-1 内存空间

在堆内存的第二段，因为没有栈空间的变量引用它，所以它分配完内存后，就等待着被回收，而原来 myString1 引用的空间的字符串并没变化。当再次调用 Insert 方法后，将在堆中再次申请一块内存，并存入新的字符串。而 myString1 引用的内存还是没有变化。这块新的内存将由 myString2 引用。因此可以看出，string 类型是一种不可改变的字符串类型，它的字符串变化都需要申请资源，因此效率较低。

4.15 StringBuilder 类

针对 string 类型的上述缺陷，.Net 类库提供了一个 StringBuilder 类以供使用。它位于 System.Text 命名空间中。当修改这个类字符串时，修改的都是字符串在堆中的本体，而不是产生新的对象。所以效率较高。代码实例如下：

```

01  using System;
02  using System.Text;
03  namespace Statement_Show
04  {
05      class Program
06      {
07          static void Main()
08          {
09              //使用构造方法创建一个默认容量 16 个字符的字符串对象
10              StringBuilder myStringBuilder1 = new StringBuilder();
11              //使用 Append 方法追加一个字符串，但是这个方法不换行，需要加换行符
12              myStringBuilder1.Append("《再别康桥》\n");
13              //使用 AppendLine 方法追加一个字符串，这个方法有换行功能
14              myStringBuilder1.AppendLine("轻轻的我走了");
15              myStringBuilder1.AppendLine("正如我轻轻的来");
16              myStringBuilder1.AppendLine("我轻轻的招手");
17              myStringBuilder1.AppendLine("作别西天的云彩");
18              //要想打印，必须调用其 ToString 方法
19              Console.WriteLine(myStringBuilder1.ToString());
20              //其他一些成员方法
21              //取得字符串的字符数
22              Console.WriteLine("在这首诗里共有 {0} 个字符", myStringBuilder1.Length);
23              //取得字符串的最大容量，最大容量是 int 类型的最大值个字符
24              Console.WriteLine("这个字符串的最大容量是 {0} 个字符",
25                  myStringBuilder1.MaxCapacity);
26              string myString = @"《再别康桥》徐志摩
27                  轻轻的我走了，
28                  正如我轻轻的来；
29                  我轻轻的招手，
30                  作别西天的云彩。
31                  那河畔的金柳，
32                  是夕阳中的新娘；
33                  波光里的艳影，
34                  在我的心头荡漾。";
35              //清空字符串
36              myStringBuilder1.Clear();

```

```

36      Console.WriteLine("清空 myStringBuilder1 后，打印试试看：{0}",
                          myStringBuilder1.ToString());
37      //从索引 0 开始插入格式化的字符串
38      myStringBuilder1.Insert(0, myString);
39      Console.WriteLine(myStringBuilder1.ToString());
40      //定义一个字符数组，初始化为 myStringBuilder1 的长度
41      char[] myChar = new char[myStringBuilder1.Length];
42      //将 myStringBuilder1 复制到这个字符数组中
43      myStringBuilder1.CopyTo(0, myChar, 0, myStringBuilder1.Length);
44      //初始化一个字符串，使用带初始容量的构造方法
45      StringBuilder myStringBuilder2 = new StringBuilder(300);
46      //采用方法重载中的将字符数组插入字符串的方法
47      myStringBuilder2.Insert(0, myChar);
48      Console.WriteLine(myStringBuilder2.ToString());
49      Console.ReadKey();
50  }
51  }
52  }

```

在代码的第 2 行，使用 using 语句引用了 System.Text 命名空间。这是因为 StringBuilder 类位于这个命名空间内。引用这个命名空间可以减少敲击键盘的次数，简化编码。

StringBuilder 类维护一个数据缓冲区，这个数据缓冲区的大小可以由构造方法确定。如果追加的数据不超过数据缓冲区的大小，则数据被追加到数据缓冲区的末尾。如果追加的数据超过了数据缓冲区的大小，则将分配一个新的，更大的缓冲区，原有数据将被复制到这个新的缓冲区中，然后再在其末尾追加数据。因此，它的效率取决于分配缓冲区的次数，如果能估计字符串的大小，则它的效率会更高。这段代码的执行结果如下：

《再别康桥》
轻轻的我走了
正如我轻轻的来
我轻轻的招手
作别西天的云彩

在这首诗里共有 41 个字符

这个字符串的最大容量是 2147483647 个字符

清空 myStringBuilder1 后，打印试试看：

《再别康桥》徐志摩

轻轻的我走了，
正如我轻轻的来；
我轻轻的招手，
作别西天的云彩。
那河畔的金柳，
是夕阳中的新娘；
波光里的艳影，
在我的心头荡漾。

《再别康桥》徐志摩

轻轻的我走了，
正如我轻轻的来；
我轻轻的招手，
作别西天的云彩。
那河畔的金柳，
是夕阳中的新娘；
波光里的艳影，
在我的心头荡漾。

4.16 Main 方法的命令行参数

做了这么多铺垫，终于进入了主题。前面介绍过，Main 方法根据返回值的不同可以分为两种。前面的所有 Main 方法的参数列表也都为空。实际上，Main 方法依据它的方法参数，可以为程序传递参数。这个参数是 string 类型的数组。实例代码如下：

```
01 using System;
02 namespace Statement_Show
03 {
04     class Program
05     {
06         static void Main(string[] args)
07         {
08             foreach (string s in args)
09             {
10                 if (s == "-help")
11                 {
12                     Console.WriteLine(@"用法: ping [-t] [-a] [-n count] [-l size] [-f] [-i
TTL] [-v TOS]
13 [-r count] [-s count] [[-j host-list] | [-k host-list]]
14 [-w timeout] [-R] [-S srcaddr] [-4] [-6] target_name ");
15                     Console.WriteLine("选项:");
16                     Console.WriteLine(@"-t           Ping 指定的主机，直到停止。
17 若要查看统计信息并继续操作 - 请键入 Control-Break;
18 若要停止 - 请键入 Control-C。
19 -a           将地址解析成主机名。
20 -n count     要发送的回显请求数。
21 -l size      发送缓冲区大小。
22 -f           在数据包中设置“不分段”标志(仅适用于 IPv4)。
23 -i TTL       生存时间。
24 -v TOS       服务类型(仅适用于 IPv4。该设置已不赞成使用，且
25 对 IP 标头中的服务字段类型没有任何影响)。
26 -r count     记录计数跃点的路由(仅适用于 IPv4)。
27 -s count     计数跃点的时间戳(仅适用于 IPv4)。
28 -j host-list 与主机列表一起的松散源路由(仅适用于 IPv4)。
29 -k host-list 与主机列表一起的严格源路由(仅适用于 IPv4)。
30 -w timeout   等待每次回复的超时时间(毫秒)。
31 -R           同样使用路由标头测试反向路由(仅适用于 IPv6)。
```

```

32      -S srcaddr      要使用的源地址。
33      -4              强制使用 IPv4。
34      -6              强制使用 IPv6。");
35          }
36      }
37      Console.ReadKey();
38  }
39 }
40 }

```

这段代码第6行的 Main 方法使用了 string 类型数组的参数。第8行代码使用 foreach 语句对 args 数组进行遍历。

第10行代码对遍历到的元素进行判断，如果发现有“-help”字符串，则打印从12行到34行的字符串。这段输出模仿了 windows 7 系统的 Ping 程序的输出。可以发现，实际上，命令行可以使用的那些程序的参数就是用 Main 方法的参数来实现的。这段代码的执行结果如下：

```

Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\Administrator>e:

E:\>Statement_Show.exe -help
用法: ping [-t] [-a] [-n count] [-l size] [-f] [-i TTL] [-v TOS]
          [-r count] [-s count] [[-j host-list] | [-k host-list]]
          [-w timeout] [-R] [-S srcaddr] [-4] [-6] target_name

选项:
    -t          Ping 指定的主机，直到停止。
                若要查看统计信息并继续操作 - 请键入 Control-Break;
                若要停止 - 请键入 Control-C。
    -a          将地址解析成主机名。
    -n count    要发送的回显请求数。
    -l size     发送缓冲区大小。
    -f          在数据包中设置“不分段”标志(仅适用于 IPv4)。
    -i TTL      生存时间。
    -v TOS      服务类型(仅适用于 IPv4。该设置已不赞成使用，且
                对 IP 标头中的服务字段类型没有任何影响)。
    -r count    记录计数跃点的路由(仅适用于 IPv4)。
    -s count    计数跃点的时间戳(仅适用于 IPv4)。
    -j host-list 与主机列表一起的松散源路由(仅适用于 IPv4)。
    -k host-list 与主机列表一起的严格源路由(仅适用于 IPv4)。
    -w timeout  等待每次回复的超时时间(毫秒)。
    -R          同样使用路由标头测试反向路由(仅适用于 IPv6)。
    -S srcaddr  要使用的源地址。
    -4          强制使用 IPv4。
    -6          强制使用 IPv6。

```

至此，对于 Main 方法的论述已经比较全面了。

第五章 方法

5.1 什么是方法

方法也叫做函数，它是能够完成一定功能的语句块。在 C#中，方法是类的一种成员，在类外没有独立的方法。要想调用方法，有两种途径，一种是通过类名调用属于类的静态方法；另一种是通过类的实例调用属于实例的成员方法。方法具有一个参数列表或空列表，它代表可以传递给方法的值或变量引用。在定义时，它也叫做形参列表。传递给方法的实际参数叫做实参。方法还有一个返回值，它是这个方法计算的结果或直接指定的返回结果。如果方法不返回值，则返回类型用 void 指定。每个方法都有一个签名，顾名思义，同每个人的签名一样，它唯一的代表这个方法。方法的签名由方法名称、参数的类型、参数的数量和修饰符组成，特别需要注意的是，方法的签名不包括方法的返回类型。

前面介绍过 Main 方法，下面看代码实例：

```
01 using System;
02 namespace Method_Show
03 {
04     class Program
05     {
06         static void Main(string[] args)
07         {
08             //调用 Test1 方法，将返回值赋给 sum
09             int sum = Program.Test1(3, 2);
10             Console.WriteLine("调用 Test1 方法返回的值是：{0}", sum);
11             Console.ReadKey();
12             return;
13         }
14         /// <summary>
15         /// 求两个整数的和
16         /// </summary>
17         /// <param name="a"></param>
18         /// <param name="b"></param>
19         /// <returns></returns>
20         static int Test1(int a, int b)
21         {
22             return a + b;
23         }
24     }
25 }
```

这段代码在 Main 方法的基础上又定义了一个新的方法，从第 14 行到第 19 行是这个方法的 XML 风格的注释。

第 20 行是方法的定义。在 C#中，类内的成员定义顺序不是必要的。因此，这个方法定义在 Main 方法后面，一样能得到 Main 方法的调用。这个新定义的方法用 static 关键字作修饰，代表它是属于类的。在类外部可以只使用类名进行调用。方法返回一个 int 类型的返回值，方法的名称为 Test1，在方法名称后面的一对括号内是方法参数列表。方法的参数是要传递给方法使用的局部变量，这里是两个 int 类型的变量。在方法参数列表后就是方法体，它是方法要执行的操作，它以一对大括号开始和结束，结尾没有分号。在方法体中是方法要执行的语句列表。这个方法执行的操作是返回局部变量 a 和 b 的和。

第 9 行代码定义了一个 `int` 类型的变量 `sum` 来获得方法的返回值。为了说明其用法，在方法前加上了类名，因为这是一个类的内部的方法间的相互调用，因此不加类名也可以。这个 `Test1` 方法含有两个整型参数，也就是说，调用它必须给它传入两个整型的参数。这里传入的是 3 和 2，它们之间用逗号隔开，在传入的时候不必使用类型关键字，类型关键字是只在方法定义中使用的，方法的定义和类一样，也可以看做是一个模板。

第 10 行代码打印 `sum` 值，这里调用 `WriteLine` 方法的时候，使用了占位符，占位符用一对大括号开始和结束，里面的占位符号是从 0 开始，逗号后接第一个要打印的元素，如果有多个，依次用逗号隔开。这段代码的执行结果如下：

调用 `Test1` 方法返回的值是：5

5.2 方法的格式化输出

在上一小节中已经涉及到了一个控制台输出的格式化操作，它使用了“`{0}`”这样的操作符。在这一小节中将通过两个实例来演示一下控制台输出的格式化以及 Windows 应用程序的窗口输出格式化操作。下面先看控制台输出格式化的代码：

```
01 using System;
02 namespace Method_Show
03 {
04     class Program
05     {
06         static void Main(string[] args)
07         {
08             //占位符的格式化输出操作
09             Console.WriteLine("师徒{0}人去西天取经，经历了九九{1}难，打倒了{2}多妖怪，最后取到了真经，他们{0}人团结友爱的精神值得赞扬！\n", 4, 81, 2000);
10             Console.WriteLine("明月{0}有，把酒问{1}”，“几时”，“青天”);
11             //数值数据的格式化操作
12             //1. 货币的格式化输出
13             Console.WriteLine("货币的格式化操作，格式为占位符后跟“:”，再跟大写的“C”或小写的“c”再跟小数位数：{0:C3}\n", 123456);
14             //2. 十进制数的格式化操作
15             Console.WriteLine("十进制数的格式化操作，格式为占位符后跟“:”，再跟大写的“D”或小写的“d”再跟整数位数：{0:D8}\n", 123456);
16             //3. 指数计数法的格式化操作
17             Console.WriteLine("指数的格式化操作，格式为占位符后跟“:”，再跟大写的“E”或小写的“e”再跟小数位数：{0:E3}\n", 123456);
18             //4. 定点小数的格式化操作
19             Console.WriteLine("定点小数的格式化操作，格式为占位符后跟“:”，再跟大写的“F”或小写的“f”再跟小数位数：{0:F4}\n", 123456);
20             //5. 通用格式的格式化操作，要求转换后的字符串最短，通用格式可能采用科学计数法//或浮点数据类型格式
21             Console.WriteLine("通用格式的格式化操作，格式为占位符后跟“:”，再跟大写的“G”或小写的“g”再跟数据位数：{0:G5}\n", 123.456);
22             //6. 基本数值格式化操作，特点是整数部分每三位用“,”分隔
23             Console.WriteLine("基本数值的格式化操作，格式为占位符后跟“:”，再跟大写的“N”或小写的“n”再跟小数位数：{0:N4}\n", 123456);
```



```

24          //7. 十六进制格式化操作，如果使用大写的 X, 则输出的十六进制也会用大写字符，如
           //果用小写 x, 则输出也会用小写 x
25          Console.WriteLine("十六进制的格式化操作，格式为占位符后跟“:”，再跟大写的“X”
           或小写的“x”再跟数据位数: {0:X8}\n", 123456);
26          Console.WriteLine("十六进制的格式化操作，格式为占位符后跟“:”，再跟大写的“X”
           或小写的“x”再跟数据位数: {0:x8}\n", 123456);
27          //8. 百分数格式化操作
28          Console.WriteLine("百分数的格式化操作，格式为占位符后跟“:”，再跟大写的“P”
           或小写的“p”再跟小数位数: {0:P3}\n", 134);
29          Console.ReadKey();
30      }
31  }
32  }

```

WriteLine 方法中的占位符形式是用大括号括起来的数字，数字从 0 开始，对应字符串后面的参数。参数可以是数字也可以是字符串，其实在运行的时候都是转成的字符串。占位符可以重复使用，比如前面用了 {0}，后面还可以用这个占位符，它对应的都是后跟的第一个参数。

在占位符中的数字加上冒号再跟特定意义的字母，则可以输出不同形式的数据，而且可以指定数据位数或小数位数。这段代码的执行结果如下：

师徒 4 人去西天取经，经历了九九 81 难，打倒了 2000 多妖怪，最后取到了真经，他们 4 人团结友爱的精神值得赞扬！

明月几时有，把酒问青天

货币的格式化操作，格式为占位符后跟“:”，再跟大写的“C”或小写的“c”再跟小数位数：
¥ 123, 456. 000

十进制数的格式化操作，格式为占位符后跟“:”，再跟大写的“D”或小写的“d”再跟整数位数：
00123456

指数的格式化操作，格式为占位符后跟“:”，再跟大写的“E”或小写的“e”再跟小数位数：
1. 235E+005

定点小数的格式化操作，格式为占位符后跟“:”，再跟大写的“F”或小写的“f”再跟小数位数：
123456. 0000

通用格式的格式化操作，格式为占位符后跟“:”，再跟大写的“G”或小写的“g”再跟数据位数：
123. 46

基本数值的格式化操作，格式为占位符后跟“:”，再跟大写的“N”或小写的“n”再跟小数位数：
123, 456. 0000

十六进制的格式化操作，格式为占位符后跟“:”，再跟大写的“X”或小写的“x”再跟数据位数：
0001E240

十六进制的格式化操作，格式为占位符后跟“:”，再跟大写的“X”或小写的“x”再跟数

据位数: 0001e240

百分数的格式化操作, 格式为占位符后跟 “:”, 再跟大写的 “P” 或小写的 “p” 再跟小数

位数: 13,400.000%

5.2.1 窗口环境下的格式化输出

现在人们编写的代码通常都是在 Windows 界面下运行的, 如何将格式化后的字符串用窗体表现出来, C#中已经给出了方法。下面建立一个 Windows 窗体应用程序, 在开发环境的项目类型中选择 “Windows 窗体应用程序”。如图 5-1 所示。

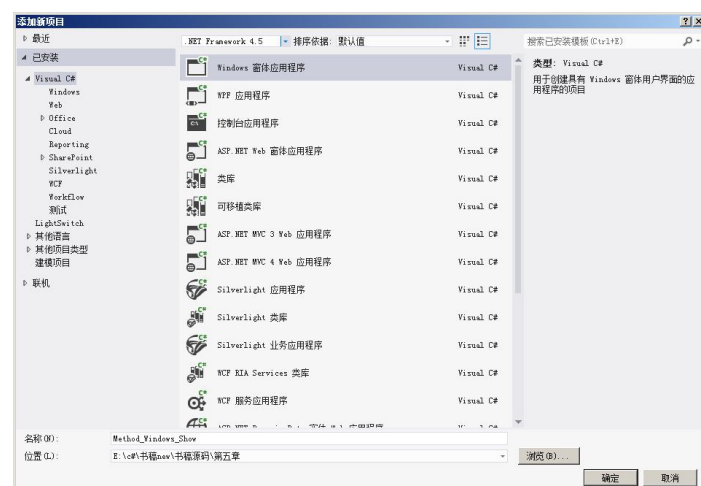


图 5-1 项目类型

在这个窗口中为项目定义一个名称, 然后单击 “确定” 按钮, 如图 5-2 所示。

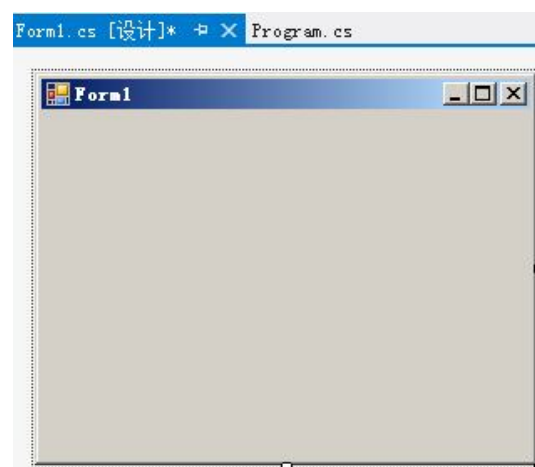


图 5-2 窗体

这时, 在代码编辑区会出现一个空白的窗体, 在代码编辑区的左侧会出现一个控件列表。如图 5-3 所示。

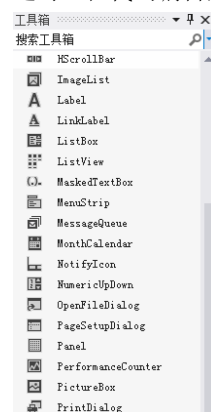


图 5-3 控件列表

在这个控件窗体中点选一下控件“Button”，然后在空白窗体上点一下鼠标左键。这时在空白新建窗体上就会出现一个按钮。或者在“工具箱”中将选择的控件用拖拽的方式拖到空白新建窗体上也有同样的效果。如图 5-4 所示。

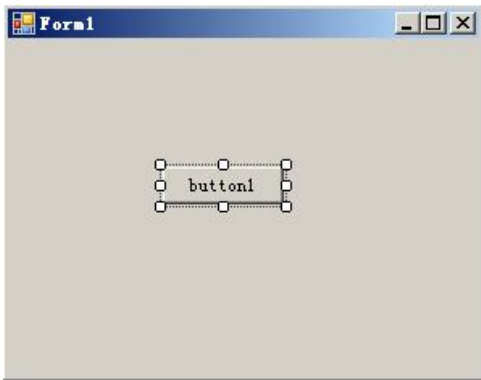


图 5-4 Button 控件

这时，要为这个按钮改一下显示的文字，应该让它显示“确认”或“取消”。如果开发环境现在在窗体编辑窗口的右侧没有“属性”窗体，则在开发环境的菜单中单击“视图”，这时会出现“视图”菜单，如图 5-5 所示。



图 5-5 视图菜单

在这个菜单中单击“属性窗口”，这时在开发环境的“解决方案资源管理器”窗体下方就会出现“属性窗口”，如图 5-6 所示。

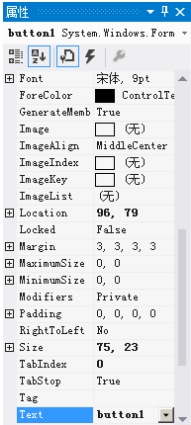


图 5-6 属性窗口

要想改变 Button 控件显示的文字，需要对这个控件的“Text”属性作设置，将它默认的“button1”改变为“输出格式化数据”。输入完毕按一下回车键，就设置完毕了。如果按钮控件不能完全显示所有文字，则用拖拽的方式再更改一下它的大小，使它能完全显示所有文字。如图 5-7 所示。



图 5-7 修改后的控件

现在想达到的效果是，单击这个控件就会弹出一个窗口，在这个窗口上显示格式化的数据。单击这个控件就是一个事件，这个事件将由操作系统传递给应用程序。所以需要为这个事件添加这个事件的处理方法。可以在窗体编辑窗口中双击这个控件产生，也可以在“属性窗体”的事件列表中添加。“属性窗体”有一个“事件”标记，是一个闪电的形状。如图 5-8 所示。

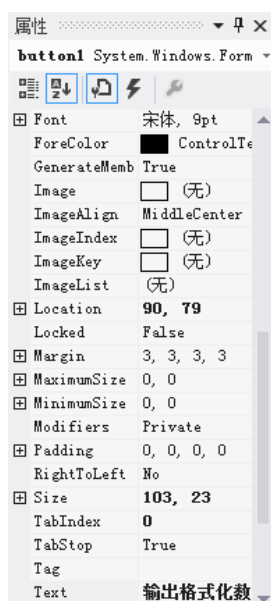


图 5-8 事件标记

点击这个标记就会切换到事件列表中，如图 5-9 所示。

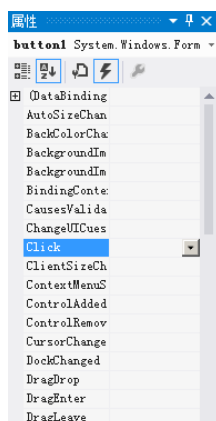


图 5-9 事件列表

在这个事件列表中的每个事件，如果选中它，在窗口下方都会有说明文字。在其中选择“MouseDown”事件。这个事件就是当鼠标单击控件时发生。双击它右侧的空白框，就会添加进一个事件处理方法。这时会自动出现代码编辑器窗口，且定位在这个事件处理方法上。修改这个事件处理方法，代码如下：

```
01 using System;
02 using System.Collections.Generic;
03 using System.ComponentModel;
04 using System.Data;
05 using System.Drawing;
06 using System.Linq;
07 using System.Text;
08 using System.Threading.Tasks;
09 using System.Windows.Forms;
10
11 namespace Method_Windows_Show
12 {
13     public partial class Form1 : Form
14     {
15         public Form1()
16         {
17             InitializeComponent();
18         }
19
20         private void button1_MouseClick(object sender, MouseEventArgs e)
21         {
22             //首先定义一个 string 类型的变量用来存储格式化字符串。而字符串的格式化操作由
23             //string 类的 Format 方法来完成。
24             string outputMessage0 = string.Format("师徒{0}人去西天取经，经历了九九{1}难，
25             打倒了{2}多妖怪，最后取到了真经，他们{0}人团结友爱的精神值得赞扬！\n", 4, 81,
26             2000);
27             string outputMessage1 = string.Format("明月{0}有，把酒问{1}\n", "几时", "青
28             天");
29             string outputMessage2 = string.Format("货币的格式化操作，格式为占位符后跟“:”，
30             再跟大写的“C”或小写的“c”再跟小数位数: {0:C3}\n", 123456);
31             string outputMessage3 = string.Format("十进制数的格式化操作，格式为占位符后
32             跟“:”，再跟大写的“D”或小写的“d”再跟整数位数: {0:D8}\n", 123456);
33             //再使用一个 string 类型的变量将格式化后的字符串串联起来
34             string concordancy = outputMessage0 + outputMessage1 + outputMessage2 +
35             outputMessage3;
36             MessageBox.Show(concordancy);
37         }
38     }
39 }
```

当从窗体设计窗口想要切换到代码编辑窗口时，按 F7 键，再想切换回来，按 Shift+F7 键。程序运行结果如图 5-10 所示。



图 5-10 主窗体

当单击“输出格式化数据”按钮时，会弹出另一个窗体，上面显示格式化的数据。如图 5-11 所示。



图 5-11 格式化数据

如果当开发环境中存在多个项目时，有可能按 F5 键进行调试时，打开的不是当前编辑的项目，这是因为没把当前项目设为启动项目的原因。在当前项目名称上单击右键，然后在出现的菜单上选择“设为启动项目”即可。

5.3 值形参

首先来介绍一下什么是方法的形参，什么是实参。形参就是方法的定义中，参数列表中的参数。它通常指定参数类型和参数名称。它是方法模板的组成部分。而实参就是在方法调用中实际传递给方法的参数。它是实际存在的参数，所以叫做实参。

在第一小节中的代码中，方法的形参只有参数类型和名称，在参数类型前面没有任何修饰关键字，这样的形参叫做值形参。下面通过一个图来说明其内存分布，如图 5-12 所示。

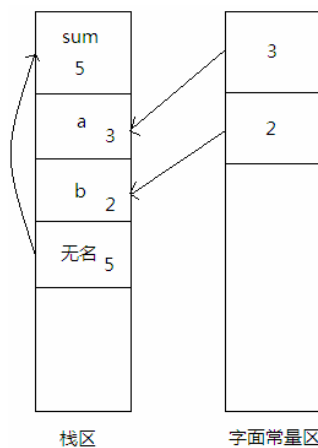


图 5-12 内存分布图

在代码执行过程中，首先在栈中会先定义一个变量 sum，Test1 的方法有两个形参 a 和 b，这时也会在栈中分配内存。然后文本值 3 和 2 将被复制给 a 和 b。然后方法进行运算，将结果返回。这个返回的结果没有名字，但是依然在内存栈中分配内存，最后将这个无名变量赋给最先分配的变量 sum。

值形参也叫值参数。可以看到，一个值形参相当于一个局部变量。它的初始值来自于为这个形参传递值的实参的复制，这也叫做按值传递。下面再通过一段代码来说明值参数的特点：

```
01 using System;
02 namespace Method_Show
03 {
```

```

04     class Program
05     {
06         static void Main(string[] args)
07         {
08             int a1 = 1, b1 = 2;
09             Console.WriteLine("调用 Test1 方法前, a1 值是: {0} b1 的值是: {1}", a1, b1);
10             //调用 Test1 方法
11             Test1(a1, b1);
12             Console.WriteLine("调用 Test1 方法后, a1 值是: {0} b1 的值是: {1}", a1, b1);
13             Console.ReadKey();
14         }
15         /// <summary>
16         /// 两个参数进行自加
17         /// </summary>
18         /// <param name="a"></param>
19         /// <param name="b"></param>
20         /// <returns></returns>
21         static void Test1(int a, int b)
22         {
23             a++;
24             b++;
25             Console.WriteLine("Test1 方法内, a 值是: {0} b 的值是: {1}", a, b);
26         }
27     }
28 }

```

首先来看代码的执行结果, 如下:

调用 Test1 方法前, a1 值是: 1 b1 的值是: 2

Test1 方法内, a 值是: 2 b 的值是: 3

调用 Test1 方法后, a1 值是: 1 b1 的值是: 2

第 8 行代码首先定义了两个局部变量, 并分别赋值为 1 和 2。第 9 行打印这两个值。

第 11 行代码调用 Test1 方法, 并且将这两个变量传递给方法。

第 21 行开始是方法的定义, 可以看到, 在方法中对传入的参数进行了自增操作。然后打印自增后的变量值。因为是按值传递, 所以 Main 方法中的 a1 和 b1 是用复制的方式传递给方法的。因此在方法中打印参数 a 和 b 的值时, 可以看到值发生了变化。

但是在第 12 行再次打印 a1 和 b1 的值时, 却发现值还是原来的值, 没有发生任何变化。这是因为方法中的参数值是 a1 和 b1 的克隆体的关系。方法内操作的只是原来局部变量的副本。当方法退出作用域, 也就是方法的块时, 这两个局部变量, 也就是 a1 和 b1 就被销毁了。

5.4 引用形参

引用形参也叫做引用参数。引用形参在方法的形参类型前有一个关键字 ref。引用形参和值形参不同, 它不创建新的存储位置。也就是说, 方法执行时, 它不为它的引用形参在栈中创建存储位置。传递给方法的引用形参必须经过初始化。下面看代码实例:

```

01     using System;
02     namespace Method_Show
03     {

```

```

04     class Program
05     {
06         static void Main(string[] args)
07         {
08             int a1 = 1, b1 = 2;
09             Console.WriteLine("调用 Test1 方法前, a1 值是: {0} b1 的值是: {1}", a1, b1);
10             //调用 Test1 方法并按引用参数传递变量, 前面需加 ref 关键字
11             Test1(ref a1, ref b1);
12             Console.WriteLine("调用 Test1 方法后, a1 值是: {0} b1 的值是: {1}", a1, b1);
13             Console.ReadKey();
14         }
15         static void Test1(ref int a, ref int b)
16         {
17             a++;
18             b++;
19             Console.WriteLine("Test1 方法内, a 值是: {0} b 的值是: {1}", a, b);
20         }
21     }
22 }

```

这段代码和前一小节的代码很相似, 不同的是, 方法的形参前加上了 ref 参数。

如果要为方法传递 ref 类型的参数, 那么这个参数必须经过初始化。第 8 行代码对要传递给方法的变量做了初始化操作。

第 15 行代码是方法的定义, 可以看到, 形参 a 和 b 的参数类型前都加上了 ref 关键字。在方法体中还是对它们进行自增操作。

第 11 行代码调用这个方法, 并且为其传递参数, 传递参数的时候要注意, 参数前同样要加 ref 关键字。

第 12 行代码打印了方法执行后, 变量 a1 和 b1 的值, 这时能观察到, 它们的值已经发生了变化。这段代码的执行结果如下:

```

调用 Test1 方法前, a1 值是: 1 b1 的值是: 2
Test1 方法内, a 值是: 2 b 的值是: 3
调用 Test1 方法后, a1 值是: 2 b1 的值是: 3

```

下面还是通过一个内存图来说明引用参数的原理, 如图 5-13 所示。

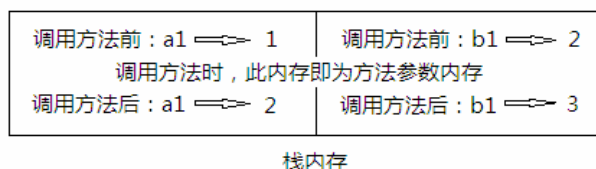


图 5-13 内存分布

首先定义两个局部变量并初始化为 1 和 2, 它们都分配在栈内存中。在定义方法时, 它的两个参数是引用形参, 它实际上不分配内存, 而是直接使用定义的局部变量 a1 和 b1 那块内存。然后, 当在方法中对变量操作时, 实际操作的是最开始定义好的两个变量 a1 和 b1。引用参数的作用是把对变量的操作由方法的内部扩展到了方法的外部, 也就是它对方法外部的变量产生了作用。和值参数相同的是, 在传递外部的变量时, 外部的变量必须已经初始化或已经赋值。没有初始化或赋值的变量如果发生值参数或引用参数的传递时会发生编译错误。如下:

```

错误 1    使用了未赋值的局部变量 "a1"
错误 2    使用了未赋值的局部变量 "b1"

```


5.5 输出形参

输出形参也叫输出参数。它和引用形参非常相似，它们都不创建新的存储位置，而是使用要传递给它们的变量的存储位置。它们的内存分配图和引用形参是一样的。它和引用参数不同的是，为它传递的实参可以不初始化，但是在方法体内必须得到初始化或赋值，而且在使用前必须赋值。也就是说，在方法体内部，首先会认为传入的输出参数是未被初始化的，因此在方法体调用结束，返回的时候，所有输出参数必须已经被初始化或赋值。因为一个方法只能有一个返回值，所以，引用参数和输出参数都用于实现多个返回值的操作。输出参数的修饰符关键字是 `out`，它位于形参类型之前。在向方法传递参数的时候，也必须加 `out` 关键字。实例代码如下：

```

01  using System;
02  namespace Method_Show
03  {
04      class Program
05      {
06          static void Main(string[] args)
07          {
08              int a1, b1;
09              //调用 Test1 方法并按输出参数传递变量，前面需加 out 关键字
10              Test1(out a1, out b1);
11              Console.WriteLine("调用 Test1 方法后, a1 值是: {0} b1 的值是: {1}", a1, b1);
12              Console.ReadKey();
13          }
14          static void Test1(out int a, out int b)
15          {
16              a = 1;
17              b = 2;
18              a++;
19              b++;
20              Console.WriteLine("Test1 方法内, a 值是: {0} b 的值是: {1}", a, b);
21          }
22      }
23  }

```

可以看到，输出参数的用法和引用参数的用法差不多，不同就是传递进方法前不需要初始化操作。但是在代码的第 18 行和第 19 行对变量自增前，必须对变量进行初始化操作。这段代码的执行结果如下：

```

Test1 方法内, a 值是: 2 b 的值是: 3
调用 Test1 方法后, a1 值是: 2 b1 的值是: 3

```

5.6 形参数组

最后一种形参类型叫做形参数组，也叫做参数数组。参数数组的作用是可以向方法传递可变数量的参数。这些参数都可被认为是这个参数数组的成员。参数数组用 `params` 关键字修饰。如果方法有多个参数，那么，参数数组必须位于参数表的最后一位。参数数组必须是一维数组形式，例如“`int[]`和`int[][]`”都可以用作参数数组，但是“`int[,]`”这样的多维数组不能用作参数数组。当传递给方法可变参数时，这些可变参数可以是零个参数、单个参数、表达式或多个参数，最后一点是，`params` 关键字不能和 `ref` 及 `out` 关键字联用。下面看代码实例：

```

01  using System;
02  namespace Method_Show

```

```
03  {
04      class Program
05      {
06          static void Test1(ref int a, ref int b, params string[] c)
07          {
08              a++;
09              b++;
10              foreach (string s in c)
11              {
12                  Console.WriteLine("参数数组的各个元素是: {0}", s);
13              }
14          }
15          static int Test2(int a, int b, params int[][] c)
16          {
17              int sum = a + b;
18              foreach (int[] i in c)
19              {
20                  foreach (int j in i)
21                  {
22                      sum += j;
23                  }
24              }
25              return sum;
26          }
27          static void Main(string[] args)
28          {
29              int a = 1;
30              int b = 2;
31              Test1(ref a, ref b, "孙悟空", "猪八戒", "沙和尚");
32              Console.WriteLine("局部变量 a 和 b 的值是: a:{0} b:{1}", a, b);
33              int[] c = new int[3] { 1, 2, 3 };
34              int[] d = new int[2] { 4, 5 };
35              int e = Test2(a, b, c, d);
36              Console.WriteLine("传递给 Test2 的参数的和是: ");
37              Console.WriteLine(e);
38              Console.ReadKey();
39          }
40      }
41  }
```

第 6 行代码定义了一个形参数组，它是 `string` 类型的。当在第 31 行代码为其传入三个字符串时，编译器会首先取得字符串的个数，然后用它来初始化方法的 `string` 类型的数组，然后用传入的三个字符串为其初始化。

第 10 行代码使用 `foreach` 语句遍历打印了数组的各个元素。

第 15 行代码定义了一个交错数组类型的形参数组，因为是一个交错数组，所以要用两层嵌套形式的

foreach 语句来遍历数组的元素。

第 18 行是第一层遍历，它先遍历出每个 int 类型数组的元素，然后第 20 行再遍历出每个 int 类型数组中的元素，然后把每个元素都与 sum 相加，再赋值给 sum。

因为要传递给方法的必须是 int 类型的数组，所以第 33 行和第 34 行定义了两个 int 类型的数组，并且做了初始化。当在第 35 行代码将它传入方法时，编译器会首先计算数组的个数，然后用它初始化形参数组的维数长度。因为有两个数组，所以，维数长度为 2。接下来用数组 c 和 d 初始化方法中的形参数组。这段代码的执行结果如下：

```
参数数组的各个元素是：孙悟空
参数数组的各个元素是：猪八戒
参数数组的各个元素是：沙和尚
局部变量 a 和 b 的值是：a:2 b:3
传递给 Test2 的参数和是：
20
```

5.7 参数默认值（可选参数）

参数的默认值也叫做可选参数。它可以让程序员在传递方法参数时，可以少指定参数的个数。下面用代码实例来说明：

```
01 using System;
02 namespace Method_Show
03 {
04     class Program
05     {
06         static void Test(ref int a, int b = 2)
07         {
08             a = a + b;
09         }
10         static void Main(string[] args)
11         {
12             int a = 3;
13             //此处不传递第二个参数，第二个参数采用默认值
14             Test(ref a);
15             Console.WriteLine("调用 Test 方法后，变量 a 的值是：{0}", a);
16             int b = 5;
17             //这里为方法传递第二个参数
18             Test(ref a, b);
19             Console.WriteLine("在传递第二个参数，调用 Test 方法后，变量 a 的值是：{0}", a);
20             Console.ReadKey();
21         }
22     }
23 }
```

这段代码中，在调用 Test 方法时，并没有传递第二个参数。但是第二个参数采用了默认值并参与了运算。参数可以采用默认值并不代表不可以传递给方法这个参数，在下次调用时，就为它传递了一个参数并参与了运算。这段代码的执行结果如下：

```
调用 Test 方法后，变量 a 的值是：5
在传递第二个参数，调用 Test 方法后，变量 a 的值是：10
```

需要注意的一点是，默认参数不可以是引用参数和输出参数。也就是不可以用 `ref` 和 `out` 关键字修饰，否则报错。而且默认参数的默认值必须是在编译的时候就可以确定，不能在代码的运行时确定。最后，默认参数必须放在方法的参数列表中的最后一项。也就是，采用默认参数的方法参数在参数列表中必须依次往后排，而不能在非默认参数的前面。下面通过代码来说明：

```
01 using System;
02 namespace Method_Show
03 {
04     class Program
05     {
06         /// <summary>
07         /// 想要进行 a+b 的计算，并通过一个 DateTime 结构的属性获得当前时间作为默认参数
08         /// </summary>
09         /// <param name="a"></param>
10         /// <param name="b"></param>
11         /// <param name="myDateTime"></param>
12         static void Test(ref int a, ref int b = 3, DateTime myDateTime = DateTime.Now)
13         {
14             a = a + b;
15             Console.WriteLine("现在的系统时间是: {0}", myDateTime.ToString());
16         }
17         static void Main(string[] args)
18         {
19             int a = 5;
20             int b = 3;
21             Test(ref a, b);
22             Console.WriteLine("调用 Test 方法后, a 的值是: {0}", a);
23             Console.ReadKey();
24         }
25     }
26 }
```

这段代码的本意是让 `b` 有一个默认值的同时还可以引用外部的变量，并且通过一个 `DateTime` 的结构属性获得当前时间作为默认值。但是很不幸，C# 的语法不允许这样做，它将在没有编译时就会报错，如下：

错误 1 `ref` 或 `out` 参数不能有默认值

错误 2 “`myDateTime`” 的默认参数值必须为编译时常量

错误 3 与 “`Method_Show.Program.Test(ref int, ref int, System.DateTime)`” 最匹配的重载方法具有一些无效参数

5.8 命名参数

命名参数的作用是在调用方法时可以以任意的顺序传递参数，而不是像传统的方法调用那样按照参数位置顺序传递参数。下面通过代码来说明命名参数的用法：

```
01 using System;
02 namespace Method_Show
03 {
04     class Program
05     {
```

```
06      static void Test(string myString1, string myString2, string myString3, string
07          myString4)
08      {
09          Console.WriteLine(myString1);
10          Console.WriteLine(myString2);
11          Console.WriteLine(myString3);
12          Console.WriteLine(myString4);
13      }
14      static void Main(string[] args)
15      {
16          //先按照参数的位置顺序依次传递
17          Test("朝辞白帝彩云间", "千里江陵一日还", "两岸猿声啼不住", "轻舟已过万重山
18              ");
19          Console.WriteLine();
20          //按照命名参数方式传递参数
21          Test(myString2: "千里江陵一日还", myString4: "轻舟已过万重山", myString1: "
22              朝辞白帝彩云间", myString3: "两岸猿声啼不住");
23          Console.WriteLine();
24          //如果在传递参数的时候,同时使用了按位置传递,则它必须位于命名参数之前
25          Test("朝辞白帝彩云间", "千里江陵一日还", myString4: "轻舟已过万重山",
26              myString3: "两岸猿声啼不住");
27          Console.ReadKey();
28      }
29  }
```

从代码实例中就可以看到用法,使用命名参数时,需要使用标签语句指明要传递的参数名称。如果同时使用按位置传递与按命名参数传递,则按位置传递的参数必须位于按命名参数传递的参数之前。这段代码的执行结果如下:

```
朝辞白帝彩云间
千里江陵一日还
两岸猿声啼不住
轻舟已过万重山
```

```
朝辞白帝彩云间
千里江陵一日还
两岸猿声啼不住
轻舟已过万重山
```

```
朝辞白帝彩云间
千里江陵一日还
两岸猿声啼不住
轻舟已过万重山
```

在一种情况下,命名参数是非常有用处的,那就是在全部参数都设置了默认参数的情况下,下面还是通过代码实例来说明:

```

01 using System;
02 namespace Method_Show
03 {
04     class Program
05     {
06         static void Test(int a = 3, int b = 4, int c = 5)
07         {
08             Console.WriteLine("a 的值是: {0}    b 的值是: {1}    c 的值是: {2}", a, b, c);
09         }
10         static void Main(string[] args)
11         {
12             Console.WriteLine("都采用默认值调用: ");
13             Test();
14             Console.WriteLine("c 采用值参数传递调用: ");
15             Test(c: 100);
16             Console.ReadKey();
17         }
18     }
19 }

```

在这段代码中定义了一个方法，这个方法的三个参数都带有默认值。现在实际的需要是，只想为第三个参数以值参数传递的方法传递一个值，而第一个和第二个都想采用默认值。这种需求如果没有命名参数就不好办。因为传递进去一个值后，首先会认为是第一个参数的传递值。而通过命名参数就可以指定为具体哪个参数传参。这段代码的执行结果如下：

都采用默认值调用：

a 的值是: 3 b 的值是: 4 c 的值是: 5

c 采用值参数传递调用：

a 的值是: 3 b 的值是: 4 c 的值是: 100

5.9 方法重载

方法重载就是允许一个类中的静态方法或实例方法可以具有相同的方法名称。在实际调用时，根据调用的具体的方法签名来调用不同的方法。在编译的时候，编译器将使用重载决策来查找与被调用的参数最佳匹配的方法，如果没有找到任何最佳的匹配就将报错。下面是代码实例：

```

01 using System;
02 namespace Method_Show
03 {
04     class Program
05     {
06         static void Test(int a)
07         {
08             Console.WriteLine("本次调用是 void Test(int a)方法");
09             Console.WriteLine();
10         }
11         static void Test(int a, int b)
12         {
13             Console.WriteLine("本次调用是 void Test(int a, int b)方法");

```

```
14         Console.WriteLine();
15     }
16     static void Test(int a, params int[] b)
17     {
18         Console.WriteLine("本次调用是 void Test(int a, params int[] b)方法");
19         Console.WriteLine();
20     }
21     static void Test(uint a, uint b)
22     {
23         Console.WriteLine("本次调用是 void Test(uint a, uint b)方法");
24         Console.WriteLine();
25     }
26     static void Test(double a, double b)
27     {
28         Console.WriteLine("本次调用是 void Test(double a, double b)方法");
29         Console.WriteLine();
30     }
31     static void Test(ref int a, ref int b)
32     {
33         Console.WriteLine("本次调用是 void Test(ref int a, ref int b)方法");
34         Console.WriteLine();
35     }
36     static void Main(string[] args)
37     {
38         int a = 3, b = 2;
39         double c = 3.33, d = 1.11;
40         //传入一个 int 类型的变量
41         Test(a);
42         //传入两个 int 类型的文本
43         Test(3, 4);
44         //传入两个 uint 类型的文本值
45         Test(3U, 4U);
46         //传入三个 int 类型的变量和文本
47         Test(a, b, 6);
48         //传入两个 int 类型的变量
49         Test(a, b);
50         //传入两个 double 类型的变量
51         Test(c, d);
52         //按引用参数方法传入两个 int 类型变量
53         Test(ref a, ref b);
54         Console.ReadKey();
55     }
56 }
57 }
```

前面谈到过，方法的签名包括方法的名称、方法的参数类型、方法的参数个数以及方法参数前的修饰符构成。在这段代码中定义的六个方法都具有相同的名称。再看构成方法签名的其它几项，从代码看来，其它几项是完全不同的。这就好像几个人的签名都摆在这里，那么调用的时候就好像在进行字迹的比对，看哪个签名符合要求就调用哪个方法。这里需要特别说明的是，在方法调用时，所传入的参数如果不符合方法定义的参数类型，则编译器首先会尝试进行隐式转换。隐式转换后面会介绍到。当传入文本 3 和 4 时，这两个文本类型默认是 int 类型，因此要想调用包含 uint 类型的方法，则需要制定 U 后缀。当传入包括 2 个以上的参数时，会自动匹配为带有参数数组的方法。当按引用参数传入时，则会调用包含引用参数的方法。这段代码的执行结果如下：

本次调用是 void Test(int a)方法

本次调用是 void Test(int a, int b)方法

本次调用是 void Test(uint a, uint b)方法

本次调用是 void Test(int a, params int[] b)方法

本次调用是 void Test(int a, int b)方法

本次调用是 void Test(double a, double b)方法

本次调用是 void Test(ref int a, ref int b)方法

微软的开发环境带有智能自动提示功能，如图 5-13 所示。

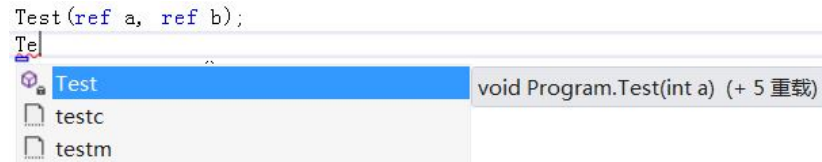


图 5-13 智能提示

在进行方法重载的代码编写时，最容易出现的问题就是将方法的返回值作为了方法的签名。但实际上它并不是方法的签名。下面看出错的代码：

```
01 using System;
02 namespace Method_Show
03 {
04     class Program
05     {
06         static void Test(int a, int b)
07         {
08         }
09         static int Test(int a, int b)
10         {
11             return 3;
12         }
13         static void Test(int a, int b = 3)
14         {
15         }
```



```

16         static void Main(string[] args)
17         {
18             Test(3, 5);
19         }
20     }
21 }

```

这段代码在编写第 6 行到第 15 行的方法定义时，不会提示任何错误。但是在第 16 行的 Main 方法调用中，编译器就会给出错误提示，如下：

在以下方法或属性之间的调用不明确：“Method_Show.Program.Test(int, int)”和“Method_Show.Program.Test(int, int)”

从错误提示中可以看出，是第 6 行和第 9 行的方法签名出现了重复形成了这个错误原因。从开发环境的错误提示中可以看到一个导致错误的原因。接下来注释掉第 9 行的方法，然后再把 Main 方法中的代码修改成如下的代码：

```
Test(3);
```

这时，可以看到的是，错误提示消失了。然后编译这段代码，这时立刻又会出现一个错误。这是个编译时错误，如下：

类型“Method_Show.Program”已定义了一个名为“Test”的具有相同参数类型的成员

这就说明方法的返回值和可选参数不构成方法重载的条件。

ref 修饰符和 out 修饰符虽然是方法签名的组成部分，但是还有一个限制，单单凭借这两个修饰符并不能构成方法的重载。还有一个出现在方法参数中的修饰符就是 params，它不是方法签名的一部分，也不能仅凭它来构成方法的重载。下面看错误的代码：

```

01 using System;
02 namespace Method_Show
03 {
04     class Program
05     {
06         static void Test(ref int a)
07         {
08         }
09         static void Test(out int a)
10         {
11             a = 100;
12         }
13         static void Test(bool[] a)
14         {
15         }
16         static void Test(params bool[] a)
17         {
18         }
19         static void Main(string[] args)
20         {
21             Test(false);
22             Console.ReadKey();
23         }

```

```
24     }  
25 }
```

这段代码的第 6 行和第 9 行定义的两个方法意图以形参类型中的引用形参和输出形参的方法来进行方法重载。

第 13 行和第 16 行定义的两个方法意图以形参数组的方法来进行方法重载。但是可惜的是，这两种方法都不能构成重载条件。错误提示如下：

错误 1 无法定义重载方法“Test”，因为它与其他方法仅在 ref 和 out 上有差别

错误 2 类型“Method_Show.Program”已定义了一个名为“Test”的具有相同参数类型的成员

其中，错误 1 是使用 ref 和 out 进行重载造成的错误。而错误 2 是因为使用形参数组造成的错误。第 16 行定义的方法和第 13 行定义的方法重复。

第六章 类型转换

6.1 什么是 CTS

CTS(Common Type System, 通用类型系统) 定义了一个能够在 CLR 上运行的语言规范。CTS 定义了可以在 CIL 中使用的预定义数据类型。所有基于 .Net Framework 的托管语言最终都生成基于预定义数据类型的 CIL 代码。下面看图 6-1。

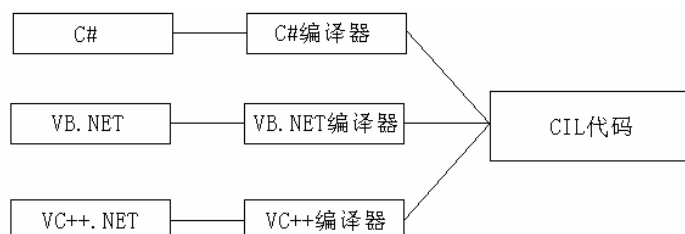


图 6-1 托管语言编译过程

被 .Net Framework 环境支持的托管语言编写的源代码，通过各自基于 .Net Framework 的编译器都编译成 CIL 代码。它们各自的语法不同，但生成的 CIL 代码几乎都是一样的，相差很小。

所以，这就解决了一个问题，使用不同编程语言的程序员可以通过 .Net Framework 这个平台进行合作。VB.NET 的程序可以调用 C# 的程序。为了让它们能够相互调用，所以底层的数据类型必须相同。

CTS 不仅仅定义了一个通用的数据类型，它还定义了一个内容丰富的类层次结构。如图 6-2 所示。

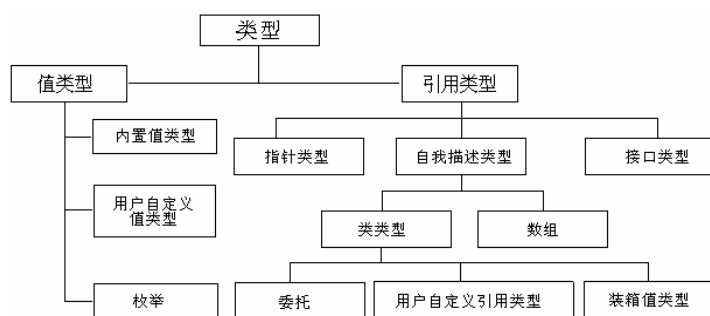


图 6-2 CTS 类层次结构

CTS 的类层次结构表现了 CIL 单一继承的面向对象编程特点，C#当然也具有单一继承的面向对象特点。

6.2 CLS 和系统数据类型

CLS(Common Language Specification, 通用语言规范) 是一个最低的标准集。它和 CTS 一起确保语言的互操作性。如果对这个概念不好理解，可以这样认为，CTS 是 CLR 能够支持的所有功能，在其上的每种语言都实现了其中的一部分功能。而 CLS 是其中大家如果要互相协作必须遵守的一个交集，或者说一个子集。如果各种语言都遵守 CLS，那么它们之间就可以互相调用而没有任何问题。下面是 C#系统数据类型表：

表 6-1 C#系统数据类型表

C#关键字	系统类型	取值范围	说明	是否符合 CLS
bool	System.Boolean	true 或 false	表示真或假	符合
sbyte	System.SByte	-128~127	有符号 8 位数	不符合
byte	System.Byte	0~255	无符号 8 位数	符合
short	System.Int16	-32768~32767	有符号 16 位整型	符合
ushort	System.UInt16	0~65535	无符号 16 位整型	不符合
int	System.Int32	-2147483648~2147483647	有符号 32 位整型	符合
uint	System.UInt32	0~4294967295	无符号 32 位整型	不符合
long	System.Int64	-9223372036854775808~9223372036854775807	有符号 64 位整型	符合

ulong	System.UInt64	0~18446744073709551615	无符号 64 位整型	不符合
char	System.Char	U+0000~U+ffff	16 位 Unicode 字符	符合
float	System.Single	$1.5 \times 10^{-45} \sim 3.4 \times 10^{38}$	32 位浮点数	符合
double	System.Double	$1.5 \times 10^{-324} \sim 1.7 \times 10^{308}$	64 位浮点数	符合
decimal	System.Decimal	$1.0 \times 10^{-28} \sim 7.9 \times 10^{28}$	128 位有符号数	符合
string	System.String	受系统内存限制	一个 Unicode 字符集合	符合
object	System.Object	所有类型都可以保存在 Object 变量中	.Net 中所有类型的基 类	符合

C#的基本数据类型不仅仅是一个供编译器识别的标记。C#的基本数据类型关键字是 Systemmm 命名空间中的类的简化符号。如果需要让 C#写的类库提供给其他语言使用,就需要只使用符合 CLS 的数据类型。从表中可以看到,所有关键字都对应于类库中的一个类,其中有符号数的二进制数的最高位用来表示正负。

从 CTS 类层次结构中可以看到,C#的类型按照存储在内存中的位置,可以分为值类型和引用类型。值类型的数据是分配在栈中的,而引用类型的数据是分配在托管堆上的,在栈中保存的是对托管堆中的地址的引用。在基本数据类型中,除了 string 类型和 object 类型外,其余都是值类型。这些值类型,如 int, float 等,都对应 System 命名空间中的一个结构。因此它们有属于自己的方法。比如,可以使用属于它们自己的结构成员的 ToString 方法将它们转成字符串。

6.3 隐式转换

类型转换就是将给定类型的表达式转换为不同类型。C#的类型转换方式分为隐式转换和显式转换。隐式转换由编译器自动完成,显式转换由程序员手动指定。隐式转换总是成功的,不会报错。隐式转换有许多种,下面来一一介绍,对于没有接触到的类型之间的转换,会在各自的章节中介绍。

6.3.1 标识转换

标识转换是在同一类型的内部进行的转换。下面看代码实例:

```

01 using System;
02 namespace Conversion
03 {
04     class Program
05     {
06         static void Main(string[] args)
07         {
08             int a = 11;
09             int b = a;
10             bool c = true;
11             bool d = c;
12             float e = 3.14F;
13             float f = e;
14             int[] g = new int[] { 1, 2, 3 };
15             int[] h = g;
16             Console.ReadKey();
17         }
18     }
19 }

```

这段代码中进行了多次赋值操作,其中第 8 行、第 10 行、第 12 行、第 14 行定义了多个类型的不同变量,前三种是值类型,最后一种是数组,它是引用类型。

第 9 行、第 11 行、第 13 行和第 15 行分别定义了一个同类型的变量,并且用先前定义并初始化的变量

对其进行赋值操作。那么，在这同类型之间发生的就是隐式转换中的标识转换。

6.3.2 基本值类型转换

基本值类型之间的转换包括下面这些类型：

- 从 sbyte 到 short、int、long、float、double 或 decimal。
- 从 byte 到 short、ushort、int、uint、long、ulong、float、double 或 decimal。
- 从 short 到 int、long、float、double 或 decimal。
- 从 ushort 到 int、uint、long、ulong、float、double 或 decimal。
- 从 int 到 long、float、double 或 decimal。
- 从 uint 到 long、ulong、float、double 或 decimal。
- 从 long 到 float、double 或 decimal。
- 从 ulong 到 float、double 或 decimal。
- 从 char 到 ushort、int、uint、long、ulong、float、double 或 decimal。
- 从 float 到 double。

在上面这几种类型到类型之间的转换由编译器自动进行操作，不会引发异常。下面看代码实例：

```
01 using System;
02 namespace Conversion
03 {
04     class Program
05     {
06         static void Main(string[] args)
07         {
08             sbyte a = -13;
09             int _a = a;
10             byte b = 13;
11             uint _b = b;
12             short c = -123;
13             float _c = c;
14             ushort d = 123;
15             ulong _d = d;
16             int e = -1234;
17             double _e = e;
18             uint f = 1234;
19             ulong _f = f;
20             long g = 12345;
21             double _g = g;
22             ulong h = 12345U;
23             decimal _h = h;
24             char i = 'a';
25             int _i = i;
26             float j = 3.14F;
27             double _j = j;
28             Console.ReadKey();
29         }
30     }
```

```
31 }
```

这段代码定义了所有基本数据类型，并且进行了不同类型之间的赋值操作。例如说第 8 行首先定义了一个 sbyte 类型的变量，并且进行了赋值操作。然后第 9 行定义了一个 int 类型的变量，并且将 a 赋值给它。可以明显看到，sbyte 和 int 是属于不同的类型，但是因为它们之间允许发生隐式转换，所以编译器没有提示任何异常信息。

第 9 行代码为了比较明显，所以定义了一个标识符同样为 a。但前面加了一个下划线作为区别的变量。这里再重申一下，可以作为变量标识符开头第一位的只有大小写字母和下划线，其余字符都不可以打头。标识符中只可以有字母、数字和下划线，不可以包括其它字符。

再看一下第 22 行代码，首先定义了 ulong 类型的变量 h，然后将文本 12345U 赋值给它。12345U 以字母 U 后缀结尾，因此它将按照值的范围，以 uint、ulong 的次序判断类型，因为 uint 符合范围，所以这个文本被认为是 uint 类型。将它赋值给 ulong 类型做初始化操作时，本身这个初始化操作就发生了一次隐式转换。因为允许 uint 到 ulong 类型的隐式类型转换，所以编译器没有提示任何异常。

从 int、uint、long 和 ulong 到 float 的转换以及从 long、ulong 到 double 的转换可能导致精度损失，但不会影响数值大小。

虽然 char 类型可以转换成整型，但是整型却不可以隐式转换为 char 类型。如果要将一个整型转换为一个 char 类型，只能通过显式转换来实现。

6.3.3 运算过程中的隐式转换

在使用算术运算符运算过程中，也存在数据类型之间的隐式转换。下面看代码实例：

```
01 using System;
02 namespace Conversion
03 {
04     class Program
05     {
06         static void Main(string[] args)
07         {
08             Type t;
09             //整型一元运算符
10             var a = +9223372036854775807;
11             t = a.GetType();
12             Console.WriteLine("变量的数据类型是: {0}", t);
13             uint b = 4294967295;
14             var _b = -a;
15             t = _b.GetType();
16             Console.WriteLine("变量的数据类型是: {0}", t);
17             //整型二元运算符
18             int c = 323;
19             uint d = 4294967000;
20             var _d = c + d;
21             t = _d.GetType();
22             Console.WriteLine("变量的数据类型是: {0}", t);
23             //二元运算符<<和>>
24             var e = 4294967000 << 2;
25             t = e.GetType();
26             Console.WriteLine("变量的数据类型是: {0}", t);
```

```

27         var f = 2457 >> 2;
28         t = f.GetType();
29         Console.WriteLine("变量的数据类型是: {0}", t);
30         Console.ReadKey();
31     }
32 }
33 }

```

首先来看整型一元运算符运算过程的转换，它的隐式转换规则如下：

1、对于一元运算符+，操作数按照 int、uint、long 和 ulong 中第一个可以完全表示操作数值的类型进行隐式转换，然后用转换后的类型的精度执行运算，结果的类型是转换后的类型。

2、对于一元运算符-，操作数按照 int 和 long 中第一个可以完全表示操作数的值的类型进行隐式转换，然后用转换后的类型的精度执行运算，结果的类型转换后的类型。对于 ulong 的操作数不能运用-运算符。

第 8 行代码定义了 Type 类型的变量 t 用来接收取得的变量类型。第 10 行代码定义了一个推断类型的变量，然后用一个文本值执行+运算，并且给这个推断类型赋值。因为这个文本值在执行+运算符时，按照一定的顺序进行隐式转换，所以在第 12 行打印其类型时，为 long 类型。

第 13 行定义了一个 uint 类型的变量，然后第 14 行对其应用-运算符。因为执行-运算符时，按照 int 和 long 进行隐式转换，所以它被转换为 long 类型，然后赋值给_b。

从第 18 行开始，是二元运算符的运算，二元运算符涉及两个操作数，它们的隐式转换规则如下：

对于+、-、*、/、%、&、^、|、==、!=、>、<、>= 和<= 二元运算符，操作数按照 int、uint、long 和 ulong 中第一个可以完全表示两个操作数的所有可能值的类型。然后用转换后的类型的精度执行运算，运算的结果的类型也属于转换后的类型。对于二元运算符，不允许一个操作数为 long 类型而另一个操作数为 ulong 类型。

参加运算的两个操作数，一个是 int 类型，一个是 uint 类型，它们进行相加运算。对于变量 c 的值 323 而言，uint 类型可以表示其值，但是这个结果的类型确是 long 类型，这是为什么呢？这是因为隐式转换的规则里有“完全表示”的规则，也就是说，并不是 uint 类型可以表示第一个值 323，就将它们都转化为 uint 类型，而是用一个类型可以完全表示 int 类型和 uint 类型。那么只有 long 类型了，所以第 22 行的结果是 long 类型。

从第 24 行开始是左移位和右移位运算符的隐式转换规则，它们的规则如下：

对于二元运算符<<和>>，运算符左边的那个操作数按照 int、uint、long 和 ulong 中第一个可以完全表示操作数的所有可能值的类型。然后用转换后的类型的精度执行运算，结果的类型转换后的类型。

有了上面的介绍，相信这条规则也很好理解了。下面是这段代码的执行结果：

```

变量的数据类型是: System.Int64
变量的数据类型是: System.Int64
变量的数据类型是: System.Int64
变量的数据类型是: System.UInt32
变量的数据类型是: System.Int32

```

介绍完整型运算过程中的隐式转换后，下面再看涉及浮点数的运算隐式转换。看代码实例：

```

01 using System;
02 namespace Conversion
03 {
04     class Program
05     {
06         static void Main(string[] args)

```

```

07     {
08         Type t;
09         //一个操作数为整型的情况
10         int a = 21;
11         float b = 3.124F;
12         var c = a + b;
13         t = c.GetType();
14         Console.WriteLine("变量的数据类型是: {0}", t);
15         //一个操作数为 double 类型的情况
16         double d = 5.6D;
17         var e = a + b + d;
18         t = e.GetType();
19         Console.WriteLine("变量的数据类型是: {0}", t);
20         Console.ReadKey();
21     }
22 }
23 }

```

对于浮点类型，有如下两条规则：

1、如果一个操作数是整型，另一个操作数是浮点型，则整型要自动隐式转换为和另一个浮点数类型相同的类型进行计算，结果为转换后的浮点类型。

2、如果参加运算的一个操作数为 double 类型，则另一个操作数不管是整型还是 float 类型，都要隐式转换为 double 类型进行计算，结果为 double 类型。

下面看第 10 行代码，第 10 行代码定义了一个 int 类型的变量并初始化，第 11 行代码定义了一个 float 类型的变量并初始化，然后第 12 行对它们进行了求和的运算，并将结果赋值给了 c。因为 b 为 float 类型，所以 a 在运算时也要转换为 float 类型。运算结果也为 float 类型。

第 16 行代码定义了一个 double 类型的变量，然后第 17 行将 a、b 和 c 进行相加。因为其中一个操作数为 double 类型，所以另外两个操作数都要转化为 double 类型进行计算，结果也为 double 类型。下面是执行结果：

```
变量的数据类型是: System.Single
```

```
变量的数据类型是: System.Double
```

接下来再来观察 decimal 类型，decimal 类型是小数类型，虽然它和浮点数很像，但是隐式转换的规则却不相同。下面看代码实例：

```

01 using System;
02 namespace Conversion
03 {
04     class Program
05     {
06         static void Main(string[] args)
07         {
08             Type t;
09             //一个操作数为整型的情况
10             int a = 23;
11             decimal b = 3.45M;
12             var c = a + b;

```



```

13         t = c.GetType();
14         Console.WriteLine("变量的数据类型是: {0}", t);
15         Console.ReadKey();
16     }
17 }
18 }

```

当两个操作数进行算术运算时，如果一个操作数为 decimal 类型，则允许隐式转换的另一个操作数必须是整型或也是 decimal 类型。

第 10 行代码定义了一个整型变量，第 11 行代码定义了一个 decimal 类型的变量，当第 12 行代码对它们进行相加操作时，a 要隐式转换成 decimal 类型。运算结果也为 decimal 类型。这段代码的执行结果如下：

```
变量的数据类型是: System.Decimal
```

对比 decimal 类型和浮点类型，decimal 类型具有精度高，取值范围小的特点。因此，在从浮点型到 decimal 类型进行转换时，可能会产生溢出。而从 decimal 类型向浮点类型转换时，可能会损失精度。所以，浮点类型和 decimal 类型之间不存在隐式转换。如果两个操作数分别是 decimal 类型和浮点类型，编译器就会要求强制转换。将这段代码中再加入下面的语句：

```

double d = 4.56;
decimal e = 5.4M;
var f = d + e;
decimal g = 3.14;

```

这时编译器就会提示 2 个错误，如下：

错误 1 运算符“+”无法应用于“double”和“decimal”类型的操作数

错误 2 不能隐式地将 Double 类型转换为“decimal”类型；请使用“M”后缀创建此类型

这是因为 decimal 类型和浮点类型之间不能进行隐式转换的原因，解决这一问题的办法就是使用强制（显式）转换。

对于 bool 类型来说，不存在 bool 类型和其它类型之间的转换。不能像在 C++ 中那样，将非 0 值转化为 true，将 0 转化为 false。可以这么说，C# 中的 bool 类型是血统纯粹的类型，它只有 true 和 false 两个值。

6.3.4 null 文本

null 文本可以隐式转换为引用类型或可以为 null 的类型。可以为 null 的类型，后面会介绍到。任何一个引用类型的变量都可以用 null 进行赋值，在这个过程中发生了隐式转换。下面看代码实例：

```

01 using System;
02 namespace Conversion
03 {
04     class Program
05     {
06         static void Main(string[] args)
07         {
08             string s = null;
09             int[] a = null;
10             Console.WriteLine(s);
11             foreach (int i in a)
12             {
13                 Console.WriteLine(i);

```

```

14         }
15         Console.ReadKey();
16     }
17 }
18 }

```

第 8 行代码和第 9 行代码定义了两个引用类型的变量，一个是 `string` 类型，另一个是 `int` 类型的一维数组。但是，和前面介绍的初始化过程不同，这里将 `null` 赋值给了它们。在这个过程中发生了隐式转换。可以这么理解，这两个变量只分配在了栈中，但是没有指向任何实例。也就是说，在堆中还没有分配内存。因此，第 10 行代码打印这个变量的值，会什么也打印不出来；而第 11 行代码打印数组中的元素，在执行时会出错，因为在数组中没有元素，所以对它不能进行遍历。错误提示如下：

未将对象引用设置到对象的实例。

6.3.5 赋值中的隐式转换

现在已经接触过简单赋值运算的隐式转换。对于赋值运算来说，有两种形式的赋值运算符。一种是简单赋值运算符`=`；还有一种就是复合赋值运算符。下面看代码实例：

```

01 using System;
02 namespace Conversion
03 {
04     class Program
05     {
06         static void Main(string[] args)
07         {
08             long a;
09             int b = -3456;
10             a = b;
11             int c = 567;
12             sbyte d = 34;
13             c += d;
14             Console.ReadKey();
15         }
16     }
17 }

```

对于简单赋值运算符来说，右操作数必须可以隐式转换为左操作数。例如第 8 行代码定义了一个 `long` 类型的变量。第 9 行定义了一个 `int` 类型的变量，当将第 9 行定义的变量 `b` 赋值给 `a` 时，发生了隐式转换。它的类型由 `int` 类型转换成了 `long` 类型。

而对于复合赋值运算符来说，右操作数也是必须可以转换成左操作数。第 13 行代码中，`d` 由原来的 `sbyte` 类型在运算时隐式转换成了 `int` 类型。

6.3.6 语句中的隐式转换

在前面介绍过的 `switch` 语句中同样存在着隐式转换，代码实例如下：

```

01 using System;
02 namespace Conversion
03 {
04     class Program
05     {
06         static void Main(string[] args)

```

```
07     {
08         for (; ; )
09         {
10             Console.WriteLine("请输入一个数值: ");
11             string s = Console.ReadLine();
12             if (s == "q")
13             {
14                 break;
15             }
16             long i = Convert.ToInt64(s);
17             switch (i)
18             {
19                 case 1:
20                     Console.WriteLine("正月一到梅花香");
21                     break;
22                 case 2:
23                     Console.WriteLine("二月芙蓉结成双");
24                     break;
25                 case 3:
26                     Console.WriteLine("三月茶花多含笑");
27                     break;
28             }
29         }
30         Console.ReadKey();
31     }
32 }
33 }
```

每个 switch 都有一个主导类型。这个主导类型就是由这段代码第 17 行中的 i 来确定。它必须是 sbyte、byte、short、ushort、int、uint、long、ulong、bool、char、string 或枚举类型，或者是对应于以上类型的可以为 null 的类型。

代码的第 16 行定义了一个 long 类型的变量来接收用户输入，它确定 switch 语句的主导类型。从第 19 行代码开始，每个 case 语句后面跟的文本值都必须能够隐式转换为主导类型。这段代码中的每个 case 语句后都是 int 类型，它可以隐式转换为主导类型 long。如果无法隐式转换为主导类型，则编译器会提示错误，如下：

错误 1 无法将类型“uint”隐式转换为“byte”。存在一个显式转换(是否缺少强制转换?)

出现这个错误的原因是，switch 语句的主导类型是 byte，而每个 case 语句后面的文本被定义成如 1U 这样的 uint 类型。

如果 switch 语句的主导类型是 string 类型，则 case 语句后可以接 null 文本。因为它可以隐式转换为 string 类型。这段代码的执行结果如下：

```
请输入一个数值:
1
正月一到梅花香
请输入一个数值:
2
```

```

二月芙蓉结成双
请输入一个数值:
3
三月茶花多含笑
请输入一个数值:
q

```

6.3.7 方法中的隐式转换

在方法执行过程中，传入方法的实参类型必须能够隐式转换为方法的形参类型，否则将提示错误。实例代码如下：

```

01 using System;
02 namespace Conversion
03 {
04     class Program
05     {
06         static void Test(long a, params int[] b)
07         {
08             Console.WriteLine(a);
09             foreach (int i in b)
10             {
11                 Console.WriteLine(i);
12             }
13         }
14         static void Main(string[] args)
15         {
16             int a = 111;
17             Test(a, 11, 22);
18             Console.ReadKey();
19         }
20     }
21 }

```

第 6 行代码定义了一个方法，它带有两个形参。其中一个是 long 类型，另一个是 int 类型的形参数组。

第 16 行代码定义了一个 int 类型的变量，第 17 行调用这个方法，将变量 a 传入进去，然后传入两个整型的文本给形参数组。因为 int 类型可以隐式转换成 long 类型，所以这个调用是正常的，没有错误。执行结果如下：

```

111
11
22

```

如果将第 17 行代码改成如下代码：

```
Test(a, "abc", "d");
```

因为传入的两个字符串无法隐式转换为 int 类型，所以会提示三个错误：

```

错误 1    与“Conversion.Program.Test(long, params int[])”最匹配的重载方法具有一些无效参数
错误 2    参数 2: 无法从“string”转换为“int”
错误 3    参数 3: 无法从“string”转换为“int”

```

6.3.8 装箱转换

装箱转换的作用是将一个值类型隐式转换为一个引用类型。对于这个值类型的要求是非可空值类型。可空的值类型对应于普通的基本数据类型，只不过它可以为空，而普通的基本数据类型不可以为空。可空值类型后面会介绍到。下面看代码实例：

```
01 using System;
02 namespace Conversion
03 {
04     class Program
05     {
06         static void Main(string[] args)
07         {
08             //向 object 类型的装箱转换
09             int a = 33;
10             object o = a;
11             //向基类的装箱转换
12             System.ValueType A = a;
13             Console.ReadKey();
14         }
15     }
16 }
```

装箱操作很简单，首先第 9 行定义了一个基本数据类型，并为它做了初始化操作。第 10 行定义了一个 object 类型的变量，然后将变量 a 赋值给它，这就完成了一个装箱操作。它实际发生的操作是，在栈中定义了一个引用类型的变量 o，然后在堆中分配了一个实例，并且将 a 的值复制到堆中的实例中。除了可以向 object 类型装箱外，还可以向所有值类型的基类 System.ValueType 类型装箱，它的原理和向 object 类型装箱是一样的。第 12 行定义了一个向 System.ValueType 类型的装箱操作。

6.4 显式转换

显式转换是程序员显式指定的类型转换，显式转换变换前后的类型显著不同，并且在转换过程中有可能丢失信息。前面介绍过的隐式转换也可以看做是安全的显式转换，下面看代码实例：

```
01 using System;
02 namespace Conversion
03 {
04     class Program
05     {
06         static void Main(string[] args)
07         {
08             int a = -33;
09             long _a = (long)a;
10             sbyte b = 22;
11             int _b = (int)b;
12             Console.ReadKey();
13         }
14     }
15 }
```

这段代码使用的是冗余的强制转换表达式。可以这么说，不使用这种冗余的强制转换表达式，它发生的就是隐式转换。以第 9 行代码为例，显式转换的方法就是在变量前加一个括号，里面是要转换到的类型。因

为是明知转换后有可能丢失信息，所以显式转换也叫做强制转换。上面的代码中，因为转换本身就是安全的，所以这种显式转换的使用就是冗余的。

6.4.1 整型之间的显式转换

下面首先列出基本数据类型之间需要显式转换的类型：

- 从 sbyte 到 byte、ushort、uint、ulong 或 char。
- 从 byte 到 sbyte 和 char。
- 从 short 到 sbyte、byte、ushort、uint、ulong 或 char。
- 从 ushort 到 sbyte、byte、short 或 char。
- 从 int 到 sbyte、byte、short、ushort、uint、ulong 或 char。
- 从 uint 到 sbyte、byte、short、ushort、int 或 char。
- 从 long 到 sbyte、byte、short、ushort、int、uint、ulong 或 char。
- 从 ulong 到 sbyte、byte、short、ushort、int、uint、long 或 char。
- 从 char 到 sbyte、byte 或 short。
- 从 float 到 sbyte、byte、short、ushort、int、uint、long、ulong、char 或 decimal。
- 从 double 到 sbyte、byte、short、ushort、int、uint、long、ulong、char、float 或 decimal。
- 从 decimal 到 sbyte、byte、short、ushort、int、uint、long、ulong、char、float 或 double。

当在这些类型之间使用隐式转换的语法之后，编译器会给出错误提示，比如说下面的代码：

```
01 using System;
02 namespace Conversion
03 {
04     class Program
05     {
06         static void Main(string[] args)
07         {
08             int a = -33;
09             byte _a = a;
10             Console.ReadKey();
11         }
12     }
13 }
```

第9行代码将一个 int 类型的变量赋值给一个 byte 类型，而且使用的是隐式转换的语法。这时编译器会给出错误提示。如下：

错误 1 无法将类型“int”隐式转换为“byte”。存在一个显式转换(是否缺少强制转换?)

也就是说，如果不知道两个类型之间是否允许隐式转换的时候，在进行隐式转换时，编译器会给出显式转换的智能提示。下面将项目的溢出检查打开，然后将上面的代码改写成显式转换语法，如下：

```
01 using System;
02 namespace Conversion
03 {
04     class Program
05     {
06         static void Main(string[] args)
07         {
08             int a = -33;
```

```

09         byte _a = (byte)a;
10         Console.ReadKey();
11     }
12 }
13 }

```

这时会发现，编译器的错误提示没有了。但是第 9 行代码需要注意，它是将一个负值强制转换赋值给一个无符号字节类型，这是不允许的，因为无符号字节类型只能是正数。下面编译这段代码，编译过程是成功的。然后再运行这段代码，问题就会出现，它会抛出下面的异常信息：

算术运算导致溢出。

这是因为设置了编译器的溢出检查，所以在编译时不会发现问题，因为语法是正确的。但是运行时会抛出错误，原因是编译时打开了溢出检查。下面使用 `unchecked` 关键字取消溢出检查，代码如下：

```

01 using System;
02 namespace Conversion
03 {
04     class Program
05     {
06         static void Main(string[] args)
07         {
08             int a = -33;
09             unchecked
10             {
11                 byte _a = (byte)a;
12                 Console.WriteLine(_a);
13             }
14             Console.ReadKey();
15         }
16     }
17 }

```

因为没有使用溢出检查，所以在运行时不会抛出错误。但是当第 12 行代码打印显式转换后的值时，就会发现这是一个意想不到的值，运行结果如下：

223

产生这个值的原因在于，当使用 `unchecked` 关键字或取消编译器的溢出检查时，显式转换总会成功。对于结果来说，如果源变量的类型大于目标类型，就像这段代码中，`int` 类型的取值是 `byte` 类型无法表示的，这时就会发生截断，截断源类型中无法表示的高位，然后将结果赋值给目标类型。

当源类型小于目标类型时，源类型会按照两种方式进行扩展，以便使位数符合目标类型位数。有两种情况，一种是源类型是有符号数，这时会按符号扩展，高位不足的位数补 0；另一种是源类型是无符号的，这时会按 0 进行扩展，高位补 0，然后将结果赋值给目标类型。下面看代码实例：

```

01 using System;
02 namespace Conversion
03 {
04     class Program
05     {
06         static void Main(string[] args)
07         {

```

```
08      int a = int.MaxValue;
09      short b = short.MaxValue;
10      unchecked
11      {
12          Console.WriteLine(Convert.ToString(a, 2));
13          byte _a = (byte)a;
14          Console.WriteLine(Convert.ToString(_a, 2));
15          Console.WriteLine(Convert.ToString(b, 2));
16          uint _b = (uint)b;
17          Console.WriteLine(Convert.ToString(_b, 2));
18      }
19      Console.ReadKey();
20  }
21 }
22 }
```

这段代码的第 8 行和第 9 行分别定义了一个 int 类型的变量和一个 short 类型的变量，并分别赋值为最大值。

接下来第 10 行使用了 unchecked 关键字，在第 12 行首先打印了变量 a 的值的二进制形式，然后将 a 显式转换为 byte 类型。因为 int 的类型大于 byte 的类型容量，所以进行了截断，截断其高位并将剩余结果赋值给 byte 类型变量。所以第 14 行打印 byte 类型值的时候，可以发现源变量的高位不见了。

第 15 行首先打印了 short 类型的源变量以便比较，然后第 16 行将 short 类型的变量显式转换为 uint 类型的变量，因为源类型小于目标类型，所以高位填 0 以扩充位数。第 17 行打印了 uint 类型变量的值，因为高位为 0，所以没有打印出来。

综上，使用 unchecked 关键字时，显式转换都会成功并且不抛出异常。

但是当使用 checked 关键字或编译器的溢出检查打开时，显式转换则并不一定成功。它取决于源类型的值是否在目标类型的范围内，如果源类型的值在目标类型范围内，则不抛出异常，转换会成功，否则抛出数据溢出异常。实例代码如下：

```
01 using System;
02 namespace Conversion
03 {
04     class Program
05     {
06         static void Main(string[] args)
07         {
08             int a = 33;
09             checked
10             {
11                 uint _a = (uint)a;
12                 Console.WriteLine(_a);
13             }
14             Console.ReadKey();
15         }
16     }
17 }
```


这段代码中，因为 `int` 类型变量的值在 `uint` 类型的值可表示的范围内，所以转换会成功，且不抛出异常。但是当把 `a` 初始化为负值时，则转换失败且抛出溢出异常。

6.4.2 decimal 到整型的显式转换

`decimal` 类型向整型进行显式转换时，它首先向 0 舍入到最接近的整数值，然后将该整数值作为结果赋值给目标类型。下面是代码实例：

```
01 using System;
02 namespace Conversion
03 {
04     class Program
05     {
06         static void Main(string[] args)
07         {
08             decimal a = 3.64M;
09             int _a = (int)a;
10             Console.WriteLine(_a);
11             //decimal b = 999999999999.9M;
12             //int _b = (int)b;
13             //Console.WriteLine(_b);
14             Console.ReadKey();
15         }
16     }
17 }
```

第 8 行代码定义了一个 `decimal` 类型的变量，然后第 9 行将它显式转换为 `int` 类型。虽然小数点后的第一位超过了 5，但是它仍然向 0 进行舍入。因此第 10 行打印 `_a` 的值将是 3。

如果将从第 11 行开始的三行代码的注释去掉，则执行这段代码时将抛出异常，异常信息如下：

未经处理的异常：OverflowException

值对于 Int32 太大或太小。

这是因为 `decimal` 值的大小超过了 `int` 类型的范围。在这种情况下，不论是否使用 `unchecked` 关键字，都会抛出异常。

6.4.3 浮点数到整型的显式转换

浮点数到整型之间的显式转换，在进行溢出检查和不进行溢出检查这两种情况下得到的结果是不同的，下面是代码实例：

```
01 using System;
02 namespace Conversion
03 {
04     class Program
05     {
06         static void Main(string[] args)
07         {
08             float a = 0.0F / 0.0F;
09             Console.WriteLine(a);
10             float b = 1.0F / 0.0F;
11             Console.WriteLine(b);
12             checked
```

```

13      {
14          int _a = (int)a;
15          int _b = (int)b;
16          Console.WriteLine(_a);
17          Console.WriteLine(_b);
18      }
19      Console.ReadKey();
20  }
21  }
22  }

```

这段代码第 8 行定义了一个非数字值，第 10 行定义了一个正无穷大的数。当在溢出检查情况下，将它们显式转换为一个 int 类型的数时，会抛出异常。得到的执行结果如下，首先会打印 a 和 b 的值：

```

非数字
正无穷大

```

然后会抛出异常信息：

```

算术运算导致溢出。

```

如果浮点数的范围在目标范围内，则浮点数会向 0 舍入到最接近的整数值，然后将这个值赋值给目标类型。如果浮点数的范围超出了目标整型的取值范围，则会抛出异常信息。下面是代码实例：

```

01  using System;
02  namespace Conversion
03  {
04      class Program
05      {
06          static void Main(string[] args)
07          {
08              float a = 5.968F;
09              Console.WriteLine(a);
10              float b = 9999999999.9F;
11              Console.WriteLine(b);
12              checked
13              {
14                  int _a = (int)a;
15                  Console.WriteLine(_a);
16                  int _b = (int)b;
17                  Console.WriteLine(_b);
18              }
19              Console.ReadKey();
20          }
21      }
22  }

```

第 8 行代码定义了一个浮点数，然后打印它的值。第 10 行代码定义了一个比较大的 float 类型数，然后打印它的值，它的值将以科学计数法的形式打印。在 checked 块内，将 a 和 b 强制转换为整型。第 14 行转换的时候，小数位数将向 0 舍入，最后打印输出为 5。而第 16 行因为浮点数的范围超出了 int 类型的范围，所以将抛出异常信息。执行结果如下：

```
5. 968
1E+11
5
```

异常信息如下：

算术运算导致溢出。

当取消溢出检查或者使用 `unchecked` 关键字的时候，显式转换总会成功。下面是代码实例：

```
01 using System;
02 namespace Conversion
03 {
04     class Program
05     {
06         static void Main(string[] args)
07         {
08             float a = 0.0F / 0.0F;
09             float b = 1.0F / 0.0F;
10             float c = 5.67F;
11             float d = 9999999999.9F;
12             unchecked
13             {
14                 int _a = (int)a;
15                 int _b = (int)b;
16                 Console.WriteLine(_a);
17                 Console.WriteLine(_b);
18                 int _c = (int)c;
19                 int _d = (int)d;
20                 Console.WriteLine(_c);
21                 Console.WriteLine(_d);
22             }
23             Console.ReadKey();
24         }
25     }
26 }
```

第 8 行和第 9 行定义了一个非数字和一个正无穷大的 `float` 类型值。第 12 代码使用 `unchecked` 关键字取消溢出检查。第 14 行和第 15 行代码对 `a` 和 `b` 进行显式转换，因为源操作数是一个非数字和一个正无穷大，当它们强制转换为 `int` 类型的时候，将是一个未指定的值。第 16 行和第 17 行将输出 `_a` 和 `_b` 的值都为 `int` 类型的最小值。

第 18 行转换后的值将是 5。

第 19 行因为源操作数的范围已经超过了 `int` 类型能表示的范围，所以发生了溢出。因为使用了取消溢出检查机制，所以它也将是一个未指定的值，这里是 `int` 类型的最小值。代码的执行结果如下：

```
-2147483648
-2147483648
5
-2147483648
```

6.4.4 浮点数到浮点数的显式转换

如果从 double 向 float 进行转换, double 值将舍入为最接近的 float 值。如果 double 值过小, 无法表示为 float 值, 则结果为正零或负零。如果 double 值过大, 无法表示为 float 值, 则结果为正无穷大或负无穷大。如果 double 值为 NaN, 则结果也为 NaN。

6.4.5 浮点数与 decimal 之间的显式转换

当从浮点数向 decimal 类型进行显式转换时, 源类型将转换为 decimal 表示形式, 并且在需要时, 将它在第 28 位小数位上舍入到最接近的数字。如果源值过小, 无法表示为 decimal, 则结果变成零。如果源值为 NaN、无穷大或者太大而无法表示为 decimal, 则将抛出算术运算溢出异常。

当从 decimal 向浮点数进行显式转换时, decimal 值将舍入为最接近的 double 或 float 值。虽然这种转换可能会损失精度, 但决不会导致引发异常。

6.4.6 拆箱转换

拆箱转换是前面介绍过的装箱转换的逆过程, 它将一个引用类型显式转换为一个值类型。不过, 区别是装箱转换是隐式转换, 拆箱转换是显式转换。下面看代码实例:

```
01 using System;
02 namespace Conversion
03 {
04     class Program
05     {
06         static void Main(string[] args)
07         {
08             int a = 33;
09             object _a = a;
10             System.ValueType v = a;
11             int b = (int)_a;
12             int c = (int)v;
13             Console.WriteLine("拆箱后 b 的值是: {0} c 的值是: {1}", b, c);
14             Console.ReadKey();
15         }
16     }
17 }
```

这段代码中的第 8 行和第 9 行代码执行了一个装箱操作, 这时, 变量 a 的值被赋值给了堆中的实例。第 11 行和第 12 行进行了拆箱转换, 拆箱转换时, 需要有一个值类型的变量来接受拆箱操作后的值。拆箱操作的语法很简单, 就是显式转换语法。显式转换后, 引用类型被转换为值类型。这段代码的执行结果如下:

拆箱后 b 的值是: 33 c 的值是: 33

第七章 枚举类型和可空类型

7.1 什么是枚举类型

枚举类型是一种值类型，它是存储在栈中的。枚举类型具有一组命名常量。下面是代码实例：

```
01 using System;
02 namespace Conversion
03 {
04     class Program
05     {
06         enum Week { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday}
07         static void Main(string[] args)
08         {
09             Console.WriteLine("请输入 1-7 的数字:");
10             for (; ; )
11             {
12                 string i = Console.ReadLine();
13                 if (i == "q")
14                     break;
15                 switch (i)
16                 {
17                     case "1":
18                         Console.WriteLine("Today is {0}", Week.Monday);
19                         break;
20                     case "2":
21                         Console.WriteLine("Today is {0}", Week.Tuesday);
22                         break;
23                     case "3":
24                         Console.WriteLine("Today is {0}", Week.Wednesday);
25                         break;
26                     case "4":
27                         Console.WriteLine("Today is {0}", Week.Thursday);
28                         break;
29                     case "5":
30                         Console.WriteLine("Today is {0}", Week.Friday);
31                         break;
32                     case "6":
33                         Console.WriteLine("Today is {0}", Week.Saturday);
34                         break;
35                     case "7":
36                         Console.WriteLine("Today is {0}", Week.Sunday);
37                         break;
38                 }
39             }
40             Console.ReadKey();
```

```

41     }
42 }
43 }

```

第 6 行代码定义了一个枚举，它位于一个类的内部，是以类的成员的身份定义的，它也可以定义在类外。那样就可以像一个类一样的使用它。定义枚举时，使用 `enum` 关键字，后跟枚举的名称，然后跟一对大括号，括号中是命名常量，每个命名常量以逗号分隔，结尾的大括号不以分号结尾。在这行代码中定义了 7 个命名常量，分别表示一周的每一天。

在 `Main` 方法内部，第 10 行定义了无限 `for` 循环，它的中断依赖于第 13 行对输入的字符串的判断，如果输入的是“q”，则使用 `break` 语句中断循环。

第 15 行代码开始的是一个 `switch` 语句，它对输入的字符串做判断，然后在相应的 `case` 语句中输出相应的每一天的名字。

以第 18 行代码为例，介绍一下枚举类型的使用。枚举类型的使用就是用枚举类型的名字后跟“.”运算符，然后跟命名常量。例如第 18 行代码中的 `Week.Monday`。这段代码的输出结果如下：

请输入 1-7 的数字：

```

1
Today is Monday
3
Today is Wednesday
4
Today is Thursday
5
Today is Friday
q

```

7.2 枚举类型的基础类型

枚举类型中的成员叫做命名常量，常量意味着它的值初始化后就不能再进行改变了。看下面的实例代码：

```

01 using System;
02 namespace Conversion
03 {
04     enum Color {Red, Yellow, Blue}
05     class Program
06     {
07         static void Main(string[] args)
08         {
09             Console.WriteLine("请输入 1-3 的数字:");
10             for (; ; )
11             {
12                 string i = Console.ReadLine();
13                 if (i == "q")
14                     break;
15                 switch (i)
16                 {
17                     case "1":
18                         Console.WriteLine("选择的颜色是：{0}", Color.Red);

```

```

19             break;
20         case "2":
21             Console.WriteLine("选择的颜色是: {0}", Color.Yellow);
22             break;
23         case "3":
24             Console.WriteLine("选择的颜色是: {0}", Color.Blue);
25             break;
26         case "4":
27             Color.Blue = (int)Color.Blue + 1;
28             break;
29     }
30 }
31 Console.ReadKey();
32 }
33 }
34 }

```

这段代码中的第 27 行代码将枚举类型中的一个成员显式转换成 int 类型，然后再进行加 1 操作，然后又对这个枚举成员进行了赋值。为何枚举成员可以转换成整型呢？这是因为每个枚举类型都有一个基础类型，这个基础类型必须是 byte、sbyte、short、ushort、int、uint、long 或 ulong 中的一种。没有显式声明基础类型的情况下，它是 int 类型。但是这段代码编译器会给出一个错误提示如下：

错误 1 赋值号左边必须是变量、属性或索引器

这就是因为枚举类型的命名成员是常量的原因，它初始化后是不可以改变其值的。下面再看一段代码：

```

01 using System;
02 namespace Conversion
03 {
04     enum Color:short {Red, Yellow, Blue}
05     enum Position { Left = -3,Center = 5,Right}
06     class Program
07     {
08         static void Main(string[] args)
09         {
10             Console.WriteLine("枚举类 Color 中的成员值为: {0}, {1},
11                               {2}", (int)Color.Red, (int)Color.Yellow, (int)Color.Blue);
12             Console.WriteLine("枚举类 Position 中的成员值为: {0}, {1}, {2}",
13                               (int)Position.Left, (int)Position.Center, (int)Position.Right);
14             Console.ReadKey();
15         }
16     }
17 }

```

第 4 行和第 5 行代码各定义了一个枚举，它们都是在类外定义的。第 4 行代码在枚举名称后面跟了一个冒号，然后跟一个 short 类型。这个语法的作用是指定枚举的基础类型为 short。枚举类型的命名常量每个都有一个具体的值，如果不指定具体值，它默认是从 0 开始的。也就是可以看成，枚举类型的成员名称是 Red, Yellow 和 Blue，它们中的每个又有一个具体的值，从 0 开始，依次递增。因此，Red 为 0, Yellow 为 1, Blue 为 2。

当然，这个值也可以初始化时指定，这就是第 5 行代码的形式，初始化时使用赋值运算符指定具体的值。如果它后面的命名常量不指定值，则从这个指定的值开始递增。

第 10 行代码和第 11 行代码将枚举成员转换成 int 类型然后打印其值。这段代码的执行结果如下：

枚举类 Color 中的成员值为：0, 1, 2

枚举类 Position 中的成员值为：-3, 5, 6

7.3 枚举类型的装箱转换

枚举类型在编译时会在后台实例化为一个派生自 System.Enum 类的结构。因此，枚举类型可以调用 System.Enum 类的成员。实例代码如下：

```
01 using System;
02 namespace Conversion
03 {
04     enum Color:short {Red, Yellow, Blue}
05     class Program
06     {
07         static void Main(string[] args)
08         {
09             Color c = Color.Red;
10             TypeCode tc = c.GetTypeCode();
11             Console.WriteLine(tc.ToString());
12             Console.ReadKey();
13         }
14     }
15 }
```

第 4 行代码定义了一个枚举，注意它的基础类型是 short 类型，也就是 System.Int16 类型。第 9 行代码定义了一个 Color 类型的变量，并且为它赋值为 Color.Red。

第 10 行代码定义了一个 TypeCode 类型的变量，TypeCode 是 .Net 类库中的一个枚举类型。接下来使用变量 c 调用了继承自 System.Enum 类型的方法 GetTypeCode()。这个方法返回 TypeCode 类型，它表示枚举的基础类型名。

第 11 行代码调用了 TypeCode 类型变量 tc 的 ToString() 方法，它将打印 c 的基础类型名称。这段代码的执行结果如下：

Int16

既然枚举继承自 System.Enum 类型，而枚举类型又是值类型，因此对它可以进行装箱操作。实例代码如下：

```
01 using System;
02 namespace Conversion
03 {
04     enum Color:short {Red, Yellow, Blue}
05     class Program
06     {
07         static void Main(string[] args)
08         {
09             Color c = Color.Red;
10             System.Enum e = c;
11             object o = c;
```



```

12         Console.ReadKey();
13     }
14 }
15 }

```

代码的第 9 行定义了一个枚举类型的变量，然后第 10 行和第 11 行分别用 `System.Enum` 类型和 `object` 类型对它进行了装箱操作。可以看到，装箱操作和基本数据类型的装箱操作没有什么两样。装箱操作后，枚举结构将由栈中复制到堆中的实例中。

7.4 隐式枚举转换

隐式枚举转换允许将 0 隐式转换为任何枚举类型。下面看代码实例：

```

01 using System;
02 namespace Conversion
03 {
04     enum Color:short {Red, Yellow, Blue}
05     enum Position {Left = 2,Center,Right}
06     class Program
07     {
08         static void Main(string[] args)
09         {
10             Color c = 0;
11             Console.WriteLine(c.ToString());
12             Position p = 0;
13             Console.WriteLine(p.ToString());
14             Position pl = (Position)3;
15             Console.WriteLine(pl.ToString());
16             Console.ReadKey();
17         }
18     }
19 }

```

第 4 行代码和第 5 行代码分别定义了两个枚举类型，枚举类型 `Color` 采用了枚举类型的默认值，而 `Position` 指定了命名常量的值，这样它的成员就不再有值 0 了。

第 10 行代码将文本 0 通过隐式转换赋值给了一个 `Color` 类型的变量，因为 `Color` 枚举含有值为 0 的成员，所以第 11 行代码可以打印出 `Red`。

第 12 行代码将文本 0 隐式转换为 `Position` 类型的变量，但是因为 `Position` 类型不再有值 0，因此第 13 行代码将打印出 0。

除了可以将 0 隐式转换为枚举类型之外，也可以将枚举类型的基础类型的值通过显式转换的方式转换为枚举类型，例如第 14 行代码，第 15 行代码将打印出 `Center`。代码的执行结果如下：

```

Red
0
Center

```

不只是基础类型中对于枚举成员指定的值可以通过显式转换的方式赋值给枚举类型，对于枚举成员没有指定的基础类型的其它值也可以通过显式转换的方式赋值给枚举类型。例如如果第 14 行代码将 20 显式转换给枚举类型，则第 15 行代码将打印出 20。

7.5 显式枚举转换

前面代码中的第 14 行代码就是一种显式枚举转换。显式枚举转换包括以下类型：

- 从 sbyte、byte、short、ushort、int、uint、long、ulong、char、float、double 或 decimal 到任何枚举类型。
- 从任何枚举类型到 sbyte、byte、short、ushort、int、uint、long、ulong、char、float、double 或 decimal。
- 从任何枚举类型到任何其他枚举类型。

下面看代码实例：

```
01 using System;
02 namespace Conversion
03 {
04     enum Color {Red, Yellow, Blue}
05     enum Position {Left = 2, Center, Right}
06     class Program
07     {
08         static void Main(string[] args)
09         {
10             byte a = 33;
11             Color _a = (Color)a;
12             Console.WriteLine(_a);
13             ulong b = 9999999999UL;
14             Color _b = (Color)b;
15             Console.WriteLine(_b);
16             long c = (long)Color.Yellow;
17             Console.WriteLine(c);
18             byte d = (byte)Color.Blue;
19             Console.WriteLine(d);
20             Color e = (Color)Position.Left;
21             Console.WriteLine(e);
22             Color f = (Color)Position.Right;
23             Console.WriteLine(f);
24             Console.ReadKey();
25         }
26     }
27 }
```

显式枚举转换通过将参与的枚举都按该枚举的基础类型进行处理。然后按照两种基础类型之间发生的隐式转换或显式转换进行。

首先代码的第 4 行和第 5 行分别定义了一个基础类型为 int 的枚举。

第 10 行代码定义了一个 byte 类型的变量，第 11 行将它显式转换给一个 Color 类型的枚举变量。因为从 byte 到 int 存在隐式转换，并且 Color 枚举里面没有关联值为 33 的成员。所以第 12 行将打印输出 33。

第 13 行定义了一个 ulong 变量 b，然后第 14 行将它通过显式转换的方式转换成 Color 类型。因为从 ulong 到 int 类型之间只存在显式转换，而且编译器的溢出检查是关闭状态，所以超出 int 类型范围的值发生了溢出。第 15 行将打印出一个未指定的值。

第 16 行将一个枚举值通过显式转换的方法转换成一个 long 类型的值，因为从 int 到 long 存在显式转换，所以第 17 行打印出了 Color.Yellow 的关联值 1。

第 18 行将 Color.Blue 强制转换成 byte 类型，在后台其实是将枚举的基础类型 int 强制转换成 byte

类型。因为从 int 到 byte 不存在隐式转换，所以只能发生显式转换。但是 Color.Blue 的关联值 2 没有超出 byte 范围，所以没有发生溢出。第 19 行将打印出 2。

第 20 行表现的是不同的枚举类型之间发生的显式转换，其实在后台发生的也是它们之间的基础类型之间的转换。因为 Position.Left 的关联值为 2，所以显式转换为 Color 类型之后，这个 Color 类型对于关联值为 2 有成员对应。因此第 21 行将打印出关联值为 2 的成员 Blue。

第 22 行将 Position.Right 成员强制转换成 Color 类型，因为它们之间的关联值没有对应，所以第 23 行将打印出关联值 4。这段代码的执行结果如下：

```
33
1215752191
1
2
Blue
4
```

7.6 枚举成员的关联值

在枚举中的每个命名常量的数值就是枚举成员的关联值，如果不显式指定，枚举成员的关联值从 0 开始依次递增，也就是说，第一个成员的关联值为 0，紧挨着它的第二个成员的关联值就是第一个关联值再加 1。然后依次递增。实例代码如下：

```
01 using System;
02 namespace Conversion
03 {
04     enum Color {Red, Yellow, Blue = Yellow, Green = Blue + 1}
05     class Program
06     {
07         static void Main(string[] args)
08         {
09             Console.WriteLine("{0}, {1}, {2}, {3}", (int)Color.Red, (int)Color.Yellow,
10                 (int)Color.Blue, (int)Color.Green);
11             Console.ReadKey();
12         }
13     }
```

这段代码演示了枚举成员的关联值的应用。从代码中可以看到，枚举成员的关联值可以重复。例如第 4 行代码中枚举的定义，Blue 成员的关联值就是使用的它的前一个成员 Yellow 的值。而最后一个成员 Green 使用的是它的前一个成员的关联值再加 1 的结果。这段代码的执行结果如下：

```
0, 1, 1, 2
```

但是需要注意的是，前面的成员不能使用它的依赖成员的关联值，否则会造成循环。下面是代码实例：

```
01 using System;
02 namespace Conversion
03 {
04     enum Color {Red = Yellow + 1, Yellow, Blue = Yellow}
05     class Program
06     {
07         static void Main(string[] args)
08         {
```

```

09         Console.WriteLine("{0},{1},{2}", (int)Color.Red, (int)Color.Yellow,
           (int)Color.Blue);
10         Console.ReadKey();
11     }
12 }
13 }

```

第 4 行代码定义枚举成员的时候，前面的成员使用了依赖它的成员的关联值。这样就造成了常量值定义上的循环，它将得到如下的错误信息：

错误 1 “Conversion.Color.Red” 的常量值计算涉及循环定义

从上面的代码也可以看出，一些算术运算符同样适用于枚举成员。因为枚举成员之间的运算就是其基础类型之间的运算。

7.7 枚举类型的拆箱转换

枚举类型的拆箱转换是装箱转换的逆运算，它将一个引用类型转换为一个值类型。拆箱转换也很简单，代码实例如下：

```

01 using System;
02 namespace Conversion
03 {
04     enum Color {Red, Yellow, Blue = Yellow}
05     class Program
06     {
07         static void Main(string[] args)
08         {
09             Color c = Color.Red;
10             System.Enum d = c;
11             object o = c;
12             Color e = (Color)d;
13             Color f = (Color)o;
14             Console.WriteLine(e);
15             Console.WriteLine(f);
16             Console.ReadKey();
17         }
18     }
19 }

```

代码的第 9 行定义了一个 Color 类型的变量，然后第 10 行和第 11 行分别对它进行了装箱操作，这时，这个 Color 类型的变量就被复制到了堆中的实例上。第 12 行和第 13 行分别用一个 Color 类型的变量接收拆箱后的值。这时，堆中的内容又被复制到了栈中，成为了一个值类型。

7.8 可空类型

可空类型有一个基础类型，它表示的是除了其基础类型可表示的值外，还可以表示一个 null 值。它的基础类型包括任何不可以为 null 的值类型。为何要定义一个这样的可空类型呢，比如数据库应用中，数据库的某个字段可以为 null，当用一个值类型来接收它的值的时候，比如数据库中的内容是 int 类型，那么就需要一个 int 类型的变量来接收，但是还需要这个变量可以接收现在的 null 值。下面是代码实例：

```

01 using System;
02 namespace Conversion
03 {

```

```

04     enum Color {Red, Yellow, Blue = Yellow}
05     class Program
06     {
07         static void Main(string[] args)
08         {
09             int? a = null;
10             Color? b = null;
11             b = Color.Blue;
12             float? c = null;
13             c = 33.33F;
14             Console.WriteLine("{0}, {1}, {2}", a, b, c);
15             Console.ReadKey();
16         }
17     }
18 }

```

代码的第 9 行、第 10 行和第 12 行都是可空类型的定义，它的语法形式很简单，就是基本类型再加一个问号。当在第 14 行打印的时候，因为变量 a 为 null，所以这一行什么也打印不出来。下面是执行结果：

```
,Yellow,33.33
```

这种基本类型名称加一个问号的语法是 .Net 类库中 `System.Nullable<T>` 的形式的简化写法。`System.Nullable<T>` 是一个泛型结构。它对使用这个结构的类型有约束条件，要求必须也得是结构才行。也就是说，必须是值类型，引用类型不行。

那么既然可空类型是一种结构，那么它也有属于它自己的属性成员，代码如下：

```

01 using System;
02 namespace Conversion
03 {
04     enum Color {Red, Yellow, Blue}
05     class Program
06     {
07         static void Main(string[] args)
08         {
09             Color? c = Color.Blue;
10             Console.Write("可空类型 Color 是否有值: ");
11             Console.WriteLine(c.HasValue);
12             Console.Write("可空类型 Color 的值是: ");
13             Console.WriteLine(c.Value);
14             Console.ReadKey();
15         }
16     }
17 }

```

这段代码的第 11 行和第 13 行各调用了可空类型的两个成员属性，一个是 `HasValue`，它用来判断可空类型是否为空；另一个是 `Value`，它用来得到可空类型的值。这段代码的执行结果如下：

```

可空类型 Color 是否有值: True
可空类型 Color 的值是: Blue

```

7.8.1 可空类型的隐式转换

从基础类型到可空类型以及从 null 文本到可空类型都存在隐式转换。适合于非可空值类型的隐式转换同样适用于相对应的可空类型。实例代码如下：

```
01 using System;
02 namespace Conversion
03 {
04     enum Color {Red, Yellow, Blue}
05     class Program
06     {
07         static void Main(string[] args)
08         {
09             int? a = null;
10             int b = 33;
11             int? _b = b;
12             int? c = 345;
13             long? _c = c;
14             int? d = null;
15             long? _d = d;
16             Console.WriteLine(_d.HasValue);
17             Console.ReadKey();
18         }
19     }
20 }
```

这段代码的第 9 行是从 null 文本到可空类型之间的隐式转换。第 10 行和第 11 行是从基础类型向可空类型之间的隐式转换。第 12 行和第 13 行是可空类型之间的隐式转换。第 14 行和第 15 行是一个为 null 的可空类型向另一个可空类型之间的隐式转换，转换后，目标可空类型也为 null。当第 16 行代码打印其 HasValue 属性的时候为 false。代码的执行结果如下：

```
False
```

因为可空类型和基础类型属于不同的结构，所以在其间进行转换的时候，经历如下过程，当从可空类型向可空类型转换的时候，首先可空类型要进行解包，变成相对应的基础类型，然后完成基础类型之间的转换，最后将结果再打包为可空类型。

如果是非可空值类型向可空类型转换，例如本实例的第 10 行和第 11 行，则非可空类型首先向目标可空类型对应的非可空类型进行转换，然后再打包为可空类型。

对于基础类型为枚举的可空类型，下面的隐式转换是存在的：

```
Color? c = 0;
```

不但 0 可以向非可空枚举类型进行隐式转换，对于可空类型同样存在隐式转换。

7.8.2 可空类型的装箱转换

可空类型前面介绍过，它是一个泛型结构，它的基础类型是不可以为 null 的值类型，也就是说，int?? 这样的语法是错误的，可空类型的基础类型不能再是一个可空类型。可空类型进行装箱的时候，很相似，它装箱后复制到堆中实例的类型也不能是一个可空类型，而是可空类型的基础类型。看下面的实例代码：

```
01 using System;
02 namespace Conversion
03 {
04     class Program
05     {
```

```
06      static void Main(string[] args)
07      {
08          int? a = null;
09          object _a = a;
10          if (_a == null)
11          {
12              Console.WriteLine("_a 为空引用");
13          }
14          int? b = 25;
15          object _b = b;
16          Console.ReadKey();
17      }
18  }
19  }
```

第 8 行代码定义了一个可空的 `int` 类型，并且将它初始化为 `null`。第 9 行代码将它进行了装箱操作。当一个为 `null` 的可空类型装箱时，实际上目标类型就是一个空引用。因此，第 10 行判断目标类型是否为 `null` 时，将打印出“_a 为空引用”。

第 14 行代码定义了一个可空类型并初始化为 25，然后将它进行了装箱操作。编译器在背后所做的其实是首先将可空类型 `b` 解包成基础类型 `int`，然后对这个基础类型进行了装箱操作。这段代码的执行结果如下：

_a 为空引用

7.8.3 可空类型的显式转换

不可为空的基础值类型之间的显式转换规则同样适用于可以为空的类型之间的显式转换。下面看代码实例：

```
01  using System;
02  namespace Conversion
03  {
04      class Program
05      {
06          static void Main(string[] args)
07          {
08              int? a = 333;
09              sbyte? _a = (sbyte?)a;
10              int b = 99999999;
11              sbyte? _b = (sbyte?)b;
12              Console.WriteLine(_b);
13              int? c = 99999999;
14              sbyte _c = (sbyte)c;
15              Console.WriteLine(_c);
16              int? d = null;
17              sbyte? _d = (sbyte?)d;
18              Console.ReadKey();
19          }
20      }
```

```
21 }
```

这段代码的第 8 行和第 9 行演示的是可空类型之间的显式转换。可空类型之间的显式转换原理是，首先将可空类型解包为基础类型，例如本例中的 a 首先被解包为 int 类型，然后它向 sbyte 类型进行显式转换，最后将结果再装包为可空类型。

第 10 行代码和第 11 行代码演示的是将基础类型显式转换为可空类型。它的转换原理是，b 首先向 sbyte 进行显式转换，然后再将结果装包为可空类型。

第 13 行代码和第 14 行代码演示的是可空类型向基础类型的显式转换。它的转换原理是，c 首先解包为基础类型，然后它向 sbyte 进行显式转换。

第 16 行演示的是为 null 的可空类型之间的显式转换。因为源操作数为 null，所以转换后的可空类型也为 null。这段代码的执行结果如下：

```
-1
-1
```

可以看到，因为溢出检查是关闭状态，所以结果发生了溢出。它的结果和基础值类型之间的显式转换结果是一样的。

7.8.4 可空类型的拆箱转换

可空类型的拆箱转换其实就是可空类型的显式转换的一种。下面看代码实例：

```
01 using System;
02 namespace NullableClass
03 {
04     class Program
05     {
06         static void Main(string[] args)
07         {
08             int? a = 333;
09             object _a = a;
10             int b = (int)_a;
11             int? c = (int?)_a;
12             int? d = null;
13             object _d = d;
14             int e = (int)_d;
15             Console.ReadKey();
16         }
17     }
18 }
```

这段代码的第 8 行和第 9 行是将一个可空类型进行装箱的操作。可空类型装箱时，实际上发生的过程是首先将可空类型解包成基础类型，然后再对基础类型进行装箱。

第 10 行和第 11 行是对装箱后的值进行拆箱转换。不同的是，第 10 行代码是用一个基础类型来接收拆箱后的值。而第 11 行代码是用一个可空类型来接收拆箱后的值。当使用可空类型接收拆箱后的值时，背后发生的操作是，拆箱后的值是一个基础类型值，它通过包装操作将结果又包装成了可空类型。

第 12 行代码定义了一个可空类型的变量并赋值为 null，然后第 13 行代码对它进行装箱操作。这时_d 变量也是一个 null 引用。当第 14 行代码使用一个基础类型变量来接收拆箱后的结果时，就会发生异常，异常信息如下：

```
未处理 NullReferenceException
未将对象引用设置到对象的实例。
```


所以，当对一个值为 `null` 的可空类型装箱时，拆箱不能用基础类型来接收拆箱值，那样将会出现异常。只能再用一个可空类型变量来接收拆箱值，最后这个可空类型变量的值也为 `null`。

第八章 类类型

8.1 最简单类的定义

前面在介绍到数组的时候已经大体介绍过类类型。类类型是 C# 中一种最基础的类型，它是一种引用类型。当定义类的时候，实际上是定义了一种数据结构的模板，程序员可以使用这个模板创造出类的实例。这个实例位于内存的堆中，而引用类型的变量位于内存的栈中，它指向了堆中的实例。堆中的实例是实实在在的数据结构。它包括字段和操作。因为定义类的实例的时候经常使用 `new` 关键字，所以也通俗的讲实例是 `new` 出来的。类的实例也叫做对象。同时，类也支持继承和多态，通过这两种机制，派生类（子类）可以扩展和专用化基类（父类）。下面看代码实例，它定义了一个最简单的类：

```
01 using System;
02 namespace ClassType
03 {
04     public class TV
05     {
06         private int channel; //电视机的频道
07         private int volume; //电视机的音量
08         /// <summary>
09         /// 增加频道
10         /// </summary>
11         public void ChPlus()
12         {
13             channel++;
14         }
15         /// <summary>
16         /// 增加音量
17         /// </summary>
18         public void VolPlus()
19         {
20             volume++;
21         }
22         /// <summary>
23         /// 将信息显示在电视机屏幕上
24         /// </summary>
25         public void Show()
26         {
27             Console.WriteLine("频道: {0}", channel);
28             Console.WriteLine("音量: {0}", volume);
29         }
30     }
31     class Program
32     {
33         static void Main(string[] args)
34         {
35             TV myTv = new TV();
```

```

36         myTv.ChPlus();
37         myTv.VolPlus();
38         myTv.Show();
39         Console.ReadKey();
40     }
41 }
42 }

```

这段代码的第 4 行到第 30 行是一个最简单的类的定义。它包含了类中最基本的字段和方法。第 4 行代码定义了类的名称，当定义类的时候，使用 `class` 关键字。在 `class` 关键字的前面是修饰符，第 4 行代码使用了 `public` 关键字作为修饰符，它表示这个类的访问权限是公开的。公开的权限意味着在本程序集和其它任何程序集中都可以访问这个类，使用这个类来实例化对象，也就是说，它的访问不受任何限制。在 `class` 关键字后面是类的名字，因为要模拟一台电视机，所以这里以 `TV` 作为名字。在 C# 中，当定义类的名字的时候，习惯上使用首字母大写的形式来定义名称。

接下来就是类体，类体由一对大括号组成，在类体中定义类的成员。这对大括号位于代码的第 5 行和第 30 行，类的定义不以分号结尾，因此第 30 行的右大括号没有分号结尾。

代码的第 6 行和第 7 行分别定义了两个字段，它们代表电视机的频道和音量。字段是一种变量，根据使用目的不同，它和类或类的实例相关联。本例的这两个字段都是和类的实例相关联，要访问它们，都需要通过类的实例来访问。当定义字段的时候，首先需要指定字段的类型，本例的两个字段都是 `int` 类型的。在字段类型的前面是字段的访问权限，本例使用了 `private` 关键字，它表示字段是私有的访问权限，只能在类的定义内部被访问。在字段类型的后面是字段的名称，因为一个表示频道，另一个表示音量，所以用 `channel` 和 `volume` 来表示。在 C# 中，字段的名称通常习惯上以小写字母开头。字段的定义都是以分号结尾。

代码的第 11 行、第 18 行和第 25 行分别定义了三个方法，这三个方法都以 `public` 关键字作为修饰符，它表示这三个方法都能在类的定义外部以实例名进行调用，因为这个类是公开访问权限的，这三个方法也是公开访问权限的，所以这三个方法能以实例名调用的方式在本程序集和其它程序集中进行调用。

第 31 行代码是定义了一个包含 `Main` 方法的类，这个类没有访问权限修饰符，类如果没有访问权限修饰符，那么它的访问权限默认是 `internal` 的，它只能被本程序集访问。

第 35 行代码首先定义了一个 `TV` 类的变量，然后使用 `new` 关键字在堆中创建了一个 `TV` 类的实例。创建类的实例的语法是 `new` 关键字后跟类名，然后是一对小括号。小括号中是传递给类的构造方法的参数。因为这个类没有定义任何构造方法，所以这里实际调用的是编译器在后台自动生成的默认构造方法，它没有方法参数。

接下来，第 36 行、第 37 行和第 38 行分别以实例名称调用了这个类的三个方法。方法实际上就是可由类或实例调用的操作。因为这三个方法都是属于实例的方法，所以调用方法就是用实例名后跟“.”运算符，然后再跟方法名称，方法名称后面的一对括号中是传递给方法的参数。这段代码的执行结果如下：

```

频道: 1
音量: 1

```

8.2 静态字段和实例字段

前面例子中的字段是属于每个实例的，因此它是实例字段。还有一种字段是属于类的，这种字段叫做静态字段。什么时候需要实例字段以及什么时候需要定义静态字段，要看代码编写的需要。如果一个字段在这个类的每个实例中都存在，而且可能各不相同，每个实例都有对它进行设置的需要，那么它就应该定义成一个实例字段。如果一个字段是类的各个实例共同的特点，则这个字段需要设置成静态字段。当定义成静态字段之后，每个实例对它的改动和设置，都会影响其它的实例。代码实例如下：

```

01 using System;
02 namespace ClassType
03 {

```

```
04     public class TV
05     {
06         private int channel; //电视机的频道
07         private int volume; //电视机的音量
08         public static string model = "39 英寸液晶"; //型号
09         /// <summary>
10         /// 增加频道
11         /// </summary>
12         public void ChPlus()
13         {
14             channel++;
15         }
16         /// <summary>
17         /// 增加音量
18         /// </summary>
19         public void VolPlus()
20         {
21             volume++;
22         }
23         /// <summary>
24         /// 将信息显示在电视机屏幕上
25         /// </summary>
26         public void Show()
27         {
28             Console.WriteLine("电视机型号是: {0}", model);
29             Console.WriteLine("频道: {0}", channel);
30             Console.WriteLine("音量: {0}", volume);
31         }
32     }
33     class Program
34     {
35         static void Main(string[] args)
36         {
37             TV myTv1 = new TV();
38             TV myTv2 = new TV();
39             Console.WriteLine("myTv1 初始屏幕上的显示是: ");
40             myTv1.Show();
41             Console.WriteLine("myTv2 初始屏幕上的显示是: ");
42             myTv2.Show();
43             TV.model = "42 英寸液晶";
44             myTv2.ChPlus();
45             Console.WriteLine("myTv1 现在屏幕上的显示是: ");
46             myTv1.Show();
47             Console.WriteLine("myTv2 现在屏幕上的显示是: ");
```

```
48         myTv2.Show();
49         Console.ReadKey();
50     }
51 }
52 }
```

这段代码的第 8 行新加入了一个字段 `model`，这个字段的修饰符中多了一个关键字 `static`。它表示这个字段是属于类的。由 `static` 关键字声明的字段叫做静态字段，静态字段在这个类的所有实例间共享。不管这个类的实例有多少个，静态字段只存在一个。首先来看这段代码的执行结果：

```
myTv1 初始屏幕上的显示是：
电视机型号是：39 英寸液晶
频道：0
音量：0
myTv2 初始屏幕上的显示是：
电视机型号是：39 英寸液晶
频道：0
音量：0
myTv1 现在屏幕上的显示是：
电视机型号是：42 英寸液晶
频道：0
音量：0
myTv2 现在屏幕上的显示是：
电视机型号是：42 英寸液晶
频道：1
音量：0
```

第 37 行代码和第 38 行代码各定义了一个 TV 类的实例，然后调用它们各自的 `Show()` 方法，这时因为两个实例都是执行的初始化操作，所以各个字段都是相同的。对于静态字段来说，它们都调用了静态字段。

接下来第 43 行代码将类的静态字段 `model` 改写，注意这里的调用方式是以类名调用的方式。类中的实例方法也可以调用静态字段，在类的定义中，实例方法调用静态字段不需要加类名调用。但是在类的外部需要类名来调用。

第 44 行代码中，`myTv2` 又调用了自己的实例方法，对实例字段 `channel` 进行了改写。然后在第 46 行和第 48 行又一次调用了实例方法 `Show()`。这时可以看到，对静态字段的改动影响了每个实例，而对 `myTv2` 实例字段的改动却只影响了它自己。

这就是实例字段和静态字段的区别，静态字段属于类的每个实例，它可以看做是公有财产。而实例字段只属于某个实例，它是每个实例的私有财产。当创建类的实例时，每个实例都包含了类的实例字段的一个副本。每个实例的实例字段集合之间都是独立的。

8.3 字段的初始化

对于一个类的字段来说，不论是静态字段还是实例字段，它们都有一个初始值。静态字段在类初始化时，它被自动设定为它的初始值；而实例字段在对象的实例化时首先被自动设定为它的初始值。因此，相对于前面介绍过的局部变量而言，字段永远都不可能遇到未初始化的错误。它的初始化是自动发生的。而他们初始化后的值就是它们所属类型的默认值。下面是实例代码：

```
01 using System;
02 namespace ClassType1
03 {
```

```

04     class Program
05     {
06         static int a;
07         static bool b;
08         private float c;
09         private string[] s;
10         static void Main(string[] args)
11         {
12             Console.WriteLine("静态字段 a 的默认值是: {0}", a);
13             Console.WriteLine("静态字段 b 的默认值是: {0}", b);
14             Program p = new Program();
15             Console.WriteLine("实例字段 c 的默认值是: {0}", p.c);
16             if (p.s == null)
17             {
18                 Console.WriteLine("实例字段 s 的默认值是: null");
19             }
20             Console.ReadKey();
21         }
22     }
23 }

```

这段代码的第 6 行和第 7 行各定义了一个静态字段，类型分别是 `int` 和 `bool`。第 8 行代码定义了一个实例字段，它的类型是 `float`。第 9 行定义了一个实例字段，它是一个引用类型，注意这里是一个要指向 `string` 类型数组的引用变量，而不是一个数组，因为它并没有实例化。

在这 4 行代码中，会发现前两个静态字段没有使用访问权限修饰符，如果字段没有使用访问权限修饰符，那它默认就是 `private` 的。它只能被类的定义内部访问。

在第 10 行是 `Main` 方法，因为 `Main` 方法必须是静态的，所以 `Main` 方法前面的 `static` 关键字是必不可少的。但是本例中的 `Main` 方法也没有访问权限关键字，同样，默认它是 `private` 的。那么，是否意味着它不能被类外部调用呢？不是这样的，将 `Main` 方法定义成 `private`，是防止其它程序集对它的访问。而操作系统是可以直接调用它的，不受它的 `private` 的影响。

在第 12 行和第 13 行输出了静态字段的默认值。对于整型和浮点型的字段来说，默认值都为 0，对于 `bool` 类型来说，默认值是 `false`。

在第 14 行创建了一个 `Program` 类的实例 `p`，通过它来调用实例字段 `c` 并在第 15 行代码输出。它显然会输出 0。

第 16 行代码对引用类型的变量 `s` 进行了判断，如果它是一个空引用，则执行 `if` 语句块中的代码。对于引用类型的字段来说，它的初始默认值就是 `null`。这段代码的执行结果如下：

```

静态字段 a 的默认值是: 0
静态字段 b 的默认值是: False
实例字段 c 的默认值是: 0
实例字段 s 的默认值是: null

```

8.4 字段的初始值设定项

上一小节的例子演示了当类初始化时和创建类的实例时，首先发生的字段操作就是将静态字段和实例字段分别设置为默认值。那么接下来编译器要做的就是看字段有没有初始值设定项，如果有初始值设定项，则执行初始值设定项对字段进行进一步的设置。代码实例如下：

```

01     using System;

```

```

02 namespace ClassType1
03 {
04     class Program
05     {
06         private static int a = 11;
07         private static bool b = true;
08         private float c = 3.14159F;
09         private string[] s = {"Hello", "World"};
10         static void Main(string[] args)
11         {
12             Console.WriteLine("静态字段 a 的初始值是: {0}", a);
13             Console.WriteLine("静态字段 b 的初始值是: {0}", b);
14             Program p = new Program();
15             Console.WriteLine("实例字段 c 的初始值是: {0}", p.c);
16             Console.WriteLine("实例字段 s 的初始值是:");
17             foreach (string i in p.s)
18             {
19                 Console.Write(i + " ");
20             }
21             Console.ReadKey();
22         }
23     }
24 }

```

这段代码的第 6 行到第 9 行分别定义了 4 个 private 权限的字段，前两个是静态字段，后两个是实例字段。在每个字段定义之后都有一个赋值语句，这就是字段的初始值设定项。类的字段成员通常都将访问权限设置为 private，这么做的原因是为了保护数据的完整性。如果这些私有数据没有必要对外公开，但是将它们的访问权限设置为 public 了，这时在开发环境的智能提示中就会看到这些成员。这样一来，一是智能提示比较混乱，二是不能随便修改的数据公开后会带来麻烦。

这段代码中，当类初始化时，首先静态字段都会被设置成它们类型的默认值，然后按照定义的文本顺序依次执行它们的初始值设定项。因此第 6 行代码和第 7 行代码的字段 a 首先被设置为 0，然后 b 设置为 false。接下来执行初始值设定项，首先 a 被设置为 11，然后 b 被设置为 true。那么什么时候发生类的初始化呢？类的初始化发生在访问类的静态成员的前一刻，也就是第 12 行代码要执行前。

当第 14 行代码执行时，它要在堆中创建一个类的实例。这时，实例字段 c 首先被初始化为 0，实例字段 s 被初始化为 null。然后按照定义的文本顺序执行它们的初始值设定项，这时，c 被设置为 3.14159，而 s 也要被实例化，它指向了堆中的 string 类型的一维数组，这个数组包含两个成员“Hello”和“World”。

当打印类的静态字段和实例字段的值时，将输出它们的初始值设定项设置的值。代码执行结果如下：

```

静态字段 a 的初始值是: 11
静态字段 b 的初始值是: True
实例字段 c 的初始值是: 3.14159
实例字段 s 的初始值是:
Hello World

```

为了说明字段的默认值和初始值设定项执行的先后顺序，下面再以一段代码实例来演示：

```

01 using System;
02 namespace ClassType1

```

```

03  {
04      class Program
05      {
06          private static int a = (int)b + 1;
07          private static float b = a + 1;
08          static void Main(string[] args)
09          {
10              Console.WriteLine("a 的值是: {0}, b 的值是: {1}", a, b);
11              Console.ReadKey();
12          }
13      }
14  }

```

这段代码的第 6 行的静态字段 a 的初始值设定项引用了 b 的值。对于这种容易造成循环的编程方式应尽量避免。这里只是为了说明代码执行的顺序。那么这两个字段的执行顺序是怎样的呢？首先 a 和 b 都被设置为默认值 0，然后按照文本定义顺序首先执行字段 a 的初始值设定项，这时 b 为 0，那么执行算术运算后，a 的值为 1。接下来执行 b 的初始值设定项，因为这时 a 为 1，所以执行算术运算后，b 为 2。代码的执行结果如下：

a 的值是: 1, b 的值是: 2

需要注意，这样的初始值设定项只能为静态字段，如果定义成实例字段，将会得到下面的错误提示：

错误 1 字段初始值无法引用非静态字段、方法或属性 “ClassType1.Program.b”
 错误 2 字段初始值无法引用非静态字段、方法或属性 “ClassType1.Program.a”
 错误 3 非静态的字段、方法或属性 “ClassType1.Program.a” 要求对象引用
 错误 4 非静态的字段、方法或属性 “ClassType1.Program.b” 要求对象引用

这是因为当字段为实例字段时，必须有类的实例的创建才能访问它们。因此，除非是静态字段，否则这样的直接引用就是错误的。

8.5 常量

常量的值在初始化后不能改变，常量可以定义为类的成员，也可以定义在方法中。定义常量使用 const 关键字。下面是代码实例：

```

01  using System;
02  namespace ClassType1
03  {
04      enum Color {Red, Yellow, Blue};
05      class A
06      {
07          public static string s = "我是类 A 中的静态字段 s";
08      }
09      class Program
10      {
11          private const int a = 11;
12          private const Color c = Color.Red;
13          private const string s = "Hello World";
14          private const A myA = new A();
15          static void Main(string[] args)
16          {

```



```

17         const int b = 12;
18         Console.WriteLine(A.s);
19         Console.WriteLine(Program.s);
20         Console.ReadKey();
21     }
22 }
23 }

```

这段代码中的第 4 行定义了一个枚举类型，第 5 行定义了一个类，在这个类中定义了一个静态字段，并做了初始化。

在类 Program 中定义了 4 个常量成员，常量在定义时必须初始化，也就是在编译时必须能明确其值。因此引用类型中除了 string 类型外，都不允许使用 new 关键字作初始化。也就是说，在常量的初始化设定项中不允许使用 new 关键字。所以第 14 行代码为类 A 的一个变量创建实例会造成错误。错误提示如下：

错误 1 “ClassType1.Program.myA” 的类型为 “ClassType1.A”。只能用 null 对引用类型(字符串除外)的常量字段进行初始化。

根据提示，对除 string 类型之外的其它引用类型，只能初始化为 null。

第 17 行代码定义了一个方法中的常量。同样，它也必须在做初始化操作。第 18 行打印了类 A 中的静态字段的值。但是第 19 行打印类 Program 中的常量字段 s 时，可以发现调用方法和第 18 行调用静态字段的方法是一样的。这是因为类中的常量成员默认是隐式静态的，它就是一个静态字段。但是在定义时不允许使用 static 关键字修饰它。

常量在初始化后就不能改变其值，代码实例如下：

```

01 using System;
02 namespace ClassType1
03 {
04     enum Color {Red, Yellow, Blue};
05     class A
06     {
07         public string s = "我是类 A 中的静态字段 s";
08     }
09     class Program
10     {
11         private const int a = 11;
12         private const Color c = Color.Red;
13         private const string s = "Hello World";
14         private const A myA = null;
15         static void Main(string[] args)
16         {
17             const int b = 12;
18             myA = new A();
19             c = Color.Yellow;
20             Console.WriteLine(myA.s);
21             Console.WriteLine(Program.s);
22             Console.ReadKey();
23         }
24     }

```

```
25 }
```

这段代码对前面的例子中的代码做了修改，首先将第 7 行的静态字段改成了实例字段，用户的意图是将第 14 行的引用类型变量初始化为 null，因为这是语法所允许的。然后在第 18 行对变量做初始化，创建一个类 A 的实例，然后在第 20 行，通过实例来调用这个实例字段。

同时第 19 行代码意图改变常量 c 的值为枚举 Color 的另一个成员 Yellow。但是这时编译器会报错如下：

```
错误 1 赋值号左边必须是变量、属性或索引器 E:\书稿\new\书稿源码\第八章
\ClassType1\Program.cs 18 13 ClassType1
```

```
错误 2 赋值号左边必须是变量、属性或索引器 E:\书稿\new\书稿源码\第八章
\ClassType1\Program.cs 19 13 ClassType1
```

可以看到，第 18 行代码和第 19 行代码同时出错，因为它们改变了常量的值。即使常量初始化为 null，再在代码中为其赋予实例也不行。

而对于 string 类型的常量来说，虽然它是引用类型，编译器允许它做初始化，但同样的，也不能通过调用 string 类型的构造方法来初始化 string 类型的常量。所以下面这样的初始化方法是允许的：

```
private const string s = "Hello World";
```

但是下面这样的初始化是不允许的：

```
private const string s = new string(new char[] { 'a', 'b' });
```

这行代码通过调用 string 类型的构造方法来初始化常量，但是因为使用 new 关键字创建实例是运行时发生的，所以即使是 string 类型，也不允许使用 new 关键字初始化常量。

一个程序集中的常量初始化操作可以互相依赖，不同程序集中的常量初始化操作可以单向依赖。下面看代码实例：

```
01 using System;
02 namespace ClassType1
03 {
04     class A
05     {
06         public const int b = Program.b + 2;
07         public const int c = 12, d = 13, e = 14;
08     }
09     class Program
10     {
11         public const int a = A.b + 1;
12         public const int b = 3;
13         public const int c = a + 10;
14         static void Main(string[] args)
15         {
16             Console.WriteLine(a);
17             Console.WriteLine(b);
18             Console.WriteLine(c);
19             Console.WriteLine(A.b);
20             Console.ReadKey();
21         }
22     }
23 }
```

在常量成员的定义形式上，它和 C# 的局部变量、成员字段都一样，可以在一行上定义多个成员。就像第 7 行代码所演示的那样，它和分行定义三个 public 的 int 类型的常量是一样的效果，只不过少敲了几个字母而已，简化了代码的输入。

在常量的初始化上，第 11 行代码依赖于第 6 行代码，而第 6 行代码又依赖于第 12 行代码。第 13 行代码又依赖于第 11 行代码。只要没有构成循环，这种语法是允许的。编译器会自动安排它们的初始化顺序。这段代码的执行结果如下：

```
6
3
16
5
```

8.5.1 类库

前面介绍过，不同程序集之间常量的初始化只能单向依赖。前面的例子中，同一程序集内部的双向依赖是允许的。下面再演示一下不同程序集之间的常量初始化依赖关系是否是单向的。在这个例子中，需要使用类库项目。什么是类库呢？类库对应扩展名为“.dll”的文件。dll 文件是 Dynamic Link Library 的缩写，它的意思是动态链接库。也就是说，它不是一种用来直接运行的应用程序，而是一种在需要时被加载的库文件。在这段代码中，它包含了一个类，这个类可以被调用。在调用这个类时，这个文件就被加载到了内存。.Net 基础类库也是以一个个这样的文件存在的。

以后在编写多层架构的应用程序时，不同的功能将被划分到不同的层，在实现上各个层就是以一个个的类库项目存在。下面首先来建立类库项目，在解决方案名称上单击右键，然后单击“添加”，再选择“新建项目”，如图 8-1 所示。

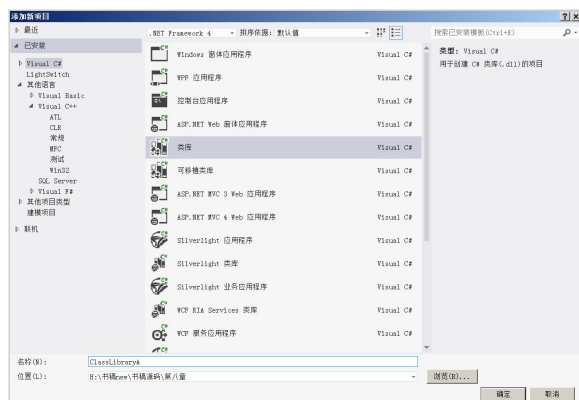


图 8-1 项目类型

在“添加新项目”窗口中选择“类库”，然后为项目起一个名字，最后点击“确定”按钮。这样，一个类库项目就建立好了。接下来需要在程序集中引用这个类库，方法是在项目的“引用”项中单击右键，然后选择“添加引用”，如图 8-2 所示。

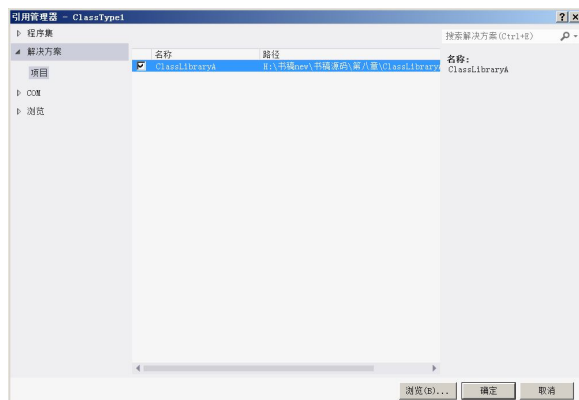


图 8-2 引用管理器

在“解决方案”一项中选择刚刚建立的类库，然后单击“确定”按钮。这时就可以看到刚刚建立的类库出

现在引用项中。如图 8-3 所示。

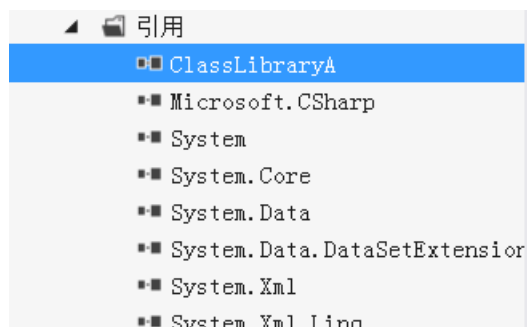


图 8-3 引用项

接下来在类库项目中输入下面的代码：

```
01 using System;
02 namespace ClassLibraryA
03 {
04     public class A
05     {
06         public const int b = 2;
07     }
08 }
```

这个类库只包含了一个类，类里有一个常量成员，并进行了初始化。然后在原来的项目中输入如下代码：

```
01 using System;
02 using ClassLibraryA;
03 namespace ClassType1
04 {
05     class Program
06     {
07         public const int a = A.b + 1;
08         public const int b = 3;
09         public const int c = a + 10;
10         static void Main(string[] args)
11         {
12             Console.WriteLine(a);
13             Console.WriteLine(b);
14             Console.WriteLine(c);
15             Console.WriteLine(A.b);
16             Console.ReadKey();
17         }
18     }
19 }
```

代码的第 2 行首先引用了类库的命名空间，注意这里的 using 语句不是物理上文件的引用，而是逻辑名称上的引用，它只是减少了代码的敲击次数。而刚刚做过的在项目上的引用才是物理文件上的引用。接下来代码定义了三个常量，常量 a 引用了类库中的常量，而常量 c 又引用了常量 a。从这里可以知道为什么不同程序集中的常量不能相互引用，而只能单向引用。因为程序集之间的物理引用只能是单向的。这段代码的执行结果如下：

3
3
13
2

8.6 静态方法和实例方法

使用 `static` 关键字修饰的方法就是静态方法，静态方法是类的一种成员方法。例如 `Main` 方法就是静态方法。静态方法只能直接访问静态成员，不能直接访问实例成员。不使用 `static` 关键字修饰的方法就是实例方法，实例方法能够访问静态成员和实例成员。下面是代码实例：

```
01 using System;
02 namespace ClassType1
03 {
04     class Program
05     {
06         private static int a = 11;
07         private int b = 12;
08         public static void Add()
09         {
10             a++;
11         }
12         public void Sub()
13         {
14             b--;
15             a--;
16         }
17         static void Main(string[] args)
18         {
19             Add();
20             Console.WriteLine(a);
21             Program p = new Program();
22             p.Sub();
23             Console.WriteLine(p.b);
24             Console.WriteLine(a);
25             Console.ReadKey();
26         }
27     }
28 }
```

这段代码的第 6 行定义了一个静态字段，第 7 行定义了一个实例字段。第 8 行定义了一个静态方法，它对静态字段 `a` 进行了自增操作。第 12 行定义了一个实例方法，它可以同时对静态字段和实例字段进行操作。

当在 `Main` 方法中调用方法成员的时候，因为 `Main` 方法是静态方法，所以它可以直接操作的只有类的静态成员。因此第 19 行它可以直接调用 `Add()` 方法，因为 `Add()` 方法是一个静态方法。第 20 行调用了类库中的静态方法 `WriteLine()` 方法直接打印静态字段 `a` 的值，因为是静态方法，所以可以直接访问类的静态字段。

当要调用类中的实例方法 `Sub()` 的时候，就不能直接调用了。静态成员要调用类中的实例成员的时候，只有一个办法，就是创建类的实例来访问。因此，第 21 行首先定义了一个类的实例。第 22 行通过类的实

例调用了类的实例方法 Sub()。当要打印类的实例字段 b 的时候，也只能通过类的实例来调用。

第 12 行代码是一个实例方法，它可以操作类的实例成员和静态成员。在方法体中对实例字段 b 进行了自减，对静态字段 a 也进行了自减。调用第 12 行的实例方法后，第 24 行重新打印了静态字段 a 的值。下面是代码的执行结果：

```
12
11
11
```

8.7 this 引用

为何静态方法不可以直接访问类的实例成员呢？这是因为静态方法没有 this 引用，而实例方法有 this 引用。所以可以直接访问类的实例成员。那么，什么是 this 引用呢？this 引用就是对所构造的实例的引用。下面用代码实例来说明：

```
01 using System;
02 namespace ClassType1
03 {
04     class Program
05     {
06         private int a = 12;
07         public void Sub()
08         {
09             this.a--;
10             this.Show();
11         }
12         public void Show()
13         {
14             Console.WriteLine(this.a);
15         }
16         static void Main(string[] args)
17         {
18             Program p = new Program();
19             p.Sub();
20             Console.ReadKey();
21         }
22     }
23 }
```

代码的第 6 行定义了一个实例字段，第 7 行定义了一个实例方法。可以看到，在实例方法的方法体中要想访问类的实例成员，不管这个成员是实例字段还是实例方法，都可以通过一个 this 关键字来引用。当在实例方法中调用类的实例成员时，当输入 this 关键字，然后输入 “.” 运算符后，开发环境都会给出智能提示，在菜单中列出了类的所有可调用的实例成员。可以这么认为，类的实例方法就是用 this 引用来达到了访问类的实例成员的目的，而静态方法因为没有 this 引用，所以不能直接访问类的实例成员，它只能通过类的实例名称来访问类的实例成员。

第九章 类的继承

9.1 静态构造方法

类的构造方法是类的成员方法的一种，它的作用是对类中的成员进行初始化操作。其中类的静态构造方法的作用是对类的静态成员进行初始化操作。实例代码如下：

```
01 using System;
02 namespace ClassType1
03 {
04     class Test
05     {
06         private static int a;
07         static Test()
08         {
09             a = 11;
10         }
11         public void Show()
12         {
13             Console.WriteLine("静态字段 a 的值是: {0}", a);
14         }
15     }
16     class Program
17     {
18         static void Main(string[] args)
19         {
20             Test t = new Test();
21             t.Show();
22             Console.ReadKey();
23         }
24     }
25 }
```

首先，这段代码定义了两个类。第 4 行代码定义了类 Test。定义类的时候，类的访问权限修饰符有两个，一个是 public，另一个是 internal。当不写任何访问修饰符的时候，类的访问权限默认是 internal。这个访问权限的意义是，这个类只能被本程序集访问，不能被本程序集以外的类访问。如果这个类是属于类库的，那么它必须是 public 的，否则调用它的程序集就不能访问它。

第 6 行代码定义了一个静态字段成员，第 7 行代码就是静态构造方法。可以看到，静态构造方法的特点是，以 static 关键字说明这个方法是静态的，方法名称要和类名完全相同，这里要注意大小写。静态构造方法不能含有参数，静态构造方法不能有返回值。在静态构造方法体内可以做初始化静态成员的操作。

第 11 行代码定义了一个实例方法，它的作用是输出静态字段的值。

在第 16 行的类 Program 中的 Main 方法中调用了这个类，第 20 行创建了这个类的一个实例，第 21 行调用了类的实例方法。从代码中可以看到，并没有显式调用类的静态构造方法。下面看代码的执行结果：

```
静态字段 a 的值是: 11
```

可以看到，静态构造方法确实是被执行了。那么本例就是静态构造方法的执行条件之一，在类的实例被创建时，类的静态构造方法将被自动调用。静态构造方法的调用次序是在静态字段的初始值设定项之后。也就是第一步是静态字段的默认值设置，第二步是执行静态字段的初始值设定项，第三步就是调用类的静

态构造方法。

下面将前面的代码实例修改一下，代码如下：

```
01 using System;
02 namespace ClassType1
03 {
04     class Test
05     {
06         private static int a;
07         static Test()
08         {
09             a++;
10         }
11         public void Show()
12         {
13             Console.WriteLine("静态字段 a 的值是: {0}", a);
14         }
15     }
16     class Program
17     {
18         static void Main(string[] args)
19         {
20             Test t = new Test();
21             t.Show();
22             Test t1 = new Test();
23             t.Show();
24             Console.ReadKey();
25         }
26     }
27 }
```

这段代码将静态构造方法做了修改，在方法体内将静态字段 a 进行自增操作。然后在代码的第 22 行又创建了一个类的实例，然后再次调用类的实例方法。下面看执行结果：

静态字段 a 的值是: 1

静态字段 a 的值是: 1

可以看到，静态字段的值并没有增加。这就是静态构造方法执行的特点，它只执行了一次。当程序集运行的时候，将会创建一个应用程序域，在一个应用程序域中，类的静态构造方法仅仅执行一次。下面再对代码实例进行修改如下：

```
01 using System;
02 namespace ClassType1
03 {
04     class Test
05     {
06         public static int a;
07         static Test()
08         {
```



```
09         Console.WriteLine("类的静态构造方法开始执行");
10         a++;
11     }
12     public void Show()
13     {
14         Console.WriteLine("静态字段 a 的值是: {0}", a);
15     }
16 }
17 class Program
18 {
19     static void Main(string[] args)
20     {
21         Console.WriteLine("静态字段 a 的值是: {0}", Test.a);
22         Console.WriteLine("静态字段 a 的值是: {0}", Test.a);
23         Console.ReadKey();
24     }
25 }
26 }
```

这段代码在类的静态构造方法中打印输出一行标记，类的静态字段的访问权限也修改为 public，这让它可以在类外被调用。在 Main 方法中两次打印输出了静态字段的值，注意在类外调用类的静态字段需要使用类名进行引用。下面是代码的执行结果：

类的静态构造方法开始执行

静态字段 a 的值是: 1

静态字段 a 的值是: 1

本段代码并没有创建类的实例。在引用类的静态成员之前，类的静态构造方法将被调用。这个被调用的类的静态成员包括静态字段和静态方法。这就是类的静态构造方法调用的第二个条件。

下面再对代码实例进行修改如下：

```
01 using System;
02 namespace ClassType1
03 {
04     class Program
05     {
06         private static int a;
07         static Program()
08         {
09             Console.WriteLine("类的静态构造方法被调用");
10             a = 11;
11         }
12         static void Main(string[] args)
13         {
14             Console.WriteLine("Main 方法被调用");
15             Console.WriteLine("静态字段 a 的值是: {0}", a);
16             Console.ReadKey();
17         }
18     }
19 }
```

```

18     }
19 }

```

这段代码在包含 Main 方法的类中定义了静态字段和静态构造方法。因为 Main 方法也是一个静态方法，而且它是类的入口点方法，那么它和类的静态构造方法之间是谁先调用呢？下面首先来看代码的执行结果：类的静态构造方法被调用

Main 方法被调用

静态字段 a 的值是：11

通过代码的执行结果可以看到，因为类的入口点方法仍然是一个静态方法，那么在任何静态成员被调用之前，静态构造方法都首先被调用。所以，可以得出如下结论，类的静态构造方法先于类的 Main 方法被调用。那么类的静态构造方法能否被显式调用呢？下面看代码实例：

```

01 using System;
02 namespace ClassType1
03 {
04     class Program
05     {
06         private static int a;
07         static Program()
08         {
09             Console.WriteLine("类的静态构造方法被调用");
10             a = 11;
11         }
12         static void Main(string[] args)
13         {
14             Program();
15             Console.ReadKey();
16         }
17     }
18 }

```

在这段代码中的第 14 行显式调用了类的静态构造方法，这时编译器会报错如下：

错误 1 “ClassType1.Program” 是“类型”，但此处被当做“变量”来使用 E:\书稿new\书稿源码\第八章\ClassType1\Program.cs 14 13 ClassType1

从错误报告中可以看出，类的静态构造方法是不能被显式调用的。

9.2 实例构造方法

类的实例构造方法是类的成员方法的一种，它的作用是对类的实例成员进行初始化操作。实例构造方法可以实现重载，在创建类的实例时，可以显式指定不同的参数来调用重载的不同的实例构造方法。下面看代码实例：

```

01 using System;
02 namespace ClassType1
03 {
04     class Program
05     {
06         private static int a;
07         private int b = 12;
08         private string c = "Hello World";

```

```
09     static Program()
10     {
11         Console.WriteLine("类的静态构造方法被调用");
12         a = 11;
13     }
14     public Program(int a, string s)
15     {
16         Console.WriteLine("带二个参数的构造方法被调用");
17         this.b = a;
18         this.c = s;
19     }
20     public Program(int a):this(a,"通过 this 关键字调用构造方法")
21     {
22         Console.WriteLine("带一个参数的构造方法被调用");
23     }
24     public void Show()
25     {
26         Console.WriteLine("静态字段 a 的值是: {0}", a);
27         Console.WriteLine("实例字段 b 的值是: {0}", b);
28         Console.WriteLine("实例字段 c 的值是: {0}", c);
29     }
30     static void Main(string[] args)
31     {
32         Program p1 = new Program(33, "这是创建的实例 P1");
33         Program p2 = new Program(34);
34         p1.Show();
35         p2.Show();
36         Console.ReadKey();
37     }
38 }
39 }
```

这段代码的第 6 行、第 7 行和第 8 行分别定义了三个字段成员，第 6 行是静态字段，第 7 行和第 8 行代码都有初始值设定项。

代码的第 14 行就是一个实例构造方法的定义，实例构造方法也是以类名作为方法名，它没有返回值，在方法名前面是访问权限修饰符，可以使用的访问权限修饰符包括 `public`、`private` 和 `protected`。其中的 `protected` 意味着构造方法只能在此类内部访问以及被从它继承的子类访问。实例构造方法可以带参数。第 14 行代码的实例构造方法使用两个传入的参数对实例字段进行了赋值。

第 20 行代码定义了带一个参数的实例构造方法，它和前一个实例构造方法形成了重载。实例构造方法可以通过 `this` 关键字调用其他的实例构造方法，方法就是在参数列表的后面使用冒号然后接 `this` 关键字，然后再跟参数列表，这个参数列表要匹配另一个重载的实例构造方法。第 20 行的构造方法只有一个参数，它将这个参数通过 `this` 关键字传递给了另一个构造方法，在用 `this` 调用另一个构造方法的时候，为其同时传入了一个字符串参数。

第 24 行的实例方法打印类的字段成员的值。

在 `Main` 方法中，第 32 行代码和第 33 行代码分别定义了两个实例，它们使用 `new` 关键字调用了不同的

实例构造方法。

第 34 行和第 35 行分别调用实例方法打印类的静态字段和实例的两个字段成员的值。下面先来看代码的执行结果：

```
类的静态构造方法被调用
带二个参数的构造方法被调用
带二个参数的构造方法被调用
带一个参数的构造方法被调用
静态字段 a 的值是：11
实例字段 b 的值是：33
实例字段 c 的值是：这是创建的实例 P1
静态字段 a 的值是：11
实例字段 b 的值是：34
实例字段 c 的值是：通过 this 关键字调用构造方法
```

现在用执行结果来介绍实例构造方法的执行过程，当第 32 行代码创建类的实例时，类的静态字段首先被设置成默认值，因为没有字段的初始值设定项，所以接着就执行类的静态构造方法。这时静态字段 a 被设置成 11。

因为第 32 行代码使用 new 调用了带有两个参数的实例构造方法，所以首先实例字段 b 被设置为 0，实例字段 c 被设置为 null。然后执行字段的初始值设定项，b 被赋值为 12，c 被赋值为“Hello World”。接下来执行实例构造方法体中的第一个语句，“带二个参数的构造方法被调用”这个字符串被打印。接下来实例 p1 的字段 b 被设置为传入的参数 33，注意构造方法的形参 a 在这里覆盖了类的静态字段 a。也就是说，这时起作用的是实例构造方法的局部变量 a。然后实例字段 c 被设置为字符串“这是创建的实例 P1”。

第 33 行代码使用 new 关键字调用了带一个参数的实例构造方法，在调用时，首先属于 p2 的实例字段 b 被设置为 0，实例字段 c 被设置为 null。然后执行字段的初始值设定项，b 被赋值为 12，c 被赋值为“Hello World”。接下来执行的是 this 引用的带两个参数的实例构造方法，“带二个参数的构造方法被调用”这个字符串被打印。然后 b 被设置为 34，c 被设置为“通过 this 关键字调用构造方法”。最后，代码控制又返回来执行带一个参数的实例构造方法体中的打印语句，“带一个参数的构造方法被调用”这个字符串被打印。至此，实例构造方法的执行完毕。

接下来的代码打印静态字段的值，可以看到两个实例打印出来的静态字段值是一样的，但是它们的实例字段的值各不相同。

可选参数和命名参数也可以用于实例构造方法，下面看代码实例：

```
01 using System;
02 namespace ClassType1
03 {
04     class Program
05     {
06         private int b;
07         private string c;
08         public Program(int a = 12, string s = "")
09         {
10             this.b = a;
11             this.c = s;
12         }
13         public void Show()
14         {
```

```

15         Console.WriteLine("实例字段 b 的值是: {0}", b);
16         Console.WriteLine("实例字段 c 的值是: {0}", c);
17     }
18     static void Main(string[] args)
19     {
20         Program p1 = new Program(); //构造方法的两个参数都采用默认值
21         Program p2 = new Program(34); //构造方法的 string 类型参数采用默认值
22         Program p3 = new Program(23, "Hello World"); //构造方法的两个参数采用传入参数
23         Program p4 = new Program(s: "今天的天气真好"); //采用命名参数, 另一个参数 a 采用默认值
24         p1.Show();
25         p2.Show();
26         p3.Show();
27         p4.Show();
28         Console.ReadKey();
29     }
30 }
31 }

```

代码的第 8 行定义了一个带有可选参数和命名参数的构造方法, 然后第 20 创建了一个类的实例, 在构造方法中没有传入任何参数, 这时, 构造方法的两个参数都采用默认值。

第 21 行代码为构造方法传入了一个 int 类型的参数, 这时, 另一个 string 类型的参数采用默认值。

第 22 行代码传入了两个参数, 构造方法的两个参数都使用了这两个传入的参数。

第 23 行代码使用了命名参数指定传入的参数是 string 类型的参数, 并将它传递给形参 s。这时另一个 int 类型的参数采用默认值。

第 24 行到第 27 行代码打印类的实例字段的值。这段代码的执行结果如下:

```

实例字段 b 的值是: 12
实例字段 c 的值是:
实例字段 b 的值是: 34
实例字段 c 的值是:
实例字段 b 的值是: 23
实例字段 c 的值是: Hello World
实例字段 b 的值是: 12
实例字段 c 的值是: 今天的天气真好

```

9.3 继承

一个类可以继承自另一个类。在 C# 中, 类与类之间只存在单一继承。也就是说, 一个类的直接基类只能有一个, 这和 C++ 语言不同。当类与类之间实现继承的时候, 子类可以将它的直接基类的所有成员当做自己的成员, 除了类的静态构造方法、实例构造方法和析构方法。但是, 虽然基类的所有成员都可以当做子类的成员, 但是如果基类的成员设置了不同的访问权限, 则派生类可以访问的成员也随之不同。C# 的继承是可以传递的, 如果类 C 从类 B 派生, 而类 B 从类 A 派生, 则类 C 将继承类 B 的所有成员, 也继承类 A 的所有成员 (各个基类的静态构造方法、实例构造方法和析构方法除外)。子类 (派生类) 可以在继承的基础上添加属于自己的成员, 但是它不能移除继承来的父类 (基类) 的成员。析构方法的作用是销毁类的实例, 后面会介绍到。下面看代码实例:

```

01 using System;
02 namespace ClassType

```

```
03 {
04     public class TV
05     {
06         private int channel = 1; //电视机的频道
07         private int volume = 20; //电视机的音量
08         public static string model = "39 英寸液晶"; //型号
09         /// <summary>
10         /// 具体设置电视机的频道和音量，因为只提供给子类使用
11         /// 所以用 protected 访问权限关键字修饰
12         /// </summary>
13         /// <param name="ch">具体设置的频道数</param>
14         /// <param name="vol">具体设置的音量值</param>
15         protected void Set(int ch, int vol)
16         {
17             channel = ch;
18             volume = vol;
19             Console.WriteLine("设置完毕");
20         }
21         /// <summary>
22         /// 增加频道
23         /// </summary>
24         public void ChPlus()
25         {
26             channel++;
27         }
28         /// <summary>
29         /// 增加音量
30         /// </summary>
31         public void VolPlus()
32         {
33             volume++;
34         }
35         /// <summary>
36         /// 将信息显示在电视机屏幕上
37         /// </summary>
38         public void Show()
39         {
40             Console.WriteLine("电视机型号是: {0}", model);
41             Console.WriteLine("频道: {0}", channel);
42             Console.WriteLine("音量: {0}", volume);
43         }
44     }
45     public class NewTV : TV
46     {
```

```

47         public void PlayUDisk()
48         {
49             this.Set(0, 30);
50             this.Show();
51             Console.WriteLine("现在开始播放 U 盘的视频文件.....");
52         }
53     }
54     class Program
55     {
56         static void Main(string[] args)
57         {
58             NewTV myNewTV = new NewTV();
59             myNewTV.ChPlus();
60             myNewTV.VolPlus();
61             myNewTV.Show();
62             myNewTV.PlayUDisk();
63             Console.ReadKey();
64         }
65     }
66 }

```

这段代码在前面模拟电视机的代码基础上又做了添加和修改。第 4 行代码定义了基类 TV。它的静态字段和实例字段都有一个初始值设定项进行了字段的初始化。

第 15 行代码添加了一个实例方法，它的访问修饰符为 protected。使用这个修饰符，只有本类的定义内部和它的派生类内部可以访问它。为什么要使用这个访问修饰符呢？因为这个方法不是给类的外部使用的。也就是说，它没有必要向用户公开。但是它的继承类又需要使用它，因此使用这个访问权限关键字可以保证一定程度的公开性，即定向公开，只为继承类开放。这个方法的作用是具体设置实例字段的值。让实例字段在模拟播放 U 盘的内容时，电视机的频道和音量能有一个特定值。除此之外，基类的其它方法没有改动。

第 45 行代码定义了一个子类，也就是派生类。它继承基类的语法就是在类名后加一个冒号，然后接一个基类的类名称。

第 47 行代码定义了一个方法，在这个方法中调用了基类的 Set 方法，并且为基类的方法传入了两个参数，这两个参数确定了在播放 U 盘的内容时，电视机的频道为 0，音量为 30。注意当调用 Set 方法的时候，使用了 this 关键字，它表示这个方法就是实例自己的，因为它是从基类继承下来的，相当于自己的财产。然后又调用了基类的 Show 方法来再一次显示频道和音量的设置值。因此，类 TV 和类 NewTV 之间的关系可以这么描述，类 TV 可以看做是一种电视机的原型机，类 NewTV 可以看做在这种原型机的基础上，电视机又进行了升级，它添加了 U 盘播放的功能，而其它功能可以直接从原型机继承，而不必再重新进行设计。

第 58 行代码定义了子类的实例，然后第 59 行、60 行和第 61 行直接调用了基类中定义的实例方法，因为这些方法都已经继承下来，完全属于子类自己。

第 62 行调用了子类定义的新添加的属于自己的方法。这段代码的执行结果如下：

```

电视机型号是：39 英寸液晶
频道：2
音量：21
设置完毕
电视机型号是：39 英寸液晶

```

频道: 0
音量: 30
现在开始播放 U 盘的视频文件.....

