

**Fortran 语言程序**

Qsh

## 第一章: Fortran 语言程序设计初步

### Fortran 语言的发展概况

本节介绍 Fortran 的起源与发展历史，讲述 Fortran 由产生到形成标准 FortranIV、Fortran77，并进一步形成新标准 Fortran90/95 的发展历程。

#### 1.1.1 Fortran 的历史

##### a) FortranI FortranIV

Fortran 是目前国际上广泛流行的一种高级语言，适用于科学计算。Fortran 是英文 **FORmula TRANslation** 的缩写，意为“公式翻译”。它是为科学、工程问题中的那些能够用数学公式表达的问题而设计的语言，主要用于数值计算。这种语言简单易学，因为可以像抄写数学教科书里的公式一样

书写数学公式，它比英文书写的自然语言更接近数学语言。Fortran 语言是第一个真正推广的高级语言。至今它已有四十多年历史，但仍历久不衰，始终是数值计算领域所使用的主要语言。Fortran 语言问世以来，根据需要几经发展，先后推出形成了很多版本。

第一代 Fortran 语言是在 1954 年提出来的，称为 FortranI。它于 1956 年在 IBM 704 计算机上得以实现。在此之前编写计算机程序是极为繁琐的，程序员需要详细了解为之编写代码的计算机的指令、寄存器和中央处理器 (CPU) 等方面的知识。源程序本身是用数学符号 (八进制码) 编写的，后来采用了助记符，即所谓机器码或汇编码，这些编码由汇编程序转换为指令字。在 50 年代书写和调试一个程序要很长时间，因为用这种方式编写程序显然是很不方便的，尽管它能使 CPU 高效地工作。正是这些原因，促使由 John Backus 率领的 IBM 公司的一个小组研究开发最早的高级程序设计语言 Fortran。其目的是开发一种容易理解、简单易学又能几乎像汇编一样高效运行的语言，他们取得了极大的成功。Fortran 语言作为第一种高级语言不仅是一次创新，也是一次革命。它使程序员摆脱了使用汇编语言的冗长乏味的负担，而且它使得不再只是计算机专家才能编写计算机程序，任何一名科学家或工程技术人员，只要稍加努力学习和使用 Fortran，就能按自己的意图编写出用于科学计算的

程序。

经过不断发展, FortranI 形成了很多不同版本, 其中最为流行的是 1958 年出现的 FortranII, 它对 FortranI 进行了很多扩充(如引进了子程序), FortranII 在很多机器上得以实现。其后出现的 FortranIII 未在任何计算机上实现。1962 年出现的 FortranIV 对原来的 Fortran 作了一些改变, 使得 FortranII 源程序在 FortranIV 编译程序下不能全部直接使用, 导致了语言不兼容的问题。这样就形成了当时同时使用 FortranII 和 FortranIV 两种程序设计语言的局面。

正因为 Fortran 满足了现实的需要, 所以它传播得很快, 在传播和使用过程中不可避免地产生了多种版本。各种 Fortran 语言的语义和语法的规定又不完全一致, 这给用户带来了极大的不便。用户迫切希望有能在各种机型上能互换通用的 Fortran 语言。因此 Fortran 语言的标准化工作变得十分迫切。1962 年 5 月, 当时的美国标准化协会 ASA(American Standard Association)(后来改名为 ANSI—American National Standards Institute, 现名为 NIST—National Institute of Standards and Technology)成立了工作组开展此项工作, 1966 年正式公布了两个美国标准文本: 标准基本 Fortran X3.10-1966(相当于 FortranII)和标准 Fortran X3.9-1966(相当于 FortranIV)。

由于 Fortran 语言在国际上的广泛使用，1972 年国际标准化组织(International Standard Organization、简称 ISO)公布了 ISO Fortran 标准，即《程序设计语言 FortranISO 1539-1972》，它分为三级，一级 Fortran 相当于 FortranIV，二级 Fortran 介于 FortranII 和 FortranIV 之间，三级 Fortran 相当于 FortranII。

FortranIV(即 Fortran66)流行了十几年，几乎统治了所有的数值计算领域。许多应用程序和程序库都是用 FortranIV 编写的。但很多编译程序并不向这一标准靠拢，它们往往为实现一些有用的功能而忽略标准；另外，在结构化程序设计方法提出以后，人们开始感到 FortranIV 已不能满足要求。FortranIV 不是结构化的语言，没有直接实现三种基本结构的语句，在程序中往往需要用一些以 GOTO 语句以实现特定的算法；而且为了使非标准的 Fortran 源程序能够交换移植，产生了“预处理程序”，通过预处理程序读入非标准的 Fortran 源程序，生成标准的 Fortran 文本，从而实现了源程序的交换移植，但这种自动生成的 Fortran 程序通常让人难以理解。

b) Fortran77    Fortran90

美国标准化协会在 1976 年对 ANSI X3.9-1966 Fortran 进行了修订，基本上把各厂家行之有效的功能都吸收了进去，此外又增加了不少新的内容，1978 年 4 月美国标准化协会正式公布将它作为美国国家标准，即 ANSI X3.9-1978 Fortran，称作 Fortran77。1980 年，Fortran77 被接受为国际标准，即《程序设计语言 Fortran ISO 1539-1980》，这种新标准并不是各非标准 Fortran 的公共子集，而是自成一体的新语言。我国制订的 Fortran 标准，基本采用了国际标准(即 Fortran77)，于 1983 年 5 月公布执行，标准号为 GB3057-82。Fortran77 还不是完全结构化的语言，但由于增加了一些结构化的语句，使 Fortran77 能用于编写结构化程序。此外，还扩充了字符处理功能。使 Fortran 不仅可用于数值计算领域，还可以适用于非数值运算领域。

因为 Fortran77 有着明显的局限性，为了引入一些新的功能，适应语言的发展，ANSI 在 80 年代初期开始准备制定 Fortran8x 标准。当初为了与前一标准相对应，设想是  $x=8$ 。由于要将 Fortran77 作为一个子集，同时又要确保程序的高效率，其标准化的工作花了十几年，最终在 1991 年通过了 Fortran90 新标准 ANSI X3.198-1991，相应的国际化标准组织的编号为 ISO/IEC1539:1991。新的 Fortran 标准废弃了过时的严格的源程序书写格式，改善了语言的正规性，并提高了程序的安全性，

功能有更大的扩充，是一个能适应现代程序设计思想的现代程序设计语言。为了保护对 Fortran77 用户在软件开发上的巨大投资，整个 Fortran77 被作为 Fortran90 的一个严格子集。

### 1.1.2 Fortran 的新发展

随着其他程序设计语言的迅速发展，Fortran 语言不再是惟一适用的程序设计语言。然而，尽管在一些特殊领域，使用其他程序语言更为合适，但在数值计算、科学和工程技术领域，Fortran 仍具有强大的优势。其强大的生命力在于它能紧跟时代的发展，不断更新标准，每次新的文本推出都在功能上有一次突破性进展。Fortran90 不仅仅是将已有的语言进行标准化，更重要的是发展了 Fortran 语言，吸取了一些其他语言的优点。所以，虽然 Fortran 语言历史悠久，但仍在日新月异地发展。

随着巨型计算机(向量机和并行机)的异军突起，出现了新的高性能 Fortran 语言(HPF)，它是 Fortran90 的一个扩展子集，主要用于分布式内存计算机上的编程，以减轻用户编写消息传递程序的负担。HPF-1.0 的语言定义是在 1992 年的超级计算国际会议上作出的，正式文本是在 1993 年公布的。其后几年的会议上又对它进行了修改、重定义、注释等工作，于 1997 年发布了 HPF2.0 语言定义。

Fortran95 包含了许多 HPF 的新功能。在 Fortran90 出现之前，在并行机上运行程序需要结合专门的矢量化子程序库，或者是依赖 Fortran 编译系统进行自动矢量化。而 Fortran90 之后，程序员在编程时可有目的的控制并行化。

在当前程序设计语言层出不穷的今天，学习 Fortran 语言的意义在于继承传统和紧跟时代。不仅一些爱好者推崇 Fortran 语言 [\[A Real Programmer\]](#)，而且科学计算编程的专家也认为，科学与工程相关专业的学生应该采用 Fortran 而非 C 和 C++ 编程 [\[F90 for Science Student\]](#)。这是因为，Fortran90 具有 C++ 所有的重要功能（尚不具备的预计将在 Fortran2k 版本中推出），然而 C 语言主要是用于微机上的廉价开发，而 Fortran 的目的是为了产生高效最优化运行的可执行程序，用 Fortran 编写的大型科学计算软件较 C 语言编写的通常要快一个量级，其程序编写更为自然和高效，且易学易懂。尤其是在高性能并行计算逐渐成为时代必然的今天，不仅巨型机而且微机和工作站也有了多处理器，其串行机上的线性内存模式已不再适用，而只有 Fortran 具备处理相应问题的标准并行化语言，其独特的数组操作充分体现了它的先进性。Fortran90 和 C++ 语言的比较可参看：<http://csepl.phy.ornl.gov/csep.html>。



### a) Fortran77 ?

既然已经有了 Fortran90, 那么是否就不用学习 Fortran77 了? 事实上, 由于很多用户在 Fortran 程序上作了巨大的投资, 许多大型科学计算 Fortran 程序(有些长达数十万条语句), 如分子动力学模拟计算(C60-C240 的碰撞: [10eV](#), [100eV](#), [300eV](#)) 等程序仍在频繁地使用。在科技领域内某些标准程序库 (International Mathematics and Statics Library, Numerical Algorithms Group) 内有数千以上的子程序是用 Fortran 写的, 特别是早期的程序都是用 Fortran77 编写的, 这些程序库已通过长期使用验证了稳定性。科学研究经常需要使用或改编以前的程序, 这时必须了解 Fortran77 的编程手法。

因此, 本教程仍然将 Fortran77 作为基础, 但随时与 Fortran90 比较不同之处。

### b) Fortran90 !

Fortran90 并没有删去任何 Fortran77 的功能, 而只是将某些功能看成是将要摒弃的。在 Fortran95 中则是已被删去的, 但考虑到历史, 厂家推出的 Fortran90/95 编译软件仍是支持这些功

能的。在新编的程序中，应尽量避免使用过时的 F77 语句或功能。目前已有一些软件可将这些功能除去并自动转换成并行化的程序。Fortran90/95 是具有强烈现代特色的语言，总结了现代软件的要求与算法应用的发展，增加了许多现代特征的新概念、新功能、新结构、新形式。Fortran90 的现代特性表现在：

加强了程序的可读性、可维护性：淘汰所有转移语句，用新的控制结构实现选择分叉与重复操作，使程序结构化。同时增加了结构块、模块及过程的调用灵活形式，使程序易读易维护，新的模块装配取代了 Fortran77 的许多旧语句，使程序员更为清晰明确地定义全局数据。增加了新的数据种别说明，使得 Fortran 程序在不同计算机编译环境下有更自由的移植性。

发展了现代算法功能：加强了数组的算法功能，引进了多种数组操作功能与概念，使数组像一个变量一样自由操作，使数组的并行化运算成为可能。增加了适于操作数据结构的派生类型，提高了文字处理功能，胜任信息管理系统、办公自动化的任务。特别是动态存储功能的引进极大地加强了它在数值计算领域中应用的威力。

扩大与编程者的友好界面：新的编程形式减少了烦琐与格式束缚，接近自然语言与公式演算。允

许在字符数据中选取不同种别，在字符串中可使用各国文字（例如汉字），还可任意使用化学、物理、数学的各种专业字符。

## Fortran 程序简例

### 1.2.1 编程实例

为了对 Fortran 程序有一个初步了解，下面先介绍几个简单的 Fortran 源程序。

#### c) 基本语句

[例 1.1] 输入两个数，求算术平均和几何平均值。[[e 121 01. f](#)][[e 121 01. f90](#)]

[计算例]

1.0    2.0

←键盘输入(a, b)值

1.500000          1.414214

←计算结果输出至屏幕

程序说明：

程序中第 1 行是注释行，对程序起说明作用。F77 注释行是以 “C” 或 “\*” 作为该行第一个字符的，F90 可在任意一行末以 “!” 开始作为注释符。第 2 行是主程序名，第 3 行是变量类型定义，第 4 行是输入语句，第 5—6 行是赋值部分，第 7 行是打印输出语句，最后是程序结束。注意 F77 的固定书写格式和 F90 的自由格式。F90 中用 “;” 将两行并为一行。

**PROGRAM 语句：** 宣布程序开始，其后跟程序名。可省略。

**REAL 语句：** 定义 a,b,...等变量为实数型数据。如为整数型，则用 **INTEGER** 定义。

**变量名：** 可用 a,b 等无具体意义的文字，或用 **average\_value** 等英文缩写。使用的字符和文字长度有具体规定。

**输入输出语句：** 可以用以下任一种默认格式，注意星号(\*)和逗号(,)。星号意指默认的输入输出硬件(键盘和屏幕)及格式。

READ \*, 变量名

READ(\*,\*) 变量名

PRINT \*, 变量或常数, 关系式

WRITE(\*,\*) 变量或常数, 关系式

**赋值语句：** 将等式右边的变量取值赋予左边的变量。

变量 = 变量、常数、关系式

例：sum = sum + x

**算术运算：**

2 项运算：加(+), 减(-), 乘(\*), 除(/), 乘方(\*\*)。

运算的优先顺序：加, 减 < 乘, 除 < 乘方, 括号中优先( )。

单项运算：(例 -a)

**数据类型：** 按定义有整数、实数、复数、双精度数等。

d) 输出字符

[例 1.2] 输入圆锥底面半径  $r$  和高  $h$ ，求体积和表面积。[[e 121 02. f90](#)]

[计算例]

Input radius  $r$  and height  $h$  ? ← 提示待输入数据的物理含义

3.0 5.2

Volume = 49.00885 ← 打印计算值

Area = 84.85442

**数据的输入：**READ \* 语句执行时进入等待数据输入的状态。数个数据输入时以英文逗号、空格或换行符作区别，单个数据中间不能有空格。

**字符的输出：**在执行 READ 语句时，计算机已经进入等待数据输入的状态，但它不会给出任何提示。除了程序员以外，谁也不会知道需要输入什么数据，即使是程序员自己可能也会忘记。另外，PRINT 语句打印的数据到底是什么物理含义也需指明。因此，需要输出文字内容。

PRINT \*, 字符常量

WRITE(\*,\*) 字符常量

字符常量： 用' '或" "括起来的文字字符。

[例 1.3] 已知放射性元素的半衰期，求给定时间后的衰减量。[[e 121 03. f90](#)]

[例 1.4] 给定一整数，求其自平方至 5 次乘方的各次乘方。[[e 121 04. f90](#)]

[例 1.5] 函数的计算。[[e 121 05. f90](#)]

[例 1.6] 分别求半径  $R=1, 3, 12.5$  时的圆周长。此源程序由二部分组成：主程序和子程序。  
[[e 121 06. f90](#)]

### 1.2.2 Fortran 程序的特点

从以上例子中可以看到：

- 一个 Fortran 程序由一个或若干个程序单位组成。主程序和辅程序分别是一个独立的程序单位。主程序单元起整体控制作用，各辅程序单元完成总问题中的一个子问题。
- 每一个程序单位都是以 END 语句结束的。END 既是一个程序单位的结束标志，又是一个独立的语句(结束语句)。主程序中的 END 语句的作用是使程序“停止运行”。辅程序中的 END 语句是“使流程返回调用程序”。
- 一个程序单位包括若干行。
  - F77 行分为下面两类。F77 规定，一行只能写一个语句，一行中不能写几个语句。如果一个语句太长，一行内写不下的话，可以写在继续行(要用“续行标志”)
  - ✦ 语句行。由一个 Fortran 语句组成， Fortran 语句分为执行语句和非执行语句。执行语句使计算机在运行时产生某些操作，如赋值语句、打印语句等。非执行语句(包括说明语句，数据语句等)将有关信息通知编译系统，以便在编译时作出相应的处理，例如类型说明语句、函数子程序语句等。Fortran 程序的基本成份是语句。
  - ✦ 非语句行，即注释行。它不是 Fortran 语句，它不被翻译成机器目标指令。不产生任何机



器操作。它仅仅是为了人们阅读程序的方便而加到程序中的。一个程序中注释行的数目不受限制，根据需要而定。但一个程序单位不能只由注释行组成。注释行的内容完全是根据程序设计人员需要而写的，一般是为程序(或程序中一部分)的作用作注释以易于理解程序。

- F90 行不分类。注释可以写在任一行末尾，而且一行不限语句数，可以将几个 F77 行合并写入一行。这样，极大地简化了程序写法，使得程序可以编写得更为清晰明了。因此 F90 的格式较 F77 有柔软性。
- Fortran 程序中的语句可以有标号。一个语句有否标号是根据需要而定，其作用是标志一个语句以便被其它语句引用。在同一个程序单元中不能有两个相同标号的语句。标号不影响语句的执行顺序。但在 F90 中因为提倡结构化程序设计，一般不使用标号。
- 一个程序单位中各类语句的位置是有一定规定的。例如 PROGRAM 语句应是主程序的第一个语句。FUNCTION 语句是函数子程序的第一个语句，END 语句只能是程序单位中最后一行。程序中语句执行的顺序一般依照它们在程序中的先后位置而定。

- F77 源程序必须按固定格式书写，即源程序中哪些内容应写在一行中的哪一列(或哪几列上)有严格的规定。而 F90 可采用自由格式。

## Fortran 程序的基本组成

### 1.3.1 字符集

不是任何一种外文字母或数字符号都能被某一计算机语言接受。每一种计算机的高级语言分别规定了它允许使用的字符。

Fortran 允许使用的字符如下：

英文字母 : A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

阿拉伯数字 : 0 1 2 3 4 5 6 7 8 9

特殊符号 : 空格 = + - \* / ( ) , . ' : " ! % & ; < > \$ ? \_ (F90 中新增的字符)

其中“\$”和“?”号在程序中没有确切意义。应当注意，在 F77 语句中不区分大小写字母，例如

写 READ 和 read 或 Read 是一样的,其它变量名和函数名中,大、小写字母也是等价的。但早期的 Fortran 卡片不允许小写,故老的程序代码都是大写的,很多人也养成了大写的习惯。

在每一种计算机系统所用的 Fortran 编译器中,可能对字符集或其功能作某些扩充。因此在使用某一具体的计算机时,应了解它的规定。例如“\$”号可用于从屏幕上连接输出输入字符于同一行。标准还允许 F90 扩充进各国文字,各种专业用符号,这要看厂商装入的编译系统是否支持这些字符的使用。某些系统还可以用这些字符作为字符常数或注释,注意这类字符是 2 字节长度,在计算字符串长时要加倍。

### 1.3.2 源码格式

#### e) 固定格式

Fortran 作为历史上第一种高级语言,其程序编写的规则与当时使用的计算机系统有很大的关系。早期的计算机系统须用卡片将源程序输入,一张卡片相应于源程序的一行。由于卡片物理宽度的限制,因此一行程序允许的字符数也是有限的。由此决定了 F77 的格式,既为 F90/95 中的固定格式。

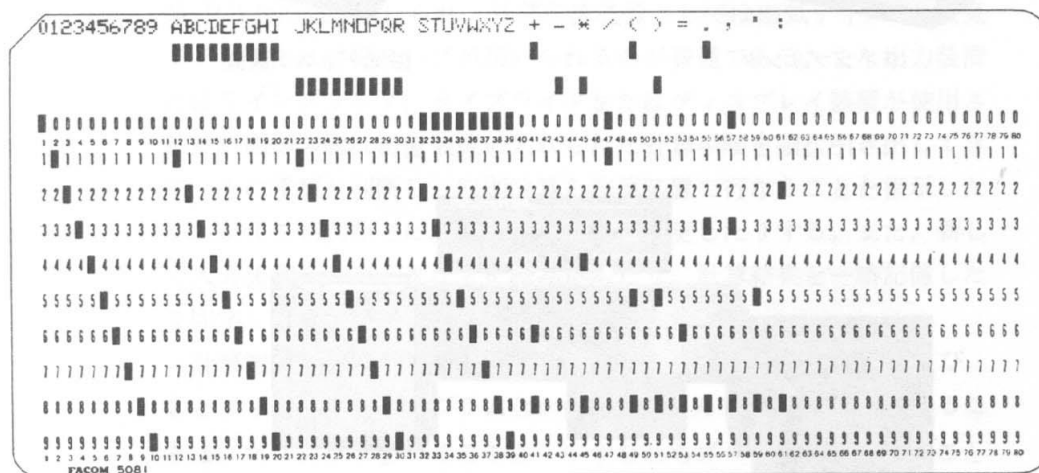


图 1-1 Fortran 卡片。

一张卡片有 12 行 80 列。其中有 10 行分别印有 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 十个数字。第 0 行上面的第一行作为第 11 行，再上面一行为第 12 行。一个字符由相应一列上的 1—3 个孔个孔来代表。如源程序某行第 11 列为字符“A”，则在该列上第 12 行和第 1 行上各穿一个方孔。注意此卡片允许的特殊字符只有十一个。

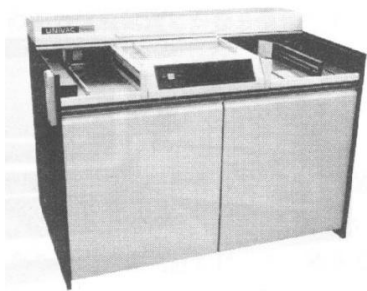


图 1-2 卡片输入机。

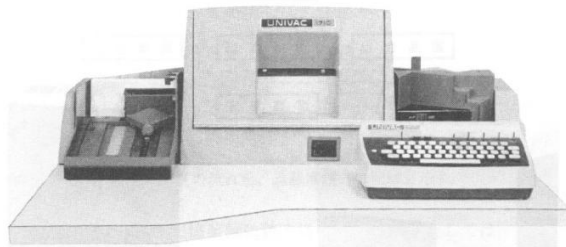


图 1-3 卡片穿孔机。

实际操作时利用卡片穿孔机，按下卡片穿孔机上某一个字符，则机器就会自动将卡片相应列中的相应位置穿上孔。如果一个源程序有 100 行，则需要 100 张卡片。在上机算题时，将穿好孔的卡片按顺序放入卡片输入机，启动机器，就将卡片上的信息输入计算机了。



标号区内不得出现标号以外的内容，但注释行例外。注释行的内容可以写在标号区内，一行中第一列为 C 或\*的，该行即被认为注释行，编译时对该行内容不作翻译，对程序运行不产生任何影响。如果在第一列上出现的不是数字、空格或 C 和\*的字符，编译时按出错处理。

### ● 续行区：第 6 列

如果在一行的第 6 列上写一个非空格和非零的字符，则该行作为其上一行的续行。注意在某些系统中，这个字符可以不限于上面所列的，如 @ } ] ~ 等字符均可使用 [[e 132 01.f](#)]。F77 允许一个语句有 19 个续行(即一个语句最多可以写成 20 行)。有的程序中第 6 列上用“1”，“2”，…表示该行是第 1 个或第 2 个续行，但用数字字符容易与第 7 列的数字形成连续的数字串而引起错觉，故最好使用固定的特殊字符。

### ● 语句区：第 7—72 列

不要求一定从第 7 列开始写语句，可以从第 7 列以后（72 列以前）的任何一列开始写，但一行只能写一个语句。如果写满了 72 列，一旦在终端上修改程序时在该行又插

入了一些字符，就会使本行最后几个字符超出语句区而引起意料不到的错误。特别注意  
到语句最后的空格将可能溢出 72 列，在某些计算机系统上将导致难以查出的错误。应  
注意，引号内的字符串中所包括的空格是有效的，不能忽略[e 132 02.f]。

### ● 注释区：第 73—80 列

在卡片输入法时代，程序员一般利用此 8 列为程序行编序号以便查找。注释区只对  
程序员提供辨别信息，不是语句的一部分，在编译时不对 72—80 列作处理[e 132 03.f]。

### f) 自由格式

在 F90 中，用自由格式编写程序有很大的自由度。与固定格式相比，不易产生键入位置的错误，  
而且易读易懂。

行：一行为 132 列。可以有复数条程序语句，语句间用分号“;”分开。语句没有位置规定。

注释行：起始用注释符“!”号，此行其后的所有字符均作为注释（字符串中的!号除外，如 print  
\*, 'help!!!!' ）。



**续行** : 当一个语句非常长以至于 132 列都书写不下时, 允许有 39 个续行。在语句行最后加上续行符 “&” 号。如果字符串跨 2 行以上, 则在续行的开始位置也要加&号。注意语句的有效字符是从 “&” 前和续行符 “&” 之后的位置算起。[[e 132 04. f90](#)]

**空格** : 在语句名和变量名中间不能有空格。需要空格的地方必须有一个以上空格 (GOTO 和 GO TO, ELSEIF 和 ELSE IF, END 构造名 (构造名有: DO, PROGRAM, FUNCTION, MODULE, SUBROUTINE) 等有两种写法的除外), 如关系运算符 ==, <= 不能写成 = =, < =。这与 F77 有很大不同之处, 因为 F77 的设计中将编译源程序的空格忽略 [[e 132 05. f](#)] [[e 132 05. f90](#)] [[e 132 06. f](#)]。

当程序员要将自己编写的 F90 程序与现有的用 F77 编写的子程序库在源码级结合起来的话, 需要特别注意格式的差别。

### g) 文件名

以上两种格式的源程序在编译时可以用选项来指定, 对应的默认文件扩展名为:

✚ 固定格式: .for 或 .f [fixed.for]

✚ 自由格式: .f90 [free.f90]

### 1.3.3 程序组成

#### a) 程序总体构造

Fortran90 程序是一种分块形式的程序，整个程序由若干个程序模块组成。各模块都有相似的语句组织形式，其中主程序起整体控制作用，各辅程序模块各自完成问题中的一个算法。在解决一个比较复杂的问题时，先把求解的问题分解为若干相对独立的子算法，每一个子算法编为一个辅程序，然后按搭积木一样将各有关程序模块组成一个程序。主程序依次调用各辅程序模块，控制各子算法的实施，通过主程序对子程序的调用，形成程序的整体运行，完成问题的解。

若某一子程序算法仍复杂，可再把它分解为若干更小的算法，分别编写为更低一层次的辅程序，由其他辅程序分别去调用。这种情况可以类推到其它子程序或更低一层次。所以，Fortran 程序 = n 个程序单位 = 1 主程序单位 + (0~n-1) 个子程序单位。在最简情况下，Fortran 程序只由一个主程序

构成而没有辅程序，所有算法都由主程序自身完成。按现代要求，即使功能比较简单的问题，也最好写成主程序调用辅程序的形式，以便于维护。

F77 中定义的辅程序对所有其它辅程序都是公开的，即除自己本身以外都可以加以引用。而在 F90 中，辅程序可以被本身应用，且可以定义不能被其它辅程序应用的内部辅程序。

## b) 程序单位

### 主程序

[PROGRAM 程序名]                      ←语句可省略

.....

END [PROGRAM [程序名]]              ←END 必须有

### 辅程序(过程)

SUBROUTINE 子程序

FUNCTION 函数

BLOCK DATA 块数据

MODULE 模块 (F90)

内部过程 CONTAINS (F90)

程序单位是 Fortran 中的基本成分，包括主程序、辅程序、块数据单元和内部过程。辅程序可以是函数辅程序或子程序辅程序。模块中包含可由其他程序单位访问的各种实体。块数据单元用来对有名公用块的数据对象规定初始值。一个执行程序总是由一个主程序单位和任意个(可以为零)其他类型的程序单位组成，任何程序单位都不能调用主程序。在主程序中定义的内部过程必须跟在 CONTAINS 语句之后，主程序是它的内部程序的宿主。过程包括函数和子程序，对一个子程序的引用是通过 CALL 语句或定义的赋值语句来引用的。

MODULE 过程在主程序中通过 USE 语句与之相联系。模块用于组装若干功能(如过程、类型定义、语义扩展等)为一集团，是 F90 中极具柔软性的程序单位，它取代了 F77 中的某些不安全的特性。

BLOCK DATA 辅程序的用处是定义全局常数或全局初始化，在 F90 中是不推荐使用的，其功能已被 MODULE 和 USE 取代。

### c) 程序体和语句顺序

各程序单位(除模块程序单位外)的程序体形式相同，共分两部分：前面是说明部分，后面是执行部分(模块程序单位只有说明部分)，两部分之间没有确切的分界，紧密衔接，但不准彼此穿插。即：  
程序单位 = 单位起始语句 + 程序体 + 单位结束语句。程序体 = 说明部分 + 执行部分。

Fortran 要求严格的语句顺序。在每个程序单位中，根据语句种类按如下的次序排列。F77 中的顺序可简单归纳为：

- ✦ PROGRAM, FUNCTION, SUBROUTINE, BLOCK DATA 等程序单位开始语句
- ✦ 变量类型和语句函数等定义语句 (说明部分)
- ✦ 执行语句, DATA 语句, FORMAT 语句 (执行部分)
- ✦ END 语句

在 F90 中将新增加的语句功能综合后，其顺序为：

PROGRAM, FUNCTION, SUBROUTINE, MODULE, BLOCK DATA		
USE		
FORMAT ENTRY	IMPLICIT NONE	
	PARAMETER	IMPLICIT
	PARAMETER & DATA	定义：派生类型、接口块、变量类型、语句 函数
	DATA	执行结构
CONTAINS		
内部过程或模块过程		
END		

整个程序中只能有一个 PROGRAM，一个 BLOCK DATA，可以有多个各自命名的 FUNCTION 和 SUBROUTINE 辅程序。

#### d) 英文名

变量名和程序名等使用的英文名是由(F77: 6; F90: 31)个字符(包含英文字母 A-Z、数字 0-9、F90:下划线 \_)构成, 且第一个字符必须是字母。其英文名的有效使用范围原则上限于一个程序单位之内(内部过程除外)。如下面的语句是错误的:

INTEGER :: 1A                   ! 不是以字母开头

INTEGER :: A\_name\_made\_up\_of\_more\_than\_31\_letters   ! 太长

INTEGER :: China:0           ! 含有不允许的字符

INTEGER :: A-3               ! 减号在此是无意义的

Fortran 没有规定保留字, 即可以用函数名或语句定义符作变量名。但为了避免混淆, 建议不要使用 Fortran 中已有特定含义的字作变量名。如 SIN 是正弦函数的名字, 如果有以下语句:

SIN=3.5

PRINT \*, SIN

则语句中的 SIN 是变量名而不代表正弦函数。系统会根据它后面有无自变量而作出判断, 又如:

```
READ *, PRINT
```

此时 PRINT 是一个变量名而不代表“打印输出”的操作。系统会认定语句的第一个字 READ 为代表操作的语句定义符，而把 PRINT 作为 READ 语句中读数的变量。但在同一个程序单位(主程序或子程序分别是一个程序单位)中，变量名和函数名或语句定义符不能同名。以下是错误的：

```
SIN=3.5
```

```
A=SIN*SIN(2.0)
```

```
PRINT *, PRINT
```

#### e) 标号和标签

可在语句开头加上标号或标签，用于指定特定的语句。有效范围限于一个程序单位内。

**标号：**1-5 位的 10 进制整数，且至少一位不为 0，前导 0 不起作用。F90 中，不可对空语句加上语句标号。

**标签：**英文名，后面接“:”（冒号） (F90)



例：

```
DO 10 k = 1, 100
```

```
...
```

```
10 CONTINUE
```

注：不能使用整型变量来指定转向语句的标号。如下是错误的。

```
n = 10
```

```
GOTO n
```

例： (F90)

```
DO k = 1, 100
```

```
loop2: DO
```

....

END DO loop2 !已有标签名时不能省略标签名。

END DO

## 数学运算

### 1.4.1 常量和变量类型

#### f) 常量

常量是指其值始终不变的一些量。整型、实型、双精度和复型常量是算数型常量，也为常数。

**整型**：默认值为 4 字节（其它为 1，2，8 字节，Compaq Visual Fortran 允许在 Alpha 机上使用 8 字节整数）。4 字节 32 位(bit)中用一位存放数值的符号，其余为数本身(用二进制表示)。第 1 位为 0 表示“正”，“1”表示“负”。由于用有限的内存单元存储一个整数，因此整数的范围是有限的： $-2^{31}-2^{31}-1$ ，即-2147483648—2147483647 之间，大约为±21 亿。

11111111 11111111 11111111 11111111 = -1

11111111 11111111 11111111 11111110 = -2

10000000 00000000 00000000 00000000 = -2147483648

01111111 11111111 11111111 11111111 = 2147483647

例：12345678    -256    (1,000,000 或 125. 是错误的)

1234567890123\_8    (8 字节整数)    (F90)

912345678901\_k    (k 为定义精度的参量)    (F90)

**实型**：默认值为 4 字节（其它为 8 字节）。实数是有小数点的数，有效位数为 7 位，其绝对值的范围与计算机系统和精度型（单精度和双精度）有关。实数有两种表示形式：

✚ **小数形式**：即日常习惯使用的小数形式。

3.141592    -0.125    3.0    -2.    等

3.14159265358979\_8    (8 字节实数)    (F90)

2.7182818\_p (p 为定义精度的参量) (F90)

✚ **指数形式**: 用指数形式表示的实数由两部分组成, 即数字部分和指数部分。

7.8E-12     -0.125E5

0.125D+45(双精度, 8 字节)

在计算机内存中存储一个实数(不论是用小数形式表示或以指数形式表示)时情况与整数不同。实数在内存中一律以指数形式存放, 它由三个部分组成: (1)数符; (2)指数包括符号; (3)数字部分。数字部分最前面有一个隐含的小数点。用 4 个字节(32bit)来存储时, 1 位存储数的符号, 7 位存储指数部分, 24 位存储数字部分。由于存储指数部分和数字部分的值(bit)长是有限的, 因此一个实数的有效数字和数的范围都是有限的。

如果以 24 位来存储一个数, 它最大可以存储十进数为  $2^{24}-1$  (减 1 是因为规定小数点后第一个数字不能为零, 以便充分利用有限的 bit 位来存储有效数字), 即 16777215。也就是说可以存放 0—16777215 这个范围内的数, 超过这个范围的数是不能存储的。可见, 只能有 7 位有效数字

(16777215 虽然是八位数，但并不是所有 8 值数都能有效存储)。如果某计算机系统用  $n$  个二进制位来存储一个实数的数字部分，则该实数的十进制数的有效数字位数大体上略小于  $n/3$ 。

一个实数的范围也是有限的，这主要是由于存储实数的指数部分的位长是有限的。如果一个数的绝对值超过此范围，就会出现“溢出”，绝对值比上界大的称为“上溢”，系统对此按数据出错处理。比下界小的称为“下溢”，大多数计算机系统将该数据按零处理。[\[e 141 01.f\]](#) [\[e 141 02.f90\]](#)

**复 型**：默认值为  $2 \times 4$  字节（其它为 8 字节）。实部与虚部用括号围起来表示：(实数, 实数)。

**逻辑型**：默认值为 4 字节（其它为 1 字节）。其值只能为：.TRUE.（真） 和 .FALSE.（假）。

**字符型**：1 个字符为 1 字节（中文系统中为 2 字节）。用 ' '(F90: " ")围起来的字符串。

例: "I'm a boy." （长为 10 字节）

'I'm a boy.' （长为 10 字节）

**数 组**：这不是一种单独的类型，可把同类型的常量用一维维数括起来表示：(/常量, 常量, ...,

常量/ )。 (F90)

### g) 变量

变量是指在程序运行期间其值是可以变化的量。系统为程序中的每一个变量开辟一个存储单元，用来存放变量的值。

常量是分为类型的，而变量是用来存放常量的，因此变量也相应地区分为整型变量 **INTEGER**、实型变量 **REAL**、双精度变量 **DOUBLE PRECISION**、复型变量 **COMPLEX**、逻辑型变量 **LOGICAL**、字符型变量 **CHARACTER**。在程序中应当说明哪些变量是整型变量，哪些变量是实型变量。变量在内存中所占的字节数和数据存储形式与相应类型的常数相同。例如，实型变量一般占 4 个字节，按指数形式存放。在程序中规定变量的类型可以用以下几种方法。

**隐含约定**：Fortran 规定，凡以字母 I, J, K, L, M, N 六个字母开头的变量名，如无另外说明则为 **整型变量**。以其它字母开头的变量为实型变量。可以将这个隐含约定称为“**I—N 规则**”，表示用 I 到 N 之间的字母开头的变量为整型。例如：I, J, IMAX, NUMBER, LINE, JOB,

KI 为整型变量，而 A, BI, COUNT, AMOUNT, TOTAL, BOOK 为实型变量。

**类型指定：**如果想改变“I—N 规则”对变量类型的约束，可以用类型说明语句专门指定某些变量的类型。Fortran 中有六个类型说明语句：

(1)INTEGER 语句(整型说明语句)

(2)REAL 语句(实型说明语句)

(3)DOUBLE PRECISION 语句(双精度说明语句)

(4)COMPLEX 语句(复型说明语句)

(5)LOGICAL 语句(逻辑型说明语句)

(6)CHARACTER 语句(字符型说明语句)

**IMPLICIT 语句(隐含说明语句)指定：**可以用 IMPLICIT 语句将某一字母开头的全部变量指定为所需的类型，还可以用一个 IMPLICIT 语句同时指定几种类型。例如：

IMPLICIT INTEGER (A, C, T-V)

IMPLICIT REAL (I, J)

IMPLICIT INTEGER (A, B), REAL(I,K), INTEGER (X-Z)

注：(1) 以上三种方法中，以类型说明语句最优先，IMPLICIT 语句次之，“I—N 规则”的隐含约定级别最低。如下程序中 IMAX 变量为整型，而其它以 I、J 开头的变量为实型：

IMPLICIT REAL(I, J)

INTEGER IMAX

(2) 类型说明语句和 IMPLICIT 语句是非执行语句。

(3) 类型说明只在本程序单位内有效。

(4) IMPLICIT 语句和类型说明语句应该出现在本程序单位中的所有执行语句之前，其中 IMPLICIT 语句又应在所有的类型说明语句之前。



### 1.4.2 内在函数

用 Fortran 解题往往要用到一些专门运算。如求三角函数  $\sin x$ ， $\cos x$ ，开根  $\sqrt{x}$ ，绝对值  $|x|$ ，对数  $\ln x$ ，指数  $\exp(x)$ ，求一组数中最大数和最小数等。Fortran 提供了一些系统内在函数来完成这些运算。程序设计者不必自己设计进行这些运算的语句组（即程序段或子程序），只需写出一个函数的名字以及结出一个或若干个自变量，就可以得到所需的值。例如：

SQRT(4.0)  $\rightarrow \sqrt{4}$

SIN(2.0)  $\rightarrow$  2(弧度)的正弦值  $\leftarrow$  注意三角函数中的自变量单位为弧度！

EXP(3.5)  $\rightarrow e^{3.5}$

LOG(3.0)  $\rightarrow \ln 3$

常用的 Fortran77 函数如下表所示。

函数名	含 义	应用例子	相当数学上的运算
-----	-----	------	----------

ABS	求绝对值	ABS (X)	$ x $
EXP	指数运算	EXP (X)	$\exp(x)$
SIN	正弦值	SIN (X)	$\sin x$
COS	余弦值	COS (X)	$\cos x$
ASIN	反正弦	ASIN (X)	$\arcsin x$
ACOS	反余弦	ACOS (X)	$\arccos x$
TAN	正切	TAN (X)	$\tan x$
ATAN	反正切	ATAN (X)	$\arctan x$
LOG	自然对数	ALOG (X)	$\ln x$
LOG10	常用对数	ALOG10 (X)	$\log_{10} x$
INT	取整	INT (X)	$\text{int}(x)$ ，取 $x$ 的整数部分
MOD	求余	MOD (X1, X2)	$x - \text{int}(x_1/x_2) \cdot x_2$
SIGN	求符号	SIGN (X1 ,	$ x_1 $ ( 当 $x_2 \geq 0$ ) , $- x_1 $ ( 当 $x_2 < 0$ )

REAL	转换为实型	X2)	$\max(i_1, i_2, i_3)$
MAX	求最大值	REAL (I)	$\min(x_1, x_2, x_3)$
MIN	求最小值	MAX0 (I1 , I2, I3) MIN(X1, X2, X3)	



### 内在函数的执行方法

Fortran 将这些内在函数分别编成单个子程序，组成函数库，存在于外部介质(如磁盘)上。在完成源程序的编译之后，用 LINK 命令实现连接，即将已翻译成二进制指令的目标程序与函数库连接。也就是将程序中出现函数名的地方用函数库中相应的一组指令代入之，组成一个统一的“可执行目标块”。例如程序中出现一个 SIN 函数，在连接时就将一组求正弦值二进制指令直接插入到程序中出现 SIN 的地方。由于是插入到程序内部的，所以称为“内在函数”。

### ● 一个内在函数可要求一个或多个自变量

例如，SQRT 函数只能有一个自变量，而 MOD 函数要求两个自变量，MAX 和 MIN 函数要求两个以上自变量。注意当自变量个数规定为 2 个时，自变量的顺序不应任意颠倒，如 MOD(8, 3) 表示 8 被 3 除的余数，其值为 2，而 MOD(3, 8) 则表示 3 被 8 除的余数，其值为 3。当自变量个数为 >2 时，自变量的顺序无关。

### ● 函数的自变量是有类型的，函数值也是有类型的

例如 MOD(8, 3) 中自变量 8 和 3 是整型，函数 MOD(8, 3) 的值“2”也是整型，如果写成 MOD(8.0, 3.0)，自变量是实型的，函数值也是实型的，其值为 2.0。

### ● 函数名分为“通用名”和“专用名”

例如求余函数的“通用名”为 MOD，“专用名”有三个(MOD, AMOD, DMOD)。根据自变量的类型就自动确定了函数值的类型，如 MOD(8, 3) 的值为整型，而 MOD(8.0, 3.0) 的值为实型。当调用子程序时如果用内在函数作为自变量(实参)，必须使用该内在函数的专用名。

### 1.4.3 算术表达式

#### a) 算术运算符和运算优先级

FORTRAN 规定可以使用五种算术运算符号。它们是：

+ 表示“加”（或正号）

- 表示“减”（或负号）

\* 表示“乘”

/ 表示“除”

\*\* 表示“乘方”

请注意：乘号用“\*”表示，不能写成“×”，以免与字母“X”混淆，也不能用“.”代替乘号。

除号不能用“÷”号。两个运算符不能紧邻，如  $A*-B$  是不合法的，应写成  $A*(-B)$ 。

不同的运算符按以下优先级次序：加，减  $<$  乘，除  $<$  乘方，括号（ ）中优先。同一优先级的两个运算，乘方按“先右后左”，其它按“先左后右”原则。

例:  $x + y*z \rightarrow x + (y*z)$

$x*y**2 \rightarrow x*(y**2)$

$x/y/z \rightarrow (x/y)/z$

$x**y**z \rightarrow x**(y**z)$

$3+5-6.0*8.0/4**2 \rightarrow 3+5-48.0/16.0 = 5$

例:  $4b$  分之  $a \rightarrow a/(4*b)$  或  $a/4/b$ , 而  $a/4*b$  和  $a/4b$  是错误的。

如无自信时可用括号 ( ), 对运算时间几乎无影响。

## b) 算术表达式的含义和表示方法

所谓**表达式**是指一个或多个运算的组合。它是由 Fortran 的运算符和括号将各运算元素(包括常量、变量、函数、数组元素)连结起来的一个有值的式子。Fortran77 允许使用四种表达式, 即: 算术表达式, 关系表达式, 逻辑表达式, 字符表达式。

**算术表达式**中各运算元素都是算术量, 使用的运算符只能是算术运算符, 表达式的值也是一个算

术量(即数值)。例如，下面就是一个 Fortran 算术表达式：  $(A-B)/C**2+SIN(X+Y)$  。

请注意：

(1) 由于用 “/” 号作为除号，因此在写除法运算式子时应加上必要的括号。

(2) 乘号不能省略，如  $a \sin x$ ，必须写成  $A*SIN(X)$ ，而不能写成  $ASIN(X)$ 。

(3) Fortran 中无大、中、小括号之分，一律用小括号。

(4) 乘方按 “先右后左” 原则处理。

(5) 对单项运算符(如  $-A$  中的符号)相当于在它前面有一个运算量 “0”，如： $-A**2$  相当于  $0-A**2$  而非  $(-A)**2$ 。

Fortran 算术表达式的求值运算的优先次序为：① 括号 > ② 函数 > ③  $**$  > ④  $*$  / > ⑤  $+$   $-$ ，即括号内的运算级别最高，加减最低。例如， $SQRT(3.*2)**2 \rightarrow (\sqrt{6})^2$ 。

### c) 表达式运算中的类型

Fortran 中的常量和变量是分类型的，允许在不同类型的算术量(包括整型、实型、双精度、复

型)之间进行算术运算,但不允许在算术量和非算术量(如逻辑型、字符型)之间进行算术运算。F77作如下规定:

- 同类型的算术量之间运算的结果仍保持原类型。

特别要注意:两个整数相除的商也是整数。例如,  $5/2$  的值是 2 而不等于 2.5,  $4*(-1)$  等于 0,  $\sqrt[3]{5}$  应写为  $5**(1./3.)$  而非  $5**(1/3)$ 。

- 如果参加运算的两个算术量为不同类型,则编译系统会自动将它们转换成同一类型后进行运算。

转换的规律是:将低级类型转换成高级类型。类型的级别如下:整型(低)→实型(高)。类型的转换是从左向右进行的,在遇到不同类型的算术量时才进行转换。例如:  $1/2*1.0$  等于 0,而  $1./2*1$  对于 0.5。

#### d) 运算的误差

整型量的运算是准确的,没有任何误差(只要在整数范围内)。而实型量的运算会出现一些误差。



例如： $11111.1 \times 1111.11$  本应得 12345654.321，但由于受实数有效位数的限制，只能得到 12345654.0。

又如： $0.001 + 1246825.0 - 1246820.0$  本应得 5.001。但由于有效位数的限制，也将产生误差。

[e 143 01.f90] 为减小误差，在写表达式时应尽量使每一次运算结果都在有效位数范围之内，尽量不要使两个相差很大的数值直接相加或相减。在判断两个实数是否相等时，要特别慎重，最好改为判断  $|A - B| \leq \varepsilon$ ，即 A 和 B 之差的绝对值如果小于  $\varepsilon$  ( $\varepsilon$  为一个很小的数，如  $10^{-10}$ )，则认为 A 和 B 相等。

总之，在实数运算中应充分考虑到其可能出现的误差，而且在运算过程中误差会不断积累而增大，有时可能达到一个可观的程度。

#### 1.4.4 赋值语句

##### e) 算术赋值语句

赋值语句的作用是将一个确定的值赋给一个变量。其一般格式为： $V = e$ ，V 代表一个变量名，e 代表一个表达式。

✦ Fortran77 的赋值语句有三类：算术赋值语句、逻辑赋值语句、字符赋值语句。这里只讨论算

术赋值语句。算术赋值语句的作用是将一个算术量赋予一个算术型变量。

- ✦ 赋值语句中的“=”号是“赋值”的符号，而不是等号。它的作用是将赋值号右边的表达式的值赋给其左边的变量。例如：赋值语句“A=3.0”的作用是将3.0送到变量A中。因此在阅读程序时对赋值号的理解应是带方向的。赋值语句“N=N+1”的作用是将N的原值加1再送回变量N中。
- ✦ 算术赋值语句兼有计算和赋值双重功能。即先计算出表达式的值；然后将该值赋给一个变量。在Fortran程序中的求值计算主要是用赋值语句来实现的。
- ✦ 赋值号左边只能是变量名(或数值元素名)，而不能是表达式，赋值号右边可以是常量、变量或表达式。“X+Y=3.0”语句是不合法的，因为在内存中找不到一个“X+Y”的单元来存放3.0这个数值。

#### f) 执行时的类型转换

一个算术赋值语句中的被赋值的变量(V)和表达式(e)的类型可以相同，也可以不相同。F77作如

下规定：

- ✦ 如果变量  $V$  与表达式  $e$  的类型相同，则直接进行赋值。
- ✦ 如果类型不同，则应先进行表达式的求值，然后将该表达式的值转换为被赋值变量的类型。如：

$$I = 3.5 * 2.1$$

表达式  $3.5 * 2.1$  的值为 7.35，实型。而变量  $I$  为整型，因此  $I$  的值等于 7。又如：

$$T = 3 * 5 / 7$$

表达式的值为 2，整型。由于  $T$  为实型，故系统先将整数 2 转换成实数 2.0，再赋给变量  $T$ ， $T$  的值为 2.0。

当赋值号两侧的类型不同时，往往会产生程序设计者事先预想不到的结果。所以在编写程序时，应尽可能使赋值号两侧保持同类型。

## 第二章：改变程序流程

### 算法和流程图

#### 2.1.1 算法

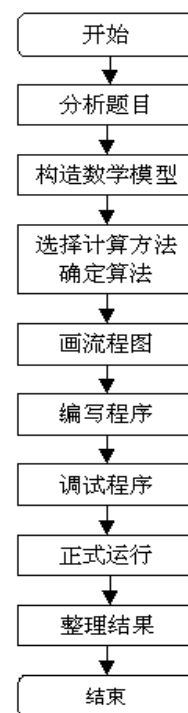
计算机语言只是一种工具。光学习语言的规则还不够，最重要的是学会针对各种类型的问题，拟定出有效的解决方法和步骤即算法。有了正确而有效的算法，可以利用任何一种计算机高级语言编写程序，使计算机进行工作。因此，设计算法是程序设计的核心。

并非只有“计算”的问题才有算法。广义地说，为解决一个问题而采取的方法和步骤，称为“算法”。不要把“计算方法”（computational method）和“算法”（algorithm）这两个词混淆。前者指的是求数值解的近似方法，后者是指解决问题的一步一步的过程。在解一个数值计算问题时，除了要选择合适的计算方法外，还要根据这个计算方法写出如何让计算机一步一步执行以求解的算

法。对于计算机外行来说，他们可以只使用别人已设计好的现成算法，只需根据算法的要求给以必要的输入，就能得到输出的结果。对他们来说，算法如同一个“黑箱子”一样，他们可以不了解“黑箱子”中的结构，只是从外部特性上了解算法的作用，即可方便地使用算法。但对于程序设计人员来说，必须会设计算法，并且根据算法编写程序。

对同一个问题，可以有不同的解题方法和步骤。例如，求  $1+2+3+\cdots+100$ ，可以先进行  $1+2$ ，再加  $3$ ，再加  $4$ ，一直加到  $100$ ，也可采取  $100+(1+99)+(2+98)+\cdots+(49+51)+50=100+50+49\times 100=5050$ 。还可以有其它的方法。当然，方法有优劣之分。有的方法只需进行很少的步骤，而有些方法则需要较多的步骤。一般说，希望采用方法简单，地进行解题，不仅需要保证算法正确，还要一个计算问题的解决过程通常包含下面几

● 确立所需解决的问题以及最后应达到  
始就对它有详细而确切的了解，避免模



运算步骤少的方法。因此，为了有效  
考虑算法的质量，选择合适的算法。  
步：

的要求。必须保证在任务一开  
棱两可和含混不清之处。

- 分析问题构造模型。在得到一个基本的物理模型后，用数学语言描述它，例如列出解题的数学公式或联立方程式，即建立数学模型。
- 选择计算方法。如定积分求值问题，可以用矩形法、梯形法或辛普生法等不同的方法。因此用计算机解题应当先确定用哪一种方法来计算。专门有一门学科“计算方法”，就是研究用什么方法最有效、最近似地实现各种数值计算的，换句话说，计算方法是研究数值计算的近似方法的。
- 确定算法和画流程图。在编写程序之前，应当整理好思路，设想好一步一步怎样运算或处理，即为“算法”。把它用框图画出来，用一个框表示要完成的一个或几个步骤，它表示工作的流程，称为流程图。它能使人们思路清楚，减少编写程序中的错误。
- 编写程序。
- 程序调试，即试算。一个复杂的程序往往不是一次上机就能通过并得到正确的结果的，需要反复试算修改，才得到正确的可供正式运行的程序。
- 正式运行得到必要的运算结果。

### 2.1.2 流程图

为了表示一个算法，可以用不同的方法。常用的有：自然语言；传统流程图；结构化流程图；伪代码；PAD 图等。这里我们主要介绍流程图。

## h) 传统流程图

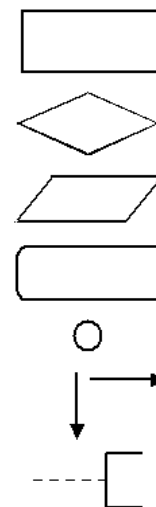
用图表示的算法就是流程图。流程图是用一些图框来表示各种类型的操作，在框内写出各个步骤，然后用带箭头的线把它们连接起来，以表示执行的先后顺序。用图形表示算法，直观形象，易于理解。

美国国家标准化协会 ANSI 曾规定了一些常用的流程图符号，为世界各国程序工作者普遍采用。最常用的流程图符号见图。

✦ 处理框（矩形框），表示一般的处理功能。

✦ 判断框（菱形框），表示对一个给定的条件成立决定如何执行其后的操作。它有

✦ 输入输出框（平行四边形框）。



件进行判断，根据给定的条件是一个入口，二个出口。

- ✦ **起止框**（圆弧形框），表示流程开始或结束。
- ✦ **连接点**（圆圈），用于将画在不同地方的流程线连接起来。如图中有两个以 1 标志的连接点(在连接点圈中写上“1” )则表示这两个点是连接在一起的，相当于一个点一样。用连接点，可以避免流程线的交叉或过长，使流程图清晰。
- ✦ **流程线**（指向线），表示流程的路径和方向。
- ✦ **注释框**，是为了对流程图中某些框的操作做必要的补充说明，以帮助阅读流程图的人更好地理解流程图的作用。它不是流程图中必要的部分，不反映流程和操作。

程序框图表示程序内各步骤的内容以及它们的关系和执行的顺序。它说明了程序的逻辑结构。框图应该足够详细，以便可以按照它顺利地写出程序，而不必在编写时临时构思，甚至出现逻辑错误。流程图不仅可以指导编写程序，而且可以在调试程序中用来检查程序的正确性。如果框图是正确的而结果不对，则按照框图逐步检查程序是很容易发现其错误的。流程图还能作为程序说明书的一部分提供给别人，以便帮助别人理解你编写程序的思路 and 结构。



例：对一个大于或等于 3 的正整数，判断它是不是一个素数。

所谓素数，是指除 1 和该数本身之外，不能被其它任何整数整除的数。例如，13 是素数，因为它不能被 2, 3, 4, ..., 12 整除。

判断一个数  $N(N > 3)$  是否素数的方法是很简单的：将  $N$  作为被除数，将 2 到  $(N-1)$  各个整数轮流作为除数，如果都不能被整除，则  $N$  为素数。算法可以表示如下：

- ① 输入  $N$  的值。
- ②  $I = 2$ 。
- ③  $N$  被  $I$  除。
- ④ 如果余数为 0，表示  $N$  能被  $I$  整除，则打印  $N$  “不是素数”，算法结束。否则继续。
- ⑤  $I = I + 1$ 。
- ⑥ 如果  $I \leq N - 1$ ，返回③。否则打印  $N$  “是素数”。然后结束。

实际上， $N$  不必被 2 到  $(N-1)$  的整数除，只需被 2 到  $N/2$  间整数除即可，甚至只需被 2 到  $\sqrt{N}$  之间的整数除即可。例如，判断 13 是否素数，只需将 13 被 2, 3 除即可，如都除不尽， $N$  必为素数。步骤⑥可改为：

⑥：如果  $I \leq \sqrt{N}$ ，返回③。否则算法结束。

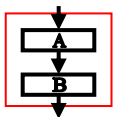
Fortran 代码文件为[\[e 212 01.f\]](#)[\[e 212 02.f\]](#)。

### i) 三种基本结构

传统的流程图用流程线指出各框的执行顺序，对流程线的使用没有严格限制。因此，使用者可以毫不受限制地使流程随意地转来转去，使流程图变得毫无规律，阅读者要花很大精力去追踪流程，使人难以理解算法的逻辑。如果我们写出的算法能限制流程的无规律任意转向，而像一本书那样，由各章各节顺序组成，那样，阅读起来就很方便，不会有任何困难，只需从头到尾顺序地看下去

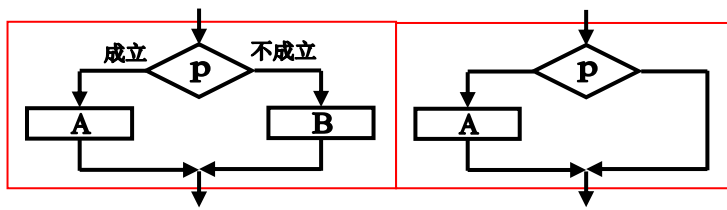
即可。

为了提高算法的质量，使算法的设计和阅读方便，必须限制箭头的滥用，即不允许无规律地使流程乱转向，只能按顺序地进行下去。但是，算法上难免会包含一些分支和循环，而不可能全部由一个一个框顺序组成。如上例不是由各框顺序进行的，包含一些流程的向前或向后的非顺序转移。为了解决这个问题，人们设想，如果规定出几种基本结构，然后由这些基本结构按一定规律组成一个算法结构，就如同用一些基本预制构件来搭成房屋一样，整个算法的结构是由上而下地将各个基本结构顺序排列起来的。1966 年，Bohra 和 Jacoplni 提出了以下三种基本结构，用这三种基本结构作为表示一个良好算法的基本单元。



● **顺序结构**：如图所示的虚线框内，A 和 B 两个框是顺序执行的。顺序结构是最简单的一种基本结构。

● **选择结构**：如图所示的虚线框中包含一个判断框。根据给定的条件 p 是否成立而选择执行 A 和

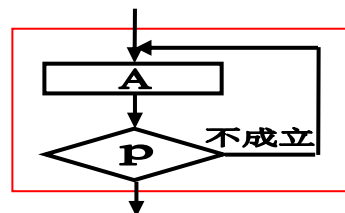


B。p 条件可以是“ $x > 0$ ”或“ $x > y$ ”等。注意，无论 p 条件是否成立，只能执行 A 或 B 之一，不可能既执行 A 又执行 B。无论走哪一条路径，在执行完 A 或 B 之后将脱离选择结构。A 或 B 两个框中可以有一个是空的，即不执行任何操作。

● **循环结构**：又称重复结构，即反复执行某一部分的操作。有两类循环结构：

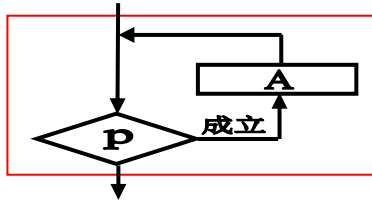
➤ **当型(While)**：当给定的条件 p 成立时，执行 A 框操作，然后再判断 p 条件是否成立。如果仍然成立，再执行 A 框，如此反复直到 p 条件不成立为止。此时不执行 A 框而脱离循环结构。

➤ **直到型(Until)**：先执行 A 框，然后判断给定的 p 条件是否成立。如果 p 条件不成立，则再

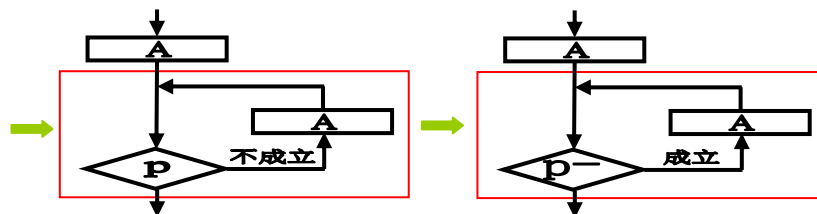


执行 A，然后再对 p 条件作判断。如此反复直到给定的 p 条件成立为止。此时脱离本循环

结构。



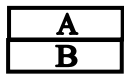
注意两种循环结构的异同：(1)两种循环结构都能处理需要重复执行的操作。(2)当型循环是“先判断(条件是否成立)，后执行(A 框)”。而直到型循环则是“先执行(A 框)，后判断(条件)”。(3)当型循环是当给定条件成立满足时执行 A 框，而直到型循环则是在给定条件不成立时执行 A 框。



同一个问题既可以用当型循环来处理，也可以用直到型循环来处理。对同一个问题，如分别

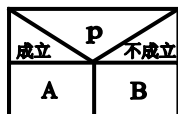
用当型循环结构和直到型循环结构来处理的话，则两者结构中的判断框内的判断条件恰为互逆条件。Fortran77 和 90/95 标准都不提供 do until 语句，Compaq Visual Fortran 也不提供此扩展（但有些计算机系统则提供），因此需要将直到型循环转换成一个当型循环结构：直到型循环等于一个 A 框加上一个当型循环，同时将给定的判断条件“取反”。

## j) 结构流程图



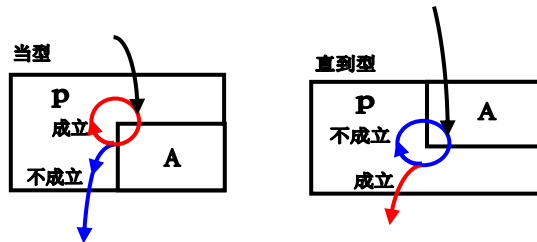
1973 年美国学者 I.Nassi 和 B.Shneiderman 提出了一种新的流程图形式。在这种流程图中，完全去掉了带箭头的流程线。全部算法写在一个矩形框内。在该框内还可以包含其它的从属于它的框，即可由一些基本的框组成一个大的框。这种适于结构化程序设计的流程图称 N-S 结构化流程图，它用以下的流程图符号：

(1)顺序结构：A 和 B 两个框组成一个顺序结构。



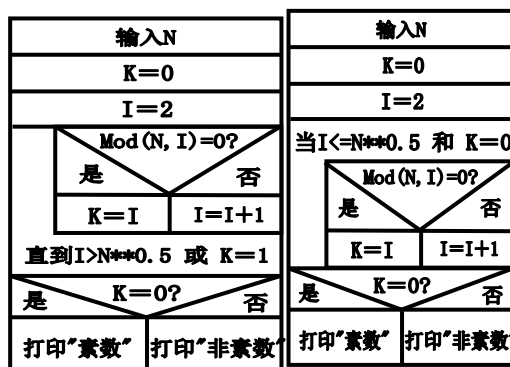
(2)选择结构：当  $p$  条件成立时执行  $A$  操作， $p$  不成立则执行  $B$  操作结构。

(3)循环结构：当型循环结构下，图符表示先判断后执行，当  $p$  条件成立时反复执行  $A$  操作，



直到  $p$  条件不成立为止。

直到型循环结构下，图符表示先执行后判断，当  $p$  条件不成立时反复执行  $A$  操作，直到  $p$  条件成立为止。



用以上三种 N-S 流程图中的基本框，可以组成复杂的 N-S 流程图，  
以表示算法。

例：将判别素数的算法用 N-S 流程图表示。

上面的非结构化流程图不是由三种基本结构组成的：图中间的循环部分有两个出口，不符合基本结构的特点。由于不能直接分解为三种基本结构，应当先作必要的变换再用 N-S 流程图的三种基本结构的符号来表示。即将第一个菱形框的两个出口汇合在一点。其方法是设一个标志值  $K$ ，它的初始状态为 0(表示  $N$  为素数)，当  $K \neq 0$  时为非素数。注意当型和直到型的判断条件。

Fortran 代码文件为[\[e 212 03.f90\]](#)。

N-S 图表示算法的优点是：比传统流程图紧凑易画，尤其是它废除了流程线。整个算法结构是由各个基本结构按顺序组成的，其上下顺序就是执行时的顺序。写算法和看算法只需从上到下进行就可以了，十分方便。归纳起来，一个结构化的算法是由一些基本结构顺序组成的；在基本结构之间不存在向前或向后的跳转，流程的转移只存在于一个基本结构范围之内(如循环中流程的跳转)；一个非结构化的算法可以用一个等价的结构化算法代替，其功能不变。如果一个算法不能分解为若干个基本结构，则它必然不是一个结构化的算法。

### k) 伪代码表示的算法



用传统的流程图和 N-S 图表示算法直观易懂，但画起来比较费事，在设计一个算法时，可能要反复修改，而修改流程图是比较麻烦的。因此，流程图适宜于表示一个算法，但在设计算法过程中使用不是很理想的(尤其是当算法比较复杂、需要反复修改时)。为了设计算法时方便，常用一种称为伪代码的工具。伪代码是用介于自然语言和计算机语言之间的文字和符号来描述算法。它如同一篇文章一样，自上而下地写下来。每一行(或几行)表示一个基本操作。它不用图形符号，因此书写方便、格式紧凑，易懂也便于向计算机语言算法(即程序)过渡。

可以用英文、汉字、中英文混合表示算法，以便于书写和阅读为原则。用伪代码写算法并无固定的、严格的语法规则，只要把意思表达清楚，并且书写的格式要写成清晰易读的形式。例如，对于电子在特殊几何构型材料中的散射问题：[\[sphere-1.doc\]](#)[\[sphere-2.doc\]](#)[\[sphere.f\]](#)

## 逻辑运算

## 2.2.1 关系表达式

### a) 关系运算符

关系运算符就是关系比较符。Fortran 中有六个关系运算符：

关系运算符		英语含义	所代表的数学符号
.GT.	>	Greater Than	> (大于)
.GE.	>=	Greater than or Equal to	≥ (大于或等于)
.LT.	<	Less Than	< (小于)
.LE.	<=	Less than or Equal to	≤ (小于或等于)
.EQ.	==	Equal to	= (等于)
.NE.	/=	Not Equal to	≠ (不等于)

			≠ (不等于)
--	--	--	---------

注意：关系运算符的两个字母(如 GT，LE…)的二侧各有一个点，不要漏写。

## b) 关系表达式

关系表达式是最简单的一种逻辑表达式。其一般形式为：

〈算术表达式〉 〈关系运算符〉 〈算术表达式〉

〈字符表达式〉 〈关系运算符〉 〈字符表达式〉

算术表达式也可以是一个算术量，即为一个数值常数、数值型变量、数值函数。

例：A+B <= 0.    A+B .LE. 0.    A\*B /= C/D    A\*B .NE. C/D

应当注意，关系表达式中关系操作符只准出现一次。数学中不等式写法与 FORTRAN 中关系表达式的写法有同有异，在编写关系表一定要严格服从它的一般形式。不可随意使用数学中的习惯用法，例如语句写法  $3 < X < 7$  就是不正确的关系表达式，因为它不符合关系表达式中只允许有一个关系操作

符的规定。

关系运算的特点是：

- ✦ 关系表达式中包括算术运算符和关系运算符，其运算的次序是先进行算术运算，然后进行关系运算。
- ✦ 不同类型的数值量进行比较时，系统会先将这两个量转换成同一个类型，然后再比较。转换的规律是将低级类型向高级类型转换。因此，执行关系表达式的过程为：算出算术表达式的值；将两个算术表达式值转换成同一类型；将两个表达式的值进行比较。
- ✦ 关系表达式的值不是一个数值，而是一个逻辑量。它的值是“真”和“假”二者之一。
- ✦ 由于实数的精度限制，因此用.EQ. (等于)和.NE. (不等于)运算符时要特别注意。要考虑到可能出现的误差，并设法弥补。如对数学式  $A=B$  的关系运算可用： $ABS(A-B).LT.1.E-30$ ；对  $A \neq B$  可用： $ABS(A-B).GT.1.E-30$ 。
- ✦ 除了算术量可进行比较外，字符关系表达式可对两个字符量进行比较。

## 2.2.2 逻辑表达式

### a) 逻辑量

Fortran 的**逻辑常量**只有两个：.TRUE.（表示“真”，即满足逻辑条件）；.FALSE.（表示“假”，即不满足逻辑条件）。

例：当  $X=3$  时，“ $X<5$ ”的值为.TRUE.（真），而“ $X\leq 0$ .”的值为.FALSE.（假）。

**逻辑型变量**用来存放逻辑常量。它的值也只能是.TRUE. 或.FALSE. 之一。可以将一逻辑常量赋予一个逻辑变量。如将 A 定义为逻辑型变量，用赋值语句对其赋值：

```
LOGICAL A
```

```
A=.TRUE.
```

逻辑赋值语句的一般形式为：**逻辑变量 = 逻辑表达式**。

### b) 逻辑运算符

Fortran 有 5 个逻辑运算符，每个操作符两边都有一点，不可省略：

逻辑运算符	含义	逻辑运算例	例子含义
. AND.	逻辑与	A. AND. B	A, B 为真时，则 A. AND. B 为真
. OR.	逻辑或	A. OR. B	A, B 之一为真，则 A. OR. B 为真
. NOT.	逻辑非	. NOT. A	A 为真，则 . NOT. A 为假
. EQV.	逻辑等价	A. EQV. B	A 和 B 值为同一逻辑常量时，A. EQV. B 为真
. NEQV.	逻辑不等价	A. NEQV. B	A 和 B 的值为不同的逻辑常量，则 A. NEQV. B 为真

注意不要将 . AND. 与 . EQV. 混淆：A. AND. B 是当 A 和 B 均为真时才为真；A. EQV. B 是当 A 和 B 均为

真或均为假时为真。

### c) 逻辑表达式的运算

下面是逻辑表达式的例子：

逻辑表达式	说明
(A. LT. B). AND. (A. GT . C)	当 $A < B$ 和 $A > C$ 时表达式值为真
(X. LT. 0. ). OR. (X. GT . 100. )	当 $X < 0$ 或 $X > 100$ 时，表达式值为真
. NOT. (X. LE. 0. )	当 $X \leq 0$ 时，表达式值为假
(A. GT. B). EQV. (C. GT . D)	当两个括弧内的值都为真或都为假时，表达式的值为真
L1. NEQV. L2	当 L1 与 L2 的逻辑值不相同，表达式的值为真

一个逻辑表达式中可以包括多个逻辑运算符，如逻辑表达式：  
A. GE. 0. 0. AND. A+C. GT. B+D. OR. . NOT. . TRUE. 中不仅有逻辑运算符，还有关系运算符和算术运算符。

Fortran 规定了以下的运算顺序：

- (1) 先计算算术表达式的值(例如上式中的 A+C 和 B+D)。
- (2) 再求关系表达式的值(例如上式中的 A. LE. 0. 0 和 A+C. GT. B+D)。
- (3) 最后进行逻辑运算，其顺序是：. NOT. > . AND. > . OR. > . EQV. 和 . NEQV. 。如果有括弧，则先进行括弧内的运算。

可用下表表示各种运算符的优先级别：

运算类别	运算符	优先级
括号	(    )	1
算术运算	**	2
	* /	3
	+ -	4



关系运算	. GT. . GE. . LT. . LE. . EQ. . NE.	5
逻辑运算	. NOT.	6
	. AND.	7
	. OR.	8
	. EQV. . NEQV.	9

### 2.2.3 逻辑 IF 语句

IF 语句不是一种选择结构，只是一条语句，它在算法较简单的场合下，可以灵活地完成二分叉选择算法。逻辑 IF 语句判别逻辑表达式的值是否为“真”，并执行一操作。其一般形式为：**IF(逻辑表达式) 执行语句**。如果条件成立（即逻辑表达式值是‘真’），则执行其后紧跟的执行语句，而后执行下一条语句；如果条件不成立，则整个 IF 语句不作任何操作，只是起下滑作用，使控制转移到 IF 语句的下一个语句。

IF 语句最大的用处是退出迭代。计算机程序中经常作连加、连乘或反复执行某段程序，并规定

只有当某个参数大于或小于某值时才停止循环，这时就可以使用 IF 语句。IF 语句用于两分叉选择的例子如：求数学中的阶梯函数（ $\theta(x) = 0, \text{ if } x \leq 0; = 1, \text{ otherwise}$ ），可写成：

```
Y=0
```

```
IF (X>0) Y=1
```

注意本例中不可写成：

```
IF (X<=0) Y=0
```

```
Y=1
```

例：IF (X>0. .AND. K\=N) Y=K+X

例：打印学生考试成绩，大于等于 80 分的为“A”，大于等于 60 分而小于 80 分的为“B”，小于 60 分的为“C”：

```
READ *, GRADE
```

```
IF (GRADE. GE. 80) PRINT *, " A"
```

```
IF (GRADE. GE. 60 .AND. GRADE. LT. 80) PRINT *, " B"
```

IF (GRADE.LT. 60) PRINT \*, " C"

逻辑 IF 语句当条件为“真”时只能执行一个执行语句而不能执行若干个语句。与块 IF 相比，逻辑 IF 语句是在一行中完成的一个选择操作，因此它又称“行 IF 语句”以与块 IF 区别。之所以用逻辑 IF 语句名称，是沿用了 Fortran66 的定义，因为在 F66 中有两种 IF 语句：算术 IF 语句(F90/95 中已废除)和逻辑 IF 语句。

逻辑 IF 语句也常与 GOTO 语句合用，但如果过多使用，程序结构将显得混乱，难于理解。应尽可能采用结构化的程序设计方法。[\[e 223 01.f\]](#)

## 选择结构

### 2.3.1 块 IF 构造

#### a) 块 IF 的组成和执行

块 IF 的一般形式可写成：

```
IF(逻辑表达式) THEN    ← 块 IF 语句
    块 1                ← then 块
ELSE
    块 2                ← else 块
END IF
```

块 IF 语句不是一个单独语句。块 IF 语句、ELSE 语句和 ENDIF 语句只能用在块 IF 中而不能单独使用，它们必须和块 IF 中其它语句联系起来共同起作用。可以说一个块 IF 是一个语句块(决不能只包括一个语句)，用来实现选择结构。但是，块 IF 可以不包含 ELSE 语句和 else 块。then 块是当块 IF 语句中的逻辑表达式为真时执行的语句组。[\[e 231 01.f90\]](#)

块 IF 的执行步骤为：先执行块 IF 语句，求出逻辑表达式的值，如果此值为“真”，则将流程转到 then 块。执行 then 块中各个执行语句。执行完 then 块后跳过 ELSE 语句和 else 块，流程转到 END IF 语句处。如果逻辑表达式的值为“假”，则流程跳过 then 块，转到 ELSE 语句及 else 块。执行完

else 块以后流程转到 END IF 语句。END IF 语句是块 IF 的“出口”，无论执行完 then 块或 else 块，流程都转到 END IF 语句，接着执行 ENDIF 语句之后的下一语句。END IF 语句本身不进行什么操作，它是块 IF 的结束符号，用它标志块 IF 的范围(块 IF 语句开始，END IF 结束)。

一个块 IF 中可完整地包含另一个(或多个)块 IF，称为块 IF 的嵌套。注意嵌套时不能交叉。

## b) ELSE IF 语句

在块 IF 中，then 块和 else 块还可以包含另一个块 IF，即上面介绍的块 IF 的嵌套。如果嵌套的层次比较多，而每一个块 IF 又都要包含块 IF 语句、ELSE 语句、ENDIF 语句，程序就会冗长。而且每一层的嵌套都向右缩进，层次多时就会发生困难，无法再向右缩进了。

F77 提供 ELSE IF 语句来处理“否则，如果……”的情况。其一般形式为：

IF(逻辑表达式 1) THEN

...                      then 块

ELSE IF(逻辑表达式 2) THEN

```

...
ELSE
...
END IF
else if 块
else 块

```

当处理多分支选择时，用数条 ELSE IF 语句往往比较方便，在程序中 ELSE IF 语句不必向右缩进，而连续用多个 ELSE IF 语句表示各个条件的选择。注意块 IF 语句必须有一个 END IF 语句与之对应。而 ELSE IF 语句不需 END IF 语句与之对应，它可以有与之配对的 ELSE 语句。

例：解  $Ax^2 + Bx + C = 0$  。 [\[e 231 02. f90\]](#)

### c) 块 IF 构造

F90 明确提出了块的概念。块是作为单元看待的一个可执行构造的序列，它可用于 IF 构造、CASE 构造和 DO 构造中，对这三种构造都可命名(即标签)。块的规则为：1) 如果一个块中包含一个可执行构造，那么它必须完整地包含在该块中。2) 禁止从块的外部转入块的内部，可以在块的内部进行控

制转移，也可以从块内转移到块的外部。3)块是可以嵌套的。

IF 构造的一般形式为：

```
[构造名:] IF( $e_1$ ) THEN  
    块 1  
[ELSE IF( $e_2$ ) THEN [构造名]  
    块 2]  
...  
[ELSE IF( $e_n$ ) THEN [构造名]  
    块 n]  
[ELSE [构造名]  
    块 n+1]  
END IF [构造名]
```

其中  $e_1, \dots, e_n$  是逻辑表达式，指出各种条件。语句块是一组语句，内容是当  $e$  成立（ $e$  逻辑表

达式值为真)时要执行的算法。IF 构造的控制机制为: (1) 检查  $e_1$  真否。真, 执行块 1, 绕过其它块, 直接转出口语句 END IF 处出口; 假, 跳过块 1, 检查  $e_2$  真否。 (2)  $e_2$  真, 执行块 2, 而后直接转出口; 假, 跳过块 2, 检查  $e_3$ , 以此类推; (3) 如果所有 ELSE IF 语句的  $e$  都是假, 那么必须执行 ELSE 语句下的块  $n+1$ 。如果 IF 构造中没有 ELSE 语句, 就什么也不执行, 转出口。由此可见, 执行 IF 构造, 自上而下顺次检查每块前面的条件, 满足条件的就执行该条件下面的块, 执行完该块后立即转向出口。以后即使还有块满足条件 (条件可以相交), 也不予理睬。因此, 一个 IF 构造中最多只执行一块, 也可能一块也不执行。

例: 输入实数  $x$ , 求下面的三角波脉冲函数  $F(x)$  的值。 [\[e 231 03. f90\]](#)

$$F(x) = \begin{cases} 0.0 & x < 28.0, \quad x \geq 32.0 \\ 0.5x - 14.0 & 28.0 \leq x < 30.0 \\ -0.5x + 16.0 & 30.0 \leq x < 32.0 \end{cases}$$

#### d) IF 构造的缺省形式

在 IF 构造中, then 块、else if 块、else 块都可以缺省。缺省有两种形式, 一种是构造中出现



该类语句，但后面没有算法语句块；一种是连该类语句一起省略。这两种情况使执行时可能结果不同，应按控制机理仔细分析。

✦ 只有 then 块

```
IF(e) THEN
```

```
    块 1
```

```
END IF
```

当 e 真，执行块 1，e 假，则不执行任何语句，此时 IF 构造相当于没有。

✦ 缺省 else if 块

如果连 ELSE IF 语句省去：

```
IF(e) THEN
```

```
    块 1
```

```
ELSE
```

```
    块 2
```

END IF

则  $e$  真执行块 1，假执行块 2。

如果保留 ELSE IF 语句，缺省下面块：

IF( $e_1$ ) THEN

块 1

ELSEIF( $e_2$ ) THEN

ELSE

块 2

END IF

则  $e_1$  真执行块 1， $e_1$  假、但  $e_2$  真时不作任何操作，转出口， $e_1$  和  $e_2$  都假时，执行语句块 2。

✚ 缺省 else 块

IF( $e_1$ ) THEN

块 1

ELSEIF ( $e_2$ ) THEN

块 2

END IF

则  $e_1$  真执行块 1,  $e_1$  假、但  $e_2$  真语句块 2,  $e_1$  和  $e_2$  都假时不作任何操作。

#### ✦ 缺省 then 块

IF THEN 语句必须有, 但后随的块可以缺省:

IF ( $e_1$ ) THEN

ELSEIF ( $e_2$ ) THEN

块 2

ELSE

块  $n+1$

END IF

则  $e_1$  真时, 整个构造等于空设, 什么也不执行。只有当  $e_1$  为假时, 才会一块块检查下去。

### e) IF 构造的嵌套

IF 构造的任一语句块中（then 块、else if 块、else 块）都可以嵌入另一个构造，被嵌入的构造可以是另一 IF 构造，也可以是另一些形态、功能不同的构造，如 CASE 构造、DO 构造，前提是必须把整个构造完整地嵌在 IF 构造的某一块中，不允许一部分嵌在一个块中，另一部分嵌在另一个块中，即被嵌入的任何构造不可跨越两块。

在 IF 构造嵌套时，为了清晰区分内层与外层构造，一般应对内、外层构造分别取名，并且内层构造要缩进几格（可用[Tab]键统一缩进）。

在某些场合 IF 构造嵌套是必需的，但嵌套过多，阅读时要一层层地记住前面各层的条件，容易出错，也不易维护，应尽量减少嵌套。办法是把条件分细，列成多句 ELSE IF 语句。

## 2.3.2 多重选择和 CASE 构造

### a) 整型

F90 增加了 CASE 构造，提供了从几个可选项中选取一个执行的手段，这是因为在某些多种条件的选择情况下使用 IF 构造显得比较繁琐，而使用 CASE 构造可是程序显得直观、简短。CASE 构造控制和块 IF 构造类似，它也是用来编写分叉选择算法，也是根据条件区分作不同的算法，不同的是它只能把某个量分成若干个孤立的离散值，按不同值作不同处理。在处理复杂、多种交叉的条件时不像 IF 构造方便，但更为简明。其一般形式为：

```
[构造名:] SELECT CASE(case 表达式)
    CASE(case 选择符) [构造名]
        块
    [CASE DEFAULT [构造名]
        块]
END SELECT [构造名]
```

其中，case 表达式是整型、字符型或逻辑型表达式，不能是实型和复型表达式。SELECT CASE 是入口语句，END SELECT 是出口语句。case 选择符是 case 值范围表，它有以下四种形式：

(值表)	表示等于该值，各值之间用逗号分开
(下界:)	表示大于或等于该值
(:上界)	表示小于或等于该值
(下界: 上界)	表示在这两个值之间(包括等于)

上面的四种表示方法可以混用，如 CASE (2:5, 9) 等价于 CASE (2, 3, 4, 5, 9)。

CASE DEFAULT 语句是可选的，当 case 表达式的值在 case 值范围以外时，执行 CASE DEFAULT 语句后面的块。CASE 构造的执行顺序是：先计算 case 表达式的值，当它落在值范围内，则执行其后的块。注意，CASE 语句是一条单独的语句。

例：错误语句      CASE (0:59)   PRINT\*, "不及格"

正确写法      CASE (0:59); PRINT\*, "不及格"

例：统计学生考试成绩：100 分为满分，85 分以上为优，70—84 为良，60—69 为及格，59 分以下不及格。 [\[e 232 01.f90\]](#)

CASE 构造的规则是：1) CASE DEFAULT 语句最多只能有一句；2) case 值必须与表达式的类型相

同；3) 给定 CASE 构造中的 case 值范围不能有重叠；5) CASE 块可以是空的，也可以包含其它块，其嵌套形式与块 IF 相同。

程序执行时，CASE 构造的控制机制是：(1)控制进入 CASE 构造后，先计算情况表达式的值；(2)如果第一个 CASE 语句选择符值与情况表达式值相等则执行语句块 1，转出口；(3)如第一个选择符值不为情况表达式的值，再查下一个 CASE 语句的选择符值，满足执行语句块 2，转出口，不满足再查下一个 CASE 语句的选择符值，直至全部情况选择符值都检查完；(4)如果全部情况选择符值都不符情况表达式的值，且又有 CASE DEFAULT 语句，则执行该语句后的 DEFAULT 块，否则直接转出口。

例：块 IF 包含 CASE 块。(有无错？)

```
if_construct: if(a.gt.0) then
    case_construct_b: select case (b)
        case (:0)
            .....
        case (0:)
```

```

        .....
    end select case_construct_b
else
    case_construct_c: select case (c)
        case (:0)
            .....
        case (0:)
            .....
    end select case_construct_c
end if if_construct

```

## b) 字符型

设要编一程序，把学生按专业划分以便检索，专业分为天体物理、应用物理、理论物理三种，把



每个专业应修的课程依次编成语句块 1、块 2、块 3，构造名为 DEPARTMENT\_22\_MAJOR，专业变量名为 MAJOR，则 CASE 构造为：

```
DEPARTMENT_22_MAJOR: SELECT CASE(MAJOR)
                                CASE('Astronomical Physics')
                                    语句块 1
                                CASE('Applied Physics')
                                    语句块 2
                                CASE('Theoretical Physics')
                                    语句块 3
                                END SELECT DEPARTMENT_22_MAJOR
```

则当变量 MAJOR 取值为 ‘Applied Physics’ 时，执行语句块 2（例如它的功能是打印出学生专业课程成绩），如此等等。

例：字符型选择。 [\[e 232 02.f90\]](#)

### c) 逻辑型

当 case 表达式为逻辑表达式时，CASE 语句中的情况选择符也要取逻辑值：真或假。由于它不可能取多于两个的值，因此在逻辑型的情况选择符内不允许写成一个值域范围（即始值:终值形式）。

设有一个关系表达式  $X \geq 3$ ，当成立时打印 ‘YES’，否则打印 ‘NO’。CASE 构造为：

```
SELECT CASE (X >= 3)
      CASE (. TRUE. )
            PRINT *, 'YES'
      CASE (. FALSE. )
            PRINT *, 'NO'
END SELECT
```

### 【作业】

[2.1] 用 IF 语句编写下面功能程序：

(1) 读入  $a, b$ , 若  $a > b$ , 则置  $X=1.23$ , 否则置  $X=32.1$ , 打印  $X$  值;

(2) 读入  $a, b, c, d$ , 若  $a+b > c+d$ , 则  $X=1.1$ , 若  $a+b=c$ , 则  $X=0.0$ , 若  $a+b < c+d$ , 则  $X=-1.1$ , 输出  $X$  值;

(3) 读入  $S$  与  $S1$ , 如  $|S-S1| \leq 10^{-7}$ , 则置  $S$  为  $S1$ 。

[2.2] 用 IF 构造嵌套方法编程: 读入  $X$ , 分三种情况求得  $Z$  值, 当  $X < 0$  时,  $Z=-100$ ; 当  $X=0$  时, 又分三种情况 ( $y < 0$ , 则  $Z=\sin X$ ;  $y=0$ , 则  $Z=0$ ;  $y > 0$ , 则  $Z=\cos X$ ); 当  $X > 0$  时,  $Z=100$ , 打印  $Z$  值。

[2.3] 读入一个整型变量  $N$  的值, 用 CASE 构造编程, 若  $N=1、2、3、5$ , 则  $Y=X$ ;  $N=4、8$ , 则  $Y=X^2$ ;  $N=6、7$ , 则  $Y=X^3$ , 其它情况则  $Y=0$ , 打印  $Y$  值。

[2.4] 设变量 `color_light` 是字符型变量, 表示灯光色彩, 编一 CASE 构造, 当该变量值是红色 (RED) 则打印 STOP 字样。当变量值是黄色 (YELLOW), 打印 WAIT 字样, 当值是绿色 (GREEN) 打印 CROSS 字样。

## 第三章：循环结构

### 单纯循环

#### 3.1.1 GOTO 语句实现循环

循环结构用于实现重复的算法。它是三种基本结构（顺序、选择、循环）之一，具有重复执行某一段语句的功能，因为它以 **DO** 作为关键字，所以又称 **DO** 构造。在程序中存在两类循环：无条件的循环和有条件的循环。无条件循环是无休止地执行一个程序段，而有条件的循环是在满足一定条件时才执行循环。

F77 允许使用 **GOTO** 语句来实现转移。**GOTO** 语句的一般形式为：**GOTO (语句标号)**。由于 **GOTO** 语句破坏了语句顺序执行的正常状况，不符合结构化原则，因此一般不提倡使用 **GOTO** 语句。只有在一个基本结构内部可以使用 **GOTO** 语句。利用 **GOTO** 语句可以实现循环处理。如果在逻辑 **IF** 语

句中使用 **GOTO** 语句就可以实现有条件的循环，循环操作能够在一定条件下结束。

### 3.1.2 有循环变量的 DO 构造

#### 1) DO 语句和循环次数

DO 构造可分为不带循环变量与带循环变量两种形式。

当需要执行的循环次数为已知时，用 DO 语句实现循环比较方便。它的一般形式为：

**DO [[标号][, ]] 循环变量=初值式, 终值式[, 增量式]**

例：循环读入学生的学号和成绩 30 次并打印。

```
DO 10, N=1, 30, 1
```

```
    READ *, NUM, GRADE
```

```
10  PRINT *, NUM, GRADE
```

上面是一个循环，第一行 DO 语句称为循环语句，DO 后面的数 10 是一个标号，表示循环的范围到标号为 10 的语句为止，也就是反复执行 READ 语句和 PRINT 语句。DO 语句中的 N 是“循环变量”，用它来控制循环次数，“N=1, 30, 1”的意思是：N 的初值为 1，终值为 30，每执行一次循环 N 的值增加 1。当 N 再变化到 31 时，由于它已超过了指定的终值 30，不再执行循环。

下面的 DO 语句是合法的：

```
DO 10, I=1, 10, 2
```

```
DO 20 N=1, 5
```

```
DO 100, X=1.2, 2.4, 0.2
```

```
DO T=2.5*2, 50./3., 0.3
```

```
DO M=1.5, 12.5, 15
```

DO 语句的一些特点是：

✦ 在上述 DO 语句的一般形式中，当循环变量的增量(步长值)为 1 时增量式可不写。

✦ 循环变量初值、终值和步长可以分别是常数、变量或表达式。如果是变量则它应预先被赋值。如果是表达式，则先计算出表达式的值。循环次数可以从循环初值、终值和步长计算出来：次数 =  $\text{INT}((\text{终值} - \text{初值} + \text{增量}) / \text{增量})$ 。如果计算出的循环次数  $< 0$  时，则按 0 处理，即一次也不执行循环。

