

## 第5章 行为模式

行为模式涉及到算法和对象间职责的分配。行为模式不仅描述对象或类的模式，还描述它们之间的通信模式。这些模式刻画了在运行时难以跟踪的复杂的控制流。它们将你的注意力从控制流转移到对象间的联系方式上来。

行为类模式使用继承机制在类间分派行为。本章包括两个这样的模式。其中 Template Method (5.10) 较为简单和常用。模板方法是一个算法的抽象定义，它逐步地定义该算法，每一步调用一个抽象操作或一个原语操作，子类定义抽象操作以具体实现该算法。另一种行为类模式是 Interpreter (5.3)。它将一个文法表示为一个类层次，并实现一个解释器作为这些类的实例上的一个操作。

行为对象模式使用对象复合而不是继承。一些行为对象模式描述了一组对等的对象怎样相互协作以完成其中任一个对象都无法单独完成的任务。这里一个重要的问题是对等的对象如何互相了解对方。对等对象可以保持显式的对对方的引用，但那会增加它们的耦合度。在极端情况下，每一个对象都要了解所有其他的对象。Mediator (5.5) 在对等对象间引入一个 mediator 对象以避免这种情况的出现。mediator 提供了松耦合所需的间接性。

Chain of Responsibility (5.1) 提供更松的耦合。它让你通过一条候选对象链隐式的向一个对象发送请求。根据运行时刻情况任一候选者都可以响应相应的请求。候选者的数目是任意的，你可以在运行时刻决定哪些候选者参与到链中。

Observer (5.7) 模式定义并保持对象间的依赖关系。典型的 Observer 的例子是 Smalltalk 中的模型/视图/控制器，其中一旦模型的状态发生变化，模型的所有视图都会得到通知。

其他的行为对象模式常将行为封装在一个对象中并将请求指派给它。Strategy (5.9) 模式将算法封装在对象中，这样可以方便地指定和改变一个对象所使用的算法。Command (5.2) 模式将请求封装在对象中，这样它就可作为参数来传递，也可以被存储在历史列表里，或者以其他方式使用。State (5.8) 模式封装一个对象的状态，使得当这个对象的状态对象变化时，该对象可改变它的行为。Visitor (5.11) 封装分布于多个类之间的行为，而 Iterator (5.4) 则抽象了访问和遍历一个集合中的对象的方式。

### 5.1 CHAIN OF RESPONSIBILITY(职责链)——对象行为型模式

#### 1. 意图

使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。

#### 2. 动机

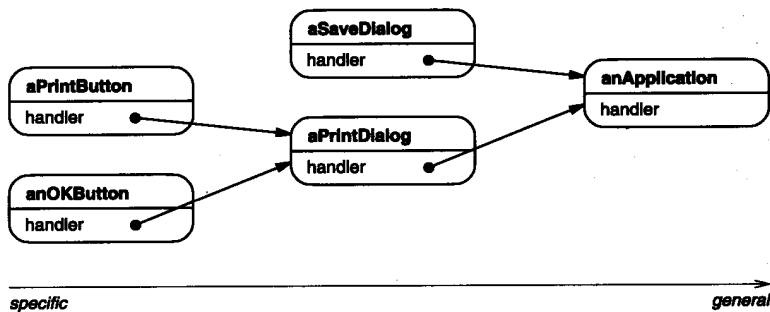
考虑一个图形用户界面中的上下文有关的帮助机制。用户在界面的任一部分上点击就可以得到帮助信息，所提供的帮助依赖于点击的是界面的哪一部分以及其上下文。例如，对话框中的按钮的帮助信息就可能和主窗口中类似的按钮不同。如果对那一部分界面没有特定的帮助信息，那么帮助系统应该显示一个关于当前上下文的较一般的帮助信息——比如说，整个

对话框。

因此很自然地，应根据普遍性 (generality) 即从最特殊到最普遍的顺序来组织帮助信息。而且，很明显，在这些用户界面对象中会有一个对象来处理帮助请求；至于是哪一个对象则取决于上下文以及可用的帮助具体到何种程度。

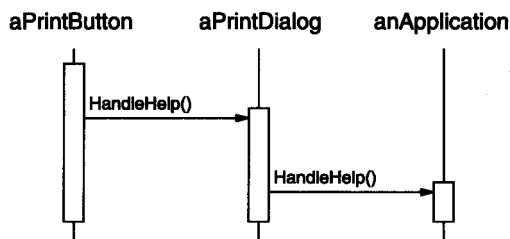
这儿的问题是提交帮助请求的对象 (如按钮) 并不明确知道谁是最终提供帮助的对象。我们要有一种办法将提交帮助请求的对象与可能提供帮助信息的对象解耦 (decouple)。Chain of Responsibility 模式告诉我们应该怎么做。

这一模式的想法是，给多个对象处理一个请求的机会，从而解耦发送者和接受者。该请求沿对象链传递直至其中一个对象处理它，如下图所示。



从第一个对象开始，链中收到请求的对象要么亲自处理它，要么转发给链中的下一个候选者。提交请求的对象并不明确地知道哪一个对象将会处理它——我们说该请求有一个隐式的接收者 (implicit receiver)。

假设用户在一个标有 “Print” 的按钮窗口组件上单击帮助，而该按钮包含在一个 PrintDialog 的实例中，该实例知道它所属的应用对象 (见前面的对象框图)。下面的交互框图 (diagram) 说明了帮助请求怎样沿链传递：

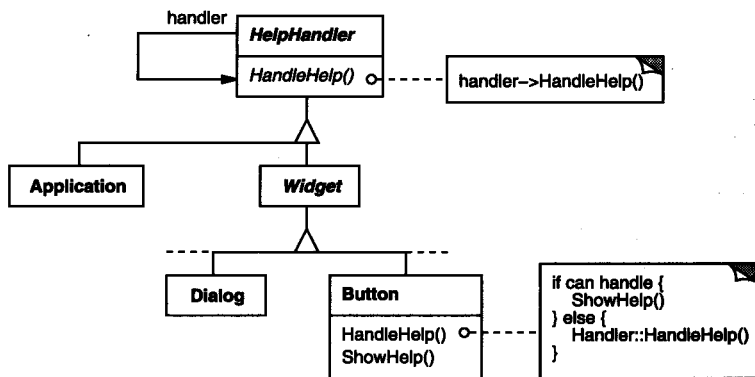


在这个例子中，既不是 aPrintButton 也不是 aPrintDialog 处理该请求；它一直被传递给 anApplication，anApplication 处理它或忽略它。提交请求的客户不直接引用最终响应它的对象。

要沿链转发请求，并保证接收者为隐式的 (implicit)，每个在链上的对象都有一致的处理请求和访问链上后继者的接口。例如，帮助系统可定义一个带有相应的 HandleHelp 操作的 HelpHandler 类。HelpHandler 可为所有候选对象类的父类，或者它可被定义为一个混入 (mixin) 类。这样想处理帮助请求的类就可将 HelpHandler 作为其一个父类，如下页上图所示。

按钮、对话框，和应用类都使用 HelpHandler 操作来处理帮助请求。HelpHandler 的

HandleHelp 操作缺省的是将请求转发给后继。子类可重定义这一操作以在适当的情况下提供帮助；否则它们可使用缺省实现转发该请求。

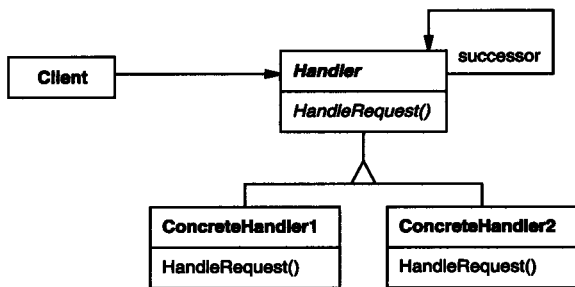


### 3. 适用性

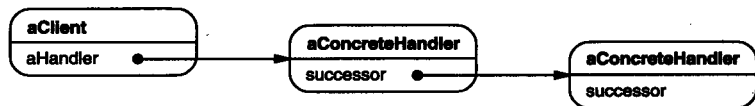
在以下条件下使用Responsibility 链：

- 有多个的对象可以处理一个请求，哪个对象处理该请求运行时刻自动确定。
- 你想在不明确指定接收者的情况下，向多个对象中的一个提交一个请求。
- 可处理一个请求的对象集合应被动态指定。

### 4. 结构



一个典型的对象结构可能如下图所示：



### 5. 参与者

- Handler (如 HelpHandler)
  - 定义一个处理请求的接口。
  - (可选) 实现后继链。
- ConcreteHandler (如 PrintButton 和 PrintDialog)
  - 处理它所负责的请求。
  - 可访问它的后继者。
  - 如果可处理该请求，就处理之；否则将该请求转发给它的后继者。
- Client

— 向链上的具体处理者 (ConcreteHandler) 对象提交请求。

## 6. 协作

- 当客户提交一个请求时，请求沿链传递直至有一个 ConcreteHandler 对象负责处理它。

## 7. 效果

Responsibility 链有下列优点和缺点 (liabilities):

1) 降低耦合度 该模式使得一个对象无需知道是其他哪一个对象处理其请求。对象仅需知道该请求会被“正确”地处理。接收者和发送者都没有对方的明确的信息，且链中的对象不需知道链的结构。

结果是，职责链可简化对象的相互连接。它们仅需保持一个指向其后继者的引用，而不需保持它所有的候选接受者的引用。

2) 增强了给对象指派职责 (Responsibility) 的灵活性 当在对象中分派职责时，职责链给你更多的灵活性。你可以通过在运行时刻对该链进行动态的增加或修改来增加或改变处理一个请求的那些职责。你可以将这种机制与静态的特例化处理对象的继承机制结合起来使用。

3) 不保证被接受 既然一个请求没有明确的接收者，那么就不能保证它一定会被处理——该请求可能一直到链的末端都得不到处理。一个请求也可能因该链没有被正确配置而得不到处理。

## 8. 实现

下面是在职责链模式中要考虑的实现问题：

1) 实现后继者链 有两种方法可以实现后继者链。

a) 定义新的链接 (通常在 Handler 中定义，但也可由 ConcreteHandlers 来定义)。

b) 使用已有的链接。

我们的例子中定义了新的链接，但你常常可使用已有的对象引用来形成后继者链。例如，在一个部分—整体层次结构中，父构件引用可定义一个部件的后继者。窗口组件 (Widget) 结构可能早已有这样的链接。Composite (4.3) 更详细地讨论了父构件引用。

当已有的链接能够支持你所需的链时，完全可以使用它们。这样你不需要明确定义链接，而且可以节省空间。但如果该结构不能反映应用所需的职责链，那么你必须定义额外的链接。

2) 连接后继者 如果没有已有的引用可定义一个链，那么你必须自己引入它们。这种情况下 Handler 不仅定义该请求的接口，通常也维护后继链接。这样 Handler 就提供了 HandleRequest 的缺省实现：HandleRequest 向后继者 (如果有的话) 转发请求。如果 ConcreteHandler 子类对该请求不感兴趣，它不需重定义转发操作，因为它的缺省实现进行无条件的转发。

此处为一个 HelpHandler 基类，它维护一个后继者链接：

```
class HelpHandler {
public:
    HelpHandler(HelpHandler* s) : _successor(s) { }
    virtual void HandleHelp();
private:
    HelpHandler* _successor;
};

void HelpHandler::HandleHelp () {
    if (_successor) {
        _successor->HandleHelp();
    }
}
```

```
    }
}
```

3) 表示请求 可以有不同的方法表示请求。最简单的形式，比如在 HandleHelp 的例子中，请求是一个硬编码的 (hard-coded) 操作调用。这种形式方便而且安全，但你只能转发 Handler 类定义的固定的一组请求。

另一选择是使用一个处理函数，这个函数以一个请求码 (如一个整型常数或一个字符串) 为参数。这种方法支持请求数目不限。唯一的要求是发送方和接受方在请求如何编码问题上应达成一致。

这种方法更为灵活，但它需要用条件语句来区分请求代码以分派请求。另外，无法用类型安全的方法来传递请求参数，因此它们必须被手工打包和解包。显然，相对于直接调用一个操作来说它不太安全。

为解决参数传递问题，我们可使用独立的请求对象来封装请求参数。Request 类可明确地描述请求，而新类型的请求可用它的子类来定义。这些子类可定义不同的请求参数。处理者必须知道请求的类型 (即它们正使用哪一个 Request 子类) 以访问这些参数。

为标识请求，Request 可定义一个访问器 (accessor) 函数以返回该类的标识符。或者，如果实现语言支持的话，接受者可使用运行时的类型信息。

以下为一个分派函数的框架 (sketch)，它使用请求对象标识请求。定义于基类 Request 中的 GetKind 操作识别请求的类型：

```
void Handler::HandleRequest (Request* theRequest) {
    switch (theRequest->GetKind()) {
        case Help:
            // cast argument to appropriate type
            HandleHelp((HelpRequest*) theRequest);
            break;

        case Print:
            HandlePrint((PrintRequest*) theRequest);
            // ...
            break;

        default:
            // ...
            break;
    }
}
```

子类可通过重定义 HandleRequest 扩展该分派函数。子类只处理它感兴趣的请求；其他的请求被转发给父类。这样就有效的扩展了 (而不是重写) HandleRequest 操作。例如，一个 ExtendedHandler 子类扩展了 MyHandler 版本的 HandleRequest:

```
class ExtendedHandler : public Handler {
public:
    virtual void HandleRequest(Request* theRequest);
    // ...
};

void ExtendedHandler::HandleRequest (Request* theRequest) {
    switch (theRequest->GetKind()) {
        case Preview:
            // handle the Preview request
            break;
        default:
            // ...
    }
}
```

```

        // let Handler handle other requests
        Handler::HandleRequest(theRequest);
    }
}

```

4) 在Smalltalk中自动转发 你可以使用Smalltalk 中的doesNotUnderstand机制转发请求。没有相应方法的消息被doseNotUnderstand 的实现捕捉 ( trap in ), 此实现可被重定义, 从而可向一个对象的后继者转发该消息。这样就不需要手工实现转发; 类仅处理它感兴趣的请求, 而依赖doesNotUnderstand 转发所有其他的请求。

#### 9. 代码示例

下面的例子举例说明了在一个像前面描述的在线帮助系统中, 职责链是如何处理请求的。帮助请求是一个显式的操作。我们将使用在窗口组件层次中的已有的父构件引用来在链中的窗口组件间传递请求, 并且我们将在 Handler类中定义一个引用以在链中的非窗口组件间传递帮助请求。

HelpHandler 类定义了处理帮助请求的接口。它维护一个帮助主题(缺省值为空), 并保持对帮助处理对象链中它的后继者的引用。关键的操作是 HandleHelp, 它可被子类重定义。HasHelp 是一个辅助操作, 用于检查是否有一个相关的帮助主题。

```

typedef int Topic;
const Topic NO_HELP_TOPIC = -1;

class HelpHandler {
public:
    HelpHandler(HelpHandler* = 0, Topic = NO_HELP_TOPIC);
    virtual bool HasHelp();
    virtual void SetHandler(HelpHandler*, Topic);
    virtual void HandleHelp();
private:
    HelpHandler* _successor;
    Topic _topic;
};

HelpHandler::HelpHandler (
    HelpHandler* h, Topic t
) : _successor(h), _topic(t) { }

bool HelpHandler::HasHelp () {
    return _topic != NO_HELP_TOPIC;
}

void HelpHandler::HandleHelp () {
    if (_successor != 0) {
        _successor->HandleHelp();
    }
}

```

所有的窗口组件都是Widget抽象类的子类。Widget是HelpHandler 的子类, 因为所有的用户界面元素都可有相关的帮助。(我们也可以使用另一种基于混入类的实现方式)

```

class Widget : public HelpHandler {
protected:
    Widget(Widget* parent, Topic t = NO_HELP_TOPIC);
private:
    Widget* _parent;
};

Widget::Widget (Widget* w, Topic t) : HelpHandler(w, t) {

```

```

    _parent = w;
}

```

在我们的例子中，按钮是链上的第一个处理者。Button类是Widget类的子类。Button构造函数有两个参数：对包含它的窗口组件的引用和其自身的帮助主题。

```

class Button : public Widget {
public:
    Button(Widget* d, Topic t = NO_HELP_TOPIC);

    virtual void HandleHelp();
    // Widget operations that Button overrides...
};

```

Button版本的HandleHelp首先测试检查其自身是否有帮助主题。如果开发者没有定义一个帮助主题，就用HelpHandler中的HandleHelp操作将该请求转发给它的后继者。如果有帮助主题，那么就显示它，并且搜索结束。

```

Button::Button (Widget* h, Topic t) : Widget(h, t) { }

void Button::HandleHelp () {
    if (HasHelp()) {
        // offer help on the button
    } else {
        HelpHandler::HandleHelp();
    }
}

```

Dialog实现了一个类似的策略，只不过它的后继者不是一个窗口组件而是任意的帮助请求处理对象。在我们的应用中这个后继者将是Application的一个实例。

```

class Dialog : public Widget {
public:
    Dialog(HelpHandler* h, Topic t = NO_HELP_TOPIC);
    virtual void HandleHelp();

    // Widget operations that Dialog overrides...
    // ...
};

Dialog::Dialog (HelpHandler* h, Topic t) : Widget(0) {
    SetHandler(h, t);
}

void Dialog::HandleHelp () {
    if (HasHelp()) {
        // offer help on the dialog
    } else {
        HelpHandler::HandleHelp();
    }
}

```

在链的末端是Application的一个实例。该应用不是一个窗口组件，因此Application不是HelpHandler的直接子类。当一个帮助请求传递到这一层时，该应用可提供关于该应用的一般性的信息，或者它可以提供一系列不同的帮助主题。

```

class Application : public HelpHandler {
public:
    Application(Topic t) : HelpHandler(0, t) { }

    virtual void HandleHelp();
    // application-specific operations...
}

```



```
};

void Application::HandleHelp () {
    // show a list of help topics
}
```

下面的代码创建并连接这些对象。此处的对话框涉及打印，因此这些对象被赋给与打印相关的主题。

```
const Topic PRINT_TOPIC = 1;
const Topic PAPER_ORIENTATION_TOPIC = 2;
const Topic APPLICATION_TOPIC = 3;

Application* application = new Application(APPLICATION_TOPIC);
Dialog* dialog = new Dialog(application, PRINT_TOPIC);
Button* button = new Button(dialog, PAPER_ORIENTATION_TOPIC);
```

我们可对链上的任意对象调用 HandleHelp 以触发相应的帮助请求。要从按钮对象开始搜索，只需对它调用 HandleHelp：

```
button->HandleHelp();
```

在这种情况下，按钮会立即处理该请求。注意任何 HelpHandler 类都可作为 Dialog 的后继者。此外，它的后继者可以被动态地改变。因此不管对话框被用在何处，你都可以得到它正确的与上下文相关的帮助信息。

#### 10. 已知应用

许多类库使用职责链模式处理用户事件。对 Handler 类它们使用不同的名字，但思想是一样的：当用户点击鼠标或按键盘，一个事件产生并沿链传播。MacApp[App89] 和 ET++[WGM88] 称之为“事件处理器”，Symantec 的 TCL 库[Sym93b]称之为“Bureaucrat”，而 NeXT 的 AppKit 命名为“Responder”。