9.4 默认构造方法

在前面的例子中, 类中并没有定义构造方法。那么为什么能使用 new 关键字创建实例呢? 例如下面的语句:

```
NewTV myNewTV = new NewTV();
```

这是因为, 在类中没有定义任何构造方法的情况下, 编译器会为类自动生成一个默认无参数的构造方法。上面的语句就是调用的这个默认的构造方法。默认构造方法的功能是什么呢? 有人说是对类的成员进行初始化。其实默认构造方法是用来调用基类的无参构造方法。下面看代码实例:

```
01 using System;
02 namespace ClassType1
03 {
04     class Program
05     {
06         private int b;
07         private string c;
08         public void Show()
09         {
10             Console.WriteLine("实例字段 b 的值是: {0}", b);
11             Console.WriteLine("实例字段 c 的值是: {0}", c);
12         }
13         static void Main(string[] args)
14         {
15             Program p1 = new Program(); //构造方法的两个参数都采用默认值
16             p1.Show();
17             Console.ReadKey();
18         }
19     }
20 }
```

这段代码定义的类中没有定义任何构造方法, 那么它的字段的初始值是默认构造方法来设置的吗? 答案是否定的, 前面已经介绍过, 它的字段的设置依靠字段的默认值, 还有字段的初始值设定项。排行第三来执行的才是构造方法。还记得介绍过的 this 引用的方式来调用同一个类的其它构造方法吗? 这时 this 引用位于构造方法的形参列表后面, 实际上这个位置的 this 构造方法的引用有一个名字, 它叫做构造方法初始值设定项。在这个位置上除了 this 关键字外, 还有一个关键字叫做 base。它是用来引用父类的构造方法。因此, 上面的例子实际上等同于下面的代码实例:

```
01 using System;
02 namespace ClassType1
03 {
04     class Program
05     {
06         private int b;
07         private string c;
08         public Program() : base()
```



```

09      {
10      }
11      public void Show()
12      {
13          Console.WriteLine("实例字段 b 的值是: {0}", b);
14          Console.WriteLine("实例字段 c 的值是: {0}", c);
15      }
16      static void Main(string[] args)
17      {
18          Program p1 = new Program(); //构造方法的两个参数都采用默认值
19          p1.Show();
20          Console.ReadKey();
21      }
22  }
23 }

```

第八行代码的无参构造方法可以看做是编译器自动生成的，实际上编译器在自动生成类的默认构造方法的时候，也自动生成了一个构造方法初始值设定项，它的形式就是 `base()`。它的作用就是调用基类的无参构造方法。而在默认构造方法体中则什么操作也没做。因为这个类是直接继承自 `Object` 类，所以这里的 `base()` 调用的就是 `Object` 类的无参构造方法，查一下 Visual Studio 开发环境的帮助文档中的 `Object` 类，就可以看到 `Object` 类的无参构造方法的描述：

```
public Object()
```

此构造函数由派生类中的构造函数调用，但它也可以直接用于创建 `Object` 类的实例。

因此，除了 `Object` 类之外，所有类的默认构造方法都隐式地生成了这个 `base()` 形式的构造方法初始值设定项用来调用直接基类的无参构造方法。因为 `Object` 类是类结构的顶点，所以它除外。如果直接基类也没有定义任何构造方法，那么 `base()` 调用的就是编译器为直接基类生成的默认构造方法。

`base()` 初始值设定项不只可以调用基类的无参构造方法，它也可以用来传递参数给直接基类的构造方法。

构造方法还有一些使用限制，如果类中定义了构造方法，不管它有没有参数，编译器不会再为类自动生成默认的构造方法。下面看代码实例：

```

01 using System;
02 namespace ClassType1
03 {
04     class T
05     {
06         public T()
07         {
08             Console.WriteLine("类 T 中的无参构造方法被调用");
09         }
10     }
11     class Program:T
12     {
13         private int b;
14         private string c;
15         public Program(int a,string s)

```

```

16      {
17          b = a;
18          c = s;
19      }
20      public Program(int a):this(a,"Hello World")
21      {
22      }
23      public void Show()
24      {
25          Console.WriteLine("实例字段 b 的值是: {0}", b);
26          Console.WriteLine("实例字段 c 的值是: {0}", c);
27      }
28      static void Main(string[] args)
29      {
30          Program p1 = new Program();
31          Program p2 = new Program(33);
32          p2.Show();
33          Console.ReadKey();
34      }
35  }
36  }

```

这段代码的第 4 行首先定义了一个基类。在基类中，第 6 行定义了一个无参的构造方法。这样，编译器就不会为类再定义一个默认构造方法。也等于用这个用户自定义的无参构造方法覆盖了编译器提供的默认构造方法。

第 11 行定义类继承自 T。它包含两个构造方法，一个是带有两个参数的；另一个使用 this 构造方法初始值设定项调用了这个带有两个参数的构造方法。如果构造方法的初始值设定项使用了 this，那么它就不可以再使用 base 了。

第 30 行使用 new 关键字调用无参的构造方法，这时编译器会提示错误如下：

错误 1 “ClassType1.Program” 不包含采用“0”个参数的构造函数 H:\书稿 new\书稿源码\第八章\ClassType1\Program.cs 30 26 ClassType1

可以看到，只要用户自定义了构造方法，则编译器不再为类生成默认构造方法。下面将第 30 行代码去掉。而仅仅由第 31 行代码调用带有一个参数的构造方法来创建实例。并对代码修改如下：

```

01 using System;
02 namespace ClassType1
03 {
04     class T
05     {
06         public T()
07         {
08             Console.WriteLine("类 T 中的无参构造方法被调用");
09         }
10     }
11     class Program:T
12     {

```

```

13     private int b;
14     private string c;
15     public Program(int a,string s)
16     {
17         Console.WriteLine("带有二个参数的构造方法被调用");
18         b = a;
19         c = s;
20     }
21     public Program(int a):this(a,"Hello World")
22     {
23         Console.WriteLine("带有一个参数的构造方法被调用");
24     }
25     public void Show()
26     {
27         Console.WriteLine("实例字段 b 的值是: {0}", b);
28         Console.WriteLine("实例字段 c 的值是: {0}", c);
29     }
30     static void Main(string[] args)
31     {
32         Program p2 = new Program(33);
33         p2.Show();
34         Console.ReadKey();
35     }
36 }
37 }

```

在前面代码实例的基础上，对各个构造方法加入了一条打印语句用来标记调用顺序。然后仅仅由第 32 行代码调用带有一个参数的构造方法。下面来看代码的执行结果：

```

类 T 中的无参构造方法被调用
带有二个参数的构造方法被调用
带有一个参数的构造方法被调用
实例字段 b 的值是: 33
实例字段 c 的值是: Hello World

```

由代码的运行结果可以得出各个构造方法的调用顺序。首先第 32 行代码根据方法重载规则调用了带有一个参数的构造方法。接下来，带有一个参数的构造方法使用 this 构造方法初始值设定项调用了带有两个参数的构造方法。带有两个参数的构造方法根据编译器为其自动生成的 base() 构造方法初始值设定项调用了基类的无参构造方法。因此，基类构造方法体中的语句首先被打印，然后控制回到带有两个参数的构造方法体中，这时，用来标记的字符串被打印，然后字段被设置。接下来，控制回到带有一个参数的构造方法中，标记字符串被打印。最后，调用实例的实例方法 Show。

因此，可以得出下面的结论，在类的继承中，如果创建派生类的实例，那么，首先基类成员要被创建，基类的成员要被设置。然后才继续创建派生类成员，接着派生类成员被设置。从逻辑上说，如果要创建子类，那么就首先要创建父类。通俗地讲，没有父亲，就没有儿子。实际上，创建子类实例的时候，在内存中，继承来的成员是它的实例的一部分，这部分继承来的成员要先被创建，才能接着创建子类的成员。

9.5 使用 base 为基类传值

base 关键字不只能起到调用基类无参构造方法的作用，它也能通过参数的传递来调用基类重载的构造

方法。下面是代码实例：

```
01 using System;
02 namespace ClassType1
03 {
04     class T
05     {
06         protected int Ta;
07         public T()
08         {
09             Console.WriteLine("类 T 中的无参构造方法被调用");
10         }
11         public T(int a)
12         {
13             Ta = a;
14         }
15     }
16     class Program:T
17     {
18         private int a;
19         public Program()
20         {
21         }
22         public Program(int a,int b):base(b)
23         {
24             Console.WriteLine("带有二个参数的构造方法被调用");
25             this.a = a;
26         }
27         public void Show()
28         {
29             Console.WriteLine("实例字段 b 的值是：{0}", a);
30             Console.WriteLine("基类实例字段 Ta 的值是：{0}",Ta);
31         }
32         static void Main(string[] args)
33         {
34             Program p1 = new Program();
35             p1.Show();
36             Program p2 = new Program(33, 32);
37             p2.Show();
38             Console.ReadKey();
39         }
40     }
41 }
```

这段代码的基类定义中，首先第 6 行定义了访问权限为 `protected` 的实例字段。定义成这样的访问权限是为了派生类能直接打印其值。

接下来第 7 行代码和第 11 行代码分别定义了两个构造方法。一个无参，一个带有一个参数。在派生类中的第 19 行代码定义了一个无参的构造方法，第 22 行代码定义了一个带有两个参数的构造方法。这个构造方法使用 `base` 关键字将形参 `b` 传递给了基类，并调用基类的带有一个参数的构造方法。然后在方法体中对自身的实例字段 `a` 进行赋值。

第 34 行代码使用 `new` 关键字调用了类的无参构造方法，无参构造方法隐式使用 `base` 关键字调用了基类的无参构造方法。因为基类和派生类的无参构造方法都没有对字段进行重新赋值，所以它们的字段值都为 0。

第 36 行代码使用两个参数调用了类的带有两个参数的构造方法。而带有两个参数的构造方法又通过 `base` 引用进行了传值。所以基类的字段首先被设置成了 32。接下来设置本类的字段 `a`，它被设置为 33。这段代码的执行结果如下：

```
类 T 中的无参构造方法被调用
实例字段 b 的值是：0
基类实例字段 Ta 的值是：0
带有二个参数的构造方法被调用
实例字段 b 的值是：33
基类实例字段 Ta 的值是：32
```

9.6 私有构造方法

私有构造方法就是访问权限修饰符为 `private` 的实例构造方法。将访问权限修饰符设置为 `private` 有两个目的，一是它不能被继承；二是它不能实例化。所以，使用私有构造方法通常用来设计工具类，即这个类通过一些静态成员来完成某些操作，而不必要实例化这个类。下面是代码实例：

```
01 using System;
02 namespace ClassType2
03 {
04     class Test
05     {
06         private Test()
07         {
08         }
09         private const float PI = 3.14159265358979F;
10         public static float Area(float r)
11         {
12             return PI * r * r;
13         }
14     }
15     class Program
16     {
17         static void Main(string[] args)
18         {
19             Console.WriteLine("请输入圆的半径：");
20             float r = Convert.ToSingle(Console.ReadLine());
21             float area = Test.Area(r);
22             Console.WriteLine("圆的面积为：{0}", area);
23             Console.ReadKey();
24         }
25     }
26 }
```

```

25     }
26 }

```

第 6 行代码简单地定义了一个 `private` 的无参构造方法就将类定义成了一个无法实例化的类。

第 9 行定义了一个常量 `PI`，第 10 行定义了一个静态的方法 `Area` 来计算圆的面积，这个方法要求传入一个 `float` 类型的圆的半径。

第 20 行代码将用户输入的字符串转化成 `float` 类型的变量，第 21 行代码调用类 `Test` 的静态方法来求圆的面积，并将结果赋值给了变量 `area`。第 22 行打印输出圆的面积。

可以看到，求圆的面积没有必要实例化类，而只需要类的静态方法的功能就可以了。可以说，项目需要的就是一个工具类，在这种情况下，定义一个私有构造方法就可以解决问题。

那么，为何定义一个私有构造方法就可以无法实例化呢？这是因为当定义一个私有构造方法之后，编译器就不再为类提供默认的构造方法了。当在类外部使用 `new` 关键字创建实例的时候，因为类的构造方法为私有的，所以，在类外就没有权限访问它，因此就不能实例化类。当在类的继承过程中，因为子类的构造方法需要首先调用基类的构造方法，而基类的构造方法却是私有的，因此子类的构造方法对基类的构造方法就没有权限访问，所以子类在创建实例的时候就没有办法完成。下面在代码中添加如下的语句：

```
Test t = new Test();
```

这时编译器会报错，如下：

```

错误 1      “ClassType2.Test.Test()” 不可访问，因为它受保护级别限制      H:\书稿new\书稿源码\第
            八章\ClassType2\Program.cs  23    22    ClassType2

```

这个错误提示明确告诉用户，类的构造方法不可访问。如果将代码改写成如下形式：

```

01 using System;
02 namespace ClassType2
03 {
04     class Test
05     {
06         private Test()
07         {
08         }
09         private const float PI = 3.14159265358979F;
10         public static float Area(float r)
11         {
12             return PI * r * r;
13         }
14     }
15     class Program:Test
16     {
17         static void Main(string[] args)
18         {
19             Console.WriteLine("请输入圆的半径: ");
20             float r = Convert.ToSingle(Console.ReadLine());
21             float area = Test.Area(r);
22             Console.WriteLine("圆的面积为: {0}", area);
23             Program p = new Program();
24             Console.ReadKey();
25         }

```

```

26     }
27 }

```

在代码的第 15 行，让类 Program 继承自 Test。然后在代码的第 23 行创建一个子类的实例，然后编译代码，这时编译器会报错如下：

```

错误 1    “ClassType2.Test.Test()” 不可访问，因为它受保护级别限制    H:\书稿new\书稿源码\第
八章\ClassType2\Program.cs 15 11  ClassType2

```

可以看到，它和实例化类的时候的错误提示是一样的，都是因为构造方法无法访问造成的错误。

9.7 析构方法

当类的实例在堆中进行分配的时候，实例会在堆中依次存放。这就使实例很容易进行下一次的堆内存分配。当对一个实例没有有效的引用时，垃圾回收器会回收实例占用的堆内存空间。有了垃圾回收器，程序员就不用再为内存的分配和回收费心，这些操作都由垃圾回收器去完成。但是对于一些特殊的非托管资源，比如文件句柄、网络连接和数据库连接，垃圾回收器不知道如何释放它们。因此对于这些不能自动释放的非托管资源，需要制定特殊的处理方法来释放它们。在 C# 中，有两种方案可以处理非托管资源：一是在类中定义一个析构方法；二是在类中实现 System.IDisposable 接口。

析构方法是用来销毁类的实例的成员，析构方法不带参数，不具有访问权限修饰关键字，没有返回值，不能显式调用。在垃圾回收期间会自动调用所涉及实例的析构方法，也就是说，析构方法的执行时间是由垃圾回收器决定的。析构方法通常用来释放非托管资源，例如数据库的连接、文件句柄、网络连接等。这些连接，如果不指定专门的语句进行释放，垃圾回收器是不知道如何释放这些资源的。所以，析构方法是释放这些资源的其中一种方式，但不一定是最好的方式。下面通过一个代码实例来说明：

```

01 using System;
02 namespace 析构方法
03 {
04     class MyClass
05     {
06         private int a;
07         //析构方法的定义以波浪符号开始，不带返回值，无参数
08         //无访问权限修饰关键字
09         ~MyClass()
10         {
11             Console.WriteLine("执行析构方法");
12             Console.ReadKey();
13         }
14     }
15     class Program
16     {
17         static void Main(string[] args)
18         {
19             MyClass myClass1 = new MyClass();
20             MyClass myClass2 = new MyClass();
21             //此时，myClass1 原来所指向的实例不可用，应该回收
22             //但是并不一定马上进行回收
23             myClass1 = myClass2;
24             Console.ReadKey();
25         }

```

```

26     }
27 }

```

代码的第 9 行定义了一个析构方法，析构方法的定义方式是以波浪符号开头，方法名称和类的名称相同，无访问权限修饰关键字、无返回值、无参数。在方法体中写释放资源的代码。下面看执行结果：

执行析构方法

执行析构方法

从代码的执行来看，在将 myClass2 的引用赋值给 myClass1 之后，原来 myClass1 所引用的实例变成了没有任何引用指向它，它应该被回收，但是析构方法并没有调用，也就是说，垃圾回收器没有马上回收它。在代码即将执行完成时，调用了两次析构方法。所以，如果对于需要马上释放的资源，写在析构方法中是不合适的，因为析构方法的运行时间不确定。在 C# 中，程序运行完成时将会调用所有未被回收的实例的析构方法。那么既然无法直接调用类的析构方法，能不能强迫垃圾回收器执行呢？因为垃圾回收器执行的时候，会自动调用类的析构方法。实际上，在 .Net 类库中有一个 GC 类，它表示的就是垃圾回收器类。可以调用它的静态方法来进行一次垃圾回收。下面修改代码如下：

```

01 using System;
02 namespace ClassType2
03 {
04     class MyClass
05     {
06         private int a;
07         //析构方法的定义以波浪符号开始，不带返回值，无参数
08         //无访问权限修饰关键字
09         ~MyClass()
10         {
11             Console.WriteLine("执行析构方法");
12             Console.ReadKey();
13         }
14     }
15     class Program
16     {
17         static void Main(string[] args)
18         {
19             MyClass myClass1 = new MyClass();
20             MyClass myClass2 = new MyClass();
21             //此时，myClass1 原来所指向的实例不可用，应该回收
22             //但是并不一定马上进行回收
23             myClass1 = myClass2;
24             System.GC.Collect();
25             Console.ReadKey();
26         }
27     }
28 }

```

在代码的第 24 行调用了 System.GC.Collect() 方法，这个方法能强制进行一次垃圾回收，这时运行程序能马上看到析构方法的执行，代码的执行结果如下：

执行析构方法

有了这个方法，能不能就说可以控制析构方法的执行了呢？答案是否定的，使用这种方式仍无法确保非托管资源能够被立即释放掉。因为垃圾回收机制有着它自己的运行机制，它不能确保未引用的资源在一次垃圾回收期间都被释放掉。

在定义析构方法的时候还有一些规则，析构方法不能被继承，在一个类中只能有一个析构方法。因为析构方法没有任何参数，因此它不能被重载。当一个类中有析构方法时，垃圾回收期间会调用类的析构方法，当一个类中没有析构方法时，垃圾回收器会直接释放掉类的实例占用的堆空间。

9.7.1 析构方法的 IL 实现

实际上，在编译器对析构方法进行编译时，在 IL 代码中生成的并不是一个和 C#代码中的析构方法一样的中间语言代码，而是调用了另一个方法。下面用反编译器打开前面代码的程序集查看，反编译器使用开发环境自带的 ILDASM 工具，打开方式是在开发环境的程序菜单中打开“Visual Studio Tools”，然后选择“*** x86 本机工具命令提示”，星号代表 Visual Studio 开发环境的版本。然后在命令行窗中输入“ildasm”，如图 9-1 所示。

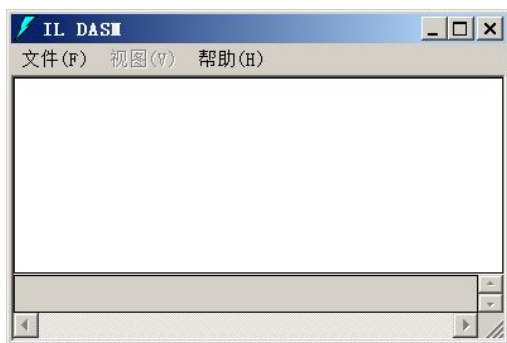


图 9-1 IL DASM

接下来单击文件菜单，然后打开编译好的程序集。如图 9-2 所示。

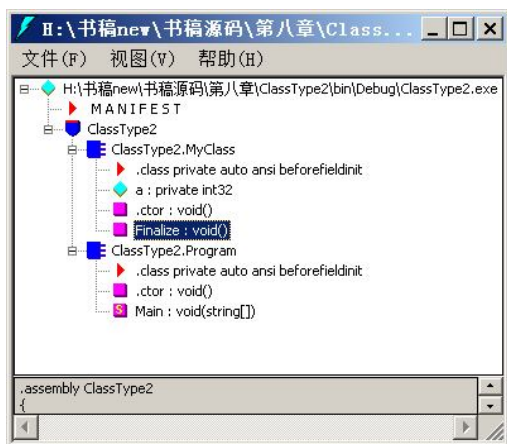


图 9-2 程序集 IL 代码

可以看到，在定义的 MyClass 类中有一个字段两个方法，字段就是代码中定义的字段 a，方法列表中的 .ctor 方法是类的默认构造方法，因为这个类中没有定义自定义的构造方法，所以这个方法是编译器自动添加的。双击打开这个方法的 IL 代码，如下所示：

```
.method public hidebysig specialname rtspecialname
    instance void .ctor() cil managed
{
    // 代码大小      7 (0x7)
    .maxstack 8
    IL_0000: ldarg.0
    IL_0001: call      instance void [mscorlib]System.Object::.ctor()
    IL_0006: ret
}
```

```
} // end of method MyClass::.ctor
```

从代码中可以看到，在 IL 代码中调用了基类 Object 的构造方法。接下来打开 Finalize 方法，代码如下：

```
.method family hidebysig virtual instance void
    Finalize() cil managed
{
    // 代码大小      31 (0x1f)
    .maxstack 1
    .try
    {
        IL_0000: nop
        IL_0001: ldstr      bytearray (67 62 4C 88 90 67 84 67 B9 65 D5 6C ) //
        gbL..g.g.e.l
        IL_0006: call      void [mscorlib]System.Console::WriteLine(string)
        IL_000b: nop
        IL_000c: call      valuetype [mscorlib]System.ConsoleKeyInfo
        [mscorlib]System.Console::ReadKey()
        IL_0011: pop
        IL_0012: nop
        IL_0013: leave.s    IL_001d
    } // end .try
    finally
    {
        IL_0015: ldarg.0
        IL_0016: call      instance void [mscorlib]System.Object::Finalize()
        IL_001b: nop
        IL_001c: endfinally
    } // end handler
    IL_001d: nop
    IL_001e: ret
} // end of method MyClass::Finalize
```

在 IL 中首先调用了 WriteLine 方法，然后是 ReadKey 方法。这两个方法是写在前面代码中的析构方法体中的，在 IL 中都被编译器写入了一个 try 块中，然后在 finally 块中调用了 Object 基类的 Finalize 方法。在 try 块执行完毕后，finally 块是必须执行的。Finalize 方法的作用是清理资源。所以可以看到，实质上，析构方法在背后其实是调用了基类的 Finalize 方法。

还有一个问题需要注意的是，当类中没有定义析构方法时，实例是被垃圾回收器一次性从堆中清除，而定义了析构方法后，就不一样了，垃圾回收器需要两次调用析构方法，第一次调用不清除对象实例，第二次调用才真正删除对象。.Net 环境使用一个专门的线程来执行 Finalize 方法，如果频繁使用析构方法，而且析构方法清理资源又需要比较长的时间，则对系统性能的影响会很显著。

9.7.2 析构方法的调用顺序

如果在类的继承链上的每个类都定义了析构方法，那么在销毁类的实例时，它们是按照一定的调用顺序进行依次调用的。下面看代码实例：

```
01 using System;
02 namespace ClassType2
03 {
```

```
04     class A
05     {
06         ~A()
07         {
08             Console.WriteLine("类 A 的析构方法被调用");
09         }
10     }
11     class B : A
12     {
13         ~B()
14         {
15             Console.WriteLine("类 B 的析构方法被调用");
16         }
17     }
18     class C : B
19     {
20     }
21     class D : C
22     {
23         ~D()
24         {
25             Console.WriteLine("类 D 的析构方法被调用");
26         }
27     }
28     class Program
29     {
30         static void Main(string[] args)
31         {
32             D myD = new D();
33             D myD1 = new D();
34             myD = myD1;
35             System.GC.Collect();
36             Console.ReadKey();
37         }
38     }
39 }
```

这段代码定义了 4 个类，A 类是第一个自定义的基类，B、C、D 类依次进行继承。但是类 C 内没有定义析构方法。

在第 32 行定义了一个类 D 的实例，第 33 行又定义了一个类 D 的实例。接下来，在第 34 行将 myD 也指向了第二个创建的实例。这样，第一个创建的实例就没有引用变量再指向它了。第 35 行强制进行了一次垃圾回收。下面首先来看执行结果：

```
类 D 的析构方法被调用
类 B 的析构方法被调用
类 A 的析构方法被调用
```

可以看到，析构方法的执行顺序和构造方法的执行顺序正好相反，离基类最远的派生类，或称作派生程度最大的派生类的析构方法最先被调用，然后向着基类的方向，各个类的析构方法被依次调用。

通过这段代码的执行还可观察到，当代码执行结束，按任意键后，按照既有的调用方向，析构方法会被再次调用一遍。通过这一点，就可以了解，析构方法的执行最好完全交给垃圾回收器去执行，释放非托管资源还是不要用这种不明确的方式。下面再看一下编译器生成的 IL 代码，如图 9-3 所示。

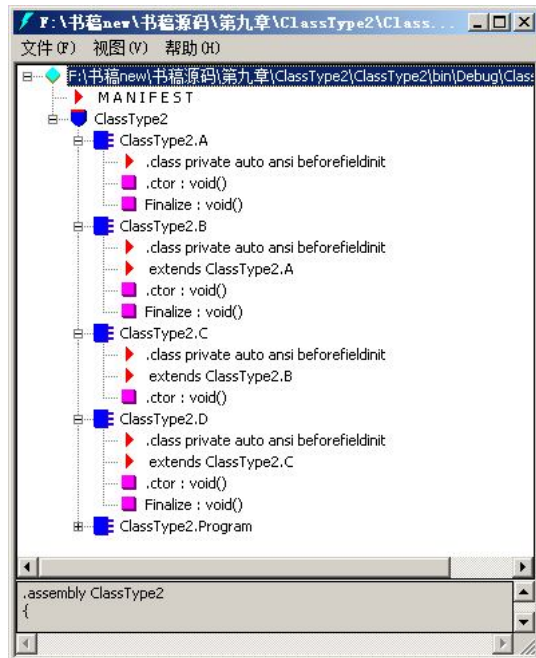


图 9-3 IL 代码

从编译器生成的 IL 代码可以看到，编译器为每个类都自动生成了 Finalize 方法。但是因为类 C 没有定义析构方法，所以它也没有生成 Finalize 方法，它直接就被垃圾回收器回收掉了。实际上，Finalize 方法是属于 Object 类的，它是一个虚方法，虚方法后面会介绍到。各个类只是自动由编译器重写了这个方法。但是在代码中不允许程序员调用这个方法，否则编译器会报错，代码实例如下：

```

01 using System;
02 namespace ClassType2
03 {
04     class A
05     {
06         ~A()
07         {
08             Console.WriteLine("类 A 的析构方法被调用");
09         }
10         public void Invoke()
11         {
12             Finalize();
13         }
14     }
15     class B : A
16     {
17         public void Finalize()
18         {
19         }

```

```

20     }
21     class Program
22     {
23         static void Main(string[] args)
24         {
25             Console.ReadKey();
26         }
27     }
28 }

```

这段代码的第 10 行通过一个方法调用了 `Object` 类的 `Finalize` 方法，第 17 行代码又重新定义了一个同名的 `Finalize` 方法，这个方法将覆盖继承来的 `Object` 类的 `Finalize` 方法。但是，这两种方式一种是不被允许的，另外一种却不被推荐，因为它将覆盖了 `Object` 的方法，却没起到资源回收的作用。它们将收到一个错误和一个警告，如下：

错误 1 无法直接调用析构函数和 `object.Finalize`。如果可用，请考虑调用 `IDisposable.Dispose`。

F:\书稿 new\书稿源码\第九章\ClassType2\ClassType2\Program.cs 12 13 ClassType2

警告 2 引入“`Finalize`”方法会妨碍析构函数调用。是否希望声明析构函数？ F:\书稿 new\书稿源码\第九章\ClassType2\ClassType2\Program.cs 17 21 ClassType2

9.8 密封类

密封类的作用就是阻止其它类从此类继承，一旦一个类被定义为密封类，则意味着，它不能再有子类。换句话说，其它类不能再从它派生了。使用密封类会接触到一个新的关键字 `sealed`。下面是代码实例：

```

01 using System;
02 namespace ClassType3
03 {
04     enum Color {Red, Yellow, Green, Blue, White}
05     class 交通指示灯
06     {
07         private string 型号;
08         private Color[] 颜色;
09         public 交通指示灯(string s, Color[] c)
10         {
11             型号 = s;
12             颜色 = c;
13         }
14         public void Show()
15         {
16             Console.WriteLine("交通灯的型号是：{0}", 型号);
17             Console.WriteLine("交通灯是{0}色交通灯", 颜色.Length);
18         }
19     }
20     sealed class 普通三色指示灯 : 交通指示灯
21     {
22         private int 车道数;
23         public 普通三色指示灯(string s, Color[] c, int num): base(s, c)
24         {

```

```

25         车道数 = num;
26     }
27     public void Show()
28     {
29         base.Show();
30         Console.WriteLine("交通灯适合 {0} 车道", 车道数);
31     }
32 }
33 class Program
34 {
35     static void Main(string[] args)
36     {
37
38         普通三色指示灯 p = new 普通三色指示灯("三车道三色交通灯", new Color[]
39         { Color.Red, Color.Yellow, Color.Green }, 3);
40         p.Show();
41         Console.ReadKey();
42     }
43 }

```

这段代码的第 5 行定义了一个基类，基类的名称用了汉字表示。因为 C# 是用 Unicode 编码，所以用中文定义标识符是允许的，这个例子是为了演示这一功能。但是在实际编程中最好不要这么做，因为考虑软件国际化的关系，要尽量使用通用语言进行编码，以利于协作。这是一个交通指示灯的类，它包含了两个成员，一个是型号，一个是表示包含颜色的 Color 类型的数组。Color 类型是一个枚举类型，在第 4 行定义。

第 9 行定义了一个构造方法，它为这两个字段进行赋值。第 14 行代码定义的实例方法打印两个实例字段的值。

第 20 行定义的类型继承自基类“交通指示灯”，这个类用关键字 sealed 进行修饰，表示这个类不能被继承。为什么要将它定义成密封类呢？因为从实际应用出发，普通三色指示灯将不再进行升级产生新产品，以后可能会去发展智能指示灯，这个普通三色指示灯就是产品线上的最终产品了，不再从这个产品升级研发新产品了。因此它不再需要创建子类。为了避免程序员错误地使用它创建派生类，所以把它定义成密封类。

第 38 行创建了类的一个实例，注意调用的构造方法的第二个参数，因为它需要的是一个 Color 类型的数组，这里可以直接采用 new 关键字为它传递参数，而不必先定义一个数组，然后再将引用传递给它。

第 39 行代码调用了派生类的 Show 方法，而这个 Show 方法在第 29 行使用 base 关键字调用了基类的 Show 方法，base 引用就是代表的基类。为何使用 base 呢，这是因为派生类的 Show 方法和基类的 Show 方法重名了，它将覆盖基类的 Show 方法，在派生类中只能访问派生类中的 Show 方法，基类的 Show 方法被隐藏了。这段代码的执行结果如下：

交通灯的型号是：三车道三色交通灯

交通灯是 3 色交通灯

交通灯适合 3 车道

虽然可以编译，但是编译器会提示一个警告信息如下：

警告 1 “ClassType3. 普通三色指示灯.Show()”隐藏了继承的成员“ClassType3. 交通指示灯.Show()”。如果是有意隐藏，请使用关键字 new。 F:\ 书 稿 new\ 书 稿 源 码 \ 第 九 章

\ClassType2\ClassType3\Program.cs 27 21 ClassType3

警告信息中提到的 new 关键字，后面会介绍到，它的作用是有意地隐藏基类的成员方法。如果让一个类从密封类继承，就会收到错误提示，代码实例如下：

```
01 using System;
02 namespace ClassType3
03 {
04     enum Color {Red, Yellow, Green, Blue, White}
05     class 交通指示灯
06     {
07         private string 型号;
08         private Color[] 颜色;
09         public 交通指示灯(string s, Color[] c)
10         {
11             型号 = s;
12             颜色 = c;
13         }
14         public void Show()
15         {
16             Console.WriteLine("交通灯的型号是：{0}", 型号);
17             Console.WriteLine("交通灯是{0}色交通灯", 颜色.Length);
18         }
19     }
20     sealed class 普通三色指示灯 : 交通指示灯
21     {
22         private int 车道数;
23         public 普通三色指示灯(string s, Color[] c, int num): base(s, c)
24         {
25             车道数 = num;
26         }
27         public void Show()
28         {
29             base.Show();
30             Console.WriteLine("交通灯适合{0}车道", 车道数);
31         }
32     }
33     class 特殊三色指示灯: 普通三色指示灯
34     {
35     }
36     class Program
37     {
38         static void Main(string[] args)
39         {
40             普通三色指示灯 p = new 普通三色指示灯("三车道三色交通灯", new Color[]
                { Color.Red, Color.Yellow, Color.Green }, 3);
```

```

41         p.Show();
42         Console.ReadKey();
43     }
44 }
45 }

```

这段代码的第 33 行从密封类创建了一个派生类，当编译这段代码时，就会收到如下错误提示：

```

错误 1    “ClassType3.特殊三色指示灯”：无法从密封类型“ClassType3.普通三色指示灯”派生 F:\
书稿 new\书稿源码\第九章\ClassType2\ClassType3\Program.cs    33    11    ClassType3

```

9.9 静态类

用 `static` 关键字修饰的类叫做静态类，静态类通常用来定义工具类。静态类不能实例化，不能从指定基类继承而来，静态类隐式从 `Object` 类继承而来。静态类只能包含静态成员和常量，因为常量是隐式静态的。静态类本身即具有密封的功能，因此，对于静态类不再能应用 `sealed` 关键字。因为静态类不能实例化，所以静态类中不能创建实例构造方法，编译器也不会自动为静态类生成默认实例构造方法。静态类中的成员必须包含 `static` 关键字，常量除外，因为常量是隐式静态的。下面是代码实例：

```

01 using System;
02 namespace ClassType3
03 {
04     static class ConsoleTools
05     {
06         /// <summary>
07         /// 静态字段，存储控制台窗口的默认前景颜色
08         /// </summary>
09         private static ConsoleColor defaultColor;
10         /// <summary>
11         /// 静态构造方法，用来将默认前景色赋值给静态字段
12         /// </summary>
13         static ConsoleTools()
14         {
15             defaultColor = Console.ForegroundColor;
16         }
17         /// <summary>
18         /// 将控制台前景色设置成红色
19         /// </summary>
20         public static void SetRed()
21         {
22             Console.ForegroundColor = ConsoleColor.Red;
23             Console.WriteLine("现在将控制台输出设置成红色字体：");
24         }
25         public static void SetGreen()
26         {
27             Console.ForegroundColor = ConsoleColor.Green;
28             Console.WriteLine("现在将控制台输出设置成绿色字体：");
29         }
30         public static void SetYellow()

```



```
31     {
32         Console.ForegroundColor = ConsoleColor.Yellow;
33         Console.WriteLine("现在将控制台输出设置成黄色字体: ");
34     }
35     /// <summary>
36     /// 将控制台前景色恢复成默认颜色
37     /// </summary>
38     public static void SetDefaultColor()
39     {
40         Console.ForegroundColor = defaultColor;
41         Console.WriteLine("现在将控制台输出设置成默认颜色字体: ");
42     }
43 }
44 class Program
45 {
46     static void Main(string[] args)
47     {
48         ConsoleTools.SetRed();
49         Console.WriteLine("白日依山尽");
50         ConsoleTools.SetYellow();
51         Console.WriteLine("黄河入海流");
52         ConsoleTools.SetGreen();
53         Console.WriteLine("欲穷千里目");
54         ConsoleTools.SetDefaultColor();
55         Console.WriteLine("更上一层楼");
56         Console.ReadKey();
57     }
58 }
59 }
```

这段代码的第 4 行定义了静态类, 注意静态类的定义要以 `static` 关键字修饰。第 9 行定义的 `ConsoleColor` 类型静态字段是用来存储控制台默认前景颜色的变量, `ConsoleColor` 是 .Net 类库中定义好的枚举类型, 它里面包含各种颜色。

第 13 行定义的静态构造方法用来在类初始化时, 将控制台的默认前景色赋值给静态字段。

第 20 行、第 25 行、第 30 行和第 38 行代码是使用 `ConsoleColor` 枚举对控制台前景色进行设置。控制台前景色的设置是对 `Console.ForegroundColor` 属性进行操作。属性的概念后面会介绍到。

第 48 行开始的代码就是首先设置控制台前景色, 然后打印一行文字。这段代码的执行结果如下:

现在将控制台输出设置成红色字体:

白日依山尽

现在将控制台输出设置成黄色字体:

黄河入海流

现在将控制台输出设置成绿色字体:

欲穷千里目

现在将控制台输出设置成默认颜色字体:

更上一层楼

下面对静态类进行继承操作，看一下编译器的提示，代码如下：

```
01 using System;
02 namespace ClassType3
03 {
04     class A { }
05     static class B : A { }
06     class C : B { }
07     class Program
08     {
09         static void Main(string[] args)
10         {
11             Console.ReadKey();
12         }
13     }
14 }
```

代码的第 5 行定义了一个静态类，将它定义成从类 A 派生，第 6 行定义了一个类，让它继承自静态类 B。这时编译器会提示两个错误，如下：

错误 1 静态类“ClassType3.B”不能从类型“ClassType3.A”派生。静态类必须从对象派生。 F:\书稿 new\书稿源码\第九章\ClassType2\ClassType3\Program.cs 5 22 ClassType3
 错误 2 “ClassType3.C”：无法从静态类“ClassType3.B”派生 F:\书稿 new\书稿源码\第九章\ClassType2\ClassType3\Program.cs 6 11 ClassType3

其中第一个错误提示的“静态类必须从对象派生”是中文翻译上的问题，其中的“对象”应该为 Object 类。下面再演示一下静态类的实例化操作，代码如下：

```
01 using System;
02 namespace ClassType3
03 {
04     static class A { }
05     class Program
06     {
07         static void Main(string[] args)
08         {
09             A myA = new A();
10             Console.ReadKey();
11         }
12     }
13 }
```

代码的第 4 行定义了一个静态类，然后第 9 行代码创建了静态类的实例，这时编译器会提示两个错误。如下：

错误 1 无法声明静态类型“ClassType3.A”的变量 F:\书稿 new\书稿源码\第九章\ClassType2\ClassType3\Program.cs 9 13 ClassType3
 错误 2 无法创建静态类“ClassType3.A”的实例 F:\书稿 new\书稿源码\第九章\ClassType2\ClassType3\Program.cs 9 21 ClassType3

创建静态类的实例将收到两个错误，一个是定义静态类的变量会提示错误，另一个是创建静态类的实例将收到错误。

9.10 分部类

在类的定义中，可以将类的定义分成多个部分。而不必在一个类体中将类的成员都定义完。在代码最后进行编译的时候，多个部分的分部类定义将被看做一个整体进行编译。定义分部类要使用上下文关键字 `partial`，它在定义分部类型的时候看做关键字，其它时候它可以用作标识符。但是建议将它看做关键字对待。在编译的时候，分部类型的所有组成部分一起进行编译，编译后，它们将合成一个整体。对于已经定义好的类型，不允许使用分部类型进行扩展。可以使用 `partial` 修饰符定义的分部类型只包括类类型、结构和接口。下面看代码实例：

```
01 //Part1.cs
02 using System;
03 namespace ClassType3
04 {
05     public partial class A
06     {
07         private string s1 = "这是 Part1.cs 中的字段 s1";
08     }
09 }
```

这是 Part1.cs 中的代码，然后在项目中再添加一个代码文件 Part2.cs，它的代码如下：

```
01 //Part2.cs
02 using System;
03 namespace ClassType3
04 {
05     public partial class A
06     {
07         private string s2;
08         public A()
09         {
10             s2 = "这是 Part2.cs 中的字段 s2";
11         }
12         public void Show()
13         {
14             Console.WriteLine(s1);
15             Console.WriteLine(s2);
16         }
17     }
18     class Program
19     {
20         static void Main(string[] args)
21         {
22             A myA = new A();
23             myA.Show();
24             Console.ReadKey();
25         }
26     }
27 }
```

首先来看 Part1.cs 中的代码实例，第 4 行定义了一个类，这个类的访问权限是 public。它用 partial 进行修饰，说明它是一个分部类。在它的内部定义了一个 string 类型的实例字段。

在 Part2.cs 中也定义了一个同样的类，这个类的名字和 Part1.cs 中的相同。并且它也用 partial 修饰。说明它是 Part1.cs 中的同名类的一部分。在定义分部类的时候，各个类的名字必须相同，并且都需要使用 partial 关键字。每个类的访问修饰符也必须定义成相同的。不能一个定义成 public，而另一个定义成 internal 的。但是有一点是需要特别声明一下的，如果一个类使用 public 关键字指定了访问权限，而类的另一部分不写 public，这时，这个分部类的访问权限并不是由默认指定的权限 internal，它也被看做是 public 的，因为类的另一部分显式指定了它。它并不造成访问权限的冲突。只有对一个分部类指定 public，而显式地指定类的另一部分为 internal，也就是说对类的另一部分使用 internal 关键字，这时才会造成冲突。Part2.cs 中的分部类又定义了一个字段，然后定义了一个实例构造方法，它对字段 s2 进行初始化赋值。然后第 12 行定义了一个方法，它打印各个字段的值。注意，这个方法虽然定义在 Part2.cs 中，但是它也能打印 Part1.cs 中的字段的值。

第 22 行代码定义了分部类的实例，第 23 行代码通过分部类的实例调用了它的实例方法 Show。最后在编译的时候，这两个文件将被合成一个文件，这两个分部类将被组合在一起，当做一个类的定义来对待。下面是执行结果：

这是 Part1.cs 中的字段 s1

这是 Part2.cs 中的字段 s2

可以看到，每个文件中的字段都被正确打印了。这个实例演示的是位于不同文件中的分部类，当分部类位于同一个文件中，效果也是相同的。分部类的最大好处就是可以将代码中的不同作用的代码分门别类地存放在不同的文件中，让程序员可以专注于特定代码的编写。例如 C# 中的窗体类程序就将窗体的定义和创建放在分部类中，这部分代码是由编译器进行生成和管理的，程序员可以不需要再花费更多的精力来编写这部分代码。这样就大大提升了工作效率。

密封类也可以是分部定义的，下面看代码实例：

```
01 //Part1.cs
02 using System;
03 namespace ClassType3
04 {
05     public sealed partial class A
06     {
07         private string s1 = "这是 Part1.cs 中的字段 s1";
08     }
09 }
```

这是 Part1.cs 中的代码，可以看到，在类的定义中使用了 sealed 关键字进行了类的修饰。在定义分部类的时候，需要注意的是，partial 修饰符必须和 class 关键字挨在一起，否则编译器会报下面的错误：

错误 2 “partial” 修饰符只能出现在紧靠 “class”、“struct”、“interface” 或 “void” 前面的位置 H:\书稿 new\书稿源码\第九章\ClassType2\ClassType3\Part1.cs 5 14 ClassType3

结构和接口后面会接触到，必须挨着 void 是指的分部方法的定义，这个后面也会介绍到，这里了解即可。

下面来看 Part2.cs 中的代码：

```
01 //Part2.cs
02 using System;
03 namespace ClassType3
04 {
05     public partial class A
06     {
```

```

07     private string s2;
08     public A()
09     {
10         s2 = "这是 Part2.cs 中的字段 s2";
11     }
12     public void Show()
13     {
14         Console.WriteLine(s1);
15         Console.WriteLine(s2);
16     }
17 }
18 public class B : A { }
19 class Program
20 {
21     static void Main(string[] args)
22     {
23         A myA = new A();
24         myA.Show();
25         Console.ReadKey();
26     }
27 }
28 }

```

可以看到，第 5 行代码并没有 `sealed` 关键字，但是它不影响密封类的定义。在分部类中，`sealed` 关键字只需要定义在类的一部分即可。当然，类的每个组成部分都定义这个关键字也可以。只不过多敲几个代码罢了。

第 18 行定义了一个类从这个密封类继承，在编译的时候，将会收到错误提示如下：

```

错误 1    "ClassType3.B" : 无法从密封类型 "ClassType3.A" 派生    H:\书稿 new\书稿源码\第九章
\ClassType2\ClassType3\Part2.cs 18 18  ClassType3

```

因此可以看到，密封类的定义是有效的。

下面再来看分部类在继承中的情况，代码实例如下：

```

01 //Part1.cs
02 using System;
03 namespace ClassType3
04 {
05     public class A { }
06     public partial class B:A
07     {
08         private string s1 = "这是 Part1.cs 中的字段 s1";
09     }
10 }

```

part1 文件中的第 5 行定义了一个基类，第 6 行指定分部类从它派生。Part2 文件中的代码如下：

```

01 //Part2.cs
02 using System;
03 namespace ClassType3

```

```
04 {
05     partial class B
06     {
07         private string s2;
08         public B()
09         {
10             s2 = "这是 Part2.cs 中的字段 s2";
11         }
12         public void Show()
13         {
14             Console.WriteLine(s1);
15             Console.WriteLine(s2);
16         }
17     }
18     class Program
19     {
20         static void Main(string[] args)
21         {
22             B myB = new B();
23             myB.Show();
24             Console.ReadKey();
25         }
26     }
27 }
```

当分部类中有一部分指定了基类，另一部分也被视同从同一个类派生。但是，分部类的各个部分不能指定不同的基类。下面再添加一个类，代码如下：

```
01 //Part1.cs
02 using System;
03 namespace ClassType3
04 {
05     public class A { }
06     public class C { }
07     public partial class B:A
08     {
09         private string s1 = "这是 Part1.cs 中的字段 s1";
10     }
11 }
```

然后修改 Part2.cs 文件中的代码如下：

```
01 //Part2.cs
02 using System;
03 namespace ClassType3
04 {
05     partial class B:C
06     {
```

```

07     private string s2;
08     public B()
09     {
10         s2 = "这是 Part2.cs 中的字段 s2";
11     }
12     public void Show()
13     {
14         Console.WriteLine(s1);
15         Console.WriteLine(s2);
16     }
17 }
18 class Program
19 {
20     static void Main(string[] args)
21     {
22         B myB = new B();
23         myB.Show();
24         Console.ReadKey();
25     }
26 }
27 }

```

在代码的第 5 行，让类继承自另一个基类 C。这样分部的两个部分就继承自不同的基类，这是不允许的。在编译的时候，编译器会报错如下：

错误 1 “ClassType3.B”的分部声明一定不能指定不同的基类 H:\书稿 new\书稿源码\第九章\ClassType2\ClassType3\Part1.cs 7 26 ClassType3

如果定义的分部类不包含基类的定义，则它仍然是默认继承自类 Object。

分部类在定义的时候必须位于同一个命名空间中，但是每个分部类的文件中的 using 指令，即命名空间名称的引入却是独立的。看下面的代码：

```

01 //Part1.cs
02 using MySystem = System;
03 namespace ClassType3
04 {
05     public partial class A
06     {
07         private string s1 = "这是 Part1.cs 中的字段 s1";
08         public void Show1()
09         {
10             MySystem.Console.WriteLine(s1);
11         }
12     }
13 }

```

在文件 Part1.cs 中，第 2 行定义了一个 System 命名空间的别名。命名空间别名后面会介绍到，这里了解即可。第 10 行使用别名调用类库中的方法，当在开发环境中输入别名的时候，会有智能提示给出 System 中的各个类。下面再来看 Part2.cs 中的代码：

```
01 //Part2.cs
02 using System;
03 using MySystem = System.Drawing;
04 namespace ClassType3
05 {
06     partial class A
07     {
08         private string s2;
09         private MySystem.Color c;
10         public A()
11         {
12             s2 = "这是 Part2.cs 中的字段 s2";
13         }
14         public void Show()
15         {
16             Console.WriteLine(s1);
17             Console.WriteLine(s2);
18         }
19     }
20     class Program
21     {
22         static void Main(string[] args)
23         {
24             A myA = new A();
25             myA.Show();
26             Console.ReadKey();
27         }
28     }
29 }
```

这段代码中的第 3 行也定义了一个命名空间的别名，这个别名和上一个文件中的别名是完全相同的。只不过它们引用的命名空间是不同的。注意导入 System.Drawing 命名空间的时候，需要物理上引入这个程序集文件。默认情况下，开发环境没有物理上引入这个程序集，当在代码中使用命名空间中的类的时候，将会提示找不到这个类。在第 9 行使用别名定义字段的时候。可以观察到开发环境的智能提示提示了不同的命名空间中的类。因此得出结论，不同文件中的分部类的命名空间引入是相互独立的。那么相同的文件中，情况又是怎样的呢？下面是代码实例：

```
01 //Part1.cs
02 using MySystem = System;
03 namespace ClassType3
04 {
05     public partial class A
06     {
07         private string s1 = "这是 Part1.cs 中的字段 s1";
08         public void Show1()
09         {
```



```

10         MySystem.Console.WriteLine(s1);
11     }
12 }
13 }
14 namespace ClassType3
15 {
16     using MySystem = System.Collections;
17     public partial class A
18     {
19         private MySystem.ArrayList myArray;
20     }
21 }

```

这是在一个文件中定义了同一个命名空间，在编译时，这两个命名空间将被合并成一个命名空间，因为它们同名。因为是在不同的命名空间中定义分部类，因此可以使用同一个别名绑定不同的命名空间。第 10 行代码中的 MySystem 和第 16 行代码中的 MySystem 别名虽然名称相同，但是允许它们绑定不同的类库中的命名空间。

9.10.1 分部方法

分部方法就和 C++ 中的形式一样，分为方法的声明和定义两个部分。分部方法要和分部类联合使用，分部方法在分部类的一部分声明，在分部类的另一部分定义。允许对分部方法只声明不定义，如果只有分部方法的声明，而没有任何定义，则编译器将在编译过程中把方法的声明和任何部分对它的调用都删除掉。下面是代码实例：

```

01 //Part1.cs
02 using System;
03 namespace ClassType3
04 {
05     public partial class A
06     {
07         private string s1 = "这是 Part1.cs 中的字段 s1";
08         partial void Show1();
09     }
10 }

```

代码的第 8 行定义了一个分部方法，注意分部方法的定义需要 partial 上下文关键字，它必须和 void 关键字挨着才有意义。也就是说，分部方法的返回类型必须是 void 的。分部方法不能有任何访问修饰符，它隐式是 private 的。第 8 行是一个方法的声明，它在形参列表后面用分号结尾，没有方法体的实现。在分部类的另一部分中进行方法的定义，看代码实例：

```

01 //Part2.cs
02 using System;
03 namespace ClassType3
04 {
05     partial class A
06     {
07         private string s2;
08         public A()
09         {

```

```
10         s2 = "这是 Part2.cs 中的字段 s2";
11     }
12     partial void Show1()
13     {
14         Console.WriteLine(s1);
15         Console.WriteLine(s2);
16     }
17     public void Show()
18     {
19         Show1();
20     }
21 }
22 class Program
23 {
24     static void Main(string[] args)
25     {
26         A myA = new A();
27         myA.Show();
28         Console.ReadKey();
29     }
30 }
31 }
```

第 12 行代码是分部方法的实现部分，实现部分和普通的方法定义是一样的。但是分部方法必须以 `partial` 上下文关键字和 `void` 联用，同样的，不能有任何访问修饰符。因为默认是 `private` 的，所以需要定义一个 `public` 的方法来调用它，以便在类外通过实例来调用。第 17 行代码定义了一个普通的实例方法来调用分部方法。

分部方法的声明和定义可以同时存在于分部类的同一部分，并且分部方法不能有 `out` 输出形参。下面看代码实例：

```
01 using System;
02 namespace ClassType3
03 {
04     public partial class A
05     {
06         private string s1 = "这是分部类第一部分中的字段";
07         partial void Show1();
08     }
09     partial class A
10     {
11         private string s2;
12         private int a;
13         public A()
14         {
15             s2 = "这是 Part2.cs 中的字段 s2";
16         }
17     }
18 }
```

```

17     partial void Set(int a, ref string s);
18     partial void Set(int a, ref string s)
19     {
20         s2 = s;
21         this.a = a;
22     }
23     partial void Show1()
24     {
25         string s = "这是分部类第二部分中的字段";
26         Set(11, ref s);
27         Console.WriteLine(s1);
28         Console.WriteLine(s2);
29     }
30     public void Show()
31     {
32         Show1();
33     }
34 }
35 class Program
36 {
37     static void Main(string[] args)
38     {
39         A myA = new A();
40         myA.Show();
41         Console.ReadKey();
42     }
43 }
44 }

```

这段代码是在一个源码文件中定义了分部类的两个部分。代码的第 17 行和第 18 行分别是分部方法的声明和定义。它们可以存在于分部类的一部分中。在分部方法中可以使用 ref 修饰符，在第 26 行调用的时候，因为向方法中传递 ref 参数时，必须使用变量的形式传入，所以第 25 行先定义了一个 string 类型的变量，并做了初始化。第 23 行调用它的方法也是一个分部方法，但是它位于分部类的两个部分中，声明和定义是分开的。下面是代码的执行结果：

这是分部类第一部分中的字段

这是分部类第二部分中的字段

如果将分部方法的 ref 修饰符改为 out，下面是代码实例：

```

01 using System;
02 namespace ClassType3
03 {
04     public partial class A
05     {
06         private string s1 = "这是分部类第一部分中的字段";
07         partial void Show1();
08     }

```

```
09     partial class A
10     {
11         private string s2;
12         private int a;
13         public A()
14         {
15             s2 = "这是 Part2.cs 中的字段 s2";
16         }
17         partial void Set(int a, out string s);
18         partial void Set(int a, out string s)
19         {
20             s = "这是分部类第二部分中的字段";
21             s2 = s;
22             this.a = a;
23         }
24         partial void Show1()
25         {
26             string s;
27             Set(11, out s);
28             Console.WriteLine(s1);
29             Console.WriteLine(s2);
30         }
31         public void Show()
32         {
33             Show1();
34         }
35     }
36     class Program
37     {
38         static void Main(string[] args)
39         {
40             A myA = new A();
41             myA.Show();
42             Console.ReadKey();
43         }
44     }
45 }
```

第 17 行和第 18 行代码将 ref 关键字修改为 out，在第 20 行为 s 做赋值操作。第 26 行在调用分部方法之前先定义了一个 string 类型的变量，然后将它以输出参数的方法传递给分部方法。第 27 行调用带有输出参数的分部方法。这时在编译阶段就会报错，信息如下：

```
错误 1    分部方法不能有 out 参数 H:\书稿 new\书稿源码\第九章\ClassType2\ClassType3\Part2.cs
          17 22 ClassType3
错误 2    分部方法不能有 out 参数 H:\书稿 new\书稿源码\第九章\ClassType2\ClassType3\Part2.cs
          18 22 ClassType3
```

从出错信息可以知道，分部方法的声明和定义部分都不能出现输出形参和输出参数。

第十章 类的成员

一个类的成员的来源共分为两类，一类是在类体中定义的成员；另一类是从基类继承来的成员。下面就对一个类的组成成员进行一一地介绍。

10.1 只读字段

字段就是一个类的变量，前面已经接触过字段。这一小节继续对字段作深入的探讨。只读字段就是在字段定义时使用 `readonly` 修饰符修饰它。对于只读字段，只能在字段的初始值设定项中改变它的值，或者在同一个类的构造方法中改变它的值。下面看代码实例：

```
01 using System;
02 namespace MemberOfClass
03 {
04     class Program
05     {
06         private readonly int a = 33;
07         public Program(int a)
08         {
09             this.a = a;
10         }
11         public void Show()
12         {
13             Console.WriteLine("字段 a 的值是: {0}", a);
14         }
15         static void Main(string[] args)
16         {
17             Program p = new Program(34);
18             p.Show();
19             Console.ReadKey();
20         }
21     }
22 }
```

代码的第 6 行定义的就是一个只读字段，代码的第 7 行是类的实例构造方法。那么对于静态字段 `a` 来说，能改变其值的只有实例字段的初始值设定项和类的实例构造方法。因此，这段代码中字段 `a` 的赋值顺序是，首先被设置为默认值 0，然后由初始值设定项设置为 33，最后由实例构造方法传入的参数确定其最终值。一旦最终值确定后，将再也不能改变。这段代码的执行结果如下：

字段 a 的值是: 34

下面将代码进行修改如下：

```
01 using System;
02 namespace MemberOfClass
03 {
04     class Program
05     {
06         private readonly int a = 33;
07         public Program(int a)
08         {
```

```
09         this.a = a;
10     }
11     public void Show()
12     {
13         Console.WriteLine("字段 a 的值是: {0}", a);
14     }
15     public void Set(int a)
16     {
17         this.a = a;
18     }
19     static void Main(string[] args)
20     {
21         Program p = new Program(34);
22         p.Show();
23         Console.ReadKey();
24     }
25 }
26 }
```

在前面代码的基础上，第 15 行代码增加了一个方法 Set，它使用传入的参数对字段 a 进行了赋值操作。当将这段代码进行编译时，编译器就会提示如下错误：

错误 1 无法对只读的字段赋值(构造函数或变量初始值指定项中除外) H:\书稿 new\书稿源码\第十章\MemberOfClass\Program.cs 17 13 MemberOfClass

只读字段不仅仅只有值类型这么简单，引用类型也可以做只读字段，它的值在运行时确定，然后再不可改变。实例代码如下：

```
01 using System;
02 namespace MemberOfClass
03 {
04     class A { }
05     class Program
06     {
07         private readonly A a = new A();
08         public Program(A a)
09         {
10             this.a = a;
11         }
12         public void Show()
13         {
14             Console.WriteLine("字段 a 的值是: {0}", a);
15         }
16         static void Main(string[] args)
17         {
18             A a = new A();
19             Program p = new Program(a);
20             p.Show();
21         }
22     }
23 }
```

```
21         Console.ReadKey();
22     }
23 }
24 }
```

第 4 行代码首先定义了一个类，第 7 行代码定义了一个 A 类型的只读字段 a，然后使用初始值设定项为这个字段创建了一个实例。第 8 行的构造方法使用传入的一个引用为这个字段重新赋值。注意，因为 A 类型是引用类型，所以字段的初始值设定项和构造方法都是运行时为只读字段赋值。这是允许的。构造方法赋值后，再也不能改变只读字段的值。这段代码的执行结果如下：

字段 a 的值是：MemberOfClass.A

因为调用的是 Object 基类的 ToString 方法，所以它打印的是 A 类的类名。

10.1.1 静态只读字段

前面的章节介绍过常量字段，常量字段要求在定义字段的时候，编译时必须能明确其值。其后其值将不能改变。如果需要一个类似常量，但其值在编译时无法确定的字段，则可以使用静态只读字段。代码实例如下：

```
01 using System;
02 namespace MemberOfClass
03 {
04     class Color
05     {
06         private byte R;
07         private byte G;
08         private byte B;
09         public Color(byte r, byte g, byte b)
10         {
11             this.R = r;
12             this.G = g;
13             this.B = b;
14         }
15         public void Show()
16         {
17             Console.WriteLine("此颜色的三基色值是：{0}, {1}, {2}", R, G, B);
18         }
19     }
20     class Program
21     {
22         private static readonly Color c1 = new Color(3, 2, 3);
23         private static readonly Color c2 = new Color(123, 100, 34);
24         static Program()
25         {
26             c1 = new Color(12, 12, 12);
27             c2 = new Color(255, 255, 255);
28         }
29         static void Main(string[] args)
30         {
```



```

31         Program.c1.Show();
32         Program.c2.Show();
33         Console.ReadKey();
34     }
35 }
36 }

```

代码的第 4 行定义了一个类，它表示颜色的三基色。在第 22 行代码和第 23 行代码定义了两个静态只读字段，并且它们都进行了实例化。如果是常量，那么在编译阶段是无法计算其值的。但是静态只读字段就可以在运行时确定其值。第 24 行代码是静态构造方法，它可以对静态只读字段重新赋值。它赋值后，静态只读字段的值就不可再改变了。

通过第 31 行代码和第 32 行代码可以看到，使用静态只读字段和使用常量的方式是一样的。静态只读字段弥补了常量在编译时必须确定其值的缺陷，而又起到了常量的效果。这段代码的执行结果如下：

```
此颜色的三基色值是：12, 12, 12
```

```
此颜色的三基色值是：255, 255, 255
```

10.1.2 常量和静态只读字段的编译差别

常量和静态只读字段的差别在编译上差别最大，常量是在编译时获取其值，而静态只读字段是在运行时获取其值。下面看代码实例：

```

01 //Class1.cs 类库
02 using System;
03 namespace ClassLibrary
04 {
05     public class Class1
06     {
07         public static readonly int a = 11;
08     }
09 }

```

这段代码建立的是一个类库，代码很简单，在类中只定义了一个静态只读字段。下面再建立一个控制台项目，在项目中引用这个类库，其代码如下：

```

01 using System;
02 namespace MemberOfClass
03 {
04     class Program
05     {
06         static void Main(string[] args)
07         {
08             Console.WriteLine(ClassLibrary.Class1.a);
09             Console.ReadKey();
10         }
11     }
12 }

```

这段代码的类只包含了一个入口点方法，在方法中打印类库中的静态字段的值。首先将它们编译，执行结果如下：

```
11
```

接下来改变类库中静态只读字段的值为 12，然后只重新编译类库，不编译控制台项目。再运行代码，得到

如下结果:

12

由此可知, 静态只读字段是运行时才进行获取的。下面将类库中的静态只读字段修改为常量, 修改后的代码如下:

```
01 //Class1.cs 类库
02 using System;
03 namespace ClassLibrary
04 {
05     public class Class1
06     {
07         public const int a = 12;
08     }
09 }
```

这段代码将静态只读字段改变为常量, 控制台项目不用修改, 将它们进行编译。这时的运行结果为:

12

接下来将类库中的常量改变为 13, 然后重新编译类库, 控制台项目不重新编译。再次运行, 结果如下:

12

这是因为常量是编译时获取, 它的值被编译进程序集中了。所以, 当类库中的值发生变化时, 因为控制台项目没有重新编译, 所以它的输出值不变。

10.1.3 只读字段的 ref 和 out 传递

除了定义只读字段的同时初始化外, 对于实例只读字段, 只能在实例构造方法中改变只读字段的值或将其作为 ref 或 out 参数传递; 对于静态只读字段, 只能在静态构造方法中对其值进行改变, 或将其作为 ref 或 out 参数传递。除此外, 在程序中的其它任何部分都不能改变只读字段的值。下面通过一段代码实例来说明:

```
using System;
namespace 只读字段 ref 和 out 传递
{
    class Program
    {
        private readonly int a;
        private readonly int b;
        private static readonly int c;
        private static readonly int d;
        public Program()
        {
            Set(ref a, out b);
        }
        static Program()
        {
            Set1(ref c, ref d);
        }
        public void Set(ref int a, out int b)
        {
            a = 21;
```

```

        b = 22;
    }

    public static void Set1(ref int c, ref int d)
    {
        c = 23;
        d = 24;
    }

    public void Show()
    {
        Console.WriteLine("只读字段 a 的值是: {0}", a);
        Console.WriteLine("只读字段 a 的值是: {0}", b);
        Console.WriteLine("静态只读字段 c 的值是: {0}", c);
        Console.WriteLine("静态只读字段 d 的值是: {0}", d);
    }

    static void Main(string[] args)
    {
        Program myP = new Program();
        myP.Show();
        Console.ReadKey();
    }
}

```

在这段代码中首先定义了 4 个只读字段，其中 2 个是实例字段，另外 2 个是静态字段。然后定义了 2 个构造方法，其中一个是实例构造方法，另外一个为静态构造方法。如果要想让只读字段作为 ref 或 out 参数传递，需要另外定义两个方法，一个是实例方法，向其传递 1 个 ref 参数和 1 个 out 参数；另外一个为静态方法，向其传递 2 个 ref 参数。为何要定义这两个方法呢？因为实例只读字段只能在实例构造方法中赋值；静态只读字段只能在静态构造方法中赋值。定义这两个方法后，在各自的构造方法中调用它们，实现在构造方法中传递 ref 或 out 只读字段。那么为何为 Set 和 Set1 方法传递 ref 参数时，没有做字段的初始化操作呢？这是因为类的字段在进入构造方法时首先将被设置为默认值的缘故。这段代码的执行结果如下：

```

只读字段 a 的值是: 21
只读字段 a 的值是: 22
静态只读字段 c 的值是: 23
静态只读字段 d 的值是: 24

```

10.2 属性

属性被定义为字段的自然扩展，它与字段一样都是类的成员，它也有相关的类型。访问属性和访问字段的方法相同。因此，如果单纯从操作来讲，会觉得属性和字段很相像。但是，属性和字段不同，它的内存分配和字段是不同的。简单来讲，属性就是方法，只不过它的操作像极了字段。属性不表示存储位置，属性使用访问器指定它的值在被读取或写入时执行的语句。虽然访问属性的方式和访问字段相同，但是因为属性并不是字段，所以不能使用 ref 和 out 关键字将属性作为字段进行传递。下面看代码实例：

```

01 using System;
02 namespace MemberOfClass
03 {
04     class Program
05     {

```

```
06      //定义三个字段，其中一个是静态字段
07      private int a;
08      private int b;
09      private static int c;
10      //定义属性，通常将属性对应的字段名大写来作为属性名
11      public int A
12      {
13          //定义 get 访问器，用于返回字段值
14          get
15          {
16              return a;
17          }
18          //定义 set 访问器，为字段赋值
19          set
20          {
21              a = value;
22          }
23      }
24      //属性中可写语句用来对返回字段值或赋值做自定义操作
25      public int B
26      {
27          get
28          {
29              int myDay = DateTime.Now.Day;
30              if (myDay == 19)
31              {
32                  return b;
33              }
34              else
35              {
36                  return 10;
37              }
38          }
39          set
40          {
41              int myDay = DateTime.Now.Day;
42              if (myDay == 19 && value > 10)
43              {
44                  b = value;
45              }
46              else
47              {
48                  b = 10;
49              }
```

```

50         }
51     }
52     //定义静态属性
53     public static int C
54     {
55         get
56         {
57             return c;
58         }
59         set
60         {
61             c = value;
62         }
63     }
64     static void Main(string[] args)
65     {
66         Program p = new Program();
67         p.A = 11;
68         p.B = 22;
69         Program.C = 33;
70         Console.WriteLine("字段 a 的值是: {0}, 字段 b 的值是: {1}, 字段 c 的值是: {2}", p.A,
p.B, Program.C);
71         Console.ReadKey();
72     }
73 }
74 }

```

属性根据是否使用 `static` 关键字, 分为静态属性和实例属性。静态属性是属于类的, 而实例属性是属于类的实例的。这段代码的第 7、8、9 行分别定义了三个字段, 其中 a、b 是实例字段, c 是静态字段。

第 11 行代码定义的就是一个属性, 它用来对字段 a 进行存取操作。当一个定义的属性和一个字段对应时, 通常字段用小写, 而对应的属性用相同字母组合的大写形式。第 14 行开始是一个 `get` 访问器; 第 19 行开始是一个 `set` 访问器。访问器的作用是指定读取和写入该属性的相关联的操作。`get` 访问器就是读取访问器, 而 `set` 访问器就是写入访问器。属性只有这两种形式的访问器, 但是属性可以使用它们的一种或将它们同时组合使用。本段代码都是组合使用的这两种访问器。

第 16 行代码是使用 `return` 语句返回字段的值, 它就是属性 A 的读取操作。第 21 行代码是将写入的值赋值给字段 a。其中的 `value` 代表写入的值。属性 A 的逻辑很简单, 它只是完成了简单的对相应字段的赋值和读取操作。

第 25 行开始的对字段 b 进行操作的属性 B 就稍显复杂一些。第 27 行开始是一个 `get` 访问器, 在它的内部对操作进行了逻辑判断。第 29 行首先定义了一个 `int` 类型的变量来接收日期值。其中 `DateTime` 是 .Net 类库中的一个结构, `Now` 是它的一个属性, 它返回的还是一个 `DateTime` 对象, 它表示的是当前的日期和时间。而 `Day` 仍然是 `DateTime` 对象的一个属性, 它取出的是当前的日期。第 30 行对取出的日期进行判断, 如果当前的日期是 19 号, 则返回字段的值, 否则返回 10。第 39 行开始的是一个 `Set` 访问器。第 41 行也是先获取当前日期, 然后第 42 行对获取的日期进行判断, 如果日期为 19 并且为属性赋的值大于 10, 则将赋的值赋值给字段 b。否则将 b 设置为 10。

第 53 行开始的是一个静态属性, 因为它用 `static` 关键字修饰。静态属性 C 的作用是对静态字段 c 进

行存取操作。

在 Main 方法中，第 66 行代码定义了类的一个实例，它可以调用属于实例的实例属性，例如第 67 行和第 68 行。它的使用和对字段的操作是一样的。第 69 行是对静态属性进行赋值操作。因为静态属性是属于类的，所以要使用类名来访问它。第 70 行代码演示了属性的读取操作，可以看到，它的操作也和字段是一样的，就好像这个字段的访问权限是 public 的。实际上它执行的是相关属性中的代码。这段代码的执行结果如下：

字段 a 的值是：11，字段 b 的值是：10，字段 c 的值是：33

10.2.1 属性的读写属性

根据属性提供的访问器的类型，属性可以分为读写属性、只读属性和只写属性。如果一个属性含有 get 访问器和 set 访问器，则这个属性具有读写属性；如果只包含 get 访问器，则这个属性具有只读属性；如果只包含 set 访问器，则这个属性具有只写属性。当为一个只读属性赋值或引用一个只写属性，编译器会报错。下面看代码实例：

```
01 using System;
02 namespace MemberOfClass1
03 {
04     public class Test
05     {
06         private int a = 22;
07         private string s = "Hello World";
08         private bool b;
09         public int A
10         {
11             get
12             {
13                 return a;
14             }
15         }
16         public string S
17         {
18             set
19             {
20                 s = value;
21             }
22         }
23         public bool B
24         {
25             get
26             {
27                 return b;
28             }
29             set
30             {
31                 b = value;
32             }
33         }
34     }
35 }
```

```

33     }
34 }
35 class Program
36 {
37     static void Main(string[] args)
38     {
39         Test myT = new Test();
40         Console.WriteLine("字段 a 的值是: {0}", myT.A);
41         myT.S = "Test";
42         myT.B = true;
43         Console.WriteLine("字段 b 的值是: {0}", myT.B);
44         Console.ReadKey();
45     }
46 }
47 }

```

第 9 行代码定义了一个只读属性，它可以返回字段 a 的值。第 16 行代码定义了一个只写属性，它可以设置字段 s 的值。第 23 行代码定义了一个读写属性，它可以引用和设置字段 b 的值。当在第 40 行引用属性 A 的时候，属性的 get 访问器将被调用。当在第 41 行和第 42 行设置属性的值时，属性的 set 访问器将被调用。不只是在赋值的时候，当属性参与数学运算的时候，set 访问器也被调用。set 访问器中的隐式参数始终命名为 value。因此，在 set 访问器中不能再定义名为 value 的常量或局部变量。目前，属性中的 get 访问器和 set 访问器都比较简单，它们执行的都是对字段进行赋值和引用。在属性实际应用中，get 访问器和 set 访问器往往复杂的多，它们经常计算多个字段和调用方法来计算属性的值。在本段代码中，由于定义了只读或者只写属性，因此，字段的设置和读取只有依赖构造方法或其它的实例方法来实现。

10.2.2 自动实现的属性

如果要实现的属性只是用来取得或赋予字段的值，没有其他的操作。则可以使用自动实现的属性。使用自动实现的属性时，会隐式生成后备字段用于该属性，并自动实现 get 和 set 访问器用来读取和写入该字段。下面通过代码实例来说明：

```

using System;
namespace 自动实现的属性
{
    class Program
    {
        //自动实现的属性不用写字段，get 和 set 访问器不用写实现
        //只用分号结尾
        public int A
        {
            get;
            set;
        }
        static void Main(string[] args)
        {
            Program myP = new Program();
            myP.A = 33;
            Console.WriteLine("属性的值为: {0}", myP.A);
        }
    }
}

```

```
        Console.ReadKey();
    }
}
```

在使用自动实现的属性时，不用写具体的访问器实现代码，只将 `get` 和 `set` 以分号结束语句即可。编译器会自动隐式生成相应字段和访问器代码。这段代码等同于下面这段代码：

```
using System;
namespace 自动实现的属性
{
    class Program
    {
        private int a;
        public int A
        {
            get
            {
                return a;
            }
            set
            {
                a = value;
            }
        }
        static void Main(string[] args)
        {
            Program myP = new Program();
            myP.A = 33;
            Console.WriteLine("属性的值为: {0}", myP.A);
            Console.ReadKey();
        }
    }
}
```

这段代码的执行结果如下：

属性的值为: 33

如果在使用自动实现的属性时，仍然定义一个相应的字段会怎么样呢？是不是编译器就不会隐式生成相应的字段呢？看下面的代码：

```
using System;
namespace 自动生成属性与字段
{
    class Program
    {
        private int a;
        public int A
        {
```



```

        get;

        set;
    }

    static void Main(string[] args)
    {
        Program myP = new Program();
        myP.A = 33;
        Console.WriteLine("属性的值为: {0}", myP.A);
        Console.ReadKey();
    }
}

```

这段代码定义了一个和属性名称相同但小写的字段，那么现在猜想，编译器会使用这个定义好的字段，而不用隐式生成字段了，但结果却不是这样，在这段代码编写完的时候，开发环境会有 1 个警告信息，如下：

警告 1 从不使用字段“MemberOfClass1.Program.a” H:\书稿new\书稿源码\第十章\MemberOfClass1\Program.cs 6 21 MemberOfClass1

警告信息提示字段 a 从未使用过，这是什么原因呢？下面用反编译器打开编译好的程序集来查看 MSIL 代码，如图 10-1 所示。

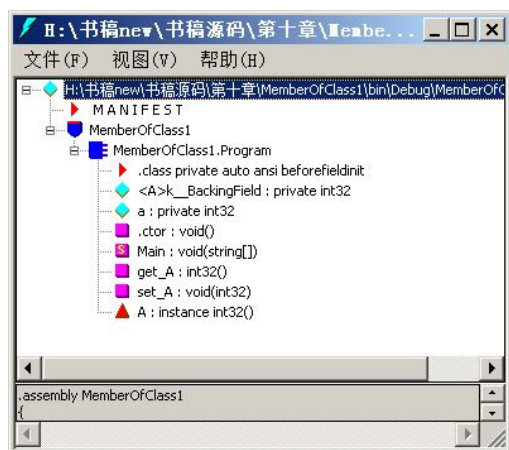


图 10-1 IL 结构

在窗口中可以看到以前缀 field 标注的就是字段，如果打开反汇编器看不到这个前缀则单击“视图”菜单，然后勾选“显示成员类型”，如图 10-2 所示。

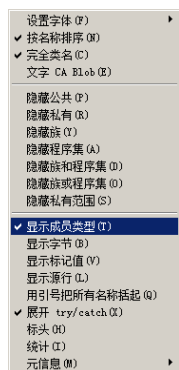


图 10-2 视图菜单

在成员列表中能够看到有一个字段，名字叫做<A>k__BackingField:private int32，冒号后是说明这个字段的类型，前面是字段的名称。这个字段才是编译器自动隐式生成的字段，所以前面代码中的问题就解决了，编译器自动隐式生成的字段和定义的字段 a 没有重名，而且 MSIL 代码可以使用 C#代码中不能使用的字符。

10.2.3 公共接口和隔离

属性的功能不只简单的和类的私有字段相对应，它还有隔离类的私有字段和向外提供公共接口的作用。

例如，有下面的代码：

```
01 using System;
02 using System.Drawing;
03 namespace MemberOfClass1
04 {
05     class Text
06     {
07         private int x, y; //文本的坐标
08         private string s; //文本
09         private Color c; //文本的前景色
10         public Text(int x, int y, string s, Color c)
11         {
12             this.x = x;
13             this.y = y;
14             this.s = s;
15             this.c = c;
16         }
17         /// <summary>
18         /// 获得文本位置的横坐标
19         /// </summary>
20         public int X
21         {
22             get
23             {
24                 return x;
25             }
26         }
27         /// <summary>
28         /// 获得文本位置的纵坐标
29         /// </summary>
30         public int Y
31         {
32             get
33             {
34                 return y;
35             }
36         }
37         /// <summary>
38         /// 获取或设置文本的内容
39         /// </summary>
40         public string Characters
41         {
```

```
42         get
43         {
44             return s;
45         }
46         set
47         {
48             s = value;
49         }
50     }
51     /// <summary>
52     /// 设置或取得文本前景色的属性
53     /// </summary>
54     public Color ForegroundColor
55     {
56         get
57         {
58             return c;
59         }
60         set
61         {
62             c = value;
63         }
64     }
65 }
66 class Program
67 {
68     static void Main(string[] args)
69     {
70         Text t = new Text(3, 3, "Hello World!", Color.Blue);
71         Console.WriteLine("文本的位置为: {0}, {1}", t.X, t.Y);
72         Console.WriteLine("文本的内容为: {0}", t.Characters);
73         Console.WriteLine("文本的前景色为: {0}", t.ForegroundColor);
74         Console.ReadKey();
75     }
76 }
77 }
```

这是表示一个文本的类，这段代码的 7-9 行定义了类的四个字段，前两个表示文本的横坐标和纵坐标，字段 s 表示文本的内容，字段 c 表示文本的前景色。

第 10 行是构造方法，它用来构造文本类的一个实例。

第 20 行是一个属性，它用来获得文本的横坐标，它是一个只读属性，不允许设置文本的坐标值。

第 30 行也是一个属性，它用来获得文本的纵坐标，它也是只读的，不允许设置文本的坐标值。

第 40 行是一个读写属性，它用来获得和设置文本的内容。

第 54 行也是一个读写属性，它用来获得和设置文本的前景颜色。这四个属性可以说构成了类对外的接口，通过它们可以获得或设置类实例的私有数据。

第 70 行定义了一个类的实例，然后第 71 行到第 73 行使用类对外的接口输出了类的内部状态数据。这段代码的执行结果如下：

文本的位置为：3,3

文本的内容为：Hello World!

文本的前景色为：Color [Blue]

现在假设有这样一种情况，这个程序集进行了升级，它添加了一个 Point 类，计划使用这个类来代替文本类中的私有字段 x 和 y。因为这样更符合封装的特性。如果没有属性这个公共的接口，而只是使用 public 的只读字段公开文本的坐标位置，现在遇到的情况就很麻烦，对类基本上无法进行升级改造。但是因为有了属性这个公共的接口，所以，类可以保持原来的对外接口不变，而内部状态数据又可以进行改变。这样，调用它的代码就完全都不必改变了。实例代码如下：

```
01 using System;
02 using System.Drawing;
03 namespace MemberOfClass1
04 {
05     class Point
06     {
07         public int X { get; set; }
08         public int Y { get; set; }
09     }
10     class Text
11     {
12         private Point p; //文本的坐标点
13         private string s; //文本
14         private Color c; //文本的前景色
15         public Text(int x, int y, string s, Color c)
16         {
17             this.p.X = x;
18             this.p.Y = y;
19             this.s = s;
20             this.c = c;
21         }
22         /// <summary>
23         /// 获得文本位置的横坐标
24         /// </summary>
25         public int X
26         {
27             get
28             {
29                 return p.X;
30             }
31         }
32         /// <summary>
33         /// 获得文本位置的纵坐标
34         /// </summary>
```

```
35     public int Y
36     {
37         get
38         {
39             return p.Y;
40         }
41     }
42     /// <summary>
43     /// 获取或设置文本的内容
44     /// </summary>
45     public string Characters
46     {
47         get
48         {
49             return s;
50         }
51         set
52         {
53             s = value;
54         }
55     }
56     /// <summary>
57     /// 设置或取得文本前景色的属性
58     /// </summary>
59     public Color ForegroundColor
60     {
61         get
62         {
63             return c;
64         }
65         set
66         {
67             c = value;
68         }
69     }
70 }
71 class Program
72 {
73     static void Main(string[] args)
74     {
75         Text t = new Text(3, 3, "Hello World!", Color.Blue);
76         Console.WriteLine("文本的位置为: {0}, {1}", t.X, t.Y);
77         Console.WriteLine("文本的内容为: {0}", t.Characters);
78         Console.WriteLine("文本的前景色为: {0}", t.ForegroundColor.ToString());
```

```
79         Console.ReadKey();
80     }
81 }
82 }
```

这段代码的第 5 行定义了一个 Point 类，然后修改文本类中的坐标字段，将原来包含两个 int 类型的字段替换成了一个 Point 对象。现在，只需对构造方法中的第 17 行和第 18 行更改，然后改变第 29 行和第 39 行的对文本坐标的读取方式。在第 75 行到第 78 行的调用代码都不必进行改动。虽然类的内部结构已经发生了很大的改变，但是从这里看来，使用类的方式没有发生变化。这就是使用属性进行接口设计的好处，它实际上也是分层模式设计的一种方法。属性本质上是一个函数成员，调用它的执行效率并不一定比直接调用字段效率要低，如果属性是非虚的，且只包含少量代码，编译器会用内联函数的方式来优化它，在调用这个属性的地方，将直接把代码粘贴过去。这样，既保证了执行效率，又保留了属性带来的设计上的灵活性。

10.2.4 非托管资源的初始化延迟

属性因为就是一种函数，所以它具有将非托管资源的初始化延迟到第一次引用该托管资源时进行。下面是代码实例：

```
01 using System;
02 using System.IO;
03 namespace MemberOfClass1
04 {
05     class InputAndOutput
06     {
07         private static StreamWriter output;
08         private static StreamReader input;
09         public static StreamWriter Output
10         {
11             get
12             {
13                 if (output == null)
14                 {
15                     output = new StreamWriter("driveinfo.txt");
16                 }
17                 return output;
18             }
19         }
20         public static StreamReader Input
21         {
22             get
23             {
24                 if (input == null)
25                 {
26                     input = new StreamReader("driveinfo.txt");
27                 }
28                 return input;
29             }
30         }
31     }
32 }
```

```

30     }
31 }
32 class Program
33 {
34     static void Main(string[] args)
35     {
36         DirectoryInfo[] di = new DirectoryInfo(@"c:\").GetDirectories();
37         Console.WriteLine("开始向文件输出: ");
38         foreach (DirectoryInfo d in di)
39         {
40             InputAndOutput.Output.WriteLine(d.Name);
41         }
42         Console.WriteLine("释放非托管资源");
43         InputAndOutput.Output.Dispose();
44         Console.WriteLine("开始向屏幕输出: ");
45         string line = string.Empty;
46         while ((line = InputAndOutput.Input.ReadLine()) != null)
47         {
48             Console.WriteLine(line);
49         }
50         Console.WriteLine("释放非托管资源");
51         InputAndOutput.Input.Dispose();
52         Console.ReadKey();
53     }
54 }
55 }

```

StreamWriter 是以一种特定的编码输出字符的类, StreamReader 是以一种特定的编码读取字符的类。它们默认使用的都是 UTF-8 编码。这段代码使用它们的目的是向文本文件中输入文本, 然后从文件中读取文本, 并向屏幕输出。这两个类包含在 System.IO 名称空间中, 对于物理文件的引用, 开发环境默认已经引用了, 为简化代码的输入, 可以使用 using 语句引用它。

第 7 行代码和第 8 行代码定义了 StreamWriter 和 StreamReader 的两个静态字段。

第 9 行代码定义了 StreamWriter 类型的属性, 它是一个只读属性, 只包括 get 访问器。在 get 访问器中, 首先判断字段 output 是否为空, 如果它为空, 则说明它没有实例化。因此第 15 行将对它进行实例化, 构造方法中的字符串是要打开并向其写入内容的文件名。

与它相似, 第 20 行代码定义了一个 StreamReader 类型的属性, 在它里面为 input 字段创建实例。构造方法中打开的就是前面要写入的文件。

第 36 行使用了 .Net 类库中的 DirectoryInfo 类, 创建 DirectoryInfo 类实例的构造方法包含的参数是指定对哪个目录创建 DirectoryInfo 实例, 然后调用 GetDirectories() 方法获得一个目录信息的数组。

第 38 行对接收到的数组进行迭代, 每迭代一次, 就调用 output 的 WriteLine 方法向文件写入目录的名称。注意, 当调用 InputAndOutput.Output.WriteLine(d.Name); 这个语句的时候, 属性要首先判断字段是否已经被实例化, 如果没有实例化, 则现在创建实例。这就是属性延迟非托管资源直到调用时执行的功能。它不必在代码一开始执行的时候就创建非托管资源。

当向文件中输出完毕后, 要在第 43 行调用 StreamWriter 类的 Dispose 方法释放非托管资源。

第 45 行定义一个空字符串用来接收从文件中读到的每一行文本。

第 46 行代码是一个循环语句，在循环语句中调用了 input 字段的 ReadLine 方法，这时因为调用的是 Input 属性，所以属性的 get 访问器被执行，input 字段在使用的时候被创建实例。每次循环，都将读取一行文本，直到文件末尾读到一个空行时跳出循环。

在第 48 行，对每一行读到的文本，都使用 Console 类的 WriteLine 方法输出到屏幕上。

第 51 行调用 Dispose 方法释放 StreamReader 实例的非托管资源。这段代码的执行结果如下：

```
开始向文件输出：
释放非托管资源
开始向屏幕输出：
$Recycle.Bin
Boot
CFLog
DBDownload
Destination Folder
Documents and Settings
Intel
KRECYCLE
ksDownloads
PerfLogs
Program Files
ProgramData
QMDownload
RavBin
Recovery
System Volume Information
Temp
testlog
Users
Windows
释放非托管资源
```

通过这段代码的演示知道，属性可以将非托管资源的申请延迟到使用时执行，它的作用就是使用时才创建非托管资源，不使用不会创建非托管资源，从而降低系统资源的消耗。

10.2.5 属性访问器的权限指定

除了属性成员本身可以指定访问权限外，属性中的访问器也可以进一步指定其访问权限。访问权限修饰符有下面几种，有几种前面已经接触并介绍过。

- public
- protected
- private
- internal
- protected internal

首先来介绍一下这几种访问修饰符的作用，public 代表公开的访问权限；protected 代表本类内部和派生类可以访问；private 代表只有本类内部可以访问；internal 代表本程序集可以访问；protected internal 代表本类内部、本类的派生类或者本程序集可以访问。再详细说一下 protected internal 权限，这两个关键字是并列的，并集的关系。因为一个类的派生类有可能位于另一个程序集中，所以，单纯看 protected 权限，就是除了本类内部可以访问外，无论派生类位于本程序集还是其它程序集，都可以访问它。而 internal

代表本程序集中的任何类都可以访问它，而不论这个类是不是本类的派生类。所以 `protected internal` 的权限就是这两种情况的并集，本程序集的任何类可以访问，本类的派生类可以访问（派生类可以位于任何程序集中）。

对于属性的访问器，可以使用的访问权限修饰符有下面几种：

- `protected`
- `private`
- `internal`
- `protected internal`

从列表中可以发现，访问器可以使用的访问权限修饰符少了 `public`。这是因为访问器的可访问性权限级别必须高于属性定义的访问权限级别。代码实例如下：

```
01 using System;
02 namespace MemberOfClass1
03 {
04     class Test
05     {
06         private int a;
07         public int A
08         {
09             public get
10             {
11                 return a;
12             }
13             public set
14             {
15                 a = value;
16             }
17         }
18     }
19 }
```

这段代码是一段出错的代码，第 7 行代码定义了一个属性，可以看到，它的可访问性是 `public`。代码的第 9 行和第 13 行定义了两个访问器。它们的访问权限也定义为 `public`。代码很好理解，现在来编译它，这时会得到如下的错误提示：

```
错误 1  不能为属性或索引器“MemberOfClass1.Test.A”的两个访问器同时指定可访问性修饰符 E:\书稿 new\书稿源码\第十章\MemberOfClass1\Program.cs 7 20 MemberOfClass1
错误 2  。“MemberOfClass1.Test.A.get”访问器的可访问性修饰符必须比属性或索引器“MemberOfClass1.Test.A”具有更强的限制 E:\书稿 new\书稿源码\第十章\MemberOfClass1\Program.cs 9 20 MemberOfClass1
错误 3  。“MemberOfClass1.Test.A.set”访问器的可访问性修饰符必须比属性或索引器“MemberOfClass1.Test.A”具有更强的限制 E:\书稿 new\书稿源码\第十章\MemberOfClass1\Program.cs 13 20 MemberOfClass1
```

出现这三个错误的原因是，对于访问器指定访问权限修饰符有如下的限制：

1. 访问器的访问修饰符只能用于一个访问器，也就是说，属性的两个访问器中只能为其中一个访问器指定访问修饰符。
2. 访问器的访问权限必须高于属性的访问权限。如果访问器没有指定访问权限，它就继承属性设置的访问

权限。因为第 7 行代码定义了 public 的访问权限，所以访问器再指定同样的访问权限就会出错。下面修改代码如下：

```

01 using System;
02 namespace MemberOfClass1
03 {
04     class Test
05     {
06         private int a;
07         public int A
08         {
09             protected internal get
10             {
11                 return a;
12             }
13         }
14     }
15 }

```

在这段代码中，对于属性的 get 访问器，将它的访问权限设置为 protected internal。但是为了不对外公布字段的设置功能，将它的 set 访问器去掉了，这时它就是一个只读属性。看起来好像没有什么问题，再次编译代码。可是又会得到一个错误信息，如下：

```

错误 1      “MemberOfClass1.Test.A”：仅当属性或索引器同时具有 get 访问器和 set 访问器时，才能
对访问器使用可访问性修饰符  E:\书稿 new\书稿源码\第十章\MemberOfClass1\Program.cs    7    20
MemberOfClass1

```

从错误信息可以知道，设定访问器的访问权限还有如下的限制：

当属性不是用来重写基类的属性时，只有属性的 get 访问器和 set 访问器同时具备时，才可以对访问器使用访问修饰符。重写的概念后面会介绍到，这里了解即可。下面列出属性的访问权限和访问器可以指定的访问权限的类别：

- 如果属性声明了 public 可访问性，则访问器的访问权限可以为 protected internal、internal、protected 或 private。
- 如果属性声明了 protected internal 可访问性，则访问器的访问权限可以为 internal、protected 或 private。
- 如果属性声明了 internal 或 protected 可访问性，则访问器的访问权限可以为 private。
- 如果属性声明了 private 可访问性，则访问器任何访问权限修饰符都不可以用。

下面看代码实例：

```

01 using System;
02 namespace MemberOfClass1
03 {
04     class Test
05     {
06         private int a;
07         private int b;
08         private int c;
09         public int A
10         {

```

```
11         get
12         {
13             return a;
14         }
15         protected internal set
16         {
17             a = value;
18         }
19     }
20     protected internal int B
21     {
22         get
23         {
24             return b;
25         }
26         private set
27         {
28             b = value;
29         }
30     }
31     private int C
32     {
33         get
34         {
35             return c;
36         }
37         set
38         {
39             c = value;
40         }
41     }
42     public Test(int a, int b, int c)
43     {
44         this.A = a;
45         this.B = b;
46         this.C = c;
47     }
48     public void Show()
49     {
50         Console.WriteLine("属性 A 的值是: {0}, 属性 B 的值是: {1}, 属性 C 的值是:
51             {2}", A, B, C);
52     }
53     class Test1 : Test
```

```

54     {
55         public Test1()
56             : base(11, 12, 13)
57         {
58         }
59         public void Set()
60         {
61             this.A = 33;
62         }
63         public void Show1()
64         {
65             Console.WriteLine("在派生类中打印属性的值:");
66             Console.WriteLine("属性 A 的值是: {0}, 属性 B 的值是: {1}", this.A, this.B);
67             Console.WriteLine("调用基类方法打印属性的值:");
68             this.Show();
69         }
70     }
71 }

```

这段代码在总体上定义了两个类，类 Test1 继承自类 Test。在基类中定义了 3 个属性，第 9 行的属性 A 是 public 权限的；第 20 行的属性 B 是 protected internal 权限的；第 31 行的属性 C 是 private 的。

属性 A 中的第 15 行 set 访问器又进一步定义了访问权限 protected internal，它能被派生类还有本程序集访问。

第 26 行的 set 访问器定义了 private 权限，它只能被本类访问。属性 C 因为设置了 private 权限，所以它的访问器不再能设置任何权限，访问器的权限也是 private 的。

因为这三个属性的 set 访问器都能被本类访问，因此第 42 行的构造方法能够为每个属性赋值。

对于没有定义访问权限修饰符的访问器来说，它将继承定义它的属性的访问权限。因此属性 A 的 get 访问器的权限为 public；属性 B 的 get 访问器的访问权限为 protected internal；属性 C 的 get 访问器的访问权限是 private。对于第 48 行定义的方法来说，这三个访问器都可以访问，因此它可以打印出它们的值。但是对于第 63 行代码定义的派生类方法来说，只有属性 A 和属性 B 的 get 访问器可以访问，所以，第 66 行代码可以打印属性 A 和属性 B 的值。

对于派生类来说，只有属性 A 的 set 访问器可以被派生类访问，所以第 59 行的方法只可以设置属性 A 的值。下面再定义一个含有入口点方法的类，来验证结果，代码如下：

```

class Program
{
    static void Main(string[] args)
    {
        Test1 myTest = new Test1();
        myTest.Set();
        myTest.Show1();
    }
}

```

这时得到的运行结果如下：

在派生类中打印属性的值：

属性 A 的值是：33, 属性 B 的值是：12

调用基类方法打印属性的值:

属性 A 的值是: 33, 属性 B 的值是: 12, 属性 C 的值是: 13

这段代码中的 A 属性的 set 访问器和 B 属性的 get 访问器都是 protected internal。下面编写代码来验证是否能在本程序集中的其它非派生类还有其它程序集中的派生类中访问它们。首先修改本程序集的代码如下:

```
01 using System;
02 namespace MemberOfClass1
03 {
04     class Test
05     {
06         private int a;
07         private int b;
08         private int c;
09         public int A
10         {
11             get
12             {
13                 return a;
14             }
15             protected internal set
16             {
17                 a = value;
18             }
19         }
20         protected internal int B
21         {
22             get
23             {
24                 return b;
25             }
26             private set
27             {
28                 b = value;
29             }
30         }
31         private int C
32         {
33             get
34             {
35                 return c;
36             }
37             set
38             {
39                 c = value;
```

```
40         }
41     }
42     public Test(int a, int b, int c)
43     {
44         this.A = a;
45         this.B = b;
46         this.C = c;
47     }
48     public void Show()
49     {
50         Console.WriteLine("属性 A 的值是: {0}, 属性 B 的值是: {1}, 属性 C 的值是: {2}", A, B, C);
51     }
52 }
53 class Test1
54 {
55     private Test t;
56     public Test1()
57     {
58         t = new Test(22, 33, 44);
59     }
60     public void Set()
61     {
62         t.A = 100;
63     }
64     public void Show1()
65     {
66         Console.WriteLine("在非派生类中打印属性的值: ");
67         Console.WriteLine("属性 A 的值是: {0}, 属性 B 的值是: {1}", t.A, t.B);
68         Console.WriteLine("调用 Test 类的方法打印属性的值: ");
69         t.Show();
70     }
71 }
72 }
```

Test 类的代码没有改变, Test1 类由原来的 Test 类的派生类更改为一个独立的类。为了访问 Test 类的成员, 它必须定义一个 Test 类的引用变量作为成员, 第 55 行代码定义了一个 Test 类的引用变量作为字段。第 56 行代码是一个构造方法, 它的作用是为 Test 类的字段 t 创建实例。

第 60 行代码是一个实例方法, 它访问了属性 A 的 set 访问器, 因为属性 A 的 set 访问器的访问权限允许本程序集类可以访问, 因此这里的设置操作是允许的。第 67 行代码打印了属性 A 和属性 B 的值, 它们将访问属性的 get 访问器。因为属性 A 的 get 访问器是 public 的; 属性 B 的 get 访问器是 protected internal 的, 它允许本程序集访问。因此这里的访问也是允许的。下面再来实验其它程序集中的派生类是否能够访问。首先将本程序集中的两个类添加 public 访问修饰符, 以便其它程序集可以访问。接下来改变本程序集的项目属性为类库, 方法是在项目名称上单击右键, 然后单击“属性”, 接下来在“应用程序”项中改变项目的输出类型为“类库”, 如图 10-3 所示。



图 10-3 输出类型

接下来新建一个控制台项目，然后在它的项目引用中物理引用这个程序集，最后编写如下代码：

```

01 using System;
02 namespace MemberOfClass1
03 {
04     class Program:Test
05     {
06         public Program()
07             : base(55, 56, 57)
08         {
09         }
10         public void Set()
11         {
12             this.A = 200;
13         }
14         public void Show1()
15         {
16             Console.WriteLine("在其它程序集中的派生类打印属性 A 的值为: {0}, B 的值为:
17                 {1}", this.A, this.B);
18         }
19         static void Main(string[] args)
20         {
21             Program p = new Program();
22             p.Set();
23             p.Show1();
24             Test1 t = new Test1();
25             t.Set();
26             t.Show1();
27             Console.ReadKey();
28         }
29     }

```

为了引用方便，这个程序集的命名空间也定义成和 Test 类所在的命名空间一样的名称。第 4 行代码定义了一个类 Program，它继承自类 Test。第 6 行定义一个构造方法，使用构造方法初始值设定项为基类的构造方法传值。第 10 行定义一个实例方法，它用来设置从基类继承来的属性 A 的值，因为属性 A 具有 protected internal 访问权限，所以它可以被其它程序集中的派生类访问。第 14 行代码打印属性 A 和属性 B 的值。因为属性 A 的 get 访问器具有 public 权限，属性 B 的 get 访问器具有 protected internal 权限，所以它们都可以被其它程序集中的派生类访问。第 20 行代码定义了一个派生类的实例用来访问 Set 方法和 Show1

方法。第 23 行代码定义了一个 Test1 类的实例用来验证 Test1 类中的操作是否正确。这段代码的执行结果如下：

在其它程序集中的派生类打印属性 A 的值为：200, B 的值为：56

在非派生类中打印属性的值：

属性 A 的值是：100, 属性 B 的值是：33

调用 Test 类的方法打印属性的值：

属性 A 的值是：100, 属性 B 的值是：33, 属性 C 的值是：44

10.3 索引器

索引器是类的一种成员，使用它的方式和使用数组的方式相同。但是索引器也并不属于变量，不能使用 ref 和 out 将索引器作为参数传递。在这一点上，它和属性是相同的。下面看代码实例：

```

01 using System;
02 namespace MemberOfClass1
03 {
04     public class CustomArray
05     {
06         private int[] a;
07         public int Length
08         {
09             get;
10             private set;
11         }
12         /// <summary>
13         /// 构造方法，用来初始化类中的数组
14         /// </summary>
15         /// <param name="length">定义数组的长度</param>
16         public CustomArray(int length)
17         {
18             this.Length = length;
19             a = new int[this.Length];
20             for (int i = 0; i < this.Length; i++)
21             {
22                 a[i] = 0;
23             }
24         }
25         public int this[int index]
26         {
27             get
28             {
29                 if (index >= 0 && index < this.Length)
30                 {
31                     return a[index];
32                 }
33                 else
34                 {

```



```
35         Console.WriteLine("访问的值不在数组范围内, 请确定索引, 现在将返回错误  
36         return -1;  
37     }  
38 }  
39 set  
40 {  
41     if (index >= 0 && index < this.Length)  
42     {  
43         if (value > 0 && value <= 100)  
44         {  
45             a[index] = value;  
46         }  
47         else  
48         {  
49             Console.WriteLine("所赋的值不在数组范围内, 请确定目标值。");  
50         }  
51     }  
52     else  
53     {  
54         Console.WriteLine("所访问索引超出数组范围");  
55     }  
56 }  
57 }  
58 }  
59 public class Program  
60 {  
61     static void Main(string[] args)  
62     {  
63         CustomArray ca = new CustomArray(20);  
64         for (int i = 0; i < 10; i++)  
65         {  
66             //因为索引器指定数组中的元素大于等于 0, 所以这里要加 1.  
67             ca[i] = i + 1;  
68         }  
69         Console.WriteLine("自定义数组类中的值是: ");  
70         for (int i = 0; i < ca.Length; i++)  
71         {  
72             Console.Write(ca[i] + " ");  
73         }  
74         Console.ReadKey();  
75     }  
76 }  
77 }
```

这段代码的目的是，类的索引器要对类中的 `int` 类型的数组进行存取操作。因为有索引的存在，因此可以使用操作数组的方式来操作类的实例。

第 6 行代码定义了一个 `int` 类型的数组变量，但是没有初始化数组。第 7 行定义了一个属性，它用来保存定义的数组的长度。它的两个访问器的访问权限并不一样，`set` 访问器是为类的内部使用的，所以设置了 `private` 权限。而 `get` 访问器因为在第 70 行被外界代码访问，所以它使用了 `public` 属性。属性 `Length` 定义成了自动实现的属性。第 16 行是类的实例构造方法，它的目的是对数组进行初始化。它带有一个参数，参数在第 18 行赋值给了属性 `Length`，然后第 19 行使用这个长度初始化数组。第 20 行使用 `for` 语句将数组进行初始化。

第 25 行开始定义的是一个索引器。索引器是以 `this` 关键字作为标志，前面是索引器的类型和访问权限。`this` 关键字后跟一对中括号，里面是索引器的参数，这个参数是有类型的。但是参数的名字可以随意命名，但通常使用 `index` 作为参数的名称。索引器和属性是很相似的，因为索引器其实也是一个函数。第 27 行开始是一个 `get` 访问器，第 29 行对索引器的参数进行判断，如果参数在数组的索引范围内，则返回数组的相应成员。否则打印出错信息，并返回 `-1`。这里其实应该设计成抛异常，但是因为异常还没有介绍过，所以这里用打印信息来代替。第 39 行开始是 `set` 访问器，在对数组内容进行赋值前，它也要对参数 `index` 进行判断，这是第 41 行的作用。对参数判断后，还要对传入的参数 `value` 进行判断，因为这是索引设计的目的，它要把范围在 1-100 之间的整数存入数组。如果存入的数值不在这个范围内，则第 49 行会打印错误。

第 59 行代码是定义的一个带有入口点方法的类，第 63 行代码定义了类 `CustomArray` 的一个实例。第 64 行代码使用 `for` 语句对索引赋值。注意第 67 行代码中，对实例 `ca` 像使用数组一样进行了赋值操作。这就是索引器的作用，它调用了索引器的 `set` 访问器；第 70 行代码打印了 `ca` 的值，它调用了索引器的 `get` 访问器。这段代码的执行结果如下：

自定义数组类中的值是：

```
1 2 3 4 5 6 7 8 9 10 0 0 0 0 0 0 0 0 0 0
```

如果将第 67 行代码修改如下：

```
ca[i] = i;
```

这时因为存入的值不符合索引器中定义的范围，所以这时会打印错误，执行结果如下：

所赋的值不在数组范围内，请确定目标值。

自定义数组类中的值是：

```
0 1 2 3 4 5 6 7 8 9 0 0 0 0 0 0 0 0 0 0
```

因为所赋的第一个值不在范围，所以打印出错信息。由数组的内容可以看到，第一个成员仍保持着初始化后的初始值。

10.3.1 索引器的重载

类中的索引器可以被重载，重载的依据是索引器的参数个数和参数类型，它们也就是索引器的签名，只要多个索引器的参数个数或参数类型不同即可实现索引器的重载。但是索引器的包含元素的类型和索引器参数的名称不是索引器签名的一部分。例如前面代码实例中的索引器是 `int` 类型的，参数的名称是 `index`。它们不是索引器的签名，不能根据它们构成索引器的重载。要实现重载，则索引器的签名必须不同于在同一个类中定义的其他索引器的签名。下面是代码实例：

```
001 using System;
002 namespace MemberOfClass1
003 {
004     public class CustomArray
005     {
006         private int[] a;
007         private int[,] b;
```

```
008     public int Length
009     {
010         get;
011         private set;
012     }
013     /// <summary>
014     /// 数组的第一维长度
015     /// </summary>
016     public int NumOfRows
017     {
018         get;
019         private set;
020     }
021     /// <summary>
022     /// 数组的第二维长度
023     /// </summary>
024     public int NumOfCols
025     {
026         get;
027         private set;
028     }
029     /// <summary>
030     /// 构造方法，用来初始化类中的数组
031     /// </summary>
032     /// <param name="length">定义数组的长度</param>
033     public CustomArray(int length,int rows,int cols)
034     {
035         this.Length = length;
036         this.NumOfRows = rows;
037         this.NumOfCols = cols;
038         a = new int[this.Length];
039         for (int i = 0; i < this.Length; i++)
040         {
041             a[i] = 0;
042         }
043         b = new int[this.NumOfRows, this.NumOfCols];
044         //初始化二维数组
045         for (int i = 0; i < this.NumOfRows; i++)
046         {
047             for (int j = 0; j < this.NumOfCols; j++)
048             {
049                 b[i, j] = 0;
050             }
051         }
```

```
052     }
053     public int this[int index]
054     {
055         get
056         {
057             if (index >= 0 && index < this.Length)
058             {
059                 return a[index];
060             }
061             else
062             {
063                 Console.WriteLine("访问的值不在数组范围内, 请确定索引, 现在将返回错误代码-1");
064                 return -1;
065             }
066         }
067         set
068         {
069             if (index >= 0 && index < this.Length)
070             {
071                 if (value > 0 && value <= 100)
072                 {
073                     a[index] = value;
074                 }
075                 else
076                 {
077                     Console.WriteLine("所赋的值不在数组范围内, 请确定目标值。");
078                 }
079             }
080             else
081             {
082                 Console.WriteLine("所访问索引超出数组范围");
083             }
084         }
085     }
086     public int this[int row, int col]
087     {
088         get
089         {
090             if ((row >= 0 && row < this.NumOfRows) && (col >= 0 && col < this.NumOfCols))
091             {
092                 return b[row, col];
093             }
094             else
```

```
095         {
096             Console.WriteLine("访问的值不在数组范围内，请确定索引, 现在将返回错误代码-1");
097             return -1;
098         }
099     }
100     set
101     {
102         if ((row >= 0 && row < this.NumOfRows) && (col >= 0 && col < this.NumOfCols))
103         {
104             if (value > 0 && value <= 100)
105             {
106                 b[row, col] = value;
107             }
108             else
109             {
110                 Console.WriteLine("所赋的值不在数组范围内，请确定目标值。");
111             }
112         }
113         else
114         {
115             Console.WriteLine("所访问索引超出数组范围");
116         }
117     }
118 }
119 }
120 public class Program
121 {
122     static void Main(string[] args)
123     {
124         CustomArray ca = new CustomArray(20, 5, 4);
125         for (int i = 0; i < 10; i++)
126         {
127             //因为索引器指定数组中的元素大于等于 0，所以这里要加 1.
128             ca[i] = i;
129         }
130         Console.WriteLine("自定义数组类中的值是：");
131         for (int i = 0; i < ca.Length; i++)
132         {
133             Console.Write(ca[i] + " ");
134         }
135         Console.WriteLine(); //换行作用
136         for (int i = 0; i < 5; i++)
137         {
```

```

138         for (int j = 0; j < 4; j++)
139         {
140             ca[i, j] = j + 1;
141         }
142     }
143     Console.WriteLine("自定义数组类中重载索引的值是: ");
144     for (int i = 0; i < 5; i++)
145     {
146         for (int j = 0; j < 4; j++)
147         {
148             Console.Write(ca[i, j] + " ");
149         }
150     }
151     Console.ReadKey();
152 }
153 }
154 }

```

在上一节代码的基础上，第 7 行又增加了一个二维数组类型的变量作为字段。第 16 行代码和第 24 行代码分别定义了两个属性，它们分别代表二维数组第一维的长度和第二维的长度。第 33 行的构造方法中增加了两个参数用来设置二维数组的第一维长度和第二维长度。在代码的第 43 行为字段 b 进行实例化。实例化数组的每一维的长度访问的就是自动属性 NumOfRows 的 get 访问器和 NumOfCols 的 get 访问器。代码的第 45 行使用了一个嵌套的 for 循环对数组 b 进行初始化。它将二维数组的每一个元素都初始化为 0。

第 86 行代码定义了一个索引器，注意索引器的参数个数是两个，每一个都是 int 类型。这个索引器和前一个索引器构成了重载。索引器的 get 访问器中，第 90 行代码对索引的两个参数进行判断，判断它们是否超出了数组 b 的索引范围。和带一个参数的索引器逻辑相同，如果符合数组的索引范围，则返回数组的相关元素，否则打印错误信息，并返回-1。第 100 行的 set 访问器中，首先也对索引器的参数进行判断，如果符合条件，接下来再判断设置的值是否符合要求，如果符合，则为数组 b 的对应索引元素进行赋值。否则打印错误消息。

在 Main 方法中，第 124 行代码初始化了类的实例，并为它传入了三个参数。如果要为带两个参数的索引器赋值，则仍需嵌套的 for 循环；打印带两个参数的索引器的值，也使用了第 144 行代码所演示的嵌套 for 循环。

第 140 行代码访问的就是带两个参数的索引器，可以看到，索引器的名字就是创建的类的实例的名字。因为在类中定义了两个索引器，所以根据索引器的参数的类型和个数不同，调用了符合相关签名的索引器。定义的这两个索引器构成了索引器重载。

10.3.2 索引器和属性的区别

从索引器和属性的定义上来看，索引器和属性是非常相似的。它们都使用了 get 和 set 访问器。在 get 和 set 访问器的访问权限上，它们也是一样的。代码实例如下：

```

001 using System;
002 namespace MemberOfClass1
003 {
004     public class CustomArray
005     {
006         private int[] a;
007         public int Length

```

```
008     {
009         get;
010         private set;
011     }
012     /// <summary>
013     /// 构造方法，用来初始化类中的数组
014     /// </summary>
015     /// <param name="length">定义数组的长度</param>
016     public CustomArray(int length)
017     {
018         this.Length = length;
019         a = new int[this.Length];
020         for (int i = 0; i < this.Length; i++)
021         {
022             a[i] = 0;
023         }
024     }
025     public int this[int index]
026     {
027         get
028         {
029             if (index >= 0 && index < this.Length)
030             {
031                 return a[index];
032             }
033             else
034             {
035                 Console.WriteLine("访问的值不在数组范围内，请确定索引，现在将返回错
误代码-1");
036                 return -1;
037             }
038         }
039         protected set
040         {
041             if (index >= 0 && index < this.Length)
042             {
043                 if (value > 0 && value <= 100)
044                 {
045                     a[index] = value;
046                 }
047                 else
048                 {
049                     Console.WriteLine("所赋的值不在数组范围内，请确定目标值。");
050                 }
051             }
052         }
053     }
054 }
```

```

051         }
052         else
053         {
054             Console.WriteLine("所访问索引超出数组范围");
055         }
056     }
057 }
058 }
059 public class Program
060 {
061     static void Main(string[] args)
062     {
063         CustomArray ca = new CustomArray(20);
064         for (int i = 0; i < 10; i++)
065         {
066             //因为索引器指定数组中的元素大于等于 0，所以这里要加 1.
067             ca[i] = i;
068         }
069         Console.WriteLine("自定义数组类中的值是：");
070         for (int i = 0; i < ca.Length; i++)
071         {
072             Console.Write(ca[i] + " ");
073         }
074         Console.ReadKey();
075     }
076 }
077 }

```

第 39 行代码为索引器的 set 访问器定义了 protected 权限。而第 25 行为整个索引器定义了 public 权限。所以，索引器的 get 访问器将继承索引器的 public 权限，它可以被类外访问。但是 set 访问器由于定义了更严格的访问权限，所以它只能被它的派生类所访问。这样，第 67 行代码就会出错。出错信息如下：

错误 1 由于 set 访问器不可访问，因此不能在此上下文中使用属性或索引器“MemberOfClass1.CustomArray.this[int]” E:\书稿 new\书稿源码\第十章\MemberOfClass1\Program.cs 67 17 MemberOfClass1

下面修改代码如下：

```

001 using System;
002 namespace MemberOfClass1
003 {
004     public class CustomArray
005     {
006         private int[] a;
007         public int Length
008         {
009             get;
010             private set;

```



```
011     }
012     /// <summary>
013     /// 构造方法，用来初始化类中的数组
014     /// </summary>
015     /// <param name="length">定义数组的长度</param>
016     public CustomArray(int length)
017     {
018         this.Length = length;
019         a = new int[this.Length];
020         for (int i = 0; i < this.Length; i++)
021         {
022             a[i] = 0;
023         }
024     }
025     public int this[int index]
026     {
027         get
028         {
029             if (index >= 0 && index < this.Length)
030             {
031                 return a[index];
032             }
033             else
034             {
035                 Console.WriteLine("访问的值不在数组范围内，请确定索引，现在将返回错误代码-1");
036                 return -1;
037             }
038         }
039         protected set
040         {
041             if (index >= 0 && index < this.Length)
042             {
043                 if (value > 0 && value <= 100)
044                 {
045                     a[index] = value;
046                 }
047                 else
048                 {
049                     Console.WriteLine("所赋的值不在数组范围内，请确定目标值。");
050                 }
051             }
052             else
053             {
```

```

054             Console.WriteLine("所访问索引超出数组范围");
055         }
056     }
057 }
058 }
059 class MyCustomArray : CustomArray
060 {
061     public MyCustomArray(int length)
062         : base(length)
063     {
064     }
065     public void SetArray()
066     {
067         for (int i = 0; i < 10; i++)
068         {
069             this[i] = i;
070         }
071     }
072 }
073 public class Program
074 {
075     static void Main(string[] args)
076     {
077         MyCustomArray myCA = new MyCustomArray(20);
078         myCA.SetArray();
079         Console.WriteLine("自定义数组类中的值是: ");
080         for (int i = 0; i < myCA.Length; i++)
081         {
082             Console.Write(myCA[i] + " ");
083         }
084         Console.ReadKey();
085     }
086 }
087 }

```

第 59 行代码定义了一个派生类继承自 CustomArray 类，然后在第 65 行定义了一个实例方法，在方法中访问基类的索引器。还是为索引器通过 for 语句赋值，但是第 69 行访问基类的索引器要直接使用 this 关键字加中括号，中括号里面是索引变量这种方式。

第 77 行定义了一个派生类的对象，然后第 78 行调用了派生类的 SetArray() 方法来设置索引器的值。因为索引器的 get 访问器是 public 权限的，所以第 82 行访问索引器的 get 访问器不会有问题。这段代码的执行结果如下：

所赋的值不在数组范围内，请确定目标值。

自定义数组类中的值是：

0 1 2 3 4 5 6 7 8 9 0 0 0 0 0 0 0 0 0 0

索引器中的访问器的访问权限和属性的访问权限规则是一样的，如下：

当索引器不是用来重写基类的索引器时，只有索引器的 `get` 访问器和 `set` 访问器同时具备时，才可以对访问器使用访问修饰符。下面列出索引器的访问权限和访问器可以指定的访问权限的类别：

- 如果索引器声明了 `public` 可访问性，则访问器的访问权限可以为 `protected internal`、`internal`、`protected` 或 `private`。
- 如果索引器声明了 `protected internal` 可访问性，则访问器的访问权限可以为 `internal`、`protected` 或 `private`。
- 如果索引器声明了 `internal` 或 `protected` 可访问性，则访问器的访问权限可以为 `private`。
- 如果索引器声明了 `private` 可访问性，则访问器任何访问权限修饰符都不可以用。

但是索引器和属性也有不同点，首先来看实例代码：

```
001 using System;
002 namespace MemberOfClass
003 {
004     class Program
005     {
006         private static int[] a;
007         private static int Length
008         {
009             get;
010             private set;
011         }
012         static Program()
013         {
014             Length = 20;
015             a = new int[Length];
016             for (int i = 0; i < Length; i++)
017             {
018                 a[i] = 0;
019             }
020         }
021         public static int this[int index]
022         {
023             get
024             {
025                 if (index >= 0 && index < Length)
026                 {
027                     return a[index];
028                 }
029                 else
030                 {
031                     Console.WriteLine("索引值超出范围");
032                     return -1;
033                 }
034             }
035             set
```

```

036         {
037             if (index >= 0 && index < Length)
038             {
039                 a[index] = value;
040             }
041             else
042             {
043                 Console.WriteLine("索引值超出范围");
044             }
045         }
046     }
047     static void Main(string[] args)
048     {
049         for (int i = 0; i < Program.Length; i++)
050         {
051             Program[i] = i;
052         }
053         Console.ReadKey();
054     }
055 }
056 }

```

这段代码的编写意图是想和静态属性一样，先在第 6 行定义一个静态的数组，然后在第 12 行通过静态构造方法对静态数组进行初始化。第 21 行计划和静态属性一样定义一个静态索引器。但是在编译这段代码的时候，编译器将会提示错误，如下：

```

错误 1    修饰符“static”对该项无效 H:\书稿 new\书稿源码\第十章\MemberOfClass\Program.cs
        21 27 MemberOfClass

```

这就是索引器和属性的不同点，属性可以定义静态的属性。但是索引器必须是实例成员，它不能定义成静态的。

10.4 委托

委托是一个类型，委托类似于 C++ 中的函数指针，使用它可以调用函数。但是，C# 中的委托是面向对象的，并且它是类型安全的。C++ 中的函数指针只是指向了成员函数，但是委托同时封装了类的实例和方法。当创建委托实例的时候，创建的实例会包含一个调用列表，在调用列表中可以包含多个方法。每个方法称作一个调用实体。调用实体可以是静态方法，也可以是实例方法。如果是实例方法，则该调用实体包含调用该实例方法的实例。委托并不关心它所调用方法所属的类，它只关心被调用方法与委托的类型是否兼容。下面是代码实例：

```

001     using System;
002     namespace MemberOfClass1
003     {
004         public delegate void D(int a, int b);
005         public class Test
006         {
007             public D myDelegate;
008             public Test()
009             {

```

```

010         myDelegate = new D(Show1);
011     }
012     private static void Show1(int a, int b)
013     {
014         Console.WriteLine("方法 Show1 被调用, 两个实参相加的值是: {0}", a + b);
015     }
016     private void Show2(int a, int b)
017     {
018         Console.WriteLine("方法 Show2 被调用, 两个实参相加的值是: {0}", a + b);
019     }
020     private void Show3(int a, int b)
021     {
022         Console.WriteLine("方法 Show3 被调用, 两个实参相加的值是: {0}", a + b);
023     }
024 }
025 public class Program
026 {
027     static void Main(string[] args)
028     {
029         Test myT = new Test();
030         myT.myDelegate(33, 22);
031         Console.ReadKey();
032     }
033 }
034 }

```

这段代码演示的是最简单的一种委托形式。委托类型可以定义在类的外部, 也可以定义在类的内部。本段代码是定义在类的外部。第 4 行代码定义的就是一个委托类型, 委托类型的关键字是 `delegate`, 关键字前是委托类型的访问权限修饰符。关键字后是委托类型的返回类型, 这个返回类型规定与委托类型兼容的方法的返回类型必须与之相同。返回类型之后是委托类型的名称。接下来是形参列表, 它指定与委托类型兼容的方法的参数类型和个数必须与之相同。

第 7 行代码定义了一个委托类型的变量, 它是一个实例字段, 访问权限是 `public` 的。注意委托类型字段的访问权限一定要比委托类型的访问权限低或与委托类型的访问权限相同才可以。第 12 行、第 16 行和第 20 行代码定义了三个方法。其中第 12 行代码是一个静态方法。因为这段代码演示的是最简单的委托使用方法, 所以只使用了其中的静态方法。

在第 8 行的构造方法中, 实例化了委托类型的变量, 注意为委托变量的调用列表添加方法, 只需要向其构造方法中传递方法名称即可。这是为委托添加调用方法的最基本的一种方法。

第 29 行定义了 `Test` 类的一个实例, 然后第 30 行调用了类的委托成员。在调用委托成员的时候, 需要向其形参列表传递实参。这就是最基本的委托的使用方法。这段代码的执行结果如下:

```
方法 Show1 被调用, 两个实参相加的值是: 55
```

下面再介绍一种委托类型的使用方法, 实例代码如下:

```

001 using System;
002 namespace MemberOfClass1
003 {
004     public delegate void D(int a, int b);

```

```

005     public class Test
006     {
007         public static void Show1(int a, int b)
008         {
009             Console.WriteLine("方法 Show1 被调用, 两个实参相加的值是: {0}", a + b);
010         }
011         public void Show2(int a, int b)
012         {
013             Console.WriteLine("方法 Show2 被调用, 两个实参相加的值是: {0}", a + b);
014         }
015         public void Show3(int a, int b)
016         {
017             Console.WriteLine("方法 Show3 被调用, 两个实参相加的值是: {0}", a + b);
018         }
019     }
020     public class Program
021     {
022         static void Main(string[] args)
023         {
024             Test myT = new Test();
025             D myDelegate = new D(Test.Show1);
026             D myDelegate1 = new D(myT.Show2);
027             D myDelegate2 = new D(myT.Show3);
028             myDelegate(22, 33);
029             myDelegate1(33, 44);
030             myDelegate2(55, 66);
031             Console.ReadKey();
032         }
033     }
034 }

```

这段代码取消了类中的委托类型字段，而是将委托类型作为一个类来看待。在包含入口点方法的类中，首先第 24 行定义了 Test 类的一个变量并做了实例化。因为要向委托传递类的实例方法，所以必须有类的实例存在，才能引用类的实例方法。

第 25 行定义了一个委托类型的变量，并实例化，这里需要注意，因为委托并不是类中的一个成员了，所以向其构造方法传递静态方法的时候，需要以类名引用。

第 26 行也定义了一个委托类型的变量，在向其传递实例方法的时候，需要以类的实例来引用。

第 27 行代码的情况同第 26 行代码一样。在向委托传递方法的时候，需要传递方法名，而不需要方法的形参列表。

第 28 行到第 30 行是对委托的调用，这时要为其传递方法的实参。这段代码的执行结果如下：

```

方法 Show1 被调用, 两个实参相加的值是: 55
方法 Show2 被调用, 两个实参相加的值是: 77
方法 Show3 被调用, 两个实参相加的值是: 121

```

10.4.1 委托的访问修饰符

当委托位于类的外部时，可以使用的访问修饰符包括 public 和 internal。如果什么也不写，默认是

internal 的。当委托位于类的内部时,可以使用的访问修饰符包括 public、protected、internal、protected internal 和 private。下面是代码实例:

```
001 using System;
002 namespace MemberOfClass1
003 {
004     public class Test
005     {
006         protected delegate void D(int a, int b);
007         private delegate void D1(int a, int b);
008         protected internal delegate void D2(int a, int b);
009         internal delegate void D3(int a, int b);
010         private D myD;
011         private D1 myD1;
012         private D2 myD2;
013         private D3 myD3;
014         public Test()
015         {
016             myD = new D(Show1);
017             myD1 = new D1(Show1);
018             myD2 = new D2(Show1);
019             myD3 = new D3(Show1);
020         }
021         public static void Show1(int a, int b)
022         {
023             Console.WriteLine("方法 Show1 被调用, 两个实参相加的值是: {0}", a + b);
024         }
025         public void Show2(int a, int b)
026         {
027             Console.WriteLine("方法 Show2 被调用, 两个实参相加的值是: {0}", a + b);
028         }
029         public void Show3(int a, int b)
030         {
031             Console.WriteLine("方法 Show3 被调用, 两个实参相加的值是: {0}", a + b);
032         }
033         public void Use()
034         {
035             myD(11, 12);
036             myD1(22, 45);
037             myD2(55, 78);
038             myD3(345, 100);
039         }
040     }
041     class Test1 : Test
042     {
```

```

043         private D Test1D;
044         private D2 Test1D2;
045         private D3 Test1D3;
046         public Test1()
047         {
048             Test1D = new D(Test.Show1);
049             Test1D2 = new D2(Test.Show1);
050             Test1D3 = new D3(Test.Show1);
051         }
052         public void Use1()
053         {
054             Test1D(22, 45);
055             Test1D2(44, 45);
056             Test1D3(77, 78);
057         }
058     }
059     public class Program
060     {
061         static void Main(string[] args)
062         {
063             Test1 myT1 = new Test1();
064             myT1.Use();
065             myT1.Use1();
066             Console.ReadKey();
067         }
068     }
069 }

```

代码的第 6 行在类的内部定义了委托类型，它作为类的成员定义，访问权限是 `protected`，它可以被本类内部访问，也可以被派生类访问。代码的第 7 行定义的委托类型，访问权限是 `private` 的，它只可以被本类内部访问。代码的第 8 行定义的 `protected internal` 访问权限的委托类型，可以被本程序集访问，还可以被派生类访问，而不管派生类位于哪个程序集。第 9 行定义的委托类型是 `internal` 的，它只可以被本程序集访问。因为所有这几种委托类型都可以被本类内部访问，所以第 10 行到第 13 行定义了它们的变量。

第 14 行的实例构造方法中，对这四个委托类型的变量进行了实例化，并为它们的调用列表加入了方法 `Show1`。`Show1` 是一个静态方法，但是在类内部传入委托类型的构造方法时，不需要使用类名引用。

第 33 行定义了实例方法，在方法内部调用了这四个委托，并为其传入实参。

第 41 行代码又定义了一个类，它继承自基类 `Test`。因为基类中的委托类型只有 `D`、`D2` 和 `D3` 可以被派生类访问，所以第 43 行到第 45 行定义了它们的变量。注意，虽然它们和基类中的委托变量是同一种类型，但是它们是不同的委托。

在第 46 行的实例构造方法中，为这三个委托类型的变量创建实例，并为其调用列表加入方法，因为静态方法 `Show1` 也被派生类所继承，所以这里传入的方法名，可以使用类名引用，也可以不使用类名引用。

第 52 行定义了一个实例方法，方法内部调用了这三个委托，并为其传入实参。第 63 行定义了派生类的实例，然后调用实例方法 `Use` 和 `Use1`。这段代码的执行结果如下：

方法 `Show1` 被调用，两个实参相加的值是：23

方法 Show1 被调用，两个实参相加的值是：67
 方法 Show1 被调用，两个实参相加的值是：133
 方法 Show1 被调用，两个实参相加的值是：445
 方法 Show1 被调用，两个实参相加的值是：67
 方法 Show1 被调用，两个实参相加的值是：89
 方法 Show1 被调用，两个实参相加的值是：155

因为 D 和 D2 的访问权限被定义成了 protected 和 protected internal。所以下面来验证在其它程序中是否可以访问它们。首先要将本段代码中的包含 Main 方法的类去掉，然后在它的项目属性中将它改变为类库。接下来新建一个控制台项目，并物理上引用这个类库。控制台项目的代码如下：

```

001 using System;
002 using MemberOfClass1;
003 namespace MemberOfClass
004 {
005     class Program:Test
006     {
007         private D pD;
008         private D2 pD2;
009         public Program()
010         {
011             pD = new D(Show1);
012             pD2 = new D2(Show1);
013         }
014         public void Use3()
015         {
016             pD(34, 33);
017             pD2(12, 11);
018         }
019         static void Main(string[] args)
020         {
021             Program p = new Program();
022             p.Use3();
023             Console.ReadKey();
024         }
025     }
026 }
```

因为第 3 行代码的命名空间和类库的命名空间是两个独立的命名空间，它们的成员不位于同一个命名空间内。所以在一个命名空间内引用另一个命名空间的成员时，需要加上另一个命名空间的名称进行引用。为了代码编写的方便，第 2 行代码首先引用了类库的命名空间。

第 5 行代码定义了一个类，它继承自基类 Test。因为是派生类，所以对于委托类型 D 和 D2 都可以访问。第 7 行代码和第 8 行代码分别定义了 D 和 D2 的两个变量。

第 9 行的实例构造方法对这两个变量进行了实例化，并为其传入方法 Show1。因为 Show1 方法被继承了下来，所以这里不需要类名引用。第 14 行代码定义了一个实例方法，它的作用是调用这两个委托，并为其传入实参。第 21 行代码定义了本类的一个实例，并调用了实例方法 Use3。这段代码的执行结果如下：

方法 Show1 被调用，两个实参相加的值是：67

方法 Show1 被调用，两个实参相加的值是：23

类 Test 中的委托类型 D2 和 D3 都具有 internal 权限，现在来验证一下，对于一个同一程序集中的非派生类是否可以访问它们。首先将类库更改回控制台项目，然后增加一个类，这个类对于 Test 类来说是独立的。它们之间只是位于一个程序集内，彼此没有继承关系。代码如下：

```
001 using System;
002 namespace MemberOfClass1
003 {
004     public class Test
005     {
006         protected delegate void D(int a, int b);
007         private delegate void D1(int a, int b);
008         protected internal delegate void D2(int a, int b);
009         internal delegate void D3(int a, int b);
010         private D myD;
011         private D1 myD1;
012         private D2 myD2;
013         private D3 myD3;
014         public Test()
015         {
016             myD = new D(Show1);
017             myD1 = new D1(Show1);
018             myD2 = new D2(Show1);
019             myD3 = new D3(Show1);
020         }
021         public static void Show1(int a, int b)
022         {
023             Console.WriteLine("方法 Show1 被调用，两个实参相加的值是：{0}", a + b);
024         }
025         public void Show2(int a, int b)
026         {
027             Console.WriteLine("方法 Show2 被调用，两个实参相加的值是：{0}", a + b);
028         }
029         public void Show3(int a, int b)
030         {
031             Console.WriteLine("方法 Show3 被调用，两个实参相加的值是：{0}", a + b);
032         }
033         public void Use()
034         {
035             myD(11, 12);
036             myD1(22, 45);
037             myD2(55, 78);
038             myD3(345, 100);
039         }
040     }
```

```

041     class Test1
042     {
043         private Test.D2 tD2;
044         private Test.D3 tD3;
045         public Test1()
046         {
047             tD2 = new Test.D2(Test.Show1);
048             tD3 = new Test.D3(Test.Show1);
049         }
050         public void Use3()
051         {
052             tD2(34, 33);
053             tD3(22, 21);
054         }
055     }
056     public class Program
057     {
058         static void Main(string[] args)
059         {
060             Test1 myT1 = new Test1();
061             myT1.Use3();
062             Console.ReadKey();
063         }
064     }
065 }

```

这段代码中，原来的类 Test 没有进行修改。在第 41 行上，定义了一个类，它是一个相对于 Test 类来说独立的类。它们的关系仅限于同在一个程序集内。

第 43 行代码和第 44 行代码定义了委托类型 D2 和 D3 的两个变量。这里需要注意，因为这两个类不是继承关系，所以要引用 Test 类中的这两个委托类型需要使用 Test 类的类名进行引用。

第 45 行代码是实例构造方法，在构造方法中将委托实例化。实例化委托类型的时候，仍然需要使用类名引用委托类型名，传递的方法名也是如此。

第 50 行定义了一个实例方法，它调用了委托，并为其传入了实参。第 60 行代码定义了类 Test1 的一个实例，然后第 61 行调用类的实例方法。这段代码的执行结果如下：

方法 Show1 被调用，两个实参相加的值是：67

方法 Show1 被调用，两个实参相加的值是：43

10.4.2 委托类型的运算符

委托类型隐式提供了+运算符，因为+运算符是二元运算符，所以它的两个操作数都要是委托类型，并且两个操作数的委托类型要相同。下面是代码实例：

```

001     using System;
002     namespace MemberOfClass1
003     {
004         public delegate void D(int a, int b);
005         public class Test
006         {

```

```
007         public static void Show1(int a, int b)
008         {
009             Console.WriteLine("方法 Show1 被调用, 两个实参相加的值是: {0}", a + b);
010         }
011         public void Show2(int a, int b)
012         {
013             Console.WriteLine("方法 Show2 被调用, 两个实参相加的值是: {0}", a + b);
014         }
015         public void Show3(int a, int b)
016         {
017             Console.WriteLine("方法 Show3 被调用, 两个实参相加的值是: {0}", a + b);
018         }
019     }
020     public class Program
021     {
022         static void Main(string[] args)
023         {
024             Test t = new Test();
025             D d1 = new D(Test.Show1);
026             D d2 = new D(t.Show2);
027             D d3 = new D(t.Show3);
028             D d4 = d1 + d2;
029             D d5 = d4 + d3;
030             d5(22, 55);
031             Console.ReadKey();
032         }
033     }
034 }
```

代码的第 4 行定义了一个委托类型, 在 `Test` 类内没有定义委托类型的成员, 而是只定义了三个方法, 其中 `Show1` 是静态方法。下面重点看 `Main` 方法内部, 在代码的第 24 行定义了一个 `Test` 类型的实例, 使用它来引用类的实例方法。

第 25 行定义了一个委托类型的变量 `d1`, 为它添加的方法是 `Show1`。第 26 行定义了一个委托变量, 为它传入 `Show2`。第 27 行定义的委托变量传入的方法是 `Show3`。

第 28 行定义了一个新的委托变量, 使用 `+` 运算符将 `d1` 和 `d2` 组合在一起。第 29 行定义了一个新的委托变量, 它将第 28 行定义的委托变量和 `d3` 组合在一起。那么, 它们现在的调用逻辑顺序是: 如果调用 `d4`, 则 `d1` 中的方法将首先被调用, 然后调用 `d2` 中的方法。如果调用 `d5`, 则首先调用 `d1` 的方法, 然后是 `d2` 的方法, 接下来是 `d3` 的方法。

代码的第 30 行调用了委托 `d5`, 下面来看执行结果:

方法 Show1 被调用, 两个实参相加的值是: 77

方法 Show2 被调用, 两个实参相加的值是: 77

方法 Show3 被调用, 两个实参相加的值是: 77

使用运算符对委托进行组合时, 如果将两个委托组合成一个委托, 它将生成一个新的委托。例如第 28 行代码中的 `d4` 就是一个新的委托, 而组成它的 `d1` 和 `d2` 保持它们的调用列表不变。

对于委托类型, 还可以使用 `-` 运算符对委托进行分拆。实例代码如下:

```

001 static void Main(string[] args)
002 {
003     Test t = new Test();
004     D d1 = new D(Test.Show1);
005     D d2 = new D(t.Show2);
006     D d3 = new D(t.Show3);
007     D d4 = d1 + d2 + d3;
008     D d5 = d4 - d2;
009     d5(22, 55);
010     Console.ReadKey();
011 }

```

这段代码只对 Main 方法中的代码进行了修改，代码的第 7 行将委托 d1、d2 和 d3 进行了组合，然后使用 - 运算符从新创建的委托中移除了 d2。这时 d2 的调用列表不变。d5 的调用顺序依次为：d1 的方法和 d3 的方法。代码的执行结果如下：

方法 Show1 被调用，两个实参相加的值是：77

方法 Show3 被调用，两个实参相加的值是：77

对于委托类型还可以使用 += 和 -= 运算符，实例代码如下：

```

001 public class Program
002 {
003     static void Main(string[] args)
004     {
005         Test t = new Test();
006         D d1 = new D(Test.Show1);
007         D d2 = new D(t.Show2);
008         D d3 = new D(t.Show3);
009         D d4 = d1 + d2 + d3;
010         d1 += d2;
011         d1 += d3;
012         d1 += d4;
013         d1 -= d2;
014         d1(22, 55);
015         Console.ReadKey();
016     }
017 }

```

这段代码的第 10 行开始使用 += 运算符为 d1 的调用列表增加委托内容。第 13 行代码使用 -= 运算符从调用列表中移除委托内容。最后 d1 的调用列表中的调用顺序为：Show1、Show2、Show3、Show1、Show3。在移除调用列表中的内容时，是从调用列表的最后面开始移除。因此第 13 行代码移除 d2 时，后加入的 d4 中包含的 d2 被移除了。代码的执行结果如下：

方法 Show1 被调用，两个实参相加的值是：77

方法 Show2 被调用，两个实参相加的值是：77

方法 Show3 被调用，两个实参相加的值是：77

方法 Show1 被调用，两个实参相加的值是：77

方法 Show3 被调用，两个实参相加的值是：77

对于委托类型，还可以直接使用 = 运算符实例化委托，= 运算符直接跟方法名称。其它运算符也可以这

样操作。实例代码如下：

```

001 using System;
002 namespace MemberOfClass1
003 {
004     public delegate void D(int a, int b);
005     public class Test
006     {
007         public static void Show1(int a, int b)
008         {
009             Console.WriteLine("方法 Show1 被调用，两个实参相加的值是：{0}", a + b);
010         }
011         public void Show2(int a, int b)
012         {
013             Console.WriteLine("方法 Show2 被调用，两个实参相加的值是：{0}", a + b);
014         }
015         public void Show3(int a, int b)
016         {
017             Console.WriteLine("方法 Show3 被调用，两个实参相加的值是：{0}", a + b);
018         }
019     }
020     public class Program
021     {
022         static void Main(string[] args)
023         {
024             Test t = new Test();
025             D d1 = Test.Show1;
026             D d2 = d1 + t.Show2;
027             d2 += t.Show3;
028             D d3 = d2 - t.Show3;
029             d3(22, 55);
030             Console.ReadKey();
031         }
032     }
033 }

```

Test 类没有进行修改。在第 25 行定义了一个委托，它没有使用 new 关键字，而是直接使用赋值运算符将一个方法名赋值给了委托。第 26 行使用+运算符将一个委托和一个方法名相加操作。第 27 行使用+=运算符对一个方法名进行操作。第 28 行使用-运算符在一个委托中移除了一个方法名。最后执行 d3 委托。以上这些操作都是合法的，可以看到，委托的使用是很方便的。这段代码的执行结果如下：

方法 Show1 被调用，两个实参相加的值是：77

方法 Show2 被调用，两个实参相加的值是：77

但是一个没有初始化的委托变量不可以直接使用+=运算符将一个方法名添加进入调用列表，要使用+=运算符，最起码委托变量要为 null 或对它实例化才行（可以使用 new 关键字或=运算符）。代码实例如下：

```

001 public class Program
002 {

```

```

003     static void Main(string[] args)
004     {
005         D d1 += Test.Show1;
006         Console.ReadKey();
007     }
008 }

```

这段代码的第 5 行直接使用 += 运算符实例化一个委托，编译器会报错如下：

```

错误 1    无效的表达式项 “+=”      E:\书稿 new\书稿源码\第十章\MemberOfClass1\Program.cs    24
        18 MemberOfClass1

```

委托是允许添加进入调用列表的委托为空的。实例代码如下：

```

001     static void Main(string[] args)
002     {
003         Test t = new Test();
004         D d1 = Test.Show1;
005         D d2 = null;
006         D d3 = new D(t.Show2);
007         d1 += d2;
008         d1 += d3;
009         d1(33, 22);
010         Console.ReadKey();
011     }

```

在这段代码中，第 5 行代码定义的委托类型没有进行实例化，只是将 null 赋值给了它。第 7 行代码中，委托 d1 使用 += 运算符将它加入了调用列表。在第 9 行执行时，这个空的委托不会引起任何异常。代码的执行结果如下：

方法 Show1 被调用，两个实参相加的值是：55

方法 Show2 被调用，两个实参相加的值是：55

10.4.3 委托调用

委托在调用时，将按照调用列表中的先后顺序依次调用各个方法。如果委托调用中包含引用和输出参数，则调用列表中每个方法的调用都会使用对同一变量的引用。如果调用列表中前面的方法对变量做了更改，那么后面的方法将会见到这一变更。如果委托含有引用参数、输出参数或返回值，则它们的最终值就是调用列表中排在最后的方法所产生的结果。实例代码如下：

```

001     using System;
002     namespace MemberOfClass1
003     {
004         public delegate void D1(ref int a, out int b);
005         public delegate int D2(int a, int b);
006         public class Test
007         {
008             public void Show1(ref int a, out int b)
009             {
010                 a++;
011                 b = a + 32;
012                 Console.WriteLine("方法 Show1 被调用");
013             }

```

```
014         public void Show2(ref int a, out int b)
015         {
016             a++;
017             b = a + 32;
018             Console.WriteLine("方法 Show2 被调用");
019         }
020         public int Show3(int a, int b)
021         {
022             Console.WriteLine("方法 Show3 被调用");
023             return a + b;
024         }
025         public int Show4(int a, int b)
026         {
027             Console.WriteLine("方法 Show4 被调用");
028             return a + b + 100;
029         }
030     }
031     public class Program
032     {
033         static void Main(string[] args)
034         {
035             int a = 0;
036             int b;
037             Test t = new Test();
038             D1 d1 = new D1(t.Show1);
039             d1 += t.Show2;
040             D2 d2 = new D2(t.Show3);
041             d2 += t.Show4;
042             d1(ref a, out b);
043             Console.WriteLine("局部变量 a 的值是: {0}, 局部变量 b 的值是: {1}", a, b);
044             int c = d2(22, 33);
045             Console.WriteLine("调用带返回值的委托后, 返回值是: {0}", c);
046             Console.ReadKey();
047         }
048     }
049 }
```

这段代码中定义了两种委托类型，第 4 行代码定义的委托类型 D1 带有两种类型的参数，第一个是 ref 参数，第二个是 out 参数。它们都是 int 类型的。第 5 行代码定义的委托类型 D2 带有返回值，返回值是 int 类型的，它带有两个值形参，都是 int 类型的。

在类中定义了四个方法，前两个方法兼容委托类型 D1，后两个方法兼容委托类型 D2。下面重点看 Main 方法中的委托调用，在代码的第 35 行和第 36 行各定义了一个局部变量用来以引用参数和输出参数方式传递给委托。接下来第 38 行代码和 39 行代码将方法 Show1 和 Show2 加入委托 d1 的调用列表。第 40 行和第 41 行代码将方法 Show3 和 Show4 加入了委托 d2 的调用列表。

第 42 行调用了委托 d1，并将局部变量传递进去，第 43 行打印局部变量 a 和 b 的值。在调用 d1 的过

程中，按调用顺序，首先执行的是 Show1 方法，执行完毕，a 的值是 1，b 的值是 33。接下来执行 Show2 方法，因为委托中的 Show2 可以见到这一变更，所以 Show2 执行完毕后，a 的值是 2，b 的值是 34。

代码的第 44 行定义了一个 int 类型的变量 c 来接收委托 d2 调用后返回的结果。委托 d2 执行时，首先是 Show3 方法执行，它返回 55。然后执行 Show4 方法，它返回 155。因为带有返回值的委托调用时，返回的是最后一个方法的返回值，所以第 45 行将打印出 155。这段代码的执行结果如下：

```
方法 Show1 被调用
方法 Show2 被调用
局部变量 a 的值是：2，局部变量 b 的值是：34
方法 Show3 被调用
方法 Show4 被调用
调用带返回值的委托后，返回值是：155
```

最后需要注意的是，如果在委托调用的时候，委托忘记了实例化，这时编译器将会抛出异常。实例代码如下：

```
001 using System;
002 namespace MemberOfClass
003 {
004     public delegate void D(int a, int b);
005     class Program
006     {
007         public void Show1(int a, int b)
008         {
009             Console.WriteLine("参数相加的值是：{0}", a + b);
010         }
011         static void Main(string[] args)
012         {
013             Program p = new Program();
014             D myD = null;
015             myD(22, 33);
016             Console.ReadKey();
017         }
018     }
019 }
```

代码比较简单，不再做过多介绍。在代码的第 14 行定义了一个委托类型的变量，并将它赋值为 null。这时它就是一个空引用。当调用它的时候，忘记了对它实例化。直接在第 15 行为委托传值并调用它。这时编译器会抛出异常信息，如下：

```
未处理 NullReferenceException
未将对象引用设置到对象的实例。
```

10.4.4 委托的基类

委托类型是从 System.Delegate 类继承而来。但是 System.Delegate 类不是委托类型，它的作用只是用来派生出委托类型。委托类型隐式为密封的，所以不允许再从委托类型进行派生。虽然委托类型是从 System.Delegate 类派生而来，查阅帮助文档可以知道，System.Delegate 类是一个抽象类，抽象类后面会介绍到。一般来说，抽象类是允许继承的，因为它就是用来继承的，帮助文档中，System.Delegate 类定义如下：

```
public abstract class Delegate : ICloneable, ISerializable
```

abstract 关键字就是标明这个类是抽象类，类名后面的冒号后并不是基类，而是它要实现的接口。接口的概念以后也会介绍到。

当定义一个类从 System.Delegate 类继承时，编译器会报告错误，如下：

错误 1 “MemberOfClass.Program.A” 不能从特殊类 “System.Delegate” 派生 H:\书稿 new\书稿源码\第十章\MemberOfClass\Program.cs 24 15 MemberOfClass

从错误代码中可以知道，System.Delegate 类虽然是抽象类，但是它有其特殊性，它只为派生委托类型而用，不能用它派生出新类。

委托类型既然是从 System.Delegate 类派生出来的，它当然也可以使用 System.Delegate 类中的成员。实例代码如下：

```
001 using System;
002 namespace MemberOfClass
003 {
004     public delegate void D(int a, int b);
005     class Program
006     {
007         public void Show1(int a, int b)
008         {
009             Console.WriteLine("参数相加的值是: {0}", a + b);
010         }
011         public void Show2(int a, int b)
012         {
013             Console.WriteLine("参数相乘的值是: {0}", a * b);
014         }
015         static void Main(string[] args)
016         {
017             Program p = new Program();
018             D myD = p.Show1;
019             myD += p.Show2;
020             myD(22, 33);
021             Console.WriteLine("此委托当前调用的方法名称是: {0}", myD.Method.Name);
022             Console.WriteLine(myD.Method.ReturnType.FullName);
023             Console.ReadKey();
024         }
025     }
026 }
```

这段代码的第 21 行，委托调用了 System.Delegate 中的属性成员返回当前执行的方法的名称。第 22 行调用了 System.Delegate 中的属性得到当前执行方法的返回值的完全限定名称。代码的执行结果如下：

参数相加的值是：55

参数相乘的值是：726

此委托当前调用的方法名称是：Show2

System.Void

在 .Net 类库中，System.Delegate 类还有一个派生类 MulticastDelegate，它是多播委托类。也就是说，它的调用列表中可以包含多个方法。当创建一个委托时，实际上，这个委托是由 MulticastDelegate 类直接派生的。所以，在继承链路上依次是 System.Delegate、System.MulticastDelegate，最后是委托

类型。System.MulticastDelegate 类也是一个抽象类，但是不允许从它派生其它类。它也是为派生委托而定义。

10.5 事件

与委托不同，委托可以看做是一个类。它可以定义在类的外部，也可以定义在类的内部。而事件是类的一种成员，它和字段类似。事件的定义让类或实例能够提供通知，通俗地讲，就是在某个时刻让类或类的实例能够通过事件来调用某个或某些方法。事件是对委托的封装，实际上，事件可以看做就是封装的委托。当说到“引发一个事件”时，可能感觉语言很晦涩，不好理解。实际上，它和“调用一个由该事件表示的委托”意思是一样的。既然，事件和委托背后都调用的是委托，那么为什么还需要事件呢？下面通过代码实例来介绍。代码如下：

```
001 using System;
002 namespace MemberOfClass
003 {
004     public delegate void D(int a,int b);
005     class Test
006     {
007         public event D myD;
008         public D myDelegate;
009         public Test()
010         {
011             myD = Show1;
012             myDelegate = Show2;
013         }
014         public void Show1(int a, int b)
015         {
016             Console.WriteLine("参数相加的值是: {0}", a + b);
017         }
018         public void Show2(int a, int b)
019         {
020             Console.WriteLine("参数相乘的值是: {0}", a * b);
021         }
022     }
023     class Program
024     {
025         static void Main(string[] args)
026         {
027             Test p = new Test();
028             p.myDelegate(22, 33);
029             p.myDelegate = p.Show1;
030             p.myDelegate(22, 33);
031             Console.ReadKey();
032         }
033     }
034 }
035 }
```

事件是基于委托的。要定义一个事件，先要有委托。第 4 行代码定义了一个委托类型。第 7 行代码就是定义的一个事件。事件的定义使用 `event` 关键字。关键字前面是访问权限修饰符，后面是委托的类型，然后接事件的变量名称。

这段代码主要演示委托的使用，所以第 8 行定义了一个委托。第 9 行的构造方法中对事件和委托都进行了实例化。事件的实例化方式和委托是一样的，本段代码就是使用赋值运算符进行的实例化。

在后面的代码中没有演示事件的使用，而是演示的委托的一种使用方法。第 29 行代码调用了委托，并为其传入了两个参数。注意第 30 行，因为定义的委托是 `public` 的，所以可以在类外将类的委托成员的调用列表进行随意改变和随意调用。第 30 行将委托成员指向了另一个方法，并在下一行代码中进行了调用。下面修改代码，在类外调用一下事件试一试，事件传递参数的方法和委托是相同的。修改的代码如下：

```
001 using System;
002 namespace MemberOfClass
003 {
004     public delegate void D(int a,int b);
005     class Test
006     {
007         public event D myD;
008         public D myDelegate;
009         public Test()
010         {
011             myD = Show1;
012             myDelegate = Show2;
013         }
014         public void Show1(int a, int b)
015         {
016             Console.WriteLine("参数相加的值是: {0}", a + b);
017         }
018         public void Show2(int a, int b)
019         {
020             Console.WriteLine("参数相乘的值是: {0}", a * b);
021         }
022     }
023     class Program
024     {
025         static void Main(string[] args)
026         {
027             Test p = new Test();
028             p.myDelegate(22, 33);
029             p.myDelegate = p.Show1;
030             p.myDelegate(22, 33);
031             p.myD = p.Show2;
032             p.myD(33, 44);
033             Console.ReadKey();
034         }
035     }
```

```
036     }
037 }
```

代码中并没有对类 Test 进行修改,但是在第 32 行调用了类的事件成员,将它也指向了另一个方法。然后第 33 行代码传入参数调用了事件。现在看来,这么做好像是没有问题的,因为委托就是这么调用的。但是编译器会报错如下:

错误 1 事件“MemberOfClass.Test.myD”只能出现在 += 或 -= 的左边(从类型“MemberOfClass.Test”中使用时除外) E:\书稿 new\书稿源码\第十章\MemberOfClass\Program.cs 32 15

MemberOfClass

错误 2 事件“MemberOfClass.Test.myD”只能出现在 += 或 -= 的左边(从类型“MemberOfClass.Test”中使用时除外) E:\书稿 new\书稿源码\第十章\MemberOfClass\Program.cs 33 15

MemberOfClass

出现这两个错误的原因是,委托的调用执行只能在它所在的类内执行。因为事件的作用就是类或实例实现在一定条件下执行方法的方式,它是给类或实例实现通知使用的,而不是用来在类外随便调用的。在类外(包括它的派生类),只能使用+=或-=从它的调用列表中添加或移除方法,而不能将它重新实例化。下面再修改代码如下:

```
001 using System;
002 namespace MemberOfClass
003 {
004     public delegate void D(int a,int b);
005     class Test
006     {
007         private event D myD;
008         public Test()
009         {
010             myD = new D(this.Show1);
011             myD += this.Show2;
012         }
013         public void Show1(int a, int b)
014         {
015             Console.WriteLine("参数相加的值是: {0}", a + b);
016         }
017         public void Show2(int a, int b)
018         {
019             Console.WriteLine("参数相乘的值是: {0}", a * b);
020         }
021         public void Count()
022         {
023             for (int i = 0; i < 5; i++)
024             {
025                 for (int j = 0; j < 5; j++)
026                 {
027                     Console.WriteLine("目前 i 的值是: {0}, j 的值是: {1}", i, j);
028                     if (i == 3 && j == 3)
029                     {
```

```
030             Console.WriteLine("符合条件，事件被触发");
031             myD(i, j);
032         }
033     }
034 }
035 }
036 }
037 class Program
038 {
039     static void Main(string[] args)
040     {
041         Test p = new Test();
042         p.Count();
043         Console.ReadKey();
044     }
045 }
046 }
```

代码的第 4 行定义了一个委托。第 7 行定义了一个事件。因为事件一般是由类内部进行处理，所以将它的访问权限设置为 `private` 的。第 8 行代码开始的构造方法为事件添加方法列表。第 10 行和第 11 行采用了两种方法为事件添加方法。第一种是用构造方法的方式，第二种采用 `+=` 运算符为事件添加方法。可以看到，事件的实例化和添加处理方式的方式是和委托一样的。如果事件还没有实例化，那么它是不能直接使用 `+=` 运算符的。

第 21 行代码开始是一个实例方法，在它内部使用了嵌套的 `for` 循环语句。当 `i` 和 `j` 同时为 3 的时候，调用类的事件。这就成为事件的触发。触发后，事件会调用为它添加的两个方法，这也称为“回调”。这段代码的执行结果如下：

```
目前 i 的值是：0，j 的值是：0
目前 i 的值是：0，j 的值是：1
目前 i 的值是：0，j 的值是：2
目前 i 的值是：0，j 的值是：3
目前 i 的值是：0，j 的值是：4
目前 i 的值是：1，j 的值是：0
目前 i 的值是：1，j 的值是：1
目前 i 的值是：1，j 的值是：2
目前 i 的值是：1，j 的值是：3
目前 i 的值是：1，j 的值是：4
目前 i 的值是：2，j 的值是：0
目前 i 的值是：2，j 的值是：1
目前 i 的值是：2，j 的值是：2
目前 i 的值是：2，j 的值是：3
目前 i 的值是：2，j 的值是：4
目前 i 的值是：3，j 的值是：0
目前 i 的值是：3，j 的值是：1
目前 i 的值是：3，j 的值是：2
目前 i 的值是：3，j 的值是：3
```

符合条件，事件被触发

参数相加的值是：6

参数相乘的值是：9

目前 i 的值是：3，j 的值是：4

目前 i 的值是：4，j 的值是：0

目前 i 的值是：4，j 的值是：1

目前 i 的值是：4，j 的值是：2

目前 i 的值是：4，j 的值是：3

目前 i 的值是：4，j 的值是：4

10.5.1 静态事件和实例事件

如果一个事件用 `static` 关键字修饰，则称这个事件为静态事件，它是属于类的。如果不包含 `static` 修饰关键字，则这个事件是实例事件，它是属于类的具体实例的。实例代码如下：

```

001     using System;
002     namespace MemberOfClass
003     {
004         public delegate void D(int a, int b);
005         class Test
006         {
007             private event D myD;
008             private static event D myD1;
009             public Test()
010             {
011                 myD = new D(this.Show1);
012                 myD += this.Show2;
013             }
014             static Test()
015             {
016                 myD1 = Sub;
017             }
018             public static void Sub(int a, int b)
019             {
020                 Console.WriteLine("参数相减的值是：{0}", a - b);
021             }
022             public void Show1(int a, int b)
023             {
024                 Console.WriteLine("参数相加的值是：{0}", a + b);
025             }
026             public void Show2(int a, int b)
027             {
028                 Console.WriteLine("参数相乘的值是：{0}", a * b);
029             }
030             public void Count()
031             {
032                 for (int i = 0; i < 5; i++)

```

```
033         {
034             for (int j = 0; j < 5; j++)
035             {
036                 Console.WriteLine("目前 i 的值是: {0}, j 的值是: {1}", i, j);
037                 if (i == 3 && j == 2)
038                 {
039                     Console.WriteLine("符合条件, 事件被触发");
040                     myD(i, j);
041                     myD1(i, j);
042                 }
043             }
044         }
045     }
046 }
047 class Program
048 {
049     static void Main(string[] args)
050     {
051         Test p = new Test();
052         p.Count();
053         Console.ReadKey();
054     }
055 }
056 }
```

代码的第 7 行和第 8 行各定义了一个实例事件和一个静态事件。代码的第 14 行是一个静态构造方法，它对静态事件进行实例化。代码的第 41 行对静态事件进行了调用。代码的执行结果如下：

```
目前 i 的值是: 0, j 的值是: 0
目前 i 的值是: 0, j 的值是: 1
目前 i 的值是: 0, j 的值是: 2
目前 i 的值是: 0, j 的值是: 3
目前 i 的值是: 0, j 的值是: 4
目前 i 的值是: 1, j 的值是: 0
目前 i 的值是: 1, j 的值是: 1
目前 i 的值是: 1, j 的值是: 2
目前 i 的值是: 1, j 的值是: 3
目前 i 的值是: 1, j 的值是: 4
目前 i 的值是: 2, j 的值是: 0
目前 i 的值是: 2, j 的值是: 1
目前 i 的值是: 2, j 的值是: 2
目前 i 的值是: 2, j 的值是: 3
目前 i 的值是: 2, j 的值是: 4
目前 i 的值是: 3, j 的值是: 0
目前 i 的值是: 3, j 的值是: 1
目前 i 的值是: 3, j 的值是: 2
```


符合条件，事件被触发

参数相加的值是：5

参数相乘的值是：6

参数相减的值是：1

目前 i 的值是：3，j 的值是：3

目前 i 的值是：3，j 的值是：4

目前 i 的值是：4，j 的值是：0

目前 i 的值是：4，j 的值是：1

目前 i 的值是：4，j 的值是：2

目前 i 的值是：4，j 的值是：3

目前 i 的值是：4，j 的值是：4

10.5.2 事件访问器

事件如同属性一样，包含访问器。事件访问器有两种，一个是添加访问器；另一个是移除访问器。当使用事件时，编译器会自动生成这两种访问器。但是在一些特殊情况下，需要人工编写这两种访问器。实例代码如下：

```
001     using System;
002     namespace MemberOfClass
003     {
004         public delegate void D(int a, int b);
005         class Test
006         {
007             private D _myD;
008             public event D myD
009             {
010                 add
011                 {
012                     _myD += value;
013                 }
014                 remove
015                 {
016                     _myD -= value;
017                 }
018             }
019             public Test()
020             {
021                 myD += this.Show1;
022                 myD += this.Show2;
023                 myD += this.Show1;
024                 myD -= this.Show1;
025             }
026             public void Show1(int a, int b)
027             {
028                 Console.WriteLine("参数相加的值是：{0}", a + b);
029             }
030         }
031     }
```

```

030     public void Show2(int a, int b)
031     {
032         Console.WriteLine("参数相乘的值是: {0}", a * b);
033     }
034     public void Count()
035     {
036         for (int i = 0; i < 5; i++)
037         {
038             for (int j = 0; j < 5; j++)
039             {
040                 Console.WriteLine("目前 i 的值是: {0}, j 的值是: {1}", i, j);
041                 if (i == 3 && j == 2)
042                 {
043                     Console.WriteLine("符合条件, 事件被触发");
044                     _myD(i, j);
045                 }
046             }
047         }
048     }
049 }
050 class Program
051 {
052     static void Main(string[] args)
053     {
054         Test p = new Test();
055         p.Count();
056         Console.ReadKey();
057     }
058 }
059 }