例：对于 `DO I=1,10,2` 其循环次数 =  $\text{INT}((10 - 1 + 2) / 2) = 5$  次。I 按序分别取值为：1,3,5,7,9。对于 `DO I=10,1,2` 则循环次数 = 0 次。I 不可取值，程序运行到这里时将跳过此循环。

✦ 循环变量的初值、终值和步长可以为正或负。初值、终值可以为零。但步长不应为 0，否则循环变量的值永远不会超过终值，从而陷入死循环。

例：对于 `DO I=-1,-3,-1` 其循环次数 =  $\text{INT}((-3 + 1 - 1) / (-1)) = 3$  次。I 按序分别取值为：-1,-2,-3。

✦ 如果循环变量的类型和初值、终值和步长的类型不一致，则按赋值的规则处理，即需先将初值、终值和步长的类型转化成循环变量的类型，然后进行处理。为避免错误，应尽量使循环变量类型与初值、终值和步长的类型一致。

例：对于 `DO I=1.5,3.6,1.2` 不要根据  $\text{INT}((3.6 - 1.5 + 1.2) / 1.2) = 2$  而认为循环次数为 2，而应当先将实

型量转化为整型量，即变成相当的循环语句 `DO I=1,3,1` 其循环次数为 3 次而不是 2 次。

例：对于 `DO X=1.5,3.6,1.2` 由于循环变量不是整型的而是实型的，它的循环次数为 2 次。`X` 取值分别是 1.5,2.7。

✦ 由于实型数在运算和存储时有一些误差，因而循环次数的理论值与实际值之间会有一些差别。这种情况在程序设计中常有发生，而且比较隐蔽不易发现。所以应该避免使用实型的循环变量，用整型循环变量计算出的循环次数是绝对准确的。

例：对于 `DO X=0.0,50.0,0.1` 理论循环次数 =  $\text{INT}(50.1/0.1) = 501$ ，但实际上在许多计算机上它只执行 500 次循环。原因是实数在内存中的误差使得增量值不是准确的 0.1，由于循环的误差积累，到执行完 500 次循环后 `X` 的值可能已超过 50.0，因而停止执行循环。改用整型循环变量时，则循环改写为：`DO I=0,500; X=I/10.`

#### m) DO 循环执行步骤

循环执行过程按序为以下几个步骤：



- (1). 计算初值式、终值式、增量式的值，并将它们转换成循环变量的类型。
- (2). 将初值赋予循环变量。
- (3). 计算应循环的次数。
- (4). 检查循环次数，若 $\leq 0$  则跳过循环体，执行循环终端语句下面的一个执行语句。如果 $> 0$ ，则执行循环体。
- (5). 执行终端语句时，循环变量增值。
- (6). 循环次数减 1。
- (7). 返回 (4)，重复执行 (4)、(5)、(6)、(7)。

#### n) 循环终端语句

上面介绍的循环中，循环终端语句为一般的执行语句。F90 规定：循环终端语句可以是除了 GOTO、块 IF、CASE、CYCLE、DO、ELSE、ELSE IF、END IF、END、END SELECT、EXIT、SELECT CASE、STOP 和 RETURN 语句以外的任一可执行语句，如打印语句、赋值语句、输入语句、逻辑 IF 语句等都可

以作为终端语句。特殊的循环终端语句是：**END DO**(常用于无语句标号时)和 **CONTINUE**(常用于有语句标号时)。**END DO** 语句使老的 **CONTINUE** 语句显得没有什么用处了，虽然 F90 的向下兼容性使 **CONTINUE** 语句仍然可用，但新编写的程序应该尽量使用以 **END DO** 结束的块 **DO** 构造。

例：用展开式  $e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^n}{n!}$  求指数函数的数值。 [\[e 312 01.f90\]](#)

#### o) 停止语句

**CONTINUE** 语句本身不进行任何机器操作，只是将流程转到逻辑上的下一个语句，因此，**CONTINUE** 语句又称为“空语句”，即进行“空操作”。与 **CONTINUE** 语句(继续功能)相对应的停止功能语句是 **STOP** 和 **PAUSE** 语句。

**PAUSE** 语句(在 F90 中不推荐使用，在 F95 中被废除)暂时中止程序的运行，将系统挂起，使程序操作员可以执行其它操作系统命令。它的一般形式是：**PAUSE [暂停值]**，暂停值为字符串常量或 5 位数以下的整型数，当程序运行至断点处将输出暂停值。如无暂停值的话，系统将输出默认的信息，WinNT/9x 系统上输出“回车才能继续”的信息。

例：PAUSE 701

PAUSE 'ERROR DETECTED'

在这些例子中，它的用处是在程序中加入断点把程序分段，以便于一段一段地调试程序。Visual Fortran 的 Debug 功能可代替这种程序调试方法，但它也可以用于一些输出情形。[\[e 312 02.f90\]](#)

STOP 语句是停止运行，一个程序单位中可以有多多个 STOP 语句，执行到任一个 STOP 语句处时，程序即完全中止运行。在子程序中如果有 STOP 语句，也是使整个程序停止运行而不是使控制返回主程序。STOP 语句的一般形式为: **STOP [停止值]**，与 PAUSE 语句类似，程序停止运行时将输出停止值。在 F66 中，END 不作为执行语句而只作为程序单位的结束标志，需要在 END 之前用 STOP 语句使程序结束运行。有些人在写 F77 程序时仍保留此习惯，在 END 语句之前又写了一个 STOP 语句。

#### p) DO 循环嵌套

在一个 DO 循环中又完整地包含另一个 DO 循环，称为 DO 循环的嵌套。嵌套层数可以不限，各层

的循环变量不允许同名。注意内循环应当完整地嵌套在外循环之内，即内循环是外循环体中的一部分，内外循环不能交叉。即：

```
do i=1,10  
    do j=1,20  
        .....  
    end do  
end do
```

程序的执行过程是外循环执行一次，内循环执行一遍。 [\[e 312 03.f90\]](#)

例：有 10 个实数，将它们按大小排列。 [\[e 312 04.f90\]](#)

#### q) DO 循环规则

- ✦ 循环变量可以在循环体中被引用，但不应当再被赋值。

例：下面写法是不正确的，循环变量 **N** 不能在循环体内被重新赋值。

```
do n=1,10
```

```
.....
```

```
    n=2*n
```

```
end do
```

- ✦ 循环的次数是根据循环变量的初值、终值和步长值计算出来的，在执行循环体期间是确定不变的。
- ✦ 可以用转移语句从循环体内转到循环体外，也可以在循环体内转移，但不允许从循环外转到循环内。(块规则)

例：下面写法是合法的(尽管不符合结构化原则)。

```
do n=1,10
```

```
.....
```

```
    if(n*x.gt.1.) go to 10
end do
```

```
10  .....
```

例：下面写法是非法的。

```
if(x.gt.1.) go to 10
```

```
do n=1,10
```

```
    .....
```

```
10  .....
```

```
end do
```

- ✦ 多个 DO 循环可以共享一条循环终端语句，但循环体必须完全包含在外围 DO 循环体内。
- ✦ 如果 DO 循环出现在 if, else if 或 else 块内，则 DO 循环范围必须完全包含在该块中。
- ✦ 如果 IF 语句和 SELECT CASE 语句出现在 DO 循环范围内，则相应的 END IF 语句和 END SELECT

语句也必须出现在这个 DO 循环体内。

### r) 隐 DO 循环

隐 DO 循环实际上是带控制循环变量的 DO 循环，但简化成只有 DO 循环的第一句，且把关键字 DO 隐去，成为  $I=m_1, m_2, m_3$  形式。它不是独立语句，只是用作为读写语句的输入输出表中一个组成部分，用来控制重复读写的次数。例如：

```
READ *,(VALUE(I),I=1,20)
```

表示读入 VALUE(1),VALUE(2), $\cdots$ ,VALUE(20)的值。

```
WRITE(*,*)(A,B,N=1,5)
```

表示在当前设备用默认格式重复输出 A、B 的值 5 次。

隐 DO 循环只能作为输入输出表的一部分出现，不能用于其它场合。此时输入输出表的一般形式为：  
(I/O 列表,循环变量名=初值,终值,增值)。也即把输入输出表与循环控制部分一起用括号括起，中间

用逗号分开，称为隐 DO 表，写在读写语句后面作为读写对象。

隐 DO 表可以嵌套，如：

```
PRINT *, ((A(I,J),I=1,3),J=1,3)
```

先把内层(A(I,J),I=1,3)隐 DO 表作为输出表，再与隐 DO 控制 J=1,3 合成外层隐 DO 表。其控制机制与嵌套 DO 循环一致，先内层循环完，外层循环变量加一步长，再循环完内层。其打印值的次序如下：

A(1,1), A(2,1), A(3,1), A(1,2), A(2,2), ..., A(3,3)

即先输出第 1 列，再输出第 2 列、第 3 列。如果把 J 作为内层，I 作为外层循环变量，则输出是按行的：

```
PRINT *, ((A(I,J),J=1,3),I=1,3)
```

则打印输出值的次序为：

A(1,1), A(1,2), A(1,3), A(2,1), A(2,2), ..., A(3,3)



## 条件循环

### 3.2.1 无循环变量的 DO 构造

#### s) 一般形式

这种 DO 构造形式非常简单，一般形式为：

```
[构造名:] DO  
    块  
END DO
```

不带循环控制变量的 DO 构造控制机制为：进入 DO 构造体后，从 DO 语句下面第一句执行起顺次执行到 END DO 前的最后一句，再返上来从 DO 语句下面第一句执行起，重复执行整个 DO 块。如此反复执行 DO 块，其间如遇到 EXIT 语句，就停止执行 DO 块，退出循环，转向执行 END DO 下面

的语句。对于不带控制变量的 DO 构造，DO 块中必须有 EXIT 语句，使它停止循环，否则循环将无休无止地一直进行下去，形成死循环。

DO 构造也可在自身中再嵌 DO 构造，例如：

```
FIRST: DO
```

```
    块 1 的第一部分
```

```
SECOND:  DO
```

```
    块 2
```

```
END DO SECOND
```

```
    块 1 的第二部分
```

```
END DO FIRST
```

t) EXIT 语句

许多实际问题中无法预先给定循环次数，而是只给出一个条件，满足此条件时就继续执行循环体，否则退出循环，故有条件循环时循环次数不是固定的。除了可以用 GOTO 语句跳出循环外(非结构化方法)，F90 还给出了其它用于条件循环方式的语句。

EXIT 语句的作用是停止循环，使控制退出循环结构，因此又称出口语句。它的一般形式是：EXIT [DO 构造名]。执行 EXIT 语句，其功能是导致循环终止。如果语句引用了 DO 构造名，则它属于该构造，否则属于它所出现的最内层 DO 构造。对于多层嵌套的 DO 构造，尤为需要指明是从哪一个构造名的 DO 构造中退出。如当 EXIT 指明从第三层结构退出，则第三层与第三层内各层嵌套的 DO 构造都将被停止执行。

单独使用将导致无条件地终止循环，没有实用意义。为了控制 DO 结构在满足某种条件时停止循环，通常将 EXIT 语句与 IF 语句结合使用，即在 DO 构造内使用：IF(逻辑表达式) EXIT。当逻辑表达式为真时，EXIT 语句被执行，循环终止，否则循环将继续进行。

例：do; if((input.eq.'n').or.(input.eq.'y')) exit

```
write(*,'(a)') 'Enter y or n:'; read(*,'(a)') input
```

```
end do
```

例：一个猜数游戏。用随机数产生器产生 1—10 之间的一个整数，你猜它，如果猜得太大或太小都会提示，猜对了结束。[\[e 321 01.f90\]](#)

例：求级数  $y = \frac{1}{1*2} + \frac{1}{2*3} + \frac{1}{3*4} + \dots + \frac{1}{n(n+1)}$  的前 n 项和，但当某项绝对值  $\leq 10^{-5}$  时，虽未满足 n 项，也因满足精度而不再加入下一项。[\[e 321 02.f90\]](#)

#### u) CYCLE 语句

CYCLE 语句的一般形式为：**CYCLE [DO 构造名]**。CYCLE 语句的功能是在循环中跳过它下面那部分的 DO 块，重新返回到块的第一个语句开始执行。运用 CYCLE 语句，可以在执行循环中某一次迭代时不执行下面的部分语句，使 DO 结构更为灵活。

CYCLE 语句与 EXIT 语句一样属于特定的 DO 构造。如果语句引用了 DO 构造名，则它属于该构造，

否则属于它所出现的最内层 DO 构造。 [\[e 321 03.f90\]](#)

### 3.2.2 DO WHILE 语句

在 F90 增强的功能中，增加了 DO WHILE 语句以支持当型循环。因此，DO 构造不仅仅能完成循环次数已知的循环，而且能够完成此数未知的循环。它的一般形式为：[构造名] DO WHILE (逻辑表达式)。DO WHILE 语句的规则和块 IF 语句所需满足的的块规则类似。

例：上面的例子中用 EXIT 语句退出循环等价于

```
do while ((input.ne.'n').and.(input.ne.'y'))
```

```
write(*,'(a)') 'Enter y or n:'; read(*,'(a)') input
```

```
end do
```

例：对一个大于或等于 3 的正整数，判断它是不是一个素数。 [\[e 322 01.f90\]](#)[\[e 322 02.f90\]](#)

例：求两个整数的最大公约数。 [\[e 322 03.f90\]](#)[\[e 322 04.f90\]](#)

例：求正弦函数的近似值。 [\[e 322 05.f90\]](#)

### 3.2.3 DO 构造的一般形式

由以上的各种 DO 循环可归纳为其下的 DO 构造一般形式：

[构造名:] DO [标号][循环控制]

块

终止语句

1) 循环控制的形式是：

[,] 循环变量=下界, 上界[, 增量] 或

[,] WHILE(逻辑表达式) 或

无 (用块中的 EXIT 和 CYCLE 语句进行实际的控制)

2) 终止语句的形式是：

END DO[构造名] 或

[标号] CONTINUE 或

[标号] 允许的语句

### 【作业】

[3.1] 用不带循环控制变量的 DO 构造求下列式（当  $m \geq 1000$  时停止）：

(1)  $1^2 + 2^2 + 3^2 + \dots + m^2$

(2)  $1^2 + 3^2 + 5^2 + \dots + (2m-1)^2$

(3)  $\prod_{n=1}^m (1 - (2n+1)^{-2})$

[3.2] 设  $f = 1 + x^{-1} + x^{-2} + x^{-3} + \dots$ ，编程求  $f(x)$  的前若干项之和，当相邻两项差的绝对值小于  $10^{-6}$  即可。

[3.3] 用牛顿迭代法求方程  $x^3 + 9.2x^2 + 16.7x + 4 = 0$  在  $x=0$  附近的根，精度满足  $|x_{n+1} - x_n| \leq 10^{-5}$  时停止迭代，并规定最多迭代 50 次。牛顿迭代公式为  $x_{n+1} = x_n - f(x_n)/f'(x_n)$ 。

## 数据类型和属性

本节主要介绍 F90 的类型说明中新的种别和属性等概念，并且引进派生数据类型等数据结构用基本语句。

### 4.1.1 类型说明语句

#### h) 一般形式

第一章中我们简单地按照 F77 的传统方法介绍了数据类型和说明语句，这里将介绍具有现代特性的类型说明语句的书写形式。F90 程序中的数据都有三个特征：**类型**、**种别**、**属性**，由类型说明语句来定义说明。其一般形式是：

类型说明[(种别说明)][, 属性说明表] :: 变量名表[=初值]

例：REAL(KIND=2), DIMENSION(1:10) :: X, Y



说明变量 X, Y 都是实型, 种别是 2, 属性是一维数组。这样 X、Y 实际上是两个种别参数为 2 的一维实型数组, 各具有 10 个元素。在某些场合下, 种别说明与属性说明可以省略, 此时它的一般形式是最基本形式:

类型说明:: 变量名表

下面逐一介绍变量在说明语句中说明类型、种别、属性的方法。

### i) 类型说明

内部类型一共有五种, 三种数值型: 整数型、实数型、复数型; 两种非数值型: 逻辑型、字符型。三种数值型变量的定义关键字分别是: INTEGER, REAL, COMPLEX。例:

```
INTEGER:: X, Y
```

```
REAL:: A, B
```

```
COMPLEX:: C, D
```

F77 中的隐式说明不利于程序的可读性与可维护性, 在 F90 中不提倡使用。在 F90 中, 每个变量

名都应该在说明部分中说明其类型，不应该使用隐式说明、为了抑制隐式说明发生作用，应该在程序说明部分开始就写出语句：

```
IMPLICIT NONE
```

既声明不使用隐式说明。

### j) 赋初值

初值是指不需要在程序执行中赋值，当程序开始时变量已有初始的量，立即可以参加运算。F90中置初值不需要专门语句，只需在类型说明语句的变量表中，把要置的初值写在指定的变量名后即可。它的形式为：

类型说明:: 变量名 1=初值 1[, 变量名 2=初值 2, ...]

例如，要让实型变量 X 有初值 1.1，Y 有初值 2.2，Z 不置初值，W 置初值 3.3，写类型说明语句：

```
REAL:: X=1.1, Y=2.2, Z, W=3.3
```

这样，既说明了类型，也为 X、Y、W 置了初值。变量赋初值后，其值在执行过程中仍可改变。需

要注意到，在过程中这种赋初值的方法实际上是对变量赋予了 **SAVE 属性**，即当过程被调用以后，变量的新值将被保存下来，过程再次被调用时变量的初值不再是类型说明语句中的初值，而是上次被保留下来的值。例如，如果希望上面的变量 X 在过程每次被调用时都有相同的初值，则应该写成：

```
REAL, SAVE :: X, Y=2.2, Z, W=4.4
```

```
X=1.1
```

#### k) DATA 语句

在 F77 中，有一个专门给变量赋初值的说明语句，即 **DATA** 语句。它的一般形式为：

**DATA 变量名表 1/初值表 1/[[,]变量名表 2/初值表 2/...]**

变量名表可以是隐 DO 循环，初值表中的常数之间须用逗号分开，重复的常数表值可以采用如下表示方法：**重复次数\*常数值**。在 DATA 语句中赋值的变量都有 SAVE 属性，除非变量名又出现在 COMMON 语句中，可以通过 SAVE 语句或类型说明中的 SAVE 属性对其显式说明。例如，可以对数组作以下的初始化：

```
DIMENSION A(10,10)
```

```
DATA A/100*1.0/ ! 按数组变量名统一初始化
```

```
DATA A(1,1), A(10,1), A(3,3) /2*2.5, 2.0/ ! 按数组元素逐个初始化
```

```
DATA ((A(I,J), I=1,5,2), J=1,5) /15*1.0/ ! 按隐 DO 循环初始化
```

对字符串作初始化:

```
CHARACTER (LEN=10) name
```

```
CHARACTER BELL, TAB, LF, FF, STARS*6
```

```
CHARACTER*8 help
```

```
DATA name, STARS /'Zhang Fei', '****' /
```

```
DATA BELL, TAB, LF, FF /7, 9, 10, 12/ ! 用 ACSII 控制字符码赋于字符变量
```

```
DATA help(1:4), help(5:8) /2*'HELP' /! 用字符子串分段赋值
```

对数值型数据作初始化:

```
INTEGER n, order, list(100)
```

```
REAL coef(4), eps(2), pi(5), x(5, 5)
COMPLEX*8 cstuff
DATA  n/0/, order/3/, list/100*0/
DATA  coef/1.0, 2*3.0, 1.0/, eps(1)/.00001/
DATA  pi/5*3.14159/
DATA  ((x(j, i), i=1, j), j=1, 5)/15*1.0/
DATA  cstuff/(-1.0, -1.0)/
```

#### 4.1.2 种别说明

##### a) 种别说明方法

种别是 F90 的新概念。一个数据, 不仅有一个类型, 并在同一类型下可分为若干种别, 种别值确定了数据的大小范围和精度。有了种别说明后, 程序更易于移植。因为在不同的计算机系统上, 同一种变量类型可以有不同的精度, 因此当程序在另一种机子上运行时可能出现溢出或下溢。规定种别后

可以避免这种情况的出现。

我们知道，一个数据通常在内序中占一个存贮单元。对整型数而言，如果该变量在程序中使用值范围很小，则只需半个存贮单元。如果变量的整数变化范围很大，则存贮时有必要占两个内存单元。实型数更复杂，除了存贮的数值范围大小不同外，要求精度也会不同，有的只要 8 位有效值即满足，有的则可能要 24 位有效值。这样，它们要求的存贮单元数量不同。为了提高效率，节约内存，按照该变量表达的值范围与表达的精度范围，把同一类划分成几个种别，不同种别分配不同数目的内存单元。

国际标准 FORTRAN90 版本没有规定每个类型必须有哪些种别，具体的种别划分由 FORTRAN 软件开发商自行设置。因此程序员在设计变量时，按该变量表达值的范围、精度要求范围，查阅手册，确定合适的种别，在说明语句中加以说明。

种别由种别选择符说明，写在类型关键字后括号内，其关键字是 KIND，后跟 '=' 号及种别值。其形式为：类型说明([KIND=]种别值)。

例如要说明变量 X 是实型，种别值是 2，说明语句是：

REAL (KIND=2) :: X

一个变量必有一个种别，如果变量的类型说明语句中没有种别说明符，如：

REAL :: X, Y

则表示变量 X, Y 的种别缺省，这时采用系统规定的标准值。

## b) 种别值

Compaq Fortran 提供了如下几类种别值：

整数：有 4 种，种别值即为字节数 n。

种别值 n	取值范围 $(-2^{8n-1}-2^{8n-1}-1)$	
INTEGER([KIND=]1) 或 INTEGER*1	-128—127	
INTEGER([KIND=]2) 或	-32768—32767	

INTEGER*2		
INTEGER([KIND=]4) 或 INTEGER*4	-2147483648—2147483647	缺省值
INTEGER([KIND=]8) 或 INTEGER*8	-9223372036854775808—9223372036854775807	仅用于 Alpha 芯片机型

实型数：有 3 种。F90 标准没有规定指数的允许范围和有效位数，。

REAL([KIND=]4) or REAL*4	通常实数的范围是 $10^{-38}$ — $10^{38}$ 之间的 7 位有效数字	缺省值
REAL([KIND=]8) or REAL*8		等价于双精度型 DOUBLE PRECISION
REAL([KIND=]16) or REAL*16		仅用于 OpenVMS、Tru64 UNIX、Linux 操作系统



复型数：有 3 种。每种表示整型数据或实型数据的方法都可以用来表示复型数据的实部和虚部。  
注意简写与完整写法之间的差别。

COMPLEX([KIND=]4) or COMPLEX*8	缺省值
COMPLEX([KIND=]8) or COMPLEX*16	等价于双精度复型 DOUBLE COMPLEX
COMPLEX([KIND=]16) or COMPLEX*32	仅用于 OpenVMS、Tru64 UNIX、Linux 操作系统

逻辑型：有 4 种

LOGICAL([KIND=]1) or LOGICAL*1	
-----------------------------------	--

LOGICAL([KIND=]2) or LOGICAL*2	
LOGICAL([KIND=]4) or LOGICAL*4	缺省值
LOGICAL([KIND=]8) or LOGICAL*8	仅用于 Alpha 芯片机型

字符型：有 1 种

CHARACTER([KIND=]1)

字节型 BYTE：取值为 1 个字节，等价于 INTEGER([KIND=]1)。

### c) 种别函数

F90 中关于种别选择的内部函数有：

**KIND(X)**：函数 KIND 用于查询变量的种别，它返回 X 的种别值，当 X 取值为 0 时，返回标准种别值即缺省值。如：KIND(0) 返回值是整型的标准种别值，KIND(0.)、KIND(.FALSE.)、KIND("A") 分别返回实型、逻辑型、字符型的标准种别值。

**SELECTED\_REAL\_KIND([n][,m])**：该函数返回实型变量对所取的值范围和精度恰当的种别值。其中 n 是指明十进制有效位的位数，m 指明值范围内以 10 为底的幂次。例如：SELECTED\_REAL\_KIND(6,70) 的返回值为 8，表示一个能表达 6 位精度、值范围在  $-10^{70}$ — $+10^{70}$  之间实型数的种别值为 8。但该机型上不能提供满足要求的种别值时，它的返回值是：-1(当精度位数达不到时)，-2(当数值范围达不到时)，-3(两者都达不到时)。对给定的实型和复型量 X，它的精度和范围可通过内部函数 **PRECISION(X)** 和 **RANGE(X)** 查出。

**SELECTED\_INT\_KIND([m])**：该函数返回整型变量对所取的值范围恰当的种别值。m 指明值的范围是  $-10^m$ — $+10^m$ 。

可以用 SELECTED\_REAL\_KIND 或 SELECTED\_INT\_KIND 定义一个 PARAMETER 常数以备后用，例如下面的语句定义了有 9 位数的整型数。

INTEGER, PARAMETER :: MY\_INT\_KIND = SELECTED\_INT\_KIND(9)

INTEGER(MY\_INT\_KIND) :: HILL

#### d) 常数种别

程序中的常数如要标明种别，方法有二。若是数值型常数或逻辑型常数则用后缀法，即后加一下划线，再跟种别值。当实数型数据的指数字母是 D 时，禁止说明种别值。如果复型数据的实部和虚部都是整数，则它的精度和范围与缺省的实型相同。如果两部分都是实型，则它的精度和范围按如下的方法确定：两部分有相同的种别，为该种别，两部分中有不同的种别，则由较大的种别值确定。例如：

21\_2+7.6\_4      表示整型种别为 2 的数 21 与实型种别为 4 的数 7.6 相加。

3.8E-5\_4

0.87D-16      禁止说明种别值

(4.7\_8, 5)      复型数据表示用括号，逗号分开前面的实部(种别值为 8 的实数)和后面的虚部(缺省种别值的整数)。

.FALSE.\_4      表示逻辑型，其常数值是假，种别值是 4。

例：用语句：

```
INTEGER, PARAMETER :: LONG=SELECTED_REAL_KIND(9, 99)
```

```
REAL :: A=2.8_LONG, B=1.23456789E60_LONG
```

来保证常数有需要的 9 位有效数字和 $-10^{-99}$ — $10^{99}$ 的指数范围。PRECISION(A) 和 RANGE(A) 的返回值应是 9 和 99。

若是字符型常数，则用前缀法，把种别值列在字符常数之前，其间用下划线连接。例如：

5\_ ‘α β γ ’      表示希腊字符串α β γ 的种别值是 5。

6\_ ‘计算数学’ 表示汉字字符串 ‘计算数学’ 的种别值是 6。

这里假定该机系统支持希腊字母与汉字，并已规定它们的种别参数是 5 和 6。对 Compaq Fortran，只有一种字符型，因此上面的字符常数应写成：1\_‘计算数学’，1\_‘α β γ ’，或：‘计算数学’，‘α β γ ’。字符串的字符不只限于 Fortran 字符集内，处理系统支持的图形符号也是允许的。

#### e) 整数的其他进制

在 F90 中，还可以根据需要定义二进制、八进制和十六进制正整数常量。二进制常量的表示方法是以字母 B 开头，后跟定界符括起来的数字串，**定界符**可以是撇号或括号，数字是 0 或 1，如：B'01011'，B(01011)。八进制常量的表示方法是以字母 O 开头，后跟定界符括起来的数字串，数字范围是 0 至 7，如：O'10472'。十六进制常量的表示方法是以字母 Z 开头，后跟定界符括起来的数字或字母串，数字范围是 0 至 9，字母范围为 A 至 F，如：Z'18E2D'。但要注意，整数的这些形式，仅限于出现在 DATA 赋值语句中。

在 Compaq Fortran 中，还可以这样定义非十进制数：**数字#数字串**，最前面的数字表示进制，省略式表示十六进制。如：

2#01011011	->	二进制
8#43051472	->	八进制
#A92D80EF	->	十六进制

#### 4.1.3 属性说明

说明语句除说明对象的类型、种别外，还可说明对象的属性。一个对象被说明具有某一属性时，就使该对象具有某种附加功能、特殊的使用方式与适用范围。一个被说明对象可以没有附加的属性说明，此时它只是最基本的变量，与 F77 中的变量相同。它也可以有多个属性说明，分别对被说明对象附加各件功能，规定它在各种场合下的使用方式。每种属性说明都有专门的关键字，各属性关键字间用逗号分开，全部属性关键字写在说明语句种别说明符之后，双分隔号::之前，各属性关键字之间次序任意。属性不仅用于说明数据，还用于说明过程。

#### a) 属性

属性说明关键字有很多，将在有关章节中详细说明。一般来说，数据属性描述了一个对象是如何在程序中被应用的，可以使用一个或多个语句来规定某个数据对象的属性。Visual Fortran 的数据属性如下表所示。

属性关键字	描述	适用范围
ALLOCATABLE	说明动态数组	数组

AUTOMATIC	声明变量在堆栈中而不是在内存中	变量
DIMENSION	说明数组	数组变量
EXTERNAL	声明外部函数的名称	过程
INTENT	说明过程哑元的用意	过程哑元
INTRINSIC	声明一个内部函数	过程
OPTIONAL	允许过程被调用时省略哑元	过程哑元
PARAMETER	声明常量	常量
POINTER	声明数据对象为指针	变量
PRIVATE	限制模块中的实体访问于本块内	常量、变量或模块
PUBLIC	允许模块中的实体被外部使用	常量、变量或模块



SAVE	保存过程执行后其中的变量值	变量或公共块
STATIC	说明变量为静态存储	变量
TARGET	声明变量为目标	变量
VOLATILE	声明对象为完全不可预测并在编译时无优化	数据对象或公共块

## b) PARAMETER 属性

PARAMETER 属性也称常数名属性。被说明对象一旦附加了 PARAMETER 属性，就不再是变量名，而是一个常数的名字，它的形式虽与变量名形式一样，但在程序中不能改变值，只能当常数使用。例如：

```
INTEGER, PARAMETER :: K=20
```

```
REAL(8), PARAMETER :: PI=3.141592654, K_PAI=K*PI, Light_Speed=2.99654E10
```

常数名被说明后，在程序中就不可再更改值，如要更改，则系统提示出错信息。这一性质，有助于防止程序中误改不可改的常数。使用常数名可以加强可读性，也有助于程序维护。设原编程序是解

四阶方程以后要扩充为能解五阶方程，修改时必须把所有反映阶数，写在程序各处的 4 改成 5，这就难免修改时有遗漏，甚至把不该改的 4 也误改为 5，造成程序混乱。使用常数名就可避免这种维护上的困难，当编四阶方程解的程序时，先定义 **Order** 是整型数 4 的常数名；

```
INTEGER, PARAMETER :: Order=4
```

在程序中所有规定阶数 4 的场合都不写 4 而用常数名 **Order** 代替，以后只要把 **Order** 由 4 改为 5 即可。

### c) DIMENSION 属性

说明一个符号名是数组名，只要在说明语句中附加数组属性关键字。数组属性关键字是：  
**DIMENSION(数组形状说明)**。

括号内数组形状说明规定数组有几维，以及每维下标的变化界限，这将在下一章中详述。例如：

```
INTEGER(KIND=2), DIMENSION(-2:8) :: X
```

说明了 **X** 是种别为 2 的整型一维数组名，数组下标以 -2 为下界，8 为上界，共有 11 个元素。

## 非数值型数据

FORTRAN 五种数据类型中，有两种非数值型数据：逻辑型、字符型。要运用这两种数据，同样要掌握它们的变量说明、常数书写格式、表达式、赋值语句、编辑符、输入输出格式。掌握这两种类型，能实现信息管理方面的强大功能，在处理办公室报表、文件、文本、文本编辑时有足够手段。并使数值型计算也变得更加丰富。

### 4.2.1 逻辑型数据

逻辑型数据与数值型数据有本质的不同。它的值不是具体的数，而是对某个论点作出的判断，常用的逻辑数据是对两个数值间等或不等关系的判断、两个字符串之间的等或不等关系的判断，也即是一个关系表达式。如果该关系成立，则称其逻辑型是‘真’，不成立则称其逻辑型是‘假’。因此逻辑型值只有两个：真或假，不像整型、实型的值可以取无穷多个。复杂的逻辑表达式的基础仍是上述关系不等式，只是这些关系表达式是以某种方式连接起来的较为复杂的判断，如逻辑表达式  $X > 3$ . AND.  $X < 5$ ，即‘ $X > 3$  而且  $X < 5$ ’，则当点  $X$  落在此区间时判断成立，逻辑表达式值为真，否则为

假。

有关逻辑型数据的运算和逻辑表达式已在第二章中作了介绍，这里仅就涉及其说明和赋值语句。

### 1) 逻辑型变量

对于逻辑值在程序中可变的量，则称逻辑变量，逻辑变量取名的法则同数值型变量，它也要作类型说明，而后才能在程序中按逻辑数据使用与读写。逻辑型类型说明关键字是 LOGICAL，其类型说明语句一般形式为：

LOGICAL[(KIND=种别值)][, 属性说明表] :: 变量名表[=初始值]

例如：LOGICAL(KIND=2), DIMENSION(1:10) :: X，说明 X 数组内有元素 X(1), X(2), ..., X(10) 都是种别值为 2 的逻辑型数据。

变量名表中要说明为逻辑型的对象可以是一般变量名、数组名、函数名、构造名等等，彼此用逗号分开，还可以对变量赋以初值。例如要说明 L1, L2 是逻辑型变量，并且置 L1 初值为真，可写成：

LOGICAL :: L1=.TRUE., L2

### m) 逻辑赋值

逻辑变量取值的方法和数值型数据类似：赋初值、读入、使用逻辑赋值语句赋值。逻辑赋值语句的一般形式是：**逻辑变量=逻辑表达式**。它通常用来将一个较长的表达式的值赋给一个逻辑型变量，而后作为条件写在 IF 语句或选择构造中，使条件变得简短明确。

例：输入实数  $x$ ，求下面的函数值。

$$F(x) = \begin{cases} x^2 & (1 < x < 2) \\ 1-x & (x \leq 1, x \geq 2) \end{cases}$$

Program Sample

```
implicit none
real :: x, y
logical :: in_side
read *, x ; print *, 'x=' , x
in_side=x>1..and.x<2.
```

```
      if(in_side) then
          y=x*x
      else
          y=1-x
      end if
      print *, 'y=' , y
End Program Sample
```

#### 4.2.2 字符型数据

##### a) 字符变量

字符常数的一般形式是由一对单引号 ‘ ’ 或一对双引号 “ ” 限定的一串字符。字符串中的字符，允许是 PORTAN 字符集的任意字符，如果系统还支持其它字符，例如汉字、希腊字、化学符号、数学符号，也可引入字符串内，用一对 ‘ ’ 或 “ ” 界定。

字符型数据除了有类型、种别外，比其它类型还多了一个长度特性，即规定它有几个字符数。字符型说明语句的关键字是 CHARACTER，其长度说明方法是紧跟在 CHARACTER 后面写一对括号，括号内写 LEN=字符长度。其一般形式是：

CHARACTER[ (LEN=整型字符长度表达式[, KIND=种别值))][, 属性说明] :: 变量名表[=初始值]

其中 LEN 后面的整常数表达式规定被说明字符变量长度，为正整数，LEN 的参数与 KIND 的参数都写在括号内，次序可以任意。在字符型说明语句中，长度说明必须有，不可省略，种别参数可以省略，此时取标准值。仅有关键字 CHARACTER 而没有括号时认为字符是一个字节长。可以省去 LEN=及 KIND=，只写参数值，此时字符长度必须写在前面。只有长度说明的语句可分为有括号和无括号两种，例如：

CHARACTER (LEN=12, KIND=1) :: A, B

CHARACTER (KIND=1, LEN=12) :: A, B

CHARACTER (12, 1) :: A, B

CHARACTER\*12 :: A, B

都是等价的，前者说明 X、Y2 是字符型变量，种别参数为 3。每个变量长度为 12。后者说明表明长度为 12，种别值为 1。

长度也可以写成一个\*号，表示长度暂不确定，待以后与程序中实际需要的长度相一致。例如：

```
CHARACTER(LEN=*), PARAMETER :: C_NAME= 'GIRL'
```

```
CHARACTER(LEN=*), PARAMETER :: C_NAME= 'BOY'
```

都是合法的说明语句，说明字符常量 C\_NAME，前者长度为 4，后者长度为 3。

用字符型变量作为过程的哑元时，可以用正整数作长度，也可以把\*作长度，后者可以与任何长度的实元作哑实结合，相当于以实元的具体长度为哑元的长度。

CHARACTER 后面说明的长度是其后所有实体名的公共长度，如果某一变量的长度与其它不同，可以在其变量名后标出自己的特有长度，方法是在变量名后写上\*及长度。例加：

```
CHARACTER(LEN=12) :: A, B*5, C, D*7, E
```

## b) 字符子串



字符数据中某一部分相连的字符为字符子串，也可以作为一个实体与字符变量一样参加操作。字符子串的一般形式是： $V(e1:e2)$ 。V 是字符型实体名，包括字符变量名、字符函数名、字符数组元素等等。e1, e2 是整型表达式或正整常数，e1 的值指明子串在 V 中的起始列号，e2 的值指明子串在 V 中的终止列号。如果 e1 省略，表示子串从第一个字符取起；e2 省略，表示子串取到末尾；如 e1, e2 都省略，表示子串从头取到尾。例如：设有字符变量 A，其取值为 ‘ABCDE12345FGH’，则下面的子串取值为：

$A(3:11)$	→	‘CDE12345F’，
$A(I+4:9)$	→	‘E1234’ (I=1)，‘1234’ (I=2)
$A(:5)$	→	‘ABCDE’
$A(11:)$	→	‘FGH’
$A(:)$	→	‘ABCDE12345FGH’
$A(3:3)$	→	‘C’

子串在程序中可直接引用，也可被其它字符实体再赋值，因此可使程序员任意地取出一部分字符，

并需要替换一部分字符，非常灵活。例如：PRINT \*, (A(I:I+1), I=6, 9)，可以打印 ‘12’、‘23’、‘34’、‘45’。

### c) 字符操作与赋值

字符操作符只有一个，称**并置符**，为两相连的斜杠“//”，其功能是把前后两个字符串拼连在一起。表达式中的操作对象可以是字符变量，字符常数、字符子串、字符函数值、字符数组元素等。例如：‘AB’//‘CDE’的值为‘ABCDE’。C//‘ABC’//A(2:6)//B(1:4)//D是把几个字符串连在一起。

字符赋值语句的一般形式是：**字符变量=字符表达式**。在字符赋值语句中，要求两边种别值一致，此外还有长度问题。当两边长度不等时，赋值时以左边的字符变量说明长度为准，当右边小于左边时，尾部补空格，否则将右边尾部的多余部分截去。

### d) 字符比较

对每种机器，系统都把本机上可以使用的各种字符按先后排列成某种次序，每个字符在排序中有

确定的位置，称为**排序号**。有了排序号，字符之间就可以比较大小及相等与否，组成关系表达式。

各种系统字符排序不一样，同一字符，在不同机上排序值可能不一样。以 ASCII 码为例，有 128 个字符，它的第一个字符序号为零，最后一个序号为 127。例如 A 的排序号为 65，数码 0 的排序号为 32。序号 0~31 及序号 127 的字符称为控制字符，用来控制操作，它们是非打印字符，屏幕上不可见，通常编程时可予忽略。从 ASCII 码字符集的排序可以看出，小写英文字母排序在大写英文字母之后，它们的序号不同。因此，虽然在书写 FORTRAN 语句时，大小写字母是等价的，但当进行字符操作和字符比较时必须看成是不同的。

字符作比较实际上是比较它们的序号大小。字符比较的一般形式是两端为字符表达式的关系表达式：**<字符表达式><关系操作符><字符表达式>**。例如：‘A’<‘B’的比较结果为真，C1//‘CCTV’>=（‘XY’//C2）//C3 式当 C1、C2、C3 是字符型变量名时是合法的，要先计算两端字符表达式的值，而后比较它们是否满足 $\geq$ 关系。比较的方法随关系操作符而异：

### >、<关系操作符

只要两边表达式第一个字符不等就可求得关系表达式的值。如：‘ABC’>‘CDE’，因第一个字符 ‘A’

<‘C’ 不满足>关系，故表达式的逻辑值为假，后面各位不再比较。如果第一位相等，则对第二位字符作比较得出整个关系表达式值为‘真’或‘假’，如此类推。如果两端字符表达式的值长度不等，则短的一端尾部添空格再按位比较。如：‘ABC12’>‘ABC’，因在 ASCII 码中‘1’的序号为 49，空格的序号为 32。故上述关系表达式的值为真。

### == 关系操作符

先作第一位比较，如不等，关系表达式值为‘假’，如相等，再比较第二位，一直比较到最后一位都相等（不足添空格），才能得出关系表达式值为真，否则值都为假。如作不等/=比较，只要有一位不等就可得出表达式的值为真，否则一直比较到末位。设有字符型变量名为 NAME，其值为‘Einstein’，则关系表达式：

NAME== ‘Einstein’          ->    真

NAME== ‘Ein’// ‘stein’    ->    真

NAME/= ‘Ein’// ‘Stein’    ->    真

NAME== ‘EINSTEIN’        ->    假

## e) 字符函数

F90 的内部函数中有许多与字符型操作有关的函数供用户使用,可以很方便地进行各种文本编辑。

### ● 字符与数值相互转换的函数: ICHAR, CHAR, IACHR, ACHAR

F90 允许把字符型的值转换成整型值(已知字符,取字符型的序号数值),也允许把数值型的值转换成字符型的值(已知序号,取它对应的字符)。函数 ICHAR 的功能是把字符型转换成数值,它的引用方式是: ICHAR(字符实体)。这里自变量的字符实体不一定是单个字符,也可以是字符变量、字符串、字符数组元素、字符函数值等等以及由它们组成的字符表达式。函数值 ICHAR 返回字符实体第一个字符的序号值。如果指明要按照 ASCII 码求它的序号,则本功能的函数名应改写为 IACHAR,例如:

```
print *, ichar('A'), ichar('A_dog'), ichar('A_dog'// 'in_my_room')
```

打印的结果是 3 个数字 65(A 的序号是 65)。

函数 CHAR 的功能是把一个序号转换成它的字符。它的引用方式是: CHAR(整型数值表达式)。先计算整型表达式的值,再以这个值为序号,返回相应的单个字符,如 CHAR(65)的值是 'A'。如果按照

ASCII 码求它的对应字符,则函数名为 **ACHAR**。函数 CHAR 可以有两个自变量,此时引用形式为: **CHAR(序号值, 种别值)**。缺省种别值时表示取标准值即 ASCII 码。

有了字符与数值的转换函数,就可处理字符循环。例如设要连续打印 DIMENSION A, DIMENSION B, ..., DIMENSION Z, 则程序可写成:

```
DO I=0, 25  
    PRINT *, 'DIMENSION '//CHAR(I+6)  
END DO
```

### ● 求字符串长度的函数: **LEN**

函数 LEN 的功能是求出一个字符实体的长度, 引用方式是: **LEN(字符实体)**。函数值返回一个正整数, 为该字符实体的长度 (包括尾随空格、中间空格在内)。如: LEN('ABCD 12 ')的值是 8 (6 个字符数加 2 个空格数)。

### ● 不计尾部空格的字符串长度函数: **LEN\_TRIM**

如：LEN\_TRIM(‘ABCD 12 ’)的值是 7（6 个字符数加 1 个空格数）。

### ● 求子串在父串位置的函数：INDEX

函数 INDEX 的功能找出指定子串在字符串内的指定位置，其引用方式是：INDEX(父串, 子串)。函数值返回一个正整数，指明子串在父串中是从第几个字符开始的。父串、子串都可以是各种形式的字符实体。如果第二个字符实体不是第一个字符实体的子串，返回函数值为 0。如：INDEX(‘Einstein’, ‘s’)的值 4。

### ● 验证字符串的函数：VERIFY

函数 VERIFY 的功能确认指定的字符集是否包含了给定字符串中所有的字符，其引用方式是：VERIFY(字符串, 字符集)。函数值返回一个正整数，指明字符串中与字符集相异的第一个字符是从左边第几个字符开始的。如果没有相异的字符，则返回函数值为 0。如：VERIFY(‘Einstein’, ‘instant’)的值为 1，VERIFY(‘instant’, ‘Einstein’)的值为 5，VERIFY(‘Einstein’, ‘abcdefghijklmnABCDEFGHIJKLMNOP’)的值为 4。

● 除去尾部空格函数：TRIM

如：TRIM('ABCD 12 ')//'34' 的值是'ABCD 1234'。

● 比较字符大小的函数：LGE、LGT、LLE、LLT

比较字符大小的四个函数都有两个字符型实体作自变量，用来判断它们之间是否满足各种>、<关系的，其函数值是一个逻辑值，满足该不等关系为真，否则为假。它们的引用方式与功能如下：

引用方式	含 义	例
LGE(a1, a2)	a1 是否 $\geq$ a2	LGE('A', 'B')值为假
LGT(a1, a2)	a1 是否 $>$ a2	LGT('A', 'B')值为假
LLE(a1, a2)	a1 是否 $\leq$ a2	LLE('A', 'B')值为真



LLT(a1, a2)	a2 是否 < a2	LLT('A', 'B') 值为 真
-------------	------------	-----------------------

## 派生数据类型

数据结构是指若干个数据的连接方式，一个复杂的数据往往是由若干个不同类型数据形成的结构。派生类型是指用户利用 FORTRAN 系统内部类型，如数值型、逻辑型、字符型等自行设计出一个新的数据类型，它们实际上是由内部类型数据形成的某种结构。本章主要目的是学会按复杂数据的客观结构形态，由程序员定义出一种派生类型，再结合上将在后面叙述的模块后，可将该类型必需的操作写成内部子程序，连同派生类型一起写在模块中，供程序各单元共同使用，成为数值计算特别是信息管理的有力工具。

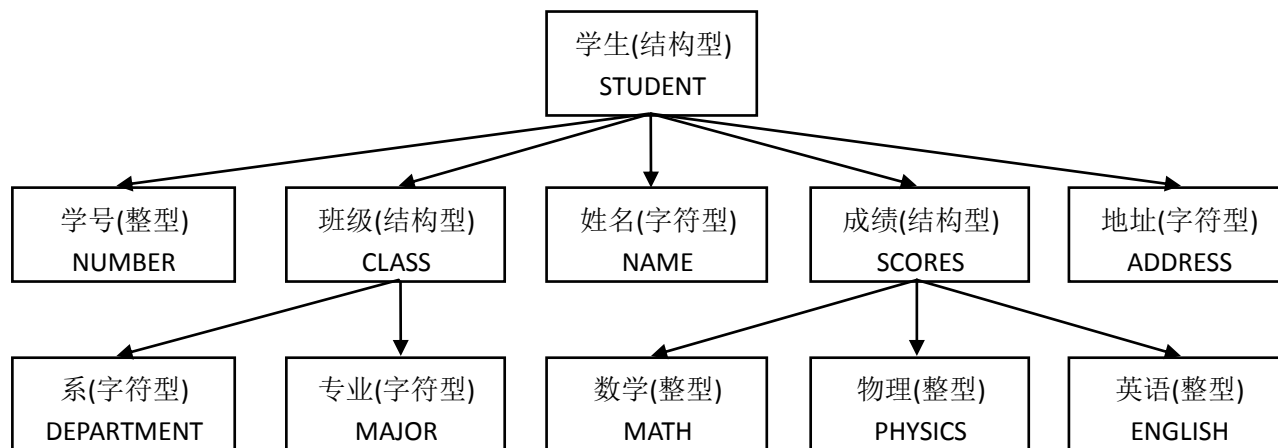
### 4.3.1 数据结构

实际生活中的数据，不像数学上的那么理想，只是一个整数、一个实数或一串字符等，实用的数

据往往是由许多单一数据彼此联系而形成一种结构。例如数组就是由许多数按前后次序排列的一种数据结构，也即数据是一种数据结构，数组名就结构名。但是数组这种结构有局限性，它一定要求结构内所有成员都是同一种内部类型：或者都是整型，或者都是逻辑型，等等。对于成员有的是整型、有的是字符的数据结构，就不能用数组结构表示，必须由程序员自行定义派生类型。

本节不讨论数据结构的严密定义，而是通过实例树立数据结构的概念，设计出能反映复杂数据结构的新类型并加以操作。如输入一个班 30 个学生的姓名及成绩，由于姓名是字符型，成绩是整型，而数据成员只能是一种类型，因此要把姓名作一数组，成绩另作一数组，排名次时要分别作相应调整，非常不便。如果能设计出一种新的派生类型，该类型既包含字符型成员，又有整型成员，再以新类型数据组成一个数组（它们都是同一个派生类型，可以组成数组）就可以在输入输出时只写一个数组名就可既传递人名又传递成绩，非常方便。

一个数据结构由若干数据组成，每个组成部分称为该结构的成员。表示结构中成员的一股形式：



**结构名%成员名**。即在所属结构名后写一个百分号（%）而后写出成员本身名称。这样的成员可以像访问变量一样被访问，包括赋值、打印、引用等。

设一个学生记录为一个结构，内由学号、班级、姓名、宿舍、成绩等有关信息组成，它们分别为不同类型。有些成员下还可再由若干数据组成，如图 71 所示。边注的英文名是程序中所取的该成员的名称。在程序中，如要访问整个学生结构，只要在被访问处写上结构名，即写上 STUDENT。访问其

中成员，如成员是一个简单数据，例如学生姓名，则在程序中被访问处写上 STUDENT%NAME 即可。访问成员中结构成员下的简单数据，则用两个%表示。如访问学生所在系，则在被访问处写 STUDENT%CLASS%DEPARTMENT。第 2 个百分号表示 DEPARTMENT 是 CLASS 的成员。如果 DEPARTMENT 下面还有成员，要访问这些成员时后面还需加上 ‘%成员名’。

### 4.3.2 派生类型

为了便于组织数据，F90 允许定义和使用派生数据类型。任何复杂的数据结构，经分析后都可分解为较简单的成员，可用自定义的派生类型反映它。派生数据类型有一个类型名称，它不能和任何内部数据类型或已定义的派生数据类型重名。定义一种派生数据类型后就可以用它来定义变量、命名常量或其它派生类型了。

#### n) 派生类型定义

定义派生类型时必须使用 TYPE 块。TYPE 块应写在程序的说明部分中，通常写在说明的前部，其

一般形式是：

TYPE[, 访问属性说明::] 派生类型名

成员 1 类型说明

.....