图形编辑器框架 Unidraw 定义了“命令”Command 对象，它封装了发给 Component 和 ComponentView 对象[VL90]的请求。一个构件或构件视图可解释一个命令以进行一个操作，这里“命令”就是请求。这对应于在实现一节中描述的“对象作为请求”的方法。构件和构件视图可以组织为层次式的结构。一个构件或构件视图可将命令解释转发给它的父构件，而父构件依次可将它转发给它的父构件，如此类推，就形成了一个职责链。

ET++ 使用职责链来处理图形的更新。当一个图形对象必须更新它的外观的一部分时，调用 InvalidateRect 操作。一个图形对象自己不能处理 InvalidateRect，因为它对它的上下文了解不够。例如，一个图形对象可被包装在一些类似滚动条 (Scrollers) 或放大器 (Zoomers) 的对象中，这些对象变换它的坐标系统。那就是说，对象可被滚动或放大以至它有一部分在视区外。因此缺省的 InvalidateRect 的实现转发请求给包装的容器对象。转发链中的最后一个对象是一个窗口 (Window) 实例。当窗口收到请求时，保证失效矩形被正确变换。窗口通知窗口系统接口并请求更新，从而处理 InvalidateRect。

#### 11. 相关模式

职责链常与 Composite (4.3) 一起使用。这种情况下，一个构件的父构件可作为它的后继。

## 5.2 COMMAND (命令) ——对象行为型模式

#### 1. 意图

将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化；对请求排队



或记录请求日志，以及支持可撤消的操作。

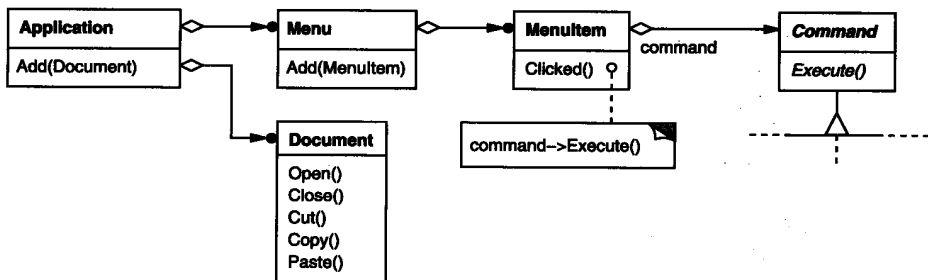
## 2. 别名

动作(Action)，事务(Transaction)

## 3. 动机

有时必须向某对象提交请求，但并不知道关于被请求的操作或请求的接受者的任何信息。例如，用户界面工具箱包括按钮和菜单这样的对象，它们执行请求响应用户输入。但工具箱不能显式的在按钮或菜单中实现该请求，因为只有使用工具箱的应用知道该由哪个对象做哪个操作。而工具箱的设计者无法知道请求的接受者或执行的操作。

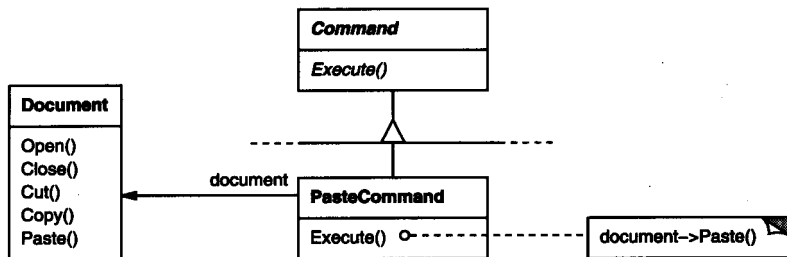
命令模式通过将请求本身变成一个对象来使工具箱对象可向未指定的应用对象提出请求。这个对象可被存储并像其他的对象一样被传递。这一模式的关键是一个抽象的 `Command` 类，它定义了一个执行操作的接口。其最简单的形式是一个抽象的 `Execute` 操作。具体的 `Command` 子类将接收者作为其一个实例变量，并实现 `Execute` 操作，指定接收者采取的动作。而接收者有执行该请求所需的具体信息。



用 `Command` 对象可很容易的实现菜单 (`Menu`)，每一菜单中的选项都是一个菜单项 (`MenuItem`) 类的实例。一个 `Application` 类创建这些菜单和它们的菜单项以及其余的用户界面。该 `Application` 类还跟踪用户已打开的 `Document` 对象。

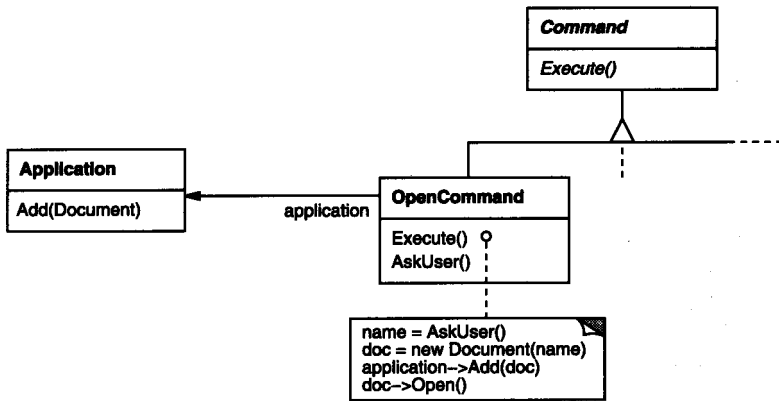
该应用为每一个菜单项配置一个具体的 `Command` 子类的实例。当用户选择了一个菜单项时，该 `MenuItem` 对象调用它的 `Command` 对象的 `Execute` 方法，而 `Execute` 执行相应操作。`MenuItem` 对象并不知道它们使用的是 `Command` 的哪一个子类。`Command` 子类里存放着请求的接收者，而 `Execute` 操作将调用该接收者的一个或多个操作。

例如，`PasteCommand` 支持从剪贴板向一个文档 (`Document`) 粘贴正文。`PasteCommand` 的接收者是一个文档对象，该对象是实例化时提供的。`Execute` 操作将调用该 `Document` 的 `Paste` 操作。

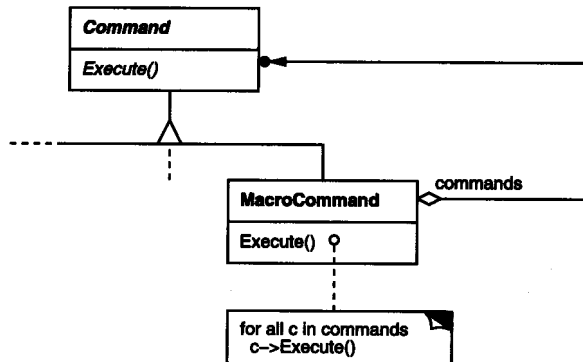


而 `OpenCommand` 的 `Execute` 操作却有所不同：它提示用户输入一个文档名，创建一个相应

的文档对象，将其入作为接收者的应用对象中，并打开该文档。



有时一个 MenuItem 需要执行一系列命令。例如，使一个页面按正常大小居中的 MenuItem 可由一个 CenterDocumentCommand 对象和一个 NormalSizeCommand 对象构建。因为这种需将多条命令串接起来的情况很常见，我们定义一个 MacroCommand 类来让一个 MenuItem 执行任意数目的命令。MacroCommand 是一个具体的 Command 子类，它执行一个命令序列。MacroCommand 没有明确的接收者，而序列中的命令各自定义其接收者。



请注意这些例子中 Command 模式是怎样解耦调用操作的对象和具有执行该操作所需信息的那个对象的。这使我们在设计用户界面时拥有很大的灵活性。一个应用如果想让一个菜单与一个按钮代表同一项功能，只需让它们共享相应具体 Command 子类的同一个实例即可。我们还可以动态地替换 Command 对象，这可用于实现上下文有关的菜单。我们也可通过将几个命令组成更大的命令的形式来支持命令脚本 (command scripting)。所有这些之所以成为可能乃是因为提交一个请求的对象仅需知道如何提交它，而不需知道该请求将会被如何执行。

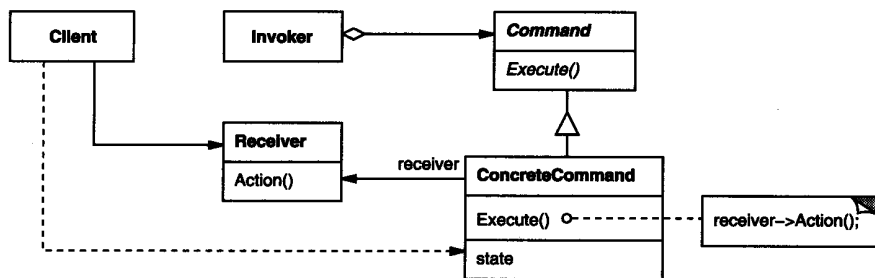
#### 4. 适用性

当你有如下需求时，可使用 Command 模式：

- 像上面讨论的 MenuItem 对象那样，抽象出待执行的动作以参数化某对象。你可用过程语言中的回调 (callback) 函数表达这种参数化机制。所谓回调函数是指函数先在某处注册，而它将在稍后某个需要的时候被调用。Command 模式是回调机制的一个面向对象的替代品。

- 在不同的时刻指定、排列和执行请求。一个 Command对象可以有一个与初始请求无关的生存期。如果一个请求的接收者可用一种与地址空间无关的方式表达，那么就可将负责该请求的命令对象传送给另一个不同的进程并在那儿实现该请求。
- 支持取消操作。Command的Excute操作可在实施操作前将状态存储起来，在取消操作时这个状态用来消除该操作的影响。Command接口必须添加一个 Unexecute操作，该操作取消上一次 Execute调用的效果。执行的命令被存储在一个历史列表中。可通过向后和向前遍历这一列表并分别调用 Unexecute和Execute来实现重数不限的“取消”和“重做”。
- 支持修改日志，这样当系统崩溃时，这些修改可以被重做一遍。在 Command接口中添加装载操作和存储操作，可以用来保持变动的一个一致的修改日志。从崩溃中恢复的过程包括从磁盘中重新读入记录下来的命令并用 Execute操作重新执行它们。
- 用构建在原语操作上的高层操作构造一个系统。这样一种结构在支持事务(transaction)的信息系统中很常见。一个事务封装了对数据的一组变动。Command模式提供了对事务进行建模的方法。Command有一个公共的接口，使得你可以用同一种方式调用所有的事务。同时使用该模式也易于添加新事务以扩展系统。

#### 5. 结构



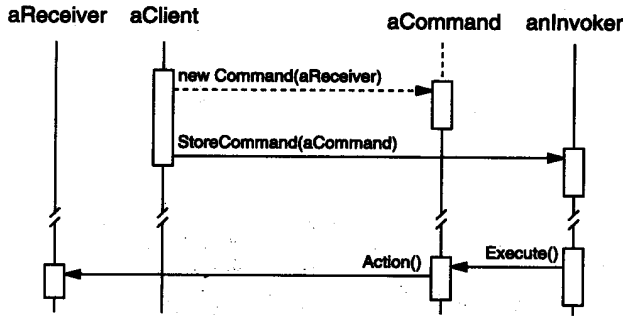
#### 6. 参与者

- Command
  - 声明执行操作的接口。
- ConcreteCommand (PasteCommand, OpenCommand)
  - 将一个接收者对象绑定于一个动作。
  - 调用接收者相应的操作，以实现 Execute。
- Client (Appliction)
  - 创建一个具体命令对象并设定它的接收者。
- Invoker (MenuItem)
  - 要求该命令执行这个请求。
- Receiver (Document, Application)
  - 知道如何实施与执行一个请求相关的操作。任何类都可能作为一个接收者。

#### 7. 协作

- Client创建一个 ConcreteCommand对象并指定它的 Receiver对象。
- 某Invoker对象存储该 ConcreteCommand对象。

- 该Invoker通过调用Command对象的Execute操作来提交一个请求。若该命令是可撤消的，ConcreteCommand就在执行Excute操作之前存储当前状态以用于取消该命令。
- ConcreteCommand对象对调用它的Receiver的一些操作以执行该请求。



下图展示了这些对象之间的交互。它说明了 Command是如何将调用者和接收者(以及它执行的请求)解耦的。

#### 8. 效果

Command模式有以下效果：

- 1) Command模式将调用操作的对象与知道如何实现该操作的对象解耦。
- 2) Command是头等的对象。它们可像其他的对象一样被操纵和扩展。
- 3) 你可将多个命令装配成一个复合命令。例如是前面描述的 MacroCommand类。一般说来，复合命令是Composite模式的一个实例。
- 4) 增加新的Command很容易，因为这无需改变已有的类。

#### 9. 实现

实现Command模式时须考虑以下问题：

1) 一个命令对象应达到何种智能程度 命令对象的能力可大可小。一个极端是它仅确定一个接收者和执行该请求的动作。另一极端是它自己实现所有功能，根本不需要额外的接收者对象。当需要定义与已有的类无关的命令，当没有合适的接收者，或当一个命令隐式地知道它的接收者时，可以使用后一极端方式。例如，创建另一个应用窗口的命令对象本身可能和任何其他对象一样有能力创建该窗口。在这两个极端间的情况是命令对象有足够的信息可以动态的找到它们的接收者。

2) 支持取消（undo）和重做（redo） 如果Command提供方法逆转(reverse)它们操作的执行(例如 Unexecute 或 Undo操作)，就可支持取消和重做功能。为达到这个目的，ConcreteCommand类可能需要存储额外的状态信息。这个状态包括：

- 接收者对象，它真正执行处理该请求的各操作。
- 接收者上执行操作的参数。
- 如果处理请求的操作会改变接收者对象中的某些值，那么这些值也必须先存储起来。接收者还必须提供一些操作，以使该命令可将接收者恢复到它先前的状态。

若应用只支持一次取消操作，那么只需存储最近一次被执行的命令。而若要支持多级的取消和重做，就需要有一个已被执行命令的历史表列(history list)，该表列的最大长度决定了取消和重做的级数。历史表列存储了已被执行的命令序列。向后遍历该表列并逆向执行

(reverse-executing)命令是取消它们的结果；向前遍历并执行命令是重执行它们。

有时可能不得不将一个可撤销的命令在它被放入历史列表中之前先拷贝下来。这是因为执行原来的请求的命令对象将在稍后执行其他的请求。如果命令的状态在各次调用之间会发生变化，那就必须进行拷贝以区分相同命令的不同调用。

例如，一个删除选定对象的删除命令 (DeleteCommand)在它每次被执行时，必须存储不同的对象集合。因此该删除命令对象在执行后必须被拷贝，并且将该拷贝放入历史表列中。如果该命令的状态在执行时从不改变，则不需要拷贝，而仅需将一个对该命令的引用放入历史表列中。在放入历史表列中之前必须被拷贝的那些 Command起着原型（参见 Prototype模式 (3.4)）的作用。

3) 避免取消操作过程中的错误积累 在实现一个可靠的、能保持原先语义的取消/重做机制时，可能会遇到滞后影响问题。由于命令重复的执行、取消执行，和重执行的过程可能会积累错误，以至一个应用的状态最终偏离初始值。这就有必要在 Command中存入更多的信息以保证这些对象可被精确地复原成它们的初始状态。这里可使用 Memento模式 (5.6) 来让该 Command访问这些信息而不暴露其他对象的内部信息。

4) 使用C++模板 对(1)不能被取消 (2)不需要参数的命令，我们可使用 C++模板来实现，这样可以避免为每一种动作和接收者都创建一个 Command子类。我们将在代码示例一节说明这种做法。

#### 10. 代码示例

此处所示的 C++代码给出了动机一节中的 Command类的实现的大致框架。我们将定义 OpenCommand、PasteCommand和MacroCommand。首先是抽象的 Command类：

```
class Command {
public:
    virtual ~Command();

    virtual void Execute() = 0;
protected:
    Command();
};
```

OpenCommand打开一个名字由用户指定的文档。注意 OpenCommand的构造器需要一个 Application对象作为参数。AskUser是一个提示用户输入要打开的文档名的实现例程。

```
class OpenCommand : public Command {
public:
    OpenCommand(Application*);

    virtual void Execute();
protected:
    virtual const char* AskUser();
private:
    Application* _application;
    char* _response;
};

OpenCommand::OpenCommand (Application* a) {
    _application = a;
}

void OpenCommand::Execute () {
    const char* name = AskUser();

    if (name != 0) {
```

```

        Document* document = new Document(name);
        _application->Add(document);
        document->Open();
    }
}

```

PasteCommand需要一个 Document对象作为其接收者。该接收者将作为一个参数给 PasteCommand的构造器。

```

class PasteCommand : public Command {
public:
    PasteCommand(Document*);

    virtual void Execute();
private:
    Document* _document;
};

PasteCommand::PasteCommand (Document* doc) {
    _document = doc;
}

void PasteCommand::Execute () {
    _document->Paste();
}

```

对于简单的不能取消和无需参数的命令，可以用一个类模板来参数化该命令的接收者。我们将为这些命令定义一个模板子类 SimpleCommand。用 Receiver 类型参数化 SimpleCommand，并维护一个接收者对象和一个动作之间的绑定，而这一动作是用指向一个成员函数的指针存储的。

```

template <class Receiver>
class SimpleCommand : public Command {
public:
    typedef void (Receiver::* Action)();

    SimpleCommand(Receiver* r, Action a) :
        _receiver(r), _action(a) { }

    virtual void Execute();
private:
    Action _action;
    Receiver* _receiver;
};

```

构造器存储接收者和对应实例变量中的动作。Execute操作实施接收者的这个动作。

```

template <class Receiver>
void SimpleCommand<Receiver>::Execute () {
    (_receiver->*_action)();
}

```

为创建一个调用 MyClass 类的一个实例上的 Action 的 Command 对象，仅需如下代码：

```

MyClass* receiver = new MyClass;
// ...
Command* aCommand =
    new SimpleCommand<MyClass>(receiver, &MyClass::Action);
// ...
aCommand->Execute();

```

记住，这一方案仅适用于简单命令。更复杂的命令不仅要维护它们的接收者，而且还要登记参数，有时还要保存用于取消操作的状态。此时就需要定义一个 Command 的子类。

MacroCommand管理一个子命令序列，它提供了增加和删除子命令的操作。这里不需要显式的接收者，因为这些子命令已经定义了它们各自的接收者。

```
class MacroCommand : public Command {
public:
    MacroCommand();
    virtual ~MacroCommand();

    virtual void Add(Command*);
    virtual void Remove(Command*);

    virtual void Execute();
private:
    List<Command*>* _cmds;
};
```

MacroCommand的关键是它的 Execute成员函数。它遍历所有的子命令并调用其各自的 Execute操作。

```
void MacroCommand::Execute () {
    ListIterator<Command*> i(_cmds);

    for (i.First(); !i.IsDone(); i.Next()) {
        Command* c = i.CurrentItem();
        c->Execute();
    }
}
```

注意，如果 MacroCommand实现取消操作，那么它的子命令必须以相对于 Execute的实现相反的顺序执行各子命令的取消操作。

最后，MacroCommand必须提供管理它的子命令的操作。MacroCommand也负责删除它的子命令。

```
void MacroCommand::Add (Command* c) {
    _cmds->Append(c);
}

void MacroCommand::Remove (Command* c) {
    _cmds->Remove(c);
}
```