```

这段代码演示了事件访问器的编写方法。使用事件访问器的前提是有一个委托类型的变量存在。第 7 行代码定义了一个委托类型的变量。第 8 行代码定义了一个带有访问器的事件，和前面介绍过的事件不同的是，它带有一对大括号组成的代码块。在代码块中编写访问器，add 访问器对应+=操作，其实就是第 12 行代码对委托字段的添加方法操作。为何可以直接对委托变量进行+=操作呢？因为在实例化类的时候，委托变量因为是类的成员，所以默认值为 null。在事件访问器中带有一个隐式形参 value，它对应的就是传入的方法名或委托。第 14 行开始是移除访问器，它实际上就是使用-=运算符对委托字段进行操作。注意在访问器中因为隐式有一个 value 参数，所以不能再自定义同名的 value 参数。第 44 行调用的时候，也是调用的委托字段。这段代码的执行结果如下：

```

目前 i 的值是: 0, j 的值是: 0
目前 i 的值是: 0, j 的值是: 1
目前 i 的值是: 0, j 的值是: 2
目前 i 的值是: 0, j 的值是: 3
目前 i 的值是: 0, j 的值是: 4
目前 i 的值是: 1, j 的值是: 0

```

```

目前 i 的值是: 1, j 的值是: 1
目前 i 的值是: 1, j 的值是: 2
目前 i 的值是: 1, j 的值是: 3
目前 i 的值是: 1, j 的值是: 4
目前 i 的值是: 2, j 的值是: 0
目前 i 的值是: 2, j 的值是: 1
目前 i 的值是: 2, j 的值是: 2
目前 i 的值是: 2, j 的值是: 3
目前 i 的值是: 2, j 的值是: 4
目前 i 的值是: 3, j 的值是: 0
目前 i 的值是: 3, j 的值是: 1
目前 i 的值是: 3, j 的值是: 2
符合条件, 事件被触发
参数相加的值是: 5
参数相乘的值是: 6
目前 i 的值是: 3, j 的值是: 3
目前 i 的值是: 3, j 的值是: 4
目前 i 的值是: 4, j 的值是: 0
目前 i 的值是: 4, j 的值是: 1
目前 i 的值是: 4, j 的值是: 2
目前 i 的值是: 4, j 的值是: 3
目前 i 的值是: 4, j 的值是: 4

```

但是, 这样定义事件访问器之后, 就不可以再使用 new 关键字或=操作符对事件进行实例化。

10.5.3 事件的访问权限

和委托不同, 事件是类的成员。所以事件可以使用的访问修饰符包括 public、protected、internal、protected internal 和 private。它们都是用在类内部的成员访问修饰符。下面是代码实例:

```

001    using System;
002    namespace MemberOfClass
003    {
004        public delegate void D(int a, int b);
005        class Test
006        {
007            private event D myD1;
008            protected event D myD2;
009            protected internal event D myD3;
010            public Test()
011            {
012                myD1 += this.Show1;
013                myD1 += this.Show2;
014                myD2 += this.Show1;
015                myD3 += this.Show2;
016            }
017            public void Show1(int a, int b)
018            {

```

```
019         Console.WriteLine("参数相加的值是: {0}", a + b);
020     }
021     public void Show2(int a, int b)
022     {
023         Console.WriteLine("参数相乘的值是: {0}", a * b);
024     }
025     public void Count()
026     {
027         int j = 12;
028         for (int i = 0; i < 5; i++)
029         {
030             Console.WriteLine("目前 i 的值是: {0}", i);
031             if (i == 3)
032             {
033                 Console.WriteLine("符合条件, 事件被触发");
034                 myD1(i, j);
035                 myD2(i, j);
036                 myD3(i, j);
037             }
038         }
039     }
040 }
041 class Test1 : Test
042 {
043     public void Show3(int a, int b)
044     {
045         Console.WriteLine("参数相加的值再乘 2 的值是: {0}", (a + b)*2);
046     }
047     public void AddMethod()
048     {
049         myD2 += this.Show3;
050     }
051 }
052 class Test2
053 {
054     private Test t = new Test();
055     public void Show4(int a, int b)
056     {
057         Console.WriteLine("参数相加的值再除以 2 的值是: {0}", (a + b) / 2);
058     }
059     public void AddMethod()
060     {
061         t.myD3 += this.Show4;
062     }
```

```

063         public void Invoke()
064         {
065             t.Count();
066         }
067     }
068     class Program
069     {
070         static void Main(string[] args)
071         {
072             Test1 t1 = new Test1();
073             t1.AddMethod();
074             t1.Count();
075             Test2 t2 = new Test2();
076             t2.AddMethod();
077             t2.Invoke();
078             Console.ReadKey();
079         }
080     }
081 }

```

这段代码的第 7 行定义了一个事件，它的访问权限是 `private` 的，它只能被本类访问。第 8 行定义的事件 `myD2`，它的访问权限是 `protected` 的，它能被本类和它的派生类访问。第 9 行定义的事件 `myD3`，它的访问权限是 `protected internal` 的，它能被它的派生类和本程序集中的其它类访问。第 10 行的构造方法中对它们添加方法列表。第 25 行的实例方法使用 `for` 循环创建一个触发事件的条件，第 34 行开始的三行代码调用事件中的方法。

第 41 行代码定义了一个 `Test1`，它继承自类 `Test`。第 47 行的实例方法对事件 `myD2` 又添加了本类的一个实例方法。对于事件来说，如果在派生类中使用 `myD2(i, j)` 这样的调用方法也是无法访问事件的。事件的调用只能在本类中执行。在派生类中所谓的访问事件，也只是使用 `+=` 运算符或 `-=` 运算符从事件的调用列表中添加方法或移除方法。

第 52 行定义了一个类 `Test2`，它在本程序集中是一个独立的类，它只派生于类 `Object`。在这个独立的类中，它只能访问类 `Test` 的 `myD3` 事件。因为前面定义的事件是类的实例成员，所以在类 `Test2` 中需要包含一个类 `Test` 的实例才能访问其 `myD3` 事件。第 54 行代码定义了一个类 `Test` 的对象，第 55 行定义了一个实例方法，第 61 行代码将这个新定义的方法加入到了事件 `myD3` 中。第 63 行代码定义的方法又调用了实例 `t` 的 `Count` 方法。

第 72 行定义了 `Test1` 类的一个实例，第 73 行代码调用了它的 `AddMethod` 方法将 `Test1` 类中定义的一个实例方法也加入到了调用列表中。第 74 行代码调用实例继承来的 `Count` 方法。

第 75 行代码定义了 `Test2` 这个独立的类的一个实例，然后调用了其 `AddMethod` 方法向事件添加方法。注意因为是包含关系，`Test2` 类的实例中包含的 `Test` 类的成员和 `Test1` 类创建时包含的 `Test` 类的成员是同一个类的两个副本，它们之间没有什么关系。第 77 行代码调用了 `Invoke` 方法，这个方法又间接调用了包含类的 `Count` 方法。这段代码的执行结果如下：

```

目前 i 的值是：0
目前 i 的值是：1
目前 i 的值是：2
目前 i 的值是：3
符合条件，事件被触发

```

```
参数相加的值是：15
参数相乘的值是：36
参数相加的值是：15
参数相加的值再乘 2 的值是：30
参数相乘的值是：36
目前 i 的值是：4
目前 i 的值是：0
目前 i 的值是：1
目前 i 的值是：2
目前 i 的值是：3
符合条件，事件被触发
参数相加的值是：15
参数相乘的值是：36
参数相加的值是：15
参数相乘的值是：36
参数相加的值再除以 2 的值是：7
目前 i 的值是：4
```

10.6 对象初始化器

对象初始化器语法的作用是让类的实例的生成更加自由。它的语法也很简单，下面通过一个代码实例来说明：

```
001    using System;
002    namespace 对象初始化器
003    {
004        class Program
005        {
006            private int x;
007            private int y;
008            public Program()
009            {
010            }
011            public Program(int a, int b)
012            {
013                x = a;
014                y = b;
015            }
016            public void Show()
017            {
018                Console.WriteLine("实例字段 x 的值是：{0}", x);
019                Console.WriteLine("实例字段 y 的值是：{0}", y);
020                Console.WriteLine();
021            }
022            static void Main(string[] args)
023            {
024                //隐式调用无参构造方法
```

```
025         Program myP1 = new Program { x = 11, y = 12 };
026         myP1.Show();
027         //显式调用无参构造方法
028         Program myP2 = new Program() { x = 13, y = 14 };
029         myP2.Show();
030         //调用带参数构造方法
031         Program myP3 = new Program(15, 16) { x = 17, y = 18 };
032         myP3.Show();
033         Console.ReadKey();
034     }
035 }
036 }
```

对象初始化器的语法就是在通常定义实例的语法上先不以分号结束语句，后跟一个大括号作为方法体，在里面写入对实例字段重新赋值的语句，然后再以分号结束语句。如第 28 行代码所示。在调用类的默认构造方法或无参构造方法的时候，可以省略类名后的参数列表小括号对。这称为隐式调用无参构造方法。如第 25 行代码所示。对象初始化语法可以让实例的构造更加灵活，也可以弥补现有构造方法的不足。这段代码的执行结果如下：

实例字段 x 的值是：11

实例字段 y 的值是：12

实例字段 x 的值是：13

实例字段 y 的值是：14

实例字段 x 的值是：17

实例字段 y 的值是：18

第十一章 多态

多态是面向对象的三个特征：封装、继承、多态中的一个特性。多态建立在继承的基础之上，先有继承，后有多态。多态的具体表现就是，派生类在继承父类的成员之后，派生类的成员可以重写基类的成员。从而达到，基类的同一成员在派生类中表现出不同的形态。下面开始介绍 C# 在面向对象的多态性方面提供的各种技术。

11.1 隐式引用转换

隐式引用转换是引用类型之间的转换。隐式引用转换总能成功，因此不需要在运行时进行检查。下面是代码实例：

```
001 using System;
002 namespace Polymorphism
003 {
004     class A
005     {
006         public void Show()
007         {
008             Console.WriteLine("正在调用 A 类中的实例方法 Show()");
009         }
010     }
011     class Program
012     {
013         static void Main(string[] args)
014         {
015             A myA = new A();
016             object o = myA;
017             Console.WriteLine(o.ToString());
018             Console.ReadKey();
019         }
020     }
021 }
```

代码的第 4 行定义了一个类，在类中定义了一个实例方法。第 15 行定义了一个这个类的实例。第 16 行定义了一个 object 类型的变量，然后将 myA 赋值给它。因为用户自定义的类都是从 object 类型派生。所以这是隐式引用转换的一种，父类的引用可以指向子类的对象。当第 17 行调用 object 类型引用 o 的成员时，它只能调用 object 类型的成员，而不能调用其指向的实例的类的成员。这段代码的执行结果如下：

Polymorphism.A

在 o 能调用的方法列表中不能看见实例方法 Show()。虽然它指向的真正实例是一个 A 类型的实例。下面将代码进行修改，如下：

```
001 using System;
002 namespace Polymorphism
003 {
004     class A
005     {
006         public void Show()
007         {
```



```

008         Console.WriteLine("正在调用 A 类中的实例方法 Show()");
009     }
010 }
011 class B : A
012 {
013     public void Show1()
014     {
015         Console.WriteLine("正在调用 B 类中的实例方法 Show()");
016     }
017 }
018 class Program
019 {
020     static void Main(string[] args)
021     {
022         A myA = new A();
023         B myB = new B();
024         myA = myB;
025         A myA1 = new B();
026         myA.Show();
027         myB.Show();
028         myB.Show1();
029         myA1.Show();
030         Console.ReadKey();
031     }
032 }
033 }

```

在这个实例中，第 4 行定义了一个基类 A，在基类 A 里面定义了一个实例方法。第 11 行定义了一个派生类 B，它继承自类 A。在派生类里面，第 13 行定义了一个实例方法 Show1()。

接下来第 22 行创建了一个类 A 的实例，第 23 行创建了一个类 B 的实例。第 24 行将类 B 的实例赋值给了 A 类型的变量 myA。这时，原来的 myA 引用的实例再没有引用变量指向它，它将在一个合适的时间里被回收。而第 24 行代码却表达了这样一个意思，同前面 object 类型的变量能够指向它的派生类之外，自定义的引用类型之间也能发生这样的操作，即基类的引用指向派生类的实例。它们之间发生的也是隐式引用转换。

第 25 行代码表达的是一种更为确切的意思，即直接可以定义一个基类的变量，然后为它创建派生类的实例。这也是一种隐式引用转换。但是，虽然发生了隐式引用转换，myA 虽然指向了 B 类型的实例，myA 却只能调用 A 类的实例成员，它不能调用它的派生类的成员方法。第 23 行定义的 B 类型的实例能调用 A 类的成员方法，也能调用 B 类的成员方法，因为这是继承的关系。A 类的引用变量 myA1 也只能调用 A 类的成员方法。这段代码的执行结果如下：

```

正在调用 A 类中的实例方法 Show()
正在调用 A 类中的实例方法 Show()
正在调用 B 类中的实例方法 Show()
正在调用 A 类中的实例方法 Show()

```

隐式引用转换，虽然基类的变量引用了派生类的实例，但是它并不能改变派生类实例的类型，派生类的实例仍旧是派生类的类型。这可以通过第一个实例的 object 类型的变量 o 最后调用 ToString() 方法打印出

了派生类的类型名称的作用可以看出。

其它隐式引用转换还有从一个类到它实现的接口之间、从接口到它的基接口之间、前面介绍过的数组到 System.Array 类之间、委托到 System.Delegate 类之间。接口的概念后面的章节会介绍到。要实现多态，需要这样的隐式引用转换。

11.2 虚方法和重写方法

使用 virtual 关键字修饰的方法就是虚方法，使用 override 关键字修饰的方法是重写方法。使用虚方法和重写方法可以实现多态。下面通过代码实例来说明：

```
001    using System;
002    namespace Polymorphism
003    {
004        class A
005        {
006            public virtual void Show()
007            {
008                Console.WriteLine("正在调用 A 类中的实例方法 Show()");
009            }
010        }
011        class B : A
012        {
013            public override void Show()
014            {
015                Console.WriteLine("正在调用 B 类中的实例方法 Show()");
016            }
017        }
018        class Program
019        {
020            static void Main(string[] args)
021            {
022                A myA = new B();
023                myA.Show();
024                Console.ReadKey();
025            }
026        }
027    }
```

这段代码演示了最简单的虚方法和重写方法的应用。第 6 行代码定义了一个虚方法，它在访问修饰符之后和方法的返回类型之间使用了一个 virtual 关键字。第 13 行代码定义了一个重写方法，它在访问修饰符之后和返回类型之间使用了 override 关键字。

第 22 行定义了一个基类的引用，它指向了派生类的实例。这时，称这个实例的编译时类型是 A 类型，运行时类型是 B 类型。在使用虚方法和重写方法时，具体执行哪个方法取决于实例的运行时类型。第 23 行代码调用了 myA 的 Show 方法。这时首先看一下执行结果，如下：

```
正在调用 B 类中的实例方法 Show()
```

可以看到，这时虽然编译时类型是 A 类型，但是调用的却不是 A 类型的实例方法，而是根据实例的运行时类型调用了 B 类型的方法。这就是多态的实现。

11.3 方法隐藏和 new 关键字

如果在派生类中的同名方法中没有使用 `override` 关键字会怎么样呢？下面通过代码实例来进行实验。代码如下：

```

001    using System;
002    namespace Polymorphism
003    {
004        class A
005        {
006            public virtual void Show()
007            {
008                Console.WriteLine("正在调用 A 类中的实例方法 Show()");
009            }
010        }
011        class B : A
012        {
013            public void Show()
014            {
015                Console.WriteLine("正在调用 B 类中的实例方法 Show()");
016            }
017        }
018        class Program
019        {
020            static void Main(string[] args)
021            {
022                A myA = new B();
023                myA.Show();
024                Console.ReadKey();
025            }
026        }
027    }

```

这段代码对前面的代码进行了修改，第 6 行的 `virtual` 关键字没有去掉，但是第 13 行的 `override` 关键字去掉了。这时执行代码，还会调用的是 B 类中的方法吗？代码的执行结果如下：

正在调用 A 类中的实例方法 Show()

这时，可以看到，调用的就是 A 类中的成员方法了。而且这时编译器会报告一个警告信息，如下：

警告 1 “Polymorphism.B.Show()” 将隐藏继承的成员 “Polymorphism.A.Show()”。若要使当前成员重写该实现，请添加关键字 `override`。否则，添加关键字 `new`。

现在来阐述一下在继承过程中的方法调用规则，以这段代码为例，对于第 22 行代码来说，在这段代码中存在下列实体，实例的编译时类型为 A，实例的运行时类型为 B，调用的方法为 Show()。当第 6 行代码中定义的方法为虚方法时，则对于运行时类型 B 而言，A 类中的方法 Show 就是派生程度最大的类，它将被调用。如果派生类中使用了 `override` 关键字，则这个方法对于运行时类型 B 来说，它是派生程度最大的类，它将被调用。

如果基类中的方法没用 `virtual` 关键字修饰，则这个方法将被调用。这时，完全由编译时类型决定调用的方法。代码如下：

```

001    using System;
002    namespace Polymorphism

```

```

003  {
004      class A
005      {
006          public void Show()
007          {
008              Console.WriteLine("正在调用 A 类中的实例方法 Show()");
009          }
010      }
011      class B : A
012      {
013          public void Show()
014          {
015              Console.WriteLine("正在调用 B 类中的实例方法 Show()");
016          }
017      }
018      class Program
019      {
020          static void Main(string[] args)
021          {
022              A myA = new B();
023              myA.Show();
024              B myB = new B();
025              myB.Show();
026              Console.ReadKey();
027          }
028      }
029  }

```

这段代码去掉了 virtual 关键字，这时 A 类中的成员方法将被调用，并且还会提示如下警告信息：

警告 1 “Polymorphism.B.Show()” 隐藏了继承的成员 “Polymorphism.A.Show()”。如果是有意隐藏，请使用关键字 new。

这时，A 类中的方法将被 B 类中的同名方法隐藏掉，myA 调用 Show 方法的时候，将会按照编译时类型调用 A 类中的实例方法。而第 25 行的 B 类的实例调用 Show 方法的时候，因为 B 从 A 继承，那么 A 类的实例方法也被继承下来，这时 B 的实例中包含了两个同名方法 Show，只不过 A 类的实例方法被隐藏掉了，B 类中的实例方法将被调用。代码的执行结果如下：

正在调用 A 类中的实例方法 Show()

正在调用 B 类中的实例方法 Show()

如果想去掉这个警告，也就是说，程序员明知道方法重名的情况下，想要有意隐藏掉 A 类的同名方法。这时就可以使用 new 关键字。代码实例如下：

```

001  using System;
002  namespace Polymorphism1
003  {
004      class A
005      {
006          public void Show()

```

```
007     {
008         Console.WriteLine("正在调用 A 类中的实例方法 Show()");
009     }
010 }
011 class B : A
012 {
013     public new void Show()
014     {
015         Console.WriteLine("正在调用 B 类中的实例方法 Show()");
016     }
017 }
018 class Program
019 {
020     static void Main(string[] args)
021     {
022         B myB = new B();
023         myB.Show();
024         Console.ReadKey();
025     }
026 }
027 }
```

第 13 行代码使用了 new 关键字隐藏了基类继承来的方法。这样，在类外只能访问 B 中的实例方法。对于基类中的同名方法是无法访问的。如果要访问它，只能在继承类的方法中使用 base 关键字。代码如下：

```
001 using System;
002 namespace Polymorphism1
003 {
004     class A
005     {
006         public void Show()
007         {
008             Console.WriteLine("正在调用 A 类中的实例方法 Show()");
009         }
010     }
011     class B : A
012     {
013         public new void Show()
014         {
015             base.Show();
016             Console.WriteLine("正在调用 B 类中的实例方法 Show()");
017         }
018     }
019     class Program
020     {
021         static void Main(string[] args)
```

```
022      {
023          B myB = new B();
024          myB.Show();
025          Console.ReadKey();
026      }
027  }
028 }
```

代码的第 15 行使用了 `base` 关键字在派生类中访问了基类的同名方法。这段代码的执行结果如下：

正在调用 A 类中的实例方法 Show()

正在调用 B 类中的实例方法 Show()

`new` 关键字除了可以隐藏基类的同名方法之外，还可以打断继承。看下面的代码：

```
001 using System;
002 namespace Polymorphism
003 {
004     class A
005     {
006         public virtual void Show()
007         {
008             Console.WriteLine("正在调用 A 类中的实例方法 Show()");
009         }
010     }
011     class B : A
012     {
013         public override void Show()
014         {
015             Console.WriteLine("正在调用 B 类中的实例方法 Show()");
016         }
017     }
018     class C : B
019     {
020         public new virtual void Show()
021         {
022             Console.WriteLine("正在调用 C 类中的实例方法 Show()");
023         }
024     }
025     class D : C
026     {
027         public override void Show()
028         {
029             Console.WriteLine("正在调用 D 类中的实例方法 Show()");
030         }
031     }
032     class Program
033     {
```

```

034         static void Main(string[] args)
035         {
036             D myD = new D();
037             A a = myD;
038             B b = myD;
039             C c = myD;
040             a.Show();
041             b.Show();
042             c.Show();
043             myD.Show();
044             Console.ReadKey();
045         }
046     }
047 }

```

这段代码定义了 4 个类，第 4 行的类 A 是基类，然后类 B、C、D 依次继承。在类 A 中，第 6 行代码定义了一个虚方法 Show。类 B 中的第 13 行代码对它进行了重写。

第 18 行代码又定义了一个同名的虚方法，但是对方法使用了 new 关键字。第 27 行代码对第 18 行的虚方法又进行了重写。

下面来看看实际调用中的情况，第 36 行代码定义了一个类 D 的实例，然后第 37 行代码到第 39 行代码依次定义类 A、类 B、类 C 的变量引用它。也就是实现了不同的基类指向派生类对象。从第 40 行代码开始，变量 a、b、c、myD 依次对 Show() 方法进行了调用。在详细分析调用过程之前，先来看看代码的执行结果，如下：

```

正在调用 B 类中的实例方法 Show()
正在调用 B 类中的实例方法 Show()
正在调用 D 类中的实例方法 Show()
正在调用 D 类中的实例方法 Show()

```

当 a 调用 Show 方法的时候，因为 a 的编译时类型为 A 类，而运行时类型为 D 类。而 Show 方法又是一个虚方法，因此将按照派生程度来实际调用方法。如果从 A 到 D，对方法 Show 都是一路进行重写，则第 40 行代码将调用 D 中的方法。但是第 20 行代码使用了 new 关键字，并且把方法重新命名为虚方法，所以按照派生程度，第 13 行的 Show 方法将是派生程度最大的方法，它将被调用。

当 b 调用 Show 方法时，道理是相同的，派生程度最大的方法就是它本身的方法。因此，调用的仍将是 B 类中的方法。

当 c 调用 Show 方法时，因为 C 类中的方法使用了 new 关键字，并且使用了 virtual 关键字重新将方法定义成虚方法，而第 27 行代码又对这个方法进行了重写，因此这里的调用可以理解成，new 关键字打断了继承，继承关系的顶点从第 20 行开始。按照派生程度最大的类被调用的原则，类 D 中的方法将被调用。

当 myD 调用 Show 方法时，因为是本类的实例调用本类的方法，所以，D 中的方法被调用。

new 关键字为何可以理解为打断继承呢？它实际上仍起着隐藏基类同名方法的作用，但是因为它不能和 override 关键字联用，所以可以理解为方法的重写不被允许，好像在逻辑上重新从本方法开始方法的继承关系一样。如果 new 关键字和 override 关键字联用，编译器将会报错，如下：

```

错误 1  标记为 override 的成员“Polymorphism.C.Show()”不能标记为 new 或 virtual H:\书稿
new\书稿源码\第十一章\Polymorphism\Program.cs 20 34 Polymorphism

```

再试一下，不使用 new，而是将第 20 行代码重新标记为 virtual 的情况。这时编译器将会给出一个警告，如下：

```

警告 1  “Polymorphism.C.Show()”将隐藏继承的成员“Polymorphism.B.Show()”。若要使当前成员重

```

写该实现，请添加关键字 `override`。否则，添加关键字 `new`。 H:\书稿 new\书稿源码\第十一章
\Polymorphism\Program.cs 20 29 Polymorphism

从警告信息可以看到，实际上，不使用 `new` 就会得到警告，只要不使用 `override` 关键字，就会起到隐藏基类同名方法的作用。继承链将仍从这里开始。代码的执行结果如下：

正在调用 B 类中的实例方法 Show()

正在调用 B 类中的实例方法 Show()

正在调用 D 类中的实例方法 Show()

正在调用 D 类中的实例方法 Show()

对于重写来说，可以使用 `override` 关键字将基类已经标记为 `override` 关键字的方法再次进行重写以实现多态性。代码实例如下：

```
001 using System;
002 namespace Polymorphism
003 {
004     class A
005     {
006         public virtual void Show()
007         {
008             Console.WriteLine("正在调用 A 类中的实例方法 Show()");
009         }
010     }
011     class B : A
012     {
013         public override void Show()
014         {
015             Console.WriteLine("正在调用 B 类中的实例方法 Show()");
016         }
017     }
018     class C : B
019     {
020         public override void Show()
021         {
022             Console.WriteLine("正在调用 C 类中的实例方法 Show()");
023         }
024     }
025     class D : C
026     {
027         public override void Show()
028         {
029             Console.WriteLine("正在调用 D 类中的实例方法 Show()");
030         }
031     }
032     class Program
033     {
034         static void Main(string[] args)
```



```

035         {
036             B myB = new B();
037             C myC = new C();
038             D myD = new D();
039             A a = myB;
040             a.Show();
041             a = myC;
042             a.Show();
043             a = myD;
044             a.Show();
045             B b = myC;
046             b.Show();
047             b = myD;
048             b.Show();
049             C c = myD;
050             c.Show();
051             myD.Show();
052             Console.ReadKey();
053         }
054     }
055 }

```

本程序对前面的代码进行了修改，第 20 行定义的方法被修改成了 `override` 关键字进行修饰。去掉了 `new` 和 `virtual` 关键字。这样，一个完整的继承关系链就形成了。代码的第 36 行到第 38 行依次定义了类 B、C、D 的三个实例。先看执行结果再进行介绍，执行结果如下：

```

正在调用 B 类中的实例方法 Show()
正在调用 C 类中的实例方法 Show()
正在调用 D 类中的实例方法 Show()
正在调用 C 类中的实例方法 Show()
正在调用 D 类中的实例方法 Show()
正在调用 D 类中的实例方法 Show()
正在调用 D 类中的实例方法 Show()

```

第 39 行首先将 `a` 指向了 B 类实例，这时第 40 行调用 `Show` 方法，将按照派生程度最大的类调用 B 类的方法。在第 41 行，`a` 又指向了 C 类的实例。第 42 行调用方法将会是 C 类中的重写方法被调用。第 43 行将 `a` 又指向了类 D，这时按照派生程度，D 中的重写方法将被调用。

现在重点来看第 45 行代码，这时定义了一个 B 类的引用指向了 C 类的实例。现在编译时类型是 B 类型，运行时类型是 C 类型。虽然 B 中的方法标记为重写方法，但此时可以将它也看做一个虚方法，继承从这里开始，派生程度最大的将是 C 中的重写方法。当第 46 行调用 `Show` 方法的时候，C 中的方法将被调用。然后第 47 行，`b` 又指向了 D 类的实例，第 48 行调用 `Show` 方法的时候，D 类中的方法将被调用。

第 49 行定义了一个 C 类型的引用指向了一个 D 类型的实例，这时第 50 行，`c` 调用 `Show` 方法的时候，C 类型中的重写方法将被看做继承的起点，按照派生程度，D 类中的重写方法将被调用。

第 51 行代码属于本类实例调用本类方法，D 中的方法将被调用。因此从这段代码的执行结果可以了解到，`override` 关键字可以在继承的各个类中依次进行重写，而每个重写的方法又可以被重新当成虚方法作为继承的起点实现方法调用的多态性。

11.4 重写方法的其它规则

在对基类的方法进行重写时，还有一些规则需要遵守。下面进行一一地列举。首先，被重写的基类方法不能是静态方法或非虚方法。下面看代码实例：

```
001 using System;
002 namespace Polymorphism
003 {
004     class A
005     {
006         public static virtual void Show()
007         {
008             Console.WriteLine("正在调用 A 类中的实例方法 Show()");
009         }
010     }
011     class B : A
012     {
013         public override void Show()
014         {
015             Console.WriteLine("正在调用 B 类中的实例方法 Show()");
016         }
017     }
018     class Program
019     {
020         static void Main(string[] args)
021         {
022             A myA = new B();
023             A.Show();
024             Console.ReadKey();
025         }
026     }
027 }
```

这段代码的第 6 行定义了一个类的静态方法，并且用 virtual 进行了标记。然后在第 13 行意图对它进行重写以实现多态。第 22 行代码定义了一个类 A 的引用指向了类 B 的实例，第 23 行调用了类 A 的静态方法意图进行多态的调用。这时编译器会报错如下：

错误 1 静态成员“Polymorphism.A.Show()”不能标记为 override、virtual 或 abstract H:\书稿 new\书稿源码\第十一章\Polymorphism\Program.cs 6 36 Polymorphism

从错误提示可以了解到，静态方法是不能用 virtual、override 进行修饰的。多态是发生在实例上的多态。必须有实例存在才能实现多态。因此，虚方法和重写方法必须都是针对实例方法进行的定义，不能定义在静态成员上。

重写方法和被重写的基类方法除了形参类型和个数相同之外，虽然方法的返回类型不是签名的一部分，它们的返回类型也必须相同。换句话说，重写方法和被重写的基类方法必须在形式上是完全相同的。代码实例如下：

```
001 using System;
002 namespace Polymorphism
003 {
004     class A
```

```

005      {
006          public virtual void Show(int a, int b)
007          {
008              Console.WriteLine("正在调用 A 类中的实例方法 Show(), 参数相加的结果是: {0}", a
              + b);
009          }
010      }
011      class B : A
012      {
013          public override void Show()
014          {
015              Console.WriteLine("正在调用 B 类中的实例方法 Show()");
016          }
017      }
018      class C : A
019      {
020          public override int Show()
021          {
022              Console.WriteLine("正在调用 C 类中的实例方法 Show()");
023          }
024      }
025      class Program
026      {
027          static void Main(string[] args)
028          {
029              A myA = new B();
030              myA.Show();
031              Console.ReadKey();
032          }
033      }
034  }

```

这段代码将会得到如下的错误提示:

错误 1 “Polymorphism.B.Show()”: 没有找到适合的方法来重写 H:\书稿 new\书稿源码\第十一章
 \Polymorphism\Program.cs 13 30 Polymorphism

错误 2 “Polymorphism.C.Show()”: 没有找到适合的方法来重写 H:\书稿 new\书稿源码\第十一章
 \Polymorphism\Program.cs 20 29 Polymorphism

出现这两个错误的原因是: 首先这段代码中定义了两个类, A 类是基类, 在其中定义了一个虚方法, 方法的返回值为 void, 并且带有两个形参。B 类和 C 类从 A 类继承, 并且都计划重写基类中的方法。但是 B 类中的重写方法没有参数, C 类中的方法虽然没有参数, 但是返回值为 int。因为重写方法有这样的条件, 重写方法必须连返回值也和它要重写的虚方法一致。所以本段代码因为这个原因导致了这两个错误的发生。

被重写的基方法必须是派生类可以访问的。下面看代码实例:

```

001      using System;
002      namespace Polymorphism
003      {

```

```

004     class A
005     {
006         public virtual void Show(ref int a,out int b)
007         {
008             b = 22;
009             Console.WriteLine("正在调用 A 类中的实例方法 Show(), 参数相加的结果是: {0}", a
+ b);
010         }
011         private virtual void Show1()
012         {
013             Console.WriteLine("正在调用 A 类中的实例方法 Show1()");
014         }
015     }
016     class B : A
017     {
018         public override void Show(ref int a,out int b)
019         {
020             base.Show(ref a, out b);
021             Console.WriteLine("正在调用 B 类中的实例方法 Show()");
022         }
023         public override void Show1()
024         {
025             Console.WriteLine("正在调用 B 类中的实例方法 Show1()");
026         }
027     }
028     class Program
029     {
030         static void Main(string[] args)
031         {
032             int a = 11;
033             int b = 12;
034             A myA = new B();
035             myA.Show(ref a, out b);
036             Console.ReadKey();
037         }
038     }
039 }

```

在基类 A 中定义了两个虚方法，虚方法 Show 包含两个形参，一个是引用类型；另一个是输出类型。虚方法 Show1 的访问权限被设置成 private 的，这样，它就只能被本类中的成员访问。

派生类 B 从基类 A 继承，当重写基类的 Show 方法时，需要注意不只是返回类型和参数个数，连参数的类型和修饰符也要一样，否则就无法重写。在第 20 行，使用了 base 关键字调用基类中的 Show 方法，并将形参按照引用和输出方式传递给基类的方法。第 23 行重写基类的 Show1 方法。但是，编译时编译器会报错，如下：

错误 1 “Polymorphism. A. Show1()”：虚拟成员或抽象成员不能是私有的 H:\书稿new\书稿源码\第

十一章\Polymorphism\Program.cs 11 30 Polymorphism

出现这个错误的原因就是第 11 行的基类虚方法使用了 `private` 的访问修饰符，导致派生类无法访问它。现在将其访问权限修改为 `public`，则会正常编译。代码执行结果如下：

正在调用 A 类中的实例方法 Show(), 参数相加的结果是：33

正在调用 B 类中的实例方法 Show()

当重写基类的虚方法的时候，重写方法的访问权限要和基类被重写的虚方法的访问权限相同。下面是代码实例：

```
001 using System;
002 namespace Polymorphism
003 {
004     class A
005     {
006         public virtual void Show()
007         {
008             Console.WriteLine("正在调用 A 类中的实例方法 Show()");
009         }
010         protected virtual void Show1()
011         {
012             Console.WriteLine("正在调用 A 类中的实例方法 Show1()");
013         }
014         protected internal virtual void Show2()
015         {
016             Console.WriteLine("正在调用 A 类中的实例方法 Show1()");
017         }
018     }
019     class B : A
020     {
021         protected override void Show()
022         {
023             Console.WriteLine("正在调用 B 类中的实例方法 Show()");
024         }
025         public override void Show1()
026         {
027             Console.WriteLine("正在调用 B 类中的实例方法 Show1()");
028         }
029         protected override void Show2()
030         {
031             Console.WriteLine("正在调用 B 类中的实例方法 Show2()");
032         }
033     }
034     class Program
035     {
036         static void Main(string[] args)
037         {
```

```

038         A myA = new B();
039         myA.Show();
040         myA.Show2();
041         Console.ReadKey();
042     }
043 }
044 }

```

本实例的基类 A 中定义了三个虚方法用来在派生类中重写。但是这三个虚方法的访问权限各不相同，第一个方法是 public 的，第二个方法是 protected，第三个方法是 protected internal。类 B 派生自类 A，当在类 B 中重写这三个方法时，方法名和参数以及返回类型都相同，但是它们的访问权限相对于基类被重写的方法做了修改，它们和基类中被重写的虚方法都不相同。在这种情况下，是违反重写规则的，重写规则要求重写方法必须和被重写的虚方法的访问权限相同，因此，编译器会报错，如下：

错误 1 “Polymorphism.B.Show()”：当重写“public”继承成员“Polymorphism.A.Show()”时，无法更改访问修饰符 H:\书稿 new\书稿源码\第十一章\Polymorphism\Program.cs 21 33

Polymorphism

错误 2 “Polymorphism.B.Show1()”：当重写“protected”继承成员“Polymorphism.A.Show1()”时，无法更改访问修饰符 H:\书稿 new\书稿源码\第十一章\Polymorphism\Program.cs 25 30

Polymorphism

错误 3 “Polymorphism.B.Show2()”：当重写“protected internal”继承成员“Polymorphism.A.Show2()”时，无法更改访问修饰符 H:\书稿 new\书稿源码\第十一章\Polymorphism\Program.cs 29 33 Polymorphism

这三个错误的类型都是一样的，都是提示当重写基类的虚方法时，无法更改访问修饰符。现在将重写方法的访问修饰符定义成和被重写的方法一样，代码的执行结果如下：

正在调用 B 类中的实例方法 Show()

正在调用 B 类中的实例方法 Show2()

但是有一种情况是允许修改基类的虚方法的访问修饰符的，那就是当基类中的虚方法是 protected internal 的，那么允许在其它程序集中的派生类中重写它的时候，定义访问修饰符为 protected，而且必须是 protected。下面先将前面的项目类型变为类库，并且移除掉含有入口点方法的类，并且将基类 A 的访问修饰符修改为 public。这样在其它程序集引用它的时候，就可以访问基类 A。然后再添加一个方法用来调用 Show2，这是因为 Show2 方法只能被本类内部和派生类访问，在类外无法访问。修改后的代码如下：

```

001 using System;
002 namespace Polymorphism
003 {
004     public class A
005     {
006         public virtual void Show()
007         {
008             Console.WriteLine("正在调用 A 类中的实例方法 Show()");
009         }
010         protected virtual void Show1()
011         {
012             Console.WriteLine("正在调用 A 类中的实例方法 Show1()");
013         }
014         protected internal virtual void Show2()

```

```
015     {
016         Console.WriteLine("正在调用 A 类中的实例方法 Show1()");
017     }
018     public void Invoke()
019     {
020         this.Show2();
021     }
022 }
023 class B : A
024 {
025     public override void Show()
026     {
027         Console.WriteLine("正在调用 B 类中的实例方法 Show()");
028     }
029     protected override void Show1()
030     {
031         Console.WriteLine("正在调用 B 类中的实例方法 Show1()");
032     }
033     protected internal override void Show2()
034     {
035         Console.WriteLine("正在调用 B 类中的实例方法 Show2()");
036     }
037 }
038 }
```

第 18 行定义的方法 `Invoke` 就是为了用来调用 `Show2` 方法。接下来新建一个控制台项目并物理引用这个类库。然后输入的代码如下：

```
001 using System;
002 using Polymorphism;
003 namespace Polymorphism1
004 {
005     class C : A
006     {
007         protected override void Show2()
008         {
009             Console.WriteLine("正在调用新建程序集中类 C 的实例方法 Show2()");
010         }
011     }
012     class Program
013     {
014         static void Main(string[] args)
015         {
016             A myA = new C();
017             myA.Invoke();
018             Console.ReadKey();
019         }
020     }
021 }
```

```

019         }
020     }
021 }

```

第 7 行定义的方法用来重写基类中的同名虚方法，基类中的虚方法的访问权限是 `protected internal` 的，但是因为重写方法所在的类位于另一个程序集中，所以派生类中的重写方法可以定义成 `protected` 的，这是唯一的一种重写方法的访问权限可以和基类的虚方法的权限不相同的情况。这段代码的执行结果如下：

正在调用新建程序集中类 C 的实例方法 Show2()

11.5 密封方法

如果使用 `sealed` 关键字修饰类的实例方法，则称这个方法为密封方法。`sealed` 关键字要和 `override` 关键字联用，也就是说，它用在方法的重写过程中。使用 `sealed` 关键字后，类的派生类就不再能重写这个方法了。下面是代码实例：

```

001 using System;
002 namespace Polymorphism
003 {
004     public class A
005     {
006         public virtual void Show()
007         {
008             Console.WriteLine("正在调用 A 类中的实例方法 Show()");
009         }
010     }
011     class B : A
012     {
013         public sealed override void Show()
014         {
015             Console.WriteLine("正在调用 B 类中的实例方法 Show()");
016         }
017     }
018     class C : B
019     {
020     }
021     public override void Show()
022     {
023         Console.WriteLine("正在调用 C 类中的实例方法 Show()");
024     }
025 }
026 class Program
027 {
028     static void Main(string[] args)
029     {
030         A myA = new C();
031         myA.Show();
032         Console.ReadKey();
033     }

```



```
034     }
035 }
```

在这个例子中，共有三个类，它们依次派生。类 A 是基类，在它里面定义了一个虚方法用来重写。第 13 行代码中，类 B 的实例方法对类 A 中的虚方法进行了重写。但是这个方法在访问修饰符和 override 关键字之间标记了 sealed 关键字。这样一来，B 类的派生类 C 将不再能重写这个方法了。这样，对于 Show 方法来说，B 类中的 Show 方法就是派生程度最大的类。

第 19 行定义的派生类 C 中，对于 Show 方法又进行了重写。因为 B 类中的方法已经是密封方法，所以在编译的时候，编译器就会报告错误，如下：

```
错误 1    “Polymorphism.C.Show()”：继承成员“Polymorphism.B.Show()”是 sealed，无法进行重写
E:\书稿 new\书稿源码\第十一章\Polymorphism\Program.cs    21    30    Polymorphism
```

下面将类 C 中的重写方法去掉，然后修改代码：

```
001 using System;
002 namespace Polymorphism
003 {
004     public class A
005     {
006         public virtual void Show()
007         {
008             Console.WriteLine("正在调用 A 类中的实例方法 Show()");
009         }
010     }
011     class B : A
012     {
013         public sealed override void Show()
014         {
015             Console.WriteLine("正在调用 B 类中的实例方法 Show()");
016         }
017     }
018 }
019 class C : B
020 {
021 }
022 class Program
023 {
024     static void Main(string[] args)
025     {
026         A myA = new C();
027         myA.Show();
028         C myC = new C();
029         myC.Show();
030         Console.ReadKey();
031     }
032 }
033 }
```

现在类 C 中没有定义任何的成员，但是因为它依次派生下来的，它将继承它所有基类中的成员。对于 Show 方法来说，实际上，类 A 中的 Show 方法和类 B 中的 Show 方法，类 C 都已经继承了下来。只不过，因为重写的关系，类 B 中的 Show 方法隐藏了类 A 中的 Show 方法。第 26 行代码定义一个类 A 的引用指向类 C 的实例之后，第 27 行代码调用 Show 方法的时候，因为类 B 中的 Show 方法是派生程度最大的方法，所以它将被调用。而第 28 行定义类 C 的实例在第 29 行调用 Show 方法的时候，它将调用继承下来的类 B 中的方法，而类 A 中的方法被隐藏调用了。如果要使用类 A 中的方法，那么只能在类 B 中的方法中使用 base 关键字间接调用。这段代码的执行结果如下：

正在调用 B 类中的实例方法 Show()

正在调用 B 类中的实例方法 Show()

11.6 抽象类

抽象类使用 abstract 关键字进行修饰，它是一种不完整的类。说它不完整，是因为它是一种偏重于逻辑设计上的类。也可以说，它提供了程序设计上的一种接口。抽象类经常包括抽象方法，从抽象类继承的类必须去实现这些抽象成员以实现多态。抽象类本身不能实例化，它只用来派生。抽象类可以继承自另一个抽象类。下面是代码实例：

```
001 using System;
002 namespace Polymorphism
003 {
004     public abstract class A
005     {
006         private int a;
007         private int b;
008         public A(int a, int b)
009         {
010             Console.WriteLine("调用抽象类的构造方法");
011             this.a = a;
012             this.b = b;
013         }
014         public virtual void Show()
015         {
016             Console.WriteLine("正在调用 A 类中的实例方法 Show()");
017             Console.WriteLine("字段 a 的值是: {0}, 字段 b 的值是: {1}", a, b);
018         }
019     }
020     class B : A
021     {
022         public B(int a, int b):base(a, b)
023         {
024             Console.WriteLine("调用派生类的构造方法");
025         }
026         public override void Show()
027         {
028             Console.WriteLine("正在调用 B 类中的实例方法 Show()");
029             base.Show();
030         }
031     }
032 }
```

```

031
032     }
033     class Program
034     {
035         static void Main(string[] args)
036         {
037             A myA = new B(22, 35);
038             myA.Show();
039             Console.ReadKey();
040         }
041     }
042 }

```

本段代码的第 4 行定义的就是一个抽象类，它用 `abstract` 关键字进行修饰。第 6 行代码和第 7 行代码是抽象类包含的两个字段。代码的第 8 行是构造方法，它对两个实例字段进行初始化。第 14 行代码定义了一个虚方法，它打印了字段 `a` 和字段 `b` 的值。

第 20 行代码中的类 `B` 从类 `A` 中派生，第 22 行代码定义了一个派生类的构造方法，它使用构造方法初始值设定项对基类的构造方法进行调用和传值。第 26 行代码定义了一个重写方法，它在方法体内部使用 `base` 关键字调用了基类被重写的虚方法。

第 37 行代码定义了类 `A` 的引用，它指向了类 `B` 的实例。当调用 `Show` 方法的时候，实现了多态。这段代码的执行结果如下：

```

调用抽象类的构造方法
调用派生类的构造方法
正在调用 B 类中的实例方法 Show()
正在调用 A 类中的实例方法 Show()
字段 a 的值是：22, 字段 b 的值是：35

```

从代码的执行结果可以观察到，使用抽象类和使用普通类看起来没什么差别。当第 37 行代码创建类 `B` 的实例的时候，首先类 `B` 的构造方法被调用，类 `B` 的构造方法又会调用类 `A` 的构造方法，因此，类 `A` 的构造方法中的 `Console.WriteLine` 方法被调用，接下来设置字段的值。然后，代码逻辑返回到类 `B` 的构造方法中，类 `B` 的构造方法中的 `Console.WriteLine` 方法被调用。接下来，当类 `A` 的引用调用 `Show` 方法的时候，因为重写的关系，类 `B` 中的 `Show` 方法被调用，打印字符串后，`base` 关键字又会调用类 `A` 的 `Show` 方法打印字段的值。

下面将 `Main` 方法中加入下列的语句：

```

A myA1 = new A(22, 33);
myA1.Show();

```

这两行语句创建了抽象类 `A` 的实例，并且调用了其实例方法。因为开始在定义抽象类的时候，为它定义了构造方法。看起来，抽象类 `A` 应该可以实例化。但是实际上，编译器会报错，如下：

```

错误 1 无法创建抽象类或接口“Polymorphism.A”的实例 E:\书稿 new\书稿源码\第十一章
\Polymorphism\Program.cs 39 22 Polymorphism

```

所以说，即使抽象类包括字段成员，虚方法以及实例构造方法，但是，抽象类就是为类的总体设计而存在的，它就是要实现继承和多态。它本身决不能实例化。类似它的还有接口类型，后面会介绍到。

下面将代码略作修改，代码如下：

```

001 using System;
002 namespace Polymorphism
003 {

```

```

004     public sealed abstract class A
005     {
006         private int a;
007         private int b;
008         public A(int a, int b)
009         {
010             Console.WriteLine("调用抽象类的构造方法");
011             this.a = a;
012             this.b = b;
013         }
014         public virtual void Show()
015         {
016             Console.WriteLine("正在调用 A 类中的实例方法 Show()");
017             Console.WriteLine("字段 a 的值是: {0}, 字段 b 的值是: {1}", a, b);
018         }
019     }
020     class B : A
021     {
022         public B(int a, int b):base(a, b)
023         {
024             Console.WriteLine("调用派生类的构造方法");
025         }
026         public override void Show()
027         {
028             Console.WriteLine("正在调用 B 类中的实例方法 Show()");
029             base.Show();
030         }
031     }
032 }
033 class Program
034 {
035     static void Main(string[] args)
036     {
037         A myA = new B(22, 35);
038         myA.Show();
039         Console.ReadKey();
040     }
041 }
042 }

```

这段代码只是在第 4 行为抽象类 A 加上了一个 sealed 关键字, 让它变成了一个密封类。这时, 编译器会报告如下错误:

错误 1 "Polymorphism.A": 抽象类不能是密封的或静态的 E:\书稿 new\书稿源码\第十一章\Polymorphism\Program.cs 4 34 Polymorphism

错误 2 "object" 不包含采用 "2" 个参数的构造函数 E:\书稿 new\书稿源码\第十一章

```
\Polymorphism\Program.cs 22 32 Polymorphism
```

错误 3 “object”并不包含“Show”的定义 E:\书稿 new\书稿源码\第十一章

```
\Polymorphism\Program.cs 29 18 Polymorphism
```

错误 4 无法声明静态类型“Polymorphism.A”的变量 E:\书稿 new\书稿源码\第十一章

```
\Polymorphism\Program.cs 37 13 Polymorphism
```

错误 5 无法将类型“Polymorphism.B”隐式转换为“Polymorphism.A” E:\书稿new\书稿源码\第

```
十一章\Polymorphism\Program.cs 37 21 Polymorphism
```

观察这些错误信息，除了第一行的错误信息外，其余的错误会感觉难以理解。实际上，造成这些错误的原因就是，抽象类是为继承和派生而来，将抽象类标记为密封的，就禁止了从它继承和派生。因此会报告如上这些奇怪的错误。所以，抽象类不能标记为 sealed 的。同理，也不能将抽象类定义成静态的，代码如下：

```
public static abstract class A
```

这时编译器会报告如下错误：

错误 1 “Polymorphism.A”：抽象类不能是密封的或静态的 E:\书稿 new\书稿源码\第十一章

```
\Polymorphism\Program.cs 4 34 Polymorphism
```

相同的情况，静态类也不能标记为 abstract 的。

11.7 抽象方法

当类的实例方法的定义中包含 abstract 关键字的时候，称这个方法是抽象方法。抽象方法就是需要被重写以实现多态的。所以抽象方法隐式为虚方法，但是不能用 virtual 关键字修饰它。下面是代码实例：

```
001 using System;
002 namespace Polymorphism
003 {
004     public abstract class A
005     {
006         private int a;
007         private int b;
008         public A(int a, int b)
009         {
010             Console.WriteLine("调用抽象类的构造方法");
011             this.a = a;
012             this.b = b;
013         }
014         public void Print()
015         {
016             Console.WriteLine("字段 a 的值是: {0}, 字段 b 的值是: {1}", a, b);
017         }
018         public abstract void Show();
019     }
020     class B : A
021     {
022         public B(int a, int b):base(a, b)
023         {
024             Console.WriteLine("调用派生类的构造方法");
025         }
026         public override void Show()
```

```

027         {
028             Console.WriteLine("正在调用 B 类中的实例方法 Show()");
029             Console.WriteLine("现在调用 A 类中的实例方法 Print()");
030             base.Print();
031         }
032     }
033     class Program
034     {
035         static void Main(string[] args)
036         {
037             A myA = new B(22, 35);
038             myA.Show();
039             Console.ReadKey();
040         }
041     }
042 }

```

本程序中，第 4 行定义了一个抽象类，第 14 行代码定义了一个普通的实例方法用来打印字段的值。第 18 行代码定义了一个抽象方法用来让派生类进行重写。在第 26 行开始的派生类的重写方法中对基类的抽象方法进行了重写，它用 base 关键字调用了基类的 Print 方法来打印私有字段的值。从第 18 行可以看到，抽象方法的格式就是使用 abstract 关键字，关键字位于访问权限修饰符之后，方法的返回值之前。在方法的形参列表后不写方法体实现，只用一个分号结束，就好像分部方法的声明一样。这段代码的执行结果如下：

调用抽象类的构造方法

调用派生类的构造方法

正在调用 B 类中的实例方法 Show()

现在调用 A 类中的实例方法 Print()

字段 a 的值是：22, 字段 b 的值是：35

抽象方法还有如下使用规则，使用抽象方法的类必须是抽象类。也就是说，如果要定义一个抽象方法，那么它所在的类也要同时定义成抽象类。下面是代码实例：

```

001 using System;
002 namespace Polymorphism1
003 {
004     class A
005     {
006         public abstract void Show();
007     }
008     class B : A
009     {
010         public override void Show()
011         {
012             Console.WriteLine("类 B 中的 Show 方法被调用");
013         }
014     }
015     class Program

```

```

016     {
017         static void Main(string[] args)
018         {
019             A myA = new B();
020             myA.Show();
021             Console.ReadKey();
022         }
023     }
024 }

```

这段代码中，在类 A 中定义了一个抽象方法，类 B 从类 A 继承。类 B 中定义了一个重写方法来重写基类中定义的抽象方法。但是这段代码在编译时将会报错，如下：

错误 1 “Polymorphism1.A.Show()”是抽象的，但它包含在非抽象类“Polymorphism1.A”中 H:\书稿 new\书稿源码\第十一章\Polymorphism1\Program.cs 6 30 Polymorphism1

从错误提示中可以知道，抽象方法必须定义在抽象类中。

在抽象方法的使用过程中还需要注意，在派生类中无法访问基类的抽象方法。实例代码如下：

```

001 using System;
002 namespace Polymorphism
003 {
004     public abstract class A
005     {
006         private int a;
007         private int b;
008         public A(int a, int b)
009         {
010             Console.WriteLine("调用抽象类的构造方法");
011             this.a = a;
012             this.b = b;
013         }
014         public void Print()
015         {
016             Console.WriteLine("字段 a 的值是: {0}, 字段 b 的值是: {1}", a, b);
017         }
018         public abstract void Show();
019     }
020     class B : A
021     {
022         public B(int a, int b):base(a,b)
023         {
024             Console.WriteLine("调用派生类的构造方法");
025         }
026         public override void Show()
027         {
028             Console.WriteLine("正在调用 B 类中的实例方法 Show()");
029             Console.WriteLine("现在调用 A 类中的实例方法 Print()");

```

```

030         base.Print();
031     }
032     public void Invoke()
033     {
034         base.Show();
035     }
036 }
037 class Program
038 {
039     static void Main(string[] args)
040     {
041         A myA = new B(22, 35);
042         myA.Show();
043         B myB = new B(35, 56);
044         myB.Invoke();
045         Console.ReadKey();
046     }
047 }
048 }

```

这段代码在派生类中定义了一个普通的实例方法，在方法中使用了 base 关键字调用了基类的 Show 方法。但是因为基类的 Show 方法是一个抽象方法，所以代码在编译时会报错。如下：

错误 1 无法调用抽象基成员：“Polymorphism.A.Show()” H:\书稿 new\书稿源码\第十一章
 \Polymorphism\Program.cs 34 13 Polymorphism

因为抽象方法的存在可以实现多态。如果使用不当，在基类中调用了抽象方法，容易造成代码的循环。代码实例如下：

```

001 using System;
002 namespace Polymorphism
003 {
004     public abstract class A
005     {
006         private int a;
007         private int b;
008         public A(int a, int b)
009         {
010             Console.WriteLine("调用抽象类的构造方法");
011             this.a = a;
012             this.b = b;
013         }
014         public void Print()
015         {
016             Console.WriteLine("字段 a 的值是：{0}, 字段 b 的值是：{1}", a, b);
017             this.Show();
018         }
019         public abstract void Show();

```



```
020     }
021     class B : A
022     {
023         public B(int a, int b):base(a,b)
024         {
025             Console.WriteLine("调用派生类的构造方法");
026         }
027         public override void Show()
028         {
029             Console.WriteLine("正在调用 B 类中的实例方法 Show()");
030             Console.WriteLine("现在调用 A 类中的实例方法 Print()");
031             base.Print();
032         }
033     }
034     class Program
035     {
036         static void Main(string[] args)
037         {
038             A myA = new B(22, 35);
039             myA.Show();
040             Console.ReadKey();
041         }
042     }
043 }
```

这段代码中，第 27 行的重写方法中使用 base 关键字调用了基类的 Print 方法。而基类中的 Print 方法又调用了基类中的 Show 方法，因为多态的关系，派生类中的 Show 方法又被调用。这样就形成了代码的循环。

抽象类可以派生自普通类，抽象方法也可以重写虚方法，从而使虚方法的原始实现不再可用。实例代码如下：

```
001     using System;
002     namespace Polymorphism1
003     {
004         class A
005         {
006             public virtual void Show()
007             {
008                 Console.WriteLine("类 A 中的 Show 方法被调用");
009             }
010         }
011         abstract class B : A
012         {
013             public override abstract void Show();
014         }
015         class C : B
```

```

016     {
017         public override void Show()
018     {
019         Console.WriteLine("类 C 中的 Show 方法被调用");
020     }
021     }
022     class Program
023     {
024         static void Main(string[] args)
025         {
026             A myA = new C();
027             myA.Show();
028             Console.ReadKey();
029         }
030     }
031 }

```

本段代码描述的情景是一个很特殊的状态，在实际编程中通常不这么用，因为抽象类一般用作顶层的构架使用。代码的第 4 行定义了一个普通类，在其中定义了一个虚方法。

第 11 行定义了一个抽象类继承自这个普通类，在抽象类中定义了一个重写方法用来重写基类的虚方法 Show。但是在 override 关键字之后，又紧跟着一个 abstract 关键字，这样，这个方法又变成了一个抽象方法，它没有方法的实现。

第 15 行定义了一个类 C 继承自抽象类 B，第 17 行定义了一个重写方法重写基类中的抽象方法。在第 26 行定义了一个 A 类的引用，它指向了派生类 C 的实例。当调用 Show 方法的时候，C 类中的 Show 方法就是派生程度最大的方法，它将被调用。这段代码的执行结果如下：

类 C 中的 Show 方法被调用

11.8 属性的多态性

首先来看属性在继承中的同名隐藏情况，下面是代码实例：

```

001     using System;
002     namespace Polymorphism1
003     {
004         class Test
005         {
006             private int a;
007             private int b;
008             public int A
009             {
010                 get
011                 {
012                     return a;
013                 }
014                 set
015                 {
016                     a = value;
017                 }
018             }
019         }
020     }

```

```
018     }
019     public int B
020     {
021         get
022         {
023             return b;
024         }
025         set
026         {
027             b = value;
028         }
029     }
030     public Test()
031     {
032         this.A = 11;
033         this.B = 12;
034     }
035 }
036 class Test1 : Test
037 {
038     private int a;
039     private int b;
040     public int A
041     {
042         get
043         {
044             return a;
045         }
046         private set
047         {
048             a = value;
049         }
050     }
051     protected int B
052     {
053         get
054         {
055             return b;
056         }
057         set
058         {
059             b = value;
060         }
061     }
```

```

062         public Test1()
063         {
064             this.A = 21;
065             this.B = 22;
066         }
067     }
068     class Program
069     {
070         static void Main(string[] args)
071         {
072             Test1 t1 = new Test1();
073             Console.WriteLine("Test1 类中的属性 A 的值是: {0}, 属性 B 的值是 {1}", t1.A,
                                t1.B);
074             Console.ReadKey();
075         }
076     }
077 }

```

在这段代码中首先定义了一个基类 Test，在基类中定义了两个字段和两个属性。属性对字段有存取的作用。在派生类中，又定义了两个和基类中的属性同名的属性，但是第 2 个属性的访问权限和基类中的同名属性不同。在每个类中各定义了两个无参的构造方法对属性进行初始化操作。第 72 行代码定义了一个 Test1 类的实例，第 73 行通过这个实例调用属性 A 和属性 B。因为派生类中的属性与基类的属性同名，所以基类的属性将被隐藏掉。t1.A 将返回派生类中的属性 A 的值。而 t1.B 将返回基类中的属性 B 的值。为什么这样呢？因为 B 类中的属性 B 是 protected 的，它在类外无法访问。这时被隐藏掉的 Test 类的属性 B 将起作用。这段代码的执行结果如下：

Test1 类中的属性 A 的值是：21, 属性 B 的值是 12

属性可以是虚属性，从而让派生类可以重写它。下面是代码实例：

```

001     using System;
002     namespace Polymorphism1
003     {
004         class Test
005         {
006             private int a;
007             public virtual int A
008             {
009                 get
010                 {
011                     return a;
012                 }
013                 set
014                 {
015                     a = value;
016                 }
017             }
018             public Test()

```

```
019         {
020             this.A = 11;
021         }
022     }
023     class Test1 : Test
024     {
025         private int a;
026         public override int A
027         {
028             get
029             {
030                 Console.WriteLine("派生类 Test1 中的属性 A 的 get 访问器被调用");
031                 return a;
032             }
033             set
034             {
035                 Console.WriteLine("派生类 Test1 中的属性 A 的 set 访问器被调用");
036                 a = value;
037             }
038         }
039         public Test1()
040         {
041             this.A = 21;
042         }
043     }
044     class Program
045     {
046         static void Main(string[] args)
047         {
048             Test t = new Test1();
049             t.A = 100;
050             Console.WriteLine("属性 A 的值是: {0}", t.A);
051             Console.ReadKey();
052         }
053     }
054 }
```

在类 Test 中定义了一个虚属性，它用来操作基类中的字段 a。第 18 行的构造方法对属性做初始化操作。类 Test1 继承自基类 Test，在第 26 行定义了一个重写属性，它对派生类中的字段进行操作。第 39 行代码是构造方法，它对派生类的属性进行设置。第 48 行定义了一个 Test 类型的引用指向一个 Test1 类型的实例。第 49 行当调用 A 属性的时候，因为多态的作用，派生类的属性将赋值。当第 50 行代码访问属性的 get 访问器的时候，派生类的属性的 get 访问器将被调用。这段代码的执行结果如下：

```
派生类 Test1 中的属性 A 的 set 访问器被调用
派生类 Test1 中的属性 A 的 set 访问器被调用
派生类 Test1 中的属性 A 的 set 访问器被调用
```

派生类 Test1 中的属性 A 的 get 访问器被调用

属性 A 的值是：100

下面再分析一下产生这个结果的原因，当第 48 行定义一个 Test 类型的引用指向 Test1 实例的时候，首先派生类的构造方法将被调用。而派生类的构造方法又会转而去调用基类的构造方法。而基类的构造方法是对属性 A 进行赋值。因为属性实现了重写，而创建实例的编译时类型是 Test 类型，运行时类型是 Test1。所以派生程度最大的就是 Test1 中的属性 A，所以这时派生类中的属性的 set 访问器将被访问。接下来，代码控制回到派生类中的构造方法中，派生类中的属性 A 又将被访问。接下来是第 49 行的赋值将再次实现多态调用派生类中的属性的 set 访问器。第 50 行代码按照派生程度最大的原则调用了派生类中的属性 A 的 get 访问器。

在定义属性为虚属性时的时候，需要注意整个属性都是 virtual 的，而不能将属性的其中一个访问器定义为 virtual 的。如果只定义其中一个为 virtual，将会收到编译器的错误提示。如下：

错误 1 修饰符“virtual”对该项无效 H:\书稿 new\书稿源码\第十一章\Polymorphism1\Program.cs
9 21 Polymorphism1

当重写基类的属性时，重写属性必须和基类的属性具有完全相同的访问权限修饰符、属性类型和名称。如果基类的属性只包含一个访问器，则重写属性也必须只包含一个访问器；如果基类的属性包含两个访问器，是读写属性，则重写属性可以包含两个访问器，也可以只包含其中一个访问器进行重写。下面是代码实例：

```
001 using System;
002 namespace Polymorphism1
003 {
004     class Test
005     {
006         private int a;
007         public virtual int A
008         {
009             get
010             {
011                 return a;
012             }
013         }
014         public virtual int B
015         {
016             get;
017             set;
018         }
019         public Test()
020         {
021             this.a = 11;
022         }
023     }
024     class Test1 : Test
025     {
026         private int a;
027         public override int A
```

```

028         {
029             get
030             {
031                 Console.WriteLine("派生类 Test1 中的属性 A 的 get 访问器被调用");
032                 return a;
033             }
034             set
035             {
036                 Console.WriteLine("派生类 Test1 中的属性 A 的 set 访问器被调用");
037                 a = value;
038             }
039         }
040         public override int B
041         {
042             get
043             {
044                 return base.B;
045             }
046         }
047         public Test1()
048         {
049             this.a = 21;
050         }
051     }
052     class Program
053     {
054         static void Main(string[] args)
055         {
056             Test t = new Test1();
057             Console.WriteLine("属性 A 的值是: {0}", t.A);
058             Console.WriteLine("属性 B 的值是: {0}", t.B);
059             Console.ReadKey();
060         }
061     }
062 }

```

本段代码中的基类 Test 中定义了两个虚属性，它包含的字段 a 通过第 19 行的实例构造方法进行初始化。属性 A 只包含了一个 get 访问器，它只能返回字段 a 的值，而没有 set 访问器可以设置它。属性 B 是一个自动实现的虚属性。在派生类中对虚属性 A 进行重写的时候，又添加了一个 set 访问器。而对属性 B 进行重写的时候却去掉了 set 访问器。在它的 get 访问器中使用 base 关键字访问了基类属性 B 的 get 访问器。第 57 行代码访问属性 A 的 get 访问器；第 58 行代码访问属性 B 的 get 访问器。这段代码包含了一个错误。下面开始编译它，看一下错误提示：

```

错误 1      “Polymorphism1.Test1.A.set”：无法重写，因为“Polymorphism1.Test.A”没有可重写的 set
访问器      H:\书稿 new\书稿源码\第十一章\Polymorphism1\Program.cs  34  13  Polymorphism1

```

出现这个错误的原因是，重写属性的访问器必须和基类的被重写的虚属性的访问器一样，或选择重写其中

一个。但是本程序的重写属性 A 却添加了一个 set 访问器，从而导致了这个错误的发生。可以看到，重写属性 B 却没有报错，减少一个访问器进行重写是允许的。下面修改代码，将重写属性 A 的错误进行改正，然后再修改一下基类属性 A 的访问器访问权限修饰符。修改后的代码如下：

```
001     using System;
002     namespace Polymorphism1
003     {
004         class Test
005         {
006             private int a;
007             public virtual int A
008             {
009                 protected get
010                 {
011                     return a;
012                 }
013                 set
014                 {
015                     a = value;
016                 }
017             }
018             public virtual int B
019             {
020                 get;
021                 set;
022             }
023             public Test()
024             {
025                 this.a = 11;
026             }
027         }
028         class Test1 : Test
029         {
030             private int a;
031             public override int A
032             {
033                 get
034                 {
035                     Console.WriteLine("派生类 Test1 中的属性 A 的 get 访问器被调用");
036                     return a;
037                 }
038             }
039             public override int B
040             {
041                 get
```



```

042         {
043             return base.B;
044         }
045     }
046     public Test1()
047     {
048         this.a = 21;
049     }
050 }
051 class Program
052 {
053     static void Main(string[] args)
054     {
055         Test t = new Test1();
056         Console.WriteLine("属性 A 的值是: {0}", t.A);
057         Console.WriteLine("属性 B 的值是: {0}", t.B);
058         Console.ReadKey();
059     }
060 }
061 }

```

这段代码的重写属性 A 挑选了基类的属性中的 get 访问器进行了重写。而重写属性 B 挑选了基类属性 B 中的 get 访问器进行了重写。在逻辑上看来，应该没有什么问题。但是这段代码在编译的时候仍然会报错。错误信息如下：

错误 1 “Polymorphism1.Test1.A.get”：当重写“protected”继承成员“Polymorphism1.Test.A.get”时，无法更改访问修饰符

出现这个错误的原因是，在属性重写的规则中，对于被重写的基类的属性的访问器，如果该访问器有访问器访问权限修饰符，则重写属性的相应访问器的访问权限修饰符要和被重写的属性一致。对于本例来说，属性 A 的 get 访问器的访问权限是 protected 的。而重写属性的 get 访问器继承了整个属性的访问权限 public，这就导致了重写属性和基类虚属性的一致，因而报错。

属性还可以是抽象的，抽象属性和抽象方法一样，都必须定义在抽象类中。下面是代码实例：

```

001 using System;
002 namespace Polymorphism1
003 {
004     abstract class Test
005     {
006         public abstract int A
007         {
008             get;
009             set;
010         }
011     }
012     class Test1 : Test
013     {
014         private int a;

```

```

015         public override int A
016     {
017         get
018         {
019             Console.WriteLine("派生类 Test1 中的属性 A 的 get 访问器被调用");
020             return a;
021         }
022         set
023         {
024             Console.WriteLine("派生类 Test1 中的属性 A 的 set 访问器被调用");
025             a = value;
026         }
027     }
028     public Test1()
029     {
030         this.A = 21;
031     }
032 }
033 class Program
034 {
035     static void Main(string[] args)
036     {
037         Test t = new Test1();
038         Console.WriteLine("属性 A 的值是: {0}", t.A);
039         t.A = 100;
040         Console.WriteLine("赋值后, 属性 A 的值是: {0}", t.A);
041         Console.ReadKey();
042     }
043 }
044 }

```

要定义一个抽象属性, 那么该抽象属性所在的类必须首先定义成抽象的。因此, 代码的第 4 行定义了一个抽象类。代码的第 6 行定义了一个抽象属性, 抽象属性和抽象方法一样, 也以 `abstract` 关键字进行修饰。但是它的访问器不能够写实现, 访问器直接以分号结束。虽然它和自动实现的属性形式一样, 但是它们的本质不同。它是需要派生类来重写它的。

在派生类中, 第 15 行代码定义了一个重写属性, 根据基类抽象属性提供了什么访问器, 重写属性就要提供相关的实现。这段代码的执行结果如下:

```

派生类 Test1 中的属性 A 的 set 访问器被调用
派生类 Test1 中的属性 A 的 get 访问器被调用
属性 A 的值是: 21
派生类 Test1 中的属性 A 的 set 访问器被调用
派生类 Test1 中的属性 A 的 get 访问器被调用
赋值后, 属性 A 的值是: 100

```

这段代码的执行顺序是这样的, 当第 37 行代码定义了一个基类的引用指向了派生类的实例的时候, 首先派生类的构造方法被调用。它调用基类的默认构造方法。然后逻辑返回执行第 30 行的代码, 当为属性赋值为

21 的时候，派生类的属性的 set 访问器被调用。然后执行第 38 行的代码，这时，因为多态的关系，派生程度最大的重写属性的 get 访问器被调用。当第 39 行代码为属性赋值的时候，仍然是多态性在起作用，派生类中的重写属性的 set 访问器被调用。第 40 行打印属性的值的时候，派生类的重写属性的 get 访问器被调用。

前面介绍过，抽象属性和自动属性的形式上很相似，但是它们的本质是不同的。它们的不同还可以体现在，自动属性必须同时具备两种访问器，也就是说，它必须是读写属性。而抽象属性却可以只提供一个访问器用作派生类的重写实现。下面是代码实例：

```
001 using System;
002 namespace Polymorphism1
003 {
004     abstract class Test
005     {
006         public abstract int A
007         {
008             get;
009         }
010     }
011     class Test1 : Test
012     {
013         private int a;
014         public override int A
015         {
016             get
017             {
018                 Console.WriteLine("派生类 Test1 中的属性 A 的 get 访问器被调用");
019                 return a;
020             }
021         }
022         public Test1()
023         {
024             this.a = 21;
025         }
026     }
027     class Program
028     {
029         static void Main(string[] args)
030         {
031             Test t = new Test1();
032             Console.WriteLine("属性 A 的值是: {0}", t.A);
033             Console.ReadKey();
034         }
035     }
036 }
```

在前面代码的基础上，抽象类 Test 中的抽象方法只定义了一个 get 访问器以供派生类重写实现。在派

生类中，第 14 行代码对基类的抽象属性进行了重写。但是，因为没有 set 访问器，所以第 22 行的构造方法中需要直接对字段 a 进行初始化赋值。在类外对属性来说，只能访问其 get 访问器，不能对它进行赋值操作。这段代码的执行结果如下：

派生类 Test1 中的属性 A 的 get 访问器被调用

属性 A 的值是：21

如果一个抽象类从一个普通类继承，那么可以将一个重写方法重新标记为抽象的，以便于派生类再次对它进行实现。实例代码如下：

```
001 using System;
002 namespace Polymorphism1
003 {
004     class Test
005     {
006         public virtual int A
007         {
008             get;
009             set;
010         }
011     }
012     abstract class Test1 : Test
013     {
014         private int a;
015         public abstract override int A
016         {
017             get;
018             set;
019         }
020         public Test1()
021         {
022             this.a = 21;
023         }
024     }
025     class Test2 : Test1
026     {
027         private int a = 31;
028         public override int A
029         {
030             get
031             {
032
033                 return a;
034             }
035             set
036             {
037                 a = value;
```

```

038         }
039     }
040 }
041 class Program
042 {
043     static void Main(string[] args)
044     {
045         Test t = new Test2();
046         Console.WriteLine("属性 A 的值是: {0}", t.A);
047         Console.ReadKey();
048     }
049 }
050 }

```

这段代码定义了三个类，它们依次进行继承。其中类 `Test` 是一个普通类，在它里面的第 6 行定义了一个虚属性。这个虚属性是一个自动实现的属性。类 `Test` 的派生类 `Test1` 是一个抽象类，这是因为要在其中定义抽象属性 `A`。抽象属性 `A` 对基类中的虚属性 `A` 进行了重写，然后它又被标记为抽象的，它的两个访问器都没有实现。具体的属性的实现由类 `Test1` 的派生类 `Test2` 里面的重写属性进行实现。当第 46 行由 `Test` 类的引用调用 `A` 属性时，将实现多态，最终类 `Test2` 中的属性 `A` 被调用。这段代码的执行结果如下：

```
属性 A 的值是: 31
```

如果在继承链中，某个中间类的重写属性被标记为 `sealed`，则它的派生类将不能再对此属性进行重写。

下面是代码实例：

```

001 using System;
002 namespace Polymorphism1
003 {
004     abstract class Test
005     {
006         public abstract int A
007         {
008             get;
009             set;
010         }
011     }
012     class Test1 : Test
013     {
014         private int a;
015         public sealed override int A
016         {
017             get
018             {
019                 return a;
020             }
021             set
022             {
023                 a = value;

```

```

024         }
025     }
026     public Test1()
027     {
028         this.a = 21;
029     }
030 }
031 class Test2 : Test1
032 {
033     public override int A
034     {
035         get
036         {
037             return base.A;
038         }
039         set
040         {
041             base.A = value;
042         }
043     }
044 }
045 class Program
046 {
047     static void Main(string[] args)
048     {
049         Test t = new Test2();
050         Console.WriteLine("属性 A 的值是: {0}", t.A);
051         Console.ReadKey();
052     }
053 }
054 }

```

第 4 行代码定义了一个抽象类，然后在其中定义了一个抽象的属性。类 Test 的派生类重写了基类的抽象属性，并且用 sealed 关键字对重写属性进行了密封。类 Test2 从类 Test1 继承，它定义了一个重写属性意图对它的基类的属性再次进行重写。但是它的基类的属性已经被定义成了一个密封属性，因此这段代码会报错，错误信息如下：

错误 1 “Polymorphism1.Test2.A.get”：继承成员“Polymorphism1.Test1.A.get”是 sealed，无法进行重写 E:\书稿 new\书稿源码\第十一章\Polymorphism1\Program.cs 35 13 Polymorphism1

错误 2 “Polymorphism1.Test2.A.set”：继承成员“Polymorphism1.Test1.A.set”是 sealed，无法进行重写 E:\书稿 new\书稿源码\第十一章\Polymorphism1\Program.cs 39 13 Polymorphism1

如果去掉类 Test1 中的属性的 sealed 关键字，这段代码的执行结果如下：

属性 A 的值是：21

索引器也包括 get 访问器和 set 访问器，它的使用规则和属性是相同的。标记为 override 的成员不能再标记为 new，因为 override 是重写，代表着继承性和多态。而 new 关键字则表示隐藏，它和 override 的作用相违背。将 virtual 和 override 联用也不行，因为 override 隐式代表成员就是虚成员。因此，如果同

时使用 new 和 override 标记或者 virturl 和 override 标记，将会得到错误报告，错误信息如下：

错误 1 标记为 override 的成员“Polymorphism1.Test2.A”不能标记为 new 或 virtual E:\书稿 new\书稿源码\第十一章\Polymorphism1\Program.cs 33 33 Polymorphism1

11.9 事件的多态性

在一个类中的事件也可以用 virtual 关键字将它定义成一个虚事件，然后在派生类中使用 override 关键字重写它。下面是代码实例：

```

001 using System;
002 namespace Polymorphism1
003 {
004     public delegate void D();
005     class Test
006     {
007         protected virtual event D myEvent1;
008         public Test()
009         {
010             myEvent1 += this.Show1;
011             myEvent1 += this.Show2;
012         }
013         public void Show1()
014         {
015             Console.WriteLine("类 Test 中的 Show1 方法被调用");
016         }
017         public void Show2()
018         {
019             Console.WriteLine("类 Test 中的 Show2 方法被调用");
020         }
021         public virtual void Invoke1()
022         {
023             Console.WriteLine("类 Test 中的 Invoke1 方法被调用");
024             myEvent1();
025         }
026     }
027     class Test1 : Test
028     {
029         protected override event D myEvent1;
030         public void Show3()
031         {
032             Console.WriteLine("类 Test1 中的 Show3 方法被调用");
033         }
034         public void Show4()
035         {
036             Console.WriteLine("类 Test1 中的 Show4 方法被调用");
037         }
038         public Test1()

```

```

039         {
040             myEvent1 += this.Show3;
041             myEvent1 += this.Show4;
042         }
043         public override void Invoke1()
044         {
045             Console.WriteLine("类 Test1 中的 Invoke1 方法被调用");
046             myEvent1();
047         }
048     }
049     class Program
050     {
051         static void Main(string[] args)
052         {
053             Test t = new Test1();
054             t.Invoke1();
055             Console.ReadKey();
056         }
057     }
058 }

```

代码的第 4 行定义了一个委托类型。在基类中，第 7 行定义了一个事件，为了派生类能够重写它，它的访问权限定义为 `protected`。因为不能访问就不能重写，所以它不能定义成 `private` 的。第 8 行的构造方法中为事件的调用列表中添加了两个本类的实例方法。第 21 行的方法中调用了事件。这个方法定义成虚方法的作用是实现在多态，根据不同的运行时类型调用不同类中的事件。

第 27 行开始是一个派生类，第 29 行重写了基类的事件。第 38 行的构造方法中将派生类的两个实例方法添加进了重写事件的调用列表中。第 43 行是一个重写方法，它也用来调用事件。第 53 行定义了一个基类的引用指向了派生类的实例。当它调用 `Invoke1` 方法的时候，实际调用的是派生类中的重写方法。下面首先来看执行结果，如下：

```

类 Test1 中的 Invoke1 方法被调用
类 Test 中的 Show1 方法被调用
类 Test 中的 Show2 方法被调用
类 Test1 中的 Show3 方法被调用
类 Test1 中的 Show4 方法被调用

```

可能看到运行结果会感到很奇怪，为什么重写事件中只添加了派生类的两个方法，但是实际执行的时候，基类中的两个方法也被执行了呢？这是因为重写事件不定义新事件，它只对被重写的事件的访问器进行了专有化的实现。当创建派生类的实例的时候，首先会调用基类的方法，这样被重写的事件背后的委托就会将基类的两个方法加入列表。当逻辑返回执行派生类的构造方法的方法体中的语句时，又会把派生类的两个方法加入了同一个委托的调用列表。这就导致了四个方法同时被调用。

重写事件必须和被重写事件具有完全一样的访问权限修饰关键字，以及类型和名称。将前面代码中基类的被重写事件的访问权限定义为 `public` 的，而派生类中的重写事件不改变访问权限关键字。这时编译代码将会看到错误信息，如下：

```

错误 1    “ Polymorphism1.Test1.myEvent1 ” : 当 重 写 “ public ” 继 承 成 员
“Polymorphism1.Test.myEvent1” 时，无法更改访问修饰符
H:\书稿 new\书稿源码\第十一章
\Polymorphism1\Program.cs    29    36    Polymorphism1

```


当用 abstract 关键字修饰事件时，则称这个事件是抽象事件，抽象事件所在的类必须定义成抽象的。下面是代码实例：

```
001     using System;
002     namespace Polymorphism1
003     {
004         public delegate void D();
005         abstract class Test
006         {
007             protected abstract event D myEvent1;
008             public Test()
009             {
010                 myEvent1 += this.Show1;
011                 myEvent1 += this.Show2;
012             }
013             public void Show1()
014             {
015                 Console.WriteLine("类 Test 中的 Show1 方法被调用");
016             }
017             public void Show2()
018             {
019                 Console.WriteLine("类 Test 中的 Show2 方法被调用");
020             }
021             public virtual void Invoke1()
022             {
023                 Console.WriteLine("类 Test 中的 Invoke1 方法被调用");
024                 //myEvent1();
025             }
026         }
027         class Test1 : Test
028         {
029             protected override event D myEvent1;
030             public void Show3()
031             {
032                 Console.WriteLine("类 Test1 中的 Show3 方法被调用");
033             }
034             public void Show4()
035             {
036                 Console.WriteLine("类 Test1 中的 Show4 方法被调用");
037             }
038             public Test1()
039             {
040                 myEvent1 += this.Show3;
041                 myEvent1 += this.Show4;
042             }
043         }
044     }
```

```

043         public override void Invoke1()
044         {
045             Console.WriteLine("类 Test1 中的 Invoke1 方法被调用");
046             myEvent1();
047         }
048     }
049     class Program
050     {
051         static void Main(string[] args)
052         {
053             Test t = new Test1();
054             t.Invoke1();
055             Console.ReadKey();
056         }
057     }
058 }

```

这段代码将第 7 行的事件成员定义为抽象的，这时，它所在的类也必须定义成抽象的。派生类中第 29 行代码对它进行了重写。第 24 行代码被注释掉了，这是因为在基类中无法直接调用抽象成员。这段代码的执行结果如下：

```

类 Test1 中的 Invoke1 方法被调用
类 Test 中的 Show1 方法被调用
类 Test 中的 Show2 方法被调用
类 Test1 中的 Show3 方法被调用
类 Test1 中的 Show4 方法被调用

```

如果将第 24 行代码的注释去掉，则编译器会报错，如下：

```

错误 1    事件 "Polymorphism1.Test.myEvent1" 只能出现在 += 或 -= 的左边    H:\书稿 new\书稿源码\第十一章\Polymorphism1\Program.cs 24    13    Polymorphism1

```

在单独定义事件的访问器的时候，add 访问器和 remove 访问器必须同时存在。下面看代码实例：

```

001     using System;
002     namespace Polymorphism1
003     {
004         public delegate void D();
005         class Test
006         {
007             protected D myD1;
008             protected virtual event D myEvent1
009             {
010                 add
011                 {
012                     myD1 += value;
013                 }
014             }
015             public Test()
016             {

```

```
017         myEvent1 += this.Show1;
018         myEvent1 += this.Show2;
019     }
020     public void Show1()
021     {
022         Console.WriteLine("类 Test 中的 Show1 方法被调用");
023     }
024     public void Show2()
025     {
026         Console.WriteLine("类 Test 中的 Show2 方法被调用");
027     }
028     public virtual void Invoke1()
029     {
030         Console.WriteLine("类 Test 中的 Invoke1 方法被调用");
031         myD1();
032     }
033 }
034 class Test1 : Test
035 {
036     protected override event D myEvent1;
037     public void Show3()
038     {
039         Console.WriteLine("类 Test1 中的 Show3 方法被调用");
040     }
041     public void Show4()
042     {
043         Console.WriteLine("类 Test1 中的 Show4 方法被调用");
044     }
045     public Test1()
046     {
047         myEvent1 += this.Show3;
048         myEvent1 += this.Show4;
049     }
050     public override void Invoke1()
051     {
052         Console.WriteLine("类 Test1 中的 Invoke1 方法被调用");
053         myEvent1();
054     }
055 }
056 class Program
057 {
058     static void Main(string[] args)
059     {
060         Test t = new Test1();
```

```

061         t.Invoke1();
062         Console.ReadKey();
063     }
064 }
065 }

```

这段代码的第 8 行以定义 add 访问器的方式定义了一个事件成员，事件成员是一个虚事件成员。在派生类中的第 36 行对其进行重写操作。但是在编译代码的时候，编译器会报高错误信息。如下：

错误 1 “Polymorphism1.Test.myEvent1”：事件属性必须同时具有 add 和 remove 访问器 E:\书稿 new\书稿源码\第十一章\Polymorphism1\Program.cs 8 35 Polymorphism1

这是因为，和属性不同，在定义事件的访问器的时候，add 访问器和 remove 访问器必须同时进行定义。下面对代码进行修改，如下：

```

001 using System;
002 namespace Polymorphism1
003 {
004     public delegate void D();
005     class Test
006     {
007         protected D myD1;
008         protected virtual event D myEvent1
009         {
010             add
011             {
012                 myD1 += value;
013             }
014             remove
015             {
016                 myD1 -= value;
017             }
018         }
019         public Test()
020         {
021             myEvent1 += this.Show1;
022             myEvent1 += this.Show2;
023         }
024         public void Show1()
025         {
026             Console.WriteLine("类 Test 中的 Show1 方法被调用");
027         }
028         public void Show2()
029         {
030             Console.WriteLine("类 Test 中的 Show2 方法被调用");
031         }
032         public virtual void Invoke1()
033         {

```

```
034         Console.WriteLine("类 Test 中的 Invoke1 方法被调用");
035         myD1();
036     }
037 }
038 class Test1 : Test
039 {
040     protected D myD2;
041     protected override event D myEvent1
042     {
043         add
044         {
045             myD2 += value;
046         }
047         remove
048         {
049             myD2 -= value;
050         }
051     }
052     public void Show3()
053     {
054         Console.WriteLine("类 Test1 中的 Show3 方法被调用");
055     }
056     public void Show4()
057     {
058         Console.WriteLine("类 Test1 中的 Show4 方法被调用");
059     }
060     public Test1()
061     {
062         myEvent1 += this.Show3;
063         myEvent1 += this.Show4;
064     }
065     public override void Invoke1()
066     {
067         Console.WriteLine("类 Test1 中的 Invoke1 方法被调用");
068         myD2();
069     }
070 }
071 class Program
072 {
073     static void Main(string[] args)
074     {
075         Test t = new Test1();
076         t.Invoke1();
077         Console.ReadKey();
```

```

078         }
079     }
080 }

```

本段程序中，在第 8 行的代码中将事件成员的 add 访问器和 remove 访问器都进行了定义。因为单独定义了访问器，所以在第 32 行的虚方法中需要调用自定义的委托类型引用 myD1。在派生类中，第 40 行代码定义了一个委托类型的引用 myD2，第 41 行代码定义了带有访问器的事件对基类中的事件进行重写。当第 76 行调用 Invoke1 方法的时候，实际上，myD2 事件将被调用。首先来看执行结果：

```

类 Test1 中的 Invoke1 方法被调用
类 Test 中的 Show1 方法被调用
类 Test 中的 Show2 方法被调用
类 Test1 中的 Show3 方法被调用
类 Test1 中的 Show4 方法被调用

```

得到这个结果可能会感到出乎意料，因为在这段代码中定义了两个委托，实际调用的是派生类中的委托。但是这个委托只在第 60 行的构造方法中添加了两个方法进入调用列表。那么基类中的两个方法是怎么执行的呢？下面修改代码，将第 68 行的代码改为调用基类的委托 myD1。下面来看执行结果：

```

类 Test1 中的 Invoke1 方法被调用

```

当打印出这行字符串后，就会抛出异常信息，如下：

```

未将对象引用设置到对象的实例。

```

也就是说，当派生类中重写基类的事件时，基类定义的委托没有实例化。而是将代码中对基类的委托的添加操作都挪到了派生类中进行。实际起作用的是派生类中的委托。下面再将 Main 方法中的代码修改如下：

```

Test t = new Test();
t.Invoke1();

```

这时，基类的引用指向了基类的实例，然后调用的肯定是基类中的 Invoke1 方法。这时委托 myD1 被调用了。但这时结果没有抛出异常。它是一个正常的结果，如下：

```

类 Test 中的 Invoke1 方法被调用
类 Test 中的 Show1 方法被调用
类 Test 中的 Show2 方法被调用

```

由此可以推断，在事件的继承和重写过程中，实际上在重写链中，只有一个事件成员会起作用，它根据多态的实现以及运行时类型对其进行方法的添加操作。

事件成员也可以重写后将其定义为抽象的，它依靠派生类对其进行重写。代码如下：

```

001 using System;
002 namespace Polymorphism1
003 {
004     public delegate void D();
005     class Test
006     {
007         protected virtual event D myEvent1;
008         public Test()
009         {
010             myEvent1 += this.Show1;
011             myEvent1 += this.Show2;
012         }
013         public void Show1()
014         {

```

```
015         Console.WriteLine("类 Test 中的 Show1 方法被调用");
016     }
017     public void Show2()
018     {
019         Console.WriteLine("类 Test 中的 Show2 方法被调用");
020     }
021     public virtual void Invoke1()
022     {
023         Console.WriteLine("类 Test 中的 Invoke1 方法被调用");
024         myEvent1();
025     }
026 }
027 abstract class Test1 : Test
028 {
029     protected abstract override event D myEvent1;
030     public void Show3()
031     {
032         Console.WriteLine("类 Test1 中的 Show3 方法被调用");
033     }
034     public void Show4()
035     {
036         Console.WriteLine("类 Test1 中的 Show4 方法被调用");
037     }
038     public Test1()
039     {
040         myEvent1 += this.Show3;
041         myEvent1 += this.Show4;
042     }
043     public override void Invoke1()
044     {
045         Console.WriteLine("类 Test1 中的 Invoke1 方法被调用");
046         //myEvent1();
047     }
048 }
049 class Test2 : Test1
050 {
051     protected override event D myEvent1;
052     public void Show5()
053     {
054         Console.WriteLine("类 Test2 中的 Show5 方法被调用");
055     }
056     public void Show6()
057     {
058         Console.WriteLine("类 Test2 中的 Show6 方法被调用");
```

```

059         }
060         public Test2()
061         {
062             myEvent1 += this.Show5;
063             myEvent1 += this.Show6;
064         }
065         public override void Invoke1()
066         {
067             Console.WriteLine("类 Test2 中的 Invoke1 方法被调用");
068             myEvent1();
069         }
070     }
071     class Program
072     {
073         static void Main(string[] args)
074         {
075             Test t = new Test2();
076             t.Invoke1();
077             Console.ReadKey();
078         }
079     }
080 }

```

这段代码从总体上共分为 3 个类，它们依次进行派生。基类中定义的事件是一个虚事件，基类也不是一个抽象类。基类的派生类 Test1 定义为一个抽象类，在它里面重写了基类的虚事件，并且将这个事件标记为抽象的。从类 Test1 派生的类 Test2 又对 Test1 中的抽象事件进行了重写，而且在类 Test2 中又定义了两个实例方法，并且在构造方法中将它们加入到了事件的调用列表中。注意第 46 行被整个注释掉了，如果不将它注释，编译器就会报错。因为它调用了抽象类中的抽象事件。这段代码的执行结果如下：

```

类 Test2 中的 Invoke1 方法被调用
类 Test 中的 Show1 方法被调用
类 Test 中的 Show2 方法被调用
类 Test1 中的 Show3 方法被调用
类 Test1 中的 Show4 方法被调用
类 Test2 中的 Show5 方法被调用
类 Test2 中的 Show6 方法被调用

```

根据运行时类型，实际上最后起作用的是派生类 Test2 中隐式生成的委托。而它的直接父类和间接父类中的隐式委托都将被作为重写使用，本身都将是一个空引用。

使用 sealed 关键字也可以将事件定义成密封的，从而不允许派生类对它进行重写。下面是代码实例：

```

001 using System;
002 namespace Polymorphism1
003 {
004     public delegate void D();
005     class Test
006     {
007         protected virtual event D myEvent1;

```



```
008     public Test()
009     {
010         myEvent1 += this.Show1;
011         myEvent1 += this.Show2;
012     }
013     public void Show1()
014     {
015         Console.WriteLine("类 Test 中的 Show1 方法被调用");
016     }
017     public void Show2()
018     {
019         Console.WriteLine("类 Test 中的 Show2 方法被调用");
020     }
021     public virtual void Invoke1()
022     {
023         Console.WriteLine("类 Test 中的 Invoke1 方法被调用");
024         myEvent1();
025     }
026 }
027 class Test1 : Test
028 {
029     protected sealed override event D myEvent1;
030     public void Show3()
031     {
032         Console.WriteLine("类 Test1 中的 Show3 方法被调用");
033     }
034     public void Show4()
035     {
036         Console.WriteLine("类 Test1 中的 Show4 方法被调用");
037     }
038     public Test1()
039     {
040         myEvent1 += this.Show3;
041         myEvent1 += this.Show4;
042     }
043     public override void Invoke1()
044     {
045         Console.WriteLine("类 Test1 中的 Invoke1 方法被调用");
046     }
047 }
048 class Test2 : Test1
049 {
050     protected override event D myEvent1;
051     public void Show5()
```

```
052     {
053         Console.WriteLine("类 Test2 中的 Show5 方法被调用");
054     }
055     public void Show6()
056     {
057         Console.WriteLine("类 Test2 中的 Show6 方法被调用");
058     }
059     public Test2()
060     {
061         myEvent1 += this.Show5;
062         myEvent1 += this.Show6;
063     }
064     public override void Invoke1()
065     {
066         Console.WriteLine("类 Test2 中的 Invoke1 方法被调用");
067         myEvent1();
068     }
069 }
070 class Program
071 {
072     static void Main(string[] args)
073     {
074         Test t = new Test2();
075         t.Invoke1();
076         Console.ReadKey();
077     }
078 }
079 }
```

这是一段出错的代码，出错的原因是，第 29 行的代码对基类的事件进行了重写，但同时它被标记为 sealed，它是一个密封事件。所以当第 50 行代码将它再次进行重写时，编译器就会报错，错误信息如下：

错误 1 “Polymorphism1.Test2.myEvent1”：继承成员“Polymorphism1.Test1.myEvent1”是 sealed，无法进行重写 E:\书稿 new\书稿源码\第十一章\Polymorphism1\Program.cs 50 36 Polymorphism1

第十二章 结构

结构与类相似，它们都包含数据成员和函数成员。但是，结构是值类型，它分配在栈上。而类是引用类型，它分配在堆上。结构类型的变量直接就表示了其数据的存储，但是类类型的变量表示的是一个引用，它本身不包含数据的存储。它指向一个分配在堆内存上的实例。那么，什么时候需要结构，什么时候需要类类型呢？如果数据结构的规模比较小，适合使用结构；如果数据结构较大，则适合使用类。使用结构的时候，因为是存储在栈中，所以效率比较高，但是栈内存是有大小限制的。在 windows 系统下，一般是 2M。而堆内存的申请一般比较慢，但是它的空间比较大，一般由计算机的虚拟内存决定。

12.1 结构的定义

结构的定义和类很相似，类使用 `class` 关键字定义，而结构使用 `struct` 关键字定义。下面是代码实例：

```
001    using System;
002    namespace StructType
003    {
004        public struct Test
005        {
006            private int a;
007            public int A
008            {
009                get
010                {
011                    return a;
012                }
013                set
014                {
015                    a = value;
016                }
017            }
018            public void Show()
019            {
020                this.A = 33;
021                Console.WriteLine("结构体中的属性 A 的值是: {0}", this.A);
022            }
023        }
024        class Program
025        {
026            static void Main(string[] args)
027            {
028                Test t = new Test();
029                t.Show();
030                Console.ReadKey();
031            }
032        }
033    }
```

第 4 行代码开始就是一个结构的定义，结构的定义使用 `struct` 关键字。在 `struct` 关键字前面是结构的访

访问修饰符，结构的访问修饰符可以是 `public`，它代表着这个结构无访问限制。还可以是 `internal`，它代表这个结构只能被本程序集访问。结构体以一对大括号开始和结束，不用分号结尾。这都和类的定义一样。在结构体内定义结构的成员。第 6 行定义了一个字段，第 7 行定义了一个属性，它用来对字段进行存取操作。第 18 行定义了一个实例方法，它对属性进行赋值，然后打印属性的值。赋值操作引用了属性的 `set` 访问器；`Console.WriteLine` 方法引用了属性的 `get` 访问器。

第 28 行定义了一个结构的实例，创建结构的实例和创建类的实例语法是一样的。但是这时结构的数据是存储在栈中的。第 29 行用结构变量调用了结构的 `Show` 方法。这段代码的执行结果如下：

结构体中的属性 A 的值是：33

12.2 结构的构造方法和析构方法

在结构中，不允许定义无形参构造方法。但每个结构隐式定义了一个无形参的构造方法，它的作用是将结构的所有值类型字段设置为默认值；引用类型字段设置为 `null` 后的实例返回。下面是代码实例：

```

001    using System;
002    namespace StructType
003    {
004        public struct Test
005        {
006            private int a;
007            public int A
008            {
009                get
010                {
011                    return a;
012                }
013                set
014                {
015                    a = value;
016                }
017            }
018            public Test(int a)
019            {
020                this.a = a;
021            }
022            public void Show()
023            {
024                Console.WriteLine("结构体中的属性 A 的值是：{0}", this.A);
025            }
026        }
027        class Program
028        {
029            static void Main(string[] args)
030            {
031                Test t = new Test();
032                t.Show();
033                Test t1 = new Test(22);

```

```

034         t1.Show();
035         Console.ReadKey();
036     }
037 }
038 }

```

在这段代码中，第 18 行定义了一个带有一个形参的实例构造方法。和类不同的是，类中定义一个自定义的构造方法之后，默认的无参构造方法将不提供。而结构不同，不论是否有自定义的构造方法，结构的默认构造方法都隐式提供。第 31 行调用结构的无参构造方法创建了一个结构的实例。第 33 行代码调用自定义的构造方法创建了结构的一个实例。这段代码的执行结果如下：

结构体中的属性 A 的值是：0

结构体中的属性 A 的值是：22

当调用默认的无参构造方法的时候，因为字段 a 的类型是 int，所以它被初始化为 0。但是，结构的实例字段不允许使用初始值设定项。将上面代码中的第 6 行加上字段的初始值设定项，代码如下：

```
private int a = 33;
```

这时编译器会报告错误，错误信息如下：

```

错误 1    “StructType.Test.a”：结构中不能有实例字段初始值设定项 H:\书稿 new\书稿源码\第十二章\StructType\Program.cs    6    21    StructType

```

虽然结构中可以定义实例构造方法，但是结构中不允许定义析构方法。当结构中含有静态字段的时候，可以使用静态构造方法对静态字段进行初始化。下面是代码实例：

```

001     using System;
002     namespace StructType
003     {
004         public struct Test
005         {
006             private int a;
007             public static int b = 12;
008             public Test(int a)
009             {
010                 this.a = a;
011             }
012             static Test()
013             {
014                 b = 100;
015             }
016             public void Show()
017             {
018                 Console.WriteLine("结构体中的属性 A 的值是：{0}", this.a);
019             }
020             public static void Show1()
021             {
022                 Console.WriteLine("结构体中的静态字段 b 的值是：{0}", b);
023             }
024         }
025     }

```

```
026      {
027          static void Main(string[] args)
028          {
029              Test t1 = new Test(22);
030              t1.Show();
031              Test.Show1();
032              Console.ReadKey();
033          }
034      }
035  }
```

第 7 行代码定义了一个静态字段，虽然结构的实例字段不允许使用字段的初始值设定项。但是结构的静态字段是允许使用初始值设定项的。第 12 行代码定义了一个静态构造方法，可以看到，静态构造方法的定义也和类没有什么不同。静态构造方法为结构的静态字段进行了再次的赋值。对于静态字段 b 来说，对它进行的操作过程是这样的，当第 29 行定义了一个结构类型的实例的时候，就会触发静态构造方法的执行。但是在静态构造方法执行前，静态字段 b 首先被设置为默认值 0。然后执行它的初始值设定项，将它设置为 12。最后执行静态构造方法将它设置为 100。

如果没有创建结构的实例，而是像第 31 行代码那样直接调用结构的静态成员也会触发结构的静态构造方法的执行。这段代码的执行结果如下：

结构体中的属性 A 的值是：22

结构体中的静态字段 b 的值是：100

下面再来通过代码验证上面这一点，修改后的代码如下：

```
001  using System;
002  namespace StructType
003  {
004      public struct Test
005      {
006          private int a;
007          public static int b = 12;
008          public Test(int a)
009          {
010              this.a = a;
011          }
012          static Test()
013          {
014              Console.WriteLine("结构体中的静态构造方法执行");
015              b = 100;
016          }
017          public void Show()
018          {
019              Console.WriteLine("结构体中的属性 A 的值是：{0}", this.a);
020          }
021          public static void Show1()
022          {
023              Console.WriteLine("结构体中的静态方法 Show1 执行");
```

```

024     }
025 }
026 class Program
027 {
028     static void Main(string[] args)
029     {
030         Test.Show1();
031         Console.ReadKey();
032     }
033 }
034 }

```

在这段代码中去掉了第 21 行代码中 WriteLine 方法对结构的静态字段的调用。然后在第 30 行代码中直接调用了结构的静态方法。从而看出，不管是结构的静态字段还是方法，只要是结构的静态成员被调用了，就会触发结构的静态构造方法的执行。代码的执行结果如下：

结构体中的静态构造方法执行

结构体中的静态方法 Show1 执行

12.3 结构类型的默认值

与类一样，在结构创建实例的时候，它的字段将首先被设置为该字段所属类型的默认值。包括结构的实例字段和结构的静态字段。在引用结构的静态成员的时候，只有结构的静态字段被首先设置为该字段所属类型的默认值。这个默认值的操作是由内存管理器或垃圾回收器来进行的，它在将内存分配以供使用之前，将内存初始化为所有位归 0，这就是字段的默认值设置的原因。

前面接触过的所有简单数据类型从根本上说都是一种结构，因此，也可以使用构造方法创建它们的实例。实例代码如下：

```

001 using System;
002 namespace StructType1
003 {
004     class Program
005     {
006         static void Main(string[] args)
007         {
008             int _i = 0;
009             int i = new int();
010             char _c = '\x0000';
011             char c = new char();
012             bool _b = false;
013             bool b = new bool();
014             float _f = 0.0F;
015             float f = new float();
016             Console.Write("_i 的值是: {0} ", _i);
017             Console.Write("_c 的值是: {0} ", _c);
018             Console.Write("_b 的值是: {0} ", _b);
019             Console.Write("_f 的值是: {0} ", _f);
020             Console.WriteLine();
021             Console.Write(" i 的值是: {0} ", i);

```

```

022         Console.WriteLine(" c 的值是: {0} ", c);
023         Console.WriteLine(" b 的值是: {0} ", b);
024         Console.WriteLine(" f 的值是: {0} ", f);
025         Console.ReadKey();
026     }
027 }
028 }

```

代码从第 8 行起, 以使用=运算符和构造方法的两种方式对一个简单类型进行初始化操作。从第 16 行起, 打印这两种方法对简单类型进行初始化后的变量的值。从结果可以看出, 使用直接赋值方法和调用结构的默认构造方法对变量的初始化来说, 是一样的效果。代码的执行结果如下:

```

_i 的值是: 0 _c 的值是:    _b 的值是: False _f 的值是: 0
i 的值是: 0 c 的值是:    b 的值是: False f 的值是: 0

```

对于值类型来说, 都有一个隐式的无参构造方法可以对它进行初始化。下面演示的是枚举类型和可空类型的调用构造方法进行初始化的方式:

```

001 using System;
002 namespace StructType1
003 {
004     public enum Color {Red,Green,Blue}
005     class Program
006     {
007         static void Main(string[] args)
008         {
009             Color c = Color.Blue;
010             Color c1 = new Color();
011             int? i = 0;
012             int? j = new int?();
013             int? k = new int?(100);
014             Console.WriteLine("c 的值是: {0} ", c);
015             Console.WriteLine(" c1 的值是: {0} ", c1);
016             Console.WriteLine(" i 的值是: {0} ", i);
017             Console.WriteLine(" j 的值是: {0} ", j);
018             Console.WriteLine(" k 的值是: {0} ", k);
019             Console.WriteLine();
020             Console.ReadKey();
021         }
022     }
023 }

```

第 9 行代码定义了一个枚举类型的变量, 它可以采用=运算符的方式进行赋值, 也可以采用第 10 行代码那样采用默认的构造方法的方式进行赋值。第 11 行代码是对一个可空类型使用=运算符赋值, 第 12 行代码是调用可空类型默认构造方法进行赋值。可空类型除了有一个无参构造方法之外, 还有一个可以传递参数的构造方法。第 13 行代码采用传递一个值类型参数的方式对一个可空类型进行初始化。这段代码的执行结果如下:

```

c 的值是: Blue c1 的值是: Red i 的值是: 0 j 的值是:    k 的值是: 100

```

从执行结果中可以看到, 对于可空类型 j 来说, 如果调用其默认构造方法, 则它的默认值被设置为 null。

因此对于简单类型，当它在类中作为字段的时候，在其被设置为默认值的操作过程中，它们被设置的默认值如下：

- 对于 sbyte、byte、short、ushort、int、uint、long 和 ulong，默认值为 0。
- 对于 char，默认值为 '\x0000'。
- 对于 float，默认值为 0.0f。
- 对于 double，默认值为 0.0d。
- 对于 decimal，默认值为 0.0m。
- 对于 bool，默认值为 false。

实际上，它们被设置的默认值就是调用它们的默认无参构造方法时返回的 0 初始化实例。下面再来看结构类型，结构类型的默认值就是调用它的无参构造方法之后，构造方法返回的结构实例。这时，结构中的字段按其类型不同，各采用其类型的默认值。下面是代码实例：

```
001 using System;
002 namespace StructType1
003 {
004     public enum Color {Red,Green,Blue}
005     public struct S
006     {
007         public int a;
008         public Color c;
009         public bool b;
010         public static float f;
011         static S()
012         {
013             Console.WriteLine("结构的静态构造方法被调用");
014             f = 1.1f;
015         }
016     }
017     class Program
018     {
019         private S myS;
020         public void Show()
021         {
022             Console.WriteLine("字段 a 的值是:{0}", myS.a);
023             Console.WriteLine("字段 c 的值是:{0}", myS.c);
024             Console.WriteLine("字段 b 的值是:{0}", myS.b);
025             Console.WriteLine("静态字段 f 的值是:{0}", S.f);
026         }
027         static void Main(string[] args)
028         {
029             Program p = new Program();
030             p.Show();
031             S[] myArray = new S[3];
032             foreach (S s in myArray)
033             {
```

```

034         Console.WriteLine("字段 a 的值是:{0}", s.a);
035         Console.WriteLine("字段 c 的值是:{0}", s.c);
036         Console.WriteLine("字段 b 的值是:{0}", s.b);
037         Console.WriteLine("静态字段 f 的值是:{0}", S.f);
038     }
039     Console.ReadKey();
040 }
041 }
042 }

```

这段代码中的第 5 行定义了一个结构，在结构中为语法演示的简单起见，定义了 4 个 public 访问权限的字段，其中字段 f 为静态字段。第 11 行代码定义了一个静态构造方法，在静态构造方法中对静态字段赋值。打印一行字符串的目的是可以方便看出静态构造方法被调用过。第 17 行定义了一个类，它包含了入口点的 Main 方法。本身它也包含了一个定义的结构 S 类型的实例字段成员，字段成员没有初始值设定项。第 20 行的实例方法打印了结构类型字段 myS 包含的结构中的各个字段的值。下面来仔细的分析代码的调用过程，第 29 行代码定义了 Program 类的一个实例，在这个过程中，字段 myS 将首先被设置为默认值。如果字段 myS 是一个引用类型，那它将被设置为 null。但是它是一个结构类型，结构类型又是一个值类型。值类型都有一个默认无参构造方法，结构类型也有。因此字段 myS 的默认值就是它的默认构造方法返回的实例。它的所有字段按其类型，如果是引用类型则设置为 null，如果是值类型，则设置为其默认构造方法返回的实例。所以第 30 行调用 Show 方法将打印出结构的各个实例字段的默认值。但是其静态字段因为在调用前会自动调用其静态构造方法，所以其值为静态构造方法设置后的值。

第 31 行代码定义了一个结构类型的数组，第 31 行对其分配堆内存，这时每个数组成员都会被设置为默认值，每个数组成员的默认值都是结构的默认构造方法返回的实例。第 32 行代码使用循环语句打印结构各个字段的值。这段代码的执行结果如下：

```

字段 a 的值是:0
字段 c 的值是:Red
字段 b 的值是:False
结构的静态构造方法被调用
静态字段 f 的值是:1.1
字段 a 的值是:0
字段 c 的值是:Red
字段 b 的值是:False
静态字段 f 的值是:1.1
字段 a 的值是:0
字段 c 的值是:Red
字段 b 的值是:False
静态字段 f 的值是:1.1
字段 a 的值是:0
字段 c 的值是:Red
字段 b 的值是:False
静态字段 f 的值是:1.1

```

对于第 8 行的枚举字段来说，其默认值为其默认构造方法的返回实例，它也是一个 0 初始化实例。也就是说，c 的值将为 0。这个值将被隐式转换为枚举类型，值为 0 的枚举成员就是 Red，因此它的值将打印出 Red。

结构类型在被设置为默认值的过程中，它的默认构造方法将被调用。但是它的静态构造方法将不被触

发。下面是代码实例：

```
001 using System;
002 namespace StructType1
003 {
004     public struct S
005     {
006         public int a;
007         public static float f;
008         static S()
009         {
010             Console.WriteLine("结构的静态构造方法被调用");
011             f = 1.1F;
012         }
013     }
014     class Program
015     {
016         private S myS;
017         public void Show()
018         {
019             Console.WriteLine("字段 a 的值是:{0}", myS.a);
020         }
021         static void Main(string[] args)
022         {
023             Program p = new Program();
024             p.Show();
025             S[] myArray = new S[3];
026             int[] myInt = new int[3];
027             foreach (S s in myArray)
028             {
029                 Console.WriteLine("字段 a 的值是:{0}", s.a);
030             }
031             Console.WriteLine("数组 myInt 中的值是:");
032             foreach (int i in myInt)
033             {
034                 Console.WriteLine(i);
035             }
036             Console.ReadKey();
037         }
038     }
039 }
```

第 4 行开始定义的结构体中只包含两个字段成员，一个是实例字段，另一个是静态字段。第 8 行是静态构造方法，它对静态字段进行赋值，并打印一行字符串表示静态构造方法被调用过了。

第 14 行代码开始是一个类的定义，在类体中定义了一个结构类型的变量。第 17 行代码打印结构中字段的值。

第 23 行创建了一个类 Program 的实例，在这个过程中，结构类型的字段将被设置为默认值。结构中的字段按其类型被设置为各自类型的默认值。第 24 行代码调用 Show 方法打印结构内字段的值。在这个过程中，因为没有调用结构的静态字段，所以结构的静态构造方法不会被执行。

第 25 行定义的结构类型的数组，也是同样的情形。第 25 行代码执行完毕，数组的每个成员只进行了初始化，但是没有赋值。这时，每个成员的值都为默认值，也就是结构的默认构造方法返回的实例。但是它们的静态构造方法不被执行。

第 26 行定义了一个简单类型的数组，因为它也是一个值类型，因此可以只初始化但不赋值。这时它们的值就是默认构造方法返回的 0 初始化实例。因为数组只初始化了，但每个结构成员都没有赋值，所以编译器会有一个警告信息，如下：

警告 1 从未对字段“StructType1.Program.myS”赋值，字段将一直保持其默认值E:\书稿 new\书稿源码\第十二章\StructType1\Program.cs 16 19 StructType1

虽然有警告信息，但是代码会正常执行。执行结果如下：

```

字段 a 的值是:0
字段 a 的值是:0
字段 a 的值是:0
字段 a 的值是:0
数组 myInt 中的值是:
0
0
0

```

12.4 结构的继承性

结构是一种值类型，它从 System.ValueType 类隐式继承。但是结构不能指定其基类，也就是除了 System.ValueType 类是它的隐式继承的基类外，不能再指定其它基类。结构也不能是抽象的，它是隐式密封的。但是不能使用 sealed 关键字修饰它。简单来说，结构不能指定基类，不能从结构派生结构或其它类。看起来，结构仿佛就是独立的。下面是代码实例：

```

001  using System;
002  namespace StructType1
003  {
004      class Test
005      { }
006      public struct S:Test
007      {
008          protected int a;
009          private int b;
010          public S()
011          {
012              a = 100;
013              b = 200;
014          }
015          public S(int a, int b)
016          {
017              this.a = a;
018              this.b = b;
019          }

```

```

020         public void Show()
021         {
022             Console.WriteLine("字段 a 的值是: {0}, 字段 b 的值是: {1}", a, b);
023         }
024     }
025     class Program
026     {
027         static void Main(string[] args)
028         {
029             S myS = new S(100, 200);
030             myS.Show();
031             Console.ReadKey();
032         }
033     }
034 }

```

这段代码会报告三个错误，下面来分析这段代码，第 4 行代码定义一个类，类中没有定义任何成员。它只作为基类使用。第 6 行代码定义一个结构，然后定义它从 Test 类派生。因为指定了结构从其它基类派生，所以这里会报告一个错误。第 8 行代码定义了一个 protected 访问权限的字段，因为不能有任何结构和类能从结构派生，结构是隐式密封的，所以定义 protected 访问权限的字段会导致一个错误。代码中定义了两个构造方法，第 10 行是无参构造方法，第 15 行又定义了一个带有两个形参的构造方法。因为 C# 为结构隐式定义无参构造方法，因此不能为结构再次自定义无参构造方法。所以第 10 行代码将导致一个错误。编译器报告的错误信息如下：

```

错误 1    接口列表中的类型“Test”不是接口 H:\书稿 new\书稿源码\第十二章
\StructType1\Program.cs 6    21    StructType1
错误 2    “StructType1.S.a”：结构中已声明新的保护成员 H:\书稿 new\书稿源码\第十二章
\StructType1\Program.cs 8    23    StructType1
错误 3    结构不能包含显式的无参数构造函数 H:\书稿 new\书稿源码\第十二章
\StructType1\Program.cs 10   16    StructType1

```

其中错误 2 中提示的“结构中已声明新的保护成员”的意思就是结构中不允许定义保护访问权限的成员。与此相似的还有，结构中也不能定义 protected internal 访问权限的成员。

对于虚方法和重写的特性来说，因为不支持从结构中再次派生新的类或结构，所以不能用 virtual 关键字修饰结构中的成员。但是对于 override 关键字来说，它可以重写 System.ValueType 类中的虚成员。实例代码如下：

```

001     using System;
002     namespace StructType1
003     {
004         public struct S
005         {
006             private int a;
007             private int b;
008             public S(int a, int b)
009             {
010                 this.a = a;
011                 this.b = b;

```

```

012     }
013     public void Show()
014     {
015         Console.WriteLine("字段 a 的值是: {0}, 字段 b 的值是: {1}", a, b);
016     }
017     public override string ToString()
018     {
019         string s = string.Format("a:{0}, b:{1}", a, b);
020         return s;
021     }
022 }
023 class Program
024 {
025     static void Main(string[] args)
026     {
027         S myS = new S(100, 200);
028         myS.Show();
029         Console.WriteLine(myS.ToString());
030         Console.ReadKey();
031     }
032 }
033 }

```

这段代码的第 6 行和第 7 行定义了两个字段，第 8 行是实例构造方法，它使用两个参数初始化结构的字段。第 17 行定义了一个重写方法，因为结构隐式派生自基类 `System.ValueType`，所以它可以重写从它继承来的虚方法。这段代码的执行结果如下：

```
字段 a 的值是: 100, 字段 b 的值是: 200
```

```
a:100, b:200
```

因为第 27 行代码定义了一个本结构类型的变量，所以第 29 行发生的不是多态。它只是调用了一个自己内部的方法。下面将入口点方法中的代码进行修改，修改后的代码如下：

```

001     static void Main(string[] args)
002     {
003         ValueType v = new S(100, 200);
004         Console.WriteLine(v.ToString());
005         Console.ReadKey();
006     }

```

第 3 行代码是将结构的基类 `System.ValueType` 类型的变量指向了新创建的结构。这时发生的是一个隐式引用转换。当使用变量 `v` 调用虚方法 `ToString` 的时候，实际调用的是结构中重写的那个 `ToString` 方法。这时的执行结果如下：

```
a:100, b:200
```

实际上，将结果类型的实例隐式转换为 `System.ValueType` 类型，这就是一种装箱操作。将结构类型隐式转换为 `Object` 类型也是装箱操作。下面看代码实例：

```

001     using System;
002     namespace StructType1
003     {

```

```
004     public struct S
005     {
006         private int a;
007         private int b;
008         public S(int a, int b)
009         {
010             this.a = a;
011             this.b = b;
012         }
013         public void Show()
014         {
015             Console.WriteLine("字段 a 的值是: {0}, 字段 b 的值是: {1}", a, b);
016         }
017         public override string ToString()
018         {
019             string s = string.Format("a: {0}, b: {1}", a, b);
020             return s;
021         }
022         public void Set(int a, int b)
023         {
024             this.a = a;
025             this.b = b;
026         }
027     }
028     class Program
029     {
030         static void Main(string[] args)
031         {
032             S myS = new S(100, 200); //myS 在栈中
033             Console.WriteLine(myS.ToString());
034             object o = myS; //结构被复制到了堆中
035             Console.WriteLine(o.ToString());
036             S myS1 = (S)o; //在栈中又定义了一个结构
037             myS1.Set(11, 12); //栈中的结构发生了改变
038             Console.WriteLine(myS1.ToString());
039             Console.WriteLine(o.ToString()); //堆中的结构没有发生改变
040             Console.ReadKey();
041         }
042     }
043 }
```

这段代码的第 22 行又添加了一个实例方法用来设置结构的字段值。下面来分析装箱操作和取消装箱操作的过程。首先第 32 行代码定义了一个结构并进行了初始化, myS 结构会在栈中分配内存。因为自定义的结构 S 重写了从基类继承来的虚方法 ToString。所以第 33 行将打印自定义的结构中的字段的值。这个调用不是装箱后的多态调用, 因为调用的主体是结构类型的变量。第 34 行代码中, 将结构隐式转换为它的间

接基类 Object 类，因为它的直接基类 ValueType 派生于 Object，所以 ValueType 类是直接基类，Object 是间接基类。当这个隐式转换发生的时候，就是一个装箱操作。这时，这个结构被从栈中复制到了堆内存中。第 35 行打印结构字段值的时候，就是一次多态的调用。因为这是父类的引用指向了子类的实例，调用是由父类的引用发起的。

第 36 行代码是一次取消装箱转换，也称为拆箱转换。它是使用显式转换将 Object 类型转换为结构类型。这时发生的操作是，堆内存中的结构又被复制到了栈中。第 37 行代码修改了栈中结构的字段值，但是它不影响堆中已经装箱的结构值。因此，第 38 行代码将打印出新值，而第 39 行代码将打印原值。代码的执行结果如下：

```
a:100, b:200
a:100, b:200
a:11, b:12
a:100, b:200
```

还有一个易混淆的概念，就是未赋值的变量和自动初始化的区别。下面先看代码实例：

```
001 static void Main(string[] args)
002 {
003     S myS;
004     myS.Show();
005     S[] s = new S[2];
006     foreach (S i in s)
007     {
008         i.Show();
009     }
010     Console.ReadKey();
011 }
```

前面结构的定义没有进行修改，修改 Main 方法中为如上代码。第 3 行代码定义了一个结构类型的变量，然后调用其 Show 方法，在输入代码的时候，编译器都会有智能提示，看起来好像没有什么问题。但是在编译的时候就会报错，错误信息如下：

```
错误 1 使用了未赋值的局部变量“myS” H:\书稿 new\书稿源码\第十二章
\StructType1\Program.cs 33 13 StructType1
```

这是因为，这个结构根本没有赋值就进行了使用，它现在只不过是一个名称，但是还没有在栈中分配内存。但是，第 5 行代码定义了结构类型的数组，它包含两个成员，每个成员都是一个结构。看起来它也没有初始化，但是当将第 3 行和第 4 行注释掉后，代码会正确执行，执行结果如下：

```
字段 a 的值是: 0, 字段 b 的值是: 0
字段 a 的值是: 0, 字段 b 的值是: 0
```

这是因为，当将结构作为字段（不论是静态字段还是实例字段）或作为数组成员的时候，结构的初始化赋值都是自动调用其默认构造方法的。这个过程由编译器来完成，不用程序员来定义。

12.5 结构的 this 和 base 引用

结构的实例构造方法不能含有 base 初始值设定项。this 引用在结构内和在类内的含义也不同。在类的内部，this 是一个值类别，它代表对实例的引用。在类内不能改变它的值。但是，在结构内，this 代表一个结构类型的变量，它的值能够被改变。代码实例如下：

```
001 using System;
002 namespace StructType
003 {
004     public class Test
```



```

005     {
006         private int a;
007         public Test()
008         {
009             a = 10;
010         }
011         public Test(int a)
012         {
013             this.a = a;
014             this = new Test();
015         }
016         public void Show()
017         {
018             Console.WriteLine("结构体中的属性 A 的值是: {0}", this.a);
019         }
020     }
021     class Program
022     {
023         static void Main(string[] args)
024         {
025             Test t = new Test(30);
026             t.Show();
027             Console.ReadKey();
028         }
029     }
030 }

```

代码的第 4 行定义了一个类，类含有一个 int 类型的字段 a。第 7 行是无参构造方法，它对 a 赋值为 10。第 11 行代码是一个带有一个形参的构造方法，它将形参的值赋值给字段 a。第 14 行它对 this 引用进行了赋值，调用类的无参构造方法将一个新的实例赋值给 this 引用。但是，编译器会报告一个错误，如下：

错误 1 无法分配到“<this>”，因为它是只读的 H:\书稿 new\书稿源码\第十二章\StructType\Program.cs 14 13 StructType

这是因为，类中的 this 引用表示对实例的引用，它是一个值类别。更确切的说，它是只读的。不可对它再进行赋值操作。下面再看结构中的不同。代码实例如下：

```

001 using System;
002 namespace StructType1
003 {
004     public struct S
005     {
006         private int a;
007         public S(int a)
008         {
009             this = new S();
010             this.a = a;
011         }

```

```
012         public void Show()
013         {
014             Console.WriteLine("字段 a 的值是: {0}", a);
015         }
016         public void Set(int a)
017         {
018             this = new S();
019             this.a = a;
020         }
021         public void Set1(ref S s)
022         {
023             s = new S(1000);
024         }
025         public void Set2()
026         {
027             Set1(ref this);
028         }
029     }
030     class Program
031     {
032         static void Main(string[] args)
033         {
034             S myS = new S(100);
035             myS.Set(200);
036             myS.Show();
037             myS.Set2();
038             myS.Show();
039             Console.ReadKey();
040         }
041     }
042 }
```

结构中的 `this` 因为它所处的位置不同而有不同的意义。在结构的实例构造方法内, `this` 相当于一个 `out` 类型的结构类型变量。在结构的实例方法内, `this` 相当于一个 `ref` 类型的结构类型变量。所以可以对 `this` 进行赋值或修改操作, 还可以将它作为 `ref` 类型或 `out` 类型的参数传递。

首先来看本段代码中定义的这个结构, 第 6 行定义了一个实例字段 `a`。这也是这个结构中唯一的字段变量。第 7 行是一个带有一个形参的实例构造方法, 第 9 行调用了结构的无参构造方法对 `this` 进行了重新赋值。这时, `this` 表示的结构实例没有变化, 还是栈中的那块内存, 但是无参构造方法创建的结构实例被复制给了 `this`。然后对 `this` 的字段 `a` 进行赋值操作。

第 21 行代码是是一个实例方法, 它带有一个 `ref` 类型的结构 `S` 类型的形参, 在第 23 行对这个引用类型的形参进行赋值, 将一个新的结构实例复制给它。第 25 行代码定义的实例方法调用了第 21 行代码定义的方法。

下面来分析 `Main` 方法中各个操作的实际内容, 当第 34 行代码调用结构的带一个参数的构造方法来创建结构的实例时, 第 7 行代码被调用。这时, `this` 引用表示这个创建的结构变量 `myS`。在构造方法中, 调用结构的无参构造方法新建了一个结构的实例, 并将它复制给了 `this` 表示的 `myS`, 然后为字段 `a` 赋值。

第 35 行代码调用 Set 方法时，对 this 引用表示的结构类型的变量 myS 又进行了一次复制操作，它将一个新创建的结构复制给了 this。然后又对 this 的字段进行了赋值操作。第 36 行打印字段的值将打印出 200。第 37 行调用方法 Set2 的时候，Set2 方法又调用了 Set1 方法，并把 this 传递给了 Set1。这时的 this 还是表示的结构类型的变量 myS。在 Set1 方法内，又使用了一个新创建的结构对它进行复制操作。在创建新的结构的过程中又会调用带一个参数的构造方法，在构造方法内部又会创建一个新的结构，用它对 this 进行复制操作，然后修改字段 a 的值。第 38 行最后打印字段 a 的值，将打印出 1000。在整个过程里，需要注意，this 引用表示的始终是结构 myS，它没有改变，改变的是对它的一次次复制和对字段的赋值。这段代码的执行结果如下：

字段 a 的值是：200

字段 a 的值是：1000

还需要注意一点的是，在结构的实例构造方法中，在结构的所有字段被明确赋值前，不能调用结构的任何实例函数成员。下面是代码实例：

```
001 using System;
002 namespace StructType1
003 {
004     public struct S
005     {
006         private int a;
007         public int A
008         {
009             get
010             {
011                 return a;
012             }
013             set
014             {
015                 a = value;
016             }
017         }
018         public S(int a)
019         {
020             this.A = a;
021         }
022         public void Show()
023         {
024             Console.WriteLine("字段 a 的值是：{0}", a);
025         }
026     }
027     class Program
028     {
029         static void Main(string[] args)
030         {
031             S myS = new S(100);
032             myS.Show();
```

```

033         Console.ReadKey();
034     }
035 }
036 }

```

这段代码将会报告两个错误。在结构体中的第 7 行代码，定义了一个实例属性成员。它对字段 a 进行存取。第 18 行代码是一个实例构造方法，它带有一个形参。在方法体内，它对属性进行赋值。因为在结构的实例构造方法内，如果结构的所有字段成员没有完全赋值，则不能调用其实例成员函数。所以第 20 行调用属性的 set 访问器的时候就会出错。错误信息如下：

错误 1 在给“this”对象的所有字段赋值之前，无法使用该对象 H:\书稿 new\书稿源码\第十二章\StructType1\Program.cs 20 13 StructType1

错误 2 在控制返回调用方之前，字段“StructType1.S.a”必须被完全赋值 H:\书稿new\书稿源码\第十二章\StructType1\Program.cs 18 16 StructType1

对于结构来说，这样赋值会出错。但是对于类来说，则没有问题。所以这是结构独有的特殊性。如果要改正这个错误，则将构造方法中的语句修改为如下的代码：

```
this.a = a;
```

12.6 结构的赋值和依赖关系

前面已经接触过结构类型的赋值，结构类型的赋值实际上发生的是复制操作，它将创建另一个结构实例的一个副本。当将结构以参数的方式传递给方法的时候，也会发生复制操作。如果不想在传递参数的时候发生复制操作，可以采用 ref 和 out 形参以引用方式传递结构。下面是代码实例：

```

001 using System;
002 namespace StructType1
003 {
004     public struct S
005     {
006         private int a;
007         public S(int a)
008         {
009             this.a = a;
010         }
011         public int A
012         {
013             get { return a; }
014             set { a = value; }
015         }
016         public void Show()
017         {
018             Console.WriteLine("字段 a 的值是: {0}", a);
019         }
020     }
021     class Program
022     {
023         public void Test1(S s)
024         {
025             s.Show();

```

```

026     }
027     public void Test2(ref S s,int a)
028     {
029         s.A = a;
030     }
031     static void Main(string[] args)
032     {
033         S myS = new S(100);
034         myS.Show();
035         Program p = new Program();
036         p.Test1(myS);
037         p.Test2(ref myS, 2000);
038         p.Test1(myS);
039         Console.ReadKey();
040     }
041 }
042 }

```

代码第 4 行开始, 定义了一个结构体。第 6 行是结构的实例字段, 第 7 行是实例构造方法, 它用来对字段 a 进行初始化。第 11 行是结构的实例属性。它用来对字段作存取操作。第 16 行定义的实例方法用来打印字段 a 的值。

在 Program 类中, 定义了两个实例方法, Test1 方法以值形参的方法传入一个结构类型的变量。Test2 方法以引用形参的方法传递一个结构类型的变量。下面来看代码实际执行中的情况。

第 33 行定义了一个结构类型的变量并为其赋初值。然后调用它的 Show 方法打印字段成员的值, 这时将打印出 100。接下来实例化了一个 Program 类的变量 p, 然后 p 调用了 Program 类的成员方法 Test1, 并为其传入结构变量 myS, 因为 Test1 是按照值参数的方式传递进入的, 所以这时方法的参数 s 将是 myS 复制给它的一个副本。第 37 行代码, p 调用了成员方法 Test2, Test2 是按照引用参数的方法将 myS 传递进来的, 所以这时不发生结构的复制操作, 在 Test2 中对属性 A 进行赋值将直接影响结构变量 myS。第 38 行再次调用了成员方法 Test1, 这时依旧会发生复制操作。代码的执行结果如下:

```

字段 a 的值是: 100
字段 a 的值是: 100
字段 a 的值是: 2000

```

当结构中定义的属性或索引器是赋值的目标时, 引用它的表达式必须是变量。如果表达式为值类别, 则会发生错误。实例代码如下:

```

001 using System;
002 namespace StructType1
003 {
004     struct Point
005     {
006         int x; //代表点的横坐标
007         int y; //代表点的纵坐标
008         public int X //对字段 x 进行存取的属性
009         {
010             get { return x; }
011             set { x = value; }

```

```
012     }
013     public int Y //对字段 y 进行存取的属性
014     {
015         get { return y; }
016         set { y = value; }
017     }
018     public Point(int x, int y) //构造方法
019     {
020         this.x = x;
021         this.y = y;
022     }
023 }
024 struct Segment
025 {
026     Point a; //线段中包含的第一个端点
027     Point b; //线段中包含的第二个端点
028     public Point A //Point 类型的属性, 对字段 a 进行存取
029     {
030         get { return a; }
031         set { a = value; }
032     }
033     public Point B //Point 类型的属性, 对字段 b 进行存取
034     {
035         get { return b; }
036         set { b = value; }
037     }
038     public Segment(Point a, Point b) //构造方法, 对字段 a 和 b 赋值
039     {
040         this.a = a;
041         this.b = b;
042     }
043 }
044 class Program
045 {
046     static void Main(string[] args)
047     {
048         Point a = new Point();
049         a.X = 100;
050         a.Y = 200;
051         Segment s = new Segment();
052         s.A = new Point(30, 30);
053         s.B = new Point(60, 60);
054         Console.ReadKey();
055     }
```

```
056     }
057 }
```

代码中定义的两个结构是包含关系，其中结构 Point 代表一个点，它有两个字段 x 和 y 代表点的两个坐标。结构 Segment 代表一条线段，它包含两个 Point 类型的字段，代表线段的两个端点。两个结构中的属性都对本结构的字段进行存取操作。下面来看 Main 方法中的代码，第 48 行定义了一个 Point 类型的变量并进行赋值。第 49 行和第 50 行调用 Point 类型的变量 a 的两个属性成员进行赋值操作。当结构的属性作为赋值的目标时，引用它的表达式必须是变量，这里 a 就是一个结构类型的变量，所以这里会正常执行。第 51 行代码定义了一个 Segment 类型的变量并实例化。第 52 行和第 53 行对它的两个属性进行赋值操作。因为 s 为变量，所以这里也会正常执行。

下面修改 Main 方法中的代码如下：

```
001 static void Main(string[] args)
002 {
003     Point a = new Point();
004     a.X = 100;
005     a.Y = 200;
006     Segment s = new Segment();
007     s.A = new Point(30, 30);
008     s.B = new Point(60, 60);
009     s.A.X = 350;
010     s.B.Y = 400;
011     Console.ReadKey();
012 }
```

这段代码的第 9 行和第 10 行将会引发两个错误。原因是表达式“s.A”和“s.B”不是一个变量。为什么这样说呢？以表达式 s.A 来说，它调用的是属性 A 的 get 访问器。实际上，属性是一个方法，它不是变量，它不表示存储位置。虽然在它的 get 访问器中有“return a;”语句，但是，这个副本必须存储在内存中时，才可以对它操作。“return a;”可以看做一种中间表达式的结果生成，它是值类型，因此不能使用它引用属性进行赋值操作。这段代码的错误信息如下：

```
错误 1 无法修改“StructType1.Segment.A”的返回值，因为它不是变量 E:\书稿new\书稿源码\第
十二章\StructType1\Program.cs 54 13 StructType1
错误 2 无法修改“StructType1.Segment.B”的返回值，因为它不是变量 E:\书稿new\书稿源码\第
十二章\StructType1\Program.cs 55 13 StructType1
```

如果将结构更改为类类型，则不会报错。原因是，类类型返回的是引用，实际的对象保存在堆中。所以它不是对引用变量进行赋值，而是对堆中的实例进行赋值。

结构不能依赖于自身的定义，这一点是和类类型相同的。下面看代码实例：

```
001 using System;
002 namespace StructType
003 {
004     public class Test
005     {
006         private Test a;
007         private int b;
008         public Test()
009         {
010             b = 10;
```

```

011         a = new Test();
012     }
013     public void Show()
014     {
015         Console.WriteLine("字段 b 的值是: {0}", this.b);
016     }
017 }
018 class Program
019 {
020     static void Main(string[] args)
021     {
022         Test t = new Test();
023         t.Show();
024         Console.ReadKey();
025     }
026 }
027 }

```

代码的第 4 行定义了一个类 Test，在类体中又包含了一个 Test 类型的引用变量。第 8 行的构造方法中对变量 a 进行实例的创建。当在第 22 行创建 Test 类的一个实例时，就会抛出异常，异常信息如下：

“System.StackOverflowException”类型的未经处理的异常在 StructType.exe 中发生
确保您没有无限循环或无限递归。

这是一个栈溢出异常，原因是出现了定义上的循环。类 Test 的定义依赖于字段 a，而字段 a 在初始化时又依赖于类 Test。这就导致了依赖关系上的循环。如果不在构造方法中实例化 a，则不会出现问题，因为 a 将被设置为默认值 null。

对于结构来说，情况就更严重一些，因为结构是值类型，它直接代表存储结构。看下面的代码：

```

001 using System;
002 namespace StructType
003 {
004     public struct Test
005     {
006         private Test a;
007         private int b;
008         public Test(int b)
009         {
010             this.b = b;
011         }
012         public void Show()
013         {
014             Console.WriteLine("字段 b 的值是: {0}", this.b);
015         }
016     }
017     class Program
018     {
019         static void Main(string[] args)

```



```

020      {
021          Test t = new Test();
022          t.Show();
023          Console.ReadKey();
024      }
025  }
026  }

```

这段代码对第 4 行进行了修改，原来的类成为了一个结构。因为结构不能自定义无参构造方法，所以在定义的构造方法中添加一个形参，用它对字段 b 进行赋值。因为结构是值类型，所以它的变量直接表示了存储位置。因此第 6 行不再是类中表示的 null，它将被设置为默认值。这就导致了结构的定义依赖它，它又依赖于结构的定义。因此编译器会报错，信息如下：

错误 1 “StructType.Test”类型的结构成员“StructType.Test.a”在结构布局中导致循环 E:\书稿 new\书稿源码\第十二章\StructType\Program.cs 6 22 StructType

不同的结构定义之间也不能形成依赖上的循环，代码实例如下：

```

001 using System;
002 namespace StructType
003 {
004     public struct Test
005     {
006         private Test1 t1;
007     }
008     public struct Test1
009     {
010         private Test2 t2;
011     }
012     public struct Test2
013     {
014         private Test t;
015     }
016     class Program
017     {
018         static void Main(string[] args)
019         {
020             Console.ReadKey();
021         }
022     }
023 }

```

本段代码中定义了三个结构，在三个结构的结构体中，依次含有下一个结构的实例字段变量。这样它们之间就是一种循环的依赖关系。这在结构的定义中是不允许的。编译器会报告错误，错误信息如下：

错误 1 “StructType.Test1”类型的结构成员“StructType.Test.t1”在结构布局中导致循环 E:\书稿 new\书稿源码\第十二章\StructType\Program.cs 6 23 StructType

错误 2 “StructType.Test2”类型的结构成员“StructType.Test1.t2”在结构布局中导致循环 E:\书稿 new\书稿源码\第十二章\StructType\Program.cs 10 23 StructType

错误 3 “StructType.Test”类型的结构成员“StructType.Test2.t”在结构布局中导致循环 E:\

书稿 new\书稿源码\第十二章\StructType\Program.cs 14 22 StructType

12.7 分部结构

与类相似，结构也可以是分部的。定义分部结构也使用 `partial` 关键字。下面是代码实例：

```
001 using System;
002 namespace StructType
003 {
004     public partial struct Test
005     {
006         private int a;
007         public Test(int a)
008         {
009             this.a = a;
010         }
011         partial void Show();
012         public void Invoke()
013         {
014             Show();
015         }
016     }
017     public partial struct Test
018     {
019         partial void Show()
020         {
021             Console.WriteLine("字段 a 的值是 {0}", a);
022         }
023     }
024     class Program
025     {
026         static void Main(string[] args)
027         {
028             Test t = new Test(100);
029             t.Invoke();
030             Console.ReadKey();
031         }
032     }
033 }
```

这段代码中的第 4 行和第 17 行定义了分部结构 `Test` 的两个部分。可以看到，分部结构的定义和分部类的定义是相似的。在第 11 行定义了一个分部方法，分部方法的使用也和类中的使用是相同的。在分部结构的另一部分中，第 19 行定义了分部方法的实现。第 12 行定义的实例方法用来调用分部方法。这段代码的执行结果如下：

字段 a 的值是 100

12.8 何时使用结构

因为结构是值类型，所以它的成员的生存期和结构变量的生存期相同。当结构变量存在时，则它的成员也存在；如果结构变量消失，则它的成员也随之消失。对于结构的初始赋值状态来说，如果结构变量是

初始已赋值的，则它的成员也是初始已赋值的；如果结构变量是初始未赋值状态，则它的成员也是初始未赋值的。既然它和类这么相似，那么在实际使用中，对两者的选择如何权衡呢？下面先看一个代码实例：

```
001 using System;
002 namespace StructType1
003 {
004     class T
005     {
006         private int a;
007         public void Show()
008         {
009             Console.WriteLine(a);
010         }
011         public T(int a)
012         {
013             this.a = a;
014         }
015     }
016     class Program
017     {
018         static void Main(string[] args)
019         {
020             T[] myT = new T[100];
021             for (int i = 0; i < 100; i++)
022             {
023                 myT[i] = new T(10);
024             }
025             Console.ReadKey();
026         }
027     }
028 }
```

这段代码定义了一个类，类很简单，对于其数据成员来说，只包含了一个 int 类型的字段。当在 Main 方法中使用这个类的时候，第 20 行代码先创建了一个 T 类型的数组，为其在堆中分配了 100 个成员的内存。第 21 行调用类的构造方法对每个数组成员进行初始化。现在来看一下其内存的分配，首先变量 myT 分配在了栈中，它指向了堆中的一块内存，这块内存中包含 100 个 T 类型的变量，当第 21 行使用 for 语句对每个 T 类型的变量进行初始化的时候，每个数组元素又指向了堆中的另外一块地方，每个这样的地方都是一个 T 类型的实例。所以在这段代码中，包含栈中的元素，共有 102 块内存空间被使用。

下面将第 4 行代码定义的类更改为结构，其余的代码不用进行修改。这时第 20 行代码执行的时候，myT 就代表了栈中的存储空间，在这个存储空间中是一个数组，数组共包含 100 个成员。当第 21 行使用 for 语句初始化数组的时候，每次循环都调用了结构的构造方法。和类的构造方法不同，类的构造方法在堆中创建实例，然后返回这个实例的引用。但是结构的构造方法通常是在栈中的一个临时位置创建实例，然后根据需要把这个实例复制过来。所以第 21 行的语句执行完毕的时候，每个结构实例都内联的存储在了 myT 所在的那块内存中了。整段代码最后只有一块内存空间被使用。因为这块内存空间还分配在栈中，所以它的效率要比类高。在定义小型的具有值语义的数据结构的时候，通常将它们定义成结构。什么是具有值语义呢？就是它的成员都是值类型。再看下面的代码：

```
001 using System;
002 namespace StructType1
003 {
004     class T1
005     {
006         public int a;
007     }
008     struct T
009     {
010         private T1 t;
011         public void Show()
012         {
013             Console.WriteLine(t.a);
014         }
015         public T(int a)
016         {
017             t = new T1();
018         }
019     }
020     class Program
021     {
022         static void Main(string[] args)
023         {
024             T[] myT = new T[100];
025             for (int i = 0; i < 100; i++)
026             {
027                 myT[i] = new T(10);
028             }
029             Console.ReadKey();
030         }
031     }
032 }
```

和前面例子不同的是，在结构 T 内含有一个类类型的变量 t。下面再来看代码执行过程中的内存分配，当第 24 行代码执行完毕的时候，栈中有一块内存被分配给数组。但是第 27 行代码每执行一次，内存就不在分配在栈中了，因为成员不是值类型的，它是引用类型的。它会指向堆内存创建的 T1 类型的实例。这样的情况就说明结构 T 不是具有值语义的，它的内存分配比较复杂。而具有值语义的结构在内存的分配上比较紧凑，高效。

12.9 结构的成员

结构的成员和类类型的成员相似，它可以包括常量、字段、方法、属性、索引器、实例构造函数、静态构造函数等等。但是它不可以包含析构方法。虽然结构的成员和类相似，但是结构不能替代类，有的需求的应用是必须使用类的。看下面的代码实例：

```
001 using System;
002 namespace StructType
003 {
```

```
004     public delegate void D(int a,int b);
005     struct TV
006     {
007         public event D myD;//在自动搜索完毕时调用
008         private int channel; //电视机当前的频道
009         private int volume; //电视机当前的音量
010         private int[] chList; //电视机的频道号
011         private string[] chName; //频道名
012         public string this[int index] //对频道名数组进行存取
013         {
014             get
015             {
016                 if (index >= 0 && index <10)
017                 {
018                     return chName[index];
019                 }
020                 else
021                     return "无此频道";
022             }
023             set
024             {
025                 if (index >= 0 && index <10)
026                 {
027                     chName[index] = value;
028                 }
029             }
030         }
031         //对字段 channel 进行存取
032         public int Channel
033         {
034             get { return channel; }
035             set { channel = value; }
036         }
037         //对字段 volume 进行存取
038         public int Volume
039         {
040             get { return volume; }
041             set { volume = value; }
042         }
043         /// <summary>
044         /// 对当前的频道和音量进行设置
045         /// </summary>
046         /// <param name="ch">设置频道</param>
047         /// <param name="vol">设置音量</param>
```

```
048     public void Set(int ch, int vol)
049     {
050         Console.WriteLine("开始设置频道和音量...");
051         this.Channel = ch;
052         this.Volume = vol;
053     }
054     public void AutoSearch()
055     {
056         Console.WriteLine("开始自动搜索.....");
057         chList = new int[10]; //对 chList 分配内存
058         chName = new string[10]; //对 chName 分配内存
059         myD += this.Set; //将方法添加进入调用列表
060         //假定一个信号源频道列表
061         string[] s = new string[10] { "cctv1", "cctv2", "cctv3", "cctv4", "cctv5",
062             "cctv6", "cctv7", "cctv8", "cctv9", "cctv10" };
063         for (int i = 0; i < 10; i++)
064         {
065             chList[i] = i + 1; //对频道号数组进行初始化
066             this[i] = s[i]; //调用索引器对频道名数组进行初始化
067         }
068         myD(1, 30); //搜索完成, 对当前的频道号和音量进行赋值
069         Console.WriteLine("自动搜索完成");
070     }
071     public void Show()
072     {
073         Console.WriteLine("现在的频道是: {0}", channel);
074         Console.WriteLine("现在的音量是: {0}", volume);
075     }
076     class Program
077     {
078         static void Main(string[] args)
079         {
080             TV myTV = new TV();
081             myTV.AutoSearch();
082             myTV.Show();
083             Console.WriteLine("频道列表: ");
084             for (int i = 0; i < 10; i++)
085             {
086                 Console.Write(myTV[i] + " ");
087             }
088             Console.WriteLine();
089             Console.WriteLine("换一个频道, 调整一下音量吧!");
090             myTV.Set(5, 50);
```

```

091         myTV.Show();
092         Console.ReadKey();
093     }
094 }
095 }

```

这段代码比较长，因此做了比较多的注释。整个程序仍旧模仿一台电视机，在这个情景的应用中，模拟了电视机自动搜索的过程。自动搜索完毕后，将电视机的当前频道和音量设置为一个特定的值。就好像电视机刚打开包装的时候，它里面什么节目列表也没有，这时为避免噪音的影响，将其音量设置为 0。然后接上信号线后，要开始自动搜索。自动搜索到的节目名要进行存储。自动搜索完毕后，因为已经有了节目列表了，要将当前的频道调整为 1，然后设定一个合适的音量，方便用户观看。这就是整个程序的需求和设计逻辑。

下面看代码，第 4 行定义的委托类型就是用来定义一个事件，它在自动搜索结束的时候被调用。这个事件就是第 7 行定义的事件。第 8 行定义了电视机当前的频道，第 9 行定义当前的音量。第 10 行和第 11 行分别定义了电视机的频道号列表和节目列表。它们是一一对应关系，即频道号数组中第一个成员，比如 int 类型的值 1，它代表 1 频道。这个频道号对应着 chName 数组中第一个成员，比如，它是 cctv1。这时，意味着 1 频道就是 cctv1。第 12 行是一个索引器，它对 chName 数组进行存取。第 32 行和第 38 行定义了两个属性，它们用来对 channel 字段和 volume 字段进行存取。第 48 行代码是一个实例方法，它对当前的节目和音量进行设置。第 54 行是一个实例方法，它演示的就是自动搜索过程。它首先对 chList 和 chName 数组分配内存，这里为简单起见，假定电视机就能存储 10 个频道。第 59 行将 Set 方法加入事件的调用列表。接下来定义一个 string 类型的数组，对它直接进行初始化，它里面存储着各个信号源的名称。然后开始一个 for 循环，对 chList 数组进行赋值，赋值是从 1 开始的。因为没有数字为 0 的频道。对频道名称数组进行赋值直接调用的就是索引器的 set 访问器。当对数组的赋值都完成的时候，调用事件，将当前的频道设置为 1，音量设置为 30。然后打印搜索完成的提示。第 70 行的 Show 方法用来打印当前的频道号和音量。

Main 方法中的代码不再多说，其中第 84 行代码使用 for 语句调用索引器的 get 方法输出电视机中存储的频道列表。下面看一下执行结果，如下：

```

001  开始自动搜索.....
002  开始设置频道和音量...
003  自动搜索完成
004  现在的频道是：0
005  现在的音量是：0
006  频道列表：
007  cctv1 cctv2 cctv3 cctv4 cctv5 cctv6 cctv7 cctv8 cctv9 cctv10
008  换一个频道, 调整一下音量吧!
009  开始设置频道和音量...
010  现在的频道是：5
011  现在的音量是：50

```

这段执行结果也标上了行号，这个行号不是程序执行得来的，只是为了方便介绍。如果细心，就可以发现第 4 行和第 5 行结果不是代码设计时想要的结果。它们在自动搜索完成时，应该被设置为 1 和 30。这是由事件的调用所赋的值。为何这里都是 0 呢？下面将 Set 方法做一下修改，代码如下：

```

public void Set(int ch, int vol)
{
    Console.WriteLine("开始设置频道和音量...");
    this.Channel = ch;
}

```

```

    this.Volume = vol;
    Show();
}

```

在 Set 方法中加入一个 Show 方法的调用，这时再看结果，如下：

```

001  开始自动搜索.....
002  开始设置频道和音量...
003  现在的频道是：1
004  现在的音量是：30
005  自动搜索完成
006  现在的频道是：0
007  现在的音量是：0
008  频道列表：
009  cctv1 cctv2 cctv3 cctv4 cctv5 cctv6 cctv7 cctv8 cctv9 cctv10
010  换一个频道, 调整一下音量!
011  开始设置频道和音量...
012  现在的频道是：5
013  现在的音量是：50
014  现在的频道是：5
015  现在的音量是：50

```

第 3 行和第 4 行结果就是执行 Set 方法中最后调用的 Show 方法得到的结果，但是奇怪的是，方法执行完毕后，再次调用 Show 方法，字段又变为 0 了。这是什么原因呢？造成这个结果的原因就是因为结构和类的本质上的不同，结构是值类型，而类是引用类型。

首先从委托类型来说，事件背后调用的实际就是委托。而委托是一个类类型，它在使用前是需要实例化的，下面是开发环境的帮助文档中的委托的构造方法的说明：

```
protected MulticastDelegate(Object target, string method)
```

可以看到，在委托的构造方法中包含了一个 Object 类型的参数，它的说明如下：

参数：target 类型：System.Object

在其上定义 method 的对象。

也就是说，在构造委托类型实例的时候，要为其传入一个 Object 类型，而本例这个传入的类型就是结构的 this 引用。而结构是值类型的，当为委托的构造方法中按值参数传入 this 的时候，就会发生复制操作。而结构又是值类型的，它的一个副本将被传入委托的构造方法，用来创建委托的实例。当在代码中对事件进行调用的时候，它背后将调用这个创建的委托。而这个委托调用 Set 方法修改字段 channel 和 volume 的值时，修改的是为委托传入的当前结构的副本的字段的值。因此，当事件调用结束，退出 Set 方法之后，当前的这个结构实例的字段还是老样子，根本没发生变化。

这就是结构不适合含有引用类型成员的需求场景。当需要逻辑上特别复杂的，含有引用的数据结构的时候，就要使用类，而不是结构。当将上面的第 5 行代码中的 struct 关键字替换成 class 的时候，也就是将它定义为一个类，则一切都正常了。代码完全按照设计的要求执行。代码执行结果如下：

```

开始自动搜索.....
开始设置频道和音量...
自动搜索完成
现在的频道是：1
现在的音量是：30
频道列表：
cctv1 cctv2 cctv3 cctv4 cctv5 cctv6 cctv7 cctv8 cctv9 cctv10

```


换一个频道, 调整一下音量吧!

开始设置频道和音量...