成员 n 类型说明

END TYPE [派生类型名]

其中，TYPE 是关键字，表示派生类型定义开始。访问属性说明关键字是 PUBLIC 或 PRIVATE，默认值是 PUBLIC，即访问方式是共用的。PRIVATE 表示该类型是专用的，这个关键字只有当 TYPE 块写在模块说明部分中时，才允许使用。如果不是在模块内定义的派生类型，不可使用 PRIVATE。派生类型名是任意取的，一旦定义完成，该类型名就成为一个新的类型，就像整型、实型、逻辑型等一样，按一种类型使用。例如可以把各种变量、各种数组说明为这种新的类型，而后按新类型特有法则操作。通常，类型的取名与物理对象的名称一致。例如上述学生结构定义成派生类型可以取派生类型名是 STUDENT。

TYPE 语句下面是结构中各成员的类型说明语句，被说明的类型一般就是 FORTRAN 内部类型（整型、实型等），也允许用一个层次较低的派生类型作为某成员的类型。派生类型的说明语句的一般形式是：**TYPE(派生类型名)::变量名**。TYPE 块中只有类型说明语句，不允许有可执行的动作语句。派生类型中字符型成员不可取\*为长度，必须是确定长度。系统不一定会按定义时的顺序储存各个成员，除非在定义中包含有 **SEQUENCE** 语句。

例如：对上面的学生记录作派生定义，由于学生结构中有两个成员 CLASS 与 SCORES 本身又是次一层的结构，因此要先对 CLASS 与 SCORES 这两个结构作出派生类型定义。CLASS 下有两个成员：DEPARTMENT(字符型)、MAJOR(字符型)。SCORES 下有三个成员：MATH(整型)、PHYSICS(整型)、ENGLISH(整型)。整个学生结构定义派生类型名为 STUDENT\_TYPE。

```
TYPE CLASS_TYPE
    CHARACTER(LEN=50) :: DEPARTMENT, MAJOR
END TYPE CLASS_TYPE

TYPE SCORES_TYPE
```

```
        INTEGER(1) :: MATH, PHYSICS, ENGLISH
    END TYPE SCORES_TYPE

    TYPE STUDENT_TYPE

        SEQUENCE

        INTEGER(4) :: NUMBER

        TYPE(CLASS_TYPE) :: CLASS

        CHARACTER(LEN=10) :: NAME, ADDRESS

        TYPE(SCORES_TYPE) :: SCORES

    END TYPE STUDENT_TYPE

    TYPE(STUDENT_TYPE), DIMENSION(40) :: STUDENT
```

注意在上面三个定义派生类型 TYPE 语句中，派生类型名取成‘名\_TYPE’形式，这是为了加强可读性。特别是当它的成员名与类型名一致时，可以这样区分。当给定属性 DIMENSION(40) 后，STUDENT 即为有 40 个学生的派生型数组，每个学生按顺序有学号、班级(系、专业)、姓名、地址、成绩(数学、

物理、英语)等成员。

### o) 缺省初始化

F95 中，允许定义派生数据类型成员时给予初始值，这时就有了缺省初始化，但没有必要为每一个成员指定初始值。它的显式初始化会覆盖缺省初始化。例如：

```
TYPE REPORT
    CHARACTER (LEN=20) REPORT_NAME
    INTEGER DAY
    CHARACTER (LEN=3) MONTH
    INTEGER :: YEAR = 1999      ! 年份这个成员有缺省初始值
END TYPE REPORT
```

而下面语句中，显式的初始化覆盖了 NOV\_REPORT 中的 YEAR 成员。

```
TYPE (REPORT), PARAMETER :: NOV_REPORT=REPORT("Sales", 15, "NOV", 2001)
```



又如，下面用 DATA 语句进行显式初始化：

```
TYPE EMPLOYEE
    INTEGER ID
    CHARACTER(LEN=40) NAME
END TYPE EMPLOYEE

TYPE(EMPLOYEE) MAN_NAME, CON_NAME

DATA MAN_NAME/EMPLOYEE(417, 'Henry Adams')/

DATA CON_NAME%ID, CON_NAME%NAME /891, "David James"/
```

## p) 结构构造函数

F90 有一种与结构有关的函数，称为结构构造函数，它用来给某结构类型中一个具体的变量赋值。定义了一个派生类型，实际上也同时建立了一个结构构造函数，这个函数名就是派生类型名，函数的自变量就是派生类型内各成员。只要派生类型一建立，就可直接调用这个结构构造函数。如果把实在

的各成员值作为实元，与结构构造函数中自变量作哑实结合，其函数值就是被定义成派生类型的一个变量的值，可以直接赋给派生类型中的某一变量名。例如，对下面的派生类型：

```
TYPE FRIEND_TYPE  
    CHARACTER(LEN=20) :: NAME  
    INTEGER :: AGE  
END TYPE FRIEND_TYPE
```

系统中就自动生成一个函数名叫 FRIEND\_TYPE 的结构函数，该函数有两个哑元自变量：NAME、AGE。其完整的函数形式为：FRIEND\_TYPE(NAME, AGE)。假设变量名 MY\_FRIEND 被说明为 FRIEND\_TYPE 类型，即程序中有说明语句：

```
TYPE(FRIEND_TYPE) :: MY_FRIEND, MY_BOY_FRIEND, MY_GIRL_FRIEND
```

于是可把 ‘Kong Ming’ 作为 NAME 的实元，25 作为 AGE 的实元，在程序执行部分中调用结构构造函数，并把函数值赋给变量 WHO，赋值语句为：MY\_FRIEND=FRIEND\_TYPE(‘Kong Ming’, 25)。此时具有 FRIEND\_TYPE 类型的变量 MY\_FRIEND 就有了 NAME 成员值和 AGE 成员值。

调用结构构造函数作哑实结合时，只要保持类型、个数、位置一致，可以有多种形式的实元与哑元结合。譬如实元可以又是另一低层次结构的结构构造函数，或另一个结构的成员等。如回到前面较复杂的学生结构例中，设已定义派生类型 STUDENT\_TYPE，并说明了变量名 ZHANG\_FEI 是 STUDENT\_TYPE 类型的，而后我们在执行部分中对 ZHANG\_FEI 的成员 CLASS 通过如下赋值语句赋值：

```
ZHANG_FEI%CLASS=CLASS_TYPE ('22th Department', 'Applied Physics')
```

这里引用 CLASS\_TYPE 的结构构造函数，说明变量 ZHANG\_FEI 的班级是 22 系应用物理专业，也即 ZHANG\_FEI%CLASS 已经有定义。因此接着可以把 ZHANG\_FEI%CLASS 作为结构构造函数 STUDENT\_TYPE 的实元，把整个函数值赋给变量 ZHANG\_FEI：

```
ZHANG_FEI=STUDENT_TYPE (0022123, ZHANG_FEI%CLASS, &  
    'ZHANG_FEI', 'He Fei', SCORES_TYPE (82, 73, 90))
```

使用结构构造函数可以非常方便地给具有这种结构的变量赋值，如果若干变量形成一个数组，就可打印出一份详细的档案，并可按照需要排序及检索。

#### q) 应用

例：利用成员特征作排序检索。如果上面 40 个学生姓名和成绩已经读入，要查找名叫张飞的成绩并打印输出，并列出数学成绩在 85 分以上的名单。将前面的派生类型定义简化后为：

```
PROGRAM STUDENT_IO  
  
    TYPE SCORES_TYPE  
        INTEGER(1) :: MATH, PHYSICS, ENGLISH  
    END TYPE SCORES_TYPE  
  
    TYPE STUDENT_TYPE  
        CHARACTER(LEN=10) :: NAME  
        TYPE(SCORES_TYPE) :: SCORES  
    END TYPE STUDENT_TYPE  
  
    TYPE(STUDENT_TYPE), DIMENSION(40) :: STUDENT  
  
    READ '(A, 3I3)', STUDENT  
  
    DO I=1, 40
```

```
        IF (STUDENT(I) %NAME= 'Zhang Fei') PRINT *,STUDENT(I) %SCORES  
    END DO  
  
    DO I=1, 40  
        IF (STUDENT(I) %SCORES%MATH>=85) PRINT *,STUDENT(I) %NAME  
    END DO  
  
END PROGRAM STUDENT_IO
```

例：建立一个计算机帐户的数据库，根据用户号码查找其帐上余额并显示。 [\[e 432 01. f90\]](#)

例：建立一个学生的数据库，包含有学生姓名和考分的记录。从键盘上输入这些记录，并将它们放入一直接存取的文件中，要求可以显示、增加和修改记录。 [\[e 432 02. f90\]](#)

### 【作业】

[4.1] 对任意读入的 100 个英文字母（可以从书中任选一段），计算字母 e 出现的频率，打印输出。

[4.1] 设机内已设有口令“physics”，编一程序，能提示用户键入口令字 (Password)，再把用户键入

的口令与 physics 作比较，若符合则继续，否则提示“口令错误，重新键入”

- [4.1] 利用派生类型编一数据结构程序，描述本班同学个人拥有的微机的硬件指标，包括：（购买者、购买价、购买日期）、（CPU 时钟频率、CPU 芯片厂家）、（硬盘容量、内存大小）、（显示器尺寸、显示器类型）、是否 DVD 光驱、微机品牌。按时钟频率及购买价排序并打印。

## 第五章:数组

### 数组类型与定义

#### 5.1.1 定义数组

F90 中变量的概念与 F77 的不同，它包含了两种变量，一种是标量，一种是数组。数组是科学和工程计算问题中常见的向量和矩阵的反映和概括。数组在 FORTRAN 程序中有着重要的意义，在批量大的情况下，如果不利用数组就失去了计算机的优越性。

#### v) 数组的描述

数组是类型相同的一组标量数据的有序集合，即要求这些数据都必须类型相同，并按某种确定方式排列。向量是一维数组，矩阵可看成是二维数组。类型可以是整型、实型、双精度型、逻辑型等任何一种。组成数组的每一个元素称为数组元素。

数组的维数称为秩(rank)，F90 中规定数组最多可以有 7 维。在某一维中元素的个数称为该维的长度(extent)。数组中所有元素的个数称为数组的大小(size)，它等于各维的长度的乘积，数组的大小可以为 0。数组的形状(shape)取决于秩和每一维的长度。

例：REAL A(10,3,2)的秩为 3，大小为  $10 \times 3 \times 2 = 60$ ，形状为 10 乘 3 乘 2，或表示成(10, 3, 2)。

数组的形状可用内在函数 SHAPE 得到，如 SHAPE(A)的结果是一维数组，其元素取值为 10,3,2。

每一维的大小都由一个下界和一个上界来指定，之间以冒号分开，即下界:上界。维界表达式是整型的数学表达式，维界值可以是正、负或零，但维上界必须大于维下界的值。维长=上界-下界+1。声明数组时下界可以省略，此时维下界为默认值 1。例如：REAL B(0:9,-1:1,4:5,9)。

数组可以用以下几种声明语句定义，F77 中为类型说明语句、DIMENSION 语句和 COMMON 语句，

F90 中还可以用类型属性说明、POINTER 语句、ALLOCATABLE 语句。例如下面都是合法的数组声明：

```
REAL          A(10,2,3)  ! 类型说明
```

```
DIMENSION    A(10,2,3)   ! DIMENSION 语句
```

```
ALLOCATABLE B(:, :)      ! ALLOCATABLE 语句
```

```
POINTER      C(:, :, :)  ! POINTER 语句
```

```
REAL, DIMENSION(2,5):: D    ! 类型说明中的 DIMENSION 属性
```

```
REAL, ALLOCATABLE:: E(:, :, :) ! 类型说明中的 ALLOCATABLE 属性
```

```
REAL, POINTER:: F(:, :) ! 类型说明中的 POINTER 属性
```

无论何种情况，数组的秩总是要指定的。数组的大小和形状是否需要声明视数组的形式而定。当 COMMON 语句指定了数组的秩和大小后，只能定义数组的类型而不能再次重复给定维界。

## w) 数组元素



在程序执行部分中，数组的三种成份可供运算或输入输出：

- 1、数组名     -> 代表数组中所有元素；
- 2、数组元素   -> 代表数组中某一个元素；
- 3、数组片段   -> 代表数组中若干个元素，它们可以相连或分离。

数组中单独的标量称为元素。标量的秩为 0，而数组的秩至少是 1。如果没有下标则指整个数组。对每一维指定确切的一个下标则定义了一个数组元素，还可以通过下标可以引用数组的一部分元素。例如 A 指整个数组，A(1)指数组 A 的第 1 个元素，A(3:4)指数组 A 的第 3 和第 4 个元素，A(1:10:2)指的是数组 A 的第 1,3,5,7,9 个元素。

数组的下标必须用逗号隔开，下标是整型常量、变量或表达式，可正、可负、也可以为 0，但必须在引用的数组的维数之内。引用下标的个数要和声明的数组的维数一致。可以使用函数或数组元素作为下标：

例：REAL A(3,3),B(3,3),C(89),R

$B(2,2)=4.5$ ;  $R=7.0$ ;  $C(\text{INT}(R)*2+1)=2.0$  !给元素  $B(2,2)$ 和  $C(15)$ 赋值

$A(1,2)=B(\text{INT}(C(15)),\text{INT}(\text{SQRT}(R)))$  !元素  $A(1,2)$ 和元素  $B(2,2)$ 的值相同

### x) 数组片段

数组片段是数组所有元素集合的一个子集。数组片段的元素可以是数组中任意的元素，它们不需连续或遵循某个规则。数组中的所有元素和片段的数据类型和种别都相同。如果指定数组的所有下标则得到的是数组元素(即标量)，如果只指定部分下标则结果是部分数组元素的集合，即数组片段，数组片段本身也是数组。

例如，如果定义  $\text{REAL } A(2,3,4)$ ，则  $A(1,2,3)$ 是数组元素，而  $A(1:2,2,2)$ ,  $A(1,1,4:2:-1)$ ,  $A(1,2:3,(/2,4/))$ 都是数组片段。

数组片段由下标列表确定，**下标列表**有两种：三元下标和向量下标。

### y) 三元下标

三元下标用三个值分别代表数组片段的下界，上界和步长。其一般形式为：**[下界]:[上界][:步长]**。如果省略下界，缺省值为数组相应维的下界；如果省略上界，缺省值为数组相应维的上界；如果省略步长，缺省值为 1。如果下标都省略了则缺省片段为这个维的全长。

例：REAL A(10)

A(1:5:2)=3.0   !把元素 A(1)，A(3)，A(5)置为 3.0

A(:5:2)=3.0   !把元素 A(1)，A(3)，A(5)置为 3.0，因为缺省下界为 1

A(2::3)=3.0   !把元素 A(2)，A(5)，A(8)置为 3.0，因为上界缺省值为 10

A(7:9)=3.0    !把元素 A(7)，A(8)，A(9)置为 3.0，因为缺省步长为 1

A(:)=3.0       !和 A=3.0 相同，将 A 的所有元素置为 3.0

对于一个多维数组的数组片段，它的每一维都可以用三元下标来声明。如果要在一个语句或过程中引用这个数组片段，则引用下标要和声明时的下标个数一样多。注意，三元下标只能算一个下标。

例: REAL A(5,9)

A(1:4:3,6:8:2)=3.0

此例中数组 A 是二维数组, 数组片段是形状为(2,2)的二维数组。上面括号内第一个三元组表示第一维下标变化范围, 第二个三元组表示第二维下标变化范围。其元素选取法是先把第二维下标定在下界值上, 而后遍历第一维下标, 选中元素为 A(1,6), A(4,6); 再把第二维下标增一步长, 再遍历第一维下标, 选中元素为 A(1,8), A(4,8), 如此重复直至全部选完。这种选取关系相当于把第二维作外层 DO 循环变量, 第一维作内层循环变量的变化。这四个元素按先后次序保持一定的形状, 上式等价于:

$$\begin{pmatrix} A(1,6) & A(1,8) \\ A(4,6) & A(4,8) \end{pmatrix}$$

数组的步长不能是 0。当步长为负值时, 数组子片段从上界开始递减至下界。例如声明一个数组 B(10), 则数组片段 B(9:2:-2)是由元素 B(9), B(7), B(5)和 B(3)组成的数组。显然下界不能比上界大, 否则产生的数组大小为 0。三元下标的值可以不在数组的边界以内, 例如对于数组 B(10), 数组片

段 B(3:15:6)是由 B(3)和 B(9)组成的数组。

## z) 向量下标

三元下标以上升或下降的顺序指定数组元素，而向量下标可以以任何顺序来指定数组元素。向量下标是一个一维整数数组(即向量)，它可以从整个数组中选择片段。

例：REAL A(10),B(5,5)

INTEGER I(4),J(3)

I=(/5,3,8,2/)     !定义向量 I

J=(/3,1,5/)       !定义向量 J

A(I)=3.0           !设置元素 A(5)，A(3)，A(8)，A(2)的值

B(2,J)=3.0         !设置元素 B(2,3)，B(2,1)和 B(2,5)的值

例：INTEGER :: a(4)=(/0,1,2,3/),b(3)=(/1,4,3/)

则 a(b)与 a 同类型，与 b 同形状，取值为(/0,3,2/)。a(b)可以不是数组片段，而是更大的数组。如上面 b 为(/2,3,2,3,2,3/)时，a(b)为(/1,2,1,2,1,2/)。

例：CHARACTER(1) :: symbol(0:1)=(/'F','M'/)

INTEGER :: bit(100)

若 bit 的元素列为 0001101100111...，则 symbol(bit)是用{F,M}字符构成的 100 字节的字符型数组 FFFMMFMMFFMMM...。

向量下标的值应该在定义的边界之内，向量下标可以有多个重复的值，此数组片段称为多对一数组片段。

例：REAL A(3,3),B(4)

INTEGER K(4)

K=(/3,1,1,2/)     !K 矢量有重复值

A=5.0                !设置 A 的所有元素

B=A(3,K)            !数组片段 B 由下列元素组成: A(3,3),A(3,1),A(3,1),A(3,2)

因为在 B(4)或 A(3,K)中有重复的元素 A(3,1)，所以它是多对一数组片段。一个多对一数组片段不能出现在赋值语句的左端。

### 5.1.2 数组类型

#### a) 显式形状 (Explicit-shape) 数组

这种数组指定了所有特征：固定的秩、每一维的长度和形状。其中下界是可以忽略的。它的一般形式是：([下界:]上界[[下界:]上界]...)

例：INTEGER M(10,10,10),K(-3:6,4:13,0:9)

维界可以是常数或变量。在过程(函数和子程序)中，数组的上界和下界可以由变量或表达式指定。使用变量或表达式的数组是自动数组和可调数组。

### b) 自动 (Automatic) 数组

自动数组是显形数组的一种，它是过程中的局域变量，自动数组必须在过程中加以声明，并且它的上下界是不定的表达式。在调用过程时，上下界通过变量或表达式求出。过程中其后的变量或表达式值的变化不会对数组的上下界产生影响。

例：SUBROUTINE EXAMPLE(N,R1,R2)

DIMENSION A(N,5),B(10\*N)

.....

N=IFIX(R1)+IFIX(R2)

此例中的 A 和 B 都是自动数组。子程序被调用时，数组 A 和 B 的上界通过传入的变量 N 来确定，



而以后 N 的值的变化对 A 和 B 的大小不会有影响。

例: SUBROUTINE SUB1(A,B)

INTEGER A,B,LOWER

COMMON/BOUND/LOWER

.....

INTEGER AUTO\_ARRAY1(B)

INTEGER AUTO\_ARRAY2(LOWER:B)

INTEGER AUTO\_ARRAY3(20,B\*A/2)

.....

END SUBROUTINE

c) 可调 (Adjustable) 数组

可调数组也是显形数组的一种，它是过程的一个哑元，至少有一个维界不是常数，这个维界当过程被调用时才被确定。其维界表达式中的变量是哑元，或者是通过 **COMMON** 语句中传递的常量。注意多维可调数组的维界表达式必须与调用时的维界相符。

例： **DIMENSION A1(10,35),A2(3,56)**

.....

**SUM1=THE\_SUM(A1,10,35)**

**SUM2=THE\_SUM(A2,3,56)**

**END**

**FUNCTION THE\_SUM(A,M,N)**

**DIMENSION A(M,N)**

**SUMX=0.0**

```
DO J=1,N  
    DO I=1,M  
        SUMX=SUMX+A(I,J)  
    END DO  
END DO  
  
THE_SUM=SUMX  
  
END FUNCTION
```

其中哑元 **M,N** 控制着可调数组的大小。下例说明了可调数组的维界 **X(-4:4,5)**在调用过程中被确定后，过程内部对维界参数的赋值不会改变维界值。

例： **DIMENSION ARRAY(9,5)**

**L=9**

**M=5**

```
CALL SUB(ARRAY,L,M)
```

```
END
```

```
SUBROUTINE SUB(X,I,J)
```

```
DIMENSION X(-I/2:I/2,J)
```

```
X(I/2,J)=999
```

```
J=1
```

```
I=2
```

```
END
```

#### d) 假定形状 (Assumed-shape) 数组

这种数组是在过程中的哑元，它从实际传递过来的数组获得形状参数。假定形状数组的秩由冒号

的个数决定。它的一般形式是：**([下界]:[下界]:...)**。如果不指定下界，则默认值为 1。上界值=过程调用时实参数组对应维的长度+下界值-1。注意它与可调数组的区别在于,可调数组属于显型数组（必须指定上界）的范围，而假定形状数组的上界是不能指定的。

例：SUBROUTINE ASSUMED(A)

```
REAL A(:, :, :)
```

此时数组 A 的秩为 3，但每一维的长度待定。当过程被调用时 A 将从传递到过程的数组获得形状：

```
REAL X(4,7,9)
```

```
CALL ASSUMED(X)
```

于是 A 获得了数组维界(4,7,9)，实际数组和假定形状数组的值必须相同。如果上面过程中数组声明了 A(3:,0:,-2:)，以哑元 X(4,7,9)调用过程时，数组 A 的实际维界是 A(3:6,0:6,-2:6)。

应用假定形状数组为哑元的过程时必须有显式的接口 **INTERFACE 语句**。

例：INTERFACE

```
SUBROUTINE SUB(M)

    INTEGER M(:,1:,5:)

END SUBROUTINE

END INTERFACE

INTEGER L(20,5:25,10)

CALL SUB(L)
```

在此例中数组 M 的维界是(1:20,1:21,5:14)。

#### e) 假定大小 (Assumed-size) 数组

这种数组是在过程中的哑元，它从实际传递过来的数组获得数组大小。除了最后一维的上界以外，其它所有特征（秩，长度和维界）都必须指定。声明一个假定大小数组时，最后一个的上界用星号\*表示。它的一般形式是：([显型维界],[显型维界],...[下界:]\* )。

例: SUBROUTINE ASSUME(A)

REAL A(2,2,\*)

假定大小数组的秩和形状可以和实际传入的数组不同，传入的数组只确定它的大小。实际数组的元素按列传递给假定大小数组，假定大小数组也按列接收。接受的过程中假定大小数组的最后一维的长度会改变来接受所有传递进来的数组元素，于是最终给出数组的大小。如上例子中的 ASSUME 子程序，如果以数组 X 为哑元来调用的话，

REAL X(7)

CALL ASSUME(X)

则数组 X 的元素与数组 A 的对应顺序是：

X(1): A(1,1,1)

X(2): A(2,1,1)

X(3): A(1,2,1)

X(4): A(2,2,1)

X(5): A(1,1,2)

X(6): A(2,1,2)

X(7): A(1,2,2)

其中数组 A 的最后一维没有必要成为完整的维，所以数组 A 始终没有确定的形状。因为假定大小数组没有形状，所以这样的数组不能仅仅通过名称来向其它过程传递。

假定大小数组可以分解成确定的数组片段。如上例中的数组 A，可分为三个片段：A(1:2,1:2,1)和 A(1:2,1,2)以及 A(1,2,2)。

假定大小数组的秩是完全确定的维数+1。上例中数组 A 的秩为 3，尽管 A 的第三维不是完整的。

#### f) 延迟形状 (Deferred-shape) 数组

可分配数组必须以延迟形状的形式来声明。它每一维的长度只有在分配数组才被确定。声明迟形



数组时，秩由冒号确定，但长度是未知的。

例：REAL,ALLOCATABLE:: A(:,,:)

INTEGER,ALLOCATABLE,TARGET:: K(:)

可分配数组可由下列方式声明：使用 **ALLOCATABLE 语句**、**DIMENSION 语句**、**TARGET 语句**或在类型声明中使用 **ALLOCATABLE 属性**。如果迟形数组以 **DIMENSION 语句**或 **TARGET 语句**声明，则在其它语句中必须给出 **ALLOCATABLE 属性**。

在迟形数组的大小、形状和维界没有确定之前，其任何部分都不能被引用，可分配数组的大小、形状和维界在 **ALLOCATE 语句**执行时才被确定。 [\[e 512 01.f90\]](#)

例：INTEGER,ALLOCATABLE:: A(:,:)

REAL,ALLOCATABLE,TARGET:: B(:,:),C(:)

ALLOCATE(A(2,3),C(5),B(SIZE(C),12))

## 数组赋值与运算

F90 中，允许把整个数组作为一个操作数进行操作，也允许在赋值语句中对整个数组进行赋值，就像对一个简单变量的操作和赋值一样。针对数组的特性，也有专门的运算方法和内在函数。

### 5.2.1 赋值

#### aa) 赋值方式

数组的赋值可以使用赋值语句和数组构造器。

F90 中数组赋值语句的形式为： $V=e$ ， $V$  代表数组名或数组片段， $e$  为数组表达式，它们必须有相同的形状（即维数相同，每维长度相同但上下界可不同）。当  $V$  大小为 0 或为长度为 0 的字符型时，则没有值赋给  $V$ 。当  $e$  为标量时，则把  $e$  处理成与  $V$  相同的形状， $V$  的每个元素均等于标量  $e$  的值。数组表达式  $e$  中允许使用+、-、\*、/、\*\*等内部算术操作符。

F90 允许赋值语句的表达式内只有标量，如  $A$  是  $2 \times 2$  数组，则  $A=3$  表示  $A = \begin{pmatrix} 3. & 3. \\ 3. & 3. \end{pmatrix}$ 。例如， $A(1:15)=2.0$ ，则  $A(1)$ 到  $A(15)$ 的每个元素都被赋值为 2.0。因此，数组赋值语句可以对整个数组一次赋值完毕，

不必用 DO 构造对每一个元素逐一赋值，如对 F77 中的初始化置零 DO 循环：

```
do i=1,10
```

```
    do j=1,10
```

```
        A(i,j)=0.
```

```
    end do
```

```
end do
```

可以只要一个赋值语句  $A=0.$  就行。

例：  $A(1:50)=E(1:99:2)+F(2:100:2,10)/G(3:101:2,5,5)$

例：  $Q(10:40:10)=A(5:20:5)//B(1:4)$

例： `INTEGER A(20)`

$A(1:20)=A(20:1:-1)$       ! 将 A 数组元素值倒序重排

**数组构造器**是由括号和斜线之间的一系列值或隐 DO 循环组成。其一般形式为：**(/取值列表/)**。取值列表可以是标量，隐 DO 循环或任意秩的数组。其中的所有值的类型都相同，以逗号隔开。如果列表中出现了数组，它的值是按列来赋的。

例：INTEGER A(6)

A=(/1,2,3,4,5,6/)      ! 斜杠与括号间不能有空格

例：C1=(/4,8,7,6/)      ! 标量表示

C2=(/B(I,1:5),B(I:J,7:9)/) ! 数组表示

C3=(/(I,I=1,4)/)      ! 隐 DO 循环

C4=(/4,A(1:5),(I,I=1,4),7/) ! 混合表示

下面是一些**数组构造器的替换格式**：

1) 用**方括号**代替括号和斜线，例如下面两个数组构造器是等价的：

例: INTEGER C(4)

C=(/4,8,7,6/)

C=[4,8,7,6]

2) 冒号三元下标(代替隐 DO 循环)来指定值的范围和步长, 例如下面两个数组构造器是等价的:

例: INTEGER D(3)

D=(/1:5:2/) ! 三元下标格式

D=(/(I,I=1,5,2)/) ! 隐 DO 循环格式

### bb) 数组的存储顺序

由于计算机的内存是一维的, 所以不管数组是几维, 它在内存中都是按一维来存储的, 数组的存储顺序对应于输入输出时给定数组元素数据的顺序。FORTRAN 规定, 一维数组在机内存贮时是按序存放的。而对二维数组, 用第一个下标指明行序, 第二个下标指明列序, 则数组是按列存放的。

即按序存放第一列内的诸元素，接着是第二列内诸元素，然后第三列、第四列直到最后一列。如  $A$  是  $3 \times 3$  数组，

$$\begin{pmatrix} A(1,1) & A(1,2) & A(1,3) \\ A(2,1) & A(2,2) & A(2,3) \\ A(3,1) & A(3,2) & A(3,3) \end{pmatrix}$$

则它在机内的存放顺序是  $A(1,1) \rightarrow A(2,1) \rightarrow A(3,1) \Rightarrow A(1,2) \rightarrow A(2,2) \rightarrow A(3,2) \Rightarrow A(1,3) \rightarrow A(2,3) \rightarrow A(3,3)$ 。

高维数组的存放次序可由二维数组类推，即最右边指标（相当于最外层循环变量）变动最慢，最左边的第一个指标变动最快。

### cc) 与 DO 循环的差异

F90 的赋值语句考虑到了在并行机上计算的功能，即使不是用的并行计算机，在形式上也是按并行化处理的。这与 F77 中的串行赋值是不同的，由此造成了数组的 F90 赋值结果与采用 DO 循环方式进行赋值的差异。

例: INTEGER :: a(0:9)=(/0,1,2,3,4,5,6,7,8,9/)

a(1:9)=a(0:8)

a 的所有元素是并行处理的, 结果是(/0,0,1,2,3,4,5,6,7,8/):

|a(0)|a(1)|a(2)|a(3)|a(4)|a(5)|a(6)|a(7)|a(8)|a(9)|

↑     ↑     ↑     ↑     ↑     ↑     ↑     ↑     ↑

|a(0)|a(1)|a(2)|a(3)|a(4)|a(5)|a(6)|a(7)|a(8)|a(9)|

但是, 如果用 DO 循环的话

DO i=1,9

a(i)=a(i-1)

END DO

A 的元素是逐一处理的, 结果是(/0,0,0,0,0,0,0,0,0,0/):

|a(0)|a(1)|a(2)|a(3)|a(4)|a(5)|a(6)|a(7)|a(8)|a(9)|

→      →      →      →      →      →      →      →      →

①      ②      ③      ④      ⑤      ⑥      ⑦      ⑧      ⑨

要用 DO 循环达到和上面同样的效果，需要数组的拷贝：

```
INTEGER :: a(0:9)=(/0,1,2,3,4,5,6,7,8,9/), b(0:9)
```

```
b=a
```

```
DO i=1,9
```

```
    a(i)=b(i-1)
```

```
END DO
```

#### dd) RESHAPE 语句

可以用内在函数 RESHAPE 语句把一个一维数组改变形状后赋给另一已知形状数组。由于 Fortran



的数组顺序不同于 C，Fortran 中左边第一维是变化最快的，RESHAPE 语句可以用于当 Fortran 过程化为 C 过程时将 Fortran 数组改序成 C 数组序。它的一般形式为：**结果=RESHAPE(源,形状[,补充[,顺序]])**。**源**可以是任意数据类型的数组，它提供结果数组的元素。当补充被省略或其大小为 0 时，源的大小必须大于 PRODUCT(形状)。**形状**为 7 个元素以下的一维固定的整型数组，它决定了结果数组的形状，其大小为正元素非负。**补充**为与源数组相同类型和属性的数组，当源数组的大小比结果数组小时用来补充元素值。**顺序**必须是和形状数组有相同形状的整型数组,其元素为(1,2,...,n)的排列，n 是形状数组的大小。当顺序被省略时，默认值为(1,2,...,n)。

因此，**结果**数组的形状与形状数组相同，与源数组的类型和属性相同，大小是形状数组元素值的乘积。结果数组的元素是源数组的元素按顺序数组指定的维别顺序排列的，当省略顺序数组时元素按通常的顺序排列。源数组元素用完后按序用补充数组的元素，必要时重复使用补充数组直至结果数组的所有元素均有其值。

例：RESHAPE((/3,4,5,6,7,8/),(/2,3/))的结果是 $\begin{pmatrix} 3 & 5 & 7 \\ 4 & 6 & 8 \end{pmatrix}$ 。

例：RESHAPE((/3,4,5,6,7,8/),(/2,4/),(/1,1/),(/2,1/))的结果是 $\begin{pmatrix} 3 & 4 & 5 & 6 \\ 7 & 8 & 1 & 1 \end{pmatrix}$ 。

例：INTEGER AR1(2,5)

REAL F(5,3,8),C(8,3,5)

AR1=RESHAPE((/1,2,3,4,5,6/),(/2,5/),(/0,0/),(/2,1/))

! AR1 的取值为 [1 2 3 4 5]

! [6 0 0 0 0]

C=RESHAPE(F,(/8,3,5/),ORDER=(/3,2,1/)) ! 将 Fortran 数组序化为 C 数组序

END

例：INTEGER B(2,3),C(8)

B=RESHAPE((/1,2,3,4,5,6/),(/2,3/)) ! 赋值给形状为(2,3)的数组

C=(/0,RESHAPE(B,(/6/)),7/) ! 赋值给向量 C 之前先把 B 转换成向量

## ee) WHERE 构造

F90 中提供了屏蔽数组赋值语句，它实质上是一种带条件的数组赋值语句，也就是说它只对某些符合条件的数组元素赋值。屏蔽数组赋值语句，为 WHERE 语句和 WHERE 构造，其形式非常类似于逻辑 IF 语句和 IF 构造。

WHERE 语句的一般形式为：WHERE(屏蔽表达式) 赋值语句

WHERE 构造的一般形式为：

[构造名:] WHERE(屏蔽表达式 1)

[块]

[ELSEWHERE(屏蔽表达式 2) [构造名]

[块]]

[ELSEWHERE [构造名]

[块]

END WHERE [构造名]

其中屏蔽表达式都是数组逻辑表达式。这里要求数组赋值语句中被定义的变量和屏蔽表达式必须都是数组，且形状相同，而且数组赋值语句不能是自定义的赋值语句。

在执行 WHERE 语句时，当数组逻辑表达式的某个数组元素的值为.TRUE.时，才对赋值语句表达式的相应元素求值，赋给被定义的数组元素。同样，在执行 WHERE 构造时，逻辑表达式的取值结果(数组元素)为.TRUE.时，执行屏蔽表达式后面的块；而元素值为.FALSE.时，执行 ELSEWHERE 后面的块。注意与 IF 构造不同的是，这里没有关键字 THEN 和 ELSE 块。

例如，要对数组 A 中绝对值大于 1 的元素取倒数，其余的元素值不变，其语句为：WHERE(ABS(A)>1.0) A=1./A。如果同时还要记录哪些元素作了变换，可用数组 B 记录变换情况，语句为：

B=.FALSE.

WHERE(ABS(A)>1.0) B=.TRUE.

WHERE(ABS(A)>1.0) A=1./A

显然，上面最后一行可以写成：WHERE(B) A=1./A

若采用 WHERE 构造，上例可改写为：

B=.FALSE.

WHERE(ABS(A)>1.0)

B=.TRUE.

A=1./A

END WHERE

这样改写后可以提高程序的执行效率。当然，还可以改写成下面的等价形式：

WHERE(ABS(A)>1.0)

B=.TRUE.

$A = 1./A$

ELSEWHERE

B=.FALSE.

END WHERE

例: INTEGER A,B,C

DIMENSION A(5),B(5),C(5)

DATA A/0,1,1,1,0/

DATA B/10,11,12,13,14/

C=-1

WHERE(A.NE.0) C=B/A

则数组 C 的元素为(-1,11,12,13,-1)。

## ff) FORALL 屏蔽赋值

FORALL 是 F95 的新增功能。它是数组屏蔽赋值（WHERE 语句和构造）功能的一对一元素的推广，其方式有 FORALL 语句和 FORALL 构造。

FORALL 语句的一般形式为：FORALL(循环三元下标[,循环三元下标]…[,屏蔽表达式]) 赋值语句

FORALL 构造的一般形式为：

[构造名:] FORALL(循环三元下标[,循环三元下标]…[,屏蔽表达式])

[块]

END FORALL [构造名]

屏蔽表达式是数组逻辑表达式，缺省值为.TRUE.。块是赋值语句，或为 WHERE 语句或构造，或为 FORALL 语句或构造。

循环三元下标的形式为：循环变量=下界:上界[:步长]。循环变量是整型的。步长不能为 0，缺省值

为 1。

例： INTEGER :: mat(0:7,0:7)

FORALL(i=0:7,j=0:7) mat(i,j)=8\*i+j

它等价于

FORALL(i=0:7,j=0:7)

mat(i,j)=8\*i+j

END FORALL

如果要用 F77 的语句写出来的话，则为

DO i=0,7

DO j=0,7

mat(i,j)=8\*i+j

END DO



END DO

例: REAL :: a(0:n)

WHERE(a>0.0) a=SQRT(a)

等价于

FORALL(i=0:n,a(i)>0.0) a(i)=SQRT(a(i))

例: WHERE(A/=0.) B=1./A

等价于

FORALL(I=1:N,J=1:N,A(I,J).NE.0.) B(I,J)=1./A(I,J)

等价于

FORALL (I=1:N,J=1:N)

```
IF(A(I,J).NE.0.) B(I,J)=1./A(I,J)
END FORALL
```

例: INTEGER :: num(0:7)=(/0,1,2,3,4,5,6,7/)

```
FORALL(i=1:7) num(i)=num(i-1)
```

等价于

```
FORALL(i=1:7)
```

```
num(i)=num(i-1)
```

```
END FORALL
```

等价于

```
a(1:7)=a(0:6)
```

不等价于

```
DO i=1,7  
  
    num(i)=num(i-1)  
  
END DO
```

[\[e 521 01. f90\]](#)

## 5.2.2 运算

### a) 基本运算

允许把整个数组或数组片段作为一个单独的对象进行运算。例如当数组 **A,B,C** 有相同大小和形状时，**C=A+B** 语句实现对数组所有元素的并行加法运算。所有的算术运算符(+, -, \*, /, \*\*)、逻辑运算符(如**.AND.**, **.OR.**, **.NOT.**)和所有关系运算符(如**.LT.**, **.EQ.**, **.GT.**)，还有很多内在函数都可以接受数组名称作为参数并对数组元素逐一运算。

例： **INTEGER, DIMENSION(100):: even,odd,plu,min,tim,div,squ**

even=(/(2\*i,i=0,49)/); odd=(/(2\*i+1,i=0,49)/)

plu=even+odd

min=even-odd

tim=even\*odd

div=even/odd

squ=even\*\*2

可以使用数组名作为参数的内在函数称为基本内在函数。

例： REAL A(5),B(5),C(5)

INTEGER D(5)

DATA PI/3.14159265/

A=(/(REAL(I)\*PI/180.,I=1,5)/)

B=COS(A); C=SQRT(A); D=CEILING(A\*180.)