## 11. 已知应用

可能最早的命令模式的例子出现在 Lieberman[Lie85]的一篇论文中。MacApp[App89]使实现可撤消操作的命令这一说法被普遍接受。而 ET++[WGM88]，InterViews[LCI+92]，和 Unidraw[VL90]也都定义了符合 Command模式的类。InterViews定义了一个 Action抽象类，它提供命令功能。它还定义了一个 ActionCallback模板，这个模板以 Action方法为参数，可自动生成 Command子类。

THINK类库[Sym93b]也使用 Command模式支持可撤消的操作。THINK中的命令被称为“任务”(Tasks)。任务对象沿着一个 Chain of Responsibility (5.1) 传递以供消费(consumption)。

Unidraw的命令对象很特别，它的行为就像是一个消息。一个 Unidraw命令可被送给另一个对象去解释，而解释的结果因接收的对象而异。此外，接收者可以委托另一个对象来进行解释，典型的情况的是委托给一个较大的结构中(比如在一个职责链中)接收者的父构件。这样，Unidraw命令的接收者是计算出来的而不是预先存储的。Unidraw的解释机制依赖于运行时的类型信息。

Coplien在C++[Cop92]中描述了C++中怎样实现 functors。Functors是一种实际上是函数的



对象。他通过重载函数调用操作符 (operator()) 达到了一定程度的使用透明性。命令模式不同，它着重于维护接收者和函数 (即动作) 之间的绑定，而不仅是维护一个函数。

## 12. 相关模式

Composite模式 (4.3) 可被用来实现宏命令。

Memento模式 (5.6) 可用来保持某个状态，命令用这一状态来取消它的效果。

在被放入历史表列前必须被拷贝的命令起到一种原型 (3.4) 的作用。

## 5.3 INTERPRETER(解释器)——类行为型模式

### 1. 意图

给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。

### 2. 动机

如果一种特定类型的问题发生的频率足够高，那么可能就值得将该问题的各个实例表述为一个简单语言中的句子。这样就可以构建一个解释器，该解释器通过解释这些句子来解决该问题。

例如，搜索匹配一个模式的字符串是一个常见问题。正则表达式是描述字符串模式的一种标准语言。与其为每一个的模式都构造一个特定的算法，不如使用一种通用的搜索算法来解释执行一个正则表达式，该正则表达式定义了待匹配字符串的集合。

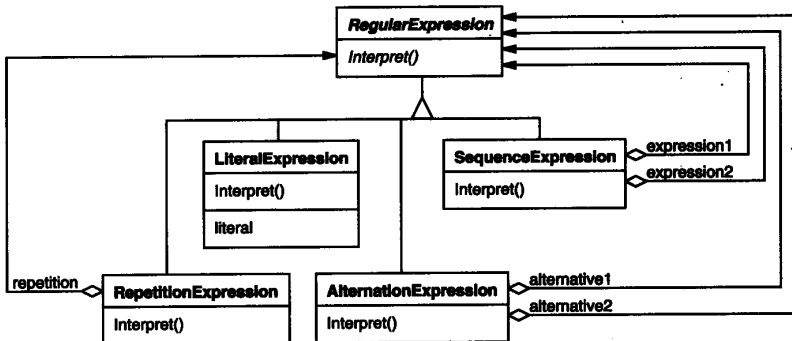
解释器模式描述了如何为简单的语言定义一个文法，如何在该语言中表示一个句子，以及如何解释这些句子。在上面的例子中，本设计模式描述了如何为正则表达式定义一个文法，如何表示一个特定的正则表达式，以及如何解释这个正则表达式。

考虑以下文法定义正则表达式：

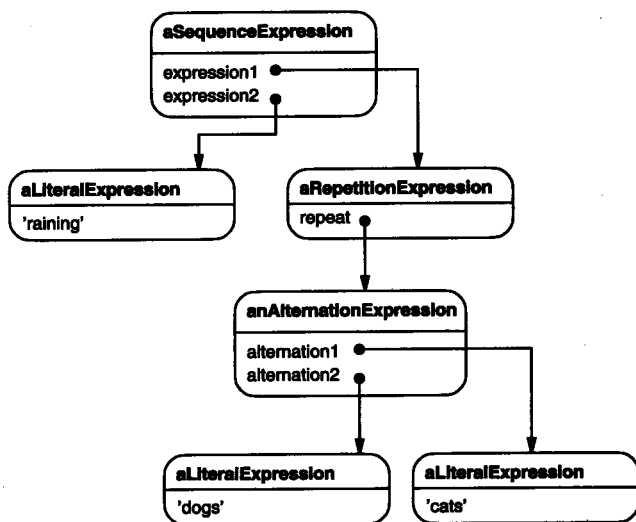
```
expression ::= literal | alternation | sequence | repetition |  
              '(' expression ')'  
alternation ::= expression '|' expression  
sequence ::= expression '&' expression  
repetition ::= expression '*'  
literal ::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' | ... }*
```

符号expression是开始符号，literal是定义简单字的终结符。

解释器模式使用类来表示每一条文法规则。在规则右边的符号是这些类的实例变量。上面的文法用五个类表示：一个抽象类RegularExpression和它四个子类LiteralExpression、AlternationExpression、SequenceExpression和RepetitionExpression后三个类定义的变量代表子表达式。



每个用这个文法定义的正则表达式都被表示为一个由这些类的实例构成的抽象语法树。例如, 抽象语法树:



表示正则表达式:

`raining & (dogs | cats) *`

如果我们为RegularExpression的每一子类都定义解释 (Interpret) 操作, 那么就得到了为这些正则表达式的一个解释器。解释器将该表达式的上下文做为一个参数。上下文包含输入字符串和关于目前它已有多少已经被匹配等信息。为匹配输入字符串的下一部分, 每一RegularExpression的子类都在当前上下文的基础上实现解释操作 (Interpret)。例如,

- LiteralExpression将检查输入是否匹配它定义的字 (literal)。
  - AlternationExpression将检查输入是否匹配它的任意一个选择项。
  - RepetitionExpression将检查输入是否含有多个它所重复的表达式。
- 等等。

### 3. 适用性

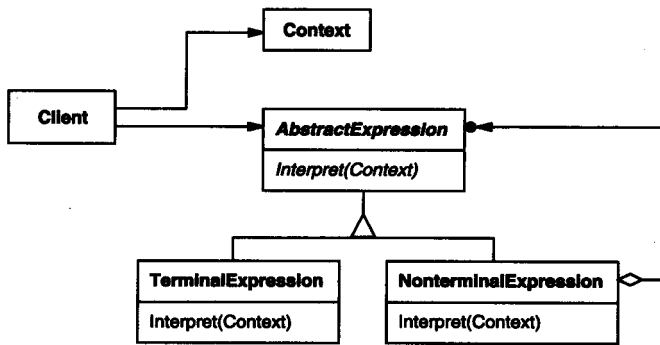
当有一个语言需要解释执行, 并且你可将该语言中的句子表示为一个抽象语法树时, 可使用解释器模式。而当存在以下情况时该模式效果最好:

- 该文法简单对于复杂的文法, 文法的类层次变得庞大而无法管理。此时语法分析程序生成器这样的工具是更好的选择。它们无需构建抽象语法树即可解释表达式, 这样可以节省空间而且还可能节省时间。
- 效率不是一个关键问题最高效的解释器通常不是通过直接解释语法分析树实现的, 而是首先将它们转换成另一种形式。例如, 正则表达式通常被转换成状态机。但即使在这种情况下, 转换器仍可用解释器模式实现, 该模式仍是有用的。

### 4. 结构 (见下页图)

### 5. 参与者

- AbstractExpression (抽象表达式, 如RegularExpression)
  - 声明一个抽象的解释操作, 这个接口为抽象语法树中所有的节点所共享。
- TerminalExpression (终结符表达式, 如LiteralExpression)



— 实现与文法中的终结符相关联的解释操作。

— 一个句子中的每个终结符需要该类的一个实例。

- NonterminalExpression (非终结符表达式, 如 AlternationExpression, RepetitionExpression, SequenceExpressions)

— 对文法中的每一条规则  $R ::= R_1 R_2 \dots R_n$  都需要一个 NonterminalExpression 类。

— 为从  $R_1$  到  $R_n$  的每个符号都维护一个 AbstractExpression 类型的实例变量。

— 为文法中的非终结符实现解释 (Interpret) 操作。解释 (Interpret) 一般要递归地调用表示  $R_1$  到  $R_n$  的那些对象的解释操作。

- Context (上下文)

— 包含解释器之外的一些全局信息。

- Client (客户)

— 构建(或被给定)表示该文法定义的语言中一个特定的句子的抽象语法树。该抽象语法树由 NonterminalExpression 和 TerminalExpression 的实例装配而成。

— 调用解释操作。

## 6. 协作

- Client 构建(或被给定)一个句子, 它是 NonterminalExpression 和 TerminalExpression 的实例的一个抽象语法树。然后初始化上下文并调用解释操作。

- 每一非终结符表达式节点定义相应子表达式的解释操作。而各终结符表达式的解释操作构成了递归的基础。

- 每一节点的解释操作用上下文来存储和访问解释器的状态。

## 7. 效果

解释器模式有下列的优点和不足:

1) 易于改变和扩展文法 因为该模式使用类来表示文法规则, 你可使用继承来改变或扩展该文法。已有的表达式可被增量式地改变, 而新的表达式可定义为旧表达式的变体。

2) 也易于实现文法 定义抽象语法树中各个节点的类的实现大体类似。这些类易于直接编写, 通常它们也可用一个编译器或语法分析程序生成器自动生成。

3) 复杂的文法难以维护 解释器模式为文法中的每一条规则至少定义了一个类(使用BNF定义的文法规则需要更多的类)。因此包含许多规则的文法可能难以管理和维护。可应用其他的设计模式来缓解这一问题。但当文法非常复杂时, 其他的技术如语法分析程序或编译器生成器更为合适。

4) 增加了新的解释表达式的方式 解释器模式使得实现新表达式“计算”变得容易。例如,你可以在表达式类上定义一个新的操作以支持优美打印或表达式的类型检查。如果你经常创建新的解释表达式的方式,那么可以考虑使用 Visitor(5.11)模式以避免修改这些代表文法的类。

### 8. 实现

Interpreter和Composite (4.3) 模式在实现上有许多相通的地方。下面是 Interpreter所要考虑的一些特殊问题:

1) 创建抽象语法树 解释器模式并未解释如何创建一个抽象的语法树。换言之,它不涉及语法分析。抽象语法树可用一个表驱动的语法分析程序来生成,也可用手写的(通常为递归下降法)语法分析程序创建,或直接由 Client提供。

2) 定义解释操作 并不一定要在表达式类中定义解释操作。如果经常要创建一种新的解释器,那么使用 Visitor (5.11) 模式将解释放入一个独立的“访问者”对象更好一些。例如,一个程序设计语言的会有许多在抽象语法树上的操作,比如类型检查、优化、代码生成,等等。恰当的做法是使用一个访问者以避免在每一个类上都定义这些操作。

3) 与Flyweight模式共享终结符 在一些文法中,一个句子可能多次出现同一个终结符。此时最好共享那个符号的单个拷贝。计算机程序的文法是很好的例子——每个程序变量在整个代码中将会出现多次。在动机一节的例子中,一个句子中终结符 dog (由LiteralExpression类描述)也可出现多次。

终结节点通常不存储关于它们在抽象语法树中位置的信息。在解释过程中,任何它们所需要的上下文信息都由父节点传递给它们。因此在共享的(内部的)状态和传入的(外部的)状态区分得很明确,这就用到了Flyweight (4.6) 模式。

例如, dog LiteralExpression的每一实例接收一个包含目前已匹配子串信息的上下文。且每一个这样的LiteralExpression在它的解释操作中做同样一件事(它检查输入的下一部分是否包含一个dog)无论该实例出现在语法树的哪个位置。

### 9. 代码示例

下面是两个例子。第一个是Smalltalk中一个完整的例子,用于检查一个序列是否匹配一个正则表达式。第二个是一个用于求布尔表达式的值的 C++程序。

正则表达式匹配器检查一个字符串是否属于一个正则表达式定义的语言。正则表达式用下列文法定义:

```
expression ::= literal | alternation | sequence | repetition |
              '(' expression ')'
alternation ::= expression '|' expression
sequence  ::= expression '&' expression
repetition ::= expression 'repeat'
literal   ::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' | ... }*
```

该文法对动机一节中的例子略做修改。因为符号“\*”在Smalltalk中不能作为后缀运算符。因此我们用repeat取代之。例如,正则表达式:

```
(( 'dog' | 'cat' ) repeat &'weather')
```

匹配输入字符串“dog dog cat weather”。

为实现这个匹配器,我们定义在(5.3)页描述的五类。类SequenceExpression包含实例变量expression 1和expression 2作为它在抽象语法树中的子结点; AlternationExpression用实例变量altercative 1和altercative 2中存储它的选择支;而 RepetitionExpression在它的实例变量

repetition中保存它所重复的表达式。LiteralExpression有一个components实例变量，它保存了一系列对象(可能为一些字符)。这些表示必须匹配输入序列的字串(literal string)。

match:操作实现了该正则表达式的一个解释器。定义抽象语法树的每一个类都实现了这一操作。它将inputState作为一个参数,表示匹配进程的当前状态,也就是读入的部分输入字符串。

这一状态由一个输入流集刻画,表示该正则表达式目前所能接收的输入集(当前已识别出的输入流,这大致等价于记录等价的有限自动机可能处于的所有状态)。

当前状态对repeat操作最为重要。例如,如果正则表达式为:

```
'a' repeat
```

那么解释器可匹配“a”,“aa”,“aaa”,等等。如果它是

```
'a' repeat & 'bc'
```

那么可以匹配“abc”,“aabc”,“aaabc”,等等。但如果正则表达式是

```
'a' repeat & 'abc'
```

那么用子表达式“‘a’ repeat”匹配输入“aabc”将产生两个输入流,一个匹配了输入的一个字符,而另一个匹配了两个字符。只有接受一个字符的那个流会匹配剩余的“abc”。

现在我们考虑match的定义:对每一个类定义相应的正则表达式。SequenceExpression匹配其序列中的每一个子表达式。通常它将从它的inputState中删除输入流。

```
match: inputState
  ^ expression2 match: (expression1 match: inputState).
```

一个AlternationExpression会返回一个状态,该状态由两个选择项的状态的并组成。

AlternationExpression的match的定义是

```
match: inputState
  | finalState |
  finalState := alternative1 match: inputState.
  finalState addAll: (alternative2 match: inputState).
  ^ finalState
```

RepetitionExpression的match:操作寻找尽可能多的可匹配的状态:

```
match: inputState
  | aState finalState |
  aState := inputState.
  finalState := inputState copy.
  [aState isEmpty]
  whileFalse:
    [aState := repetition match: aState.
     finalState addAll: aState].
  ^ finalState
```

它的输出通常比它的输入包含更多的状态,因为RepetitionExpression可匹配输入的重复体的一次、两次或多次出现。而输出状态要表示所有这些可能性以允许随后的正则表达式的元素决定哪一个状态是正确的。

最后,LiteralExpression的match:对每一可能的输入流匹配它的组成部分。它仅保留那些获得匹配的输入流:

```
match: inputState
  | finalState tStream |
  finalState := Set new.
  inputState
    do:
      [:stream | tStream := stream copy.
```

```

        (tStream nextAvailable:
            components size
        ) = components
            ifTrue: [finalState add: tStream]
    ].
    ^ finalState

```

其中nextAvailable:消息推进输入流（即读入文字）。这是唯一一个推进输入流的 match:操作。注意返回的状态包含的是输入流的拷贝，这就保证匹配一个 literal不会改变输入流。这一点很重要，因为每个 AlternationExpression 的选择项看到的应该是相同的输入流。

现在我们已经定义了组成抽象语法树的各个类，下面说明怎样构建语法树。我们犯不着为正则表达式写一个语法分析程序，而只要在 RegularExpression 类上定义一些操作，就可以“计算”一个 Smalltalk 表达式，得到的结果就是对应于该正则表达式的一棵抽象语法树。这使我们可以把 Smalltalk 内置编译器当作一个正则表达式的语法分析程序来使用。

为构建抽象语法树，我们需要将“|”、“repeat”，和“&”定义为 RegularExpression 上的操作。这些操作在 RegularExpression 类中定义如下：

```

& aNode
    ^ SequenceExpression new
        expression1: self expression2: aNode asRExp

repeat
    ^ RepetitionExpression new repetition: self
    | aNode
        ^ AlternationExpression new
            alternative1: self alternative2: aNode asRExp

asRExp
    ^ self

```

asRExp 操作将把 literals 转化为 RegularExpression。这些操作在类 String 中定义：

```

& aNode
    ^ SequenceExpression new
        expression1: self asRExp expression2: aNode asRExp

repeat
    ^ RepetitionExpression new repetition: self

| aNode
    ^ AlternationExpression new
        alternative1: self asRExp alternative2: aNode asRExp

asRExp
    ^ LiteralExpression new components: self

```

如果我们在类层次的更高层（Smalltalk 中的 SequenceableCollection，Smalltalk/V 中的 IndexedCollection）中定义这些操作，那么象 Array 和 OrderedCollection 这样的类也有这些操作的定义，这就使得正则表达式可以匹配任何类型的对象序列。

第二个例子是在 C++ 中实现的对布尔表达式进行操作和求值。在这个语言中终结符是布尔变量，即常量 true 和 false。非终结符表示包含运算符 and, or 和 not 的布尔表达式。文法定义如下<sup>①</sup>：

```

BooleanExp ::= VariableExp | Constant | OrExp | AndExp | NotExp |
              '(' BooleanExp ')'

```

① 为简单起见，我们忽略了操作符的优先次序且假定由构造该语法树的对象负责处理这件事。



```

AndExp ::= BooleanExp 'and' BooleanExp
OrExp  ::= BooleanExp 'or' BooleanExp
NotExp ::= 'not' BooleanExp
Constant ::= 'true' | 'false'
VariableExp ::= 'A' | 'B' | ... | 'X' | 'Y' | 'Z'

```

这里我们定义布尔表达式上的两个操作。第一个操作是求值 (evaluate)，即在一个上下文中求一个布尔表达式的值，当然，该上下文必须为每个变量都赋以一个“真”或“假”的布尔值。第二个操作是替换 (replace)，即用一个表达式来替换一个变量以产生一个新的布尔表达式。替换操作说明了解释器模式不仅可以用于求表达式的值，而且还可用作其它用途。在这个例子中，它就被用来对表达式本身进行操作。

此处我们仅给出 BooleanExp, VariableExp 和 AndExp 类的细节。类 OrExp 和 NotExp 与 AndExp 相似。Constant 类表示布尔常量。

BooleanExp 为所有定义一个布尔表达式的类定义了一个接口：

```

class BooleanExp {
public:
    BooleanExp();
    virtual ~BooleanExp();

    virtual bool Evaluate(Context&) = 0;
    virtual BooleanExp* Replace(const char*, BooleanExp&) = 0;
    virtual BooleanExp* Copy() const = 0;
};