现在的频道是: 5

现在的音量是: 50

出现这个结果的原因就是当委托进行实例化的时候, 传入的 `this` 是类类型, 它是一个引用, 而不是复制的副本, 它仍然代表当前的实例, 所以当前实例的字段受到影响, 被 `Set` 方法修改了。

第十三章 命名空间

C#代码在逻辑上的最外层，是使用命名空间对代码进行组织 and 管理的。命名空间从根本上解决了程序员在定义类的时候，名称重名的问题。因此在内部上，命名空间可以用于代码的组织；在对外发布上，命名空间向其它使用者公布代码的总体构成，方便第三方使用。

13.1 编译单元和命名空间的定义

一个 C#程序由一个或多个编译单元组成，一个编译单元就对应着项目中的一个以“.cs”为扩展名的源文件。当最后编译程序时，所有的编译单元一起进行处理。编译单元之间可以依赖，甚至这种依赖是循环的。下面是代码实例，在这个项目中定义了两个编译单元，它们都以“.cs”为扩展名的源文件形式存在。

```
001 //Class1.cs
002 using System;
003 class Test
004 {
005     public void Show()
006     {
007         Console.WriteLine("位于 Test 类中的 Show 方法被调用");
008     }
009 }
```

这是 Class1.cs 文件，在这个源文件中只包含了一个类，名为 Test。在类中定义了一个实例方法。可以看到，这个类的外面是没有定义命名空间的。下面再建立一个源文件，名为 Program.cs，代码如下：

```
001 //Program.cs
002 using System;
003 class Program
004 {
005     static void Main(string[] args)
006     {
007         Test t = new Test();
008         t.Show();
009         Console.ReadKey();
010     }
011 }
```

这个源文件中的类也没有被命名空间包裹，它调用了 Test 类，创建了 Test 类的一个实例，然后调用它的方法。在开发环境输入代码的时候，就可以观察到，对于类 Test 都有相关的智能提示。如果类的外面没有定义命名空间，则它们就属于一个全局命名空间，这个全局命名空间没有名称。所以虽然这两个类位于不同的源文件中，但是因为它们属于一个程序集中，这两个类的访问权限修饰符是默认的 internal。这两个类就可以互相进行调用。代码的执行结果如下：

位于 Test 类中的 Show 方法被调用

因为是位于一个全局的命名空间内，所以在这个全局命名空间内定义的类型名也不能重名。除非它们是分部类。实例代码如下：

```
001 //Class1.cs
002 using System;
003 partial class Test
004 {
005     public void Show()
```

```

006      {
007          Console.WriteLine("位于 Test 类中的 Show 方法被调用");
008      }
009  }
010  class Test1
011  { }

```

修改 Class1.cs 中的第 3 行代码，将原来的独立的类更改为分部的类。再在第 10 行添加一个类 Test1。下面在修改 Program.cs 文件，代码如下：

```

001  //Program.cs
002  using System;
003  partial class Test
004  {
005      static void Main(string[] args)
006      {
007          Test t = new Test();
008          t.Show();
009          Console.ReadKey();
010      }
011  }
012  class Test1
013  { }

```

修改第 3 行的类为分部类，并且修改其名称和 Class1 文件中的类相同。然后在第 12 行添加一个类，也命名为 Test1。接下来开始编译程序集，编译器会报告一个错误，如下：

```

错误 1    命名空间“<全局命名空间>”已经包含了“Test1”的定义H:\书稿 new\书稿源码\第十三章
\NameSpace1\Program.cs 12    7    NameSpace1

```

编译器会报告说，全局命名空间中的类重名了。但是因为分部类允许类名重复，因为它们就是一个类，编译阶段要合到一起的，所以分部类不会报错。

定义一个命名空间的时候，使用 namespace 关键字，然后后跟命名空间的名称，再跟一对大括号。大括号可以以分号结尾，也可以不加分号。通常情况下是不加分号的。下面是代码实例：

```

001  using System;
002  namespace Program
003  {
004      class Test
005      {
006          public void Show()
007          {
008              Console.WriteLine("位于 Test 类中的 Show 方法被调用");
009          }
010      }
011      class Program
012      {
013          static void Main(string[] args)
014          {
015              Test t = new Test();

```

```

016         t.Show();
017         Console.ReadKey();
018     }
019 }
020 };

```

这是定义了命名空间的程序集，第 20 行的命名空间结尾的地方加了分号，这也是语法所允许的，编译器不会报错。第 2 行是命名空间的定义，命名空间不像类的成员一样可以加访问修饰符。但是没有访问修饰符不代表没有访问权限，命名空间的访问权限隐式是 public 的，但是不能为其再自定义 public 访问权限修饰符。如果为命名空间加上了形如：

```
public namespace Program
```

的访问修饰符，编译器就会报错，错误信息如下：

```

错误 1    命名空间声明不能有修饰符或特性    H:\书稿new\书稿源码\第十三章\NameSpace1\Program.cs
        2    1    NameSpace1

```

命名空间声明的位置如果在全局命名空间下，则它就是全局命名空间的成员。命名空间的定义还可以在其它命名空间里，在这种情况下，它就是另一个命名空间的成员。如果两个命名空间的名称相同，且又定义在相同的命名空间内部，则这两个同名的命名空间就是可扩充的命名空间，实际上，它们将合成一个命名空间。在它们的内部不能定义相同的成员。代码如下：

```

001    using System;
002    namespace Program
003    {
004        class Test
005        {
006            public void Show()
007            {
008                Console.WriteLine("位于 Test 类中的 Show 方法被调用");
009            }
010        }
011    }
012    namespace Program
013    {
014        class Program
015        {
016            static void Main(string[] args)
017            {
018                Test t = new Test();
019                t.Show();
020                Console.ReadKey();
021            }
022        }
023    }

```

这段代码在全局命名空间下面定义了两个相同的命名空间 Program。它们将被看做是一个命名空间的扩充。在其内部的成员也将在编译时合并在一起。因此在代码的第 18 行可以直接创建类 Test 的实例，而不用完全限定名称。这段代码的执行结果如下：

```
位于 Test 类中的 Show 方法被调用
```

但是，在这两个相同名称的命名空间内部，不能定义相同名称的成员。现在对上面的代码做如下修改：

```

001  using System;
002  namespace Program
003  {
004      class Test
005      {
006          public void Show()
007          {
008              Console.WriteLine("位于 Test 类中的 Show 方法被调用");
009          }
010      }
011  }
012  namespace Program
013  {
014      class Test
015      { }
016      class Program
017      {
018          static void Main(string[] args)
019          {
020              Test t = new Test();
021              t.Show();
022              Console.ReadKey();
023          }
024      }
025  }

```

在第 12 行代码定义的命名空间内又定义了和第一个命名空间内相同名称的类，在编译时，编译器会报错，如下：

错误 1 命名空间“Program”已经包含了“Test”的定义 H:\书稿 new\书稿源码\第十三章\NameSpace1\Program.cs 14 11 NameSpace1

命名空间可以使用“.”运算符划分层次，它和命名空间的嵌套是相同的效果。代码实例如下：

```

001  using System;
002  namespace Netc.Program
003  {
004      class Test
005      {
006          public void Show()
007          {
008              Console.WriteLine("位于 Test 类中的 Show 方法被调用");
009          }
010      }
011      class Program
012      {
013          static void Main(string[] args)

```

```
014     {
015         Netc.Program.Test t = new Test();
016         t.Show();
017         Console.ReadKey();
018     }
019 }
020 }
```

第 2 行代码定义了一个划分层次的命名空间，对命名空间划分层次可以更好的组织代码的结构。比如说，可以在命名空间的第一层定义公司名；第二层定义项目名等等。这样就可以大大减少类名重复的可能性，而且代码变得更有层次性了。这样划分层次的命名空间和下面的嵌套的命名空间是一样的效果。代码如下：

```
001 using System;
002 namespace Netc
003 {
004     namespace Program
005     {
006         class Test
007         {
008             public void Show()
009             {
010                 Console.WriteLine("位于 Test 类中的 Show 方法被调用");
011             }
012         }
013         class Program
014         {
015             static void Main(string[] args)
016             {
017                 Netc.Program.Test t = new Test();
018                 t.Show();
019                 Console.ReadKey();
020             }
021         }
022     }
023 }
```

第 2 行代码定义了一个名称为 Netc 的命名空间，然后在其内部又定义了一个名称为 Program 的命名空间。因为嵌套的命名空间和划分层次的命名空间作用是一样的，所以第 17 行仍可以使用划分层次的命名空间名称来引用其中定义的类。

可以使用 using 语句来简化命名空间名称的输入，但是 using 语句必须位于任何类型的声明之前。代码实例如下：

```
001 using System;
002 namespace Netc
003 {
004     namespace Program
005     {
```

```

006      using System.Data;
007      class Test
008      {
009          using System.Data;
010          public void Show()
011          {
012              Console.WriteLine("位于 Test 类中的 Show 方法被调用");
013          }
014      }
015      class Program
016      {
017          static void Main(string[] args)
018          {
019              Netc.Program.Test t = new Test();
020              t.Show();
021              Console.ReadKey();
022          }
023      }
024  }
025  }

```

在代码的第一句定义 using 语句肯定没什么问题，但是在其它地方定义 using 语句就会有限制。using 语句仅允许定义在类的定义之前。第 9 行代码将 using 语句定义在了类的成员的位置。这样编译器就会报错，错误信息如下：

```

错误 1    类、结构或接口成员声明中的标记“using”无效    H:\书稿 new\书稿源码\第十三章
\NameSpace1\Program.cs 9    13    NameSpace1
错误 2    类、结构或接口成员声明中的标记“;”无效    H:\书稿 new\书稿源码\第十三章
\NameSpace1\Program.cs 9    30    NameSpace1

```

13.2 using 别名语句

using 关键字为包容它的编译单元或命名空间体内引入命名空间或类的别名。它的作用范围依据它的定义位置不同而不同。下面是代码实例：

```

001      using System;
002      using myNamespace = Netc.N1;
003      namespace Netc.N1
004      {
005          class Test
006          {
007              public void Show()
008              {
009                  Console.WriteLine("位于 Test 类中的 Show 方法被调用");
010              }
011          }
012      }
013      namespace N2
014      {

```

```
015     class Test1
016     {
017         public myNamespace.Test t = new myNamespace.Test();
018     }
019 }
020 namespace N3
021 {
022     class Program
023     {
024         static void Main(string[] args)
025         {
026             myNamespace.Test t = new myNamespace.Test();
027             t.Show();
028             Console.ReadKey();
029         }
030     }
031 }
```

这段代码的第 3 行定义了一个命名空间 `Netc.N1`，在其中定义了一个类 `Test`。在第 13 行和第 20 行各定义了一个命名空间，在其中各定义了一个类，在每个命名空间内的类中，都要使用命名空间 `Netc.N1` 中的类 `Test`。在代码的第 2 行使用 `using` 语句定义了一个命名空间的别名。语法是 `using` 关键字后跟别名，然后是赋值运算符，最后是要为其定义别名的命名空间的完全限定名称。语句要以分号结束。

因为这个别名定义在全局命名空间内，它对于下面定义的两个命名空间都是可见的。它的作用域范围就是整个编译单元。下面再将它定义在命名空间中，代码实例如下：

```
001 using System;
002 namespace Netc.N1
003 {
004     class Test
005     {
006         public void Show()
007         {
008             Console.WriteLine("位于 Test 类中的 Show 方法被调用");
009         }
010     }
011 }
012 namespace N2
013 {
014     using myNamespace = Netc.N1;
015     class Test1
016     {
017         public myNamespace.Test t = new myNamespace.Test();
018     }
019 }
020 namespace N3
021 {
```



```

022     class Program
023     {
024         static void Main(string[] args)
025         {
026             myNamespace.Test t = new myNamespace.Test();
027             t.Show();
028             Console.ReadKey();
029         }
030     }
031 }

```

这时,“using myNamespace = Netc.N1;”这句别名指令被定义在命名空间 N2 中。那么它的作用域范围就是 N2 命名空间体内。其它命名空间将看不到这个别名。在命名空间 N3 内部,第 26 行引用这个命名空间的别名将会引发错误,错误信息如下:

```

错误 1    未能找到类型或命名空间名称“myNamespace”(是否缺少 using 指令或程序集引用?)    H:\
书稿 new\书稿源码\第十三章\NameSpace1\Program.cs    26    13    NameSpace1
错误 2    未能找到类型或命名空间名称“myNamespace”(是否缺少 using 指令或程序集引用?)    H:\
书稿 new\书稿源码\第十三章\NameSpace1\Program.cs    26    38    NameSpace1

```

因为第 26 句代码两处引用了这个别名,所以会报两个错误。

using 指令不仅仅能为命名空间定义别名,还能为类定义别名。代码实例如下:

```

001     using System;
002     namespace Netc.N1
003     {
004         class Test
005         {
006             public void Show()
007             {
008                 Console.WriteLine("位于 Test 类中的 Show 方法被调用");
009             }
010         }
011     }
012     namespace N2
013     {
014         using myNamespace = Netc.N1;
015         using myClass = Netc.N1.Test;
016         class Program
017         {
018             static void Main(string[] args)
019             {
020                 myClass t = new myClass();
021                 t.Show();
022                 myNamespace.Test t1 = new myNamespace.Test();
023                 t1.Show();
024                 Console.ReadKey();
025             }

```

```
026     }
027 }
```

代码的第 14 行在命名空间 N2 中为 Netc.N1 命名空间定义了一个别名，然后又为其中的 Test 类定义了一个别名。这两个别名的作用域范围就是 N2 命名空间。在第 20 行，使用类的别名创建了一个 Test 类的实例；第 22 行使用命名空间的别名创建了一个类 Test 的实例。它们起到了同样的效果。代码的执行结果如下：

位于 Test 类中的 Show 方法被调用

位于 Test 类中的 Show 方法被调用

不仅仅在类的实例被创建的时候可以使用别名，在类的继承过程中也可以使用别名。代码实例如下：

```
001 using System;
002 namespace Netc.N1
003 {
004     class Test
005     {
006         public void Show()
007         {
008             Console.WriteLine("位于 Test 类中的 Show 方法被调用");
009         }
010     }
011 }
012 namespace N2
013 {
014     using Test = Netc.N1.Test;
015     using myNamespace = Netc.N1;
016     class Test1 : myNamespace.Test
017     { }
018     class Program :Test
019     {
020         static void Main(string[] args)
021         {
022             Program p = new Program();
023             p.Show();
024             Console.ReadKey();
025         }
026     }
027 }
```

在命名空间 N2 中，自定义的类如果要从 Netc.N1 命名空间中的类继承时，可以使用类 Test 的完全限定名称指定其直接基类。也可以使用第 14 行那样的类名的别名指定 Netc.N1.Test 类的别名，然后使用这个别名指定其直接基类。定义别名的时候，别名可以和 Netc.N1 中的类名相同，看起来就好像在 N2 中可以直接引用 Test 类一样。使用第 15 行定义的别名就需要使用这个别名再引用其中的类名，第 16 行代码就是这么做的。

在编译单元或者命名空间中定义的别名是唯一的，它不可以和另一个命名空间的名称或命名空间内的类名名称重复。下面是代码实例，这是一段错误的代码：

```
001 using System;
```

```

002    using N2 = Netc.N1;
003    namespace Netc.N1
004    {
005        class Test
006        {
007            public void Show()
008            {
009                Console.WriteLine("位于 Test 类中的 Show 方法被调用");
010            }
011        }
012    }
013    namespace N2
014    {
015        using Test1 = Netc.N1.Test;
016        class Test1 : Test1
017        { }
018        class Program :Test1
019        {
020            static void Main(string[] args)
021            {
022                Program p = new Program();
023                p.Show();
024                Console.ReadKey();
025            }
026        }
027    }

```

在第 2 行定义的别名和第 13 行定义的命名空间的名称重复了；第 15 行定义的类型名的别名和第 16 行定义的类的名称重复。在编译时，编译器会报错，错误信息如下：

```

错误 1    命名空间“N2”包含与别名“Test1”冲突的定义    H:\书稿 new\书稿源码\第十三章
\NameSpace1\Program.cs 16  19  NameSpace1
错误 2    命名空间“N2”包含与别名“Test1”冲突的定义    H:\书稿 new\书稿源码\第十三章
\NameSpace1\Program.cs 18  20  NameSpace1
错误 3    “N2.Program”不包含“Show”的定义，并且找不到可接受类型为“N2.Program”的第一个参
数的扩展方法“Show”（是否缺少 using 指令或程序集引用？） H:\书稿 new\书稿源码\第十三章
\NameSpace1\Program.cs 23  15  NameSpace1

```

命名空间具有可扩充性，但是在其中任何一个命名空间部分内定义的别名则不具有扩充性，它的作用域是具有严格限制的。实例代码如下：

```

001    using System;
002    namespace Netc.N1
003    {
004        using myNamespace = Netc.N1;
005        class Test
006        {
007            public void Show()

```

```

008         {
009             Console.WriteLine("位于 Test 类中的 Show 方法被调用");
010         }
011     }
012 }
013 namespace Netc.N1
014 {
015     class Program
016     {
017         static void Main(string[] args)
018         {
019             myNamespace.Test t = new myNamespace.Test();
020             t.Show();
021             Test t1 = new Test();
022             t1.Show();
023             Console.ReadKey();
024         }
025     }
026 }

```

这段代码中的第 2 行和第 13 行定义了相同名称的命名空间，在编译时，它们将合在一起作为一个命名空间进行编译。因此，第 21 行的代码中，可以直接引用 Test 类型创建实例。但是在第 2 行定义的命名空间体内定义的别名，却不能扩充到第 13 行定义的命名空间体内，在其中是看不到别名 myNamespace 的。因此，第 19 行引用这个别名将会导致错误。错误信息如下：

```

错误 1    未能找到类型或命名空间名称“myNamespace”（是否缺少 using 指令或程序集引用？）    H:\
书稿 new\书稿源码\第十三章\NameSpace1\Program.cs    19    13    NameSpace1
错误 2    未能找到类型或命名空间名称“myNamespace”（是否缺少 using 指令或程序集引用？）    H:\
书稿 new\书稿源码\第十三章\NameSpace1\Program.cs    19    38    NameSpace1

```

如果要想使这个别名在第 13 行定义的命名空间内也可用，办法就是把这个别名的定义挪到第 1 行代码的下边，也就是将它定义在全局命名空间内，这时它将对整个编译单元都是有效的。

13.2.1 别名的名称隐藏

如果在定义别名的命名空间内再定义同名的类，则会引发错误。但是如果将别名定义在全局命名空间范围内则会发生名称隐藏。下面是代码实例：

```

001 using System;
002 using Test = System.Console;
003 namespace Netc.N1
004 {
005     class Test
006     {
007         public void Show()
008         {
009             Console.WriteLine("位于 Test 类中的 Show 方法被调用");
010         }
011     }
012     class Program

```

```

013     {
014         static void Main(string[] args)
015         {
016             Test t = new Test();
017             t.Show();
018             Test.WriteLine("Hello World");
019             Console.ReadKey();
020         }
021     }
022 }

```

代码的第 2 行定义了一个 .Net 类库中 System.Console 类的别名，这个别名因为定义在全局命名空间下面，所以它对于命名空间 Netc.N1 内的成员是可见的。但是在 Netc.N1 命名空间内又定义了一个同名的类 Test。这时，将会发生什么呢？Netc.N1 命名空间内的 Test 将隐藏全局命名空间内的别名 Test。当第 16 行代码定义 Test 类的实例的时候，它引用的是命名空间内定义的类 Test，而第 18 行意图使用别名引用 System.Console 类中的 WriteLine 方法将会引发错误。错误信息如下：

```

错误 1      “Netc.N1.Test” 并不包含 “WriteLine” 的定义    H:\书稿 new\书稿源码\第十三章
\NameSpace1\Program.cs 18 18  NameSpace1

```

使用 using 语句指定别名的时候，如果有多个 using 语句，则这些 using 语句所定义的别名之间不可以互相引用。看下面的代码实例：

```

001     using System;
002     namespace N1
003     {
004         using myN1 = N1;
005         using myN2 = myN1.N2;
006         using myN3 = myN2.N3;
007         namespace N2
008         {
009             namespace N3
010             {
011                 class Test
012                 {
013                     public void Show()
014                     {
015                         Console.WriteLine("位于 Test 类中的 Show 方法被调用");
016                     }
017                 }
018                 class Program
019                 {
020                     static void Main(string[] args)
021                     {
022                         myN3.Test t = new myN3.Test();
023                         t.Show();
024                         Console.ReadKey();
025                     }

```

```

026         }
027     }
028 }
029 }

```

这段代码首先定义了一个命名空间 N1，然后在其中依次定义嵌套的命名空间 N2 和 N3。在代码的第 4 行到第 6 行使用 using 语句为每个命名空间建立别名。为了简单起见，下一个别名引用了上一个别名。但是这在语法上是不允许的，多个 using 语句之间不能直接引用。所以，在第 22 行代码引用这个别名的时候就会报错。错误信息如下：

错误 1 未能找到类型或命名空间名称“myN1”（是否缺少 using 指令或程序集引用？） H:\书稿 new\书稿源码\第十三章\NameSpace1\Program.cs 5 18 NameSpace1

错误 2 未能找到类型或命名空间名称“myN2”（是否缺少 using 指令或程序集引用？） H:\书稿 new\书稿源码\第十三章\NameSpace1\Program.cs 6 18 NameSpace1

13.3 using 命名空间语句

使用 using 语句可以将指定的命名空间所包含的类导入到定义 using 语句的编译单元或命名空间中。使得不必使用类型的完全限定名称就可以使用类的标识符引用指定的类。前面已经接触过使用 using 语句导入 System 命名空间。导入后，就可以直接使用 Console 类的静态方法。下面看代码实例：

```

001 using System;
002 namespace N1
003 {
004     namespace N2.N3
005     {
006         class Test
007         {
008             public void Show()
009             {
010                 Console.WriteLine("位于 Test 类中的 Show 方法被调用");
011             }
012         }
013     }
014 }
015 namespace Program
016 {
017     using N1;
018     class Program
019     {
020         static void Main(string[] args)
021         {
022             N2.N3.Test t = new N2.N3.Test();
023             t.Show();
024             Console.ReadKey();
025         }
026     }
027 }

```

这段代码的第 2 行定义了一个命名空间 N1，然后在其中又定义了一个嵌套的命名空间。嵌套的命名空

间又进行了层次的划分。然后在其中定义了一个类 Test。在第 15 行定义的命名空间 Program 中，首相使用 using 语句导入了 N1。但是，在第 22 行使用它的嵌套命名空间的限定名引用类 Test 却会导致错误。错误信息如下：

```
错误 1    未能找到类型或命名空间名称“N2” (是否缺少 using 指令或程序集引用?)    H:\书稿 new\
书稿源码\第十三章\NameSpace1\Program.cs    22    13    NameSpace1
错误 2    未能找到类型或命名空间名称“N2” (是否缺少 using 指令或程序集引用?)    H:\书稿 new\
书稿源码\第十三章\NameSpace1\Program.cs    22    32    NameSpace1
```

出现这两个错误的原因是，使用 using 语句导入的只能是该命名空间下的类，而不包括其下的命名空间。所以，要改正这一错误，可以将第 17 行的代码修改为“using N1.N2.N3”。这样，在第 22 行就可以直接使用 Test 类的标识符名来直接引用这个类。

使用命名空间的别名有隐藏现象的发生，使用 using 导入命名空间的时候，如果被导入的命名空间内含有和本编译单元或者本命名空间中的类名重名的类，则本编译单元或本命名空间内的类将会隐藏掉导入的同名类。代码实例如下：

```
001    using System;
002    namespace N1
003    {
004        class Test
005        {
006            public void Show()
007            {
008                Console.WriteLine("位于 N1.Test 类中的 Show 方法被调用");
009            }
010        }
011    }
012    namespace N2
013    {
014        class Test
015        {
016            public void Show()
017            {
018                Console.WriteLine("位于 N2.Test 类中的 Show 方法被调用");
019            }
020        }
021        class Test1
022        {
023            public void Show()
024            {
025                Console.WriteLine("位于 N2.Test1 类中的 Show 方法被调用");
026            }
027        }
028    }
029    namespace N3
030    {
031        using N1;
```

```

032     using N2;
033     /// <summary>
034     /// 隐藏掉 N2 中的 Test1 类
035     /// </summary>
036     class Test1
037     {
038         public void Show()
039         {
040             Console.WriteLine("位于 N3. Test1 类中的 Show 方法被调用");
041         }
042     }
043     class Program
044     {
045         static void Main(string[] args)
046         {
047             Test1 t1 = new Test1();
048             t1.Show();
049             Test t = new Test();
050             t.Show();
051             Console.ReadKey();
052         }
053     }
054 }

```

本程序中的命名空间 N1 和 N2 中各定义了一个同名的类 Test。在 N2 中还定义了一个类 Test1。在命名空间 N3 中首先使用 using 语句导入了命名空间 N1 和 N2。在命名空间 N3 中又定义了一个和 N2 中同名的类 Test1。当在 Main 方法中，引用 Test1 类创建实例的时候，因为 Test1 就位于类 Program 所在的命名空间 N3 中，所以 Test1 将隐藏掉导入的 N2 中的 Test1。当引用导入的类 Test 创建类的实例的时候，因为 N1 和 N2 都导入了同名的类，所以发生歧义。编译器会报告错误，错误信息如下：

错误 1 “Test”是“N1.Test”和“N2.Test”之间的不明确的引用 H:\书稿 new\书稿源码\第十三章\NameSpace1\Program.cs 49 13 NameSpace1

错误 2 “Test”是“N1.Test”和“N2.Test”之间的不明确的引用 H:\书稿 new\书稿源码\第十三章\NameSpace1\Program.cs 49 26 NameSpace1

那么，当引入的命名空间中包含同名的类名时，如何解决呢？可以使用前面介绍过的命名空间别名的方式来指定要使用的类。实例代码如下：

```

001     using System;
002     namespace N1
003     {
004         class Test
005         {
006             public void Show()
007             {
008                 Console.WriteLine("位于 N1. Test 类中的 Show 方法被调用");
009             }
010         }

```



```
011     }
012     namespace N2
013     {
014         class Test
015         {
016             public void Show()
017             {
018                 Console.WriteLine("位于 N2. Test 类中的 Show 方法被调用");
019             }
020         }
021         class Test1
022         {
023             public void Show()
024             {
025                 Console.WriteLine("位于 N2. Test1 类中的 Show 方法被调用");
026             }
027         }
028     }
029     namespace N3
030     {
031         using N1;
032         using N2;
033         using Test = N1.Test;
034         /// <summary>
035         /// 隐藏掉 N2 中的 Test1 类
036         /// </summary>
037         class Test1
038         {
039             public void Show()
040             {
041                 Console.WriteLine("位于 N3. Test1 类中的 Show 方法被调用");
042             }
043         }
044         class Program
045         {
046             static void Main(string[] args)
047             {
048                 Test1 t1 = new Test1();
049                 t1.Show();
050                 Test t = new Test();
051                 t.Show();
052                 Console.ReadKey();
053             }
054         }
```

```
055    }
```

在代码的第 33 行定义了一个别名 Test 来指定为 N1 命名空间中的 Test 类。这样一来，第 50 行代码的调用就没有问题了。代码的执行结果如下：

位于 N3.Test1 类中的 Show 方法被调用

位于 N1.Test 类中的 Show 方法被调用

13.4 命名空间别名限定符

命名空间别名限定符由两个冒号组成，形如“::”。它的用处就是让类型名称的查找不受引入的新类或新成员的影响。下面通过代码实例来介绍：

```
001    using System;
002    namespace N1
003    {
004        namespace N2
005        {
006            class Program
007            {
008                static void Main(string[] args)
009                {
010                    N2.Test.Show();
011                    Console.ReadKey();
012                }
013            }
014        }
015    }
016    namespace N2
017    {
018        class Test
019        {
020            public static void Show()
021            {
022                Console.WriteLine("N2.Test 中的 Show 方法");
023            }
024        }
025    }
```

这段代码演示了这样一种情况，如果在全局命名空间中定义了两个命名空间，分别是 N1 和 N2。但是在命名空间 N1 中，又定义了一个命名空间 N2。在全局命名空间中的 N2 命名空间内，定义了一个类 Test。在其中定义了一个静态方法。如果要在 Main 方法中引用这个方法，使用第 10 行代码的语法是不行的。因为 N1 中的命名空间 N2 隐藏了全局命名空间中的 N2。这时，编译器会报错，如下：

错误 1 命名空间“N1.N2”中不存在类型或命名空间名称“Test”。是否缺少程序集引用? H:\书稿
new\书稿源码\第十三章\NameSpace1\Program.cs 10 20 NameSpace1

怎么能避免局部定义的命名空间 N2 的影响呢？这时就需要命名空间别名限定符。修改代码如下：

```
001    using System;
002    namespace N1
003    {
004        namespace N2
```

```
005     {
006         class Program
007         {
008             static void Main(string[] args)
009             {
010                 global::N2.Test.Show();
011                 Console.ReadKey();
012             }
013         }
014     }
015 }
016 namespace N2
017 {
018     class Test
019     {
020         public static void Show()
021         {
022             Console.WriteLine("N2.Test 中的 Show 方法");
023         }
024     }
025 }
```

第 10 行代码使用了命名空间别名限定符，限定符左边的 `global` 表示从全局命名空间开始查找后面的类。而不是从局部的 `N2` 命名空间开始。因为全局命名空间下有名称为 `N2` 的命名空间，所以，从它开始查找后，`Test` 类名可以匹配。因此，最后全局命名空间 `N2` 中的类 `Test` 将被正确引用。这段代码的执行结果如下：

N2.Test 中的 Show 方法

还有一种极端的情况，但是这种状况很难遇到。就是自定义的类所在的命名空间和类库中的命名空间重名了。代码实例如下：

```
001 using System;
002 namespace N1
003 {
004     namespace System
005     {
006         class Program
007         {
008             private int Console;
009             static void Main(string[] args)
010             {
011                 //Console.WriteLine("Hello World");
012                 //System.Console.WriteLine("Hello World");
013                 global::System.Console.WriteLine("Hello World");
014                 global::System.Console.ReadKey();
015             }
016         }
017     }
```

```
018    }
```

这段代码的第 4 行在外层命名空间 N1 里面又定义了一个命名空间 System，很不幸，这个命名空间的名称和 .Net 类库中的 System 命名空间重名了。在类 Program 中又定义了一个字段，名字为 Console。更不幸的是，这个字段的名称和 System.Console 类重名了。因此，它们将隐藏掉类库中的 Console 类。在第 11 行和第 12 行注释掉的代码中，调用 Console 类的成员都会出错。如果没有字段 Console，那么第 1 行的代码就会很有用，可以在类 Program 中直接使用 Console 类。但是因为字段 Console 的存在，这也变得不可行。因此唯一的办法就是使用命名空间别名限定符。让查找从全局命名空间开始。使用方法就是第 13 行和第 14 行的代码。

当在类中定义的别名和类中的成员重名时，命名空间别名限定符也显得特别有用。实例代码如下：

```
001    using System;
002    namespace N1
003    {
004        class Test
005        {
006        }
007    }
008    namespace N1
009    {
010        using Test = System.Collections;
011        class Program
012        {
013            static void Main(string[] args)
014            {
015                Test.ArrayList ta = new Test.ArrayList();
016                Console.ReadKey();
017            }
018        }
019    }
```

在这段代码中定义的两个命名空间重名了，因此它们在编译时将看做是一个命名空间。在第 4 行定义了一个名为 Test 的类，第 10 行定义了一个 System.Collections 命名空间的别名。这个别名和类 Test 重名了。在第 15 行使用别名调用类库中的类时，编译器就会报错，错误信息如下：

错误 1 命名空间“N1”包含与别名“Test”冲突的定义 H:\书稿 new\书稿源码\第十三章
\NameSpace1\Program.cs 15 13 NameSpace1

错误 2 命名空间“N1”包含与别名“Test”冲突的定义 H:\书稿 new\书稿源码\第十三章
\NameSpace1\Program.cs 15 37 NameSpace1

避免这个错误的办法还是使用命名空间别名限定符。使用方法如下：

```
001    using System;
002    namespace N1
003    {
004        class Test
005        {
006        }
007    }
008    namespace N1
```

```

009  {
010      using Test = System.Collections;
011      class Program
012      {
013          static void Main(string[] args)
014          {
015              Test::ArrayList ta = new Test::ArrayList();
016              Console.ReadKey();
017          }
018      }
019  }

```

第 15 行使用了命名空间别名限定符，让查找从 Test 别名开始即可解决问题。但是需要注意的是，命名空间别名限定符的右边不能是方法名。例如下面的使用方法是错误的：

```

001  using System;
002  namespace N1
003  {
004      class Test
005      {
006      }
007  }
008  namespace N1
009  {
010      using Test = System.Console;
011      class Program
012      {
013          static void Main(string[] args)
014          {
015              Test::WriteLine("Hello World");
016              Console.ReadKey();
017          }
018      }
019  }

```

别名 Test 定位到类名，第 15 行从别名开始查找方法名是不允许的。编译器会报错，信息如下：

错误 1 命名空间别名限定符“::”始终解析为类型或命名空间，因此在这里是非法的。请考虑改用“.”。

H:\书稿 new\书稿源码\第十三章\NameSpace1\Program.cs 15 17 NameSpace1

13.5 调用外部方法

外部方法就是这个方法不是本程序集中定义的成员，它是由其它非托管语言编写的动态链接库。例如 C++ 的动态链接库。通常在代码中要调用连接的设备的驱动程序的时候，需要调用其编写好的动态链接库。在调用前，需要将动态链接库复制到本程序中。然后使用 extern 关键字声明方法成员。实例代码如下：

```

001  using System;
002  using System.Runtime.InteropServices;
003  namespace N1
004  {
005      class Program

```

```
006      {
007          [DllImport("externDll.dll")]
008          extern static void Show(int a, int b);
009          static void Main(string[] args)
010          {
011              Show(11, 22);
012              Console.ReadKey();
013          }
014      }
015  }
```

外部的动态链接库中定义了如下的这样一个方法：

```
extern "C" void DLL_EXPORT Show (int a,int b)
{
    std::cout << "这是一个外部方法, 参数a和参数b的值是: " << a << " " << b << std::endl;
}
```

要使用外部的的方法，首先需要导入 System.Runtime.InteropServices 命名空间。然后第 7 行要使用特性指定要导入的方法所在的动态链接库文件。特性在以后会接触到，这里了解即可。特性以一对中括号开始和结束。要使用外部的动态链接库，需要使用 DllImport("externDll.dll")这样的语句，它其实就是构造方法。然后第 8 行使用 extern 关键字表示这个方法是外部导入进来的。外部方法在本程序集中声明时必须是静态的，方法的签名要和其在动态链接库中声明的一样。在使用外部方法前，要先像第 8 行这样进行方法的声明，然后再使用。使用方法就如第 11 行所演示的这样，和调用一个普通的方法没有什么区别。这段代码的执行结果如下：

```
这是一个外部方法, 参数 a 和参数 b 的值是: 11 22
```

最后还需要注意的是，导入的外部方法不能是泛型方法。

第十四章 接口

在类从基类进行继承时，只可以指定一个直接基类。在 C# 中是不允许指定多重继承的。也就是说，一个类不可以有多个基类。但是，如果要实现的功能不在这个指定的基类中，可以以接口的方式将要实现的功能写在接口中。然后让指定类实现接口。在实现接口的时候，可以指定实现多个接口，就好像 C++ 中的多重继承一样。接口更类似于一个协议，它代表某种功能，这个功能可能很多类都需要，那么它们都可以以实现接口的方法来继承接口中定义的功能。接口可以从多个接口继承，一个类也可以实现多个接口。接口的成员包括方法、属性、事件和索引器，但是接口本身不实现它们。它们等着实现接口的类来实现它们。

14.1 接口的定义

接口使用 `interface` 关键字进行定义，下面通过一段实例代码来演示接口的创建：

```
001 using System;
002 namespace InterfaceType1
003 {
004     interface Action
005     {
006         void Eat();
007         void Sleep();
008         void Fly();
009         void Sing();
010     }
011     class Bird : Action
012     {
013         public void Eat()
014         {
015             Console.WriteLine("我是一只麻雀，我喜欢吃小虫和谷子");
016         }
017         public void Sleep()
018         {
019             Console.WriteLine("我到晚上必须睡觉");
020         }
021         public void Fly()
022         {
023             Console.WriteLine("我们要成群结队地飞行");
024         }
025         public void Sing()
026         {
027             Console.WriteLine("我唱歌当然没有百灵好听");
028         }
029     }
030     class Program
031     {
032         static void Main(string[] args)
033         {
034             Bird Sparrow = new Bird();
```

```

035         Sparrow.Eat();
036         Sparrow.Sleep();
037         Sparrow.Fly();
038         Sparrow.Sing();
039         Console.ReadKey();
040     }
041 }
042 }

```

如同第 4 行演示的那样，定义接口的时候，使用 `interface` 关键字，后跟接口的名称。接口体用一对大括号开始和结束。在里面定义成员的时候，需要注意，成员不能指定任何访问权限修饰符，它隐式是 `public` 的。而且成员不能提供任何实现，它的实现需要实现它的类去做，接口中只能提供成员的声明。

第 11 行定义的类实现了这个接口，实现接口的方法和类的继承的语法一样。需要在类名后使用冒号，然后跟要实现的接口的名称。在类体中需要按照接口中成员的声明进行逐项的定义。实现接口中的成员时，访问权限只能是 `public` 的。

第 34 行创建了一个类的实例，然后这个实例可以调用实现的各个成员。就像类的继承那样，这些成员都好像是继承来的，唯一不同的是，接口中不定义实现，谁实现接口，谁实现接口中的那些成员。这段代码的执行结果如下：

```

我是一只麻雀，我喜欢吃小虫和谷子
我到晚上必须睡觉
我们要成群结队地飞行
我唱歌当然没有百灵好听

```

接口中的成员是隐式 `public` 的，不能为它指定其它访问权限修饰符，即使指定 `public` 也不行。实现接口的类在实现接口中的成员时也必须指定成员的访问权限为 `public` 的，其它访问权限不允许指定。下面是代码实例：

```

001     using System;
002     namespace InterfaceType1
003     {
004         interface Action
005         {
006             protected void Eat();
007             void Sleep();
008             void Fly();
009             void Sing();
010         }
011         class Sparrow : Action
012         {
013             protected void Eat()
014             {
015                 Console.WriteLine("我是一只麻雀，我喜欢吃小虫和谷子");
016             }
017             internal void Sleep()
018             {
019                 Console.WriteLine("我到晚上必须睡觉");
020             }

```



```

021         public void Fly()
022         {
023             Console.WriteLine("我们要成群结队地飞行");
024         }
025         public void Sing()
026         {
027             Console.WriteLine("我唱歌当然没有百灵好听");
028         }
029     }
030     class Program
031     {
032         static void Main(string[] args)
033         {
034             Sparrow mySparrow = new Sparrow();
035             Console.ReadKey();
036         }
037     }
038 }

```

第 6 行代码将接口中的成员定义为 protected 访问权限；第 13 行代码将实现的方法定义为 protected 访问权限；第 17 行代码将实现方法定义为 internal 访问权限。在编译时，编译器会报告错误，错误如下：

错误 1 修饰符“protected”对该项无效 H:\书稿 new\书稿源码\第十四章\InterfaceType1\Program.cs 6 24 InterfaceType1

下面将第 6 行代码中的访问权限修饰符去掉，这时再编译的时候，编译器又会提示如下的错误：

错误 1 “InterfaceType1.Sparrow”不实现接口成员“InterfaceType1.Action.Sleep()”。“InterfaceType1.Sparrow.Sleep()”无法实现接口成员，因为它不是公共的。 H:\书稿 new\书稿源码\第十四章\InterfaceType1\Program.cs 11 11 InterfaceType1

错误 2 “InterfaceType1.Sparrow”不实现接口成员“InterfaceType1.Action.Eat()”。“InterfaceType1.Sparrow.Eat()”无法实现接口成员，因为它不是公共的。 H:\书稿 new\书稿源码\第十四章\InterfaceType1\Program.cs 11 11 InterfaceType1

造成这两个错误的原因就是没有将实现的成员定义为 public 访问权限。

14.2 接口的其它成员

接口的成员只能是方法、属性、事件或索引器。它的成员不能是字段、常量、运算符、实例构造方法、析构方法、内部类和静态成员。原因就是，接口更像是待实现的某种功能，它没有实体。举个通俗一点例子来说，一个人可以继承它的父亲的全部财产，这个人可以看做派生类，它的父亲可以看做基类。它们都代表实体，因此有其数据成员。但是，这个人还想具备一些本领，比如绘画、音乐、舞蹈等。那么他可以去找会这些本领的老师学习。学成后，老师的本领他拥有了，但是不代表老师的个人财产他也继承下来了。所以这里的老师可以看做是接口，它可以看做是某种功能的模型。要具备什么功能，就实现什么接口。接口是为外部提供的功能规范，它本身不能是一件工具，不能用接口去做什么。所以，接口也不能含有静态成员。下面看代码实例：

```

001     using System;
002     namespace InterfaceType1
003     {
004         delegate void D();
005         interface Action

```

```
006     {
007         event D myD;
008         int Weight
009         {
010             get;
011             set;
012         }
013         void Eat();
014         void Sleep();
015         void Fly();
016         void Sing();
017     }
018     class Sparrow : Action
019     {
020         private int weight;
021         public event D myD;
022         public Sparrow()
023         {
024             Weight = 50;
025             myD += this.Sing;
026         }
027         public int Weight
028         {
029             get
030             {
031                 return weight;
032             }
033             set
034             {
035                 weight = value;
036             }
037         }
038         public void Eat()
039         {
040             Console.WriteLine("我是一只麻雀，我喜欢吃小虫和谷子");
041         }
042         public void Sleep()
043         {
044             Console.WriteLine("我到晚上必须睡觉");
045         }
046         public void Fly()
047         {
048             myD();
049             Console.WriteLine("我们要成群结队地飞行");
```

```

050         Console.WriteLine("我现在的体重是 {0} 克, 我得减肥了, 有点飞不动了", Weight);
051     }
052     public void Sing()
053     {
054         Console.WriteLine("叽叽喳喳");
055     }
056 }
057 class Program
058 {
059     static void Main(string[] args)
060     {
061         Sparrow mySparrow = new Sparrow();
062         mySparrow.Eat();
063         mySparrow.Sleep();
064         mySparrow.Fly();
065         Console.ReadKey();
066     }
067 }
068 }

```

第 4 行代码定义了一个委托, 在接口中, 使用这个委托类型定义了一个事件。第 8 行代码定义了一个属性, 在接口中的属性成员, 只写 get 和 set, 并且以分号结束。形式和自动实现的属性很相似, 但它不是自动实现的属性。接口是不代表物理存储位置的。因为索引器也包含 get 访问器和 set 访问器, 它和属性很相似, 所以这里以属性作为代表, 就不演示索引器了。

在实现接口的类 Sparrow 中, 第 20 行代码定义了一个 int 类型的字段, 它代表麻雀的体重。要实现接口中的事件, 只需要在类中使用 public 访问权限修饰符修饰这个事件就可以了。第 22 行是构造方法, 它对属性 Weight 进行赋值, 然后为事件 myD 添加调用列表。而这个属性是对接口中的属性成员的实现。属性在实现时也必须使用 public 访问权限关键字, 然后定义具体的 get 访问器和 set 访问器。在第 46 行的方法体中, 首先调用事件, 第 50 行引用属性的 get 访问器打印字段 weight 的值。这段代码的执行结果如下:

```

我是一只麻雀, 我喜欢吃小虫和谷子
我到晚上必须睡觉
叽叽喳喳
我们要成群结队地飞行
我现在的体重是 50 克, 我得减肥了, 有点飞不动了

```

要实现一个接口, 对于接口中定义的成员就要全部实现。不允许只实现其中一部分。

14.3 指定基类和接口

如果一个类从一个基类继承, 而且这个类还要实现接口。则在指定基类的时候, 基类的名称要放在第一位。对于隐式定义的基类, 则不需要特别指出。例如定义一个类, 它隐式从 Object 类继承 (前面例子中的 Sparrow 类); 定义一个结构, 它隐式从 System.ValueType 类继承。它们都不需要特别指定。但是从一个自定义类派生, 则需要指定基类。实例代码如下:

```

001     using System;
002     namespace InterfaceType1
003     {
004         delegate void D();
005         class Car

```

```
006     {
007         private string brand; //汽车的品牌
008         private int speed; //当前速度
009         private event D myD; //加减速时调用
010         public void SpeedUp() //加速
011         {
012             speed += 10;
013             myD();
014         }
015         public void Deceleration() //减速
016         {
017             if (speed > 0)
018             {
019                 speed -= 10;
020                 myD();
021             }
022             else
023             {
024                 Console.WriteLine("已经停车了，不能再减速了");
025             }
026         }
027         public Car(string name)
028         {
029             brand = name;
030             myD += this.Show;
031         }
032         public void Show()
033         {
034             Console.WriteLine("当前速度: {0} 公里/小时", speed);
035         }
036     }
037     interface Arms
038     {
039         void MachineGun(); //机枪
040         void Rocket(); //火箭弹
041         void Pilotless(); //无人驾驶
042     }
043     interface Flight
044     {
045         void Fly();
046     }
047     class NewCar : Car, Arms, Flight
048     {
049         public NewCar(string name)
```

```
050         : base(name)
051     {
052     }
053     public void MachineGun()
054     {
055         Console.WriteLine("轮胎侧上方两挺机枪开火");
056     }
057     public void Rocket()
058     {
059         Console.WriteLine("前灯下方火箭弹发射器发射");
060     }
061     public void Pilotless()
062     {
063         Console.WriteLine("开启无人驾驶系统");
064     }
065     public void Fly()
066     {
067         Console.WriteLine("现在开启飞行模式，可以飞上天空了");
068     }
069 }
070 class Program
071 {
072     static void Main(string[] args)
073     {
074         NewCar car007 = new NewCar("007 的座驾");
075         car007.SpeedUp();
076         car007.SpeedUp();
077         car007.MachineGun();
078         car007.Pilotless();
079         car007.Rocket();
080         car007.Fly();
081         Console.ReadKey();
082     }
083 }
084 }
```

这段代码模拟了 007 的汽车。第 4 行代码定义的委托类型是由类中的事件使用的。第 5 行代码定义了一个 Car 类，在它的内部定义了表示汽车频道，汽车当前速度的字段。第 9 行定义的事件由表示加速和减速的方法调用，每加速和减速一次，这个事件就被调用一次。在第 27 行的构造方法中，对表示汽车品牌的字段赋值，然后为事件添加显示当前速度的 Show 方法。

第 37 行代码定义了一个接口，它代表武器系统。本来汽车没有武器系统，但是可以通过实现这个接口来添加这些功能。

第 43 行代码另外定义了一个接口表示飞行的功能，普通的汽车也没有这样的功能，通过实现这个接口来添加飞行的能力。

第 47 行代码定义了一个 Car 类型的派生类，它表示 007 的汽车。注意它指定的基类的位置和要实现的

接口的位置。它的直接基类必须位于冒号后的第一位，然后是要实现的接口。对于派生来说，直接基类必须只有一个，但是实现的接口可以有多个。

第 49 行的构造方法中，使用 base 构造方法初始值设定项为基类的构造方法传值。从第 53 行开始，依次对要实现的两个接口中的所有成员进行实现代码的编写。这段代码的执行结果如下：

```
当前速度：10 公里/小时
当前速度：20 公里/小时
轮胎侧上方两挺机枪开火
开启无人驾驶系统
前灯下方火箭弹发射器发射
现在开启飞行模式，可以飞上天空了
```

如果在指定基类的时候，没有把基类的标识放在类名后的第一位。而是把接口放在第一位了，则编译器会报告如下的错误信息：

```
错误 1    “object” 不包含采用“1”个参数的构造函数 H:\书稿 new\书稿源码\第十四章
\InterfaceType1\Program.cs 50 15 InterfaceType1
错误 2    “InterfaceType1.NewCar” 不包含“SpeedUp”的定义，并且找不到可接受类型为
“InterfaceType1.NewCar”的第一个参数的扩展方法“SpeedUp”（是否缺少 using 指令或程序集引用？）
H:\书稿 new\书稿源码\第十四章\InterfaceType1\Program.cs 75 20 InterfaceType1
错误 3    “InterfaceType1.NewCar” 不包含“SpeedUp”的定义，并且找不到可接受类型为
“InterfaceType1.NewCar”的第一个参数的扩展方法“SpeedUp”（是否缺少 using 指令或程序集引用？）
H:\书稿 new\书稿源码\第十四章\InterfaceType1\Program.cs 76 20 InterfaceType1
```

在错误信息中没有明确告诉基类没有在继承列表的第一位，而是报告了基类成员找不到的错误。这是因为，编译器把指定在第一位的接口当做基类来处理了。

14.4 接口的隐式引用转换

接口与实现它的类之间也存在着隐式转换。实例代码如下：

```
001    using System;
002    namespace InterfaceType2
003    {
004        interface ITest
005        {
006            void Show1();
007            void Show2();
008        }
009        class Test2 : ITest
010        {
011            private int a;
012            private int b;
013            public Test2(int a, int b)
014            {
015                this.a = a;
016                this.b = b;
017            }
018            public void Show1()
019            {
020                Console.WriteLine("字段 a 的值是：{0}", a);
```

```

021     }
022     public void Show2()
023     {
024         Console.WriteLine("字段 b 的值是: {0}", b);
025     }
026 }
027 class Program
028 {
029     static void Main(string[] args)
030     {
031         Test2 myT = new Test2(100, 200);
032         ITest myIT = myT; //隐式转换
033         myIT.Show1();
034         myIT.Show2();
035         ITest myIT1 = new Test2(300, 400); //隐式转换
036         myIT1.Show1();
037         myIT1.Show2();
038         Console.ReadKey();
039     }
040 }
041 }

```

本段代码中，第 4 行定义了一个接口，它有两个方法成员。这两个方法成员用来打印类中的字段。第 9 行代码定义了一个类实现了接口 ITest。它对接口中的两个方法进行了实现。

第 31 行定义了一个类的实例，第 32 行将它赋值给了一个接口类型的变量。这时发生的就是隐式转换。第 35 行代码实现的是同样的效果，不过它更直接，它直接定义一个接口类型的变量指向了一个实现类的实例，这也是隐式转换。第 36 行和第 37 行使用接口类型的变量调用接口中的方法成员，实际它却调用了实现类中的已经实现的方法。这时发生的就是方法调用的多态。代码的执行结果如下：

```

字段 a 的值是: 100
字段 b 的值是: 200
字段 a 的值是: 300
字段 b 的值是: 400

```

14.5 接口作为参数传递

当接口作为参数进行传递的时候，依然可以发生隐式转换和多态调用。下面看代码实例：

```

001 using System;
002 namespace InterfaceType2
003 {
004     interface ITest
005     {
006         void Show1();
007         void Show2();
008     }
009     class Test2 : ITest
010     {
011         private int a;

```

```

012     private int b;
013     public Test2(int a, int b)
014     {
015         this.a = a;
016         this.b = b;
017     }
018     public void Show1()
019     {
020         Console.WriteLine("字段 a 的值是: {0}", a);
021     }
022     public void Show2()
023     {
024         Console.WriteLine("字段 b 的值是: {0}", b);
025     }
026     public void Invoke(ITest i)
027     {
028         i.Show1();
029         i.Show2();
030     }
031 }
032 class Program
033 {
034     static void Main(string[] args)
035     {
036         Test2 myTest = new Test2(300, 400);
037         myTest.Invoke(myTest);
038         Console.ReadKey();
039     }
040 }
041 }

```

这段代码中第 26 行又增加了一个实例方法 Invoke，这个方法带有一个接口类型的参数。由接口类型的变量调用接口成员方法。第 36 行代码定义了一个 Test2 类型的变量，并创建了实例。第 37 行将这个变量传入 Invoke 方法。这时首先会发生隐式转换，方法的局部变量 i 指向了实现类的实例。然后使用这个变量调用接口方法的时候，发生了多态调用。实现类的方法实际被调用。代码的执行结果如下：

```

字段 a 的值是: 300
字段 b 的值是: 400

```

由此可见，接口虽然和类的继承过程中发生的虚方法和重写不同，它是通过对接口中的成员方法声明在实现类中实现的办法来达到多态的目的。

14.6 接口的继承

不只类可以从其它类派生，接口也可以从其它接口继承，而且可以从多个接口继承，就好像 C++ 中的多重继承一样。实例代码如下：

```

001     using System;
002     namespace InterfaceType2
003     {

```



```
004     interface ITest1
005     {
006         void Show1();
007         void Show2();
008     }
009     interface ITest2
010     {
011         void Show3();
012     }
013     interface ITest3 : ITest1, ITest2
014     {
015         void Show4();
016     }
017     class Test2 : ITest3
018     {
019         private int a;
020         private int b;
021         private int c;
022         private int d;
023         public Test2(int a, int b,int c,int d)
024         {
025             this.a = a;
026             this.b = b;
027             this.c = c;
028             this.d = d;
029         }
030         public void Show1()
031         {
032             Console.WriteLine("字段 a 的值是: {0}", a);
033         }
034         public void Show2()
035         {
036             Console.WriteLine("字段 b 的值是: {0}", b);
037         }
038         public void Show3()
039         {
040             Console.WriteLine("字段 c 的值是: {0}", c);
041         }
042         public void Show4()
043         {
044             Console.WriteLine("字段 d 的值是: {0}", d);
045         }
046         public void Invoke(ITest1 i)
047         {
```

```
048         i.Show1();
049         i.Show2();
050     }
051     public void Invoke1(ITest3 i)
052     {
053         i.Show3();
054         i.Show4();
055     }
056 }
057 struct St : ITest3
058 {
059     private int a;
060     private int b;
061     private int c;
062     private int d;
063     public St(int a, int b, int c, int d)
064     {
065         this.a = a;
066         this.b = b;
067         this.c = c;
068         this.d = d;
069     }
070     public void Show1()
071     {
072         Console.WriteLine("结构字段 a 的值是: {0}", a);
073     }
074     public void Show2()
075     {
076         Console.WriteLine("结构字段 b 的值是: {0}", b);
077     }
078     public void Show3()
079     {
080         Console.WriteLine("结构字段 c 的值是: {0}", c);
081     }
082     public void Show4()
083     {
084         Console.WriteLine("结构字段 d 的值是: {0}", d);
085     }
086 }
087 class Program
088 {
089     static void Main(string[] args)
090     {
091         Test2 myTest = new Test2(100, 200, 300, 400);
```

```

092         myTest.Invoke(myTest);
093         myTest.Invoke1(myTest);
094         ITest3 i = new St(500, 600, 700, 800);
095         i.Show1();
096         i.Show2();
097         i.Show3();
098         i.Show4();
099         Console.ReadKey();
100     }
101 }
102 }

```

这段代码中的第 4 行和第 9 行代码各定义了一个独立的接口，第 13 行代码定义的接口 ITest3 同时从这两个接口派生。这时，它将拥有这两个基接口中所有定义的接口成员。第 17 行定义的类实现了接口 ITest3。当类实现某个接口的时候，它将实现该接口继承下来的所有成员。所以，类 Test2 将要实现 4 个方法。第 46 行定义的 Invoke 方法以 ITest1 接口的变量作为参数，那么它可以调用的方法只能是 ITest1 中所定义的接口方法。第 51 行定义的方法 Invoke1 以 ITest3 接口的变量作为参数，它可以调用它自己的包括继承下来的全部接口方法。但是这里只让它调用了它的直接基接口的方法和它自己定义的方法。当第 92 行代码和第 93 行代码调用 Invoke 和 Invoke1 方法的时候，都会发生隐式转换和多态。

不仅仅类能实现接口，结构也可以实现接口。第 57 行代码定义了一个结构 St，它实现了 ITest3 接口。那么它就把 ITest3 自己的和它继承自基接口的全部接口成员都实现一遍。第 94 行代码定义了一个 ITest3 的变量指向了结构的实例，这时接口 ITest3 的变量调用继承下来的接口成员都会发生多态调用。这段代码的执行结果如下：

```

字段 a 的值是：100
字段 b 的值是：200
字段 c 的值是：300
字段 d 的值是：400
结构字段 a 的值是：500
结构字段 b 的值是：600
结构字段 c 的值是：700
结构字段 d 的值是：800

```

当类实现某个接口后，它的派生类仍可以重写此类实现的接口的方法，前提是此类实现接口的方法时，再加上 virtual 关键字。代码实例如下：

```

001     using System;
002     namespace InterfaceType2
003     {
004         interface ITest1
005         {
006             void Show1();
007             void Show2();
008         }
009         class Test2 : ITest1
010         {
011             private int a;
012             private int b;

```

```
013     public Test2(int a, int b)
014     {
015         this.a = a;
016         this.b = b;
017     }
018     public virtual void Show1()
019     {
020         Console.WriteLine("字段 a 的值是: {0}", a);
021     }
022     public virtual void Show2()
023     {
024         Console.WriteLine("字段 b 的值是: {0}", b);
025     }
026 }
027 class Test3 : Test2
028 {
029     public Test3(int a, int b)
030         : base(a, b)
031     { }
032     public override void Show1()
033     {
034         Console.WriteLine("此时调用的是类 Test3 中重写的基类的实现方法 Show1");
035         base.Show1();
036     }
037     public override void Show2()
038     {
039         Console.WriteLine("此时调用的是类 Test3 中重写的基类的实现方法 Show2");
040         base.Show2();
041     }
042 }
043 class Program
044 {
045     static void Main(string[] args)
046     {
047         ITest1 myIT = new Test3(100, 200);
048         myIT.Show1();
049         myIT.Show2();
050         Console.ReadKey();
051     }
052 }
053 }
```

类 Test2 实现了接口 ITest1, 但是在实现接口 ITest1 中的方法时, 又加上了 virtual 关键字, 将实现的方法又定义成了可重写的虚方法。

第 27 行代码定义的类 Test3 从类 Test2 派生, 在类 Test3 中又重写了基类中的虚方法。第 47 行定义

了一个接口 ITest1 的变量 myIT，它指向了一个 Test3 类的实例。当使用 myIT 调用接口中的方法时，发生了多态调用。代码的执行结果如下：

此时调用的是类 Test3 中重写的基类的实现方法 Show1

字段 a 的值是：100

此时调用的是类 Test3 中重写的基类的实现方法 Show2

字段 b 的值是：200

14.7 分部接口

除了类和结构可以是分部的，接口也可以是分部的。下面看代码实例：

```
001 using System;
002 namespace InterfaceType2
003 {
004     partial interface ITest1
005     {
006         void Show1();
007         void Show2();
008     }
009     partial interface ITest1
010     {
011         void Show3();
012         void Show4();
013     }
014     class Test2 : ITest1
015     {
016         private int a;
017         private int b;
018         private int c;
019         private int d;
020         public Test2(int a, int b, int c, int d)
021         {
022             this.a = a;
023             this.b = b;
024             this.c = c;
025             this.d = d;
026         }
027         public void Show1()
028         {
029             Console.WriteLine("字段 a 的值是：{0}", a);
030         }
031         public void Show2()
032         {
033             Console.WriteLine("字段 b 的值是：{0}", b);
034         }
035         public void Show3()
036         {
```

```

037         Console.WriteLine("字段 c 的值是: {0}", c);
038     }
039     public void Show4()
040     {
041         Console.WriteLine("字段 d 的值是: {0}", d);
042     }
043 }
044 class Program
045 {
046     static void Main(string[] args)
047     {
048         ITest1 myIT = new Test2(100, 200, 300, 400);
049         myIT.Show1();
050         myIT.Show2();
051         myIT.Show3();
052         myIT.Show4();
053         Console.ReadKey();
054     }
055 }
056 }

```

分部的接口使用 `partial` 关键字进行修饰，它们有着一样的名字。第 4 行和第 9 行各定义了一个分部的接口。在最后编译器进行编译的时候，它们将合在一起处理。第 14 行定义类实现接口时，将把每个分部接口中的成员都要进行一一实现。第 48 行定义了一个接口类型的变量指向了实现类的实例，在调用接口中的成员方法时，将发生多态调用。代码的执行结果如下：

```

字段 a 的值是: 100
字段 b 的值是: 200
字段 c 的值是: 300
字段 d 的值是: 400

```

14.8 显式接口成员实现

一个类的基类只能有一个。但是，类或结构可以实现的接口却可以有多个。当一个类或结构实现多个接口的时候，有可能遇到多个接口中包含同名的成员。因此，这就会造成命名冲突。先看一下下面的代码实例：

```

001 using System;
002 namespace InterfaceType2
003 {
004     interface ITest1
005     {
006         void Show();
007     }
008     interface ITest2
009     {
010         void Show();
011     }
012     interface ITest3

```

```
013     {
014         void Show();
015     }
016     class Test : ITest1, ITest2, ITest3
017     {
018         private int a;
019         private int b;
020         private int c;
021         public Test(int a, int b, int c)
022         {
023             this.a = a;
024             this.b = b;
025             this.c = c;
026         }
027         public void Show()
028         {
029             Console.WriteLine("字段 a 的值是: {0}", a);
030         }
031     }
032     class Program
033     {
034         static void Main(string[] args)
035         {
036             Test t = new Test(100, 200, 300);
037             ITest1 it1 = t;
038             it1.Show();
039             ITest2 it2 = t;
040             it2.Show();
041             ITest3 it3 = t;
042             it3.Show();
043             Console.ReadKey();
044         }
045     }
046 }
```

在本程序中的三个接口 ITest1、ITest2 和 ITest3 中，都包含了一个同名的成员方法 Show。第 16 行代码中的类 Test 实现了这三个接口。第 27 行代码中，对于接口中的 Show 方法进行了实现。但是，因为方法同名的关系，这个实现等于是将每个接口中的方法都进行了同名的实现。虽然每个接口中的方法名称相同，但是现在想要它们完成不同的功能，比如，使用 ITest1 的引用来调用方法，打印字段 a 的值；而使用 ITest2 的引用来调用方法，则打印 b 的值。但是现在的情况是，从第 37 行开始，每个不同的引用调用 Show 方法都打印了 a 的值。

对这一情况的解决办法就是使用接口成员的显式实现。在进行接口成员的显式实现之前，先来介绍一下接口成员的完全限定名称。实例代码如下：

```
001     using System;
002     namespace InterfaceType1
```

```
003  {
004      interface A
005      {
006          void Show1();
007      }
008      interface B : A
009      {
010          void Show2();
011      }
012  }
```

在这段代码中，命名空间 `InterfaceType1` 下面定义了两个接口 `A` 和 `B`。接口 `B` 从接口 `A` 继承。这时如果有一个类实现接口 `B`，那么在这个类中，`Show1` 方法的完全限定名称就是 `InterfaceType1.A.Show1()`；而 `Show2` 的完全限定名称就是 `InterfaceType1.B.Show2()`。虽然接口 `B` 继承自接口 `A`，但和类不同，不能使用 `InterfaceType1.B.Show1()` 来引用 `Show1()`。

当进行接口成员的显式实现的时候，就是使用接口成员的完全限定名称来实现接口成员。下面是代码实例：

```
001  using System;
002  namespace InterfaceType2
003  {
004      interface ITest1
005      {
006          void Show();
007      }
008      interface ITest2
009      {
010          void Show();
011      }
012      interface ITest3
013      {
014          void Show();
015      }
016      class Test : ITest1, ITest2, ITest3
017      {
018          private int a;
019          private int b;
020          private int c;
021          public Test(int a, int b, int c)
022          {
023              this.a = a;
024              this.b = b;
025              this.c = c;
026          }
027          void ITest1.Show()
028          {
```



```

029         Console.WriteLine("字段 a 的值是: {0}", a);
030     }
031     void ITest2.Show()
032     {
033         Console.WriteLine("字段 b 的值是: {0}", b);
034     }
035     void ITest3.Show()
036     {
037         Console.WriteLine("字段 c 的值是: {0}", c);
038     }
039 }
040 class Program
041 {
042     static void Main(string[] args)
043     {
044         Test t = new Test(100, 200, 300);
045         ITest1 it1 = t;
046         it1.Show();
047         ITest2 it2 = t;
048         it2.Show();
049         ITest3 it3 = t;
050         it3.Show();
051         Console.ReadKey();
052     }
053 }
054 }

```

这段代码在类 `Test` 中实现接口的成员方法时，使用完全限定名称引用接口成员，然后予以实现。类 `Test` 也定义在命名空间 `InterfaceType2` 中，所以在引用接口的成员的时候，可以不使用命名空间的名称。在开发环境显式实现接口的时候，可以在类名后面引入的用逗号分隔的基接口的名称列表上的每个接口名上单击右键，然后选择“实现接口”，再在下一级菜单上选择“显式实现接口”。这时，开发环境会自动用接口成员的完全限定名称来生成类的方法成员。程序员只要写方法中的实现代码即可，大大方便了代码的编写。接口成员显式实现后，即可实现用不同的接口引用调用不同的类成员方法，实现了真正意义上的多态。代码执行结果如下：

```

字段 a 的值是: 100
字段 b 的值是: 200
字段 c 的值是: 300

```

仔细观察显式接口成员实现的代码即可发现，每个显式实现的接口成员都没有访问权限修饰符。也就是说，它们都是 `private` 访问权限的。因此，直接用类的变量是无法引用它们的。要访问它们的唯一办法就是用接口名称的变量来引用。也就是用多态的方法来引用。如果为显式实现的接口成员加上访问权限修饰符，编译器就会报告错误。下面看代码实例：

```

001     using System;
002     namespace InterfaceType2
003     {
004         interface ITest1

```

```
005     {
006         void Show();
007     }
008     interface ITest2
009     {
010         void Show();
011     }
012     interface ITest3
013     {
014         void Show();
015     }
016     class Test : ITest1, ITest2, ITest3
017     {
018         private int a;
019         private int b;
020         private int c;
021         public Test(int a, int b, int c)
022         {
023             this.a = a;
024             this.b = b;
025             this.c = c;
026         }
027         public void ITest1.Show()
028         {
029             Console.WriteLine("字段 a 的值是: {0}", a);
030         }
031         protected virtual void ITest2.Show()
032         {
033             Console.WriteLine("字段 b 的值是: {0}", b);
034         }
035         void ITest3.Show()
036         {
037             Console.WriteLine("字段 c 的值是: {0}", c);
038         }
039     }
040     class Program
041     {
042         static void Main(string[] args)
043         {
044             Test t = new Test(100, 200, 300);
045             //t.Show();
046             //t.ITest1.Show();
047             ITest1 it1 = t;
048             it1.Show();
```

```

049         ITest2 it2 = t;
050         it2.Show();
051         ITest3 it3 = t;
052         it3.Show();
053         Console.ReadKey();
054     }
055 }
056 }

```

在代码的第 27 行为 ITest1.Show 方法加上了 public 访问权限修饰符, 代码的第 31 行为 ITest2.Show 方法加上了 protected virtual 关键字, 目的是在派生类中对它进行重写。但是, 这在语法上是不允许的。编译器会报错, 错误信息如下:

```

错误 1    修饰符“public”对该项无效 H:\书稿 new\书稿源码\第十四章\InterfaceType2\Program.cs
          27 21 InterfaceType2
错误 2    修饰符“virtual”对该项无效 H:\书稿 new\书稿源码\第十四章\InterfaceType2\Program.cs
          31 32 InterfaceType2
错误 3    修饰符“protected”对该项无效 H:\书稿 new\书稿源码\第十四章\InterfaceType2\Program.cs 31 32 InterfaceType2

```

如果将加上的访问权限修饰符和虚方法关键字去掉, 将第 45 行和第 46 行的代码中的注释去掉, 编译器则会报告如下的错误:

```

错误 1    “InterfaceType2.Test”不包含“Show”的定义, 并且找不到可接受类型为
          “InterfaceType2.Test”的第一个参数的扩展方法“Show”(是否缺少 using 指令或程序集引用?) H:\
          书稿 new\书稿源码\第十四章\InterfaceType2\Program.cs 45 15 InterfaceType2
错误 2    “InterfaceType2.Test”不包含“ITest1”的定义, 并且找不到可接受类型为
          “InterfaceType2.Test”的第一个参数的扩展方法“ITest1”(是否缺少 using 指令或程序集引用?)
          H:\书稿 new\书稿源码\第十四章\InterfaceType2\Program.cs 46 15 InterfaceType2

```

可以看到, 直接使用类的变量无法引用显式实现的接口成员, 即使使用完全限定名称也不行。显式实现的接口成员没有访问权限修饰符, 默认它们是 private 的。但是使用多态的方式又可以访问它们, 在功能上来说, 它们仿佛又是 public 的。因为显式实现的接口是 private 访问权限的, 所以它们可以被类内部的其它方法成员所访问。也就是说, 这些被实现的接口成员用来被内部调用, 而不对外发布。在使用类的变量引用类的成员时, 在智能提示中看不到这些显式实现的接口成员。

14.9 显式引用转换

前面介绍过隐式引用转换, 在隐式引用转换过程中, 基类的引用可以指向派生类的实例。隐式引用转换总是成功的。但是显式引用转换就不一定了。显式引用转换是指派生类的引用指向了基类的实例。下面先观察一段代码:

```

001 using System;
002 namespace InterfaceType1
003 {
004     class Father
005     {
006         private int a;
007         public Father(int a)
008         {
009             this.a = a;
010         }

```

```

011         public virtual void Show()
012         {
013             Console.WriteLine("父亲的年龄是: {0}", a);
014         }
015     }
016     class Son : Father
017     {
018         private int b;
019         public Son(int a, int b)
020             : base(a)
021         {
022             this.b = b;
023         }
024         public override void Show()
025         {
026             Console.WriteLine("儿子的年龄是: {0}", b);
027         }
028     }
029     class Program
030     {
031         static void Main(string[] args)
032         {
033             Son s = new Father(50);
034             s.Show();
035         }
036     }
037 }

```

这段代码中首先定义了一个类 Father，它是基类。然后定义了一个 Son 类从 Father 类派生。基类中又定义了一个打印字段的方法 Show。它标记为虚方法，在派生类中重写这个方法以实现重写。

在第 31 行，定义了一个派生类的引用指向了基类的实例，然后调用派生类的 Show 方法。在编译时，编译器会报告如下错误：

错误 1 无法将类型“InterfaceType1.Father”隐式转换为“InterfaceType1.Son”。存在一个显式转换(是否缺少强制转换?) E:\书稿 new\书稿源码\第十四章\InterfaceType1\Program.cs 33 21

InterfaceType1

错误 1 提示将基类无法隐式转换为子类，存在一个显式转换。这是因为，从基类是无法向派生类进行隐式转换的。下面修改代码如下：

```

Son s = (Son)new Father(50);
s.Show();

```

在修改的代码中，使用强制转换的方式将基类实例强制转换为派生类。但是这时编译器还会报错，信息如下：

未处理 InvalidCastException

无法将类型为“InterfaceType1.Father”的对象强制转换为类型“InterfaceType1.Son”。

产生这个错误的原因是，在进行引用类型的显式转换的时候，能否正确执行显式转换，要根据实例的类型判断显式转换是否能成功。引用类型之间的转换有这样的规则，基类可以指向子类的实例，但是子类不能

指向基类的实例。举个通俗一点的例子来说明这一点，比如说人制造了机器人，机器人就好比派生类。人可以指挥机器人的行动。但是反过来，机器人不可以指挥人的行动。将它换成引用类型来讲，就是基类的引用可以指向子类的实例，但是子类的引用不允许指向基类的实例。注意，这里指的是实例，而不是变量。再观察下面的代码：

```
001 object o = new Son(50, 20);
002 Father f = o;
003 f.Show();
004 Console.ReadKey();
```

第一行代码中将一个 Son 类型的实例隐式转换为 object 类型，然后将这个 object 类型的变量赋给 Father 类型的变量。但是，编译器会报告如下错误：

错误 1 无法将类型“object”隐式转换为“InterfaceType1.Father”。存在一个显式转换(是否缺少强制转换?) E:\书稿 new\书稿源码\第十四章\InterfaceType1\Program.cs 34 24 InterfaceType1

这是编译时进行的检查，在 object 和 Father 类型之间是无法隐式转换的。下面再修改代码如下：

```
001 object o = new Son(50, 20);
002 Father f = (Father)o;
003 f.Show();
004 Console.ReadKey();
```

在第 2 行代码中将 object 类型的变量通过显式转换的方法转换为 Father 类型。在这个过程中将在运行时进行类型检查。是否转换能成功，要看转换的实际类型是否符合基类引用指向子类实例的规则。因为 o 实际指向的类型是 Son 类型，它能够转换给它的基类 Father。所以这个转换能成功。代码的执行结果如下：

儿子的年龄是：20

对于接口来说，同样存在着这样的显式转换。实例代码如下：

```
001 using System;
002 namespace InterfaceType1
003 {
004     interface Sing
005     {
006         void sing();
007     }
008     class Father
009     {
010         private int a;
011         public Father(int a)
012         {
013             this.a = a;
014         }
015         public virtual void Show()
016         {
017             Console.WriteLine("父亲的年龄是：{0}", a);
018         }
019     }
020     class Son : Father, Sing
021     {
022         private int b;
```

```

023         public Son(int a, int b)
024             : base(a)
025         {
026             this.b = b;
027         }
028         public override void Show()
029         {
030             base.Show();
031             Console.WriteLine("儿子的年龄是: {0}", b);
032         }
033         public void sing()
034         {
035             Console.WriteLine("儿子会唱歌");
036         }
037     }
038     class Program
039     {
040         static void Main(string[] args)
041         {
042             Sing s = new Son(50, 20);
043             s.sing();
044             Father f = (Father)s;
045             f.Show();
046             Console.ReadKey();
047         }
048     }
049 }

```

在代码的第 4 行定义了一个接口，它是一个关于唱歌的功能。第 20 行，Son 类从基类 Father 类派生后，又实现了这个接口。它表示，儿子从爸爸那里继承成员之后，儿子又会了一项新本领，他会唱歌。在第 42 行，可以使用接口的引用指向实现类的实例，它调用 sing 方法实现多态。第 44 行，从 Sing 接口到 Father 类不存在隐式的转换，但是允许在编译时通过显式转换的方法对它们进行转换。具体能不能成功，要看运行时对类实例的检查，因为 s 最终的实例是 Son 类型，它是可以隐式转换为 Father 类的。所以这里的显式转换能够成功。代码的执行结果如下：

```

儿子会唱歌
父亲的年龄是: 50
儿子的年龄是: 20

```

14.10 值类型装箱之后的显式转换

值类型装箱为引用类型之后，也允许进行引用类型之间的显式转换。下面看代码实例：

```

001 using System;
002 namespace InterfaceType1
003 {
004     interface Sing
005     {
006         void sing();

```

```

007     }
008     struct Son : Sing
009     {
010         private int b;
011         public Son(int b)
012         {
013             this.b = b;
014         }
015         public void Show()
016         {
017             Console.WriteLine("儿子的年龄是: {0}", b);
018         }
019         public void sing()
020         {
021             Console.WriteLine("儿子会唱歌");
022         }
023     }
024     class Program
025     {
026         static void Main(string[] args)
027         {
028             object o = new Son(20);
029             Sing s = (Sing)o;
030             s.sing();
031             Son s1 = (Son)s;
032             s1.Show();
033             Console.ReadKey();
034         }
035     }
036 }

```

本段代码中，第 8 行定义了一个结构 Son，它实现了 Sing 接口。在第 28 行，定义了一个 object 类型的变量，然后将 Son 结构的实例装箱后赋值给了 object 类型的变量 o。这时，结构的实例从栈中复制到了堆中。第 29 行定义了一个接口 Sing 类型的变量，然后将 object 类型强制转换成了 Sing 类型。因为结构 Son 实现了接口 Sing，所以这个强制转换是能够成功的。

第 31 行将 Sing 类型的变量强制转换成了 Son 类型。这是一个取消装箱转换，因为 s 实际的类型是装箱后的结构，所以，这个取消装箱转换也能成功。结构被从堆中再次复制到了栈中。代码的执行结果如下：

儿子会唱歌

儿子的年龄是: 20

14.11 is 运算符

is 运算符的作用是动态检查对象的运行时类型是否与指定类型兼容。is 运算符运算的结果为 bool 类型，表示检查的类型是否可以转换成目标类型。下面看代码实例：

```

001     using System;
002     namespace InterfaceType1
003     {

```

```
004     interface A
005     {
006         void Show();
007     }
008     class B : A
009     {
010         public void Show()
011         {
012             Console.WriteLine("类 B 中实现接口成员的 Show 方法");
013         }
014     }
015     struct C : A
016     {
017         public void Show()
018         {
019             Console.WriteLine("结构 C 中实现接口成员的 Show 方法");
020         }
021     }
022     class Program
023     {
024         static void Main(string[] args)
025         {
026             int a = 100;
027             if (a is int)
028             {
029                 Console.WriteLine("a--int 可以成功转换");
030             }
031             else
032             {
033                 Console.WriteLine("a--int 类型不兼容，不可以成功转换");
034             }
035             if (a is int?)
036             {
037                 Console.WriteLine("a--int?可以成功转换");
038             }
039             else
040             {
041                 Console.WriteLine("a--int?类型不兼容，不可以成功转换");
042             }
043             if (a is object)
044             {
045                 Console.WriteLine("a--object 可以成功转换");
046             }
047             else
```



```
048      {
049          Console.WriteLine("a--object 类型不兼容，不可以成功转换");
050      }
051      B myB = new B();
052      C myC = new C();
053      if (myB is C)
054      {
055          Console.WriteLine("myB--C 可以成功转换");
056      }
057      else
058      {
059          Console.WriteLine("myB--C 类型不兼容，不可以成功转换");
060      }
061      if (myC is B)
062      {
063          Console.WriteLine("myC--B 可以成功转换");
064      }
065      else
066      {
067          Console.WriteLine("myC--B 类型不兼容，不可以成功转换");
068      }
069      if (myB is A)
070      {
071          Console.WriteLine("myB--A 可以成功转换");
072      }
073      else
074      {
075          Console.WriteLine("myB--A 类型不兼容，不可以成功转换");
076      }
077      if (myC is A)
078      {
079          Console.WriteLine("myC--A 可以成功转换");
080      }
081      else
082      {
083          Console.WriteLine("myC--A 类型不兼容，不可以成功转换");
084      }
085      if (null is A)
086      {
087          Console.WriteLine("null--A 可以成功转换");
088      }
089      else
090      {
091          Console.WriteLine("null--A 类型不兼容，不可以成功转换");
```

```

092         }
093         Console.ReadKey();
094     }
095 }
096 }

```

is 运算符的左边是变量，运算符的右边是要转换到的类型。代码第 4 行定义了一个接口 A，第 8 行和第 15 行的类和结构都实现了接口 A。代码第 26 行定义了一个 int 类型的变量并赋了初始值。第 27 行判断，a 是否能转换成 int。如果变量的类型和目标类型相同，is 运算符返回 true。第 35 行判断变量能否转换成基础类型与它相同的可空类型，在这种情况下，is 运算符也将返回 true。第 43 行判断 a 能否转换成 object 类型，因为 a 可以通过装箱的方式隐式转换成 object 类型，所以也将返回 true。第 51 行定义了类 B 的一个引用变量并实例化；第 52 行定义了结构 C 的一个变量并实例化。第 53 行判断 B 类型的变量能否转换成 C 类型，因为 myB 的运行时类型是 B，类型 B 和结构 C 之间不存在隐式转换，所以 is 运算符将返回 false。第 61 行判断结构类型的变量 myC 能否转换成类型 B，和上一个例子同理，它也将返回 false。第 69 行判断 myB 能否转换成接口 A 类型，因为 B 实现了 A，所以 is 运算符将返回 true。第 77 行判断结构类型的变量 myC 能否转换成接口 A 类型，同理，因为结构实现了接口，所以将返回 true。第 85 行判断 null 文本能否转换成接口 A，即使接口 A 是引用类型，但是只要 is 运算符的左操作数是 null，或者左操作数是引用类型，但是它为 null，is 运算符都将返回 false。这段代码的执行结果如下：

```

a--int 可以成功转换
a--int?可以成功转换
a--object 可以成功转换
myB--C 类型不兼容，不可以成功转换
myC--B 类型不兼容，不可以成功转换
myB--A 可以成功转换
myC--A 可以成功转换
null--A 类型不兼容，不可以成功转换

```

这段代码是演示了 is 运算符的判断方式，下面再演示一下它的应用。代码如下：

```

001     using System;
002     namespace InterfaceType1
003     {
004         interface A
005         {
006             void Show();
007         }
008         class B : A
009         {
010             public void Show()
011             {
012                 Console.WriteLine("类 B 中实现接口成员的 Show 方法被调用");
013             }
014         }
015         class Program
016         {
017             static void Main(string[] args)
018             {

```

```

019         B myB1 = new B();
020         B myB2 = null;
021         object o = myB1;
022         if (myB2 is A)
023         {
024             Console.WriteLine("空引用可以转换成接口类型 A");
025         }
026         else
027         {
028             Console.WriteLine("空引用不可以转换成接口类型 A");
029         }
030         if (o is A)
031         {
032             A myA = (A)o;
033             myA.Show();
034         }
035         else
036         {
037             Console.WriteLine("o 的运行时类型不与 A 类型兼容，不可以转换");
038         }
039         Console.ReadKey();
040     }
041 }
042 }

```

这段代码定义了一个接口 A，类 B 实现了接口 A。在 Main 方法中，第 19 行定义了一个类 B 的实例，第 20 行定义了一个类 B 的变量，然后将 null 赋值给它。第 21 行将 myB1 隐式转换成 object 类型。第 22 行判断 myB2 能否转换成 A 类型，因为 myB2 是空引用，所以第 22 行的 is 运算符将返回 false。第 30 行就是一般环境下，is 运算符的用法。如果不知道 o 的运行时类型是什么类型，那么就可以使用 is 运算符将变量与目标类型进行判断。因为 o 的运行时类型是 B 类型，而 B 类型实现了接口 A。所以这里的判断结果是 true。if 语句的代码块内的内容将被执行。在第 32 行，o 被显式转换成 A 类型，然后调用接口中的成员方法 Show。这时将发生多态的调用，类中的实现方法被正确调用。如果在实际编程过程中，不对两个类型能否正确转换进行判断，将导致异常的发生。

14.12 as 运算符

as 运算符用于类型之间转换时使用。与 is 运算符返回 bool 类型值不同，使用 as 运算符，如果可以正确转换，它将返回左操作数；如果不可以转换，它将返回 null。as 运算符的右操作数必须是引用类型或可以为 null 的类型。使用 as 运算符不会发生异常。下面是代码实例：

```

001     using System;
002     namespace InterfaceType1
003     {
004         interface A
005         {
006             void Show();
007         }
008         class B : A

```

```
009      {
010          public void Show()
011          {
012              Console.WriteLine("类 B 中实现接口成员的 Show 方法被调用");
013          }
014      }
015      class Program
016      {
017          static void Main(string[] args)
018          {
019              B myB1 = new B();
020              B myB2 = null;
021              object o = myB1 as object;
022              if (o == null)
023              {
024                  Console.WriteLine("转换不成功");
025              }
026              else
027              {
028                  A myA1 = o as A;
029                  if (myA1 == null)
030                  {
031                      Console.WriteLine("转换不成功");
032                  }
033                  else
034                  {
035                      myA1.Show();
036                  }
037              }
038              A myA2 = myB2 as A;
039              if (myA2 == null)
040              {
041                  Console.WriteLine("转换成功");
042              }
043              Console.ReadKey();
044          }
045      }
046  }
```

这段代码中接口与类的关系还是沿用前面的例子，只对 Main 方法中的代码做了修改。第 19 行代码定义了一个 B 类型的引用并做了实例化。第 20 行代码定义了一个 B 类型的空引用。第 21 行使用 as 运算符进行判断并进行类型转换。它的意思是，如果 myB1 能够转换成 object 类型，则把 myB1 转换给 object 类型的变量 o，如果转换不成功，则将 null 赋值给 o。第 22 行对转换运算后的 o 进行判断，如果不为空，则说明转换成功了。这时定义一个 A 类型的变量 myA1，然后将 o 再通过 as 运算符进行判断。如果 o 能够转换成 A 类型，则将 o 转换给 myA1。否则将 null 赋值给它。第 29 行对转换运算后的 myA1 进行判断，如果不

为空，则调用接口成员方法 Show。第 38 行使用 as 运算符判断 myB2 能否转换成 A 类型。使用 as 判断一个空引用能否转换给一个引用类型，总是返回 null。这段代码的执行结果如下：

类 B 中实现接口成员的 Show 方法被调用

转换成功

14.13 接口成员显示实现的内部调用

前面介绍过，接口成员显式实现时隐式为 private。它可以被内部方法调用，而不对外发布。在内部调用的时候，可以使用 is 或 as 运算符对传入的变量进行类型检查以调用正确的显式实现方法。实例代码如下：

```
001 using System;
002 namespace InterfaceType2
003 {
004     interface ITest1
005     {
006         void Show();
007     }
008     interface ITest2
009     {
010         void Show();
011     }
012     interface ITest3
013     {
014         void Show();
015     }
016     class Test : ITest1, ITest2, ITest3
017     {
018         private int a;
019         private int b;
020         private int c;
021         public Test(int a, int b, int c)
022         {
023             this.a = a;
024             this.b = b;
025             this.c = c;
026         }
027         void ITest1.Show()
028         {
029             Console.WriteLine("字段 a 的值是: {0}", a);
030         }
031         void ITest2.Show()
032         {
033             Console.WriteLine("字段 b 的值是: {0}", b);
034         }
035         void ITest3.Show()
036         {
```

```
037         Console.WriteLine("字段 c 的值是: {0}", c);
038     }
039     public void Invoke(Test t)
040     {
041         if (t is ITest1)
042         {
043             ITest1 it1 = t;
044             it1.Show();
045         }
046         if (t is ITest2)
047         {
048             ITest2 it2 = t;
049             it2.Show();
050         }
051         if (t is ITest3)
052         {
053             ITest3 it3 = t;
054             it3.Show();
055         }
056     }
057 }
058 class Program
059 {
060     static void Main(string[] args)
061     {
062         Test t = new Test(100, 200, 300);
063         t.Invoke(t);
064         Console.ReadKey();
065     }
066 }
067 }
```

在这段代码中的第 39 行定义了一个类内部的实例方法 Invoke。这个实例方法需要传入一个 Test 类型的变量。在方法体中使用 is 运算符对变量 t 进行判断,判断它是否能够转换成目标接口类型。如果能够转换,则说明类 Test 实现了那个接口。然后执行 if 语句代码块中的代码,将变量转换成接口类型,接下来调用接口中的方法实现多态。这段代码的执行结果如下:

```
字段 a 的值是: 100
字段 b 的值是: 200
字段 c 的值是: 300
```

14.14 接口成员隐藏

当接口进行单一的继承时,派生程度大的成员将隐藏掉派生程度小的相同名称或相同签名成员。代码实例如下:

```
001     using System;
002     namespace InterfaceType2
003     {
```

```
004     interface ITest1
005     {
006         int Show
007         {
008             get;
009             set;
010         }
011     }
012     interface ITest2:ITest1
013     {
014         void Show(float a);
015     }
016     interface ITest3:ITest2
017     {
018         void Show(double b);
019     }
020     class Test : ITest3
021     {
022         private int a;
023         private float b;
024         private double c;
025         public Test(int a, float b, double c)
026         {
027             this.a = a;
028             this.b = b;
029             this.c = c;
030         }
031     }
032     class Program
033     {
034         static void Main(string[] args)
035         {
036             Test t = new Test(100, 200, 300);
037             Console.ReadKey();
038         }
039     }
040 }
```

这段代码中，接口 ITest1 中声明了一个名称为 Show 的属性。第 12 行的接口 ITest2 从 ITest1 继承，在它里面定义了一个方法 Show，它有一个 float 类型的参数。第 16 行代码定义了一个接口 ITest3，它从 ITest2 继承，在 ITest3 中定义了一个方法 Show，它有一个 double 类型的参数。对于属性和方法来说，它们的名称不能相同。但是对于方法来说，它们只要签名不同，就不构成隐藏。因此对于这段代码，编译器将会报告如下错误：

警告 1 “InterfaceType2. ITest2. Show(float)” 隐藏了继承的成员 “InterfaceType2. ITest1. Show”。如果是有意隐藏，请使用关键字 new。 H:\书稿 new\书稿源码\第十四章\InterfaceType2\Program.cs

17 14 InterfaceType2

对于事件和方法来说，方法的名称也不能和事件的名称相同，否则也将造成成员的隐藏。代码如下：

```

001 using System;
002 namespace InterfaceType2
003 {
004     public delegate void D();
005     interface ITest1
006     {
007         event D Show;
008     }
009     interface ITest2:ITest1
010     {
011         void Show(float a);
012     }
013     interface ITest3:ITest2
014     {
015         void Show(double b);
016     }
017     class Test : ITest3
018     {
019         private int a;
020         private float b;
021         private double c;
022         public Test(int a, float b, double c)
023         {
024             this.a = a;
025             this.b = b;
026             this.c = c;
027         }
028     }
029     class Program
030     {
031         static void Main(string[] args)
032         {
033             Test t = new Test(100, 200, 300);
034             Console.ReadKey();
035         }
036     }
037 }

```

这段代码的第 7 行定义了一个事件，事件的名称和它的继承接口中的方法成员名称相同。这时也会发生隐藏。但是 ITest2 中的方法和 ITest3 中的方法因为签名不同，不构成隐藏。编译器会报告如下信息：

警告 1 “InterfaceType2. ITest2. Show(float)” 隐藏了继承的成员 “InterfaceType2. ITest1. Show”。如果是有意隐藏，请使用关键字 new。 H:\书稿 new\书稿源码\第十四章\InterfaceType2\Program.cs

11 14 InterfaceType2

错误 2 “InterfaceType2.Test” 不实现接口成员 “InterfaceType2.ITest1.Show” H:\书稿 new\书稿源码\第十四章\InterfaceType2\Program.cs 17 11 InterfaceType2

错误 3 “InterfaceType2.Test” 不实现接口成员 “InterfaceType2.ITest2.Show(float)” H:\书稿 new\书稿源码\第十四章\InterfaceType2\Program.cs 17 11 InterfaceType2

错误 4 “InterfaceType2.Test” 不实现接口成员 “InterfaceType2.ITest3.Show(double)” H:\书稿 new\书稿源码\第十四章\InterfaceType2\Program.cs 17 11 InterfaceType2

在编译器中的三个错误是因为，虽然发生了成员隐藏，但是对于每个接口的成员都需要实现。如果每个接口中的成员都是方法成员，如果它们的名称相同，但是只要它们的签名不同，就不构成隐藏。实例代码如下：

```

001 using System;
002 namespace InterfaceType2
003 {
004     interface ITest1
005     {
006         void Show(int a);
007     }
008     interface ITest2:ITest1
009     {
010         void Show(float b);
011     }
012     interface ITest3:ITest2
013     {
014         void Show(double c);
015     }
016     class Test : ITest3
017     {
018         private int a;
019         private float b;
020         private double c;
021         public Test(int a, float b, double c)
022         {
023             this.a = a;
024             this.b = b;
025             this.c = c;
026         }
027         public void Show(int a)
028         {
029             this.a = a;
030             Console.WriteLine("字段 a 的值是: {0}", this.a);
031         }
032         public void Show(float b)
033         {
034             this.b = b;
035             Console.WriteLine("字段 a 的值是: {0}", this.b);

```

```

036         }
037         public void Show(double c)
038         {
039             this.c = c;
040             Console.WriteLine("字段 a 的值是: {0}", this.c);
041         }
042     }
043     class Program
044     {
045         static void Main(string[] args)
046         {
047             Test t = new Test(100, 200, 300);
048             t.Show(100);
049             t.Show(200.01F);
050             t.Show(300.05);
051             Console.ReadKey();
052         }
053     }
054 }

```

在这段代码中的三个接口中声明了三个签名不同的同名方法,但是它们因为签名不同不构成成员隐藏。可以正常在实现类中进行方法的实现。第 48 行到第 50 行使用不同类型的参数传递进去实现了方法的重载。代码的执行结果如下:

```

字段 a 的值是: 100
字段 a 的值是: 200.01
字段 a 的值是: 300.05

```

但是对于类从多个接口进行多重继承来说,则不存在方法的隐藏。这时叫做成员的多义性。即使不同的接口中定义了名称相同的属性和方法,编译器也不会提示成员发生了隐藏。对于成员的多义性,也需要接口成员的显式实现来解决。

14.15 接口的重新实现

一个接口可以重复的加入基类和派生类的继承列表。这时,基类和派生类可以重复实现接口。实例代码如下:

```

001 using System;
002 namespace InterfaceType2
003 {
004     interface A
005     {
006         void Show();
007     }
008     class Test:A
009     {
010         private int a;
011         public Test(int a)
012         {
013             this.a = a;

```

```

014     }
015     public void Show()
016     {
017         Console.WriteLine("基类中的方法被调用, 字段 a 的值是: {0}", this.a);
018     }
019 }
020 class Test1 : Test, A
021 {
022     private int b;
023     public Test1(int a, int b)
024         : base(a)
025     {
026         this.b = b;
027     }
028     void A.Show()
029     {
030         Console.WriteLine("派生类的方法被调用, 字段 b 的值是: {0}", this.b);
031     }
032 }
033 class Program
034 {
035     static void Main(string[] args)
036     {
037         Test t = new Test1(100, 200);
038         A myA = t;
039         myA.Show();
040         Console.ReadKey();
041     }
042 }
043 }

```

在这段代码中, 接口 A 被类 Test 和 Test 的派生类 Test1 同时进行了实现。如果在类 Test1 中不使用显式实现的方式, 则 Test1 中的方法 Show 将隐藏掉基类中的方法 Show。编译器就会有一个警告信息。当把派生类中的方法以显式实现的方式定义后, 编译器就不会有警告信息。而且多态调用也能得到实现。当在第 38 行用 A 类型的变量 myA 指向 test1 的实例的时候, 使用 myA 调用 Show 方法将实现多态。代码的执行结果如下:

```
派生类的方法被调用, 字段 b 的值是: 200
```

14.16 接口与抽象类

抽象类和接口一样, 它们都不能实例化。但是抽象类可以包含数据成员和方法的实现。当抽象类实现接口时, 它可以实现接口中的成员, 也可以不实现接口中的成员, 而把接口中的成员重新映射为抽象的。实例代码如下:

```

001     using System;
002     namespace InterfaceType2
003     {
004         interface A

```

```
005      {
006          void Show1();
007          void Show2();
008      }
009      abstract class Test:A
010      {
011          public abstract void Show1();
012          public abstract void Show2();
013      }
014      class Test1 : Test
015      {
016          private int a;
017          private int b;
018          public Test1(int a, int b)
019          {
020              this.a = a;
021              this.b = b;
022          }
023          public override void Show1()
024          {
025              Console.WriteLine("派生类 Test1 的方法被调用, 字段 a 的值是: {0}", this.a);
026          }
027          public override void Show2()
028          {
029              Console.WriteLine("派生类 Test1 的方法被调用, 字段 b 的值是: {0}", this.b);
030          }
031      }
032      class Program
033      {
034          static void Main(string[] args)
035          {
036              Test t = new Test1(100,200);
037              t.Show1();
038              t.Show2();
039              Console.ReadKey();
040          }
041      }
042  }
```

代码的第 4 行定义了一个接口 A，在接口中声明了两个接口的成员方法。抽象类 Test 实现接口 A，在抽象类中，对于接口中的方法，可以将它们重新映射为抽象方法而不予实现。这样它就把接口中的这两个成员方法交给抽象类的派生类去实现。类 Test1 从类 Test 中派生，它重写了抽象类中的两个抽象方法。第 36 行定义了抽象类的引用变量指向派生类的实例，通过调用两个抽象方法实现了多态。这段代码的执行结果如下：

派生类 Test1 的方法被调用，字段 a 的值是：100

派生类 Test1 的方法被调用，字段 b 的值是：200

抽象类实现接口的时候，可以在抽象类中实现接口中的方法。也可以使用显式实现接口成员的方式实现接口的方法成员。实例代码如下：

```
001     using System;
002     namespace InterfaceType2
003     {
004         interface A
005         {
006             void Show1();
007             void Show2();
008             void Show3();
009             void Show4();
010         }
011         abstract class Test:A
012         {
013             public abstract void Show1();
014             public abstract void Show2();
015             public void Show3()
016             {
017                 Console.WriteLine("抽象类中实现的接口方法 Show3 被调用");
018             }
019             void A.Show4()
020             {
021                 Invoke();
022             }
023             public abstract void Invoke();
024         }
025         class Test1 : Test
026         {
027             private int a;
028             private int b;
029             public Test1(int a, int b)
030             {
031                 this.a = a;
032                 this.b = b;
033             }
034             public override void Show1()
035             {
036                 Console.WriteLine("派生类 Test1 的方法被调用，字段 a 的值是：{0}", this.a);
037             }
038             public override void Show2()
039             {
040                 Console.WriteLine("派生类 Test1 的方法被调用，字段 b 的值是：{0}", this.b);
041             }
042         }
043     }
```

```
042         public override void Invoke()
043     {
044         Console.WriteLine("派生类重写的抽象类中的 Invoke 方法被调用");
045     }
046 }
047 class Program
048 {
049     static void Main(string[] args)
050     {
051         A myA = new Test1(100,200);
052         myA.Show1();
053         myA.Show2();
054         myA.Show3();
055         myA.Show4();
056         Console.ReadKey();
057     }
058 }
059 }
```

接口 A 中定义了 4 个接口成员方法，抽象类 Test 实现了 A。对于接口 A 中的方法，抽象类 Test 采用了三种方式进行了实现。Show1 和 Show2 方法被重新映射成了抽象方法，在抽象类的派生类 Test1 中进行了实现。Show3 方法在抽象类中直接进行了实现。Show4 方法在第 19 行代码中进行了显式实现。对于显式实现的接口中的方法来说，它的访问权限不能是 public 的，不能在类外部直接调用。只能通过多态的方式调用。但是，抽象类是不允许实例化的，不能实例化抽象类来实现多态调用。因此第 21 行，它调用了另外一个方法 Invoke。而 Invoke 方法是在抽象类中定义的一个抽象方法，它在抽象类的派生类 Test1 中进行了重写。

在独立的类 Program 中的 Main 方法中，第 51 行定义了一个接口 A 的引用变量指向了派生类 Test1 的实例。然后依次由接口 A 的引用变量 myA 调用它的 4 个成员方法。当调用 Show1 和 Show2 方法的时候，按照派生程度最大的原则，派生类 Test1 中的重写方法被调用。在调用 Show3 的时候，抽象类中的实现方法被调用。需要注意的是 Show4 的调用，它按照派生程度最大的原则，因为 myA 的编译时类型是 A，myA 的运行时类型是 Test1。因此依照派生程度最大的原则，首先第 19 行的显式实现的方法被调用，它又调用了抽象方法 Invoke。接着按照继承链往下走，Invoke 方法又被派生类 Test1 重写。因此在派生程度最大的派生链的终点就是派生类中重写的 Invoke 方法，它最终被调用。所以，在类的继承过程中，确定哪个方法最终被调用，就是不管方法调用有多复杂，只要确定好起点（编译时类型）和终点（运行时类型），然后在这条路径中找派生程度最大的方法就可以了。这段代码的执行结果如下：

```
派生类 Test1 的方法被调用，字段 a 的值是：100
派生类 Test1 的方法被调用，字段 b 的值是：200
抽象类中实现的接口方法 Show3 被调用
派生类重写的抽象类中的 Invoke 方法被调用
```

第十五章 内部类和成员访问

在已经介绍过的类型中，都是在编译单元或命名空间内部定义的类型。这样的类型叫做非嵌套类型。而在类或结构内部定义的类型，叫做嵌套类型或内部类。

15.1 嵌套类型的定义方法

下面定义一个简单的嵌套类型来介绍一下嵌套类型的定义方法，实例代码如下：

```
001 using System;
002 namespace NestedType1
003 {
004     class A
005     {
006         private B myB;
007         public A()
008         {
009             myB = new B(100);
010         }
011         class B
012         {
013             private int a;
014             public B(int a)
015             {
016                 this.a = a;
017             }
018             public void Show1()
019             {
020                 Console.WriteLine("嵌套类中字段 a 的值是: {0}", this.a);
021             }
022         }
023         public void Show2()
024         {
025             myB.Show1();
026         }
027     }
028     class Program
029     {
030         static void Main(string[] args)
031         {
032             A myA = new A();
033             myA.Show2();
034             Console.ReadKey();
035         }
036     }
037 }
```

本段代码的第 4 行定义了一个类 A，它定义在命名空间 NestedType1 中。因此，它是一个非嵌套类。

在 A 内部，第 11 行代码定义了一个类 B。它位于类 A 中，它就是一个嵌套类，通常也叫做内部类。内部类和它的外部类的成员位于同等的位置。因此，嵌套类是有访问权限修饰符的。本例因为是简单介绍嵌套类的定义，所以没有加访问权限修饰符。没有任何访问权限修饰符，和类的其它成员一样，它也是默认 `private` 的。代码的第 6 行定义了外部类 A 的一个字段 `myB`，它是内部类 B 的一个引用变量。第 7 行的构造方法中对它进行了实例化，调用了内部类的构造方法。在内部类中，定义成员的方法和普通类的定义方法都是一样的。它包含字段、构造方法和普通实例方法等。在第 23 行的实例方法中，通过内部类的实例字段调用了内部类中的实例方法，打印输出内部类实例的字段值。

在 Main 方法中，首先定义了一个外部类的实例，然后调用外部类实例的方法，这个方法又调用了内部类实例的方法，从而输出了内部类实例的字段值。这段代码的执行结果如下：

嵌套类中字段 a 的值是：100

15.2 嵌套类型的完全限定名称

嵌套类型具有完全限定名称，它的完全限定名称从包含它的外部类型所在的命名空间开始。下面看代码实例：

```
001 using System;
002 namespace NestedType1
003 {
004     class A
005     {
006         private B myB;
007         public A()
008         {
009             myB = new B(100);
010         }
011         public class B
012         {
013             private int a;
014             public B(int a)
015             {
016                 this.a = a;
017             }
018             public void Show1()
019             {
020                 Console.WriteLine("嵌套类中字段 a 的值是：{0}", this.a);
021             }
022         }
023         public void Show2()
024         {
025             myB.Show1();
026         }
027     }
028     class Program
029     {
030         static void Main(string[] args)
031         {
```



```
032         NestedType1.A.B myB = new NestedType1.A.B(30);
033         myB.Show1();
034         Console.ReadKey();
035     }
036 }
037 }
```

这段代码将类 A 中的内部类 B 定义为 public 可访问性，这样在外部类中可以直接引用这个内部类。在第 32 行引用的就是内部类 B 的完全限定名称。内部类 B 的完全限定名称就是从包含它的外部类所在的命名空间开始依次进行引用。因为内部类的访问权限设置为 public，所以可以使用内部类的变量调用内部类定义的实例方法 Show1。这段代码的执行结果如下：

嵌套类中字段 a 的值是：30

不只是类可以包含内部类，结构也可以包含内部类。实例代码如下：

```
001 using System;
002 namespace NestedType1
003 {
004     struct A
005     {
006         private B myB;
007         private C myC;
008         private int i;
009         public A(int i)
010         {
011             myB = new B(100);
012             myC = new C(200);
013             this.i = i;
014         }
015         public class B
016         {
017             private int a;
018             public B(int a)
019             {
020                 this.a = a;
021             }
022             public void Show1()
023             {
024                 Console.WriteLine("嵌套类中字段 a 的值是：{0}", this.a);
025             }
026         }
027         public struct C
028         {
029             private int b;
030             public C(int b)
031             {
032                 this.b = b;
```

```

033         }
034         public void Show2()
035         {
036             Console.WriteLine("嵌套结构中字段 b 的值是: {0}", this.b);
037         }
038     }
039     public void Show()
040     {
041         myB.Show1();
042         myC.Show2();
043     }
044 }
045 class Program
046 {
047     static void Main(string[] args)
048     {
049         NestedType1.A.B myB = new NestedType1.A.B(100);
050         myB.Show1();
051         A.C myC = new A.C(200);
052         myC.Show2();
053         A myA = new A(20);
054         myA.Show();
055         Console.ReadKey();
056     }
057 }
058 }

```

这段代码在第 4 行定义了一个结构 A，和类相同，结构中也可以包含内部类与内部结构。第 15 行和第 27 行各定义了一个内部类和内部结构。在结构 A 中包含了内部类 B 和内部结构 C 的字段变量，第 9 行开始的构造方法中实例化了内部类和内部结构的字段。内部结构的用法和内部类的用法是一样的。在第 49 行使用完全限定名称的方式定义了一个内部类的实例，内部结构也可以使用这样的完全限定名称来定义内部结构的实例。因为同在一个命名空间内，所以第 51 行使用结构的完全限定名称来实例化内部结构的实例时，没有使用命名空间名称。因为内部类和内部结构都定义为 public 访问权限的，所以使用完全限定名称的方式在类外定义内部类与内部结构的变量时，可以直接调用它们的实例方法。也可以定义一个外部结构的变量，通过它调用外部结构的实例方法的方式，间接调用各个内部类与内部结构的实例方法打印输出内部类与内部结构的字段值。这段代码的执行结果如下：

```

嵌套类中字段 a 的值是: 100
嵌套结构中字段 b 的值是: 200
嵌套类中字段 a 的值是: 100
嵌套结构中字段 b 的值是: 200

```

15.3 声明空间

从前面这个内部类与内部结构的例子可以看到，内部类也可以使用访问权限修饰符。内部类完全作为一种类的成员存在。因为存在嵌套，它的访问权限就比较复杂。在介绍嵌套类的可访问性之前，先介绍一下声明空间的概念。C#语言中的各种元素需要先声明才能使用，最大的一层声明空间就是命名空间。各种类都是使用命名空间进行组织和管理。各种元素的组织结构如下所示：

命名空间包含:

- 类型声明
 - a. 类
 - 1. 字段
 - 2. 方法
 - 3. 属性
 - 4. 常量
 - 5. 事件
 - 6. 索引器
 - 7. 嵌套类
 - b. 接口
 - 1. 抽象方法
 - 2. 抽象属性
 - 3. 抽象索引器
 - c. 结构
 - (同类)
 - d. 枚举
 - e. 委托
- 嵌套的命名空间

声明空间就是各种元素声明的区域,有若干种声明空间,在每种声明空间中,各种元素可以定义自己的名称,但是除了方法重载外,在同一个声明空间中,成员的名称是唯一的,即使是不同种类的成员也不允许定义相同的名称。例如,在一个声明空间中,定义相同名称的字段和方法将会引发编译时错误。实例代码如下:

```

001 using System;
002 namespace DeclarationSpace
003 {
004     class Program
005     {
006         private int a;
007         public void a()
008         {
009         }
010         static void Main(string[] args)
011         {
012         }
013     }
014 }
```

在这段代码中,Program 类中定义了同名的字段和方法,在编写代码的过程中不会报告语法错误,但是在编译的时候,就会提示错误。错误信息如下:

```

错误 1 类型“DeclarationSpace.Program”已经包含“a”的定义E:\书稿 new\书稿源码\第十五章
\ConsoleApplication1\DeclarationSpace.cs 7 21 ConsoleApplication1
```

声明空间包含如下几种类型:

1. 在C#程序的所有源文件中,存在一个全局声明空间,这个声明空间没有名称,它存在于所有源文件中,并且它是组合起来的,也就是说,所有源文件的全局声明空间是一个。

2. 如果声明一个命名空间，如果这个命名空间不是嵌套命名空间，也就是说，它没有声明在任何已经声明的命名空间下，则它属于全局声明空间。

3. 在 C# 程序的所有源文件中，如果一个命名空间的声明和另一个命名空间的声明名称具有完全相同的完全限定名称，则这两个命名空间属于一个命名空间的组合部分。即使这个新定义的命名空间和 .Net 类库的命名空间名称相同，也属于同一个命名空间的组合部分。实例代码如下：

```
001 using System;
002 namespace System
003 {
004     class Program
005     {
006         public void a()
007         {
008         }
009         static void Main(string[] args)
010         {
011         }
012     }
013 }
014 namespace Test
015 {
016     class Program
017     {
018     }
019 }
```

在这段代码中的第 2 行首先定义了一个命名空间，它的名称和类库中的 System 命名空间相同，第 14 行又定义了一个命名空间，在它里面定义一个类，如果在这个类中使用 System 命名空间中的成员，编译器的智能提示中就可以看到刚刚定义的 Program 类。

由此可以知道，自定义的 System 命名空间和类库中的 System 命名空间组合成了一个命名空间。

4. 类、接口、结构的声明创建一个新的声明空间，在这个新创建的声明空间中，除了实例构造方法以及实例构造方法的重载、析构方法和静态构造方法外，其余成员的声明不能和类或结构的名称同名。在这个声明空间中，可以同名的只有重载构造方法、重载成员方法、重载索引器以及运算符重载。可以同名的原因是它们的方法签名是不同的。

5. 在类的继承中，基类的声明空间和派生类的声明空间不同，因此允许重名，重名有两种方式，一种是多态的实现，另一种是派生类的同名成员隐藏了基类继承来的成员。

6. 委托的声明创建一个新的声明空间，它的形参以及泛型委托的类型参数不能重名。

7. 枚举的声明创建一个新的声明空间，枚举的成员不能重名。

8. 方法、索引器、运算符重载、实例构造方法以及匿名方法的声明都会创建一个新的声明空间，这个声明空间称作局部变量声明空间。这个声明空间包括形参和泛型方法的类型参数的声明，而方法体的声明空间将嵌套在这个局部变量声明空间中。因此，此局部变量声明空间和嵌套的局部变量声明空间中的成员声明如有重名，将发生编译时错误。下面是代码实例：

```
001 using System;
002 namespace System
003 {
004     class Program
```

```

005     {
006         public static int Method1(int a,int b)
007     {
008         const int a = 3;
009         int b;
010         return a;
011     }
012     static void Main(string[] args)
013     {
014     }
015
016     }
017 }

```

在这段代码中，第 7 行开始的嵌套的局部变量声明空间内的成员名称和第 6 行代码中新创建的局部变量声明空间内的形参名称相同，编译器会报错，错误信息如下：

```

错误 1    类型“DeclarationSpace.Program”已经包含“a”的定义E:\书稿 new\书稿源码\第十五章
\ConsoleApplication1\DeclarationSpace.cs 7    21 ConsoleApplication1
错误 2    不能在此范围内声明名为“a”的局部变量，因为这样会使“a”具有不同的含义，而它已在“父
级或当前”范围中表示其他内容了 E:\书稿 new\书稿源码\第十五章
\ConsoleApplication1\DeclarationSpace.cs 8    23 ConsoleApplication1
错误 3    不能在此范围内声明名为“b”的局部变量，因为这样会使“b”具有不同的含义，而它已在“父
级或当前”范围中表示其他内容了 E:\书稿 new\书稿源码\第十五章
\ConsoleApplication1\DeclarationSpace.cs 9    17 ConsoleApplication1

```

9. 每个代码块、switch 语句、for 语句、foreach 语句以及 using 语句都会创建一个新的局部变量声明空间，它们的代码块创建的局部变量声明空间将嵌套在父级的局部变量声明空间中，因此在这两个声明空间内不能有重名的成员。代码实例如下：

```

001 using System;
002 namespace System
003 {
004     class Program
005     {
006         static void Main(string[] args)
007         {
008             for (int i = 0; i < 12; i++)
009             {
010                 int i = 1;//此处重名
011             }
012             int a = 11;
013             switch (a)
014             {
015                 case 11:
016                 {
017                     int a = 12;//此处重名
018                     break;

```

```

019         }
020     }
021     {
022         int b = 13;
023     {
024         int b = 14;//此处重名
025     }
026     }
027 }
028 }
029 }

```

在这段代码中第 8 行代码中的 for 语句创建的两个声明空间有重名现象、第 13 行代码中 switch 语句创建的局部变量声明空间和父一级的块有重名现象，两个嵌套的块中的局部变量声明空间有重名现象。在编译时，编译器会报告错误。错误信息如下：

错误 1 不能在此范围内声明名为“i”的局部变量，因为这样会使“i”具有不同的含义，而它已在“父级或当前”范围中表示其他内容了 E:\书稿 new\书稿源码\第十五章\ConsoleApplication1\DeclarationSpace.cs 10 21 ConsoleApplication1

错误 2 不能在此范围内声明名为“a”的局部变量，因为这样会使“a”具有不同的含义，而它已在“父级或当前”范围中表示其他内容了 E:\书稿 new\书稿源码\第十五章\ConsoleApplication1\DeclarationSpace.cs 17 29 ConsoleApplication1

错误 3 不能在此范围内声明名为“b”的局部变量，因为这样会使“b”具有不同的含义，而它已在“父级或当前”范围中表示其他内容了 E:\书稿 new\书稿源码\第十五章\ConsoleApplication1\DeclarationSpace.cs 24 25 ConsoleApplication1

10. 每个代码块或 switch 语句块都为标签的使用创建一个新的声明空间。在这个声明空间中还可以包含嵌套块，嵌套块和父一级块中不能包含相同名称的标签声明。下面是代码实例：

```

001 using System;
002 namespace System
003 {
004     class Program
005     {
006         static void Main(string[] args)
007         {
008             goto label1;
009             {
010                 goto label1;
011                 label1:
012                 {
013                     Console.WriteLine();
014                     goto label1;
015                     label1:
016                     Console.WriteLine();
017                 }
018             }
019         }

```

```

020
021     }
022 }

```

这段代码的错误提示如下：

```

错误 1    goto 语句范围内没有“label1”这样的标签    E:\书稿 new\书稿源码\第十五章
\ConsoleApplication1\DeclarationSpace.cs 8    13 ConsoleApplication1
警告 2    检测到无法访问的代码    E:\书稿 new\书稿源码\第十五章
\ConsoleApplication1\DeclarationSpace.cs 10 17 ConsoleApplication1
错误 3    在包含的范围中标签“label1”遮盖了具有同样名称的另一个标签    E:\书稿new\书稿源码\第
十五章\ConsoleApplication1\DeclarationSpace.cs 15 17 ConsoleApplication1

```

这段代码编译器会给出两个错误和一个警告，产生的原因是，goto 语句用来引用标签，而第一个 goto 语句位于最高一级的块声明空间中，它无法看到子一级的块声明空间中的内容，因此提示第一个错误。第二个警告信息产生的原因是因为最后一个标签的声明空间是嵌套局部变量声明空间，它和父一级的声明空间中的标签重名导致错误，最后一个 goto 语句不知道要引用谁，从而导致检测到无法访问的代码。

15.4 成员类型

在 C# 中，命名空间和类具有成员。命名空间是用来组织各个类的，它是一种逻辑结构。类是一种数据结构，它是最基本的 C# 类型，类外不能有成员。这和 C++ 语言不同，C# 是完全面向对象的语言。类封装了字段和方法，类可以用来创建实例，实例也称为对象。类还可以实现继承和多态，派生类扩展和专用化了基类。命名空间是最大的一层组织结构，它包含类。要想访问它们的成员，必须使用“.”标记。类的成员从继承性来说有两种，一种是创建类的时候声明的，一种是从基类继承来的。当类发生继承的时候，派生类将获得除了基类的构造方法、析构方法和静态构造方法之外的所有成员，而不管该成员在基类中的访问权限是什么。访问权限只决定了派生类是否可以访问该成员，而不决定是否继承。

15.4.1 命名空间的成员

一个命名空间和类在声明的时候，无论怎样都会声明在一个命名空间中，这个命名空间就是全局命名空间，它也是一个声明空间。如果声明的命名空间和类位于一个已经声明好的命名空间中时，则称这个命名空间和类是该命名空间的成员。命名空间不像类一样具有访问权限，不能用访问权限关键字修饰命名空间，命名空间是开放式可访问的。下面是代码实例：

```

001 using System;
002 namespace MemberOfNamespace
003 {
004     class Program
005     {
006         static void Main(string[] args)
007         {
008             Program1.Print();
009             MemberOfNamespace.Program1.Print();
010         }
011     }
012 }
013 namespace MemberOfNamespace
014 {
015     class Program1
016     {
017         public static void Print()

```

```

018      {
019          Console.WriteLine("Hello World!");
020      }
021  }
022  }

```

在这段代码中，定义了两个同名的命名空间。这两个命名空间现在是在一个源文件中，即使它们不在同一个源文件中，它们也是一个命名空间，它们都是一个命名空间的组成部分。因此在一个命名空间的类中访问另一个类的静态方法不需要使用命名空间名称。这段代码的第二种调用使用了完全限定名称的方式访问了同一个静态方法。完全限定名的方式体现了命名空间的成员的访问方式，在底层，编译器生成的中间语言都是用这种完全限定名称的方式访问命名空间的成员。

15.4.2 类的成员

类的成员从是否可创建实例来划分，可以分为两类，一种是静态成员，它属于类；另外一种实例成员，它属于实例或者叫做对象，类的实例使用 `new` 运算符创建，它将在堆中分配内存，并由垃圾回收器进行回收管理。

类的成员如果从继承性来划分，可以分为在本类中创建的成员和从基类中继承来的成员，在这里，`object` 类除外，因为在 C# 中，`object` 类是所有类的基类，因此它不再存在基类。`object` 如果第一个字母小写，则它是一种简写类型名称，它对应的是 `System.Object` 类。和它相似的基本类型还有 `string` 类型，它对应的是 `System.String` 类型。

在本类中可以声明的成员包括：常量（隐式静态的，属于类）、字段（静态字段和实例字段）、方法（静态方法和实例方法以及运算符重载）、属性（静态属性和实例属性）、索引器（静态索引器和实例索引器）、事件、嵌套类、构造方法（静态构造方法和实例构造方法）、析构方法。

在基类中继承来的成员和本类声明的成员不同的就是基类的构造方法（静态构造方法和实例构造方法）、析构方法不能被继承。

如果一个类在声明的时候没有指定基类，则它隐式继承自 `System.Object` 类，它将自然拥有 `object` 类的成员。

15.4.3 结构的成员

结构和类很相似，它也是一种数据结构。但是，结构不是只有一个 `object` 类作为基类，它派生自 `System.ValueType` 类。`object` 类成为了它的间接基类。

在 C# 中的每种基本数据类型都是一种结构，它们都继承自 `System.ValueType` 类。它们的简化名称和类库中的完全限定结构名的对应关系如下：`sbyte` 对应于 `System.SByte`，`byte` 对应于 `System.Byte`，`short` 对应于 `System.Int16`，`ushort` 对应于 `System.UInt16`，`int` 对应于 `System.Int32`，`uint` 对应于 `System.UInt32`，`long` 对应于 `System.Int64`，`ulong` 对应于 `System.UInt64`，`char` 对应于 `System.Char`，`float` 对应于 `System.Single`，`double` 对应于 `System.Double`，`decimal` 对应于 `System.Decimal`，`bool` 对应于 `System.Boolean`。它们都是结构。

15.4.4 接口的成员

接口的成员包括在本接口中声明的和继承自所有基接口的成员。因为接口中的成员只提供声明，不提供实现，但是这和所有类型都继承自 `object` 类有所冲突。微软官方的解释是，`object` 的成员严格来讲不是任何接口的成员，但是通过开发环境的智能提示是可以查找到 `object` 的成员的。下面是代码实例：

```

001  using System;
002  namespace MemberOfInterface
003  {
004      interface A
005      {
006          void Print();

```



```
007     }
008     class Program:A
009     {
010         static void Main(string[] args)
011         {
012             A myInterface = new Program();
013             Console.WriteLine(myInterface.ToString());
014             Console.ReadKey();
015         }
016         public void Print()
017         {
018             Console.WriteLine("Hello World!");
019         }
020         public override string ToString()
021         {
022             string s = "调用了类中重写的 ToString 方法";
023             return s;
024         }
025     }
026 }
```

在这段代码中，首先定义了一个接口，然后定义一个类实现了这个接口，并重写了 ToString 方法。在 Main 方法中声明一个接口类型的变量引用了类的实例，然后通过接口类型的引用调用 ToString 方法。这段代码的执行结果如下：

调用了类中重写的 ToString 方法

通过接口类型的引用实现了多态的调用，这就说明，其实接口类型还是继承了 object 类的方法，否则，它不能够实现多态的方法调用。

15.4.5 枚举的成员

枚举是一种值类型，它继承自 System.Enum，这个类是一个抽象类，但是它很特殊，不能创建一个类从这个类继承。如果声明一个类从这个类继承，就会出现编译错误。枚举的间接基类是 System.ValueType 和 System.Object 类。因此它会继承它们的所有成员。

15.4.6 数组的成员

数组的成员是从 System.Array 类继承的成员。System.Array 类是一个抽象类，但是和 Enum 类一样，不能定义一个类从这个类派生，只有系统和编译器可以从 System.Array 类显式派生。如果定义一个数组，它会自动拥有 System.Array 类的成员。

15.4.7 委托的成员

委托的成员是从 System.Delegate 类继承的成员。虽然创建的委托类是从 System.Delegate 类继承，但是 System.Delegate 类不是一个委托类。这个类用来派生委托类型，并且只有系统和编译器可以从这个类显式派生。System.Delegate 类有一个派生类 System.MulticastDelegate，它代表多播委托类。同样的，只有系统和编译器可以从这个类显式派生。定义委托类应使用 C#语言提供的 delegate 关键字。当使用这个关键字定义委托时，是从 System.MulticastDelegate 类派生。

15.5 成员访问性

在声明成员的时候，可以使用关键字确定成员的可访问性。在确定成员的可访问性的时候，由包含它的那个类型的可访问性和该成员声明的可访问性两部分结合起来确定其可访问性。每个成员都有其可访问域，只有当代码位于该成员的可访问域中时，才可以访问该成员。

15.5.1 可访问性关键字

当定义一个成员时，它的可访问性关键字包括以下几种：

- `public` 用此关键字定义成员，成员的访问不受限制
- `protected` 用此关键字定义成员，此类中或从此类派生的类可以访问。
- `internal` 用此关键字定义成员，本程序集可以访问。
- `protected internal` 用此关键字定义成员，本程序集或从此类派生的类可以访问。
- `private` 用此关键字定义成员，只有此类中可以访问。

在这几种关键字中特别说明一点的是 `protected internal` 关键字，它是本程序集或从此类派生的类可以访问，并不是并且的关系。例如，如果新建一个程序集，然后定义一个类，那么这个类因为和此成员不在一个程序集内，所以不可以访问它。但是如果定义的这个类从此成员派生，那么它就可以访问这个成员。如果在同一个程序集内定义一个类，虽然这个类不从此类派生，但是它也可以访问 `protected internal` 关键字修饰的成员，因为它们位于一个程序集内。

当定义一个成员的可访问性的时候，可以使用的可访问性关键字受限于该成员所在位置的上下文。看下面的实例代码：

```
001 using System;
002 namespace AccessOfNamespace
003 {
004     internal class Program
005     {
006         /// <summary>
007         /// Main 方法不能被其他程序集调用
008         /// </summary>
009         /// <param name="args"></param>
010         public static void Main(string[] args)
011         {
012         }
013     }
014 }
```

在这个例子中，虽然 `Main` 方法是 `public` 的，但是因为包含它的类是 `internal` 的，所以不能从其他程序集访问这个 `Main` 方法。这就是可访问性关键字受限于该成员所在位置的上下文的例子。当定义一个成员的时候，不指定可访问性关键字，那么该成员定义处的上下文会为该成员指定一个默认的可访问性关键字。例如在类中定义成员，如果不指定可访问性关键字，那么默认都是 `private` 的。

15.5.2 成员访问性指定

命名空间的可访问性不需要指定，它默认是 `public` 的。如果指定了命名空间的 `public` 关键字，编译器会报错。定义类的时候，可以使用的可访问性关键字包括 `public` 和 `internal`，也就是说，类的可访问性只能是公开的或本程序集可访问的。当不指定类的可访问性关键字的时候，上下文会自动给它指定一个 `internal` 的可访问性。

类的成员可以具有上一小节介绍的 5 种可访问性之一，如果不指定任何可访问性关键字，则上下文会自动为类的成员赋予 `private` 可访问性。

结构因为定义在命名空间中，所以结构可以使用的可访问性关键字包括 `public` 和 `internal`，当不指定类的可访问性关键字的时候，上下文会自动给它指定一个 `internal` 的可访问性。

结构因为是隐式密封的，所以它不能被继承，结构中声明的成员可以使用的可访问性关键字包括 `public`、`internal` 或 `private`。但是，结构从它的基类继承的成员则不受此限制。因为不能从结构继承，因此，结构的成员不能具有 `protected` 和 `protected internal` 可访问性。

接口的成员隐式具有 public 可访问性，但是不能用 public 关键字修饰。其他可访问性修饰关键字当然更不行。在实现接口的成员时，必须使用 public 关键字。实例代码如下：

```
001 using System;
002 namespace AccessOfNamespace
003 {
004     interface IMyInteface
005     {
006         void Test();
007     }
008     class Program:IMyInteface
009     {
010         public void Test()
011         {
012             Console.WriteLine("实现接口中的抽象方法");
013         }
014         static void Main(string[] args)
015         {
016         }
017     }
018 }
```

在这个例子中，如果第 10 行代码中实现的方法前面不用 public 关键字，而是使用 protected 或其他的关键字，则编译器会报错。

枚举成员隐式具有 public 可访问性，但是不能用 public 关键字进行修饰。实例代码如下，首先新建一个类库：

```
001 using System;
002 namespace Netc
003 {
004     public enum Color1 { white,red,green,blue };
005     public class A
006     {
007         enum Color2 { black, orange, yellow }
008     }
009 }
```

在这个类库中的命名空间下定义了一个 public 的枚举类型 Color1，Color1 因为直接定义在命名空间下面，所以它的可访问性只能是 public 或 internal 的，它的成员，例如 white、red 等的可访问性隐式是 public 的。但是它受上下文的影响，如果枚举类型本身是 internal 的，那么它们在其他的程序集中也是不可访问的。首先将第 4 行的 public 更改为 internal，然后再新建一个程序集引用这个类库，代码如下：

```
001 using System;
002 using Netc;
003 namespace TestEnum
004 {
005     class Program
006     {
007         static void Main(string[] args)
```

```

008      {
009          Netc.Color1 myColor = Netc.Color1.blue;
010          int temp = (int)A.Color2.black;
011      }
012  }
013 }

```

这时会提示 4 个错误，错误信息如下：

```

错误 1    “Netc.Color1” 不可访问，因为它受保护级别限制  E:\书稿 new\书稿源码\第十五章
\TestEnum\Program.cs    9    18    TestEnum
错误 2    “Netc.Color1” 不可访问，因为它受保护级别限制  E:\书稿 new\书稿源码\第十五章
\TestEnum\Program.cs    9    40    TestEnum
错误 3    “Netc.Color1” 并不包含“blue”的定义E:\书稿 new\书稿源码\第十五章
\TestEnum\Program.cs    9    47    TestEnum
错误 4    “Netc.A.Color2” 不可访问，因为它受保护级别限制    E:\书稿 new\书稿源码\第十五章
\TestEnum\Program.cs    10   31    TestEnum

```

枚举定义的位置可以在命名空间下，也可以在一个类的内部。在这个例子中，产生这 4 个错误的原因是，枚举 Color1 因为访问权限更改为 internal，所以在其他程序集中无法访问；Color2 因为定义在了一个类的内部，在没有可访问性修饰符的情况下，它默认是 private 的，因此不可访问。在这两种情况下，虽然枚举的成员都是 public 的，但是都收到了上下文的影响，公开访问的权限受到了限制。

15.5.3 嵌套类型的可访问域

由于成员的可访问性声明的存在，形成了该成员的可访问域。成员的可访问域由程序的代码文本节组成，并且文本节可以是不连续的。

C#中的预定义类型，例如 object、int、double 等类型的可访问域无限制。也就是说，在程序文本的任何地方都可以使用它们。

如果成员不是在某个类型内部声明的，就称这个成员是顶级的。顶级成员的声明有两种，未绑定的和绑定的。未绑定类型包括非泛型类型和未绑定泛型类型；绑定类型包括非泛型类型和构造类型。顶级类型可以使用的可访问性修饰符只有两个，public 和 internal。

- 顶级未绑定类型如果使用了 public 修饰，则它的可访问域是本程序集的程序文本以及引用并且使用它的其它任何程序集的程序文本。
- 顶级未绑定类型如果使用了 internal 修饰，则它的可访问域是本程序集的程序文本。

因为顶级类型涉及到了泛型类型，所以后面会介绍到。现在来看嵌套类型的可访问域。当嵌套类型位于类中的时候，它可以具有 public、protected、private、internal、protected internal 这 5 种可访问性。默认情况下，也就是不加上任何访问权限修饰符的情况下，嵌套类型是 private 的。现在看第一种情况：

1. 如果嵌套成员的可访问性为 public，那么它的可访问域就是包含它的类型的可访问域，这就是上下文约束。下面看代码实例：

```

001  using System;
002  namespace AccessibilityDomain2
003  {
004      internal class A
005      {
006          public class B
007          {
008          }
009          public int a;

```

```

010     }
011     class Program
012     {
013         static void Main(string[] args)
014         {
015             A.B myB = new A.B();
016             A myA = new A();
017             myA.a = 11;
018         }
019     }
020 }

```

在这段代码中，第 4 行中，可访问性为 `internal` 的类 `A` 中定义了一个内部类 `B`，类 `B` 的可访问性为 `public`，但是因为包含它的类 `A` 的可访问域为本程序集，所以即使类 `B` 的可访问性为 `public`，它也只能在本程序集中被引用。如果类 `A` 的可访问性是 `public`，则内部类的可访问域也是不受限制。

2. 如果嵌套成员的可访问性为 `private`，那么嵌套成员的可访问域就是包含它的类型的程序文本。下面看代码实例：

```

001 using System;
002 namespace AccessibilityDomain2
003 {
004     internal class A
005     {
006         private class B
007         {
008             public int b;
009         }
010         public int a;
011         private B myB = new B();
012         public void Get()
013         {
014             B myB1 = new B();
015             myB1.b = 11;
016             myB = myB1;
017             a = 33;
018             Console.WriteLine("a: {0}, myB.b: {1}", a, myB.b);
019         }
020     }
021     class Program
022     {
023         static void Main(string[] args)
024         {
025             //此处不能访问
026             //A.B myB = new A.B();
027             A myA = new A();
028             myA.a = 11;

```

```

029     }
030     }
031 }

```

在这段代码中，内部类 B 的可访问性设置为 `private`，因此它的可访问域只能是包含它的类 A 的程序文本，在类外，如果使用 A.B 的方式是无法访问的。这段代码中的字段 a 的可访问域是本程序集，内部类 B 的可访问域是 A 的程序文本，字段 myB 的可访问域是 A 的程序文本，还需要注意的是它的可访问性只能比内部类 B 的可访问性相同或比它低。方法 Get 的可访问域是本程序集，内部类 B 中的字段 b 因为可访问性是 `public` 的，因此它的可访问域是包含它的类的可访问域，也是类 A 的程序文本。

3. 如果嵌套成员的可访问性是 `Internal`，那么嵌套成员的可访问域就是包含嵌套成员的类的可访问域与本程序集之间的交集。下面是代码实例：

```

001 using System;
002 namespace AccessibilityDomain2
003 {
004     public class Test
005     {
006         internal class A
007         {
008             private int b;
009             public A(int b)
010             {
011                 this.b = b;
012             }
013         }
014     }
015     class Program
016     {
017         static void Main(string[] args)
018         {
019             Test.A myA = new Test.A(100); //在本程序集中可以访问
020         }
021     }
022 }

```

在代码的第 6 行定义的内部类的访问权限是 `internal` 的，它外部的类的访问权限是 `public` 的。因此内部类的可访问域就是公共的访问权限和本程序集的交集，结果还是本程序集。因此，在第 19 行可以以完全限定名称的方式引用内部类。如果外部类 Test 的访问权限是 `internal`，则内部类的访问权限取交集后还是本程序集。

4. 如果嵌套成员的可访问性是 `protected`，则先取外部类的程序文本和从外部类派生的类型的程序文本的并集。嵌套成员的可访问域是外部类的可访问域与前面所取的并集的交集。看下面的代码实例：

```

001 using System;
002 namespace AccessibilityDomain2
003 {
004     public class Test
005     {
006         protected class A

```

```

007         {
008             private int b;
009             public A(int b)
010             {
011                 this.b = b;
012             }
013         }
014     }
015     class Test1 : Test
016     {
017         A myA = new A(100); //在这里可以访问
018     }
019     class Program
020     {
021         static void Main(string[] args)
022         {
023             Test.A myA = new Test.A(100); //在这里不可以访问
024         }
025     }
026 }

```

在这段代码中，第 6 行的代码中内部类 A 的访问权限设置成了 protected。所以它的可访问域的确定方式是：先取外部类的程序文本和从外部类派生的类型的程序文本的并集。外部类的程序文本就是类 Test 的类体，类 Test 的派生类的程序文本就是第 15 行代码中的类 Test 的派生类 Test1 的类体。接下来取外部类的可访问域，类 Test 的可访问性是公开权限，所以它和前面的并集取交集就是类 Test 内部和它的派生类内部。所以，第 17 行可以引用内部类 A；而第 23 行代码中的引用是不允许的。

如果外部类的访问权限是 internal 的，则外部类的可访问域是本程序集。它与外部类和外部类的派生类的并集的交集就是本程序集中外部类内部和外部类的派生类的内部。

5. 如果内部类的可访问性是 protected internal，则它的可访问域是先取本程序集和从外部类派生的类的文本的并集，然后取外部类的可访问域与前面的并集的交集。实例代码如下：

```

001     using System;
002     namespace AccessibilityDomain2
003     {
004         public class Test
005         {
006             protected internal class A
007             {
008                 private int b;
009                 public A(int b)
010                 {
011                     this.b = b;
012                 }
013             }
014         }
015         class Test1 : Test

```

```

016      {
017          A myA = new A(100); //在这里可以访问
018      }
019      class Program
020      {
021          static void Main(string[] args)
022          {
023              Test.A myA = new Test.A(100); //在这里可以访问
024          }
025      }
026  }

```

在这段代码中，第6行代码的访问权限为 `protected internal`。它的可访问域是先取本程序集和外部类的派生类的程序文本。然后取外部类的可访问域（无访问限制）与前面并集的交集，结果就是本程序集和外部类的派生类可以访问。

如果外部类的可访问性是 `internal` 的，那么结果就是先取本程序集和外部类的派生类的程序文本。然后取外部类的可访问域（本程序集）与前面并集的交集，结果就是本程序集中可以访问（包括本程序集中的派生类）。

从上面的定义可以得出结论，嵌套类型的可访问域至少为包含它的外部类的程序文本。而成员的可访问域不会比定义该成员的类型可访问域更广。

以上几种情况是外部类为类类型时的可访问性规则。如果外部类是结构类型的话，因为结构类型不允许派生。所以内部类可以使用的可访问性权限只有三种，`public`、`private` 和 `internal`。

15.5.4 Protected 成员的访问方法

对于类中定义的 `protected` 成员的访问方法有下面几种：

1. 非限定名称方式，看下面的代码：

```

001      using System;
002      namespace AccessOfProtected
003      {
004          class A
005          {
006              protected int a;
007              public void Set()
008              {
009                  a = 11;
010              }
011          }
012          class B : A
013          {
014              public void Set1()
015              {
016                  a = 12;
017              }
018          }
019          class Program
020          {

```



```
021         static void Main(string[] args)
022         {
023         }
024     }
025 }
```

对于 protected 成员，在定义成员的类中和继承类中，可以使用非限定名称的方式来访问。例如第 9 行和第 16 行代码。

2. 以限定名称的方式访问，实例代码如下：

```
001     using System;
002     namespace AccessOfProtected
003     {
004         class A
005         {
006             protected int a;
007             public void Set(A myA)
008             {
009                 myA.a = 11;
010             }
011         }
012         class B : A
013         {
014             //此处不能访问
015             //public void Set1(A myA)
016             //{
017             //    myA.a = 12;
018             //}
019             public void Set2(B myB)
020             {
021                 myB.a = 13;
022             }
023         }
024         class Program
025         {
026             static void Main(string[] args)
027             {
028             }
029         }
030     }
```

在这段代码中，通过实例引用的方式访问了 protected 成员。需要注意的是 protected 成员可以在定义的程序文本中访问以及派生类访问。在派生类中使用基类的实例无法访问字段 a，因为在派生类中使用基类的实例引用字段 a，并不是从基类的程序文本中或派生类的实例访问。

3. 以 base. 成员的方式访问。实例代码如下：

```
001     using System;
002     namespace AccessOfProtected
```

```
003  {
004      class A
005      {
006          protected int a;
007      }
008      class B : A
009      {
010          public void Set2()
011          {
012              base.a = 13;
013          }
014      }
015      class Program
016      {
017          static void Main(string[] args)
018          {
019          }
020      }
021  }
```

15.5.5 可访问性约束

在 C# 中，对于成员的可访问性的设置并不是随便设置的，它要受上下文的影响。存在下述可访问性约束条件。

1. 类的直接基类的可访问性要高于或等同于这个类的可访问性
2. 接口的基接口的可访问性要高于或等同于这个接口的可访问性。

这两个约束条件的实例代码如下：

```
001  using System;
002  namespace Accessible1
003  {
004      internal class A
005      {
006          private class InterA
007          {
008          }
009      }
010      internal interface IA
011      { }
012      internal class B : A
013      { }
014      internal interface IB : IA
015      { }
016      public class C : A
017      { }
018      public interface IC : IA
019      { }
```

```

020     public class D : A. InterA
021     { }
022     class Program
023     {
024         static void Main(string[] args)
025         {
026         }
027     }
028 }

```

在这段代码中类 B 从类 A 继承，接口 IB 从接口 IA 继承都没问题，因为它们的可访问性与它们的基类相当。但是类 C 和接口 IC 在编译时就会报错，因为它们的可访问性高于它们的基类。类 D 想从一个内部类继承，但是这个内部类是 private 的，因此无法访问。

3. 委托类型的返回类型和参数类型的可访问性要高于或等同于委托类型本身的可访问性。
4. 常量的类型的可访问性要高于或等同于常量本身的可访问性。
5. 方法的返回类型和参数类型的可访问性必须高于或等同于方法本身的可访问性。

实例代码如下：

```

001     using System;
002     namespace Accessible2
003     {
004         internal class A
005         {
006         }
007         public delegate A DA(A a);
008         public class Program
009         {
010             private const int b = 11;
011             public static DA myDA;
012             public static A Test(A a)
013             {
014                 return a;
015             }
016             static void Main(string[] args)
017             {
018                 myDA = new DA(Test);
019                 A myA = new A();
020                 myDA(myA);
021             }
022         }
023     }

```

这一段代码在编译时会报告 4 个错误，错误信息如下：

```

错误 1    可访问性不一致：参数类型“Accessible2.A”比委托“Accessible2.DA”的可访问性低  H:\
书稿 new\书稿源码\第十五章\TestEnum\Program.cs 7    23    TestEnum
错误 2    可访问性不一致：返回类型“Accessible2.A”比委托“Accessible2.DA”的可访问性低  H:\
书稿 new\书稿源码\第十五章\TestEnum\Program.cs 7    23    TestEnum

```

错误 3 可访问性不一致：参数类型“Accessible2.A”比方法“Accessible2.Program.Test(Accessible2.A)”的可访问性低 H:\书稿 new\书稿源码\第十五章\TestEnum\Program.cs 12 25 TestEnum

错误 4 可访问性不一致：返回类型“Accessible2.A”比方法“Accessible2.Program.Test(Accessible2.A)”的可访问性低 H:\书稿 new\书稿源码\第十五章\TestEnum\Program.cs 12 25 TestEnum

产生这 4 个错误提示的原因是，类 A 的可访问性是 internal 的，当定义委托类型时，委托类型的参数和返回类型都为 A，所以，这个委托类型的可访问性只能比类 A 更低或等同，但是代码中为它设置了 public 的可访问性，违反了可访问性约束条件。接下来又在类中定义了一个静态的方法，这个方法的返回类型和参数也都为 A，但是这个方法的可访问性被设置为 public，它高于了类 A 的可访问性，所以编译出错。代码中的常量 b 则没问题，它是 int 类型的，int 类型的可访问性为 public，在这段代码中，b 的可访问性是 private 的，它低于 int 类型的可访问性。

6. 字段的类型的可访问性必须高于或等同于字段本身的可访问性。

7. 属性的类型的可访问性必须高于或等同于属性本身的可访问性。

8. 事件的类型和可访问性必须高于或等同于事件本身的可访问性。

这三个规则的代码实例如下：

```
001 using System;
002 namespace Accessible2
003 {
004     internal class A
005     {
006     }
007     internal delegate A DA(A a);
008     public class Program
009     {
010         public A p = new A();
011         public A P
012         {
013             get
014             {
015                 return p;
016             }
017             set
018             {
019                 p = value;
020             }
021         }
022         public event DA eDA;
023         private static A Test(A a)
024         {
025             return a;
026         }
027         static void Main(string[] args)
028         {
```

```

029         A a = new A();
030         Program pro = new Program();
031         pro.eDA = new DA(Test);
032         pro.eDA(a);
033     }
034 }
035 }

```

在编译这段代码的时候，会报告 3 个错误，错误信息如下：

错误 1 可访问性不一致：字段类型 “Accessible2.A” 比字段 “Accessible2.Program.p” 的可访问性低 H:\书稿 new\书稿源码\第十五章\TestEnum\Program.cs 10 18 TestEnum

错误 2 可访问性不一致：属性类型 “Accessible2.A” 比属性 “Accessible2.Program.P” 的可访问性低 H:\书稿 new\书稿源码\第十五章\TestEnum\Program.cs 11 18 TestEnum

错误 3 可访问性不一致：字段类型 “Accessible2.DA” 比字段 “Accessible2.Program.eDA” 的可访问性低 H:\书稿 new\书稿源码\第十五章\TestEnum\Program.cs 22 25 TestEnum

产生这 3 个错误的原因是，字段 p 的类型是类型 A，它的可访问性比字段 p 的可访问性低。属性类型也是类 A，它的可访问性比属性本身的可访问性低。事件 eDA 的类型是委托类型 DA，它的可访问性比 eDA 本身的可访问性低。

9. 索引器的类型和参数类型的可访问性必须高于或等同于索引器本身的可访问性。

10. 运算符的返回类型和参数类型的可访问性必须高于或等同于运算符本身的可访问性。

11. 实例构造方法的参数类型的可访问性必须高于或等同于实例构造方法本身的可访问性。

这三个规则的实例代码如下：

```

001 using System;
002 namespace Accessible2
003 {
004     internal class A
005     {
006         internal int a;
007         internal A(int a)
008         {
009             this.a = a;
010         }
011         public static A operator +(A a1, A a2)
012         {
013             A temp = new A(a1.a + a2.a);
014             return temp;
015         }
016     }
017     public class Program
018     {
019         private A[] p = new A[10];
020         public A this[int index]
021         {
022             get
023             {

```

```

024         return p[index];
025     }
026 }
027 public Program(A a)
028 {
029     for (int i = 0; i < 10; i++)
030     {
031         p[i] = a;
032     }
033 }
034 static void Main(string[] args)
035 {
036     A a1 = new A(11);
037     A a2 = new A(12);
038     Program pro1 = new Program(a1);
039     Program pro2 = new Program(a2);
040     A[] a = new A[10];
041     for (int i = 0; i < 10; i++)
042     {
043         a[i] = pro1[i] + pro2[i];
044         Console.WriteLine("a[{0}]:{1}", i, a[i].a);
045     }
046     Console.ReadKey();
047 }
048 }
049 }

```

这段代码在编译的时候会收到 2 个错误，错误信息如下：

错误 1 可访问性不一致：索引器返回类型“Accessible2.A”比索引器“Accessible2.Program.this[int]”的可访问性低 H:\书稿 new\书稿源码\第十五章\TestEnum\Program.cs 20 18 TestEnum

错误 2 可访问性不一致：参数类型“Accessible2.A”比方法“Accessible2.Program.Program(Accessible2.A)”的可访问性低 H:\书稿 new\书稿源码\第十五章\TestEnum\Program.cs 27 16 TestEnum

首先来看运算符重载方法（这里了解即可，后面会介绍到），这个方法看起来好像应该报错，实际上它是正确的，虽然运算符重载方法返回的类型和参数类型都是 internal 的，而方法本身是 public 的，但是实际上受上下文的影响，运算符重载方法的实际可访问性被限制为 internal，所以不会出问题。但是索引器和构造方法的设定可访问性为 public，而类 A 的可访问性为 internal，它低于索引器和构造方法本身的设定可访问性，因此编译器将报错。如果将类 A 的可访问性设置为 public，则执行结果如下：

```

a[0]:23
a[1]:23
a[2]:23
a[3]:23
a[4]:23
a[5]:23

```

```
a[6]:23
a[7]:23
a[8]:23
a[9]:23
```

15.6 嵌套类型的成员隐藏

在类的继承过程中，嵌套类型可以隐藏继承来的基类同名成员。实例代码如下：

```
001 using System;
002 namespace NestedType1
003 {
004     class A
005     {
006         public void Show()
007         {
008             Console.WriteLine("类 A 中的实例方法 Show 方法被调用");
009         }
010     }
011     class B : A
012     {
013         public class Show
014         {
015             public void Test()
016             {
017                 Console.WriteLine("内部类 Show 中的实例方法 Test 方法被调用");
018             }
019         }
020         public void Invoke()
021         {
022             Show myShow = new Show();
023             myShow.Test();
024             base.Show();
025         }
026     }
027     class Program
028     {
029         static void Main(string[] args)
030         {
031             B myB = new B();
032             myB.Invoke();
033             Console.ReadKey();
034         }
035     }
036 }
```

在这段代码中，第 4 行定义了一个类 A，在类 A 中包含一个实例方法 Show。类 B 从类 A 继承，在类 B 中，第 13 行代码定义了一个内部类。内部类的名称也叫做 Show。在类内部定义了一个实例方法。在第 20

行代码中，定义了一个实例方法 `Invoke`。在方法中定义了一个内部类的实例，然后用这个实例调用 `Test` 方法。在这个方法中还计划调用基类的 `Show` 方法。但是这个方法被内部类隐藏掉了，要想访问它，只能使用 `base` 关键字进行调用。在编译时，编译器将会提示警告信息，如下：

警告 1 “NestedType1.B.Show” 隐藏了继承的成员 “NestedType1.A.Show()”。如果是有意隐藏，请使用关键字 `new`。 H:\书稿 new\书稿源码\第十五章\NestedType1\Program.cs 13 22 NestedType1

如果在内部类定义的时候，使用 `new` 关键字，则会避免这个警告。这段代码的执行结果如下：

内部类 Show 中的实例方法 Test 方法被调用

类 A 中的实例方法 Show 方法被调用

15.7 嵌套类对包含类成员的访问

嵌套类型可以访问包含它的类的全部成员。代码实例如下：

```
001 using System;
002 namespace NestedType1
003 {
004     class A
005     {
006         private int a;
007         protected int b;
008         protected internal int c;
009         internal static void Show()
010         {
011             Console.WriteLine("类 A 中的静态方法 Show 被调用");
012         }
013         public class B
014         {
015             public void Test(A myA)
016             {
017                 Console.WriteLine("外部类的字段 a 的值是: {0}", myA.a);
018                 Console.WriteLine("外部类的字段 b 的值是: {0}", myA.b);
019                 Console.WriteLine("外部类的字段 c 的值是: {0}", myA.c);
020                 A.Show();
021             }
022         }
023         public A()
024         {
025             this.a = 100;
026             this.b = 200;
027             this.c = 300;
028         }
029     }
030     class Program
031     {
032         static void Main(string[] args)
033         {
034             A myA = new A();
```



```

035         A.B myB = new A.B();
036         myB.Test(myA);
037         Console.ReadKey();
038     }
039 }
040 }

```

内部类对外部类的访问，如果访问的是实例成员，则需要通过外部类的实例来访问；反过来，外部类对内部类的访问也是一样，如果要访问实例成员，则需要通过类的实例来访问。如果访问的是静态成员，则直接通过类名引用即可。

在第 13 行的内部类 B 中定义了一个实例方法 Test。内部类可以访问外部类的全部成员，但是必须通过外部类的实例来引用。因此对于实例方法 Test，要为其传入一个外部类型的实例。但是对于外部类型的静态方法来说，在第 20 行直接使用外部类的名称引用即可。因为外部类的可访问性是 internal，而内部类的可访问性是 public。所以内部类的可访问域是 internal 的，即在本程序集中的任何位置都可以引用内部类。因此，在第 35 行的独立类 Program 中可以创建内部类的实例。第 36 行代码调用内部类的实例方法，然后将外部类的实例传递进去。这段代码的执行结果如下：

```

外部类的字段 a 的值是：100
外部类的字段 b 的值是：200
外部类的字段 c 的值是：300
类 A 中的静态方法 Show 被调用

```

15.8 内部类的 this 访问

虽然嵌套类型是定义在一个类的内部的成员，看起来它好像和外部类的其它成员一样，实际上，它和外部类并不具有特别的关系。上一小节已经介绍过了，外部类和内部类的互相访问都需要通过对方的实例来进行。对外部类和内部类来讲，如果要访问对方的成员，除了向自己的实例方法传递对方类型的实例之外，还可以使用 this 引用来传递对方的实例。在内部类中，this 代表的是它自己，并不代表外部类。下面是代码实例：

```

001     using System;
002     namespace NestedType1
003     {
004         class A
005         {
006             private int a;
007             public void Show()
008             {
009                 Console.WriteLine("类 A 中的字段 a 的值是：{0}", a);
010             }
011             public class B
012             {
013                 private A myA;
014                 public B(A a)
015                 {
016                     this.myA = a;
017                 }
018                 public void Invoke()
019                 {

```

```

020         myA.Show();
021     }
022 }
023 public A()
024 {
025     this.a = 100;
026 }
027 }
028 class Program
029 {
030     static void Main(string[] args)
031     {
032         A.B myB = new A.B(new A());
033         myB.Invoke();
034         Console.ReadKey();
035     }
036 }
037 }

```

在这段代码中的内部类 B 中，包含一个外部类的变量字段。第 14 行代码中的构造方法中，传入了一个外部类型的实例用来初始化内部类的字段变量。注意第 16 行使用的 this 引用代表的是内部类，而不是外部类。第 23 行代码是外部类的构造方法，它使用的 this 引用代表的是外部类。这两个 this 引用虽然总体上说都在外部类的类体中使用，但是它们代表的不是同一个类。

第 32 行代码实例化了一个内部类的实例，直接使用 new 运算符为它传入外部类的实例。第 33 行调用内部类的实例方法。这段代码的执行结果如下：

类 A 中的字段 a 的值是：100

15.9 内部接口

和类与结构相似，接口也可以作为类的成员，这时它就是一个嵌套接口。实例代码如下：

```

001 using System;
002 namespace NameHide
003 {
004     class A
005     {
006         public interface B
007         {
008             void Show();
009         }
010         public class C : B
011         {
012             public void Show()
013             {
014                 Console.WriteLine("实现接口 B 的方法 Show 被调用");
015             }
016         }
017     }

```

```

018     class Program
019     {
020         static void Main(string[] args)
021         {
022             A.C myC = new A.C();
023             myC.Show();
024             A.B myB = myC;
025             myB.Show();
026             Console.ReadKey();
027         }
028     }
029 }

```

在类 A 中，定义了一个接口 B。接着又定义了一个内部类 C 实现了 B。因为 C 的可访问性为 public，它的外部类的访问性为 internal。在类的外部可以访问这个内部类。第 22 行定义了内部类 C 的实例 myC，然后调用内部类的实例方法 Show。因为接口 B 的可访问性与内部类 C 的一样，所以可以定义一个接口类型的引用指向内部类的实例。第 25 行由接口的引用调用其 Show 方法实现多态。这段代码的执行结果如下：

实现接口 B 的方法 Show 被调用

实现接口 B 的方法 Show 被调用

那么，如果类的内部定义了一个嵌套接口。这个类能否实现这个内部接口呢？代码如下：

```

001     using System;
002     namespace NameHide
003     {
004         class A:B
005         {
006             public interface B
007             {
008                 void Show();
009             }
010         }
011         class Program
012         {
013             static void Main(string[] args)
014             {
015                 Console.ReadKey();
016             }
017         }
018     }

```

第 4 行代码定义了一个类 A，在它内部定义了一个接口 B。然后让 A 实现它内部的接口 B。这时，编译器会报告如下的错误：

```

错误 1    未能找到类型或命名空间名称“B”（是否缺少 using 指令或程序集引用？）H:\书稿 new\书稿源码\第十五章\NestedType1\Program.cs    4    13    NestedType1

```

同理，类也不能从它内部的类继承。

15.10 静态类中的内部类

静态类除了不能从指定基类继承外，静态类也不能实现接口。在静态类中可以定义内部类，虽然外部

类是静态的，但是静态类中的内部类除非特别指定 `static` 关键字，否则，静态类中的内部类不是静态的。下面看代码实例：

```

001    using System;
002    namespace NameHide
003    {
004        static class A
005        {
006            static class B
007            {
008                public static void Show()
009                {
010                    Console.WriteLine("内部静态类中的方法被调用");
011                }
012            }
013            public static void Invoke()
014            {
015                B.Show();
016            }
017        }
018        class Program
019        {
020            static void Main(string[] args)
021            {
022                A.Invoke();
023                Console.ReadKey();
024            }
025        }
026    }

```

内部类也可以是静态的，这段代码演示了静态内部类的创建。第 4 行定义的静态类没有写任何访问修饰符，所以它是 `Internal` 的。它的内部类 B 是一个静态类，它也没有标注任何访问修饰符，它的默认访问权限是 `private` 的。因此在外无法直接访问。在第 13 行定义了一个静态的方法用来调用静态的内部类。第 22 行直接使用静态类 A 的类名引用其中的 `public` 权限的静态方法，再由这个静态方法调用静态类中的静态方法。这段代码的执行结果如下：

内部静态类中的方法被调用

静态类的内部类也可以是非静态类，实例代码如下：

```

001    using System;
002    namespace NameHide
003    {
004        static class A
005        {
006            private class B
007            {
008                public void Show()
009            {

```

```
010         Console.WriteLine("内部静态类中的方法被调用");
011     }
012 }
013 public static void Invoke()
014 {
015     B myB = new B();
016     myB.Show();
017 }
018 }
019 class Program
020 {
021     static void Main(string[] args)
022     {
023         A.Invoke();
024         Console.ReadKey();
025     }
026 }
027 }
```

这段代码中，外部类是一个静态类，而定义的内部类是一个非静态类。并且内部类的可访问性为 `private`。因此它需要一个外部类的方法来调用。因为外部类是一个静态类，所以外部类的方法也必须是静态方法。在静态方法 `Invoke` 中，首先定义了内部类的一个实例，然后用这个实例调用内部类的实例成员方法。这段代码的执行结果如下：

内部静态类中的方法被调用

第十六章 预处理指令

预处理指令不会被编译为可执行代码，但是它们影响编译过程的各个方面。包括跳过源文件中的代码块、报告错误、设定警告条件以及标注源代码的不同区域。

16.1 声明指令

声明指令包括 `define` 和 `undef` 两个指令，它们的作用是定义或取消条件编译符号。下面是代码实例：

```
001  #define A
002  #define A
003  #undef A
004  #undef A
005  #undef A
006  using System;
007  namespace Preprocessed
008  {
009  #define B
010  #undef B
011      class Program
012      {
013          static void Main(string[] args)
014          {
015              Console.ReadKey();
016          }
017      }
018  }
```

定义一个条件编译符号的指令是 `define`，它后跟一个标识符。取消编译符号的指令是 `undef`，它后面跟要取消的编译符号。在标识符之后不加分号结尾。定义编译符号和取消编译符号的指令都可多次使用，不会出错。第 1 行和第 2 行代码就是定义了一个编译符号 A。第 3 行到第 5 行就是取消了编译符号 A。定义条件编译符号和取消编译符号指令只能位于代码中第一个标记的前面，也就是说它必须位于所有实代码前面。第 9 行代码和第 10 行代码将会出错。这个位置不允许定义条件编译符号和取消条件编译符号指令。这两行代码将引起下面的错误信息：

```
错误 1 不能在文件的第一个标记之后定义或取消定义预处理器符号 H:\书稿 new\书稿源码\第十六章\Preprocessed\Program.cs 9 2 Preprocessed
错误 2 不能在文件的第一个标记之后定义或取消定义预处理器符号 H:\书稿 new\书稿源码\第十六章\Preprocessed\Program.cs 10 2 Preprocessed
```

最后需要注意的是，定义的条件编译符号只在定义它的那个编译单元有效。

16.2 条件判断指令

条件判断指令的作用是基于声明指令定义的条件编译符号进行判断，按照结果包含源代码或排除源代码的一部分。下面来看实例代码：

```
001  #define A
002  #define B
003  #define C
004  using System;
005  namespace Preprocessed
006  {
```

```
007     class Program
008     {
009         public static void Show1()
010         {
011             #if A
012                 Show2();
013             #endif
014             #if D
015                 Show3();
016             #elif B
017                 Show4();
018             #else
019                 Show5();
020             #endif
021         }
022         public static void Show2()
023         {
024             Console.WriteLine("Show2 方法被调用");
025         }
026         public static void Show3()
027         {
028             Console.WriteLine("Show3 方法被调用");
029         }
030         public static void Show4()
031         {
032             Console.WriteLine("Show4 方法被调用");
033         }
034         public static void Show5()
035         {
036             Console.WriteLine("Show5 方法被调用");
037         }
038         static void Main(string[] args)
039         {
040             Show1();
041             Console.ReadKey();
042         }
043     }
044 }
```

代码的第 1 行到第 3 行定义了三个条件编译符号。在使用 `define` 和 `undef` 指令的时候，这两个指令必须放在所有实代码前面。在类 `Program` 中定义了 5 个静态方法，在 `Show1` 方法中使用条件判断指令对定义的条件编译符号进行判断，按照结果取舍调用的其它静态方法。条件判断指令包括 `#if`、`#elif`、`#else` 和 `#endif`。其中 `#if` 和 `#endif` 必须成对使用，在其中可以包括若干个 `#elif` 和 `#else` 指令。现在来看第 11 行到第 20 行的代码，第 11 行定义了一个 `#if` 语句。有 `#if` 语句就必须有与它成对使用的 `#endif` 语句，例如第 13 行的 `#endif` 语句就是与它成对使用的语句。第 11 行代码的意思是，如果定义了 `A`，则调用 `Show2` 方

法。第 14 行代码中，如果定义了 D，则调用 Show3 方法。否则如果定义了 B，则调用 Show4 方法。如果 D 与 B 都没定义，则调用 Show5 方法。第 14 行的语句与第 20 行的语句成对使用。这里的 #elif 和 else if 的意思是一样的。但是不能使用 #else if 来代替 #elif。因为在语法中，是没有 #else if 语句的。这段代码的执行结果如下：

Show2 方法被调用

Show4 方法被调用

如果条件编译指令生效，尽管被跳过的源代码在语法上不正确，也不会造成错误。这是条件编译指令的特性，但是在实际编写代码时要尽量避免这种情况的发生。实例代码如下：

```
001  #define A
002  #define B
003  #define C
004  using System;
005  namespace Preprocessed
006  {
007      class Program
008      {
009          public static void Show1()
010          {
011              #if A
012                  Show2();
013              #endif
014              #if D
015                  Show3(
016                  #elif B
017                      Show4());
018              #else
019                  Show5 else
020              #endif
021          }
022          public static void Show2()
023          {
024              Console.WriteLine("Show2 方法被调用");
025          }
026          public static void Show3()
027          {
028              Console.WriteLine("Show3 方法被调用");
029          }
030          public static void Show4()
031          {
032              Console.WriteLine("Show4 方法被调用");
033          }
034          public static void Show5()
035          {
036              Console.WriteLine("Show5 方法被调用");
```



```
037     }
038     static void Main(string[] args)
039     {
040         Show1();
041         Console.ReadKey();
042     }
043 }
044 }
```

第 15 行代码和第 19 行代码尽管语法是错误的，但编译器不会报错。因为在条件编译的逻辑上，它们是被跳过的代码。

当预处理指令出现在以@符号开始的原文字符串中时，不作为预处理指令处理。实例代码如下：

```
001 #define A
002 using System;
003 namespace Preprocessed1
004 {
005     class Program
006     {
007         static void Main(string[] args)
008         {
009             Console.WriteLine(@"#if A Hello #endif World");
010             Console.ReadKey();
011         }
012     }
013 }
```

这段代码的第 9 行中，WriteLine 方法中的参数部分是一个原文字符串，在原文字符串中，预处理指令是无效的。

在预处理指令的使用过程中，注释符号的优先级是高于预处理指令的。看下面的代码：

```
001 #define A
002 // #undef A
003 using System;
004 namespace Preprocessed1
005 {
006     #if A
007         /*
008     #else
009         /* */ class Test
010         {
011             private int a;
012             public Test()
013             {
014                 a = 100;
015             }
016         }
017     #endif
```

```

018     class Program
019     {
020         static void Main(string[] args)
021         {
022             Test t = new Test();
023             Console.ReadKey();
024         }
025     }
026 }

```

在代码的第 1 行定义了一个条件编译符号 A，第 2 行的取消条件编译符号的指令暂时注释掉了。首先来看代码的执行，第 9 行到第 16 行的类 Test 的定义，无论第 2 行的取消指令是否有效，这个类始终会被定义。原因是，注释符号的优先级高于预处理指令。如果 A 是有效的，第 7 行的多行注释符号的左半部分是有效的，它和第 9 行的多行注释的右半部分进行组合，将它们之间的内容注释掉了。这时是不管第 8 行的 #else 语句是否有效的。这种情况下，类 Test 的定义是有效的。

当取消第 2 行的注释符号之后，A 的定义是无效的。这时，第 7 行的代码被跳过。第 8 行的 #else 指令变得有效。但是它下面定义了一对多行注释符号。这一对注释符号变成什么内容也没注释。它后面的类的定义依然有效，没有被跳过。

16.3 预处理指令中的运算符

在预处理指令中，还可以使用 !、==、!=、&& 和 || 帮助进行条件的判断。使用这些运算符后总会产生一个 bool 值。对于已经定义的编译符号来说，它具有 bool 值 true；未定义的编译符号具有 bool 值 false。实例代码如下：

```

001 #define A
002 #define Version1
003 #define Version2
004 #define B
005 #define C
006 #define D
007 #undef A
008 using System;
009 namespace Preprocessed1
010 {
011     class Program
012     {
013         public static void Show()
014         {
015             #if A == B
016                 Show1();
017             #elif (Version1 == true) && (B == true)
018                 Show2();
019             #endif
020             #if Version2 != A
021                 Show3();
022             #endif
023             #if (C == true) || (D == false) || (!A == true)

```

```

024         Show4();
025     #endif
026 }
027 public static void Show1()
028 {
029     Console.WriteLine("Show1 方法被调用");
030 }
031 public static void Show2()
032 {
033     Console.WriteLine("Show2 方法被调用");
034 }
035 public static void Show3()
036 {
037     Console.WriteLine("Show3 方法被调用");
038 }
039 public static void Show4()
040 {
041     Console.WriteLine("Show4 方法被调用");
042 }
043 static void Main(string[] args)
044 {
045     Show();
046     Console.ReadKey();
047 }
048 }
049 }

```

在第 1 行到第 7 行代码中定义的这些条件编译符号，因为 A 被取消掉了。所以除了 A 之外，都为 true，A 为 false。第 15 行代码判断 A 和 B 是否相等，因为 A 为 false，而 B 为 true。所以将返回 false。第 16 行代码将被跳过。第 17 行代码判断 Version1 和 B 是否同时为 true，这个表达式将返回 true。因此第 18 行代码将是有效的。第 20 行代码判断 Version2 是否和 A 不相等，因为 Version2 为 true，而 A 为 false。所以这个表达式将返回 true，第 21 行代码有效。第 23 行代码判断 C 为 true 或者 D 为 false 或者非 A 为 true，只要这三个判断表达式中有一个为 true，则整个表达式返回 true。因为 C 为 true，D 为 true，非 A 为 true，只有 D 那个表达式返回 false。但是整个表达式的逻辑是 true。第 24 行代码是有效的。这段代码的执行结果如下：

```

Show2 方法被调用
Show3 方法被调用
Show4 方法被调用

```

16.4 诊断指令

预处理指令中的诊断指令包含两个指令，#error 和#warning 指令。它们的作用是产生一个错误提示和一个警告信息。实例代码如下：

```

001 #define Version1
002 #define Version2
003 using System;
004 namespace Preprocessed1

```

```
005  {
006      class Program
007      {
008          public static void Show()
009          {
010              #if Version1 == true
011              #warning 这时编译的是一个测试版本，发布版本请编译 Version2
012              Show1();
013              Show2();
014              #endif
015              #if Version2 == true
016              #error 两个版本的代码不能同时进行编译
017              #endif
018          }
019          public static void Show1()
020          {
021              Console.WriteLine("Show1 方法被调用");
022          }
023          public static void Show2()
024          {
025              Console.WriteLine("Show2 方法被调用");
026          }
027          public static void Show3()
028          {
029              Console.WriteLine("Show3 方法被调用");
030          }
031          public static void Show4()
032          {
033              Console.WriteLine("Show4 方法被调用");
034          }
035          static void Main(string[] args)
036          {
037              Show();
038              Console.ReadKey();
039          }
040      }
041  }
```

这段代码中定义了两个表示版本的编译符号，Version1 表示测试版本；Version2 表示正式发布的版本。第 10 行进行判断，如果 Version1 为 true，则提示一条警告信息。#warning 指令的用法是在#warning 指令后跟要提示的警告信息的内容。如果是测试版本则调用第 12 行和第 13 行的两个方法。第 15 行代码判断是否定义了 Version2，如果定义了 Version2 而且前面的代码定义了 Version1，则提示一条错误信息。两个版本的代码不能一起编译。#error 指令的用法和#warning 指令的用法是一样的。编译器提示的信息如下：

```
警告 1    #warning: '这时编译的是一个测试版本，发布版本请编译 Version2' E:\书稿new\书稿源码\第十六章\Preprocessed1\Program.cs 11 22 Preprocessed1
```

错误 2 #error: '两个版本的代码不能同时进行编译' E:\书稿 new\书稿源码\第十六章
\Preprocessed1\Program.cs 16 20 Preprocessed1

16.5 区域指令

区域指令起到规整源代码的作用，它不会起到其它的作用。区域指令包括#region 和#endregion 两个指令，它们必须成对使用。实例代码如下：

```
001 using System;
002 namespace Preprocessed1
003 {
004     class Program
005     {
006         public static void Show()
007         {
008             Show1();
009             Show2();
010             Show3();
011             Show4();
012         }
013         #region 被调用的 4 个静态方法
014         public static void Show1()
015         {
016             Console.WriteLine("Show1 方法被调用");
017         }
018         public static void Show2()
019         {
020             Console.WriteLine("Show2 方法被调用");
021         }
022         public static void Show3()
023         {
024             Console.WriteLine("Show3 方法被调用");
025         }
026         public static void Show4()
027         {
028             Console.WriteLine("Show4 方法被调用");
029         }
030         #endregion
031         static void Main(string[] args)
032         {
033             Show();
034             Console.ReadKey();
035         }
036     }
037 }
```

代码的第 13 行和第 30 行就是成对使用的区域指令。在第 13 行的#region 指令后还可以再加上提示的消息。使用这对指令包裹代码代码后，在开发环境的编辑器中的第 13 行前面就会有一个“-”号，单击它

就可以将区域指令包裹的代码都折叠起来，并给出一个消息的提示。这时的代码实例如下：

```

001  using System;
002  namespace Preprocessed1
003  {
004      class Program
005      {
006          public static void Show()
007          {
008              Show1();
009              Show2();
010              Show3();
011              Show4();
012          }
013          被调用的 4 个静态方法
014          static void Main(string[] args)
015          {
016              Show();
017              Console.ReadKey();
018          }
019      }
020  }
```

因为这 4 个方法的形式和目的都差不多，全部打开则很占空间。如果要对它们再进行编辑，则单击第 13 行代码前面的“+”号，这时所有的方法又都可以打开进行编辑了。

16.6 行指令

行指令用于变更编译器在输出警告信息和错误信息时报告的行号和源文件名称。还可以改变调用方信息特性所使用的行号和源文件名称。行指令包括#line、#line default 和#line hidden。实例代码如下：

```

001  #define Version1
002  #define Version2
003  #line 100 "test.cs"
004  using System;
005  namespace Preprocessed1
006  {
007      class Program
008      {
009          public static void Show()
010          {
011              #if Version1
012              #warning 编译版本为 Version1
013              #endif
014              #if Version1 == true && Version2 == true
015              #error 两个版本不能同时进行编译
016              #endif
017              #line default
018              #warning 编译版本为 Version1 (使用#line default 指令后)
```

```

019  #error 两个版本不能同时进行编译（使用#line default 指令后）
020          Show1();;
021      }
022      public static void Show1()
023      {
024          Console.WriteLine("Show1 方法被调用");
025      }
026      static void Main(string[] args)
027      {
028          Show();
029          Console.ReadKey();
030      }
031  }
032  }