其中 COS, SQRT 和 CEILING 都是基本内在函数。当两个以上数组出现在赋值语句或表达式中时, 数组的形状应该相同(称为**相容**)。例如数组 A(2,3), B(2,3), C(2:3,6:8)都是相容的, 而数组 D(4,5), E(5,4), F(5,2,2)都是不相容的。

例:  $X(1:10)=X(10:1:-1)$  !使用三元下标颠倒数组 X

如果数组片段指定的部分相容, 则它也可以用于表达式和赋值。这样, 不相容的数组也可以互相使用。例如, 一个较小的完整数组可以和一个相容的大数组的片段进行运算:

例: REAL A(5),B(4,7)

A=20.; B=5.; A=A-B(2,1:5)

## b) 数组与数组

两个数组作算术操作的结果仍是一个形状相同的数组, 它的每个位置上元素的值是参与操作的相同位置上一对元素操作后的结果值。例如有数组 A、数组 B, 形状如下:

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix}, \quad B = \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{pmatrix},$$

则执行赋值语句 **C=A+B** 后，数组 **C** 的值即为

$$C = A + B = \begin{pmatrix} a_{11} + b_{11} & a_{12} + b_{12} & a_{13} + b_{13} \\ a_{21} + b_{21} & a_{22} + b_{22} & a_{23} + b_{23} \end{pmatrix}。$$

对于其它内部操作，如+、-、\*、/、\*\*等，操作结果也是一个形状相同的数组。它的每个位置上的元素值是参与操作的数组相同位置上的一对元素（不管它们的下标是否相同）作相同操作的结果。

### c) 数组与标量

数组表达式允许数组与标量作算术运算。当 **A**、**B** 为形状相同的数组时，赋值语句 **A=B+2** 是合法的。表达式是数组 **B** 与标量值 **2** 相加，这时可以把标量看作形状与 **B** 相同的数组，其每一元素的值都是 **2**，而后与 **B** 数组相加，其结果为：

$$A = B + 2 = \begin{pmatrix} b_{11} + 2 & b_{12} + 2 & b_{13} + 2 \\ b_{21} + 2 & b_{22} + 2 & b_{23} + 2 \end{pmatrix},$$

也即数组与标量操作相当于数组内每一元素与该标量操作。

#### d) 数组内在函数

数组表达式中允许对数组求基本函数，其函数值仍是一个形状相同的数组，它的每个位置上的元素值就是被操作数组对应位置元素取该函数值，例如 **A**、**B** 为形状相同的一维数组，则语句 **B=SQRT(A)** 表示：**B(1)=SQRT(A(1))**、**B(2)=SQRT(A(2))**,…。

F90 中增加了许多新的数组专用内在函数，使得数组的运算更加灵活方便。下面是一部分新增的数组内在函数。

##### ✦ 矩阵乘积函数：**MATMUL(A,B)**

描述：执行数值或逻辑型矩阵 **A** 与 **B** 的矩阵乘法。

说明：矩阵 **A** 和 **B** 必须是秩为 1 或 2 的数值型或逻辑型的有值数组，且 **A** 与 **B** 的类型必须相同，**A**、**B** 中至少有一个秩是 2。**A** 与 **B** 的矩阵乘积规则和结果值与数学上的定义相同，**A** 的最后一维的长度必须和 **B** 的第一维的长度相同。

✚ 向量点乘函数: `DOT_PRODUCT(A,B)`

描述: 执行数值或逻辑型向量 A 与 B 的点积乘法。

说明: 向量 A 和 B 必须是秩为 1(即向量)的数值型或逻辑型的有值数组, 且 A 与 B 的类型必须相同。

向量 A 与 B 点乘的结果是标量, 其规则和结果值与数学上的定义相同。

例: `DOT_PRODUCT((/1,2,3/),(/2,3,4/))` 的值为 20。

✚ 数组归约函数: 包括 SUM、PRODUCT、MAXVAL、MINVAL、COUNT、ANY 和 ALL 函数。

★ 元素求和函数: `SUM(ARRAY[,DIM][,MASK])`

描述: 沿着维 DIM, 对在 MASK 真值中的数组 ARRAY 的所有元素求和。

说明: ARRAY 是被求和的数组名。DIM 用于指明选哪一维来求函数值。当 “DIM=1” 时分别按列求和, “DIM=2” 时分别按行求和。MASK 是屏蔽表达式, 不满足条件的元素则被屏蔽, 不参加求函数值。

例: `SUM((/4,5,6/))` 的值是 15。



例：SUM(C,MASK=C>0.0)是对 C 的所有正元素值的求和。

例：若  $A = \begin{pmatrix} 2 & 3 & 4 \\ 1 & 2 & 3 \end{pmatrix}$ ，则SUM(A,DIM=1)的值是[3,5,7]，SUM(A,DIM=2)的值是[9,6]。

★ 元素连乘求积函数：PRODUCT(ARRAY[,DIM][,MASK])

描述：沿着维 DIM，对在 MASK 真值中的数组 ARRAY 的所有元素求连乘积。

例：PRODUCT((/4,5,6/))的值是 120。

例：PRODUCT(C,MASK=C>0.0)是对 C 的所有正元素值的求连乘积。

例：若  $A = \begin{pmatrix} 2 & 3 & 4 \\ 1 & 2 & 3 \end{pmatrix}$ ，则PRODUCT(A,DIM=1)的值是[2,6,12]，SUM(A,DIM=2)的值是[24,6]。

✦ 数组查询函数：包括 SIZE、SHAPE、ALLOCATED、LBOUND 和 UBOUND 函数。

★ 求数组大小函数：SIZE(ARRAY[,DIM])

描述：求数组 ARRAY 沿着维 DIM 的长度或数组元素的总数目。

★ 求数组形状函数： **SHAPE(ARRAY)**。

描述：求数组或标量的形状。

✦ 数组的构造函数：包括 MERGE、PACK，UNPACK 和 SPREAD 函数。

描述：数组构造函数用于从已有数组的元素构造出新数组。

★ 合并数组函数： **MERGE(TSOURCE,FSOURCE,MASK)**

描述：在 MASK 的控制下，合并数组 TSOURCE 和 FSOURCE。

说明：数组 TSOURCE 可以是任一类型，数组 FSOURCE 必须与 TSOURCE 具有相同的类型和类型参数。MASK 必须是逻辑型数组。若 MASK 为真，则结果是 TSOURCE，若为假，则结果是 FSOURCE。

例：MERGE(1.,0.,R<0)的结果是：当 R=-2，取值为 1.0；当 R=2，取值为 0.0。

例：若  $A = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$ ， $B = \begin{pmatrix} 8 & 9 & 0 \\ 1 & 2 & 3 \end{pmatrix}$ ， $M = \begin{pmatrix} F & T & T \\ T & T & F \end{pmatrix}$ ，则  $\text{MERGE}(A,B,M) = \begin{pmatrix} 8 & 3 & 5 \\ 2 & 4 & 3 \end{pmatrix}$ 。

★ 压缩数组函数： **PACK(ARRAY,MASK[,VECTOR])**

描述：在 MASK 控制下，将数组 ARRAY 压缩成向量数组。

说明：ARRAY 可是任意类型的数组。MASK 必须是逻辑型数组并与 ARRAY 相容。VECTOR(可选)必须为向量数组，且与 ARRAY 具有相同的类型和类型参数。结果是秩为 1 的数组，其类型和类型参数与 ARRAY 相同；若 VECTOR 存在，结果大小等于 VECTOR 的大小，否则其大小是 MASK 中真元素的个数 t，若 MASK 为标量且为真值，这时结果的大小与 ARRAY 相同。结果的值按数组中元素排序，ARRAY 中的第 i 个元素对应于 MASK 的第 i 个真元素。若 VECTOR 存在，且大小  $n > t$ ，则结果中第 i 个元素值为 VECTOR(i)， $i = t + 1, \dots, n$ 。

例：若  $A = \begin{pmatrix} 0 & 0 & 0 \\ 9 & 0 & 0 \\ 0 & 0 & 7 \end{pmatrix}$ ，其非 0 元素可由 PACK 函数收集，PACK(A, MASK=A.NE.0)的结果为[9,7]，

PACK(A, A.NE.0, VECTOR=(/1,2,3,4,5,6/))的结果值为[9,7,3,4,5,6]。

★ 拷贝数组函数：SPREAD(SOURCE, DIM, NCOPIES)

描述：将数组 SOURCE 沿着 DIM 方向拷贝 NCOPIES 次后扩展成一新的数组。

说明：当 DIM=1 时，即沿着第一维下标变化的方向扩展，也即向下扩展。DIM=2 时，沿着第二维下标变化方向扩展即向右扩展。

例：若  $A = \begin{pmatrix} 1.1 & 1.2 & 2.3 & 1.4 \\ 2.1 & 2.2 & 2.3 & 2.4 \\ 3.1 & 3.2 & 3.3 & 3.4 \end{pmatrix}$ ，片段  $A(2,2:4)=[2.2,2.3,2.4]$ ，

则  $\text{SPREAD}(A(2,2:4),1,3)$  为  $\begin{pmatrix} 2.2 & 2.3 & 2.4 \\ 2.2 & 2.3 & 2.4 \\ 2.2 & 2.3 & 2.4 \end{pmatrix}$ ， $\text{SPREAD}(A(1:3,1),2,3)$  为  $\begin{pmatrix} 1.1 & 1.1 & 1.1 \\ 2.1 & 2.1 & 2.1 \\ 3.1 & 3.1 & 3.1 \end{pmatrix}$ 。

例： $\text{SPREAD}("B",1,4)$  为数组  $(/"B","B","B","B"/)$ 。

✦ 数组重构形函数： **RESHAPE(SOURCE,SHAPE[,PAD][,ORDER])**

✦ 数组运算函数：包括 TRANSPOSE、EOSHIFT 和 CSHIFT 三个函数。

★ 矩阵转置函数： **TRANSPOSE(MATRIX)**

描述：将秩为 2 的数组转置。

例：若A是数组 $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$ ，则TRANSPOSE(A)为 $\begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}$

★ 去端移动函数：EOSHIFT(ARRAY,SHIFT[,BOUNDARY][,DIM])

描述：对秩为 1 的数组表达式作去端移位，或沿着维 DIM 对秩为>1 的数组表达式在所有秩为 1 的完整数组片段上作去端移位。在一个数组片段的一端被移出的元素被丢弃，并在另一端移入相同的 BOUNDARY 的值。不同的片段可以有不同的 BOUNDARY 值，并可在不同的方向上移动不同的位数。SHIFT 为正值去端左移，为负值去端右移。

例：若 A 是数组[2,4,6,8,10]，则 EOSHIFT(A,SHIFT=3)去端左移 3 位的结果是[8,10,0,0,0]；而用 EOSHIFT(A,SHIFT=-2,BOUNDARY=36)去端右移 2 位的结果为[36,36,2,4,6]。

一个秩为 2 的数组的各行可以移动相同或不同数量的元素，边界元素也可以相同或不同。

例：若B是数组 $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$ ，

则EOSHIFT(B,SHIFT=1,BOUNDARY='\*',DIM=2)为 $\begin{pmatrix} 2 & 3 & * \\ 5 & 6 & * \\ 8 & 9 & * \end{pmatrix}$ , EOSHIFT(B,SHIFT=-1,DIM=1)为 $\begin{pmatrix} 0 & 0 & 0 \\ 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$

EOSHIFT(B,SHIFT=(/1,-1,0/),BOUNDARY=(/'\*', '?' , '/'/),DIM=2)为 $\begin{pmatrix} 2 & 3 & * \\ ? & 4 & 5 \\ 7 & 8 & 9 \end{pmatrix}$

★ 循环替换函数: CSHIFT(ARRAY,SHIFT[,DIM])

描述: 将秩为 1 的数组的所有元素或高维数组的指定维上的元素进行循环移动。在一端上移走的元素被插到另一端。SHIFT=正值时被移向左端, 负值移向右端。

例: 若 A 是数组[1,2,3,4,5,6], 则 CSHIFT(A,SHIFT=2)是[3,4,5,6,1,2]; 而用 CSHIFT(A,SHIFT=-2)是[5,6,1,2,3,4]。

✚ 数组定位函数: 有 MAXLOC 和 MINLOC 两个函数。

★ 最大值元素定位函数: MAXLOC(ARRAY[,DIM][,MASK])

描述：根据 MASK 的真值条件确定 ARRAY 的所有元素中或沿维 DIM 中第一个最大值元素出现的位置。

例：MAXLOC(/3,8,5,8/)的值为[2]。

例：若A是数组 $\begin{pmatrix} 4 & 0 & -3 & 2 \\ 3 & 1 & -2 & 6 \\ -1 & -4 & 5 & 5 \end{pmatrix}$ ，则MAXLOC(A,MASK=A.LT.5)为[1,1]，MAXLOC(A,DIM=1)为

[1,2,3,2]，MAXLOC(A,DIM=2)为[1,4,3]。

要注意的是，表达式中数组进行操作后，不再保留原来的下标形式，由系统将它们按线下界为1重新排列。这样，参与操作的两个数组原来的下标是否相同，可以不必考虑。但在引用某些与下标有关的函数时，要记住此时的函数值以整理过的新下标值出现。例如定义一个数组A及其初值如下：

```
INTEGER, DIMENSION(0:6),PARAMETER :: A=(/3,7,0,-2,3,6,-1/)
```

因值7是A中元素的最大值，元素的下标是A(1)，即数组A中最大值元素的下标为1。但内部函

数MAXLOC的返回值是 2 而不是 1，即返回的不是原来的下标值，而是以 1 为维下界整理过的下标值。

F90 数组运算内在函数表

函数名称	描述
ALL(mask[,dim])	判断全部数组值在指定维上是否都满足mask的条件
ANY(mask[,dim])	判断是否有数组值在指定维上满足mask的条件
COUNT(mask[,dim])	统计在指定维上满足mask的条件的元



	素个数
CSHIFT(array,shift[,dim])	进行指定维上的循环替换
DOT_PRODUCT(vector_a,vector_b)	进行两个向量的点乘
EOSHIFT(array,shift[,boundary][,dim])	在指定维上替换掉数组末端，复制边界值到数组末尾
LBOUND(array[,dim])	返回指定维上的下界
MATMUL(matrix_a,matrix_b)	进行两个矩阵(二维数组)的乘积

MAXLOC(array[,dim][,mask])	返回数组的全部元素或指定维元素当满足mask条件的最大值的位罝
MAXVAL(array[,dim][,mask])	返回在指定维上满足mask条件的最大值
MERGE(tsource,fsource,mask)	按mask条件组合两个数组
MINLOC(array[,dim][,mask])	返回数组的全部元素或指定维元素当满足mask条件的最

	小值的位置
MINVAL(array[,dim][,mask])	返回在指定维上满足mask条件的最小值
PACK(array,mask[,vector])	使用mask条件把一个数组压缩至vector大小的向量
PRODUCT(array[,dim][,mask])	返回在指定维上满足mask条件的元素的乘积
RESHAPE(source,shape[,pad][,order])	使用顺序order和补充pad数组元素来

	改变数组形状
SHAPE(source)	返回数组的形状
SIZE(array[,dim])	返回数组在指定维上的长度
SPREAD(source,dim,ncopies)	通过增加一维来复制数组
SUM(array[,dim][,mask])	返回在指定维上满足mask条件的元素的和
TRANSPOSE(matrix)	转置二维数组
UBOUND(array[,dim])	返回指定维上的上

	界
UNPACK(vector,mask,field)	把向量在mask条件下填充field的元素 解压至数组

### e) 数组的输入输出

数组的输入输出可以用隐 DO 循环指定每维循环变量，也可单给出数组名、数组元素或数组片段。

#### ✦ 一维数组:

一维数组的输入输出比较简单，它在机内存贮时作线性排列，依次读入（输出）的数逐一按下标值递增赋给（打印）各元素：

例：INTEGER :: a(10)

READ \*, (a(i), i=1,10)    ← 可以 1 行输入 10 个值，也可输入 1 个值(隐 DO 循环)

READ \*, a                    ← 同上                    (数组名)

DO i=1,10

    READ \*,a(i)            ← 1 行只能输入 1 个值，输入 2 个以上被忽略

END DO

READ \*, a(3),a(4:6)       ← 输入数组元素和数组片段： a(3),a(4),a(5),a(6)

PRINT '(10I5)',(a(i),i=1,10) ← 1 行打印 10 个值 (隐 DO 循环)

PRINT '(10I5)',a           ← 同上                    (数组名)

PRINT '(5I5)',a(1:5)       ← 1 行打印 5 个值 (数组片段)

DO i=1,10

    PRINT '(I5)',a(i)       ← 1 行打印 1 个值，共打 10 行

END DO

## ✦ 二维数组:

二维数组的输入输出读写语句使用与一维情况时一样，数组元素输入输出的顺序是按前面提到的数组的存储顺序来进行的。由于二维数组的顺序是首先按列存放，因此对于输入 $3 \times 3$ 数组的 A，执行语句 `READ *,A` 时，则读入数据的顺序是  $A(1,1) \rightarrow A(2,1) \rightarrow A(3,1) \Rightarrow A(1,2) \rightarrow A(2,2) \rightarrow A(3,2) \Rightarrow A(1,3) \rightarrow A(2,3) \rightarrow A(3,3)$ 。按列存贮方式与我们数学上按行处理的习惯不一致，在给数组各元素赋值时，先输入第一列元素的值，再输入第二、第三列的值这样机内收到的矩阵才是正确的。

如一定要按行方式输入矩阵值，可以通过交换隐 `DO` 循环内外层循环变量的方法来实现。如 `READ *,A` 与 `READ *,((A(I,J),I=1,3),J=1,3)` 是等价的，交换内外层循环变量后：`READ *,((A(J,I),I=1,3),J=1,3)` 或 `READ *,((A(I,J),J=1,3),I=1,3)`，这时输入顺序是按行的： $A(1,1) \rightarrow A(1,2) \rightarrow A(1,3) \Rightarrow A(2,1) \rightarrow A(2,2) \rightarrow A(2,3) \Rightarrow A(3,1) \rightarrow A(3,2) \rightarrow A(3,3)$ 。

### 5.2.3 数组的动态分配

#### a) 可分配数组

数组可以是静态的也可以是动态的。如果数组是静态的，则在编译时就被分配了固定的储存空间，并且直到程序退出时才被释放。程序运行时静态数组的大小不能改变。静态数组的缺陷是，即使数组已经使用完毕，它仍占据着内存空间，浪费了系统资源。在给定的计算机内存资源情况下，耗费了其他数组可以利用的内存，并且超过资源的数组将导致程序执行错误。因此，**F90** 增加了动态的数组功能，动态数组的储存在程序运行当中是可以分配、改变和释放的。

动态数组只有两种：可分配数组和自动数组。自动数组和可分配数组很类似，区别在于当程序开始或结束时，自动数组会自动分配和释放内存。当用户分配动态存储空间时，数组的大小是在运行时而不是在编译时确定的。动态分配可以用于标量和任何类型的数组。当用户给数组指定了可分配属性时并没有立即分配内存，而是直到使用 **ALLOCATE** 语句后才分配。随后还可以用 **DEALLOCATE** 语句释放内存空间，这时数组可以以其它形状或目的来使用。

应该注意的是，**WinNT/9x** 上运行的 **Visual Fortran** 动态内存分配受一些因素的限制，包括交换文件的大小和其它同时运行的应用程序所需的内存大小。如果动态分配的内存太大或试图使用其它应用程序的保护内存会产生一般内存保护错误。碰到这类问题可以通过控制面板来改变虚拟内存的



大小或交换文件的大小，还有一些编程技术可以降低内存需要。

## b) ALLOCATE 语句

ALLOCATE 语句动态创建可分配数组，使内存和对象相联系。分配的对象可以被命名为任何有 ALLOCATABLE 属性的变量。它的一般形式为：

**ALLOCATE(数组名[维界符],数组名[(维界符[,维界符...])] ] ...[,STAT=状态值])。**

例：REAL A(:),B(:, :, :)

ALLOCATABLE A,B

ALLOCATE(A(-2:40),B(3,3,3))

当数组被分配时，内存分配给指定大小的数组。ALLOCATE 语句中的秩必须和可分配数组的秩相同。在分配的同时，ALLOCATE 语句中的上下界决定了数组的大小和形状。边界的值可以是正数、负数或零，缺省的下界为 1。如果维上界比下界小，则该维的长度为零，并且数组的大小为零。大小为

零的数组不能被赋值。

当前被分配的数组不能被再分配，否则会引起运行错误。错误状态可以由 `ALLOCATE` 语句中的 `STAT` 值获得。如果指定 `STAT` 选项，语句的成功执行时将返回 0，否则返回正值。若未指定 `STAT` 选项且出现错误时，程序将中止执行。

例： `INTEGER, ALLOCATABLE :: A(:),B(:)`

`INTEGER ERR_MESSAGE`

`ALLOCATE(A(10:25),B(SIZE(A)),STAT=ERR_MESSAGE)`

`IF(ERR_MESSAGE.NE.0) PRINT *,'ALLOCATION ERROR'`

可以用内在函数 `ALLOCATED` 来判断一个数组是否已被分配。它的形式为：`ALLOCATED(数组名)`。返回值是逻辑标量，已被分配时为真，现在还未被分配时为假，当数组的分配状态未定义时它也是未定义的。

例： `REAL, ALLOCATABLE :: A(:)`

...

```
IF(.NOT.ALLOCATED(A)) ALLOCATE(A(5))
```

### c) DEALLOCATE 语句

DEALLOCATE 语句用来释放已分配数组的内存。它的一般形式为：**DEALLOCATE(数组名[,数组名]...[,STAT=状态值])**。

例： INTEGER, ALLOCATABLE :: A(:),B(:)

```
INTEGER ERR_MESSAGE
```

```
ALLOCATE(A(10:25),B(SIZE(A)))
```

```
DEALLOCATE(A,B,STAT=ERR_MESSAGE)
```

```
IF(ERR_MESSAGE.NE.0) PRINT *,'DEALLOCATION ERROR'
```

例： INTEGER,DIMENSION(:),ALLOCATABLE :: freq

```
READ *,limit
```

```
ALLOCATE(freq(1:limit))
```

```
DEALLOCATE(freq)
```

只有被 **ALLOCATE** 语句分配的内存空间才可以被 **DEALLOCATE** 语句释放，否则产生运行错误。可以使用 **ALLOCATED** 语句判断数组是否被分配，错误状态可以由 **ALLOCATE** 语句中的 **STAT** 值获得。

当过程的执行被 **RETURN** 或 **END** 语句中止时，除非可分配数组是有 **SAVE** 属性的，否则它的分配状态变成未定义的。但是，**RETURN** 和 **END** 语句并不释放数组分配的内存，所以应该在退出子程序前主动释放数组分配的内存。

可分配数组的联合状态可以是已分配的（该数组被 **ALLOCATE** 语句分配，可以被引用、定义或释放）或是目前未联合（该数组从未分配或上一个操作是释放，数组不能被引用或定义）。

当可分配数组赋值时就被定义。

例： **INTEGER, ALLOCATABLE :: A(:)**

ALLOCATE(A(100))      ! A 被分配但未定义，A 的分配状态是已分配

A(1:100)=1      ! A 被定义

DEALLOCATE(A)      ! A 被释放，A 的分配状态是未分配

### 【作业】

[5.1] 用类型说明语句定义：

- (1) 一个整型数组，10 个元素，名 I；
- (2) 一个逻辑型数组，2 维，第一维维界是 0:7，第 2 维维界是-7:0，名 L；
- (3) 一个字符型数组，1 维，下标从 1 变到 100，名 C；

(4) 把矩阵 $\begin{pmatrix} 1.1 & 1.2 & 1.3 \\ 1.4 & 1.5 & 1.6 \end{pmatrix}$ 定义成数组，名 R；

[5.2] 设数组 $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$ ， $B = \begin{pmatrix} -1.1 & -2.1 & -3.1 \\ -4.1 & -5.1 & -6.1 \end{pmatrix}$ ， $C = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$ ， $D = (1 \ 2 \ 3)$ ，E 数组描述为(1:2, 1:3)，F 数组

描述为(1:3)，问下列数组表达式是否合法，不合法的说明理由，合法的写出计算结果：

(1)  $E = A + B$

(2)  $E = \text{ABS}(B) + 2$

(3)  $E = B + C$

(4)  $C = A + C$

(5)  $F = D * D$

(6)  $F = A(1:2, 1) + B(1:3, 1)$

(7)  $F = A(1, 1:3) + B(1, 1:3)$

[5.3] 用 WHERE 构造，使上题中 A 元素 $\leq 2$ 时，置 B 相应元素为自身的绝对值，否则使 B 中元素加 1，写出最后 B 的内容。

[5.4] 读入一个二维数组 A，形状为(1:5, 1:3)，任意输入具体数据。

(1) 用 DO 循环求全部数组元素和，再用数组内部专用求和函数求和，比较两者结果。

(2) 用 DO 循环求数组元素连乘积，再用数组内部求连乘积函数求整个数组元素之积，比较两者结果。

[5.5] 用动态数组存放任意读入的 N 个储款数，统计这次存款总数，而后释放数组，再把它存贮任意读入的 M 个取款数，统计共被取出的款额，求收支相抵的余额。

## 第六章：过程和模块

### 程序单元结构

#### 6.1.1 概述

FORTRAN 程序应由一个主程序单元和若干个过程程序单元组成。编程前应先作结构化分析，把

问题分解为若干个子功能，每个子功能编成一个独立的程序单元，称为过程程序单元。再编一个主程序单元，控制整个解题过程。在主程序单元中用简练的形式调用每个过程，每次调用就是执行该过程单元，完成该子功能，从而最终完成解题任务。本章介绍 F90 中的程序单元、过程、范围、模块等概念。

F90 中，共有四种程序单元：

- 主程序
- 过程或辅程序
- 块数据单元
- 模块

其中模块是 F90 新增加的一种程序单元。通过使用模块，可以方便地共享数据和过程。各种程序单元中，除主程序外，各程序单元可以被其他程序单元甚至自身调用。一个程序单元不需要包含可执行语句。包含内部子程序或函数的程序单元称为宿主程序。下表是对这四种程序单元类型的定义：



程序 单元	定义
主 程 序	主程序是程序开始执行的标志，其第一条语句不能是 SUBROUTINE，FUNCTION，MODULE 和 BLOCK DATA。主程序可以用 PROGRAM 语句作为第一条语句，但不是必需的
过程	子程序或函数
块 数 据 单 元	在命名的公共块中提供变量初始值的程序单元
模块	包含数据对象定义、类型定义、函数或子程序接口和其它程序可访问的函数或子程序

程序单元之间的关系有：

- **联合**：这种机制允许不同的程序单元共享变量，从而不用重新定义变量就可以以不同的名字处理同一数据。
- **范围**：它描述的是一个名称(或者是全局的或者是局部的)作用的范围。

### 6.1.2 主程序

程序的执行始终是开始于主程序的第一条可执行语句，所以每个完整的 Fortran 程序必须有且仅有一个主程序。其形式为：

[PROGRAM [程序名]]

[说明部分]

[可执行部分]

[CONTAINS

内部过程]

END [PROGRAM[程序名]]

可以看到，主程序单元中允许包含内部过程。主程序名和外部过程名和公共块名都被认为是全局名称。全局名称在一个程序中必须是唯一的。

例：PROGRAM MAIN

REAL X(10),A(10,10),B(10)   !说明部分

...

CALL GAUSS(A,B,X)           !执行部分，调用内部过程 GAUSS

CONTAINS

SUBROUTINE GAUSS(R,S,T)   !内部过程的开始

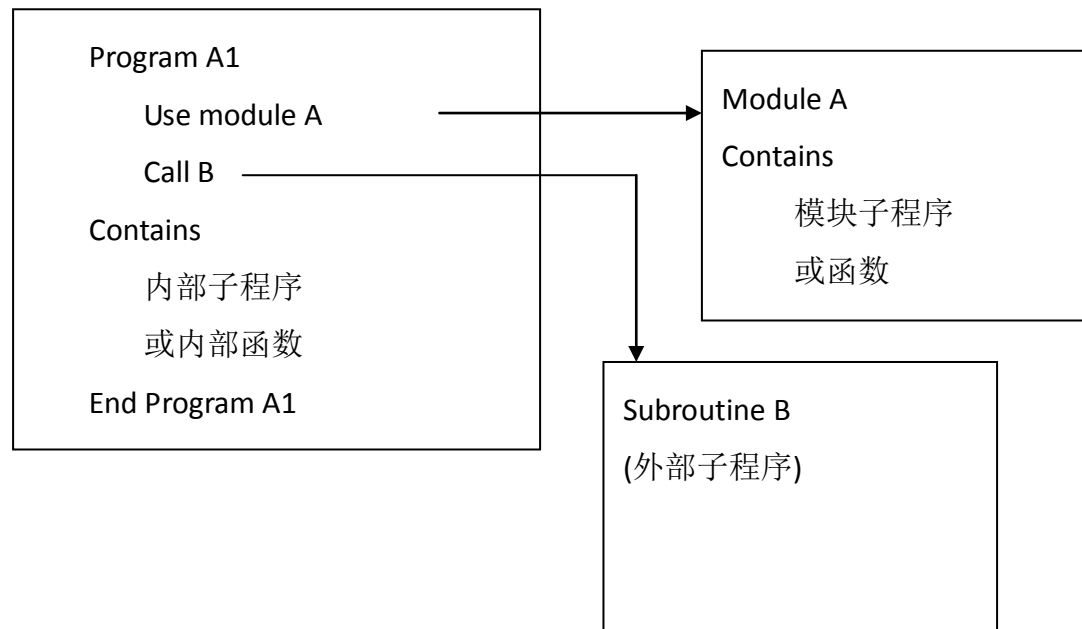
...

END SUBROUTINE GAUSS

!内部过程的结束

EHD PROGRAM MAIN

下图显示的是一个 F90 程序的标准结构。每个框都可以是一个独立的源文件：



### 6.1.3 过程

## gg) 外部过程

如果过程是一个独立于主程序单元的程序单元，它就是外部过程。外部过程也可以通过非 Fortran 语言(通常是汇编语言)来定义。外部过程的形式有两种：

外部函数 : FUNCTION 语句

[说明部分]

[可执行部分]

[CONTAINS

内部过程]

END [FUNCTION 函数名]

外部子程序: SUBROUTINE 语句

[说明部分]

```
[可执行部分]

[CONTAINS

内部过程]

END [SUBROUTINE 子程序名]
```

#### hh) 内部过程

内部过程是包含于外部过程、模块或主程序单元之中的程序单元。在宿主程序单元中，以 **CONTAINS** 语句把内部过程与其他部分分开。内部过程也具有两种形式：

**内部函数**：FUNCTION 语句

```
[说明部分]

[可执行部分]

END [FUNCTION 函数名]
```

内部子程序: SUBROUTINE 语句

[说明部分]

[可执行部分]

END [SUBROUTINE 子程序名]

注意内部过程中不能再包含内部过程，即 F90 不是过程嵌套式语言。

## ii) 内在过程

内在过程是 Fortran 含于编译器程序库中的过程，它不需任何其它声明或说明即可使用。F90 中定义了 113 个内在过程，它们为科学计算提供了极大方便，内在过程除了第一章中介绍的部分内在函数和第五章中介绍的部分数组函数外，还有其他一些内在函数及内在子程序，其具体使用方法请查看 Visual Fortran 的联机帮助。内在过程共分为四类：

- 1、 查询函数：它们的返回值是根据变元的性质而非变元的取值。

- 2、基本过程：由标量变元指明的基本过程，但可以用标量或数组实元进行调用。有许多函数是这种基本函数，并有一种基本内在子程序(MVBITS)。如果变元是标量，则结果也是标量。如果变元是数组，则对数组的每个元素施加过程，其结果是与变元形状相同的数组。例如， $a$  是数组的话，则  $b=\sin(a)$  也是数组。
- 3、变换函数：其变元是数组，但过程不施加于每个元素，而是把变元变换成另一数组。
- 4、非基本过程：只能以标量实元进行调用，除了 MVBITS 以外的所有内在子程序都是非基本过程。

如果要用内在过程名作为其它过程的实元，需要用 INTRINSIC 语句加以声明。

#### 6.1.4 块数据

块数据单元是一种为有名公用块中的变量定义初始值的一种程序单元，它只包含数据声明和初始值，不包含可执行语句。其一般形式为：

**BLOCK DATA**[块数据名]



[说明部分]

END [BLOCK DATA[块数据名]]

变量一般由 **DATA** 语句来初始化。公共块中命名的变量只能在块数据单元或某个过程中初始化一次，或只能由所有的过程完全一致地初始化。更好的编程法是用模块而不是块数据单元来声明和初始化变量。

在块数据单元的说明部分，不能包含 **ALLOCATABLE**，**AUTOMATIC**，**EXTERNAL**，**INTENT**，**OPTIONAL**，**PRIVATE** 或 **PUBLIC** 等属性说明符。

例：BLOCK DATA BLKDAT

INTEGER S,X

LOGICAL T,W

DOUBLE PRECISION U

DIMENSION R(3)

```
COMMON/AREA1/R,S,U,T/AREA2/W,X,Y
```

```
DATA R/1.0,2*2.0/,T/.FALSE./,U/0.214537D-7/,W/.TRUE./,Y/3.5/
```

```
END BLOCK DATA BLKDAT
```

## 过程

### 6.2.1 什么是过程

Fortran 编译系统提供内在函数，可以在任意程序单元中引用。但是在许多时候仅仅使用系统的内在函数并不能满足程序设计的需要，因此就需要自行编制相应的函数或子程序来扩充程序的处理能力。

过程是在程序的执行中可被直接调用的、封装在一起的、进行计算或处理的语句序列。它是任何一种过程型程序设计语言的重要组成部分，对 Fortran 语言也不例外。F90 中，一个过程的定义就是指它是一个函数或是一个子程序。

过程的引用就是调用一个过程。建立过程的目的就是建立可多次重复执行的程序段，以便多次调用它们。通常过程是带有参数的，在 Fortran 中把参数称为变元（实元或哑元），过程定义中的变元是哑元，过程引用中的变元是实元。在调用过程时，要用实元代替哑元，这就是哑实结合。

## jj) 分类

过程包括下面几种类型：

- ✦ **外部过程**：它是在某个外部程序单元中定义的独立过程，或是用非 Fortran 语言编写的过程。
- ✦ **内部过程**：在程序单元内部定义而且只能被该程序单元调用。
- ✦ **内在过程**：由编译系统内部定义，不用任何附加声明或说明就可以可直接引用。
- ✦ **模块过程**：它在模块中定义，可以被所有使用该模块的程序调用。包含过程的模块称为宿主。
- ✦ **哑过程**：如果一个哑元被指明为过程或作为过程名出现在过程引用中，那么该哑元代表的过程为一个哑过程。
- ✦ **语句函数**：它是由单个语句定义的函数，其形式为：**函数名([哑元名表])=标量表达式**。F90 不

推荐使用，因为它不符合过程的一般规则。

## kk) 特性

过程的特性包括，将过程分为函数和子程序的分类特性和它的哑元的特性，对于函数还包括有结果的特性。

一个哑元可以是一个虚拟数据对象、哑过程或作为选择返回指示符的星号。当一个哑元不是星号时，它可有 **OPTIONAL** 属性，表示对该过程引用时不需要有实元与该哑元结合。一个哑元数据对象的主要特性包括它的类型、种别值、形状、输入输出意向(**INTENT**)、是否可选(**OPTIONAL**)、是否一个指针(**POINTER**)或目标(**TARGET**)。哑过程的特性包括其接口是否显式给出、作为过程的特性(如果其接口显式给出)以及它是否可选。函数结果的主要特性包括它的类型、类别值、秩以及是否指针等。

## 6.2.2 外部过程

外部过程是程序员编写的函数或子程序，它独立于主程序外部。外部过程可以单独以源文件存储和编译，也可以包括在主程序源代码的 **END** 语句后。外部过程本身也可以包含有内部函数或内部子程序。

### a) 子程序

子程序以 **SUBROUTINE** 语句开始，**END** 语句结束的过程。其一般形式为：

[前缀]**SUBROUTINE** 子程序名[(哑元列表)]

...

**END** [**SUBROUTINE**[子程序名]]

其中，哑元可以是变量名、数组名、过程名、指针名等均可作为哑元。哑元表内列出本程序体内要用到的全部哑元，列在括号内，彼此用逗号分隔（哑元都要在说明语句中说明类型）。前缀是 F90 中新增的，它可以是：类型说明 [关键词]或关键词 [类型说明]。关键词是下面之一：**RECURSIVE**(F90)，**PURE**(F95)，**ELEMENTAL**(F95)。RECURSIVE 表示过程时可以直接或间接地调用自身，即递归调用，

其过程是递归过程。

子程序名只是用来作标识，不代表任何值。**哑元列表**是子程序与调用单元之间进行数据传送的主要渠道，当有一个以上哑元时，它们之间用逗号隔开，如果没有哑元，则一对括号可以省略。从 **SUBROUTINE** 语句后面一直到 **END** 语句则是子程序体。它的说明部分应包括对哑元和子程序中所用变量、数组等的说明，它的执行语句部分完成运算和操作功能。其中的 **END** 语句或 **RETURN** 语句使执行流程返回到调用单元。

哑元表中的哑元个数，理论上不受限制，但从软件工程观点看，不宜过多，一般不应超过六、七个。如果太多，意味着该子程序的算法较复杂，应该把该过程再分解为几个子功能分别编写成几个子程序。

调用子程序时必须用一条独立的 **CALL** 语句，其形式为：**CALL 子程序名[(实元列表)]**

例：读入 3 个整数，按大小顺序重排。

```
INTEGER :: i,j,k
```

```
READ *, i,j,k
```

```
IF(i<j) CALL swap(i,j)
```

```
IF(j<k) CALL swap(j,k)
```

```
IF(i<j) CALL swap(i,j)
```

```
PRINT *,i,j,k
```

```
END
```

```
!
```

```
SUBROUTINE swap(p,q)
```

```
    INTEGER :: p,q,r
```

```
    r=p;p=q;q=r
```

```
RETURN
```

```
END
```

调用子程序时的实元是和哑元相同类型的变量、数组元素、数组和常数。当用 **CALL** 语句予以调用时，哑元和实元才按列表顺序一一对应，取得同一数值。但使用常数时要避免下例中的常数替换式，否则将导致不可预料的错误，应尽可能用变量作实元。

例：SUBROUTINE bai(x)

```
REAL :: x
```

```
x = 2. * x
```

```
END
```

```
s=1.
```

```
CALL bai(1.)
```

```
PRINT*,s
```

```
END
```



## b) 函数

自定义函数和子程序比较相近，共同的特征是作为过程的集合，子程序的很多规则也可应用。两者的不同之处在于函数返回一个值，且通常不改变哑元的值，因此，它代表数学上的一个函数。而子程序通常完成一项更为复杂的任务，通过变元或其他手段返回几个结果。函数的一般形式为：

[前缀] FUNCTION 函数名([哑元列表])[RESULT(结果名)]

...

END [FUNCTION 函数名]

如果没有哑元，则哑元表是一个空括号。**RESULT** 是关键字，要照写，后面括号内的变量名就是函数计算的结果值。函数结果变量名有值，必须在说明语句中说明类型，在程序执行部分中至少赋值一次。在引用时，它的值就是函数值。函数结果名不可列入哑元表。如果没有结果名，则函数名就是结果名。**F90** 中之所以增添了结果名功能，就是为了区别函数字面上的名称(函数名)和实际运算结果变量的名称。

函数调用与调用内在函数形式一样，在主调程序任何处，作为表达式的一项写下：**函数名(实元表)**就完成调用。如果函数无哑元，则调用形式是在表达式中写上函数名，后跟空括号：函数名()。系统在该位置上返回以该实元为自变量的函数结果值，参加表达式的运算。

例如，下面都是合法的 FUNCTION 语句：

```
FUNCTION FUN1
```

```
FUNCTION FUN2()
```

```
INTEGER FUNCTION FUN3(A,B)
```

```
RECURSIVE FUNCTION FUN4(X,Y,Z)
```

```
REAL RECURSIVE FUNCTION FUN5(M,N) RESULT(R_FUN5)
```

例：将一个 4 字节的整数用 16 进制表示出来。 [\[e 622 01.f90\]](#)

```
FUNCTION hex(n)
```

```
    CHARACTER(LEN=8) :: hex
```

```
CHARACTER(LEN=1) :: h(0:15)=(/'0','1','2','3','4','5','6',&  
                                '7','8','9','A','B','C','D','E','F'/)
```

```
INTEGER :: n,j,nn
```

```
hex = ''
```

```
DO j=8,1,-1
```

```
    nn=n/16
```

```
    hex(j:j)=h(n-16*nn)
```

```
    IF(nn==0) EXIT
```

```
    n=nn
```

！ 将 n 指定为 INTENT(IN)将导致错误

```
END DO
```

```
END FUNCTION
```

```
PROGRAM main
```

```
    CHARACTER(LEN=8)::hex    !函数名的类型在调用单元也须加以声明
```

```
    INTEGER::i
```

```
    PRINT *, 'Input a positive integer, or negative one to stop:'
```

```
    DO
```

```
        READ *,i; IF(i<0) EXIT
```

```
        PRINT *,hex(i)
```

```
    END DO
```

```
END
```

在本例中，函数值是赋给函数名的，如果要将值赋给非函数名的另一结果名，则

```
FUNCTION hex(n) RESULT(hx)
```

```
    CHARACTER(LEN=8)::hx
```

...

hx(j:j)=

但在此例中就没有必要用函数了，可以用子程序，如：

```
SUBROUTINE hex(n,hx)
```

```
CHARACTER(LEN=8),INTENT(OUT)::hx
```

例：分形图形的计算。分形(fractal)是 Mandelbrot 将自然界的复杂图形(如海岸线，树叶形，雪花结晶型)进行数学理想化后提出的一种概念，其核心是图形的任意细小部分都与图形的整体具有自相似性，这种图形的维数不是整数，而是有分数维的。一个典型例子是 Koch 曲线，它有雪花形，可通过对一段直线反复进行某一简单操作而得到，把这个过程用数学语言描述，即为在复空间定义的一种简单迭代过程，它是一个图形的缩小映射，从而产生自相似曲线。一种简单的缩小映射是：

$$\begin{aligned}f_0(z) &= az + b\bar{z} \\ f_1(z) &= c(z-1) + d(\bar{z}-1) + 1\end{aligned}$$

迭代过程是用  $z_0 = 0$  作为初始值代入得  $f_0(z_0), f_1(z_0)$  然后反复将得到的值作为初始值代入得一序列复数

值  $f_0(f_0(z_0)), f_0(f_1(z_0)), f_1(f_0(z_0)), f_1(f_1(z_0)), \dots$ 。然后将此序列复数在复空间中绘出即得到相似曲线。程序中要计算迭代到  $n$  阶所需的迭代函数，其个数为  $2^n - 1$ ：用 **0** 记  $f_0(z_0)$ ，**01** 记  $f_0(f_1(z_0))$

1 阶：0,1

2 阶：00,01,10,11

3 阶：000,001,010,011,100,101,110,111[\[e 622 02.f\]](#)

### c) EXTERNAL 属性和哑过程

指定 EXTERNAL 语句或属性说明实元实际上是外部过程。其一般形式为：

类型定义语句：类型,EXTERNAL :: 外部函数名[,外部函数名]...

或 EXTERNAL 语句：EXTERNAL [外部函数名][,子程序名][,块数据名]...

哑元也可以是一个过程，这时作为哑元的过程称为哑过程。只有在多层调用（至少两层调用）时才能用哑过程。如果要用外部过程名作实元，则过程名必须出现在一个 EXTERNAL 语句中，或被该

作用范围中的一个类型语句指明具有 **EXTERNAL** 属性，或在该作用范围内被一个接口块 **INTERFACE** 声明为一个过程。这就是说，在主调程序中，除了把一个实际存在的过程名作实元与哑过程名结合外，还需对该实过程名作特别说明，以便让编译系统明白，该实元不是一般的简单变量，而是一个函数（或子程序），或者用接口块来通知编译系统。

例： **REAL FUNCTION CALCULATE(A,B,FUNC)**

**EXTERNAL FUNC**

**REAL,INTENT(IN) :: A,B**

**REAL F,X**

**...**

**F=FUNC(X) !调用自定义的外部函数**

**...**

**END FUNCTION CALCULATE**

程序中的 `EXTERNAL` 语句说明函数 `FUNC` 是外部函数，也可用类型声明语句说明函数 `FUNC` 具有 `EXTERNAL` 属性，把程序改写为：

例： `REAL FUNCTION CALCULATE(A,B,FUNC)`

`REAL EXTERNAL :: FUNC`

...