```

类 Context 定义从变量到布尔值的一个映射，这些布尔值我们可用 C++ 中的常量 true 和 false 来表示。Context 有以下接口：

```

class Context {
public:
    bool Lookup(const char*) const;
    void Assign(VariableExp*, bool);
};

```

一个 VariableExp 表示一个有名变量：

```

class VariableExp : public BooleanExp {
public:
    VariableExp(const char*);
    virtual ~VariableExp();

    virtual bool Evaluate(Context&);
    virtual BooleanExp* Replace(const char*, BooleanExp&);
    virtual BooleanExp* Copy() const;
private:
    char* _name;
};

```

构造器将变量的名字作为参数：

```

VariableExp::VariableExp (const char* name) {
    _name = strdup(name);
}

```

求一个变量的值，返回它在当前上下文中的值。

```

bool VariableExp::Evaluate (Context& aContext) {
    return aContext.Lookup(_name);
}

```

拷贝一个变量返回一个新的 VariableExp：



```
BooleanExp* VariableExp::Copy () const {
    return new VariableExp(_name);
}
```

在用一个表达式替换一个变量时，我们检查该待替换变量是否就是本对象代表的变量：

```
BooleanExp* VariableExp::Replace (
    const char* name, BooleanExp& exp
) {
    if (strcmp(name, _name) == 0) {
        return exp.Copy();
    } else {
        return new VariableExp(_name);
    }
}
```

AndExp表示由两个布尔表达式与操作得到的表达式。

```
class AndExp : public BooleanExp {
public:
    AndExp(BooleanExp*, BooleanExp*);
    virtual ~AndExp();

    virtual bool Evaluate(Context&);
    virtual BooleanExp* Replace(const char*, BooleanExp&);
    virtual BooleanExp* Copy() const;
private:
    BooleanExp* _operand1;
    BooleanExp* _operand2;
};

AndExp::AndExp (BooleanExp* op1, BooleanExp* op2) {
    _operand1 = op1;
    _operand2 = op2;
}
```

一个AndExp的值求是它的操作数的值的逻辑“与”。

```
bool AndExp::Evaluate (Context& aContext) {
    return
        _operand1->Evaluate(aContext) &&
        _operand2->Evaluate(aContext);
}
```

AndExp的Copy和Replace操作将递归调用它的操作数的Copy和Replace操作：

```
BooleanExp* AndExp::Copy () const {
    return
        new AndExp(_operand1->Copy(), _operand2->Copy());
}

BooleanExp* AndExp::Replace (const char* name, BooleanExp& exp) {
    return
        new AndExp(
            _operand1->Replace(name, exp),
            _operand2->Replace(name, exp)
        );
}
```

现在我们可以定义布尔表达式

```
(true and x) or (y and (not x))
```

并对给定的以true或false赋值的x和y求这个表达式值：

```
BooleanExp* expression;
```

```

Context context;

VariableExp* x = new VariableExp("X");
VariableExp* y = new VariableExp("Y");

expression = new OrExp(
    new AndExp(new Constant(true), x),
    new AndExp(y, new NotExp(x))
);

context.Assign(x, false);
context.Assign(y, true);

bool result = expression->Evaluate(context);

```

对x和y的这一赋值，求得该表达式值为 true。要对其它赋值情况求该表达式的值仅需改变上下文对象即可。

最后，我们可用一个新的表达式替换变量 y，并重新求值：

```

VariableExp* z = new VariableExp("Z");
NotExp not_z(z);

BooleanExp* replacement = expression->Replace("Y", not_z);

context.Assign(z, true);

result = replacement->Evaluate(context);

```

这个例子说明了解释器模式一个很重要的特点：可以用多种操作来“解释”一个句子。在为 BooleanExp 定义的三种操作中，Evaluate 最切合我们关于一个解释器应该做什么的想法——即，它解释一个程序或表达式并返回一个简单的结果。但是，替换操作也可被视为一个解释器。这个解释器的上下文是被替换变量的名字和替换它的表达式，而它的结果是一个新的表达式。甚至拷贝也可被视为一个上下文为空的解释器。将替换和拷贝视为解释器可能有点怪，因为它们仅仅是树上的基本操作。Visitor(5.11)中的例子说明了这三个操作都可以被重新组织为独立的“解释器”访问者，从而显示了它们之间深刻的相似性。

解释器模式不仅仅是分布在一个使用 Composite(4.3)模式的类层次上的操作。我们之所以认为 Evaluate 是一个解释器，是因为我们认为 BooleanExp 类层次表示一个语言。对于一个用于表示汽车部件装配的类层次，即使它也使用复合模式，我们还是不太可能将 Weight 和 Copy 这样的操作视为解释器，因为我们不会把汽车部件当作一个语言。这是一个看问题的角度问题；如果我们真有“汽车部件语言”的语法，那么也许可以认为在那些部件上的操作是以某种方式解释该语言。

#### 10. 已知应用

解释器模式在使用面向对象语言实现的编译器中得到了广泛应用，如 Smalltalk 编译器。SPECTalk 使用该模式解释输入文件格式的描述 [Sza92]。QOCA 约束—求解工具使用它对约束进行计算 [HHMV92]。

在最宽泛的概念下(即，分布在基于 Composite(4.3)模式的类层次上的一种操作)，几乎每个使用复合模式的系统也都使用了解释器模式。但一般只有在用一个类层次来定义某个语言时，才强调使用解释器模式。

#### 11. 相关模式

Composite 模式 (4.3)：抽象语法树是一个复合模式的实例。

Flyweight模式 (4.6): 说明了如何在抽象语法树中共享终结符。

Iterator (5.4): 解释器可用一个迭代器遍历该结构。

Visitor (5.11): 可用来在一个类中维护抽象语法树中的各节点的行为。

## 5.4 ITERATOR(迭代器)——对象行为型模式

### 1. 意图

提供一种方法顺序访问一个聚合对象中各个元素, 而又不需暴露该对象的内部表示。

### 2. 别名

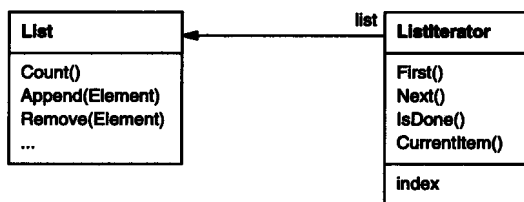
游标 (Cursor)

### 3. 动机

一个聚合对象, 如列表(list), 应该提供一种方法来让别人可以访问它的元素, 而又不需暴露它的内部结构。此外, 针对不同的需要, 可能要以不同的方式遍历这个列表。但是即使可以预见到所需的那些遍历操作, 你可能也不希望列表的接口中充斥着各种不同遍历的操作。有时还可能需要在同一个表列上同时进行多个遍历。

迭代器模式都可帮你解决所有这些问题。这一模式的关键思想是将对列表的访问和遍历从列表对象中分离出来并放入一个迭代器 (iterator) 对象中。迭代器类定义了一个访问该列表元素的接口。迭代器对象负责跟踪当前的元素; 即, 它知道哪些元素已经遍历过了。

例如, 一个列表 (List) 类可能需要一个列表迭代器 (ListIterator), 它们之间的关系如下图:



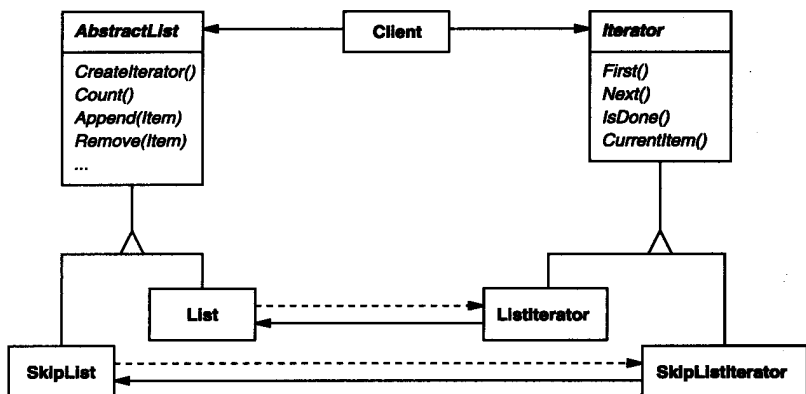
在实例化列表迭代器之前, 必须提供待遍历的列表。一旦有了该列表迭代器的实例, 就可以顺序地访问该列表的各个元素。CurrentItem操作返回表列中的当前元素, First操作初始化迭代器, 使当前元素指向列表的第一个元素, Next操作将当前元素指针向前推进一步, 指向下一个元素, 而IsDone检查是否已越过最后一个元素, 也就是完成了这次遍历。

将遍历机制与列表对象分离使我们可以定义不同的迭代器来实现不同的遍历策略, 而无需在列表接口中列举它们。例如, 过滤表列迭代器(FilteringListIterator)可能只访问那些满足特定过滤约束条件的元素。

注意迭代器和列表是耦合在一起的, 而且客户对象必须知道遍历的是一个列表而不是其他聚合结构。最好能有一种办法使得不需改变客户代码即可改变该聚合类。可以通过将迭代器的概念推广到多态迭代(polymorphic iteration)来达到这个目标。

例如, 假定我们还有一个列表的特殊实现, 比如说 SkipList[Pug90]。SkipList是一种具有类似于平衡树性质的随机数据结构。我们希望我们的代码对 List和SkipList对象都适用。

首先, 定义一个抽象列表类 AbstractList, 它提供操作列表的公共接口。类似地, 我们也需要一个抽象的迭代器类 Iterator, 它定义公共的迭代接口。然后我们可以为每个不同的列表实现定义具体的Iterator子类。这样迭代机制就与具体的聚合类无关了。



余下的问题是如何创建迭代器。既然要使这些代码不依赖于具体的列表子类，就不能仅仅简单地实例化一个特定的类，而要让列表对象负责创建相应的迭代器。这需要列表对象提供 `CreateIterator` 这样的操作，客户请求调用该操作以获得一个迭代器对象。

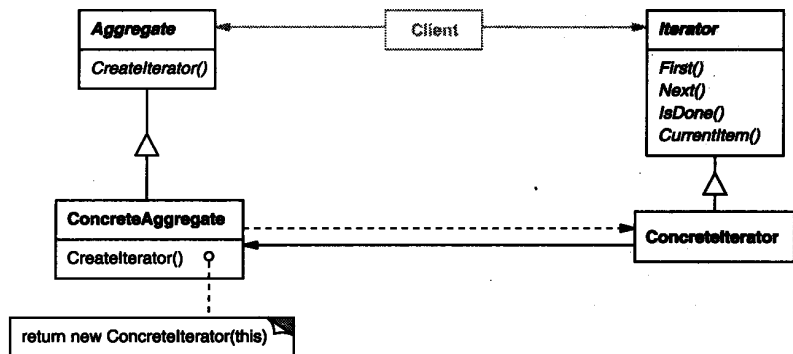
创建迭代器是一个 Factory Method 模式（3.3）的例子。我们在这里用它来使得一个客户可向一个列表对象请求合适的迭代器。Factory Method 模式产生两个类层次，一个是列表的，一个是迭代器的。CreateIterator “联系” 这两个类层次。

#### 4. 适用性

迭代器模式可用来：

- 访问一个聚合对象的内容而无需暴露它的内部表示。
- 支持对聚合对象的多种遍历。
- 为遍历不同的聚合结构提供一个统一的接口（即，支持多态迭代）。

#### 5. 结构



#### 6. 参与者

- **Iterator**（迭代器）
  - 迭代器定义访问和遍历元素的接口。
- **ConcreteIterator**（具体迭代器）
  - 具体迭代器实现迭代器接口。
  - 对该聚合遍历时跟踪当前位置。
- **Aggregate**（聚合）

— 聚合定义创建相应迭代器对象的接口。

- ConcreteAggregate (具体聚合)

— 具体聚合实现创建相应迭代器的接口, 该操作返回 ConcreteIterator 的一个适当的实例。

## 7. 协作

- ConcreteIterator 跟踪聚合中的当前对象, 并能够计算出待遍历的后继对象。

## 8. 效果

迭代器模式有三个重要的作用:

1) 它支持以不同的方式遍历一个聚合 复杂的聚合可用多种方式进行遍历。例如, 代码生成和语义检查要遍历语法分析树。代码生成可以按中序或者按前序来遍历语法分析树。迭代器模式使得改变遍历算法变得很容易: 仅需用一个不同的迭代器的实例代替原先的实例即可。你也可以自己定义迭代器的子类以支持新的遍历。

2) 迭代器简化了聚合的接口 有了迭代器的遍历接口, 聚合本身就不再需要类似的遍历接口了。这样就简化了聚合的接口。

3) 在同一个聚合上可以有多个遍历 每个迭代器保持它自己的遍历状态。因此你可以同时进行多个遍历。

## 9. 实现

迭代器在实现上有许多变化和选择。下面是一些较重要的实现。实现迭代器模式时常常需要根据所使用的语言提供的控制结构来进行权衡。一些语言 (例如, CLU[LG86]) 甚至直接支持这一模式。

1) 谁控制该迭代 一个基本的问题是决定由哪一方来控制该迭代, 是迭代器还是使用该迭代器的客户。当由客户来控制迭代时, 该迭代器称为一个外部迭代器 (external iterator), 而当由迭代器控制迭代时, 该迭代器称为一个内部迭代器 (internal iterator)<sup>①</sup>。使用外部迭代器的客户必须主动推进遍历的步伐, 显式地向迭代器请求下一个元素。相反地, 若使用内部迭代器, 客户只需向其提交一个待执行的操作, 而迭代器将对聚合中的每一个元素实施该操作。

外部迭代器比内部迭代器更灵活。例如, 若要比两个集合是否相等, 这个功能很容易用外部迭代器实现, 而几乎无法用内部迭代器实现。在象 C++ 这样不提供匿名函数、闭包, 或象 Smalltalk 和 CLOS 这样不提供连续 (continuation) 的语言中, 内部迭代器的弱点更为明显。但另一方面, 内部迭代器的使用较为容易, 因为它们已经定义好了迭代逻辑。

2) 谁定义遍历算法 迭代器不是唯一可定义遍历算法的地方。聚合本身也可以定义遍历算法, 并在遍历过程中用迭代器来存储当前迭代的状态。我们称这种迭代器为一个游标 (cursor), 因为它仅用来指示当前位置。客户会以这个游标为一个参数调用该聚合的 Next 操作, 而 Next 操作将改变这个指示器的状态<sup>②</sup>。

如果迭代器负责遍历算法, 那么将易于在相同的聚合上使用不同的迭代算法, 同时也易于在不同的聚合上重用相同的算法。从另一方面说, 遍历算法可能需要访问聚合的私有变量。如果这样, 将遍历算法放入迭代器中会破坏聚合的封装性。

3) 迭代器健壮程度如何 在遍历一个聚合的同时更改这个聚合可能是危险的。如果在遍

① Booch 分别称外部和内部迭代器为主动 (active) 和被动 (passive) 迭代器 [Boo94]。“主动”和“被动”两个词描述了客户的作用, 而不是指迭代器主动与否。

② 指示器是 Memento 模式的一个简单例子并且有许多和它相同的实现问题。

历聚合的时候增加或删除该聚合元素，可能会导致两次访问同一个元素或者遗漏掉某个元素。一个简单的解决办法是拷贝该聚合，并对该拷贝实施遍历，但一般来说这样做代价太高。

一个健壮的迭代器(robust iterator)保证插入和删除操作不会干扰遍历，且不需拷贝该聚合。有许多方法来实现健壮的迭代器。其中大多数需要向这个聚合注册该迭代器。当插入或删除元素时，该聚合要么调整迭代器的内部状态，要么在内部的维护额外的信息以保证正确的遍历。

Kofler在ET++[Kof93]中对如何实现健壮的迭代器做了很充分的讨论。Murray讨论了如何为USL StandardComponents 列表类实现健壮的迭代器[Mur93]。

4) 附加的迭代器操作 迭代器的最小接口由First、Next、IsDone和CurrentItem<sup>①</sup>操作组成。其他一些操作可能也很有用。例如，对有序的聚合可用一个Previous操作将迭代器定位到前一个元素。SkipTo操作用于已排序并做了索引的聚合中，它将迭代器定位到符合指定条件的元素对象上。

5) 在C++中使用多态的迭代器 使用多态迭代器是有代价的。它们要求用一个Factory Method动态的分配迭代器对象。因此仅当必须多态时才使用它们。否则使用在栈中分配内存的具体的迭代器。

多态迭代器有另一个缺点：客户必须负责删除它们。这容易导致错误，因为你容易忘记释放一个使用堆分配的迭代器对象，当一个操作有多个出口时尤其如此。而且其间如果有异常被触发的话，迭代器对象将永远不会被释放。

Proxy (4.4) 模式提供了一个补救方法。我们可使用一个栈分配的Proxy作为实际迭代器的中间代理。该代理在其析构器中删除该迭代器。这样当该代理生命周期结束时，实际迭代器将同它一起被释放。即使是在发生异常时，该代理机制能保证正确地清除迭代器对象。这就是著名的C++“资源分配即初始化”技术[ES90]的一个应用。下面的代码示例给出了一个例子。

6) 迭代器可有特权访问 迭代器可被看为创建它的聚合的一个扩展。迭代器和聚合紧密耦合。在C++中我们可让迭代器作为它的聚合的一个友元(friend)来表示这种紧密的关系。这样你就不需要在聚合类中定义一些仅为迭代器所使用的操作。

但是，这样的特权访问可能使定义新的遍历变得很难，因为它将要求改变该聚合的接口增加另一个友元。为避免这一问题，迭代器类可包含一些protected操作来访问聚合类的重要的非公共可见的成员。迭代器子类(且只有迭代器子类)可使用这些protected操作来得到对该聚合的特权访问。

7) 用于复合对象的迭代器 在Composite(4.3)模式中的那些递归聚合结构上，外部迭代器可能难以实现，因为在该结构中不同对象处于嵌套聚合的多个不同层次，因此一个外部迭代器为跟踪当前的对象必须存储一条纵贯该Composite的路径。有时使用一个内部迭代器会更容易一些。它仅需递归地调用自己即可，这样就隐式地将路径存储在调用栈中，而无需显式地维护当前对象位置。

如果复合中的节点有一个接口可以从一个节点移到它的兄弟节点、父节点和子节点，那么基于游标的迭代器是个更好的选择。游标只需跟踪当前的节点；它可依赖这种节点接口来遍历

① 甚至可以将Next、IsDone和CurrentItem并入到一个操作中，该操作前进到下一个对象并返回这个对象，如果遍历结束，那么这个操作返回一个特定的值(例如，0)标志该迭代结束。这样我们就使这个接口变得更小了。



该复合对象。

复合常常需要用多种方法遍历。前序，后序，中序以及广度优先遍历都是常用的。你可用不同的迭代器类来支持不同的遍历。

8) 空迭代器 一个空迭代器(NullIterator)是一个退化的迭代器，它有助于处理边界条件。根据定义，一个NullIterator总是已经完成了遍历：即，它的IsDone操作总是返回true。