```

使用#line 指令后，这一行指令的后续代码在编译器的提示信息中的行数将从#line 指令指定的行号开始计算。源代码文件名也使用#line 指令指定的文件名。第 3 行指定行号为 100 后，第 4 行代码如果有提示信息的输出，将使用行号 100。然后行号从 100 开始累加。第 12 行和第 15 行的输出信息将采用新指定的行号。第 17 行使用#line default 指令将行号指定为原始值，第 18 行和第 19 行的输出信息将回归正常。编译器的输出信息如下：

```

警告 1  #warning: '编译版本为 Version1（使用#line default 指令后）' Program.cs 18 10
错误 2  #error: '两个版本不能同时进行编译（使用#line default 指令后）' Program.cs 19 8
警告 3  #warning: '编译版本为 Version1' test.cs 108 10
错误 4  #error: '两个版本不能同时进行编译' test.cs 111 8

```

#line hidden 指令对错误信息中报告的文件号和行号无效，它用在代码调试的环境。在调试时，#line hidden 指令和后面的 #line 指令（不是 #line hidden）之间的所有行都没有行号信息。在调试器中逐句执行代码时，将全部跳过这些行。

16.7 pragma 指令

pragma 指令目前在 C# 中用来控制编译器的警告信息。C# 语言的将来版本可能包含更多的 pragma 指令。pragma 指令目前主要使用的是 pragma warning 指令。它用来禁用或恢复指定的警告信息。实例代码如下：

```

001  #define Version1
002  #define Version2
003  //#pragma warning disable 1030
004  using System;
005  namespace Preprocessed1
006  {
007      class Program
008      {
009          public static void Show()
010          {
011              #if Version1
012                  #warning 编译版本为 Version1
013              #endif
014                  Show1();;
015          }

```

```

016         public static void Show1()
017         {
018             Console.WriteLine("Show1 方法被调用");
019         }
020         static void Main(string[] args)
021         {
022             Show();
023             Console.ReadKey();
024         }
025     }
026 }

```

这段代码中的第 12 行将会在编译器的错误列表中输出一条警告信息。警告信息如下：

```

警告 1    #warning: '编译版本为 Version1' H:\书稿 new\书稿源码\第十六章
\Preprocessed1\Program.cs 12 10 Preprocessed1

```

如果不想让这条警告信息出现，可以去掉第 3 行代码的注释。第 3 行代码就是一条禁用警告信息的指令。指令的用法是 `#pragma warning disable` 后跟警告代码。当将这条指令的注释去掉之后，警告信息就不会再提示。警告代码如何获取呢？可以将项目进行生成，在“输出”项目中可以看到警告代码，如下：

```

1>----- 已启动生成: 项目: Preprocessed1, 配置: Debug Any CPU -----
1>H:\书稿 new\书稿源码\第十六章\Preprocessed1\Program.cs(12,10,12,23): warning CS1030:
#warning: "编译版本为 Version1"
1> Preprocessed1 -> H:\书稿 new\书稿源码\第十六章\Preprocessed1\bin\Debug\Preprocessed1.exe
===== 生成: 成功 1 个, 失败 0 个, 最新 0 个, 跳过 0 个 =====

```

输出信息中的 warning cs1030 中的 1030 就是警告代码，可以使用 `#pragma warning disable 1030` 指令禁用这条警告。如果在开发环境中看不到“输出”项，则单击开发环境的“视图”菜单，然后选择“输出”项。

如果要在编译时控制编译某段代码时不显示警告信息，在编译超过这段代码的时候再打开警告信息。则需要使用 `#pragma warning restore` 指令。实例代码如下：

```

001     #define Version1
002     #define Version2
003     #pragma warning disable 1030
004     using System;
005     namespace Preprocessed1
006     {
007         class Program
008         {
009             public static void Show()
010             {
011                 #if Version1
012                 #warning 编译版本为 Version1
013                 #endif
014                 #pragma warning restore 1030
015                 Show1();
016             }
017             public static void Show1()

```



```
018      {
019          Console.WriteLine("Show1 方法被调用");
020      }
021      static void Main(string[] args)
022      {
023          Show();
024          Console.ReadKey();
025      }
026  }
027 }
```

这段代码在第 3 行禁用掉编号为 1030 的警告信息之后，因为输出警告信息的代码在第 12 行。在编译器编译操作越过第 12 行之后，又可以打开禁用警告信息的操作。恢复操作使用 `#pragma warning restore 1030` 指令。如果想要取消掉全部警告信息，可以使用 `#pragma warning disable` 指令，不接任何警告信息代码。要想恢复，则使用 `#pragma warning restore` 指令，不接任何警告编号。

第十七章 匿名方法和 lambda 表达式

17.1 匿名方法的定义

匿名方法用在无需单独定义方法实体的场合，匿名方法和委托的使用息息相关。下面看代码实例：

```

001  using System;
002  namespace AnonymousFunction
003  {
004      public delegate int D(int a,int b);
005      class Program
006      {
007          private D myD;
008          private int Show1(int a, int b)
009          {
010              return a + b;
011          }
012          public Program()
013          {
014              myD += Show1;
015          }
016          public void Invoke()
017          {
018              int i = myD(100, 200);
019              Console.WriteLine("两个 int 类型的参数相加的结果是 {0}", i);
020          }
021          static void Main(string[] args)
022          {
023              Program p = new Program();
024              p.Invoke();
025              Console.ReadKey();
026          }
027      }
028  }

```

在这段代码中，类 Program 中定义了一个委托类型的字段 myD，第 8 行定义了一个方法 Show1，这个方法将在构造方法中添加进入委托的调用列表中。第 16 行定义了一个实例方法，在方法中调用委托成员。在这个例子中，可以观察到，实例方法 Show1 的作用就是被委托调用，再没有其它的成员来调用它。它更像是一种功能，在需要的时候调用一次。然后就再没有其它的成员来调用它。因此，它就没有必要这样定义一个单独的方法出来。它可以定义成一个匿名的方法，只供委托进行调用。下面是修改后的代码：

```

001  using System;
002  namespace AnonymousFunction
003  {
004      public delegate int D(int a,int b);
005      class Program
006      {
007          private D myD;

```

```

008     private int Show1(int a, int b)
009     {
010         return a + b;
011     }
012     public Program()
013     {
014         myD += delegate(int a, int b) { return a + b; };
015     }
016     public void Invoke()
017     {
018         int i = myD(100, 200);
019         Console.WriteLine("两个 int 类型的参数相加的结果是 {0}", i);
020     }
021     static void Main(string[] args)
022     {
023         Program p = new Program();
024         p.Invoke();
025         Console.ReadKey();
026     }
027 }
028 }

```

这段代码对委托的调用列表的添加没有使用方法 Show1 的名称，在第 14 行的构造方法体中，定义了一个匿名方法直接添加进了委托的调用列表中。匿名方法的定义方法就是，使用 delegate 关键字，然后后跟方法的形参列表，再跟一对大括号，大括号就是方法体，在其中定义要执行的语句。每条语句以分号结尾。这就是匿名方法的定义方式，很简单。在使用时，直接调用将它添加进调用列表的委托，为委托传入参数即可。匿名方法的返回类型没有直接指定，它由方法体中的 return 语句的返回值推导而得。这段代码的执行结果如下：

```
两个 int 类型的参数相加的结果是 300
```

17.2 lambda 表达式

lambda 表达式和上一小节的匿名方法表达式是匿名方法的两种语法格式。lambda 表达式的语法比匿名方法更简单。那么，什么是 lambda 表达式呢？lambda 表达式以 lambda 运算符 “=>” 作为标记。下面是代码实例：

```

001     using System;
002     namespace AnonymousFunction
003     {
004         public delegate int D(int a,int b);
005         class Program
006         {
007             private D myD;
008             public Program()
009             {
010                 myD += (int a, int b) =>{ return a + b; };
011             }
012             public void Invoke()

```

```
013     {
014         int i = myD(100, 200);
015         Console.WriteLine("两个 int 类型的参数相加的结果是 {0}", i);
016     }
017     static void Main(string[] args)
018     {
019         Program p = new Program();
020         p.Invoke();
021         Console.ReadKey();
022     }
023 }
024 }
```

代码的第 10 行由匿名方法的语法形式修改为 lambda 表达式的形式。可以观察到, lambda 表达式使用 lambda 表达式运算符将整个表达式分为两个部分, 运算符左边是形参列表, 右边是方法体。返回类型依然根据 return 语句推导而得。

17.3 匿名方法和 lambda 表达式的参数类型化

匿名方法的参数必须显式类型化, 也就是必须定义参数类型。匿名方法的参数列表也可以不提供, 在这种情况下, 匿名函数可以转换为带有不含 out 参数的参数列表的委托。实例代码如下:

```
001 using System;
002 namespace AnonymousFunction
003 {
004     public delegate int D(int a,int b);
005     class Program
006     {
007         private D myD;
008         private D myD1;
009         public Program()
010         {
011             myD += delegate(int a, int b){ return a + b; };
012             myD1 = delegate{return 100 * 20;};
013         }
014         public void Invoke()
015         {
016             int i = myD(100, 200);
017             Console.WriteLine("两个 int 类型的参数相加的结果是 {0}", i);
018             int i1 = myD1(10, 10);
019             Console.WriteLine("不带参数的匿名方法的返回结果是 {0}", i1);
020         }
021         static void Main(string[] args)
022         {
023             Program p = new Program();
024             p.Invoke();
025             Console.ReadKey();
026         }
027     }
028 }
```

```
027     }
028 }
```

第 11 行代码和第 12 行代码各定义了一个匿名方法。第 11 行代码的匿名方法带有一个形参列表，在匿名方法中，如果带有形参，形参必须显式提供参数类型。匿名方法也可以不定义参数列表，在这种情况下，它可以转换成带有不含 out 参数的参数列表的委托。例如第 12 行代码中的匿名方法的方法体中直接返回了两个 int 类型的文本的乘积。虽然委托类型带有参数，但是这个不带参数的匿名方法可以转换赋值给它。在第 16 行和第 18 行进行了正常的调用。这段代码的执行结果如下：

两个 int 类型的参数相加的结果是 300

不带参数的匿名方法的返回结果是 2000

lambda 表达式的形参可以显式类型化，也可以隐式类型化。但是 lambda 表达式的形参列表完全受调用它的委托类型的制约。看下面的代码实例：

```
001 using System;
002 namespace AnonymousFunction
003 {
004     public delegate int D(int a,int b);
005     class Program
006     {
007         private D myD;
008         private D myD1;
009         public Program()
010         {
011             myD += (int a, int b)=>{ return a + b; };
012             myD1 = ()=>{return 100 * 20;};
013         }
014         public void Invoke()
015         {
016             int i = myD(100, 200);
017             Console.WriteLine("两个 int 类型的参数相加的结果是 {0}", i);
018             int i1 = myD1(10, 10);
019             Console.WriteLine("不带参数的匿名方法的返回结果是 {0}", i1);
020         }
021         static void Main(string[] args)
022         {
023             Program p = new Program();
024             p.Invoke();
025             Console.ReadKey();
026         }
027     }
028 }
```

在这段代码中，第 11 行的 lambda 表达式的形参列表是显式类型化的，它必须和第 4 行的委托类型的形参列表一致。lambda 表达式的形参列表不能省略，必须提供。而且为空也不行。它必须和调用它的委托类型的形参列表一致。因为第 12 行代码中的 lambda 表达式的形参列表和委托类型不一致，所以编译器会报错。错误信息如下：

错误 1 委托“AnonymousFunction.D”未采用 0 个参数 H:\书稿 new\书稿源码\第十七章

\AnonymousFunction\Program.cs 12 20 AnonymousFunction

lambda 表达式的形参列表如果是隐式类型化，它的类型由调用它的委托类型提供类型。实例代码如下：

```

001 using System;
002 namespace AnonymousFunction
003 {
004     public delegate int D(int a,int b);
005     public delegate float D1(float a,float b);
006     class Program
007     {
008         private D myD;
009         private D myD1;
010         private D1 myD2;
011         public Program()
012         {
013             myD += (a, b)=>{ return a + b; };
014             myD1 = (a,b)=>{return 100 * 20;};
015             myD2 += (a, b) => { return a / b; };
016         }
017         public void Invoke()
018         {
019             int i = myD(100, 200);
020             Console.WriteLine("两个 int 类型的参数相加的结果是 {0}", i);
021             int i1 = myD1(10, 10);
022             Console.WriteLine("不带参数的匿名方法的返回结果是 {0}", i1);
023             float i2 = myD2(120.0F, 10);
024             Console.WriteLine("参数类型为 float 的匿名方法的返回结果是 {0}", i2);
025         }
026         static void Main(string[] args)
027         {
028             Program p = new Program();
029             p.Invoke();
030             Console.ReadKey();
031         }
032     }
033 }

```

在代码的第 5 行又定义了一个委托类型 D1，它的返回类型和形参都是 float 类型。第 10 行定义了一个 D1 类型的字段。第 15 行将一个 lambda 表达式形式的匿名方法添加进入 myD2 的调用列表。虽然第 15 行的 lambda 表达式的形参列表和第 13 行的形参列表相同，但是它们的形参类型是由调用它们的委托来提供的。因此它们是不同的类型。这段代码的执行结果如下：

两个 int 类型的参数相加的结果是 300

不带参数的匿名方法的返回结果是 2000

参数类型为 float 的匿名方法的返回结果是 12

如果 lambda 表达式的形参只有一个，那么它可以省略形参列表外面的括号。实例代码如下：

```

001 using System;

```

```
002 namespace AnonymousFunction
003 {
004     public delegate int D(int a);
005     class Program
006     {
007         private D myD;
008         private D myD1;
009         public Program()
010         {
011             myD += (a)=>{ return a + 300; };
012             myD1 = a => { return a * 100; };
013         }
014         public void Invoke()
015         {
016             int i = myD(100);
017             Console.WriteLine("lambda 表达式的返回结果是 {0}", i);
018             int i1 = myD1(200);
019             Console.WriteLine("lambda 表达式的返回结果是 {0}", i1);
020         }
021         static void Main(string[] args)
022         {
023             Program p = new Program();
024             p.Invoke();
025             Console.ReadKey();
026         }
027     }
028 }
```

第 4 行代码定义的委托类型只有一个形参。在第 11 行和第 12 行定义 lambda 表达式的时候，lambda 运算符左面的形参可以是隐式类型化的，并且隐式类型化的形参可以带括号，也可以不带括号。这段代码的执行结果如下：

lambda 表达式的返回结果是 400

lambda 表达式的返回结果是 20000

lambda 表达式的方法体可以只是一个表达式，而匿名方法的方法体必须是语句块。实例代码如下：

```
001 using System;
002 namespace AnonymousFunction
003 {
004     public delegate int D(int a);
005     class Program
006     {
007         private D myD;
008         private D myD1;
009         public Program()
010         {
011             myD += a => (a + 300);
```

```

012         myD1 = delegate(int a) { return (a * 100); };
013     }
014     public void Invoke()
015     {
016         int i = myD(100);
017         Console.WriteLine("lambda 表达式的返回结果是{0}", i);
018         int i1 = myD1(200);
019         Console.WriteLine("lambda 表达式的返回结果是{0}", i1);
020     }
021     static void Main(string[] args)
022     {
023         Program p = new Program();
024         p.Invoke();
025         Console.ReadKey();
026     }
027 }
028 }

```

代码的第 11 行和第 12 行各定义了一个 lambda 表达式和一个匿名方法。第 11 行的 lambda 表达式的方法体部分可以只使用一个语句，返回值也不用 return 关键字。但是第 12 行定义的匿名方法的方法体必须是大括号形式的语句块。这段代码的执行结果如下：

```

lambda 表达式的返回结果是 400
lambda 表达式的返回结果是 20000

```

17.4 匿名方法的签名

匿名方法的签名包括匿名方法的形参的名称和形参的类型。但是 lambda 表达式的形参类型有可能不提供。匿名方法的参数范围包括匿名方法的方法体，因此匿名方法的参数列表和它的方法体一起构成一个声明空间。下面看一个代码实例：

```

001     using System;
002     namespace AnonymousFunction
003     {
004         public delegate int D(int a);
005         class Program
006         {
007             private D myD;
008             private D myD1;
009             public Program()
010             {
011                 myD += a => { int a = 100; return a * 1000; };
012                 myD1 = delegate(int b) { int b = 20; return (b * 100); };
013             }
014             public void Invoke()
015             {
016                 int i = myD(100);
017                 Console.WriteLine("lambda 表达式的返回结果是{0}", i);
018                 int i1 = myD1(200);

```



```

019         Console.WriteLine("lambda 表达式的返回结果是 {0}", i1);
020     }
021     static void Main(string[] args)
022     {
023         Program p = new Program();
024         p.Invoke();
025         Console.ReadKey();
026     }
027 }
028 }

```

这段代码中的第 11 行和第 12 行的匿名方法都会引发错误。在第 11 行中，因为形参和 lambda 表达式的方法体一起构成一个声明空间，因此它们中的参数名称不能相同。第 12 行代码中的匿名方法也是这种情况。它们的错误提示如下：

```

错误 1    运算符 “+=” 无法应用于 “AnonymousFunction.D” 和 “lambda 表达式” 类型的操作数    H:\
书稿 new\书稿源码\第十七章\AnonymousFunction\Program.cs 11 13 AnonymousFunction
错误 2    不能在此范围内声明名为 “a” 的局部变量，因为这样会使 “a” 具有不同的含义，而它已在 “父
级或当前” 范围中表示其他内容了    H:\书稿 new\书稿源码\第十七章\AnonymousFunction\Program.cs
11 31 AnonymousFunction
错误 3    不能在此范围内声明名为 “b” 的局部变量，因为这样会使 “b” 具有不同的含义，而它已在 “父
级或当前” 范围中表示其他内容了    H:\书稿 new\书稿源码\第十七章\AnonymousFunction\Program.cs
12 42 AnonymousFunction

```

匿名方法可以不提供形参列表，在这种情况下，调用它的委托不能带有 out 类型的形参。实例代码如下：

```

001 using System;
002 namespace AnonymousFunction
003 {
004     public delegate int D(out int a);
005     class Program
006     {
007         private D myD;
008         public Program()
009         {
010             myD += delegate { int i = 10; return i * 100; };
011         }
012         public void Invoke()
013         {
014             int i;
015             int i1 = myD(out i);
016             Console.WriteLine("lambda 表达式的返回结果是 {0}", i1);
017         }
018         static void Main(string[] args)
019         {
020             Program p = new Program();
021             p.Invoke();

```

```

022         Console.ReadKey();
023     }
024 }
025 }

```

第4行定义的委托类型带有一个 out 类型的形参。第10行将一个不带形参列表的匿名方法转换赋值给委托类型的字段时，编译器会报告错误。错误信息如下：

错误 1 运算符“+=”无法应用于“AnonymousFunction.D”和“匿名方法”类型的操作数 H:\书稿 new\书稿源码\第十七章\AnonymousFunction\Program.cs 10 13 AnonymousFunction

错误 2 参数 1 必须使用关键字“out”声明 H:\书稿 new\书稿源码\第十七章\AnonymousFunction\Program.cs 10 20 AnonymousFunction

错误 3 无法将不含参数列表的匿名方法块转换为委托类型“AnonymousFunction.D”，原因是该方法块具有一个或多个 out 参数 H:\书稿 new\书稿源码\第十七章\AnonymousFunction\Program.cs 10 20 AnonymousFunction

出现错误的原因是，委托类型的形参是 out 类型，它要求转换赋值给它的匿名方法中必须对形参进行初始化操作。因此修改代码如下：

```

001 using System;
002 namespace AnonymousFunction
003 {
004     public delegate int D(out int a);
005     class Program
006     {
007         private D myD;
008         public Program()
009         {
010             myD += delegate(out int i) { i = 10; return i * 100; };
011         }
012         public void Invoke()
013         {
014             int i;
015             int i1 = myD(out i);
016             Console.WriteLine("lambda 表达式的返回结果是 {0}", i1);
017         }
018         static void Main(string[] args)
019         {
020             Program p = new Program();
021             p.Invoke();
022             Console.ReadKey();
023         }
024     }
025 }

```

这段代码对前面的代码进行了修改，第10行代码定义的匿名方法必须带有一个形参列表，而且形参列表的参数必须是 out 类型的。在方法体中对参数必须做初始化。这段代码的执行结果如下：

```
lambda 表达式的返回结果是 1000
```

17.5 匿名方法体

前面介绍过，匿名方法的语法形式有两种，第一种是传统的使用 delegate 关键字的语法形式；另外一种 lambda 表达式形式的匿名方法。在这两种语法形式中，除了在匿名方法的签名中，即形参列表中指定的 ref 或 out 类型的形参外，匿名方法体中不能访问其它的 ref 或 out 参数。实例代码如下：

```

001 using System;
002 namespace AnonymousFunction
003 {
004     public delegate int D(out int a);
005     class Program
006     {
007         private D myD;
008         public void AddList(ref int a,out int b)
009         {
010             a = 20;
011             b = 100;
012             myD += delegate(out int i) { a = 100; b = 200; i = 10; return i * 100; };
013             myD += (out int i1) => { a = 200; b = 300; i1 = 20; return i1 * 200; };
014         }
015         public void Invoke()
016         {
017             int i;
018             int i1 = myD(out i);
019             Console.WriteLine("lambda 表达式的返回结果是{0}", i1);
020         }
021         static void Main(string[] args)
022         {
023             int a = 10;
024             int b;
025             Program p = new Program();
026             p.AddList(ref a,out b);
027             p.Invoke();
028             Console.ReadKey();
029         }
030     }
031 }

```

在这段代码中，对委托的方法列表添加使用了一个类的实例方法 AddList。这个方法又带有 ref 和 out 类型的参数。在其中的匿名方法中，除了匿名方法本身的形参中的 out 参数外，匿名方法体中不能访问包裹它的方法体中的 ref 形参和 out 形参。这时编译器将会报错，错误信息如下：

```

错误 1    运算符“+=”无法应用于“AnonymousFunction.D”和“lambda 表达式”类型的操作数  H:\书稿 new\书稿源码\第十七章\AnonymousFunction\Program.cs 13 13 AnonymousFunction
错误 2    不能在匿名方法、lambda 表达式或查询表达式内使用 ref 或 out 参数“a” H:\书稿 new\书稿源码\第十七章\AnonymousFunction\Program.cs 13 38 AnonymousFunction
错误 3    不能在匿名方法、lambda 表达式或查询表达式内使用 ref 或 out 参数“b” H:\书稿 new\书稿源码\第十七章\AnonymousFunction\Program.cs 13 47 AnonymousFunction
错误 4    不能在匿名方法、lambda 表达式或查询表达式内使用 ref 或 out 参数“a” H:\书稿 new\

```

书稿源码\第十七章\AnonymousFunction\Program.cs 12 42 AnonymousFunction

错误 5 不能在匿名方法、lambda 表达式或查询表达式内使用 ref 或 out 参数“b” H:\书稿 new\书稿源码\第十七章\AnonymousFunction\Program.cs 12 51 AnonymousFunction

修改代码中的错误，正确的代码如下：

```

001    using System;
002    namespace AnonymousFunction
003    {
004        public delegate int D(out int a);
005        class Program
006        {
007            private D myD;
008            public void AddList(ref int a,out int b)
009            {
010                a = 20;
011                b = 100;
012                myD += delegate(out int i) { i = 10; return i * 100; };
013                myD += (out int i1) => { i1 = 20; return i1 * 200; };
014            }
015            public void Invoke()
016            {
017                int i;
018                int i1 = myD(out i);
019                Console.WriteLine("lambda 表达式的返回结果是 {0}", i1);
020            }
021            static void Main(string[] args)
022            {
023                int a = 10;
024                int b;
025                Program p = new Program();
026                p.AddList(ref a,out b);
027                p.Invoke();
028                Console.ReadKey();
029            }
030        }
031    }

```

这段代码中对于委托类型的字段 myD 来说，为它添加了两个匿名方法。在这两个匿名方法体中去掉了不可访问的 ref 和 out 参数 a 和 b。它们都带有一个 out 类型的形参。在实际调用委托的时候，这两个匿名方法将依次修改传递给它们的 out 类型的参数。在第 15 行的方法中，第一次修改 i 的值为 10，第二次为 20。这段代码的执行结果如下：

lambda 表达式的返回结果是 4000

前面介绍过，传统形式的使用 delegate 关键字的匿名方法可以不提供形参列表，如果没有形参列表，那么可以将它转换给一个带有形参的委托来调用。但是委托的形参不能是 out 类型的。如果委托带有形参，且不是 out 类型。委托的形参不能在匿名方法的方法体中访问。实例代码如下：

```

001    using System;

```

```

002 namespace AnonymousFunction
003 {
004     public delegate int D(int a);
005     class Program
006     {
007         private D myD;
008         public void AddList()
009         {
010             myD += delegate { a = 10; return a * 100; };
011         }
012         public void Invoke()
013         {
014             int i = 10;
015             int i1 = myD(i);
016             Console.WriteLine("lambda 表达式的返回结果是{0}", i1);
017         }
018         static void Main(string[] args)
019         {
020             Program p = new Program();
021             p.AddList();
022             p.Invoke();
023             Console.ReadKey();
024         }
025     }
026 }

```

第 4 行定义的委托类型带有一个 int 类型的值形参，第 10 行定义的匿名方法中，没有提供形参列表。但是在方法体中定义了对委托形参的访问。在方法 Invoke 中可以以传参的方式调用匿名方法，但是因为委托的形参在匿名方法中无法访问。所以编译器会报错，错误信息如下：

```

错误 1 当前上下文中不存在名称“a” H:\ 书 稿 new\ 书 稿 源 码 \ 第 十 七 章
\AnonymousFunction\Program.cs 10 31 AnonymousFunction
错误 2 当前上下文中不存在名称“a” H:\ 书 稿 new\ 书 稿 源 码 \ 第 十 七 章
\AnonymousFunction\Program.cs 10 46 AnonymousFunction

```

在匿名方法中还可以传入类的实例，从而在匿名方法体中访问类的成员。但是要注意委托类型和类类型之间的可访问性约束。看下面的代码实例：

```

001 using System;
002 namespace AnonymousFunction
003 {
004     public delegate void D(T t);
005     class T
006     {
007         private int a;
008         public T(int a)
009         {
010             this.a = a;

```

```

011     }
012     public void Show()
013     {
014         Console.WriteLine("字段 a 的值是: {0}", a);
015     }
016 }
017 class Program
018 {
019     private D myD;
020     public void AddList()
021     {
022         myD += delegate(T t) { t.Show(); };
023         myD += (T t) => { t.Show(); };
024     }
025     public void Invoke()
026     {
027         T myt = new T(100);
028         myD(myt);
029     }
030     static void Main(string[] args)
031     {
032         Program p = new Program();
033         p.AddList();
034         p.Invoke();
035         Console.ReadKey();
036     }
037 }
038 }

```

代码的第 4 行和第 5 行分别是一个委托类型和一个类的定义，委托的可访问性为 public，类的可访问性为 internal。在第 22 行和第 23 行，以两种语法定义了两个匿名方法，在方法体中调用类的成员方法。但是这段代码在编译的时候，编译器会报错，错误信息如下：

错误 1 可访问性不一致：参数类型“AnonymousFunction.T”比委托“AnonymousFunction.D”的可访问性低 H:\书稿 new\书稿源码\第十七章\AnonymousFunction\Program.cs 4 26

AnonymousFunction

导致这个错误的原因是，因为委托类型的可访问性无限制，它带有一个类型 T 的参数。而类 T 的可访问性为 internal，它只能在本程序集中访问。因此它们之间的访问性出现了冲突，所以会报告这样的错误。如果将委托的可访问性修改为 internal。代码的执行结果如下：

字段 a 的值是: 100

字段 a 的值是: 100

如果在一个机构的内部使用匿名方法，需要注意在匿名方法体内不能使用 this 引用。如果在匿名方法体内引用 this，则编译器会报错。代码实例如下：

```

001 using System;
002 namespace AnonymousFunction
003 {

```

```

004     delegate void D();
005     struct T
006     {
007         private int a;
008         private D myD;
009         public void AddList()
010         {
011             myD += () => { this.Show(); };
012             myD();
013         }
014         public void Show()
015         {
016             Console.WriteLine("字段 a 的值是: {0}", a);
017         }
018     }
019     class Program
020     {
021         static void Main(string[] args)
022         {
023             T t = new T();
024             t.AddList();
025             Console.ReadKey();
026         }
027     }
028 }

```

这段代码中的委托类型是一个无参数，无返回值类型。在第 5 行定义的结构中的实例方法 AddList 中，为委托字段添加一个匿名方法。在方法体中，使用 this 引用调用了结构的实例方法 Show。在第 12 行调用委托。但是，在代码编译时，编译器会报错，错误信息如下：

```

错误 1    运算符“+=”无法应用于“AnonymousFunction.D”和“lambda 表达式”类型的操作数    E:\
书稿 new\书稿源码\第十七章\AnonymousFunction\Program.cs 11 13 AnonymousFunction
错误 2    结构内部的匿名方法、lambda 表达式和查询表达式无法访问“this”的实例成员。请考虑将
“this”复制到匿名方法、lambda 表达式或查询表达式外部的某个局部变量并改用该局部变量。    E:\
书稿 new\书稿源码\第十七章\AnonymousFunction\Program.cs 11 28 AnonymousFunction

```

如果将结构改为类，则不会有上述问题。修改后的代码如下：

```

001     using System;
002     namespace AnonymousFunction
003     {
004         delegate void D();
005         class T
006         {
007             private int a;
008             private D myD;
009             public void AddList()
010             {

```

```

011         this.a = 100;
012         myD += () => { this.Show(); };
013         myD();
014     }
015     public void Show()
016     {
017         Console.WriteLine("字段 a 的值是: {0}", a);
018     }
019 }
020 class Program
021 {
022     static void Main(string[] args)
023     {
024         T t = new T();
025         t.AddList();
026         Console.ReadKey();
027     }
028 }
029 }

```

在类中的实例方法 AddList 中，为字段 a 作了赋值操作。代码可以正常执行，执行结果如下：

```

字段 a 的值是: 100

```

17.5.1 外层变量

匿名方法的外层变量就是范围包含 lambda 表达式或匿名方法的任何局部变量、值参数或参数数组。实例代码如下：

```

001 using System;
002 namespace AnonymousFunction
003 {
004     delegate void D();
005     class T
006     {
007         private int a;
008         private D myD;
009         private int[] myArray;
010         public T(int a)
011         {
012             this.a = a;
013             myArray = new int[20];
014             for (int i = 0; i < 20; i++)
015             {
016                 myArray[i] = i;
017             }
018         }
019         public void AddList()
020         {

```



```

021         int i = 30;
022         myD += () => {
023             i++;
024             this.a = 200;
025             this.Show();
026             Console.WriteLine("数组中的所有值为: ");
027             foreach (int il in myArray)
028             {
029                 Console.Write(il + " ");
030             }
031         };
032         myD();
033     }
034     public void Show()
035     {
036         Console.WriteLine("字段 a 的值是: {0}", a);
037     }
038 }
039 class Program
040 {
041     static void Main(string[] args)
042     {
043         T t = new T(100);
044         t.AddList();
045         Console.ReadKey();
046     }
047 }
048 }

```

在这段代码中，类 T 中定义了数组字段，在包含匿名方法的实例方法中定义了一个局部变量 i。在匿名方法中，对于局部变量 i，类的实例的 this 引用和类的数组字段，以及类的实例方法，都可以访问和引用。类的实例的 this 引用对于匿名方法来说，是一种值参数。这段代码的执行结果如下：

字段 a 的值是: 200

数组中的所有值为:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

外层变量中不包含引用类型和输出类型的参数，这样的例子前面作过演示了。下面再看参数数组的演示实例：

```

001 using System;
002 namespace AnonymousFunction
003 {
004     delegate void D();
005     class T
006     {
007         private int a;
008         private D myD;

```

```
009     private int[] myArray;
010     public T(int a)
011     {
012         this.a = a;
013         myArray = new int[20];
014         for (int i = 0; i < 20; i++)
015         {
016             myArray[i] = i;
017         }
018     }
019     public void AddList(int i, params string[] s)
020     {
021         myD += () => {
022             i++;
023             this.a = 200;
024             this.Show();
025             Console.WriteLine("数组中的所有值为: ");
026             foreach (int il in myArray)
027             {
028                 Console.Write(il + " ");
029             }
030             Console.WriteLine();
031             Console.WriteLine("外层变量中的参数数组的值包含: ");
032             foreach (string s0 in s)
033             {
034                 Console.WriteLine(s0);
035             }
036         };
037         myD();
038     }
039     public void Show()
040     {
041         Console.WriteLine("字段 a 的值是: {0}", a);
042     }
043 }
044 class Program
045 {
046     static void Main(string[] args)
047     {
048         T t = new T(100);
049         t.AddList(10, "好好学习, ", "天天向上");
050         Console.ReadKey();
051     }
052 }
```

```
053 }
```

在实例方法 AddList 中定义了两个形参，一个是普通的值参数；另一个是参数数组。在匿名方法中，第 32 行可以引用这个参数数组，打印它的全部值。在第 49 行调用这个方法时，可以给它传入任意个数的字符串。这段代码的执行结果如下：

字段 a 的值是：200

数组中的所有值为：

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

外层变量中的参数数组的值包含：

好好学习，

天天向上

17.5.2 外层变量的捕获

当外层变量被匿名方法引用时，则称外层变量被匿名方法捕获。这时，这个变量的生存期就会发生变化。一般来说，局部变量在其关联的代码块执行结束时，就会消失。但是，被捕获的变量的生存期将延长至引用匿名方法的委托被垃圾回收时为止。下面是代码实例：

```
001 using System;
002 namespace AnonymousFunction
003 {
004     delegate int D();
005     class T
006     {
007         public static D Show()
008         {
009             int i = 0;
010             D myD = () => { return (i += 10); };
011             return myD;
012         }
013     }
014     class Program
015     {
016         static void Main(string[] args)
017         {
018             D d = T.Show();
019             Console.WriteLine(d());
020             Console.WriteLine(d());
021             Console.WriteLine(d());
022             Console.ReadKey();
023         }
024     }
025 }
```

代码的第 4 行定义了一个委托类型，它没有形参，返回一个 int 类型的值。在类 T 中定义了一个静态方法 Show。这个方法返回一个委托类型。在方法体中首相定义了一个局部变量 i。然后定义了一个委托类型的变量，将匿名方法转换赋值给它。匿名方法中对局部变量 i 进行操作，然后返回其值。最后方法返回委托类型的变量。正常情况下，当方法 Show 执行完毕，变量 i 就会因为超出其作用域而消失。但是这个静态方法返回了在其中定义的委托变量，可以说，方法执行完毕后，委托并没有消失。因此，这个被匿名方法

捕获的局部变量也不会消亡，它会继续存储着其值。在第 18 行定义了委托类型的变量来获得这个返回的委托类型，然后用它三次调用委托。这时可以看到返回的 `i` 的值一直在发生着变化。`i` 并没有因为方法执行完毕而消失。代码的执行结果如下：

```
10
20
30
```

对于不安全代码来说，局部变量 `i` 不再是固定变量，它被视作可移动变量。

17.5.3 局部变量的实例化与捕获

当定义一个局部变量的时候，就可以说，这个局部变量被实例化了。如果这个局部变量没有超出其作用域，每次再对它进行赋值或更改其值，它被看做已经实例化了，每次的赋值不是再次实例化。看下面的代码：

```
001 using System;
002 namespace AnonymousFunction
003 {
004     class Program
005     {
006         static void Main(string[] args)
007         {
008             for (int i = 0; i < 3; i++)
009             {
010                 int a = i * 2;
011             }
012             Console.ReadKey();
013         }
014     }
015 }
```

这段代码中，第 8 行定义了一个 `for` 语句循环，共循环三次。每次循环都重新定义一个 `int` 类型的变量 `a`。并对它进行初始化。这时可以说，局部变量 `a` 实例化了 3 次，初始化了 3 次。现在修改代码如下：

```
001 using System;
002 namespace AnonymousFunction
003 {
004     class Program
005     {
006         static void Main(string[] args)
007         {
008             int a = 0;
009             for (int i = 0; i < 3; i++)
010             {
011                 a = i * 2;
012             }
013             Console.ReadKey();
014         }
015     }
016 }
```

这段代码中，将局部变量 `a` 的定义挪到了第 8 行，从第 9 行开始 `for` 语句的循环。这时可以说，局部变量 `a` 的实例化是 1 次，初始化了 4 次。

当局部变量是一个未捕获的变量时，它的生存期只要超过它的作用域时就结束了。但是，当它被匿名方法捕获时，它的生存期就会发生变化。实例代码如下：

```
001    using System;
002    namespace AnonymousFunction
003    {
004        delegate void D();
005        class Program
006        {
007            static D[] Test()
008            {
009                D[] myD = new D[3];
010                for (int i = 0; i < 3; i++)
011                {
012                    int a = i * 2;
013                    myD[i] = () => { a++; Console.WriteLine("本次捕获的值是: {0}", a); };
014                }
015                return myD;
016            }
017            static void Main(string[] args)
018            {
019                foreach (D d in Test())
020                {
021                    d();
022                }
023                Console.ReadKey();
024            }
025        }
026    }
```

在类 `Program` 中，定义了一个静态方法 `Test`。它返回一个委托类型的数组。在第 9 行首先定义了一个委托类型的数组 `myD`，然后进行了初始化。在 `for` 语句中，因为循环了 3 次，所以局部变量 `a` 会实例化 3 次，并初始化 3 次。然后依次对委托类型数组中的每个成员添加调用的匿名方法。最后返回委托类型的数组变量。

在 `Main` 方法中，使用 `foreach` 语句调用了 `Test` 方法，然后针对委托类型数组中的每个元素，依次执行委托。在执行委托前，因为捕获的 3 次实例化的变量值分别为 0、2 和 4，这时每个匿名方法捕获的局部变量的值经过自加后，都会再打印出来，分别为 1、3 和 5。可以观察到，每个捕获的局部变量的值都是不同的。代码的执行结果如下：

```
本次捕获的值是: 1
本次捕获的值是: 3
本次捕获的值是: 5
```

如果将这段代码中的局部变量 `a` 转移到循环语句外面，则情况又有所不同。更改后的代码如下：

```
001    using System;
002    namespace AnonymousFunction
```

```

003  {
004      delegate void D();
005      class Program
006      {
007          static D[] Test()
008          {
009              D[] myD = new D[3];
010              int a;
011              for (int i = 0; i < 3; i++)
012              {
013                  a = i * 2;
014                  myD[i] = () => { a++; Console.WriteLine("本次捕获的值是: {0}", a); };
015              }
016              return myD;
017          }
018          static void Main(string[] args)
019          {
020              foreach (D d in Test())
021              {
022                  d();
023              }
024              Console.ReadKey();
025          }
026      }
027  }

```

因为局部变量 `a` 在第 10 行定义，而它在循环的外面，所以 `a` 只实例化一次，但进行了 3 次初始化。这时，当执行 `Test` 方法的时候，数组 `myD` 的每个元素都被添加了一个匿名方法。因为局部变量只做了一次实例化，所以每个匿名方法捕获的外部变量都是一个。所以当 3 次执行第 22 行的委托调用时，都是对同一个被捕获的局部变量 `a` 进行操作。进行调用前，局部变量储存的值是最后一次初始化的值 4，所以每次调用委托打印出来的自增值分别是 5、6、7。代码的执行结果如下：

```

本次捕获的值是：5
本次捕获的值是：6
本次捕获的值是：7

```

如果将这段代码再进行更改，改为捕获 `for` 语句中的迭代变量 `i`，那么结果会如何呢？下面是修改后的代码：

```

001  using System;
002  namespace AnonymousFunction
003  {
004      delegate void D();
005      class Program
006      {
007          static D[] Test()
008          {
009              D[] myD = new D[3];

```

```

010         for (int i = 0; i < 3; i++)
011         {
012             myD[i] = () => { i++; Console.WriteLine("本次捕获的值是: {0}", i); };
013         }
014         return myD;
015     }
016     static void Main(string[] args)
017     {
018         foreach (D d in Test())
019         {
020             d();
021         }
022         Console.ReadKey();
023     }
024 }
025 }

```

在 for 循环语句中，局部变量 i 也是只实例化 1 次，但是进行了自增操作。下面来分析过程，因为只实例化 1 次，所以每个匿名方法捕获的变量都是一个。这一点现在可以确定。在进行循环时，for 语句的执行过程是这样的，首先 i 赋值为 0，接下来进行判断，判断其是否小于 3，如果符合，则执行语句块中的语句。等语句块中的语句执行完毕，再进行自增操作。所以，最后一次循环之后，i 还是要再自增一次的。最后的值就为 3。当第 20 行开始执行委托的时候，因为捕获的变量为 3，所以再经过匿名方法中的自增之后，打印出来的值就是 4、5 和 6。这段代码的执行结果如下：

```

本次捕获的值是：4
本次捕获的值是：5
本次捕获的值是：6

```

每个匿名方法还可以捕获同一个实例和不同的实例，实例代码如下：

```

001     using System;
002     namespace AnonymousFunction
003     {
004         delegate void D();
005         class Program
006         {
007             static D[] Test()
008             {
009                 D[] myD = new D[3];
010                 int a = 0;
011                 for (int i = 0; i < 3; i++)
012                 {
013                     int b = 0;
014                     myD[i] = () => { a++; b++; Console.WriteLine("a 的值是: {0}, b 的值是: {1}", a, b); };
015                 }
016                 return myD;
017             }

```

```

018         static void Main(string[] args)
019         {
020             foreach (D d in Test())
021             {
022                 d();
023             }
024             Console.ReadKey();
025         }
026     }
027 }

```

在这段代码中，局部变量 `a` 只进行了一次实例化，而局部变量 `b` 进行了三次实例化。所以匿名方法捕获了相同的实例 `a`，捕获了不同的实例 `b`。在第 22 行调用的时候，因为 `a` 的值捕获时为 0，所以每次打印都进行自增为 1、2 和 3。而对于局部变量 `b` 来说，每次捕获的实例初始都为 0，所以每个匿名方法都从 0 开始自增，所以结果都为 1。代码的执行结果如下：

```

a 的值是: 1,b 的值是: 1
a 的值是: 2,b 的值是: 1
a 的值是: 3,b 的值是: 1

```

两个不同的匿名方法还可以捕获同一个实例，实例代码如下：

```

001     using System;
002     namespace AnonymousFunction
003     {
004         delegate int D();
005         delegate void D1(int a);
006         class Program
007         {
008             static void Main(string[] args)
009             {
010                 int a = 0;
011                 D1 myD1 = (int i) => { a = i; };
012                 D myD = () => { return a; };
013                 myD1(100);
014                 Console.WriteLine("a 的值是: {0}",myD());
015                 myD1(200);
016                 Console.WriteLine("a 的值是: {0}", myD());
017                 Console.ReadKey();
018             }
019         }
020     }

```

在这段代码中，使用两个匿名方法捕获同一个局部变量实现了两个匿名方法之间的通信。每次一个匿名方法对局部变量进行赋值后，另一个匿名方法都可以把它再读出来。这段代码的执行结果如下：

```

a 的值是: 100
a 的值是: 200

```

17.6 匿名方法中的语句

在使用匿名方法的过程中，在匿名方法的内部不能使用 `goto`、`break` 和 `continue` 语句将代码逻辑跳转

到匿名方法外部；同理，在匿名方法外部也不能使用 goto、break 和 continue 语句将代码逻辑跳转到匿名方法内部。实例代码如下：

```

001 using System;
002 namespace AnonymousFunction
003 {
004     delegate void D();
005     class Program
006     {
007         static void Main(string[] args)
008         {
009             D myD1 = () => {
010                 for (int i = 0; i < 10; i++)
011                 {
012                     if (i == 5)
013                         break;
014                 }
015                 Console.WriteLine("i 的值现在是 5。"); //此行代码依旧执行
016             };
017             D myD2 = () => {
018                 for (int i = 0; i < 20; i++)
019                 {
020                     if (i == 10)
021                         continue;
022                 }
023             };
024             for (int i = 0; i < 10; i++)
025             {
026                 if (i == 5)
027                 {
028                     myD2();
029                     Console.WriteLine("i 的值是 {0}", i); //此行代码依旧执行
030                 }
031             }
032             myD1();
033             Console.ReadKey();
034         }
035     }
036 }

```

在代码的第 9 行定义了一个匿名方法，在匿名方法中定义了一个 for 循环语句。当迭代变量 i 等于 5 的时候，调用 break 语句。但是这个 break 语句并不能把代码逻辑跳转到匿名方法之外。第 15 行代码依旧会执行。

第 17 行代码定义了一个匿名方法，在它里面定义了一个 for 语句，当变量 i 等于 10 的时候，调用 continue 语句。在 Main 方法中，第 24 行定义了一个 for 语句，当变量 i 等于 5 的时候，调用委托 myD2。

这时委托执行循环，将执行 continue 语句。但是这个语句并没有对外层的 Main 方法中的逻辑造成影响。第 29 行代码依旧会执行。这段代码的执行结果如下：

i 的值是 5

i 的值现在是 5。

接下来再来看 goto 语句和 return 语句，实例代码如下：

```
001 using System;
002 namespace AnonymousFunction
003 {
004     delegate void D();
005     class Program
006     {
007         static void Main(string[] args)
008         {
009             D myD1 = () => { goto label1;};
010             myD1();
011             Console.WriteLine("这是第 11 行代码");
012             Console.WriteLine("这是第 12 行代码");
013             label1:
014                 Console.WriteLine("代码现在执行到 label1 处");
015                 D myD2 = () => { return; };
016                 myD2();
017                 Console.WriteLine("这是第 17 行代码");
018                 Console.ReadKey();
019             }
020         }
021     }
```

在第 9 行代码中定义了一个匿名方法，方法体中定义了一个 goto 语句跳转到 label1 标签处。在第 15 行定义了一个匿名方法，方法体中只有一个 return 语句。在第 10 行执行 myD1 委托；在第 16 行执行 myD2 委托。当执行 myD1 的时候，代码逻辑并没有跳转到第 13 行。而是报错，错误信息如下：

```
错误 1    goto 语句范围内没有“label1”这样的标签  H:\书稿 new\书稿源码\第十七章
\AnonymousFunction\Program.cs    9    30    AnonymousFunction
```

这是因为匿名方法内的 goto 语句无法跳转到匿名方法外部。下面修改代码如下：

```
001 using System;
002 namespace AnonymousFunction
003 {
004     delegate void D();
005     class Program
006     {
007         static void Main(string[] args)
008         {
009             D myD1 = () => {};
010             myD1();
011             Console.WriteLine("这是第 11 行代码");
012             Console.WriteLine("这是第 12 行代码");
```

```

013         label1:
014             Console.WriteLine("代码现在执行到 label1 处");
015             D myD2 = () => { return; };
016             myD2();
017             Console.WriteLine("这是第 17 行代码");
018             Console.ReadKey();
019         }
020     }
021 }

```

第 9 行的代码中，匿名方法是一个空方法。因此第 10 行执行的时候将不会跳转。但是，第 16 行执行的时候，整个 Main 方法并没有结束，第 17 行代码仍旧会执行。代码的执行结果如下：

这是第 11 行代码

这是第 12 行代码

代码现在执行到 label1 处

这是第 17 行代码

这是因为，匿名方法中的 return 语句只能用于匿名方法内部的返回，而不能对外层的方法起作用。下面再次修改代码如下：

```

001 using System;
002 namespace AnonymousFunction
003 {
004     delegate void D();
005     class Program
006     {
007         static void Main(string[] args)
008         {
009             goto label1;
010             D myD1 = () => { label1:Console.WriteLine("Hello World"); };
011             Console.ReadKey();
012         }
013     }
014 }

```

在这段代码中，Main 方法中的第 9 行代码意图使用 goto 语句将代码逻辑跳转到第 10 行的匿名方法内部的标签处，执行那里的 WriteLine 方法。但是这时编译器会报错，错误信息如下：

```

错误 1    goto 语句范围内没有“label1”这样的标签    H:\书稿 new\书稿源码\第十七章
\AnonymousFunction\Program.cs    9    13    AnonymousFunction
警告 2    检测到无法访问的代码    H:\书稿 new\书稿源码\第十七章\AnonymousFunction\Program.cs
    10    13    AnonymousFunction
警告 3    这个标签尚未被引用 H:\书稿 new\书稿源码\第十七章\AnonymousFunction\Program.cs    10
    30    AnonymousFunction

```

这是因为，外部方法中的 goto 语句不允许将代码逻辑跳转到它内部的匿名方法内。

17.7 方法组转换

前面介绍过，匿名方法可以隐式转换为兼容的委托类型。然后通过委托来调用匿名方法。可以隐式转换为委托类型的还有方法组。什么是方法组呢？方法组就是一组重载的方法。将方法组转换为委托类型的时候，委托可以根据与它匹配的原则，调用方法组中正确的重载方法。下面是代码实例：

```
001 using System;
002 namespace AnonymousFunction
003 {
004     delegate void D();
005     delegate void D1(int a);
006     class Program
007     {
008         public void Show()
009         {
010             Console.WriteLine("无参 Show 方法");
011         }
012         public void Show(int a)
013         {
014             Console.WriteLine("参数为 int 的 Show 方法, 参数值为: {0}", a);
015         }
016         public void Show(float f)
017         {
018             Console.WriteLine("参数为 float 的 Show 方法, 参数值为: {0}", f);
019         }
020         public void InvokeDelegatel(D d)
021         {
022             d();
023         }
024         public void InvokeDelegate2(D1 d)
025         {
026             d(12);
027         }
028         static void Main(string[] args)
029         {
030             Program p = new Program();
031             p.InvokeDelegatel(new D(p.Show));
032             p.InvokeDelegate2(p.Show);
033             Console.ReadKey();
034         }
035     }
036 }
```

这段代码中定义了两个委托类型，一个带有参数，一个不带参数。在类 Program 中定义了 3 个重载方法 Show。在第 20 行定义的方法中，要求传入一个 D 类型的委托变量。第 24 行定义的方法中，要求传入一个 D1 类型的委托。在它们的方法体中，对传入的委托进行了调用。

第 31 行代码中调用 InvokeDelegatel 方法的时候，因为要求传入一个 D 类型的委托，所以使用了 new 关键字创建了 D 类型委托的实例，为委托 D 的构造方法传入方法名的时候，其实就是方法组转换。因为 Show 这个方法名代表了一组重载的方法，它包含了第 8 行、第 12 行和第 16 行定义的三个方法。

第 32 行中，为 InvokeDelegate2 方法传入委托的时候，更直接，直接就使用了方法名，它也是一种方法组转换。在第 22 行和第 26 行调用委托的时候，根据参数的不同，委托调用了正确的方法。

如果提供了方法组转换，但是实际定义的方法中没有和委托类型相匹配的方法，则编译器会报错。看下面的实例代码：

```

001 using System;
002 namespace AnonymousFunction
003 {
004     delegate void D();
005     delegate void D1(int a);
006     class Program
007     {
008         public void Show()
009         {
010             Console.WriteLine("无参 Show 方法");
011         }
012         public void Show(float f)
013         {
014             Console.WriteLine("参数为 float 的 Show 方法，参数值为: {0}", f);
015         }
016         public void InvokeDelegate1(D d)
017         {
018             d();
019         }
020         public void InvokeDelegate2(D1 d)
021         {
022             d(12);
023         }
024         static void Main(string[] args)
025         {
026             Program p = new Program();
027             p.InvokeDelegate1((D)p.Show);
028             p.InvokeDelegate2(p.Show);
029             Console.ReadKey();
030         }
031     }
032 }

```

在这段代码中去掉了和委托 D1 相匹配的带有 int 类型参数的 Show 方法。因此第 28 行代码就会出错。编译器提示如下的错误信息：

```

错误 1    “AnonymousFunction.Program.Show(float)” 的重载均与委托 “AnonymousFunction.D1” 不匹
配 H:\书稿 new\书稿源码\第十七章\AnonymousFunction\Program.cs 28 31 AnonymousFunction

```

在进行方法组转换的时候，还可以使用显式转换来进行。第 27 行代码演示的就是这一语法。对方法名直接使用显式转换，将其转换到目的委托类型，这在语法上是允许的。

17.8 外层变量的明确赋值

匿名方法在使用外层变量的时候，如果匿名方法没有为外层变量赋值，就进行引用，则编译器会报错。实例代码如下：

```

001 using System;

```

```

002 namespace AnonymousFunction
003 {
004     delegate void D();
005     class Program
006     {
007         static void Main(string[] args)
008         {
009             int a;
010             D d1 = () => { if (a > 3) { Console.WriteLine("a 的值大于 3"); } };
011             a = 5;
012             Console.ReadKey();
013         }
014     }
015 }

```

在使用一个变量的时候，变量必须先做初始化赋值。如果引用一个没有做初始化赋值的变量，编译器会报错。对于匿名方法也是一样，第 9 行代码定义了一个局部变量 a，在匿名方法中对它进行判断。因为 a 的赋值在匿名方法的定义之后，所以编译器会报错，错误信息如下：

错误 1 使用了未赋值的局部变量“a” H:\ 书 稿 new\ 书 稿 源 码 \ 第 十 七 章
\AnonymousFunction\Program.cs 10 32 AnonymousFunction

下面再看一个代码实例：

```

001 using System;
002 namespace AnonymousFunction
003 {
004     delegate void D();
005     class Program
006     {
007         static void Main(string[] args)
008         {
009             int a;
010             D d1 = () => { a = 5; if (a > 3) { Console.WriteLine("a 的值大于 3"); } };
011             d1();
012             Console.WriteLine("现在 a 的值是：{0}", a);
013             Console.ReadKey();
014         }
015     }
016 }

```

在这段代码中仍旧在第 9 行定义了一个变量，并且没有赋值。在第 10 行的匿名方法的定义中，对 a 进行了赋值。并且第 11 行调用了这个匿名方法。在第 12 行打印 a 的值。看起来好像没有什么问题，但是编译器依旧报错如下：

错误 1 使用了未赋值的局部变量“a” H:\ 书 稿 new\ 书 稿 源 码 \ 第 十 七 章
\AnonymousFunction\Program.cs 12 45 AnonymousFunction

从错误信息中可以看到，第 12 行代码的调用仍是使用了未赋值的局部变量。这是为什么呢？这是因为 C# 中对于一个变量是否已赋值的判断是基于静态流程分析。判断一个变量已经赋值，包括下面的几种情况：

- 在定义变量的同时，对它赋值。

- 在使用变量之前的任何可到达变量的路径上，包含以下内容，则称这个变量已经赋值：
 1. 变量作为左操作数的简单赋值操作。
 2. 变量已经作为 out 类型的形参进行了调用，因为 out 类型的形参在方法中是必须赋值的。
 3. 对变量使用了 new 形式的对象创建表达式 (包含简单类型的变量和引用类型的变量)。
 4. 对变量使用了初始值设定项。

第十八章 泛型

在 C# 中，泛型的作用就是在同一份代码上可以使用多种类型。泛型使用类型参数。使用类型参数的泛型类和方法，可以将具体类型的指定推迟到客户端代码的定义中。这样，泛型类和方法就纯粹的变成了可以适应多种类型的类和方法的模板。因此使用泛型编程就可以最大限度地重用代码、保护类型的安全以及提高性能。

18.1 泛型类的定义

下面创建一个简单的泛型类来介绍泛型的用法。实例代码如下：

```
001     using System;
002     namespace Generic1
003     {
004         class Test<T>
005         {
006             T a;
007             T b;
008             public Test(T t1, T t2)
009             {
010                 this.a = t1;
011                 this.b = t2;
012             }
013             public void Show()
014             {
015                 Console.WriteLine("字段 a 的值是: {0}, 字段 b 的值是: {1}", a, b);
016             }
017         }
018         class Program
019         {
020             static void Main(string[] args)
021             {
022                 Test<int> t = new Test<int>(100, 200);
023                 t.Show();
024                 Console.ReadKey();
025             }
026         }
027     }
```

在这段代码的第 4 行定义了一个泛型类型，泛型类型的定义方式是，class 关键字表示这是一个类类型。class 关键字后是泛型类的名称，类名后紧跟着一对左右尖括号。在尖括号中是泛型类的类型形参。类型形参的命名最好都以 T 字母开头，除非单个的 T 能明了地表示其含义，否则应该后跟表示这个形参用途的字母组合。这里的 T 就是一个通用的类型。

第 6 行代码和第 7 行代码定义了两个 T 类型，也就是通用类型的字段。第 8 行代码中的构造方法中传入两个 T 类型的参数，用来对字段初始化。第 13 行的实例方法打印字段的值。可以看到，泛型类就好像一个蓝图一样，可以用它构造不同参数类型的类。现在定义的这个泛型类也叫做未绑定类型，因为还没有为它指定它的类型形参的具体类型。

再来看类 Program 中的代码，类 Program 中的代码也可以叫做客户端代码，它具体地使用了一个泛型

类。在使用泛型类创建类实例的时候，需要像第 22 行代码那样为泛型类传入类型实参，也就是原来在泛型类定义中的那个 T 参数。在定义时，它叫做类型形参；在具体指定类型的时候，它叫做类型实参。第 22 行代码就传入了一个 int 类型的类型实参。在调用构造方法的时候，依然需要在类名后加一对尖括号，在尖括号中写入类型实参。然后后跟构造方法的形参列表。这里传入的 int 类型就是类型实参，它将泛型专有化了。好像就是根据蓝图造出了一个房子一样，这个类型参数为 int 的类 Test 就叫做构造类型。它也是一个绑定类型。

使用泛型类的方式就是这样，包括调用 .Net 类库中的现有泛型类来说，例如一些集合类，都是使用这样在尖括号中指定使用类型的方式。每个指定特定类型实参而使用的类型都叫做构造类型或绑定类型。这段代码的执行结果如下：

字段 a 的值是：100, 字段 b 的值是：200

可以看到，传入类型实参之后，类的使用和普通的非泛型类的使用没有什么两样。要想具体的使用泛型类，那么只有使用绑定类型，而未绑定的泛型，只能在 typeof 运算符中使用。实例代码如下：

```

001    using System;
002    namespace Generic1
003    {
004        class Test<T>
005        {
006            T a;
007            T b;
008            public Test(T t1, T t2)
009            {
010                this.a = t1;
011                this.b = t2;
012            }
013            public void Show()
014            {
015                Console.WriteLine("字段 a 的值是：{0}, 字段 b 的值是：{1}", a, b);
016            }
017        }
018        class Program
019        {
020            static void Main(string[] args)
021            {
022                Type t = typeof(Test<>);
023                Console.WriteLine(t.FullName);
024                Console.WriteLine("Test<>是否是一个泛型类：{0}", t.IsGenericType);
025                Type t1 = typeof(Test<int>);
026                Console.WriteLine(t1.FullName);
027                Console.WriteLine("Test<>是否是一个泛型类：{0}", t1.IsGenericType);
028                Console.ReadKey();
029            }
030        }
031    }

```

在代码的第 22 行定义了一个 Type 类型的引用 t 来获得泛型类型的类型信息。使用 typeof 关键字来取

得泛型类型的信息。第 23 行调用 `Type` 类型的属性 `FullName` 打印输出这些信息。第 24 行调用 `IsGenericType` 属性来获得这个类是否是一个泛型类，它返回一个 `bool` 值。第 25 行为泛型类指定类型实参，然后再次使用 `typeof` 关键字获得信息。这段代码的执行结果如下：

```
Generic1.Test`1
Test<>是否是一个泛型类: True
Generic1.Test`1[[System.Int32, mscorlib, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089]]
Test<>是否是一个泛型类: True
```

从代码的执行结果可以观察到，`typeof(Test<>)` 的结果不依赖于类型实参，而 `typeof(Test<int>)` 的结果则依赖类型实参。

介绍完泛型类的定义之后，现在来介绍一下绑定类型和未绑定类型。绑定类型包括非泛型类型和构造类型；未绑定类型包括泛型和非泛型类型。未绑定的泛型类型本身不是一种类型，它不能用作变量、参数或返回值的类型。它也不能用作其它类的基类。前面介绍过，引用未绑定泛型类型的唯一一种方法就是 `typeof` 表达式。

18.2 类型形参

类型形参是一个标识符，它在为构造类型提供类型实参的时候，代表一个占位符。类型形参是一个占位符，而类型实参是一个实际类型，它在创建构造类型的时候替换掉类型形参。每个类、结构或接口在定义的时候创建一个新的声明空间，类型形参不能与该类的其它类型形参或成员同名。也不能与泛型类同名。下面是代码实例：

```
001 using System;
002 namespace Generic1
003 {
004     class Test<T, T, Test>
005     {
006         T a;
007         T b;
008         public void T()
009         { }
010         public Test(T t1, T t2)
011         {
012             this.a = t1;
013             this.b = t2;
014         }
015         public void Show()
016         {
017             Console.WriteLine("字段 a 的值是: {0}, 字段 b 的值是: {1}", a, b);
018         }
019     }
020     class Program
021     {
022         static void Main(string[] args)
023         {
024             Console.ReadKey();
025         }
026     }
027 }
```

```
026     }
027 }
```

在这段代码中，类型形参的名字重复了，而且另一个类型形参和类名同名了。这时，编译器会报告如下的错误：

```
错误 1    重复的类型参数“T”      H:\书稿 new\书稿源码\第十八章\Generic1\Program.cs    4    18
          Generic1
错误 2    类型参数“Test”与包含类型或方法同名 H:\书稿 new\书稿源码\第十八章
          \Generic1\Program.cs    4    20    Generic1
```

如果泛型类中的类型形参还没有被类型实参替换，也就是说，类型是一个未绑定类型。这时，不可以使用 using 语句为它定义别名。实例代码如下：

```
001 using System;
002 namespace Generic1
003 {
004     class Test<T1,T2>
005     {
006         T1 a;
007         T2 b;
008         public Test(T1 t1,T2 t2)
009         {
010             this.a = t1;
011             this.b = t2;
012         }
013         public void Show()
014         {
015             Console.WriteLine("字段 a 的值是: {0}, 字段 b 的值是: {1}", a, b);
016         }
017     }
018 }
019 namespace Generic2
020 {
021     using myGeneric<T1,T2> = Generic1.Test<T1, T2>;
022     class Program
023     {
024         static void Main(string[] args)
025         {
026             myGeneric<int,int> t = new myGeneric(100, 200);
027             t.Show();
028             Console.ReadKey();
029         }
030     }
031 }
```

在第 21 行代码中使用 using 语句为未绑定类型 Test<T1,T2>定义了一个别名。在编译的时候，编译器会报告错误信息，信息如下：

```
错误 1    应输入 ; H:\书稿 new\书稿源码\第十八章\Generic1\Program.cs    21    20    Generic1
```

错误 2 未能找到类型或命名空间名称“myGeneric” (是否缺少 using 指令或程序集引用?) H:\书稿
new\书稿源码\第十八章\Generic1\Program.cs 21 11 Generic1

错误 3 未能找到类型或命名空间名称“myGeneric” (是否缺少 using 指令或程序集引用?) H:\书稿
new\书稿源码\第十八章\Generic1\Program.cs 26 13 Generic1

错误 4 未能找到类型或命名空间名称“myGeneric” (是否缺少 using 指令或程序集引用?) H:\书稿
new\书稿源码\第十八章\Generic1\Program.cs 26 40 Generic1

下面改正代码中的错误，修改后的代码如下：

```

001 using System;
002 namespace Generic1
003 {
004     class Test<T1,T2>
005     {
006         T1 a;
007         T2 b;
008         public Test(T1 t1,T2 t2)
009         {
010             this.a = t1;
011             this.b = t2;
012         }
013         public void Show()
014         {
015             Console.WriteLine("字段 a 的值是: {0}, 字段 b 的值是: {1}", a, b);
016         }
017     }
018 }
019 namespace Generic2
020 {
021     using myGeneric = Generic1.Test<int, float>;
022     class Program
023     {
024         static void Main(string[] args)
025         {
026             myGeneric t = new myGeneric(100, 200.0F);
027             t.Show();
028             Console.ReadKey();
029         }
030     }
031 }

```

这段代码的第 4 行定义了一个泛型类，它带有两个类型形参。在定义两个类型形参的时候，需要注意它们的名称不能重名。在复杂一些的代码中，最好使用 T 开头的表明形参用途的标识符名称。使用 T 开头是一个惯例，但不是必须的。在第 21 行使用类型实参替换了类型形参。这两个类型实参也不是一样的类型。第 26 行定义了一个构造类型的实例。第 27 行调用实例方法。这段代码的执行结果如下：

字段 a 的值是: 100, 字段 b 的值是: 200

18.3 顶级成员的可访问域

接触过泛型类之后，现在再介绍一下顶级成员的可访问域。如果成员不是在某个类型内部声明的，就称这个成员是顶级的。顶级成员的声明有两种，未绑定的和绑定的。未绑定类型包括非泛型类型和未绑定泛型类型；绑定类型包括非泛型类型和构造类型。顶级类型可以使用的可访问性修饰符只有两个，`public` 和 `internal`。

- 顶级未绑定类型如果使用了 `public` 修饰，则它的可访问域是本程序集的程序文本以及引用并且使用它的其它任何程序集的程序文本
- 顶级未绑定类型如果使用了 `internal` 修饰，则它的可访问域是本程序集的程序文本

下面看代码实例，首先新建一个类库，代码如下：

```
001 using System;
002 namespace AccessibilityDomainLib
003 {
004     public class A
005     {
006     }
007     public class B<T>
008     {
009         T a;
010         public T Get(T b)
011         {
012             a = b;
013             return a;
014         }
015     }
016     internal class C
017     {
018     }
019 }
```

在这段代码中，定义了三个顶级的类型，两个普通的类和一个未绑定的泛型类型。类 A 和泛型类型 B 都采用了 `public` 进行修饰，类 C 使用 `internal` 进行修饰。接下来再新建一个程序集引用它，代码如下：

```
001 using System;
002 using AccessibilityDomainLib
003 namespace AccessibilityDomainTest1
004 {
005     class Program
006     {
007         static void Main(string[] args)
008         {
009             A myA = new A();
010             B<int> B = new B<int>();
011             int a = B.Get(11);
012             //如果将类 C 的可访问性设置为 public，则在此可使用
013             //C myC = new C();
014         }
015     }
```

```
016    }
```

在这段代码中使用了类库中定义的类 A 和构造类型 B，但是如果 C 的可访问性为 internal，则它只能在其所在的程序集中访问，在此程序集中不能访问。除非将它的可访问性设置为 public。如果顶级成员是构造类型的话，它的可访问域就是它的未绑定泛型和传递给它的类型实参的可访问域的交集。下面是代码实例：

```
001    using System;
002    namespace AccessibilityDomain2
003    {
004        public class A<T1,T2>
005        {
006            T1 a;
007            T2 b;
008            public void Set(T1 a, T2 b)
009            {
010                this.a = a;
011                this.b = b;
012            }
013        }
014        internal class B
015        {
016        }
017        class Program
018        {
019            static void Main(string[] args)
020            {
021                B myB = new B();
022                A<int, B> myA = new A<int, B>();
023                myA.Set(11, myB);
024            }
025        }
026    }
```

在这段代码中，类 A 是一个未绑定的泛型类型，并且它的可访问性是 public 的，类 B 的可访问性是 internal 的，它只能在本程序集中访问。因此下面的 Main 方法中，构造类型传递了两个类型实参，一个为 int 类型，一个是类 B，因为 int 类型的可访问域无限制，类 B 的可访问域是本程序集，因此构造类型 A 的可访问域就是本程序集，接下来将类 Program 去掉，再新建一个程序集，并且引用本程序集，看是否能使用构造类型 A，代码如下：

```
001    using System;
002    using AccessibilityDomain2;
003    namespace AccessibilityDomain2Test
004    {
005        class Program
006        {
007            static void Main(string[] args)
008            {
009                A<int, AccessibilityDomain2.B> myA = new A<int, AccessibilityDomain2.B>();
```

```

010      }
011    }
012  }

```

虽然在开发环境的智能提示中可以看到类 A，但是当创建类的实例时，编译器会提示错误，错误信息如下：

错误 1 “AccessibilityDomain2.B” 不可访问，因为它受保护级别限制 H:\书稿new\书稿源码\第十八章\AccessibilityDomain2Test\Program.cs 9 41 AccessibilityDomain2Test

错误 2 “AccessibilityDomain2.A<int, AccessibilityDomain2.B>.A()” 不可访问，因为它受保护级别限制 H:\书稿new\书稿源码\第十八章\AccessibilityDomain2Test\Program.cs 9 50
AccessibilityDomain2Test

错误 3 “AccessibilityDomain2.B” 不可访问，因为它受保护级别限制 H:\书稿new\书稿源码\第十八章\AccessibilityDomain2Test\Program.cs 9 82 AccessibilityDomain2Test

18.4 泛型类型的继承

泛型类型之间可以进行继承，在构造类型时，基类的类型形参必须能够替换为类型实参。也就是说，构造类型时，基类不能还留有类型形参不能替换。基类在构造时，必须成为构造类型。下面是代码实例：

```

001 using System;
002 namespace Generic1
003 {
004     class Father<T1, T2>
005     {
006         private T1[] t1;
007         private T2 t2;
008         public Father(int a, T2 t)
009         {
010             t1 = new T1[a];
011             t2 = t;
012         }
013         public void Show()
014         {
015             Console.WriteLine(t2.ToString());
016             foreach (T1 t in t1)
017             {
018                 Console.WriteLine(t);
019             }
020         }
021         public T1 this[int index]
022         {
023             get
024             {
025                 return t1[index];
026             }
027             set
028             {
029                 t1[index] = value;
030             }
031         }
032     }
033 }

```

```

031     }
032 }
033 class Son<T1, T2>:Father<T1, T2>
034 {
035     public Son(int a, T2 t)
036         : base(a, t)
037     { }
038 }
039 class Program
040 {
041     static void Main(string[] args)
042     {
043         Son<int, string> s = new Son<int, string>(10, "Hello World");
044         for (int i = 0; i < 10; i++)
045         {
046             s[i] = i * 2 + 5;
047         }
048         s.Show();
049         Console.ReadKey();
050     }
051 }
052 }

```

这段代码首先在第 4 行定义了一个基类 Father，它带有两个类型形参 T1 和 T2。类中定义了两个字段，第一个是 T1 类型的数组；第二个是 T2 类型的字段变量。在第 8 行的构造方法中传入了一个 int 类型的参数对数组进行初始化。另一个参数用来初始化字段 t2。第 13 行的实例方法中打印字段 t2 的值，然后使用循环输出数组的值。第 21 行代码定义了一个索引器，索引器对数组 t1 进行存取。

第 33 行代码定义了一个派生类 Son 从基类派生。派生类也带有两个类型形参，注意在派生时，因为要使用派生类为基类传入类型实参，所以派生类的类型形参的名字必须和基类相同。

在 Main 方法中，第 43 行定义了一个 Son 构造类型的实例 s，然后进行了实例化。在构造泛型类的时候，传入的类型实参是 int 和 string。它也被传递给基类，对基类也进行了构造。这时，基类的类型实参也是 int 和 string。

第 44 行使用 for 语句调用了索引器的 set 访问器对类内的数组成员进行了赋值。第 48 行输出了类的数组成员的值和字段的值。这段代码的执行结果如下：

```
Hello World
```

```

5
7
9
11
13
15
17
19
21
23

```


如果在派生类由泛型类进行构造的时候，基类还留有类型形参没有指定类型实参，这时编译器会报错。也就是说，不允许未绑定的泛型类型充当类的基类。将前面的代码进行修改，代码如下：

```
001 using System;
002 namespace Generic1
003 {
004     class Father<T1, T2, T3>
005     {
006         private T1[] t1;
007         private T2 t2;
008         private T3 t3;
009         public Father(int a, T2 t)
010         {
011             t1 = new T1[a];
012             t2 = t;
013         }
014         public void Show()
015         {
016             Console.WriteLine(t2.ToString());
017             foreach (T1 t in t1)
018             {
019                 Console.WriteLine(t);
020             }
021         }
022         public T1 this[int index]
023         {
024             get
025             {
026                 return t1[index];
027             }
028             set
029             {
030                 t1[index] = value;
031             }
032         }
033     }
034     class Son<T1, T2>:Father<T1, T2, T3>
035     {
036         public Son(int a, T2 t)
037             : base(a, t)
038         { }
039     }
040     class Program
041     {
042         static void Main(string[] args)
```

```

043     {
044         Son<int, string> s = new Son<int, string>(10, "Hello World");
045         for (int i = 0; i < 10; i++)
046         {
047             s[i] = i * 2 + 5;
048         }
049         s.Show();
050         Console.ReadKey();
051     }
052 }
053 }

```

这段代码的第 4 行定义基类的时候，又定义了一个类型形参 T3。第 8 行增加了 T3 类型的字段。在派生类继承基类的时候，只对应传入了两个类型实参。而新增加的类型形参 T3 没有进行类型实参的替换。因此，编译器会报告错误。错误信息如下：

```

错误 1  未能找到类型或命名空间名称“T3” (是否缺少 using 指令或程序集引用?)    H:\书稿 new\
书稿源码\第十八章\Generic1\Program.cs 34    35    Generic1

```

如果在不去掉类型形参 T3 的情况下改正这个错误，可以在派生类继承基类的时候，为类型形参 T3 指定类型实参。在派生类继承基类的时候，也可以直接指定基类的类型实参。实例代码如下：

```

001 using System;
002 namespace Generic1
003 {
004     class Father<U1,U2,U3>
005     {
006         private U1[] t1;
007         private U2 t2;
008         private U3 t3;
009         public Father(int a,U2 t)
010         {
011             t1 = new U1[a];
012             t2 = t;
013             t3 = (U3)(object)100;
014         }
015         public virtual void Show()
016         {
017             Console.WriteLine(t2.ToString());
018             foreach (U1 t in t1)
019             {
020                 Console.WriteLine(t);
021             }
022         }
023         public U1 this[int index]
024         {
025             get
026             {

```

```
027         return t1[index];
028     }
029     set
030     {
031         t1[index] = value;
032     }
033 }
034 }
035 class Son<T1>:Father<int, string, int>
036 {
037     private T1 t1;
038     public Son(int a, string s, T1 t)
039         : base(a, s)
040     { t1 = t; }
041     public override void Show()
042     {
043         Console.WriteLine("派生类中的 Show 方法被调用, 字段 t1 的值是: {0}", t1);
044     }
045 }
046 class Program
047 {
048     static void Main(string[] args)
049     {
050         Son<int> s = new Son<int>(10, "Hello World", 200);
051         for (int i = 0; i < 10; i++)
052         {
053             s[i] = i * 2 + 5;
054         }
055         s.Show();
056         Father<int, string, int> f = s;
057         f.Show();
058         Console.ReadKey();
059     }
060 }
061 }
```

为了在派生类继承基类的时候, 直接指定基类的类型实参。在第 4 行代码中, 将基类中的类型形参更名为 U1、U2 和 U3。为什么要更改为这三个类型形参呢? 这是避免派生类和基类的类型形参互相影响。因为如果有和派生类的类型形参名相同的类型形参。它们之间就会有对应关系。这样就不能将基类中的类型形参指定了实参, 而派生类中的类型形参却在运行时指定。在第 13 行代码中演示了一种特殊的语法。如果不在构造方法的方法参数中传入 U3 类型的参数, 那么可以采用两次显式转换的方法将一个 int 类型的文本转换成 U3 类型。因为一个 int 类型是不可以通过显式转换直接转换成 U3 类型的。但是因为 C# 中的任何类型均从 object 类型派生。所以可以先将 int 类型的值转成 object 类型, 再显式转换成 U3 类型。这样就可以完成对 U3 字段的赋值。但是, 这种转换方法很危险, 需要程序员主观知道转换是会成功的才行。这里的 int 类型的文本是有大小的, 可以知道转换会成功。

第 35 行定义派生类从基类继承的时候，对于基类直接指定了类型实参。第 38 行的构造方法采用传递类型形参参数的方法对字段 t1 进行赋值。因为基类的泛型已经指定了类型实参，所以派生类的构造方法在使用 base 关键字调用基类的构造方法的时候，需要为其传入正确类型的参数。因为指定了基类的类型实参，所以可以定义一个基类的构造类型的变量，将派生类的实例赋值给它以实现多态。需要注意，定义基类的构造类型的变量的时候，指定的类型实参要符合基类已经指定的类型实参。这段代码的执行结果如下：

派生类中的 Show 方法被调用，字段 t1 的值是：200

派生类中的 Show 方法被调用，字段 t1 的值是：200

还有一点需要注意，在从基类派生子类的时候，还需要注意可访问性的问题。基类的访问权限必须至少和派生类的访问权限相同。否则会导致编译错误。

18.5 泛型方法

泛型方法可以将类型参数用作返回值的类型或者某个形参的类型。泛型类和非泛型类中都可以定义泛型方法。但是需要注意，并不是方法只要有类型形参，就说这个方法是泛型方法。只有方法有自己的类型形参列表时，这个方法才是泛型方法。实例代码如下：

```
001     using System;
002     namespace Generic2
003     {
004         class Program
005         {
006             private T Test<T>(T t1, T t2)
007             {
008                 int t = (int)(object)t1 + (int)(object)t2;
009                 return (T)(object)t;
010             }
011             private void Swap<T>(ref T t1, ref T t2)
012             {
013                 T temp;
014                 temp = t1;
015                 t1 = t2;
016                 t2 = temp;
017             }
018             static void Main(string[] args)
019             {
020                 Program p = new Program();
021                 int f1 = 3;
022                 int f2 = 33;
023                 float f3 = p.Test<int>(f1, f2);
024                 int i1 = 100;
025                 int i2 = 200;
026                 int i3 = p.Test<int>(i1, i2);
027                 Console.WriteLine("f3 的值是：{0}, i3 的值是：{1}", f3, i3);
028                 p.Swap<int>(ref i1, ref i2);
029                 Console.WriteLine("i1 的值是：{0}, i2 的值是：{1}", i1, i2);
030                 Console.WriteLine("省略类型实参，依靠编译器的推断来调用泛型方法：");
031                 p.Swap(ref i1, ref i2);
```

```

032         Console.WriteLine("i1 的值是: {0}, i2 的值是: {1}", i1, i2);
033         Console.ReadKey();
034     }
035 }
036 }

```

在这段代码中定义了两个泛型方法，是否是泛型方法，要看其是否有类型形参列表，即方法名称后面是否有一个尖括号，里面是类型形参列表。可以看到，这两个方法都是泛型方法。在第 6 行的泛型方法 Test 中，类型形参的类型只有一种 T，方法带有两个 T 类型的参数，返回值的类型也为 T 类型。在第 8 行，明知两个操作数可以转换成 int 类型的情况下，使用强制转换对它们进行了相加操作，将结果返回。第 11 行代码中的 Swap 方法是交换两个参数的值，在第 13 行定义了一个临时的 T 类型的变量，然后让它暂时存储参数 t1 的值，然后交换两个参数的值，最后将存储的值赋值给 t2 完成两个参数值的交换。因为传入参数的时候，是以引用参数的方式传入的参数，所以原始的变量值也会发生交换。

因为这两个方法都是类的实例方法，不是静态方法。所以第 20 行代码定义了一个类的实例，使用它来调用这两个方法。在调用泛型方法的时候，需要为其指定类型实参。但是注意第 31 行代码，编译器可以根据为方法实际传入的参数类型，进行推断来得出类型实参的类型。所以，直接调用泛型方法而不指定类型实参也是可行的。这段代码的执行结果如下：

```

f3 的值是: 36, i3 的值是: 300
i1 的值是: 200, i2 的值是: 100
省略类型实参，依靠编译器的推断来调用泛型方法:
i1 的值是: 100, i2 的值是: 200

```

下面再来修改代码的第 21、22 和 23 行，如下：

```

float f1 = 3.33F;
float f2 = 33.33F;
float f3 = p.Test<float>(f1, f2);

```

这时编译代码的时候，编译器会抛出异常信息，如下：

```

未处理 InvalidCastException
指定的转换无效

```

问题出在第 8 行代码上，当调用第 23 行代码的时候，因为传入的类型实参为 float 类型，float 类型的取值是可以为无穷大的，将它显式转换为 int 类型就会抛出转换异常。因此在使用这样的强制转换语法的时候一定要小心处理，将各种可能的问题都考虑到。

18.5.1 泛型方法的重载

当依靠编译器的类型推断来判断类型实参的时候，泛型方法的返回值以及后面要介绍的约束不能作为推断的依据。编译器仅仅能够依靠为泛型方法传入的参数类型来推断类型实参类型。下面看代码实例：

```

001     using System;
002     namespace Generic2
003     {
004         class Program
005         {
006             private T Swap<T>(ref int t1, ref int t2)
007             {
008                 T temp;
009                 temp = (T)(object)t1;
010                 t1 = t2;
011                 t2 = (int)(object)temp;

```

```

012         return temp;
013     }
014     static void Main(string[] args)
015     {
016         Program p = new Program();
017         int i1 = 100;
018         int i2 = 200;
019         Console.WriteLine("省略类型实参，依靠编译器的推断来调用泛型方法：");
020         int i3 = p.Swap(ref i1, ref i2);
021         Console.WriteLine("i1 的值是：{0}, i2 的值是：{1}", i1, i2);
022         Console.ReadKey();
023     }
024 }
025 }

```

这段代码中对 Swap 方法做了修改，为它增加了一个返回值。方法的参数类型固定为 int 类型。在这种情况下，使用了两次强制转换的方法进行了方法内部的赋值操作。需要注意，在其中发生了装箱和拆箱转换。这会影响代码的执行效率。所以，泛型的使用也要依据具体的需要而使用。在第 20 行，依然没有指定类型实参，而是想要依靠编译器的推断来正确的调用方法，但是编译器是不允许使用方法的返回值作为推断依据的，编译器会报告错误，如下：

错误 1 无法从用法中推导出方法“Generic2.Program.Swap<T>(ref int, ref int)”的类型实参。请尝试显式指定类型实参。 H:\书稿 new\书稿源码\第十八章\Generic2\Generic2.cs 20 22 Generic2

使用不同的类型参数，泛型方法可以实现重载。下面是代码实例：

```

001 using System;
002 namespace Generic2
003 {
004     class Program
005     {
006         private void Test<T>(int a)
007         {
008             T t;
009             t = (T)(object)a;
010             Console.WriteLine("局部变量 t 的值是：{0}", t);
011         }
012         private void Test<T>()
013         {
014             T t;
015             t = (T)(object)100;
016             Console.WriteLine("局部变量 t 的值是：{0}", t);
017         }
018         private void Test<T, U>(int a, int b)
019         {
020             T t;
021             U u;
022             t = (T)(object)a;

```

```

023         u = (U)(object)b;
024         Console.WriteLine("局部变量 t 的值是: {0}", t);
025         Console.WriteLine("局部变量 u 的值是: {0}", u);
026     }
027     static void Main(string[] args)
028     {
029         Program p = new Program();
030         p.Test<int>();
031         p.Test<int>(20);
032         p.Test<int, int>(100, 200);
033         Console.ReadKey();
034     }
035 }
036 }

```

对于泛型方法来说，不同的类型形参个数，方法形参类型都可以构成重载。第 30 行代码匹配的是第 12 行代码的方法；第 31 行代码匹配的是第 6 行定义的方法；第 32 行代码匹配的是第 18 行定义的泛型方法。这段代码的执行结果如下：

```

局部变量 t 的值是: 100
局部变量 t 的值是: 20
局部变量 t 的值是: 100
局部变量 u 的值是: 200

```

在泛型类中定义泛型方法的时候，需要注意泛型类的类型形参名称不能和泛型方法的类型形参名称相同。实例代码如下：

```

001     using System;
002     namespace Generic3
003     {
004         class Test<T>
005         {
006             public void Show<T>()
007             {
008                 Console.WriteLine("Hello World");
009             }
010         }
011         class Program
012         {
013             static void Main(string[] args)
014             {
015                 Test<int> t = new Test<int>();
016                 t.Show<float>();
017                 Console.ReadKey();
018             }
019         }
020     }

```

在第 6 行定义类中的泛型方法时，定义的类型形参的名称和类的类型形参的名称相同，这时编译器会报告

一个警告信息，如下：

警告 1 类型参数“T”与外部类型“Generic3.Test<T>”中的类型参数同名 H:\书稿new\书稿源码\第十八章\Generic3\Program.cs 6 26 Generic3

如果要去掉这个警告信息，可以为泛型方法指定不同名称的类型形参。实例代码如下：

```
001 using System;
002 namespace Generic3
003 {
004     class Test<T1>
005     {
006         public void Show<T2>(T2 t2)
007         {
008             Console.WriteLine("Hello World");
009             Console.WriteLine("t2 的类型是: {0}", t2.GetType().ToString());
010         }
011     }
012     class Program
013     {
014         static void Main(string[] args)
015         {
016             Test<int> t = new Test<int>();
017             t.Show<float>(34.12F);
018             Console.ReadKey();
019         }
020     }
021 }
```

使用了不同的类型形参的名称，就可以为类的类型形参和泛型方法的类型形参指定不同的类型实参。这段代码的执行结果如下：

```
Hello World
t2 的类型是: System.Single
```

在泛型类中定义方法的时候，还要注意两个重载的方法，可能因为实际定义的类型实参造成方法完全相同的签名。实例代码如下：

```
001 using System;
002 namespace Generic3
003 {
004     class Test0
005     { }
006     class Test<T1,T2>
007     {
008         public void Show(T2 t2, T1 t1)
009         {
010             Console.WriteLine("Show(T2 t2, T1 t1)被调用");
011             Console.WriteLine("t1 的类型是: {0}", t1.GetType().ToString());
012             Console.WriteLine("t2 的类型是: {0}", t2.GetType().ToString());
013         }
014     }
015 }
```



```
014     public void Show(T1 t1, T2 t2)
015     {
016         Console.WriteLine("Show(T1 t1, T2 t2)被调用");
017         Console.WriteLine("t1 的类型是: {0}", t1.GetType().ToString());
018         Console.WriteLine("t2 的类型是: {0}", t2.GetType().ToString());
019     }
020     public void Show(Test0 t, T1 t1)
021     {
022         Console.WriteLine("Show(Test0 t, T1 t1)被调用");
023     }
024     public void Show(T1 t1, Test0 t)
025     {
026         Console.WriteLine("Show(T1 t1, Test0 t)被调用");
027     }
028     public void Show(ref T1 t1)
029     {
030         Console.WriteLine("Show(ref T1 t1)被调用");
031     }
032     public void Show(out T2 t2)
033     {
034         t2 = (T2)(object)100;
035         Console.WriteLine("Show(out T2 t2)被调用");
036     }
037 }
038 class Program
039 {
040     static void Main(string[] args)
041     {
042         Test0 t0 = new Test0();
043         Test<int, int> t = new Test<int, int>();
044         t.Show(100, 200);
045         Test<Test0, Test0> t1 = new Test<Test0, Test0>();
046         t1.Show(t0, t0);
047         Test<int, int> t2 = new Test<int, int>();
048         int a = 10;
049         t2.Show(ref a);
050         t2.Show(out a);
051         Console.ReadKey();
052     }
053 }
054 }
```

在泛型类中定义方法的时候，虽然在定义的时候，方法采用不同名称的类型形参好像实现了重载。但是在将类型实参传递进去的时候，就有可能造成两个方法的签名完全相同。例如第 8 行和第 14 行定义的两个重载方法，当类型实参相同时，例如都为 `int` 类型的时候，就会签名完全相同。第 20 行和第 24 行定义

的两个方法，当类型实参都为 Test0 的时候，就会形成完全相同的签名。第 28 行和第 32 行定义的两个重载方法，当类型实参相同的时候，它们依靠 ref 和 out 关键字是无法实现重载的。编译器报告的错误信息如下：

错误 1 在以下方法或属性之间的调用不明确：“Generic3.Test<T1,T2>.Show(T2, T1)” 和 “Generic3.Test<T1,T2>.Show(T1, T2)” H:\书稿 new\书稿源码\第十八章\Generic3\Program.cs 44

13 Generic3

错误 2 在以下方法或属性之间的调用不明确：“Generic3.Test<T1,T2>.Show(Generic3.Test0, T1)” 和 “Generic3.Test<T1,T2>.Show(T1, Generic3.Test0)” H:\书稿 new\书稿源码\第十八章\Generic3\Program.cs 46 13 Generic3

可以看到，虽然使用 ref 和 out 参数的两个方法可以正常调用，但是从重载的定义上来说，它们没有实现严格意义上的重载，因为它们只是参数传递上的差异，但是参数的类型在将类型实参传入时，是完全相同的。

在实现泛型方法的重载时，只有最明确的方法被调用。下面是代码实例：

```
001 using System;
002 namespace Generic3
003 {
004     class Test
005     {
006         public static void Show<T>()
007         {
008             Console.WriteLine("Show<T>() 被调用");
009         }
010         public static void Show<T>(T t)
011         {
012             Console.WriteLine("Show<T>(T t) 被调用");
013         }
014         public static void Show(int a)
015         {
016             Console.WriteLine("Show(int a) 被调用");
017         }
018         public static void Show<T>(int a)
019         {
020             Console.WriteLine("Show<T>(int a) 被调用");
021         }
022     }
023     class Program
024     {
025         static void Main(string[] args)
026         {
027             Test.Show<int>();
028             Test.Show(100);
029             Test.Show<int>(100);
030             Test.Show("Hello World");
031             Console.ReadKey();
```

```

032     }
033     }
034 }

```

这段代码的第 27 行指定了类型实参是 `int`，这时很明确，第 6 行定义的方法将被调用。第 28 行为方法传入一个 `int` 类型的实参，这时第 14 行代码将被调用。第 29 行定义类型实参为 `int`，传入参数是 `int` 类型，这时，第 18 行定义的方法被调用。第 30 行使用编译器推导的方式传入一个 `string` 类型的实参。第 10 行定义的方法将给调用，它的类型实参推导为 `string`。虽然，第 28 行代码的方法调用也可以推断得出。但是因为第 14 行的方法有明确定义，所以编译器不会调用第 10 行的方法定义。由此可见，编译器总会调用最明确的方法定义，只有在明确匹配都不符合的情况下，才调用泛型的方法匹配。

18.6 泛型的 default 关键字

在介绍泛型的 `default` 的时候，通过代码来介绍最为直接。下面是代码实例：

```

001 using System;
002 namespace Generic2
003 {
004     class Program
005     {
006         private void Test<T>(string a)
007         {
008             T t = default(T);
009             if (t is int)
010             {
011                 t = (T)(object)Convert.ToInt32(a);
012             }
013             if (t == null)
014             {
015                 t = (T)(object)a;
016             }
017             if (t is float)
018             {
019                 t = (T)(object)Convert.ToSingle(a);
020             }
021             Console.WriteLine("局部变量 t 的值是: {0}", t);
022         }
023         static void Main(string[] args)
024         {
025             Program p = new Program();
026             p.Test<int>("1000");
027             p.Test<string>("2000");
028             p.Test<float>("3000");
029             Console.ReadKey();
030         }
031     }
032 }

```

在泛型方法 `Test` 中，首先定义了一个类型形参类型的变量 `t`，因为泛型就好像是一个模板一样，如果要在

模板中写判断类的语句，就非常困难，因为不知道要传入的类型实参是什么类型的。假如没有第 8 行的 default 语句。那么在第 9 行判断 t 的类型的时候，就会出错，编译会提示如下的错误信息：

错误 1 使用了未赋值的局部变量“t” Generic2.cs 9 17 Generic2

这是因为变量没有赋值之前是无法直接引用的。但是，在模板中又不知道要传入的类型实参到底是什么类型。因此可以使用 default(T) 语句将变量设置为给它传入的那个类型的默认值。它是一种运行时语法，因为只有运行时才能知道类型实参是什么类型。default 语句后面的括号中就是泛型的类型形参的名称标识。

如果类型实参是数值类型，则 default 语句将变量设置为 0。如果类型实参是引用类型，则 default 语句将变量设置为 null。如果类型实参是结构，则 default 将返回初始化为零或 null 的每个结构成员。如果类型是可空类型，则 default 语句将变量设置为 null。

如果为泛型方法传入的类型实参是 int 类型，则 t 的值将为 0，第 9 行能够判断出它的类型是 int。于是使用 Convert 类型的 ToInt32 方法将字符串转化为 int 类型，再强制转换为 T 类型。如果传入的是引用类型，如第 27 行代码将类型实参定义为 string 类型，则 t 的默认值是 null。那么第 13 行就不能再使用 is 运算符，而是要使用“==”运算符判断变量是否为 null。这时，字符串可以直接使用显式转换的方法转换为 T 类型。如果类型实参是 float 类型，则 t 的默认值仍然为 0。这时第 17 行代码仍旧可以判断出它的类型是 float 类型，于是使用 Convert 类型的 ToSingle 方法将字符串类型转换为 float 类型，再强制转换为 T 类型。这样一来，泛型方法的模板就写好了。代码的执行结果如下：

局部变量 t 的值是：1000

局部变量 t 的值是：2000

局部变量 t 的值是：3000

18.7 泛型类的静态成员

泛型类就好比一个模板一样，它只有传入类型实参变成构造类型之后，它才能去创建实例。类的静态成员对于类的所有实例是共享的，它只有一份。但是，对于泛型类来说，情况就不是这样了。实例代码如下：

```
001 using System;
002 namespace Generic2
003 {
004     class Test<T>
005     {
006         public static int a;
007     }
008     class Program
009     {
010         static void Main(string[] args)
011         {
012             Test<int>.a = 100;
013             Test<float>.a = 200;
014             Console.WriteLine("Test<int>中，静态字段 a 的值是：{0}", Test<int>.a);
015             Console.WriteLine("Test<float>中，静态字段 a 的值是：{0}", Test<float>.a);
016             Console.WriteLine("Test<string>中，静态字段 a 的值是：{0}", Test<string>.a);
017             Console.ReadKey();
018         }
019     }
020 }
```

这段代码的执行结果如下：

Test<int>中，静态字段 a 的值是：100

Test<float>中，静态字段 a 的值是：200

Test<string>中，静态字段 a 的值是：0

可以看到，对于泛型类的每个构造类型来说，都有自己的一份静态成员的副本。在所有同一个泛型类的构造类型中，泛型类的静态成员并不共享。

18.8 泛型接口

使用泛型不只可以定义泛型类，也可以定义泛型接口。使用泛型定义的接口同样使用类型形参。下面是代码实例：

```
001 using System;
002 namespace Generic2
003 {
004     interface ITest<T>
005     {
006         void Show();
007         T A { get; set; }
008     }
009     class Test<T>:ITest<T>
010     {
011         private T t;
012         public void Show()
013         {
014             Console.WriteLine("字段 t 的值是：{0}", t);
015         }
016         public T A
017         {
018             get
019             {
020                 return t;
021             }
022             set
023             {
024                 t = value;
025             }
026         }
027         public Test(T t)
028         {
029             this.t = t;
030         }
031     }
032     class Program
033     {
034         static void Main(string[] args)
035         {
```

```

036         ITest<int> myITest = new Test<int>(100);
037         myITest.A = 2000;
038         myITest.Show();
039         Console.WriteLine("属性 A 的值是: {0}", myITest.A);
040         Console.ReadKey();
041     }
042 }
043 }

```

泛型接口的定义方法和普通的接口没有太大的区别，只不过它带有类型形参。具体的类型需要定义类的实例的时候，使用类型实参替换类型形参。这段代码定义的是一个泛型接口，然后一个泛型类实现这个泛型接口。第 4 行代码定义的泛型接口只有一个类型形参 T。在泛型接口中定义了一个普通的方法声明和一个泛型属性，属性的返回类型是 T。

第 9 行定义的泛型类实现了泛型接口，需要注意，泛型类也带有一个类型形参 T。泛型接口中的类型形参需要由泛型类的构造类型将类型实参指定给它。泛型类中的类型形参和泛型接口中的类型形参是对应的关系。在指定类型实参的时候，它们是同时被替换为类型实参的。在泛型类中，第 11 行代码定义了 T 类型的字段，第 12 行实现了接口中的普通方法声明。第 16 行实现了接口中的泛型属性。第 27 行是构造方法，在构造方法中对字段进行了赋值。

第 36 行定义了一个接口的构造类型变量指向了实现类的构造类型实例。为泛型接口和泛型类传入的类型实参是 int 类型。第 37 行代码调用了接口的属性的 set 访问器。它实际上实现了属性的多态。第 38 行调用了接口的 Show 方法，它发生的也是多态的方法调用。这段代码的执行结果如下：

```
字段 t 的值是: 2000
```

```
属性 A 的值是: 2000
```

18.8.1 非泛型类实现泛型接口

非泛型类实现泛型接口的时候，必须指定接口的类型实参。不可以非泛型类中包含接口中类型形参的成员。下面是代码实例：

```

001     using System;
002     namespace Generic2
003     {
004         interface ITest<T>
005         {
006             void Show();
007             T A { get; set; }
008         }
009         class Test:ITest<T>
010         {
011             private T t;
012             public void Show()
013             {
014                 Console.WriteLine("字段 t 的值是: {0}", t);
015             }
016             public T A
017             {
018                 get
019                 {

```

```

020         return t;
021     }
022     set
023     {
024         t = value;
025     }
026 }
027 public Test(T t)
028 {
029     this.t = t;
030 }
031 }
032 class Program
033 {
034     static void Main(string[] args)
035     {
036         Console.ReadKey();
037     }
038 }
039 }

```

在这段代码中，非泛型类实现了泛型接口。但是非泛型类没有定义类型形参列表。但是在实现接口的成员的时候，又使用了类型形参。因此，编译器会报错，错误信息如下：

```

错误 1    未能找到类型或命名空间名称“T” (是否缺少 using 指令或程序集引用?) H:\书稿 new\书稿源码\第十八章\Generic2\Generic2.cs 9    22    Generic2
错误 2    未能找到类型或命名空间名称“T” (是否缺少 using 指令或程序集引用?) H:\书稿 new\书稿源码\第十八章\Generic2\Generic2.cs 11   17    Generic2
错误 3    未能找到类型或命名空间名称“T” (是否缺少 using 指令或程序集引用?) H:\书稿 new\书稿源码\第十八章\Generic2\Generic2.cs 16   16    Generic2
错误 4    未能找到类型或命名空间名称“T” (是否缺少 using 指令或程序集引用?) H:\书稿 new\书稿源码\第十八章\Generic2\Generic2.cs 27   21    Generic2

```

出现这四个错误的原因就是非泛型类没有类型形参列表，但是又使用了类型形参。这样编译器就会找不到类型形参 T 的定义。当创建类的实例的时候，因为无法指定类型实参，所以无法创建接口的构造类型。在非泛型类实现泛型接口的时候，必须直接指定类型实参。修改后的代码如下：

```

001 using System;
002 namespace Generic2
003 {
004     interface ITest<T>
005     {
006         void Show();
007         T A { get; set; }
008     }
009     class Test:ITest<int>
010     {
011         private int t;

```

```

012         public void Show()
013         {
014             Console.WriteLine("字段 t 的值是: {0}", t);
015         }
016         public int A
017         {
018             get
019             {
020                 return t;
021             }
022             set
023             {
024                 t = value;
025             }
026         }
027         public Test(int t)
028         {
029             this.t = t;
030         }
031     }
032     class Program
033     {
034         static void Main(string[] args)
035         {
036             ITest<int> myITest = new Test(100);
037             myITest.A = 2000;
038             myITest.Show();
039             Console.WriteLine("属性 A 的值为: {0}", myITest.A);
040             Console.ReadKey();
041         }
042     }
043 }

```

在第 9 行指定要实现的接口的时候，直接为接口指定了类型实参。因为接口的类型实参为 `int` 类型，在非泛型类实现它的时候，相关实现的成员类型至少要和接口的类型实参相当。在这个例子中也使用了 `int` 类型。创建接口的变量指向类的实例，则需要指定接口的类型实参也为 `int` 类型。如果在创建类的实例的时候，指定了其它类型的类型实参，则编译器会报告无法转换的错误。使用的代码如下：

```
ITest<float> myITest = new Test(100);
```

编译器报告的错误信息如下：

```

错误 1    无法将类型“Generic2.Test”隐式转换为“Generic2.ITest<float>”。存在一个显式转换(是否缺少强制转换?)  H:\书稿 new\书稿源码\第十八章\Generic2\Generic2.cs 36 36  Generic2

```

18.8.2 多个泛型接口的实现

当实现多个泛型接口的时候，泛型接口依次列在类的继承列表中。实例代码如下：

```

001     using System;
002     namespace Generic2

```



```
003 {
004     interface ITest<T>
005     {
006         void Show();
007         T A { get; set; }
008     }
009     interface ITest1<U, V>
010     {
011         void Show1();
012         U Set1(U u);
013         V Set2(V v);
014     }
015     class Test<T, U, V>:ITest<T>, ITest1<U, V>
016     {
017         private T t;
018         private U u;
019         private V v;
020         public void Show()
021         {
022             Console.WriteLine("字段 t 的值是: {0}", t);
023         }
024         public void Show1()
025         {
026             Show();
027             Console.WriteLine("字段 u 的值是: {0}, 字段 v 的值是 {1}", u, v);
028         }
029         public T A
030         {
031             get
032             {
033                 return t;
034             }
035             set
036             {
037                 t = value;
038             }
039         }
040         public Test(T t, U u, V v)
041         {
042             this.t = t;
043             this.u = u;
044             this.v = v;
045         }
046         public U Set1(U u)
```

```

047         {
048             this.u = u;
049             return this.u;
050         }
051         public V Set2(V v)
052         {
053             this.v = v;
054             return this.v;
055         }
056     }
057     class Program
058     {
059         static void Main(string[] args)
060         {
061             Test<int, bool, double> myTest = new Test<int, bool, double>(100, true,
062                                     200.1);
063             myTest.A = 1000;
064             myTest.Show1();
065             myTest.Set1(false);
066             myTest.Set2(300.2);
067             Console.WriteLine("属性 A 的值为: {0}", myTest.A);
068             Console.ReadKey();
069         }
070     }

```

第 4 行和第 9 行定义了两个泛型接口，接口 `ITest` 带有一个类型形参 `T`；接口 `ITest1` 带有两个类型形参。在第 15 行，泛型类 `Test` 实现接口 `ITest` 和 `ITest1` 的时候，需要有三个类型形参与它们对应。在类 `Test` 中需要对这两个接口中的所有成员进行实现。

当在第 61 行定义类 `Test` 的构造类型的实例的时候，需要指定三个类型实参。它们分别用来替换两个泛型接口中的类型形参。这段代码的执行结果如下：

```

字段 t 的值是：1000
字段 u 的值是：True, 字段 v 的值是 200.1
属性 A 的值为：1000

```

当定义接口类型的变量指向实现类的构造类型的实例的时候，虽然接口类型不包含全部的类型形参，但是全部的类型形参都要在调用实现类的构造方法的时候指定类型实参。代码如下：

```

001     static void Main(string[] args)
002     {
003         ITest<int> myTest = new Test<int, bool, double>(100, true, 200.1);
004         myTest.A = 1000;
005         ITest1<bool, double> myTest1 = (ITest1<bool, double>)myTest;
006         myTest1.Set1(false);
007         myTest1.Set2(200.35);
008         myTest1.Show1();
009         Console.WriteLine("属性 A 的值为: {0}", myTest.A);

```

```

010     Console.ReadKey();
011 }

```

代码的第3行定义了一个 ITest 接口的变量指向了 Test 类的实例。这里的类型实参的指定要和传递给 Test 类的类型实参相同。第5行定义了一个 ITest1 接口的变量，然后将 ITest 接口的变量赋值给它。因为它们之间不能互相赋值，所以需要有一个显式转换。因为它们都指向一个实例，转换是一定会成功的。显示转换的时候，需要指定的 ITest1 类型的变量的类型实参也必须和 Test 实例的类型实参相同。这样当使用 myTest 和 myTest1 调用接口的成员的时候，发生了多态调用。代码的执行结果如下：

字段 t 的值是：1000

字段 u 的值是：False, 字段 v 的值是 200.35

属性 A 的值为：1000

当类实现泛型接口的时候，如果泛型接口有多个类型形参的时候，也可以只指定其中一部分为类型实参。另外一部分留给泛型类去替换。实例代码如下：

```

001     using System;
002     namespace Generic2
003     {
004         interface ITest<U, V>
005         {
006             void Show();
007             U Set1(U u);
008             V Set2(V v);
009         }
010         class Test<U>:ITest<U,string>
011         {
012             private U u;
013             private string v;
014             public void Show()
015             {
016                 Console.WriteLine("字段 u 的值是：{0}", u);
017                 Console.WriteLine("字段 v 的值是：{0}", v);
018             }
019             public U Set1(U u)
020             {
021                 this.u = u;
022                 return this.u;
023             }
024             public string Set2(string v)
025             {
026                 this.v = v;
027                 return this.v;
028             }
029             public Test(U u, string s)
030             {
031                 this.u = u;
032                 this.v = s;

```

```

033         }
034     }
035     class Program
036     {
037         static void Main(string[] args)
038         {
039             ITest<int,string> myTest = new Test<int>(100, "Hello World");
040             myTest.Set1(1000);
041             myTest.Set2("问余何意栖碧山，笑而不答心自闲。桃花流水窅然去，别有天地非人间。");
042             myTest.Show();
043             Console.ReadKey();
044         }
045     }
046 }

```

在这段代码中的第 4 行定义的泛型接口含有两个类型形参，当类 Test 实现接口 ITest 的时候，指定了类型形参 V 为 string 类型。当在第 39 行定义一个接口类型的变量指向实现类的实例的时候，ITest 的第二个类型形参一定也要指定为 string。但是因为类 Test 只有一个类型形参，所以在使用 new 关键字创建类的实例的时候，只需要指定一个类型实参就可以了。当使用接口的变量调用接口的成员的时候，实现了多态的调用。代码的执行结果如下：

字段 u 的值是：1000

字段 v 的值是：问余何意栖碧山，笑而不答心自闲。桃花流水窅然去，别有天地非人间。

18.9 类型参数的约束

类型参数的约束就是在定义泛型类型时，指定有哪些种类的类型实参可以替换泛型类型的类型形参。如果使用类型形参的约束所不允许使用的类型实参种类来构造泛型类，则编译器会报告错误。类型参数的约束就是指的对类型形参施加的这些限制。

18.9.1 值类型约束

值类型约束指定用于泛型类的类型实参必须是不可以为 null 的值类型。它包括结构、枚举和具有值类型约束的类型形参。虽然可以为 null 的类型是值类型，但是它不满足条件。同时，具有值类型约束的类型形参还不能具有构造方法约束。下面是代码实例：

```

001     using System;
002     namespace Generic2
003     {
004         struct A
005         {
006             private int a;
007             public string Show()
008             {
009                 string s = string.Format("结构中字段 a 的值是：{0}", a);
010                 return s;
011             }
012             public A(int a)
013             {
014                 this.a = a;

```

```
015     }
016     public override string ToString()
017     {
018         return Show();
019     }
020 }
021 enum Color { Red, Green, Blue}
022 class Test<T>
023     where T:struct
024 {
025     private T t;
026     public Test(T t)
027     {
028         this.t = t;
029     }
030     public void Show()
031     {
032         Console.WriteLine(t.ToString());
033     }
034 }
035 class Program
036 {
037     static void Main(string[] args)
038     {
039         Test<Color> t1 = new Test<Color>(Color.Blue);
040         t1.Show();
041         A myA = new A(100);
042         Test<A> t2 = new Test<A>(myA);
043         t2.Show();
044         Test<int> t3 = new Test<int>(2000);
045         t3.Show();
046         Console.ReadKey();
047     }
048 }
049 }
```

在代码的第 4 行定义了一个结构，结构中定义了一个 Show 方法，它用来按照格式化字符串的方式返回一个字符串，打印字段的值。第 12 行定义一个构造方法用来对字段赋值。第 16 行重写了基类的 ToString 方法，重写方法中调用了自定义的 Show 实例方法。第 21 行代码定义了一个枚举类型。

第 22 行定义了一个泛型类，泛型类的形参列表中只有一个类型形参 T。对类型形参 T 定义了一个约束，替换它的类型实参必须是值类型约束。定义约束的语法就是在尖括号类型形参列表的后面使用 where 语句，然后接要对其定义的类型形参标识符，再接一个冒号，最后是约束的类型。因为本段代码定义了一个值类型约束，所以这里的约束类型使用了 struct 关键字。

第 25 行定义了一个泛型类的字段，第 26 行是构造方法。第 30 行定义了一个实例方法 Show。它用来调用字段 t 的 ToString 方法。

从第 39 行开始，分别对泛型类进行构造类型的实例化，依次传入的类型实参包括枚举 Color 类型、结构 A 类型和 int 类型，不要忘记，基本数据类型都是结构。然后对它们的实例调用 Show 方法。这段代码的执行结果如下：

Blue

结构中字段 a 的值是：100

2000

在开发环境中编写代码的时候，如果定义的泛型类有类型形参约束，则开发环境的智能提示会提示开发人员约束的类型，如图 18-1 所示。

```
class Program
{
    static void Main(string[] args)
    {
        Test<Color> t1 = new Test<Color>(Color.Blue);
        t1.Show();
        Test<
    }
}
```

class Generic2.Test<T> where T : struct

18-1 约束提示

下面对代码进行修改，修改后的代码如下：

```
001 using System;
002 namespace Generic2
003 {
004     struct A
005     {
006         private int a;
007         public string Show()
008         {
009             string s = string.Format("结构中字段 a 的值是：{0}", a);
010             return s;
011         }
012         public A(int a)
013         {
014             this.a = a;
015         }
016         public override string ToString()
017         {
018             return Show();
019         }
020     }
021     enum Color { Red, Green, Blue}
022     class B
023     {
024         private int b;
025         public B(int b)
026         {
027             this.b = b;
028         }
029     }
030 }
```

```

029         public override string ToString()
030         {
031             string s = string.Format("类中字段 b 的值是: {0}", b);
032             return s;
033         }
034     }
035     class Test<T>
036     where T:struct
037     {
038         private T t;
039         public Test(T t)
040         {
041             this.t = t;
042         }
043         public void Show()
044         {
045             Console.WriteLine(t.ToString());
046         }
047     }
048     class Program
049     {
050         static void Main(string[] args)
051         {
052             Test<Color> t1 = new Test<Color>(Color.Blue);
053             t1.Show();
054             A myA = new A(100);
055             Test<A> t2 = new Test<A>(myA);
056             t2.Show();
057             Test<int> t3 = new Test<int>(2000);
058             t3.Show();
059             B myB = new B(1000);
060             Test<B> t4 = new Test<B>(myB);
061             t4.Show();
062             Console.ReadKey();
063         }
064     }
065 }

```

这段代码在第 21 行定义的枚举类型之后, 又定义了一个类类型 B。与结构相似, 在其中也定义了字段、构造方法以及重写的 ToString 方法等。在第 59 行定义了类 B 的一个实例 myB, 然后在第 60 行创建 Test 类的构造类型实例, 并将 myB 作为参数传递进去。但是因为 myB 是一个引用类型, 所以编译器会报告 2 个错误。错误信息如下:

```

错误 1    类型 "Generic2.B" 必须是不可以为 null 值的类型才能用作泛型类型或方法
"Generic2.Test<T>" 中的参数 "T"      H:\书稿 new\书稿源码\第十八章\Generic2\Generic2.cs 60
18  Generic2

```

错误 2 类型 “Generic2.B” 必须是不可以为 null 值的类型才能用作泛型类型或方法
 “Generic2.Test<T>” 中的参数 “T” H:\书稿 new\书稿源码\第十八章\Generic2\Generic2.cs 60
 35 Generic2

这两个错误提示第 60 行代码使用了非值类型的类型实参，这是编译器对第 60 行创建泛型类的构造类型传递的类型实参进行检查的原因。

值类型约束还可以是具有值类型约束的另一个类型形参。实例代码如下：

```
001 using System;
002 namespace Generic2
003 {
004     struct A
005     {
006         private int a;
007         public string Show()
008         {
009             string s = string.Format("结构中字段 a 的值是: {0}", a);
010             return s;
011         }
012         public A(int a)
013         {
014             this.a = a;
015         }
016         public override string ToString()
017         {
018             return Show();
019         }
020     }
021     enum Color { Red, Green, Blue}
022     class B
023     {
024         private int b;
025         public B(int b)
026         {
027             this.b = b;
028         }
029         public override string ToString()
030         {
031             string s = string.Format("类中字段 b 的值是: {0}", b);
032             return s;
033         }
034     }
035     class Test<T,U>
036         where T:struct
037         where U:T
038     {
```



```

039         private T t;
040         private U u;
041         public Test(T t, U u)
042         {
043             this.t = t;
044             this.u = u;
045         }
046         public void Show()
047         {
048             Console.WriteLine(t.ToString());
049             Console.WriteLine(u.ToString());
050         }
051     }
052     class Program
053     {
054         static void Main(string[] args)
055         {
056             A myA = new A(1000);
057             Color c = Color.Green;
058             Test<A, Color> myTest = new Test<A, Color>(myA, c);
059             myTest.Show();
060             Console.ReadKey();
061         }
062     }
063 }

```

代码第 35 行定义的泛型类中，类型形参列表中有两个类型形参。类型形参 T 的约束为 struct，对于另一个类型形参也定义了一个约束。这时也需要使用新的 where 语句。这个新的 where 语句跟在已经定义的对类型形参 T 的约束之后。在本段代码中，使用了另一个类型形参的名字作为约束。意思是，对于这个类型形参，也就是类型形参 U，它的约束和对类型形参 T 的一样。什么意思呢？看第 58 行代码，当创建泛型类 Test 的构造类型的时候，需要为其传入两个类型实参。原以为 where U:T 的意思是 U 也必须是值类型。但是结果却不是这样，编译器这时报错了，错误信息如下：

```

错误 1    不能将类型“Generic2.Color”用作泛型类型或方法“Generic2.Test<T,U>”中的类型形参“U”。
          没有从“Generic2.Color”到“Generic2.A”的装箱转换。    E:\书稿 new\书稿源码\第十八章
          \Generic2\Generic2.cs 58 21 Generic2
错误 2    不能将类型“Generic2.Color”用作泛型类型或方法“Generic2.Test<T,U>”中的类型形参“U”。
          没有从“Generic2.Color”到“Generic2.A”的装箱转换。    E:\书稿 new\书稿源码\第十八章
          \Generic2\Generic2.cs 58 49 Generic2

```

根据编译器的提示可以知道，where U:T 的意思是，T 使用什么类型的类型实参，则 U 也必须使用相同类型的类型实参。在这种情况下，称 T 已经被密封。U 被强制性地使用与 T 一样的类型。在这种情况下，就没有必要使用两个类型形参，因为这两个类型形参是一样的类型。当使用另一个类型形参作为约束时，称这两个约束是互相依赖的。

18.9.2 引用类型约束

引用类型约束指的是类型形参必须是引用类型，它包括类类型、接口类型、委托类型或数组类型。下面是代码实例：

```
001 using System;
002 namespace Generic2
003 {
004     struct A
005     {
006         private int a;
007         public string Show()
008         {
009             string s = string.Format("结构中字段 a 的值是: {0}", a);
010             return s;
011         }
012         public A(int a)
013         {
014             this.a = a;
015         }
016         public override string ToString()
017         {
018             return Show();
019         }
020     }
021     delegate void D();
022     interface IFly
023     {
024         void Fly();
025     }
026     class B:IFly
027     {
028         private int b;
029         public B(int b)
030         {
031             this.b = b;
032         }
033         public void Fly()
034         {
035             Console.WriteLine("我现在能够飞翔");
036         }
037         public override string ToString()
038         {
039             string s = string.Format("类中字段 b 的值是: {0}", b);
040             return s;
041         }
042     }
043     class Test<T>
044     where T:class
```

```
045     {
046         private T t;
047         public Test(T t)
048         {
049             this.t = t;
050         }
051         public void Show()
052         {
053             Console.WriteLine(t.ToString());
054         }
055         public T A
056         {
057             get
058             {
059                 return t;
060             }
061             set
062             {
063                 t = value;
064             }
065         }
066     }
067     class Program
068     {
069         static void Main(string[] args)
070         {
071             B myB = new B(1000);
072             Test<B> myTest1 = new Test<B>(myB);
073             myTest1.Show();
074             myTest1.A.Fly();
075             IFly myIFly = new B(2000);
076             Test<IFly> myTest2 = new Test<IFly>(myIFly);
077             myTest2.Show();
078             myTest2.A.Fly();
079             int[] myArray = new int[10];
080             for (int i = 0; i < 10; i++)
081             {
082                 myArray[i] = i * 2 + 10;
083             }
084             Test<int[]> myTest3 = new Test<int[]>(myArray);
085             myTest3.Show();
086             Console.WriteLine("数组中的值是: ");
087             foreach (int i in myTest3.A)
088             {
```

```

089         Console.WriteLine(i);
090     }
091     D myD = new D(myIFly.Fly);
092     Test<D> myTest4 = new Test<D>(myD);
093     myTest4.Show();
094     myTest4.A();
095     Console.ReadKey();
096 }
097 }
098 }

```

代码从第 21 行开始定义不同类型的引用类型，第 21 行定义的是一个委托类型 D；第 22 行定义了一个接口 IFly；第 26 行定义了一个类 B，它实现了接口 IFly，并且实现了接口中定义的方法 Fly。第 43 行的泛型类 Test 的类型参数列表中含有一个类型形参 T。对它的约束使用了 class 关键字。class 关键字代表的就是引用类型。

下面来看实际的调用过程，第 71 行代码定义了类 B 的一个实例，然后第 72 行构造泛型类，使用类 B 作为类型实参。第 73 行和第 74 行分别调用了泛型类的 Show 方法和属性中的 get 访问器。get 访问器返回了 myB 的实例，然后用这个实例又调用了它实现接口的实例方法 Fly。第 75 行定义了一个接口类型的变量指向了派生类的实例，然后将接口类型作为类型实参构造泛型类。接下来将接口的变量 myIFly 传递进去。当调用 Show 方法时和属性 A 的 Fly 方法时，实际实现了多态的方法调用。第 79 行定义了一个数组类型并初始化。第 80 行的 for 语句对数组中的每个元素进行赋值。第 84 行以数组类型作为类型实参构造泛型类的构造类型，并将数组传递进去。第 87 行使用 foreach 语句访问构造类型的属性的 get 访问器，它返回的就是数组实例。然后循环输出数组的成员。第 91 行代码定义了委托的变量，它的调用列表中加入了接口的 Fly 方法，因为 myIFly 指向了派生类的实例，所以调用委托也将产生多态的调用。第 92 行代码使用委托类型作为类型实参创建了泛型类的构造类型，然后为其传入前面创建的委托。第 94 行访问构造类型的属性的 get 访问器，它将返回委托的实例，然后直接接一个无参的参数列表调用它。这段代码的执行结果如下：

类中字段 b 的值是：1000

我现在能够飞翔

类中字段 b 的值是：2000

我现在能够飞翔

System.Int32[]

数组中的值是：

10

12

14

16

18

20

22

24

26

28

Generic2.D

我现在能够飞翔

在这段代码中开始的时候定义了一个结构，现在试着用这个结构作为类型实参来构造泛型类，修改的

代码如下：

```
static void Main(string[] args)
{
    A myA = new A(100);
    Test<A> myTest = new Test<A>(myA);
    myTest.Show();
    Console.ReadKey();
}
```

当用结构作为类型实参来构造泛型类的时候，编译器会报错，错误信息如下：

错误 1 类型“Generic2.A”必须是引用类型才能用作泛型类型或方法“Generic2.Test<T>”中的参数“T” E:\书稿 new\书稿源码\第十八章\Generic2\Generic2.cs 72 18 Generic2

错误 2 类型“Generic2.A”必须是引用类型才能用作泛型类型或方法“Generic2.Test<T>”中的参数“T” E:\书稿 new\书稿源码\第十八章\Generic2\Generic2.cs 72 39 Generic2

出现这个错误的原因是，只有引用类型才能构造泛型类，因为类型形参 T 的约束是 class。

18.9.3 构造方法约束

如果泛型类的类型形参指定了构造方法约束，则用于替换类型形参的类型实参类型必须具有公共可访问性的无参构造方法。下面是代码实例：

```
001 using System;
002 namespace Generic2
003 {
004     struct A
005     {
006         private int a;
007         public string Show()
008         {
009             string s = string.Format("结构中字段 a 的值是: {0}", a);
010             return s;
011         }
012         public A(int a)
013         {
014             this.a = a;
015         }
016         public override string ToString()
017         {
018             return Show();
019         }
020     }
021     class B
022     {
023         private int b;
024         public B(int b)
025         {
026             this.b = b;
027         }
028     }
029 }
```

```
028     public override string ToString()
029     {
030         string s = string.Format("类中字段 b 的值是: {0}", b);
031         return s;
032     }
033 }
034 class Test<T>
035     where T : new()
036 {
037     private T t;
038     public Test(T t)
039     {
040         this.t = t;
041     }
042     public void Show()
043     {
044         Console.WriteLine(t.ToString());
045     }
046     public T A
047     {
048         get
049         {
050             return t;
051         }
052         set
053         {
054             t = value;
055         }
056     }
057 }
058 class Program
059 {
060     static void Main(string[] args)
061     {
062         A myA = new A(1000);
063         Test<A> myTest = new Test<A>(myA);
064         myTest.Show();
065         B myB = new B(2000);
066         Test<B> myTest1 = new Test<B>(myB);
067         Console.ReadKey();
068     }
069 }
070 }
```

这段代码定义了一个结构和一个类 B。在类 B 中定义了一个带参数的构造方法。这样一来，编译器为

它提供的无参构造方法就不再提供了。第 34 行代码定义泛型类的时候，使用 where 语句指定了构造方法约束。构造方法约束就是在 where 语句中的形参类型后的冒号后接 new() 语句。它就表示构造泛型类的时候，该类型实参的类型必须具有无参构造方法。因为结构不论是否定义带参构造方法，编译器都会为其提供无参构造方法，所以结构类型是适合的。

第 62 行和第 65 行各定义了一个结构的实例和类的实例，然后是使用它们的类型来构造泛型类。当使用类 B 来构造泛型类的时候，编译器会报错，错误信息如下：

错误 1 “Generic2.B” 必须是具有公共的无参数构造函数的非抽象类型，才能用作泛型类型或方法
“Generic2.Test<T>” 中的参数 “T” E:\书稿 new\书稿源码\第十八章\Generic3\Program.cs 66

18 Generic3

错误 2 “Generic2.B” 必须是具有公共的无参数构造函数的非抽象类型，才能用作泛型类型或方法
“Generic2.Test<T>” 中的参数 “T” E:\书稿 new\书稿源码\第十八章\Generic3\Program.cs 66

40 Generic3

出现错误的原因是类型 B 的无参构造方法因为自定义带参构造方法而取消掉了，下面为它定义一个无参构造方法，则代码会正确执行。执行结果如下：

结构中字段 a 的值是：1000

类中字段 b 的值是：2000

对泛型类中的类型参数使用 new() 构造方法约束之后，一个非常大的好处就是可以在泛型类中对类型参数 T 进行实例化。在前面接触过的代码实例中，对类型参数的对象是几乎不可操作的，要想对它们进行操作，都需要进行复杂的显式转换。有了 new() 构造方法约束之后，这种现象有了大大地改观。下面看实例代码：

```
001 using System;
002 namespace Generic2
003 {
004     struct SA
005     {
006         private int a;
007         public string Show()
008         {
009             string s = string.Format("结构中字段 a 的值是：{0}", a);
010             return s;
011         }
012         public SA(int a)
013         {
014             this.a = a;
015         }
016         public int A
017         {
018             get
019             {
020                 return a;
021             }
022             set
023             {
024                 a = value;

```

```
025         }
026     }
027     public override string ToString()
028     {
029         return Show();
030     }
031 }
032 class CB
033 {
034     private int b;
035     public CB(int b)
036     {
037         this.b = b;
038     }
039     public CB()
040     {
041         this.b = 200;
042     }
043     public int B
044     {
045         get
046         {
047             return b;
048         }
049         set
050         {
051             b = value;
052         }
053     }
054     public override string ToString()
055     {
056         string s = string.Format("类中字段 b 的值是: {0}", b);
057         return s;
058     }
059 }
060 class Test<T>
061     where T : new()
062 {
063     private T t;
064     public Test()
065     {
066         this.t = new T();
067     }
068     public void Show()
```



```

069         {
070             Console.WriteLine(t.ToString());
071         }
072     public T A
073     {
074         get
075         {
076             return t;
077         }
078         set
079         {
080             t = value;
081         }
082     }
083 }
084 class Program
085 {
086     static void Main(string[] args)
087     {
088         Test<SA> myTest1 = new Test<SA>();
089         myTest1.Show();
090         Test<CB> myTest2 = new Test<CB>();
091         myTest2.Show();
092         myTest2.A.B = 2000;
093         myTest2.Show();
094         Console.ReadKey();
095     }
096 }
097 }

```

在这段代码中，泛型类 Test 使用了构造方法约束。使用了这个约束之后，就意味着类型参数的对象可以调用无参的构造方法。在第 64 行的构造方法中，对类型参数的字段 t 由原来的在类外实例化，然后传值进来的方式，改变成了在泛型类中直接调用类型参数的无参构造方法进行实例化。这样就把类型参数对象的实例化挪到了泛型类的内部。

在第 88 行定义一个类型实参为结构类型的构造类型的时候，已经不需要先定义结构类型的实例了。第 90 行也是这样，类型实参的对象在泛型类中进行了实例化。因此就可以使用构造类型的属性 A 的 get 访问器再调用类 CB 的属性的方式访问类 CB 的内部数据成员。代码变得更为简洁。这段代码的执行结果如下：

结构中字段 a 的值是：0

类中字段 b 的值是：200

类中字段 b 的值是：2000

使用 new() 构造方法约束的时候，需要注意，类型实参的类型不能是抽象类。因为抽象类是无法创建实例的。下面是代码实例：

```

001     using System;
002     namespace Generic2
003     {

```

```
004     abstract class TestClass
005     {
006         private int b;
007         public TestClass(int b)
008         {
009             this.b = b;
010         }
011         public TestClass()
012         {
013             this.b = 200;
014         }
015         public int B
016         {
017             get
018             {
019                 return b;
020             }
021             set
022             {
023                 b = value;
024             }
025         }
026         public override string ToString()
027         {
028             string s = string.Format("类中字段 b 的值是: {0}", b);
029             return s;
030         }
031     }
032     class Test<T>
033     where T : new()
034     {
035         private T t;
036         public Test()
037         {
038             this.t = new T();
039         }
040         public void Show()
041         {
042             Console.WriteLine(t.ToString());
043         }
044         public T A
045         {
046             get
047             {
```

```

048         return t;
049     }
050     set
051     {
052         t = value;
053     }
054 }
055 }
056 class Program
057 {
058     static void Main(string[] args)
059     {
060         Test<TestClass> myTest1 = new Test<TestClass>();
061         myTest1.Show();
062         Console.ReadKey();
063     }
064 }
065 }

```

在代码的第 4 行定义了一个抽象类。虽然抽象类也有无参的构造方法，但是它是用来继承的。它的构造方法用来被派生类调用。因此当第 60 行使用抽象类作为类型实参构造泛型类的时候，编译器会报告如下错误：

```

错误 1    “Generic2.TestClass” 必须是具有公共的无参数构造函数的非抽象类型，才能用作泛型类型
或方法 “Generic2.Test<T>” 中的参数 “T”    H:\书稿 new\书稿源码\第十八章\Generic3\Program.cs
60  18  Generic3
错误 2    “Generic2.TestClass” 必须是具有公共的无参数构造函数的非抽象类型，才能用作泛型类型
或方法 “Generic2.Test<T>” 中的参数 “T”    H:\书稿 new\书稿源码\第十八章\Generic3\Program.cs
60  48  Generic3

```

出现这样错误的原因就是在泛型类中的构造方法中，当调用抽象类的无参构造方法创建类的实例的时候，因为类是抽象的而无法创建。

18.9.4 基类约束

基类约束是指类型参数必须是指定基类或派生自指定基类。有了基类约束，泛型类就可以在类中调用类型实参的成员。实例代码如下：

```

001 using System;
002 namespace Generic2
003 {
004     class TestClass
005     {
006         private int b;
007         public TestClass(int b)
008         {
009             this.b = b;
010         }
011         public TestClass()
012         {

```

```
013         this.b = 200;
014     }
015     public int B
016     {
017         get
018         {
019             return b;
020         }
021         set
022         {
023             b = value;
024         }
025     }
026     public override string ToString()
027     {
028         string s = string.Format("类中字段 b 的值是: {0}", b);
029         return s;
030     }
031     public int Add(int i)
032     {
033         return this.b + i;
034     }
035 }
036 class Test<T>
037     where T : TestClass
038 {
039     private T t;
040     public Test(T t)
041     {
042         this.t = t;
043     }
044     public void Show()
045     {
046         Console.WriteLine(t.ToString());
047     }
048     public T A
049     {
050         get
051         {
052             return t;
053         }
054         set
055         {
056             t = value;
```

```

057         }
058     }
059     public int Sum(int a)
060     {
061         return t.Add(a) + t.B;
062     }
063 }
064 class Program
065 {
066     static void Main(string[] args)
067     {
068         TestClass myTestClass = new TestClass(1000);
069         Test<TestClass> myTest1 = new Test<TestClass>(myTestClass);
070         myTest1.Show();
071         myTest1.A.B = 1500;
072         myTest1.Show();
073         int sum = myTest1.Sum(30000);
074         Console.WriteLine("求和的值是: {0}", sum);
075         Console.ReadKey();
076     }
077 }
078 }

```

在第 4 行代码定义的类中，除了为类定义的带参数构造方法和无参构造方法外，还为类定义了属性以及实例方法。第 36 行定义的泛型类使用了基类约束。基类约束的语法就是在 where 语句的冒号后面加上基类的名称。泛型类含有一个类型形参字段，但是第 40 行的构造方法还是采用传递类型实参实例的方式构造类型形参字段。为什么这样呢？因为在约束中使用了基类约束，没有使用构造方法约束。这样泛型类就无法在类中实例化形参字段。在第 59 行代码定义的泛型类的实例方法中，可以看到，泛型类可以调用基类中定义的实例方法。

第 68 行依旧实例化了一个 TestClass 类的实例，然后第 69 行使用 TestClass 类构造了泛型类。为它传入 TestClass 类的实例。第 70 行调用实例方法 Show，Show 方法又间接调用了类型实参重写的 ToString 方法。第 71 行引用泛型类的属性的 get 访问器返回类型实参的实例，然后又调用了类型实参的属性的 set 访问器为之赋值。第 73 行定义一个局部变量，它负责接收泛型类的实例方法 Sum 返回值。而泛型类的 Sum 方法间接调用了类型实参的实例方法 Add。

可以看到，使用基类约束，泛型类与类型实参之间形成了一种紧耦合的状态。基类约束使泛型类更注重于在继承方面的应用。因为它只限制约束中定义的基类及其派生类可用于泛型类，它将泛型类专用化了。

使用基类约束，基类的派生类也可以用于泛型类。下面是代码实例：

```

001     using System;
002     namespace Generic2
003     {
004         class TestClass
005         {
006             private int b;
007             public TestClass(int b)
008             {

```

```
009         this.b = b;
010     }
011     public TestClass()
012     {
013         this.b = 200;
014     }
015     public int B
016     {
017         get
018         {
019             return b;
020         }
021         set
022         {
023             b = value;
024         }
025     }
026     public override string ToString()
027     {
028         string s = string.Format("类中字段 b 的值是: {0}", b);
029         return s;
030     }
031     public int Add(int i)
032     {
033         return this.b + i;
034     }
035 }
036 class TestClass1 : TestClass
037 {
038     private int c;
039     public TestClass1(int b, int c)
040         : base(b)
041     {
042         this.c = c;
043     }
044     public override string ToString()
045     {
046         string s1 = string.Format("派生类中字段 c 的值是: {0}", c);
047         string s = base.ToString() + "\r\n" + s1;
048         return s;
049     }
050 }
051 class Test<T>
052     where T : TestClass
```

```
053     {
054         private T t;
055         public Test(T t)
056         {
057             this.t = t;
058         }
059         public void Show()
060         {
061             Console.WriteLine(t.ToString());
062         }
063         public T A
064         {
065             get
066             {
067                 return t;
068             }
069             set
070             {
071                 t = value;
072             }
073         }
074         public int Sum(int a)
075         {
076             return t.Add(a) + t.B;
077         }
078     }
079     class Program
080     {
081         static void Main(string[] args)
082         {
083             TestClass1 myTestClass1 = new TestClass1(1000, 2000);
084             Test<TestClass1> myTest1 = new Test<TestClass1>(myTestClass1);
085             myTest1.Show();
086             myTest1.A.B = 1500;
087             myTest1.Show();
088             int sum = myTest1.Sum(30000);
089             Console.WriteLine("求和的值是: {0}", sum);
090             TestClass myTest = new TestClass1(10, 20);
091             Test<TestClass> myTest2 = new Test<TestClass>(myTest);
092             myTest2.Show();
093             Console.ReadKey();
094         }
095     }
096 }
```

这段代码以 `TestClass` 为基类，定义了一个它的派生类 `TestClass1`。在派生类中又新定义了一个字段 `c`。然后再次重写 `ToString` 方法。

泛型类的基类约束定义的基类是 `TestClass` 类，因此，第 83 行定义的派生类实例和第 90 行定义的基类的变量指向派生类的实例，都可以派生类作为类型实参和以基类作为类型实参的方式构造泛型类，并且传入泛型类。当以基类的类型作为类型实参，并且传入的变量是基类的引用变量，它实际指向派生类的实例的时候，泛型类的构造类型变量 `myTest2` 调用实例方法 `Show` 实现了多态的调用。`Show` 方法内部调用的是派生类的 `ToString` 方法。这段代码的执行结果如下：

```

类中字段 b 的值是：1000
派生类中字段 c 的值是：2000
类中字段 b 的值是：1500
派生类中字段 c 的值是：2000
求和的值是：33000
类中字段 b 的值是：10
派生类中字段 c 的值是：20

```

基类约束还有一些规则。首先，基类约束的类型必须是类类型。这一点当然毋庸置疑。基类约束的类型也不能是密封类。并且约束的类型也不能是以下类型：`System.Array`、`System.Delegate`、`System.Enum` 和 `System.ValueType`。这几个类型有个特点，它们都是特殊类型的基类。前面曾经介绍过，从这几个类不允许自定义派生类。下面是代码实例：

```

001     using System;
002     namespace Generic2
003     {
004
005         sealed class TestClass
006         {
007             private int b;
008             public TestClass(int b)
009             {
010                 this.b = b;
011             }
012             public TestClass()
013             {
014                 this.b = 200;
015             }
016             public int B
017             {
018                 get
019                 {
020                     return b;
021                 }
022                 set
023                 {
024                     b = value;
025                 }
026             }

```



```
027         public override string ToString()
028         {
029             string s = string.Format("类中字段 b 的值是: {0}", b);
030             return s;
031         }
032         public int Add(int i)
033         {
034             return this.b + i;
035         }
036     }
037     class Test<T>
038     where T : TestClass
039     {
040         private T t;
041         public Test(T t)
042         {
043             this.t = t;
044         }
045         public void Show()
046         {
047             Console.WriteLine(t.ToString());
048         }
049         public T A
050         {
051             get
052             {
053                 return t;
054             }
055             set
056             {
057                 t = value;
058             }
059         }
060         public int Sum(int a)
061         {
062             return t.Add(a) + t.B;
063         }
064     }
065     class Program
066     {
067         static void Main(string[] args)
068         {
069             TestClass myTestClass = new TestClass(1000);
070             Test<TestClass> myTest = new Test<TestClass>(myTestClass);
```

```

071         myTest.Show();
072         Console.ReadKey();
073     }
074 }
075 }

```

这段代码将类 `TestClass` 标记为密封类，在第 70 行仍用这个密封类类型作为类型实参来构造泛型类。在编译时，编译器会提示如下错误信息：

```

错误 1    “Generic2.TestClass” 不是有效的约束。作为约束使用的类型必须是接口、非密封类或类型
参数。    H:\书稿 new\书稿源码\第十八章\Generic3\Program.cs  38  19  Generic3

```

如果将第 38 行的基类约束替换成如下代码：

```
where T : System.Array
```

并且去掉泛型类中的 `Sum` 方法中对 `Add` 方法的调用，这时编译器会提示如下错误：

```

错误 1    约束不能是特殊类 “System.Array” H:\书稿 new\书稿源码\第十八章\Generic3\Program.cs
      38  19  Generic3

```

如果在基类约束中指定前面介绍的几个特殊类，都会提示这样的错误。同时，基类约束也不能是 `object` 类。因为 `object` 类是所有类的基类，加上它等于没有约束。如果使用 `object` 类作为约束，还会提示如下错误：

```

错误 1    约束不能是特殊类 “object” H:\书稿 new\书稿源码\第十八章\Generic3\Program.cs  38
      19  Generic3

```

18.9.5 接口约束

使用接口约束后，对于定义接口约束的类型形参来说，替换它的类型实参必须是指定接口类型或实现指定接口类型的类类型。并且在 `where` 语句中指定的接口约束只能有一次，不能对同一个接口名使用多次。下面是代码实例：

```

001 using System;
002 namespace Generic2
003 {
004     interface IFly
005     {
006         void Fly();
007     }
008     class TestClass:IFly
009     {
010         private int b;
011         public TestClass(int b)
012         {
013             this.b = b;
014         }
015         public TestClass()
016         {
017             this.b = 200;
018         }
019         public int B
020         {
021             get
022             {

```

```
023         return b;
024     }
025     set
026     {
027         b = value;
028     }
029 }
030 public override string ToString()
031 {
032     string s = string.Format("类中字段 b 的值是: {0}", b);
033     return s;
034 }
035 public int Add(int i)
036 {
037     return this.b + i;
038 }
039 public void Fly()
040 {
041     Console.WriteLine("我现在开始飞翔");
042 }
043 }
044 class Test<T>
045     where T : IFly
046 {
047     private T t;
048     public Test(T t)
049     {
050         this.t = t;
051     }
052     public void Show()
053     {
054         Console.WriteLine(t.ToString());
055     }
056     public T A
057     {
058         get
059         {
060             return t;
061         }
062         set
063         {
064             t = value;
065         }
066     }
```

```

067     }
068     class Program
069     {
070         static void Main(string[] args)
071         {
072             IFly myIFly = new TestClass(1000);
073             Test<IFly> myTest = new Test<IFly>(myIFly);
074             myTest.A.Fly();
075             myTest.Show();
076             TestClass myTestClass = new TestClass(2000);
077             Test<TestClass> myTest1 = new Test<TestClass>(myTestClass);
078             myTest1.Show();
079             myTest1.A.Fly();
080             Console.ReadKey();
081         }
082     }
083 }

```

在本段代码中首先定义了一个接口 IFly，类 TestClass 实现了这个接口。第 44 行代码的泛型类中，对类型形参 T 指定了接口约束。接口约束的语法和基类约束很相似，只不过使用接口的标识符替换了类的标识符。在第 73 行构造泛型类时，可以使用接口的类型。传入的接口类型的变量指向了实现类的实例。因此在泛型类的实例方法的调用过程中实现了多态。第 77 行代码也可以用实现类的类型作类型实参来构造泛型类。因为指定为类型实参的类实现了接口。代码的执行结果如下：

我现在开始飞翔

类中字段 b 的值是：1000

类中字段 b 的值是：2000

我现在开始飞翔

18.9.6 主要约束和次要约束

主要约束和次要约束的意思就是可以给一个类型形参指定多个约束，这些约束同时对指定的类型形参起作用。回忆前面讲过的构造方法约束，如果要在指定基类约束的同时，还要在泛型类中创建类型实参的实例，该如何做呢？在这种情况下，可以为类型形参指定两个约束，一个是基类约束；另一个是构造方法约束。实例代码如下：

```

001     using System;
002     namespace Generic2
003     {
004         class TestClass
005         {
006             private int b;
007             public TestClass(int b)
008             {
009                 this.b = b;
010             }
011             public TestClass()
012             {
013                 this.b = 200;

```

```
014     }
015     public int B
016     {
017         get
018         {
019             return b;
020         }
021         set
022         {
023             b = value;
024         }
025     }
026     public override string ToString()
027     {
028         string s = string.Format("类中字段 b 的值是: {0}", b);
029         return s;
030     }
031 }
032 class Test<T>
033     where T : TestClass, new()
034 {
035     private T t;
036     public Test()
037     {
038         this.t = new T();
039     }
040     public void Show()
041     {
042         Console.WriteLine(t.ToString());
043     }
044     public T A
045     {
046         get
047         {
048             return t;
049         }
050         set
051         {
052             t = value;
053         }
054     }
055 }
056 class Program
057 {
```

```

058         static void Main(string[] args)
059         {
060             Test<TestClass> myTest = new Test<TestClass>();
061             myTest.Show();
062             myTest.A.B = 1500;
063             myTest.Show();
064             Console.ReadKey();
065         }
066     }
067 }

```

在本段代码中，因为要在泛型类的构造方法中使用无参构造方法创建类型形参字段的一个实例。如果只指定基类约束，虽然类 TestClass 中有无参构造方法，但因为没有构造方法约束，所以仍然不能在构造方法中使用无参构造方法创建字段的实例。这时可以为类型形参的约束添加一个次要的约束。把它和主要约束用逗号隔开。因此，第 33 行在 where 语句中又添加了 new() 构造方法约束。使用构造方法约束的时候需要注意，如果有多个约束，构造方法约束一定要放在列表的最后面。这段代码的执行结果如下：

类中字段 b 的值是：200

类中字段 b 的值是：1500

当存在多个约束的时候，第一个是主要约束，其余的多个是次要约束，new() 约束要放在最后。其中主要约束可以是基类约束、接口约束、引用类型约束和值类型约束。次要约束可以是类型形参约束和接口约束。如果多个约束中有 new() 构造方法约束，则它要放在最后面。最后，指定的类型形参只能有一个基类约束。如果指定了基类约束，那么它一定要做主要约束，放在约束列表的第一位。下面看代码实例：

```

001     using System;
002     namespace Generic2
003     {
004         interface IFly
005         {
006             void Fly();
007         }
008         class TestClass:IFly
009         {
010             private int b;
011             public TestClass(int b)
012             {
013                 this.b = b;
014             }
015             public TestClass()
016             {
017                 this.b = 200;
018             }
019             public int B
020             {
021                 get
022                 {
023                     return b;

```

```
024         }
025         set
026         {
027             b = value;
028         }
029     }
030     public override string ToString()
031     {
032         string s = string.Format("类中字段 b 的值是: {0}", b);
033         return s;
034     }
035     public void Fly()
036     {
037         Console.WriteLine("我现在开始飞翔");
038     }
039 }
040 class Test<T>
041     where T : IFly, TestClass, new()
042 {
043     private T t;
044     public Test()
045     {
046         this.t = new T();
047     }
048     public void Show()
049     {
050         Console.WriteLine(t.ToString());
051     }
052     public T A
053     {
054         get
055         {
056             return t;
057         }
058         set
059         {
060             t = value;
061         }
062     }
063 }
064 class Program
065 {
066     static void Main(string[] args)
067     {
```

```
068         Test<TestClass> myTest = new Test<TestClass>();
069         myTest.Show();
070         myTest.A.B = 1500;
071         myTest.Show();
072         Console.ReadKey();
073     }
074 }
075 }
```

第 8 行的类 TestClass 实现了定义的接口 IFly。在泛型类 Test 中，第 41 行代码指定了一个约束列表。类型形参 T 必须是接口 IFly 类型或实现了 IFly 接口的实现类类型，它的类型也必须是类 TestClass 或 TestClass 的派生类，而且类型实参必须具有无参构造方法。但是这段代码中定义约束列表的时候，基类约束写在了接口约束之后。这时，编译器会报告错误，错误信息如下：

错误 1 类类型约束“Generic2.TestClass”必须在其他任何约束之前 H:\书稿 new\书稿源码\第十八章\Generic3\Program.cs 41 24 Generic3

在泛型类的约束列表中，所有列出的关系是同时符合的关系。如果类型实参有一项不符合，则编译器会报告错误。下面修改代码如下：

```
001 using System;
002 namespace Generic2
003 {
004     interface IFly
005     {
006         void Fly();
007     }
008     class TestClass
009     {
010         private int b;
011         public TestClass(int b)
012         {
013             this.b = b;
014         }
015         public TestClass()
016         {
017             this.b = 200;
018         }
019         public int B
020         {
021             get
022             {
023                 return b;
024             }
025             set
026             {
027                 b = value;
028             }
029         }
030     }
031 }
```



```
029     }
030     public override string ToString()
031     {
032         string s = string.Format("类中字段 b 的值是: {0}", b);
033         return s;
034     }
035 }
036 class Test<T>
037     where T : TestClass, IFly, new()
038 {
039     private T t;
040     public Test()
041     {
042         this.t = new T();
043     }
044     public void Show()
045     {
046         Console.WriteLine(t.ToString());
047     }
048     public T A
049     {
050         get
051         {
052             return t;
053         }
054         set
055         {
056             t = value;
057         }
058     }
059 }
060 class Program
061 {
062     static void Main(string[] args)
063     {
064         Test<TestClass> myTest = new Test<TestClass>();
065         myTest.Show();
066         myTest.A.B = 1500;
067         myTest.Show();
068         Console.ReadKey();
069     }
070 }
071 }
```

第 8 行的类没有实现接口 IFly, 而第 37 行的约束却要求类型实参必须是类 TestClass 或它的派生类, 而且

实现了接口 IFly，且必须具有无参构造方法。由此可以推导得出，传入的类型实参是 TestClass 或它的派生类，而且必须实现了 IFly 接口。但是因为传递进去的类型实参是 TestClass 类，它没有实现 IFly 接口。所以编译器会报告如下错误：

错误 1 不能将类型“Generic2.TestClass”用作泛型类型或方法“Generic2.Test<T>”中的类型参数“T”。没有从“Generic2.TestClass”到“Generic2.IFly”的隐式引用转换。 H:\书稿 new\书稿源

码\第十八章\Generic3\Program.cs 64 18 Generic3

错误 2 不能将类型“Generic2.TestClass”用作泛型类型或方法“Generic2.Test<T>”中的类型参数“T”。没有从“Generic2.TestClass”到“Generic2.IFly”的隐式引用转换。 H:\书稿 new\书稿源

码\第十八章\Generic3\Program.cs 64 47 Generic3

下面再看一下，如果在约束的列表中指定了两个基类约束会怎么样？实例代码如下：

```
001 using System;
002 namespace Generic2
003 {
004     class Father
005     {
006         private int age; //表示年龄
007         public Father() //构造方法
008         {
009             age = 50;
010         }
011         public int Age //对 age 存取
012         {
013             get
014             {
015                 return age;
016             }
017             set
018             {
019                 age = value;
020             }
021         }
022     }
023     class Son : Father
024     {
025         private int age;
026         public Son()
027         {
028             age = 30;
029         }
030         public new int Age //隐藏基类同名属性
031         {
032             get
033             {
034                 return age;
```

```
035         }
036         set
037         {
038             age = value;
039         }
040     }
041 }
042 class Grandson : Son
043 {
044     private int age;
045     public Grandson()
046     {
047         age = 2;
048     }
049     public new int Age //隐藏基类同名属性
050     {
051         get
052         {
053             return age;
054         }
055         set
056         {
057             age = value;
058         }
059     }
060 }
061 class Test<T>
062     where T : Father, Son, new()
063 {
064     private T t;
065     public Test()
066     {
067         t = new T(); //构造类型实参实例
068     }
069     public void Show()
070     {
071         Console.WriteLine("年龄是: {0}", t.Age); //访问类型实参的属性
072     }
073 }
074 class Program
075 {
076     static void Main(string[] args)
077     {
078         Test<Grandson> myTest = new Test<Grandson>();
```

```

079         myTest.Show();
080         Console.ReadKey();
081     }
082 }
083 }

```

这段代码定义了三个类，父亲类、儿子类和孙子类，它们依次派生。在每个类中，成员也比较简单，一个代表年龄的字段，一个属性用来存取字段和一个无参的构造方法。因为要想实现在泛型类中根据传入的类型实参的不同调用不同的属性。所以在类的继承过程中使用了相同的属性名，并且使用了 `new` 关键字将继承来的同名属性进行了隐藏。

但是，第 62 行的约束中却写入了两个基类约束。在第 78 行使用类型实参构造泛型类的时候，使用了直接和间接派生自 `Father` 类的 `Grandson` 类。然后，调用实例方法 `Show` 来输出年龄。因为在基类约束中不允许定义两个基类约束，所以编译器会报错，错误信息如下：

```

错误 1    类类型约束“Generic2.Son”必须其他任何约束之前    H:\书稿 new\书稿源码\第十八章
\Generic2\Generic2.cs 62 26  Generic2

```

出现这个错误的原因是，`Grandson` 是派生自 `Son` 的，所以编译器认为应该 `Son` 在主要约束的位置。改正这个错误，将 `Son` 的基类约束去掉。则代码的执行结果如下：

```

年龄是：50

```

18.9.7 约束类型的成员确定

从前一小节的代码执行结果可以看到，虽然代码得以执行了，但是结果却不是想象中的结果。在派生类之间使用 `new` 关键字隐藏基类的属性，意图就是根据传入的类型实参的不同，调用不同类型的同名属性。但是，传入的类型是 `Grandson` 类型，调用的却是 `Father` 类型的属性。这是怎么回事呢？这里面涉及一个约束类型的成员确定问题。下面修改代码，修改后的代码如下：

```

001     using System;
002     namespace Generic2
003     {
004         class Father
005         {
006             private int age; //表示年龄
007             public Father()
008             {
009                 age = 50;
010             }
011             public int Age //对 age 存取
012             {
013                 get
014                 {
015                     return age;
016                 }
017                 set
018                 {
019                     age = value;
020                 }
021             }
022         }

```

```
023     class Son : Father
024     {
025         private int age;
026         public Son()
027         {
028             age = 30;
029         }
030         public new int Age //隐藏基类的属性
031         {
032             get
033             {
034                 return age;
035             }
036             set
037             {
038                 age = value;
039             }
040         }
041     }
042     class Grandson : Son
043     {
044         private int age;
045         public Grandson()
046         {
047             age = 2;
048         }
049         public new int Age //隐藏基类的属性
050         {
051             get
052             {
053                 return age;
054             }
055             set
056             {
057                 age = value;
058             }
059         }
060     }
061     class Test<T>
062     where T : Father, new() //只定义一个基类约束
063     {
064         private T t;
065         public Test()
066         {
```

```

067         t = new T();
068         Console.WriteLine(t.GetType().FullName); //确定类型实参的类型
069     }
070     public void Show()
071     {
072         Console.WriteLine(t.GetType().FullName); //确定类型实参的类型
073         Console.WriteLine("年龄是: {0}", t.Age);
074     }
075 }
076 class Program
077 {
078     static void Main(string[] args)
079     {
080         Test<Grandson> myTest = new Test<Grandson>();
081         myTest.Show(); //调用泛型类的实例方法, 间接调用类型实参的同名属性
082         Console.ReadKey();
083     }
084 }
085 }

```

修改后的代码在第 62 行只保留了一个基类约束。在第 68 行和第 72 行各加上了取得类型形参类型的语句。在第 80 行传入的类型实参依旧是 Grandson 类型。首先来看一下代码的执行结果, 如下:

```

Generic2.Grandson
Generic2.Grandson
年龄是: 50

```

从代码的执行结果可以观察到, 在调用类型实参的 Age 属性之前, 类型实参的类型的确是 Grandson 类型。但是为何调用的却是 Father 类型的属性呢? 难道是 Age 属性的隐藏没有起作用? 下面改变 Main 方法中的代码, 不使用泛型类, 直接定义 Grandson 的实例来调用其 Age 属性, 代码如下:

```

static void Main(string[] args)
{
    //Test<Grandson> myTest = new Test<Grandson>();
    //myTest.Show(); //调用泛型类的实例方法, 间接调用类型实参的同名属性
    Grandson myGrandson = new Grandson();
    Console.WriteLine("Grandson 的年龄是: {0}", myGrandson.Age);
    Console.ReadKey();
}

```

这时代码的执行结果是这样:

```

Grandson 的年龄是: 2

```

可以看到, 这才是想要的结果, 它正确调用了相应的属性。可见, 不是 Age 属性隐藏的问题。实际上, 出现这个问题的原因就是, 泛型类的约束类的成员是在确定类型实参之前确定的。换句话说, 泛型类的类型形参的成员根据给出的约束先确定, 然后再根据替换的类型实参来调用约束中的成员。这就是一种多态的调用。那么, 在调用 t.Age 的时候, 虽然可以确定这时 t 的类型是 Grandson。但是 Age 的调用却是多态的调用。在内部, 它的调用是用约束的类型来调用的, 代码更像是这样:

```

Father t = new Grandson();
Console.WriteLine("年龄是: {0}", t.Age);

```

因为父类和子类之间的 Age 属性没有重写，而只是隐藏的关系。所以将根据 Father 引用变量的类型调用 Father 类型的 Age 属性。下面再修改代码来验证这一点，修改的代码如下：

```
001 using System;
002 namespace Generic2
003 {
004     class Father
005     {
006         private int age; //表示年龄
007         public Father()
008         {
009             age = 50;
010         }
011         public virtual int Age //对 age 存取
012         {
013             get
014             {
015                 return age;
016             }
017             set
018             {
019                 age = value;
020             }
021         }
022         public void ShowAge()
023         {
024             Console.WriteLine("父亲的年龄是：{0}", age);
025         }
026     }
027     class Son : Father
028     {
029         private int age;
030         public Son()
031         {
032             age = 30;
033         }
034         public override int Age //重写基类的属性
035         {
036             get
037             {
038                 return age;
039             }
040             set
041             {
042                 age = value;
```

```
043     }
044     }
045 }
046 class Grandson : Son
047 {
048     private int age;
049     public Grandson()
050     {
051         age = 2;
052     }
053     public override int Age //重写基类的属性
054     {
055         get
056         {
057             return age;
058         }
059         set
060         {
061             age = value;
062         }
063     }
064     public new void ShowAge() //隐藏基类的同名方法
065     {
066         Console.WriteLine("孙子的年龄是: {0}", age);
067     }
068 }
069 class Test<T>
070     where T : Father, new() //只定义一个基类约束
071 {
072     private T t;
073     public Test()
074     {
075         t = new T();
076         Console.WriteLine(t.GetType().FullName); //确定类型实参的类型
077     }
078     public void Show()
079     {
080         Console.WriteLine(t.GetType().FullName); //确定类型实参的类型
081         Console.WriteLine("年龄是: {0}", t.Age);
082         t.ShowAge(); //在智能提示中, t 的成员出现了 ShowAge 方法
083     }
084 }
085 class Program
086 {
```



```

087         static void Main(string[] args)
088         {
089             Test<Grandson> myTest = new Test<Grandson>();
090             myTest.Show(); //调用泛型类的实例方法，间接调用类型实参的同名属性
091             Console.ReadKey();
092         }
093     }
094 }

```

在第 22 行代码，为 Father 类型增加了一个实例方法 ShowAge。然后在每个派生的类中，都重写每个类的基类的 Age 属性。在 Grandson 类中也定义一个 ShowAge 方法，使用 new 关键字隐藏继承来的 ShowAge 方法。当在第 82 行定义调用 ShowAge 方法的语句时，可以看到，虽然这时第 89 行的传入类型实参的语句还没有调用，但是在开发环境的智能提示中已经可以看到 Father 类型的 ShowAge 方法了。这就说明，这个方法成员的确定是由约束的类型来确定的。下面执行代码，结果如下：

```

Generic2.Grandson
Generic2.Grandson
年龄是：2
父亲的年龄是：50

```

从结果可以得知，因为属性 Age 实现了重写，所以得到了多态的调用。但是 ShowAge 方法因为没有重写，所以只有 Father 类型的 ShowAge 方法得到了调用。

关于约束类型的成员还有一个可访问性的问题，对于约束中的类型，它的可访问性不能比泛型类还低。至少应该与它有相同的可访问性。下面看代码实例：

```

001     using System;
002     namespace Generic2
003     {
004         class Father //可访问性默认为 internal
005         {
006             private int age;
007             public Father()
008             {
009                 age = 50;
010             }
011             public virtual int Age
012             {
013                 get
014                 {
015                     return age;
016                 }
017                 set
018                 {
019                     age = value;
020                 }
021             }
022             public void ShowAge()
023             {

```

```

024         Console.WriteLine("父亲的年龄是: {0}", age);
025     }
026 }
027 public class Test<T> //改变泛型类的可访问性为 public
028     where T : Father, new()
029 {
030     private T t;
031     public Test()
032     {
033         t = new T();
034     }
035     public void Show()
036     {
037         Console.WriteLine("年龄是: {0}", t.Age);
038         t.ShowAge();
039     }
040 }
041 class Program
042 {
043     static void Main(string[] args)
044     {
045         Test<Father> myTest = new Test<Father>();
046         myTest.Show();
047         Console.ReadKey();
048     }
049 }
050 }

```

代码的第 4 行将 Father 类的可访问性设置为 internal，而第 27 行将泛型类的可访问性设置为 public 类型。这时编译代码，编译器会提示错误，错误信息如下：

错误 1 可访问性不一致：约束类型“Generic2.Father”的可访问性比“Generic2.Test<T>”低 H:\书稿 new\书稿源码\第十八章\Generic2\Generic2.cs27 18 Generic2

错误的原因就是基类约束的类型的可访问性比泛型类的可访问性低的原因，这在语法逻辑上是不允许的。

18.9.8 约束类型的泛型

在对一个类型形参定义约束的时候，约束可以有多个，它们包括一个主要约束和多个次要约束。约束类型的本身也可以是一个泛型类。下面是代码实例：

```

001 using System;
002 namespace Generic2
003 {
004     interface ICompare<T>
005     {
006         string CompareAge(T t);
007         string CompareHeight(T t);
008     }
009     class Person