也可以用显式的接口块来代替 `EXTERNAL` 语句，把上面的程序改写为：

例： `REAL FUNCTION CALCULATE(A,B,FUNC)`

`INTERFACE`

`REAL FUNCTION FUNC(X)`

`REAL,INTENT(IN) :: A,B`

`END FUNCTION FUNC`

`END INTERFACE`



```
REAL F,X
```

```
...
```

```
F=FUNC(X) !调用自定义的外部函数
```

```
...
```

```
END FUNCTION CALCULATE
```

例：采用梯形公式近似求[a,b]区间上函数 f(x)的定积分

```
FUNCTION INTEGRAL(F,A,B,N) RESULT(INT_RES)
```

```
IMPLICIT NONE
```

```
REAL :: INT_RES
```

```
INTERFACE
```

```
    FUNCTION F(X)
```

```
        REAL :: F,X
```

```
        END FUNCTION

    END INTERFACE

    REAL,INTENT(IN) :: A,B

    INTEGER,INTENT(IN) :: N

    REAL :: H,SUM

    INTEGER :: I

    H=(B-A)/N

    SUM=(F(A)+F(B))/2

    DO I=1,N-1

        SUM=SUM+F(A+I*H)

    END DO

    INT_RES=H*SUM
```

## END FUNCTION INTEGRAL

在上面的函数 **INTEGRAL** 中，哑元 **F** 被一个接口块说明为一个过程，因此它是一个哑过程。在调用该过程时，必须对该哑元提供一个另外定义的函数过程作为实元与之结合，同时也需要写上接口块。

当用 **EXTERNAL** 语句说明 **SIN**, **COS** 等名字时，它们被解释为自定义函数，而非内在函数(三角函数)。

### d) ENTRY 语句

**ENTRY** 语句允许在特定的可执行语句中插入外部过程或模块过程。内部过程不能有 **ENTRY** 语句。使用语句的目的是，把多个有条件进入过程的相关函数或子程序组织起来。它的形式和用法与函数和子程序类似：语句包括一个入口点的名称，一个可选哑元列表，一个可选的 **RESULT** 结果名(在函数的情况下)。一个过程可以有一个或多个 **ENTRY** 语句。**F90** 的块结构如 **CASE** 语句所实现的控制流方式比通过 **ENTRY** 方式为好，故 **F90** 中并不提倡用 **ENTRY** 语句。

函数过程中的 **ENTRY** 语句：定义了另一个函数，函数名由 **ENTRY** 语句指定，结果名即函数名或由

RESULT 指定。

例：计算双曲三角函数 sinh，cosh 和 tanh 的函数过程。

```
REAL FUNCTION TANH(X)
```

```
    TSINH(X)=EXP(X)-EXP(-X)
```

```
    TCOSH(X)=EXP(X)+EXP(-X)
```

```
    TANH=TSINH(X)/TCOSH(X)
```

```
    RETURN
```

```
ENTRY SINH(X)
```

```
    SINH=TSINH(X)/2.
```

```
    RETURN
```

```
ENTRY COSH(X)

COSH=TCOSH(X)/2.

RETURN

END
```

函数过程中的 ENTRY 语句，定义了另外两个函数，函数名由 ENTRY 语句指定。

例：PROGRAM TEST

```
...

CALL WAHAHA(A,B,C,D)

...

END

SUBROUTINE HAHA(X,Y,Z)
```

```
...  
ENTRY WAHAHA(Q,R,S,T)  
...  
END SUBROUTINE
```

### 6.2.3 变元的性质

#### a) INTENT 属性

哑实结合是在两个程序单元间传递数值的主要手段，主程序中实元 2.0 与过程中哑元 X 结合，就使 X 有值 2.0，也即把主程序中 2.0 的值传递给子程序中的 X，该值可供子程序运算。反之，如果子程序中的变量 Y 在子程序执行完后有值 3.0，它与实元 R 结合后则使调用程序单元中的实元变量 R 得值 3.0。

在 F77 中，不能确切地说明哑元的目的。它们到底是用于把数据传入到过程中的，还是用于把数

据传出到调用它的程序单元中的，或是两者兼而有之的，这个概念是含糊的。在 F90 中，为了避免当过程内部变量值变化后返回到引用的程序单元时可能造成的混淆情况，在过程的变量类型的定义中，可以对哑元指定意图说明的 **INTENT** 属性。哑元按数据传输特性可分为输入输出两用、仅用于输入和仅用于输出。其一般形式为：

在类型定义语句中： 类型, **INTENT**(意图说明符) :: 哑元名表

或用 **INTENT** 语句： **INTENT**(意图说明符) :: 哑元名表

意图说明符为以下字符串：

**IN** 指明哑元仅用于向过程提供数据，过程的执行期间哑元不能被重定义或成为未定义的，相联合的实元可以是常数、变量、数组以及它们的算术表达式。

**OUT** 指明哑元用于把过程中的数据传回调用过程的程序，与之相结合的实元只允许是变量，不得为常数或算术表达式。

**INOUT** 指明哑元既可以用于向过程提供数据，也可用于向调用程序返回数据，与之相

结合的实元只允许是变量。

**INTENT** 属性不能在主程序说明语句中出现，只能在过程的哑元说明语句中使用。它是可选的，可省略。但现代特性的编程中应提倡使用 **INTENT** 属性，因为这样能增加可读性和可维护性，还能防止编程中的一些错误。因为一旦哑实结合，哑元和实元始终是同一个值，如果过程中给有属性 **INTENT(IN)**的哑元重新赋值，也将改变调用程序单元中实元的值，而这是不应该的。这样，如在程序执行部分中误把有 **INTENT(IN)**属性的哑元赋值时，操作系统就会提示。

例：给出 10 个步长的分布值，打印分布图形。 [\[e 623 01.f90\]](#)

## b) **SAVE** 属性

在过程中变量的定义和取值当过程被调用结束后有可能变为不确定的，因此当过程再次被调用时，变量的取值在不同编译器下可能取值不同。为了避免这种情况的出现，在 **F77** 中可用 **SAVE** 语句，在 **F90** 中对变量增加了 **SAVE** 属性，其形式为：

在类型定义语句中： 类型, **SAVE**, [其它属性] :: 变量名表



或用 SAVE 语句 : **SAVE [变量名表]**

在过程中设定初始值时要注意，类型定义中的初始值赋值法隐含了 SAVE 属性。[\[e 623 02.f90\]](#)

### c) 关键字变元

哑实结合必须遵循**三个一致**的原则，否则运行出错：哑元与实元位置一致；哑元与实元个数一致；哑元与实元类型一致。这就要求记住每个哑元的名及其位置，阅读实元表时对其中每个表达式要追溯到它原来的哑元是什么，非常不便，为此 F90 可以通过如下方法放宽这三个一致的原则：用关键字变元放宽位置一致；用可选择变元放宽个数一致；用类属过程放宽类型一致。

关键字变元是调用过程时变元的一种现代形式，它的写法是：**哑元名 = 实元表达式**。调用时，实元表中不仅要写出实元表达式，还要写出它对应的哑元变量名，这个哑元变量称为关键字，并用‘=’号与实元连接。使用关键字后，就不必记住哑元原来的次序，填写的实元次序可以任意。例如，对于子程序语句（对函数一样可用）：

```
SUBROUTINE FACTORIAL(N,F_VALUE)
```

主调程序中，调用语句使用关键字变元时形式如下：

```
CALL FACTORIAL(N=M,F_VALUE=F_M)
```

```
CALL FACTORIAL(F_VALUE=F_M,N=M)
```

F90 也允许在调用语句中，前面部分实元不用关键字变元，只从某一个变元开始用关键字变元。此时，前面未使用关键字变元仍要保持与原来哑元次序相同，后面使用关键字变元的部分可以按任意次序排列。例如，对于

```
SUBROUTINE TEST(A,B,C,D)
```

调用时可以使用如下形式：

```
CALL TEST(1,10,100,1000)
```

```
CALL TEST(1,10,D=1000,C=100)
```

```
CALL TEST(D=1000,C=100,A=1,B=10)
```

但是，以下形式是错误的：

CALL TEST(10,1,C=100,D=1000) 头两个实元次序颠倒

CALL TEST(1,10,C=100,1000) 关键字变元后面都要写成关键字变元形式

主调程序中如采用关键字变元调用过程，就必须写出被调子程序的接口块。

#### d) 可选择变元与 OPTIONAL 属性

某些过程中,虽然哑元表中列出好几个哑元，但在实际调用时不一定每次都全部用到。这种情况下，F90 允许只对哑元表中部分哑元作哑实结合，另一部分哑元则按需要可选择结合，称为选择变元。例如内在数组函数 SUM，它的完整的函数及哑元表为 SUM(ARRAY,DIM,MASK)，其中，后二个哑元 DIM、MASK 就是可选择变元。主调程序调用时，可以不选后两个变元，只对第一个变元作哑实结合，如 SUM(A)；也可选上第二个哑元，如 SUM(A,DIM=2)；或后两个可选变元都选中，如 SUM(A,DIM=2,MASK=A>0)。

编写有可选择变元的过程时，可选择变元必须说明具有 OPTIONAL 属性，并要使用 PRESET 内在函数。一个哑元是否为可选哑元，看它是否有 OPTIONAL 属性，有 OPTIONAL 属性的哑元是可选变元，

没有 **OPTIONAL** 属性的哑元是必选的。内在函数 **PRESET** 用来反映它的自变量是否在程序执行部分中出现。**PRESET(A)**的值是一个逻辑值，当 **A** 出现（被使用到）时，函数值为真，否则为假。利用 **PRESET** 函数的真假值，可以通过 **IF** 构造作出变元是否出现时的不同算法。

例如，要求编一子程序，既能求四边形同长(**A+B+C+D**)的值，也能求三角形周长(**A+B+C**)的值。此时 **D** 就是可选择变元，并规定当 **D** 不出现时，置 **D** 值为零。子程序如下：

```
SUBROUTINE SUM(S,A,B,C,D)

IMPLICIT NONE

REAL,INTENT(IN) :: A,B,C

REAL,INTENT(IN),OPTIONAL :: D

REAL,INTENT(OUT) :: S

REAL :: TEMP

IF(PRESET(D)) THEN
```

```
        TEMP=D  
ELSE  
        TEMP=0.  
END IF  
  
S=A+B+C+TEMP  
  
END SUBROUTINE SUM
```

#### e) 哑元改名

过程的优点是具有广泛通用性，一旦编好，解各种问题的主程序都可调用它。但是在为不同目的而使用时，具体问题的物理名称不同。为了加强可读性与可维护性，在不同领域使用过程时，需把哑元名称改为与该领域中的物理名称一致，而 F90 允许改变变元名称。名称的改变是在接口块中进行的，所以主调程序中要写出接口块。

例如，对于上面求边长的子程序，如调用时欲把哑元名 A,B,C,D 改为物理意义明确的名称 UPPER,DOWN,LEFT,RIGHT，只需在主调程序中写入接口块，在接口块的哑元表中用新的哑元名即可：

```
PROGRAM SUMMATION

INTERFACE

    SUBROUTINE SUM(S,UPPER,DOWN,LEFT,RIGHT)

        IMPLICIT NONE

        REAL,INTENT(IN) :: UPPER,DOWN,LEFT

        REAL,INTENT(IN),OPTIONAL :: RIGHT

        REAL,INTENT(OUT) :: S

        REAL :: TEMP

    END SUBROUTINE SUM

END INTERFACE
```

```
READ *, UPPER,DOWN,LEFT,RIGHT

CALL SUBROUTINE SUM(S,UPPER,DOWN,LEFT,RIGHT)

.....

END PROGRAM SUMMATION
```

#### f) INTRINSIC 属性

与 EXTERNAL 语句或属性说明的实元是外部过程相对应，INTRINSIC 语句或属性用来说明实元实际上是内在过程。其一般形式为：

类型定义语句： 类型,INTRINSIC :: 内在函数名[,内在函数名]…

或 INTRINSIC 语句： INTRINSIC 内在过程名[,内在过程名]…

内在过程名必须是内在过程的通用名或专用名。如果是专用名，则可以在其作用范围单元中作为一个过程的实元，但它必须出现在一个 INTRINSIC 语句中，或被该单元中的一个类型声明语句指明

具有 INTRINSIC 属性。需要注意的是，一个内在过程名只能在 INTRINSIC 语句中出现一次，并且不能同时出现在 INTRINSIC 语句和 EXTERNAL 语句中。

例：PROGRAM MAIN

REAL F

REAL,INTRINSIC :: ALOG

F=CALCULATE(0.,1.,ALOG) !使用内在函数 ALOG 作实元

...

END PROGRAM

注意这里必须用专用名 ALOG，而不能用通用名 LOG。

## 6.2.4 其它过程

### a) 内部过程



内部过程是包含在程序单元里的 **CONTAINS** 结构中的函数过程和子程序过程，只有包含它们的程序单元才能够使用该过程。**CONTAINS** 结构标志着程序中可执行部分和程序包含的任意内部子程序之间的分界，它将内部过程与主过程分开。如果程序包含内部子程序，则必须有 **CONTAINS** 结构。在 **F77** 中，一个源文件可以包含一个主程序和几个分别独立的函数或子程序（相当于 **F90** 中的外部过程）。在 **F90** 中，可以将若干个过程用 **CONTAINS** 结构包含在主程序里，它们与宿主程序单元共享变量名。而且，外部过程等其它程序单元都可以有自己的内部过程。

例：program internal

real a,b,c

call find

print \*,c

contains

subroutine find

```
        read *, a,b  
        c=sqrt(a**2+b**2)  
    end subroutine find  
end
```

使用内部过程的规则:在宿主中不要定义子程序名和函数名的类型,也不能指定它们是有 **EXTERNAL** 属性。宿主中的变量名和数组名等在内部过程中有效,有相同的数值。但同一名若在内部过程中又进行了类型声明,则此名被视为其过程中的独立变量,无相同的数值。内部过程中也可引用另一内部过程。

因此,写成内部过程的子程序有一个重要特征,即它们通常没有说明语句。它们使用到的变量等实体的说明已统一出现在主程序说明部分中,它们的变量与主程序同名变量的值是相通的。主程序内可以直接引用内部过程变量的值,或赋之以值,也即主程序内定义的变量是全局的,作用于以 **PROGRAM** 语句到 **END PROGRAM** 语句之间的所有场合。内部过程的第二个重要特征是它们一般

没有哑元。主程序调用时也不需要哑实结合，因为如果内部过程的变量在主程序中被说明，它们就可以直接引用，无需通过哑实结合来传递值。调用内部过程时只要简单地写一个过程名。这是内部过程与外部过程的很大区别。在特殊情况下，内部过程也可保留由自己单独说明的少量哑元。如果变量是在内部子程序中单独说明的，它只是局部变量，作用域只能是局部的，对其主程序的其它部分不起作用。主程序调用时仍需通过哑实结合来传送数据。

## b) 递归过程

**F90** 允许过程递归调用，即在过程内又调用过程自身，这种过程称为递归过程。递归过程包括递归函数和递归子程序，它只是外部过程或内部过程的一种特殊情况，则过程定义语句的前缀中必须出现关键字 **RECURSIVE**。递归调用时，过程直接或间接引用自身或由该过程中的 **ENTRY** 语句定义的函数。

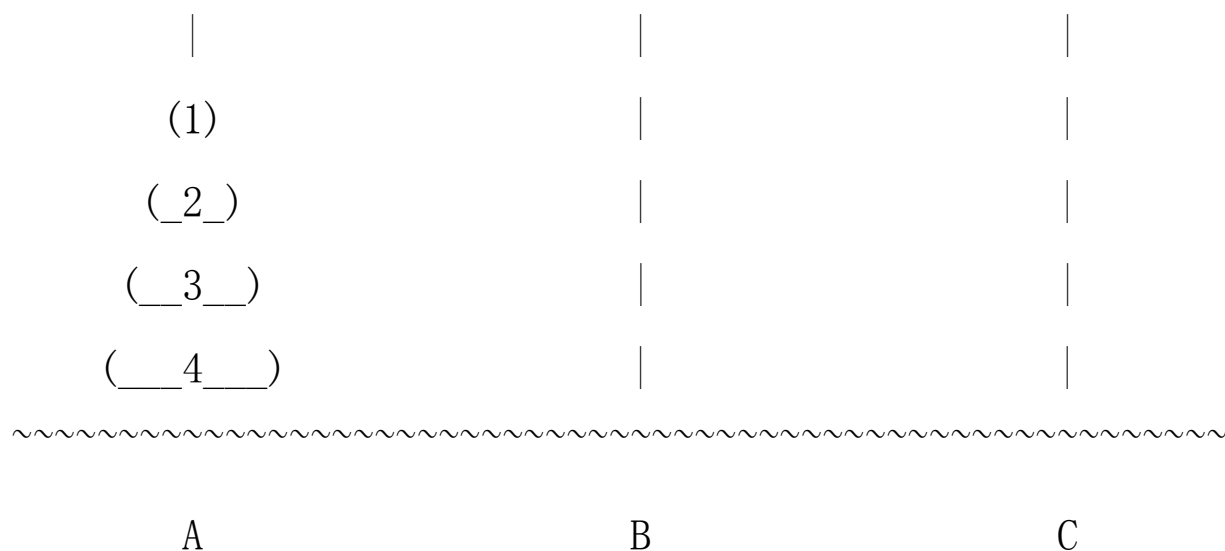
几乎所有递归程序都可用一般程序替代。但递归程序简单明白，清晰易懂，有利于阅读与维护。某些问题当用一般程序编写显得太复杂时，可以用递归程序来实现。递归程序的缺点是效率低，

比起一般程序来，在 CPU 时间与占用内存等方面的开销都要大得多。因此，要根据具体问题的需要，决定是写成递归程序还是一般程序。

下面是一个经典的递归例子，求  $N$  的阶乘  $N!$ 。若用递归过程编程，可使用程序简单明了。因为  $N! = N(N-1)! = \cdots = N(N-1)(N-2) \cdots 2 \cdot 1$ ，如果设  $N!$  的函数过程名为  $\text{FACTORIAL}(N)$ ，则求  $(N-1)!$  调用  $\text{FACTORIAL}(N-1)$ ，如此递推下去，直到求得  $\text{FACTORIAL}(1)=1!$  为止。然后，系统又自动地一层层向上回归，最后求得  $\text{FACTORIAL}(N)$  的值。上述递推与回归的过程即为递归过程。 [\[e 624 01.f90\]](#)

调用递归函数的主程序写法与非递归函数的写法没有什么两样，只是因被调用的是递归函数，最好写明递归函数接口块。在具体调用时，过程只需调用一次，填进实元即可，各级递归调用与回归会自动进行。

为了加深对递归过程的理解，再来看一个关于汉诺塔(The Tower of Hanoi)的例子。据传说，在汉诺有座寺庙，里面有一个举行仪式的塔，它由 3 个柱子和柱子上的 64 个金盘组成。这些金盘大小不一，在修建寺庙时，64 个金盘放在 A 柱子上，并且最大的在最下面，往上依次减小，最小的在最上面。



庙里的和尚的工作就是移动这些金盘，直到 64 个金盘被移到 B 柱子上，到那时就是世界末日来临之时。移动这些金盘必须遵循两个规则：把金盘从一个柱子移到另一个柱子上，一次只能移动一个金盘；一个大的金盘永远不能压在一个较小的金盘上面。这个问题可以用递归子程序来解决。其思路是：为了将上面 4 个盘子从 A 移到 B，需先将上 3 个盘子从 A 移到 C，然后将第 4 个盘子从 A 移到 B，再将上 3 个盘子从 C 移到 B。为了完成 3 个盘子的移动，又需先完成 2 个盘子的移动....。这个递归过程可用递归子程序完成。 [\[e 624 02.f90\]](#)

例：求二项式的展开系数。已知二项式的展开式为： $(a+b)^n = \sum_{i=1}^n C_n^k a^{n-k} b^k$ ，共有  $N+1$  个系数，其系数呈杨辉三角形形式：

$$\begin{array}{ccccccc}
 (a+b)^0 & & & & & & 1 \\
 (a+b)^1 & & & & 1 & & 1 \\
 (a+b)^2 & & & 1 & & 2 & & 1 \\
 (a+b)^3 & & 1 & & 3 & & 3 & & 1 \\
 (a+b)^4 & & 1 & & 4 & & 6 & & 4 & & 1 \\
 (a+b)^5 & 1 & & 5 & & 10 & & 10 & & 5 & & 1
 \end{array}$$

这些系数间有明显的规律，即除了首尾两项系数为 1 外，当  $N>1$  时， $N$  阶中间各项系数是  $N-1$  阶的相应两项系数之和，也即如果把  $N$  阶的系数列为数组  $C(I,N)$ ，则除了  $C(1,N)$ 和  $C(N+1,N)$ 恒为 1 外， $C(I,N)=C(I,N-1)+C(I-1,N-1)$ 。当  $N=1$  时，只有两个取值为 1 的系数  $C(1,1)$ 和  $C(2,1)$ 。因此，任意  $N$  阶

的系数可由  $N-1$  阶的系数求得，直到  $N=1$  为止，可写成递归子程序。[[e 624 03.f90](#)]

### c) 类属过程

类属过程是过程的一种，它允许用不同类型的实元与同一个哑元结合，也即放宽了哑实元结合时类型必须一致的条件。在内在基本函数中，已经有许多类属过程，例如求绝对值的函数 `ABS(X)`，哑元为 `X`，与它结合的实元可以是整型、实型与复型，也即使用同一个过程名 `ABS`，调用函数 `ABS(-1)`、`ABS(-1.)`、`ABS(-1.,1.)` 都是合法的，并且一般来说，其函数值类型就取实元的类型，如上述前两个引用的函数值是 `1` 与 `1.0`。

编写类属过程的方法是先编写着若干个功能相同的过程，它们分别以整型、实型、复型等作哑元类型。而后在主调程序中编写接口，为接口取一个统一的名，接口内分别列出哑元类型不同的过程说明部分语句。这个统一的接口名就是类属过程名，可以在后面执行部分中用不同类型的实元作哑实结合。

例如，要编写求两数之和的类属函数时，分别编写哑元是实型和整型的函数：

```
FUNCTION SUM_REAL(A,B) RESULT(SUM_REAL_RESULT)
```

```
REAL :: A,B,SUM_REAL_RESULT
```

```
SUM_REAL_RESULT=A+B
```

```
END FUNCTION SUM_REAL
```

```
FUNCTION SUM_INTEGER(A,B) RESULT(SUM_INTEGER_RESULT)
```

```
INTEGER :: A,B,SUM_INTEGER_RESULT
```

```
SUM_INTEGER_RESULT=A+B
```

```
END FUNCTION SUM_INTEGER
```

现在把这两个函数过程综合成一个类属函数，类属函数名取为 **MY\_SUM**，在主调程序应写明如下接口：

```
PROGRAM SUMMATION
```

```
INTERFACE MY_SUM
```



```
FUNCTION SUM_REAL(A,B) RESULT(SUM_REAL_RESULT)
```

```
REAL :: A,B,SUM_REAL_RESULT
```

```
END FUNCTION SUM_REAL
```

```
FUNCTION SUM_INTEGER(A,B) RESULT(SUM_INTEGER_RESULT)
```

```
INTEGER :: A,B,SUM_INTEGER_RESULT
```

```
END FUNCTION SUM_INTEGER
```

```
END INTERFACE
```

```
IMPLICIT NONE
```

```
REAL :: X,Y
```

```
INTEGER :: I,J
```

```
READ *, X,Y,I,J
```

```
PRINT *, MY_SUM(X,Y),MY_SUM(I,J)
```

## END PROGRAM SUMMATION

### d) 多层调用

主程序调用过程，称为第一层调用。被调用的过程可能也要调用更小的下属过程，这种调用称为第二层调用，依此类推。这样逐层调用下属过程，称为多层调用。

在多层调用中，函数可以调用下属的函数过程或子程序过程，子程序过程也可调用下属的函数过程或子程序过程，并不要求被调用的过程与自己的性质相同。多层调用中的程序控制流程是：控制从主程序(第 0 层)的第一个执行语句开始，顺次往下执行，当遇到调用第一个第一层过程时，控制转入该过程的第一条可执行语句，并把第 0 层中的实元值赋给该第一层过程的哑元，顺次往下执行。如果再遇到调用第二层过程的语句，控制又转入该层过程的第一条可执行语句，并把第一层的实元值赋给第二层的哑元。如果最下一层子程序内没有调用语句，则控制遇到过程结束语句时，返回上层过程调用语句后，继续执行尚未执行的语句，直到第一层过程结束语句，控制再返回到主程序的调用语句，继续执行完后面的语句。直至遇到主程序结束语句，结束控制。

因此，这个控制机制是按顺序进行的，这一点也可由执行 Visual Fortran 的 Debug 功能时，跟踪过程的顺序执行和变量的变化可见。

### 6.2.5 过程接口

#### a) 接口形式

一个内部过程总是由程序单元中的语句来调用的。一般来讲，编译程序知道内部过程的一切情况，如知道该过程是一个函数或子程序、过程名、哑元的名字、变量类型和属性、函数结果的特性等等。这个信息的集合被称为过程的接口(interface)。对于内部过程、内在过程和模块，过程接口对编译程序而言是已知的和显式给出的，故称显式接口。

由于主调程序与被调过程是分别编译的，而 F90 扩充了过程的许多功能，这些功能单靠简单的调用语句无法反映，编译系统也无法知晓。如在调用一个外部过程或一个哑过程时，编译系统通常不知道该过程的各种情况，这种接口是隐式的。在 F77 中可用 EXTERNAL 语句来指明一个外部过程或哑过程，但此语句仅说明每一个外部名是一个外部过程名或哑过程名，并没有指明过程的接口，

所以接口仍是隐式的。为了全面准确地通知编译系统，在主调程序中有时需要加入接口块，以说明主调程序与被调程序的接口。接口块是 **F90** 中引进的新颖程序块，它显式指明了过程接口的机制。通过接口块可用为一个外部过程或哑过程指明一个显式的接口。这比 **EXTERNAL** 语句提供了更多的信息，也提高了程序的可读性。

过程接口确定过程被调用的形式，它由过程的特性、过程名、各哑元的名字和特性以及过程的类属标识符(可以省略)组成，一般它们都被写在一个过程的开头部分。此接口块被放在主调程序的说明部分中，通常还应写在类型说明语句之前，它的内容是被调用的过程中的说明部分，功能是通知编译系统，主调程序调用的过程中各种变元的类型、属性、性质等。

## b) **INTERFACE** 语句

接口块应写在主调程序(主程序或过程)的说明部分中，一般是写在最前面：

**PROGRAM** 程序名

接口块

主调程序内变元说明

执行语句

END PROGRAM

过程接口块的一般形式为:

INTERFACE [类属说明]

[接口体]...

[模块过程语句]...

END INTERFACE [类属说明]

其中类属说明的形式为:

- |                 |    |              |
|-----------------|----|--------------|
| ✦ 类属名           | -> | 类属过程         |
| ✦ OPERATOR      | -> | 重载操作符、自定义操作符 |
| ✦ ASSIGNMENT(=) | -> | 重载赋值号        |

接口体的形式为：

- 函数语句

[说明部分]

函数 END 语句

- 子程序语句

[说明部分]

子程序 END 语句

模块过程语句的形式为：MODULE PROCEDURE 过程名表。

这里，接口体是过程头的精确拷贝，定义了过程中的变量和函数结果。使用接口块时应该注意的  
是：

- ✦ 接口块以 INTERFACE 语句开始，END INTERFACE 语句结束，块内只能取被调用过程中的说明部分，不允许出现任何可执行语句。接口块内的语句构成接口体。

- ✦ 接口体中不能含有 ENTRY 语句、DATA 语句、FORMAT 语句、语句函数语句。
- ✦ 接口块不允许出现在 BLOCK DATA 程序单元中。
- ✦ 接口块中可以有多多个接口体，即一个接口块中可以说明多个被调用过程，每个过程用自己的开始语句与结束语句定界，排列次序任意。

例：interface

```
        subroutine swap(x,y)

            real x,y

        end subroutine

end interface

real a,b

read *,a,b

call swap(a,b)
```

```
end
```

```
subroutine swap(x,y)
```

```
    real x,y
```

```
    z=x;x=y;y=z
```

```
end subroutine
```

例：求正方矩阵对角元的和(trace)。 [\[e 625 01.f90\]](#)

例：INTERFACE

```
    SUBROUTINE EXT1(X,Y,Z)
```

```
        REAL,DIMENSION(100,100)::X,Y,Z
```

```
    END SUBROUTINE EXT1
```

```
    SUBROUTINE EXT2(X,Z)
```

```
        REAL X
```



```
        COMPLEX(KIND=4) Z(2000)

END SUBROUTINE EXT2

FUNCTION EXT3(P,Q)

    LOGICAL EXT3

    INTEGER P(1000)

    LOGICAL Q(1000)

END FUNCTION EXT3

END INTERFACE
```

这个接口块对于三个外部过程 EXT1、EXT2 和 EXT3 说明了显式接口。它是无类属名的接口块。

### c) 必需接口

接口块并不是每个主调程序都必须写的。若仅使用 F77 语言编写的子程序，则无需在主调程序单

元中写接口块。但若使用 F90 提供的现代化手段编写程序，通常需要在引用程序单元中写入接口块，否则编译容易出错，而且 F90 不提倡使用 COMMON 语句进行单元间的数据传递，其功能已由模块中的接口块代替。确切地说，凡遇下列情况之一时，主调程序必须有接口块：

1、如果外部过程具有以下特征：

- ✦ 过程的哑元有可选择属性。
- ✦ 过程的哑元是假定形数组、指针变量、目标变量。
- ✦ 函数过程的结果是数组或指针。
- ✦ 对于字符型函数过程的结果、其长度不是常数，也非假定长度(\*)。

2、如果调用过程时出现：

- ✦ 实元是关键字变元。
- ✦ 用一个类属名调用。
- ✦ 用超载赋值号(对于子程序)。
- ✦ 用超载操作符(对于函数)。

3、如果过程前缀关键词是 **ELEMENTAL**。

#### d) 超载操作符

超载操作符的形式仍是系统内部操作符，如+、-、\*、/等，但通过编出恰当的过程，可以扩充这些操作符的功能。例如；‘+’本来用于对数值作算术操作，但经恰当的编程后‘+’也可用于字符型操作，这就像车辆超载一样，故称为超载操作符。定义超载操作符，需先编写一个实现此超载(扩充)功能的函数过程，再在主调程序中编写过程的接口，在接口语句后加上超载说明，其一般形式为：

**INTERFACE OPERATOR(被超载使用的操作符号)**

例如，要使‘+’超载能执行如下操作：把两个字符串的最后一个字母接起来。实现超载时，先编一个能实现该操作功能的函数：

```
FUNCTION ADD(A,B) RESULT(ADD_RESULT)
```

```
IMPLICIT NONE
```

```
CHARACTER(LEN=*),INTENT(IN) :: A,B
```

```
CHARACTER(LEN=2) :: ADD_RESULT
```

```
ADD_RESULT=A(LEN_TRIM(A):LEN_TRIM(A))/B(LEN_TRIM(B):LEN_TRIM(B))
```

```
END FUNCTION ADD
```

其中内在函数 `LEN_TRIM` 为字符串去掉了尾部空格后的实际长度。现在要把这个函数功能定义为超载的操作符 ‘+’，则应在主调程序中编写如下接口块：

```
INTERFACE OPERATOR(+)
```

```
FUNCTION ADD(A,B) RESULT(ADD_RESULT)
```

```
IMPLICIT NONE
```

```
CHARACTER(LEN=*),INTENT(IN) :: A,B
```

```
CHARACTER(LEN=2) :: ADD_RESULT
```

```
END FUNCTION ADD
```

## END INTERFACE

接口的作用是向编译系统提示，遇到操作符‘+’时，如果操作数不是数值，就不是原来意义的加法，操作含义要到 **FUNCTION ADD** 中找。当主调程序有了上述接口块后，在下面执行部分中执行字符串操作 **CH1+CH2** 时，+号作超载用。[\[e 625 02.f90\]](#)

### e) 自定义操作符

如果用内部操作符超载后仍不能达到算法要求的功能，还可自定义一个新形式的操作符作崭新的操作。自定义新操作符可由字母组成，两边用小数点(.)作定界符，以便与一般变量用关键字区别开来。它的形式与逻辑操作符**.AND.**、**.OR.**等形式一致。与我们可定义名义上的加法**.add.**，减法**.minus.**等。操作符名最多有 31 个字符组成，名称不得与 **FORTRAN** 关键字相同，如不可取**.AND.**或**.GT.**等。  
[\[pi.f90\]](#)

### f) 超载赋值号

赋值号赋值号(=)也可超载。根据规定，赋值号两边的类型必须相同，如两边都是数值型，或者都是逻辑型、字符型等。但 F90 也允许使用超载赋值号，即允许把一个类型的表达式赋给另一类型的变量。例如，可以把逻辑表达式赋给一个数值型变量，并使之构成一一对应关系。

实现赋值号超载的方法是，先编一个实现两个不同类型变量值之间一一对应关系的子程序过程（非函数过程），而后在主调程序中编一个接口，用 **ASSIGNMENT** 接口语句实现给赋值号定义不同的语义。接口语句的形式是：

#### **INTERFACE ASSIGNMENT(=)**

在具有 **ASSIGNMENT** 选择的接口块中，包含的子程序过程只有两个变元，第一个具有 **INTENT(OUT)** 或 **INTENT(INOUT)** 属性，第二个具有 **INTENT(IN)** 属性。作赋值运算时，第一个变元为被赋值的对象。

注意：要实现超载赋值号功能，必须编成子程序过程形式。相反，要实现超载操作符功能，必须编成函数过程形式，不可混用。

例：编一程序把逻辑量超载赋值给整型变量。先编一个实现这一功能的子程序，

```
SUBROUTINE LOG_INT(I,L)

    IMPLICIT NONE

    LOGICAL, INTENT(IN) :: L

    INTEGER, INTENT(OUT):: I

    IF(L) I=1

    IF(.NOT.L) I=0

END SUBROUTINE LOG_INT
```

再在主程序内编写接口,

```
INTERFACE ASSIGNMENT(=)

    SUBROUTINE LOG_INT(I,L)

        IMPLICIT NONE

        LOGICAL, INTENT(IN) :: L
```

```
INTEGER, INTENT(OUT):: I  
  
END SUBROUTINE LOG_INT  
  
END INTERFACE
```

此后，在主调程序内可任意赋值。此接口块定义的赋值运算与通常的赋值含义不同，但在程序中仍可用等号“=”来使用。如 `I=1` 是通常的赋值，而 `I=.TRUE.` 是超载赋值，但 `I` 的取值认为 1。

## 6.2.6 作用域

### a) 作用域单元

在用带有语句标号的 `GOTO` 语句实现语句间的转移时会遇到这样一个问题，即程序中在什么情况下可以使用相同的语句标号而不会产生歧义？这就是语句标号的作用域问题。实际上主程序或每个过程都有一套独立的语句标号，包括宿主程序含有几个内部过程的情况。类似地，对变量名称也会有这样的问题，这就是名称的作用域问题。对于语句标号来说，它的作用域是主程序或过程，



但不包括它含有的内部过程，相同的语句标号可以用在宿主程序和内部过程里而不会产生歧义。

**F90** 标准以作用域单元的形式来定义名称。作用域单元是一个程序或程序的一部分，在作用域单元中定义一个名称，这个名称在作用域单元中有效。作用域单元可以是整个程序、程序单元、一个单独的语句或语句的一部分。名称可以是常量名、变量名、过程名、操作符或任何其它名称。名称可以用于程序、过程、变量、数组、哑元、命名常量、派生类型或块结构。名称有三种：全局名称、局部名称和语句名称。

作用域单元有以下几种：**1**、派生类型定义；**2**、过程接口体，不包括其内部的派生类型定义和过程接口体；**3**、程序单元或过程，不包括内部的派生类型定义、过程接口体以及内部过程。

下面是包括 5 个作用域单元的例子，作用域单元可以包含其他作用域单元。

```
MODULE SCOPE1                ! Scoping unit 1
...
CONTAINS                     ! Scoping unit 1
```

FUNCTION SCOPE2	! Scoping unit 2
TYPE SCOPE3	! Scoping unit 3
...	! Scoping unit 3
END TYPE SCOPE3	! Scoping unit 3
INTERFACE	! Scoping unit 3
...	! Scoping unit 4
END INTERFACE	! Scoping unit 3
...	! Scoping unit 2
CONTAINS	! Scoping unit 2
SUBROUTINE SCOPE5	! Scoping unit 5
...	! Scoping unit 5
END SUBROUTINE SCOPE5	! Scoping unit 5

END FUNCTION SCOPE2          ! Scoping unit 2

END MODULESCOPE1          ! Scoping unit 1

## b) 名称的作用域

### ✦ 全局名称

全局名称用来识别程序单元、公共块和外部过程。全局名称在程序的任何地方都是有效的，所以只能在程序中定义一次。例如，如果用户在一个程序中使用了名为 **Son** 的子程序，就不能再在该程序中使用名为 **Son** 的公共块或函数。

### ✦ 局部名称

局部名称是用来识别变量(标量和数组)、常量、命名常量、语句函数、内部过程、模块过程、哑元过程、内在过程、一般标识符、派生类型和名称列表组的名称。派生数据类型的成员和关键字变元(哑元)也是局部名称。局部名称可以覆盖全局名称和同一程序单元中的其它局部名称(关键字变

元、类属名称和公共块名称除外)。如果一个名称对某个程序单元是局部的，同样的名称即可以作为全局名称又可以作为其它程序单元中的局部名称。

### ✚ 内在过程的双重名称

因为 **FORTRAN** 语言的关键字不予保留，所以用户可以创建名称和 **FORTRAN** 内在过程名称一样的变量、常量或过程。一旦创建了这样的名称，原来的内在函数就不能再被访问。例如，下面定义了新的函数 **sin**：

```
SUBROUTINE sub
```

```
...
```

```
CONTAINS
```

```
FUNCTION sin(x)
```

```
...
```

END FUNCTION sin

END SUBROUTINE sub

任何在子程序 `sub` 中对 `sin` 的引用都会调用其中定义的内部函数，而不是原来的内在函数。

类似地，任何与同名内在过程的标准类型不同的类型声明都会产生一个局部名称。下面的例子声明了一个名为 `sin` 的变量：

CHARACTER(LEN=5) :: sin

任何使用这个字符变量的程序或内部过程都不能再使用原来的内在函数。如果该变量在模块中以 `PRIVATE` 属性声明，模块外的程序单元仍可以使用内在函数 `sin`。

### 📌 语句名称

语句名称的作用域是一条语句。语句名称可以出现在语句函数的语句中、一个 `DATA` 语句的隐 `DO` 循环中、一个数组构造器中。在语句函数语句中作为哑元出现的变量名，在其出现之处的范围就

是语句域。DO 变量的域(必须是整数)是隐 DO 列表。例如：

```
DIMENSION x(10)
```

```
Add(a,b)=a+b
```

```
DO n=1,10
```

```
    x(n)=add(y,z)
```

```
END DO
```

在此例中，a 和 b 的域被限制在语句函数内部。n 的域是整个 DO 循环。

### ✦ 公共块名称

公共块名称是全局名称。因为局部名称可以和全局名称重名，在局部实体中对这个名称的引用指的是局部名称。当公共块在 **SAVE** 语句中命名时，它应该用斜杠围起来以和其它同名的局部变量区分。例如：

```
COMMON/happy/cat,dog,mouse
```

```
CHARACTER(20) happy
```

```
SAVE /happy/  !SAVE 的是公共块而不是变量
```

### ✦ 函数结果名

函数结果是另一个允许出现重名的例子。对每一个在函数过程中的 **FUNCTION** 语句或 **ENTRY** 语句都可有一个结果变量。如果没有用 **RESULT** 指定另外的变量名，结果变量和函数定义时同名。

### ✦ 派生类型成员名称

如果一个变量是其它名称的成员，它的域和包含该变量的名称的域一样。例如，一个模块中定义的派生类型和模块的域一致，也和任何使用该模块的域一致；承认派生类型的程序同样承认派生类型的成员；对于数组的情况也是类似的，在数组适用的域内，数组片段也是有效的。

### ✦ 其它情形

其它实体，例如语句标号、I/O 单元、操作符和赋值号等都有域的概念。对于这些实体应遵守下面的规则：1、语句标号始终被认为是局部的，两个域相同的单元不能使用相同的标号；2、内在操作符(例如+，-，\*，\*\*，或/)是全局的，但自定义的操作符是局部实体，特殊操作符的域由定义该操作符的过程的范围决定。可以通过使用过程接口块使自定义的操作符成为全局的；3、赋值符号(=)是全局实体，可以在一个接口块中确定附加的一般赋值操作。

## 模块

模块也是一种在主程序单元之外独立编写的程序单元。它有独特的形式，即模块程序单元内没有可执行语句。除了说明语句外，最多包含内部过程。模块的主要作用是供其它程序单元引用。一个程序单元如果引用模块，实际上就是把该模块内的全部语句复制到本程序单元中，并且所有与模块中的名字相同的变量等，彼此取值相通、共享存贮单元。所以引用模块起两个作用：共享与复制。



如果模块程序单元中包含有内部过程，这些过程也可供其它程序单元使用。

### 6.3.1 数据共享

不同的程序单元是各个分别编译的，编译工作中很重要的一件事是给程序单元中各变量分配内部存贮单元（内存单元）。由于是分别编译的，程序单元 A 中的变量 X 所占存贮单元与程序单元 B 中变量 X 所占存贮单元不可能是在同一个存贮单元中，因此，程序单元中同名的变量名并不能相互传递数值。

要想使两个不同的程序单元间的变量相互传递数值有两个办法，一个是哑实结合，一个就是数据共享。

#### 11) 共享方式

共享就是让编译系统把两个变量名分配在同一个内存单元中。已知访问一个变量名，实际上就是访问它所分配的内存单元中存贮的值。既然 A 程序单元中的变量 X 与 B 程序单元中的变量 Y 分配

在同一存贮单元中，访问 X 或访问 Y 都是访问同一个内存单元，取得的数也是相同的。

如果两个变量共享存贮单元，当 A 程序单元中  $X=2$  时，B 程序单元中 Y 也是 2。如果 A 程序单元中执行语句  $X=X+1$ ，则 B 程序单元中 Y 的值也变为 3，反之也然。当然，前提是 X 和 Y 的类型必须一致。如果类型不一致，其结果将很难预料。

共享方式有以下几种：使用 COMMON 语句和 EQUIVALENCE 语句(F77)，使用模块(F90)。另外，使用 INCLUDE 语句可进行语句段复制。

### mm) COMMON 语句

F77 中不同程序单元间数据的共享通常是用 COMMON 和 EQUIVALENCE 语句，使用这些语句共享数据的效率不高，编程时容易出错。但 F90 仍保留了这两个语句以兼容以前的标准。

COMMON 语句用于在不同程序单元之间进行实体的数据批量传递，它比哑实结合的方法进行数据传递的速度要快。其方法是开辟一个公共块，公共块可以是无名的(只能有一个)，也可以是有名称的，其一般形式为：

**COMMON** **[[公共块名 1]/变量名表 1[[,]/[公共块名 2]/变量名表 2]...**

其中的公共块名可以和变量名相同，变量实体名不得是哑元、可分配数组、自动对象、函数名或函数结果名或 **ENTRY** 名，并且不得有 **PARAMETER** 属性。不同程序单元中相同公用名下的变量名可以是不同的。相同公用名中的变量在不同程序单元中，按位置一一对应共享同一存储单元中的数值。由于 **COMMON** 语句是说明语句，它的位置必须在可执行语句之前，通常是紧跟在程序单元的起始句之后，一个程序单元可以有多条 **COMMON** 语句。例如：下面的 **COMMON** 语句段

```
COMMON/happy/we,you,they
```

```
COMMON/      /our,your,their
```

```
COMMON/happy/i,he,she
```

```
COMMON/angry/dog,cat,mouse
```

```
COMMON my,his,her
```

等价于语句段，

```
COMMON/happy/we,you,they,i,he,she
```

```
COMMON/angry/dog,cat,mouse
```

```
COMMON/      /our,your,their,my,his,her
```

其中，有两条有名公共块语句，而 ‘//’ 是无名的。

由于各个程序单元中的变量名是独立的，它们并不会因为名字相同而建立起数值的联系。例如，主程序中名为 **x** 的变量和子程序中的 **x** 变量虽然同名，但它们各有自己的存储单元，互不相关。但如果我们在主程序和子程序的说明部分各自都增加一条无名共用区语句：**COMMON x**，则 **FORTRAN** 编译程序在存储区中开辟了一个公用数据区，主程序和子程序中的 **COMMON** 语句中的第一个变量共同占用公共块的第一个存储单元，达到数据共享。

例如：主程序中的语句

```
COMMON X,Y,Z(3),I
```

和子程序中的语句

```
COMMON A,B,C(3),J
```

使得无名公共块中变量 X 和 A，Y 和 B，数组 Z 和 C，I 和 J 分别被分配在相同的存储单元中。占同一个存储单元的那些变量在不同的程序单位中，它们的名字不需要相同。

COMMON 语句还可用来声明数组，例如：

```
COMMON /food/restruant(100),McDonald(10)
```

这条语句已经按 I-N 规则声明了实型数组和整型数组，无需用 DIMENSION 语句或属性对数组名重新说明。如果要重新定义类型的话，则数组大小不得在 COMMON 语句和类型说明语句中重复出现。例如：

```
SUBROUTINE unit1
```

```
REAL(8)      x(5)
```

```
INTEGER      J
```

```
CHARACTER    str*12
```

```
TYPE(member) club(50)
```

```
COMMON/blocka/x,j,str,club
```

```
...
```

```
SUBROUTINE unit2
```

```
REAL(8)      z(5)
```

```
INTEGER      m
```

```
CHARACTER    chr*12
```

```
TYPE(member) myclub(50)
```

```
COMMON/blocka/z,m,chr,myclub
```

nn) EQUIVALENCE 语句

**EQUIVALENCE** 语句是说明语句，它必须出现在程序单元的执行语句之前。它的作用是使同一个程序单元中的两个或更多的变量共用同一个存储单元，以节省内存。这里特别需要强调的是同一个程序单元。因此，主程序和过程之间以及过程相互之间不同变量不能用 **EQUIVALENCE** 语句来指定共用存储单元。等价语句的另一种用途是允许用两个或更多的变量名代表同一个量。等价语句的形式为：

**EQUIVALENCE** (变量名表 1),(变量名表 2),...

等价语句后面的每一对括号内的变量表中，可以是变量名、数组名或数组元素等，但至少应该有两个变量名，之间用逗号分开。例如，**EQUIVALENCE (A,B)**语句指定本程序单位中的变量 **A** 和 **B** 同占一个存储单元，也就是说这两个变量都从同一个存储单元中取值，只要其中一个变量得到某个值，其它一个变量也就必须具有相同的值。但必须强调这不是数学上的等值，而是由于共享一个存储单元而得到的方便。

如果数组出现在变量名表中，则它们的大小必须相同，等价时按数组元素的排列顺序一一对应。

## oo) INCLUDE 语句

可以用 INCLUDE 语句将另一个文件中的原程序段包括进来，实现复制功能。比如，可以将程序中多处引用的同一段语句块放入一个文件中，这样在调用时可以保持语句的形式和文字是完全相同的。INCLUDE 语句的功能是让编译器停止读取当前文件而从一个文件中读取语句，读完后再继续读取当前文件中的下一条语句。它的一般形式是：

**INCLUDE '文件名[/[NO]LIST]'**

文件名是一被当前使用的操作系统认可的字符串，/[NO]LIST 选项指明包含文件中的代码是否出现在编译源程序的列单中，缺省值是/NOLIST。包含文件中可以有其它的 INCLUDE 语句，但不能是递归的，否则将层层套圈直到耗尽系统资源。

例：主程序

```
PROGRAM
```

```
    INCLUDE 'COMMON.FOR'
```



```
REAL,DIMENSION(M) :: Z

CALL CUBE

DO I=1,M

    Z(I)=X(I)+SQRT(Y(I))

    ...

END DO

END
```

```
SUBROUTINE CUBE

    INCLUDE 'COMMON.FOR'

    DO I=1,M

        X(I)=Y(I)**3
```

```
END DO
```

```
RETURN
```

```
END
```

包含文件 COMMON.FOR 是:

```
INTEGER,PARAMETER :: M=100
```

```
REAL,DIMENSION(M) :: X,Y
```

## pp) 模块

模块是 F90 中新增加的、使数据共享的最现代的手段。只要是出现在模块中的变量，都能与引用该模块的程序单元中的变量共享。模块中如果有内部过程，这些过程也可各引用该模块的程序单元所共用，因而又起了过程库的作用。模块的共享关系示意图如下，双向箭头表示数据可存可取，单根连线表示模块内部过程供下面外部过程调用。

模块的功能是提供一种方便有效的常量、变量、类型定义及过程的共享途径。它可代替 **COMMON**、**EQUIVALENCE** 和 **INCLUDE** 语句的功能。模块的用途主要有：

- ✦ 包含通常使用的过程
- ✦ 声明全局变量和派生类型
- ✦ 声明外部过程的接口块
- ✦ 初始化全局数据和全局可分配数组
- ✦ 封装数据和处理数据的过程

模块程序单元有它独特的形式，即单元内没有可执行语句，除了说明语句外，最多包含内部过程。模块这种程序单元不能被直接执行，必须用 **USE** 语句引用。

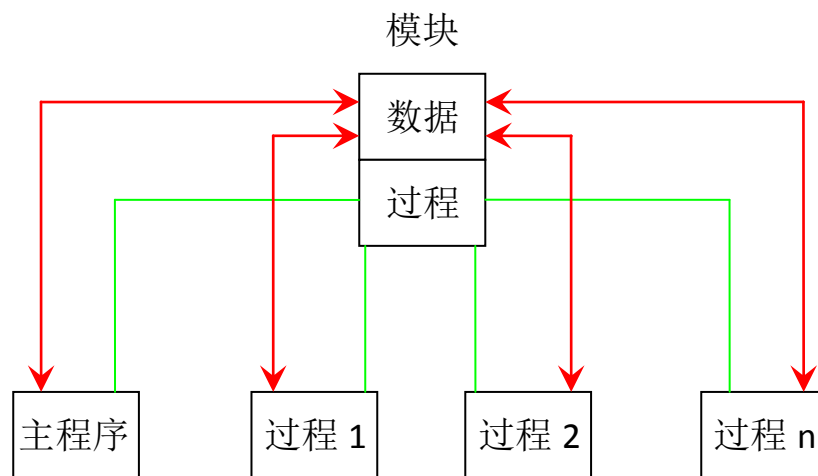
存储在公共块 **COMMON** 中的变量可以被摘录和保存到模块之中。这样，通过引用模块，其它程序单元就可使用这些变量而不必包括 **COMMON** 块，从而不必在多个程序单元中声明变量。还可以保证在所有使用这些变量的程序单元中，变量声明是一致的，且是用同一个值初始化的。如果一个程序单元中，只使用模块中的部分变量或需要重新命名部分变量，该单元可以指定这些变量具有

ONLY 属性。

在程序中使用模块和使用 `INCLUDE` 语句很类似。每种情况下，都由某个独立文件中内容提供程序运行所需的信息。使用模块比使用 `INCLUDE` 配语句的优势在于：

- ✦ 封装数据和操作这些数据的过程：模块提供了封装相关定义和操作的简便方法，可以把多个相关但不同的文件中的信息结合到一个模块中，而不需使用多个 `INCLUDE` 语句。
- ✦ 变量、常数和模块过程的命名控制：允许临时重命名常数、变量甚至过程。
- ✦ 隐式接口：模块过程的参数和返回值对于主调程序是可知的，并且和调用过程匹配。
- ✦ 数据类型和操作符定义：允许为派生类型定义特殊操作符，还可以把内在操作符扩展到其它数据类型。
- ✦ 只在程序中一个位置指定信息：这样能保证使用该信息的不同的程序单元对信息的解释是一致的。例如，某个程序单元中的一条 `IMPLICIT` 或 `EXPLICIT` 语句可能会引起对一个包括文件的解释和其它包括这个文件的程序单元的解释不同。





### 6.3.2 模块的用法

#### a) 定义模块

模块单元的一般形式是：

**MODULE** 模块名

类型说明部分

[CONTAINS

内部过程

...