空迭代器使得更容易遍历树形结构的聚合（如复合对象）。在遍历过程中的每一节点，都可向当前的元素请求遍历其各个子结点的迭代器。该聚合元素将返回一个具体的迭代器。但叶节点元素返回NullIterator的一个实例。这就使我们可以用一种统一的方式实现在整个结构上的遍历。

#### 10. 代码示例

我们将看看一个简单List类的实现，它是我们的基础库(附录C)的一部分。我们将给出两个迭代器的实现，一个以从前到后的次序遍历该表列，而另一个以从后到前的次序遍历(基础库只支持第一种)。然后我们说明如何使用这些迭代器，以及如何避免限定于一种特定的实现。在此之后，我们将改变原来的设计以保证迭代器被正确的删除。最后一个例子示例一个内部迭代器并与其相应的外部迭代器进行比较。

1) 列表和迭代器接口 首先让我们看与实现迭代器相关的部分List接口。完整的接口请参考附录C。

```
template <class Item>
class List {
public:
    List(long size = DEFAULT_LIST_CAPACITY);

    long Count() const;
    Item& Get(long index) const;
    // ...
};
```

该List类通过它的公共接口提供了一个合理的有效的途径以支持迭代。它足以实现这两种遍历。因此没有必再要给迭代器对底层数据结构的访问特权，也就是说，迭代器类不是列表的友元。为确保对不同遍历的透明使用，我们定义一个抽象的迭代器类，它定义了迭代器接口。

```
template <class Item>
class Iterator {
public:
    virtual void First() = 0;
    virtual void Next() = 0;
    virtual bool IsDone() const = 0;
    virtual Item CurrentItem() const = 0;
protected:
    Iterator();
};
```

2) 迭代器子类的实现 列表迭代器是迭代器的一个子类。

```
template <class Item>
class ListIterator : public Iterator<Item> {
public:
    ListIterator(const List<Item>* aList);
    virtual void First();
    virtual void Next();
    virtual bool IsDone() const;
    virtual Item CurrentItem() const;
```



```
private:
    const List<Item>* _list;
    long _current;
};
```

ListIterator的实现简单直接。它存储List和列表当前位置的索引\_current。

```
template <class Item>
ListIterator<Item>::ListIterator (
    const List<Item>* aList
) : _list(aList), _current(0) {
}
```

First将迭代器置于第一个元素：

```
template <class Item>
void ListIterator<Item>::First () {
    _current = 0;
}
```

Next使当前元素向前推进一步：

```
template <class Item>
void ListIterator<Item>::Next () {
    _current++;
}
```

IsDone检查指向当前元素的索引是否超出了列表：

```
template <class Item>
bool ListIterator<Item>::IsDone () const {
    return _current >= _list->Count();
}
```

最后，CurrentItem返回当前索引指向的元素。若迭代已经终止，则抛出一个IteratorOutOfBounds异常：

```
template <class Item>
Item ListIterator<Item>::CurrentItem () const {
    if (IsDone()) {
        throw IteratorOutOfBounds;
    }
    return _list->Get(_current);
}
```

ReverseListIterator的实现是几乎是一样的，只不过它的First操作将\_current置于列表的末尾，而Next操作将\_current减一，向表头的方向前进一步。

3) 使用迭代器 假定有一个雇员（Employee）对象的List，而我们想打印出列表包含的所有雇员的信息。Employee类用一个Print操作来打印本身的信息。为打印这个列表，我们定义一个PrintEmployee操作，此操作以一个迭代器为参数，并使用该迭代器遍历和打印这个列表：

```
void PrintEmployees (Iterator<Employee*>& i) {
    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Print();
    }
}
```

前面我们已经实现了从后向前和从前向后两种遍历的迭代器，我们可用这个操作以两种次序打印雇员信息：

```
List<Employee*>* employees;
// ...
ListIterator<Employee*> forward(employees);
ReverseListIterator<Employee*> backward(employees);
```

```
PrintEmployees(forward);
PrintEmployees(backward);
```

4) 避免限定于一种特定的列表实现 考虑一个List的变体skiplist会对迭代代码产生什么影响。List的SkipList子类必须提供一个实现 Iterator接口的相应的迭代器 SkipListIterator。在内部, 为了进行高效的迭代, SkipListIterator必须保持多个索引。既然 SkipListIterator实现了Iterator, PrintEmployee操作也可用于用SkipList存储的雇员列表。

```
SkipList<Employee*>* employees;
// ...

SkipListIterator<Employee*> iterator(employees);
PrintEmployees(iterator);
```

尽管这种方法是可行的, 但最好能够无需明确指定具体的 List实现(此处即为SkipList)。为此可以引入一个AbstractList类, 它为不同的列表实现给出一个标准接口。List和SkipList成为AbstractList的子类。

为支持多态迭代, AbstractList定义一个Factory Method, 称为CreateIterator。各个列表子类重定义这个方法以返回相应的迭代器。

```
template <class Item>
class AbstractList {
public:
    virtual Iterator<Item>* CreateIterator() const = 0;
    // ...
};
```

另一个办法是定义一个一般的mixin类Traversable, 它定义一个用于创建迭代器接口。聚合类通过混入(继承) Traversable来支持多态迭代。

List重定义CreateIterator, 返回一个ListIterator对象:

```
template <class Item>
Iterator<Item>* List<Item>::CreateIterator () const {
    return new ListIterator<Item>(this);
}
```

现在我们可以写出不依赖于具体列表表示的打印雇员信息的代码。

```
// we know only that we have an AbstractList
AbstractList<Employee*>* employees;
// ...

Iterator<Employee*>* iterator = employees->CreateIterator();
PrintEmployees(*iterator);
delete iterator;
```

5) 保证迭代器被删除 注意CreateCreateIterator返回的是一个动态分配的迭代器对象。在使用完毕后, 必须删除这个迭代器, 否则会造成内存泄漏。为方便客户, 我们提供一个IteratorPtr作为迭代器的代理, 这个机制可以保证在Iterator对象离开作用域时清除它。

IteratorPtr总是在栈上分配<sup>①</sup>。C++自动调用它的析构器, 而该析构器将删除真正的迭代器。IteratorPtr重载了操作符“->”和“\*”, 使得可将IteratorPtr用作一个指向迭代器的指针。IteratorPtr的成员都实现为内联的, 这样它们不会产生任何额外开销。

```
template <class Item>
class IteratorPtr {
```

① 你只需定义私有的new和delete操作符即可在编译时保证这一点。不需要附加的实现。

```

public:
    IteratorPtr(Iterator<Item>* i): _i(i) { }
    ~IteratorPtr() { delete _i; }

    Iterator<Item>* operator->() { return _i; }
    Iterator<Item>& operator*() { return *_i; }
private:
    // disallow copy and assignment to avoid
    // multiple deletions of _i:

    IteratorPtr(const IteratorPtr&);
    IteratorPtr& operator=(const IteratorPtr&);
private:
    Iterator<Item>* _i;
};

IteratorPtr简化了打印代码：
AbstractList<Employee*>* employees;
// ...

IteratorPtr<Employee*> iterator(employees->CreateIterator());
PrintEmployees(*iterator);

```

6) 一个内部的ListIterator 最后，让我们看看一个内部的或被动的ListIterator类是怎么实现的。此时由迭代器来控制迭代，并对列表中的每一个元素施行同一个操作。

问题是如何实现一个抽象的迭代器，可以支持不同的作用于列表各个元素的操作。有些语言支持所谓的匿名函数或闭包，使用这些机制可以较方便地实现抽象的迭代器。但是 C++并不支持这些机制。此时，至少有两种办法可供选择：(1)给迭代器传递一个函数指针(全局的或静态的)。(2)依赖于子类生成。在第一种情况下，迭代器在迭代过程中的每一步调用传递给它的操作，在第二种情况下，迭代器调用子类重定义了的操作以实现一个特定的行为。

这两种选择都不是尽善尽美。常常需要在迭代时累积(accumulate)状态，而用函数来实现这个功能并不太适合；因为我们将不得不使用静态变量来记住这个状态。Iterator子类给我们提供了一个方便的存储累积状态的地方，比如存放在一个实例变量中。但为每一个不同的遍历创建一个子类需要做更多的工作。

下面是第二种实现办法的一个大体框架，它利用了子类生成。这里我们称内部迭代器为一个ListTraverser。

```

template <class Item>
class ListTraverser {
public:
    ListTraverser(List<Item>* aList);
    bool Traverse();
protected:
    virtual bool ProcessItem(const Item&) = 0;
private:
    ListIterator<Item> _iterator;
};

```

ListTraverser以一个List实例为参数。在内部，它使用一个外部ListIterator进行遍历。Traverse启动遍历并对每一元素项调用ProcessItem操作。内部迭代器可在某次ProcessItem操作返回false时提前终止本次遍历。而Traverse返回一个布尔值指示本次遍历是否提前终止。

```

template <class Item>
ListTraverser<Item>::ListTraverser (
    List<Item>* aList
) : _iterator(aList) { }

```

```

template <class Item>
bool ListTraverser<Item>::Traverse () {
    bool result = false;

    for (
        _iterator.First();
        !_iterator.IsDone();
        _iterator.Next()
    ) {
        result = ProcessItem(_iterator.CurrentItem());

        if (result == false) {
            break;
        }
    }
    return result;
}

```

让我们使用一个ListTraverser来打印雇员列表中的头10个雇员。为达到这个目的，必须定义一个ListTraverser的子类并重定义其ProcessItem操作。我们用一个\_count实例变量中对已打印的雇员进行计数。

```

class PrintNEmployees : public ListTraverser<Employee*> {
public:
    PrintNEmployees(List<Employee*>* aList, int n) :
        ListTraverser<Employee*>(aList),
        _total(n), _count(0) { }

protected:
    bool ProcessItem(Employee* const&);
private:
    int _total;
    int _count;
};

bool PrintNEmployees::ProcessItem (Employee* const& e) {
    _count++;
    e->Print();
    return _count < _total;
}

```

下面是PrintNEmployees怎样打印列表中的头10个雇员的代码：

```

List<Employee*>* employees;
// ...

PrintNEmployees pa(employees, 10);
pa.Traverse();

```

注意这里客户不需要说明如何进行迭代循环。整个迭代逻辑可以重用。这是内部迭代器的主要优点。但其实现比外部迭代器要复杂一些，因为必须定义一个新的类。与使用外部迭代器比较：

```

ListIterator<Employee*> i(employees);
int count = 0;

for (i.First(); !i.IsDone(); i.Next()) {
    count++;
    i.CurrentItem()->Print();

    if (count >= 10) {
        break;
    }
}

```

内部迭代器可以封装不同类型的迭代。例如，FilteringListTraverser封装的迭代仅处理能通过测试的那些列表元素：

```
template <class Item>
class FilteringListTraverser {
public:
    FilteringListTraverser(List<Item>* aList);
    bool Traverse();
protected:
    virtual bool ProcessItem(const Item&) = 0;
    virtual bool TestItem(const Item&) = 0;
private:
    ListIterator<Item> _iterator;
};
```

这个类接口除了增加了用于测试的成员函数 TestItem外与ListTraverser相同,它的子类将重定义TestItem以指定所需的测试。

Traverse根据测试的结果决定是否越过当前元素继续遍历：

```
template <class Item>
void FilteringListTraverser<Item>::Traverse () {
    bool result = false;

    for (
        _iterator.First();
        !_iterator.IsDone();
        _iterator.Next()
    ) {
        if (TestItem(_iterator.CurrentItem())) {
            result = ProcessItem(_iterator.CurrentItem());
            if (result == false) {
                break;
            }
        }
    }
    return result;
}
```

这个类的一中变体是让 Traverse返回值指示是否至少有一个元素通过测试<sup>⊖</sup>。

## 11. 已知应用

迭代器在面向对象系统中很普遍。大多数集合类库都以不同的形式提供了迭代器。

这里是一个流行的集合类库——Booch构件[Boo94]中的一个例子，该类库提供了一个队列的两种实现：固定大小的(有界的)实现和动态增长的(无界的)实现。队列的接口由一个抽象的Queue类定义。为了支持不同队列实现上的多态迭代，队列迭代器的实现基于抽象的 Queue类接口。这样做的优点在于，不需要每个队列都实现一个Factory Method来提供合适的迭代器。但是，它要求抽象Queue类的接口的功能足够强大以有效地实现通用迭代器。

在Smalltalk中不需显式定义迭代器。标准的集合类(包, 集合, 字典, 有序集, 字符串, 等等)都定义一个内部迭代器方法 do:，它以一个程序块(即闭包)为参数。集合中的每个元素先被绑定于与程序块中的局部变量，然后该程序块被执行。 Smalltalk也包括一些Stream类，这些Stream类支持一个类似于迭代器的接口。ReadStream实质上是一个迭代器，而且对所有的顺序集合它都可作为一个外部迭代器。对于非顺序的集合类如集合和字典没有标准的外部迭代器。

⊖ 在这些例子中的Traverse操作是一个带原语操作 TestItem和ProcessItem的Template Method。(5.10)

ET++容器类[WGM88]提供了前面讨论的多态迭代器和负责清除迭代器的 Proxy。Unidraw 图形编辑框架使用基于指示器的迭代器 [VL90]。

ObjectWindow2.0[Bor94]为容器提供了一个迭代器类层次。你可对不同的容器类型用相同的方法迭代。ObjectWindow 迭代语法靠重载算后增量算符 ++ 推进迭代。

## 12. 相关模式

Composite(4.3)：迭代器常被应用到象复合这样的递归结构上。

Factory Method(3.3)：多态迭代器靠 Factory Method 来例化适当的迭代器子类。

Memento(5.6)：常与迭代器模式一起使用。迭代器可使用一个 memento 来捕获一个迭代的状态。迭代器在其内部存储 memento。

## 5.5 MEDIATOR(中介者)——对象行为型模式

### 1. 意图

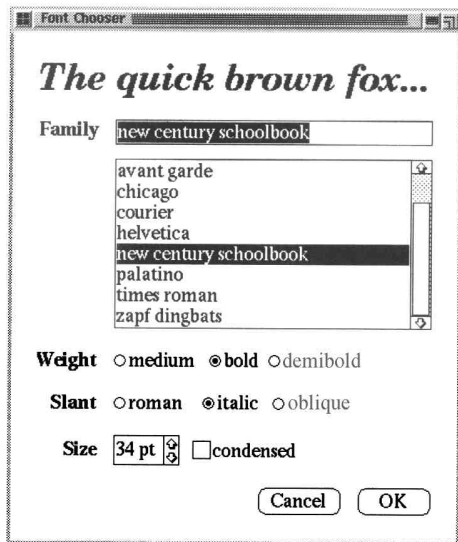
用一个中介对象来封装一系列的对象交互。中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。

### 2. 动机

面向对象设计鼓励将行为分布到各个对象中。这种分布可能会导致对象间有许多连接。在最坏的情况下，每一个对象都知道其他所有对象。

虽然将一个系统分割成许多对象通常可以增强可复用性，但是对象间相互连接的激增又会降低其可复用性。大量的相互连接使得一个对象似乎不太可能在没有其他对象的支持下工作——系统表现为一个不可分割的整体。而且，对系统的行为进行任何较大的改动都十分困难，因为行为被分布在许多对象中。结果是，你可能不得不定义很多子类以定制系统的行为。

例如，考虑一个图形用户界面中对话框的实现。对话框使用一个窗口来展现一系列的窗口组件，如按钮、菜单和输入域等，如下图所示。



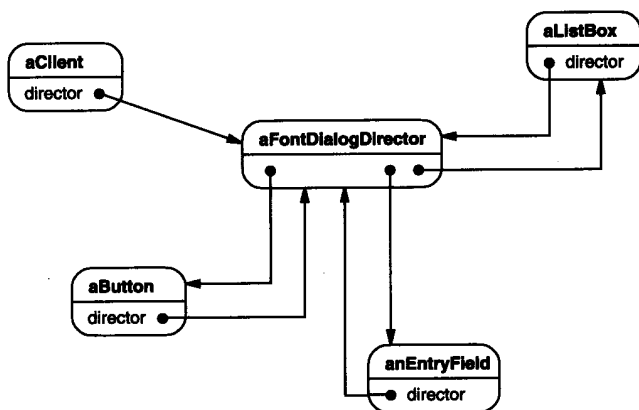
通常对话框中的窗口组件间存在依赖关系。例如，当一个特定的输入域为空时，某个按钮不能使用；在称为列表框的一系列选项中选择一个表目可能会改变一个输入域的内容；反过来，

在输入域中输入正文可能会自动的选择一个或多个列表框中相应的表目；一旦正文出现在输入域中，其他一些按钮可能就变得能够使用了，这些按钮允许用户做一些操作，比如改变或删除这些正文所指的东西。

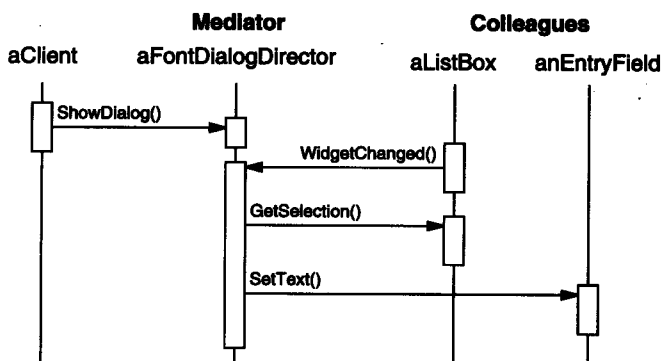
不同的对话框会有不同的窗口组件间的依赖关系。因此即使对话框显示相同类型的窗口组件，也不能简单地直接重用已有的窗口组件类；而必须定制它们以反映特定对话框的依赖关系。由于涉及很多个类，用逐个生成子类的办法来定制它们会很冗长。

可以通过将集体行为封装在一个单独的中介者(mediator)对象中以避免这个问题。中介者负责控制和协调一组对象间的交互。中介者充当一个中介以使组中的对象不再相互显式引用。这些对象仅知道中介者，从而减少了相互连接的数目。

例如，FontDialogDirector可作为一个对话框中的窗口组件间的中介者。FontDialogDirector对象知道对话框中的各窗口组件，并协调它们之间的交互。它充当窗口组件间通信的中转中心，如下图所示。



下面的交互图说明了各对象如何协作处理一个列表框中选项的变化。



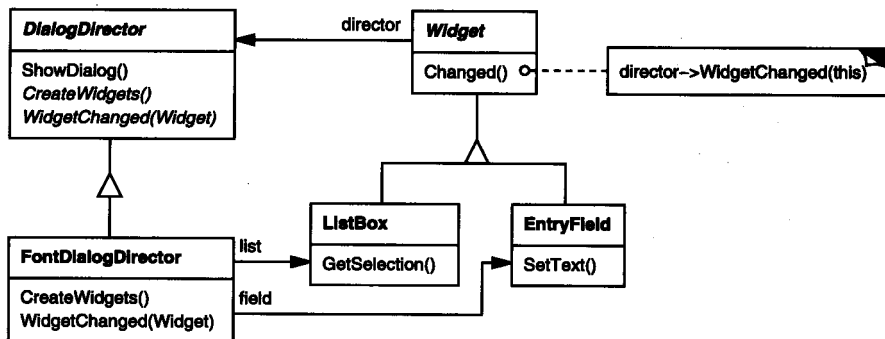
下面一系列事件使一个列表框的选择被传送给一个输入域：

- 1) 列表框告诉它的操作者它被改变了。
- 2) 导控者从列表框中得到选中的选择项。
- 3) 导控者将该选择项传递给入口域。
- 4) 现在入口域已有正文，导控者使得用于发起一个动作（如“半黑体”，“斜体”）的某个（某些）按钮可用。