```

```
010     {
011         private int age;
012         private int height;
013         public Person(int age, int height)
014         {
015             this.age = age;
016             this.height = height;
017         }
018         public Person()
019         {
020             this.age = 30;
021             this.height = 178;
022         }
023         public int Age
024         {
025             get
026             {
027                 return age;
028             }
029             set
030             {
031                 age = value;
032             }
033         }
034         public int Height
035         {
036             get
037             {
038                 return height;
039             }
040             set
041             {
042                 height = value;
043             }
044         }
045         public virtual void Show()
046         {
047             Console.WriteLine("本人年龄是: {0}, 本人身高是: {1}", Age, Height);
048         }
049     }
050     class Student : Person, ICompare<Student>
051     {
052         private string name; //表示学生姓名
053         public Student()
```

```
054         : base(20, 170)
055     {
056         this.name = "学生";
057     }
058     public string Name //对 name 字段进行存取
059     {
060         get
061         {
062             return name;
063         }
064         set
065         {
066             name = value;
067         }
068     }
069     /// <summary>
070     /// 重写基类的 Show 方法, 打印学生的个人信息
071     /// </summary>
072     public override void Show()
073     {
074         base.Show();
075         Console.WriteLine("学生的姓名是: {0}", Name);
076     }
077     /// <summary>
078     /// 重写接口的方法
079     /// </summary>
080     /// <param name="t">对方的实例</param>
081     /// <returns>比较结果字符串</returns>
082     public string CompareAge(Student t)
083     {
084         if (this.Age > t.Age)
085         {
086             return this.Name + "年龄较大";
087         }
088         if (this.Age == t.Age)
089         {
090             return this.Name + "与" + t.Name + "一样大";
091         }
092         if (this.Age < t.Age)
093         {
094             return t.Name + "年龄较大";
095         }
096         else
097             return "无法比较"; //在所有可能的返回路径上都要有返回值
```

```
098     }
099     /// <summary>
100     /// 重写接口的方法
101     /// </summary>
102     /// <param name="t">对方的实例</param>
103     /// <returns>比较结果字符串</returns>
104     public string CompareHeight(Student t)
105     {
106         if (this.Height > t.Height)
107         {
108             return this.Name + "身高较高";
109         }
110         if (this.Height == t.Height)
111         {
112             return this.Name + "与" + t.Name + "一样高";
113         }
114         if (this.Height < t.Height)
115         {
116             return t.Name + "身高较高";
117         }
118         else
119             return "无法比较"; //在所有可能的返回路径上都要有返回值
120     }
121 }
122 class StudentInfo<T>
123     where T : Person, ICompare<T>, new()
124 {
125     private T t;
126     public StudentInfo()
127     {
128         this.t = new T();
129     }
130     public void Show()
131     {
132         Console.WriteLine(t.ToString());
133     }
134     public T A
135     {
136         get
137         {
138             return t;
139         }
140         set
141         {
```

```

142         t = value;
143     }
144 }
145 }
146 class Program
147 {
148     static void Main(string[] args)
149     {
150         StudentInfo<Student> student1 = new StudentInfo<Student>();
151         student1.A.Age = 28;
152         student1.A.Height = 180;
153         student1.A.Name = "令狐冲";
154         student1.A.Show();
155         StudentInfo<Student> student2 = new StudentInfo<Student>();
156         student2.A.Age = 30;
157         student2.A.Height = 179;
158         student2.A.Name = "段誉";
159         student2.A.Show();
160         Console.WriteLine("令狐冲与段誉比身高的结果是: {0}",
            student1.A.CompareHeight(student2.A));
161         Console.WriteLine("令狐冲与段誉比年龄的结果是:
            {0}", student1.A.CompareAge(student2.A));
162         Console.ReadKey();
163     }
164 }
165 }

```

这段代码比较长，因此相关的地方做了注释。第 4 行代码定义了一个泛型接口，在接口里定义了两个方法的声明。一个是比较年龄的方法；另一个是比较身高的方法；第 9 行定义了一个 Person 类，它在这段代码中要用作基类使用，它定义了一个人的最基本的信息，包括年龄、身高以及年龄和身高的属性。又定义了一个虚方法用来打印年龄和身高的信息。第 50 行定义了 Student 类，它继承自类 Person。同时它又实现了泛型接口。泛型接口的类型实参也定义成 Student 类。因为实现的方法中要比较两个 Student 对象的内部数据。Student 类又定义了 string 类型的字段用来表示学生的名字。同时与之对应的还有相应的属性。在无参构造方法中，对类的字段做了最基本的设置。它主要是给构造方法约束使用，设置的值没有什么特殊的意义。在实际使用的时候，是要使用相关的属性进行重新设置的。第 72 行重写了基类的 Show 方法，它又添加打印了学生的名字。从第 82 行开始，实现了接口的两个方法。需要注意的是，因为方法有返回值，所以尽管三个 if 语句已经包括了三种比较情况，但还是需要一个 else 语句将所有可能的返回路径补充完整。否则编译器会有相关提示信息。

第 122 行定义了泛型类，它有三个约束条件，要求传入的类型实参是 Person 类或它的派生类。并且实现了 ICompare<T>接口，还要有无参构造方法。

从第 150 行代码开始定义了两个 StudentInfo 类的实例，传入的类型实参是 Student 类型。这时，泛型接口约束中的类型实参也是 Student 类型，这时它才开始构建构造类型接口。第 160 行代码和第 161 行代码调用接口方法比较两个人的年龄和身高。代码的执行结果如下：

```

本人年龄是: 28, 本人身高是: 180
学生的姓名是: 令狐冲

```

本人年龄是：30，本人身高是：179

学生的姓名是：段誉

令狐冲与段誉比身高的结果是：令狐冲身高较高

令狐冲与段誉比年龄的结果是：段誉年龄较大

泛型类在派生过程中，它的类型形参是不被继承的。同时，约束也不被继承。因此，在泛型类的继承过程中，有时需要同时指定约束。下面是代码实例：

```
001 using System;
002 namespace Generic2
003 {
004     interface ICompare<T>
005     {
006         string CompareAge(T t);
007         string CompareHeight(T t);
008     }
009     class Person
010     {
011         private int age;
012         private int height;
013         public Person(int age,int height)
014         {
015             this.age = age;
016             this.height = height;
017         }
018         public Person()
019         {
020             this.age = 30;
021             this.height = 178;
022         }
023         public int Age
024         {
025             get
026             {
027                 return age;
028             }
029             set
030             {
031                 age = value;
032             }
033         }
034         public int Height
035         {
036             get
037             {
038                 return height;
```

```
039         }
040         set
041         {
042             height = value;
043         }
044     }
045     public virtual void Show()
046     {
047         Console.WriteLine("本人年龄是: {0}, 本人身高是: {1}", Age, Height);
048     }
049 }
050 class Student : Person
051 {
052     private string name; //表示学生姓名
053     public Student()
054         : base(20, 170)
055     {
056         this.name = "学生";
057     }
058     public string Name //对 name 字段进行存取
059     {
060         get
061         {
062             return name;
063         }
064         set
065         {
066             name = value;
067         }
068     }
069     /// <summary>
070     /// 重写基类的 Show 方法, 打印学生的个人信息
071     /// </summary>
072     public override void Show()
073     {
074         base.Show();
075         Console.WriteLine("学生的姓名是: {0}", Name);
076     }
077 }
078 class StudentInfo<T>:ICompare<T>
079     where T :Student,new ()
080 {
081     private T t;
082     public StudentInfo()
```



```
083     {
084         this.t = new T();
085     }
086     public void Show()
087     {
088         Console.WriteLine(t.ToString());
089     }
090     public T A
091     {
092         get
093         {
094             return t;
095         }
096         set
097         {
098             t = value;
099         }
100     }
101     /// <summary>
102     /// 重写接口的方法
103     /// </summary>
104     /// <param name="t">对方的实例</param>
105     /// <returns>比较结果字符串</returns>
106     public string CompareAge(T t)
107     {
108         if (this.A.Age > t.Age)
109         {
110             return this.A.Name + "年龄较大";
111         }
112         if (this.A.Age == t.Age)
113         {
114             return this.A.Name + "与" + t.Name + "一样大";
115         }
116         if (this.A.Age < t.Age)
117         {
118             return t.Name + "年龄较大";
119         }
120         else
121             return "无法比较"; //在所有可能的返回路径上都要有返回值
122     }
123     /// <summary>
124     /// 重写接口的方法
125     /// </summary>
126     /// <param name="t">对方的实例</param>
```

```
127     /// <returns>比较结果字符串</returns>
128     public string CompareHeight(T t)
129     {
130         if (this.A.Height > t.Height)
131         {
132             return this.A.Name + "身高较高";
133         }
134         if (this.A.Height == t.Height)
135         {
136             return this.A.Name + "与" + t.Name + "一样高";
137         }
138         if (this.A.Height < t.Height)
139         {
140             return t.Name + "身高较高";
141         }
142         else
143             return "无法比较"; //在所有可能的返回路径上都要有返回值
144     }
145 }
146 class Program
147 {
148     static void Main(string[] args)
149     {
150         StudentInfo<Student> student1 = new StudentInfo<Student>();
151         student1.A.Age = 28;
152         student1.A.Height = 180;
153         student1.A.Name = "令狐冲";
154         student1.A.Show();
155         StudentInfo<Student> student2 = new StudentInfo<Student>();
156         student2.A.Age = 30;
157         student2.A.Height = 179;
158         student2.A.Name = "段誉";
159         student2.A.Show();
160         Console.WriteLine("令狐冲与段誉比身高的结果是: {0}",
            student1.CompareHeight(student2.A));
161         Console.WriteLine("令狐冲与段誉比年龄的结果是:
            {0}", student1.CompareAge(student2.A));
162         Console.ReadKey();
163     }
164 }
165 }
```

这段代码在基本的功能上和前一段代码没有什么不同。但是它的内部逻辑发生了变化。下面简要介绍一下。在类的继承上, Student 类派生自 Person 类, 但它不再实现接口 ICompare<T>。接口由泛型类 StudentInfo 来实现。因为要求类型实参为 Student 类型, 所以如第 79 行那样, 可以在泛型类实现泛型接

口的过程中，同时对类型形参 T 施加基类约束和构造方法约束。因为泛型接口由泛型类进行了实现，所以第 160 行和第 161 行使用泛型类的实例来调用接口的方法实现 Student 类的实例的内部数据的比较。代码的执行结果如下：

```
本人年龄是：28，本人身高是：180
学生的姓名是：令狐冲
本人年龄是：30，本人身高是：179
学生的姓名是：段誉
令狐冲与段誉比身高的结果是：令狐冲身高较高
令狐冲与段誉比年龄的结果是：段誉年龄较大
```

18.9.9 引用类型约束的比较运算符

没有指定约束的类型形参叫做未绑定的类型形参，前面介绍过，对于未绑定的类型形参来说，对它的操作是少之又少的，很多操作都不可用。下面看代码实例：

```
001    class TestClass<T>
002    {
003        T t1;
004        T t2;
005        public TestClass(T t1, T t2)
006        {
007            this.t1 = t1;
008            this.t2 = t2;
009        }
010        public bool Compare()
011        {
012            return t1 == t2;
013        }
014    }
```

这段代码定义了一个泛型类，在类中定义了两个字段。第 10 行代码定义了一个实例方法用来比较两个字段。但是因为编译器不知道 T 是什么类型。因此就无法使用比较运算符。而且编译器会报错，错误信息如下：

```
错误 1    运算符“==”无法应用于“T”和“T”类型的操作数 H:\书稿 new\书稿源码\第十八章
\Generic1\Generic1.cs 15 20 Generic1
```

只有为类型形参指定约束，编译器才可以知道类型形参的类型。才能知道对类型形参可以进行怎样的操作。实例代码如下：

```
001    using System;
002    namespace Generic2
003    {
004        class TestClass<T>
005            where T:class
006        {
007            T t1;
008            T t2;
009            public TestClass(T t1, T t2)
010            {
011                this.t1 = t1;
```

```

012         this.t2 = t2;
013     }
014     public bool Compare()
015     {
016         return t1 == t2;
017     }
018 }
019 class Program
020 {
021     static void Main(string[] args)
022     {
023         string s1 = "Hello World";
024         string s2 = new System.Text.StringBuilder("Hello World").ToString();
025         if(s1 == s2)
026         {
027             Console.WriteLine("s1 和 s2 是一样的字符串");
028         }
029         TestClass<string> myTestClass1 = new TestClass<string>(s1, s2);
030         if (myTestClass1.Compare() == true)
031         {
032             Console.WriteLine("泛型类中的两个字段的值是一样的");
033         }
034         else
035         {
036             Console.WriteLine("泛型类中的两个字段的值不一样");
037         }
038         Console.ReadKey();
039     }
040 }
041 }

```

这段代码为泛型类 TestClass 的类型形参加上了引用类型的约束。这样，泛型类的类型形参只能是引用类型。因此可以对它使用判断运算符。第 14 行代码对两个 T 类型的字段进行判断。第 23 行定义了一个 string 类型的字符串，并初始化为“Hello World”。第 24 行使用 StringBuilder 方法构建一个字符串内容也为“Hello World”的字符串。第 25 行代码使用“==”运算符对它们进行判断。这个运算符是 string 类型重载过的。它会判断两个字符串是否相等。在 C# 中，如果使用 string 类型直接定义变量并初始化的方式，例如第 23 行那样的定义方式，则两个字符串会引用同一个字符串实例。那样，无论如何两个字符串都会相等。但是，如果调用 stringBuilder 方法则会新创建一个字符串。使用第 25 行代码的方式，如果两个字符串的内容相同，则返回 true。但是，第 30 行代码调用了泛型类的实例方法来判断两个字符串是否相同，这时，Compare 方法却返回了 false。代码的执行结果如下：

s1 和 s2 是一样的字符串

泛型类中的两个字段的值不一样

这是因为，对类型形参使用了引用约束之后，在类内使用“==”运算符判断的是两个字段的值是否引用的同一个实例。即使类型实参重载了“==”运算符用来判断字段的内容也不行。在编译器开始编译时，因为是引用类型的约束，所以它会使用对所有类型都有效的运算符。

那么，为类型形参指定基类约束可不可以呢？下面将引用约束换为基类约束，使用 String 类型指定。代码如下：

```
where T:String
```

但是，这时编译器又会报告错误，错误信息如下：

```
错误 1 “string”不是有效的约束。作为约束使用的类型必须是接口、非密封类或类型参数。 H:\
书稿 new\书稿源码\第十八章\Generic2\Generic2.cs5 17 Generic2
```

从这两个例子可以得出，未绑定的类型形参不能使用“==”和“!=”运算符。因为不知道具体的类型实参是否支持这两个运算符。但是，对于类型形参，可以像前面介绍过的那样，将它们显式转换为 object 类型或任何的接口类型。对于比较运算来说，可以将它们与 null 进行比较。下面是代码实例：

```
001 using System;
002 namespace Generic2
003 {
004     class TestClass<T>
005     {
006         T t1;
007         public TestClass(T t1)
008         {
009             this.t1 = t1;
010         }
011         public bool Compare()
012         {
013             return t1 == null;
014         }
015     }
016     class Program
017     {
018         static void Main(string[] args)
019         {
020             TestClass<string> myTestClass1 = new TestClass<string>("Hello World");
021             Console.WriteLine(myTestClass1.Compare());
022             TestClass<int> myTestClass2 = new TestClass<int>(1000);
023             Console.WriteLine(myTestClass2.Compare());
024             string s = null;
025             TestClass<string> myTestClass3 = new TestClass<string>(s);
026             Console.WriteLine(myTestClass3.Compare());
027             Console.ReadKey();
028         }
029     }
030 }
```

第 20 行为泛型类传入一个 string 类型的实例，这时 Compare 方法将返回 false。第 22 行为泛型类传入一个 int 类型的值，这时将返回 false。对于值类型，未绑定的类型形参与 null 的比较将始终返回 false。第 24 行定义一个 string 类型的空引用，然后将它传入泛型类，这时将返回 true。代码的执行结果如下：

```
False
False
```

True

18.9.10 类型形参约束规则

前面已经接触过值类型约束可以是另一个具有值类型约束的形参。这种约束就是类型形参约束。当一个类型形参以另一个类型形参作为约束的时候，则称这个类型形参依赖那个作为约束的类型形参。在类型形参的依赖关系中不能出现循环。实例代码如下：

```

001    using System;
002    namespace Generic2
003    {
004        class TestClass<T,U>
005            where T:U
006            where U:T
007        {
008            T t1;
009            U u1;
010            public TestClass(T t1,U u1)
011            {
012                this.t1 = t1;
013                this.u1 = u1;
014            }
015            public bool Compare()
016            {
017                return t1 == null;
018            }
019        }
020        class Program
021        {
022            static void Main(string[] args)
023            {
024                TestClass<string, string> myTestClass1 = new TestClass<string, string>("Hello
025                World", "Hello World");
026                Console.WriteLine(myTestClass1.Compare());
027                Console.ReadKey();
028            }
029        }

```

在这段代码中，泛型类 TestClass 又两个类型形参。在代码中对它们定义了类型形参约束。类型形参 T 依赖于 U，类型形参 U 依赖于 T。这时在编译的时候，编译器会报错，错误信息如下：

```

错误 1    涉及“T”和“U”的循环约束依赖项 H:\书稿 new\书稿源码\第十八章\Generic2\Generic2.cs
4        21    Generic2

```

错误信息的原因就是两个类型形参之间形成了类型形参约束的循环依赖关系。

类型形参约束存在继承的依赖关系。如果类型形参 A 依赖于类型形参 B，而类型形参 B 又依赖类型形参 C，则类型形参 A 依赖类型形参 C。实例代码如下：

```

001    using System;
002    namespace Generic2

```

```
003  {
004      class Base
005      {
006          private int a;
007          public Base()
008          {
009              this.a = 100;
010          }
011          public virtual void Show()
012          {
013              Console.WriteLine("基类字段 a 的值是: {0}", a);
014          }
015      }
016      class Derive : Base
017      {
018          private int b;
019          public Derive()
020          {
021              this.b = 200;
022          }
023          public override void Show()
024          {
025              base.Show();
026              Console.WriteLine("派生类字段 b 的值是: {0}", b);
027          }
028      }
029      class TestClass<T,U,V>
030          where T:U
031          where U:V
032          where V:Base
033      {
034          T t1;
035          U u1;
036          V v1;
037          public TestClass(T t1,U u1,V v1)
038          {
039              this.t1 = t1;
040              this.u1 = u1;
041              this.v1 = v1;
042          }
043          public void Print()
044          {
045              t1.Show();
046              u1.Show();
```

```

047         v1.Show();
048     }
049 }
050 class Program
051 {
052     static void Main(string[] args)
053     {
054         Base myBase1 = new Derive();
055         Derive myDerive1 = new Derive();
056         Derive myDerive2 = new Derive();
057         TestClass<Derive, Derive, Base> myTestClass = new TestClass<Derive,
058             Derive, Base>(myDerive1, myDerive2, myBase1);
059         myTestClass.Print();
059         Console.ReadKey();
060     }
061 }
062 }

```

从代码的第 30 行开始，类型形参 T 依赖于类型形参 U，类型形参 U 依赖类型形参 V。而类型形参 V 的约束是基类约束，它必须是类 Base 或它的派生类。所以类型形参 T 和类型形参 U 都必须能够隐式转换为 Base 类。第 57 行为泛型类的构造类型传入类型实参的时候，就遵守着这一规则。前两个类型实参都是第三个类型实参的派生类。代码的执行结果如下：

```

基类字段 a 的值是：100
派生类字段 b 的值是：200
基类字段 a 的值是：100
派生类字段 b 的值是：200
基类字段 a 的值是：100
派生类字段 b 的值是：200

```

如果类型形参有两个，第一个具有值类型约束，则第二个类型形参不能具有基类约束。下面是代码实例：

```

001 using System;
002 namespace Generic2
003 {
004     struct S
005     {
006         private int s;
007         public S(int i)
008         {
009             s = i;
010         }
011         public void Show()
012         {
013             Console.WriteLine("结构中的字段 s 的值是：{0}", s);
014         }
015     }

```



```
016     class Base
017     {
018         private int a;
019         public Base()
020         {
021             this.a = 100;
022         }
023         public virtual void Show()
024         {
025             Console.WriteLine("基类字段 a 的值是: {0}", a);
026         }
027     }
028     class Derive : Base
029     {
030         private int b;
031         public Derive()
032         {
033             this.b = 200;
034         }
035         public override void Show()
036         {
037             base.Show();
038             Console.WriteLine("派生类字段 b 的值是: {0}", b);
039         }
040     }
041     class TestClass<T,U>
042         where T:struct,U
043         where U:Base
044     {
045         T t1;
046         U u1;
047         public TestClass(T t1,U u1)
048         {
049             this.t1 = t1;
050             this.u1 = u1;
051         }
052         public void Print()
053         {
054             t1.Show();
055             u1.Show();
056         }
057     }
058     class Program
059     {
```

```

060         static void Main(string[] args)
061         {
062             Base myBase1 = new Derive();
063             S myS = new S();
064             TestClass<S,Base> myTestClass = new TestClass<S,Base>(myS, myBase1);
065             myTestClass.Print();
066             Console.ReadKey();
067         }
068     }
069 }

```

在这段代码中又定义了一个结构类型 S。第 41 行的泛型类包含两个类型形参，对它们的约束是，类型形参 T 的主要约束是值类型约束，次要约束是类型形参约束 U。类型形参 U 的约束是基类约束，类型是 Base。因为类型形参 T 的次要约束依赖于类型形参 U。因此，T 的约束为值类型约束，而 U 的约束为基类约束是不被语法所允许的。编译器会报告错误，错误信息如下：

错误 1 不能将类型“Generic2.S”用作泛型类型或方法“Generic2.TestClass<T,U>”中的类型形参“T”。没有从“Generic2.S”到“Generic2.Base”的装箱转换。H:\书稿 new\书稿源码\第十八章\Generic2\Generic2.cs 64 23 Generic2

错误 2 不能将类型“Generic2.S”用作泛型类型或方法“Generic2.TestClass<T,U>”中的类型形参“T”。没有从“Generic2.S”到“Generic2.Base”的装箱转换。H:\书稿 new\书稿源码\第十八章\Generic2\Generic2.cs 64 59 Generic2

如果存在三个类型形参 A、B、C，A 依赖 B 和 C。而 B 和 C 都是基类约束，则 A 必须能够隐式转换为 B 和 C。实例代码如下：

```

001     using System;
002     namespace Generic2
003     {
004         class Base
005         {
006             private int a;
007             public Base()
008             {
009                 this.a = 100;
010             }
011             public virtual void Show()
012             {
013                 Console.WriteLine("基类字段 a 的值是: {0}", a);
014             }
015         }
016         class Derive : Base
017         {
018             private int b;
019             public Derive()
020             {
021                 this.b = 200;
022             }
023         }
024     }

```

```
023         public override void Show()
024         {
025             Console.WriteLine("派生类字段 b 的值是: {0}", b);
026         }
027     }
028     class TestClass<T,U,V>
029     where T:U,V
030     where U:Base
031     where V:Base
032     {
033         T t1;
034         U u1;
035         V v1;
036         public TestClass(T t1,U u1,V v1)
037         {
038             this.t1 = t1;
039             this.u1 = u1;
040             this.v1 = v1;
041         }
042         public void Print()
043         {
044             t1.Show();
045             u1.Show();
046             v1.Show();
047         }
048     }
049     class Program
050     {
051         static void Main(string[] args)
052         {
053             Base myBase1 = new Derive();
054             Derive myDerive1 = new Derive();
055             Derive myDerive2 = new Derive();
056             TestClass<Base, Base, Base> myTestClass = new TestClass<Base, Base,
057             Base>(myBase1, myDerive1, myDerive2);
058             TestClass<Derive, Base, Base> myTestClass1 = new TestClass<Derive, Base,
059             Base>(myDerive1, myDerive1, myDerive2);
060             Console.ReadKey();
061         }
062     }
```

泛型类 TestClass 有三个约束，类型形参 T 依赖于类型形参 U 和类型形参 V。类型形参 U 和类型形参 V 都是基类约束，类型为 Base。在这种情况下，第 56 行代码中的类型实参 T 与 U 和 V 类型相同；第 57 行代码中的类型 T 可以隐式转换为 U 和 V 都是允许的。传入类的实例的时候，可以传入派生类以实现多态的调

用。例如第 56 行的调用，基类约束为 Base，但是传入了 Derive 类的实例。如果第 29 行到第 31 行的约束是如下形式：

```
where T:U, V
where U:Derive
where V:Base
```

那么，这时类型形参 T 的类型实参只能是 Derive 类型。构造类型如下：

```
static void Main(string[] args)
{
    Base myBase1 = new Derive();
    Derive myDerive1 = new Derive();
    Derive myDerive2 = new Derive();
    TestClass<Derive, Derive, Base> myTestClass = new TestClass<Derive, Derive,
    Base>(myDerive1, myDerive1, myDerive2);
    Console.ReadKey();
}
```

因为类型形参 T 依赖类型形参 U，所以它只能是 Derive 类型或它的派生类。同时，它也依赖于类型形参 V，所以它也只能是类型 Base，或它的派生类。因为需要同时满足条件，所以类型形参 T 的类型实参只能是 Derive 类型。

18.10 泛型方法的约束

泛型方法也可以具有约束，它的约束的使用方法和泛型类是一样的。实例代码如下：

```
001 using System;
002 namespace Generic2
003 {
004     interface ISing
005     {
006         void Sing();
007     }
008     class Person : IComparable<Person>, ISing
009     {
010         public int Age
011         {
012             get;
013             set;
014         }
015         public string Name
016         {
017             get;
018             set;
019         }
020         public int CompareTo(Person other)
021         {
022             int balance = 0;
023             if (this.Age < other.Age)
024             {
```

```
025         balance = other.Age - this.Age;
026         Console.WriteLine(other.Name + "的年龄大，他们相差{0}岁", balance);
027         return balance;
028     }
029     if (this.Age == other.Age)
030     {
031         balance = 0;
032         Console.WriteLine("他们的年龄一样大");
033         return balance;
034     }
035     if (this.Age > other.Age)
036     {
037         balance = this.Age - other.Age;
038         Console.WriteLine(this.Name + "的年龄大，他们相差{0}岁", balance);
039         return balance;
040     }
041     else
042         return -1;
043 }
044 public void Sing()
045 {
046     Console.WriteLine("我唱歌很好听");
047 }
048 }
049 class Test
050 {
051     public static void GenericMethod<T>(T t1, T t2)
052         where T : IComparable<T>, ISing
053     {
054         t1.Sing();
055         t2.Sing();
056         t1.CompareTo(t2);
057     }
058 }
059
060 class Program
061 {
062     static void Main(string[] args)
063     {
064         Person p1 = new Person();
065         p1.Name = "孙悟空";
066         p1.Age = 500;
067         Person p2 = new Person();
068         p2.Name = "猪八戒";
```

```

069         p2.Age = 300;
070         Test.GenericMethod<Person>(p1, p2);
071         Console.ReadKey();
072     }
073 }
074 }

```

第八行的代码中，定义了一个 Person 类，它实现了 .Net 类库中的 IComparable<T> 泛型接口。这个接口只含有一个方法，就是比较两个对象，并返回一个 int 类型的值。同时，Person 类还实现了自定义的 ISing 接口。在第 49 行的类 Test 中，定义了一个静态的泛型方法，它带有两个 T 类型的形参。在第 52 行对类型形参定义一个约束，类型实参必须实现 IComparable<T> 和 ISing 接口。因为有这两个接口约束，所以在泛型方法中，类型形参 T 可以调用接口中的方法。

第 64 行开始定义了两个 Person 类的实例，然后调用 Test 类中的泛型方法，为其传入 Person 类的类型实参，并传入实例。因为 Person 类型同时实现了约束中的接口。所以使用 Person 类来创建构造方法是语法允许的。这段代码的执行结果如下：

```

我唱歌很好听
我唱歌很好听
孙悟空的年龄大，他们相差 200 岁

```

18.11 泛型委托

在定义委托类型的时候，也可以为委托类型指定类型形参，让它可以应用多种类型实参。这样的委托类型就是泛型委托。实例代码如下：

```

001 using System;
002 namespace Generic2
003 {
004     delegate int D<T>(T t)
005     where T:IComparable<T>;//因为这个委托要调用比较方法，所以需要接口约束
006     delegate void D1<T>() where T:ISing;//这个委托需要调用 Sing 方法，所以需要接口约束
007     interface ISing
008     {
009         void Sing();
010     }
011     class Person : IComparable<Person>, ISing
012     {
013         public int Age
014         {
015             get;
016             set;
017         }
018         public string Name
019         {
020             get;
021             set;
022         }
023         public int CompareTo(Person other)
024         {

```

```
025         int balance = 0;
026         if (this.Age < other.Age)
027         {
028             balance = other.Age - this.Age;
029             Console.WriteLine(other.Name + "的年龄大，他们相差{0}岁", balance);
030             return balance;
031         }
032         if (this.Age == other.Age)
033         {
034             balance = 0;
035             Console.WriteLine("他们的年龄一样大");
036             return balance;
037         }
038         if (this.Age > other.Age)
039         {
040             balance = this.Age - other.Age;
041             Console.WriteLine(this.Name + "的年龄大，他们相差{0}岁", balance);
042             return balance;
043         }
044         else
045             return -1;
046     }
047     public void Sing()
048     {
049         Console.WriteLine("唱歌很好听");
050     }
051 }
052 class Test
053 {
054     public static void InvokeAdd(Person p1, Person p2)
055     {
056         D<Person> myD1 = p1.CompareTo;
057         myD1(p2);
058         Console.Write(p1.Name); //打印调用方的 Name 属性，与 myD2 委托的输出拼接
059         D1<Person> myD2 = p1.Sing;
060         myD2();
061     }
062 }
063 class Program
064 {
065     static void Main(string[] args)
066     {
067         Person p1 = new Person();
068         p1.Name = "孙悟空";
```

```

069         p1.Age = 500;
070         Person p2 = new Person();
071         p2.Name = "猪八戒";
072         p2.Age = 300;
073         Test.InvokeAdd(p1, p2);
074         Console.ReadKey();
075     }
076 }
077 }

```

代码的第 4 行和第 6 行定义了两个泛型委托，泛型委托类型的语法形式就是在委托的名称标识符后带有一个类型形参列表。泛型委托也可以带有约束，它的约束可以写在委托类型的同一行，也可以写在委托类型的下一行。但是，委托类型结尾的分号必须放在 where 语句的后面。第 52 行代码中，在类 Test 中定义了一个静态实例方法，它带有两个 Person 类型的参数。这个方法的功能是为委托添加方法和调用委托。在第 56 行和第 59 行使用 Person 类型创建了泛型委托的构造类型。第 57 行和第 60 行调用了委托。这段代码的执行结果如下：

```

孙悟空的年龄大，他们相差 200 岁
孙悟空唱歌很好听

```

在 .Net 类库中，微软也定义了泛型委托可以供程序员直接使用。下面看代码实例：

```

001 using System;
002 namespace Generic2
003 {
004     interface ISing
005     {
006         void Sing();
007     }
008     class Person : IComparable<Person>, ISing
009     {
010         public int Age
011         {
012             get;
013             set;
014         }
015         public string Name
016         {
017             get;
018             set;
019         }
020         public int CompareTo(Person other)
021         {
022             int balance = 0;
023             if (this.Age < other.Age)
024             {
025                 balance = other.Age - this.Age;
026                 Console.WriteLine(other.Name + "的年龄大，他们相差 {0} 岁", balance);

```



```
027         return balance;
028     }
029     if (this.Age == other.Age)
030     {
031         balance = 0;
032         Console.WriteLine("他们的年龄一样大");
033         return balance;
034     }
035     if (this.Age > other.Age)
036     {
037         balance = this.Age - other.Age;
038         Console.WriteLine(this.Name + "的年龄大，他们相差{0}岁", balance);
039         return balance;
040     }
041     else
042         return -1;
043 }
044 public void Sing()
045 {
046     Console.WriteLine("唱歌很好听");
047 }
048 }
049 class Test
050 {
051     public static void InvokeAdd(Person p1, Person p2)
052     {
053         Func<Person, int> myFunc = p1.CompareTo;
054         Func<Person, int> myFunc1 = (Person p) => { Console.Write(p.Name); p.Sing();
055             return 0; };
056         myFunc(p2);
057         myFunc1(p1);
058     }
059 }
060 class Program
061 {
062     static void Main(string[] args)
063     {
064         Person p1 = new Person();
065         p1.Name = "孙悟空";
066         p1.Age = 500;
067         Person p2 = new Person();
068         p2.Name = "猪八戒";
069         p2.Age = 300;
070         Test.InvokeAdd(p1, p2);
```

```

070         Console.ReadKey();
071     }
072 }
073 }

```

在这段代码中，类和接口的定义与前一个例子相比没有进行修改。重点看静态方法 `InvokeAdd` 方法中的语句。在这个静态方法中，定义了两个委托类型的实例 `myFunc` 和 `myFunc1`。它们的委托类型使用的是 .Net 类库中的 `Func` 委托类型。这个委托类型的定义如下：

```
public delegate TResult Func<in T, out TResult>(T arg)
```

`Func` 委托类型还有含多个类型参数的形式。本例中使用了带有两个类型形参的类型。这个 `Func` 委托的类型形参前面的 `in` 和 `out` 表示的是逆变和协变，这两个概念在后边的章节中会介绍到。它带有一个 `T` 类型的参数。返回值类型是类型形参 `TResult` 的类型。也就是说，这个 `Func` 委托的前一个类型形参是方法的参数类型，后一个类型形参是方法的返回值类型。

第 53 行代码使用 `Person` 类型和 `int` 类型创建了 `Func` 的构造类型，并为它添加了调用方法。这个方法就是 `Person` 类中的 `CompareTo` 方法，因为实现这个方法的时候，它带有一个 `Person` 类型的参数，返回值为 `int` 类型的，它是符合这个 `Func` 的定义的。

第 54 行定义了一个 `Func` 的委托，并且将一个匿名方法添加进入了它的调用列表。这行的意图是调用 `Person` 类的 `Sing` 方法，但是因为 `Sing` 方法不符合 `Func` 的定义，因此用一个匿名方法包裹了它。匿名方法带有一个 `Person` 类型的参数，返回值 `0` 为 `int` 类型。第 55 行和 56 行对委托进行了调用。而实际实例的传入是在第 69 行，将新创建的两个 `Person` 类型的实例传递进入了方法，间接调用了委托。代码的执行结果如下：

```

孙悟空的年龄大，他们相差 200 岁
孙悟空唱歌很好听

```

18.12 表达式树类型

表达式目录树允许将 `lambda` 表达式表示为数据结构而非可执行代码。它使 `lambda` 表达式的结构变得更加透明而明确。在与表达式目录树中的数据进行交互时，其方式就像与任何其他数据结构交互时一样。在 .Net 类库中有一个 `System.Linq.Expressions.Expression<TDelegate>` 泛型类，它可以用表达式目录树的形式将强类型 `lambda` 表达式表示为数据结构。它的定义如下：

```
public sealed class Expression<TDelegate> : LambdaExpression
```

这是一个泛型的密封类，不允许继承。前面介绍过，`lambda` 表达式可以隐式转换为委托。如果存在 `lambda` 表达式到委托的转换，则存在到表达式目录树类型 `Expression<TDelegate>` 的转换。但是需要注意，使用 `delegate` 关键字的匿名方法不可以转换成表达式目录树类型。下面是代码实例：

```

001     using System;
002     using System.Linq.Expressions; //引入表达式目录树类型的命名空间
003     namespace Generic2
004     {
005         class Person: IComparable<Person>
006         {
007             public string Name
008             {
009                 get;
010                 set;
011             }
012             public string ID
013             {

```

```
014         get;
015         set;
016     }
017     public int Age
018     {
019         get;
020         set;
021     }
022     public void Print()
023     {
024         Console.WriteLine("姓名: {0}", Name);
025         Console.WriteLine("身份证号: {0}", ID);
026         Console.WriteLine("年龄: {0}", Age);
027     }
028     public int CompareTo(Person other)
029     {
030         int balance = 0;
031         if (this.Age < other.Age)
032         {
033             balance = other.Age - this.Age;
034             Console.WriteLine(other.Name + "的年龄大, 他们相差 {0} 岁", balance);
035             return balance;
036         }
037         if (this.Age == other.Age)
038         {
039             balance = 0;
040             Console.WriteLine("他们的年龄一样大");
041             return balance;
042         }
043         if (this.Age > other.Age)
044         {
045             balance = this.Age - other.Age;
046             Console.WriteLine(this.Name + "的年龄大, 他们相差 {0} 岁", balance);
047             return balance;
048         }
049         else
050             return -1;
051     }
052 }
053 class Program
054 {
055     static void Main(string[] args)
056     {
057         Person p1 = new Person();
```

```

058         p1.Name = "孙悟空";
059         p1.ID = "123456789";
060         p1.Age = 500;
061         p1.Print();
062         Person p2 = new Person();
063         p2.Name = "猪八戒";
064         p2.ID = "987654321";
065         p2.Age = 300;
066         p2.Print();
067         Func<Person, int> myFunc = p1.CompareTo;
068         myFunc(p2);
069         //Expression<Func<Person, int>> exp = (Person p) => { return
           p1.CompareTo(p); };
070         Expression<Func<int, bool>> exp1 = x => x > 400;
071         Console.WriteLine(exp1.ReturnType.ToString());
072         Func<int, bool> myFunc1 = exp1.Compile();
073         Console.WriteLine(myFunc1(p1.Age));
074         Console.WriteLine(myFunc1(p2.Age));
075         Console.ReadKey();
076     }
077 }
078 }

```

在这段代码中,第 67 行使用了 Func 委托,将 Person 类中的 CompareTo 方法添加进了委托的调用列表。第 68 行调用了委托。虽然,CompareTo 方法可以添加进入 Func 委托,但是第 69 行的 lambda 表达式却不可以隐式转换成表达式目录树类型。这是因为它具有语句体。如果将第 69 行的注释去掉,则编译器会报告错误,如下:

```

错误 1    无法将具有语句体的 lambda 表达式转换为表达式树 H:\书稿 new\书稿源码\第十八章
\Generic1\Generic1.cs 69 49 Generic1

```

第 70 行将一个 lambda 表达式转换成了 Expression 类型的实例。需要注意,Expression 类型的类型实参需要是能将此 lambda 表达式隐式转换成的 Func 委托的构造类型。一旦将 lambda 表达式转换成了表达式目录树,就可以像使用类一样的操作它。第 71 行打印了 lambda 表达式的返回类型,第 72 行调用它的 Compile 方法将表达式目录树又编译为可执行代码,并生成委托。第 73 行和第 74 行调用了这个生成的委托。这段代码的执行结果如下:

```

姓名: 孙悟空
身份证号: 123456789
年龄: 500
姓名: 猪八戒
身份证号: 987654321
年龄: 300
孙悟空的年龄大,他们相差 200 岁
System.Boolean
True
False

```

18.13 可以为空的类型

前面的章节已经接触过可以为 null 的类型，并且使用了它的简化定义方式。实际上，可以为空的类型有一个基础类型，它的基础类型是一个泛型结构，名字叫做 `Nullable<T>`。其中 `T` 是基础类型。可空类型可以简写为 `T?` 的形式。下面是 `Nullable<T>` 的定义：

```
public struct Nullable<T>
where T : struct, new()
```

可以看到，对于类型形参 `T` 来说，定义了两个约束，一个是值类型约束，另一个是构造方法约束。下面是代码实例：

```
001 using System;
002 namespace Generic2
003 {
004     class Program
005     {
006         static void Main(string[] args)
007         {
008             Nullable<int> myNullable = new Nullable<int>(1000);
009             Console.WriteLine(myNullable.HasValue);
010             Console.WriteLine(myNullable.Value.ToString());
011             Nullable<int> myNullable1 = null;
012             Console.WriteLine(myNullable1.HasValue);
013             //引发 System.InvalidOperationException 异常
014             //Console.WriteLine(myNullable1.Value.ToString());
015             Console.ReadKey();
016         }
017     }
018 }
```

除了以形如 `int?` 的形式定义可空类型外，还可以使用泛型结构定义可空类型。如同第 8 行代码所示。使用哪种类型的可空类型，那么泛型结构的类型实参就替换为哪种基础类型。第 9 行和第 10 行调用泛型结构的成员属性输出泛型结构是否有基础类型值和基础类型的值。第 11 行定义了一个可空类型为 `null`。这时它的 `HasValue` 属性为 `false`，同时不能访问其 `Value` 属性。如果去掉第 14 行的注释，则编译器会抛异常，异常的类型是 `System.InvalidOperationException`。这段代码的执行结果如下：

```
True
1000
False
```

18.14 泛型类继承中的方法重写

在泛型类继承过程中，方法的重写要依据给定的类型实参来确定。下面看代码实例：

```
001 using System;
002 namespace Generic2
003 {
004     abstract class Test<T>
005     {
006         public virtual void Method1(T t)
007         {
008             Console.WriteLine(t);
009         }
010     }
011 }
```

```
010         public abstract T Method2(T t);
011     }
012     class Test1 : Test<string>
013     {
014         public override void Method1(string t)
015         {
016             base.Method1(t);
017         }
018         public override string Method2(string t)
019         {
020             return t + "Hello World";
021         }
022     }
023     class Test2<T, U> : Test<T>
024     {
025         public override void Method1(T t)
026         {
027             base.Method1(t);
028         }
029         public override T Method2(T t)
030         {
031             U u1 = (U)(object)t;
032             T t1 = (T)(object)((int)(object)u1 + 1000);
033             return t1;
034         }
035     }
036     class Program
037     {
038         static void Main(string[] args)
039         {
040             Test<string> myTest1 = new Test1();
041             myTest1.Method1("杨花落尽子规啼，闻道龙标过五溪。");
042             Console.WriteLine("Method2 的返回值是: {0}", myTest1.Method2("C#的"));
043             Test<int> myTest2 = new Test2<int, int>();
044             myTest2.Method1(1320);
045             Console.WriteLine("Method2 的返回值是: {0}", myTest2.Method2(5000));
046             Console.ReadKey();
047         }
048     }
049 }
```

这段代码的第 4 行定义了一个抽象基类，它是一个泛型类，带有一个类型形参 T。第 6 行代码中的虚方法 Method1 带有一个 T 类型的形参；第 10 行代码定义了一个抽象方法，它也带有一个 T 类型的形参，返回值也是 T 类型。

第 12 行代码定义类 Test1 派生自泛型类 Test<string>，可以看到，它将基类的类型形参定义为

string 类型。因此，第 14 行和第 18 行的重写方法都需要将相应的参数类型替换为指定的 string 类型。

第 23 行代码定义了一个泛型类，它带有两个类型形参 T 和 U。它在从 Test 类型派生的时候，因为对基类没有指定类型实参，所以派生类的类型形参需要有一个和基类的类型形参发生映射的关系。这样，在指定基类的类型实参的时候，或指定派生类的类型实参的时候，都会同时指定派生类或基类的映射类型形参的类型。例如，第 43 行代码定义了一个基类的变量指向了派生类的实例，在指定基类的类型实参的同时，也就指定了派生类的类型形参 T 为 int 类型，另一个类型形参 U 由创建派生类的实例的时候指定。这段代码的执行结果如下：

杨花落尽子规啼，闻道龙标过五溪。

Method2 的返回值是：C#的 Hello World

1320

Method2 的返回值是：6000

18.15 静态构造方法对约束类型的进一步指定

静态构造方法对于一个构造类型来说，只执行一次。因此，使用静态构造方法可以对类型形参的约束指定的类型作进一步的检查。下面是代码实例：

```

001    using System;
002    namespace Generic2
003    {
004        class Person
005        { }
006        class Test<T>
007            where T:class
008        {
009            static Test()
010            {
011                if (typeof(T).FullName != "Generic2.Person")
012                {
013                    Console.WriteLine("类型实参不符合约束条件");
014                }
015                else
016                {
017                    Console.WriteLine("类型实参符合约束条件");
018                }
019            }
020        }
021        class Program
022        {
023            static void Main(string[] args)
024            {
025                Test<Person> myTest = new Test<Person>();
026                //Test<string> myTest1 = new Test<string>();
027                Console.ReadKey();
028            }
029        }
030    }

```

在第 6 行定义的泛型类 Test 中，对类型形参 T 定义了一个引用类型约束。如果想要再进一步定义类型实参的类型，可以使用静态构造方法起到和基类约束一样的功能。第 11 行使用 typeof 关键字取得类型实参的类型名，然后判断是否是指定的类型。如果注释掉第 25 行的代码，让第 26 行的代码生效，则编译器会提示错误信息，错误信息如下：

类型实参不符合约束条件

这种判断类型的方法可以应用到值类型约束上，因为值类型中的很多类型都属于特殊类型，不能用在约束中。使用静态构造方法作类型判断可以弥补这一不足。实例代码如下：

```

001    using System;
002    namespace Generic2
003    {
004        class Person
005        { }
006        class Test<T>
007            where T:struct
008        {
009            static Test()
010            {
011                if (typeof(T).FullName != "System.Int32")
012                {
013                    Console.WriteLine("类型实参不符合约束条件");
014                }
015                else
016                {
017                    Console.WriteLine("类型实参符合约束条件");
018                }
019            }
020        }
021        class Program
022        {
023            static void Main(string[] args)
024            {
025                Test<int> myTest1 = new Test<int>();
026                Console.ReadKey();
027            }
028        }
029    }

```

在这段代码中，只有类型实参使用了 int 类型，才会执行第 17 行的语句。其它类型都会执行第 13 行的语句。

18.16 泛型类的内部类

在泛型类中可以包含嵌套的内部类，内部类可以是非泛型的，也可以是泛型内部类。不论内部类是否是泛型类，内部类都是外部泛型类的成员。下面是代码实例：

```

001    using System;
002    namespace Generic2
003    {

```



```
004     class List<T> //链表的泛型类
005     {
006         private InnerData id; //链表的节点
007         private class InnerData //表示链表节点的内部类
008         {
009             /// <summary>
010             /// 节点内的数据
011             /// </summary>
012             public T Data
013             {
014                 get;
015                 set;
016             }
017             public InnerData i; //节点含有的指向下一个节点的引用
018         }
019         /// <summary>
020         /// 构造方法，初始化第一个节点
021         /// </summary>
022         public List()
023         {
024             id = new InnerData();
025             id.i = null;
026         }
027         /// <summary>
028         /// 向链表中添加数据
029         /// </summary>
030         /// <param name="t">添加进来的数据</param>
031         public void AddData(T t)
032         {
033             if (id.i == null)
034             {
035                 id.Data = t;
036                 InnerData temp = id;
037                 id = new InnerData();
038                 id.i = temp;
039             }
040             else
041             {
042                 InnerData temp = id;
043                 id.Data = t;
044                 id = new InnerData();
045                 id.i = temp;
046             }
047         }
048     }
```

```
048      /// <summary>
049      /// 将指向链表的引用修改为下一个引用
050      /// </summary>
051      public void Remove()
052      {
053          id = id.i;
054      }
055      /// <summary>
056      /// 输出链表中全部数据
057      /// </summary>
058      public void Show()
059      {
060          InnerData temp = id.i;
061          for (; ; )
062          {
063              Console.WriteLine(temp.Data);
064              temp = temp.i;
065              if (temp == null)
066              {
067                  return;
068              }
069          }
070      }
071  }
072  class Person
073  {
074      public string Name
075      {
076          get;
077          set;
078      }
079      public string Age
080      {
081          get;
082          set;
083      }
084      public string Occupation
085      {
086          get;
087          set;
088      }
089      public Person(string name, string age, string occupation)
090      {
091          this.Name = name;
```

```

092         this.Age = age;
093         this.Occupation = occupation;
094     }
095     /// <summary>
096     /// 重写 object 类的 ToString 方法，让链表调用输出
097     /// </summary>
098     /// <returns></returns>
099     public override string ToString()
100     {
101         return string.Format("姓名: {0} 年龄: {1} 职业: {2}", Name, Age, Occupation);
102     }
103 }
104 class Program
105 {
106     static void Main(string[] args)
107     {
108         List<int> myList = new List<int>();
109         myList.AddData(100);
110         myList.AddData(200);
111         myList.AddData(300);
112         myList.Show();
113         myList.Remove();
114         myList.Show();
115         Person p1 = new Person("郭靖", "40", "襄阳守备");
116         Person p2 = new Person("黄蓉", "36", "郭靖之妻");
117         Person p3 = new Person("杨过", "30", "神雕大侠");
118         List<Person> myList1 = new List<Person>();
119         myList1.AddData(p1);
120         myList1.AddData(p2);
121         myList1.AddData(p3);
122         myList1.Show();
123         Console.ReadKey();
124     }
125 }
126 }

```

这段代码创建了一个链表的泛型类，在泛型类中包含了一个非泛型的内部类表示一个链表的节点。第 6 行的代码定义了一个内部节点的引用。注意它不是实际存储的数据，它只是指向了堆内存中的实际数据。第 7 行是表示链表的节点的内部类，它含有一个 T 类型的属性和一个内部类的引用，这个引用的作用是指向下一个节点。第 22 行是链表的构造方法，它用来创建一个链表，并创建第一个节点。第一个节点因为没有指向任何其它节点，所以它的节点引用为 null。第 31 行是向链表中添加数据的操作，添加的数据作为参数传递进来。首先要判断节点内的引用是否为 null，如果为 null，则说明这是链表的第一个节点。因此，就把数据赋值给它，然后创建一个临时的节点引用，将当前节点的引用赋值给它，这样这个临时的节点引用也指向了当前节点。然后为当前节点的引用创建一个新节点，将这个新节点中的引用指向原来的节点。这样，两个节点就像一个链条一样链接了起来。

如果当前节点的引用不为 null，则说明当前节点不是链表的第一个节点。在这种情况下，首先建立一个临时节点引用指向当前节点，然后为当前节点的数据赋值。接下来新建一个节点，将新节点的引用指向原来的节点。

第 58 行的实例方法 Show 打印链表中的所有数据。因为每次添加数据的时候，都会新建一个没有数据的空节点，所以在遍历链表中的内容的时候，需要首先建立一个临时节点指向当前节点的下一个节点。接下来进入一个无限循环，打印输出临时节点的数据。然后将临时节点的节点引用指向它的下一个节点，这样循环输出每个节点的数据，直到节点的引用为 null，说明到了链表中的节点的最后一个节点。接下来使用 return 语句返回。

为测试链表对各种数据的泛型支持，第 72 行定义了一个类 Person。为支持链表泛型类中的输出操作，需要重写 object 的 ToString 方法。

从第 108 行开始，就是使用不同的类型创建链表的构造类型，然后为其添加和删除数据并进行打印。这段代码的执行结果如下：

```
300
200
100
200
100
姓名：郭靖 年龄：40 职业：襄阳守备
姓名：黄蓉 年龄：36 职业：郭靖之妻
姓名：杨过 年龄：30 职业：神雕大侠
```

泛型类的内部类也可以是一个泛型类。下面是代码实例：

```
001 using System;
002 namespace Generic2
003 {
004     class List<T,V> //链表的泛型类
005     {
006         private InnerData<V> id; //链表的节点
007         private class InnerData<U> //表示链表节点的内部类
008         {
009             /// <summary>
010             /// 节点内的数据
011             /// </summary>
012             public T Data
013             {
014                 get;
015                 set;
016             }
017             public InnerData<U> i; //节点含有的指向下一个节点的引用
018             public U ID
019             {
020                 get;
021                 set;
022             }
023         }
024     }
025 }
```

```
024      /// <summary>
025      /// 构造方法, 初始化第一个节点
026      /// </summary>
027      public List(V v)
028      {
029          id = new InnerData<V>();
030          id.i = null;
031          id.ID = v;
032      }
033      /// <summary>
034      /// 向链表中添加数据
035      /// </summary>
036      /// <param name="t">添加进来的数据</param>
037      public void AddData(T t)
038      {
039          if (id.i == null)
040          {
041              id.Data = t;
042              InnerData<V> temp = id;
043              id = new InnerData<V>();
044              id.i = temp;
045          }
046          else
047          {
048              InnerData<V> temp = id;
049              id.Data = t;
050              id = new InnerData<V>();
051              id.i = temp;
052          }
053      }
054      /// <summary>
055      /// 将指向链表的引用修改为下一个引用
056      /// </summary>
057      public void Remove()
058      {
059          id = id.i;
060      }
061      /// <summary>
062      /// 输出链表中全部数据
063      /// </summary>
064      public void Show()
065      {
066          InnerData<V> temp = id.i;
067          for (; ; )
```

```
068         {
069             Console.WriteLine(temp.Data);
070             temp = temp.i;
071             if (temp == null)
072             {
073                 return;
074             }
075         }
076     }
077     public void ShowID()
078     {
079         Console.WriteLine("链表封装数据类型为: {0}", id.ID.ToString());
080     }
081 }
082 class Person
083 {
084     public string Name
085     {
086         get;
087         set;
088     }
089     public string Age
090     {
091         get;
092         set;
093     }
094     public string Occupation
095     {
096         get;
097         set;
098     }
099     public Person(string name, string age, string occupation)
100     {
101         this.Name = name;
102         this.Age = age;
103         this.Occupation = occupation;
104     }
105     /// <summary>
106     /// 重写 object 类的 ToString 方法, 让链表调用输出
107     /// </summary>
108     /// <returns></returns>
109     public override string ToString()
110     {
111         return string.Format("姓名: {0} 年龄: {1} 职业: {2}", Name, Age, Occupation);
```

```

112     }
113 }
114 class Program
115 {
116     static void Main(string[] args)
117     {
118         List<int,string> myList = new List<int,string>("数据类型为 int");
119         myList.ShowID();
120         myList.AddData(100);
121         myList.AddData(200);
122         myList.AddData(300);
123         myList.Show();
124         myList.Remove();
125         myList.Show();
126         Person p1 = new Person("郭靖", "40", "襄阳守备");
127         Person p2 = new Person("黄蓉", "36", "郭靖之妻");
128         Person p3 = new Person("杨过", "30", "神雕大侠");
129         List<Person,int> myList1 = new List<Person,int>(1002);
130         myList1.ShowID();
131         myList1.AddData(p1);
132         myList1.AddData(p2);
133         myList1.Show();
134         Console.ReadKey();
135     }
136 }
137 }

```

这段代码中的 List 泛型类定义了两个类型形参，T 仍然用作链表的内部数据类型。V 用来向内部类传递类型实参。第 7 行定义的内部类有一个类型形参 U，它代表装入链表的数据类型的标识。需要注意，内部类如果也是泛型的，那么它的类型形参的标识符不能和外部类的相同。否则，内部类的类型形参将隐藏外部类的类型形参。那么该如何确定内部类的类型实参呢？可以通过用外部类的类型形参替换内部类的类型形参的方式。例如第 6 行代码，它使用外部类的类型形参，替换掉了内部类的类型形参。这意味着，一旦为外部类指定类型实参，则它会同时指定了内部类的类型实参。在后来的外部类中使用内部类的引用变量时，都用外部类的类型形参 V 替换掉内部类的类型形参 U。这样就完成了内部类型实参的间接指定。

在构造外部类的实例时，通过构造方法的参数为内部类的字段 ID 赋值。表示了链表装载的数据的类型描述。

所以，综合的操作可知，当泛型类的内部有一个泛型的内部类时，内部类不能和外部类使用相同的类型形参。否则，内部类的类型形参会隐藏掉外部类的类型形参。可以采用让外部类和内部类使用不相同的类型形参，但是在使用中进行映射传递的方式来为内部类指定类型实参。具体使用方法是，内部类的定义使用和外部类不同的类型形参。在外部类使用内部类时，用外部类的类型形参替换内部类的类型形参，例如第 6 行代码的使用方式。这段代码的执行结果如下：

链表封装数据类型为：数据类型为 int

```

300
200
100

```

200

100

链表封装数据类型为: 1002

姓名: 黄蓉 年龄: 36 职业: 郭靖之妻

姓名: 郭靖 年龄: 40 职业: 襄阳守备

对于内部类来说, 如果它是一个泛型, 而且访问条件许可, 那么就可以直接引用它来创建实例。代码如下:

```

001  using System;
002  namespace Generic2
003  {
004      class Test<T>
005      {
006          public class InnerTest<U>
007          {
008              private U a;
009              public InnerTest(U u)
010              {
011                  a = u;
012              }
013              public void Show()
014              {
015                  Console.WriteLine("内部类的字段 a 的值是: {0}", a);
016              }
017              public static void Show1()
018              {
019                  Console.WriteLine("内部类的 Show1 方法被调用");
020              }
021          }
022      }
023      class Program
024      {
025          static void Main(string[] args)
026          {
027              //Test<T>.InnerTest<int> myInnerTest = new Test<T>.InnerTest<int>(1000);
028              //myInnerTest.Show();
029              //Test<T>.InnerTest<int>.Show1();
030              Test<int>.InnerTest<int> myInnerTest1 = new Test<int>.InnerTest<int>(1000);
031              myInnerTest1.Show();
032              Test<int>.InnerTest<int>.Show1();
033              Console.ReadKey();
034          }
035      }
036  }

```

当在泛型类中有一个内部泛型类时, 可以直接指定类型实参对它进行引用。但是外部类的类型形参不

论是否有用，都需要进行类型实参的指定。如果将第 27 行到第 29 行代码都取消注释，则编译器会报错，错误信息如下：

错误 1 未能找到类型或命名空间名称“T” (是否缺少 using 指令或程序集引用?) H:\书稿 new\书稿源码\第十八章\Generic2\Generic2.cs 27 18 Generic2

错误 2 未能找到类型或命名空间名称“T” (是否缺少 using 指令或程序集引用?) H:\书稿 new\书稿源码\第十八章\Generic2\Generic2.cs 27 59 Generic2

错误 3 未能找到类型或命名空间名称“T” (是否缺少 using 指令或程序集引用?) H:\书稿 new\书稿源码\第十八章\Generic2\Generic2.cs 29 18 Generic2

代码的执行结果如下：

内部类的字段 a 的值是：1000

内部类的 Show1 方法被调用

第十九章 异常

异常用来处理程序系统级的和应用程序级的错误状态。它是一种结构化的，类型安全的处理机制。在 C# 中，所有异常都由 System.Exception 类派生。System.Exception 类是所有异常类的基类。在 C# 中，.Net 类库为各种异常都定义了相应的异常类，使得对于代码出错的判断变得非常简单。在代码的执行过程中，如果代码的执行出现了一些意外的情况，使某些操作无法完成，这时系统就会抛出异常，用来提示程序开发人员问题可能出现在哪个地方。异常是一个判断代码逻辑问题的工具。

19.1 try 语句

try 语句的作用是将认为可能出现问题的代码置于 try 块中，当问题发生时，代码的异常就会被捕捉到并进行处理。try 语句通常与 catch 语句或 finally 语句联用。下面是代码实例：

```
001 using System;
002 namespace Exception1
003 {
004     class Program
005     {
006         static void Main(string[] args)
007         {
008             int a = 10;
009             int b = 0;
010             try
011             {
012                 int c = a / b;
013                 Console.WriteLine(c);
014             }
015         }
016     }
017 }
```

这段代码定义了两个 int 类型的变量，然后使用了一个 try 语句。try 语句使用一个语句块将要检查的代码包裹起来。第 12 行代码和第 13 行代码都包裹在 try 块中，这样，当这两个语句出问题的时候，抛出的异常就会被捕捉到以便进行处理。但是处理并不是由这个 try 语句进行的。try 语句的意思就是本语句块可能会跑异常，下面准备处理吧。这段代码在编译时，编译器会报错错误，错误信息如下：

```
错误 1 应输入 catch 或 finallyH:\书稿 new\书稿源码\第十九章\Exception1\Program.cs 15 9
Exception1
```

从错误信息可以看出，try 语句是必须后接 catch 语句或 finally 语句的。

19.2 catch 语句

catch 语句是用来捕捉 try 语句抛出的异常，并可以使用一个语句块对异常进行处理。下面再补充前面的代码如下：

```
001 using System;
002 namespace Exception1
003 {
004     class Program
005     {
006         static void Main(string[] args)
007         {
```

```

008         int a = 10;
009         int b = 0;
010         try
011         {
012             int c = a / b;
013             Console.WriteLine(c);
014         }
015         catch (Exception e)
016         {
017             Console.WriteLine(e.Message);
018             Console.WriteLine(e.Source);
019             Console.WriteLine(e.StackTrace);
020             Console.ReadKey();
021         }
022     }
023 }
024 }

```

如果对 try 语句可能抛出哪类异常不熟悉,那么可以使用 exception 类的一个变量来接收抛出的异常。为什么可以这么做呢?因为 Exception 是所有异常类的基类,所有异常类都可以隐式转换到 Exception 类。通过它的变量引用,还可以实现多态的调用。第 15 行代码定义为捕捉所有的可能异常,然后可以像第 17 行代码到第 19 行代码那样调用 Exception 类的成员,分别打印异常信息、异常发生的源以及异常的堆栈信息。下面是代码的执行结果:

尝试除以零。

Exception1

在 Exception1.Program.Main(String[] args) 位置 h:\书稿 new\书稿源码\第十九章\Exception1\Program.cs:行号 12

虽然使用 Exception 类型的变量可以捕捉所有异常,但是这不是一种好方法。因为它就好比一张细密的网一样把代表异常信息的大大小小的鱼都给捕捉到了,但是却无法细分。实例代码如下:

```

001     using System;
002     namespace Exception1
003     {
004         class Program
005         {
006             static void Main(string[] args)
007             {
008                 int a = 10;
009                 int b = 0;
010                 try
011                 {
012                     //int c = a / b;
013                     //Console.WriteLine(c);
014                     int[] myArray = new int[3] { 0, 1, 3 };
015                     Console.WriteLine(myArray[3]);
016                 }

```

```

017         catch (Exception e)
018         {
019             Console.WriteLine(e.Message);
020             Console.WriteLine(e.Source);
021             Console.WriteLine(e.StackTrace);
022             Console.ReadKey();
023         }
024     }
025 }
026 }

```

在第 14 行代码的位置，定义了一个 int 类型的数组，它包含 3 个成员。第 15 行对数组的访问明显已经越界了。但是，如果不注释掉第 12 行代码和第 13 行代码的话，catch 语句捕捉到的异常信息还会和以前一样。数组越界的异常不会显露出来。只有把第 12 行和第 13 行代码都注释掉了，这时 catch 语句才会输出数组越界的信息。产生这个结果的原因是，当第 12 行发生异常之后，代码的逻辑就会立即跳转到后面的 catch 语句中，第 13 行、第 14 行和第 15 行代码都不会得到执行。执行结果如下：

索引超出了数组界限。

Exception1

在 Exception1.Program.Main(String[] args) 位置 h:\书稿 new\书稿源码\第十九章\Exception1\Program.cs:行号 15

可见，使用 Exception 类的变量捕捉异常并不是一种万能的办法。它会将所有异常都捕获下来。当代码可能产生不同的异常，而且应该对不同的异常进行处理时，这个方法就不适合。只有在没有其它捕捉办法的情况下才使用 Exception 类的变量捕捉异常。

19.3 finally 语句

finally 语句也可以和 try 语句联用，它的意思是，它的代码块中的语句是无论如何也要执行的。下面是代码实例：

```

001     using System;
002     namespace Exception1
003     {
004         class Program
005         {
006             static void Main(string[] args)
007             {
008                 int a = 10;
009                 int b = 0;
010                 try
011                 {
012                     int[] myArray = new int[3] { 0, 1, 3 };
013                     Console.WriteLine(myArray[3]);
014                 }
015                 catch (Exception e)
016                 {
017                     Console.WriteLine(e.Message);
018                     return;
019                 }

```

```
020         finally
021         {
022             a = 1000;
023             b = 2000;
024             Console.WriteLine("a 的值是: {0},b 的值是: {1}", a, b);
025             Console.ReadKey();
026         }
027     }
028 }
029 }
```

在这段代码中，第 13 行发生了数组的越界访问，它会抛出异常。第 15 行会捕捉到异常，并打印异常的信息。需要注意第 18 行，它使用 `return` 语句将控制返回到了调用方。这样看起来，到这行代码的时候，方法就应该终止了。但是，实际上没有发生这样的情况，因为后面还有一个 `finally` 语句。`finally` 语句是必须被执行的。即使在它的前面有 `return` 语句也不行。代码的执行结果如下：

索引超出了数组界限。

a 的值是: 1000,b 的值是: 2000

19.4 try-catch-finally 语句的联用

`try-catch-finally` 语句如果联用，可以实现比较严密的程序逻辑。在一些场合特别有用。例如，代码有一些非托管资源需要在程序结束前释放，但是程序有可能有比较多的异常会抛出。在这种情况下，就可以使用多个 `catch` 语句尽可能详细的捕捉异常。对于代码有可能异常终止的情况，可以使用 `finally` 语句保证非托管资源得到释放。下面是代码实例：

```
001 using System;
002 namespace Exception2
003 {
004     class Program
005     {
006         static void Main(string[] args)
007         {
008             int a = 10;
009             int b = 0;
010             try
011             {
012                 int c = a / b;
013             }
014             catch (DivideByZeroException e)
015             {
016                 Console.WriteLine(e.Message);
017             }
018             try
019             {
020                 int[] myArray = new int[3] { 0, 1, 3 };
021                 Console.WriteLine(myArray[3]);
022             }
023             catch (IndexOutOfRangeException e)
```

```

024      {
025          Console.WriteLine(e.Message);
026      }
027      finally
028      {
029          a = 1000;
030          b = 2000;
031          Console.WriteLine("a 的值是: {0},b 的值是: {1}", a, b);
032          Console.ReadKey();
033      }
034  }
035  }
036  }

```

在这段代码中，使用了多个 try 语句和多个 catch 语句将异常信息进行细分，但是它必须要求事先知道可能会抛出哪种异常。有一个办法就是根据代码输出的异常信息进行捕捉。如果实在不知道具体会抛出哪种异常，那就只好使用 Exception 基类。在这段代码中，对每个可能抛出异常的语句都使用了 try 语句，然后后跟处理的 catch 语句。在 catch 语句中写明具体的抛出异常的类型。然后在代码块中进行处理。对必须执行的代码，放置到 finally 块中。

这段代码的第 14 行和第 23 行都指定了具体的异常类型，它们都是 System.Exception 类的派生类。在 catch 语句中定义的异常类的变量 e 是一个局部变量，它的范围是它所在的整个 catch 块。超出这个 catch 块，它就会消失。所以下一个 catch 块还可以定义和使用局部变量名 e。这段代码的执行结果如下：

尝试除以零。

索引超出了数组界限。

a 的值是: 1000,b 的值是: 2000

try 语句只能接 catch 语句，多个 try 语句不能联用，但是多个 catch 语句可以一起作为可能的异常和一个 try 语句配合使用。如果对捕捉到的异常没有兴趣，而是只需要执行一些代码，那么这时可以使用不带任何参数的 catch 语句。这样的语句只能放在所有 catch 语句的最后面。下面是代码实例：

```

001  using System;
002  namespace Exception2
003  {
004      class Program
005      {
006          static void Main(string[] args)
007          {
008              int a = 10;
009              int b = 0;
010              try
011              {
012                  int[] myArray = new int[3] { 0, 1, 3 };
013                  Console.WriteLine(myArray[3]);
014              }
015              catch (DivideByZeroException e)
016              {
017                  Console.WriteLine(e.Message);

```

```

018         }
019         catch (IndexOutOfRangeException e)
020         {
021             Console.WriteLine(e.Message);
022         }
023         catch
024         {
025             Console.WriteLine("索引访问越界");
026         }
027         finally
028         {
029             a = 1000;
030             b = 2000;
031             Console.WriteLine("a 的值是: {0}, b 的值是: {1}", a, b);
032             Console.ReadKey();
033         }
034     }
035 }
036 }

```

在这段代码中, 第 10 行的 try 语句后跟了 3 个 catch 语句。因为它会抛出索引越界异常, 所以第 15 行的 catch 语句不会执行, 因为它捕捉不到 DivideByZeroException 类型的异常。它的下一个 catch 语句因为捕捉的是 IndexOutOfRangeException 类型的异常, 它的代码块将得以执行。它捕捉到了符合的异常, 则第 23 行的 catch 语句不执行。但是 finally 块将必须被执行。代码的执行结果如下:

索引超出了数组界限。

a 的值是: 1000, b 的值是: 2000

下面将第 19 行的 catch 语句都注释掉, 这样一来, 就没有一个明确的 IndexOutOfRangeException 类型的捕捉语句。这时, 最后的一个 catch 语句就会执行。代码执行结果如下:

索引访问越界

a 的值是: 1000, b 的值是: 2000

这种既 not 指定异常类型和异常变量的 catch 语句叫做常规 catch 语句。try 语句只能有一个 catch 语句, 并且这个 catch 语句只能是最后一个 catch 语句。需要注意, try 语句只能有一个 catch 语句是针对每个 try 语句而言, 并不是整个方法中只能有一个 catch 语句。下面看代码实例:

```

001 using System;
002 namespace Exception2
003 {
004     class Program
005     {
006         static void Main(string[] args)
007         {
008             int a = 10;
009             int b = 0;
010             try
011             {
012                 int c = a / b;

```

```
013     }
014     catch
015     {
016         Console.WriteLine("0 不能做除数");
017     }
018     try
019     {
020         int[] myArray = new int[3] { 0, 1, 3 };
021         Console.WriteLine(myArray[3]);
022     }
023     catch (DivideByZeroException e)
024     {
025         Console.WriteLine(e.Message);
026     }
027     catch
028     {
029         Console.WriteLine("索引访问越界");
030     }
031     finally
032     {
033         a = 1000;
034         b = 2000;
035         Console.WriteLine("a 的值是: {0}, b 的值是: {1}", a, b);
036         Console.ReadKey();
037     }
038 }
039 }
040 }
```

在这段代码中，针对每个 try 语句都有 catch 语句，而且它们都能正常的执行。代码执行结果如下：

0 不能做除数

索引访问越界

a 的值是: 1000, b 的值是: 2000

19.5 throw 语句

前面介绍过的语句都是用来捕获和处理异常，但是 throw 语句却用于引发一个异常。也就是说，一个异常的产生可以由 throw 语句引发。另外一种引发异常的方式就是像介绍过的除数为 0 的情况。这种是非正常的代码执行使操作无法完成而导致的。下面是代码实例：

```
001 using System;
002 namespace Exception2
003 {
004     class Program
005     {
006         static void Main(string[] args)
007         {
008             int a = 10;
```



```
009         int b = 0;
010         try
011         {
012             try
013             {
014                 int c = a / b;
015             }
016             catch (DivideByZeroException e)
017             {
018                 Console.WriteLine(e.Message);
019                 Console.WriteLine("再将这个异常抛出");
020                 throw;
021             }
022         }
023         catch
024         {
025             Console.WriteLine("0 真的作了除数");
026         }
027         try
028         {
029             try
030             {
031                 int[] myArray = new int[3] { 0, 1, 3 };
032                 Console.WriteLine(myArray[3]);
033             }
034             catch (IndexOutOfRangeException e)
035             {
036                 Console.WriteLine(e.Message);
037                 Console.WriteLine("再将这个异常抛出");
038                 throw;
039             }
040         }
041         catch
042         {
043             Console.WriteLine("索引访问真的越界了");
044         }
045         finally
046         {
047             a = 1000;
048             b = 2000;
049             Console.WriteLine("a 的值是: {0}, b 的值是: {1}", a, b);
050             Console.ReadKey();
051         }
052     }
```

```
053     }
054 }
```

在第 20 行和第 38 行代码中，catch 语句虽然已经捕捉到了明确的异常，但是可以使用不带任何表达式的 throw 语句将异常重新抛出交由它们后面的 catch 语句来捕捉处理。但是需要注意，如果要用 catch 语句来捕捉异常，则必须使用 try 块。因此在使用 throw 语句的 try-catch 语句之外，再使用一个 try 块将它包裹起来。这样才能再次使用 catch 语句捕获这个再次抛出的异常。代码的执行结果如下：

尝试除以零。

再将这个异常抛出

0 真的作了除数

索引超出了数组界限。

再将这个异常抛出

索引访问真的越界了

a 的值是：1000,b 的值是：2000

throw 语句还可以抛出自定义的异常，实例代码如下：

```
001 using System;
002 namespace Exception2
003 {
004     class Program
005     {
006         static void Main(string[] args)
007         {
008             int a = 10;
009             int b = 0;
010             try
011             {
012                 throw new IndexOutOfRangeException("数组索引越界了");
013             }
014             catch (IndexOutOfRangeException e)
015             {
016                 Console.WriteLine(e.Message);
017             }
018             try
019             {
020                 throw new NullReferenceException("这是一个空引用");
021             }
022             catch (NullReferenceException e)
023             {
024                 Console.WriteLine(e.Message);
025             }
026             try
027             {
028                 throw new BadImageFormatException("图像格式错误");
029             }
030             catch (BadImageFormatException e)
```

```

031      {
032          Console.WriteLine(e.Message);
033      }
034      finally
035      {
036          a = 1000;
037          b = 2000;
038          Console.WriteLine("a 的值是: {0},b 的值是: {1}", a, b);
039          Console.ReadKey();
040      }
041  }
042  }
043  }

```

在这段代码中，使用了 .Net 类库中定义好的异常类。使用 throw 语句抛出异常类的实例。每个异常类都有一个重载的构造方法可以传入一个异常信息描述字符串。所以，抛出异常的方法就是先定义一个指定异常类的实例，然后使用 throw 语句抛出。如果要捕捉到抛出的异常，然后处理，必须使用 try 语句包裹 throw 语句才行。这段代码的执行结果如下：

数组索引越界了

这是一个空引用

图像格式错误

a 的值是: 1000,b 的值是: 2000

19.6 自定义异常

在代码中也可以创建自定义异常，创建自定义异常类的时候，让自定义异常类从 Exception 类派生或指定的异常类派生。下面是代码实例：

```

001  using System;
002  namespace Exception2
003  {
004      class MyDivideByZeroException : DivideByZeroException
005      {
006          private string helpLink;
007          public override string HelpLink
008          {
009              get
010              {
011                  return helpLink;
012              }
013              set
014              {
015                  helpLink = value;
016              }
017          }
018          public override string Message
019          {
020              get

```

```
021         {
022             return "分母和分子不能同时为零";
023         }
024     }
025     public override string StackTrace
026     {
027         get
028         {
029             return base.StackTrace;
030         }
031     }
032     public MyDivideByZeroException()
033     {
034         HelpLink = "http://www.myNetc.com/a.html";
035     }
036     public override string Source
037     {
038         get
039         {
040             return this.GetType().FullName;
041         }
042         set
043         {
044             base.Source = value;
045         }
046     }
047 }
048 class Math
049 {
050     private int a;
051     private int b;
052     private int c;
053     public Math(int a, int b)
054     {
055         this.a = a;
056         this.b = b;
057     }
058     public int Div()
059     {
060         if (a == 0 && b == 0)
061         {
062             throw new MyDivideByZeroException();
063         }
064         if (b == 0)
```

```
065         {
066             throw new DivideByZeroException("除数不能为零，请确认除数");
067         }
068         c = a / b;
069         return c;
070     }
071 }
072 class Program
073 {
074     static void Main(string[] args)
075     {
076         Math m1 = new Math(100, 0);
077         int c = 0;
078         try
079         {
080             c = m1.Div();
081             Console.WriteLine("a 除以 b 的商为: {0}", c);
082         }
083         catch (MyDivideByZeroException e)
084         {
085             Console.WriteLine(e.Message);
086             Console.WriteLine(e.StackTrace);
087             Console.WriteLine(e.Source);
088             Console.WriteLine(e.HelpLink);
089         }
090         catch (DivideByZeroException e)
091         {
092             Console.WriteLine(e.Message);
093             Console.WriteLine(e.StackTrace);
094             Console.WriteLine(e.Source);
095             Console.WriteLine(e.HelpLink);
096         }
097         Math m2 = new Math(0, 0);
098         try
099         {
100             c = m2.Div();
101             Console.WriteLine("a 除以 b 的商为: {0}", c);
102         }
103         catch (MyDivideByZeroException e)
104         {
105             Console.WriteLine(e.Message);
106             Console.WriteLine(e.StackTrace);
107             Console.WriteLine(e.Source);
108             Console.WriteLine(e.HelpLink);
```

```

109         }
110         catch (DivideByZeroException e)
111         {
112             Console.WriteLine(e.Message);
113             Console.WriteLine(e.StackTrace);
114             Console.WriteLine(e.Source);
115             Console.WriteLine(e.HelpLink);
116         }
117         Console.ReadKey();
118     }
119 }
120 }

```

在代码的第 4 行,定义了一个 `MyDivideByZeroException` 类,它派生自异常类 `DivideByZeroException`。在这个自定义的异常类中,可以对基类的成员进行重写,以适应自己的程序的要求。第 6 行代码定义了一个字符串字段,这个字段的作用是自定义帮助信息的链接。第 7 行对基类的 `HelpLink` 属性进行了重写,让它的 `get` 访问器返回本类中的这个字段。第 18 行代码对基类的 `Message` 属性进行了重写,让它的 `get` 访问器返回自定义的字符串。对于 `StackTrace` 属性,因为基类的 `StackTrace` 属性信息就比较完整,因此可以让它返回基类的 `StackTrace` 属性。对于 `Source` 属性,因为基类的这个属性不能输出类名,所以重写它的 `get` 访问器,让它输出命名空间和本程序的类名。在第 32 行的构造方法中对 `HelpLink` 属性进行定义。

在第 48 行开始,定义了一个类,在类中对 `int` 类型的字段进行除法运算。在进行除法运算的 `Div` 方法中,要对参与运算的被除数和除数进行判断,如果被除数和除数同时为 0,则抛出自定义的异常的实例。如果只有除数为 0,则抛出异常基类的实例。在 `Main` 方法中,如果要捕捉异常,则需要使用 `try` 语句将能够抛出异常的方法包裹起来,然后使用 `catch` 语句捕获异常并进行处理。在捕获异常的时候,一定要注意把自定义的捕获异常的 `catch` 语句放在前面,把捕获它的基类异常的 `catch` 语句放在后面。这样才能保证异常不被自定义异常的基类全部捕获。这段代码的执行结果如下:

除数不能为零,请确认除数

在 `Exception2.Math.Div()` 位置 `h:\书稿 new\书稿源码\第十九章\Exception2\Program.cs`:行号 66

在 `Exception2.Program.Main(String[] args)` 位置 `h:\书稿 new\书稿源码\第十九章\Exception2\Program.cs`:行号 80

`Exception2`

分母和分子不能同时为零

在 `Exception2.Math.Div()` 位置 `h:\书稿 new\书稿源码\第十九章\Exception2\Program.cs`:行号 62

在 `Exception2.Program.Main(String[] args)` 位置 `h:\书稿 new\书稿源码\第十九章\Exception2\Program.cs`:行号 100

`Exception2.MyDivideByZeroException`

`http://www.myNetc.com/a.html`

从执行结果中可以看到,当被除数和除数同时为 0 的时候,输出的是自定义异常的信息。

19.7 异常处理中的标签语句

如果在 `try` 块中有 `goto` 语句,而且 `try` 块和 `finally` 语句联用的情况下,如果 `goto` 语句意图把代码的执行跳转到 `try` 块之外的标记语句处,并且 `finally` 块的结束点不可到达,则代码会出现循环的现象。代码实例如下:

```

001 using System;
002 namespace Exception2
003 {
004     class Program
005     {
006         static void Main(string[] args)
007         {
008             int a = 100;
009             int b = 0;
010             int c = 0;
011             Label1:
012             Console.WriteLine("此处是 Label1 标签");
013             try
014             {
015                 goto Label1;
016                 c = a / b;
017             }
018             catch (DivideByZeroException e)
019             {
020                 Console.WriteLine(e.Message);
021             }
022             finally
023             {
024                 Console.WriteLine("此语句必须被执行");
025             }
026             Console.ReadKey();
027         }
028     }
029 }

```

在这段代码中，第 15 行的 goto 语句跳转到第 11 行的标签语句处，造成代码的循环。这时 finally 块的内容也不能正常执行。编译器会报告一个警告信息，信息如下：

```

警告 1    检测到无法访问的代码    H:\书稿 new\书稿源码\第十九章\Exception2\Program.cs 16    17
Exception2

```

19.8 异常处理中的 break 语句

对于 break 语句来说，break 语句不能退出 finally 块。因为它是用作循环体的语句。实例代码如下：

```

001 using System;
002 namespace Exception1
003 {
004     class Program
005     {
006         static void Main(string[] args)
007         {
008             int a = 10;
009             int b = 0;

```

```

010         try
011         {
012             int c = a / b;
013         }
014         catch (DivideByZeroException e)
015         {
016             Console.WriteLine(e.Message);
017         }
018         finally
019         {
020             break;
021             a = 1000;
022             b = 2000;
023             Console.WriteLine("a 的值是: {0}, b 的值是: {1}", a, b);
024         }
025         try
026         {
027             int[] myArray = new int[3] { 0, 1, 3 };
028             Console.WriteLine(myArray[3]);
029         }
030         catch (IndexOutOfRangeException e)
031         {
032             Console.WriteLine(e.Message);
033         }
034         finally
035         {
036             Console.WriteLine("此语句必须被执行");
037             Console.ReadKey();
038         }
039     }
040 }
041 }

```

在这段代码中, 对于 try 语句来说, 有两个对应的 finally 块。在第 18 行的 finally 块中, 使用了 break 语句意图退出 finally 块, 但是编译器会报告错误, 错误信息如下:

```

错误 1  没有要中断或继续的封闭循环 H:\书稿 new\书稿源码\第十九章\Exception1\Program.cs 20
17  Exception1

```

如果 break 语句和包含它的循环语句必须位于 finally 块中, 则它退出的位置也必须位于同一个 finally 块中。实例代码如下:

```

001     using System;
002     namespace Exception1
003     {
004         class Program
005         {
006             static void Main(string[] args)

```



```
007      {
008          int a = 10;
009          int b = 0;
010          try
011          {
012              int c = a / b;
013          }
014          catch (DivideByZeroException e)
015          {
016              Console.WriteLine(e.Message);
017          }
018          finally
019          {
020              for (int i = 0; i < 10; i++)
021              {
022                  if(i == 5)
023                      break;
024              }
025              a = 1000;
026              b = 2000;
027              Console.WriteLine("a 的值是: {0},b 的值是: {1}", a, b);
028          }
029          try
030          {
031              int[] myArray = new int[3] { 0, 1, 3 };
032              Console.WriteLine(myArray[3]);
033          }
034          catch (IndexOutOfRangeException e)
035          {
036              Console.WriteLine(e.Message);
037          }
038          finally
039          {
040              Console.WriteLine("此语句必须被执行");
041              Console.ReadKey();
042          }
043      }
044  }
045 }
```

在代码的第 20 行,定义了一个 for 语句循环,当变量 i 等于 5 的时候,使用 break 语句退出循环。因为退出循环后,代码的执行逻辑仍在这个 finally 语句中,所以这样的语法是允许的。代码的执行结果如下:

尝试除以零。

a 的值是: 1000,b 的值是: 2000

索引超出了数组界限。

此语句必须被执行

如果使用 break 语句退出 try 块的时候，每个 try 块都有对应的 finally 块。这时，代码逻辑从最里层的 finally 块开始执行，逐步向外执行 finally 块。代码实例如下：

```
001     using System;
002     namespace Exception2
003     {
004         class Program
005         {
006             static void Main(string[] args)
007             {
008                 for (int i = 0; i < 1; i++)
009                 {
010                     try
011                     {
012                         try
013                         {
014                             try
015                             {
016                                 break;
017                                 throw new IndexOutOfRangeException();
018                             }
019                             catch (IndexOutOfRangeException e)
020                             {
021                                 throw;
022                             }
023                             finally
024                             {
025                                 Console.WriteLine("最里层的 finally 语句被执行");
026                             }
027                         }
028                         catch (IndexOutOfRangeException e)
029                         {
030                             throw;
031                         }
032                         finally
033                         {
034                             Console.WriteLine("中间层的 finally 语句被执行");
035                         }
036                     }
037                     catch (IndexOutOfRangeException e)
038                     {
039                         Console.WriteLine("最外层的 catch 语句被执行");
040                     }
041                 }
042             }
043         }
044     }
```

```

041         finally
042         {
043             Console.WriteLine("最外层的 finally 语句被执行");
044         }
045     }
046     Console.ReadKey();
047 }
048 }
049 }

```

需要注意，`break` 语句只能用来跳出循环，因此，本段代码的情景发生在循环的内部有 `try-catch-finally` 语句。`finally` 语句必须和 `try` 配合使用。每个 `try` 语句都有一个 `finally` 语句作最保守的代码执行。因此，如果想要 `finally` 语句执行，必须其对应的 `try` 是可达的。`finally` 语句必须被执行的前提是其对应的 `try` 可执行。而并不是只要代码中有 `finally` 语句，这个 `finally` 语句就一定要执行。这段代码在最里层的 `try` 块中，第 16 行定义了一个 `break` 语句。这样就保证了所有 `try` 语句都可被执行。当代码执行 `break` 语句的时候，它将退出包裹所有 `try-catch-finally` 块的 `for` 语句。在这种情况下，最里层的 `finally` 块先执行，然后依次向外执行 `finally` 语句。代码的执行结果如下：

```

最里层的 finally 语句被执行
中间层的 finally 语句被执行
最外层的 finally 语句被执行

```

19.9 异常处理中的 `continue` 语句

`continue` 语句因为是用于循环的，所以它同样不能退出 `finally` 块。当循环语句和 `continue` 语句位于 `finally` 块中时，它的退出目标也必须位于同一个 `finally` 块中，否则会发生编译时错误。实例代码如下：

```

001 using System;
002 namespace Exception1
003 {
004     class Program
005     {
006         static void Main(string[] args)
007         {
008             for (int i = 0; i < 5; i++)
009             {
010                 try
011                 {
012                     throw new IndexOutOfRangeException();
013                 }
014                 catch (IndexOutOfRangeException e)
015                 {
016                     Console.WriteLine(e.Message);
017                 }
018                 finally
019                 {
020                     continue;
021                     Console.WriteLine("执行 finally 块中的语句");

```

```

022         }
023     }
024     Console.ReadKey();
025 }
026 }
027 }

```

在这段代码中,第 8 行开始的 for 语句体中是不断循环抛出的异常和处理异常的 catch 语句和 finally 语句。当代码执行第一次循环的时候,执行到第 20 行代码,就会马上跳出,开始执行下一次循环。它后面的语句将不被执行。在语法上,这是不允许的。编译器会报错,错误信息如下:

```

错误 1    控制不能离开 finally 子句主体    E:\书稿new\书稿源码\第十九章\Exception1\Program.cs
          20  21  Exception1
警告 2    检测到无法访问的代码    E:\书稿 new\书稿源码\第十九章\Exception1\Program.cs 21  21
          Exception1

```

伴随着错误,还有一个警告信息。因为 continue 语句后面的语句不能执行。

continue 语句如果位于多重嵌套的 try 语句中,而且 try 语句有对应的 finally 语句,则 finally 语句的执行逻辑和前面介绍过的带有 break 语句的 try 语句对应的 finally 语句的执行逻辑是一样的。实例代码如下:

```

001  using System;
002  namespace Exception2
003  {
004      class Program
005      {
006          static void Main(string[] args)
007          {
008              for (int i = 0; i < 1; i++)
009              {
010                  try
011                  {
012                      try
013                      {
014                          try
015                          {
016                              continue;
017                              throw new IndexOutOfRangeException();
018                          }
019                          catch (IndexOutOfRangeException e)
020                          {
021                              throw;
022                          }
023                          finally
024                          {
025                              Console.WriteLine("最里层的 finally 语句被执行");
026                          }
027                      }
028                  }
029              }
030          }
031      }
032  }

```

```

028             catch (IndexOutOfRangeException e)
029             {
030                 throw;
031             }
032             finally
033             {
034                 Console.WriteLine("中间层的 finally 语句被执行");
035             }
036         }
037         catch (IndexOutOfRangeException e)
038         {
039             Console.WriteLine("最外层的 catch 语句被执行");
040         }
041         finally
042         {
043             Console.WriteLine("最外层的 finally 语句被执行");
044         }
045     }
046     Console.ReadKey();
047 }
048 }
049 }

```

这段代码是修改自前面介绍过的代码，因此代码的逻辑不再多说。这段代码中使用了 continue 语句。而包裹它的 for 语句只循环一次。再做第一次循环的时候，执行到 continue 语句，代码逻辑会立刻跳出。在这个过程中，finally 语句按照从里到外的原则依次执行。当所有的 finally 语句都执行完毕后，局部变量 i 会自增，然后再判断 i 是否小于 1，因为不符合条件，所以循环结束。第 46 行代码被执行。代码的执行结果如下：

```

最里层的 finally 语句被执行
中间层的 finally 语句被执行
最外层的 finally 语句被执行

```

19.10 异常处理中的 goto 语句

goto 语句也不能用来退出 finally 块，实例代码如下：

```

001 using System;
002 namespace Exception1
003 {
004     class Program
005     {
006         static void Main(string[] args)
007         {
008             Label1:
009                 Console.WriteLine("此处是 Label1 标签");
010                 try
011                 {
012                     throw new IndexOutOfRangeException();

```

```

013         }
014         catch (IndexOutOfRangeException e)
015         {
016             Console.WriteLine(e.Message);
017         }
018         finally
019         {
020             goto Label1;
021             Console.WriteLine("执行 finally 块中的语句");
022         }
023         Console.ReadKey();
024     }
025 }
026 }

```

在这段代码的第 8 行，有一个标签语句。在第 20 行的 finally 块中有一个 goto 语句，它跳转到第 8 行的标签语句处。当代码执行的时候，第 9 行的代码会得到正常执行。但是当执行 finally 语句块中的语句时，因为 goto 语句的存在，它后面的代码将不会得到执行。这属于语法上的错误，编译器会报告错误。错误信息如下：

```

错误 1    控制不能离开 finally 子句主体    E:\书稿new\书稿源码\第十九章\Exception1\Program.cs
          20  21  Exception1
警告 2    检测到无法访问的代码    E:\书稿 new\书稿源码\第十九章\Exception1\Program.cs 21  21
          Exception1

```

同时，编译器也会给出一个警告，警告第 21 行的代码不会得到执行，它是无法访问的代码。

与 continue 语句一样，goto 语句也可以位于多重嵌套的 try 语句中。实例代码如下：

```

001 using System;
002 namespace Exception2
003 {
004     class Program
005     {
006         static void Main(string[] args)
007         {
008             int j = 0;
009             Label1:
010             if (j == 5)
011             {
012                 Console.ReadKey();
013                 return;
014             }
015             for (int i = 0; i < 1; i++)
016             {
017                 try
018                 {
019                     try
020                     {

```

```
021         try
022         {
023             j = 5;
024             goto Label1;
025             throw new IndexOutOfRangeException();
026         }
027         catch (IndexOutOfRangeException e)
028         {
029             throw;
030         }
031         finally
032         {
033             Console.WriteLine("最里层的 finally 语句被执行");
034         }
035     }
036     catch (IndexOutOfRangeException e)
037     {
038         throw;
039     }
040     finally
041     {
042         Console.WriteLine("中间层的 finally 语句被执行");
043     }
044 }
045 catch (IndexOutOfRangeException e)
046 {
047     Console.WriteLine("最外层的 catch 语句被执行");
048 }
049 finally
050 {
051     Console.WriteLine("最外层的 finally 语句被执行");
052 }
053 }
054 }
055 }
056 }
```

代码的第 8 行定义了一个局部变量，第 9 行是一个标签语句。第 10 行对局部变量进行判断，如果符合条件，则 Main 方法退出。在第 23 行代码，对局部变量 j 进行赋值。让它符合退出条件。第 24 行使用 goto 语句跳转到外面的 Label1 标签语句处。这时嵌套的 finally 语句开始从里层向外层执行。执行的顺序与前面介绍过的 continue 语句一样。代码的执行结果如下：

```
最里层的 finally 语句被执行
中间层的 finally 语句被执行
最外层的 finally 语句被执行
```

19.11 异常处理中的 return 语句

对于异常处理来说，return 语句不能出现在 finally 块中，因为 finally 块是必须执行完毕的。下面是代码实例：

```

001 using System;
002 namespace Exception1
003 {
004     class Program
005     {
006         static void Main(string[] args)
007         {
008             try
009             {
010                 throw new IndexOutOfRangeException();
011             }
012             catch (IndexOutOfRangeException e)
013             {
014                 Console.WriteLine(e.Message);
015             }
016             finally
017             {
018                 return;
019             }
020             Console.ReadKey();
021         }
022     }
023 }

```

对于 finally 块来说，如果其中有 return 语句，不论 return 语句后面有没有其它语句，编译器都会报告错误。错误信息如下：

```

错误 1    控制不能离开 finally 子句主体    E:\书稿new\书稿源码\第十九章\Exception1\Program.cs
        18  21  Exception1

```

如果 return 语句位于嵌套的 try 和 finally 块中，那么 finally 的执行顺序和前面介绍过的 continue 等语句一样。实例代码如下：

```

001 using System;
002 namespace Exception2
003 {
004     class Program
005     {
006         static void Main(string[] args)
007         {
008             for (int i = 0; i < 1; i++)
009             {
010                 try
011                 {
012                     try
013                     {

```



```
014         try
015         {
016             return;
017             throw new IndexOutOfRangeException();
018         }
019         catch (IndexOutOfRangeException e)
020         {
021             throw;
022         }
023         finally
024         {
025             Console.WriteLine("最里层的 finally 语句被执行");
026         }
027     }
028     catch (IndexOutOfRangeException e)
029     {
030         throw;
031     }
032     finally
033     {
034         Console.WriteLine("中间层的 finally 语句被执行");
035     }
036 }
037 catch (IndexOutOfRangeException e)
038 {
039     Console.WriteLine("最外层的 catch 语句被执行");
040 }
041 finally
042 {
043     Console.WriteLine("最外层的 finally 语句被执行");
044     Console.ReadKey();
045 }
046 }
047 }
048 }
049 }
```

第 16 行代码中, return 语句位于 try 块中, 为了让控制台的显示暂停, 第 44 行的最外层的 finally 块中加入了 Console.ReadKey() 语句。当代码执行到第 16 行的时候, 因为要退出 Main 方法, 所以所有的 finally 语句要从内向外执行。代码的执行结果如下:

```
最里层的 finally 语句被执行
中间层的 finally 语句被执行
最外层的 finally 语句被执行
```

return 语句也可以位于 catch 块中, 如果将这段代码中的 return 语句挪到第 19 行的 catch 块中, 让它位于 throw 语句的前面。这时, finally 语句的执行顺序和前面的结果一样。

第二十章 特性与反射

特性使得程序员可以为程序中的程序实体定义声明性的信息。例如，在定义类中的成员时，可以指定其访问权限修饰符。这些修饰符可以看做是声明性的信息。特性也与它们类似。特性附加到类或方法上之后，运行时环境就可以检索这些特性信息。在前面章节介绍过使用 C++ 的动态链接库，需要使用一个 [DllImport("***.dll")] 的特性。特性实质上就是一个类。类似 DllImport("***.dll") 这样的语法实际上就是调用特性类的构造方法创建一个特性类的实例。

20.1 特性的定义

下面通过一个代码实例来说明如何定义特性。代码实例如下：

```

001 using System;
002 namespace Attribute1
003 {
004     class AppInfo : Attribute //自定义特性类需从 Attribute 类派生
005     {
006         /// <summary>
007         /// 表示作者姓名
008         /// </summary>
009         public string Name
010         {
011             get;
012             set;
013         }
014         /// <summary>
015         /// 程序集版本
016         /// </summary>
017         public string Version
018         {
019             get;
020             set;
021         }
022         /// <summary>
023         /// 帮助信息
024         /// </summary>
025         public string HelpInfo
026         {
027             get;
028             set;
029         }
030         /// <summary>
031         /// 构造方法
032         /// </summary>
033         /// <param name="name"></param>
034         public AppInfo(string name)
035         {

```

```
036         Name = name;
037     }
038 }
039 //为 Person 类指定特性
040 [AppInfo("Tom And Jarry",Version = "1.0",HelpInfo = "http://www.aaa.com/help.html")]
041 class person
042 {
043     public string Name
044     {
045         get;
046         set;
047     }
048     public string ID
049     {
050         get;
051         set;
052     }
053     public int Age
054     {
055         get;
056         set;
057     }
058     public void Print()
059     {
060         Console.WriteLine("姓名: {0}, 年龄: {1}, 身份证号: {2}", Name, Age, ID);
061     }
062 }
063 class Student
064 {
065     public void Sing()
066     {
067         Console.WriteLine("我会唱歌");
068     }
069     public void Dance()
070     {
071         Console.WriteLine("我会跳舞");
072     }
073 }
074 class Program
075 {
076     static void OutPutInfo(Type t)
077     {
078         Console.WriteLine("现在开始通过反射获取 {0} 类的特性信息", t.ToString());
079         Attribute[] attArray = Attribute.GetCustomAttributes(t);
```

```

080         foreach (Attribute att in attArray)
081         {
082             if (att is AppInfo)
083             {
084                 AppInfo ai = (AppInfo)att;
085                 Console.WriteLine("作者: {0}, 版本: {1}, 帮助信息: {2}", ai.Name,
                                ai.Version, ai.HelpInfo);
086             }
087         }
088     }
089     static void Main(string[] args)
090     {
091         OutPutInfo(typeof(person));
092         OutPutInfo(typeof(Student));
093         Console.ReadKey();
094     }
095 }
096 }

```

在这段代码的第 4 行定义了一个自定义特性类 AppInfo。自定义特性类需要从 System.Attribute 基类派生。第 6 行到第 29 行代码定义了三个属性，分别表示程序集的作者、版本和帮助信息。第 34 行代码是构造方法，它只定义了一个形参 name，用来为 Name 属性赋值。

第 41 行代码定义了一个类 Person，它的定义很简单，定义了三个属性和一个实例方法。实例方法 Print 用来输出三个属性。

第 63 行又定义了一个类 Student，它定义了两个实例方法，分别表示学生能够唱歌和跳舞。

在第 40 行代码，对类 Person 指定了特性。应用特性需要在指定的程序实体上使用一对中括号。在中括号中首先指定特性名称，然后跟一对小括号，在小括号中第一个参数是指定的特性的构造方法的参数实参，后两个赋值语句是引用特性类的公共读写属性提供附加信息。需要注意，特性类中的属性成员必须是读写属性。

为类 Student 没有指定特性。

现在来看如何通过反射的技术来在代码中使用特性。特性定义好后，却不通过反射获取特性的信息，则特性的定义就没有任何意义。例如前面介绍过的，使用特性引用 C++ 的动态链接库的例子中。对 .Net 类库中 DllImport 特性的反射获取就是由 C# 的编译器来执行的，它能影响编译器的编译工作。现在来看如何获取特性的信息。在第 76 行定义了一个静态方法 OutPutInfo，它定义了一个 Type 类型的变量 t。Type 类前面已经接触过了，它是用来获取类型信息的类。获取特性信息要使用 Attribute.GetCustomAttributes 方法，它返回一个类型上应用的特性的数组。特性的数组是 Attribute 类型，每个数组的成员就是指定类上应用的特性实例。第 80 行通过 foreach 语句获取每个数组的元素。第 82 行判断特性是否是自定义的 AppInfo 类，如果符合，则将 Attribute 类型转换为 AppInfo 类型。然后依次打印输出其成员。

在 Main 方法中，调用静态方法 OutPutInfo，并使用 typeof 运算符得到类的类型并作为参数传递给 OutPutInfo 方法。在 Main 方法中调用两次 OutPutInfo 方法，来获取类 Person 和类 Student 的特性信息。因为 Student 类没有指定特性，所以没有特性信息显示。代码的执行结果如下：

现在开始通过反射获取 Attribute1.person 类的特性信息

作者: Tom And Jarry, 版本: 1.0, 帮助信息: <http://www.aaa.com/help.html>

现在开始通过反射获取 Attribute1.Student 类的特性信息

20.2 反射

对特性的使用是通过反射的机制实现的。那么什么是反射呢？通过反射机制可以提供描述程序集、模块和类型的实例。这个实例就是 `Type` 类型的实例。其实前面的章节已经接触过了。如果代码中定义了特性，就可以通过反射来访问它们。反射主要使用 `System.Reflection` 命名空间和 `System.Type` 类型。其实 `System.Type` 类就是继承自 `System.Reflection` 命名空间中的 `MemberInfo` 类，这个类的作用是获取有关成员属性的信息并提供对成员元数据的访问。实际上，在类中定义的特性，在编译的时候就是以元数据的形式插入到程序集的。那么，什么又是元数据呢？其实元数据就是一种二进制的信息，它在编译的时候嵌入到程序集中。当执行程序集的时候，运行时环境就将元数据加载到内存中。并引用它的实例来发现有关代码的类、成员、继承等信息。程序集中的元数据存储以下类型的信息：

- 程序集的说明。
 1. 标识（名称、版本、区域性、公钥）。
 2. 导出的类型。
 3. 该程序集所依赖的其他程序集。
 4. 运行所需的安全权限。
- 类型的说明。
 1. 名称、可见性、基类和实现的接口。
 2. 成员（方法、字段、属性、事件、嵌套的类型）。
- 特性。

修饰类型和成员的其他说明性元素。

下面使用一段代码来说明元数据的作用。代码如下：

```

001 using System;
002 namespace Attribute2
003 {
004     class Program
005     {
006         public int Add(int a, int b)
007         {
008             return a + b;
009         }
010         static void Main(string[] args)
011         {
012             int a = 10;
013             int b = 20;
014             Program p = new Program();
015             Console.WriteLine("a + b 的值是: {0}", p.Add(a, b));
016             Console.ReadKey();
017         }
018     }
019 }

```

这段代码很简单，在类中定义了一个普通的实例方法，并在 `Main` 方法中进行了调用。下面使用 `ildasm` 工具打开编译好的程序集。并在 `ildasm` 中的“视图”菜单中选中“显示标记值”项。接下来打开 `Main` 方法的 MSIL 中间代码。代码如下：

```

.method /*06000002*/ private hidebysig static
    void Main(string[] args) cil managed
{

```

```

.entrypoint
// 代码大小      44 (0x2c)
.maxstack 4
.locals /*11000002*/ init ([0] int32 a,
    [1] int32 b,
    [2] class Attribute2.Program/*02000002*/ p)
IL_0000: nop
IL_0001: ldc.i4.s 10
IL_0003: stloc.0
IL_0004: ldc.i4.s 20
IL_0006: stloc.1
IL_0007: newobj instance void Attribute2.Program/*02000002*/::.ctor() /* 06000003 */
IL_000c: stloc.2
IL_000d: ldstr bytearray (61 00 20 00 2B 00 20 00 62 00 84 76 3C 50 2F 66 //
a. .+. .b..v<P/f
    1A FF 7B 00 30 00 7D 00 ) // ..{.0.}. /* 70000001 */
IL_0012: ldloc.2
IL_0013: ldloc.0
IL_0014: ldloc.1
IL_0015: callvirt instance int32 Attribute2.Program/*02000002*/::Add(int32,
    int32) /* 06000001 */
IL_001a: box [mscorlib/*23000001*/]System.Int32/*01000013*/
IL_001f: call void
[mscorlib/*23000001*/]System.Console/*01000014*/::WriteLine(string,object) /* 0A000011 */
IL_0024: nop
IL_0025: call valuetype [mscorlib/*23000001*/]System.ConsoleKeyInfo/*01000015*/
[mscorlib/*23000001*/]System.Console/*01000014*/::ReadKey() /* 0A000012 */
IL_002a: pop
IL_002b: ret
} // end of method Program::Main

```

在这段代码的 IL_0015 标记处，就是 Main 方法中对 Add 方法的调用。可以看到，在其后有一个/*06000001 */的标记。这个标记就是元数据标记。它的意思是让运行时环境参考 MethodDef 表的第一行。MethodDef 表是一个元数据表，它也位于程序集中。在这个表中定义了方法的相对虚拟地址和访问权限等信息。运行时环境依据这些信息计算该方法的起始内存地址等信息。从这个例子可以看出，元数据在代码编译时起着非常重要的作用。

下面再通过一个例子使用反射类来分析程序集中的元数据。代码如下：

```

001 using System;
002 using System.Reflection;
003 namespace Attribute2
004 {
005     class Test
006     {
007         public void ShowMethods(Type t)
008     {

```

```
009         Console.WriteLine("通过反射得到的方法信息");
010         MethodInfo[] myMethods = t.GetMethods();
011         int count = 0;
012         foreach (MethodInfo m in myMethods)
013         {
014             count++;
015             Console.WriteLine(count.ToString() + "." + m.Name);
016         }
017     }
018     public void ShowFields(Type t)
019     {
020         Console.WriteLine("通过反射得到的字段成员信息");
021         FieldInfo[] myFields = t.GetFields();
022         int count = 0;
023         foreach (FieldInfo f in myFields)
024         {
025             count++;
026             Console.WriteLine(count.ToString() + "." + f.Name);
027         }
028     }
029     public void ShowProps(Type t)
030     {
031         Console.WriteLine("通过反射得到的属性成员信息");
032         PropertyInfo[] myProps = t.GetProperties();
033         int count = 0;
034         foreach (PropertyInfo p in myProps)
035         {
036             count++;
037             Console.WriteLine(count.ToString() + "." + p.Name);
038         }
039     }
040     public void ShowInterfaces(Type t)
041     {
042         Console.WriteLine("通过反射得到的接口信息");
043         Type[] myInterfaces = t.GetInterfaces();
044         int count = 0;
045         foreach (Type t1 in myInterfaces)
046         {
047             count++;
048             Console.WriteLine(count.ToString() + "." + t1.Name);
049         }
050     }
051     public void ShowClassInfo(Type t)
052     {
```

```
053         Console.WriteLine("通过反射得到的本类信息");
054         Console.WriteLine("本类派生自: {0}", t.BaseType.Name);
055         if (t.IsClass)
056         {
057             Console.WriteLine("此类是类类型");
058         }
059         if (!t.IsAbstract)
060         {
061             Console.WriteLine("此类不是抽象类");
062         }
063         if (!t.IsEnum)
064         {
065             Console.WriteLine("此类不是枚举类型");
066         }
067         if (t.IsVisible)
068         {
069             Console.WriteLine("其它程序集可以访问此类");
070         }
071         else
072         {
073             Console.WriteLine("其它程序集不可以访问此类");
074         }
075         if (!t.IsGenericType)
076         {
077             Console.WriteLine("此类不是泛型类型");
078         }
079     }
080 }
081 interface Hobby
082 {
083     void Sing();
084     void Dance();
085 }
086 class Person
087 {
088     //字段设置成公有只是为了让反射能够取得字段信息
089     public string name;
090     public string id;
091     public int age;
092     public string Name
093     {
094         get
095         {
096             return name;
```



```
097         }
098         set
099         {
100             name = value;
101         }
102     }
103     public string ID
104     {
105         get
106         {
107             return id;
108         }
109         set
110         {
111             id = value;
112         }
113     }
114     public int Age
115     {
116         get
117         {
118             return age;
119         }
120         set
121         {
122             age = value;
123         }
124     }
125     public Person(string name,int age,string id)
126     {
127         this.name = name;
128         this.age = age;
129         this.id = id;
130     }
131     public virtual string Print()
132     {
133         string s = string.Format("姓名: {0}, 年龄: {1}, 身份证号: {2}", Name, Age, ID);
134         return s;
135     }
136 }
137 class Student : Person, Hobby
138 {
139     private string Seat
140     {
```

```
141         get;
142         set;
143     }
144     public Student(string name, int age, string id, string seat)
145         : base(name, age, id)
146     {
147         this.Seat = seat;
148     }
149     public override string Print()
150     {
151         string s = base.Print() + "座位号: " + this.Seat;
152         return s;
153     }
154     public void Sing()
155     {
156         Console.WriteLine("我会唱歌");
157     }
158     public void Dance()
159     {
160         Console.WriteLine("我会跳舞");
161     }
162 }
163 class Program
164 {
165     static void Main(string[] args)
166     {
167         Test myTest = new Test();
168         myTest.ShowFields(typeof(Student));
169         myTest.ShowProps(typeof(Student));
170         myTest.ShowMethods(typeof(Student));
171         myTest.ShowInterfaces(typeof(Student));
172         myTest.ShowClassInfo(typeof(Student));
173         Console.ReadKey();
174     }
175 }
176 }
```

在代码的第 4 行定义了一个类 `Test`，在 `Test` 中定义了 5 个方法通过反射获取类的信息。第 7 行的 `ShowMethods` 方法带有一个 `Type` 类型的变量。这个方法用来获取类中定义的方法信息。反射类的方法使用了 `Type` 类的 `GetMethods` 方法，它返回一个 `MethodInfo` 类型的数组。`MethodInfo` 类位于 `System.Reflection` 命名空间。为了简化代码的输入，第 2 行引用了这个命名空间。第 18 行定义了一个 `ShowFields` 方法，它调用了 `GetFields` 方法。这个方法返回一个 `FieldInfo` 类型的数组。`FieldInfo` 类型也位于 `System.Reflection` 命名空间。第 29 行定义了一个 `ShowProps` 方法，它使用了 `GetProperties` 方法，这个方法返回一个 `PropertyInfo` 类型的数组。第 40 行定义了一个 `ShowInterfaces` 方法，这个方法使用了 `GetInterfaces` 方法，`GetInterfaces` 方法返回一个 `Type` 类型的数组。第 51 行定义了一个 `ShowClassInfo`

方法，这个方法调用了 Type 类型的若干属性对类的各种信息和情况进行了判断。在获得类的成员数组后，各个方法使用了 foreach 循环迭代输出了各个成员的名称。

第 81 行定义了一个接口 Hobby，第 86 行定义了一个类 Person。第 137 行定义了一个类 Student 派生自类 Person，且实现了接口 Hobby。

在类 Program 中，定义了一个类 Test 的实例。然后用这个实例调用类的各个实现反射功能的方法。方法的参数就是类 Student 的类型。需要注意的是，GetFields 方法只能返回 public 访问权限的字段。为了演示这个方法，类 Person 中的字段设置成了 public 访问权限。这段代码的执行结果如下：

通过反射得到的字段成员信息

1. name
2. id
3. age

通过反射得到的属性成员信息

1. Name
2. ID
3. Age

通过反射得到的方法信息

1. Print
2. Sing
3. Dance
4. get_Name
5. set_Name
6. get_ID
7. set_ID
8. get_Age
9. set_Age
10. ToString
11. Equals
12. GetHashCode
13. GetType

通过反射得到的接口信息

1. Hobby

通过反射得到的本类信息

本类派生自：Person

此类是类类型

此类不是抽象类

此类不是枚举类型

其它程序集不可以访问此类

此类不是泛型类型

20.3 特性的实例化过程和名称搜寻

特性在使用反射进行引用时需要实例化。编译器会在当前程序集和引用的类库中寻找这个特性的名称。在创建自定义的特性的时候，对于命名有一个一般性的约定，就是用 Attribute 字符串结尾。当然，特性的名称不一定非得定义为这样的名字。但是，这是一种约定俗成的习惯。实例代码如下：

```
001    using System;
002    namespace Attribute1
```

```
003  {
004      class AppInfoAttribute : Attribute //自定义特性类需从 Attribute 类派生
005      {
006          /// <summary>
007          /// 表示作者姓名
008          /// </summary>
009          public string Name
010          {
011              get;
012              set;
013          }
014          /// <summary>
015          /// 程序集版本
016          /// </summary>
017          public string Version
018          {
019              get;
020              set;
021          }
022          /// <summary>
023          /// 帮助信息
024          /// </summary>
025          public string HelpInfo
026          {
027              get;
028              set;
029          }
030          /// <summary>
031          /// 构造方法
032          /// </summary>
033          /// <param name="name"></param>
034          public AppInfoAttribute(string name)
035          {
036              Name = name;
037          }
038      }
039      //为 Person 类指定特性
040      [AppInfo("Tom And Jarry", Version = "1.0", HelpInfo = "http://www.aaa.com/help.html")]
041      class person
042      {
043          public string Name
044          {
045              get;
046              set;
```

```
047     }
048     public string ID
049     {
050         get;
051         set;
052     }
053     public int Age
054     {
055         get;
056         set;
057     }
058     public void Print()
059     {
060         Console.WriteLine("姓名: {0}, 年龄: {1}, 身份证号: {2}", Name, Age, ID);
061     }
062 }
063 class Student
064 {
065     public void Sing()
066     {
067         Console.WriteLine("我会唱歌");
068     }
069     public void Dance()
070     {
071         Console.WriteLine("我会跳舞");
072     }
073 }
074 class Program
075 {
076     static void OutPutInfo(Type t)
077     {
078         Console.WriteLine("现在开始通过反射获取 {0} 类的特性信息", t.ToString());
079         Attribute[] attArray = Attribute.GetCustomAttributes(t);
080         foreach (Attribute att in attArray)
081         {
082             if (att is AppInfoAttribute)
083             {
084                 AppInfoAttribute ai = (AppInfoAttribute)att;
085                 Console.WriteLine("作者: {0}, 版本: {1}, 帮助信息: {2}", ai.Name,
086                                     ai.Version, ai.HelpInfo);
087             }
088         }
089     static void Main(string[] args)
```

```

090      {
091          OutPutInfo(typeof(person));
092          OutPutInfo(typeof(Student));
093          Console.ReadKey();
094      }
095  }
096  }

```

在这段代码中，第 4 行定义了特性类 AppInfoAttribute。这个名称以 Attribute 字符串结尾。当在第 40 行使用它时，为类指定的特性名称可以不包含 Attribute 字符串。编译器在搜索到这个特性的时候，会自动尝试为它加上这个字符串。搜索的位置就是本程序集和引用的所有类库。当通过反射来获取信息的时候，就会使用这个搜索到的特性的构造方法创建特性类的实例。在本段代码中，为类指定特性时，已经指定了构造方法的参数。

```
AppInfo("Tom And Jerry",Version = "1.0",HelpInfo = "http://www.aaa.com/help.html")
```

这条语句等价于：

```

AppInfoAttribute *** = new AppInfoAttribute("Tom And Jerry");
***.Version = "1.0";
***.HelpInfo = "http://www.aaa.com/help.html";

```

这个 AppInfoAttribute 实例是一个匿名的实例。它在第 79 行的语句：

```
Attribute[] attArray = Attribute.GetCustomAttributes(t);
```

为 Attribute 类型的数组赋值时创建。在本例中，构造方法必须参数叫做定位参数，它是不能省略的。另外两个叫做命名参数，它们的顺序可以随意指定，参数的个数也可以指定。如果不指定它们，则它们采用默认值。如果第 40 行代码改为如下的语句：

```
[AppInfo("Tom And Jerry")]
```

则代码的执行结果如下：

现在开始通过反射获取 Attribute1.person 类的特性信息

作者：Tom And Jerry，版本：，帮助信息：

现在开始通过反射获取 Attribute1.Student 类的特性信息

20.4 为自定义特性指定特性

当自定义特性的时候，还可以为自定义特性指定 AttributeUsage 特性。这个特性是被 C# 的编译器支持的，它的作用是定义使用自定义特性类的方式。实例代码如下：

```

001  using System;
002  namespace Attribute1
003  {
004      [AttributeUsage(AttributeTargets.Class|AttributeTargets.Interface)]
005      class AppInfoAttribute : Attribute //自定义特性类需从 Attribute 类派生
006      {
007          /// <summary>
008          /// 表示作者姓名
009          /// </summary>
010          public string Name
011          {
012              get;
013              set;
014          }

```

```
015      /// <summary>
016      /// 程序集版本
017      /// </summary>
018      public string Version
019      {
020          get;
021          set;
022      }
023      /// <summary>
024      /// 帮助信息
025      /// </summary>
026      public string HelpInfo
027      {
028          get;
029          set;
030      }
031      /// <summary>
032      /// 构造方法
033      /// </summary>
034      /// <param name="name"></param>
035      public AppInfoAttribute(string name)
036      {
037          Name = name;
038      }
039  }
040  //[AppInfo("Tom And Jerry", Version = "1.0", HelpInfo = "http://www.aaa.com/help.html")]
041  //struct PersonInfo
042  //{
043  //    private string ID;
044  //}
045  [AppInfo("Tom And Jerry", Version = "1.0", HelpInfo = "http://www.aaa.com/help.html")]
046  interface Hobby
047  {
048      void Sing();
049      void Dance();
050  }
051  //为Person 类指定特性
052  [AppInfo("Tom And Jerry", Version = "1.0", HelpInfo = "http://www.aaa.com/help.html")]
053  class Person
054  {
055      public string Name
056      {
057          get;
058          set;
```

```
059     }
060     public string ID
061     {
062         get;
063         set;
064     }
065     public int Age
066     {
067         get;
068         set;
069     }
070     public void Print()
071     {
072         Console.WriteLine("姓名: {0}, 年龄: {1}, 身份证号: {2}", Name, Age, ID);
073     }
074 }
075 class Student:Person, Hobby
076 {
077     public void Sing()
078     {
079         Console.WriteLine("我会唱歌");
080     }
081     public void Dance()
082     {
083         Console.WriteLine("我会跳舞");
084     }
085 }
086 class Program
087 {
088     static void OutPutInfo(Type t)
089     {
090         Console.WriteLine("现在开始通过反射获取 {0} 类的特性信息", t.ToString());
091         Attribute[] attArray = Attribute.GetCustomAttributes(t);
092         foreach (Attribute att in attArray)
093         {
094             if (att is AppInfoAttribute)
095             {
096                 AppInfoAttribute ai = (AppInfoAttribute)att;
097                 Console.WriteLine("作者: {0}, 版本: {1}, 帮助信息: {2}", ai.Name,
098                                     ai.Version, ai.HelpInfo);
099             }
100         }
101     static void Main(string[] args)
```



```

102      {
103          OutPutInfo(typeof(Person));
104          OutPutInfo(typeof(Student));
105          OutPutInfo(typeof(Hobby));
106          //OutPutInfo(typeof(PersonInfo));
107          Console.ReadKey();
108      }
109  }
110 }

```

AttributeUsage 特性有一个定位参数，这个定位参数能够指定特性可以应用在何种声明上。定位参数是 AttributeTargets 类型，它是一个枚举。它的取值及作用如下所示：

- Assembly 可以对程序集应用特性。
- Module 可以对模块应用特性。说明:Module 指的是可移植的可执行文件 (.dll 或 .exe)，而非 Visual Basic 标准模块。
- Class 可以对类应用特性。
- Struct 可以对结构应用特性，即值类型。
- Enum 可以对枚举应用特性。
- Constructor 可以对构造函数应用特性。
- Method 可以对方法应用特性。
- Property 可以对属性应用特性。
- Field 可以对字段应用特性。
- Event 可以对事件应用特性。
- Interface 可以对接口应用特性。
- Parameter 可以对参数应用特性。
- Delegate 可以对委托应用特性。
- ReturnValue 可以对返回值应用特性。
- GenericParameter 可以对泛型参数应用特性。
- All 可以对任何应用程序元素应用特性。

如果要同时满足几种应用，可以使用“|”运算符将枚举值联合使用。例如第 4 行代码将 AttributeTargets.Class 枚举成员与 AttributeTargets.Interface 枚举成员用“|”运算符联用，表示这个自定义的特性只能用在类和接口上。第 46 行和第 53 行的接口与类上都定义了 AppInfoAttribute 这个自定义的特性。第 105 行也对定义的接口 Hobby 进行了特性的反射读取。代码的执行结果如下：

现在开始通过反射获取 Attribute1.Person 类的特性信息

作者: Tom And Jarry, 版本: 1.0, 帮助信息: <http://www.aaa.com/help.html>

现在开始通过反射获取 Attribute1.Student 类的特性信息

作者: Tom And Jarry, 版本: 1.0, 帮助信息: <http://www.aaa.com/help.html>

现在开始通过反射获取 Attribute1.Hobby 类的特性信息

作者: Tom And Jarry, 版本: 1.0, 帮助信息: <http://www.aaa.com/help.html>

在代码的第 41 行定义了一个结构，并且为这个结构也应用了特性。当把本段代码的注释去掉的时候，在编译的时候，编译器会报错。错误信息如下：

```

错误 1    特性“AppInfo”对此声明类型无效。它只对“class, interface”声明有效。 H:\书稿 new\
书稿源码\第二十章\Attribute1\Program.cs  40  6    Attribute1

```

因为自定义的特性也应用了特性，它不允许对结构应用自定义的 AppInfoAttribute 特性。所以导致编译器报告这个错误。

20.4.1 AttributeUsage 特性的 AllowMultiple 命名参数

AttributeUsage 特性还有一个命名参数可用，它的名字是 AllowMultiple。它的作用是针对程序实体，是否可以多次指定同一个自定义的特性。它的取值为 true 或 false。如果为 true，则说明特性是多次性特性，对于程序实体可以多次指定同一个特性；如果为 false 或未指定这个命名参数，则说明特性是一次性特性，在给定的程序实体上只能指定一次。下面是代码实例：

```
001    using System;
002    namespace Attribute1
003    {
004        [AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface, AllowMultiple
            = true)]
005        class AppInfoAttribute : Attribute
006        {
007            public string Name
008            {
009                get;
010                set;
011            }
012            public string Version
013            {
014                get;
015                set;
016            }
017            public string HelpInfo
018            {
019                get;
020                set;
021            }
022            public AppInfoAttribute(string name)
023            {
024                Name = name;
025            }
026        }
027        [AppInfo("Tom And Jerry", Version = "1.0", HelpInfo = "http://a/help.html")]
028        [AppInfo("Tom And Jerry", Version = "1.0", HelpInfo = "http://a/help.html")]
029        interface Hobby
030        {
031            void Sing();
032            void Dance();
033        }
034        [AppInfo("Tom And Jerry", Version = "1.0", HelpInfo = "http://a/help.html")]
035        [AppInfo("Tom And Jerry", Version = "1.0", HelpInfo = "http://a/help.html")]
036        class Person
037        {
038            public string Name
```

```
039         {
040             get;
041             set;
042         }
043     public string ID
044     {
045         get;
046         set;
047     }
048     public int Age
049     {
050         get;
051         set;
052     }
053     public void Print()
054     {
055         Console.WriteLine("姓名: {0}, 年龄: {1}, 身份证号: {2}", Name, Age, ID);
056     }
057 }
058 class Student : Person, Hobby
059 {
060     public void Sing()
061     {
062         Console.WriteLine("我会唱歌");
063     }
064     public void Dance()
065     {
066         Console.WriteLine("我会跳舞");
067     }
068 }
069 class Program
070 {
071     static void OutPutInfo(Type t)
072     {
073         Console.WriteLine("现在开始通过反射获取 {0} 类的特性信息", t.ToString());
074         Attribute[] attArray = Attribute.GetCustomAttributes(t);
075         foreach (Attribute att in attArray)
076         {
077             if (att is AppInfoAttribute)
078             {
079                 AppInfoAttribute ai = (AppInfoAttribute)att;
080                 Console.WriteLine("作者: {0}, 版本: {1}, 帮助信息: {2}", ai.Name,
081                                     ai.Version, ai.HelpInfo);
081             }
082         }
083     }
084 }
```

```

082         }
083     }
084     static void Main(string[] args)
085     {
086         OutPutInfo(typeof(Person));
087         OutPutInfo(typeof(Student));
088         OutPutInfo(typeof(Hobby));
089         Console.ReadKey();
090     }
091 }
092 }

```

在代码的第 4 行, 为自定义特性类指定的 `AttributeUsage` 特性的命名参数 `AllowMultiple` 被赋值为 `true`。这说明 `AppInfoAttribute` 特性是多次性特性。在为程序实体指定 `AppInfoAttribute` 特性的时候, 可以多次指定。接下来, 为接口 `Hobby` 和类 `Person` 指定了两次 `AppInfoAttribute` 特性。这时的代码执行结果如下:

现在开始通过反射获取 `Attribute1.Person` 类的特性信息

作者: Tom And Jerry, 版本: 1.0, 帮助信息: <http://a/help.html>

作者: Tom And Jerry, 版本: 1.0, 帮助信息: <http://a/help.html>

现在开始通过反射获取 `Attribute1.Student` 类的特性信息

作者: Tom And Jerry, 版本: 1.0, 帮助信息: <http://a/help.html>

作者: Tom And Jerry, 版本: 1.0, 帮助信息: <http://a/help.html>

现在开始通过反射获取 `Attribute1.Hobby` 类的特性信息

作者: Tom And Jerry, 版本: 1.0, 帮助信息: <http://a/help.html>

作者: Tom And Jerry, 版本: 1.0, 帮助信息: <http://a/help.html>

可以看到, 使用反射获取特性的时候, 对同一个程序实体都获取到了两个特性实例。对于一个程序实体如果多次指定特性, 还可以使用下面的语法:

```
[AppInfo("Tom And Jerry", Version = "1.0", HelpInfo = "http://a/help.html"), AppInfo("Tom And
Jerry", Version = "1.0", HelpInfo = "http://a/help.html")]
```

它的效果和每个特性都使用一对中括号的方式是一样的。

20.4.2 AttributeUsage 特性的 Inherited 命名参数

从前面的代码中, 第 87 行代码使用反射获取了类 `Student` 的特性。但是在代码中, 对于 `Student` 类并没有应用特性。但是 `Student` 类从 `Person` 类派生, 并且实现了接口 `Hobby`。对 `Person` 类和接口 `Hobby` 都指定了特性 `AppInfoAttribute`。默认情况下, 特性是能够被继承中的程序实体所继承的。`AttributeUsage` 特性还有一个 `Inherited` 命名参数, 可以使用它来指定特性是否能够被继承。下面是代码实例:

```

001     using System;
002     namespace Attribute1
003     {
004         [AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface, AllowMultiple
            = true, Inherited = true)]
005         class AppInfoAttribute : Attribute
006         {
007             public string Name
008             {
009                 get;

```

```
010         set;
011     }
012     public string Version
013     {
014         get;
015         set;
016     }
017     public string HelpInfo
018     {
019         get;
020         set;
021     }
022     public AppInfoAttribute(string name)
023     {
024         Name = name;
025     }
026 }
027 [AppInfo("Tom And Jerry", Version = "1.0", HelpInfo = "http://a/help.html")]
028 [AppInfo("Tom And Jerry", Version = "1.0", HelpInfo = "http://a/help.html")]
029 interface Hobby
030 {
031     void Sing();
032     void Dance();
033 }
034 class Person
035 {
036     public string Name
037     {
038         get;
039         set;
040     }
041     public string ID
042     {
043         get;
044         set;
045     }
046     public int Age
047     {
048         get;
049         set;
050     }
051     public void Print()
052     {
053         Console.WriteLine("姓名: {0}, 年龄: {1}, 身份证号: {2}", Name, Age, ID);
```

```

054     }
055 }
056 class Student : Person, Hobby
057 {
058     public void Sing()
059     {
060         Console.WriteLine("我会唱歌");
061     }
062     public void Dance()
063     {
064         Console.WriteLine("我会跳舞");
065     }
066 }
067 class Program
068 {
069     static void OutPutInfo(Type t)
070     {
071         Console.WriteLine("现在开始通过反射获取 {0} 类的特性信息", t.ToString());
072         Attribute[] attArray = Attribute.GetCustomAttributes(t);
073         foreach (Attribute att in attArray)
074         {
075             if (att is AppInfoAttribute)
076             {
077                 AppInfoAttribute ai = (AppInfoAttribute)att;
078                 Console.WriteLine("作者: {0}, 版本: {1}, 帮助信息: {2}", ai.Name,
                                ai.Version, ai.HelpInfo);
079             }
080         }
081     }
082     static void Main(string[] args)
083     {
084         OutPutInfo(typeof(Student));
085         Console.ReadKey();
086     }
087 }
088 }

```

在代码的第4行，为自定义特性指定了 `Inherited` 命名参数。将它赋值为 `true` 表示特性将会被从指定了此特性的基类派生到子类。接下来对接口指定了两次特性。下面看代码的执行结果：

现在开始通过反射获取 `Attribute1.Student` 类的特性信息

可以看到，特性并没有被继承下来。这是因为特性的派生只是针对类而言。对接口指定特性后，特性不会被实现类继承。`Inherited` 命名参数是对类起作用的。下面将特性加到类 `Person` 上。这时的代码执行结果如下：

现在开始通过反射获取 `Attribute1.Student` 类的特性信息

作者: Tom And Jarry, 版本: 1.0, 帮助信息: <http://a/help.html>

作者: Tom And Jarry, 版本: 1.0, 帮助信息: <http://a/help.html>

对于一个什么特性也不指定的自定义特性类来说, 默认情况下, 虽然在特性类上没有指定 `AttributeUsage` 特性, 但是它是隐式起作用的。对于什么特性也不指定的自定义特性来说, 它相当于指定了下面的特性:

```
[AttributeUsage(AttributeTargets.All, AllowMultiple = false, Inherited = true)]
```

也就是说, 自定义特性是可以指定给所有类型的程序实体, 并且不允许多次指定的, 同时它可以被派生类继承。

20.5 定位参数和命名参数

自定义特性可以具有定位参数和命名参数。自定义特性的定位参数由自定义特性的公共实例构造方法来确定; 自定义的特性类的命名参数由类的公共读写属性和公共读写字段来确定。下面是代码实例:

```
001 using System;
002 namespace Attribute1
003 {
004     [AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface)]
005     class AppInfoAttribute : Attribute
006     {
007         public string platform;
008         private string style;
009         public string Style
010         {
011             get
012             {
013                 return style;
014             }
015         }
016         public string Name
017         {
018             get;
019             set;
020         }
021         public string Version
022         {
023             get;
024             set;
025         }
026         public string HelpInfo
027         {
028             get;
029             set;
030         }
031         public AppInfoAttribute(string name, string version)
032         {
033             Name = name;
034             Version = version;
```

```
035         platform = "Windows 7";
036         HelpInfo = "a.html";
037         style = "控制台";
038     }
039 }
040 [AppInfo("Tom And Jarry", "1.0", HelpInfo = "http://a/help.html", platform = "Windows
8")]
041 interface Hobby
042 {
043     void Sing();
044     void Dance();
045 }
046 [AppInfo("秋叶无声", "2.0", HelpInfo = "http://b/help.html")]
047 class Person
048 {
049     public string Name
050     {
051         get;
052         set;
053     }
054     public string ID
055     {
056         get;
057         set;
058     }
059     public int Age
060     {
061         get;
062         set;
063     }
064     public void Print()
065     {
066         Console.WriteLine("姓名: {0}, 年龄: {1}, 身份证号: {2}", Name, Age, ID);
067     }
068 }
069 class Student : Person, Hobby
070 {
071     public void Sing()
072     {
073         Console.WriteLine("我会唱歌");
074     }
075     public void Dance()
076     {
077         Console.WriteLine("我会跳舞");
```



```

078     }
079     }
080     class Program
081     {
082         static void OutPutInfo(Type t)
083         {
084             Console.WriteLine("现在开始通过反射获取 {0} 类的特性信息", t.ToString());
085             Attribute[] attArray = Attribute.GetCustomAttributes(t);
086             foreach (Attribute att in attArray)
087             {
088                 if (att is AppInfoAttribute)
089                 {
090                     AppInfoAttribute ai = (AppInfoAttribute)att;
091                     Console.WriteLine("作者: {0}, 版本: {1}, 帮助信息: {2}", ai.Name,
092                                     ai.Version, ai.HelpInfo);
093                     Console.WriteLine("平台: {0}, 程序风格: {1}", ai.platform, ai.Style);
094                 }
095             }
096             static void Main(string[] args)
097             {
098                 OutPutInfo(typeof(Student));
099                 OutPutInfo(typeof(Hobby));
100                 Console.ReadKey();
101             }
102         }
103     }

```

本段代码中的自定义特性类增加了一个公共的字段 platform; 一个 private 的字段 style。属性 Style 对它有读取操作, 但没有定义 set 访问器。在第 31 行的构造方法中带有两个构造方法参数。因此, 在第 40 行和第 46 行代码中, 对接口和类指定特性的时候, 就需要两个定位参数。但是命名参数个数可以随意。代码的执行结果如下:

现在开始通过反射获取 Attribute1.Student 类的特性信息

作者: 秋叶无声, 版本: 2.0, 帮助信息: <http://b/help.html>

平台: Windows 7, 程序风格: 控制台

现在开始通过反射获取 Attribute1.Hobby 类的特性信息

作者: Tom And Jarry, 版本: 1.0, 帮助信息: <http://a/help.html>

平台: Windows 8, 程序风格: 控制台

但是, 如果将定位参数改成一个的话, 编译器会提示错误。错误信息如下:

错误 1 “Attribute1.AppInfoAttribute” 不包含采用 “1” 个参数的构造函数 H:\书稿 new\书稿源码\第二十章\Attribute2\Program.cs 40 6 Attribute2

错误 2 “Attribute1.AppInfoAttribute” 不包含采用 “1” 个参数的构造函数 H:\书稿 new\书稿源码\第二十章\Attribute2\Program.cs 46 6 Attribute2

从错误提示可以知道, 特性的构造方法中有几个参数, 在指定特性的时候就必须提供几个定位参数。

在自定义的特性中, 有一个 Style 属性, 它是一个只读属性。虽然它的访问权限也是 public 的, 但是

因为它只有 get 访问器的缘故，它不能做命名参数。如果将它指定为命名参数，编译器同样会报错。错误信息如下：

错误 1 “Style”不是有效的命名特性参数。命名特性参数必须是非只读、非静态或非常数的字段，或者是公共的和非静态的读写属性。 H:\书稿 new\书稿源码\第二十章\Attribute2\Program.cs 46 28
Attribute2

通过错误信息可知，命名参数不能是静态成员，因为它是需要创建特性类的实例来访问的。同时，只读成员和常量也不能做命名参数。

但是，如果特性类中提供了多个重载的构造方法，则定位参数可以指定符合重载的构造方法的个数。

下面是代码实例：

```
001 using System;
002 namespace Attribute1
003 {
004     [AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface)]
005     class AppInfoAttribute : Attribute
006     {
007         public string Style
008         {
009             get;
010             set;
011         }
012         public string Platform
013         {
014             get;
015             set;
016         }
017         public string Name
018         {
019             get;
020             set;
021         }
022         public string Version
023         {
024             get;
025             set;
026         }
027         public string HelpInfo
028         {
029             get;
030             set;
031         }
032         public AppInfoAttribute(string name, string version)
033         {
034             Name = name;
035             Version = version;
```

```
036         Platform = "Windows 7";
037         HelpInfo = "a.html";
038         Style = "控制台";
039     }
040     public AppInfoAttribute(string name, string version, string platform)
041     {
042         Name = name;
043         Version = version;
044         Platform = platform;
045         HelpInfo = "b.html";
046         Style = "控制台";
047     }
048 }
049 [AppInfo("Tom And Jerry", "1.0", HelpInfo = "http://a/help.html", Platform = "Windows
8")]
050 interface Hobby
051 {
052     void Sing();
053     void Dance();
054 }
055 [AppInfo("秋叶无声", "2.0", "Windows 7")]
056 class Person
057 {
058     public string Name
059     {
060         get;
061         set;
062     }
063     public string ID
064     {
065         get;
066         set;
067     }
068     public int Age
069     {
070         get;
071         set;
072     }
073     public void Print()
074     {
075         Console.WriteLine("姓名: {0}, 年龄: {1}, 身份证号: {2}", Name, Age, ID);
076     }
077 }
078 class Student : Person, Hobby
```

```

079     {
080         public void Sing()
081         {
082             Console.WriteLine("我会唱歌");
083         }
084         public void Dance()
085         {
086             Console.WriteLine("我会跳舞");
087         }
088     }
089     class Program
090     {
091         static void OutPutInfo(Type t)
092         {
093             Console.WriteLine("现在开始通过反射获取 {0} 类的特性信息", t.ToString());
094             Attribute[] attArray = Attribute.GetCustomAttributes(t);
095             foreach (Attribute att in attArray)
096             {
097                 if (att is AppInfoAttribute)
098                 {
099                     AppInfoAttribute ai = (AppInfoAttribute)att;
100                     Console.WriteLine("作者: {0}, 版本: {1}, 帮助信息: {2}", ai.Name,
101                                     ai.Version, ai.HelpInfo);
102                     Console.WriteLine("平台: {0}, 程序风格: {1}", ai.Platform, ai.Style);
103                 }
104             }
105             static void Main(string[] args)
106             {
107                 OutPutInfo(typeof(Student));
108                 OutPutInfo(typeof(Hobby));
109                 Console.ReadKey();
110             }
111         }
112     }

```

这段代码在第 40 行又定义了一个构造方法，它带有三个构造方法参数。因此第 55 行为类 Person 指定特性的时候，可以采用 3 个定位参数。在为接口指定特性的时候，可以采用 2 个定位参数。只要使用的定位参数的类型和个数可以匹配重载的特性类构造方法即可。这段代码的执行结果如下：

现在开始通过反射获取 Attribute1.Student 类的特性信息

作者：秋叶无声，版本：2.0，帮助信息：b.html

平台：Windows 7，程序风格：控制台

现在开始通过反射获取 Attribute1.Hobby 类的特性信息

作者：Tom And Jarry，版本：1.0，帮助信息：<http://a/help.html>

平台：Windows 8，程序风格：控制台

20.6 定位参数和命名参数的类型限定

对于特性类的定位参数和命名参数来说，它们的参数类型是有限制的。它们的参数类型必须是以下几种：

- 基本数据类型 bool、byte、char、double、float、int、long、sbyte、short、string、uint、ulong、ushort。
- object 类型。
- System.Type 类型。
- 枚举类型，前提是该枚举类型具有 public 可访问性，而且所有嵌套着它的类型（如果有）也必须具有 public 可访问性。
- 以上类型的一维数组。

下面是代码实例：

```
001 using System;
002 namespace Attribute1
003 {
004     public class ColorClass
005     {
006         public enum Color { Red, Green, Blue };
007     }
008     struct Data
009     {
010         public int bit;
011         public string type;
012         public Data(int bit, string type)
013         {
014             this.bit = bit;
015             this.type = type;
016         }
017     }
018     class AppInfoAttribute : Attribute
019     {
020         public ColorClass.Color style;
021         public string Platform
022         {
023             get;
024             set;
025         }
026         public string Name
027         {
028             get;
029             set;
030         }
031         public string Version
032         {
033             get;
```

```
034         set;
035     }
036     public string[] dataType;
037     public Data data;
038     public AppInfoAttribute(string name, string version, string platform)
039     {
040         Name = name;
041         Version = version;
042         Platform = platform;
043         dataType = new string[3] { "int", "enum", "string" };
044         style = ColorClass.Color.Blue;
045         data = new Data(32, "Int32");
046     }
047     public AppInfoAttribute(string name, string version)
048     {
049         Name = name;
050         Version = version;
051         Platform = "Windows 8";
052         dataType = new string[3] { "int", "enum", "string" };
053         style = ColorClass.Color.Blue;
054         data = new Data(64, "Int64");
055     }
056 }
057 [AppInfo("Tom And Jarry", "2.0", style = ColorClass.Color.Red)]
058 interface Hobby
059 {
060     void Sing();
061     void Dance();
062 }
063 [AppInfo("秋叶无声", "2.0", "Windows 7")]
064 class Person
065 {
066     public string Name
067     {
068         get;
069         set;
070     }
071     public string ID
072     {
073         get;
074         set;
075     }
076     public int Age
077     {
```

```
078         get;
079         set;
080     }
081     public void Print()
082     {
083         Console.WriteLine("姓名: {0}, 年龄: {1}, 身份证号: {2}", Name, Age, ID);
084     }
085 }
086 class Student : Person, Hobby
087 {
088     public void Sing()
089     {
090         Console.WriteLine("我会唱歌");
091     }
092     public void Dance()
093     {
094         Console.WriteLine("我会跳舞");
095     }
096 }
097 class Program
098 {
099     static void OutPutInfo(Type t)
100     {
101         Console.WriteLine("现在开始通过反射获取 {0} 类的特性信息", t.ToString());
102         Attribute[] attArray = Attribute.GetCustomAttributes(t);
103         foreach (Attribute att in attArray)
104         {
105             if (att is AppInfoAttribute)
106             {
107                 AppInfoAttribute ai = (AppInfoAttribute)att;
108                 Console.WriteLine("作者: {0}, 版本: {1}, 平台: {2}", ai.Name, ai.Version,
109                                     ai.Platform);
110                 Console.WriteLine("结构信息 (位数): {0}, 结构信息 (类型): {1}",
111                                     ai.data.bit, ai.data.type);
112                 Console.WriteLine("程序字体颜色: {0}", ai.style.ToString());
113             }
114         }
115     }
116     static void Main(string[] args)
117     {
118         OutPutInfo(typeof(Student));
119         OutPutInfo(typeof(Hobby));
120         Console.ReadKey();
121     }
122 }
```

```
120     }
121 }
```

代码的第 4 行定义了一个类，类中包含一个 public 访问权限的枚举类型。第 8 行定义了一个结构。定义的这两个类型都是为了在特性类中使用。但是，前面的列表中并没有包含结构类型。是否能够在特性类中定义结构的成员或类成员呢？答案是肯定的。这段代码中并没有使用定位参数和命名参数的形式为 struct 类型或 class 类型赋值，而是把 struct 类型的成员的初始化定义在了特性类的构造方法中。第 45 行和第 54 行代码使用 new 关键字在构造方法中为特性类的结构成员初始化。代码的执行结果如下：

现在开始通过反射获取 Attribute1.Student 类的特性信息

作者：秋叶无声，版本：2.0，平台：Windows 7

结构信息（位数）：32，结构信息（类型）：Int32

程序字体颜色：Blue

现在开始通过反射获取 Attribute1.Hobby 类的特性信息

作者：Tom And Jarry，版本：2.0，平台：Windows 8

结构信息（位数）：64，结构信息（类型）：Int64

程序字体颜色：Red

接下来修改代码，在为程序实体应用特性的时候，以命名参数的方式为结构成员赋值。代码如下：

```
[AppInfo("Tom And Jarry", "2.0", style = ColorClass.Color.Red, data = new Data(64, "Int64"))]
```

这时，编译器会报告如下的错误：

```
错误 1      “data”不是有效的命名特性参数，因为它不是有效的特性参数类型 H:\书稿new\书稿源码\第二十章\Attribute2\Program.cs    57    66    Attribute2
```

出现这个错误的原因是，结构和类是不能作定位参数和命名参数的。下面将特性类中的 data 成员改成 object 类型。因为 object 类型是可以作定位参数和命名参数的。相应的，为类型指定特性类的时候，命名参数的指定方式也需要修改，如下：

```
[AppInfo("Tom And Jarry", "2.0", style = ColorClass.Color.Red, data = (object)new Data(64, "Int64"))]
```

同时，第 109 行读取特性类的成员的方式也要进行修改，代码如下：

```
Console.WriteLine("结构信息（位数）：{0}，结构信息（类型）：{1}", ((Data)ai.data).bit, ((Data)ai.data).type);
```

在这行代码中需要通过显式转换的方法将 object 类型转换为 Data 类型，这样才能读取其成员。否则，object 类型是没有成员的。但是这样做也不行，编译的时候，编译器也会报错。错误信息如下：

```
错误 1      特性实参必须是特性形参类型的常量表达式、typeof 表达式或数组创建表达式 H:\书稿 new\书稿源码\第二十章\Attribute2\Program.cs    57    81    Attribute2
```

对于特性的定位参数和命名参数是不能使用 new 关键字语法创建类或结构的实例的。它必须是常量表达式、typeof 表达式或数组创建表达式。下面是使用数组创建表达式的代码实例：

```
001    using System;
002    namespace Attribute1
003    {
004        public class ColorClass
005        {
006            public enum Color { Red, Green, Blue };
007        }
008        struct Data
009        {
010            public int bit;
```



```
011         public string type;
012         public Data(int bit, string type)
013         {
014             this.bit = bit;
015             this.type = type;
016         }
017     }
018     class AppInfoAttribute : Attribute
019     {
020         public ColorClass.Color style;
021         public string Platform
022         {
023             get;
024             set;
025         }
026         public string Name
027         {
028             get;
029             set;
030         }
031         public string Version
032         {
033             get;
034             set;
035         }
036         public string[] dataType;
037         public Data data;
038         public AppInfoAttribute(string name, string version, string platform)
039         {
040             Name = name;
041             Version = version;
042             Platform = platform;
043             dataType = new string[3] { "int", "enum", "string" };
044             style = ColorClass.Color.Blue;
045             data = new Data(32, "Int32");
046         }
047         public AppInfoAttribute(string name, string version)
048         {
049             Name = name;
050             Version = version;
051             Platform = "Windows 8";
052             dataType = new string[3] { "int", "enum", "string" };
053             style = ColorClass.Color.Blue;
054             data = new Data(64, "Int64");
```

```
055     }
056 }
057 [AppInfo("Tom And Jarry", "2.0", style = ColorClass.Color.Red, dataType = new
string[3] {"Float", "Double", "Bool"})]
058 interface Hobby
059 {
060     void Sing();
061     void Dance();
062 }
063 [AppInfo("秋叶无声", "2.0", "Windows 7")]
064 class Person
065 {
066     public string Name
067     {
068         get;
069         set;
070     }
071     public string ID
072     {
073         get;
074         set;
075     }
076     public int Age
077     {
078         get;
079         set;
080     }
081     public void Print()
082     {
083         Console.WriteLine("姓名: {0}, 年龄: {1}, 身份证号: {2}", Name, Age, ID);
084     }
085 }
086 class Student : Person, Hobby
087 {
088     public void Sing()
089     {
090         Console.WriteLine("我会唱歌");
091     }
092     public void Dance()
093     {
094         Console.WriteLine("我会跳舞");
095     }
096 }
097 class Program
```

```

098     {
099         static void OutPutInfo(Type t)
100     {
101         Console.WriteLine("现在开始通过反射获取 {0} 类的特性信息", t.ToString());
102         Attribute[] attArray = Attribute.GetCustomAttributes(t);
103         foreach (Attribute att in attArray)
104         {
105             if (att is AppInfoAttribute)
106             {
107                 AppInfoAttribute ai = (AppInfoAttribute)att;
108                 Console.WriteLine("作者: {0}, 版本: {1}, 平台: {2}", ai.Name, ai.Version,
109                                     ai.Platform);
110                 Console.WriteLine("结构信息 (位数): {0}, 结构信息 (类型): {1}",
111                                     ai.data.bit, ai.data.type);
112                 Console.WriteLine("数据信息: {0}, {1}, {2}",
113                                     ai.dataType[0], ai.dataType[1], ai.dataType[2]);
114                 Console.WriteLine("程序字体颜色: {0}", ai.style.ToString());
115             }
116         }
117     }
118     static void Main(string[] args)
119     {
120         OutPutInfo(typeof(Student));
121         OutPutInfo(typeof(Hobby));
122         Console.ReadKey();
123     }
124 }

```

在第 57 行代码中, 虽然使用 new 关键字创建类或结构的实例是不允许的, 但是创建数组的实例是允许的。代码的执行结果如下:

现在开始通过反射获取 Attribute1.Student 类的特性信息

作者: 秋叶无声, 版本: 2.0, 平台: Windows 7

结构信息 (位数): 32, 结构信息 (类型): Int32

数据信息: int, enum, string

程序字体颜色: Blue

现在开始通过反射获取 Attribute1.Hobby 类的特性信息

作者: Tom And Jarry, 版本: 2.0, 平台: Windows 8

结构信息 (位数): 64, 结构信息 (类型): Int64

数据信息: Float, Double, Bool

程序字体颜色: Red

下面再来讨论如果在一个类中定义一个结构或类的静态成员并先进行实例化, 然后将它转换成 object 类型的变量以命名参数赋值, 看看是否在语法上允许。实例代码如下:

```

001     using System;
002     namespace Attribute1

```

```
003  {
004      //定义一个类，在类的成员中定义好结构的实例
005      public class ValueClass
006      {
007          public enum Color { Red, Green, Blue };
008          //成员定义成静态的
009          public static Data data1 = new Data(64, "Int32");
010          public static Data data2 = new Data(32, "Int32");
011      }
012      //因为类 ValueClass 的可访问性为 public, 两个 Data 类成员也都为 public 权限
013      //所以为了可访问性的一致性，结构也要定义成 public 访问权限。
014      public struct Data
015      {
016          public int bit;
017          public string type;
018          public Data(int bit, string type)
019          {
020              this.bit = bit;
021              this.type = type;
022          }
023      }
024      class AppInfoAttribute : Attribute
025      {
026          public ValueClass.Color style;
027          public string Platform
028          {
029              get;
030              set;
031          }
032          public string Name
033          {
034              get;
035              set;
036          }
037          public string Version
038          {
039              get;
040              set;
041          }
042          public string[] dataType;
043          public object data; //此处仍定义为 object 类型
044          public AppInfoAttribute(string name, string version, string platform)
045          {
046              Name = name;
```

```
047         Version = version;
048         Platform = platform;
049         dataType = new string[3] { "int", "enum", "string" };
050         style = ValueClass.Color.Blue;
051         data = new Data(32, "Int32");
052     }
053     public AppInfoAttribute(string name, string version)
054     {
055         Name = name;
056         Version = version;
057         Platform = "Windows 8";
058         dataType = new string[3] { "int", "enum", "string" };
059         style = ValueClass.Color.Blue;
060         data = new Data(64, "Int64");
061     }
062 }
063 [AppInfo("Tom And Jerry", "2.0", style = ValueClass.Color.Red, data =
(object)ValueClass.data2)]
064 interface Hobby
065 {
066     void Sing();
067     void Dance();
068 }
069 [AppInfo("秋叶无声", "2.0", "Windows 7", data = (object)ValueClass.data1)]
070 class Person
071 {
072     public string Name
073     {
074         get;
075         set;
076     }
077     public string ID
078     {
079         get;
080         set;
081     }
082     public int Age
083     {
084         get;
085         set;
086     }
087     public void Print()
088     {
089         Console.WriteLine("姓名: {0}, 年龄: {1}, 身份证号: {2}", Name, Age, ID);
```

```

090     }
091 }
092 class Program
093 {
094     static void OutPutInfo(Type t)
095     {
096         Console.WriteLine("现在开始通过反射获取 {0} 类的特性信息", t.ToString());
097         Attribute[] attArray = Attribute.GetCustomAttributes(t);
098         foreach (Attribute att in attArray)
099         {
100             if (att is AppInfoAttribute)
101             {
102                 AppInfoAttribute ai = (AppInfoAttribute)att;
103                 Console.WriteLine("作者: {0}, 版本: {1}, 平台: {2}", ai.Name, ai.Version,
104                                     ai.Platform);
105                 Console.WriteLine("结构信息 (位数): {0}, 结构信息 (类型): {1}",
106                                     ((Data)ai.data).bit, ((Data)ai.data).type);
107                 Console.WriteLine("数据信息: {0}, {1}, {2}",
108                                     ai.dataType[0], ai.dataType[1], ai.dataType[2]);
109                 Console.WriteLine("程序字体颜色: {0}", ai.style.ToString());
110             }
111         }
112     }
113 }
114 static void Main(string[] args)
115 {
116     OutPutInfo(typeof(Person));
117     OutPutInfo(typeof(Hobby));
118     Console.ReadKey();
119 }
120 }

```

代码的第 9 行和第 10 行先定义了结构的静态成员，然后将其在命名参数的赋值过程中转换成 object 类型进行赋值。这样在命名参数赋值中就没有了 new 关键字表达式。但是这样做也不行的，编译器会检查到在第 9 行和第 10 行使用了 new 关键字表达式为结构创建实例。而结构是不允许作为定位参数和命名参数类型的。编译器的错误提示如下：

错误 1 特性实参必须是特性形参类型的常量表达式、typeof 表达式或数组创建表达式 H:\书稿 new\书稿源码\第二十章\Attribute2\Program.cs 63 82 Attribute2

错误 2 特性实参必须是特性形参类型的常量表达式、typeof 表达式或数组创建表达式 H:\书稿 new\书稿源码\第二十章\Attribute2\Program.cs 69 57 Attribute2

20.7 特性名称的两义性

在自定义特性的时候，虽然名称约定特性的名称要以 Attribute 字符串后缀作为结尾。但是当然可以使用一个名称不以 Attribute 字符串后缀作为结尾。这个约定不是硬性的规定。当自定义一个特性，并且为程序实体指定该特性的时候，编译器会自动为特性加上 Attribute 字符串后缀进行名称的查找。如果在程序集中存在一个特性和一个以这个特性的名称再加上 Attribute 字符串后缀的特性，则会出现两义性名

称冲突。编译器将不知道使用哪个特性。代码实例如下：

```
001 using System;
002 namespace Attribute1
003 {
004     class AppInfo : Attribute
005     {
006         public string Platform
007         {
008             get;
009             set;
010         }
011         public string Name
012         {
013             get;
014             set;
015         }
016         public string Version
017         {
018             get;
019             set;
020         }
021         public AppInfo(string name, string version, string platform)
022         {
023             Name = name;
024             Version = version;
025             Platform = platform;
026         }
027         public AppInfo(string name, string version)
028         {
029             Name = name;
030             Version = version;
031             Platform = "Windows 2003";
032         }
033     }
034     class AppInfoAttribute : Attribute
035     {
036         public string Platform
037         {
038             get;
039             set;
040         }
041         public string Name
042         {
043             get;
```

```
044         set;
045     }
046     public string Version
047     {
048         get;
049         set;
050     }
051     public AppInfoAttribute(string name, string version, string platform)
052     {
053         Name = name;
054         Version = version;
055         Platform = platform;
056     }
057     public AppInfoAttribute(string name, string version)
058     {
059         Name = name;
060         Version = version;
061         Platform = "Windows 8";
062     }
063 }
064 [AppInfo("Tom And Jerry", "2.0")]
065 interface Hobby
066 {
067     void Sing();
068     void Dance();
069 }
070 [AppInfo("秋叶无声", "2.0", "Windows 7")]
071 class Person
072 {
073     public string Name
074     {
075         get;
076         set;
077     }
078     public string ID
079     {
080         get;
081         set;
082     }
083     public int Age
084     {
085         get;
086         set;
087     }
```



```

088         public void Print()
089         {
090             Console.WriteLine("姓名: {0}, 年龄: {1}, 身份证号: {2}", Name, Age, ID);
091         }
092     }
093     class Program
094     {
095         static void OutPutInfo(Type t)
096         {
097             Console.WriteLine("现在开始通过反射获取 {0} 类的特性信息", t.ToString());
098             Attribute[] attArray = Attribute.GetCustomAttributes(t);
099             foreach (Attribute att in attArray)
100             {
101                 if (att is AppInfoAttribute)
102                 {
103                     AppInfoAttribute ai = (AppInfoAttribute)att;
104                     Console.WriteLine("作者: {0}, 版本: {1}, 平台: {2}", ai.Name, ai.Version,
105                                     ai.Platform);
106                 }
107             }
108             static void Main(string[] args)
109             {
110                 OutPutInfo(typeof(Person));
111                 OutPutInfo(typeof(Hobby));
112                 Console.ReadKey();
113             }
114         }
115     }

```

这段代码中,定义了两个特性。分别是 AppInfo 类和 AppInfoAttribute 类。当为接口 Hobby 和类 Person 指定特性的时候,都指定了名称 AppInfo。这时编译器会在程序集中查找这个特性类。它以名称 AppInfo 查找时,可以找到相关的特性类;它以 AppInfoAttribute 查找时,也可以找到相关的特性类。但是编译器不知道究竟该不该为 AppInfo 名称加上 Attribute 后缀。这时就会出现两义性的问题。编译器会报告如下的错误:

错误 1 “AppInfo”在“Attribute1.AppInfo”和“Attribute1.AppInfoAttribute”之间不明确;请使用“@AppInfo”或“AppInfoAttribute” H:\书稿new\书稿源码\第二十章\Attribute2\Program.cs

64 6 Attribute2

错误 2 “AppInfo”在“Attribute1.AppInfo”和“Attribute1.AppInfoAttribute”之间不明确;请使用“@AppInfo”或“AppInfoAttribute” H:\书稿new\书稿源码\第二十章\Attribute2\Program.cs

70 6 Attribute2

在这种情况下,该怎么处理呢?一种方法就是在特性类的命名时完全遵守约定的命名规则。另外一种已经在错误提示中给出了解决办法。就是使用原义字符串。代码如下:

```

[AppInfo("Tom And Jarry", "2.0")]
interface Hobby

```

类的特性指定如下：

```
[AppInfoAttribute("秋叶无声", "2.0", "Windows 7")]
```

```
class Person
```

这样一来，编译器就能够区别特性的名称并进行查找。同时，在反射的方法中也要加入判断特性类的类型是否是 AppInfo 类的代码：

```
if (att is AppInfo)
{
    AppInfo ai = (AppInfo)att;
    Console.WriteLine("作者: {0}, 版本: {1}, 平台: {2}", ai.Name, ai.Version, ai.Platform);
}
```

的执行结果如下：

现在开始通过反射获取 Attribute1.Person 类的特性信息

作者: 秋叶无声, 版本: 2.0, 平台: Windows 7

现在开始通过反射获取 Attribute1.Hobby 类的特性信息

作者: Tom And Jarry, 版本: 2.0, 平台: Windows 2003

20.8 特性目标

前面介绍过对自定义特性使用 AttributeUsage 特性可以定义自定义特性应用的目标。而定义好的特性还可以使用特性目标更具体的确定其应用目标。特性目标的使用是遵循下列的语法：

```
[target : 目标值]
```

目标值可以从如下的列表中进行选择：

- assembly 整个程序集
- module 当前程序集模块
- field 在类或结构中的字段
- event Event
- method 方法或 get 和 set 属性访问器
- param 方法参数或 set 属性访问器参数
- property 属性
- return 方法、属性索引器或 get 属性访问器的返回值
- type 结构、类、接口、枚举或委托

下面是代码实例：

首先定义一个类库，在类库中定义一个特性类，它用于程序集目标的特性指定。代码如下：

```
001 using System;
002 namespace Attribute1
003 {
004     [AttributeUsage(AttributeTargets.Assembly)]
005     public class AppVersionAttribute : Attribute
006     {
007         public string Version
008         {
009             get;
010             set;
011         }
012         public AppVersionAttribute(string version)
013         {
```

```
014         Version = version;
015     }
016 }
```

在这段代码的第 5 行定义了一个特性，名称为 AppVersionAttribute。需要注意，它的名称必须以 Attribute 结束，即遵从约定的命名规则。因为这个程序集要作类库使用。如果不加特性后缀。在引用它的程序集中是访问不到这个特性类的。也就是说，编译器对这个特性类必须加后缀进行查找，而不是以它的不加后缀的原名查找。第 4 行对这个自定义的特性指定了 AttributeUsage 特性，因为这个特性就是专用于程序集目标，所以使用了 AttributeTargets.Assembly 枚举。那么为何要先定义一个类库出来呢？这是因为使用程序集目标的特性指定，指定用的特性类不能位于被指定的程序集中。所以要在被指定的程序集外定义一个类库，在这个类库中定义一个程序集。