[内部过程]]

END MODULE [模块名]

**MODULE** 语句下面写各种变量、数组等实体的类型说明语句，以及派生类型定义及接口块。注意到其中只有说明部分，没有执行部分。自 **CONTAINS** 语句开始连同它后面的各内部过程是可选的，一般不用。通常在为某一个派生类型规定新的操作符时，就把实现这些新操作的过程作为模块的内部过程放在 **CONTAINS** 后面，以便把这种操作定义供各外部过程共享。当模块有内部过程时，必须把整个过程完整地写入。各内部过程（可以是函数或子程序）次序可以任意。

模块程序单元可以不至一个，每个模块都独立编写，而后与主程序单元、外部过程单元一起输入机中编译、连接，才可以运行。

```
例:  MODULE DATA_MODULE  
  
      REAL, DIMENSION(1:10) :: A  
  
      INTEGER, PARAMETER :: I=15  
  
      INTEGER :: B=5  
  
      END MODULE DATA_MODULE
```

这个例子中，模块的实体只有类型说明语句，没有内部过程。如果有某一个外部过程引用了这一模块，则相当于把其中的三条类型说明语句移到该外部过程的说明部分中，**I** 及 **B** 的初值传递给外部过程中同名的实体。

```
例:  MODULE STUDENT_MODULE  
  
      TYPE STUDENT_TYPE  
  
      CHARACTER(LEN=20) :: NAME  
  
      INTEGER :: SCORE
```



```
END TYPE STUDENT_TYPE
```

```
END MODULE STUDENT_MODULE
```

该模块内把学生数据结构定义成一个派生类型。对于每个引用这个模块的外部过程而言，相当于把这一派生类型的定义移到自己的说明部分中，因而可以使用这个类型来说明程序体内的变量名。

例：     MODULE MY\_MODULE

```
REAL, PARAMETER :: Pi=3.141592654
```

```
CONTAINS
```

```
SUBROUTINE SWAP(X,Y)
```

```
REAL :: TEMP,X,Y
```

```
TEMP=X
```

```
X=Y
```

```
Y=TEMP
```

END SUBROUTINE SWAP

END MODULE MY\_MODULE

该模块内有一内部过程 **SWAP**，引用这个模块的外部过程都将包含有此内部过程。

## b) 引用模块

任何程序单元，要共享模块程序单元内的内容，只需引用该模块名，引用方法是在本程序单元说明部分的最前面加上 **USE** 语句。通过模块共享可以取代各程序单元间哑实结合，使有哑元的过程改为无哑元的过程。**USE** 语句的一般形式为：

**USE 模块名 1, 模块名 2, ... 模块名 n**

**USE** 语句表示在本程序单元中引用了模块 1、模块 2、...、模块 n，即相当于把这些模块中的语句内容都移植到本程序单元内。使之其中的全部公用实体成为可访问的。**USE** 语句还可以有多种引用方式，可视不同需要灵活引用模块，譬如可以只对模块中一部分变量共享等。它包括两方面：模块定义时规定只有哪些内容允许共享、引用模块时只要求共享哪些内容。

✦ 模块的 **PRIVATE** 属性:

当定义派生类型的 **TYPE** 块写在模块中时, 可以限制该派生类型定义的使用范围, 以及类型定义内各成员的使用范围。譬如规定模块内的该派生类型或派生类型内的成员只供本模块内引用, 不许模块外程序单元引用。其形式是:

**TYPE, PRIVATE :: 派生类型名**

**成员 1 类型说明**

**.....**

**成员 n 类型说明**

**END TYPE [派生类型名]**

使用 **PRIVATE** 专用特性后, 可以禁止一切外部过程 (包括主程序) 访问派生类型的内部成员, 而只是把派生定义类型作为一个整体黑箱使用。以后如果派生类型内部成员要改动, 只需改写派生类型部分, 引用模块的各程序单元可不必改动。

✦ 模块内的**变量改名**:

如果需要对多个模块进行访问，而在不同的模块中可能用到了相同的名字，因此允许 **USE** 语句对被访问的实体重新命名，以解决局部实体和模块中访问实体之间的名字冲突问题。要重新命名时，**USE** 语句具有下列形式：

**USE 模块名 [,改名列表]**

其中，改名列表的形式为：**局部名 1=>块中名 1, 局部名 2=>块中名 2, ...**。其中，局部名是使用 **USE** 语句的程序单元中用的名字，块中名是待改的模块内部使用的变量名。

如模块中定义的变量名是 **A、B**，程序单元中同名变量 **A、B** 与之共享，但若要在程序单元中把变量名改为 **C、D**，则只需在单元内把引用语句写成：

**USE 模块名, C=>A, D=>B**

即可，而无需修改模块。

✚ **ONLY 选项：**

可以规定只取模块中一部分变量与本程序单元中实体共享，即只需要使用模块中的部分实体，其

它实体没有共享关系。这时可在 USE 语句中使用 ONLY 选项。这时 USE 语句具有下列形式：

USE 模块名, ONLY : [,仅用名列表]

其中，仅用名列表的形式为：[局部名 1=>]块中名 1, [局部名 2=>]块中名 2, ...。

### 6.3.3 模块的应用

引入模块给程序设计带来了诸多方便，模块可应用于以下几个方面。

#### a) 全局数据

如果数据是在整个可执行程序中都要用到的全局数据，可以把它们放在一个模块中统一说明，在需要这些数据的程序单元内使用 USE 语句导出它们即可。例如：

```
MODULE DATA_MODULE
```

```
SAVE
```

```
REAL :: A(10), B, C(50,10)
```

```
INTEGER,PARAMETER :: I=10
```

```
COMPLEX D(I,I)
```

```
END MODULE DATA_MODULE
```

如果要在程序单元中访问模块中定义的全部数据对象，可使用语句：

```
USE DATA_MODULE
```

如果只需访问其中的 A 和 D，则可使用语句：

```
USE DATA_MODULE, ONLY : A,D
```

如果要避免名字冲突而改名的话，则可使用语句：

```
USE DATA_MODULE, ONLY : A_MODULE=>A, D_MODULE=>D
```

## b) 过程共享

利用模块，还可以把整个可执行程序中都要用到的全局过程封装在一起，即把这些过程放在一个模块中统一说明，而在需要这些过程的程序单元内使用 **USE** 语句导出它们即可，这就是模块过程。使用的方式是，首先在模块中把待用的全局过程定义为模块的内部过程，即写在 **CONTAINS** 语句之后，在调用过程单元中写上 **USE** 语句以调用此模块，再写上接口块，接口块的实体是模块过程语句：

**USE MODULE PROCEDURE** 模块过程名表

例：     **PROGRAM CHANGE\_KIND**

**USE Module1**

**INTERFACE DEFAULT**

**MODULE PROCEDURE Sub1, Sub2**

**END INTERFACE**

integer(2) in

integer indef

indef = DEFAULT(in)

END PROGRAM

MODULE Module1

CONTAINS

FUNCTION Sub1(y)

REAL(8) y

sub1 = REAL(y)

END FUNCTION



```
FUNCTION Sub2(z)

    INTEGER(2) z

    sub2 = INT(z)

END FUNCTION

END MODULE
```

### c) 公用派生类型

模块还可用来封装一些派生类型，供其它程序单元公用。例如：

```
MODULE SPARSE_MATRIX

    TYPE NONEZERO

        REAL A

        INTEGER I,J
```

```
END TYPE
```

```
END MODULE SPARE_MATRIX
```

定义了一个由实数和两个整数构成的数据类型，用以表示稀疏矩阵的非零元素。其中的实数表示元素的值，两个整数表示该元素的下标。于是，派生类型 **NONZERO** 就可被其他单元通过 **USE** 语句来共用。

#### d) 全局可分配数组

在许多程序中需要一些全局公用的可分配数组，它们的大小在程序执行前是不知道的，这时也可用模块来实现。例如：

```
PROGRAM MAIN
```

```
...
```

```
CALL CONFIGURE_ARRAYS    !分配数组
```

```
CALL COMPUTE
```

!用分配好的数组进行计算

```
...
```

```
END PROGRAM MAIN
```

```
MODULE WORK_ARRAYS
```

```
  INTEGER N
```

```
  REAL,ALLOTABLE,SAVE :: A(:),B(:,,:),C(:,:::)
```

```
END MODULE WORK_ARRAYS
```

```
SUBROUTINE CONFIGURE_ARRAYS
```

```
  USE WORK_ARRAYS
```

```
  READ *,N
```

```
        ALLOCATE(A(N),B(N,N),C(N,N,2*N))  
  
    END SUBROUTINE CONFIGURE_ARRAYS  
  
SUBROUTINE COMPUTE  
  
    USE WORK_ARRAYS  
  
    ...  
  
END SUBROUTINE COMPUTE
```

#### e) 抽象数据类型和超载运算

可以用模块来自定义抽象数据类型，为此只需在模块中先定义派生类型，再随后定义可在这种类型值上进行的运算即可。例如：

```
MODULE INTERVAL_ARITHMETIC
```

```
TYPE INTERVAL
```

```
    REAL LOWER,UPPER
```

```
END TYPE INTERVAL
```

```
INTERFACE OPERATOR(+)
```

```
    MODULE PROCEDURE COMB_INTERVALS
```

```
END INTERFACE
```

```
INTERFACE OPERATOR(*)
```

```
    MODULE PROCEDURE INTERSECTION_INTERVALS
```

```
END INTERFACE
```

CONTAINS

```
FUNCTION COMB_INTERVALS(A,B)
```

```
    TYPE(INTEGER) COMB_INTERVALS,A,B
```

```
    COMB_INTERVALS%LOWER=MIN(A&LOWER,B%LOWER)
```

```
    COMB_INTERVALS%UPPER=MAX(A&UPPER,B%UPPER)
```

```
END FUNCTION
```

```
FUNCTION INTERSECTION_INTERVALS(A,B)
```

```
    TYPE(INTEGER) INTERSECTION_INTERVALS,A,B
```

```
    INTERSECTION_INTERVALS%LOWER=MAX(A&LOWER,B%LOWER)
```

```
    INTERSECTION_INTERVALS%UPPER=MIN(A&UPPER,B%UPPER)
```

END FUNCTION

END MODULE INTERVAL\_ARITHMETIC

模块中定义了派生类型 **INTERVAL**，它表示一个实数区间，通过接口块说明定义的运算符 ‘+’ 和 ‘\*’ 号，它们分别是求两个区间的并计和交集。

例：求  $\pi$  的计算。 [\[pi.f90\]](#)

### 【作业】

[6.1] 设函数

$$f(x) = \begin{cases} 1 + \sqrt{1 + x^2} & (x < 0) \\ 0 & (x = 0) \\ 1 - \sqrt{1 + x^2} & (x > 0) \end{cases}$$

要求编写成函数子程序及主程序，求  $x$  值分别取值 0.5 和  $\sqrt{15}$ 、 $\sin(0.3)$  时的函数值，并打印输出。

[6.2] 编写函数子程序，求数组  $A_{3 \times 5} \times B_{5 \times 3}$  的积  $C_{3 \times 3}$ 。在主程序中求  $A1_{3 \times 5} \times B1_{5 \times 3}$ 。

[6.3] 使用内部过程，由主程序读入 10 个整数，再进行从小到大排列，然后由主程序输出。

## 第七章:输入输出和文件

### 输入输出编辑

#### 7.1.1 输入输出语句

##### qq) 相关语句

输入输出语句决定了作用在数据上的 I/O 操作。

数据传输语句有：READ，ACCEPT，WRITE，PRINT 和 REWRITE。文件连接、查询和定位语句有：BACKSPACE，CLOSE，DELETE，ENDFILE，INQUIRE，OPEN，REWIND 和 UNLOCK。下表给出了它们的简要描述。另外，内在函数 EOF 可以用来判断在文件当前位置之后是否还有剩余数据。



语句	功能
ACCEPT	输入数据，和格式化顺序 READ 语句类似
BACKSPACE	定位到文件上一个记录开始处
CLOSE	断开和一个单元(文件和设备)的连接
DELETE	从相关文件中删去一条记录
ENDFILE	写一个文件结束记录
INQUIRE	返回一个单元或外部文件的属性
OPEN	使一个单元号和一个文件或设备相连接
PRINT	向星号单元(屏幕)输出数据
READ	从一个文件向 I/O 列表中的项目输入数据
REWIND	重新定位于文件的开头

REWRITE	覆盖当前记录
UNLOCK	释放先前被 READ 语句锁定的相关或顺序文件中的一个记录
WRITE	从一个 I/O 列表中的项目向文件输出数据

一个记录是一个数字或字符的序列。有三种记录形式，即格式化记录、非格式化记录和文件结束记录。一条格式化记录的数据需要进行内部和外部形式间的转化，格式化 I/O 语句有确切的格式说明符或名称列表，只有格式化 I/O 语句才能读写格式化记录。非格式化记录保持其内部形式，而内部形式依赖于处理器。只有非格式化 I/O 语句才能读写非格式化记录。文件结束记录是文件的最后一个记录，可以在顺序文件中用 ENDFILE 语句写出一个文件结束记录。

## rr) WRITE 语句

用于输出的 WRITE 语句的一般形式为：

**WRITE**(**[UNIT=]**单元|\***],[FMT=]**格式说明符|**[NML=]**名称列表组名|\***],[REC=**记录号**],[IOSTAT=**状态变量名**],[ERR=**错误标号**)]** **[I/O 列表]**

上面一般形式中的各项不是同时具有的，根据文件的属性(外部文件(顺序文件(格式化、格式化直接列表、格式化名称列表、非格式化)和直接文件(格式化、非格式化)) and 内部文件)不同而择其项。如果省略 **UNIT=**，则第一个参数必须是“单元”。如果省略了 **FMT=**或 **NML=**，则格式说明符或名称列表组名必须是第二个参数(格式化文件)。其后的几项参数次序可以任意。

**单元**：外部文件时是一个指定设备号的整型表达式，内部文件时是一个字符串、变量、数组、数组元素或非字符数组。

**格式说明符**：对于格式写操作是必需的，非格式写操作时不能有。

**名称列表组名**：如果它被说明，则 **I/O 列表**必须省略。

**错误标号**：在同一个程序单位中的一个可执行语句的标号。如果指定了它，**I/O 错误**将把控制传递给此标号处的语句，省略时取决于状态变量名的存在与否。

**状态变量名**：一个整型变量、数组元素。当无错误时，它返回值为 0，有错误时则返回错误信息号。

**记录号**：一个整数表达式指定要被写的记录序号，仅用于直接文件。文件中的第一条记录的记录号为 1，缺省值为文件中的当前位置。

### ss) PRINT 和 TYPE 语句

PRINT 语句向屏幕输出，TYPE 语句是 PRINT 语句的别名，其规则是：**PRINT {\*|格式}[I/O 列表]**和 **PRINT 名称列表**。

例：下面的语句是等价的：

```
PRINT      '(A11)', 'Abbottsford'
```

```
WRITE (*, '(A11)') 'Abbottsford'
```

```
TYPE      '(A11)', 'Abbottsford'
```

例：一个输出的趣例。 [\[e 711 01.f90\]](#)

## tt) READ 语句

用于输入的 READ 语句的一般形式为：

READ({[UNIT=]单元|\*},{[FMT=]格式说明符|[NML=]名称列表组名|\*}) [, REC=记录号][,IOSTAT=状态变量名][,ERR=错误标号][,END=文件结束标号][,EOR=记录结束标号]) [I/O 列表]

文件结束标号：读到文件结束记录时把控制传递给标号处的语句。

记录结束标号：读完一个记录时把控制传递给标号处的语句。

### 7.1.2 I/O 列表

## g) NAMELIST 语句

名称列表 NAMELIST 语句将一组变量用一个名字相关联。这个组名可以在输入和输出中被引用。其一般形式为：NAMELIST/名称列表组名/变量列表[,]/组名/变量列表]...。

例：integer int1

logical log1

real r1

character(20) char20

namelist/mylist/int1,log1,r1,char20

例: NAMELIST/INPUT/NAME,GRADE,DATE/OUTPUT/TOTAL,NAME

此例的 INPUT 组中包括 NAME,GRADE,DATE 三个变量。OUTPUT 组中包括 TOTAL,NAME 两个变量。

## h) I/O 列表实体

I/O 列表提供将要传输的数据的信息。数据传输语句(READ, WRITE 和 PRINT)需要如何传递数据和传递什么数据的信息。其中传递什么数据由 I/O 列表(iolist)中列出的将要读写的项确定。指定 I/O 列表可以有以下方法:

✦ 无实体。I/O 列表可以是空列表。结果记录要么是零长度,要么是只包含填充字符。如果使用

只有字符串而没有加列表的格式，结果记录将包含这个字符串。

例：WRITE(1,FMT='(2I8)')

WRITE(1," ('string')" )

- ✦ 变量名、数组元素名、派生类型名、派生类型元素名或子字符串名。 [\[e 711 02.f90\]](#)
- ✦ 指定数组名或数组片段。没有下标的数组指的是按列存储的所有数组元素。 [\[e 711 03.f90\]](#)
- ✦ 表达式。WRITE 和 PRINT 语句中的输出列表可以包含表达式。表达式的类型可以是数值、逻辑型、字符型或派生类型。

例：PRINT \*,'(I5)',2\*3

- ✦ 隐 DO 列表。它和一般的 DO 循环类似，起始、终值和增量值决定了循环次数。

例：WRITE(\*,\*) (my\_data(i),I=2,30,3)

除此之外，指定名称列表后可以对其中的所有变量用一个 I/O 语句进行读写。

### 7.1.3 非格式输入输出编辑

I/O 编辑告知系统如何在内存中的变量和外部设备及外部文件之间传递数据。有三种方法，即格式化 I/O，直接列表 I/O，名称列表 I/O。当用户使用格式 I/O 时，必须指定数据在外部设备上如何出现的显示格式或 **FORMAT** 语句。后两种方法中的格式不是由程序员指定的，而是按系统默认的格式进行输入输出。

#### a) 直接列表 I/O

使用直接列表 I/O 可以从一个 I/O 列表中读写数据而不需要格式或 **FORMAT** 语句，它可以用于读写外部或内部文件。输入输出由 I/O 列表中的数据项的数目和类型决定，格式指定符是星号(\*)。例如下例中 **READ** 语句需要两个 2 字节整数，两个 8 字节实数和七个字符。

```
INTEGER(2)  INT1,INT2
```

```
REAL(8)     REAL1,REAL2
```

```
CHARACTER   CHAR(7)
```

```
READ(10,*)  INT1,INT2,REAL1,REAL2,CHAR(7)
```



1) **直接列表输入**：必须提供包含要读变量名称的 I/O 列表。直接列表输入数据记录是一系列由逗号或空格分开的值。在直接列表数据记录中的每个数据项必须要么是某个值要么是 Null(空)值。只有字符串常量可以包含嵌入的空格，两个数字之间的空格被解释为分隔符。与分隔符(逗号，斜杠或其它空格)相邻的空格被忽略。例如 5, 6 /7 等价于 5,6/7。重复输入的值可以通过给该值乘上要重复的次数来实现。例如：对于程序 `REAL R(10); READ(5,*) R` 如果输入 `10*3.14`，则数组 R 的十个元素都被置为 3.14。空值对其映射的变量没有影响，有值的变量仍保留其值，没有值的变量仍保持为空。指定空值的方法是，用无间值的连续分隔符，如 `3.14,,,14` 记录中 3.14 后面两个值为空。还可以用星号格式说明一组 Null，如 `10*` 等价于 10 个空值。斜杠(/)将结束输入流，输入列表中的其它项就像其值为空一样不会产生任何影响。例如：

```
INTEGER I1,I2,I3
```

```
I1=1
```

```
I2=2
```

```
READ(*,*) I1,I2,I3
```

如果用户输入：8, , /9 则 I1 被赋成 8, I2 的输入为空，所以它的值仍为 2，然后因为有斜杠所以输入记录被中止，I3 的值仍为空(0)。

2) **直接列表输出**：程序员写出 I/O 列表，Fortran 系统提供输出格式。如果需要，系统会创建新的记录。为了在打印记录时提供走纸控制，每个输出记录自动以空格作为开头。每种内部数据类型有其默认的输出格式。

## b) 名称列表 I/O

名称列表 I/O 是功能强大的读入数据或向文件(终端)写入数据的方法。通过在名称列表组中确定一个或多个变量，可以用一个单独的输入输出语句读写它们所有的值。

1) **名称列表输出**：名称列表中变量的值通过 WRITE 语句写入一个文件或屏幕，其格式指定符是名称列表名。注意，这里不需要 I/O 列表也不允许出现 I/O 列表。名称列表输出的一般格式为：WRITE(\*,[NML=]namelist)。名称列表输出文件中第一个输出记录是连字符(&)，紧接着是大写的名称列表组名称。然后是一个等号和变量当前的取值。名称列表输出由斜杠结束。变量的输出格式

与直接列表 I/O 中的相同。下面的例子先声明一些变量，然后放入名称列表中并初始化，最后用名称列表 I/O 写入屏幕和文件：[\[e 711 04.f90\]](#)

2) **名称列表输入**：一条名称列表 READ 语句的操作几乎是一个 WRITE 操作的逆操作。READ 语句先从当前位置开始扫描文件(终端或者在磁盘上)，直到找到一个紧接着名称列表组名称的开始符号(连字符&或美元符\$)。在组名称后面必须至少有一个空格或回车来把名称和后面的赋值对区分开，语句找到组名称后会继续扫描寻找给组中的变量赋值的赋值语句。以斜杠、连字符或美元符号为结束符号。END 可以在结束符号&或\$后出现，但不能出现在斜杠之后。[\[e 711 05.f90\]](#)

**赋值规则**：一个赋值对由一个变量名、数组元素或子字符串和后面的等号以及一个或多个值和值分隔符组成。等号前后可以有任意个或没有空格。值分隔符可以是逗号或 tab。如果逗号前面没有值则认为是空值，相应的变量或数组元素原来的值不变。字符串必须用撇号或引号隔开。变量出现的顺序是任意的。同一个变量可以在多个赋值对中出现，变量最终值由最后一个赋值决定。名称列表中的所有变量不一定都要赋值，其中没有出现的或与空值联系的变量保持它们原来的值。在输入文件中的变量名如果不在名称列表组中会产生运行错误。数组赋值是按行顺序进行的，所

赋的值不能多于数组中元素的个数，重复赋值时可以在要赋的值前加上重复系数和星号。如  $7*5$  把 5 赋给数组或变量列表的下七个元素。如果星号后没有值，则认为是空值，相应的元素原来的值不变。

例：matrix=10,50\*25,50\*,-101

matrix(42)=63

第一条语句给元素 0 赋成 10，元素 1 至元素 50 赋成 25，元素 51 至元素 100 为空，元素 101 赋成 -101。第二条语句把元素 42 的值该为 63。

#### 7.1.4 格式化输出编辑

##### a) 格式化 I/O 和 ASSIGN 语句

格式信息包含在格式列表中，并由 PRINT，WRITE 和 READ 语句使用。这些语句可以包含格式列表本身，或包含有格式列表的 FORMAT 语句标号，或包含一个变量以设置编辑列表或语句标号。

**格式列表**是一系列格式描述符，之间用逗号隔开。这些格式描述符描述了将要传输的数据，例如要被读写的数字，数据类型和长度。下面是 **FORMAT** 和 **WRITE** 语句中的格式列表的例子：

例：100 **FORMAT**('A= ',I5,'B= ',F7.2)

例：**WRITE**(\*,'(F8.5,2I3,A20)') **REAL1**,**INT1**,**INT2**," format list example"

**FORMAT** 语句可以出现在程序的任何地方(但必须在 **PROGRAM** 语句之后和 **END** 语句之前，如果在子程序中用 **FORMAT** 语句，则应在子程序定义语句之后)。系统按 **WRITE** 语句中指定的语句标号找到相应的 **FORMAT** 语句，并按所规定的格式对输出数据进行编辑。

格式列表(包括外部的括号)是字符常量，在 **READ** 或 **WRITE** 语句中出现时应被单引号或双引号包括。当格式列表在 **FORMAT** 语句中出现时整个格式列表并不用引号。编辑列表也可以包含另一个格式列表，在格式列表最外层括号中最多允许 8 层嵌套的括号。格式列表由可重复和不可重复的编辑描述符组成。**可重复编辑描述符**描述了数据项，如 2I3 指定编辑描述符 I3 重复两次，这样可以写两个 3 位的整数。**不可重复编辑描述符**可以改变数据格式，如 SP 使正数输出时带有加号。

指定包含格式列表的格式方法是：

**FORMAT 语句标号**：如果指定 I/O 中的 FORMAT 语句标号，在 FORMAT 语句中的格式列表规定了数据的格式。

例：       WRITE(\*,9000) int1,real1(3),char1

          9000 FORMAT(I5,3F4.5,A16)

**整型变量名**：可以使用 ASSIGN 语句把 FORMAT 语句的标号和一个整型变量相联系，随后使用这个变量来引用 FORMAT 语句。ASSIGN 语句的形式为：**ASSIGN 语句标号 TO 整型变量**

例：       ASSIGN 9000 TO MYFMT

          WRITE(\*,MYFMT) iolist

          9000 FORMAT(I5,3F4.5,A16)

**字符表达式或变量**：可以把格式表达式写成字符表达式并在 READ，WRITE 或 PRINT 语句中使用，也可把格式列表赋给变量名。

例：WRITE(\*,'(I5,3F4.5,A16)') iolist

例：CHARACTER(80) MYLIST

MYLIST='(I5,3F4.5,A16)'

WRITE(\*,MYLIST) iolist

**数组或数组元素**：如果把格式列表写成字符表达式并将其赋值给一个数组，可以使用数组作为格式指定符。如果把格式列表写成字符表达式并将其赋值给一个数组元素，可以使用数组元素作为格式指定符。

例：CHARACTER(6) array(3)

DATA array/'(I5',' ,3F4.5',' ,A16)'/

WRITE(\*,array) iolist

例：CHARACTER(80) array(5)

array(2)='(I5,3F4.5,A16)'

WRITE(\*,array(2)) iolist

## b) 输出规则

- 1) 如果输出字符的数目超过了指定的区域宽度或指数宽度，则整个区域将被写为星号(\*)。如果实数小数点后包含的数字位数比区域允许的多，结果会四舍五入。
- 2) 输出时数字是右对齐的，如果输出数字少于规定宽度前面会补上空格。
- 3) 规定复数的格式时需要两个连续的 F，G 和 D 编辑描述符，而且两个编辑符可以不同，第一个指定实部，第二个指定虚部。
- 4) 在可重复编辑描述符之间可以出现不可重复编辑描述符。

## c) 可重复编辑描述符

可重复编辑符告知 Fortran 输入输出系统如何解释 FORMAT 语句中的数据项。它可以根据 I/O 列表中数据项的需要重复任意次。**重复方式**是指定一个非零的无符号整数常量或一个被尖括号括起来



的整型表达式。例如<J+K>I5 就说明 I5 格式的数据项应该重复 J+K 次。

重复编辑描述符有：

- 1) 整数编辑(I)
- 2) 二进制(B)、八进制(O)和十六进制(Z)编辑
- 3) 没有指数的实型编辑(F)
- 4) 有指数的实型编辑(E)
- 5) 双精度实型编辑(D)
- 6) 工程计数法编辑(EN)
- 7) 科学计数法编辑(ES)
- 8) 逻辑编辑(L)
- 9) 字符编辑(A)

## 10)普通编辑(G)

其中 I(整数)、B(二进制)、O(八进制)、F(单精度实型)、E(有指数的实型)、EN(工程计数法实型)、ES(科学计数法实型)、G(普通)和 D(双精度实型)编辑描述符用于数字数据的输入输出。

### d) I 编辑符

I 编辑符用于整型数据的输入输出，其一般形式为：**lw[.m]**。一个数据所占的宽度称为“字段宽度”，**w** 用来指定字段宽度，负数的符号也包含在字段宽度内。如 I3 表示相应的整数的输出占 3 列位置。FORMAT 语句中各编辑符与 WRITE 语句中的各输出项按排列的顺序一一对应。数字在指定的区域内向右端靠齐，如果数字位数比 lw 的 w 小则左边补以空格。如果应输出的列数超过了规定的字段宽度，则不输出有效数据，而在该字段宽度范围内充满星号。m 表示需要输出的最少数字位数，不足添 0，如果应输出的数字超过 m，则按实际应输出的值数输出(但不能超过 w)。

在任何需要整型常量作为编辑符的地方都可以在 FORMAT 语句中指定数值表达式。如果表达式不是整型的，则它将在使用之前被转化为整型。

#### e) F 编辑符

**F 编辑符用于实数的小数形式输出**，其一般形式为：**Fw.d**。其中 **w** 的含义仍为“字段宽度”，包含一位小数点和一位负号(如为负时)，**d** 的含义是输出数据的小数位数。用 **F** 编辑符输出时，由于难以事先确切估计出数据值的大小，输出大的数时容易产生“宽度不够”的错误，输出小的数时会出现丢掉有用数字的情况，这是用 **F** 编辑符所应注意的一个问题。

#### f) 可变格式的输出

可变格式即用尖括号括起来的数值表达式。

例：     WRITE(6,20) INT1

          20 FORMAT(I<MAX(20,5)>)

例：     WRITE(6,FMT=30) REAL1(10), REAL3

          30 FORMAT(<J+K>X,<2\*M>F8.3)

例：用于可变格式输出时的一例。 [\[e 711 06.f90\]](#)

### g) E 编辑符

**E 编辑符用来输出指数形式的实数**，其一般形式为：**Ew.d**。其中 **w** 仍为字段宽度，**d** 为以指数形式出现的数据的数字部分的小数位。例如，若  $A = -746.578$ ，则 `WRITE(*,'(E15.6,E12.4,E9.3,E8.3)')`  
`A,A,A,A` 语句的输出结果为：`-0.746578E+03 -0.7466E+03-.747E+03*****`

小数部分 **d** 位，再加上一个小数点、小数点前的 0、一位负号、指数部分 4 位，需要  $w \geq d+7$ 。用 E 编辑符可以避免“大数印错，小数印丢”的情况，可见用 x 编辑符输出实数的优点是它能容纳任意大小的数据，不必事先估计数值的大小，但输出的是指数形式，看起来不大直观。

小数点前的 0 与 Fortran 编译器有关，有的系统不提供 0。

### h) G 编辑符

**G 编辑符综合了 F 编辑符和 E 编辑符各自的长处**。它可以根据输出的实数大小决定用 F 型格式输出

或 E 型格式输出。当输出大数值或小数值时自动按 E 格式，当输出的数不大不小时用 F 格式。其一般格式为：**Gw.d**。绝对值小于 0.1 或大于  $10^d$  的数用 E 格式，否则用 F 格式，有效位数为 d 位。如  $A=758321.6$ ，用 **G14.7**，由于  $d=7$ ，系统采用 F 格式输出： $758321.6$ 。而当  $A=75.83216E+06$  时，用 E 格式输出： $0.7583216E+08$ 。

注意用 F 格式输出时，不是按 **Fw.d** 格式输出，d 规定的不是小数位数而是打印出来的全部数字的位数，小数位数根据数值大小和 d 的大小而定，这样做的目的是使数的整数部分能全部保留，而适当截去小数部分，以保证输出数值的正确性。

#### i) D 编辑符

**D 编辑符用于双精度数据的输出**。其一般形式为：**Dw.d**。使用方法与 E 编辑符相仿，只是把字母“E”换成“D”。F 编辑符也可用于双精度数据的输出，和用于实型数据输出相似。

#### j) L 编辑符

**L 编辑符用于逻辑型数据的输出。**其一般形式为：**Lw**。对值为“真”的，在输出时打印一个字母 T，“假”则以一个字母 F 表示。**w>1** 时左边补空格。逻辑常量(.TRUE.或.FALSE.)可以直接输出。

#### k) A 编辑符

**A 编辑符用于字符型数据的输出。**其一般形式为：**A[w]**。**w** 为字段宽度。如果待输出的字符串的长度大于 **w**，则左边补空格，否则只输出最左边的 **w** 个字符。用 **A** 时按字符串定义的长度输出。

#### 1) B、O、Z 编辑符

**二进制(B)、八进制(O)和十六进制(Z)**是 F90 新增的编辑描述符，用于整数、实数和逻辑量的输出。其一般形式为：**Bw[.m]**，**Ow[.m]**，**Zw[.m]**。**w** 为字段宽度。**m** 表示需要输出的最少数字位数，缺省值为 1。如果输出少于指定的宽度，则左边补空格。但对于二进制数，如果以 0 填补可读性会更好一些，例如 **O0010101** 显示了 **I0101** 所有的 8 位。可以使用类似 **B8.8**，**B16.16**，**B32.32** 等的格式强迫开始以 0 填补。

编辑描述符不包含小数点或负号。相应进制的数据只能由允许的数字组成：**B** 描述符允许 **0** 和 **1**，**O** 描述符允许 **0-7**，**Z** 描述符允许 **0-9** 和 **A-F**。它们都可用于整型、字符型、实型或逻辑型。因为没有负号，所以 **B**，**O** 和 **Z** 的负值应根据所使用编码转换来表示。对二进制、八进制和十六进制的译码，尤其是负数的译码是和 **CPU** 有关的。使用 **B**，**O** 和 **Z** 编辑描述符和以 **B**，**O** 和 **Z** 存储数据的程序可能不能直接在计算机之间移植。

使用 **B**，**O** 和 **Z** 编辑可以在二进制、八进制和十六进制形式的外部数据和内部数值表示之间互相转换。内部数据的每个字节对应 **8** 位二进制字符，**3** 个八进制字符和 **2** 个十六进制字符。例如，十进制 **255** 的输出就是二进制字符 **11111111**、八进制字符 **377** 和十六进制字符 **FF**。类似地，一个数据存储中的 **INTEGER(4)** 输出时是 **32** 个二进制字符、**12** 个八进制字符或 **8** 个十六进制字符。**w** 被忽略时，缺省区域宽度为  $8*n$  个二进制字符， $3*n$  个八进制字符或  $2*n$  个十六进制字符。**n** 是 **I / O** 列表中的数据项的字节长度。例如，一个为 **INTEGER(2)** 的值是由 **4** 个十六进制字符表示的。

[\[e 711 07.f90\]](#)

m) **EN、ES 编辑符**

工程计数法(EN)和科学计数法(ES)也是 F90 新增的编辑描述符。EN 的一般形式为：ENw.d[Ee]。EN 和 E 编辑描述符基本类似，区别在于 EN 输出数据的非指数部分的绝对值强制在 1 到 1000 的范围内，且指数可以被 3 整除。包括指数的区域的宽度是 w 个字符，小数点后 d 个字符，指数宽度 e 是可选的。ES 的一般形式为：ESw.d[Ee]。它和 E 编辑描述符也基本类似，区别在于 ES 输出数据的非指数部分的绝对值强制在 1 到 10 的范围内，而非 E 的 0 到 1 的范围。[\[e 711 08.f90\]](#)

n) 不可重复编辑描述符

不可重复编辑描述符可以改变解释重复编辑符的方式，还可以改变完成输入输出的方式。

形式	名称	用途	可否用于输	可否用于输



			入	出
‘ ’ 或” ”	撇号编辑	传递 <b>string</b> 到输出单元	否	是
nH	Hollerith 编辑	传递下 n 个字符到输出单元	否	是
Q	字符计数 编辑	返回记录中剩余字符的数目	是	否
T,TL,TR	位置编辑 (Tab)	指定记录的位置	是	是
nX	位置编辑	指定记录的位置	是	是
SP,SS,S	可选加号	控制加号的输出	否	是

	的编辑			
/	斜杠编辑	指向下一个记录或写记录结束符	是	是
\	反斜杠编辑	延续相同的记录	否	是
\$	美元符号编辑	延续相同的记录	否	是
:	格式控制结束	如果 I/O 列表中没有其它记录则结束语句	否	是
kP	指数比例编辑	设置后面的 F 和 E 编辑符的指数比例	是	是

BN,BZ	空格解释	指定对数值空格的解释	是	否
-------	------	------------	---	---

用来分隔列表项的逗号在下列不可重复编辑符前面或后面时可以忽略：

- 1) 在 P 编辑符和紧接着的 F、E、EN、ES、D 或 G 编辑符之间，例如：I3， 2PF8.6， 4F4.3。
- 2) 在撇号、双引号、反斜杠、美元符号或冒号编辑符前后，例如：A14， A35， I5\$。
- 3) 当可选的重复系数没有出现时，在斜杠编辑符之前；所有情况时在斜杠之后。
- 4) 在 nH 或 X 编辑符之后，例如：2I3， 8HF5.3,A12。

#### o) 撇号编辑符

撇号编辑符(单撇号'或双撇号")用来插入所需的字符串，如 WRITE(\*,'(1X,'I=',I3,'J=',I4)') I,J。如果需要输出的字符包括单撇号，则用两个连续的单撇号代表一个被输出的撇号(撇号编辑符为单撇号时)或用双撇号的编辑符，如 WRITE(\*,'('I' 'm a boy')')或 WRITE(\*,'(" I'm a boy" )')。

#### p) H 编辑符

**H 编辑符**它用来输出字符常量，其一般形式为：**nH 字符串**。n 为字符串中字符个数。它的作用与撇号编辑符相似。例如，上面用撇号编辑符的例子也可用 H 编辑符：`WRITE(*,'(9Hl'm a boy)')`。用 H 编辑符必须准确地数出字符串中字符(包括符号,.)的个数，数错了就会造成错误。因此不建议使用 H 编辑符，而应该用 A 编辑符和撇号编辑符来输出字符串。F77 之所以保留 H 编辑符主要是为了与 F66 兼容，但在 F95 中已被废除。

#### q) X 编辑符

**X 编辑符**用来在输出时产生空格。没有空格的输出时数据是连成一片的，难以区分开，为此需要插入空格。它的一般形式为：**nX**，n 为插入的空格数，如 `WRITE(*,'(1X,'I=',I3,5X,'J=',I4)')` I,J 在数据 I 和字符串 'J=' 之间插入 5 个空格。注意第一项中的 1X，在行式打印机上可作为纵向走纸控制符，但在输出到文件和屏幕时，按 Visual Fortran 的默认编译它仅仅为空格一格。