注意导航者是如何在对话框和入口域间进行中介的。窗口组件间的通信都通过导航者间接地进行。它们不必互相知道；它们仅需知道导航者。而且，由于所有这些行为都局部于一个类中，只要扩展或替换这个类，就可以改变和替换这些行为。

这里展示的是FontDialogDirector抽象怎样被集成到一个类库中，如下图所示。



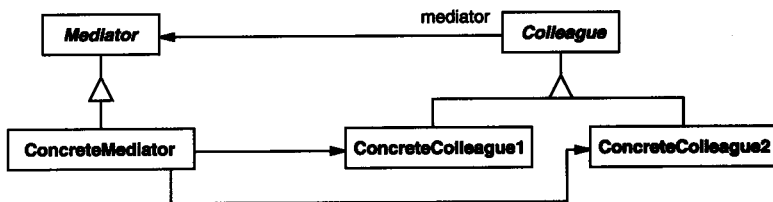
DialogDirector是一个抽象类，它定义了一个对话框的总体行为。客户调用 ShowDialog操作将对话框显示在屏幕上。CreateWidgets是创建一个对话框的窗口组件的抽象操作。WidgetChanged是另一个抽象操作；窗口组件调用它来通知它的导航者它们被改变了。DialogDirector的子类将重定义CreateWidgets以创建正确的窗口组件，并重定义WidgetChanged以处理其变化。

### 3. 适用性

在下列情况下使用中介者模式：

- 一组对象以定义良好但是复杂的方式进行通信。产生的相互依赖关系结构混乱且难以理解。
- 一个对象引用其他很多对象并且直接与这些对象通信，导致难以复用该对象。
- 想定制一个分布在多个类中的行为，而又不想生成太多的子类。

### 4. 结构

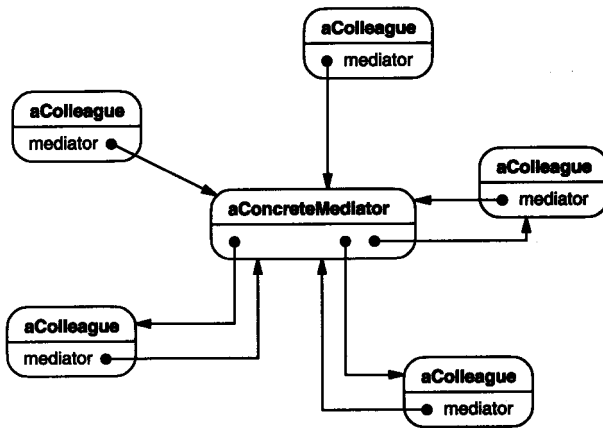


一个典型的对象结构可能如下页图所示。

### 5. 参与者

- Mediator(中介者，如DialogDirector)
  - 中介者定义一个接口用于与各同事（Colleague）对象通信。
- ConcreteMediator(具体中介者，如FontDialogDirector)
  - 具体中介者通过协调各同事对象实现协作行为。
  - 了解并维护它的各个同事。
- Colleague class(同事类，如ListBox, EntryField)

- 每一个同事类都知道它的中介者对象。
- 每一个同事对象在需与其他的同事通信的时候，与它的中介者通信。



## 6. 协作

- 同事向一个中介者对象发送和接收请求。中介者在各同事间适当地转发请求以实现协作行为。

## 7. 效果

中介者模式有以下优点和缺点：

1) 减少了子类生成 Mediator将原本分布于多个对象间的行为集中在一起。改变这些行为只需生成Mediator的子类即可。这样各个Colleague类可被重用。

2) 它将各Colleague解耦 Mediator有利于各Colleague间的松耦合。你可以独立的改变和复用各Colleague类和Mediator类。

3) 它简化了对象协议 用Mediator和各Colleague间的一对多的交互来代替多对多的交互。一对多的关系更易于理解、维护和扩展。

4) 它对对象如何协作进行了抽象 将中介作为一个独立的概念并将其封装在一个对象中，使你注意力从对象各自本身的行为转移到它们之间的交互上来。这有助于弄清楚一个系统中的对象是如何交互的。

5) 它使控制集中化 中介者模式将交互的复杂性变为中介者的复杂性。因为中介者封装了协议，它可能变得比任何一个Colleague都复杂。这可能使得中介者自身成为一个难于维护的庞然大物。

## 8. 实现

下面是与中介者模式有关的一些实现问题：

1) 忽略抽象的Mediator类 当各Colleague仅与一个Mediator一起工作时，没有必要定义一个抽象的Mediator类。Mediator类提供的抽象耦合已经使各Colleague可与不同的Mediator子类一起工作，反之亦然。

2) Colleague——Mediator通信 当一个感兴趣的事件发生时，Colleague必须与其Mediator通信。一种实现方法是使用Observer(5.7)模式，将Mediator实现为一个Observer，各Colleague作为Subject，一旦其状态改变就发送通知给Mediator。Mediator作出的响应是将状态改变的结果传播给其他的Colleague。

另一个方法是在 Mediator 中定义一个特殊的通知接口, 各 Colleague 在通信时直接调用该接口。Windows 下的 Smalltalk/V 使用某种形式的代理机制: 当与 Mediator 通信时, Colleague 将自身作为一个参数传递给 Mediator, 使其可以识别发送者。代码示例一节使用这种方法。而 Smalltalk/V 的实现方法将稍后在已知应用一节中讨论。

### 9. 代码示例

我们将使用一个 DialogDirector 来实现在动机一节中所示的字体对话框。抽象类 DialogDirector 为导航者定义了一个接口。

```
class DialogDirector {
public:
    virtual ~DialogDirector();

    virtual void ShowDialog();
    virtual void WidgetChanged(Widget*) = 0;

protected:
    DialogDirector();
    virtual void CreateWidgets() = 0;
};
```

Widget 是窗口组件的抽象基类。一个窗口组件知道它的导航者。

```
class Widget {
public:
    Widget(DialogDirector*);
    virtual void Changed();

    virtual void HandleMouse(MouseEvent& event);
    // ...

private:
    DialogDirector* _director;
};
```

Changed 调用导航者的 WidgetChanged 操作。通知导航者某个重要事件发生了。

```
void Widget::Changed () {
    _director->WidgetChanged(this);
}
```

DialogDirector 的子类重定义 WidgetChanged 以导航相应的窗口组件。窗口组件把对自身的一个引用作为 WidgetChanged 的参数, 使得导航者可以识别哪个窗口组件改变了。

DialogDirector 子类重定义纯虚函数 CreateWidgets, 在对话框中构建窗口组件。

ListBox、EntryField 和 Button 是 Widget 的子类, 用作特定的用户界面构成元素。ListBox 提供了一个 GetSelection 操作来得到当前的选择项, 而 EntryField 的 SetText 操作则将新的正文放入该域中。

```
class ListBox : public Widget {
public:
    ListBox(DialogDirector*);

    virtual const char* GetSelection();
    virtual void SetList(List<char*>* listItems);
    virtual void HandleMouse(MouseEvent& event);
    // ...
};

class EntryField : public Widget {
public:
```

```

    EntryField(DialogDirector*);

    virtual void SetText(const char* text);
    virtual const char* GetText();
    virtual void HandleMouse(MouseEvent& event);
    // ...
};

```

Button是一个简单的窗口组件，它一旦被按下就调用Changed。这是在其HandleMouse的实现中完成的：

```

class Button : public Widget {
public:
    Button(DialogDirector*);

    virtual void SetText(const char* text);
    virtual void HandleMouse(MouseEvent& event);
    // ...
};

void Button::HandleMouse (MouseEvent& event) {
    // ...
    Changed();
}

```

FontDialogDirector类在对话框中的窗口组件间进行中介。FontDialogDirector是DialogDirector的子类：

```

class FontDialogDirector : public DialogDirector {
public:
    FontDialogDirector();
    virtual ~FontDialogDirector();
    virtual void WidgetChanged(Widget*);

protected:
    virtual void CreateWidgets();

private:
    Button* _ok;
    Button* _cancel;
    ListBox* _fontList;
    EntryField* _fontName;
};

```

FontDialogDirector跟踪它显示的窗口组件。它重定义CreateWidgets以创建窗口组件并初始化对它们的引用：

```

void FontDialogDirector::CreateWidgets () {
    _ok = new Button(this);
    _cancel = new Button(this);
    _fontList = new ListBox(this);
    _fontName = new EntryField(this);

    // fill the listBox with the available font names

    // assemble the widgets in the dialog
}

```

WidgetChanged保证窗口组件正确地协同工作：

```

void FontDialogDirector::WidgetChanged (
    Widget* theChangedWidget
) {
    if (theChangedWidget == _fontList) {

```

```
_fontName->SetText(_fontList->GetSelection());
```

```
    } else if (theChangedWidget == _ok) {
        // apply font change and dismiss dialog
        // ...
    } else if (theChangedWidget == _cancel) {
        // dismiss dialog
    }
}
```

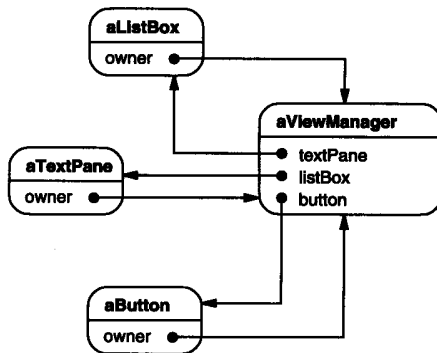
WidgetChanged的复杂度随对话框的复杂度的增加而增加。在实践中，大对话框并不受欢迎，其原因是多方面的，其中一个重要原因是中介者的复杂性可能会抵消该模式在其他方面的带来的好处。

#### 10. 已知应用

ET++[WGM88]和THINK C类库[Sym93b]都在对话框中使用类似导控者的对象作为窗口组件间的中介者。

Windows下的Smalltalk/V的应用结构基于中介者结构[LaL94]。在这个环境中，一个应用由一个包含一组窗格 (pane)的窗口组成。该类库包含若干预定义的 Pane对象；比如说TextPane、ListBox、Button，等等。这些窗格无需继承即可直接使用。应用开发者仅需由 ViewManager衍生子类，ViewManager类负责窗格间的协调工作。ViewManager是一个中介者，而每一个窗格只知道它的view manager，它被看作该窗格的“主人”。窗格不直接互相引用。

下面的对象图显示了一个应用运行时刻的情景。



Smalltalk/V的Pane-ViewManager通信使用一种事件机制。当一个窗格想从中介者得到信息或当它想通知中介者一些重要的事情发生时，它产生一个事件。事件定义一个符号（如#select）来标识该事件。为处理该事件，视管理者为该窗格注册一个候选方法。这个方法是该事件的处理程序；一旦该事件发生它就会被调用。

下面的代码片段说明了在一个 ViewManager子类中，一个 ListPane对象如何被创建以及 ViewManager如何为#select事件注册一个事件处理程序：

```
self addSubpane: (ListPane new
    paneName: 'myListPane';
    owner: self;
    when: #select perform: #listSelect:).
```

另一个中介者模式的应用是用于协调复杂的更新。一个例子是在 Observer(5.7)中提到的 ChangeManager类。ChangeManager在subject和Observer间进行协调以避免冗余的更新。当一

一个对象改变时，它通知ChangeManager，ChangeManager随即通知依赖于该对象的那些对象以协调这个更新。

一个类似的应用出现在Unidraw绘图框架[VL90]中，它使用一个称为CSolver的类来实现“连接器”间的连接约束。图形编辑器中的对象可用不同的方式表现出相互依附。连接器用于自动维护连接的应用中，如框图编辑器和电路设计系统。CSolver是连接器间的中介者。它解释连接约束并更新连接器的位置以反映这些约束。

### 11. 相关模式

Facade(4.5)与中介者的不同之处在于它是对一个对象子系统进行抽象，从而提供了一个更为方便的接口。它的协议是单向的，即Facade对象对这个子系统类提出请求，但反之则不行。相反，Mediator提供了各Colleague对象不支持或不能支持的协作行为，而且协议是多向的。

Colleague可使用Observer(5.7)模式与Mediator通信。

## 5.6 MEMENTO（备忘录）——对象行为型模式

### 1. 意图

在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样以后就可将该对象恢复到原先保存的状态。

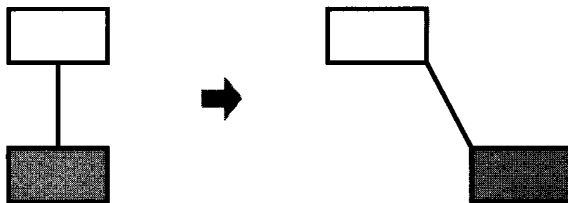
### 2. 别名

Token

### 3. 动机

有时有必要记录一个对象的内部状态。为了允许用户取消不确定的操作或从错误中恢复过来，需要实现检查点和取消机制，而要实现这些机制，你必须事先将状态信息保存在某处，这样才能将对象恢复到它们先前的状态。但是对象通常封装了其部分或所有的状态信息，使得其状态不能被其他对象访问，也就不可能在该对象之外保存其状态。而暴露其内部状态又将违反封装的原则，可能有损应用的可靠性和可扩展性。

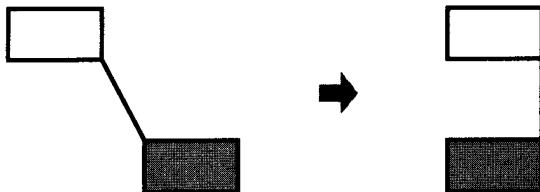
例如，考虑一个图形编辑器，它支持图形对象间的连线。用户可用一条直线连接两个矩形，而当用户移动任意一个矩形时，这两个矩形仍能保持连接。在移动过程中，编辑器自动伸展这条直线以保持该连接。



一个众所周知的保持对象间连接关系的方法是使用一个约束解释系统。我们可将这一功能封装在一个ConstraintSolver对象中。ConstraintSolver在连接生成时，记录这些连接并产生描述它们的数学方程。当用户生成一个连接或修改图形时，ConstraintSolver就求解这些方程。并根据它的计算结果重新调整图形，使各个对象保持正确的连接。

在这一应用中，支持取消操并不象看起来那么容易。一个显而易见的方法是，每次移动时

保存移动的距离，而在取消这次移动时该对象移回相等的距离。然而，这不能保证所有的对象都会出现在它们原先出现的地方。设想在移动过程中某连接中有一些松弛。在这种情况下，简单地将矩形移回它原来的位置并不一定能得到预想的结果。



一般来说, ConstraintSolver的公共接口可能不足以精确地逆转它对其他对象的作用。为重建先前的状态, 取消操作机制必须与 ConstraintSolver更紧密的结合, 但我们同时也应避免将 ConstraintSolver的内部暴露给取消操作机制。

我们可用备忘录(Memento)模式解决这一问题。一个备忘录(memento)是一个对象, 它存储另一个对象在某个瞬间的内部状态, 而后者称为备忘录的原发器(originator)。当需要设置原发器的检查点时, 取消操作机制会向原发器请求一个备忘录。原发器用描述当前状态的信息初始化该备忘录。只有原发器可以向备忘录中存取信息, 备忘录对其他的对象“不可见”。

在刚才讨论的图形编辑器的例子中, ConstraintSolver可作为一个原发器。下面的事件序列描述了取消操作的过程:

- 1) 作为移动操作的一个副作用, 编辑器向 ConstraintSolver请求一个备忘录。
- 2) ConstraintSolver创建并返回一个备忘录, 在这个例子中该备忘录是 SolverState类的一个实例。SolverState备忘录包含一些描述 ConstraintSolver的内部等式和变量当前状态的数据结构。
- 3) 此后当用户取消移动操作时, 编辑器将 SolverState备忘录送回给 ConstraintSolver。
- 4) 根据 SolverState备忘录中的信息, ConstraintSolver改变它的内部结构以精确地将它的等式和变量返回到它们各自先前的状态。

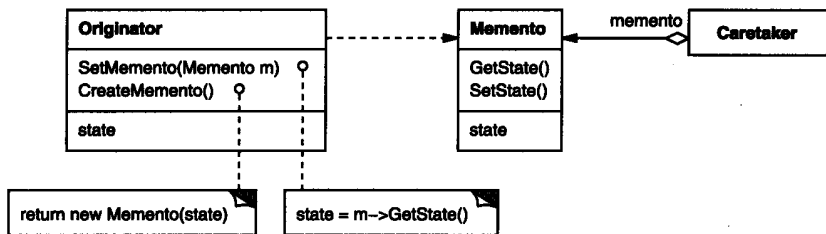
这一方案允许 ConstraintSolver把恢复先前状态所需的信息交给其他的对象, 而又不暴露它的内部结构和表示。

#### 4. 适用性

在以下情况下使用备忘录模式:

- 必须保存一个对象在某一个时刻的(部分)状态, 这样以后需要时它才能恢复到先前的状态。
- 如果一个用接口来让其它对象直接得到这些状态, 将会暴露对象的实现细节并破坏对象的封装性。

#### 5. 结构





## 6. 参与者

- Memento(备忘录, 如 SolverState)

- 备忘录存储原发器对象的内部状态。原发器根据需要决定备忘录存储原发器的哪些内部状态。
- 防止原发器以外的其他对象访问备忘录。备忘录实际上有两个接口, 管理者 (caretaker) 只能看到备忘录的窄接口——它只能将备忘录传递给其他对象。相反, 原发器能够看到一个宽接口, 允许它访问返回到先前状态所需的所有数据。理想的情况是只允许生成本备忘录的那个原发器访问本备忘录的内部状态。

- Originator(原发器, 如 ConstraintSolver)

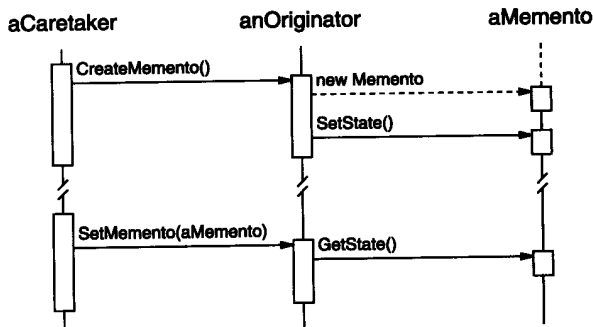
- 原发器创建一个备忘录, 用以记录当前时刻它的内部状态。
- 使用备忘录恢复内部状态。

- Caretaker(负责人, 如 undo mechanism)

- 负责保存好备忘录。
- 不能对备忘录的内容进行操作或检查。

## 7. 协作

- 管理者向原发器请求一个备忘录, 保留一段时间后, 将其送回给原发器, 如下面的交互图所示。



有时管理者不会将备忘录返回给原发器, 因为原发器可能根本不需要退到先前的状态。

- 备忘录是被动的。只有创建备忘录的原发器会对它的状态进行赋值和检索。

## 8. 效果

备忘录模式有以下一些效果:

1) 保持封装边界 使用备忘录可以避免暴露一些只应由原发器管理却又必须存储在原发器之外的信息。该模式把可能很复杂的 Originator 内部信息对其他对象屏蔽起来, 从而保持了封装边界。

2) 它简化了原发器 在其他的保持封装性的设计中, Originator 负责保持客户请求过的内部状态版本。这就把所有存储管理的重任交给了 Originator。让客户管理它们请求的状态将会简化 Originator, 并且使得客户工作结束时无需通知原发器。

3) 使用备忘录可能代价很高 如果原发器在生成备忘录时必须拷贝并存储大量的信息, 或者客户非常频繁地创建备忘录和恢复原发器状态, 可能会导致非常大的开销。除非封装和恢复 Originator 状态的开销不大, 否则该模式可能并不合适。参见实现一节中关于增量式改变的

讨论。

4) 定义窄接口和宽接口 在一些语言中可能难以保证只有原发器可访问备忘录的状态。

5) 维护备忘录的潜在代价 管理器负责删除它所维护的备忘录。然而，管理器不知道备忘录中有多少个状态。因此当存储备忘录时，一个本来很小的管理器，可能会产生大量的存储开销。

## 9. 实现

下面是当实现备忘录模式时应考虑的两个问题：

1) 语言支持 备忘录有两个接口：一个为原发器所使用的宽接口，一个为其他对象所使用的窄接口。理想的实现语言应可支持两级的静态保护。在 C++ 中，可将 Originator 作为 Memento 的一个友元，并使 Memento 宽接口为私有的。只有窄接口应该被声明为公共的。例如：

```
class State;

class Originator {
public:
    Memento* CreateMemento();
    void SetMemento(const Memento*);
    // ...
private:
    State* _state;      // internal data structures
    // ...
};

class Memento {
public:
    // narrow public interface
    virtual ~Memento();
private:
    // private members accessible only to Originator
    friend class Originator;
    Memento();

    void SetState(State*);
    State* GetState();
    // ...
private:
    State* _state;
    // ...
};
```

2) 存储增量式改变 如果备忘录的创建及其返回（给它们的原发器）的顺序是可预测的，备忘录可以仅存储原发器内部状态的增量改变。

例如，一个包含可撤消的命令的历史列表可使用备忘录以保证当命令被取消时，它们可以被恢复到正确的状态（参见 Command(5.2)）。历史列表定义了一个特定的顺序，按照这个顺序命令可以被取消和重做。这意味着备忘录可以只存储一个命令所产生的增量改变而不是它所影响的每一个对象的完整状态。在前面动机一节给出的例子中，约束解释器可以仅存储那些变化了的内部结构，以保持直线与矩形相连，而不是存储这些对象的绝对位置。

## 10. 代码示例

此处给出的 C++ 代码展示的是前面讨论过的 ConstraintSolver 的例子。我们使用 MoveCommand 命令对象（参见 Command(5.2)）来执行（取消）一个图形对象从一个位置到另一个位置的移动变换。图形编辑器调用命令对象的 Execute 操作来移动一个图形对象，而用 Unexecute 来

取消该移动。命令对象存储它的目标、移动的距离和一个 ConstraintSolverMemento 的实例,它是一个包含约束解释器状态的备忘录。

```
class Graphic;
// base class for graphical objects in the graphical editor

class MoveCommand {
public:
    MoveCommand(Graphic* target, const Point& delta);
    void Execute();
    void Unexecute();
private:
    ConstraintSolverMemento* _state;
    Point _delta;
    Graphic* _target;
};
```

连接约束由 ConstraintSolver 类创建。它的关键成员函数是 Solve, 它解释那些由 AddConstraint 操作注册的约束。为支持取消操作, ConstraintSolver 用 CreateMemento 操作将自身状态存储在外部的一个 ConstraintSolverMemento 实例中。调用 SetMemento 可使约束解释器返回到先前某个状态。ConstraintSolver 是一个 Singleton(3.5)。

```
class ConstraintSolver {
public:
    static ConstraintSolver* Instance();

    void Solve();
    void AddConstraint(
        Graphic* startConnection, Graphic* endConnection
    );
    void RemoveConstraint(
        Graphic* startConnection, Graphic* endConnection
    );
    ConstraintSolverMemento* CreateMemento();
    void SetMemento(ConstraintSolverMemento*);
private:
    // nontrivial state and operations for enforcing
    // connectivity semantics
};

class ConstraintSolverMemento {
public:
    virtual ~ConstraintSolverMemento();
private:
    friend class ConstraintSolver;
    ConstraintSolverMemento();

    // private constraint solver state
};
```

给定这些接口, 我们可以实现 MoveCommand 的成员函数 Execute 和 Unexecute 如下:

```
void MoveCommand::Execute () {
    ConstraintSolver* solver = ConstraintSolver::Instance();
    _state = solver->CreateMemento(); // create a memento
    _target->Move(_delta);
    solver->Solve();
}

void MoveCommand::Unexecute () {
```

```

ConstraintSolver* solver = ConstraintSolver::Instance();
_target->Move(-_delta);
solver->SetMemento(_state); // restore solver state
solver->Solve();
}

```

Execute在移动图形前先获取一个ConstraintSolverMemento备忘录。Unexecute先将图形移回,再将约束解释器的状态设回原先的状态,并最后让约束解释器解释这些约束。

### 11. 已知应用

前面的代码示例是来自于Unidraw中通过Csolver类[VL90]实现的对连接的支持。

Dylan中的Collection[App92]提供了一个反映备忘录模式的迭代接口。Dylan的集合有一个“状态”对象的概念,它是一个表示迭代状态的备忘录。每一个集合可以按照它所选择的任意方式表示迭代的当前状态;该表示对客户完全不可见。Dylan的迭代方法转换为C++可表示如下:

```

template <class Item>
class Collection {
public:
    Collection();

    IterationState* CreateInitialState();
    void Next (IterationState*);
    bool IsDone(const IterationState*) const;
    Item CurrentItem(const IterationState*) const;
    IterationState* Copy(const IterationState*) const;

    void Append(const Item&);
    void Remove(const Item&);
    // ...
};

```

CreateInitialState为该集合返回一个已初始化的IterationState对象。Next将状态对象推进到迭代的下一个位置;实际上它将迭代索引加一。如果Next已经超出集合中的最后一个元素,IsDone返回true。CurrentItem返回状态对象当前所指的那个元素。Copy返回给定状态对象的一个拷贝。这可用来标记迭代过程中的某一点。

给定一个类ItemType,我们可以象下面这样在它的实例的集合上进行迭代<sup>⊖</sup>:

```

class ItemType {
public:
    void Process();
    // ...
};

Collection<ItemType*> aCollection;
IterationState* state;

state = aCollection.CreateInitialState();

while (!aCollection.IsDone(state)) {
    aCollection.CurrentItem(state)->Process();
    aCollection.Next(state);
}
delete state;

```

⊖ 注意我们在迭代的最后删除该状态对象。但如果ProcessItem抛出一个异常,delete将不会被调用,这样就产生了垃圾。在C++中这是一个问题,但在Dylan中则没有这个问题,因为Dylan有垃圾回收机制。我们在第5章讨论了这个问题的一个解决方法。

基于备忘录的迭代接口有两个有趣的优点：

1) 在同一个集合上中可有多个状态一起工作。(Iterator(5.4)模式也是这样。)

2) 它不需要为支持迭代而破坏一个集合的封装性。备忘录仅由集合自身来解释；任何其他对象都不能访问它。支持迭代的其它方法要求将迭代器类作为它们的集合类的友元（参见 Iterator(5.4)），从而破坏了封装性。这一情况在基于备忘录的实现中不再存在，此时 Collection 是 IteratorState 的一个友元。

QOCA约束解释工具在备忘录中存储增量信息 [HHMV92]。客户可得到刻画某约束系统当前解释的备忘录。该备忘录仅包括从上一次解释以来发生改变的那些约束变量。通常每次新的解释仅有一小部分解释器变量发生改变。这个发生变化的变量子集已足以将解释器恢复到先前的解释；恢复更前的解释要求经过中间的解释逐步地恢复。所以不能以任意的顺序设定备忘录；QOCA依赖一种历史机制来恢复到先前的解释。

## 12. 相关模式

Command(5.2): 命令可使用备忘录来为可撤销的操作维护状态。

Iterator(5.4): 如前所述备忘录可用于迭代。

## 5.7 OBSERVER (观察者) ——对象行为型模式

### 1. 意图

定义对象间的一种一对多的依赖关系,当一个对象的状态发生改变时,所有依赖于它的对象都得到通知并被自动更新。

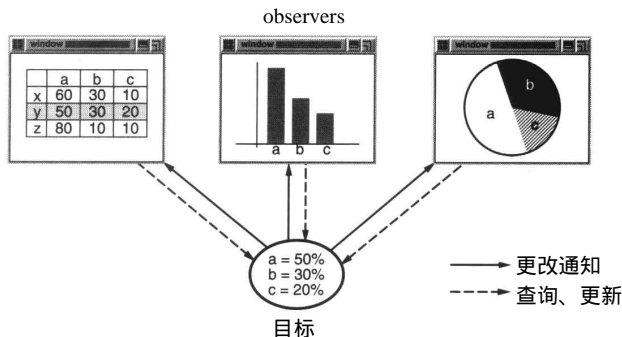
### 2. 别名

依赖(Dependents), 发布-订阅(Publish-Subscribe)

### 3. 动机

将一个系统分割成一系列相互协作的类有一个常见的副作用：需要维护相关对象间的一致性。我们不希望为了维持一致性而使各类紧密耦合，因为这样降低了它们的可重用性。

例如，许多图形用户界面工具箱将用户应用的界面表示与底下的应用数据分离 [KP88, LVC89, P+88, WGM88]。定义应用数据的类和负责界面表示的类可以各自独立地复用。当然它们也可一起工作。一个表格对象和一个柱状图对象可使用不同的表示形式描述同一个应用数据对象的信息。表格对象和柱状图对象互相并不知道对方的存在，这样使你可以根据需要单独复用表格或柱状图。但在这里是它们表现的似乎互相知道。当用户改变表格中的信息时，柱状图能立即反映这一变化，反过来也是如此。



这一行为意味着表格对象和棒状图对象都依赖于数据对象，因此数据对象的任何状态改变都应立即通知它们。同时也没有理由将依赖于该数据对象的对象的数目限定为两个，对相同的数据可以有任意数目的不同用户界面。

Observer模式描述了如何建立这种关系。这一模式中的关键对象是目标(subject)和观察者(observer)。一个目标可以有任意数目的依赖它的观察者。一旦目标的状态发生改变，所有的观察者都得到通知。作为对这个通知的响应，每个观察者都将查询目标以使其状态与目标的状态同步。

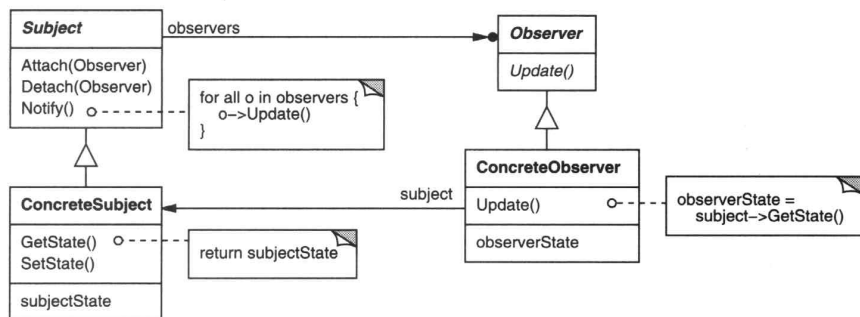
这种交互也称为发布 - 订阅 (publish-subscribe)。目标是通知的发布者。它发出通知时并不需知道谁是它的观察者。可以有任意数目的观察者订阅并接收通知。

#### 4. 适用性

在以下任一情况下可以使用观察者模式：

- 当一个抽象模型有两个方面，其中一个方面依赖于另一方面。将这二者封装在独立的对象中以使它们可以各自独立地改变和复用。
- 当对一个对象的改变需要同时改变其它对象，而不知道具体有多少对象有待改变。
- 当一个对象必须通知其它对象，而它又不能假定其它对象是谁。换言之，你不希望这些对象是紧密耦合的。

#### 5. 结构



#### 6. 参与者

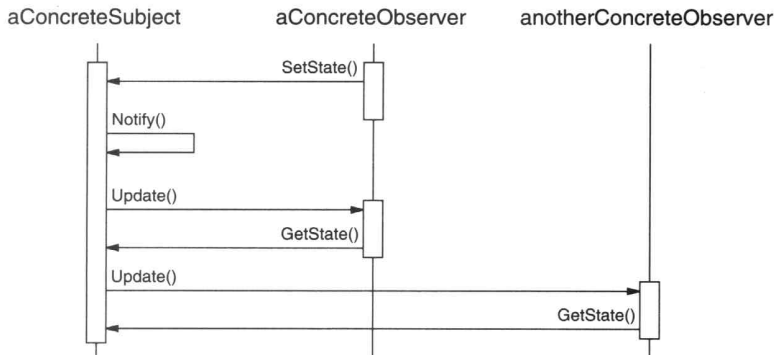
- Subject (目标)
  - 目标知道它的观察者。可以有任意多个观察者观察同一个目标。
  - 提供注册和删除观察者对象的接口。
- Observer (观察者)
  - 为那些在目标发生改变时需获得通知的对象定义一个更新接口。
- ConcreteSubject (具体目标)
  - 将有关状态存入各 ConcreteObserver 对象。
  - 当它的状态发生改变时，向它的各个观察者发出通知。
- ConcreteObserver (具体观察者)
  - 维护一个指向 ConcreteSubject 对象的引用。
  - 存储有关状态，这些状态应与目标的状态保持一致。
  - 实现 Observer 的更新接口以使自身状态与目标的状态保持一致。



## 7. 协作

- 当 ConcreteSubject 发生任何可能导致其观察者与其本身状态不一致的改变时，它将通知它的各个观察者。
- 在得到一个具体目标的改变通知后，ConcreteObserver 对象可向目标对象查询信息。ConcreteObserver 使用这些信息以使它的状态与目标对象的状态一致。

下面的交互图说明了一个目标对象和两个观察者之间的协作：



注意发出改变请求的 Observer 对象并不立即更新，而是将其推迟到它从目标得到一个通知之后。Notify 不总是由目标对象调用。它也可被一个观察者或其它对象调用。实现一节将讨论一些常用的变化。

## 8. 效果

Observer 模式允许你独立的改变目标和观察者。你可以单独复用目标对象而无需同时复用其观察者，反之亦然。它也使你可以在不改动目标和其他的观察者的前提下增加观察者。

下面是观察者模式其它一些优缺点：

1) 目标和观察者间的抽象耦合 一个目标所知道的仅仅是它有一系列观察者，每个都符合抽象的 Observer 类的简单接口。目标不知道任何一个观察者属于哪一个具体的类。这样目标和观察者之间的耦合是抽象的和最小的。

因为目标和观察者不是紧密耦合的，它们可以属于一个系统中的不同抽象层次。一个处于较低层次的目标对象可与一个处于较高层次的观察者通信并通知它，这样就保持了系统层次的完整。如果目标和观察者混在一块，那么得到的对象要么横贯两个层次（违反了层次性），要么必须放在这两层的某一层中（这可能会损害层次抽象）。

2) 支持广播通信 不像通常的请求，目标发送的通知不需指定它的接收者。通知被自动广播给所有已向该目标对象登记的有关对象。目标对象并不关心到底有多少对象对自己感兴趣；它唯一的责任就是通知它的各观察者。这给了你在任何时刻增加和删除观察者的自由。处理还是忽略一个通知取决于观察者。

3) 意外的更新 因为一个观察者并不知道其它观察者的存在，它可能对改变目标的最终代价一无所知。在目标上一个看似无害的操作可能会引起一系列对观察者以及依赖于这些观察者的那些对象的更新。此外，如果依赖准则的定义或维护不当，常常会引起错误的更新，这种错误通常很难捕捉。

简单的更新协议不提供具体细节说明目标中什么被改变了，这就使得上述问题更加严重。如果没有其他协议帮助观察者发现什么发生了改变，它们可能会被迫尽力减少改变。



## 9. 实现

这一节讨论一些与实现依赖机制相关的问题。

1) 创建目标到其观察者之间的映射 一个目标对象跟踪它应通知的观察者的最简单的方法是显式地在目标中保存对它们的引用。然而,当目标很多而观察者较少时,这样存储可能代价太高。一个解决办法是用时间换空间,用一个关联查找机制(例如一个hash表)来维护目标到观察者的映射。这样一个没有观察者的目标就不产生存储开销。但另一方面,这一方法增加了访问观察者的开销。

2) 观察多个目标 在某些情况下,一个观察者依赖于多个目标可能是有意义的。例如,一个表格对象可能依赖于多个数据源。在这种情况下,必须扩展 Update 接口以使观察者知道是哪一个目标送来的通知。目标对象可以简单地将自己作为 Update 操作的一个参数,让观察者知道应去检查哪一个目标。

3) 谁触发更新 目标和它的观察者依赖于通知机制来保持一致。但到底哪一个对象调用 Notify 来触发更新? 此时有两个选择:

a) 由目标对象的状态设定操作在改变目标对象的状态后自动调用 Notify。这种方法的优点是客户不需要记住要在目标对象上调用 Notify,缺点是多个连续的操作会产生多次连续的更新,可能效率较低。

b) 让客户负责在适当的时候调用 Notify。这样做的优点是客户可以在一系列的状态改变完成后再一次性地触发更新,避免了不必要的中间更新。缺点是给客户增加了触发更新的责任。由于客户可能会忘记调用 Notify,这种方式较易出错。

4) 对已删除目标的悬挂引用 删除一个目标时应注意不要在其观察者中遗留对该目标的悬挂引用。一种避免悬挂引用的方法是,当一个目标被删除时,让它通知它的观察者将对该目标的引用复位。一般来说,不能简单地删除观察者,因为其他的对象可能会引用它们,或者也可能它们还在观察其他的目标。

5) 在发出通知前确保目标的状态自身是一致的 在发出通知前确保状态自身一致这一点很重要,因为观察者在更新其状态的过程中需要查询目标的当前状态。

当 Subject 的子类调用继承的该项操作时,很容易无意中违反这条自身一致的准则。例如,下面的代码序列中,在目标尚处于一种不一致的状态时,通知就被触发了:

```
void MySubject::Operation (int newValue) {
    BaseClassSubject::Operation(newValue);
    // trigger notification

    _myInstVar += newValue;
    // update subclass state (too late!)
}
```

你可以用抽象的 Subject 类中的模板方法 (Template Method(5.10)) 发送通知来避免这种错误。定义那些子类可以重定义的原语操作,并将 Notify 作为模板方法中的最后一个操作,这样当子类重定义了 Subject 的操作时,还可以保证该对象的状态是自身一致的。

```
void Text::Cut (TextRange r) {
    ReplaceRange(r);          // redefined in subclasses
    Notify();
}
```