接下来新建一个控制台项目，项目的代码虽然较长，但这也算是对本章的重点内容的一个总结和回顾。代码如下：

```
001 using System;
002 using Attribute1;
003 using System.Reflection;
004 [assembly:AppVersion("1.0.0.1")]
005 namespace Attribute1
006 {
007     [AttributeUsage(AttributeTargets.All)]
008     class AppInfoAttribute : Attribute
009     {
010         public string Platform
011         {
012             get;
013             set;
014         }
015         public string Name
016         {
017             get;
018             set;
019         }
020         public string Version
021         {
022             get;
023             set;
024         }
025         public AppInfoAttribute(string name, string version, string platform)
026         {
027             Name = name;
028             Version = version;
029             Platform = platform;
030         }
031         public AppInfoAttribute(string name, string version)
032         {
```

```
033         Name = name;
034         Version = version;
035         Platform = "Windows 8";
036     }
037 }
038 interface Hobby
039 {
040     [return: AppInfo("Tom and Jarry", "1.0")]
041     void Sing();
042     void Dance();
043 }
044 [type:AppInfo("秋叶无声", "3.0", "Windows XP")]
045 class Person
046 {
047     [property: AppInfo("秋叶无声", "2.0", "Windows 7")]
048     public string Name
049     {
050         get;
051         set;
052     }
053     public string ID
054     {
055         get;
056         set;
057     }
058     public int Age
059     {
060         get;
061         set;
062     }
063     public void Print([param: AppInfo("秋叶无声", "2.0")]string s)
064     {
065         s = "#";
066         Console.WriteLine(s + " 姓名: {0}, 年龄: {1}, 身份证号: {2}", Name, Age, ID);
067     }
068 }
069 class Program
070 {
071     static void GetAssemblyAttribute(string name)
072     {
073         Console.WriteLine("现在开始通过反射获取 {0} 程序集的特性信息", name);
074         Assembly asb = Assembly.Load(name);
075         Attribute atb = asb.GetCustomAttribute(typeof(AppVersionAttribute));
076         if (atb is AppVersionAttribute)
```

```
077         {
078             AppVersionAttribute ai = (AppVersionAttribute)atb;
079             Console.WriteLine("程序集版本: {0}", ai.Version);
080         }
081     }
082     static void GetTypeAttribute(Type t)
083     {
084         Console.WriteLine("现在开始通过反射获取 {0} 类的特性信息", t.ToString());
085         Attribute[] attArray = Attribute.GetCustomAttributes(t);
086         foreach (Attribute att in attArray)
087         {
088             if (att is AppInfoAttribute)
089             {
090                 AppInfoAttribute ai = (AppInfoAttribute)att;
091                 Console.WriteLine("作者: {0}, 版本: {1}, 平台: {2}", ai.Name,
                                ai.Version, ai.Platform);
092             }
093         }
094     }
095     static void GetPropAttribute(Type t)
096     {
097         Console.WriteLine("现在开始通过反射获取 {0} 类的属性的特性信息",
                                t.ToString());
098         PropertyInfo props = t.GetProperty("Name");
099         object[] o = props.GetCustomAttributes(typeof(AppInfoAttribute), false);
100         Attribute[] attArray = (Attribute[])o;
101         foreach (Attribute att in attArray)
102         {
103             if (att is AppInfoAttribute)
104             {
105                 AppInfoAttribute ai = (AppInfoAttribute)att;
106                 Console.WriteLine("作者: {0}, 版本: {1}, 平台: {2}", ai.Name, ai.Version,
                                ai.Platform);
107             }
108         }
109     }
110     static void GetMethodParam(Type t)
111     {
112         Console.WriteLine("现在开始通过反射获取 {0} 类的方法参数特性信息",
                                t.ToString());
113         MethodInfo mti = t.GetMethod("Print");
114         ParameterInfo[] Params = mti.GetParameters();
115         foreach (ParameterInfo p in Params)
116         {
```

```
117         object[] o = p.GetCustomAttributes(typeof(AppInfoAttribute), false);
118         Attribute[] attArray = (Attribute[])o;
119         foreach (Attribute att in attArray)
120         {
121             if (att is AppInfoAttribute)
122             {
123                 AppInfoAttribute ai = (AppInfoAttribute)att;
124                 Console.WriteLine("作者: {0}, 版本: {1}, 平台: {2}", ai.Name,
125                                     ai.Version, ai.Platform);
126             }
127         }
128     }
129     static void GetMethodReturnAttribute(Type t)
130     {
131         Console.WriteLine("现在开始通过反射获取 {0} 类的方法返回值的特性信息",
132                             t.ToString());
133         MethodInfo mti = t.GetMethod("Sing");
134         object[] o = mti.ReturnTypeCustomAttributes.GetCustomAttributes
135             (typeof(AppInfoAttribute), false);
136         Attribute[] attArray = (Attribute[])o;
137         foreach (Attribute att in attArray)
138         {
139             if (att is AppInfoAttribute)
140             {
141                 AppInfoAttribute ai = (AppInfoAttribute)att;
142                 Console.WriteLine("作者: {0}, 版本: {1}, 平台: {2}", ai.Name, ai.Version,
143                                     ai.Platform);
144             }
145         }
146     }
147     static void Main(string[] args)
148     {
149         GetAssemblyAttribute("Attribute2");
150         GetTypeAttribute(typeof(Person));
151         GetPropAttribute(typeof(Person));
152         GetMethodReturnAttribute(typeof(Hobby));
153         GetMethodParam(typeof(Person));
154         Console.ReadKey();
155     }
```

这段代码因为要使用类库，所以使用 `using Attribute1` 语句引用了类库。其次要在项目的引用中物理引用自定义的类库项目。第 3 行代码引用了 `System.Reflection` 名称空间。这是因为要在项目中使用这个

名称空间内的 Method 等类型。

第 4 行代码使用特性目标为 assembly 的语法指定了程序集使用自定义的特性 AssemblyVersion。

代码的第 8 行定义了一个自定义的特性 AppInfoAttribute。对这个自定义的特性指定了特性 AttributeUsage，并且它的定位参数使用了 AttributeTargets.All 枚举，意思是可以对所有的程序实体指定本特性。

代码的第 10 行到第 24 行为代码的简单起见，只定义了三个特性的属性成员。第 25 行和第 31 行开始，定义了两个构造方法，一个带有三个参数，另一个带有两个参数。

第 38 行代码定义了一个接口 Hobby，在第 40 行，对其的方法 Sing 指定了自定义的特性。自定义特性的目标是 return。意思是对方法的返回值指定特性。需要注意，在这里，特性目标是不能省略的。如果不加特性目标，就会认为这个特性是对方法指定的。

代码的第 45 行定义了一个类 Person，第 44 行对它指定了自定义特性。特性目标为 type。在这里，这个特性目标可以省略。因为不加特性目标，这个特性在这个位置上也是对类指定的。

在第 47 行，使用 property 目标对属性 Name 指定了特性，在这里，property 目标也可以省略。在第 63 行，对实例方法 Print 的参数 s 指定了特性。特性目标为 param。这里的特性目标也可以省略。因为它是直接在程序实体前面指定的。

第 71 行代码定义了一个 GetAssemblyAttribute 方法，它用来获取程序集上指定的特性信息。要获得程序集上指定的特性信息，首先要获得这个程序集。获得程序集需要使用 .Net 类库中的 Assembly 类的 Load 方法，方法需要加载的程序集名称。程序集名称可以在项目的项目属性中查看到。方法是在项目名称上单击右键，然后单击“属性”。如图 20-1 所示。

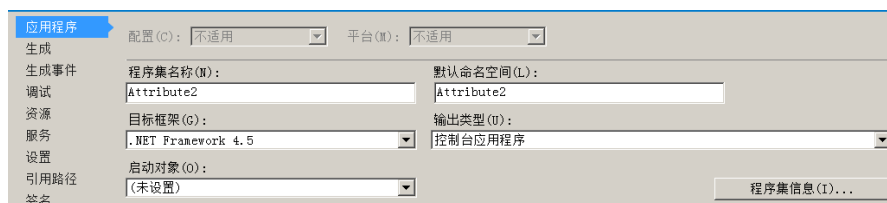


图 20-1 项目属性

在窗口中有一个“程序集名称”项，在它里面就是本程序集的名称。接下来使用程序集变量调用 GetCustomAttribute 方法，传入的参数是自定义特性的类型。然后将 Attribute 类型显式转换为 AssemblyVersionAttribute 类型进行输出。

第 82 行代码定义的 GetTypeAttribute 方法是输出类的自定义特性信息，这个方法的内容前面介绍过，就不再赘述了。

第 95 行代码定义的 GetPropAttribute 方法用来通过反射得到属性中的自定义特性信息。在方法内部使用了 Type 类的变量调用了 GetProperty(“Name”)方法。方法的参数是要获得其属性信息的属性名称。这个方法返回了一个 PropertyInfo 类型的变量，接下来使用这个变量调用 GetCustomAttributes(typeof(AppInfoAttribute), false) 方法，这个方法返回一个 object 类型的数组。方法有两个参数，第一个参数是要获得的自定义特性的类型，第二个 bool 值的作用是否是向这个属性的继承链上进行查询。如果为 false 则只反射本属性的特性信息。第 100 行将 object 数组显式转换为 Attribute 数组。接下来就好办了，使用循环将 Attribute 类型转换为自定义特性的类型，然后打印输出。

第 110 行定义了一个 GetMethodParam 方法，它是通过反射得到方法的参数的特性信息。它的实现方法和前一个方法类似，不再多说。

第 129 行代码定义了一个 GetMethodReturnAttribute 方法，它用来得到方法的返回值的自定义特性信息。它主要使用了 MethodInfo 类的 ReturnTypeCustomAttributes 属性。这个属性返回的是 ICustomAttributeProvider 接口变量。然后使用这个变量调用 GetCustomAttributes 方法以返回一个特性数组。

这段代码的执行结果如下：

现在开始通过反射获取 Attribute2 程序集的特性信息

程序集版本: 1.0.0.1

现在开始通过反射获取 Attribute1.Person 类的特性信息

作者: 秋叶无声, 版本: 3.0, 平台: Windows XP

现在开始通过反射获取 Attribute1.Person 类的属性的特性信息

作者: 秋叶无声, 版本: 2.0, 平台: Windows 7

现在开始通过反射获取 Attribute1.Hobby 类的方法返回值的特性信息

作者: Tom and Jarry, 版本: 1.0, 平台: Windows 8

现在开始通过反射获取 Attribute1.Person 类的方法参数特性信息

作者: 秋叶无声, 版本: 2.0, 平台: Windows 8

20.9 分部类型的特性

最后再来介绍一下分部类的特性指定。分部类的特性是通过组合分部类的各个部分类确定的, 如果在分部类的多个部分指定了同一个特性, 则视为对该类多次指定了同一特性。代码实例如下:

```
001     using System;
002     using System.Reflection;
003     namespace Attribute1
004     {
005         [AttributeUsage(AttributeTargets.All, AllowMultiple = true)]
006         class AppInfoAttribute : Attribute
007         {
008             public string Platform
009             {
010                 get;
011                 set;
012             }
013             public string Name
014             {
015                 get;
016                 set;
017             }
018             public string Version
019             {
020                 get;
021                 set;
022             }
023             public AppInfoAttribute(string name, string version, string platform)
024             {
025                 Name = name;
026                 Version = version;
027                 Platform = platform;
028             }
029             public AppInfoAttribute(string name, string version)
030             {
031                 Name = name;
032                 Version = version;
```



```
033         Platform = "Windows 8";
034     }
035 }
036 public class AppVersionAttribute : Attribute
037 {
038     public string Version
039     {
040         get;
041         set;
042     }
043     public AppVersionAttribute(string version)
044     {
045         Version = version;
046     }
047 }
048 [AppInfo("秋叶无声", "3.0", "Windows XP")]
049 [AppVersion("1.0.0.0")]
050 partial class Person
051 {
052     public string Name
053     {
054         get;
055         set;
056     }
057 }
058 [AppInfo("秋叶无声", "2.0", "Windows 7")]
059 partial class Person
060 {
061     public string ID
062     {
063         get;
064         set;
065     }
066     public int Age
067     {
068         get;
069         set;
070     }
071     public void Print(string s)
072     {
073         s = "#";
074         Console.WriteLine(s + " 姓名: {0}, 年龄: {1}, 身份证号: {2}", Name, Age, ID);
075     }
076 }
```

```
077     class Program
078     {
079         static void GetTypeAttribute(Type t)
080         {
081             Console.WriteLine("现在开始通过反射获取 {0} 类的特性信息", t.ToString());
082             Attribute[] attArray = Attribute.GetCustomAttributes(t);
083             foreach (Attribute att in attArray)
084             {
085                 if (att is AppInfoAttribute)
086                 {
087                     AppInfoAttribute ai = (AppInfoAttribute)att;
088                     Console.WriteLine("作者: {0}, 版本: {1}, 平台: {2}", ai.Name,
089                                     ai.Version, ai.Platform);
090                 }
091                 if (att is AppVersionAttribute)
092                 {
093                     AppVersionAttribute ava = (AppVersionAttribute)att;
094                     Console.WriteLine("程序集版本: {0}", ava.Version);
095                 }
096             }
097         }
098         static void Main(string[] args)
099         {
100             GetTypeAttribute(typeof(Person));
101             Console.ReadKey();
102         }
103     }
```

这段代码定义了两个特性类和一个分部类。对每个分部类的组成部分都指定了 AppInfo 特性，对特性 AppVersion 只指定了一次。最后，它们都被编译器合并到了一起，并通过反射得到输出。代码的执行结果如下：

现在开始通过反射获取 Attribute1.Person 类的特性信息

作者: 秋叶无声, 版本: 3.0, 平台: Windows XP

程序集版本: 1.0.0.0

作者: 秋叶无声, 版本: 2.0, 平台: Windows 7

从代码的执行结果可以看到，对每个分部类的部分进行的 AppInfo 特性的指定被视同对同一类指定了两次同类的特性。

第二十一章 C#语言高级特性

21.1 扩展方法

扩展方法用于向已经编译好的程序集（例如 .Net 类库）添加功能。一旦一个程序集编译好了之后，要想向其中添加功能，只有拥有程序集的代码才行。而扩展方法允许在不改动程序集的情况下，为程序集扩展功能。扩展方法必须是静态类中的静态方法。下面是代码实例：

```

001 using System;
002 namespace ExtensionMethod1
003 {
004     static class Extension
005     {
006         public static void Cube(this int i)
007         {
008             int a = i * i * i;
009             Console.WriteLine("a 的立方值是: {0}", a);
010         }
011     }
012     class Program
013     {
014         static void Main(string[] args)
015         {
016             int a = 100;
017             Console.WriteLine("a 的平方值是: {0}", a * a);
018             a.Cube();
019             Console.ReadKey();
020         }
021     }
022 }

```

扩展方法除了必须是静态类的静态方法之外，它的第一个参数必须是以 `this` 关键字修饰的参数。这个参数的类型就是要扩展的类型。在这段代码中，第 4 行代码定义了一个静态类，第 6 行代码定义了一个静态方法。它带有一个 `this` 关键字修饰的 `int` 类型的参数。第 8 行定义了一个局部变量，它用来接收参数的立方值。第 9 行代码打印输出这个结果。这个扩展方法因为 `this` 关键字修饰的是一个 `int` 类型的参数，所以它扩展了 .Net 类库中的 `int` 类型的功能。

第 16 行代码定义了一个局部变量 `a`，第 18 行可以使用这个变量调用它的扩展方法。这就好像这个扩展方法是 `int` 结构的实例方法一样。这段代码的执行结果如下：

```

a 的平方值是: 10000
a 的立方值是: 1000000

```

21.1.1 为自定义类添加扩展方法

除了为 .Net 类库中的类添加扩展方法外，也可以为自定义类添加扩展方法。实例代码如下：

```

001 using System;
002 namespace ExtensionMethod1
003 {
004     class Hummingbird
005     {

```

```

006         public void Show()
007         {
008             Console.WriteLine("我是一种鸟，我喜欢吃花蜜");
009             Console.WriteLine("我能前后随意飞行");
010         }
011     }
012     static class Extension
013     {
014         public static void HumExt(this Hummingbird h, string s)
015         {
016             h.Show();
017             Console.WriteLine(s);
018         }
019     }
020     class Program
021     {
022         static void Main(string[] args)
023         {
024             Hummingbird myHummingbird = new Hummingbird();
025             myHummingbird.HumExt("我拍打翅膀能发出嗡嗡声");
026             Console.ReadKey();
027         }
028     }
029 }

```

这段代码定义了一个类 Hummingbird，在类中只有一个方法 Show。在第 12 行定义了一个静态类，在其中定义了一个扩展方法 HumExt。扩展方法可以包括多个参数。第一个参数可以用来调用被扩展的类中的成员。代码的执行结果如下：

```

我是一种鸟，我喜欢吃花蜜
我能前后随意飞行
我拍打翅膀能发出嗡嗡声

```

实际上，扩展方法背后发生的调用过程是将调用扩展方法的实例作为扩展方法的参数。这个隐式的转换过程是由 C# 的编译器来完成的。以上例来介绍，实际的调用代码如下：

```

001     using System;
002     namespace ExtensionMethod1
003     {
004         class Hummingbird
005         {
006             public void Show()
007             {
008                 Console.WriteLine("我是一种鸟，我喜欢吃花蜜");
009                 Console.WriteLine("我能前后随意飞行");
010             }
011         }
012         static class Extension

```

```

013     {
014         public static void HumExt(this Hummingbird h, string s)
015         {
016             h.Show();
017             Console.WriteLine(s);
018         }
019     }
020     class Program
021     {
022         static void Main(string[] args)
023         {
024             Hummingbird myHummingbird = new Hummingbird();
025             Extension.HumExt(myHummingbird, "我拍打翅膀能发出嗡嗡声");
026             Console.ReadKey();
027         }
028     }
029 }

```

代码中的类与扩展方法没有进行改动，但是在第 25 行调用的时候，是以静态类的类名来引用静态方法的。然后将第 24 行代码定义的 Hummingbird 类的实例作为参数传递给扩展方法。调用方法虽改变了，但结果和原来是一样的。代码的执行结果如下：

```

我是一种鸟，我喜欢吃花蜜
我能前后随意飞行
我拍打翅膀能发出嗡嗡声

```

从代码的执行结果可以看出，实际上扩展方法只是 C# 编译器对 C# 语法的一种巧妙的使用。但却极大地方便了代码的扩展和编写。

21.1.2 使用扩展方法扩展接口

实际上，在 .Net 类库中有许多扩展方法，它们都以类库的形式存在着。下面使用类库实现的方式来对一个接口进行扩展。首先定义一个接口，然后定义它的实现类。代码如下：

```

001     using System;
002     namespace ExtensionMethod1
003     {
004         interface Hobby
005         {
006             void Sing();
007             void Dance();
008         }
009         class Student:Hobby
010         {
011             public void Sing()
012             {
013                 Console.WriteLine("我能唱歌");
014             }
015             public void Dance()
016             {

```

```

017         Console.WriteLine("我能跳舞");
018     }
019 }
020 class Program
021 {
022     static void Main(string[] args)
023     {
024         Console.ReadKey();
025     }
026 }
027 }

```

这段代码定义了一个接口和一个实现类，在类 Program 中的 Main 方法中还没有写类的实例的创建和方法的调用。下面再建立一个类库，代码如下：

```

001 using System;
002 namespace ExtentionLibrary
003 {
004     public static class ExtentionHobby
005     {
006         public static void Play(this object h)
007         {
008             Console.WriteLine("我能弹钢琴");
009         }
010     }
011 }

```

在类库中定义一个扩展方法，只不过这个扩展方法使用的参数是 object 类型。它可以为所有的类型提供这个扩展方法。接下来让原来的类引用这个类库。为其 Main 方法中添加如下代码：

```

static void Main(string[] args)
{
    Hobby h = new Student();
    h.Sing();
    h.Dance();
    h.Play();
    Console.ReadKey();
}

```

在这段代码中，定义了一个接口的变量指向了实现类的实例。然后用接口的变量就可以引用这个新添加的扩展方法。也可以说，这个扩展方法扩展了这个接口。代码的执行结果如下：

```

我能唱歌
我能跳舞
我能弹钢琴

```

其实，这段代码有一个缺陷，就是扩展方法的参数类型是 object 针对的是所有类型，它不能针对 Hobby 接口专有扩展。下面再演示一下 .Net 类库中的实现方式。先将接口的定义放在一个类库中，类库的代码如下：

```

001 using System;
002 namespace InterfaceLibrary

```

```
003 {
004     public interface Hobby
005     {
006         void Sing();
007         void Dance();
008     }
009     public class Student : Hobby
010     {
011         public void Sing()
012         {
013             Console.WriteLine("我能唱歌");
014         }
015         public void Dance()
016         {
017             Console.WriteLine("我能跳舞");
018         }
019     }
020 }
```

在这个类库中包含了接口的定义和实现类的定义，需要注意，接口和类都需要 public 访问性，以便外部程序集能够访问。接下来再新建一个类库，这个类库专门用于扩展。让这个类库引用定义和实现接口的类库。代码如下：

```
001 using System;
002 using InterfaceLibrary;
003 namespace ExtentionLibrary
004 {
005     public static class ExtentionHobby
006     {
007         public static void Play(this Hobby h)
008         {
009             Console.WriteLine("我能弹钢琴");
010         }
011     }
012 }
```

在代码的第 2 行使用 using 语句引用定义接口和实现接口的类库。然后在扩展方法中就可以使用接口的名称定义参数。然后在调用类中引用这两个类库，代码如下：

```
001 using System;
002 using ExtentionLibrary;
003 using InterfaceLibrary;
004 namespace ExtensionMethod1
005 {
006     class Program
007     {
008         static void Main(string[] args)
009         {
```

```
010         Hobby h = new Student();
011         h.Sing();
012         h.Dance();
013         h.Play();
014         Console.ReadKey();
015     }
016 }
017 }
```

在代码的开始需要引用这两个类库的名称空间以简化代码的输入。在物理上也需要引用创建好的两个类库。然后在 Main 方法中就可以使用它们了。这就是 .Net 类库的实现方式的模拟。如果在编译过程中出现 .Net 架构版本不一致的问题，需要在各个项目的项目属性中统一定义 .Net 的框架版本。

21.2 用户自定义转换

用户自定义转换的作用是扩充预定义的隐式转换和显式转换。implicit 关键字用来定义用户定义的隐式转换；explicit 关键字用来定义用户定义的显式转换。下面是代码实例：

```
001 using System;
002 namespace AdFeatures1
003 {
004     class Person
005     {
006         public string Name;
007         public string Age;
008         public string ID;
009         public Person(string name, string age, string id)
010         {
011             this.Name = name;
012             this.Age = age;
013             this.ID = id;
014         }
015     }
016     class Teacher : Person
017     {
018         public void Occupation()
019         {
020             Console.WriteLine("我的职业是教学生");
021         }
022         public Teacher(string name, string age, string id)
023             : base(name, age, id)
024         { }
025     }
026     class Volunteer : Person
027     {
028         public void Occupation()
029         {
030             Console.WriteLine("我的业余工作是志愿者");
```



```

031     }
032     public Volunteer(string name, string age, string id)
033         : base(name, age, id)
034     { }
035 }
036 class Program
037 {
038     static void Main(string[] args)
039     {
040         Teacher myTeacher = new Teacher("秋叶无声", "30", "123456");
041         Volunteer myVolunteer = new Volunteer("Tom", "35", "654321");
042         myTeacher.Occupation();
043         myVolunteer.Occupation();
044         Teacher myTeacher1 = myVolunteer;
045         Volunteer myVolunteer1 = myTeacher;
046         Teacher myTeacher2 = (Teacher)myVolunteer;
047         Volunteer myVolunteer2 = (Volunteer)myTeacher;
048         Console.ReadKey();
049     }
050 }
051 }

```

在这段代码的第 4 行定义了一个基类 Person，第 16 行的类 Teacher 和第 26 行的类 Volunteer 都从类 Person 继承。在类 Program 中的 Main 方法中，分别定义了 Teacher 和 Volunteer 类的实例。然后在第 44 行和第 45 行定义了 Teacher 类和 Volunteer 类的变量，将 myVolunteer 和 myTeacher 以隐式转换的方式赋值给它们。第 46 行和第 47 行再次定义了这两个类的变量，然后以显式转换的方式为它们赋值。但是，因为这两个类之间没有继承关系。所以编译器不允许它们之间进行隐式和显式转换。编译器会报错，错误信息如下：

```

错误 1    无法将类型“AdFeatures1.Volunteer”隐式转换为“AdFeatures1.Teacher” H:\书稿 new\
书稿源码\第二十一章\AdFeatures1\Program.cs    44    34    AdFeatures1
错误 2    无法将类型“AdFeatures1.Teacher”隐式转换为“AdFeatures1.Volunteer” H:\书稿 new\
书稿源码\第二十一章\AdFeatures1\Program.cs    45    38    AdFeatures1
错误 3    无法将类型“AdFeatures1.Volunteer”转换为“AdFeatures1.Teacher” H:\书稿 new\书稿源
码\第二十一章\AdFeatures1\Program.cs 46    34    AdFeatures1
错误 4    无法将类型“AdFeatures1.Teacher”转换为“AdFeatures1.Volunteer” H:\书稿 new\书稿源
码\第二十一章\AdFeatures1\Program.cs 47    38    AdFeatures1

```

既然，它们之间不能转换，那么将它们先转换成 object 类型，然后再转换成目标类型会怎么样呢？代码修改如下：

```

001     static void Main(string[] args)
002     {
003         Teacher myTeacher = new Teacher("秋叶无声", "30", "123456");
004         Volunteer myVolunteer = new Volunteer("Tom", "35", "654321");
005         myTeacher.Occupation();
006         myVolunteer.Occupation();
007         Teacher myTeacher1 = (Teacher)(object)myVolunteer;

```

```

008     Volunteer myVolunteer1 = (Volunteer)(object)myTeacher;
009     Console.ReadKey();
010 }

```

这时，在编辑代码的时候，编译器不会报错了。因为从编译时来看，这样的转换语法是允许的。下面将它进行编译，但这时编译器会抛出异常。接下来修改代码，捕捉异常。代码如下：

```

001     static void Main(string[] args)
002     {
003         Teacher myTeacher = new Teacher("秋叶无声", "30", "123456");
004         Volunteer myVolunteer = new Volunteer("Tom", "35", "654321");
005         myTeacher.Occupation();
006         myVolunteer.Occupation();
007         try
008         {
009             Teacher myTeacher1 = (Teacher)(object)myVolunteer;
010         }
011         catch (Exception e)
012         {
013             Console.WriteLine(e.Message);
014         }
015         try
016         {
017             Volunteer myVolunteer1 = (Volunteer)(object)myTeacher;
018         }
019         catch (Exception e)
020         {
021             Console.WriteLine(e.Message);
022         }
023         Console.ReadKey();
024     }

```

这段代码在进行显式转换的时候，进行了代码的捕捉。下面来看结果：

我的职业是教学生

我的业余工作是志愿者

无法将类型为“AdFeatures1.Volunteer”的对象强制转换为类型“AdFeatures1.Teacher”。

无法将类型为“AdFeatures1.Teacher”的对象强制转换为类型“AdFeatures1.Volunteer”。

这两个异常是无效转换异常。可以看到，无论使用什么办法，都不可以将不相关的两个类型进行互相转换。如果要在它们之间进行转换，只有自定义一个转换出来。下面是代码实例：

```

001     using System;
002     namespace AdFeatures1
003     {
004         class Person
005         {
006             public string Name;
007             public string Age;
008             public string ID;

```

```
009     public Person(string name, string age, string id)
010     {
011         this.Name = name;
012         this.Age = age;
013         this.ID = id;
014     }
015     public void Print()
016     {
017         Console.WriteLine("信息: 姓名: {0}, 年龄: {1}, ID: {2}", Name, Age, ID);
018     }
019 }
020 class Teacher : Person
021 {
022     public void Occupation()
023     {
024         Console.WriteLine("我的职业是教学生");
025     }
026     public Teacher(string name, string age, string id)
027         : base(name, age, id)
028     { }
029     public static implicit operator Teacher(Volunteer v)
030     {
031         Teacher p = new Teacher(v.Name, v.Age, v.ID);
032         return p;
033     }
034 }
035 class Volunteer : Person
036 {
037     public void Occupation()
038     {
039         Console.WriteLine("我的业余工作是志愿者");
040     }
041     public Volunteer(string name, string age, string id)
042         : base(name, age, id)
043     { }
044     public static implicit operator Volunteer(Teacher t)
045     {
046         return new Volunteer(t.Name, t.Age, t.ID);
047     }
048 }
049 class Program
050 {
051     static void Main(string[] args)
052     {
```

```

053         Teacher myTeacher = new Teacher("秋叶无声", "30", "123456");
054         Volunteer myVolunteer = new Volunteer("Tom", "35", "654321");
055         myTeacher.Print();
056         myVolunteer.Print();
057         Console.WriteLine("现在开始使用隐式转换");
058         Teacher myTeacher1 = myVolunteer;
059         myTeacher1.Print();
060         myTeacher1.Occupation();
061         Volunteer myVolunteer1 = myTeacher;
062         myVolunteer1.Print();
063         myVolunteer1.Occupation();
064         Console.ReadKey();
065     }
066 }
067 }

```

在类 Teacher 中，第 29 行代码定义的就是一个自定义的隐式转换。自定义的隐式转换必须是 public 访问性的，并且是静态的方法。在 public 和 static 之后，要使用 implicit 关键字表示这是一个隐式转换。然后要接 operator 关键字，再跟要转换到的类型。在方法参数中是源类型的变量。因此第 29 行代码的意思是要使用源类型为 Volunteer 类型的实例作为模板，将它转换为 Teacher 类型。在方法体内部就是创建一个 Teacher 类型的实例，构造方法的参数要使用传入的源类型实例的各个内部数据值。最后需要使用 return 语句返回这个新创建的实例（要转换到的目标类型）。实际上它完成的就是内部数据的赋值操作。

第 44 行代码中的 Volunteer 类型内部也定义了这样一个方法。这样一来，这两种类型之间就可以相互进行隐式转换了。编译器也不再会报错。代码的执行结果如下：

```

信息：姓名:秋叶无声，年龄：30，ID:123456
信息：姓名:Tom，年龄：35，ID:654321
现在开始使用隐式转换
信息：姓名:Tom，年龄：35，ID:654321
我的职业是教学生
信息：姓名:秋叶无声，年龄：30，ID:123456
我的业余工作是志愿者

```

从代码的执行结果看，它们之间确实好像完成了转换。其实背后发生了一次实例的创建过程。

为它们定义了隐式转换之后，显式转换自然是被允许的。下面修改代码，代码如下：

```

001     static void Main(string[] args)
002     {
003         Teacher myTeacher = new Teacher("秋叶无声", "30", "123456");
004         Volunteer myVolunteer = new Volunteer("Tom", "35", "654321");
005         myTeacher.Print();
006         myVolunteer.Print();
007         Console.WriteLine("现在开始使用隐式转换");
008         Teacher myTeacher1 = (Teacher)myVolunteer;
009         myTeacher1.Print();
010         myTeacher1.Occupation();
011         Volunteer myVolunteer1 = (Volunteer)myTeacher;
012         myVolunteer1.Print();

```

```

013         myVolunteer1.Occupation();
014         Console.ReadKey();
015     }

```

代码的第 8 行和第 11 行使用了显式转换，这时的运行结果和前面是一样的。结果如下：

信息：姓名:秋叶无声，年龄：30，ID:123456

信息：姓名:Tom，年龄：35，ID:654321

现在开始使用隐式转换

信息：姓名:Tom，年龄：35，ID:654321

我的职业是教学生

信息：姓名:秋叶无声，年龄：30，ID:123456

我的业余工作是志愿者

接下来，将代码中的隐式转换方法去掉，为它们再定义显式转换方法。代码如下：

```

public static explicit operator Teacher(Volunteer v)
{
    Teacher p = new Teacher(v.Name, v.Age, v.ID);
    return p;
}

```

这是 Teacher 类中的显式转换方法。Occupation 类中的显式转换方法如下：

```

public static explicit operator Volunteer(Teacher t)
{
    return new Volunteer(t.Name, t.Age, t.ID);
}

```

可以看到，显式转换方法只是把关键字 implicit 换成了 explicit。其它没有发生变化。这时在 Main 方法中使用显式转换语法是允许的，代码的运行结果和前面一样。但是，如果将它们的转换改成隐式转换，则编译器会报错，错误信息如下：

错误 1 无法将类型“AdFeatures1.Volunteer”隐式转换为“AdFeatures1.Teacher”。存在一个显式转换(是否缺少强制转换?) H:\书稿 new\书稿源码\第二十一章\AdFeatures1\Program.cs 58 34

AdFeatures1

错误 2 无法将类型“AdFeatures1.Teacher”隐式转换为“AdFeatures1.Volunteer”。存在一个显式转换(是否缺少强制转换?) H:\书稿 new\书稿源码\第二十一章\AdFeatures1\Program.cs 61 38

AdFeatures1

从结果可以看到，与隐式转换不同，定义了显式转换后，隐式转换的方式就不可以了。

另外，在定义自定义转换的时候，要转换到的目标并不一定和所在的类相同。下面看代码实例：

```

001 using System;
002 namespace AdFeatures1
003 {
004     class Person
005     {
006         public string Name;
007         public string Age;
008         public string ID;
009         public Person(string name, string age, string id)
010         {
011             this.Name = name;

```

```
012         this.Age = age;
013         this.ID = id;
014     }
015     public void Print()
016     {
017         Console.WriteLine("信息: 姓名:{0}, 年龄: {1}, ID: {2}", Name, Age, ID);
018     }
019 }
020 class Teacher : Person
021 {
022     public void Occupation()
023     {
024         Console.WriteLine("我的职业是教学生");
025     }
026     public Teacher(string name, string age, string id)
027         : base(name, age, id)
028     { }
029     /// <summary>
030     /// 在 Teacher 类中定义向 Volunteer 类的转换
031     /// </summary>
032     /// <param name="t">Teacher 类型</param>
033     /// <returns></returns>
034     public static explicit operator Volunteer(Teacher t)
035     {
036         Volunteer v = new Volunteer(t.Name, t.Age, t.ID);
037         return v;
038     }
039 }
040 class Volunteer : Person
041 {
042     public void Occupation()
043     {
044         Console.WriteLine("我的业余工作是志愿者");
045     }
046     public Volunteer(string name, string age, string id)
047         : base(name, age, id)
048     { }
049     /// <summary>
050     /// 在 Volunteer 类中定义向 Teacher 类的转换
051     /// </summary>
052     /// <param name="v">Volunteer 类型</param>
053     /// <returns></returns>
054     public static explicit operator Teacher(Volunteer v)
055     {
```

```

056         return new Teacher(v.Name, v.Age, v.ID);
057     }
058 }
059 class Program
060 {
061     static void Main(string[] args)
062     {
063         Teacher myTeacher = new Teacher("秋叶无声", "30", "123456");
064         Volunteer myVolunteer = new Volunteer("Tom", "35", "654321");
065         myTeacher.Print();
066         myVolunteer.Print();
067         Console.WriteLine("现在开始使用隐式转换");
068         Teacher myTeacher1 = (Teacher)myVolunteer;
069         myTeacher1.Print();
070         myTeacher1.Occupation();
071         Volunteer myVolunteer1 = (Volunteer)myTeacher;
072         myVolunteer1.Print();
073         myVolunteer1.Occupation();
074         Console.ReadKey();
075     }
076 }
077 }

```

在这段代码中，在 Teacher 类中并没有定义向 Teacher 类的转换，而是定义了向 Volunteer 类的转换。在 Volunteer 类中也是这样。但是，这不影响代码的运行结果。只要关于要转换的两种类型之间有这样的自定义转换就可以了，而不必管它们是定义在源类中还是目标类中。代码的执行结果如下：

```

信息：姓名:秋叶无声，年龄：30，ID:123456
信息：姓名:Tom，年龄：35，ID:654321
现在开始使用隐式转换
信息：姓名:Tom，年龄：35，ID:654321
我的职业是教学生
信息：姓名:秋叶无声，年龄：30，ID:123456
我的业余工作是志愿者

```

21.2.1 泛型类的自定义转换

对于已经存在的预定义转换，不能再对它们进行自定义转换。下面看代码实例：

```

001 using System;
002 namespace AdFeatures1
003 {
004     class Person
005     {
006         public string Name;
007         public string Age;
008         public string ID;
009         public Person(string name, string age, string id)
010         {

```

```

011         this.Name = name;
012         this.Age = age;
013         this.ID = id;
014     }
015     public void Print()
016     {
017         Console.WriteLine("信息: 姓名: {0}, 年龄: {1}, ID: {2}", Name, Age, ID);
018     }
019     public static implicit operator Person(Teacher t)
020     {
021         return new Person(t.Name, t.Age, t.ID);
022     }
023 }
024 class Teacher : Person
025 {
026     public void Occupation()
027     {
028         Console.WriteLine("我的职业是教学生");
029     }
030     public Teacher(string name, string age, string id)
031         : base(name, age, id)
032     { }
033 }
034 class Program
035 {
036     static void Main(string[] args)
037     {
038         Teacher myTeacher = new Teacher("秋叶无声", "30", "123456");
039         Person p = myTeacher;
040         p.Print();
041         Console.ReadKey();
042     }
043 }
044 }

```

在基类 Person 中定义了一个隐式的自定义转换，将派生类 Teacher 转换成基类。但是编译器会报告如下的错误：

```

错误 1    “AdFeatures1.Person.implicit operator AdFeatures1.Person(AdFeatures1.Teacher)”：不
          允许进行以派生类为转换源或目标的用户定义转换  H:\书稿 new\书稿源码\第二十一章
          \ExtensionMethod1\Program.cs    19    23    ExtensionMethod1

```

对于这样的预定义转换来说，是不允许再定义自定义的转换的。但是，在泛型类中，这就不同了。下面看一下泛型类中如何进行自定义的转换，代码如下：

```

001     using System;
002     namespace AdFeatures1
003     {

```



```
004     class Test1<T>
005     {
006         public T A
007         {
008             get;
009             set;
010         }
011         public Test1(T t)
012         {
013             A = t;
014         }
015     }
016     class Test2<T> : Test1<T>
017     {
018         public T B
019         {
020             get;
021             set;
022         }
023         public Test2(T t1, T t2)
024             : base(t1)
025         {
026             B = t2;
027         }
028         public static implicit operator Test1<int>(Test2<T> t2)
029         {
030             Test1<int> t = new Test1<int>((int)(object)t2.A);
031             return t;
032         }
033     }
034     class Program
035     {
036         static void Main(string[] args)
037         {
038             Test2<int> t2 = new Test2<int>(100, 200);
039             Test1<int> t1 = t2;
040             Console.WriteLine(t1.A);
041             Console.ReadKey();
042         }
043     }
044 }
```

在代码的第4行定义了泛型类 Test1，它带有一个类型参数 T。第16行的泛型类 Test2 也带有一个类型参数 T。Test2 派生于 Test1。在 Test2 中定义了一个转换方法。它将 Test2 类转换为 Test1 类。虽然 Test1 和 Test2 之间是继承关系。但是因为在定义转换方法的时候，Test1 已经指定了类型实参 int。所以它们被

看做没有关系的两个类。因此这样的转换是允许的。代码的执行结果如下：

100

但是，如果将它定义成如下的形式则是不允许的，代码如下：

```
public static implicit operator Test1<T>(Test2<T> t2)
```

因为类型形参完全相同，所以它们被看做就是派生类向基类的自定义转换。在编译时，编译器会报告如下的错误：

```
错误 1 “ AdFeatures1.Test2<T>.implicit operator AdFeatures1.Test1<T>(AdFeatures1.
Test2<T>)”：不允许进行以基类为转换源或目标的用户定义转换 H:\书稿 new\书稿源码\第二十一章
\ExtensionMethod1\Program.cs 28 23 ExtensionMethod1
```

但是，在泛型类中，对于特定类型的实参，还会有例外发生。实例代码如下：

```
001 using System;
002 namespace AdFeatures1
003 {
004     class Test1<T>
005     {
006         public T A
007         {
008             get;
009             set;
010         }
011         public Test1(T t)
012         {
013             A = t;
014         }
015         public static explicit operator T(Test1<T> t)
016         {
017             T t1 = t.A;
018             return t1;
019         }
020     }
021     class Program
022     {
023         static void Main(string[] args)
024         {
025             Test1<int> t1 = new Test1<int>(1000);
026             int i = (int)t1;
027             Console.WriteLine(i);
028             Console.ReadKey();
029         }
030     }
031 }
```

在泛型类 Test1 中，定义了一个自定义的显式转换，它将 Test1 类转换为类型实参类型。这样的转换是允许的，代码的执行结果如下：

1000

但是，当类型实参是 `object` 类型的时候，情况就不同了，它变成了派生类向基类的预定义转换。下面是 `Main` 方法中的代码：

```
001 static void Main(string[] args)
002 {
003     Test1<object> t1 = new Test1<object>(1000);
004     object i = (object)t1;
005     Console.WriteLine(i);
006     Console.ReadKey();
007 }
```

当在第 3 行创建 `Test1` 的实例的时候，为其传入的参数首先会进行装箱。但是在第 5 行打印 `i` 的值的时候却没发生拆箱操作。结果如下：

```
AdFeatures1.Test1`1[System.Object]
```

它打印的是泛型类的类型信息。其中的 `1` 指的是泛型类的类型形参。可以看到 `i` 的类型是泛型类。为什么没有发生拆箱操作呢？这是因为有如下的规则：

1. 如果存在从源类型到目标类型的预定义隐式转换，则从源类型到目标类型的所有自定义的转换（不论隐式或显式）都将被忽略。

2. 如果存在从源类型到目标类型的预定义显式转换，则从源类型到目标类型的所有自定义的显式转换将被忽略。

本段代码符合第一种情况，因为从泛型类到 `object` 类型之间存在隐式转换，所以自定义的转换被忽略了。这时第 4 行代码就是将一个泛型类的构造类型隐式转换成了 `object` 类型。虽然它用了显式转换的方式，但是实际上是隐式转换。这时，变量 `i` 的类型是一个 `object` 类型，它指向了一个构造类型的实例。而不是将 1000 装箱后的值。所以它不能打印出 1000。

在自定义转换的时候，还需要注意，接口类型之间不可以定义自定义转换。自定义转换方法的签名包括源类型和目标类型，不包括 `implicit` 和 `explicit` 关键字。所以在同一个类中不能定义具有相同源类型和目标类型的隐式转换和显式转换。在自定义转换操作的时候，应该本着一个原则来确定是该定义隐式转换还是定义显式转换。如果在转换过程中有信息丢失或异常可能发生，则应该定义成显式转换。如果转换过程丝毫没有问题，则应该定义成隐式转换。

最后需要注意的是，`is` 和 `as` 运算符不能用于自定义的判断和转换。

21.2.2 提升转换运算符

如果存在一个从不可以为空的值类型到另一个不可以为空的值类型之间的自定义转换，则存在一个它们之间的可以为空的类型之间的提升转换。下面是代码实例：

```
001 using System;
002 namespace AdFeatures1
003 {
004     struct TestA
005     {
006         private int a;
007         public TestA(int a)
008         {
009             this.a = a;
010         }
011         public static implicit operator TestA(TestB b)
012         {
013             TestA ta = new TestA(b.B);
```

```
014         return ta;
015     }
016     public int A
017     {
018         get
019         {
020             return a;
021         }
022         set
023         {
024             a = value;
025         }
026     }
027 }
028 struct TestB
029 {
030     private int b;
031     public TestB(int b)
032     {
033         this.b = b;
034     }
035     public int B
036     {
037         get
038         {
039             return b;
040         }
041         set
042         {
043             b = value;
044         }
045     }
046 }
047 class Program
048 {
049     static void Main(string[] args)
050     {
051         TestB myTestB = new TestB(1000);
052         TestA myTestA = myTestB;
053         Console.WriteLine(myTestA.A);
054         TestB myTestB1 = new TestB(2000);
055         TestB? myTestB2 = new TestB?(myTestB1);
056         TestA? myTestA1 = myTestB2;
057         Console.WriteLine(myTestA1.Value.A);
```

```

058         Console.ReadKey();
059     }
060 }
061 }

```

这段代码定义两个结构，这两个结构没有任何继承关系。在结构 TestA 中的第 11 行定义了一个自定义的隐式转换。这样，TestB 可以隐式转换到 TestA。在代码的第 51 行到第 53 行演示了这一转换过程。对于相对于它们的可空类型，存在着提升的自定义隐式转换。要将结构变为可空类型，首先需要先创建一个结构的实例。然后在第 55 行对它装包成为可空类型。然后看第 56 行的隐式转换，提升的隐式转换首先将 myTestB2 进行解包，然后执行其基本类型的自定义隐式转换。转换完成后，再对结果进行装包，装包的结果就是 TestA 的可空类型。代码的执行结果如下：

```

1000
2000

```

21.3 运算符重载

运算符重载就是对于已有的预定义的运算符，除了它们自己的预定义实现外，还可以在类或结构中自定义其实现方法，以便让运算符实现自定义类的功能。但是并不是所有的运算符都可以被重载，只有下列运算符可以被重载：

一元运算符：

+ - ! ~ ++ -- true false

二元运算符：

+ - * / % & | ^ << >> == != > < >= <=

下列运算符不可以被重载：

成员访问和方法调用运算符、=、&&、||、??、?:、=>、checked、unchecked、new、typeof、default、as 和 is 运算符。

另外，强制转换运算符()不可以被重载，它通过自定义转换来实现。元素访问[]运算符也不可以被重载，它通过索引器来实现。当重载一个二元运算符时，会隐式实现其相应的赋值运算符。例如重载+运算符后，它的运算符重载也是+=运算符的重载。在进行比较运算符的重载时，必须成对重载其相反意义的运算符。例如重载了==，就需要同时重载!=。类似如此的还有< 和 > 以及 <= 和 >=。下面是代码实例：

```

001 using System;
002 namespace AdFeatures1
003 {
004     class Person
005     {
006         public string Name;
007         public string Age;
008         public string ID;
009         public Person(string name, string age, string id)
010         {
011             this.Name = name;
012             this.Age = age;
013             this.ID = id;
014         }
015         public void Print()
016         {
017             Console.WriteLine("信息: 姓名: {0}, 年龄: {1}, ID: {2}", Name, Age, ID);

```

```

018     }
019     public static Person operator +(Person p1, Person p2)
020     {
021         Person p = new Person(p1.Name + " " + p2.Name, p1.Age + " " + p2.Age, p1.ID
            + " " + p2.ID);
022         return p;
023     }
024     public static bool operator ==(Person p1, Person p2)
025     {
026         if (p1.Name == p2.Name)
027             return true;
028         else
029             return false;
030     }
031     public static bool operator !=(Person p1, Person p2)
032     {
033         if (p1.Name != p2.Name)
034             return true;
035         else
036             return false;
037     }
038 }
039 class Program
040 {
041     static void Main(string[] args)
042     {
043         Person p1 = new Person("令狐冲", "26", "12345678");
044         Person p2 = new Person("任盈盈", "21", "87654321");
045         Person p3 = p1 + p2;
046         p3.Print();
047         Console.WriteLine("两个人的比较结果: {0}", p1 == p2);
048         Console.WriteLine("两个人再次比较的结果: {0}", p1 != p2);
049         Console.ReadKey();
050     }
051 }
052 }

```

在这段代码中定义了一个类 `Person`，它包含三个字段，分别是姓名、年龄和身份证号，都是 `string` 类型。在代码的第 19 行定义了 `+` 运算符的重载。运算符重载的语法是，运算符重载方法必须是 `public` 访问性的静态方法，然后跟方法的返回值类型，接下来是 `operator` 关键字，再接要重载的运算符。方法的参数就是要参加运算的两个类的实例。本段代码中二元 `+` 运算符的实现思路是，将每个实例的字段都拼接起来，然后用这个拼接起来的字符串创建新的实例。然后返回这个实例。

第 24 行和第 31 行的代码是通过比较两个实例的 `Name` 字段来返回 `bool` 值。

第 43 行代码和第 44 行代码定义了两个类的实例，然后第 45 行定义一个 `Person` 类型的变量，再将预先定义的两个实例相加，最后把结果赋值给这个 `Person` 类型的变量。代码的第 47 行和第 48 行使用重载的

比较运算符比较两个实例。

代码的执行结果如下：

信息：姓名：令狐冲 任盈盈，年龄：26 21，ID:12345678 87654321

两个人的比较结果：False

两个人再次比较的结果：True

因为重载了==运算符，所以第 31 行代码也可以简化为如下的代码：

```
public static bool operator !=(Person p1, Person p2)
{
    return !(p1 == p2);
}
```

这段代码在比较两个实例的时候，虽然比较的结果是预期的结果，但是在编译的时候，编译器会报告如下的警告信息：

警告 1 “AdFeatures1.Person” 定义运算符 == 或运算符 !=，但不重写 Object.GetHashCode() H:\书稿 new\书稿源码\第二十一章\ExtensionMethod1\Program.cs 4 11 ExtensionMethod1

警告 2 “AdFeatures1.Person” 定义运算符 == 或运算符 !=，但不重写 Object.Equals(object o) H:\书稿 new\书稿源码\第二十一章\ExtensionMethod1\Program.cs 4 11 ExtensionMethod1

这是因为，对于相等运算符和不相等运算符进行重载的时候，必须重写从 object 类继承来的 GetHashCode 方法与 Equals 方法。

21.3.1 GetHashCode 方法与 Equals 方法

默认情况下，对于值类型，使用==运算符比较的是操作数的值；对于引用类型，比较的是两个引用是否指向同一个实例。如果对于自定义类中的==相等比较运算符进行了运算符重载，则必须同时重写自定义类的 Equals 方法。Equals 方法是 object 基类的虚方法。它的作用是对两个实例进行比较，默认情况下，Equals 方法比较的方法和==运算符的比较方法是相同的，对于值类型，比较其按位相等性；对于引用类型，比较其是否引用同一个对象。当类中重载了运算符==时，必须重写 Equals 方法，让它的返回值返回与==运算符同样的结果。以便让使用 Equals 方法的类库代码，比如集合的类型的行为方式和自定义类的==运算符的方式一致。在重写 Equals 方法的同时，也必须同时重写 GetHashCode 方法，否则 Hashtable（类库中的类）可能不工作。总之，记住一点，重写 Equals 方法必须重写 GetHashCode 方法；重写 GetHashCode 方法也必须同时重写 Equals 方法。这是为了保证 .Net 类库中的一些类处理方法的一致性。

GetHashCode 方法是获得对象的哈希代码。哈希代码是一个用于标识对象的值。object 类提供的 GetHashCode 方法不保证对不同的对象返回一个唯一的哈希代码，因此，对象的哈希代码不能用作标识对象的唯一值。自定义值类型的时候，必须重写此方法。

重写 GetHashCode 方法的时候，需要按照下面的原则来重写：

1. 如果两个实例的比较结果相等，则这两个实例的 GetHashCode 方法应该返回相同的值。但是，如果两个实例的比较结果不相等，这两个实例的 GetHashCode 方法不一定返回不相同的值，这就是哈希代码的不唯一型。
2. 一个实例如果没有修改过内部数据信息，则总是返回同一个哈希代码值。

下面对代码进行修改，如下：

```
001 using System;
002 namespace AdFeatures1
003 {
004     class Person
005     {
006         public string Name;
007         public int Age;
```

```
008         public string ID;
009         public Person(string name, int age, string id)
010         {
011             this.Name = name;
012             this.Age = age;
013             this.ID = id;
014         }
015         public void Print()
016         {
017             Console.WriteLine("信息: 姓名:{0}, 年龄: {1}, ID:{2}", Name, Age, ID);
018         }
019         public static Person operator +(Person p1, Person p2)
020         {
021             Person p = new Person(p1.Name + " " + p2.Name, (p1.Age + p2.Age), p1.ID + "
022             " + p2.ID);
023             return p;
024         }
025         public static bool operator ==(Person p1, Person p2)
026         {
027             if (p1.Name == p2.Name && p1.Age == p2.Age)
028                 return true;
029             else
030                 return false;
031         }
032         public static bool operator !=(Person p1, Person p2)
033         {
034             return !(p1 == p2);
035         }
036         public override int GetHashCode()
037         {
038             return Name.GetHashCode() ^ Age;
039         }
040         public override bool Equals(object p)
041         {
042             if (this.Name == ((Person)p).Name && this.Age == ((Person)p).Age)
043                 return true;
044             else
045                 return false;
046         }
047     }
048     class Program
049     {
050         static void Main(string[] args)
```



```

051         Person p1 = new Person("令狐冲", 26, "12345678");
052         Person p2 = new Person("任盈盈", 21, "87654321");
053         Person p3 = p1 + p2;
054         p3.Print();
055         Console.WriteLine("两个人的比较结果: {0}", p1 == p2);
056         Console.WriteLine("两个人再次比较的结果: {0}", p1 != p2);
057         Person p4 = new Person("令狐冲", 26, "12345678");
058         Console.WriteLine("两个人的比较结果: {0}", p1 == p4);
059         Console.WriteLine("p1 的哈希码是: {0}", p1.GetHashCode());
060         Console.WriteLine("p4 的哈希码是: {0}", p4.GetHashCode());
061         Console.WriteLine("使用 Equals 比较 p1 与 p4: {0}", p1.Equals(p4));
062         Console.ReadKey();
063     }
064 }
065 }

```

这段代码对前面的代码进行了修改, 它将 Age 字段设置成 int 类型。在第 24 行的重载 == 运算符方法中, 对实例的 Name 和 Age 字段同时进行比较, 只有这两项都相同, 才认为实例相等。在第 35 行重写了基类的 GetHashCode 方法, 它返回的结果是 Name 字段的哈希码与 Age 字段相异或的结果。因为 string 类型重写了 GetHashCode 方法, 对于相同的字符串, 它的哈希码相同。又因为本段代码中的 == 运算符重载判断了两个字段, 因此这里使用 Name 字段的哈希码与 Age 字段相异或, 可以保证相等的实例的哈希码相同。

第 39 行重写了基类的 Equals 方法, Equals 方法和 GetHashCode 方法中, 有一个重写了, 另一个也必须重写。这是 .Net 类库的要求。Equals 方法的重写逻辑思想和 == 运算符重载方法的实现方法一样, 这样就保证了它们返回同样的结果。在重写 Equals 方法的时候, 需要注意它的参数是 object 类型的, 在判断的时候需要使用显式转换变成 Person 类型。

在代码的第 57 行又定义了一个变量 p4, 它进行实例化的时候, 参数和 p1 的实例相同。在第 59 行和第 60 行取得它们的哈希码, 可以看到, 它们的哈希码完全相同。第 61 行代码使用 Equals 方法比较 p1 与 p4, 得到的结果和使用 == 运算符得到的结果相同。代码的执行结果如下:

```

信息: 姓名: 令狐冲 任盈盈, 年龄: 47, ID: 12345678 87654321
两个人的比较结果: False
两个人再次比较的结果: True
两个人的比较结果: True
p1 的哈希码是: -1547411818
p4 的哈希码是: -1547411818
使用 Equals 比较 p1 与 p4: True

```

21.3.2 自增和自减运算符的重载

下面再来演示一下自增运算符的重载, 它和自减运算符的重载实现方式大致相同。自增运算符和自减运算符当运算符前置和后置的时候, 处理是不同的, 前置的时候, 操作数先进行自增或自减, 然后返回结果。如果运算符后置, 则先返回结果, 然后操作数再进行自增或自减。实例代码如下:

```

001     using System;
002     namespace AdFeatures1
003     {
004         class Point
005         {
006             public int X

```

```
007         {
008             get;
009             set;
010         }
011         public int Y
012         {
013             get;
014             set;
015         }
016         public Point(int x, int y)
017         {
018             this.X = x;
019             this.Y = y;
020         }
021         public void Show()
022         {
023             Console.WriteLine("当前坐标: {0}, {1}", X, Y);
024         }
025         //前置++
026         public static Point operator ++(Point p)
027         {
028             return new Point((p.X + 1), (p.Y) + 1);
029         }
030     }
031     class Program
032     {
033         static void Main(string[] args)
034         {
035             Point p1 = new Point(10, 15);
036             p1.Show();
037             p1++.Show();
038             p1.Show();
039             Console.ReadKey();
040         }
041     }
042 }
```

代码中首先定义了一个 Point 类, Point 类含有两个 int 类型的字段表示其横坐标和纵坐标。代码的第 26 行定义了一个前置自增运算符。它返回了一个 Point 类型的实例, 构造参数的值为参数的横坐标值和纵坐标值都加 1 的值。先不写后置++运算符重载。在代码的第 37 行就使用了后置运算符。这时的代码的执行结果如下:

```
当前坐标: 10, 15
当前坐标: 10, 15
当前坐标: 11, 16
```

可以看到, 没有后置自增运算符的重载, 但是却起到了后置自增运算符的作用。这是因为, 在 C# 中不需要

写后置自增运算符，如果再写一个++运算符的重载，编译器就会报错。在定义++运算符的时候只要使用 new 关键字返回一个新类型即可。如果编译器检测到使用了后置自增运算符，则它先返回原来操作数的一个临时变量。下面再添加前置运算符的操作，看一下执行结果。代码如下：

```
static void Main(string[] args)
{
    Point p1 = new Point(10, 15);
    p1.Show();
    p1++.Show();
    p1.Show();
    ++p1;
    p1.Show();
    Console.ReadKey();
}
```

执行结果如下：

```
当前坐标: 10, 15
当前坐标: 10, 15
当前坐标: 11, 16
当前坐标: 12, 17
```

21.4 匿名类

匿名类提供了一种方便的方法，可以把一组只读属性封装到单个对象中，而无需显式定义一个类型。它的类型名在背后由编译器生成，在代码中不能使用生成的类名。每个只读属性的类型由编译器进行推断。下面定义一个匿名类，代码如下：

```
001    using System;
002    namespace AdFeatures1
003    {
004        class Program
005        {
006            static void Main(string[] args)
007            {
008                var Person = new {Name = "秋叶无声", Age = 30, ID = "12345678"};
009                Console.WriteLine("Name: {0}, Age: {1}, ID: {2}", Person.Name, Person.Age,
010                                Person.ID);
011                Console.ReadKey();
012            }
013        }
```

代码的第 8 行定义了一个匿名类，并且用一个推断类型变量引用它。匿名类的创建使用 new 关键字，然后跟一对大括号。在大括号中定义属性名并赋值。各个属性之间以逗号分隔。最后一个属性不加分号。整个语句要以分号结尾。在第 9 行打印输出的时候，要以推断类型的变量 Person 来引用各个属性。代码的执行结果如下：

```
Name:秋叶无声, Age:30, ID:12345678
```

匿名类的各个属性是只读的，如果在代码中改变它们的值，就会发生错误。看下面的代码：

```
001    using System;
002    namespace AdFeatures1
```

```

003  {
004      class Program
005      {
006          static void Main(string[] args)
007          {
008              var Person = new {Name = "秋叶无声", Age = 30, ID = "12345678" };
009              Person.Name = "Tom and Jerry";
010              Person.Age = 28;
011              Person.ID = "87654321";
012              Console.WriteLine("Name: {0}, Age: {1}, ID: {2}", Person.Name, Person.Age,
013                               Person.ID);
014              Console.ReadKey();
015          }
016      }

```

这段代码在定义匿名类之后，又对它的各个属性进行了重新赋值。在编译时，编译器会报告如下的错误：

错误 1 无法对属性或索引器“AnonymousType#1.Name”赋值 -- 它是只读的 H:\书稿new\书稿源码\第二十一章\AdFeatures1\Program.cs 9 13 AdFeatures1

错误 3 无法对属性或索引器“AnonymousType#1.ID”赋值 -- 它是只读的 H:\书稿new\书稿源码\第二十一章\AdFeatures1\Program.cs 11 13 AdFeatures1

错误 2 无法对属性或索引器“AnonymousType#1.Age”赋值 -- 它是只读的 H:\书稿new\书稿源码\第二十一章\AdFeatures1\Program.cs 10 13 AdFeatures1

匿名类的属性还可以是已有的自定义类的成员。下面是代码实例：

```

001  using System;
002  namespace AdFeatures1
003  {
004      class Person
005      {
006          public string Name
007          {
008              get;
009              set;
010          }
011          public int Age
012          {
013              get;
014              set;
015          }
016          public string ID
017          {
018              get;
019              set;
020          }

```

```

021     public Person(string name, int age, string id)
022     {
023         Name = name;
024         Age = age;
025         ID = id;
026     }
027 }
028 class Program
029 {
030     static void Main(string[] args)
031     {
032         Person p1 = new Person("Tom and Jarry", 28, "abce1234");
033         var Person = new {Name = p1.Name, Age = p1.Age, ID = p1.ID };
034         Console.WriteLine("Name: {0}, Age: {1}, ID: {2}", Person.Name, Person.Age,
035                             Person.ID);
036         Console.ReadKey();
037     }
038 }

```

本段代码中首先定义了一个类 Person，在其中含有三个属性。第 32 行定义了一个 Person 类的实例。第 33 行定义了一个匿名类的变量并进行了初始化。匿名类的变量和自定义类的名称相同，这时，它将隐藏掉自定义的类的名称。在初始化过程中，匿名类的各个属性都采用自定义类实例的属性值进行赋值。代码的执行结果如下：

```
Name:Tom and Jarry, Age:28, ID:abce1234
```

如果两个匿名类型的初始值设定项以相同的顺序定义了相同名称和类型的属性，则它们会生成相同匿名类型的实例。下面是代码实例：

```

001     using System;
002     namespace AdFeatures1
003     {
004         class Program
005         {
006             static void Main(string[] args)
007             {
008                 var Person1 = new {Name = "秋叶无声", Age = 27, ID = "12456tew" };
009                 Console.WriteLine("Name: {0}, Age: {1}, ID: {2}", Person1.Name, Person1.Age,
010                                     Person1.ID);
011                 var Person2 = new { Name = "Tom and Jarry", Age = 30, ID = "6574321" };
012                 Console.WriteLine("Name: {0}, Age: {1}, ID: {2}", Person2.Name, Person2.Age,
013                                     Person2.ID);
014                 Person2 = Person1;
015                 Console.WriteLine("Name: {0}, Age: {1}, ID: {2}", Person2.Name, Person2.Age,
016                                     Person2.ID);
017                 Console.ReadKey();
018             }
019         }
020     }

```

```
016     }
017 }
```

本程序中，第 8 行代码定义的匿名类和第 10 行代码定义的匿名类中的属性名称和类型都相同，因此它们是同一类型。在第 12 行可以使用赋值运算符对它们的变量进行赋值。赋值后，Person1 和 Person2 都指向同一实例。代码的执行结果如下：

```
Name:秋叶无声, Age:27, ID:12456tew
Name:Tom and Jarry, Age:30, ID:6574321
Name:秋叶无声, Age:27, ID:12456tew
```

21.5 构建迭代器和可枚举类型

对于一个数组来说，前面已经接触过了，它可以使用 foreach 语句迭代输出数组的内部成员。实例代码如下：

```
001 using System;
002 namespace AdFeatures2
003 {
004     class Program
005     {
006         static void Main(string[] args)
007         {
008             string[] s = new string[] { "薄雾浓云愁永昼", "瑞脑销金兽", "佳节又重阳",
009                                     "玉枕纱橱", "半夜凉初透", "东篱把酒黄昏后", "有暗香盈袖", "莫道不消魂",
010                                     "帘卷西风", "人比黄花瘦" };
011             foreach (string s1 in s)
012             {
013                 Console.WriteLine(s1);
014             }
015             Console.ReadKey();
016         }
017     }
018 }
```

这段代码能够以 foreach 语句迭代输出数组 s 中的内容，是因为数组是一个可枚举的类型，它实现了 .Net 类库中的 IEnumerable 和 IEnumerator 接口。这段代码的执行结果如下：

```
薄雾浓云愁永昼
瑞脑销金兽
佳节又重阳
玉枕纱橱
半夜凉初透
东篱把酒黄昏后
有暗香盈袖
莫道不消魂
帘卷西风
人比黄花瘦
```

如果自定义一个类，也想要使用 foreach 语句像上面这样迭代输出结果行不行呢？下面定义一段代码来试一下：

```
001 using System;
```

```
002 namespace AdFeatures1
003 {
004     class Person
005     {
006         public string Name;
007         public int Age;
008         public string ID;
009         public Person(string name, int age, string id)
010         {
011             this.Name = name;
012             this.Age = age;
013             this.ID = id;
014         }
015         public void Print()
016         {
017             Console.WriteLine("职员信息: 姓名:{0}, 年龄: {1}, ID:{2}", Name, Age, ID);
018         }
019     }
020     class Company
021     {
022         private Person[] clerk;
023         public Company()
024         {
025             clerk = new Person[5];
026             clerk[0] = new Person("令狐冲", 23, "0001");
027             clerk[1] = new Person("任盈盈", 20, "0002");
028             clerk[2] = new Person("向问天", 30, "0003");
029             clerk[3] = new Person("任我行", 50, "0004");
030             clerk[4] = new Person("风清扬", 60, "0005");
031         }
032     }
033     class Program
034     {
035         static void Main(string[] args)
036         {
037             Company newCompany = new Company();
038             foreach (Person p in newCompany)
039             {
040                 p.Print();
041             }
042             Console.ReadKey();
043         }
044     }
045 }
```

在这段代码中首先定义了一个 Person 类，接下来又定义了一个 Company 类，在 Company 类中含有一个 Person 类型的数组。在第 23 行的构造方法中对数组进行了初始化。在 Main 方法中，定义了一个 Company 类的实例，然后意图使用 foreach 语句迭代输出类中的 clerk 数组中的成员信息。但是不幸的是，编译器会报错。错误信息如下：

错误 1 “AdFeatures1.Company” 不包含 “GetEnumerator” 的公共定义，因此 foreach 语句不能作用于 “AdFeatures1.Company” 类型的变量 E:\ 书 稿 new\ 书 稿 源 码 \ 第 二 十 一 章 \ExtensionMethod1\Program.cs 38 13 ExtensionMethod1

这是因为，这个自定义类没有实现 GetEnumerator 方法的原因。下面修改代码如下：

```
001 using System;
002 using System.Collections;
003 namespace AdFeatures1
004 {
005     class Person
006     {
007         public string Name;
008         public int Age;
009         public string ID;
010         public Person(string name, int age, string id)
011         {
012             this.Name = name;
013             this.Age = age;
014             this.ID = id;
015         }
016         public void Print()
017         {
018             Console.WriteLine("职员信息: 姓名:{0}, 年龄: {1}, ID:{2}", Name, Age, ID);
019         }
020     }
021     class Company:IEnumerable
022     {
023         private Person[] clerk;
024         public Company()
025         {
026             clerk = new Person[5];
027             clerk[0] = new Person("令狐冲", 23, "0001");
028             clerk[1] = new Person("任盈盈", 20, "0002");
029             clerk[2] = new Person("向问天", 30, "0003");
030             clerk[3] = new Person("任我行", 50, "0004");
031             clerk[4] = new Person("风清扬", 60, "0005");
032         }
033         public IEnumerator GetEnumerator()
034         {
035             foreach (Person p in clerk)
036             {
```



```

037         yield return p;
038     }
039 }
040 }
041 class Program
042 {
043     static void Main(string[] args)
044     {
045         Company newCompany = new Company();
046         foreach (Person p in newCompany)
047         {
048             p.Print();
049         }
050         Console.ReadKey();
051     }
052 }
053 }

```

修改后的代码实现了自定义类 Company 为一个可枚举类。构建一个可枚举类，需要实现 .Net 类库中的 IEnumerable 接口。如果要简化代码的输入，需要在第 2 行引用 System.Collections 命名空间。在这个接口中有一个方法成员，它返回一个 IEnumerator 接口类型的变量。IEnumerator 接口是一个实现迭代器的接口。它里面有实现迭代的方法。不过，现在不必那么麻烦的再去实现 IEnumerator 接口中的迭代器方法。可以在实现 IEnumerable 接口中的 GetEnumerator 方法的时候，使用 yield return 语句。

在第 33 行实现 GetEnumerator 方法的时候，对 Person 类型的数组成员 clerk 使用了 foreach 循环。然后在每次迭代中，使用 yield return 语句返回每个数组成员。

这样一来，在 Main 方法中就可以先定义一个自定义类的实例，然后在第 46 行，对这个实例使用 foreach 循环迭代输出其成员数组中的内容。代码的执行结果如下：

```

职员信息: 姓名:令狐冲,   年龄: 23,   ID:0001
职员信息: 姓名:任盈盈,   年龄: 20,   ID:0002
职员信息: 姓名:向问天,   年龄: 30,   ID:0003
职员信息: 姓名:任我行,   年龄: 50,   ID:0004
职员信息: 姓名:风清扬,   年龄: 60,   ID:0005

```

在使用 yield return 的时候，也可以不使用 foreach 语句。下面修改代码：

```

001 using System;
002 using System.Collections;
003 namespace AdFeatures1
004 {
005     class Person
006     {
007         public string Name;
008         public int Age;
009         public string ID;
010         public Person(string name, int age, string id)
011         {
012             this.Name = name;

```

```
013         this.Age = age;
014         this.ID = id;
015     }
016     public void Print()
017     {
018         Console.WriteLine("职员信息: 姓名:{0}, 年龄: {1}, ID:{2}", Name, Age, ID);
019     }
020 }
021 class Company:IEnumerable
022 {
023     private Person[] clerk;
024     public Company()
025     {
026         clerk = new Person[5];
027         clerk[0] = new Person("令狐冲", 23, "0001");
028         clerk[1] = new Person("任盈盈", 20, "0002");
029         clerk[2] = new Person("向问天", 30, "0003");
030         clerk[3] = new Person("任我行", 50, "0004");
031         clerk[4] = new Person("风清扬", 60, "0005");
032     }
033     public IEnumerator GetEnumerator()
034     {
035         yield return clerk[0];
036         yield return clerk[1];
037         yield return clerk[2];
038         yield return clerk[3];
039         yield return clerk[4];
040         for (int i = 0; i < clerk.Length; i++)
041         {
042             yield return clerk[i];
043             if (i == 3)
044             {
045                 yield break;
046             }
047         }
048     }
049 }
050 class Program
051 {
052     static void Main(string[] args)
053     {
054         Company newCompany = new Company();
055         foreach (Person p in newCompany)
056         {
```

```

057         p.Print();
058     }
059     Console.ReadKey();
060 }
061 }
062 }

```

第 33 行的 GetEnumerator 方法中首先使用了多个 yield return 语句逐个输出了数组中的内容。接下来使用 for 循环又一次输出了数组中的成员。当局部变量 i 为 3 的时候，使用 yield break 语句打断了循环。因此，最后一个成员将不会迭代出来。在 Main 方法中使用 foreach 语句迭代的时候，迭代的内容完全受 GetEnumerator 方法中 yield return 语句返回的内容数量的制约，它返回多少，foreach 语句就迭代出多少，不管这些内容是否重复。代码的执行结果如下：

```

职员信息: 姓名:令狐冲,   年龄: 23,   ID:0001
职员信息: 姓名:任盈盈,   年龄: 20,   ID:0002
职员信息: 姓名:向问天,   年龄: 30,   ID:0003
职员信息: 姓名:任我行,   年龄: 50,   ID:0004
职员信息: 姓名:风清扬,   年龄: 60,   ID:0005
职员信息: 姓名:令狐冲,   年龄: 23,   ID:0001
职员信息: 姓名:任盈盈,   年龄: 20,   ID:0002
职员信息: 姓名:向问天,   年龄: 30,   ID:0003
职员信息: 姓名:任我行,   年龄: 50,   ID:0004

```

21.5.1 自定义迭代器

除了使用传统的 foreach 语句的方式迭代数据外，还可以自定义一个方法来进行自定义类中的成员的迭代。实例代码如下：

```

001 using System;
002 using System.Collections;
003 namespace AdFeatures1
004 {
005     class Person
006     {
007         public string Name;
008         public int Age;
009         public string ID;
010         public Person(string name, int age, string id)
011         {
012             this.Name = name;
013             this.Age = age;
014             this.ID = id;
015         }
016         public void Print()
017         {
018             Console.WriteLine("职员信息: 姓名:{0},   年龄: {1},   ID:{2}", Name, Age, ID);
019         }
020     }
021     class Company:IEnumerable

```

```
022     {
023         private Person[] clerk;
024         public Company()
025         {
026             clerk = new Person[5];
027             clerk[0] = new Person("令狐冲", 23, "0001");
028             clerk[1] = new Person("任盈盈", 20, "0002");
029             clerk[2] = new Person("向问天", 30, "0003");
030             clerk[3] = new Person("任我行", 50, "0004");
031             clerk[4] = new Person("风清扬", 60, "0005");
032         }
033         public IEnumerator GetEnumerator()
034         {
035             yield return clerk[0];
036             yield return clerk[1];
037             yield return clerk[2];
038             yield return clerk[3];
039             yield return clerk[4];
040             for (int i = 0; i < clerk.Length; i++)
041             {
042                 yield return clerk[i];
043                 if (i == 3)
044                 {
045                     yield break;
046                 }
047             }
048         }
049         /// <summary>
050         /// 按正序和逆序迭代
051         /// </summary>
052         /// <param name="b">如果 b 为 true, 正序迭代; 否则, 逆序迭代</param>
053         /// <returns></returns>
054         public IEnumerable SequenceGet(bool b)
055         {
056             if (b == true)
057             {
058                 for (int i = 0; i < clerk.Length; i++)
059                 {
060                     yield return clerk[i];
061                 }
062             }
063             else
064             {
065                 for (int i = (clerk.Length-1); i >=0;i--)
```

```

066         {
067             yield return clerk[i];
068         }
069     }
070 }
071
072 class Program
073 {
074     static void Main(string[] args)
075     {
076         Company newCompany = new Company();
077         foreach (Person p in newCompany)
078         {
079             p.Print();
080         }
081         Console.WriteLine("下面使用自定义迭代器正序输出");
082         foreach (Person p in newCompany.SequenceGet(true))
083         {
084             p.Print();
085         }
086         Console.WriteLine("下面使用自定义迭代器逆序输出");
087         foreach (Person p in newCompany.SequenceGet(false))
088         {
089             p.Print();
090         }
091         Console.ReadKey();
092     }
093 }
094 }

```

构建自定义迭代器的时候，其实就是定义一个方法，这个方法返回一个 `IEnumerable` 接口变量。就像第 54 行演示的那样。第 54 行定义了一个 `SequenceGet` 方法，它含有一个 `bool` 类型的参数。如果传入 `true`，则按正序迭代；传入 `false`，则按逆序迭代。在 `Main` 方法中使用时，`foreach` 语句的查询目标不再是创建的 `Company` 实例，而是 `Company` 实例调用的 `SequenceGet` 方法。在调用这个方法的时候，为其传入 `bool` 值的实参。并且，自定义迭代器并不依赖 `IEnumerable` 接口，它只要使用 `yield return` 语句返回这个类型的返回值即可。代码的执行结果如下：

```

职员信息: 姓名:令狐冲,   年龄: 23,   ID:0001
职员信息: 姓名:任盈盈,   年龄: 20,   ID:0002
职员信息: 姓名:向问天,   年龄: 30,   ID:0003
职员信息: 姓名:任我行,   年龄: 50,   ID:0004
职员信息: 姓名:风清扬,   年龄: 60,   ID:0005
职员信息: 姓名:令狐冲,   年龄: 23,   ID:0001
职员信息: 姓名:任盈盈,   年龄: 20,   ID:0002
职员信息: 姓名:向问天,   年龄: 30,   ID:0003
职员信息: 姓名:任我行,   年龄: 50,   ID:0004

```

下面使用自定义迭代器正序输出

职员信息: 姓名:令狐冲, 年龄: 23, ID:0001
 职员信息: 姓名:任盈盈, 年龄: 20, ID:0002
 职员信息: 姓名:向问天, 年龄: 30, ID:0003
 职员信息: 姓名:任我行, 年龄: 50, ID:0004
 职员信息: 姓名:风清扬, 年龄: 60, ID:0005

下面使用自定义迭代器逆序输出

职员信息: 姓名:风清扬, 年龄: 60, ID:0005
 职员信息: 姓名:任我行, 年龄: 50, ID:0004
 职员信息: 姓名:向问天, 年龄: 30, ID:0003
 职员信息: 姓名:任盈盈, 年龄: 20, ID:0002
 职员信息: 姓名:令狐冲, 年龄: 23, ID:0001

21.5.2 泛型 IEnumerable 接口

对应于非泛型的 IEnumerable 接口, 还有一个泛型的 IEnumerable<T> 接口。它位于 System.Collections.Generic 命名空间中。当实现泛型的 IEnumerable<T>接口的时候, 需要实现两个方法。下面是代码实例:

```
001 using System;
002 using System.Collections;
003 using System.Collections.Generic;
004 namespace AdFeatures1
005 {
006     class Person
007     {
008         public string Name;
009         public int Age;
010         public string ID;
011         public Person(string name, int age, string id)
012         {
013             this.Name = name;
014             this.Age = age;
015             this.ID = id;
016         }
017         public void Print()
018         {
019             Console.WriteLine("职员信息: 姓名:{0}, 年龄: {1}, ID: {2}", Name, Age, ID);
020         }
021     }
022     class Teacher
023     {
024         public string Name
025         {
026             get;
027             set;
028         }
029     }
030 }
```

```
029         public string Subject
030     {
031         get;
032         set;
033     }
034     public Teacher(string name, string subject)
035     {
036         Name = name;
037         Subject = subject;
038     }
039     public void Print()
040     {
041         Console.WriteLine("教师信息: 姓名:{0}, 所教科目: {1}", Name, Subject);
042     }
043 }
044 class Company<T> : IEnumerable<T>
045 {
046     private T[] clerk;
047     public Company(T[] t)
048     {
049         clerk = t;
050     }
051     public IEnumerator<T> GetEnumerator()
052     {
053         foreach (T t in clerk)
054         {
055             yield return t;
056         }
057     }
058     IEnumerator IEnumerable.GetEnumerator()
059     {
060         foreach (T t in clerk)
061         {
062             yield return t;
063         }
064     }
065 }
066 class Program
067 {
068     static void Main(string[] args)
069     {
070         Person[] p = new Person[5];
071         p[0] = new Person("令狐冲", 26, "0001");
072         p[1] = new Person("任盈盈", 23, "0002");
```

```

073         p[2] = new Person("向问天", 30, "0003");
074         p[3] = new Person("任我行", 50, "0004");
075         p[4] = new Person("风清扬", 60, "0005");
076         Company<Person> newCompany = new Company<Person>(p);
077         foreach (Person p1 in newCompany)
078         {
079             p1.Print();
080         }
081         Teacher[] t = new Teacher[5];
082         t[0] = new Teacher("张老师", "语文");
083         t[1] = new Teacher("李老师", "数学");
084         t[2] = new Teacher("赵老师", "英语");
085         t[3] = new Teacher("王老师", "生物");
086         t[4] = new Teacher("刘老师", "化学");
087         Company<Teacher> newCompany1 = new Company<Teacher>(t);
088         foreach (Teacher t1 in newCompany1)
089         {
090             t1.Print();
091         }
092         Console.ReadKey();
093     }
094 }
095 }

```

如果实现 `IEnumerable<T>` 接口，为简化代码的输入，需要导入 `System.Collections.Generic` 名称空间。本段代码定义了两个类 `Person` 和 `Teacher` 作为泛型类的类型实参。第 44 行定义了一个泛型类。它带有一个类型形参 `T`，并且实现了 `IEnumerable<T>` 接口。实现这个接口就要实现它的方法声明。它包含两个同名的 `GetEnumerator()` 方法，其中一个是继承自基接口 `IEnumerable`。因此，实现它们的时候其中有一个就需要显式实现。在开发环境中，自动给出的实现是，返回 `IEnumerator<T>` 的 `GetEnumerator()` 方法正常实现；返回 `IEnumerator` 的显式实现。实现的方法和前面介绍过的没有什么不同。在 `Main` 方法中的第 76 行和第 87 行各定义了一个类型实参是 `Person` 和 `Teacher` 的构造类型实例。然后将数组传递进去。使用方法仍旧是使用 `foreach` 循环。代码的执行结果如下：

```

职员信息: 姓名:令狐冲,   年龄: 26,   ID:0001
职员信息: 姓名:任盈盈,   年龄: 23,   ID:0002
职员信息: 姓名:向问天,   年龄: 30,   ID:0003
职员信息: 姓名:任我行,   年龄: 50,   ID:0004
职员信息: 姓名:风清扬,   年龄: 60,   ID:0005
教师信息: 姓名:张老师,   所教科目: 语文
教师信息: 姓名:李老师,   所教科目: 数学
教师信息: 姓名:赵老师,   所教科目: 英语
教师信息: 姓名:王老师,   所教科目: 生物
教师信息: 姓名:刘老师,   所教科目: 化学

```

21.6 数组的协变

数组的协变就是允许将派生程度更大的数组隐式转换为派生程度更小的类型的数组。下面是代码实例：

```

001     using System;

```



```
002 using System.Collections;
003 using System.Collections.Generic;
004 namespace AdFeatures1
005 {
006     class Person
007     {
008         public string Name;
009         public int Age;
010         public string ID;
011         public Person(string name, int age, string id)
012         {
013             this.Name = name;
014             this.Age = age;
015             this.ID = id;
016         }
017         public virtual void Print()
018         {
019             Console.WriteLine("职员信息: 姓名:{0}, 年龄: {1}, ID: {2}", Name, Age, ID);
020         }
021     }
022     class Executives:Person
023     {
024         public void Work()
025         {
026             Console.WriteLine("我管理产品研发");
027         }
028         public Executives(string name, int age, string id)
029             : base(name, age, id)
030         { }
031         public override void Print()
032         {
033             base.Print();
034             Work();
035         }
036     }
037     class Program
038     {
039         static void Main(string[] args)
040         {
041             Executives[] myExe = new Executives[3];
042             myExe[0] = new Executives("孙经理", 30, "0001");
043             myExe[1] = new Executives("王经理", 32, "0002");
044             myExe[2] = new Executives("李经理", 33, "0003");
045             Person[] myPerson = myExe;
```

```

046         foreach (Person p in myPerson)
047         {
048             p.Print();
049         }
050         Console.ReadKey();
051     }
052 }
053 }

```

这段代码演示了数组的协变，为了实现多态的效果，定义了继承关系的两个类。类 Executives 派生自类 Person。并且对 Print 方法作了重写。代码的第 41 行定义了一个派生类的数组并进行初始化。第 45 行定义了一个基类的数组变量，然后将派生类的变量赋值给它。这个过程就是发生的将派生程度更大的数组隐式转换为派生程度更小的数组。当数组的协变发生时，派生程度更小的数组，在本段代码中就是基类的数组，它的秩是必须与派生类数组相同的。也就是说，发生数组的协变时，两个进行转换的数组的维数是一样的。接下来，就可以使用基类数组的成员调用基类的 Print 方法，因为基类数组的成员实际是指向派生类成员的，所以在这个过程中发生了多态的调用。代码的执行结果如下：

```

职员信息：姓名:孙经理，  年龄：30，  ID:0001
我管理产品研发
职员信息：姓名:王经理，  年龄：32，  ID:0002
我管理产品研发
职员信息：姓名:李经理，  年龄：33，  ID:0003
我管理产品研发

```

在数组的协变过程中，需要注意的一点是，数组的协变不是类型安全的，它有可能会出错。实例代码如下：

```

001 using System;
002 namespace AdFeatures1
003 {
004     class Program
005     {
006         static void Main(string[] args)
007         {
008             object[] myArray1 = new string[2];
009             myArray1[0] = "Hello";
010             myArray1[1] = "World";
011             foreach (object o in myArray1)
012             {
013                 Console.WriteLine(o.ToString());
014             }
015             object[] myArray2 = new string[5];
016             try
017             {
018                 for (int i = 0; i < 5; i++)
019                 {
020                     myArray2[i] = i;
021                 }

```

```

022         }
023         catch (Exception e)
024         {
025             Console.WriteLine(e.Message);
026         }
027         Console.ReadKey();
028     }
029 }
030 }

```

这段代码中，第 8 行代码定义了一个 object 类型的数组变量，然后使用数组协变将一个 string 类型的数组隐式转换给它。第 9 行和第 10 行对数组进行赋值。赋值的类型和协变前的类型相同。这是允许的。第 11 行的打印也会很正常。第 15 行又定义了一个数组协变，依旧是将 string 类型的数组进行协变。但是，在第 18 行的 for 语句中，对数组赋值却是 int 类型。这个赋值过程现在可以看做是装箱的转换。但是，在实际执行过程中，编译器要进行运行时检查。因为是将 string 类型的数组协变为 object 类型的数组，所以为其赋值为 int 类型的值会引发异常。所以，在数组的协变操作中要注意类型安全性问题。代码的执行结果如下：

```
Hello
```

```
World
```

尝试访问类型与数组不兼容的元素。

最后需要注意的是，数组的协变只存在于引用类型之间，对于值类型的数组不存在数组的协变。例如，为了修改错误，将代码中的第 15 行修改如下：

```
object[] myArray2 = new int[5];
```

这时编译器会报告错误，错误信息如下：

```

错误 1    无法将类型“int[]”隐式转换为“object[]” H:\书稿 new\书稿源码\第二十一章
\ExtensionMethod1\Program.cs    17    33    ExtensionMethod1

```

21.7 委托的协变和逆变

委托的协变是指，可以给委托的调用列表添加这样的方法，方法的返回值与委托指定的返回值相比，派生程度更大，这就是协变。协变只能支持委托的返回类型，不支持委托的参数；如果给委托的调用列表添加这样的方法，方法的参数与委托的参数类型相比，派生程度更小，这就是逆变。下面先来看一个委托协变的代码实例：

```

001     using System;
002     namespace AdFeatures1
003     {
004         class Father
005         {
006             public int FAge
007             {
008                 get;
009                 set;
010             }
011             public Father(int age)
012             {
013                 FAge = age;
014             }

```

```
015         public virtual void Print()
016     {
017         Console.WriteLine("父亲的年龄是: {0} 岁", FAge);
018     }
019 }
020 class Son : Father
021 {
022     public int SAge
023     {
024         get;
025         set;
026     }
027     public Son(int age1, int age2)
028         : base(age1)
029     {
030         this.SAge = age2;
031     }
032     public override void Print()
033     {
034         base.Print();
035         Console.WriteLine("儿子的年龄是: {0} 岁", SAge);
036     }
037 }
038 delegate Father D1(Son s);
039 class Program
040 {
041     static Son Invoke(Son s)
042     {
043         return s;
044     }
045     static void Main(string[] args)
046     {
047         D1 myD1 = Invoke;
048         Father f = myD1(new Son(50, 20));
049         f.Print();
050         Console.ReadKey();
051     }
052 }
053 }
```

为了演示委托的协变和逆变, 代码先定义了一个 Father 类, 然后定义了它的派生类 Son 类。为了使协变的作用表现的很清楚, 对基类的 Print 方法又进行了重写。在代码的第 38 行, 定义了一个委托, 它的参数类型是 Son 类型, 返回类型是 Father 类型。在第 41 行, 定义了一个静态方法, 它的参数是 Son 类型, 返回类型是 Son 类型。方法传递进去一个 Son 类型的实例, 然后将它的引用返回。在第 47 行, 将这个方法添加给了委托 D1。因为委托返回的是 Father 类型, 但是方法返回的是 Son 类型, 所以方法的返回类型的

派生程度大于委托的返回类型的派生程度。这就是委托的协变。第 48 行定义了一个 Father 类的变量用来接收委托的返回值。然后调用 Print 方法。因为实际返回的是 Son 类型的实例，所以这就形成了基类的引用指向了子类的实例。所以就形成了多态的调用。代码的执行结果如下：

父亲的年龄是：50 岁

儿子的年龄是：20 岁

现在修改代码，试一试委托的参数可不可以发生协变。代码修改如下：

```
001    using System;
002    namespace AdFeatures1
003    {
004        class Father
005        {
006            public int FAge
007            {
008                get;
009                set;
010            }
011            public Father(int age)
012            {
013                FAge = age;
014            }
015            public virtual void Print()
016            {
017                Console.WriteLine("父亲的年龄是：{0}岁", FAge);
018            }
019        }
020        class Son : Father
021        {
022            public int SAge
023            {
024                get;
025                set;
026            }
027            public Son(int age1, int age2)
028                : base(age1)
029            {
030                this.SAge = age2;
031            }
032            public override void Print()
033            {
034                base.Print();
035                Console.WriteLine("儿子的年龄是：{0}岁", SAge);
036            }
037        }
038        delegate Father D1(Father s);
```

```

039     class Program
040     {
041         static Father Invoke(Son s)
042         {
043             Father f = s;
044             return f;
045         }
046         static void Main(string[] args)
047         {
048             D1 myD1 = Invoke;
049             Father f = myD1(new Son(50, 20));
050             f.Print();
051             Console.ReadKey();
052         }
053     }
054 }

```

在这段代码中，第 38 行的委托类型定义修改为，返回类型和参数都是 Father 类型。第 41 行的代码定义了一个静态方法，它的参数是 Son 类型，返回类型是 Father 类型。在方法体中，直接定义一个 Father 类型的变量指向了子类。这当然是允许的。第 48 行将这个方法的添加进委托 myD1 的调用列表。因为委托类型的参数是 Father 类型，而实际传递给它 Son 类型的参数，这个操作的意图是发生委托参数的协变。第 49 行调用委托，为其传入一个 Son 类型的实例。这时编译器会报告错误，错误信息如下：

```

错误 1    “Invoke”的重载均与委托“AdFeatures1.D1”不匹配    H:\书稿 new\书稿源码\第二十一章
\ExtensionMethod1\Program.cs    48    23    ExtensionMethod1

```

这说明，委托的参数是不允许发生协变的。下面再来看一下委托的逆变。实例代码如下：

```

001     using System;
002     namespace AdFeatures1
003     {
004         class Father
005         {
006             public int FAge
007             {
008                 get;
009                 set;
010             }
011             public Father(int age)
012             {
013                 FAge = age;
014             }
015             public virtual void Print()
016             {
017                 Console.WriteLine("父亲的年龄是：{0}岁", FAge);
018             }
019         }
020         class Son : Father

```

```

021    {
022        public int SAge
023    {
024        get;
025        set;
026    }
027    public Son(int age1, int age2)
028        : base(age1)
029    {
030        this.SAge = age2;
031    }
032    public override void Print()
033    {
034        base.Print();
035        Console.WriteLine("儿子的年龄是: {0}岁", SAge);
036    }
037    }
038    delegate Father D1(Son s);
039    class Program
040    {
041        static Father Invoke(Father f)
042        {
043            return f;
044        }
045        static void Main(string[] args)
046        {
047            D1 myD1 = Invoke;
048            Father f = myD1(new Father(50));
049            f.Print();
050            Console.ReadKey();
051        }
052    }
053 }

```

这段代码的第 38 行的委托类型的参数定义为 Son 类型，而返回值为 Father 类型。第 41 行定义了一个静态方法，它的参数是 Father 类型，返回值是 Father 类型。第 47 行将它添加给了委托 myD1。因为方法的参数是 Father 类型，它比委托类型定义中的 Son 类型参数派生程度更小。所以这就是委托的逆变。在第 48 行的委托调用中，实际传入的也是一个 Father 类型的实例。但是，编译器会报告错误，错误信息如下：

错误 1 委托“AdFeatures1.D1”有一些无效参数 H:\书稿 new\书稿源码\第二十一章\ExtensionMethod1\Program.cs 48 24 ExtensionMethod1

错误 2 参数 1: 无法从“AdFeatures1.Father”转换为“AdFeatures1.Son” H:\书稿 new\书稿源码\第二十一章\ExtensionMethod1\Program.cs 48 29 ExtensionMethod1

出现这个错误的原因是，委托的协变与逆变都是指的是方法的参数和返回值与委托类型的关系，而不是实际传入方法的参数。在调用委托的时候，实际传入的参数必须符合隐式转换和显式转换的规则。也就是说，决不允许子类的引用指向父类的实例。而这个例子中发生的就是实际传入的是父类的实例，而委托的参数

是子类的引用。所以编译器会报错。修改代码如下：

```
Father f = myD1(new Son(50, 30));
```

这段代码为方法传入的是子类的实例，而方法的参数定义的是 Father 类型，这就形成了一个隐式转换。而委托调用的时候，因为委托的参数要求是 Son 类型，实际传过来的参数的运行时类型是也是 Son 类型的。所以调用没有问题。而委托的返回类型是 Father 类型，当用一个 Father 类型的引用接收委托的返回值的时候，又形成了父类的引用指向子类的实例。当调用 Print 方法的时候，又形成了多态的调用。代码的执行结果如下：

父亲的年龄是：50 岁

儿子的年龄是：30 岁

21.7.1 泛型委托的协变与逆变

接下来再来介绍泛型委托的协变和逆变。下面先来看一下泛型委托的协变，代码如下：

```
001 using System;
002 namespace AdFeatures1
003 {
004     class Father
005     {
006         public int FAge
007         {
008             get;
009             set;
010         }
011         public Father(int age)
012         {
013             FAge = age;
014         }
015         public virtual void Print()
016         {
017             Console.WriteLine("父亲的年龄是：{0}岁", FAge);
018         }
019     }
020     class Son : Father
021     {
022         public int SAge
023         {
024             get;
025             set;
026         }
027         public Son(int age1, int age2)
028             : base(age1)
029         {
030             this.SAge = age2;
031         }
032         public override void Print()
033         {
```



```

034         base.Print();
035         Console.WriteLine("儿子的年龄是: {0} 岁", SAge);
036     }
037 }
038 delegate R D1<T,R>(T t);
039 class Program
040 {
041     static Son Invoke(Son s)
042     {
043         return s;
044     }
045     static void Main(string[] args)
046     {
047         D1<Son,Father> myD1 = Invoke;
048         Father f = myD1(new Son(50,30));
049         f.Print();
050         Console.ReadKey();
051     }
052 }
053 }

```

这段代码的第 38 行定义了一个泛型委托，它带有两个类型形参。类型形参 T 是委托的参数，类型形参 R 是委托的返回类型。第 41 行代码定义了一个静态方法，它的返回类型是 Son 类型，参数类型也是 Son 类型。第 47 行使用类型实参构建了泛型委托的构造类型。这时委托的返回类型是 Father 类型，参数是 Son 类型。在将 Invoke 方法添加进委托的时候，因为方法的返回类型是 Son 类型，所以这时发生了委托的协变。因为委托的返回类型是 Father 类型，而方法将传入的 Son 类型实例进行了返回，所以当 Father 类型的变量 f 调用 Print 方法的时候发生了多态的调用。代码的执行结果如下：

父亲的年龄是：50 岁

儿子的年龄是：30 岁

下面再来看泛型委托的逆变，实例代码如下：

```

001 using System;
002 namespace AdFeatures1
003 {
004     class Father
005     {
006         public int FAge
007         {
008             get;
009             set;
010         }
011         public Father(int age)
012         {
013             FAge = age;
014         }
015         public virtual void Print()

```

```
016         {
017             Console.WriteLine("父亲的年龄是: {0}岁", FAge);
018         }
019     }
020     class Son : Father
021     {
022         public int SAge
023         {
024             get;
025             set;
026         }
027         public Son(int age1, int age2)
028             : base(age1)
029         {
030             this.SAge = age2;
031         }
032         public override void Print()
033         {
034             base.Print();
035             Console.WriteLine("儿子的年龄是: {0}岁", SAge);
036         }
037     }
038     delegate R D1<T,R>(T t);
039     class Program
040     {
041         static Son Invoke(Father f)
042         {
043             Son s = new Son(f.FAge, 30);
044             return s;
045         }
046         static void Main(string[] args)
047         {
048             D1<Son, Father> myD1 = Invoke;
049             Father f = myD1(new Son(50, 30));
050             f.Print();
051             Console.ReadKey();
052         }
053     }
054 }
```

这段代码不只发生了泛型委托的逆变,同时也发生了协变。第38行的委托类型的定义没有发生变化。第41行定义的方法含有一个Father类型的参数,返回类型是Son类型。在它的内部创建了一个Son类型的实例,为Son类型的构造方法传入的参数使用了参数f的FAge属性的值。然后将这个创建的Son类型的实例返回。

第48行定义泛型委托的构造类型的时候,委托的构造类型的参数是Son类型,返回值是Father类型。

在将 Invoke 添加进入委托的调用列表的时候，因为方法的返回值是 Son，而委托的返回类型是 Father，所以发生了委托的协变；而方法的参数是 Father 类型，委托的参数是 Son 类型，这时发生了委托的逆变。所以这个例子同时发生了委托的协变和逆变。在调用委托的时候，必须为其传入 Son 类型的实例才能不出错。当用委托的返回值调用 Print 方法的时候，依旧发生了多态调用。代码的执行结果如下：

父亲的年龄是：50 岁

儿子的年龄是：30 岁

21.7.2 变体泛型委托

在介绍变体泛型委托之前，先来看一个代码实例：

```
001 using System;
002 namespace AdFeatures1
003 {
004     class Father
005     {
006         public int FAge
007         {
008             get;
009             set;
010         }
011         public Father(int age)
012         {
013             FAge = age;
014         }
015         public virtual void Print()
016         {
017             Console.WriteLine("父亲的年龄是：{0}岁", FAge);
018         }
019     }
020     class Son : Father
021     {
022         public int SAge
023         {
024             get;
025             set;
026         }
027         public Son(int age1, int age2)
028             : base(age1)
029         {
030             this.SAge = age2;
031         }
032         public override void Print()
033         {
034             base.Print();
035             Console.WriteLine("儿子的年龄是：{0}岁", SAge);
036         }
037     }
038 }
```

```

037     }
038     delegate R D1<R>(Son s);
039     class Program
040     {
041         static Son Invoke(Son s)
042         {
043             return s;
044         }
045         static void Main(string[] args)
046         {
047             D1<Son> myD1 = Invoke;
048             D1<Father> myD2 = myD1;
049             Father f = myD2(new Son(50, 30));
050             f.Print();
051             Console.ReadKey();
052         }
053     }
054 }

```

在代码的第 38 行定义了一个泛型委托，它的类型形参是它的返回值。它的参数是 Son 类型。在第 47 行，为委托类型的类型形参指定了 Son 类型的类型实参，并且将方法 Invoke 添加进它的调用列表。方法 Invoke 是完全匹配委托的构造类型的。接下来在第 48 行定义了一个返回值为 Father 类型的泛型委托的构造类型，然后将委托 myD1 赋值给它。因为 myD1 的返回值类型和 myD2 的返回值类型之间可以发生协变，所以这样的转换应该是成功的。但是编译器会报告错误，错误信息如下：

```

错误 1 无法将类型“AdFeatures1.D1<AdFeatures1.Son>”隐式转换为
“AdFeatures1.D1<AdFeatures1.Father>”
H:\书稿 new\书稿源码\第二十一章
\ExtensionMethod1\Program.cs 48 31 ExtensionMethod1

```

从错误信息中可以知道，这个隐式转换无法完成。下面修改代码，修改后的代码如下：

```

delegate R D1<out R>(Son s);

```

对代码的修改只是在第 38 行的委托类型的定义中，为类型形参加上了 out 修饰符。这个修饰符注明这个类型形参可以发生协变。这样，这个委托就叫做变体泛型委托。这时代码可以正常执行。下面再次修改代码：

```

001     using System;
002     namespace AdFeatures1
003     {
004         class Father
005         {
006             public int FAge
007             {
008                 get;
009                 set;
010             }
011             public Father(int age)
012             {
013                 FAge = age;
014             }

```

```
015         public virtual void Print()
016     {
017         Console.WriteLine("父亲的年龄是: {0} 岁", FAge);
018     }
019 }
020 class Son : Father
021 {
022     public int SAge
023     {
024         get;
025         set;
026     }
027     public Son(int age1, int age2)
028         : base(age1)
029     {
030         this.SAge = age2;
031     }
032     public override void Print()
033     {
034         base.Print();
035         Console.WriteLine("儿子的年龄是: {0} 岁", SAge);
036     }
037 }
038 delegate Father D1<in R>(R r);
039 class Program
040 {
041     static Son Invoke(Father f)
042     {
043         Son mySon = new Son(f.FAge, 20);
044         return mySon;
045     }
046     static void Main(string[] args)
047     {
048         D1<Father> myD1 = Invoke;
049         D1<Son> myD2 = myD1;
050         Father f = myD2(new Son(50, 30));
051         f.Print();
052         Console.ReadKey();
053     }
054 }
055 }
```

在代码的第 38 行定义了一个委托类型，它带有一个类型形参，这个类型形参作委托类型的参数。并且它用 `in` 关键字进行了修饰，表明这个类型形参支持逆变。第 41 行的方法定义了一个 `Father` 类型的参数，返回值为 `Son` 类型。在方法体中使用 `Father` 类型的成员数据创建了 `Son` 类型的实例。在第 48 行将 `Invoke`

方法添加给了委托 myD1。这时委托 myD1 的参数是 Father 类型，返回值为 Fahter 类型。它发生了返回值的协变。然后又定义了类型参数为 Son 类型的委托变量，然后将 myD1 赋值给它。虽然 myD1 的参数是 Father 类型，但是因为类型形参标注了可以逆变，所以这个转换赋值是成功的。当调用 myD2 的时候需要传入 Son 类型的实例才符合要求，因为 myD2 要求参数为 Son 类型。myD2 的返回类型是 Father 类型，用返回值调用 Print 方法时又发生了多态的调用。代码的执行结果如下：

父亲的年龄是：50 岁

儿子的年龄是：20 岁

如果将第 38 行中的 in 关键字去掉，则编译器会报告如下的错误：

错误 1 无法将类型 “AdFeatures1.D1<AdFeatures1.Father>” 隐式转换为 “AdFeatures1.D1<AdFeatures1.Son>” H:\书稿 new\书稿源码\第二十一章\ExtensionMethod1\Program.cs 49 28 ExtensionMethod1

21.8 变体泛型接口

非泛型接口与实现类之间不存在协变与逆变，看下面的代码：

```
001 using System;
002 namespace AdFeatures1
003 {
004     class Father
005     {
006         public int FAge
007         {
008             get;
009             set;
010         }
011         public Father(int age)
012         {
013             FAge = age;
014         }
015         public virtual void Print()
016         {
017             Console.WriteLine("父亲的年龄是：{0}岁", FAge);
018         }
019     }
020     class Son : Father
021     {
022         public int SAge
023         {
024             get;
025             set;
026         }
027         public Son(int age1, int age2)
028             : base(age1)
029         {
030             this.SAge = age2;
031         }
032     }
033 }
```

```

032         public override void Print()
033         {
034             base.Print();
035             Console.WriteLine("儿子的年龄是: {0} 岁", SAge);
036         }
037     }
038     interface Hobby
039     {
040         Father Sing();
041     }
042     class Test : Hobby
043     {
044         public Son Sing()
045         {
046             Son mySon = new Son(50, 30);
047             Console.WriteLine("我会唱歌");
048             return mySon;
049         }
050     }
051     class Program
052     {
053         static void Main(string[] args)
054         {
055             Test myTest = new Test();
056             Son s = myTest.Sing();
057             s.Print();
058             Console.ReadKey();
059         }
060     }
061 }

```

这段代码在定义了 Father 基类和 Son 派生类之外, 在第 38 行又定义了一个接口, 接口中的 Sing 方法返回一个 Father 类的值。在第 42 行定义了一个类 Test 实现了接口 Hobby。在类 Test 中实现接口中的 Sing 方法的时候, 返回了一个 Son 类型的引用。因为接口中的方法规定返回类型为 Father, 而方法的实现返回的类型是 Son, 它们之间能不能发生协变呢? 在编译的时候, 编译器会报告如下的错误:

```

错误 1    “ AdFeatures1.Test ” 不 实 现 接 口 成 员 “ AdFeatures1.Hobby.Sing() ” 。
“ AdFeatures1.Test.Sing() ” 无法实现 “ AdFeatures1.Hobby.Sing() ”, 因为它没有匹配的返回类型
“ AdFeatures1.Father”。 H:\书稿 new\书稿源码\第二十一章\ExtensionMethod1\Program.cs 42 11
ExtensionMethod1

```

可以看到, 实现接口中的方法的时候, 实现方法的返回值必须和接口中定义的方法的返回值相同。所以, 在非泛型接口与实现类之间是没有逆变与协变这种概念的。

逆变与协变只发生在泛型接口中。协变允许实现方法的返回值具有比泛型接口中的方法的返回值派生程度更大的类型。下面看代码实例:

```

001     using System;
002     namespace AdFeatures1

```

```
003  {
004      class Father
005      {
006          public int FAge
007          {
008              get;
009              set;
010          }
011          public Father()
012          {
013              FAge = 50;
014          }
015          public virtual void Print()
016          {
017              Console.WriteLine("父亲的年龄是: {0}岁", FAge);
018          }
019      }
020      class Son : Father
021      {
022          public int SAge
023          {
024              get;
025              set;
026          }
027          public Son()
028          {
029              this.SAge = 30;
030          }
031          public override void Print()
032          {
033              base.Print();
034              Console.WriteLine("儿子的年龄是: {0}岁", SAge);
035          }
036      }
037      interface Hobby<out T>
038      {
039          T Sing();
040      }
041      class Test<T> : Hobby<T>
042          where T : class, new()
043      {
044          public T Sing()
045          {
046              T t = new T();
```



```

047         Console.WriteLine("我会唱歌");
048         return t;
049     }
050 }
051 class Program
052 {
053     static void Main(string[] args)
054     {
055         Test<Son> myTest = new Test<Son>();
056         Hobby<Father> myHobby = myTest;
057         myHobby.Sing().Print();
058         Console.ReadKey();
059     }
060 }
061 }

```

首先，基类与派生类都定义了无参的构造方法，第 37 行定义了泛型接口，它的类型形参使用 `out` 关键字修饰，表示接口中的方法的返回类型可以比泛型接口指定的类型实参的返回类型派生程度更大。第 41 行的泛型类实现了泛型接口。泛型类也带有一个类型形参，它的类型与泛型接口一样。第 42 行对泛型类和泛型接口中的类型形参 `T` 作了约束，使它们可以使用构造方法。第 44 行对接口中的方法进行了实现，其中用到了构造方法的调用。

第 55 行定义了一个 `Test` 类的实例，类型实参为 `Son` 类型，这时，`Sing` 方法也返回 `Son` 类型。第 56 行就是接口的协变的应用，在这一行定义了一个接口类型的变量，类型实参为 `Father` 类型。这时接口中的方法声明的返回类型也是 `Father` 类型的。接下来将 `myTest` 赋值给它的时候，可以说，实际的 `Sing` 方法的返回类型的派生程度已经高于 `myHobby` 的方法的返回类型了。第 57 行因为 `myHobby.Sing` 方法返回的是 `Son` 类型，所以调用了 `Son` 类型中的 `Print` 方法。代码执行结果如下：

```

我会唱歌
父亲的年龄是：50 岁
儿子的年龄是：30 岁

```

再来看泛型接口的逆变，“逆变”允许接口方法具有与泛型形参指定的实参类型相比，派生程度更小的实参类型。下面是代码实例：

```

001     using System;
002     namespace AdFeatures1
003     {
004         class Father
005         {
006             public int FAge
007             {
008                 get;
009                 set;
010             }
011             public Father()
012             {
013                 FAge = 50;
014             }
015         }
016     }

```

```
015         public virtual void Print()
016     {
017         Console.WriteLine("父亲的年龄是: {0}岁", FAge);
018     }
019 }
020 class Son : Father
021 {
022     public int SAge
023     {
024         get;
025         set;
026     }
027     public Son()
028     {
029         this.SAge = 30;
030     }
031     public override void Print()
032     {
033         base.Print();
034         Console.WriteLine("儿子的年龄是: {0}岁", SAge);
035     }
036 }
037 interface Hobby<in T>
038 {
039     void Sing(T t);
040 }
041 class Test<T> : Hobby<T>
042     where T : Father, new()
043 {
044     public void Sing(T t)
045     {
046         Console.WriteLine("我会唱歌");
047         t.Print();
048     }
049 }
050 class Program
051 {
052     static void Main(string[] args)
053     {
054         Hobby<Son> myHobby = new Test<Father>();
055         myHobby.Sing(new Son());
056         Console.ReadKey();
057     }
058 }
```

```
059    }
```

在这段代码中的第 37 行，泛型接口的类型形参使用了 `in` 关键字进行修饰，说明类型形参是逆变的，只能做方法的参数。类 `Test` 实现了接口 `Hobby`，为了能够调用 `Father` 类的方法，对类型形参使用了约束。在第 47 行调用的 `Print` 是 `Father` 类的方法。

因为类型形参是逆变的，所以第 54 行能够定义一个以 `Son` 为类型实参的接口引用，指向了以 `Father` 类为类型实参的类的实例。当 `myHobby` 调用 `Sing` 方法的时候，能够为其传入 `Son` 类型的实例以实现多态。代码的执行结果如下：

```
我会唱歌
```

```
父亲的年龄是：50 岁
```

```
儿子的年龄是：30 岁
```

如果将修饰类型形参的 `in` 关键字去掉，则编译器会报告如下的错误：

```
错误 1 无法将类型“AdFeatures1.Test<AdFeatures1.Father>”隐式转换为
“AdFeatures1.Hobby<AdFeatures1.Son>”。存在一个显式转换(是否缺少强制转换?) H:\书稿 new\书稿源码\第二十一章\ExtensionMethod1\Program.cs 54 34 ExtensionMethod1
```

21.9 dynamic 类型

`dynamic` 类型与 `object` 类型一样，可以引用任何实例。但是，与 `object` 不同的是，使用 `dynamic` 引用实例的时候，其解析会延迟到运行时。也就是说，纵然 `dynamic` 的引用是错误的，在编译时也不会报告任何错误。但是在运行时，会引发异常。`dynamic` 的主要用途就是允许进行动态绑定。下面看一下代码实例：

```
001    using System;
002    namespace AdFeatures3
003    {
004        class Car
005        {
006            private int speed;
007            public void SpeedAdd(int s)
008            {
009                speed += s;
010            }
011            public void Show()
012            {
013                Console.WriteLine("汽车现在的速度是：{0}公里/小时", speed);
014            }
015        }
016        class Program
017        {
018            static void Main(string[] args)
019            {
020                dynamic d1 = "我们出发吧";
021                Type t = d1.GetType();
022                Console.WriteLine(t.ToString());
023                Console.WriteLine(d1.ToString());
024                dynamic d2 = new Car();
025                d2.SpeedAdd(80);
```

```

026         d2.Show();
027         Console.WriteLine("速度太快了，现在减速吧");
028         try
029         {
030             d2.SpeedSub(30);
031         }
032         catch (Exception e)
033         {
034             Console.WriteLine(e.Message);
035         }
036         Console.ReadKey();
037     }
038 }
039 }

```

这段代码中定义了一个 Car 类，它含有一个 int 类型的字段 speed。SpeedAdd 方法可以对它赋值。Show 方法显示当前的 speed 值。在第 20 行，首先定义了一个 dynamic 类的变量并将一个字符串赋值给它。第 21 行代码取得它的类型。在调用 GetType 方法的时候，可以发现，开发环境不再有智能提示，而是提示类型在运行时解析。第 22 行打印这个动态类型的实际类型。

第 24 行定义了一个 Car 类型的实例，并将它赋值给 dynamic 类型的变量 d2。第 25 行调用了 SpeedAdd 方法将 speed 赋值为 80。第 30 行调用了 SpeedSub 方法，意图将 30 赋值给 speed。查看 Car 类可以发现，在类中没有定义这个方法，但是，编译这段代码却不会出错。对 SpeedSub 方法的调用在代码中进行了异常的捕获。在代码实际运行的时候，运行时环境将会捕捉到这一异常信息。代码的执行结果如下：

```

System.String
我们出发吧
汽车现在的速度是：80 公里/小时
速度太快了，现在减速吧
“AdFeatures3.Car”未包含“SpeedSub”的定义

```

从代码的执行结果来看，在运行时，可以正常解析出 dynamic 类型变量引用的类型。它的作用可以认为同 object 类型相同。但是它有两点与 object 类型不同：

1. 对 dynamic 类型的表达式运算可以动态绑定。
2. 类型推断在 dynamic 和 object 都是候选项的时候，会优先考虑 dynamic 类型。

它们之间的相同点是：

1. object 与 dynamic 之间存在隐式标识转换，将 dynamic 替换为 object 时相同的构造类型之间也存在隐式标识转换。
2. 与 object 之间的隐式和显式转换也适用于 dynamic。
3. 在将 dynamic 替换为 object 时相同的方法签名视为是相同的签名。

下面是代码实例：

```

001     using System;
002     namespace AdFeatures3
003     {
004         class Test<T>
005         {
006             private T t;
007             public Test(T t)

```

```

008      {
009          this.t = t;
010      }
011      public void Show()
012      {
013          Console.WriteLine("字段 t 的值是: {0}", t);
014      }
015      public void Show2(T t)
016      {
017          Console.WriteLine(t.ToString());
018      }
019  }
020  class Program
021  {
022      static void Main(string[] args)
023      {
024          Test<object> myTest = new Test<object>(1000);
025          myTest.Show();
026          object o = 2000;
027          myTest.Show2(o);
028          dynamic d = 2000;
029          myTest.Show2(d);
030          Test<dynamic> myTest1 = myTest;
031          myTest1.Show();
032          Console.ReadKey();
033      }
034  }
035  }

```

这段代码将 `Test` 类型定义为一个泛型类，它带有一个类型形参。在第 24 行为其指定的类型实参是 `object` 类型，第 25 行、第 26 行和第 27 行的代码可以正常运行。第 28 行代码中，定义了一个 `dynamic` 类型的变量，并赋值为 2000，将它可以与 `object` 一样的方法传入方法 `Show2`，它们的方法签名被认为是同样的签名。第 30 行使用 `dynamic` 类型创建了泛型类的构造类型，并定义了一个变量，这时，以 `object` 为类型实参的实例可以以隐式标识转换的方式赋值给它。第 31 行，这个新的构造类型变量调用了 `Show` 方法。代码的执行结果如下：

```

字段 t 的值是: 1000
2000
2000
字段 t 的值是: 1000

```

可以看到，`dynamic` 在运行时与 `object` 类型没有什么区别，使用 `dynamic` 的表达式被称为动态表达式。

21.9.1 隐式动态转换

`dynamic` 类型的表达式可以隐式动态转换到任何类型，这种转换是运行时的，如果在运行时发现没有发生任何转换，则会引发运行时异常。下面是代码实例：

```

001  using System;
002  namespace AdFeatures3

```

```

003  {
004      class Program
005      {
006          static void Main(string[] args)
007          {
008              dynamic d = "Hello World";
009              string s = d;
010              try
011              {
012                  int i = d;
013              }
014              catch (Exception e)
015              {
016                  Console.WriteLine(e.Message);
017              }
018              object o = "Hello World";
019              string s1 = o;//dynamic 可以隐式转换, 但 object 不行
020              Console.ReadKey();
021          }
022      }
023  }

```

在这段代码中, 第 18 行定义了一个 object 类型的变量 o, 然后将它指向了一个 string 类型的实例。在第 19 行将它隐式转换到 string 类型, 但是编译器会报错, 因为不运行有这样的隐式转换。对于 object 类型来说不行, 但是 dynamic 类型就不同了。第 9 行将一个 dynamic 类型的变量可以隐式转换到 string 类型, 因为其原始类型就是 string 类型的。但是第 12 行就会在运行时出错, 因为 d 的原始类型是 string 类型的, 不能将它在运行时转换为一个 int 类型。将第 19 行注释掉, 代码的执行结果如下:

无法将类型“string”隐式转换为“int”

21.9.2 显式动态转换

从 dynamic 类型的表达式可以以显式转换的方式转换到任何其它类型, 转换是动态绑定的。如果没有发生转换, 则会引发运行时异常。下面是代码实例:

```

001  using System;
002  namespace AdFeatures3
003  {
004      class Test
005      {
006          string s;
007          public Test(string s)
008          {
009              this.s = s;
010          }
011          public static explicit operator Test(int i)
012          {
013              string s1 = Convert.ToString(i);
014              return new Test(s1);

```

```

015     }
016     public void Print()
017     {
018         Console.WriteLine(s);
019     }
020 }
021 class Program
022 {
023     static void Main(string[] args)
024     {
025         object o = 100;
026         try
027         {
028             Test t1 = (Test)o;
029         }
030         catch (Exception e)
031         {
032             Console.WriteLine(e.Message);
033         }
034         dynamic d = 200;
035         Test t2 = (Test)d;
036         t2.Print();
037         Console.ReadKey();
038     }
039 }
040 }

```

代码的第 4 行自定义了一个类 Test, Test 类中定义了一个 string 类型的字段。第 11 行定义了一个显式转换方法, 将一个 int 类型的值转换为 Test 类型。第 25 行定义了一个 object 类型的变量, 为它赋值 100。第 28 行将它强制转换为 Test 类型。因为 object 的类型是编译时类型, 所以编译器会在运行时抛出异常, 因为 object 类型不符合显式转换方法的参数要求。第 34 行代码定义了一个 dynamic 类型, 并赋值为 200。第 35 行使用 dynamic 类型显式转换为 Test 类型则没有问题, 因为 dynamic 类型是运行时类型, 它在得到它的运行时的具体类型前始终是挂起状态。而它的运行时类型是 int 类型, 符合显式转换方法的要求, 所以转换会成功。代码的执行结果如下:

无法将类型为“System.Int32”的对象强制转换为类型“AdFeatures3.Test”。

200

21.9.3 静态绑定与动态绑定

在 C# 中, 静态绑定在编译时进行, 而 dynamic 类型是动态绑定的, 它在运行时进行。下面是代码实例:

```

001 using System;
002 namespace AdFeatures3
003 {
004     class Program
005     {
006         static void Main(string[] args)
007         {

```

```
008         object o = 100;
009         bool b = true;
010         dynamic d = "Hello World";
011         Console.WriteLine(o);
012         Console.WriteLine(b);
013         Console.WriteLine(d);
014         Console.ReadKey();
015     }
016 }
017 }
```

这段代码中，第 11 行和第 12 行是静态绑定的，其绑定在编译时进行。绑定类型分别是 `object` 类型和 `bool` 类型。第 13 行代码是动态绑定的，其绑定类型是 `dynamic` 类型，运行时类型是 `string` 类型。

在静态绑定时，构成表达式的类型视为该表达式的编译时类型；当进行动态绑定时，如果其编译时类型为 `dynamic` 类型，则表达式被视为其在运行时的实际值的类型；如果编译时类型为类型形参，则其表达式的构成类型为类型形参在运行时绑定到的类型。

21.9.4 as 和 is 运算符对 dynamic 类型的转换

`as` 运算符不是动态绑定的，但 `is` 运算符能够检查数据的运行时类型。对于使用 `dynamic` 的 `as` 表达式来说，如果类型能够转换成功，则编译器先把 `dynamic` 类型转换成 `object`，然后再由 `object` 类型转换成目标类型；如果不能够转换成功，则返回 `null`。下面是代码实例：

```
001 using System;
002 namespace AdFeatures3
003 {
004     class Person
005     {
006         public int Age
007         {
008             get;
009             set;
010         }
011         public Person(int age)
012         {
013             this.Age = age;
014         }
015         public virtual void Print()
016         {
017             Console.WriteLine("Person 的年龄是: {0}", Age);
018         }
019     }
020     class Teacher : Person
021     {
022         public Teacher(int age)
023             : base(age)
024         { }
025         public override void Print()
```



```

026      {
027          base.Print();
028      }
029  }
030  class Program
031  {
032      static void Main(string[] args)
033      {
034          dynamic d1 = "Hello World";
035          Person p1 = d1 as Person;
036          try
037          {
038              p1.Print();
039          }
040          catch (Exception e)
041          {
042              Console.WriteLine(e.Message);
043          }
044          dynamic d2 = new Teacher(30);
045          Person p2 = d2 as Person;
046          p2.Print();
047          Console.ReadKey();
048      }
049  }
050  }

```

在代码的第 34 行，定义了一个 `dynamic` 类型，并赋值为 `Hello World`。第 35 行将它使用 `as` 运算符转换成 `Person` 类型。这时，C#编译器将把表达式扩展成下面的形式来进行转换：`E is T ? (T)(object)(E) : (T)null`。首先编译器调用了 `is` 运算符来判断 `dynamic` 类型的运行时类型，如果运行时类型可以进行转换，则先将它的类型转为 `object` 类型，然后再次进行转换成目标类型；如果运行时类型不能进行转换，则返回 `null`。

当第 35 行进行 `as` 转换的时候，会先调用 `is` 运算符进行运行时的判断，因为 `d1` 的运行时类型是 `string` 类型，所以它是不能转换成 `Person` 类型的，因此 `p1` 的值将为 `null`。第 44 行定义了一个 `dynamic` 类型的变量指向了 `Teacher` 类型的实例。接下来将它用 `as` 运算符转换成 `Person` 类型，这时编译器会调用 `is` 运算符判断 `d2` 的运行时类型，因为 `d2` 的运行时类型是 `Teacher`，它可以隐式转换成 `Person` 类型，所以 `as` 运算符将它先转换成 `object` 类型，然后再次显式转换成 `Person` 类型。这就是 `as` 运算符的运行机制。这段代码的执行结果如下：

未将对象引用设置到对象的实例。

Person 的年龄是：30

21.9.5 dynamic 类型的应用

`dynamic` 类型就像一种不受规则约束的类型，拥有很大的自由。虽然使用它可以完成用普通类型不能完成的许多任务，但是随之带来的风险也很大。要考虑任何可能出错的原因，除非程序员能够确认自己把握所有可能的情况，否则应该尽量少用这种类型，因为它没有编译器的类型安全检查。类型的安全性要人为的把握。下面看一段代码：

```
001 using System;
```

```

002 namespace AdFeatures3
003 {
004     class Test<T, U>
005     {
006         private T t;
007         private U u;
008         public Test(T t, U u)
009         {
010             this.t = t;
011             this.u = u;
012         }
013         public void Add()
014         {
015             dynamic d = (dynamic)t + (dynamic)u;
016             Console.WriteLine(d);
017         }
018     }
019     class Program
020     {
021         static void Main(string[] args)
022         {
023             Test<int, float> myTest = new Test<int, float>(1000, 1020.3F);
024             myTest.Add();
025             Console.ReadKey();
026         }
027     }
028 }

```

这段代码定义了一个泛型类，泛型类带有两个类型形参。在泛型类中含有这两个类型形参的字段。第13行的方法对两个字段相加。然后打印其结果。如果不采用 dynamic 类型，这样写代码根本行不通，因为不知道类型形参 T 和 U 的具体类型就无法使用+运算符。如果采用约束的话，同时应该定义运算符重载才可以。但是使用 dynamic 类型就简单得多，因为它的具体类型要到运行时才知道，目前编译器认为这种语法完全是正确的，类型安全由程序员来掌握。因此，使用 dynamic 后，代码很简单。传入正确的类型实参也能很好的运行。代码的执行结果如下：

2020.3

再来看一下 dynamic 类型在反射中的应用，代码实例如下：

```

001 using System;
002 using System.Reflection;
003 namespace Attribute1
004 {
005     [AttributeUsage(AttributeTargets.All, AllowMultiple = true)]
006     class AppInfoAttribute : Attribute
007     {
008         public string Platform
009         {

```

```
010         get;
011         set;
012     }
013     public string Name
014     {
015         get;
016         set;
017     }
018     public string Version
019     {
020         get;
021         set;
022     }
023     public AppInfoAttribute(string name, string version, string platform)
024     {
025         Name = name;
026         Version = version;
027         Platform = platform;
028     }
029     public AppInfoAttribute(string name, string version)
030     {
031         Name = name;
032         Version = version;
033         Platform = "Windows 8";
034     }
035 }
036 public class AppVersionAttribute : Attribute
037 {
038     public string Version
039     {
040         get;
041         set;
042     }
043     public AppVersionAttribute(string version)
044     {
045         Version = version;
046     }
047 }
048 [AppInfo("秋叶无声", "3.0", "Windows XP")]
049 [AppVersion("1.0.0.0")]
050 class Person
051 {
052     public string Name
053     {
```

```
054         get;
055         set;
056     }
057     public string ID
058     {
059         get;
060         set;
061     }
062     public int Age
063     {
064         get;
065         set;
066     }
067     [return:AppInfo("秋叶无声", "2.0", "Windows 7")]
068     public void Print(string s)
069     {
070         s = "#";
071         Console.WriteLine(s + " 姓名: {0}, 年龄: {1}, 身份证号: {2}", Name, Age, ID);
072     }
073 }
074 class Program
075 {
076     static void GetTypeAttribute(Type t)
077     {
078         Console.WriteLine("现在开始通过反射获取 {0} 类的特性信息", t.ToString());
079         Attribute[] attArray = Attribute.GetCustomAttributes(t);
080         foreach (Attribute att in attArray)
081         {
082             if (att is AppInfoAttribute)
083             {
084                 AppInfoAttribute ai = (AppInfoAttribute)att;
085                 Console.WriteLine("作者: {0}, 版本: {1}, 平台: {2}", ai.Name, ai.Version,
                                ai.Platform);
086             }
087             if (att is AppVersionAttribute)
088             {
089                 AppVersionAttribute ava = (AppVersionAttribute)att;
090                 Console.WriteLine("程序集版本: {0}", ava.Version);
091             }
092         }
093     }
094     static void GetMethodReturnAttribute(Type t)
095     {
096         Console.WriteLine("现在开始通过反射获取 {0} 类的方法返回值的特性信息",
```

```

097         t.ToString());
098         MethodInfo mti = t.GetMethod("Print");
099         dynamic o = mti.ReturnTypeCustomAttributes.
           GetCustomAttributes(typeof(AppInfoAttribute), false);
100         foreach (Attribute att in o)
101         {
102             if (att is AppInfoAttribute)
103             {
104                 AppInfoAttribute ai = (AppInfoAttribute)att;
105                 Console.WriteLine("作者: {0}, 版本: {1}, 平台: {2}", ai.Name,
                                   ai.Version, ai.Platform);
106             }
107         }
108     }
109     static void Main(string[] args)
110     {
111         GetTypeAttribute(typeof(Person));
112         GetMethodReturnAttribute(typeof(Person));
113         Console.ReadKey();
114     }
115 }
116 }

```

这段代码使用反射获取定义在类型中的实例方法上的自定义特性信息。在第 99 行得到返回类型的特性数组的时候，使用了 `dynamic` 类型，这样就避免了下面这样的转换：

```
Attribute[] attArray = (Attribute[])o;
```

`foreach` 语句可以直接对 `dynamic` 类型的变量 `o` 进行迭代。而不用管它的运行时类型是什么。实际上，做这一步的时候，只有程序员知道 `o` 的运行时类型是什么，只有确认它的运行时类型才可以使用 `dynamic` 类型。代码的执行结果如下：

现在开始通过反射获取 `Attribute1.Person` 类的特性信息

作者：秋叶无声，版本：3.0，平台：Windows XP

程序集版本：1.0.0.0

现在开始通过反射获取 `Attribute1.Person` 类的方法返回值的特性信息

作者：秋叶无声，版本：2.0，平台：Windows 7

21.10 使用 `using` 语句获取资源和释放资源

使用 `using` 语句除了可以引用命名空间外，还可以使用它获取非托管资源和释放非托管资源。前面介绍过，使用类的析构方法不能准确地释放非托管资源，因为它的运行时刻由垃圾回收器来掌握。因此，在使用非托管资源的过程中，要通过非托管资源实现的 `System.IDisposable` 接口中的方法 `Dispose` 来立即释放资源。非托管资源通常已经实现了这个接口，只需要在使用托管资源之后，调用这个接口中的方法即可完成非托管资源的释放工作。调用这个方法来释放资源比较明确，否则，只能等垃圾回收器在合适的时间来释放非托管资源。

通过 `using` 语句来使用非托管资源的话，`using` 语句的工作分为获取、使用和释放三个阶段。下面是代码实例：

```

001 using System;
002 using System.IO;

```

```

003 namespace AdFeatures3
004 {
005     class Program
006     {
007         static void Main(string[] args)
008         {
009             using (TextWriter tw = File.CreateText("a.csv"))
010             {
011                 tw.WriteLine("ID    ,用户姓名    ,地址");
012                 tw.WriteLine("0001 , 令狐冲    ,华山剑派");
013                 tw.WriteLine("0002 , 任盈盈    ,黑木崖");
014                 tw.WriteLine("0003 , 东方不败    ,黑木崖");
015                 tw.WriteLine("0004 , 风清扬    ,华山后山");
016             }
017             using(TextReader tr = File.OpenText("a.csv"))
018             {
019                 string s;
020                 while((s = tr.ReadLine()) != null)
021                 {
022                     Console.WriteLine(s);
023                 }
024             }
025             Console.ReadKey();
026         }
027     }
028 }

```

在实际应用过程中，经常会有和数据库的交互以及和文件的交互这样的需求。可以采用手动实现连接的建立或文件的打开，在操作数据完毕后，再手动释放连接或关闭文件方式来编写代码。也可以采用本段代码的这种方法，使用 using 语句打开资源，然后在 using 语句后面的代码块内进行数据操作。当代码块内的语句执行完毕的时候，using 语句会自动调用资源的 Dispose 方法来释放资源。

因为要使用文件资源，所以代码的第 2 行引入了 System.IO 命名空间。在代码的第 9 行的 using 语句中，定义了一个 TextWriter 变量接收返回值，使用 File 类的 CreateText 方法在程序集所在的目录下建立一个.csv 文件。然后在 using 的语句块中，调用 TextWriter 类的 WriteLine 方法来向文件中写入数据。当语句块中的代码执行完毕后，调用 Dispose 方法来释放资源。

代码的第 17 行使用 File 类的 OpenText 方法来打开刚刚建立的文件，然后将它赋值给一个 TextReader 类的变量。第 20 行代码使用 TextReader 类的 ReadLine 方法来读取文件中的内容，每读取一行数据，就使用 Console 类的 WriteLine 方法将它在控制台上输出出来。当读到文件的最后的空行时，跳出循环，代码块执行完毕。这时会调用 Dispose 方法来释放资源。代码的执行结果如下：

```

ID    ,用户姓名    ,地址
0001 , 令狐冲    ,华山剑派
0002 , 任盈盈    ,黑木崖
0003 , 东方不败    ,黑木崖
0004 , 风清扬    ,华山后山

```

最后还需要注意一点的是：在 using 语句中的用来引用资源的变量是只读的，在 using 语句中不可以对它

进行更改，如果将这段代码更改为如下的形式：

```

001 using System;
002 using System.IO;
003 namespace AdFeatures3
004 {
005     class Program
006     {
007         static void Main(string[] args)
008         {
009             using (TextWriter tw = File.CreateText("a.csv"))
010             {
011                 tw.WriteLine("ID    ,用户姓名    ,地址");
012                 tw.WriteLine("0001 , 令狐冲    ,华山剑派");
013                 tw.WriteLine("0002 , 任盈盈    ,黑木崖");
014                 tw.WriteLine("0003 , 东方不败    ,黑木崖");
015                 tw.WriteLine("0004 , 风清扬    ,华山后山");
016                 tw.Dispose();
017                 tw = File.CreateText("b.csv");
018             }
019             using(TextReader tr = File.OpenText("a.csv"))
020             {
021                 string s;
022                 while((s = tr.ReadLine()) != null)
023                 {
024                     Console.WriteLine(s);
025                 }
026             }
027             Console.ReadKey();
028         }
029     }
030 }

```

在代码的第 16 行释放资源之后，又为变量创建了一个资源的连接。但这时编译器会报告错误，错误信息如下：

```

错误 1    “tw” 是一个 “using 变量”，无法为它赋值    E:\书稿 new\书稿源码\第二十一章
\AdFeatures3\AdFeatures3.cs 17    17    AdFeatures3

```

第二十二章 集合与查询表达式

语言集成查询 (LINQ) 是一组技术的名称, 它们建立在将查询功能集成到 C# 语言的基础上。通过 LINQ, 现在查询已经变成了 C# 语言的组成部分, 就如同它的其它部分, 如类、接口、属性等等。在 C# 中, LINQ 最显著的应用就是查询表达式。查询表达式是使用 C# 3.0 中引入的声明性查询语法编写的。使用查询表达式, 就可以通过最简化的代码执行对数据源的复杂的筛选、排序和分组操作。查询表达式又提供了统一的接口, 可以使用查询表达式查询和转换 SQL 数据库、ADO.NET 数据集、XML 文档和流以及 .NET 集合中的数据。因为本章在介绍查询表达式的时候要使用数据源, 所以下面先简要介绍一下集合类。

22.1 非泛型集合

关于集合, 在前面的章节中已经接触过, 曾经建立的链表就是集合的一种。数组也是一种集合, 但是它的缺点是大小固定, 不能动态增长和收缩。在 .Net 类库中的 System.Collections 中有几个常用的集合类, 现在来一一的简单介绍。下面先来看 ArrayList 类, 它是大小可以增长的数组:

```
001 using System;
002 using System.Collections;
003 namespace LinQ1
004 {
005     class Person
006     {
007         public int Age
008         {
009             get;
010             set;
011         }
012         public Person(int age)
013         {
014             Age = age;
015         }
016         public void Print()
017         {
018             Console.WriteLine("年龄是: {0}", Age);
019         }
020         public override string ToString()
021         {
022             string s = string.Format("年龄是: {0}", Age);
023             return s;
024         }
025     }
026     class Program
027     {
028         static void Main(string[] args)
029         {
030             ArrayList myArrayList = new ArrayList(10);
031             myArrayList.Add("Hello World");
032             myArrayList.Add(1000);
```



```

033         myArrayList.Add(true);
034         myArrayList.Add(new Person(30));
035         Console.WriteLine("现在集合中有 {0} 个元素", myArrayList.Count);
036         Console.WriteLine("集合中可包含 {0} 个元素", myArrayList.Capacity);
037         myArrayList.Insert(1, "这是一个动态数组");
038         foreach (object o in myArrayList)
039         {
040             Console.WriteLine(o.ToString());
041         }
042         Console.ReadKey();
043     }
044 }
045 }

```

非泛型集合可以加入集合的成员都是 `object` 类型的。也就是说，定义一个集合后，不只是 C# 的基础类型，包括自定义类型也可以加入集合。因此本段代码先定义了一个类 `Person`，并且重写了 `ToString` 方法，以方便打印输出。

在第 30 行代码处，定义了一个 `ArrayList` 类型的实例，为构造方法传入一个整数 10。它表示这个动态数组的大小可以包括 10 个成员。因为 `ArrayList` 的内部实现就是数组，所以需要有一个范围值。如果装入数组的成员超过了这个值，还可以继续自动增长。构造方法也可以不带参数，那样 `ArrayList` 会默认使用一个值。向集合中装入数据就是使用 `Add` 方法，它的参数是 `object` 类型。第 34 行添加了自定义的类的实例。第 35 行引用了 `ArrayList` 的 `Count` 属性输出了集合中的成员个数。第 36 行引用了 `ArrayList` 的 `Capacity` 属性输出了集合的初始化大小。使用 `Insert` 方法可以在指定索引处添加成员。

如果要迭代输出集合的成员，可以使用 `foreach` 语句。它循环调用了各个成员的 `ToString` 方法。代码的执行结果如下：

```

现在集合中有 4 个元素
集合中可包含 10 个元素
Hello World
这是一个动态数组
1000
True
年龄是：30

```

成员的输出还可以使用下面的方式，代码如下：

```

001     using System;
002     using System.Collections;
003     namespace LinQ1
004     {
005         class Person
006         {
007             public int Age
008             {
009                 get;
010                 set;
011             }
012             public Person(int age)

```

```
013     {
014         Age = age;
015     }
016     public void Print()
017     {
018         Console.WriteLine("年龄是: {0}", Age);
019     }
020     public override string ToString()
021     {
022         string s = string.Format("年龄是: {0}", Age);
023         return s;
024     }
025 }
026 class Program
027 {
028     static void Main(string[] args)
029     {
030         ArrayList myArrayList = new ArrayList(10);
031         myArrayList.Add("Hello World");
032         myArrayList.Add(1000);
033         myArrayList.Add(true);
034         myArrayList.Add(new Person(30));
035         myArrayList.Insert(1, "这是一个动态数组");
036         for (int i = 0; i < myArrayList.Count; i++)
037         {
038             Console.WriteLine(myArrayList[i].ToString());
039         }
040         myArrayList.Remove(1000);
041         Console.WriteLine();
042         for (int i = 0; i < myArrayList.Count; i++)
043         {
044             Console.WriteLine(myArrayList[i].ToString());
045         }
046         Console.ReadKey();
047     }
048 }
049 }
```

这段代码的第 36 行使用了 for 语句进行了集合的循环输出。其中范围的确定是靠 ArrayList 的 Count 属性。使用 ArrayList 的 Remove 方法可以移除集合中第一个有参数特征的成员。例如第 40 行将移除集合中第一个成员 1000。如果还有一个成员也是 1000, 则不会一次性移除掉。使用 ArrayList 的 Clear 方法可以将全部成员删除掉。下面是代码的执行结果:

```
Hello World
这是一个动态数组
1000
```

True
年龄是: 30

Hello World
这是一个动态数组

True
年龄是: 30

集合中的成员还可以进行排序, 对于自定义类型来说, 需要实现 `IComparable` 接口。下面是代码实例:

```
001 using System;
002 using System.Collections;
003 namespace LinQ1
004 {
005     class Person:IComparable
006     {
007         public int Age
008         {
009             get;
010             set;
011         }
012         public Person(int age)
013         {
014             Age = age;
015         }
016         public void Print()
017         {
018             Console.WriteLine("年龄是: {0}", Age);
019         }
020         public override string ToString()
021         {
022             string s = string.Format("年龄是: {0}", Age);
023             return s;
024         }
025         public int CompareTo(object obj)
026         {
027             if (this.Age < ((Person)obj).Age)
028                 return -1;
029             else if (this.Age == ((Person)obj).Age)
030                 return 0;
031             else
032                 return 1;
033         }
034     }
035     class Program
036     {
```

```

037     static void Main(string[] args)
038     {
039         ArrayList myArrayList = new ArrayList(10);
040         myArrayList.Add(10);
041         myArrayList.Add(20);
042         myArrayList.Add(60);
043         myArrayList.Add(50);
044         myArrayList.Insert(1, 1000);
045         for (int i = 0; i < myArrayList.Count; i++)
046         {
047             Console.WriteLine(myArrayList[i].ToString());
048         }
049         Console.WriteLine("现在进行排序: ");
050         myArrayList.Sort();
051         for (int i = 0; i < myArrayList.Count; i++)
052         {
053             Console.WriteLine(myArrayList[i].ToString());
054         }
055         myArrayList.Clear();
056         myArrayList.Add(new Person(50));
057         myArrayList.Add(new Person(20));
058         myArrayList.Add(new Person(30));
059         myArrayList.Add(new Person(10));
060         myArrayList.Add(new Person(70));
061         Console.WriteLine("现在进行排序: ");
062         myArrayList.Sort();
063         for (int i = 0; i < myArrayList.Count; i++)
064         {
065             Console.WriteLine(myArrayList[i].ToString());
066         }
067         Console.ReadKey();
068     }
069 }
070 }

```

这段代码首先为 Person 类实现了 IComparable 接口, 这个接口只有一个方法 CompareTo, 它比较两个实例的大小, 如果第一个小于第二个则返回一个小于 0 的值; 如果相等则返回 0; 如果第一个大于第二个, 则返回一个正数。在实现这个方法的时候, 采用了比较 Person 类的 Age 属性的方法。对于 .Net 类库中的简单类型来说, 已经实现了这个接口, 所以对于 int 类型的成员可以正常排序。第 55 行代码调用 Clear 方法清除掉了全部集合成员, 然后重新添加 Person 类型的实例。再次调用 Sort 方法即进行了排序。如果自定义的类没有实现 IComparable 接口, 则编译器会抛异常。这段代码的执行结果如下:

```

10
1000
20
60

```

```
50
现在进行排序：
10
20
50
60
1000
现在进行排序：
年龄是：10
年龄是：20
年龄是：30
年龄是：50
年龄是：70
```

22.1.1 Hashtable 集合

Hashtable 集合是根据键的哈希代码进行组织的键/值对的集合。下面是代码实例：

```
001 using System;
002 using System.Collections;
003 namespace LinQ1
004 {
005     class Person: IComparable
006     {
007         public int Age
008         {
009             get;
010             set;
011         }
012         public Person(int age)
013         {
014             Age = age;
015         }
016         public void Print()
017         {
018             Console.WriteLine("年龄是：{0}", Age);
019         }
020         public override string ToString()
021         {
022             string s = string.Format("年龄是：{0}", Age);
023             return s;
024         }
025         public int CompareTo(object obj)
026         {
027             if (this.Age < ((Person)obj).Age)
028                 return -1;
029             else if (this.Age == ((Person)obj).Age)
```

```
030         return 0;
031     else
032         return 1;
033     }
034 }
035 class Program
036 {
037     static void Main(string[] args)
038     {
039         Hashtable myHashtable = new Hashtable();
040         myHashtable.Add("001", 10);
041         myHashtable.Add("002", 20);
042         myHashtable.Add("003", 40);
043         myHashtable.Add("004", 30);
044         myHashtable.Add("005", 50);
045         Console.WriteLine("编号  值");
046         foreach (DictionaryEntry de in myHashtable)
047         {
048             Console.WriteLine("{0}    {1}", de.Key, de.Value);
049         }
050         Console.WriteLine("现在查找是否包含 003 号的值:
051         {0}", myHashtable.Contains("003"));
052         Console.WriteLine("003 号的值是: {0}", myHashtable["003"]);
053         myHashtable.Clear();
054         Console.WriteLine();
055         myHashtable.Add("李老师", new Person(50));
056         myHashtable.Add("王老师", new Person(20));
057         myHashtable.Add("刘老师", new Person(30));
058         myHashtable.Add("张老师", new Person(10));
059         myHashtable.Add("赵老师", new Person(70));
060         Console.WriteLine("姓名      年龄");
061         foreach (DictionaryEntry de in myHashtable)
062         {
063             Console.WriteLine("{0}    {1}", de.Key, de.Value);
064         }
065         Console.WriteLine("现在查找是否包含李老师的年龄信息: {0}",
066         myHashtable.Contains("李老师"));
067         Console.WriteLine("李老师的年龄是: {0}", myHashtable["李老师"]);
068         Console.ReadKey();
069     }
070 }
```

在代码的第 2 行,为了简化代码的输入,引用了包含非泛型集合的 System.Collections 命名空间。代码的第 39 行定义了一个 Hashtable 类的实例。为 Hashtable 集合添加的成员都是键值对的形式。类型都是

object 类型的。当循环输出集合中的内容的时候，在 foreach 循环中使用了 DictionaryEntry 结构的实例，集合中的键对应的是 DictionaryEntry 结构的 Key 属性；值对应的是 DictionaryEntry 结构的 Value 属性。集合的查找可以调用 Hashtable 类的 Contains 方法，它用来查找键。如果要查找值，可以调用 ContainsValue 方法。Hashtable 集合实现了索引器，因此可以使用第 51 行那样的语法输出它的具体键对应的值。这段代码的执行结果如下：

```

编号  值
001   10
002   20
005   50
004   30
003   40
现在查找是否包含 003 号的值: True
003 号的值是: 40

姓名      年龄
赵老师    年龄是: 70
王老师    年龄是: 20
李老师    年龄是: 50
张老师    年龄是: 10
刘老师    年龄是: 30
现在查找是否包含李老师的年龄信息: True
李老师的年龄是: 年龄是: 50

```

22.1.2 Queue 集合

Queue 集合是一个先进先出的队列，下面是代码实例：

```

001  using System;
002  using System.Collections;
003  namespace LinQ1
004  {
005      class Person: IComparable
006      {
007          public int Age
008          {
009              get;
010              set;
011          }
012          public Person(int age)
013          {
014              Age = age;
015          }
016          public void Print()
017          {
018              Console.WriteLine("年龄是: {0}", Age);
019          }
020          public override string ToString()

```

```
021     {
022         string s = string.Format("年龄是: {0}", Age);
023         return s;
024     }
025     public int CompareTo(object obj)
026     {
027         if (this.Age < ((Person)obj).Age)
028             return -1;
029         else if (this.Age == ((Person)obj).Age)
030             return 0;
031         else
032             return 1;
033     }
034 }
035 class Program
036 {
037     static void Main(string[] args)
038     {
039         Queue myQueue = new Queue();
040         myQueue.Enqueue(10);
041         myQueue.Enqueue(20);
042         myQueue.Enqueue(40);
043         myQueue.Enqueue(30);
044         myQueue.Enqueue(50);
045         foreach (int i in myQueue)
046         {
047             Console.WriteLine("队列中的值是: {0}", i);
048         }
049         myQueue.Clear();
050         Console.WriteLine();
051         myQueue.Enqueue(new Person(50));
052         myQueue.Enqueue(new Person(20));
053         myQueue.Enqueue(new Person(30));
054         myQueue.Enqueue(new Person(10));
055         myQueue.Enqueue(new Person(70));
056         foreach (Person p in myQueue)
057         {
058             Console.WriteLine("年龄是: {0}", p.Age);
059         }
060         Console.WriteLine("取得队列第一个值, 然后删除它");
061         Person p1 = (Person)myQueue.Dequeue();
062         p1.Print();
063         Console.WriteLine("现在队列中的值是: ");
064         foreach (Person p in myQueue)
```



```

065      {
066          Console.WriteLine("年龄是: {0}", p.Age);
067      }
068      Console.ReadKey();
069  }
070  }
071  }

```

在代码的第39行定义了一个队列的集合,接下来调用队列的 Enqueue 方法将成员添加到队列的结尾处。打印队列中的成员只需要使用 foreach 语句就可以了。调用队列的 Dequeue 方法可以返回队列中的第一个成员,然后在队列中删除它。如果调用队列的 Peek 成员可以返回队列的第一个成员,但不删除它。代码的执行结果如下:

```

队列中的值是: 10
队列中的值是: 20
队列中的值是: 40
队列中的值是: 30
队列中的值是: 50

```

```

年龄是: 50
年龄是: 20
年龄是: 30
年龄是: 10
年龄是: 70
取得队列第一个值, 然后删除它
年龄是: 50
现在队列中的值是:
年龄是: 20
年龄是: 30
年龄是: 10
年龄是: 70

```

22.1.3 SortedList 集合

SortedList 是表示键/值对的集合,这些键值对按键排序并可按照键和索引进行访问。代码实例如下:

```

001  using System;
002  using System.Collections;
003  namespace LinQ1
004  {
005      class Person: IComparable
006      {
007          public int Age
008          {
009              get;
010              set;
011          }
012          public Person(int age)
013          {

```

```
014         Age = age;
015     }
016     public void Print()
017     {
018         Console.WriteLine("年龄是: {0}", Age);
019     }
020     public override string ToString()
021     {
022         string s = string.Format("年龄是: {0}", Age);
023         return s;
024     }
025     public int CompareTo(object obj)
026     {
027         if (this.Age < ((Person)obj).Age)
028             return -1;
029         else if (this.Age == ((Person)obj).Age)
030             return 0;
031         else
032             return 1;
033     }
034 }
035 class Program
036 {
037     static void Main(string[] args)
038     {
039         SortedList mySortedList = new SortedList();
040         mySortedList.Add("01", 10);
041         mySortedList.Add("04", 30);
042         mySortedList.Add("03", 20);
043         mySortedList.Add("02", 60);
044         mySortedList.Add("05", 50);
045         foreach (DictionaryEntry de in mySortedList)
046         {
047             Console.WriteLine("集合中的键值对是: {0}, {1}", de.Key, de.Value);
048         }
049         Console.WriteLine("集合的值中包含 20 吗? {0}",
050             mySortedList.ContainsValue(20));
051         mySortedList.Clear();
052         Console.WriteLine();
053         mySortedList.Add(new Person(50), "李老师");
054         mySortedList.Add(new Person(20), "王老师");
055         mySortedList.Add(new Person(30), "周老师");
056         mySortedList.Add(new Person(10), "杨老师");
057         mySortedList.Add(new Person(70), "孙老师");
```

```

057         foreach (DictionaryEntry de in mySortedList)
058         {
059             Console.WriteLine("{0}, 姓名是: {1}", de.Key, de.Value);
060         }
061         Console.WriteLine("取得队列第一个值, 然后删除它");
062         string name = (string)mySortedList.GetByIndex(2);
063         Console.WriteLine("获得的老师的名字是: {0}", name);
064         Console.ReadKey();
065     }
066 }
067 }

```

SortedList 集合的添加数据的方式和 Hashtable 集合很相似。它也是键值对的集合。不过它的排序是自动的。判断集合中是否包含指定值可以调用集合的 ContainsValue 方法。第 52 行开始添加的数据的键是自定义的 Person 类型, 因为集合要使用键进行排序, 所以这个自定义的类一定要实现了 IComparable 接口。调用集合的 GetByIndex 方法可以取得指定索引处的值, 因为键是 Person 类型, 而值是 string 类型, 所以要用 string 类型的变量来接收 GetByIndex 返回的值。代码的执行结果如下:

```

集合中的键值对是: 01, 10
集合中的键值对是: 02, 60
集合中的键值对是: 03, 20
集合中的键值对是: 04, 30
集合中的键值对是: 05, 50
集合的值中包含 20 吗? True

```

```

年龄是: 10, 姓名是: 杨老师
年龄是: 20, 姓名是: 王老师
年龄是: 30, 姓名是: 周老师
年龄是: 50, 姓名是: 李老师
年龄是: 70, 姓名是: 孙老师
取得队列第一个值, 然后删除它
获得的老师的名字是: 周老师

```

如果自定义的类没有实现 IComparable 接口, 则编译器在编译过程中会报错, 错误信息如下:

```

未处理 InvalidOperationException
未能比较数组中的两个元素

```

22.1.4 Stack 集合

stack 集合是一个类似栈的集合, 添加进集合的成员是后进先出的。下面是代码实例:

```

001     using System;
002     using System.Collections;
003     namespace Linq1
004     {
005         class Person:IComparable
006         {
007             public int Age
008             {
009                 get;

```

```
010         set;
011     }
012     public Person(int age)
013     {
014         Age = age;
015     }
016     public void Print()
017     {
018         Console.WriteLine("年龄是: {0}", Age);
019     }
020     public override string ToString()
021     {
022         string s = string.Format("年龄是: {0}", Age);
023         return s;
024     }
025     public int CompareTo(object obj)
026     {
027         if (this.Age < ((Person)obj).Age)
028             return -1;
029         else if (this.Age == ((Person)obj).Age)
030             return 0;
031         else
032             return 1;
033     }
034 }
035 class Program
036 {
037     static void Main(string[] args)
038     {
039         Stack myStack = new Stack();
040         myStack.Push(10);
041         myStack.Push(30);
042         myStack.Push(50);
043         myStack.Push(40);
044         myStack.Push(20);
045         foreach (int i in myStack)
046         {
047             Console.WriteLine("集合中的值是: {0}", i);
048         }
049         Console.WriteLine("集合的值中包含 20 吗? {0}", myStack.Contains(20));
050         myStack.Clear();
051         Console.WriteLine();
052         myStack.Push(new Person(50));
053         myStack.Push(new Person(20));
```

```

054         myStack.Push(new Person(30));
055         myStack.Push(new Person(10));
056         myStack.Push(new Person(70));
057         foreach (Person p in myStack)
058         {
059             Console.WriteLine("年龄是: {0}", p.Age);
060         }
061         Console.WriteLine("取得栈集合中第一个值, 然后删除它");
062         Person p1 = (Person)myStack.Pop();
063         Console.WriteLine("获得的老师的年龄是: {0}", p1.Age);
064         Console.ReadKey();
065     }
066 }
067 }

```

代码第39行定义了一个栈集合,为栈集合中添加成员的方法是调用集合的Push方法,可以使用foreach循环对其进行遍历。调用栈集合的Pop方法可以返回栈集合第一个元素,然后将它删除。调用栈集合的Peek方法可以返回栈集合第一个元素,但不删除它。这段代码的执行结果如下:

```

集合中的值是: 20
集合中的值是: 40
集合中的值是: 50
集合中的值是: 30
集合中的值是: 10
集合的值中包含 20 吗? True

年龄是: 70
年龄是: 10
年龄是: 30
年龄是: 20
年龄是: 50
取得栈集合中第一个值, 然后删除它
获得的老师的年龄是: 70

```

22.2 泛型集合

对于前面介绍过的非泛型集合来说,都有泛型集合与之相对。但是对于 ArrayList 集合来说,相对应的泛型的类名字叫做 List<T>类。List<T>类可以通过索引进行访问。下面是代码实例:

```

001     using System;
002     using System.Collections;
003     using System.Collections.Generic;
004     namespace LinQ1
005     {
006         class Person: IComparable
007         {
008             public int Age
009             {
010                 get;

```

```
011         set;
012     }
013     public Person(int age)
014     {
015         Age = age;
016     }
017     public void Print()
018     {
019         Console.WriteLine("年龄是: {0}", Age);
020     }
021     public override string ToString()
022     {
023         string s = string.Format("年龄是: {0}", Age);
024         return s;
025     }
026     public int CompareTo(object obj)
027     {
028         if (this.Age < ((Person)obj).Age)
029             return -1;
030         else if (this.Age == ((Person)obj).Age)
031             return 0;
032         else
033             return 1;
034     }
035 }
036 class Program
037 {
038     static void Main(string[] args)
039     {
040         List<int> myList = new List<int>();
041         myList.Add(10);
042         myList.Add(30);
043         myList.Add(50);
044         myList.Add(40);
045         myList.Add(20);
046         foreach (int i in myList)
047         {
048             Console.WriteLine("集合中的值是: {0}", i);
049         }
050         Console.WriteLine("集合的值中包含 20 吗? {0}", myList.Contains(20));
051         myList.Clear();
052         Console.WriteLine();
053         List<Person> myList1 = new List<Person>();
054         myList1.Add(new Person(50));
```

```

055         myList1.Add(new Person(20));
056         myList1.Add(new Person(30));
057         myList1.Add(new Person(10));
058         myList1.Add(new Person(70));
059         foreach (Person p in myList1)
060         {
061             Console.WriteLine("年龄是: {0}", p.Age);
062         }
063         Console.WriteLine("现在开始排序");
064         myList1.Sort();
065         foreach (Person p in myList1)
066         {
067             Console.WriteLine("年龄是: {0}", p.Age);
068         }
069         Console.WriteLine();
070         Console.WriteLine("现在开始反转成员");
071         myList1.Reverse();
072         foreach (Person p in myList1)
073         {
074             Console.WriteLine("年龄是: {0}", p.Age);
075         }
076         Console.WriteLine();
077         Console.WriteLine("现在移除索引为 2 处的成员");
078         myList1.RemoveAt(2);
079         foreach (Person p in myList1)
080         {
081             Console.WriteLine("年龄是: {0}", p.Age);
082         }
083         Console.ReadKey();
084     }
085 }
086 }

```

要想简化代码并使用泛型集合的类，需要引用 `System.Collections.Generic` 命名空间。使用 `List<T>` 类型时，需要为其指定类型实参。虽然使用泛型集合需要指定添加的类型，但是泛型集合的效率要比非泛型集合高。因此，推荐使用泛型集合。原因是：非泛型集合使用 `object` 类型的参数，当使用值类型的时候都需要装包，取出时，都需要拆包。效率比较低。而添加引用类型的成员时，取出时又要显式转换。因此，泛型集合与之相比就有很大的优点。`List` 泛型集合添加数据也使用 `Add` 方法，使用 `foreach` 语句可以对其进行遍历。当调用 `Sort` 方法的时候，可以实现集合的排序，但是这要求成员类实现了 `IComparable` 接口。当调用 `Reverse` 方法的时候，可以对集合进行反转。这个功能也基于实现 `IComparable` 接口。第 78 行代码调用了 `RemoveAt` 方法删除了指定索引处的成员。这段代码的执行结果如下：

```

集合中的值是: 10
集合中的值是: 30
集合中的值是: 50
集合中的值是: 40

```

集合中的值是: 20
集合的值中包含 20 吗? True

年龄是: 50
年龄是: 20
年龄是: 30
年龄是: 10
年龄是: 70
现在开始排序
年龄是: 10
年龄是: 20
年龄是: 30
年龄是: 50
年龄是: 70

现在开始反转成员
年龄是: 70
年龄是: 50
年龄是: 30
年龄是: 20
年龄是: 10

现在移除索引为 2 处的成员
年龄是: 70
年龄是: 50
年龄是: 20
年龄是: 10

22.2.1 Dictionary<TKey, TValue>集合

Dictionary<TKey, TValue>集合表示键和值的集合, 也叫字典集合。下面是代码实例:

```
001 using System;
002 using System.Collections;
003 using System.Collections.Generic;
004 namespace LinQ1
005 {
006     class Person: IComparable
007     {
008         public int Age
009         {
010             get;
011             set;
012         }
013         public Person(int age)
014         {
015             Age = age;
```



```
016     }
017     public void Print()
018     {
019         Console.WriteLine("年龄是: {0}", Age);
020     }
021     public override string ToString()
022     {
023         string s = string.Format("年龄是: {0}", Age);
024         return s;
025     }
026     public int CompareTo(object obj)
027     {
028         if (this.Age < ((Person)obj).Age)
029             return -1;
030         else if (this.Age == ((Person)obj).Age)
031             return 0;
032         else
033             return 1;
034     }
035 }
036 class Program
037 {
038     static void Main(string[] args)
039     {
040         Dictionary<string, Person> myDictionary = new Dictionary<string, Person>();
041         myDictionary.Add("王老师", new Person(50));
042         myDictionary.Add("张老师", new Person(20));
043         myDictionary.Add("李老师", new Person(30));
044         myDictionary.Add("杨老师", new Person(36));
045         myDictionary.Add("孙老师", new Person(70));
046         foreach (KeyValuePair<string, Person> kvp in myDictionary)
047         {
048             Console.WriteLine("姓名是: {0}, 年龄是: {1}", kvp.Key, kvp.Value);
049         }
050         Console.WriteLine("修改李老师的年龄");
051         myDictionary["李老师"] = new Person(40);
052         foreach (KeyValuePair<string, Person> kvp in myDictionary)
053         {
054             Console.WriteLine("姓名是: {0}, 年龄是: {1}", kvp.Key, kvp.Value);
055         }
056         Person p;
057         if (myDictionary.TryGetValue("杨老师", out p))
058         {
059             Console.WriteLine("得到的老师的年龄是: {0}", p.Age);
```

```
060      }
061      Console.WriteLine("还可以获得集合中的值集合(Person 类实例)");
062      Dictionary<string, Person>.ValueCollection vc = myDictionary.Values;
063      foreach (Person p1 in vc)
064      {
065          Console.WriteLine("老师的年龄是: {0}", p1.Age);
066      }
067      Console.ReadKey();
068  }
069  }
070 }
```