#### r) 纵向控制符

在把格式记录中的信息传送到打印设备上(打印机或终端)时，格式说明中的第一个字符不被印出，这个字符作为纵向间隔控制标志，称为纵向控制符。它们的功能在下表中列出。

格式说明的首字符	纵向间隔控制功能	常用形式
(空格)	移到下一行开头	1X, ' ', 1H
0(数字 0)	移到下面第二行开头	'0', 1H0
1(数字 1)	移到下一页第一行开头	'1', 1H1

+(加号)	移到当前行开头	'+', 1H+
其它字符	移到下一行开头	(非标准规定)

要使这些功能在 Visual Fortran 上实现，必须按以下步骤修改默认值：对于输出到终端的情形，在菜单选项中 **Project -> Setting -> Fortran -> Compatibility** 选取 **Enable VMS Compatibility** 项。对于输出到文件的情形，在打开文件的 **OPEN** 语句中加上说明项 **CARRIAGECONTROL='FORTRAN'**。这时每行记录的第一个字符被当作控制符，可能产生输出的数字或字符被吃掉的情形。另外重叠印刷功能仅对于行式打印机有效，对于终端和文件的输出其效果是覆盖。

#### s) 斜杠编辑符

斜杠(/)编辑符的作用是结束记录在本行的输出并从下一行开始输出下一个记录。如果有两个连续

的斜杠，相当于增加一个空行输出。如果在编辑符的最后出现斜杠，也是再输出一个空行。用  $n$  个连续的斜杠，可以达到输出  $n-1$  个空行的效果。如 `WRITE(*,'(I3,I4/I1,I2//3F8.2/)' ) I,J,M,N,X,Y,Z` 的输出第一行为 `I,J` 的值，第二行为 `M,N` 的值，第三行为空行，第四行是 `X,Y,Z` 的值，第五行为空行。

#### t) 反斜杠编辑符和美元编辑符

反斜杠(`\`)编辑符和美元(`$`)编辑符的作用相同，都是在输出一个记录行后取消回车符。常用于输出字符串与输入数据显示于屏幕同一行的情形。

例： `Write(*,'(" Please Enter Your Age =" , $)')`

`Read(*,*) My_age`

当屏幕上输出字符串 `Please Enter Your Age =`后没有换行，`My_age` 的数值可紧接在`=`号后输入。

#### u) T, TL, TR 编辑符

位置编辑符(`T,TL,TR`)在用于输出时，指出将要输出到记录上的下一个字符的位置。它们的一般形式

为：**Tn**，**TLn**，**TRn**。n 是非零正整数。

**T** 指明记录相对于左 **Tab** 端的位置，记录上的下一个字符输出第 n 个字符的位置上。对于行式打印输出，因为记录的第一个字符作为纵向控制符不被打印，所以 **Tn** 是定位在打印记录的第 n-1 个字符的位置上。在这个位置之前若没有字符输出，则填满空格。

**TL** 用于输出时，指明把记录上的下一个字符输出到从当前位置向左移 n 个字符的位置上。如果左移已到记录的第一列，则不再向左移，即向左移至多回退到第一列。**TR** 用于输出时，指明把记录上的下一个字符传输到从当前位置向右移 n 个字符的位置上。如 `WRITE(*,'(TR10,F6.2,TL16,F6.2)')`  
4.25, -21.46 语句的输出结果是-21.46          4.25。

#### v) 冒号编辑符

当 I/O 列表中没有更多的数据项时，**冒号(:)编辑符使格式控制结束**。此编辑将常常用于 **FORMAT** 语句中没有要输出的数据项时的输出结束。



## w) P 编辑符

**P 编辑符设置比例因子以改变小数点位置**，它用于实数变量编辑描述符如 **F**、**E** 和 **G** 编辑符。其作用范围延续到下一个比例因子的设置处。它的一般形式是：**kP**。**k** 是一有符号整数以指定小数点向左或向右移几位，**k** 取值范围在-128 至 127 之间。在每一个输入输出语句开始时，这个比例因子被初始化为 0。输出时，正 **k** 向右移，负 **k** 向左移(输入时相反)。比例因子对下面格式编辑符的影响：在用 **F** 编辑符输出时，这个要输出的值在显示以前将乘以  $10^k$ 。在用 **E** 编辑符输出时，这个要输出的值的实数部分在显示以前将乘以  $10^k$ ，其指数部分减 **k**。

例：Format	Value	Output
1PE12.3	-270.139	-2.701E+02
1P,E12.2	-270.139	-2.70E+02
-1PE12.2	-270.139	-0.03E+04

例：dimension a(6)

```
a=25.;write(*, "(' ',f8.2,2pf8.2,f8.2)") a
```

其输出是:      25.00 2500.00 2500.00

2500.00 2500.00 2500.00

#### x) SP, SS, S 编辑符

SP, SS 和 S 编辑符在数字输出字段中控制着任选加号(+)的打印。SP 在其后所有正数值域的位置输出加号, SS 为不输出加号, S 重新储存 SS 使其后不输出加号。

#### y) 输出格式指定和 I/O 列表

在输出语句执行时, I/O 列表中的每一项都和一个可重复编辑符联系(I/O 列表中的复型数据需要两个编辑符), 非重复编辑符不和 I/O 列表中的数据项联系。如果 I/O 列表包含一个或多个数据项, 则在格式指定时至少有一个可重复编辑符。空的编辑指定()只能用在 I/O 列表没有数据项的情况。一条编辑指定为空的 FORMAT()格式 WRITE 语句输出的是回车换行。

在格式输入输出过程中，格式控制器从左向右扫描格式数据项。下面列出了格式控制器可能碰到的具体情况及相应的解释：

- 1) 如果 I/O 列表中出现了可重复编辑符和相应的数据项，该数据项和编辑符是互相联系的，该数据项的输出会在编辑符的格式控制下执行。如果没有相应的数据项格式控制器将中止输出，即多余的编辑符无效。
- 2) 如果 I/O 列表中项数多于格式说明中的可重复编辑符个数，即 **WRITE** 语句中的输出项表列中还有未输出的元素，而格式说明中的编辑符已用完，则重新使用该格式说明，但从下一行开始产生一个新记录。
- 3) 如果在格式说明中包含有重复使用的编辑符组，则当格式说明用完后再重新使用时，只有最右面的一个编辑符组(包括其重复系数)和它右面的编辑符被重复使用。
- 4) 遇格式说明的右括号(即最后面一个括号)或斜杠“/”时，结束本记录的输出，但不意味停止全部输出。只要 I/O 列表中还有未输出的量，将重复使用格式说明或按斜杠右面的格式说明组织输出。

右括号与斜杠的不同是：当扫描到右括号而列表中已无数据项时，输出即告结束。而斜杠只表示结束本行输出，即使此时已无输出变量要输出，输出并未停止，它会重新开始一个新记录，直到遇到右括号或非重复编辑符为止。

5) 如果出现冒号编辑符(中止格式控制)且 I/O 列表中没有其它项，则格式控制器将中止输入输出。

### 7.1.5 格式化输入编辑

#### a) 输入规则

1) 输入时，全部是空格的区域始终解释为 O。拖后空格和分散空格的解释由 BN 和 BZ 两个编辑描述符和 OPEN 语句中 BLANK=选项来控制。正号(+)可选的，但对指数例外。

2) F、E、EN、ES、G 和 D 编辑下的输入中，输入区域的显式小数点将覆盖编辑描述符中对小数点位置的指定。例如，`READ(*,'(F4.2)')` x 当输入 123.4 时，`x=123.4` 而非按格式指定的 23.40。

3) 当用 I、B、O、Z、F、E、EN、ES、G、D 或 L 编辑描述符时，输入区域可以包含逗号，表示结

束该区域。下一个区域开始于逗号后的第一个字符。丢掉的字符没有意义。但在使用如 T、TL、TR 或 nX 等显式编辑描述符时不能使用这个性质，因为这样将会改变数据中的字符位置。

### b) 整数输入

整数输入用 I 编辑符。Iw 中的 w 规定输入数据所占的列数。在规定的字段宽度内，空格按零处理 (BZ)，除非用 BN 另行解释。因此必须使输入的数据在规定的字段宽度内向右端对齐。符号也包括在字段宽度内。

### c) 实数、复数和双精度数的输入

实数用 F 编辑符输入：当输入数据不带小数点，由系统按指定格式自动加上小数点。输入数据可以自带小数点，如果 Fw.d 指定的小数点位置与输入数据的小数点位置不一致，按“输入数据自带小数点优先”原则。

输入实数时可以任选 F、E、EN、ES、G 编辑符，输入数据的形式可以是小数形式或指数形式。如

果用指数形式输入数据而其数字部分不带小数点，则按照 **Fw.d**(或 **Ew.d**)中 **d** 的值对其加上小数点。

复数输入时，按两个实数输入，双精度数用 **D** 编辑符输入。

#### d) 逻辑型和字符型数据的输入

逻辑型数据用 **L** 编辑符输入。输入的数据可以是 **.TRUE.**(真)或 **.FALSE.**(假)，也可以是以 **T** 或 **F** 字母开头的任何字符串。

字符型数据的输入用 **A** 编辑符。不指定字段宽度 **w** 则自动按字符变量的定义的长度截取所需字符。

#### e) **BN**、**BZ** 编辑符

**BN**，**BZ** 编辑符在数字输入字段中控制着空格的解释。**BN** 编辑符忽略数字输入字段中内嵌和后续空格，使格式控制器仅使用字段上的所有非空格字符，并使它们向右对齐。例如，对于 **READ(\*,'(BN,I6)')** **n**，输入以下 3 个值后按回车键：

123

123

123 456

READ 语句将这三个输入值都理解为 123。因为只有前 6 个字符才作为 n 的数值，且 123 前后的空格都被忽略了。

BZ 编辑符使结尾空格符和分散空格符为零，而开头空格符仍为零。例如，对于 READ(\*,'(I6)') n，当输入 123 后按回车键，则 n 值为 123000。

#### f) Q 编辑符和可变格式输入

Q 编辑描述符返回当前输入记录中剩余的字符数。对应的 I/O 列表中的数据项必须是整型或逻辑型的。

例：READ(4,'(E15.7,I4,Q,(80A1))')XRAY, KK, NCHRS, (ICHR(I), I=1, NCHRS)

输入语句首先读出前面 XAY 和 KK 的变量值。记录中后面剩余的字符数赋值给 NCHRS，然后字符数

组 ICHAR 读入的字符数即为准确的待读字符数。通过将 Q 作为格式说明的第一项，可以确定待输入记录的长度。

注意由 Q 返回的长度是记录左边的字符串的长度，不是实数，或整数或其他数据类型的长度。一旦由 Q 读出长度，此值可以立即使用，或者用于可变格式表达式中。下面是可变格式输入的例子：

例：INTEGER width, value

```
width=2
```

```
READ (*,10) width, value
```

```
10 FORMAT(I1, I <width>)
```

```
PRINT *, value
```

```
END
```

当输入 3123 时，打印输出值是 123 而非 12。



### g) 输入格式指定和 I/O 列表

一条编辑指定为空的 `FORMAT()` 格式 `READ` 语句将跳过相邻的下一个记录，除非输入输出设置成 `ADVANCE='NO'`，这时文件位置将保持不变。记录中的字符如果少于编辑符指定的长度，在右侧会填以空格，除非在 `OPEN` 语中指定 `PAD='NO'`。用户输入的空格的解释取决于空格编辑描符(`BN` 或 `BZ`)的作用或 `OPEN` 语句中的 `BLANK=` 选项。`BN` 和 `BZ` 的优先级比 `BLANK=` 选项要高。

## 文件的存取

### 7.2.1 逻辑设备和文件

#### uu) 逻辑设备

在 `Fortran` 中对文件和外部设备的操作都是通过逻辑设备进行的。在对文件和外部设备进行操作之前，都要把它们连接到逻辑设备上。内部文件的设备描述符和外部文件的设备描述符是不同的：

内部文件用一个字符变量或其它变量名来描述，外部文件用 **OPEN** 语句打开文件时的数字(单元号)作为文件的描述，或是用默认的设备单元号包括星号(\*)作为文件的描述。对于一个设备描述符，不能同时连接一个以上文件同样，一个文件也不能同时与一个以上设备描述符连接。在 **F90** 中，可以用 **OPEN** 语句打开一个已经打开的文件，但是通过这种方式打开的文件只能对这个文件的输入输出属性选项进行修改，而不能对其进行输入输出操作。

除下列三种情况，必须在每个输入输出语句中使用设备描述符：

1) 使用 **PRINT** 进行输出，**PRINT** 语句的输出是把数据输出到标准输出设备上(单元号 6，即屏幕)。

例：**PRINT I**

2) 只包含一个 I/O 列表和格式描述符的 **READ** 语句以及名称列表 **READ** 语句，其形式为：**READ** 格式描述符 [,I/O 列表]和 **READ** 名称列表，它是从标准输入设备上(单元号 5，即键盘)输入数据的。

3) 对文件按文件名进行的 **INQUIRE** 操作，这时查询的参数是文件名，而不是连接着文件的设备号。

## vv) 外部文件

根据所操作的文件是否在内存中可以把文件分为内部文件和外部文件。当把内存中的数据记录到磁盘的文件中或输入输出到其他外部设备如打印机、显示器、键盘上时，被称为外部文件。连接着一个外部文件的设备描述必须是一整型表达式或是星号(\*),其整型表达式的取值范围在-32768到 32767 之间。

例：OPEN(UNIT=10,FILE=' output.dat' )

WRITE(10,'(A)') 'how are you?'

在 Fortran 中有 4 个预定义的外部文件(设备):

设备号	连接的设备
星号(*)	总是键盘和显示器
0	缺省状态下是键盘和显示器

5	缺省状态下是键盘
6	缺省状态下是显示器

不能关闭星号设备号。设备号 0, 5, 6 可以通过 **OPEN** 语句连接到其他文件上，当在程序中关闭设备号是 0, 5, 6 的外部文件后，如果下次还要使用这些设备号进行输入输出操作，这些设备号将自动连接到它们各自的缺省的设备上去。

## ww) 内部文件

内存中的数据也可以像磁盘上的文件一样进行操作。把连接到设备描述符上的、进行与文件相类似的操作的一块内存中的数据称为内部文件。连接着一个内部文件的设备描述符是一个字符串或是一个字符数组。使用内部文件的规则是：对内部文件只能使用格式化的输入输出操作，包括用格式描述符和直接列表来限定格式的输入输出操作。只能用 **READ** 和 **WRITE** 语句对内部文件进行操作，不能用文件连接(**OPEN**)、文件指针位置的设置(**REWIND**, **BACKSPACE**)或是文件属性查询(**INQUIRE**)语句。

利用内部文件及输入输出系统的格式化功能，可以实现数据在外部的字符表示和在内存表示之间的转换。即可以通过从一个内部文件中读取数据到一个变量或数组中实现 ASCII 的存储方式到数字、逻辑或字符等内存存储方式的转换，或者是通过把一个变量写到内部文件中，实现内存存储方式到 ASCII 存储方式的转换。

在进行内部文件的写操作时，当写入的内容长度小于内部文件的一个记录的长度时，记录中剩余的空间被空格填满；大于时多余的数据将被删除。在进行内部文件的读操作时，当内部文件的一个记录的长度小于希望读取的数据长度时，没有读取到数据的变量填为空格；大于时多余的内部文件数据被删除。

例：CHARACTER(10) str

CHARACTER(14) fname

str = " 1    2    3"

READ(str,\*) n1,n2,n3    !直接列表 READ 语句设置 n1=1, n2=2, n3=3

i=4

WRITE(fname,200) i      !格式化 WRITE 语句设置 fname ='FM004.DAT'.

200 FORMAT('FM',I3.3,'.DAT')

### 7.2.2 外部文件分类

Fortran 支持两种文件的访问方式(顺序访问和直接访问)和三种文件的结构(有格式、无格式、二进制)。顺序访问或直接访问可以用于这三种结构的文件进行的每一种。因此,共有 6 种文件类型。

#### a) 格式化文件

在格式化文件中,记录数据内容的记录是以 ASCII 字符的方式存在的,每一条记录是以 ASCII 码中的回车符 CR(0D)加换行符 LF(0A)来结束的,可以用文本编辑软件打开格式文件并直接看懂其内容。即存放在文件中的数字就是平时所看到的数字字符,字符串也就是平时所看到的字符串。而若用文本编辑软件打开无格式文件或二进制文件,看到的则是一些十六进制的字符。因此如果要使文

件中的内容可以被人直接看懂，应用格式文件。

**OPEN** 语句默认的打开文件是格式文件，也可以使用 **FORM= 'FORMATTED'** 设置项的 **OPEN** 语句以明确文件是格式化的。

## b) 无格式文件

无格式文件由一系列物理块组成的记录组成，所存储的记录序列的存放方式与其在内存中的存放非常相似，所以在输入输出时几乎不需作转化。由于去掉了格式控制，与有格式文件相比，在使用数据信息时所做的处理更简洁更迅速；同样也是这个原因使得无格式文件中即使存放着数字，也不能用文本编辑软件打开并看到它们。

使用无格式文件之前，应该先打开或建立一个无格式文件。通过带有 **FORM= 'UNFORMATTED'** 设置项的 **OPEN** 语句来打开或建立一个文件，或者用省略 **FORM** 选项的 **OPEN** 语句来打开或建立一个无格式直接访问文件。

### c) 二进制文件

二进制文件是处理最快、最简洁的一种文件，也是最紧凑的存储格式，适合于大批量数据的存储。在程序中可以用带有 **FORM=' BINARY'** 选项的 **OPEN** 语句来打开或建立二进制文件。

### d) 顺序访问文件

存放在顺序文件中的数据必须一个记录接一个记录地按顺序被访问。也就是说，程序中要读写第 **N** 条记录时，必须至少已对前面的 **N-1** 记录进行过读操作。在输入输出操作中，有些方法只有在顺序访问的文件中才可能实现，包括低级的输入输出操作、直接列表和名称列表输入输出操作。内部文件也必须使用顺序文件。键盘、显示器和打印机等顺序访问的外部设备必须连接成顺序文件。**OPEN** 语句默认的打开文件是顺序文件，也可以使用 **ACCESS= 'SEQUENTIAL'** 设置项的 **OPEN** 语句以明确文件是顺序文件。

当对顺序文件进行输出时，在 **OPEN** 语句之后总是把 **WRITE** 语句输出的记录作为文件的开头，当前的 **WRITE** 语句所输出的记录总作为文件的最后一条记录。如果所写的顺序文件是一个已经存在



的文件，则文件原来的内容将全部丢失。对一个顺序文件在读操作后立即进行写操作，则当前写的这个记录就成了文件的最后一个记录，在写操作后立即进行读操作，则必然遇到文件结束记录。如对同一个文件写操作后要进行读操作时，必须使用能够对文件的指针进行重定位的语句重新设置文件的指针。

#### e) 直接访问文件

存放在直接访问文件中的记录可以以任意顺序进行读写操作。文件中的记录从 1 开始连续编号，记录的长度是通过 **OPEN** 语句中的 **RECL** 选项来描述的。直接文件中的记录是通过指定要访问的记录号来实现的。因此，如果想要实现数据的随机访问可以使用直接访问文件。直接文件应用的一个最常见的实例就是数据库。在程序中可以用带有 **ACCESS=' DIRECT'** 设置项的 **OPEN** 语句来打开或建立直接文件。

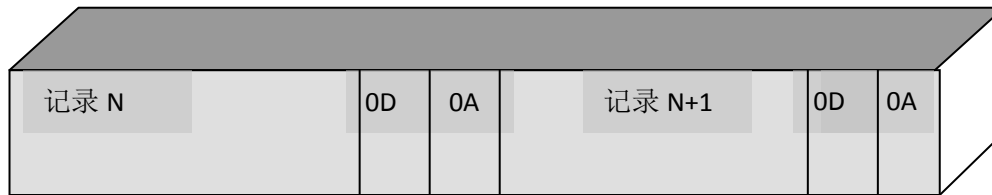
直接文件中的每个记录的长度必须相等。如果实际输出的记录长度不等，则应取输出的所有记录中最大的长度作为每个记录的长度。如果使用一个老文件，在 **OPEN** 语句中说明的记录的长度必

须与实际的记录长度一致。特别要注意尾随的空格符，它占一个字节。回车换行符不计入记录长度。用直接方式建立的文件可以使用顺序方式打开进行读操作。用顺序方式建立的文件，只要记录长度相等，也可以用直接方式打开进行读操作。

### 7.2.3 文件记录的存取

#### i) 格式化顺序文件

一个格式化文件是一个由按顺序写到文件中的有格式记录序列组成的，当要对文件进行读操作时，读取的顺序就是记录在文件中的存放顺序。文件中记录的长度不一定相同，记录也可以是空的。记录用回车符(0DH)和换行符(0AH)分开。



例： OPEN(3,FILE='FSEQ')

! FSEQ is a formatted sequential file by default.

WRITE(3,'(A,I3)') 'RECORD',1

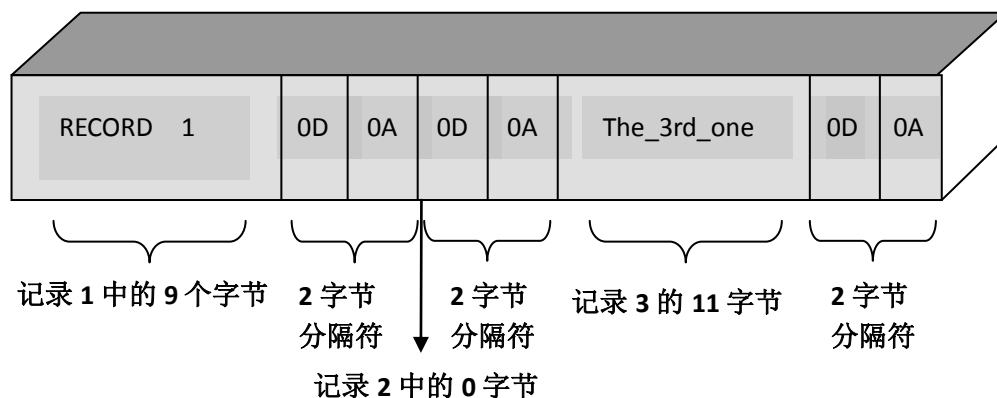
WRITE(3,'()')

WRITE(3,'(A11)') 'The 3rd one'

CLOSE(3)

END

[\[e 7 2 1.f90\]](#)



## j) 格式化直接文件

在格式化直接文件中，所有记录的长度都相同并且可以以任意顺序读写。记录的长度由 **OPEN** 语句中的 **RECL=**选项指定，该长度应该大于或等于最长的记录中的字节数。**CR** 和 **LF** 是分隔符，不包括在 **RECL** 中。一旦某个直接访问记录被写入就不能再删除它，但可以覆盖这个记录。在输出到一个格式化直接文件时如果数据没有占满一个记录，则编译系统将在剩下的位置上补以空格，保证文件只包含长度相同的完整的记录。从文件中读数据时，当 **I/O** 列表或格式描述符中要读取的数据

多于记录中的数据时，编译器也会以空格填充未读数据的变量。可以通过在打开文件的 **OPEN** 语句中设置 **PAD=NO** 来避免填补空格，此时输入记录必须有和输入列表和格式描述符所要求的一样的数据，否则会产生错误。**PAD=NO** 对输出没有影响。

例： **OPEN(3,FILE='FDIR',FORM='FORMATTED',ACCESS='DIRECT',RECL=10)**

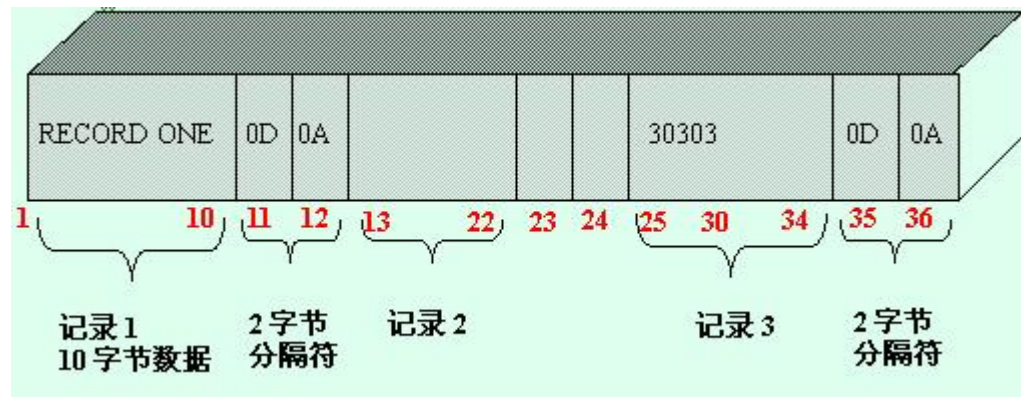
**WRITE(3,'(A10)',REC=1) 'RECORD ONE'**

**WRITE(3,'(I5)',REC=3) 30303**

**CLOSE(3)**

**END**

[\[e 7 2 2.f90\]](#)

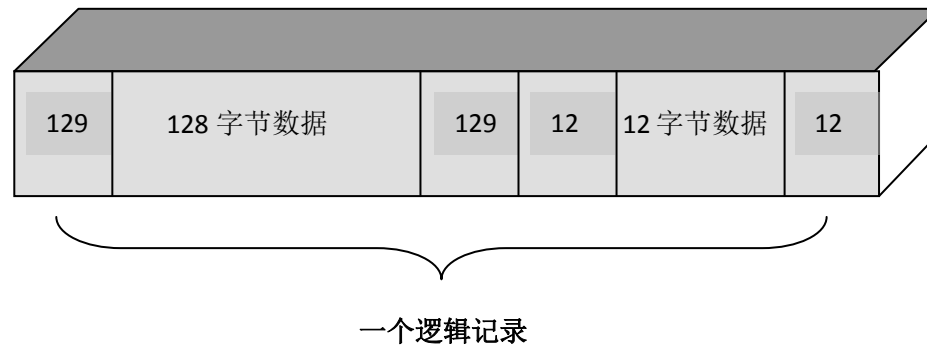


### k) 无格式顺序文件

无格式顺序文件中记录的长度可以不同，文件以 130 或少于 130 字节为一个物理块进行组织。每个物理块包含着输入到文件中的数据(最多 128 字节)，编译系统在物理块之间加入两个 1 字节长的长度值以说明每个记录的起始和结束位置。

一个逻辑记录包含一个或多个物理块，其大小可在程序中指定，编译系统会相应地使用需要数量的物理块。当创建一个包含多个物理块的逻辑记录时，编译系统把长度值置为 129 以表示当前物

理块的数据和下一个物理块相连接。例如，如果写入了 140 字节的数据，则逻辑记录的结构如图



所示。

无格式顺序文件中的第一个和最后一个字节是保留字节：第一个字节的值为 75，最后一个字节的值为 130。Fortran 使用这些字节作为错误检测和文件结束的判断。

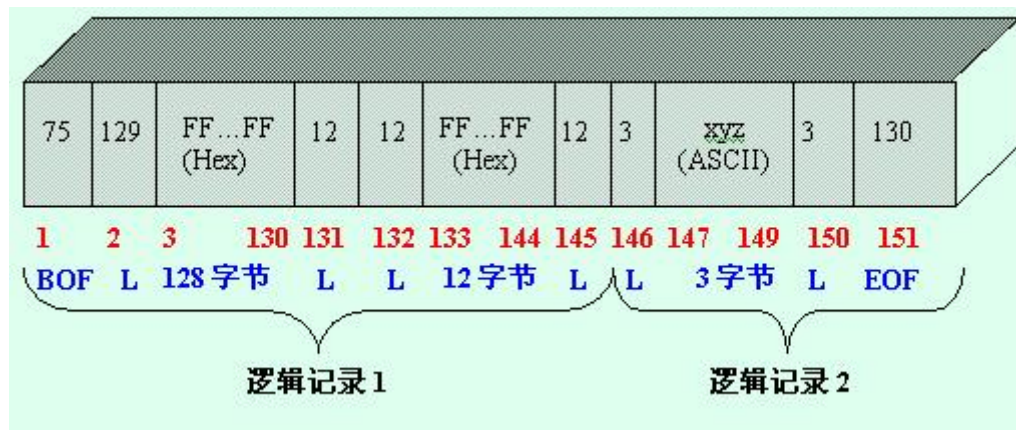
例：CHARACTER xyz(3)

INTEGER(4) idata(35)

```
DATA idata/35*-1/,xyz/'x','y','z'/  
  
OPEN(3,FILE='USEQ',FORM='UNFORMATTED')  
  
WRITE(3) idata  
  
WRITE(3) xyz  
  
CLOSE(3)  
  
END
```

产生的数据文件结构如下，但是在 Visual Fortran 中需要加上编译指定参数/fpscomp，可在 Fortran Compatibility 选项框中选取 I/O Format 项，否则文件的结构是不同的。 [\[e 7 2 3.f90\]](#)





## 1) 无格式直接文件

无格式直接文件是一系列非格式的记录，可以以任意顺序读写记录。记录的长度都相同，由 OPEN 语句中的 RECL=选项指定。没有字节分隔符或其它表示记录结构的字节。

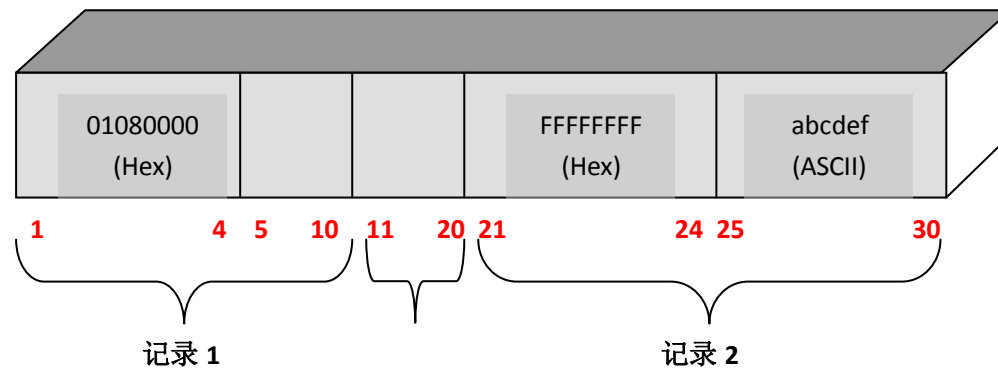
例：OPEN(3,FILE='UDIR',RECL=10,FORM='UNFORMATTED',ACCESS='DIRECT')

WRITE(3,REC=3).TRUE., 'abcdef'

WRITE(3,REC=1) 2049

CLOSE(3)

END



[\[e 7 2 4.f90\]](#)

### m) 二进制顺序文件

二进制顺序文件是一系列按同一顺序和同样二进制数个数来读写的值。其中没有记录边界，没有

说明文件结构的特殊字节。数据读写时长度和形式都不改变。数据记录的长度可以不等。对于任何输入输出数据，内存中的字节序列就是文件中的字节序列。二进制顺序文件是处理最简洁、速度最快的文件。

例：INTEGER bells(4)

CHARACTER(4) wys(3),cvar

DATA bells/4\*7/,cvar/'is'/,wys/'What',' you',' see'/

OPEN(3,FILE='BSEQ',FORM='BINARY')

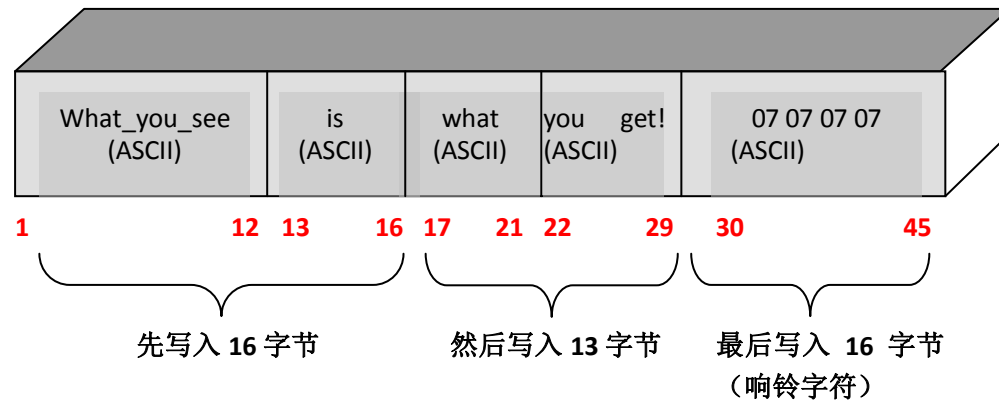
WRITE(3) wys,cvar

WRITE(3) 'what','you get!'

WRITE(3) bells

CLOSE(3)

END



[\[e 7 2 5.f90\]](#)

## n) 二进制直接文件

二进制直接文件存储一系列二进制数记录，它们可以按任何顺序访问。与二进制顺序文件不同的是，这些记录的长度是相等的，由 **OPEN** 语句中的 **RECL=**选项指定。在二进制直接文件中可以写入部分记录，记录中未使用的部分将以未定义数据填充。

在二进制直接文件中可以只使用一条读或写语句来读写到多于一条的记录，而这样的操作在无格式直接文件中将引发错误。在无格式直接文件中所能进行的一切操作在二进制直接文件中都是合法的，另外，在二进制直接文件中提供了一种不依赖于填充二进制数据 0 的对记录的某部分进行读写操作的功能。二进制直接文件是所有 6 类文件中使用最灵活的一类。

例：OPEN(3,FILE='BDIR',RECL=10,FORM='BINARY',ACCESS='DIRECT')

WRITE(3,REC=1) 'abcdefghijklmno'

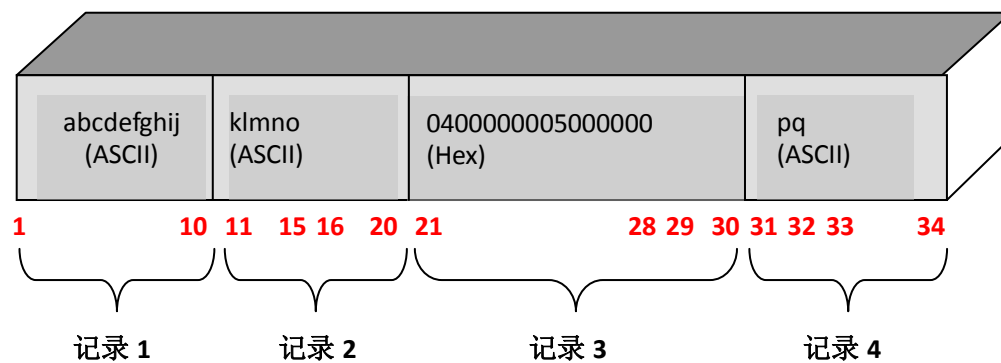
WRITE(3) 4,5

WRITE(3,REC=4) 'pq'

CLOSE(3)

END

[\[e 7 2 6.f90\]](#)



## 7.2.4 文件操作语句

### a) OPEN 语句

OPEN 语句用于把设备号与文件名连接起来，并且对文件的各项性质进行指定。它的一般形式为：

OPEN([UNIT=]unit[,ACCESS=access][,ACTION=action][,BLANK=blanks][,BLOCKSIZE=blocksize][,CARRIAGE CONTROL=carriagecontrol][,DELIM=delim][,ERR=err][,FILE=file][,FORM=form][,IOFOCUS=iofocus][,IOSTAT=iostat][,PAD=pad][,POSITION=position][,RECL=recl][,SHARE=share][,STATUS=status])

其中的各项参数的意义及取值如下：

1) **UNIT**：设备号说明。**unit** 是大于或等于 0 的正整数，设备号说明是 **OPEN** 语句的第一项时可以省略 “**UNIT=**”。

2) **ACCESS**：存取方式说明。**access** 是字符串表达式：

**APPEND** 追加方式

**SEQUENTIAL** 顺序访问方式

**DIRECT** 直接访问方式

当省略此说明项时为顺序访问方式。

3) **ACTION**：描述文件的读写属性。**action** 是字符串表达式：

**READ** 文件为只读方式打开

**WRITE** 文件为只写方式打开

READWRITE 文件为可读写方式打开

当省略此说明项时，文件打开顺序：READWRITE->READ->WRITE。

4) **BLANK**：说明数据格式输入字段中空格的含义。blank 是字符串表达式：

NULL 空格忽略不计，相当于在格式描述符中的 BN 编辑符

ZERO 空格处理成数字 0，相当于 BZ 编辑符

当省略此说明项时为 ZERO。此说明只能用于格式输入。

5) **BLOCKSIZE**：指定以字节为单位的设备缓存的大小，默认值为一 4 字节整数。

6) **CARRIAGECONTROL**：指明处理文件中的第一个字符的方式，其值为字符串表达式：

Fortran 对第一个字符作一般的 Fortran 解释

LIST 指出在文件的每两个记录之间有一个空格

默认状态下，对于连接到打印机和显示器这样的设备，设置值为 Fortran，对于连接到文件的设备，



设置值为 LIST。当 FORM 被设成 UNFORMATTED 和 BINARY 时，其值被忽略。

7) **DELIM**: 指明分隔直接列表或格式化名称列表记录的方式，其值为字符串表达式:

APOSTROPHE 用单撇号(')分隔

QUOTE 用双撇号(")分隔

NONE 不用分隔符

如果在 OPEN 语句中设置了分隔符，则在文件中的单撇号和双撇号都是成对出现的。

8) **ERR**: 出错处理说明。其值是同一程序中的一条语句的标号，当 OPEN 语句执行出错时执行此语句。如果省略该项，则出错时给出出错信息并终止运行。

9) **FILE**: 文件名。file 是一字符串表达式，可以是空、合法的数据文件名字、设备名字或是作为内部文件的变量。在 WinNT/9x 中允许使用长度大于 8 的文件名和长度大于 3 的文件扩展名。省略此项时，编译器将自动产生一个文件名唯一的临时文件，这个临时文件将在结束运行或与文件连接的设备关闭后被删除掉。

10) **FORM**: 记录格式说明。form 是字符串表达式:

**FORMATTED**                      记录按有格式存放。

**UNFORMATTED**                  记录按无格式存放。

当省略此说明项时为: 对顺序文件是有格式的; 对直接文件是无格式的。

11) **IOFUS**: 指出一个新 Quickwin 子窗口是否为活动窗口, 其值为逻辑值。缺省值为真。

12) **IOSTAT**: 出错状态说明。iostat 是一个缺省长度为 4 的整形变量。当执行此 OPEN 语句时系统给变量赋值:

零 没有发生错误

负数 文件结尾

正数 发生错误, 其值视具体计算机系统而定

若省略该项则没有此功能。

13) **PAD**: 从格式化文件中记录的数据少于要读取的数据时, 是否用空格来填充没有从记录中读到数据的变量。**pad** 是字符串表达式:

YES      填充(默认值)

NO    不填充

14) **POSITION**: 指定打开顺序文件的访问位置, **position** 是字符串表达式:

**ASIA** 已被连接的文件的访问位置是固定的, 未被连接的文件的访问位置是文件的开始处。

**REWIND** 把文件的访问位置定在文件的开始处(文件已存在)。

**APPEND** 把文件的访问位置定在文件的末尾处(文件已存在)。

对于一个新文件, 文件的访问位置总是被定在文件的开始处。

15) **RECL**: 记录长度(单位为字节)说明。**recl** 是指定的正整型量或算术表达式, 用来指定直接文件中的每条记录的字节数, 或顺序文件中的记录的最大长度。

16) **SHARE**: 指明当文件打开时是否实现文件的锁定。share 是字符串表达式:

**DENYRW** 动态读写模式。不允许其他的进程打开这个文件。

**DENYWR** 动态写模式。不允许其他的进程以写的方式打开这个文件。

**DENYRD** 动态读模式。不允许其他的进程以读的方式打开这个文件。

**DENYNONE** 默认的非动态模式。允许其他的进程打开这个文件。

17) **STATUS**: 文件状态说明。status 是字符串表达式:

**OLD**                   表示指定的文件是已经存在的老文件。这一状态一般用于读操作，如果用于写操作则重写文件，原文件内容将被覆盖。如果指定的文件并不存在，则系统将给出出错信息。

**NEW**                  表示指定的文件尚不存在。执行 **OPEN** 语句时将在磁盘上建立该文件并使其状态改变为 **OLD**。**NEW** 状态一般用于写操作。如果指定的文件名已经存在将给出出错信息(有的系统不给出信息而是把这个已经存在的文件冲掉使原来的内容不复存在)。

**SCRATCH** 表示与设备号相连接的文件在关闭时将被自动删除。注意：此状态不能与 **FILE** 说明共存，只能用于由计算机系统指定的文件名，使该文件作为程序运行过程中的一个临时性文件。

**REPLACE** 表示替换一个有相同名字的文件，如果没有同名的文件存在，将产生一个新文件。

**UNKNOWN** 表示文件可以是已存在的或不存在的。系统打开文件状态的次序为：**OLO->NEW->**创建新文件。**STATUS** 的设置值只影响磁盘文件，像键盘和显示器这样的设备将忽略这一设置。

若省略该项时默认的状态为 **UNKNOWN**。

## b) **ENDFILE** 语句

**ENDFILE** 语句的功能是在文件上写一条文件结束记录，这时文件定位在结束记录的后面。它的一般形式为：

**ENDFILE{unit | ([UNIT=]unit[,ERR=err][,IOSTAT=iostat])}**

由于用 **ENDFILE** 语句在文件中写入一条结束记录后，文件的指针被定位在结束记录之后，所以若再想向同一个文件中添加更多的记录，就必须使用 **BACKSPACE** 或 **REWIND** 语句对文件进行文件指针定位的操作。在直接访问文件中使用 **ENDFILE** 语句在文件中写入一条结束记录后，新的结束记录后的所有老的记录都将被删除掉。

### c) CLOSE 语句

**CLOSE** 语句解除设备号与文件的连接，又称关闭文件。它的一般形式为：

**CLOSE([UNIT=]unit[,ERR=err][,IOSTAT=iostat][,STATUS | DISPOSE | DISP=status])**

其中除 **STATUS** 以外的各项参数的意义及取值与 **OPEN** 语句中的相同。**STATUS** 是文件关闭后状态说明，其值是一字符串：

**DELETE**    与设备连接的文件不保留，被删除

**KEEP(或 SAVE)**                      与设备号连接的文件保留下来不被删除

PRINT	将文件递交给打印机打印并被保留(仅对顺序文件)
PRINT/DELETE	将文件递交给打印机后被删除
SUBMIT	插入一个进程以执行文件
SUBMIT/DELETE	插入一个进程以执行文件，当插入完成后被删除

默认设置将删除带有 **SCRATCH** 属性的临时文件，对其它文件为 **KEEP**。

在程序中，没有必要显示的进行文件的关闭，一般情况下，当程序退出时将以各个文件的默认状态关闭所有的文件。**CLOSE** 语句不必与 **OPEN** 语句出现存同一程序单元中。

#### d) 文件指针定位语句

**REWIND** 语句：称为反绕语句，它使指定设备号的文件指针指向文件的开头，通常用于顺序文件的操作。它的一般形式为：

**REWIND{unit | ([UNIT=]unit[,ERR=err][,IOSTAT=iostat])**

**BACKSPACE** 语句：称为回退语句，它使指定设备号的文件指针退回一个记录位置，一般用于顺序文件。它的一般形式为：

**BACKSPACE{unit| ([UNIT=]unit[,ERR=err][,IOSTAT=iostat])**

除了以下几种情况外，使用 **BACKSPACE** 语句正好使文件的指针向前移动一条记录：本条记录前再没有记录时文件指针的位置不变；文件指针的位置在一条记录的中间时，文件指针移到本条记录的开始处；本记录的前一记录是文件结束记录时，文件指针移到文件结束记录之前。

### 7.2.5 使用硬件设备

在 Fortran 中标准的输入设备是键盘，标准的输出设备是显示器(控制台)。一般的输入输出语句都是针对标准设备进行操作的，如果想对除键盘和显示器以外的其他的物理设备进行读写操作，就应该把物理设备名描述为文件名，这样就可以像操作文件一样对其进行操作，绝大多数设备名没有扩展名。以下是 WinNT/9x 下的一些设备名。



设备	描述
CON	控制台(即屏幕，标准输出设备)
PRN	打印机
COM1	1#串行通信口
COM2	2#串行通信口
COM3	3#串行通信口
COM4	4#串行通信口
LPT1	1#并行通信口
LPT2	2#并行通信口
LPT3	3#并行通信口
LPT4	4#并行通信口

NUL	空(NLTLL)设备。放弃输出，不包含任何输入
AUX	1#串行通信口
LINE1	1#串行通信口
USER1	标准输出
ERRI	标准错误
CONOUT\$	标准输出
CONIN\$	标准输入

如果使用了这些名字的扩展名，例如 LPT1.TXT，Fortran 将会写入一个文件而不是相应的设备。下面是打开物理设备作为单元的例子：

例：OPEN(UNIT=4,FILE=' PRN' )

例: OPEN(UNIT=7,FILE=' COM2' ,ERR=100)