使用 Dictionary 泛型类集合的时候,要为其指定类型实参,因为本段代码要定义的键值对是老师的姓名和 Person 类实例。所以 Dictionary 的构造类型是 Dictionary<string,Person>。为集合添加成员需要调用集合的 Add 方法。使用 foreach 语句遍历集合成员的时候,需要使用 KeyValuePair<string,Person> 结构。可以使用第 51 行那样的索引器来对指定的键进行值的修改和赋值。调用 TryGetValue 方法可以得到指定键的值,这个值是输出参数。ValueCollection 类是 Dictionary 泛型类的派生类,但因为它反向引用了基类的成员,所以要使用 Dictionary<string, Person>.ValueCollection 的定义形式。通过 Dictionary 泛型类的 Values 属性可以返回 ValueCollection 类的实例,它表示值的集合。接下来可以使用 foreach 语句遍历 ValueCollection 集合。这段代码的执行结果如下:

```
姓名是: 王老师, 年龄是: 年龄是: 50
姓名是: 张老师, 年龄是: 年龄是: 20
姓名是: 李老师, 年龄是: 年龄是: 30
姓名是: 杨老师, 年龄是: 年龄是: 36
姓名是: 孙老师, 年龄是: 年龄是: 70
修改李老师的年龄
姓名是: 王老师, 年龄是: 年龄是: 50
姓名是: 张老师, 年龄是: 年龄是: 20
姓名是: 李老师, 年龄是: 年龄是: 40
姓名是: 杨老师, 年龄是: 年龄是: 36
姓名是: 孙老师, 年龄是: 年龄是: 70
得到的老师的年龄是: 36
还可以获得集合中的值集合(Person 类实例)
老师的年龄是: 50
老师的年龄是: 20
老师的年龄是: 40
老师的年龄是: 36
老师的年龄是: 70
```

为 Dictionary 泛型集合添加成员的时候,需要注意键不能重复,如果使用下面的语句为集合添加成员,编译器就会抛出异常。代码如下:

```
try
{
    myDictionary.Add("孙老师", new Person(35));
}
catch (Exception e)
```

```

{
    Console.WriteLine(e.Message);
}
foreach (KeyValuePair<string, Person> kvp in myDictionary)
{
    Console.WriteLine("姓名是: {0}, 年龄是: {1}", kvp.Key, kvp.Value);
}

```

因为集合中已经包含了一个键为“孙老师”的键值对，则编译器会抛出异常。这时添加的这段代码的运行结果如下：

已添加了具有相同键的项。

姓名是: 王老师, 年龄是: 50

姓名是: 张老师, 年龄是: 20

姓名是: 李老师, 年龄是: 40

姓名是: 杨老师, 年龄是: 36

姓名是: 孙老师, 年龄是: 70

22.2.2 SortedList<TKey, TValue>集合

SortedList 泛型集合也是键值对的集合，它按照键进行自动排序。下面是代码实例：

```

001 using System;
002 using System.Collections;
003 using System.Collections.Generic;
004 namespace LinQ1
005 {
006     class Person: IComparable
007     {
008         public int Age
009         {
010             get;
011             set;
012         }
013         public Person(int age)
014         {
015             Age = age;
016         }
017         public void Print()
018         {
019             Console.WriteLine("年龄是: {0}", Age);
020         }
021         public override string ToString()
022         {
023             string s = string.Format("年龄是: {0}", Age);
024             return s;
025         }
026         public int CompareTo(object obj)
027         {

```

```
028         if (this.Age < ((Person)obj).Age)
029             return -1;
030         else if (this.Age == ((Person)obj).Age)
031             return 0;
032         else
033             return 1;
034     }
035 }
036 class Program
037 {
038     static void Main(string[] args)
039     {
040         SortedList<string, Person> mySortedList = new SortedList<string, Person>();
041         mySortedList.Add("王老师", new Person(50));
042         mySortedList.Add("张老师", new Person(20));
043         mySortedList.Add("李老师", new Person(30));
044         mySortedList.Add("杨老师", new Person(36));
045         mySortedList.Add("孙老师", new Person(70));
046         foreach (KeyValuePair<string, Person> kvp in mySortedList)
047         {
048             Console.WriteLine("姓名是: {0}, 年龄是: {1}", kvp.Key, kvp.Value);
049         }
050         Console.WriteLine("修改李老师的年龄");
051         mySortedList["李老师"] = new Person(40);
052         foreach (KeyValuePair<string, Person> kvp in mySortedList)
053         {
054             Console.WriteLine("姓名是: {0}, 年龄是: {1}", kvp.Key, kvp.Value);
055         }
056         Person p;
057         if (mySortedList.TryGetValue("杨老师", out p))
058         {
059             Console.WriteLine("得到的老师的年龄是: {0}", p.Age);
060         }
061         Console.WriteLine("还可以获得集合中的值集合(Person 类实例)");
062         IList<Person> il = mySortedList.Values;
063         foreach (Person p1 in il)
064         {
065             Console.WriteLine("老师的年龄是: {0}", p1.Age);
066         }
067         try
068         {
069             mySortedList.Add("孙老师", new Person(35));
070         }
071         catch (Exception e)
```

```

072         {
073             Console.WriteLine(e.Message);
074         }
075         foreach (KeyValuePair<string, Person> kvp in mySortedList)
076         {
077             Console.WriteLine("姓名是: {0}, 年龄是: {1}", kvp.Key, kvp.Value);
078         }
079         Console.ReadKey();
080     }
081 }
082 }

```

SortedList 泛型的使用方法和 Dictionary 泛型类集合的使用方法相同，唯一不同的是第 62 行调用 SortedList.Values 属性时，需要 IList<Person> 接口变量来引用它。在使用 foreach 语句遍历输出集合成员的时候，可以看到，它的排序方法是按照 string 类型进行的排序，因为 string 类型也实现了比较的接口。SortedList 泛型也不允许重复添加键值对成员。在重复添加键值对的时候，编译器会抛异常出来。代码的执行结果如下：

```

姓名是: 李老师, 年龄是: 30
姓名是: 孙老师, 年龄是: 70
姓名是: 王老师, 年龄是: 50
姓名是: 杨老师, 年龄是: 36
姓名是: 张老师, 年龄是: 20
修改李老师的年龄
姓名是: 李老师, 年龄是: 40
姓名是: 孙老师, 年龄是: 70
姓名是: 王老师, 年龄是: 50
姓名是: 杨老师, 年龄是: 36
姓名是: 张老师, 年龄是: 20
得到的老师的年龄是: 36
还可以获得集合中的值集合(Person 类实例)
老师的年龄是: 40
老师的年龄是: 70
老师的年龄是: 50
老师的年龄是: 36
老师的年龄是: 20
已存在具有相同键的条目。
姓名是: 李老师, 年龄是: 40
姓名是: 孙老师, 年龄是: 70
姓名是: 王老师, 年龄是: 50
姓名是: 杨老师, 年龄是: 36
姓名是: 张老师, 年龄是: 20

```

22.2.3 Queue<T>集合

Queue<T>集合是对象的先进先出集合。下面是代码实例：

```

001 using System;
002 using System.Collections;

```

```
003 using System.Collections.Generic;
004 namespace LinQ1
005 {
006     class Person:IComparable
007     {
008         public int Age
009         {
010             get;
011             set;
012         }
013         public Person(int age)
014         {
015             Age = age;
016         }
017         public void Print()
018         {
019             Console.WriteLine("年龄是: {0}", Age);
020         }
021         public override string ToString()
022         {
023             string s = string.Format("年龄是: {0}", Age);
024             return s;
025         }
026         public int CompareTo(object obj)
027         {
028             if (this.Age < ((Person)obj).Age)
029                 return -1;
030             else if (this.Age == ((Person)obj).Age)
031                 return 0;
032             else
033                 return 1;
034         }
035     }
036     class Program
037     {
038         static void Main(string[] args)
039         {
040             Queue<Person> myQueue = new Queue<Person>();
041             myQueue.Enqueue(new Person(50));
042             myQueue.Enqueue(new Person(23));
043             myQueue.Enqueue(new Person(30));
044             myQueue.Enqueue(new Person(55));
045             myQueue.Enqueue(new Person(40));
046             foreach (Person p in myQueue)
```

```

047         {
048             Console.WriteLine("年龄是: {0}", p.Age);
049         }
050         Person p1 = myQueue.Dequeue();
051         Console.WriteLine("获得并移除的队列成员的年龄是: {0}", p1.Age);
052         Person[] myPersonArray = new Person[4];
053         myQueue.CopyTo(myPersonArray, 0);
054         Console.WriteLine("复制出来的 Person 类型的数组中的元素有: ");
055         foreach (Person p in myPersonArray)
056         {
057             Console.WriteLine("队列成员的年龄是: {0}", p.Age);
058         }
059         Console.ReadKey();
060     }
061 }
062 }

```

Queue 泛型集合添加成员的方式和 Queue 非泛型集合的添加成员方式一样。第 50 行调用了集合的 Dequeue 返回队列的第一个成员，然后移除这个成员。第 52 行代码定义了一个数组并创建了实例。第 53 行代码调用了集合的 CopyTo 方法将集合成员复制到了数组中，注意这个方法第二个参数是目标数组的索引值，意思是从索引值处开始向其复制。这段代码的执行结果如下：

```

年龄是: 50
年龄是: 23
年龄是: 30
年龄是: 55
年龄是: 40
获得并移除的队列成员的年龄是: 50
复制出来的 Person 类型的数组中的元素有:
队列成员的年龄是: 23
队列成员的年龄是: 30
队列成员的年龄是: 55
队列成员的年龄是: 40

```

22.2.4 Stack<T>集合

Stack<T>集合的使用方式和非泛型的 Stack 集合的使用方式大致相同。下面是代码实例：

```

001     using System;
002     using System.Collections;
003     using System.Collections.Generic;
004     namespace LinQ1
005     {
006         class Person: IComparable
007         {
008             public int Age
009             {
010                 get;
011                 set;

```

```
012     }
013     public Person(int age)
014     {
015         Age = age;
016     }
017     public void Print()
018     {
019         Console.WriteLine("年龄是: {0}", Age);
020     }
021     public override string ToString()
022     {
023         string s = string.Format("年龄是: {0}", Age);
024         return s;
025     }
026     public int CompareTo(object obj)
027     {
028         if (this.Age < ((Person)obj).Age)
029             return -1;
030         else if (this.Age == ((Person)obj).Age)
031             return 0;
032         else
033             return 1;
034     }
035 }
036 class Program
037 {
038     static void Main(string[] args)
039     {
040         Stack<Person> myStack = new Stack<Person>();
041         myStack.Push(new Person(50));
042         myStack.Push(new Person(23));
043         myStack.Push(new Person(30));
044         myStack.Push(new Person(55));
045         myStack.Push(new Person(40));
046         foreach (Person p in myStack)
047         {
048             Console.WriteLine("年龄是: {0}", p.Age);
049         }
050         Person p1 = myStack.Pop();
051         Console.WriteLine("获得并移除的栈集合成员的年龄是: {0}", p1.Age);
052         Person[] myPersonArray = new Person[4];
053         myStack.CopyTo(myPersonArray, 0);
054         Console.WriteLine("复制出来的 Person 类型的数组中的元素有: ");
055         foreach (Person p in myPersonArray)
```



```

056         {
057             Console.WriteLine("队列成员的年龄是: {0}", p.Age);
058         }
059         Console.ReadKey();
060     }
061 }
062 }

```

对于 Stack 泛型集合的成员添加方法仍旧是调用集合的 Push，每次调用 Push 方法都会把成员添加到栈集合的顶部。使用 Pop 方法可以弹出栈集合的顶端元素，然后删除顶端元素。CopyTo 方法的使用和上一小节的 Queue 集合的使用方法一样。代码的执行结果如下：

```

年龄是: 40
年龄是: 55
年龄是: 30
年龄是: 23
年龄是: 50
获得并移除的栈集合成员的年龄是: 40
复制出来的 Person 类型的数组中的元素有:
队列成员的年龄是: 55
队列成员的年龄是: 30
队列成员的年龄是: 23
队列成员的年龄是: 50

```

22.2.5 SortedSet<T>

SortedSet<T>集合表示按排序顺序保持的对象的集合。添加它里面的成员始终是排序的。下面是代码实例：

```

001 using System;
002 using System.Collections;
003 using System.Collections.Generic;
004 namespace LinQ1
005 {
006     class Person: IComparable
007     {
008         public int Age
009         {
010             get;
011             set;
012         }
013         public Person(int age)
014         {
015             Age = age;
016         }
017         public void Print()
018         {
019             Console.WriteLine("年龄是: {0}", Age);
020         }
021     }
022 }

```

```
021     public override string ToString()
022     {
023         string s = string.Format("年龄是: {0}", Age);
024         return s;
025     }
026     public int CompareTo(object obj)
027     {
028         if (this.Age < ((Person)obj).Age)
029             return -1;
030         else if (this.Age == ((Person)obj).Age)
031             return 0;
032         else
033             return 1;
034     }
035 }
036 class Program
037 {
038     static void Main(string[] args)
039     {
040         SortedSet<Person> mySortedSet = new SortedSet<Person>();
041         Person p1 = new Person(50);
042         Person p2 = new Person(23);
043         Person p3 = new Person(30);
044         Person p4 = new Person(55);
045         Person p5 = new Person(40);
046         mySortedSet.Add(p1);
047         mySortedSet.Add(p2);
048         mySortedSet.Add(p3);
049         mySortedSet.Add(p4);
050         mySortedSet.Add(p5);
051         foreach (Person p in mySortedSet)
052         {
053             Console.WriteLine("排序集合中的年龄是: {0}", p.Age);
054         }
055         mySortedSet.Remove(p3);
056         Console.WriteLine("删除 p3 后排序");
057         foreach (Person p in mySortedSet)
058         {
059             Console.WriteLine("排序集合中的年龄是: {0}", p.Age);
060         }
061         Person p6 = new Person(32);
062         Console.WriteLine("添加 p6 后的排序");
063         mySortedSet.Add(p6);
064         foreach (Person p in mySortedSet)
```

```

065         {
066             Console.WriteLine("排序集合中的年龄是: {0}", p.Age);
067         }
068         Console.ReadKey();
069     }
070 }
071 }

```

为 SortedSet 集合添加成员需要调用 Add 方法, 不论从集合中删除成员还是添加成员, 集合中的成员始终都是排序的。当然, 这需要集合中的元素实现 IComparable 接口。代码的执行结果如下:

```

排序集合中的年龄是: 23
排序集合中的年龄是: 30
排序集合中的年龄是: 40
排序集合中的年龄是: 50
排序集合中的年龄是: 55
删除 p3 后排序
排序集合中的年龄是: 23
排序集合中的年龄是: 40
排序集合中的年龄是: 50
排序集合中的年龄是: 55
添加 p6 后的排序
排序集合中的年龄是: 23
排序集合中的年龄是: 32
排序集合中的年龄是: 40
排序集合中的年龄是: 50
排序集合中的年龄是: 55

```

22.2.6 LinkedList<T>集合

LinkedList<T>集合是一个双向链表。它的每个节点都有两个引用, 一个指向它的前一个节点, 一个指向它的后一个节点。下面是代码实例:

```

001 using System;
002 using System.Collections;
003 using System.Collections.Generic;
004 namespace LinQ1
005 {
006     class Person:IComparable
007     {
008         public int Age
009         {
010             get;
011             set;
012         }
013         public Person(int age)
014         {
015             Age = age;
016         }
017     }
018 }

```

```
017     public void Print()
018     {
019         Console.WriteLine("年龄是: {0}", Age);
020     }
021     public override string ToString()
022     {
023         string s = string.Format("年龄是: {0}", Age);
024         return s;
025     }
026     public int CompareTo(object obj)
027     {
028         if (this.Age < ((Person)obj).Age)
029             return -1;
030         else if (this.Age == ((Person)obj).Age)
031             return 0;
032         else
033             return 1;
034     }
035 }
036 class Program
037 {
038     static void Main(string[] args)
039     {
040         LinkedList<Person> myLinkedList = new LinkedList<Person>();
041         Person p1 = new Person(50);
042         Person p2 = new Person(23);
043         Person p3 = new Person(30);
044         Person p4 = new Person(55);
045         Person p5 = new Person(40);
046         myLinkedList.AddFirst(p1);
047         myLinkedList.AddFirst(p2);
048         myLinkedList.AddLast(p3);
049         myLinkedList.AddFirst(p4);
050         myLinkedList.AddLast(p5);
051         foreach (Person p in myLinkedList)
052         {
053             p.Print();
054         }
055         Console.WriteLine("现在开始查找成员 p3");
056         LinkedListNode<Person> llNode = myLinkedList.Find(p3);
057         llNode.Value.Print();
058         Console.WriteLine("现在开始正向遍历节点");
059         LinkedListNode<Person> temp = myLinkedList.First;
060         while(temp != null)
```

```

061      {
062          temp.Value.Print();
063          temp = temp.Next;
064      }
065      Console.WriteLine("现在开始反向遍历节点");
066      LinkedListNode<Person> temp1 = myLinkedList.Last;
067      while (temp1 != null)
068      {
069          temp1.Value.Print();
070          temp1 = temp1.Previous;
071      }
072      Console.ReadKey();
073  }
074  }
075  }

```

双向链表的添加节点的方式是使用 `AddFirst` 方法向链表的前端添加节点, 或使用 `AddLast` 向链表的尾端添加节点。对双向链表可以使用 `foreach` 语句进行遍历, 还可以使用 `Find` 方法查找成员。`Find` 方法返回的是 `LinkedListNode<Person>` 泛型类实例, 它代表双向链表的其中一个节点。访问它的 `Value` 属性即可返回节点中包括的 `Person` 类型实例。除了使用 `foreach` 语句遍历双向链表外, 还可以自定义一个 `LinkedListNode<Person>` 泛型类变量来正向遍历链表或反向遍历链表。

第 59 行代码是定义一个 `LinkedListNode<Person>` 泛型类变量指向链表的第一个节点, 然后在 `while` 循环中引用节点的 `Value` 属性获得 `Person` 类实例。接下来将这个变量指向链表的下一个节点, 直到链表的最后一个节点时, 它的 `Next` 属性为 `null`, 则跳出循环。

反向遍历链表和正向遍历链表的思路相同, 只不过一开始让 `LinkedListNode<Person>` 泛型类变量指向链表的最后一个节点, 然后依次向前引用链表节点的 `Previous` 属性, 即指向前一个链表节点的引用 (`LinkedListNode<Person>` 类型的引用)。代码的执行结果如下:

```

年龄是: 55
年龄是: 23
年龄是: 50
年龄是: 30
年龄是: 40
现在开始查找成员 p3
年龄是: 30
现在开始正向遍历节点
年龄是: 55
年龄是: 23
年龄是: 50
年龄是: 30
年龄是: 40
现在开始反向遍历节点
年龄是: 40
年龄是: 30
年龄是: 50
年龄是: 23

```

年龄是：55

22.3 基本查询表达式

使用 LINQ 查询表达式，可以同许多种数据类型进行交互。例如针对数组和集合；针对 XML；针对 ADO.NET DataSet 等等。但是，查询表达式只是在查询上可以替代各种对象的专有 API。它还不能完全替代它们。例如在编程中经常和 SQL 数据库进行交互，但是一些例如创建数据库、维护数据库等操作，不能用 LINQ 查询表达式来代替。下面来看一个最基本的查询表达式的构成，实例代码如下：

```
001    using System;
002    using System.Collections.Generic;
003    using System.Linq;
004    namespace LINQ2
005    {
006        class Program
007        {
008            static void Main(string[] args)
009            {
010                int[] myArray = new int[20];
011                for (int i = 0; i < 20; i++)
012                {
013                    myArray[i] = i;
014                }
015                IEnumerable<int> mySubSet = from num in myArray where num < 15 select num;
016                Console.WriteLine("查询出的结果包括：");
017                foreach (int i in mySubSet)
018                {
019                    Console.Write(i + " ");
020                }
021                Console.ReadKey();
022            }
023        }
024    }
```

这段代码从一个数组中检索数据。使用查询表达式需要引用 System.Linq 命名空间，使用 IEnumerable<T> 泛型接口需要引用 System.Collections.Generic 命名空间。第 15 行代码是一个查询表达式的演示。查询表达式必须以 from 子句开头，from 关键字后跟的是范围变量。然后是 in 关键字，它指定要查询的数据源。而范围变量指定的是查询的范围。在指定了数据源之后，where 关键字指定查询的条件，select 关键字指定范围变量。查询表达式的返回结果必须是一个实现了 IEnumerable<T> 或者 IEnumerable 接口的实例。

查询表达式可以查询的数据源的类型必须实现了 IEnumerable、IEnumerable<T> 或者它们的派生接口。查询表达式是强类型的，也就是说类型安全的。它的范围变量和数据源都必须能够推断出具体的类型。对于范围变量来说，如果数据源实现了 IEnumerable<T> 接口，则编译器就能够推断出范围变量的类型，如果数据源没有实现 IEnumerable<T> 接口，则需要指定范围变量的类型。

和 foreach 语句中的迭代用的变量不同，查询表达式中的范围变量不存储数据，它只是提供了一种语法上的便利，对查询进行描述。对于查询表达式的返回结果，可以继续使用 foreach 语句进行迭代，因为它实现了 IEnumerable<T> 接口或者 IEnumerable 接口。在这里使用 IEnumerable<T> 接口即可，因为它就是派生自 IEnumerable 接口。在本例中，虽然数组 myArray 的基类 Array 类没有实现 IEnumerable<T> 接口，

但是因为它实现了 IEnumerable 接口，所以返回类型使用 IEnumerable<T>接口是允许的，但是需要为类型形参 T 指定类型实参。这段代码的执行结果如下：

查询出的结果包括：

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

下面再对查询表达式进行简化，实际上，查询表达式的返回结果可以让 C# 的编译器再多做一点事情，即返回结果的类型也由编译器负责推断得出。下面是修改后的代码：

```
001 using System;
002 using System.Collections.Generic;
003 using System.Collections;
004 using System.Linq;
005 namespace LINQ2
006 {
007     class Program
008     {
009         static void Main(string[] args)
010         {
011             int[] myArray = new int[20];
012             for (int i = 0; i < 20; i++)
013             {
014                 myArray[i] = i;
015             }
016             var mySubSet = from num in myArray where num < 15 select num;
017             Console.WriteLine("查询出的结果包括：");
018             foreach (var i in mySubSet)
019             {
020                 Console.Write(i + " ");
021             }
022             Console.ReadKey();
023         }
024     }
025 }
```

这段代码使用 var 推断类型替换了查询表达式的返回结果类型和 foreach 语句中的迭代变量类型。它们的类型都由编译器推断得出。程序运行的结果和前一个例子是一样的。正因为查询表达式的返回结果实现了 IEnumerable 接口或者 IEnumerable<T>接口，所以才能使用 foreach 语句对结果进行迭代输出。

22.3.1 从非泛型集合中查询数据

非泛型集合没有实现 IEnumerable<T>接口，所以编译器无法推断得出集合中的具体类型。在这样的情况下，必须指定范围变量的具体类型，而不能依靠编译器去推断得出。下面是代码实例：

```
001 using System;
002 using System.Collections;
003 using System.Collections.Generic;
004 using System.Linq;
005 namespace LinQ1
006 {
007     class Person: IComparable
```

```
008     {
009         public int Age
010     {
011         get;
012         set;
013     }
014     public Person(int age)
015     {
016         Age = age;
017     }
018     public void Print()
019     {
020         Console.WriteLine("年龄是: {0}", Age);
021     }
022     public override string ToString()
023     {
024         string s = string.Format("年龄是: {0}", Age);
025         return s;
026     }
027     public int CompareTo(object obj)
028     {
029         if (this.Age < ((Person)obj).Age)
030             return -1;
031         else if (this.Age == ((Person)obj).Age)
032             return 0;
033         else
034             return 1;
035     }
036 }
037 class Program
038 {
039     static void Main(string[] args)
040     {
041         ArrayList myArrayList = new ArrayList();
042         Person p1 = new Person(50);
043         Person p2 = new Person(23);
044         Person p3 = new Person(30);
045         Person p4 = new Person(55);
046         Person p5 = new Person(40);
047         myArrayList.Add(p1);
048         myArrayList.Add(p2);
049         myArrayList.Add(p3);
050         myArrayList.Add(p4);
051         myArrayList.Add(p5);
```



```

052         var mySubSet = from Person num in myArrayList where num.Age <= 40 select num;
053         foreach (var v in mySubSet)
054         {
055             v.Print();
056         }
057         Console.ReadKey();
058     }
059 }
060 }

```

本段代码在第 2 行引用了 System.Collections 命名空间, 在第 4 行引用了 System.Linq 命名空间。使用 LINQ 查询表达式是必须要引用 System.Linq 命名空间的, 否则编译器会报告如下错误:

错误 1 找不到源类型“System.Collections.ArrayList”的查询模式的实现。找不到“Cast”。是否缺少引用, 或者缺少用于“System.Linq”的 using 指令? H:\书稿 new\书稿源码\第二十二章\LinQ1\Program.cs 52 47 LinQ1

本段代码使用查询表达式查询的是一个 ArrayList 集合, 在对非泛型集合查询的时候, 是必须要指定范围变量的类型的。因为集合中是 Person 类型, 所以第 52 行中, 范围变量指定为 Person 类型。在使用 where 关键字指定查询条件时, 使用的是 Person 类的 Age 属性, 因此 where 关键字后是 num.Age。查询表达式的返回类型用 var 推断类型, 在 foreach 语句中的迭代变量也用 var 类型。实际上, foreach 语句中的迭代变量就是 Person 类型, 所以可以调用其 Print 方法。代码的执行结果如下:

年龄是: 23

年龄是: 30

年龄是: 40

22.4 筛选

在前面的例子中, where 子句中就是筛选表达式。筛选的作用是让查询表达式只返回那些筛选子句为 true 的结果。还可以使用逻辑运算符来联合使用多个筛选表达式。下面是代码实例:

```

001 using System;
002 using System.Collections;
003 using System.Collections.Generic;
004 using System.Linq;
005 namespace LinQ1
006 {
007     class Person:IComparable
008     {
009         public string Name
010         {
011             get;
012             set;
013         }
014         public int Age
015         {
016             get;
017             set;
018         }
019         public string Address

```

```
020     {
021         get;
022         set;
023     }
024     public Person(string name,int age,string address)
025     {
026         this.Name = name;
027         this.Age = age;
028         this.Address = address;
029     }
030     public void Print()
031     {
032         Console.WriteLine("姓名: {0}, 年龄: {1}, 地址: {2}",Name, Age, Address);
033     }
034     public override string ToString()
035     {
036         string s = string.Format("年龄是: {0}", Age);
037         return s;
038     }
039     public int CompareTo(object obj)
040     {
041         if (this.Age < ((Person)obj).Age)
042             return -1;
043         else if (this.Age == ((Person)obj).Age)
044             return 0;
045         else
046             return 1;
047     }
048 }
049 class Program
050 {
051     static void Main(string[] args)
052     {
053         ArrayList myArrayList = new ArrayList();
054         Person p1 = new Person("云飞扬", 50, "陕西");
055         Person p2 = new Person("令狐冲", 23, "陕西");
056         Person p3 = new Person("任盈盈", 18, "陕西");
057         Person p4 = new Person("向问天", 40, "河北");
058         Person p5 = new Person("陆大有", 20, "河北");
059         myArrayList.Add(p1);
060         myArrayList.Add(p2);
061         myArrayList.Add(p3);
062         myArrayList.Add(p4);
063         myArrayList.Add(p5);
```

```
064         var mySubSet = from Person num in myArrayList where num.Age <= 40 && num.Address
                                == "陕西" select num;
065         foreach (var v in mySubSet)
066         {
067             v.Print();
068         }
069         Console.ReadKey();
070     }
071 }
072 }
```

这段代码中为自定义类又添加了姓名和地址属性。在第 64 行的查询表达式中进行筛选的时候，使用两种条件联合进行筛选，只有符合年龄小于等于 40 并且地址为“陕西”的数据才会被当做结果返回。这段代码的执行结果如下：

姓名：令狐冲，年龄：23，地址：陕西

姓名：任盈盈，年龄：18，地址：陕西

22.4.1 where 子句

where 子句就是负责查询表达式的筛选操作。如果没有 where 语句的存在，则没有一切筛选条件，会返回集合中的所有成员。下面是 where 子句的用法：

```
001 using System;
002 using System.Collections;
003 using System.Collections.Generic;
004 using System.Linq;
005 namespace LinQ1
006 {
007     class Person:IComparable
008     {
009         public string Name
010         {
011             get;
012             set;
013         }
014         public int Age
015         {
016             get;
017             set;
018         }
019         public string Address
020         {
021             get;
022             set;
023         }
024         public Person(string name,int age,string address)
025         {
026             this.Name = name;
```

```
027         this.Age = age;
028         this.Address = address;
029     }
030     public void Print()
031     {
032         Console.WriteLine("姓名: {0}, 年龄: {1}, 地址: {2}", Name, Age, Address);
033     }
034     public override string ToString()
035     {
036         string s = string.Format("年龄是: {0}", Age);
037         return s;
038     }
039     public int CompareTo(object obj)
040     {
041         if (this.Age < ((Person)obj).Age)
042             return -1;
043         else if (this.Age == ((Person)obj).Age)
044             return 0;
045         else
046             return 1;
047     }
048 }
049 class Program
050 {
051     static void Main(string[] args)
052     {
053         ArrayList myArrayList = new ArrayList();
054         Person p1 = new Person("云飞扬", 50, "陕西");
055         Person p2 = new Person("令狐冲", 23, "陕西");
056         Person p3 = new Person("任盈盈", 18, "陕西");
057         Person p4 = new Person("向问天", 40, "河北");
058         Person p5 = new Person("陆大有", 20, "河北");
059         myArrayList.Add(p1);
060         myArrayList.Add(p2);
061         myArrayList.Add(p3);
062         myArrayList.Add(p4);
063         myArrayList.Add(p5);
064         Console.WriteLine("如果没有 where 子句, 查询表达式也可以执行");
065         var mySubSet = from Person num in myArrayList select num;
066         foreach (var v in mySubSet)
067         {
068             v.Print();
069         }
070         Console.WriteLine("where 子句还可以调用方法");
```

```

071         var mySubSet1 = from Person num in myArrayList where Test(num.Age) select num;
072         foreach (var v in mySubSet1)
073         {
074             v.Print();
075         }
076         Console.ReadKey();
077     }
078     static bool Test(int i)
079     {
080         if (i % 10 == 0)
081             return true;
082         else
083             return false;
084     }
085 }
086 }

```

在代码的第 65 行演示的是没有 where 子句的用法，因为有了筛选方法，所以集合中的所有成员都查询了出来。where 子句还可以调用方法。第 78 行定义了一个静态方法，它用参数对 10 求模，只有值为 0 的才返回 true，否则返回 false。第 71 行代码的 where 子句调用了这个方法，为它传入 Person 类的 Age 属性。这就保证了只有 Age 属性是 10 的整数倍的成员才能查询出来。代码的执行结果如下：

如果没有 where 子句，查询表达式也可以执行

```

姓名：云飞扬，年龄：50，地址：陕西
姓名：令狐冲，年龄：23，地址：陕西
姓名：任盈盈，年龄：18，地址：陕西
姓名：向问天，年龄：40，地址：河北
姓名：陆大有，年龄：20，地址：河北

```

where 子句还可以调用方法

```

姓名：云飞扬，年龄：50，地址：陕西
姓名：向问天，年龄：40，地址：河北
姓名：陆大有，年龄：20，地址：河北

```

22.5 排序

如果要对查询表达式返回的序列进行排序，则可以使用 orderby 子句。下面看一下它的简单用法。

```

001 using System;
002 using System.Collections;
003 using System.Collections.Generic;
004 using System.Linq;
005 namespace Linq1
006 {
007     class Person:IComparable
008     {
009         public string Name
010         {
011             get;
012             set;

```

```
013     }
014     public int Age
015     {
016         get;
017         set;
018     }
019     public string Address
020     {
021         get;
022         set;
023     }
024     public Person(string name, int age, string address)
025     {
026         this.Name = name;
027         this.Age = age;
028         this.Address = address;
029     }
030     public void Print()
031     {
032         Console.WriteLine("姓名: {0}, 年龄: {1}, 地址: {2}", Name, Age, Address);
033     }
034     public override string ToString()
035     {
036         string s = string.Format("年龄是: {0}", Age);
037         return s;
038     }
039     public int CompareTo(object obj)
040     {
041         if (this.Age < ((Person)obj).Age)
042             return -1;
043         else if (this.Age == ((Person)obj).Age)
044             return 0;
045         else
046             return 1;
047     }
048 }
049 class Program
050 {
051     static void Main(string[] args)
052     {
053         ArrayList myArrayList = new ArrayList();
054         Person p1 = new Person("云飞扬", 50, "陕西");
055         Person p2 = new Person("令狐冲", 23, "陕西");
056         Person p3 = new Person("任盈盈", 18, "陕西");
```

```

057         Person p4 = new Person("向问天", 40, "河北");
058         Person p5 = new Person("陆大有", 20, "河北");
059         myArrayList.Add(p1);
060         myArrayList.Add(p2);
061         myArrayList.Add(p3);
062         myArrayList.Add(p4);
063         myArrayList.Add(p5);
064         var mySubSet = from Person num in myArrayList where num.Age < 60 orderby num.Age
                        select num;
065         foreach (var v in mySubSet)
066         {
067             v.Print();
068         }
069         Console.ReadKey();
070     }
071 }
072 }

```

在这段代码中第 64 行使用 orderby 子句为返回的结果进行了排序，orderby 子句可以放在 where 子句后面。orderby 后紧跟着要按照什么数据进行排序。这段代码中是按照 Person 类的 Age 属性进行排序。如果在 num.Age 后什么关键字也不跟，则默认按照升序排列。代码的执行结果如下：

```

姓名：任盈盈，年龄：18，地址：陕西
姓名：陆大有，年龄：20，地址：河北
姓名：令狐冲，年龄：23，地址：陕西
姓名：向问天，年龄：40，地址：河北
姓名：云飞扬，年龄：50，地址：陕西

```

可以看到，返回的结果集按照年龄进行了升序排列。

22.5.1 orderby 子句

orderby 子句还可以接 ascending 关键字，意思是按照指定的元素以升序排列；如果接 descending 关键字，则是按照指定的元素以降序排列。使用 orderby 子句还可以指定主要排序和次要排序。下面是代码实例：

```

001     using System;
002     using System.Collections;
003     using System.Collections.Generic;
004     using System.Linq;
005     namespace Linq1
006     {
007         class Person:IComparable
008         {
009             public string Name
010             {
011                 get;
012                 set;
013             }
014             public int Age

```

```
015         {
016             get;
017             set;
018         }
019         public string Address
020         {
021             get;
022             set;
023         }
024         public Person(string name,int age,string address)
025         {
026             this.Name = name;
027             this.Age = age;
028             this.Address = address;
029         }
030         public void Print()
031         {
032             Console.WriteLine("姓名: {0}, 年龄: {1}, 地址: {2}", Name, Age, Address);
033         }
034         public override string ToString()
035         {
036             string s = string.Format("年龄是: {0}", Age);
037             return s;
038         }
039         public int CompareTo(object obj)
040         {
041             if (this.Age < ((Person)obj).Age)
042                 return -1;
043             else if (this.Age == ((Person)obj).Age)
044                 return 0;
045             else
046                 return 1;
047         }
048     }
049     class Program
050     {
051         static void Main(string[] args)
052         {
053             ArrayList myArrayList = new ArrayList();
054             Person p1 = new Person("云飞扬", 50, "陕西");
055             Person p2 = new Person("令狐冲", 23, "陕西");
056             Person p3 = new Person("任盈盈", 18, "陕西");
057             Person p4 = new Person("向问天", 40, "河北");
058             Person p5 = new Person("陆大有", 20, "河北");
```



```
059         Person p6 = new Person("任盈盈", 50, "陕西");
060         Person p7 = new Person("陆大有", 45, "浙江");
061         myArrayList.Add(p1);
062         myArrayList.Add(p2);
063         myArrayList.Add(p3);
064         myArrayList.Add(p4);
065         myArrayList.Add(p5);
066         myArrayList.Add(p6);
067         myArrayList.Add(p7);
068         Console.WriteLine("现在将查询到的结果按照年龄升序排列");
069         var mySubSet = from Person num in myArrayList where num.Age < 60 orderby
                        num.Age ascending select num;
070         foreach (var v in mySubSet)
071         {
072             v.Print();
073         }
074         Console.WriteLine("现在将查询到的结果按照年龄降序排列");
075         var mySubSet1 = from Person num in myArrayList where num.Age < 60 orderby
                        num.Age descending select num;
076         foreach (var v in mySubSet1)
077         {
078             v.Print();
079         }
080         Console.WriteLine("现在将姓名设为主要排序(升序), 将年龄设为次要排序(降序)");
081         var mySubSet2 = from Person num in myArrayList where num.Age < 60 orderby
                        num.Name ascending, num.Age descending select num;
082         foreach (var v in mySubSet2)
083         {
084             v.Print();
085         }
086         Console.WriteLine("现在按照 Person 实例排序");
087         var mySubSet3 = from Person num in myArrayList where num.Age < 60 orderby num
                        ascending select num;
088         foreach (var v in mySubSet3)
089         {
090             v.Print();
091         }
092         Console.ReadKey();
093     }
094 }
095 }
```

首先为集合再添加两个成员, 成员的名称分别是任盈盈和陆大有, 但是和集合中原有的成员年龄有区别。这是假设现实中有同名成员。第 69 行的查询表达式按照年龄的升序排序, 语法是 orderby 语句后跟成

员的 Age 属性，然后接 ascending 关键字；第 75 行按照 Age 属性的降序排序，它的语法和升序排序相似，只不过 ascending 关键字换成了 descending 关键字。第 80 行的表达式语句使用了主要排序和次要排序。它的作用是，如果主要排序中有重复，则按照次要排序执行排序。主要排序和次要排序的使用方法就是将主要排序写在前面，然后用逗号分隔，后面再跟一个排序，就是次要排序。第 87 行使用成员本身进行了排序。如果要按照自定义类型进行排序，那么这个自定义类型必须实现了 IComparable 接口，也就是说，自定义类的两个实例互相之间应该有比较的方法。这段代码的执行结果如下：

现在将查询到的结果按照年龄升序排列

姓名：任盈盈，年龄：18，地址：陕西

姓名：陆大有，年龄：20，地址：河北

姓名：令狐冲，年龄：23，地址：陕西

姓名：向问天，年龄：40，地址：河北

姓名：陆大有，年龄：45，地址：浙江

姓名：云飞扬，年龄：50，地址：陕西

姓名：任盈盈，年龄：50，地址：陕西

现在将查询到的结果按照年龄降序排列

姓名：云飞扬，年龄：50，地址：陕西

姓名：任盈盈，年龄：50，地址：陕西

姓名：陆大有，年龄：45，地址：浙江

姓名：向问天，年龄：40，地址：河北

姓名：令狐冲，年龄：23，地址：陕西

姓名：陆大有，年龄：20，地址：河北

姓名：任盈盈，年龄：18，地址：陕西

现在将姓名设为主要排序(升序)，将年龄设为次要排序(降序)

姓名：令狐冲，年龄：23，地址：陕西

姓名：陆大有，年龄：45，地址：浙江

姓名：陆大有，年龄：20，地址：河北

姓名：任盈盈，年龄：50，地址：陕西

姓名：任盈盈，年龄：18，地址：陕西

姓名：向问天，年龄：40，地址：河北

姓名：云飞扬，年龄：50，地址：陕西

现在按照 Person 实例排序

姓名：任盈盈，年龄：18，地址：陕西

姓名：陆大有，年龄：20，地址：河北

姓名：令狐冲，年龄：23，地址：陕西

姓名：向问天，年龄：40，地址：河北

姓名：陆大有，年龄：45，地址：浙江

姓名：云飞扬，年龄：50，地址：陕西

姓名：任盈盈，年龄：50，地址：陕西

22.6 分组

使用 group 子句可以对查询到的结果进行分组。使用 group 之后，返回结果是一个 IGrouping< TKey, TElement>类型的实例。下面是代码实例：

```
001    using System;
002    using System.Collections;
003    using System.Collections.Generic;
```

```
004 using System.Linq;
005 namespace LinQ1
006 {
007     class Person:IComparable
008     {
009         public string Name
010         {
011             get;
012             set;
013         }
014         public int Age
015         {
016             get;
017             set;
018         }
019         public string Address
020         {
021             get;
022             set;
023         }
024         public Person(string name,int age,string address)
025         {
026             this.Name = name;
027             this.Age = age;
028             this.Address = address;
029         }
030         public void Print()
031         {
032             Console.WriteLine("姓名: {0}, 年龄: {1}, 地址: {2}",Name, Age, Address);
033         }
034         public override string ToString()
035         {
036             string s = string.Format("年龄是: {0}", Age);
037             return s;
038         }
039         public int CompareTo(object obj)
040         {
041             if (this.Age < ((Person)obj).Age)
042                 return -1;
043             else if (this.Age == ((Person)obj).Age)
044                 return 0;
045             else
046                 return 1;
047         }
048     }
049 }
```

```
048     }
049     class Program
050     {
051         static void Main(string[] args)
052         {
053             ArrayList myArrayList = new ArrayList();
054             Person p1 = new Person("云飞扬", 50, "陕西");
055             Person p2 = new Person("令狐冲", 23, "陕西");
056             Person p3 = new Person("任盈盈", 18, "陕西");
057             Person p4 = new Person("向问天", 40, "河北");
058             Person p5 = new Person("陆大有", 20, "河北");
059             Person p6 = new Person("任盈盈", 50, "陕西");
060             Person p7 = new Person("陆大有", 45, "浙江");
061             myArrayList.Add(p1);
062             myArrayList.Add(p2);
063             myArrayList.Add(p3);
064             myArrayList.Add(p4);
065             myArrayList.Add(p5);
066             myArrayList.Add(p6);
067             myArrayList.Add(p7);
068             var mySubSet = from Person num in myArrayList group num by num.Address;
069             foreach (var v in mySubSet)
070             {
071                 Console.WriteLine(v.Key);
072                 foreach (var vl in v)
073                 {
074                     vl.Print();
075                 }
076             }
077             Console.ReadKey();
078         }
079     }
080 }
```

使用 group 分组的时候, 首先用 group 关键字修饰范围变量, 然后接 by 关键字, 它指定使用哪个元素作为分组的依据。返回的是一个 IGrouping<TKey, TElement>类型的实例, 它是一个键值对。键就是分组的凭据, 值就是实际的集合成员。因此遍历查询表达式的返回值需要嵌套的 foreach 语句, 外层的 foreach 语句遍历 IGrouping<TKey, TElement>类型的键; 内层的 foreach 语句遍历值所对应的的的成员。代码的执行结果如下:

陕西

姓名: 云飞扬, 年龄: 50, 地址: 陕西

姓名: 令狐冲, 年龄: 23, 地址: 陕西

姓名: 任盈盈, 年龄: 18, 地址: 陕西

姓名: 任盈盈, 年龄: 50, 地址: 陕西

河北

姓名: 向问天, 年龄: 40, 地址: 河北

姓名: 陆大有, 年龄: 20, 地址: 河北

浙江

姓名: 陆大有, 年龄: 45, 地址: 浙江

22.6.1 group 子句

使用 group 子句之后, 还可以使用 into 关键字将返回的结果赋予一个临时的标识符。然后再针对这个标识符进行继续的查询。代码如下:

```
001    using System;
002    using System.Collections;
003    using System.Collections.Generic;
004    using System.Linq;
005    namespace LinQ1
006    {
007        class Person:IComparable
008        {
009            public string Name
010            {
011                get;
012                set;
013            }
014            public int Age
015            {
016                get;
017                set;
018            }
019            public string Address
020            {
021                get;
022                set;
023            }
024            public Person(string name,int age,string address)
025            {
026                this.Name = name;
027                this.Age = age;
028                this.Address = address;
029            }
030            public void Print()
031            {
032                Console.WriteLine("姓名: {0}, 年龄: {1}, 地址: {2}",Name, Age, Address);
033            }
034            public override string ToString()
035            {
036                string s = string.Format("年龄是: {0}", Age);
037                return s;
```

```
038     }
039     public int CompareTo(object obj)
040     {
041         if (this.Age < ((Person)obj).Age)
042             return -1;
043         else if (this.Age == ((Person)obj).Age)
044             return 0;
045         else
046             return 1;
047     }
048 }
049 class Program
050 {
051     static void Main(string[] args)
052     {
053         ArrayList myArrayList = new ArrayList();
054         Person p1 = new Person("云飞扬", 50, "贵州");
055         Person p2 = new Person("令狐冲", 23, "云南");
056         Person p3 = new Person("任盈盈", 18, "陕西");
057         Person p4 = new Person("向问天", 40, "河北");
058         Person p5 = new Person("陆大有", 20, "河北");
059         Person p6 = new Person("任盈盈", 50, "陕西");
060         Person p7 = new Person("陆大有", 45, "浙江");
061         myArrayList.Add(p1);
062         myArrayList.Add(p2);
063         myArrayList.Add(p3);
064         myArrayList.Add(p4);
065         myArrayList.Add(p5);
066         myArrayList.Add(p6);
067         myArrayList.Add(p7);
068         var mySubSet = from Person num in myArrayList group num by num.Address into
069             g orderby g.Key ascending select g;
070         foreach (var v in mySubSet)
071         {
072             Console.WriteLine(v.Key);
073             foreach (var v1 in v)
074             {
075                 v1.Print();
076             }
077         }
078         Console.ReadKey();
079     }
080 }
```

在代码的第 68 行，首先按照 Person 类的成员的地址进行分组。接下来使用 into 语句将结果赋予一个新的标识符 g，然后对于这个 g 进行排序，排序的凭据是前面返回的结果键值对中的键，也就是成员地址。排序顺序是升序。最后是 select g 结尾，意思是针对 g 查询。代码的执行结果如下：

```
贵州
姓名: 云飞扬, 年龄: 50, 地址: 贵州
河北
姓名: 向问天, 年龄: 40, 地址: 河北
姓名: 陆大有, 年龄: 20, 地址: 河北
陕西
姓名: 任盈盈, 年龄: 18, 地址: 陕西
姓名: 任盈盈, 年龄: 50, 地址: 陕西
云南
姓名: 令狐冲, 年龄: 23, 地址: 云南
浙江
姓名: 陆大有, 年龄: 45, 地址: 浙江
```

还可以组合多个条件进行分组，下面是代码实例：

```
001    using System;
002    using System.Collections;
003    using System.Collections.Generic;
004    using System.Linq;
005    namespace Linq1
006    {
007        class Person:IComparable
008        {
009            public string Name
010            {
011                get;
012                set;
013            }
014            public int Age
015            {
016                get;
017                set;
018            }
019            public string Address
020            {
021                get;
022                set;
023            }
024            public Person(string name,int age,string address)
025            {
026                this.Name = name;
027                this.Age = age;
028                this.Address = address;
```

```
029     }
030     public void Print()
031     {
032         Console.WriteLine("姓名: {0}, 年龄: {1}, 地址: {2}", Name, Age, Address);
033     }
034     public override string ToString()
035     {
036         string s = string.Format("年龄是: {0}", Age);
037         return s;
038     }
039     public int CompareTo(object obj)
040     {
041         if (this.Age < ((Person)obj).Age)
042             return -1;
043         else if (this.Age == ((Person)obj).Age)
044             return 0;
045         else
046             return 1;
047     }
048 }
049 class Program
050 {
051     static void Main(string[] args)
052     {
053         ArrayList myArrayList = new ArrayList();
054         Person p1 = new Person("云飞扬", 50, "贵州");
055         Person p2 = new Person("令狐冲", 20, "云南");
056         Person p3 = new Person("任盈盈", 18, "陕西");
057         Person p4 = new Person("向问天", 40, "河北");
058         Person p5 = new Person("陆大有", 20, "河北");
059         Person p6 = new Person("任盈盈", 50, "陕西");
060         Person p7 = new Person("陆大有", 45, "河北");
061         myArrayList.Add(p1);
062         myArrayList.Add(p2);
063         myArrayList.Add(p3);
064         myArrayList.Add(p4);
065         myArrayList.Add(p5);
066         myArrayList.Add(p6);
067         myArrayList.Add(p7);
068         var mySubSet = from Person num in myArrayList group num by new { 姓名 =
                                num.Name, 地址 = num.Address };
069         foreach (var v in mySubSet)
070         {
071             Console.WriteLine(v.Key);
```



```
072         foreach (var vl in v)
073         {
074             vl.Print();
075         }
076         Console.WriteLine();
077     }
078     Console.ReadKey();
079 }
080 }
081 }
```

如果在 by 关键字之后使用 new 关键字，然后跟一对大括号，在里面定义多个集合成员的元素，则会按照这个组合的逻辑分组。代码的执行结果如下：

```
{ 姓名 = 云飞扬, 地址 = 贵州 }
姓名: 云飞扬, 年龄: 50, 地址: 贵州

{ 姓名 = 令狐冲, 地址 = 云南 }
姓名: 令狐冲, 年龄: 20, 地址: 云南

{ 姓名 = 任盈盈, 地址 = 陕西 }
姓名: 任盈盈, 年龄: 18, 地址: 陕西
姓名: 任盈盈, 年龄: 50, 地址: 陕西

{ 姓名 = 向问天, 地址 = 河北 }
姓名: 向问天, 年龄: 40, 地址: 河北

{ 姓名 = 陆大有, 地址 = 河北 }
姓名: 陆大有, 年龄: 20, 地址: 河北
姓名: 陆大有, 年龄: 45, 地址: 河北
```

22.7 联接

联接可以联合查询两个数据源，返回同时符合查询条件的结果集。下面是代码实例：

```
001 using System;
002 using System.Collections;
003 using System.Collections.Generic;
004 using System.Linq;
005 namespace LinQ1
006 {
007     class Person: IComparable
008     {
009         public string Name
010         {
011             get;
012             set;
013         }
014         public int Age
```

```
015         {
016             get;
017             set;
018         }
019     public string Address
020     {
021         get;
022         set;
023     }
024     public Person(string name,int age,string address)
025     {
026         this.Name = name;
027         this.Age = age;
028         this.Address = address;
029     }
030     public void Print()
031     {
032         Console.WriteLine("姓名: {0}, 年龄: {1}, 地址: {2}",Name, Age, Address);
033     }
034     public override string ToString()
035     {
036         string s = string.Format("年龄是: {0}", Age);
037         return s;
038     }
039     public int CompareTo(object obj)
040     {
041         if (this.Age < ((Person)obj).Age)
042             return -1;
043         else if (this.Age == ((Person)obj).Age)
044             return 0;
045         else
046             return 1;
047     }
048 }
049 class Work
050 {
051     public string Job;
052     public int Times;
053     public string Address;
054     public Work(string job, int times,string address)
055     {
056         Job = job;
057         Times = times;
058         Address = address;
```

```
059     }
060     public void Print()
061     {
062         Console.WriteLine("职务: {0}, 工龄: {1}, 地址: {2}", Job, Times, Address);
063     }
064 }
065 class Program
066 {
067     static void Main(string[] args)
068     {
069         ArrayList myArrayList = new ArrayList();
070         Person p1 = new Person("云飞扬", 50, "贵州");
071         Person p2 = new Person("令狐冲", 20, "云南");
072         Person p3 = new Person("任盈盈", 18, "陕西");
073         Person p4 = new Person("向问天", 40, "河北");
074         Person p5 = new Person("陆大有", 20, "河北");
075         Person p6 = new Person("任盈盈", 50, "陕西");
076         Person p7 = new Person("陆大有", 45, "河北");
077         myArrayList.Add(p1);
078         myArrayList.Add(p2);
079         myArrayList.Add(p3);
080         myArrayList.Add(p4);
081         myArrayList.Add(p5);
082         myArrayList.Add(p6);
083         myArrayList.Add(p7);
084         ArrayList myArrayList1 = new ArrayList();
085         Work w1 = new Work("侠客", 10, "河北");
086         Work w2 = new Work("侠客", 20, "陕西");
087         Work w3 = new Work("侠客", 20, "贵州");
088         Work w4 = new Work("侠客", 30, "云南");
089         myArrayList1.Add(w1);
090         myArrayList1.Add(w2);
091         myArrayList1.Add(w3);
092         myArrayList1.Add(w4);
093         var mySubSet = from Person num in myArrayList join Work work in myArrayList1
094             on num.Address equals work.Address select new {姓名 = num.Name, 工作 =
095             work.Job };
096         foreach (var v in mySubSet)
097         {
098             Console.WriteLine("{0}—{1}", v.姓名, v.工作);
099         }
100         Console.ReadKey();
101     }
102 }
```

```
101 }
```

因为联接查询需要使用两个类，所以在前面代码的基础上又定义了一个类 Work，在其中又定义了一个 Address 属性用来联接查询。第 89 行到第 92 行将类 Work 的实例依次添加进入了集合。第 93 行是联接查询表达式，它的类别是内部联接，内部联接指的是对于第一个集合内的成员，如果第二个集合中有一个匹配项，则该元素就会在结果集中出现一次。

查询表达式中的“from Person num in myArrayList”指定了第一个数据源；第二个数据源用 join 子句指定，“join Work work in myArrayList1”子句又指定了一个范围变量 work。“on num.Address equals work.Address”子句指定了第一个集合中哪个项和第二个集合中的哪个项进行匹配，匹配的关键字是 equals。最后的“select new {姓名 = num.Name, 工作 = work.Job}”子句指定了 foreach 语句中的迭代变量（匿名类实例）可以引用的数据。select 语句返回的是一个匿名类。代码的执行结果如下：

```
云飞扬--侠客
令狐冲--侠客
任盈盈--侠客
向问天--侠客
陆大有--侠客
任盈盈--侠客
陆大有--侠客
```

22.7.1 join 子句

使用复合键联接可以用多个属性来比较元素。下面是代码实例：

```
001 using System;
002 using System.Collections;
003 using System.Collections.Generic;
004 using System.Linq;
005 namespace Linq1
006 {
007     class Person:IComparable
008     {
009         public string Name
010         {
011             get;
012             set;
013         }
014         public int Age
015         {
016             get;
017             set;
018         }
019         public string Address
020         {
021             get;
022             set;
023         }
024         public Person(string name,int age,string address)
025         {
```

```
026         this.Name = name;
027         this.Age = age;
028         this.Address = address;
029     }
030     public void Print()
031     {
032         Console.WriteLine("姓名: {0}, 年龄: {1}, 地址: {2}", Name, Age, Address);
033     }
034     public override string ToString()
035     {
036         string s = string.Format("年龄是: {0}", Age);
037         return s;
038     }
039     public int CompareTo(object obj)
040     {
041         if (this.Age < ((Person)obj).Age)
042             return -1;
043         else if (this.Age == ((Person)obj).Age)
044             return 0;
045         else
046             return 1;
047     }
048 }
049 class Work
050 {
051     public string Name
052     {
053         get;
054         set;
055     }
056     public int Times
057     {
058         get;
059         set;
060     }
061     public string Address
062     {
063         get;
064         set;
065     }
066     public Work(string name, int times, string address)
067     {
068         Name = name;
069         Times = times;
```

```
070         Address = address;
071     }
072     public void Print()
073     {
074         Console.WriteLine("姓名: {0}, 工龄: {1}, 地址: {2}", Name, Times, Address);
075     }
076 }
077 class Program
078 {
079     static void Main(string[] args)
080     {
081         ArrayList myArrayList = new ArrayList();
082         Person p1 = new Person("云飞扬", 50, "贵州");
083         Person p2 = new Person("令狐冲", 20, "云南");
084         Person p3 = new Person("任盈盈", 20, "陕西");
085         Person p4 = new Person("向问天", 10, "河北");
086         Person p5 = new Person("陆大有", 20, "河北");
087         Person p6 = new Person("任盈盈", 50, "陕西");
088         Person p7 = new Person("陆大有", 45, "河北");
089         myArrayList.Add(p1);
090         myArrayList.Add(p2);
091         myArrayList.Add(p3);
092         myArrayList.Add(p4);
093         myArrayList.Add(p5);
094         myArrayList.Add(p6);
095         myArrayList.Add(p7);
096         ArrayList myArrayList1 = new ArrayList();
097         Work w1 = new Work("侠客甲", 10, "河北");
098         Work w2 = new Work("任盈盈", 20, "陕西");
099         Work w3 = new Work("云飞扬", 20, "贵州");
100         Work w4 = new Work("令狐冲", 30, "云南");
101         myArrayList1.Add(w1);
102         myArrayList1.Add(w2);
103         myArrayList1.Add(w3);
104         myArrayList1.Add(w4);
105         var mySubSet = from Person num in myArrayList join Work work in myArrayList1
106             on new {num.Name,num.Address } equals new {work.Name,work.Address } select
107             num.Name + " " + num.Address;
108         foreach (var v in mySubSet)
109         {
110             Console.WriteLine(v);
111         }
112         Console.ReadKey();
113     }
114 }
```

```
112     }  
113 }
```

使用复合键联接查询的时候，第一个集合的选取的元素属性必须相同，它和第二个集合中选取的元素属性也要有对应关系。最后的 select 语句返回字符串的组合。代码的执行结果如下：

```
云飞扬 贵州  
令狐冲 云南  
任盈盈 陕西  
任盈盈 陕西
```

含有 into 表达式的 join 子句称为分组联接。下面是代码实例：

```
001 using System;  
002 using System.Collections;  
003 using System.Collections.Generic;  
004 using System.Linq;  
005 namespace LinQ1  
006 {  
007     class Person: IComparable  
008     {  
009         public string Name  
010         {  
011             get;  
012             set;  
013         }  
014         public int Age  
015         {  
016             get;  
017             set;  
018         }  
019         public string Address  
020         {  
021             get;  
022             set;  
023         }  
024         public Person(string name,int age,string address)  
025         {  
026             this.Name = name;  
027             this.Age = age;  
028             this.Address = address;  
029         }  
030         public void Print()  
031         {  
032             Console.WriteLine("姓名: {0}, 年龄: {1}, 地址: {2}",Name, Age, Address);  
033         }  
034         public override string ToString()  
035         {
```

```
036         string s = string.Format("年龄是: {0}", Age);
037         return s;
038     }
039     public int CompareTo(object obj)
040     {
041         if (this.Age < ((Person)obj).Age)
042             return -1;
043         else if (this.Age == ((Person)obj).Age)
044             return 0;
045         else
046             return 1;
047     }
048 }
049 class Work
050 {
051     public string Job
052     {
053         get;
054         set;
055     }
056     public int Times
057     {
058         get;
059         set;
060     }
061     public string Address
062     {
063         get;
064         set;
065     }
066     public Work(string job, int times, string address)
067     {
068         Job = job;
069         Times = times;
070         Address = address;
071     }
072     public void Print()
073     {
074         Console.WriteLine("工作: {0}, 工龄: {1}, 地址: {2}", Job, Times, Address);
075     }
076 }
077 class Program
078 {
079     static void Main(string[] args)
```



```

080      {
081          List<Person> myList = new List<Person>();
082          Person p1 = new Person("云飞扬", 50, "贵州");
083          Person p2 = new Person("令狐冲", 20, "云南");
084          Person p3 = new Person("任盈盈", 20, "陕西");
085          Person p4 = new Person("向问天", 10, "河北");
086          Person p5 = new Person("陆大有", 20, "河北");
087          Person p6 = new Person("任盈盈", 50, "陕西");
088          Person p7 = new Person("陆大有", 45, "河北");
089          myList.Add(p1);
090          myList.Add(p2);
091          myList.Add(p3);
092          myList.Add(p4);
093          myList.Add(p5);
094          myList.Add(p6);
095          myList.Add(p7);
096          List<Work> myList1 = new List<Work>();
097          Work w1 = new Work("侠客", 10, "河北");
098          Work w2 = new Work("侠客", 20, "陕西");
099          Work w3 = new Work("侠客", 20, "贵州");
100          Work w4 = new Work("侠客", 30, "云南");
101          myList1.Add(w1);
102          myList1.Add(w2);
103          myList1.Add(w3);
104          myList1.Add(w4);
105          var mySubSet = from num in myList join work in myList1 on num.Address equals
                           work.Address into g from num1 in g where num1.Times < 30 select num1;
106          foreach (var v in mySubSet)
107          {
108              v.print();
109          }
110          Console.ReadKey();
111      }
112  }
113 }

```

这段代码使用了 List<T>集合装载数据，这样在查询表达式中就不用再指明范围变量的类型，从而简化了编码。分组联接可以在查询到的结果集中再次执行查询。第 105 行代码中的 into 关键字定义的标识符中，还可以再次使用 where 子句选择符合第二个集合查询条件的结果集。

分组联接查询到的结果集是第二个集合中的元素，代码的执行结果如下：

```

工作：侠客，工龄：20，地址：贵州
工作：侠客，工龄：20，地址：陕西
工作：侠客，工龄：10，地址：河北
工作：侠客，工龄：10，地址：河北
工作：侠客，工龄：20，地址：陕西

```

工作：侠客，工龄：10，地址：河北

22.8 投影 (select 子句)

通常情况下，select 子句只表示范围变量，一个查询表达式必须以 select 子句结尾或 group 子句结尾。select 子句还提供了一种功能强大的机制，可用于将源数据转换（或投影）为新类型。下面看一下如何使用 select 将多个输入序列联接到一个输出字符串序列。

```
001    using System;
002    using System.Collections;
003    using System.Collections.Generic;
004    using System.Linq;
005    namespace LinQ1
006    {
007        class Person:IComparable
008        {
009            public string Name
010            {
011                get;
012                set;
013            }
014            public int Age
015            {
016                get;
017                set;
018            }
019            public string Address
020            {
021                get;
022                set;
023            }
024            public Person(string name,int age,string address)
025            {
026                this.Name = name;
027                this.Age = age;
028                this.Address = address;
029            }
030            public void Print()
031            {
032                Console.WriteLine("姓名: {0}, 年龄: {1}, 地址: {2}",Name, Age, Address);
033            }
034            public override string ToString()
035            {
036                string s = string.Format("年龄是: {0}", Age);
037                return s;
038            }
039            public int CompareTo(object obj)
```

```
040     {
041         if (this.Age < ((Person)obj).Age)
042             return -1;
043         else if (this.Age == ((Person)obj).Age)
044             return 0;
045         else
046             return 1;
047     }
048 }
049 class Work
050 {
051     public string Job
052     {
053         get;
054         set;
055     }
056     public int Times
057     {
058         get;
059         set;
060     }
061     public string Address
062     {
063         get;
064         set;
065     }
066     public Work(string job, int times, string address)
067     {
068         Job = job;
069         Times = times;
070         Address = address;
071     }
072     public void Print()
073     {
074         Console.WriteLine("工作: {0}, 工龄: {1}, 地址: {2}", Job, Times, Address);
075     }
076 }
077 class Program
078 {
079     static void Main(string[] args)
080     {
081         List<Person> myList = new List<Person>();
082         Person p1 = new Person("云飞扬", 50, "贵州");
083         Person p2 = new Person("令狐冲", 20, "云南");
```

```
084         Person p3 = new Person("任盈盈", 20, "陕西");
085         Person p4 = new Person("向问天", 10, "河北");
086         Person p5 = new Person("陆大有", 20, "河北");
087         Person p6 = new Person("任盈盈", 50, "陕西");
088         Person p7 = new Person("陆大有", 45, "河北");
089         myList.Add(p1);
090         myList.Add(p2);
091         myList.Add(p3);
092         myList.Add(p4);
093         myList.Add(p5);
094         myList.Add(p6);
095         myList.Add(p7);
096         List<Work> myList1 = new List<Work>();
097         Work w1 = new Work("侠客", 10, "河北");
098         Work w2 = new Work("侠客", 20, "陕西");
099         Work w3 = new Work("侠客", 20, "贵州");
100         Work w4 = new Work("侠客", 30, "云南");
101         myList1.Add(w1);
102         myList1.Add(w2);
103         myList1.Add(w3);
104         myList1.Add(w4);
105         var mySubSet = (from num in myList select num.Name).Concat(from work in
                             myList1 select work.Job);
106         foreach (var v in mySubSet)
107         {
108             Console.WriteLine(v.ToString());
109         }
110         Console.ReadKey();
111     }
112 }
113 }
```

使用“.”运算符可以更具体的选择集合成员的子成员。第 105 行的查询表达式把两个数据源的范围变量的子成员联接成一个新的输出字符串序列。联接的方法是两个子查询用“.”运算符接 Concat 关键字联合起来。代码的执行结果如下：

```
云飞扬
令狐冲
任盈盈
向问天
陆大有
任盈盈
陆大有
侠客
侠客
侠客
```

侠客

使用 `select` 还可以创建包含集合成员多个属性的匿名类型，这个语法前面已经接触过了，就是使用 `new` 关键字。代码如下：

```
001    using System;
002    using System.Collections;
003    using System.Collections.Generic;
004    using System.Linq;
005    namespace LinQ1
006    {
007        class Person:IComparable
008        {
009            public string Name
010            {
011                get;
012                set;
013            }
014            public int Age
015            {
016                get;
017                set;
018            }
019            public string Address
020            {
021                get;
022                set;
023            }
024            public Person(string name,int age,string address)
025            {
026                this.Name = name;
027                this.Age = age;
028                this.Address = address;
029            }
030            public void Print()
031            {
032                Console.WriteLine("姓名: {0}, 年龄: {1}, 地址: {2}",Name, Age, Address);
033            }
034            public override string ToString()
035            {
036                string s = string.Format("年龄是: {0}", Age);
037                return s;
038            }
039            public int CompareTo(object obj)
040            {
041                if (this.Age < ((Person)obj).Age)
```

```
042         return -1;
043     else if (this.Age == ((Person)obj).Age)
044         return 0;
045     else
046         return 1;
047     }
048 }
049 class Work
050 {
051     public string Job
052     {
053         get;
054         set;
055     }
056     public int Times
057     {
058         get;
059         set;
060     }
061     public string Address
062     {
063         get;
064         set;
065     }
066     public Work(string job, int times, string address)
067     {
068         Job = job;
069         Times = times;
070         Address = address;
071     }
072     public void Print()
073     {
074         Console.WriteLine("工作: {0}, 工龄: {1}, 地址: {2}", Job, Times, Address);
075     }
076 }
077 class Program
078 {
079     static void Main(string[] args)
080     {
081         List<Person> myList = new List<Person>();
082         Person p1 = new Person("云飞扬", 50, "贵州");
083         Person p2 = new Person("令狐冲", 20, "云南");
084         Person p3 = new Person("任盈盈", 20, "陕西");
085         Person p4 = new Person("向问天", 10, "河北");
```

```

086         Person p5 = new Person("陆大有", 20, "河北");
087         Person p6 = new Person("任盈盈", 50, "陕西");
088         Person p7 = new Person("陆大有", 45, "河北");
089         myList.Add(p1);
090         myList.Add(p2);
091         myList.Add(p3);
092         myList.Add(p4);
093         myList.Add(p5);
094         myList.Add(p6);
095         myList.Add(p7);
096         List<Work> myList1 = new List<Work>();
097         Work w1 = new Work("侠客", 10, "河北");
098         Work w2 = new Work("侠客", 20, "陕西");
099         Work w3 = new Work("侠客", 20, "贵州");
100         Work w4 = new Work("侠客", 30, "云南");
101         myList1.Add(w1);
102         myList1.Add(w2);
103         myList1.Add(w3);
104         myList1.Add(w4);
105         var mySubSet = from num in myList select new { 姓名 = num.Name, 地址 =
                        num.Address };
106         foreach (var v in mySubSet)
107         {
108             Console.WriteLine(v.姓名 + " " + v.地址);
109         }
110         Console.ReadKey();
111     }
112 }
113 }

```

这段代码使用 new 关键字创建了包含成员 Name 属性和 Address 属性的新匿名类型, 在 foreach 语句中可以访问这两个匿名类的属性。其中推断类型 v 就是匿名类的一个实例。代码的执行结果如下:

```

云飞扬 贵州
令狐冲 云南
任盈盈 陕西
向问天 河北
陆大有 河北
任盈盈 陕西
陆大有 河北

```

22.9 聚合

查询表达式还可以对结果集进行聚合操作。下面是代码实例:

```

001     using System;
002     using System.Collections;
003     using System.Collections.Generic;
004     using System.Linq;

```

```
005 namespace LinQ1
006 {
007     class Person:Comparable
008     {
009         public string Name
010         {
011             get;
012             set;
013         }
014         public int Age
015         {
016             get;
017             set;
018         }
019         public string Address
020         {
021             get;
022             set;
023         }
024         public Person(string name,int age,string address)
025         {
026             this.Name = name;
027             this.Age = age;
028             this.Address = address;
029         }
030         public void Print()
031         {
032             Console.WriteLine("姓名: {0}, 年龄: {1}, 地址: {2}",Name, Age, Address);
033         }
034         public override string ToString()
035         {
036             string s = string.Format("年龄是: {0}", Age);
037             return s;
038         }
039         public int CompareTo(object obj)
040         {
041             if (this.Age < ((Person)obj).Age)
042                 return -1;
043             else if (this.Age == ((Person)obj).Age)
044                 return 0;
045             else
046                 return 1;
047         }
048     }
```



```
049     class Work
050     {
051         public string Job
052         {
053             get;
054             set;
055         }
056         public int Times
057         {
058             get;
059             set;
060         }
061         public string Address
062         {
063             get;
064             set;
065         }
066         public Work(string job, int times, string address)
067         {
068             Job = job;
069             Times = times;
070             Address = address;
071         }
072         public void Print()
073         {
074             Console.WriteLine("工作: {0}, 工龄: {1}, 地址: {2}", Job, Times, Address);
075         }
076     }
077     class Program
078     {
079         static void Main(string[] args)
080         {
081             List<Person> myList = new List<Person>();
082             Person p1 = new Person("云飞扬", 50, "贵州");
083             Person p2 = new Person("令狐冲", 20, "云南");
084             Person p3 = new Person("任盈盈", 20, "陕西");
085             Person p4 = new Person("向问天", 10, "河北");
086             Person p5 = new Person("陆大有", 20, "河北");
087             Person p6 = new Person("任盈盈", 50, "陕西");
088             Person p7 = new Person("陆大有", 45, "河北");
089             myList.Add(p1);
090             myList.Add(p2);
091             myList.Add(p3);
092             myList.Add(p4);
```

```
093         myList.Add(p5);
094         myList.Add(p6);
095         myList.Add(p7);
096         var mySubSet = (from num in myList select num.Age).Max();
097         Console.WriteLine("成员年龄的最大值是: {0}", mySubSet);
098         var mySubSet1 = (from num in myList select num.Age).Min();
099         Console.WriteLine("成员年龄的最小值是: {0}", mySubSet1);
100         var mySubSet2 = (from num in myList select num.Age).Average();
101         Console.WriteLine("成员年龄的平均值是: {0}", mySubSet2);
102         var mySubSet3 = (from num in myList select num.Age).Sum();
103         Console.WriteLine("成员年龄的和是: {0}", mySubSet3);
104         Console.ReadKey();
105     }
106 }
107 }
```

对查询结果集进行聚合的方法就是对结果集直接调用其聚合方法。引用聚合方法的时候，要把整个查询表达式用括号括起来以保证其优先级。代码的执行结果如下：

```
成员年龄的最大值是: 50
成员年龄的最小值是: 10
成员年龄的平均值是: 30.7142857142857
成员年龄的和是: 215
```

第二十三章 多线程与并行编程

使用 C# 语言可以编写同时执行多项任务的应用程序。这些任务可以在不同的线程上运行，从而对用户输入的响应更快。因为不同的任务在不同的线程上运行，对于用户的接口来说，它是活动的。使用多线程编程，程序的缩放性比较好，因为可以随时增加线程来执行任务。当应用程序开始运行时，会创建一个应用程序域。在应用程序域中可以创建多个线程。在任意一个时刻，一个线程只能运行在一个应用程序域中。

23.1 同步调用

在 C# 的实际编程中，在应用程序中使用多线程的方法，用得最多的就是使用委托。在 .Net 类库中，几乎所有委托都可以异步调用成员，即创建一个新的线程，在新的线程上调用方法。这是 .Net 类库的最大的优点，它自身隐含了线程的创建和管理。下面来介绍一下什么是同步，什么是异步。当应用程序的 Main 方法执行的时候，它将创建一个主线程。在这个主线程上，可以使用多种方法来创建新的线程。下面通过一个代码实例来演示一下同步调用，代码如下：

```
001 using System;
002 using System.Threading;
003 namespace Parallel1
004 {
005     public delegate void D1();
006     class Program
007     {
008         static void Method()
009         {
010             for (int i = 0; i < 3; i++)
011             {
012                 Console.WriteLine("开始循环，当前线程 ID 是:{0}",
013                                     Thread.CurrentThread.ManagedThreadId);
014                 Console.WriteLine("开始循环，当前线程名称是:{0}",
015                                     Thread.CurrentThread.Name);
016                 Console.WriteLine("开始循环，当前线程状态是:{0}",
017                                     Thread.CurrentThread.ThreadState.ToString());
018                 Console.WriteLine();
019                 Thread.Sleep(2000);
020             }
021             Console.WriteLine("等待用户输入");
022             Console.ReadKey();
023         }
024         static void Main(string[] args)
025         {
026             Console.WriteLine("开始执行 Main 方法，当前主线程 ID 是: {0}",
027                                 Thread.CurrentThread.ManagedThreadId);
028             Console.WriteLine("下面开始使用委托同步调用方法");
029             Thread.CurrentThread.Name = "主线程";
030             D1 myD1 = Method;
031             myD1();
032             Console.WriteLine("程序执行完毕");
033             Console.ReadKey();
034         }
035     }
036 }
```

```

030         }
031     }
032 }

```

创建和管理多线程的类位于 `System.Threading` 命名空间中。在本段代码中调用了当前线程的信息，需要用到这个命名空间，所以在第 2 行导入这个命名空间。代码的第 5 行创建了一个委托，它用来调用方法 `Method`。在静态方法中定义了一个 `for` 循环，它共循环 3 次。在每次循环的时候，都调用了 `Thread.CurrentThread` 属性，然后用这个属性再调用 `ManagedThreadId` 属性获得当前执行线程的 ID，当一个线程创建后，系统都会要给它分配一个识别它的 ID 号。调用 `Name` 属性获得当前线程的名称，这个属性也可以设置当前线程的名称。调用 `ThreadState` 属性获得当前线程的状态。接下来调用 `Thread.Sleep` 静态方法让当前线程暂停 2 秒钟。这个方法是以毫秒计时的。线程暂停也叫做挂起。当整个方法执行完毕后，需要等待用户的按键输入。

在 `Main` 方法中，首先第 23 行代码输出了 `Main` 方法的线程 ID，然后调用 `Thread.CurrentThread.Name` 属性的 `set` 访问器为当前线程定义一个名称。接下来为委托添加方法，再调用这个委托。在委托调用的方法执行完毕后，打印“程序执行完毕”。下面是代码的执行结果：

```
开始执行 Main 方法，当前主线程 ID 是：9
```

```
下面开始使用委托同步调用方法
```

```
开始循环，当前线程 ID 是：9
```

```
开始循环，当前线程名称是：主线程
```

```
开始循环，当前线程状态是：Running
```

```
开始循环，当前线程 ID 是：9
```

```
开始循环，当前线程名称是：主线程
```

```
开始循环，当前线程状态是：Running
```

```
开始循环，当前线程 ID 是：9
```

```
开始循环，当前线程名称是：主线程
```

```
开始循环，当前线程状态是：Running
```

```
等待用户输入
```

```
程序执行完毕
```

从代码的执行结果可以观察到，通常的委托调用中，被调用的方法的线程和 `Main` 方法的线程是一个 ID，名称也相同。也就是说，它们是在一个线程上执行的。当委托调用的方法执行时，`Main` 方法不再往下执行而是等待委托调用的方法执行完毕之后，它才继续执行。也可以说，在代码的逻辑转去执行其它方法时，`Main` 方法被阻塞了，它一直等着代码逻辑回来，才可以继续往下执行。在整段代码的执行过程中，代码的执行路线只有一条。这样的方法调用就叫做同步方法调用。

23.2 异步调用

下面再来看一下异步方法调用，它就是一个多线程的调用方法了。现在的计算机基本上都步入到多核处理器时代了，如果使用单一线程的程序，是无法全面发挥多核处理器的优势的。因此，多线程和并行编程基本上是未来程序的发展方向了。下面是代码实例：

```

001 using System;
002 using System.Threading;
003 namespace Parallel1
004 {
005     public delegate void D1();

```

```

006     class Program
007     {
008         static void Method()
009         {
010             for (int i = 0; i < 3; i++)
011             {
012                 Console.WriteLine("开始循环, 当前线程 ID 是: {0}",
                                Thread.CurrentThread.ManagedThreadId);
013                 Console.WriteLine("开始循环, 当前线程名称是: {0}",
                                Thread.CurrentThread.Name);
014                 Console.WriteLine("开始循环, 当前线程状态是: {0}",
                                Thread.CurrentThread.ThreadState.ToString());
015                 Console.WriteLine();
016                 Thread.Sleep(2000);
017             }
018             Console.WriteLine("等待用户输入");
019             Console.ReadKey();
020         }
021         static void Main(string[] args)
022         {
023             Console.WriteLine("开始执行 Main 方法, 当前主线程 ID 是:
                                {0}\n", Thread.CurrentThread.ManagedThreadId);
024             Console.WriteLine("下面开始使用委托异步调用方法\n");
025             Dl myDl = Method;
026             myDl.BeginInvoke(null, null);
027             Console.WriteLine("异步调用的下一条主线程语句开始执行\n");
028             Console.ReadKey();
029         }
030     }
031 }

```

使用委托异步调用方法时, 需要调用委托实例的 BeginInvoke 方法。它带有两个参数, 先不看这两个参数的作用, 因为其为引用类型, 所以为其先传入 null, 这在语法上是允许的。委托基本上都有这个方法, 它就是做异步调用使用的。调用这个方法之后, 就会新创建一个线程。下面看代码的执行结果:

开始执行 Main 方法, 当前主线程 ID 是: 10

下面开始使用委托异步调用方法

异步调用的下一条主线程语句开始执行

开始循环, 当前线程 ID 是: 7

开始循环, 当前线程名称是:

开始循环, 当前线程状态是: Background

开始循环, 当前线程 ID 是: 7

开始循环, 当前线程名称是:
开始循环, 当前线程状态是:Background

开始循环, 当前线程 ID 是:7
开始循环, 当前线程名称是:
开始循环, 当前线程状态是:Background

等待用户输入

在 Main 方法执行之后, 会创建一个主线程, 这个主线程的 ID 是 10, 为委托添加调用方法之后, 就开始调用委托的 BeginInvoke 方法, 这时就会创建一个新的线程。新的线程执行之后, 并不阻碍 Main 方法的主线程的执行, 几乎立刻就会执行第 27 行的代码。接着, 次线程的输出才开始, 可以看到, 它的线程 ID 和主线程不同, 是 7。因为没有为新的线程定义名称, 所以它的名称为 null。新的线程的状态是 Background。这表明它是在后台运行的。当新的线程上的方法执行完毕后, 会等待用户的输入。当用户输入后, 代码逻辑回到主线程中, 这时主线程还在阻塞状态, 等待着新线程的回归, 它才能结束。

23.3 IAsyncResult 接口

前面的例子中, 被异步调用的方法是没有返回值的。但是在实际编程中, 通常方法都有返回值。那么该怎么在主线程中获得新线程中的方法的返回值呢? 这时就需要使用 System.IAsyncResult 接口类型的变量来接收 BeginInvoke 方法的返回值。委托的异步调用方法 BeginInvoke 方法返回一个实现了 IAsyncResult 接口的实例。因此, 可以使用 IAsyncResult 接口的变量来接收它。接收它有什么用呢? 下面看代码实例:

```
001 using System;
002 using System.Threading;
003 namespace Parallel1
004 {
005     public delegate string D1(); // 返回值为字符串
006     class Program
007     {
008         static string Method()
009         {
010             for (int i = 0; i < 3; i++)
011             {
012                 Console.WriteLine("开始循环, 当前线程 ID 是: {0}",
013                                     Thread.CurrentThread.ManagedThreadId);
014                 Console.WriteLine("开始循环, 当前线程名称是: {0}",
015                                     Thread.CurrentThread.Name);
016                 Console.WriteLine("开始循环, 当前线程状态是: {0}",
017                                     Thread.CurrentThread.ThreadState.ToString());
018                 Console.WriteLine();
019                 Thread.Sleep(2000);
020             }
021             Console.WriteLine("执行完毕, 现在返回值\n");
022             return "我是返回值"; // 执行完毕后, 将这个字符串返回
023         }
024         static void Main(string[] args)
025         {
026         }
```

```

023         Console.WriteLine("开始执行 Main 方法，当前主线程 ID 是：
           {0}\n", Thread.CurrentThread.ManagedThreadId);
024         Console.WriteLine("下面开始使用委托异步调用方法\n");
025         Dl myDl = Method;
026         System.IAsyncResult ir = myDl.BeginInvoke(null, null);
027         Console.WriteLine("异步调用的下一条主线程语句(1)开始执行\n");
028         string s = myDl.EndInvoke(ir);
029         Console.WriteLine("新线程中返回的值是：{0}\n", s);
030         Console.WriteLine("异步调用的下一条主线程语句(2)开始执行\n");
031         Console.WriteLine("异步调用的下一条主线程语句(3)开始执行\n");
032         Console.ReadKey();
033     }
034 }
035 }

```

委托的异步调用方法 `BeginInvoke` 返回一个实现了 `IAsyncResult` 接口的实例，那么这个接口是用来做什么的呢？它就像一个句柄一样，作为参数传递给委托的 `EndInvoke` 方法。委托的 `EndInvoke` 方法用于返回异步调用的方法的返回值。使用 `IAsyncResult` 接口的参数就好像拿着句柄能够准确得到异步调用的返回值一样。下面先来看代码的执行结果：

开始执行 Main 方法，当前主线程 ID 是：10

下面开始使用委托异步调用方法

异步调用的下一条主线程语句(1)开始执行

开始循环，当前线程 ID 是:11

开始循环，当前线程名称是：

开始循环，当前线程状态是:Background

开始循环，当前线程 ID 是:11

开始循环，当前线程名称是：

开始循环，当前线程状态是:Background

开始循环，当前线程 ID 是:11

开始循环，当前线程名称是：

开始循环，当前线程状态是:Background

执行完毕，现在返回值

新线程中返回的值是：我是返回值

异步调用的下一条主线程语句(2)开始执行

异步调用的下一条主线程语句(3)开始执行

当启动新的线程之后，第 27 行代码会立即执行。第 28 行代码定义了一个 `string` 类型的变量来接收

EndInvoke 方法的返回值。但是，在这一行代码调用 EndInvoke 方法并传入 IAsyncResult 接口的参数的时候，并不是马上就关闭委托异步调用的方法。它的作用不是结束异步调用方法，而是使用参数在等待异步方法结束之后返回其值。但是，新的问题来了，虽然新的线程已经运行了，但是第 28 行代码没有完成的时候，第 29 行、第 30 行和第 31 行的代码都没有执行。也就是说，主线程在等待新线程的结果的过程中仍旧阻塞了。这当然就违反了使用多线程编程的初衷，即，用户接口不受影响。

23.4 异步调用完成的判断

既然 EndInvoke 方法能够阻塞主线程，那么是否可以这样做：如果能判断出新线程中的异步调用什么时候结束，那么就可以在它结束前，尽情地处理主线程中的任务，而在它结束后再调用 EndInvoke 方法？答案是肯定的，IAsyncResult 接口有一个用来判断异步调用是否结束的属性。下面来看代码实例：

```
001 using System;
002 using System.Threading;
003 namespace Parallel1
004 {
005     public delegate string D1();
006     class Program
007     {
008         static string Method()
009         {
010             for (int i = 0; i < 3; i++)
011             {
012                 Console.WriteLine("开始循环，当前线程 ID 是:{0}",
013                                     Thread.CurrentThread.ManagedThreadId);
014                 Console.WriteLine("开始循环，当前线程名称是:{0}",
015                                     Thread.CurrentThread.Name);
016                 Console.WriteLine("开始循环，当前线程状态是:{0}",
017                                     Thread.CurrentThread.ThreadState.ToString());
018                 Console.WriteLine();
019                 Thread.Sleep(2000);
020             }
021             Console.WriteLine("执行完毕，现在返回值\n");
022             return "我是返回值";
023         }
024         static void Main(string[] args)
025         {
026             Console.WriteLine("开始执行 Main 方法，当前主线程 ID 是：
027                               {0}\n", Thread.CurrentThread.ManagedThreadId);
028             Console.WriteLine("下面开始使用委托异步调用方法\n");
029             D1 myD1 = Method;
030             System.IAsyncResult ir = myD1.BeginInvoke(null, null);
031             for (int i = 0; ; i++)
032             {
033                 if (ir.IsCompleted == true)
034                 {
035                     break;
036                 }
037             }
038         }
039     }
040 }
```



```
032         }
033         Console.WriteLine("主线程语句 {0} 开始执行\n", i + 1);
034         Thread.Sleep(1000);
035     }
036     string s = myDl.EndInvoke(ir);
037     Console.WriteLine("新线程中返回的值是: {0}\n", s);
038     Console.ReadKey();
039 }
040 }
041 }
```

这段代码第 27 行使用了一个 for 的无限循环,在语句块中,判断每次循环的时候,异步调用是否已经结束了。如果结束了,就跳出循环,然后得到异步调用的结果并打印出来。如果没有结束,就继续执行主线程中的语句。这段代码的执行结果如下:

开始执行 Main 方法,当前主线程 ID 是: 8

下面开始使用委托异步调用方法

主线程语句 1 开始执行

开始循环,当前线程 ID 是:9

开始循环,当前线程名称是:

开始循环,当前线程状态是:Background

主线程语句 2 开始执行

开始循环,当前线程 ID 是:9

开始循环,当前线程名称是:

开始循环,当前线程状态是:Background

主线程语句 3 开始执行

主线程语句 4 开始执行

开始循环,当前线程 ID 是:9

开始循环,当前线程名称是:

开始循环,当前线程状态是:Background

主线程语句 5 开始执行

主线程语句 6 开始执行

执行完毕,现在返回值

新线程中返回的值是: 我是返回值

从代码的执行结果可以看出，主线程中的语句和新线程中的语句交替执行了，有点多线程的味道了。还可以有一种方法可以实现这一功能，那就是调用 `IAsyncResult` 接口的 `AsyncWaitHandle` 属性的 `WaitOne` 方法。下面是代码实例：

```
001 using System;
002 using System.Threading;
003 namespace Parallel1
004 {
005     public delegate string D1();
006     class Program
007     {
008         static string Method()
009         {
010             for (int i = 0; i < 3; i++)
011             {
012                 Console.WriteLine("开始循环，当前线程 ID 是: {0}",
013                                     Thread.CurrentThread.ManagedThreadId);
014                 Console.WriteLine("开始循环，当前线程名称是: {0}",
015                                     Thread.CurrentThread.Name);
016                 Console.WriteLine("开始循环，当前线程状态是: {0}",
017                                     Thread.CurrentThread.ThreadState.ToString());
018                 Console.WriteLine();
019                 Thread.Sleep(2000);
020             }
021             Console.WriteLine("执行完毕，现在返回值\n");
022             return "我是返回值";
023         }
024         static void Main(string[] args)
025         {
026             Console.WriteLine("开始执行 Main 方法，当前主线程 ID 是:
027                                 {0}\n", Thread.CurrentThread.ManagedThreadId);
028             Console.WriteLine("下面开始使用委托异步调用方法\n");
029             D1 myD1 = Method;
030             System.IAsyncResult ir = myD1.BeginInvoke(null, null);
031             while(!ir.AsyncWaitHandle.WaitOne(500))
032             {
033                 Console.WriteLine("主线程语句开始执行\n");
034             }
035             string s = myD1.EndInvoke(ir);
036             Console.WriteLine("新线程中返回的值是: {0}\n", s);
037             Console.ReadKey();
038         }
039     }
040 }
```

`AsyncWaitHandle` 属性的 `WaitOne` 方法的作用是阻止当前线程，当它收到异步调用完成的信号时，则

返回 true，否则返回 false。在方法中可以指定等待的时间，单位是毫秒。第 27 行代码的意思就是，等候异步调用的信号 500 毫秒，如果没有结束则返回 false。这时 while 语句块中的语句将执行。如果异步调用结束了，将返回 true，这时会跳出循环，去执行第 31 行的代码。代码的执行结果如下：

开始执行 Main 方法，当前主线程 ID 是：8

下面开始使用委托异步调用方法

开始循环，当前线程 ID 是:9

开始循环，当前线程名称是：

开始循环，当前线程状态是:Background

主线程语句开始执行

主线程语句开始执行

主线程语句开始执行

主线程语句开始执行

开始循环，当前线程 ID 是:9

开始循环，当前线程名称是：

开始循环，当前线程状态是:Background

主线程语句开始执行

主线程语句开始执行

主线程语句开始执行

主线程语句开始执行

开始循环，当前线程 ID 是:9

开始循环，当前线程名称是：

开始循环，当前线程状态是:Background

主线程语句开始执行

主线程语句开始执行

主线程语句开始执行

执行完毕，现在返回值

新线程中返回的值是：我是返回值

23.5 异步调用结束自动回调

目前接触过的例子对异步调用是否结束的判断，都是通过轮询的方法。这种方法比较笨拙，在实际实现比较复杂的任务时，也不好写代码逻辑。下面就开始接触 BeginInvoke 方法的参数，让异步调用结束的时候能自动的调用处理方法。这就是方法的回调。下面是代码实例：

```
001 using System;
002 using System.Threading;
003 namespace Parallel1
004 {
005     public delegate string D1();
006     class Program
007     {
008         static D1 myD1;
009         static string Method()
010         {
011             for (int i = 0; i < 3; i++)
012             {
013                 Console.WriteLine("开始循环，当前线程 ID 是: {0}",
014                                     Thread.CurrentThread.ManagedThreadId);
015                 Console.WriteLine("开始循环，当前线程名称是: {0}",
016                                     Thread.CurrentThread.Name);
017                 Console.WriteLine("开始循环，当前线程状态是: {0}",
018                                     Thread.CurrentThread.ThreadState.ToString());
019                 Console.WriteLine();
020                 Thread.Sleep(2000);
021             }
022             Console.WriteLine("执行完毕，现在返回值\n");
023             return "我是返回值";
024         }
025         static void GetResult(IAsyncResult ir)
026         {
027             string s = myD1.EndInvoke(ir);
028             Console.WriteLine("新线程中返回的值是: {0}\n", s);
029             int i = (int)ir.AsyncState;
030             Console.WriteLine("主线程中传来的状态数据是 {0}", i);
031         }
032         static void Main(string[] args)
033         {
034             Console.WriteLine("开始执行 Main 方法，当前主线程 ID 是:
035                                 {0}\n", Thread.CurrentThread.ManagedThreadId);
036             Console.WriteLine("下面开始使用委托异步调用方法\n");
037             myD1 = Method;
038             AsyncCallback acb = GetResult;
039             System.IAsyncResult ir = myD1.BeginInvoke(acb, 12345);
040             for (int i = 0; i < 5; i++)
```

```

037         {
038             Console.WriteLine("主方法第 {0} 条语句执行", i);
039             Thread.Sleep(1000);
040         }
041         Console.ReadKey();
042     }
043 }
044 }

```

因为在两个方法中都要用到委托，所以将委托在第 8 行以字段的形式进行了定义。第 22 行代码定义了一个自定义的静态方法用来获取异步调用的返回值，它的参数为 `IAsyncResult` 类型。在方法体中，使用委托字段调用了 `EndInvoke` 方法来获取返回值。下面再回头来看第 35 行的 `BeginInvoke` 方法，这次为它传入的参数不再是 `null`。`BeginInvoke` 方法的第一个参数是 `AsyncCallback` 委托，它的定义如下：

```
public delegate void AsyncCallback(IAsyncResult ar)
```

这个委托的作用就是回调方法，就是当异步委托结束的时候，调用为它传入的 `AsyncCallback` 委托。而 `AsyncCallback` 委托又指向了静态方法 `GetResult`。`BeginInvoke` 方法还带有一个 `object` 类型的参数，它是用来由主线程向新线程传递状态信息的。在调用 `AsyncCallback` 委托的时候，`BeginInvoke` 方法会为其传入 `IAsyncResult` 类型的参数。因此第 35 行中的使用 `IAsyncResult` 类型的变量 `ir` 来接收 `BeginInvoke` 方法的返回结果就没有意义了。可以直接调用而不用接收返回值。在第 26 行定义了一个 `int` 类型的变量来接收主线程传来的数据。这个数据由 `IAsyncResult` 类型的参数 `ir` 引用 `IAsyncResult` 类型的 `AsyncState` 属性得来。第 36 行开始是主线程中要执行的语句。下面是代码实例：

```
开始执行 Main 方法，当前主线程 ID 是：10
```

```
下面开始使用委托异步调用方法
```

```
主方法第 0 条语句执行
```

```
开始循环，当前线程 ID 是:6
```

```
开始循环，当前线程名称是：
```

```
开始循环，当前线程状态是:Background
```

```
主方法第 1 条语句执行
```

```
开始循环，当前线程 ID 是:6
```

```
开始循环，当前线程名称是：
```

```
开始循环，当前线程状态是:Background
```

```
主方法第 2 条语句执行
```

```
主方法第 3 条语句执行
```

```
开始循环，当前线程 ID 是:6
```

```
开始循环，当前线程名称是：
```

```
开始循环，当前线程状态是:Background
```

```
主方法第 4 条语句执行
```

```
执行完毕，现在返回值
```

```
新线程中返回的值是：我是返回值
```

主线程中传来的状态数据是 12345

从结果可以看到，主线程和新线程都正常执行了，当新线程执行完毕后会自动调用处理方法来打印输出返回值。

23.6 创建线程

在程序开发过程中，如果不使用委托的方式来建立多线程程序，还可以创建新线程来分配代码的工作任务。下面是代码实例：

```
001    using System;
002    using System.Threading;
003    using System.Collections.Generic;
004    namespace Parallel2
005    {
006        class Regimental
007        {
008            public string Name
009            {
010                get;
011                set;
012            }
013            public int[] i;
014            public Regimental(string name)
015            {
016                this.Name = name;
017                i = new int[5];
018                for (int j = 0; j < 5; j++)
019                {
020                    i[j] = j + 1;
021                }
022            }
023        }
024        class Program
025        {
026            static void Report1()
027            {
028                Regimental r = new Regimental("红一团");
029                Thread.CurrentThread.Name = "线程 1";
030                foreach (int i in r.i)
031                {
032                    Console.WriteLine(Thread.CurrentThread.Name + "-" +
                                Thread.CurrentThread.ManagedThreadId + "-" + r.Name + "-" +
                                i.ToString());
033                    Thread.Sleep(1000);
034                }
035            }
036        }
037    }
```

```

036         static void Report2()
037         {
038             Regimental r = new Regimental("红二团");
039             Thread.CurrentThread.Name = "线程 2";
040             foreach (int i in r.i)
041             {
042                 Console.WriteLine(Thread.CurrentThread.Name + "-" +
                                Thread.CurrentThread.ManagedThreadId + "-" + r.Name + "-" +
                                i.ToString());
043                 Thread.Sleep(1000);
044             }
045         }
046         static void Main(string[] args)
047         {
048             Thread t1 = new Thread(new ThreadStart(Report1));
049             Thread t2 = new Thread(new ThreadStart(Report2));
050             Thread.CurrentThread.Name = "主线程";
051             t1.Start();
052             t2.Start();
053             for (int i = 0; i < 10; i++)
054             {
055                 Console.WriteLine("旅长正在计数");
056                 Thread.Sleep(1000);
057             }
058             Console.WriteLine("计数完毕");
059             Console.ReadKey();
060         }
061     }
062 }

```

这段代码模仿了两个团的士兵报数的场景，代表旅长的主线程进行计数。首相定义了一个表示团的类 `Regimental`，它有一个 `string` 类型的属性代表名称。`int` 类型的数组 `i` 表示每个士兵。因为篇幅的问题，只初始化为含有 5 个成员。创建多线程的时候，方法必须由委托来调用。当使用 `ThreadStart` 委托的时候，需要一个不带参数不带返回值的方法。因此第 26 行和第 36 行各定义了一个静态方法供委托调用。在方法中实例化 `Regimental` 类，然后为调用它的线程定义一个名称。接下来使用 `foreach` 语句迭代输出数组中的内容，附加的信息有当前线程的名称、当前线程的 ID 以及当前实例的 `Name` 属性。每当输出一行的时候，将线程暂停 1 秒钟。

想要创建多线程的时候，需要先实例化 `Thread` 类的实例，`Thread` 类的构造方法含有一个 `ThreadStart` 类型的委托，它不带参数，且无返回值。因此，静态方法 `Report1` 和 `Report2` 都与之匹配。将它们传递给 `ThreadStart` 类型的委托，就表示这个线程准备调用传给它的方法。

第 50 行代码为主线程定义了一个名称。然后就开始启动线程。启动线程使用 `Thread` 类的 `Start` 方法。再启动两个线程之后，使用一个 `for` 语句为主线程也开始一个循环。下面来看代码的执行结果：

```

旅长正在计数
线程 1-10-红一团-1
线程 2-11-红二团-1

```

```
旅长正在计数
线程 2-11-红二团-2
线程 1-10-红一团-2
旅长正在计数
线程 2-11-红二团-3
线程 1-10-红一团-3
线程 1-10-红一团-4
线程 2-11-红二团-4
旅长正在计数
旅长正在计数
线程 1-10-红一团-5
线程 2-11-红二团-5
旅长正在计数
旅长正在计数
旅长正在计数
旅长正在计数
旅长正在计数
计数完毕
```

从代码的结果可以看到，主线程并没有因为其它两个新线程而阻塞。三个线程都在很好地运行。但是如果被调用的方法有参数，则 ThreadStart 类型的委托就不适合了，这时就需要另一个委托类型作为 Thread 类的构造方法的参数。下面看代码实例：

```
001    using System;
002    using System.Threading;
003    using System.Collections.Generic;
004    namespace Parallel2
005    {
006        class Regimental
007        {
008            public string Name
009            {
010                get;
011                set;
012            }
013            public int[] i;
014            public Regimental(string name)
015            {
016                this.Name = name;
017                i = new int[5];
018                for (int j = 0; j < 5; j++)
019                {
020                    i[j] = j + 1;
021                }
022            }
023        }
```



```
024     class Program
025     {
026         static void Report1(object r)
027         {
028             Thread.CurrentThread.Name = "线程 1";
029             foreach (int i in ((Regimental)r).i)
030             {
031                 Console.WriteLine(Thread.CurrentThread.Name + "-" +
                                Thread.CurrentThread.ManagedThreadId + "-" + ((Regimental)r).Name + "-"
                                + i.ToString());
032                 Thread.Sleep(1000);
033             }
034         }
035         static void Report2(object r)
036         {
037             Thread.CurrentThread.Name = "线程 2";
038             foreach (int i in ((Regimental)r).i)
039             {
040                 Console.WriteLine(Thread.CurrentThread.Name + "-" +
                                Thread.CurrentThread.ManagedThreadId + "-" + ((Regimental)r).Name + "-"
                                + i.ToString());
041                 Thread.Sleep(1000);
042             }
043         }
044         static void Main(string[] args)
045         {
046             Regimental r1 = new Regimental("红一团");
047             Regimental r2 = new Regimental("红二团");
048             Thread t1 = new Thread(new ParameterizedThreadStart(Report1));
049             Thread t2 = new Thread(new ParameterizedThreadStart(Report2));
050             Thread.CurrentThread.Name = "主线程";
051             t1.Start(r1);
052             t2.Start(r2);
053             for (int i = 0; i < 10; i++)
054             {
055                 Console.WriteLine("旅长正在计数");
056                 Thread.Sleep(1000);
057             }
058             Console.WriteLine("计数完毕");
059             Console.ReadKey();
060         }
061     }
062 }
```

这段代码将两个静态的要被调用的方法做了修改，为其增加了 object 类型的参数，在方法中，将参数

进行了显式转换，转换为 `Regimental` 类型之后进行使用。为什么参数要使用 `object` 类型的呢？因为，如果要在多线程上调用带有参数的方法，则 `Thread` 的构造方法需要一个 `ParameterizedThreadStart` 类型的委托，这个委托的参数类型是 `object` 类型的。也就是说，可以为它传入任何类型的参数。那么为方法传递的参数在什么地方传入呢？在第 51 行和第 52 行，调用 `Thread` 类的 `Start` 方法时，可以为其传入方法所需要的参数。这段代码的执行结果如下：

```
旅长正在计数
线程 2-11-红二团-1
线程 1-10-红一团-1
线程 2-11-红二团-2
线程 1-10-红一团-2
旅长正在计数
线程 2-11-红二团-3
旅长正在计数
线程 1-10-红一团-3
旅长正在计数
线程 1-10-红一团-4
线程 2-11-红二团-4
旅长正在计数
线程 1-10-红一团-5
线程 2-11-红二团-5
旅长正在计数
旅长正在计数
旅长正在计数
旅长正在计数
旅长正在计数
计数完毕
```

23.7 通知事件

当多个线程一起运行时，如果主线程要获得新线程的数据，但是主线程可能提前于次线程之前结束，那么这时获得的数据有可能是错误。下面先看代码实例：

```
001    using System;
002    using System.Threading;
003    using System.Collections.Generic;
004    namespace Parallel2
005    {
006        class Regimental
007        {
008            public string Name
009            {
010                get;
011                set;
012            }
013            public int[] i;
014            public Regimental(string name)
015            {
```

```
016         this.Name = name;
017         i = new int[5];
018         for (int j = 0; j < 5; j++)
019         {
020             i[j] = j + 1;
021         }
022     }
023 }
024 class Program
025 {
026     private static int i1;
027     private static int i2;
028     static void Report1(object r)
029     {
030         Thread.CurrentThread.Name = "线程 1";
031         foreach (int i in ((Regimental)r).i)
032         {
033             Console.WriteLine(Thread.CurrentThread.Name + "-" +
034                                Thread.CurrentThread.ManagedThreadId + "-" + ((Regimental)r).Name + "-"
035                                + i.ToString());
036             i1 = i;
037             Thread.Sleep(1000);
038         }
039     }
040     static void Report2(object r)
041     {
042         Thread.CurrentThread.Name = "线程 2";
043         foreach (int i in ((Regimental)r).i)
044         {
045             Console.WriteLine(Thread.CurrentThread.Name + "-" +
046                                Thread.CurrentThread.ManagedThreadId + "-" + ((Regimental)r).Name + "-"
047                                + i.ToString());
048             i2 = i;
049             Thread.Sleep(1000);
050         }
051     }
052     static void Main(string[] args)
053     {
054         Regimental r1 = new Regimental("红一团");
055         Regimental r2 = new Regimental("红二团");
056         Thread t1 = new Thread(new ParameterizedThreadStart(Report1));
057         Thread t2 = new Thread(new ParameterizedThreadStart(Report2));
058         Thread.CurrentThread.Name = "主线程";
059         t1.Start(r1);
```

```

056         t2.Start(r2);
057         for (int i = 0; i < 2; i++)
058         {
059             Console.WriteLine("旅长正在计数");
060             Thread.Sleep(1000);
061         }
062         Console.WriteLine("计数完毕");
063         Console.WriteLine("i1:{0}", i1);
064         Console.WriteLine("i2:{0}", i2);
065         Console.ReadKey();
066     }
067 }
068 }

```

在代码的第 26 行和第 27 行定义了两个静态的字段，两个静态的方法对这两个字段要进行赋值。但是，主线程的循环改为只循环 2 次，这时新创建的线程调用的方法还没有执行完，第 63 行代码和第 64 行代码就执行完毕了。这时的数据就是错误的。代码执行如下：

```

旅长正在计数
线程 1-11-红一团-1
线程 2-12-红二团-1
旅长正在计数
线程 1-11-红一团-2
线程 2-12-红二团-2
计数完毕
i1:2
i2:2
线程 2-12-红二团-3
线程 1-11-红一团-3
线程 2-12-红二团-4
线程 1-11-红一团-4
线程 2-12-红二团-5
线程 1-11-红一团-5

```

i1 和 i2 本应该为 5。但是因为主线程的语句提前执行，所以都被赋值为 2，这当然不是想要的结果。要想改正这个问题，就需要次线程能够在执行完毕时通知主线程。在这种情况下，可以使用 `AutoResetEvent` 类。这是一个密封类，它可以给主线程提供通知。下面修改代码：

```

001     using System;
002     using System.Threading;
003     using System.Collections.Generic;
004     namespace Parallel2
005     {
006         class Regimental
007         {
008             public string Name
009             {
010                 get;

```

```
011         set;
012     }
013     public int[] i;
014     public Regimental(string name)
015     {
016         this.Name = name;
017         i = new int[5];
018         for (int j = 0; j < 5; j++)
019         {
020             i[j] = j + 1;
021         }
022     }
023 }
024 class Program
025 {
026     private static int i1;
027     private static int i2;
028     private static AutoResetEvent are1 = new AutoResetEvent(false);
029     private static AutoResetEvent are2 = new AutoResetEvent(false);
030     static void Report1(object r)
031     {
032         Thread.CurrentThread.Name = "线程 1";
033         foreach (int i in ((Regimental)r).i)
034         {
035             Console.WriteLine(Thread.CurrentThread.Name + "-" +
036                               Thread.CurrentThread.ManagedThreadId + "-" + ((Regimental)r).Name + "-"
037                               + i.ToString());
038             i1 = i;
039             Thread.Sleep(3000);
040         }
041         are1.Set();
042     }
043     static void Report2(object r)
044     {
045         Thread.CurrentThread.Name = "线程 2";
046         foreach (int i in ((Regimental)r).i)
047         {
048             Console.WriteLine(Thread.CurrentThread.Name + "-" +
049                               Thread.CurrentThread.ManagedThreadId + "-" + ((Regimental)r).Name + "-"
050                               + i.ToString());
049             i2 = i;
050             Thread.Sleep(1000);
051         }
052         are2.Set();
053     }
054 }
```

```

051         }
052         static void Main(string[] args)
053         {
054             Regimental r1 = new Regimental("红一团");
055             Regimental r2 = new Regimental("红二团");
056             Thread t1 = new Thread(new ParameterizedThreadStart(Report1));
057             Thread t2 = new Thread(new ParameterizedThreadStart(Report2));
058             Thread.CurrentThread.Name = "主线程";
059             t1.Start(r1);
060             t2.Start(r2);
061             for (int i = 0; i < 2; i++)
062             {
063                 Console.WriteLine("旅长正在计数");
064                 Thread.Sleep(1000);
065             }
066             Console.WriteLine("计数完毕");
067             are1.WaitOne();//阻止当前线程,直到收到信号
068             Console.WriteLine("i1:{0}", i1);
069             are2.WaitOne();
070             Console.WriteLine("i2:{0}", i2);
071             Console.ReadKey();
072         }
073     }
074 }

```

要想让次线程能够通知主线程,首先在类 Program 中需要定义 AutoResetEvent 类型的字段。因为要启动两个次线程,所以定义了两个 AutoResetEvent 类型的字段,并用 false 作为参数初始化它们的实例。false 参数的作用是表示初始状态为非终止状态。然后在 Main 方法中每个需要获取字段 i1 和 i2 的语句前都调用 AutoResetEvent 类的 WaitOne 方法。这样,当主线程执行到这里时,就会暂停,等待次线程来改变状态。在每个次线程调用的方法内,当执行完毕的时候,调用 AutoResetEvent 类的 Set 方法来改变主线程中 AutoResetEvent 类实例的状态,改变了,主线程的执行将继续。这样就保证了第 68 行和第 69 行读取到的是最终的结果。代码的执行结果如下:

```

旅长正在计数
线程 1-11-红一团-1
线程 2-12-红二团-1
线程 2-12-红二团-2
旅长正在计数
计数完毕
线程 2-12-红二团-3
线程 1-11-红一团-2
线程 2-12-红二团-4
线程 2-12-红二团-5
线程 1-11-红一团-3
线程 1-11-红一团-4
线程 1-11-红一团-5

```

i1:5

i2:5

23.8 后台线程

目前介绍过的例子中所接触的都是前台线程，前台线程必须都执行完毕，应用程序域才会卸载。如果不特殊指定，按照前面的方法建立的线程都是前台线程。下面是代码实例：

```

001    using System;
002    using System.Threading;
003    using System.Collections.Generic;
004    namespace Parallel2
005    {
006        class Regimental
007        {
008            public string Name
009            {
010                get;
011                set;
012            }
013            public int[] i;
014            public Regimental(string name)
015            {
016                this.Name = name;
017                i = new int[5];
018                for (int j = 0; j < 5; j++)
019                {
020                    i[j] = j + 1;
021                }
022            }
023        }
024        class Program
025        {
026            static void Report1(object r)
027            {
028                Thread.CurrentThread.Name = "线程 1";
029                foreach (int i in ((Regimental)r).i)
030                {
031                    Console.WriteLine(Thread.CurrentThread.Name + "-" +
032                                     Thread.CurrentThread.ManagedThreadId + "-" + ((Regimental)r).Name + "-" +
033                                     i.ToString());
034                    Thread.Sleep(2000);
035                }
036            }
037            static void Report2(object r)
038            {
039                Thread.CurrentThread.Name = "线程 2";

```

```

038         foreach (int i in ((Regimental)r).i)
039         {
040             Console.WriteLine(Thread.CurrentThread.Name + "-" +
                                Thread.CurrentThread.ManagedThreadId + "-" + ((Regimental)r).Name + "-"
                                + i.ToString());
041             Thread.Sleep(1000);
042         }
043     }
044     static void Main(string[] args)
045     {
046         Regimental r1 = new Regimental("红一团");
047         Regimental r2 = new Regimental("红二团");
048         Thread t1 = new Thread(new ParameterizedThreadStart(Report1));
049         Thread t2 = new Thread(new ParameterizedThreadStart(Report2));
050         Thread.CurrentThread.Name = "主线程";
051         t1.Start(r1);
052         t2.Start(r2);
053         for (int i = 0; i < 2; i++)
054         {
055             Console.WriteLine("旅长正在计数");
056             Thread.Sleep(1000);
057         }
058         Console.WriteLine("计数完毕");
059     }
060 }
061 }

```

在这段代码中，Main 方法的最后一句取消掉了 Console.ReadKey 方法。因为第 53 行的 for 语句循环次数比较少，所以主线程的语句会先结束。但是这时次线程的语句还没结束，这时应用程序域会等到所有次线程都执行完成之后，整个应用程序域才会卸载掉。这段代码的执行结果如下：

```

001  旅长正在计数
002  线程 1-10-红一团-1
003  线程 2-11-红二团-1
004  旅长正在计数
005  线程 2-11-红二团-2
006  计数完毕
007  线程 1-10-红一团-2
008  线程 2-11-红二团-3
009  线程 2-11-红二团-4
010  线程 1-10-红一团-3
011  线程 2-11-红二团-5
012  线程 1-10-红一团-4
013  线程 1-10-红一团-5

```

从执行结果上可以观察到，只有所有前台线程都结束了，整个应用程序域才会结束，即使 Main 方法有最后一句 Console.ReadKey 语句。当在第 6 行输出完毕后，立刻按一下按键，当结果输出到第 13 行之后，窗口

会立刻消失。因为主线程也是一个前台线程，它早已经执行完毕了，它并不会等待次要线程执行完毕。

而线程的类别如果是后台线程就不一样了，当前台线程执行完毕后，如果后台线程还在执行，它也会被打断，整个应用程序域退出。下面是代码实例：

```
001 using System;
002 using System.Threading;
003 using System.Collections.Generic;
004 namespace Parallel2
005 {
006     class Regimental
007     {
008         public string Name
009         {
010             get;
011             set;
012         }
013         public int[] i;
014         public Regimental(string name)
015         {
016             this.Name = name;
017             i = new int[5];
018             for (int j = 0; j < 5; j++)
019             {
020                 i[j] = j + 1;
021             }
022         }
023     }
024     class Program
025     {
026         static void Report1(object r)
027         {
028             Thread.CurrentThread.Name = "线程 1";
029             foreach (int i in ((Regimental)r).i)
030             {
031                 Console.WriteLine(Thread.CurrentThread.Name + "-" +
032                                     Thread.CurrentThread.ManagedThreadId + "-" + ((Regimental)r).Name + "-" +
033                                     i.ToString());
034                 Thread.Sleep(8000);
035             }
036         }
037         static void Report2(object r)
038         {
039             Thread.CurrentThread.Name = "线程 2";
040             foreach (int i in ((Regimental)r).i)
```

```

040      Console.WriteLine(Thread.CurrentThread.Name + "-" +
      Thread.CurrentThread.ManagedThreadId + "-" + ((Regimental)r).Name + "-"
      + i.ToString());
041      Thread.Sleep(1000);
042  }
043  }
044  static void Main(string[] args)
045  {
046      Regimental r1 = new Regimental("红一团");
047      Regimental r2 = new Regimental("红二团");
048      Thread t1 = new Thread(new ParameterizedThreadStart(Report1));
049      t1.IsBackground = true;
050      Thread t2 = new Thread(new ParameterizedThreadStart(Report2));
051      Thread.CurrentThread.Name = "主线程";
052      t1.Start(r1);
053      t2.Start(r2);
054      for (int i = 0; i < 2; i++)
055      {
056          Console.WriteLine("旅长正在计数");
057          Thread.Sleep(1000);
058      }
059      Console.WriteLine("计数完毕");
060  }
061  }
062  }

```

这段代码在定义线程 t1 之后，第 49 行将 Thread 的 IsBackground 属性设置为 true。这样 t1 就是一个后台线程。第 32 行代码将 t1 的挂起时间设置为 8 秒，保证当所有前台线程都执行完毕之后，t1 还没有执行完毕。接下来运行代码，执行结果如下：

```

旅长正在计数
线程 1-11-红一团-1
线程 2-12-红二团-1
旅长正在计数
线程 2-12-红二团-2
计数完毕
线程 2-12-红二团-3
线程 2-12-红二团-4
线程 2-12-红二团-5

```

从代码的运行结果可以观察到，当主线程和 t2 线程都执行完毕之后，应用程序立刻退出了，而这时 t1 调用的方法刚刚执行完第一次循环。因为它是后台线程，所以它被打断了。

23.9 lock 关键字

在一个多线程程序中，一个数据可能被多个线程共享。在这种情况下，从数据中读取或向数据文件写入的时候，就有可能发生问题。比如，两个线程同时从一个数据中读取，那么输出的内容究竟是哪个线程读取到的？两个线程同时向数据中写入，那么数据的结果究竟应该是多少？这都是不可知、不可控的问题。要想解决这些问题，就需要一种技术对数据的多线程共享问题进行控制。lock 关键字就是其中的一个。下

面是代码实例:

```
001    using System;
002    using System.Threading;
003    using System.Collections.Generic;
004    namespace Parallel2
005    {
006        class Person
007        {
008            public string Name
009            {
010                get;
011                set;
012            }
013            public int Age
014            {
015                get;
016                set;
017            }
018            public int[] ID;
019            public Person(string name)
020            {
021                this.Name = name;
022                Age = 10;
023                ID = new int[12];
024                Random r = new Random(0);
025                for (int i = 0; i < 12; i++)
026                {
027                    ID[i] = r.Next(10);
028                }
029            }
030            public void Set()
031            {
032                for (int i = 0; i < 3; i++)
033                {
034                    Age = Age + i;
035                    Console.WriteLine("Age 的值是: {0}", Age);
036                }
037            }
038            public void ShowAge()
039            {
040                Console.WriteLine("Person 的年龄是: {0}", Age);
041            }
042            public void ShowID()
043            {
```

```

044         foreach (int i in ID)
045         {
046             Console.WriteLine(i);
047             Thread.Sleep(1000);
048         }
049     }
050 }
051 class Program
052 {
053     static void Main(string[] args)
054     {
055         Person p = new Person("秋叶无声");
056         Console.WriteLine("Person 的 ID 号码是: ");
057         Thread t1 = new Thread(new ThreadStart(p.ShowID));
058         Thread t2 = new Thread(new ThreadStart(p.ShowID));
059         t1.Start();
060         t2.Start();
061         Console.ReadKey();
062     }
063 }
064 }

```

这段代码中的 Person 中定义了一个 int 类型的数组作为每个人的标识使用。第 24 行代码创建了 Random 类的一个实例来生成随机数，在构造方法中传入 0 是表示这个随机数最小是 0。第 25 行的 for 循环中对数组进行初始化。调用了 Random 类的 Next 方法，参数 10 的意思是生成的随机数小于 10。第 42 行的实例方法使用循环迭代输出数组中的每个值。每循环一次，挂起当前线程 1 秒钟。第 57 行和第 58 行定义了两个 Thread 类的实例，为其指定要调用的方法是 ShowID 方法。这两个线程同时访问同一个共享资源。代码的执行结果如下：

```

Person 的 ID 号码是：
778877552255994499222244

```

这时，无法确定哪个数字是哪个线程输出的。它们都混在一起。如果修改 Main 方法中的代码为如下的代码：

```

001     static void Main(string[] args)
002     {
003         Person p = new Person("秋叶无声");
004         Thread t3 = new Thread(new ThreadStart(p.Set));
005         Thread t4 = new Thread(new ThreadStart(p.Set));
006         t3.Start();
007         t4.Start();
008         Console.ReadKey();
009     }

```

这段代码中的第 4 行和第 5 行定义了两个 Thread 类的实例，它们调用的方法是 Set 方法，这个方法使用循环对 Age 属性进行赋值操作。因为两个线程同时对同一个资源进行操作，所以结果也变得混乱不堪。代码的执行结果如下：

```

Age 的值是：10
Age 的值是：11

```

Age 的值是: 13

Age 的值是: 13

Age 的值是: 14

Age 的值是: 16

可以看到, Age 的值还有不变的时候。这是一种错误的结果。

解决这个问题的一种办法就是使用 lock 关键字。使用 lock 关键字之后, 如果一个线程已经在操作数据, 则另一个要对资源进行操作的线程会停止执行, 直到前一个线程释放资源之后, 它才可以继续操作数据。lock 语句的作用就是获取某个给定对象的互斥锁, 执行完语句之后, 再释放该锁。当一个互斥锁被占用时, 在同一个线程中的代码可以获取或释放该锁, 但是其它线程中的代码在该锁被释放前是无法获取该锁的。下面看代码实例:

```
001 using System;
002 using System.Threading;
003 using System.Collections.Generic;
004 namespace Parallel2
005 {
006     class Person
007     {
008         private object myLock = new object();
009         public string Name
010         {
011             get;
012             set;
013         }
014         public int Age
015         {
016             get;
017             set;
018         }
019         public int[] ID;
020         public Person(string name)
021         {
022             this.Name = name;
023             Age = 10;
024             ID = new int[12];
025             Random r = new Random(0);
026             for (int i = 0; i < 12; i++)
027             {
028                 ID[i] = r.Next(10);
029             }
030         }
031         public void Set()
032         {
033             for (int i = 0; i < 3; i++)
034             {
```

```

035         Age = Age + i;
036         Console.WriteLine("Age 的值是: {0}", Age);
037     }
038 }
039 public void ShowAge()
040 {
041     Console.WriteLine("Person 的年龄是: {0}", Age);
042 }
043 public void ShowID()
044 {
045     lock (myLock)
046     {
047         foreach (int i in ID)
048         {
049             Console.Write(i);
050             Thread.Sleep(1000);
051         }
052     }
053 }
054 }
055 class Program
056 {
057     static void Main(string[] args)
058     {
059         Person p = new Person("秋叶无声");
060         Thread t1 = new Thread(new ThreadStart(p.ShowID));
061         Thread t2 = new Thread(new ThreadStart(p.ShowID));
062         t1.Start();
063         t2.Start();
064         Console.ReadKey();
065     }
066 }
067 }

```

在代码的第 45 行对循环迭代输出 ID 数组的代码块进行了 lock 锁定。lock 关键字需要定义一个标识，这个标识必须是一个引用类型。在本段代码中，要锁定的目标是一个 public 访问性的方法，当锁定的目标是一个公共成员方法时，必须先定义一个私有或保护类型字段，本段代码是定义了一个私有的 object 类型字段用作锁的标识。对于私有的方法，可以锁定 this 引用。但是，对于共有的方法，最好不锁定 this 引用，因为公有的成员能够被外部访问，lock(this) 可能会有问题，因为不受控制的代码也可能会锁定该对象。这样就可能造成死锁，即两个或更多个线程等待释放同一对象。锁定公共的字段也可能有问题，尤其是字符串。前面介绍过，在一段代码中有可能多个字符串引用的是同一个实例，因此，当一个字符串被锁定时，可能会造成其它代码无法访问它。对于类的成员，像本例中这样定义一个私有字段来锁定比较好。锁定后，代码的执行结果如下：

```
787525949224787525949224
```

从结果可以看到，两个线程不再能同时访问一个资源，它们的输出不再掺到一起了。

这段代码是锁定实例成员的例子，如果需要锁定的是静态成员应该怎么办呢？下面是代码实例：

```
001 using System;
002 using System.Threading;
003 using System.Collections.Generic;
004 namespace Parallel2
005 {
006     class Person
007     {
008         protected object myLock = new object();
009         private static readonly object mySLock = new object();
010         private static int[] sa;
011         public string Name
012         {
013             get;
014             set;
015         }
016         public int Age
017         {
018             get;
019             set;
020         }
021         public int[] ID;
022         public Person(string name)
023         {
024             this.Name = name;
025             Age = 10;
026             ID = new int[12];
027             Random r = new Random(0);
028             for (int i = 0; i < 12; i++)
029             {
030                 ID[i] = r.Next(10);
031             }
032         }
033         static Person()
034         {
035             sa = new int[10];
036             for (int i = 0; i < 10; i++)
037             {
038                 sa[i] = i * 2 + 1;
039             }
040         }
041         public void Set()
042         {
043             for (int i = 0; i < 3; i++)
```

```
044         {
045             Age = Age + i;
046             Console.WriteLine("Age 的值是: {0}", Age);
047         }
048     }
049     public void ShowAge()
050     {
051         Console.WriteLine("Person 的年龄是: {0}", Age);
052     }
053     public void ShowID()
054     {
055         lock (myLock)
056         {
057             foreach (int i in ID)
058             {
059                 Console.Write(i);
060                 Thread.Sleep(1000);
061             }
062         }
063     }
064     public static void ShowSA()
065     {
066         lock (mySLock)
067         {
068             foreach (int i in sa)
069             {
070                 Console.Write(i);
071                 Thread.Sleep(1000);
072             }
073         }
074     }
075 }
076 class Program
077 {
078     static void Main(string[] args)
079     {
080         Person p = new Person("秋叶无声");
081         Thread t1 = new Thread(new ThreadStart(Person.ShowSA));
082         Thread t2 = new Thread(new ThreadStart(Person.ShowSA));
083         t1.Start();
084         t2.Start();
085         Console.ReadKey();
086     }
087 }
```



```
088    }
```

这段代码的第 10 行定义了静态的数组成员，第 64 行定义了一个静态的方法来读取这个数组。现在要对这个静态方法进行锁定，方法是先定义一个私有静态的字段，例如第 9 行定义了一个私有的静态的 object 类型的字段，并做了初始化。然后在第 66 行锁定这个静态的字段。这样就可以完成锁定。代码的执行结果如下：

```
135791113151719135791113151719
```

综上所述，使用 lock 互斥锁的时候，如果锁定实例方法，则定义一个私有的 object 字段进行锁定；如果锁定静态方法，则定义一个私有的静态 object 字段进行锁定。

23.10 监视器

监视器的功能也是锁定对象，它锁定的对象也必须是引用类型，而不能是值类型。下面是代码实例：

```
001    using System;
002    using System.Threading;
003    using System.Collections.Generic;
004    namespace Parallel2
005    {
006        class Person
007        {
008            private object myLock = new object();
009            private static int[] sa;
010            public string Name
011            {
012                get;
013                set;
014            }
015            public int Age
016            {
017                get;
018                set;
019            }
020            public int[] ID;
021            public Person(string name)
022            {
023                this.Name = name;
024                Age = 10;
025                ID = new int[12];
026                Random r = new Random(0);
027                for (int i = 0; i < 12; i++)
028                {
029                    ID[i] = r.Next(10);
030                }
031            }
032            public void Set()
033            {
034                for (int i = 0; i < 3; i++)
```

```
035         {
036             Age = Age + i;
037             Console.WriteLine("Age 的值是: {0}", Age);
038         }
039     }
040     public void ShowAge()
041     {
042         Console.WriteLine("Person 的年龄是: {0}", Age);
043     }
044     public void ShowID()
045     {
046         Monitor.Enter(myLock);
047         try
048         {
049             foreach (int i in ID)
050             {
051                 Console.Write(i);
052                 Thread.Sleep(1000);
053             }
054         }
055         finally
056         {
057             Monitor.Exit(myLock);
058         }
059     }
060 }
061 class Program
062 {
063     static void Main(string[] args)
064     {
065         Person p = new Person("秋叶无声");
066         Thread t1 = new Thread(new ThreadStart(p.ShowID));
067         Thread t2 = new Thread(new ThreadStart(p.ShowID));
068         t1.Start();
069         t2.Start();
070         Console.ReadKey();
071     }
072 }
073 }
```

使用监视器的时候,同样需要先定义一个私有的字段,如果锁定的是静态方法,则需要定义一个静态的字段;如果锁定的是一个实例方法,则需要定义一个实例的私有字段。字段都可以定义成 object 类型的。如同第 46 行所示,定义监视器的时候,使用的是 Monitor 类,它有一个 Enter 方法,方法的参数就是定义的锁定标识符。然后使用 try 语句将要锁定的语句包裹起来,在 finally 语句中调用 Monitor 类的 Exit 方法释放锁定。Exit 方法的参数就是前面定义的锁定标识符。这段代码的执行结果如下:

787525949224787525949224

23.11 使用特性定义类的线程安全性

使用 Synchronization 特性可以在类级别上宏观控制整个类的所有方法成员都是线程安全的。当然，它也不会出现多个线程同时访问的共享数据实施保护，这无疑降低了代码的性能。下面是代码实例：

```
001 using System;
002 using System.Threading;
003 using System.Runtime.Remoting.Contexts;
004 namespace Parallel2
005 {
006     [Synchronization]
007     class Person:ContextBoundObject
008     {
009         public string Name
010         {
011             get;
012             set;
013         }
014         public int Age
015         {
016             get;
017             set;
018         }
019         public int[] ID;
020         public Person(string name)
021         {
022             this.Name = name;
023             Age = 10;
024             ID = new int[12];
025             Random r = new Random(0);
026             for (int i = 0; i < 12; i++)
027             {
028                 ID[i] = r.Next(10);
029             }
030         }
031         public void Set()
032         {
033             for (int i = 0; i < 3; i++)
034             {
035                 Age = Age + i;
036                 Console.WriteLine("Age 的值是: {0}", Age);
037             }
038         }
039         public void ShowAge()
040         {
```

```

041         Console.WriteLine("Person 的年龄是: {0}", Age);
042     }
043     public void ShowID()
044     {
045         foreach (int i in ID)
046         {
047             Console.Write(i);
048             Thread.Sleep(1000);
049         }
050     }
051 }
052 class Program
053 {
054     static void Main(string[] args)
055     {
056         Person p = new Person("秋叶无声");
057         Thread t1 = new Thread(new ThreadStart(p.ShowID));
058         Thread t2 = new Thread(new ThreadStart(p.ShowID));
059         t1.Start();
060         t2.Start();
061         Console.ReadKey();
062     }
063 }
064 }

```

使用使用 Synchronization 特性时，需要把这个特性定义在类上，然后让这个类派生自 ContextBoundObject 类。这样，整个类的成员就都是线程安全的了。

23.12 可变字段

使用 volatile 修饰的字段叫做可变字段。可变字段的作用是什么呢？实际上，在代码中创建多线程的程序时，每个线程都会有要操作的字段的一个副本。也就是说，线程对字段的改变不一定能马上体现到其它线程中。在程序中可以认为存在一个主内存区域，在这里存放着字段的原始变量，每创建一个操作这个变量的新线程的时候，这个变量就被复制到这个线程中。因此，多个线程导致了字段的不同步。实例代码如下：

```

001 using System;
002 using System.Threading;
003 using System.Collections.Generic;
004 namespace Parallel2
005 {
006     class TestClass
007     {
008         public int a;
009         public void SetA()
010         {
011             for(int i = 0;i<10;i++)
012             {

```

```
013         a = i;
014         Console.Write(a + " ");
015         Thread.Sleep(1000);
016     }
017 }
018 public void GetA()
019 {
020     Console.Write(a + " ");
021 }
022 }
023 class Program
024 {
025     static void Main(string[] args)
026     {
027         TestClass t = new TestClass();
028         Thread t1 = new Thread(new ThreadStart(t.SetA));
029         Thread t2 = new Thread(new ThreadStart(t.SetA));
030         t1.Start();
031         t2.Start();
032         for (int i = 0; i < 15; i++)
033         {
034             t.GetA();
035             Thread.Sleep(1000);
036         }
037         Console.ReadKey();
038     }
039 }
040 }
```

在这段代码中,共有三个线程操作字段 `a`,两个创建的线程 `t1` 和 `t2`,还有主线程的第 32 行的 `for` 语句。在这种情况下,字段 `a` 就是一个不同步的字段。每个字段对它的操作并不一定能马上反映到其它操作它的线程中。对类指定 `Synchronization` 特性可以避免这个问题。对方法锁定只可以保证两个新创建的线程不混乱地访问 `a`,但是不能保证主线程和新创建的线程能及时的访问到变量的变化。在这种情况下,可以用 `volatile` 修饰字段 `a`。这样,这个字段就是可变字段,每个访问它的线程都能访问到它的原始版本。

23.13 线程池

线程池是一种多线程处理方式,可以将任务添加到线程队列,然后自动启动这些任务。它其实就是一个在后台执行多个任务的线程集合。通常它用于服务器程序,每个传入请求都分配给线程池中的线程。如果池中的某个线程完成了任务,它又会回到线程池中继续接受任务。每个线程池的线程数是有限制的。下面是代码实例:

```
001 using System;
002 using System.Threading;
003 namespace Parallel1
004 {
005     class Person
006     {
```

```
007     private int Age
008     {
009         get;
010         set;
011     }
012     public Person(int age)
013     {
014         Age = age;
015     }
016     public void Print()
017     {
018         Console.WriteLine("Person 的年龄是: {0}", Age);
019     }
020 }
021 class Program
022 {
023     static void GetResult(object o)
024     {
025         Console.WriteLine("线程池中当前处理线程 ID: {0}",
026             Thread.CurrentThread.ManagedThreadId);
027         ((Person)o).Print();
028         Console.WriteLine();
029     }
030     static void Main(string[] args)
031     {
032         Console.WriteLine("开始执行 Main 方法, 当前主线程 ID 是:
033             {0}\n", Thread.CurrentThread.ManagedThreadId);
034         WaitCallback wcb = new WaitCallback(GetResult);
035         Console.WriteLine("下面开始使用线程池异步调用方法\n");
036         for (int i = 0; i < 5; i++)
037         {
038             Console.WriteLine("正在添加第 {0} 个任务", i + 1);
039             ThreadPool.UnsafeQueueUserWorkItem(wcb, new Person(i + 10));
040         }
041         for (int i = 0; i < 5; i++)
042         {
043             Console.WriteLine("主方法第 {0} 条语句执行", i);
044             Thread.Sleep(1000);
045         }
046         Console.ReadKey();
047     }
048 }
```

为了给线程池中的任务传入数据, 第 5 行代码开始定义了一个类 Person。为了简便, 它只含有一个 Age

属性和一个打印 Age 属性的方法。

先不看第 23 行的静态方法，先看第 32 行代码，它定义了一个 WaitCallback 类型的委托。这个委托类型是 .Net 类库中的。它是线程池中的线程回调处理方法时用的委托。为这个委托添加的方法是 GetResult 方法，就是第 23 行定义的方法。这个处理方法按照委托类型 WaitCallback 的定义是必须以 object 类型作为接收处理的参数。因为本例中要给线程池调用的方法传入 Person 类型的实例，所以在 GetResult 方法中除了打印当前线程 ID 外，还将传入的 object 类型的参数转换成 Person 类型。然后调用其 Print 方法来打印 Age 属性。

第 34 行开始使用一个 for 循环语句为线程池中的 5 个线程添加回调方法和传入的待处理的参数。为线程池中的线程添加参数，就是反复的调用 ThreadPool.UnsafeQueueUserWorkItem 方法。为了有区别的传入参数，在对 Person 类进行实例化的时候，调用它的构造方法使用了局部变量 i。

线程池是由 CLR 管理的，它是动态的。它的启动不用手工编写代码，只需要为其传入参数，线程池中的线程立刻就会开始启动。

第 39 行定义了一个 for 循环，在主线程中重复调用第 41 行的输出语句，以便比较主线程在线程池工作的时候是否阻塞。这段代码的执行结果如下：

开始执行 Main 方法，当前主线程 ID 是：9

下面开始使用线程池异步调用方法

正在添加第 1 个任务

正在添加第 2 个任务

正在添加第 3 个任务

正在添加第 4 个任务

正在添加第 5 个任务

主方法第 0 条语句执行

线程池中当前处理线程 ID：11

线程池中当前处理线程 ID：10

线程池中当前处理线程 ID：12

Person 的年龄是:12

线程池中当前处理线程 ID：12

Person 的年龄是:13

线程池中当前处理线程 ID：12

Person 的年龄是:14

Person 的年龄是:11

Person 的年龄是:10

主方法第 1 条语句执行

主方法第 2 条语句执行

主方法第 3 条语句执行

主方法第 4 条语句执行

从代码的执行结果可以看到，线程池的运行是极为迅速的。以致让处理方法中的紧挨着的语句都分离开进

行了执行。

23.14 线程计时器

线程计时器是一个可以按照预定的时间间隔回调方法的类。它的回调委托使用的是 TimerCallback 委托类型。下面是代码实例：

```
001    using System;
002    using System.Threading;
003    using System.Text;
004    namespace Parallel1
005    {
006        class Person
007        {
008            public string Name
009            {
010                get;
011                set;
012            }
013            public int Age
014            {
015                get;
016                set;
017            }
018            public string Address
019            {
020                get;
021                set;
022            }
023            public Person(string name,int age,string address)
024            {
025                Name = name;
026                Age = age;
027                Address = address;
028            }
029            public void Print()
030            {
031                Console.WriteLine("姓名: {0}, 年龄: {1}, 地址: {2}",Name, Age, Address);
032            }
033        }
034        class Program
035        {
036            static void GetData(object o)
037            {
038                Console.Clear();
039                ((Person)o).Print();
040                Console.WriteLine(DateTime.Now.ToString());
```



```

041     }
042     static void Main(string[] args)
043     {
044         Person p = new Person("秋叶无声的电子钟", 30, "建国街 30 号");
045         TimerCallback tcb = new TimerCallback(GetData);
046         Timer t = new Timer(tcb, p, 0, 1000);
047         Console.ReadKey();
048     }
049 }
050 }

```

为了在控制台上显示个人的信息，从第 6 行代码开始，定义了一个 Person 类。它的成员包括姓名、年龄和地址。先来看代码的第 45 行，在这一行代码中定义了一个 TimerCallback 委托。它的参数是 object，没有返回值。这个委托就是 Timer 类要回调的委托。为这个委托定义一个回调方法，就是第 36 行的 GetData 方法。它带有一个 object 类型的参数。需要注意这个方法必须是静态的，这是 TimerCallback 委托要求的。在第 38 行对控制台进行清屏，以便除去上一次的输出。这样看起来，输出的信息就在控制台上的位置没有发生变化。第 39 行将传递进来的参数进行类型转换，然后调用 Person 类的 Print 方法打印个人信息。第 40 行调用 DateTime 结构的 Now 属性向控制台打印当前时间。

第 46 行定义了一个 Timer 类的实例，为其传入的参数依次是：回调委托、状态数据、第一次调用委托前的间隔时间和每次调用回调委托的间隔时间。在这行代码中，为构造方法传递的状态数据就是第 44 行定义的 Person 类的实例，这个实例将由 GetData 方法接受并处理。代码的执行结果如下：

```

姓名：秋叶无声的电子钟，年龄：30，地址：建国街 30 号
2013/11/8 22:10:14

```

23.15 async 与 await 关键字

async 用来修饰方法，表示这个方法是一个异步方法。用 async 修饰一个方法之后，这个方法内一定要使用 await 关键字。await 关键字表示等待一个线程的执行完毕。使用这两个关键字的条件是 .Net Framework 的版本必须是 4.5。下面是代码实例：

```

001     using System;
002     using System.Threading;
003     using System.Threading.Tasks;
004     namespace Parallel2
005     {
006         class Program
007         {
008             private static int t1;
009             static async void GetNum()
010             {
011                 Task<int> t = new Task<int>(GetNum1);
012                 t.Start();
013                 t1 = await t;
014                 are.Set();
015             }
016             static int GetNum1()
017             {
018                 int a = 0;

```

```

019         for (int i = 0; i < 10; i++)
020         {
021             a = i;
022             Console.WriteLine(a);
023             Thread.Sleep(1000);
024         }
025         return a;
026     }
027     private static AutoResetEvent are = new AutoResetEvent(false);
028     static void Main(string[] args)
029     {
030         GetNum();
031         for (int i = 0; i < 5; i++)
032         {
033             Console.WriteLine("执行主线程中的语句");
034             Thread.Sleep(1000);
035         }
036         are.WaitOne();
037         Console.WriteLine("a 的值是: {0}", t1);
038         Console.ReadKey();
039     }
040 }
041 }

```

使用 `async` 修改方法之后，这个异步方法的返回值必须是 `void`、`Task` 或者 `Task<T>` 类型。`Task` 类和它的泛型类表示一个可以返回值的异步操作。使用这个类需要引用 `System.Threading.Tasks` 命名空间。并且 `await` 能够等候的也必须是能够返回 `Task` 类型的方法或线程。代码的第 9 行定义了一个异步方法 `GetNum`，在方法中的第 11 行定义了一个 `Task<int>` 的变量，然后对它实例化。`Task<T>` 的构造方法需要一个 `Func<TResult>` 类型的委托，这个委托不需参数但是需要返回值。因此这里传递给它的是 `GetNum1` 方法。第 12 行调用 `Task<T>` 类的 `start` 方法，这个方法可以方便的开始一个线程。第 13 行的 `await t` 的意思就是当前线程暂时挂起，等待线程 `t` 的返回。返回后将线程的返回值赋值给静态字段 `t1`。需要注意，`t1` 的类型必须是 `Task` 的类型实参类型。

代码第 27 行定义了一个通知事件，用来告诉主线程，这个异步方法结束了。在代码的第 30 行调用异步方法。然后开始主线程的 `for` 语句。第 36 行调用 `AutoResetEvent` 类的 `WaitOne` 方法等待异步调用的结束。当 `GetNum` 方法执行完毕的时候，第 14 行代码将被调用。这时开始执行主线程中的第 37 行代码获得正确的静态字段 `t1` 的值。代码的执行结果如下：

```

0
执行主线程中的语句
1
执行主线程中的语句
2
执行主线程中的语句
3
执行主线程中的语句
4

```

执行主线程中的语句

```
5
6
7
8
9
```

a 的值是: 9

对于 Task<T>创建线程的调用通常采取下面的调用方法, 这样的方法更简便:

```
001 static async void GetNum()
002 {
003     Task<int> t = Task<int>.Factory.StartNew(GetNum1);
004     t1 = await t;
005     are.Set();
006 }
```

使用第 3 行代码的方式就不用再次调用 Start 方法开始线程, 而是通过访问 Task 类的工厂方法直接开始线程。它的效果和前面例子中是一样的。

async 后跟的返回值除了 void 之外, 就是 Task<T>或 Task 类型了。使用 Task<T>类型的返回值, 对于这个例子来说, 实际在方法中只需返回 int 类型的值就可以了。Task<T>就可以封装它。下面是修改后的代码实例:

```
001 using System;
002 using System.Threading;
003 using System.Threading.Tasks;
004 namespace Parallel2
005 {
006     class Program
007     {
008         static async Task<int> GetNum()
009         {
010             Task<int> t = Task<int>.Factory.StartNew(GetNum1);
011             int result = await t;
012             return result;
013         }
014         static int GetNum1()
015         {
016             int a = 0;
017             for (int i = 0; i < 10; i++)
018             {
019                 a = i;
020                 Console.Write(a + " ");
021                 Console.WriteLine("正在执行创建线程语句, 当前线程{0}",
022                                     Thread.CurrentThread.ManagedThreadId);
022                 Thread.Sleep(500);
023             }
024             return a;
025         }
026     }
027 }
```

```

025     }
026     static async void GetNum2()
027     {
028         int i = await GetNum();
029         Console.WriteLine("i 的值是: {0}", i);
030     }
031     static void Main(string[] args)
032     {
033         GetNum2();
034         for (int i = 0; i < 5; i++)
035         {
036             Console.WriteLine("正在执行主线程语句, 当前线程 {0}",
                                Thread.CurrentThread.ManagedThreadId);
037             Thread.Sleep(1000);
038         }
039         Console.ReadKey();
040     }
041 }
042 }

```

首先来看三个静态方法的调用过程, 它们的调用逻辑顺序是: GetNum2 调用了 GetNum, GetNum 又调用了 GetNum1。第 8 行的 GetNum 返回的就是封装后的 Task<T>类型, 在方法体中, 可以看到, 它实际上返回的是 int 类型, 然后进行了封装。实际上, 线程的返回值是 int 类型, 如果要想从返回值是 Task<T>类型的方法中获得它, 就必须以 await 关键字修饰这个方法的调用, 就如同第 28 行所示。当然, 使用这个关键字之后, 这个方法也必须用 async 定义成异步方法。当用 await 调用返回类型为 Task<T>的方法之后, 就可以直接取出其实际线程调用方法的返回值。第 28 行直接用一个 int 类型的变量 i 来获得。当在 Main 方法中调用 GetNum2 的时候, 不用加 async。而且 Main 方法中也不用加 async, 实际上 Main 方法也不允许加 async 关键字。这段代码的执行结果如下:

```

0 正在执行创建线程语句, 当前线程 10
正在执行主线程语句, 当前线程 9
1 正在执行创建线程语句, 当前线程 10
2 正在执行创建线程语句, 当前线程 10
正在执行主线程语句, 当前线程 9
3 正在执行创建线程语句, 当前线程 10
4 正在执行创建线程语句, 当前线程 10
正在执行主线程语句, 当前线程 9
5 正在执行创建线程语句, 当前线程 10
正在执行主线程语句, 当前线程 9
6 正在执行创建线程语句, 当前线程 10
7 正在执行创建线程语句, 当前线程 10
正在执行主线程语句, 当前线程 9
8 正在执行创建线程语句, 当前线程 10
9 正在执行创建线程语句, 当前线程 10
i 的值是: 9

```

除了基本类型外, await 也能返回自定类型, 下面这个实例返回的是自定义类 Person 的实例:

```
001 using System;
002 using System.Threading;
003 using System.Threading.Tasks;
004 namespace Parallel2
005 {
006     class Person
007     {
008         public int Age
009         {
010             get;
011             set;
012         }
013         public Person(int age)
014         {
015             Age = age;
016         }
017     }
018     class Program
019     {
020         static async Task<Person> GetNum()
021         {
022             Task<Person> t = Task<Person>.Factory.StartNew(GetNum1);
023             Person result = await t;
024             return result;
025         }
026         static Person GetNum1()
027         {
028             Person[] p = new Person[10];
029             for (int i = 0; i < 10; i++)
030             {
031                 p[i] = new Person(i + 10);
032                 Console.WriteLine("正在执行创建线程语句，当前线程{0}",
033                                     Thread.CurrentThread.ManagedThreadId);
034                 Thread.Sleep(500);
035             }
036             return p[9];
037         }
038         static async void GetNum2()
039         {
040             Person i = await GetNum();
041             Console.WriteLine("Person 的年龄是: {0}", i.Age);
042         }
043         static void Main(string[] args)
044         {
```

```

044         GetNum2();
045         for (int i = 0; i < 5; i++)
046         {
047             Console.WriteLine("正在执行主线程语句，当前线程{0}",
                                Thread.CurrentThread.ManagedThreadId);
048             Thread.Sleep(1000);
049         }
050         Console.ReadKey();
051     }
052 }
053 }

```

在代码的开始，定义了一个类 Person。它的成员只有一个 int 类型的字段 Age，用来表示人的年龄。因为 GetNum1 返回的是一个 Person 类的实例，所以 Task<T> 的类型实参也应该定义成 Person 类型。GetNum1 方法有下列改动，在第 28 行定义了一个 Person 类型的数组，然后使用 for 循环对它初始化。最后返回数组中最后一个成员。在 GetNum2 方法中，使用 Person 类型变量接收 await 语句的返回值。第 40 行代码进行打印输出。可以看到，当异步方法的返回值为 void 的时候，这个方法的作用类似于事件处理作用。代码的执行结果如下：

```

正在执行主线程语句，当前线程 10
正在执行创建线程语句，当前线程 11
正在执行创建线程语句，当前线程 11
正在执行主线程语句，当前线程 10
正在执行创建线程语句，当前线程 11
正在执行创建线程语句，当前线程 11
正在执行主线程语句，当前线程 10
正在执行创建线程语句，当前线程 11
正在执行创建线程语句，当前线程 11
正在执行主线程语句，当前线程 10
正在执行创建线程语句，当前线程 11
正在执行创建线程语句，当前线程 11
正在执行主线程语句，当前线程 10
正在执行创建线程语句，当前线程 11
正在执行创建线程语句，当前线程 11
Person 的年龄是：19

```

23.16 并行编程中的 For 方法

为了更好地发挥多 CPU 计算机和多核心处理器的性能，可以使用 .Net 类库中的任务并行库编程。相关类都位于 System.Threading.Tasks 命名空间下。使用任务并行库编程后，线程池的工作分配交由任务并行库进行管理。下面介绍并行编程中比较重要的一个类 Parallel，在这个类中定义了两个方法可以迭代循环相关的语句。下面是代码实例：

```

001 using System;
002 using System.Threading;
003 using System.Threading.Tasks;
004 namespace Parallel2
005 {
006     class Program

```

```

007     {
008         static void GetNum1(int a)
009         {
010             for (int i = a; i < 3; i++)
011             {
012                 Console.WriteLine("正在执行创建线程语句, 当前线程 {0}",
                                Thread.CurrentThread.ManagedThreadId);
013                 Thread.Sleep(1000);
014             }
015         }
016         static void Main(string[] args)
017         {
018             Parallel.For(0, 10, GetNum1);
019             for (int i = 0; i < 5; i++)
020             {
021                 Console.WriteLine("正在执行主线程语句, 当前线程 {0}",
                                Thread.CurrentThread.ManagedThreadId);
022                 Thread.Sleep(1000);
023             }
024             Console.ReadKey();
025         }
026     }
027 }

```

在代码的第 18 行调用了 `Parallel` 类的 `For` 方法, 它的用法和 `for` 循环差不多, 它有多个重载方法。这里选择了其中的一个演示。它的第一个参数和第二个参数是循环条件范围。本段代码就是假定有一个循环变量 `i`, 它相当于一个 `for` 循环, 语句如下:

```

for (int i = 0; i < 10; i++)
{
}

```

最后一个参数是一个 `Action<Int32>` 类型的委托。这个委托和 `Func` 委托相似, 都是 .Net 类库中定义好的。这个委托有一个 `int` 类型的参数, 没有返回值。那么它的参数如何传递呢? 前面假定的局部变量 `i` 就是参数。每循环一次, 就把这个变量传入委托调用的方法一次。为了匹配委托, 第 8 行定义的方法带有一个 `int` 类型的参数, 没有返回值。这个参数用作 `for` 循环的初始化。当 `For` 方法第一次循环的时候, 循环变量为 0, 这个值将传入 `GetNum1` 方法, 这时第 10 行的语句循环 3 次; 接着 `For` 方法的循环变量为 1, 这个变量传入 `GetNum1` 方法后, 它循环打印 2 次; 然后循环变量 2 被传入, 打印 1 次。当循环变量为 3 和 3 以上的值时, 传入 `GetNum1` 方法均不具备循环条件。因此最终 `GetNum1` 中的循环共循环了 6 次。下面先来看代码的执行结果:

```

正在执行创建线程语句, 当前线程 10
正在执行创建线程语句, 当前线程 11
正在执行创建线程语句, 当前线程 10
正在执行创建线程语句, 当前线程 11
正在执行创建线程语句, 当前线程 10
正在执行创建线程语句, 当前线程 11
正在执行主线程语句, 当前线程 10

```

```
正在执行主线程语句，当前线程 10
正在执行主线程语句，当前线程 10
正在执行主线程语句，当前线程 10
正在执行主线程语句，当前线程 10
```

可以看到，任务并行库调用了线程池。当线程 10 调用完毕后，它又回到了线程池中。当调用主线程中的其它语句时，仍然使用了这个线程。下面在 Main 方法中第一条语句中输出 Main 方法的主线程 ID，下面是执行结果：

```
正在执行主线程语句，当前线程 10
正在执行创建线程语句，当前线程 10
正在执行创建线程语句，当前线程 11
正在执行创建线程语句，当前线程 10
正在执行创建线程语句，当前线程 11
正在执行创建线程语句，当前线程 10
正在执行创建线程语句，当前线程 11
正在执行主线程语句，当前线程 10
正在执行主线程语句，当前线程 10
正在执行主线程语句，当前线程 10
正在执行主线程语句，当前线程 10
正在执行主线程语句，当前线程 10
```

从执行结果可以观察到，主线程的 ID 就是 10。当主线程挂起后，开始执行 For 方法。这时线程池管理的线程是包括主线程的。主线程又被分配来做 For 方法中调用的委托。当线程执行完毕后，主线程又来执行 Main 方法中的语句。

这个例子中，For 方法执行后，第 20 行开始的代码就挂起了，主线程被分配做其它的工作。这时的并行任务是不完善的。如果主线程是窗口的话，这时窗口将会因主线程的阻塞问题而变得不可用。这样一来，互动性和用户体验就不是很好。下面来改正这个问题，代码如下：

```
001 using System;
002 using System.Threading;
003 using System.Threading.Tasks;
004 namespace Parallel2
005 {
006     class Program
007     {
008         static void GetNum1(int a)
009         {
010             for (int i = a; i < 3; i++)
011             {
012                 Console.WriteLine("正在执行创建线程语句，当前线程{0}",
                                Thread.CurrentThread.ManagedThreadId);
013                 Thread.Sleep(1000);
014             }
015         }
016         static void InvokeFor()
017         {
018             Parallel.For(0, 10, GetNum1);
```



```

019     }
020     static void Main(string[] args)
021     {
022         Console.WriteLine("正在执行主线程语句，当前线程{0}",
            Thread.CurrentThread.ManagedThreadId);
023         Task.Factory.StartNew(InvokeFor);
024         for (int i = 0; i < 5; i++)
025         {
026             Console.WriteLine("正在执行主线程语句，当前线程{0}",
                Thread.CurrentThread.ManagedThreadId);
027             Thread.Sleep(1000);
028         }
029         Console.ReadKey();
030     }
031 }
032 }

```

这段代码将 `Parallel.For` 方法定义在了静态方法 `InvokeFor` 中进行调用。目的就是创建一个无参数无返回值的方法，用来让 `Task.Factory.StartNew` 工厂方法来创建线程。`Task.Factory.StartNew` 方法需要一个 `Action` 类型的委托，这个委托有一个无参数和无返回值的重载。当在第 23 行调用 `Task.Factory.StartNew` 方法来创建线程的时候，主线程就不再被重新分配别的任务而阻塞了。下面是代码实例：

```

正在执行主线程语句，当前线程 9
正在执行主线程语句，当前线程 9
正在执行创建线程语句，当前线程 10
正在执行创建线程语句，当前线程 11
正在执行主线程语句，当前线程 9
正在执行创建线程语句，当前线程 11
正在执行创建线程语句，当前线程 10
正在执行主线程语句，当前线程 9
正在执行创建线程语句，当前线程 10
正在执行创建线程语句，当前线程 11
正在执行主线程语句，当前线程 9
正在执行主线程语句，当前线程 9

```

可以看到，这次真正的实现了并行任务的执行。

23.17 并行编程中的 `ForEach` 方法

`Parallel` 类还有一个 `ForEach` 方法可以对指定数据源进行并行迭代。下面是代码实例：

```

001 using System;
002 using System.Threading;
003 using System.Threading.Tasks;
004 namespace Parallel2
005 {
006     class Program
007     {
008         private static int[] a = new int[10] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

```

```

009         static void GetNum1(int j)
010         {
011             Console.WriteLine("{0} 正在执行创建线程语句, 当前线程
                {1}", j, Thread.CurrentThread.ManagedThreadId);
012             Thread.Sleep(1000);
013         }
014         static void InvokeFor()
015         {
016             Parallel.ForEach<int>(a, new Action<int>(GetNum1));
017         }
018         static void Main(string[] args)
019         {
020             Console.WriteLine("正在执行主线程语句, 当前线程 {0}",
                Thread.CurrentThread.ManagedThreadId);
021             Task.Factory.StartNew(InvokeFor);
022             for (int i = 0; i < 5; i++)
023             {
024                 Console.WriteLine("正在执行主线程语句, 当前线程 {0}",
                    Thread.CurrentThread.ManagedThreadId);
025                 Thread.Sleep(1000);
026             }
027             Console.ReadKey();
028         }
029     }
030 }

```

代码的第 16 行调用了 `Parallel.ForEach<int>` 方法, 这是一个泛型方法。它的第一个参数是指定的类型实参类型的集合, 这个集合必须是可以枚举的。它的第二个参数是一个 `Action<T>` 类型的委托, 这个委托的类型实参必须和 `ForEach<int>` 方法的类型实参一样。当然这个 `Action` 委托也有多个重载, 这里只选择了其中一个。为了匹配这个委托, 第 9 行代码定义了一个带有 `int` 类型参数的方法。下面来介绍 `ForEach` 方法的参数的作用。它的第一个参数是一个可枚举的集合, 在方法执行的时候, 每当对这个集合迭代一次, 这个集合的成员就作为参数传递给调用的 `Action` 委托。第 8 行定义了一个 `int` 类型的数组作为集合参数传递进来。那么, 第一次迭代的时候, 数组中的第一个成员 1 就作为参数传递给 `Action` 委托。因为这个委托调用了第 9 行的方法, 当参数 1 传递给它的时候, `GetNum1` 方法会打印它的值和当前的线程。这就是 `ForEach` 方法的执行过程。代码的执行结果如下

```

正在执行主线程语句, 当前线程 10
正在执行主线程语句, 当前线程 10
1 正在执行创建线程语句, 当前线程 7
6 正在执行创建线程语句, 当前线程 11
正在执行主线程语句, 当前线程 10
2 正在执行创建线程语句, 当前线程 7
4 正在执行创建线程语句, 当前线程 12
7 正在执行创建线程语句, 当前线程 11
正在执行主线程语句, 当前线程 10
3 正在执行创建线程语句, 当前线程 7

```

```

5 正在执行创建线程语句，当前线程 12
8 正在执行创建线程语句，当前线程 11
9 正在执行创建线程语句，当前线程 13
正在执行主线程语句，当前线程 10
10 正在执行创建线程语句，当前线程 7
正在执行主线程语句，当前线程 10

```

从结果中可以看到，ForEach 方法使用了线程池中的 4 个线程对集合进行了迭代。还有，因为并行迭代的关系，输出并不是按照集合中的成员定义的顺序执行。

23.18 取消并行线程

一般来说，并行编程执行的都是很繁重的任务。当并行的线程没有执行完毕的时候，因为时间关系要停止它们该怎么办呢？可以通过在被调用的方法中定义取消标记的办法来通知线程的结束。下面是代码实例：

```

001  using System;
002  using System.Threading;
003  using System.Threading.Tasks;
004  namespace Parallel2
005  {
006      class Program
007      {
008          private static int[] a = new int[10] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
009          private static CancellationTokenSource cts = new CancellationTokenSource();
010          private static ParallelOptions po = new ParallelOptions();
011          static void GetNum1(int j)
012          {
013              try
014              {
015                  po.CancellationToken.ThrowIfCancellationRequested();
016                  Console.WriteLine("{0} 正在执行创建线程语句，当前线程 {1}", j,
017                      Thread.CurrentThread.ManagedThreadId);
018                  Thread.Sleep(1000);
019              }
020              catch (OperationCanceledException e)
021              {
022                  Console.WriteLine(e.Message);
023              }
024          }
025          static void InvokeFor()
026          {
027              po.CancellationToken = cts.Token;
028              po.MaxDegreeOfParallelism = Environment.ProcessorCount;
029              Parallel.ForEach<int>(a, new Action<int>(GetNum1));
030          }
031          static void Main(string[] args)
032          {

```

```

032         Console.WriteLine("正在执行主线程语句, 当前线程{0}",
                                Thread.CurrentThread.ManagedThreadId);
033         Task.Factory.StartNew(InvokeFor);
034         for (int i = 0; i < 5; i++)
035         {
036             Console.WriteLine("正在执行主线程语句, 当前线程{0}",
                                    Thread.CurrentThread.ManagedThreadId);
037             if (i == 2)
038             {
039                 cts.Cancel();
040             }
041             Thread.Sleep(500);
042         }
043         Console.ReadKey();
044     }
045 }
046 }

```

首先在类中定义一个静态的 `CancellationTokenSource` 类的实例, 这个实例的作用是通知取消标记用户执行了取消命令。然后再定义一个 `ParallelOptions` 类的实例, 这个类的作用是存储用于配置 `Parallel` 类的方法操作的选项。然后在第 26 行的 `InvokeFor` 方法中调用 `ForEach` 方法前, 设置方法选项的取消标记为第 9 行定义的取消标记。然后在第 27 行设置方法选项的最大并行度为本计算机上的处理器个数。处理器个数通过 `Environment.ProcessorCount` 属性获得。在实际调用的方法中, 第 13 行使用 `try` 语句将要执行的语句包裹起来, 在要执行的语句前, 调用 `ParallelOptions` 类的实例 `po` 的 `CancellationToken.ThrowIfCancellationRequested` 方法, 这个方法的作用是当取消操作发生时, 能够抛出一个 `OperationCanceledException` 异常通知用户线程已经取消。第 19 行代码捕获异常并显示出来。第 37 行判断当循环变量为 2 的时候, 调用定义的 `CancellationTokenSource` 类的实例 `cts` 的 `Cancel` 方法取消并行线程的任务。这段代码的执行结果如下:

```

正在执行主线程语句, 当前线程 8
正在执行主线程语句, 当前线程 8
1 正在执行创建线程语句, 当前线程 9
3 正在执行创建线程语句, 当前线程 10
5 正在执行创建线程语句, 当前线程 11
7 正在执行创建线程语句, 当前线程 12
正在执行主线程语句, 当前线程 8
正在执行主线程语句, 当前线程 8
2 正在执行创建线程语句, 当前线程 9
已取消该操作。
已取消该操作。
已取消该操作。
已取消该操作。
已取消该操作。
正在执行主线程语句, 当前线程 8
正在执行主线程语句, 当前线程 8

```

从代码的执行结果可以观察到, 当取消并行线程的任务后, 所有准备执行任务的线程都被取消了, 并抛出

了异常信息通知用户，该操作已经取消。

参考文献

微软 MSDN 文档