顺便提一句,在文档中记录是哪一个 Subject 操作触发通知总是应该的。

6) 避免特定于观察者的更新协议——推/拉模型 观察者模式的实现经常需要让目标广播关于其改变的其他一些信息。目标将这些信息作为 Update 操作一个参数传递出去。这些信息的量可能很小，也可能很大。

一个极端情况是，目标向观察者发送关于改变的详细信息，而不管它们需要与否。我们称之为推模型(push model)。另一个极端是拉模型(pull model)；目标除最小通知外什么也不送出，而在此之后由观察者显式地向目标询问细节。

拉模型强调的是目标不知道它的观察者，而推模型假定目标知道一些观察者的需要的信息。推模型可能使得观察者相对难以复用，因为目标对观察者的假定可能并不总是正确的。另一方面。拉模型可能效率较差，因为观察者对象需在没有目标对象帮助的情况下确定什么改变了。

7) 显式地指定感兴趣的改变 你可以扩展目标的注册接口，让各观察者注册为仅对特定事件感兴趣，以提高更新的效率。当一个事件发生时，目标仅通知那些已注册为对该事件感兴趣的观察者。支持这种做法一种途径是，对使用目标对象的方面(aspects)的概念。可用如下代码将观察者对象注册为对目标对象的某特定事件感兴趣：

```
void Subject::Attach(Observer*, Aspect& interest);
```

此处interest指定感兴趣的事件。在通知的时刻，目标将这方面的改变作为 Update 操作的一个参数提供给它的观察者，例如：

```
void Observer::Update(Subject*, Aspect& interest);
```

8) 封装复杂的更新语义 当目标和观察者间的依赖关系特别复杂时，可能需要一个维护这些关系的对象。我们称这样的对象为更改管理器(ChangeManager)。它的目的是尽量减少观察者反映其目标的状态变化所需的工作量。例如，如果一个操作涉及到对几个相互依赖的目标进行改动，就必须保证仅在所有的目标都已更改完毕后，才一次性地通知它们的观察者，而不是每个目标都通知观察者。

ChangeManager有三个责任：

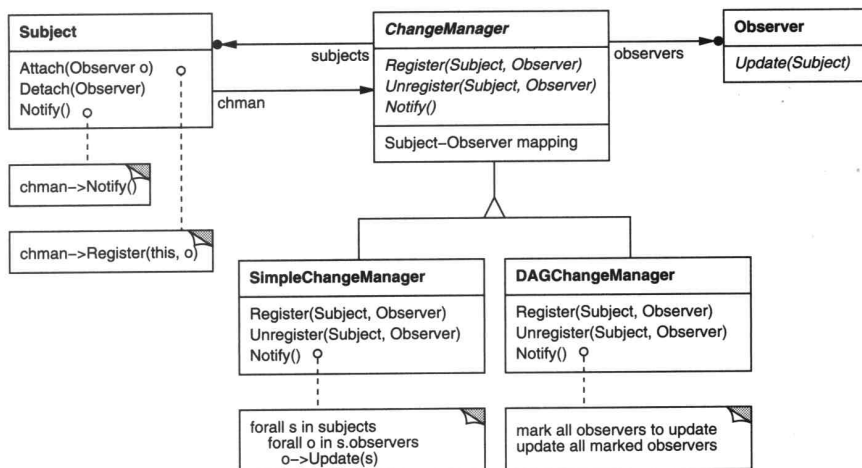
- a) 它将一个目标映射到它的观察者并提供一个接口来维护这个映射。这就不需要由目标来维护对其观察者的引用，反之亦然。
- b) 它定义一个特定的更新策略。
- c) 根据一个目标的请求，它更新所有依赖于这个目标的观察者。

下页的框图描述了一个简单的基于 ChangeManager 的 Observer 模式的实现。有两种特殊的 ChangeManager。SimpleChangeManager总是更新每一个目标的所有观察者，比较简单。相反，DAGChangeManager处理目标及其观察者之间依赖关系构成的无环有向图。当一个观察者观察多个目标时，DAGChangeManager要比SimpleChangeManager更好一些。在这种情况下，两个或更多个目标中产生的改变可能会产生冗余的更新。DAGChangeManager保证观察者仅接收一个更新。当然，当不存在多重更新的问题时，SimpleChangeManager更好一些。

ChangeManager是一个Mediator(5.5)模式的实例。通常只有一个ChangeManager，并且它是全局可见的。这里Singleton(3.5)模式可能有用。

9) 结合目标类和观察者类 用不支持多重继承的语言(如Smalltalk)书写的类库通常不单独定义Subject和Observer类，而是将它们的接口结合到一个类中。这就允许你定义一个既是一个目标又是一个观察者的对象，而不需要多重继承。例如在 Smalltalk 中，Subject和Observer接口

定义于根类 Object 中，使得它们对所有的类都可用。



## 10. 代码示例

一个抽象类定义了 Observer 接口：

```

class Subject;

class Observer {
public:
    virtual ~Observer();
    virtual void Update(Subject* theChangedSubject) = 0;
protected:
    Observer();
};
  
```

这种实现方式支持一个观察者有多个目标。当观察者观察多个目标时，作为参数传递给 Update 操作的目标让观察者可以判定是哪一个目标发生了改变。

类似地，一个抽象类定义了 Subject 接口：

```

class Subject {
public:
    virtual ~Subject();

    virtual void Attach(Observer*);
    virtual void Detach(Observer*);
    virtual void Notify();
protected:
    Subject();
private:
    List<Observer*> *_observers;
};

void Subject::Attach (Observer* o) {
    _observers->Append(o);
}

void Subject::Detach (Observer* o) {
    _observers->Remove(o);
}

void Subject::Notify () {
    ListIterator<Observer*> i(_observers);
  
```

```

    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Update(this);
    }
}

```

ClockTimer是一个用于存储和维护一天时间的具体目标。它每秒钟通知一次它的观察者。ClockTimer提供了一个接口用于取出单个的时间单位如小时，分钟，和秒。

```

class ClockTimer : public Subject {
public:
    ClockTimer();

    virtual int GetHour();
    virtual int GetMinute();
    virtual int GetSecond();

    void Tick();
};

```

Tick操作由一个内部计时器以固定的时间间隔调用，从而提供一个精确的时间基准。Tick更新ClockTimer的内部状态并调用Notify通知观察者：

```

void ClockTimer::Tick () {
    // update internal time-keeping state
    // ...
    Notify();
}

```

现在我们可以定义一个 DigitalClock类来显示时间。它从一个用户界面工具箱提供的 Widget类继承了它的图形功能。通过继承 Observer, Observer接口被融入DigitalClock的接口。

```

class DigitalClock: public Widget, public Observer {
public:
    DigitalClock(ClockTimer*);
    virtual ~DigitalClock();

    virtual void Update(Subject*);
        // overrides Observer operation

    virtual void Draw();
        // overrides Widget operation;
        // defines how to draw the digital clock
private:
    ClockTimer* _subject;
};

DigitalClock::DigitalClock (ClockTimer* s) {
    _subject = s;
    _subject->Attach(this);
}

DigitalClock::~DigitalClock () {
    _subject->Detach(this);
}

```

在Update操作画出时钟图形之前，它进行检查，以保证发出通知的目标是该时钟的目标：

```

void DigitalClock::Update (Subject* theChangedSubject) {
    if (theChangedSubject == _subject) {
        Draw();
    }
}

```

```
void DigitalClock::Draw () {  
    // get the new values from the subject  
  
    int hour = _subject->GetHour();  
    int minute = _subject->GetMinute();  
    // etc.  
  
    // draw the digital clock  
}
```

一个AnalogClock可用相同的方法定义.

```
class AnalogClock : public Widget, public Observer {  
public:  
    AnalogClock(ClockTimer*);  
    virtual void Update(Subject*);  
    virtual void Draw();  
    // ...  
};
```

下面的代码创建一个AnalogClock和一个DigitalClock, 它们总是显示相同时间:

```
ClockTimer* timer = new ClockTimer;  
AnalogClock* analogClock = new AnalogClock(timer);  
DigitalClock* digitalClock = new DigitalClock(timer);
```

一旦timer走动, 两个时钟都会被更新并正确地重新显示.

#### 11. 已知应用

最早的可能也是最著名的 Observer模式的例子出现在 Smalltalk的Model/View/Controller(MVC)结构中, 它是Smalltalk环境[KP88]中的用户界面框架。MVC的Model类担任目标的角色, 而View是观察者的基类。Smalltalk, ET++[WGM88], 和THINK类库[Sym93b]都将Subject和Observer接口放入系统中所有其他类的父类中, 从而提供一个通用的依赖机制。

其他的使用这一模式的用户界面工具有 InterViews[LVC89], Andrew Toolkit[P+88]和Unidraw[VL90]。InterViews显式地定义了Observer和Observable(目标)类。Andrew分别称它们为“视”和“数据对象”。Unidraw将图形编辑器对象分割成 View(观察者)和Subject两部分。

#### 12. 相关模式

Mediator(5.5): 通过封装复杂的更新语义, ChangeManager充当目标和观察者之间的中介者。

Singleton(3.5): ChangeManager可使用Singleton模式来保证它是唯一的并且是可全局访问的。

## 5.8 STATE (状态) ——对象行为型模式

#### 1. 意图

允许一个对象在其内部状态改变时改变它的行为。对象看起来似乎修改了它的类。

#### 2. 别名

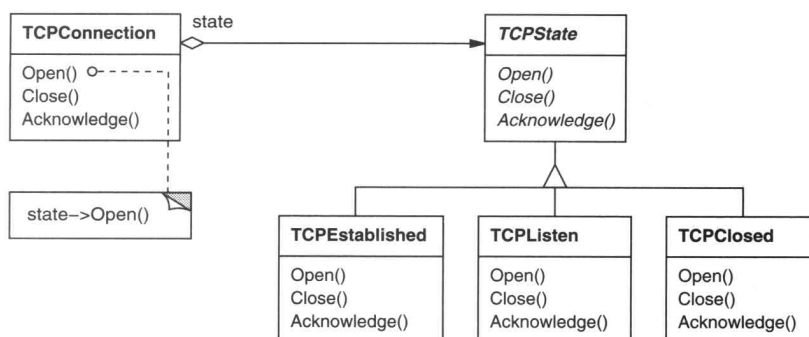
状态对象 ( Objects for States )

#### 3. 动机

考虑一个表示网络连接的类 TCPConnection。一个TCPConnection对象的状态处于若干不同状态之一: 连接已建立 ( Established )、正在监听(Listening)、连接已关闭(Closed)。当一个TCPConnection对象收到其他对象的请求时, 它根据自身的当前状态作出不同的反应。例如,

一个Open请求的结果依赖于该连接是处于连接已关闭状态还是连接已建立状态。State模式描述了TCPConnection如何在每一种状态下表现出不同的行为。

这一模式的关键思想是引入了一个称为 TCPState的抽象类来表示网络的连接状态。TCPState类为各表示不同的操作状态的子类声明了一个公共接口。TCPState的子类实现与特定状态相关的行为。例如，TCPEstablished和TCPClosed类分别实现了特定于TCPConnection的连接已建立状态和连接已关闭状态的行为。



TCPConnection类维护一个表示TCP连接当前状态的状态对象（一个TCPState子类的实例）。TCPConnection类将所有与状态相关的请求委托给这个状态对象。TCPConnection使用它的TCPState子类实例来执行特定于连接状态的操作。

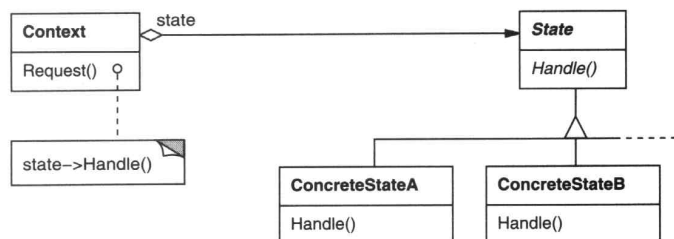
一旦连接状态改变，TCPConnection对象就会改变它所使用状态对象。例如当连接从已建立状态转为已关闭状态时，TCPConnection会用一个TCPClosed的实例来代替原来的TCPEstablished的实例。

#### 4. 适用性

在下面的两种情况下均可使用State模式：

- 一个对象的行为取决于它的状态，并且它必须在运行时刻根据状态改变它的行为。
- 一个操作中含有庞大的多分支的条件语句，且这些分支依赖于该对象的状态。这个状态通常用一个或多个枚举常量表示。通常，有多个操作包含这一相同的条件结构。State模式将每一个条件分支放入一个独立的类中。这使得你可以根据对象自身的情况将对象的状态作为一个对象，这一对象可以不依赖于其他对象而独立变化。

#### 5. 结构



#### 6. 参与者

- Context(环境，如TCPConnection)
  - 定义客户感兴趣的接口。



— 维护一个 ConcreteState 子类的实例，这个实例定义当前状态。

- State(状态，如 TCPState)

— 定义一个接口以封装与 Context 的一个特定状态相关的行为。

- ConcreteState subclasses(具体状态子类，如 TCPEstablished, TCPListen, TCPClosed)

— 每一子类实现一个与 Context 的一个状态相关的行为。

## 7. 协作

- Context 将与状态相关的请求委托给当前的 ConcreteState 对象处理。

- Context 可将自身作为一个参数传递给处理该请求的状态对象。这使得状态对象在必要时可访问 Context。

- Context 是客户使用的主要接口。客户可用状态对象来配置一个 Context，一旦一个 Context 配置完毕，它的客户不再需要直接与状态对象打交道。

- Context 或 ConcreteState 子类都可决定哪个状态是另外哪一个的后继者，以及是在何种条件下进行状态转换。

## 8. 效果

State 模式有下面一些效果：

1) 它将与特定状态相关的行为局部化，并且将不同状态的行为分割开来。State 模式将所有与一个特定的状态相关的行为都放入一个对象中。因为所有与状态相关的代码都存在于某一个 State 子类中，所以通过定义新的子类可以很容易的增加新的状态和转换。

另一个方法是使用数据值定义内部状态并且让 Context 操作来显式地检查这些数据。但这样将会使整个 Context 的实现中遍布看起来很相似的条件语句或 case 语句。增加一个新的状态可能需要改变若干个操作，这就使得维护变得复杂了。

State 模式避免了这个问题，但可能会引入另一个问题，因为该模式将不同状态的行为分布在多个 State 子类中。这就增加了子类的数目，相对于单个类的实现来说不够紧凑。但是如果有许多状态时这样的分布实际上更好一些，否则需要使用巨大的条件语句。

正如很长的过程一样，巨大的条件语句是不受欢迎的。它们形成一大整块并且使得代码不够清晰，这又使得它们难以修改和扩展。State 模式提供了一个更好的方法来组织与特定状态相关的代码。决定状态转移的逻辑不在单块的 if 或 switch 语句中，而是分布在 State 子类之间。将每一个状态转换和动作封装到一个类中，就把着眼点从执行状态提高到整个对象的状态。这将使代码结构化并使其意图更加清晰。

2) 它使得状态转换显式化。当一个对象仅以内部数据值来定义当前状态时，其状态仅表现为对一些变量的赋值，这不够明确。为不同的状态引入独立的对象使得转换变得更加明确。而且，State 对象可保证 Context 不会发生内部状态不一致的情况，因为从 Context 的角度看，状态转换是原子的——只需重新绑定一个变量（即 Context 的 State 对象变量），而无需为多个变量赋值[dCLF93]。

3) State 对象可被共享。如果 State 对象没有实例变量——即它们表示的状态完全以它们的类型来编码——那么各 Context 对象可以共享一个 State 对象。当状态以这种方式被共享时，它们必然是没有内部状态，只有行为的轻量级对象（参见 Flyweight (4.6)）。

## 9. 实现

实现 State 模式有多方面的考虑：



1) 谁定义状态转换 State模式不指定哪一个参与者定义状态转换准则。如果该准则是固定的,那么它们可在Context中完全实现。然而若让State子类自身指定它们的后继状态以及何时进行转换,通常更灵活更合适。这需要 Context增加一个接口,让State对象显式地设定Context的当前状态。

用这种方法分散转换逻辑可以很容易地定义新的State子类来修改和扩展该逻辑。这样做的一个缺点是,一个State子类至少拥有一个其他子类的信息,这就在各子类之间产生了实现依赖。

2) 基于表的另一种方法 在C++ Programming Style[Car92]中,Cargil描述了另一种将结构加载在状态驱动的代码上的方法:他使用表将输入映射到状态转换。对每一个状态,一张表将每一个可能的输入映射到一个后继状态。实际上,这种方法将条件代码(和State模式下的虚函数)映射为一个查找表。

表的主要好处是它们的规则性:你可以通过更改数据而不是更改程序代码来改变状态转换的准则。然而它也有一些缺点:

- 对表的查找通常不如(虚)函数调用效率高。
- 用统一的、表格的形式表示转换逻辑使得转换准则变得不够明确而难以理解。
- 通常难以加入伴随状态转换的一些动作。表驱动的方法描述了状态和它们之间的转换,但必须扩充这个机制以便在每一个转换上能够进行任意的计算。

表驱动的状态机和State模式的主要区别可以被总结如下:State模式对与状态相关的行为进行建模,而表驱动的方法着重于定义状态转换。

3) 创建和销毁State对象 一个常见的值得考虑的实现上的权衡是,究竟是(1)仅当需要State对象时才创建它们并随后销毁它们,还是(2)提前创建它们并且始终不销毁它们。

当将要进入的状态在运行时是不可知的,并且上下文不经常改变状态时,第一种选择较为可取。这种方法避免创建不会被用到的对象,如果State对象存储大量的信息时这一点很重要。当状态改变很频繁时,第二种方法较好。在这种情况下最好避免销毁状态,因为可能很快再次需要用到它们。此时可以预先一次付清创建各个状态对象的开销,并且在运行过程中根本不存在销毁状态对象的开销。但是这种方法可能不太方便,因为Context必须保存对所有可能会进入的那些状态的引用。

4) 使用动态继承 改变一个响应特定请求的行为可以用在运行时刻改变这个对象的类的办法实现,但这在大多数面向对象程序设计语言中都是不可能的。Self[US87]和其他一些基于委托的语言却是例外,它们提供这种机制,从而直接支持State模式。Self中的对象可将操作委托给其他对象以达到某种形式的动态继承。在运行时刻改变委托的目标有效地改变了继承的结构。这一机制允许对象改变它们的行为,也就是改变它们的类。

#### 10. 代码示例

下面的例子给出了在动机一节描述的TCP连接例子的C++代码。这个例子是TCP协议的一个简化版本,它并未完整描述TCP连接的协议及其所有状态<sup>①</sup>。

首先,我们定义类TCPConnection,它提供了一个传送数据的接口并处理改变状态的请求。

```
class TCPOctetStream;  
class TCPState;
```

① 这个例子基于由Lynch和Rose描述的TCP连接协议[LR93]。

```

class TCPConnection {
public:
    TCPConnection();

    void ActiveOpen();
    void PassiveOpen();
    void Close();
    void Send();
    void Acknowledge();
    void Synchronize();

    void ProcessOctet(TCPOctetStream*);
private:
    friend class TCPState;
    void ChangeState(TCPState*);
private:
    TCPState* _state;
};

```

TCPConnection在\_state成员变量中保持一个 TCPState类的实例。类 TCPState复制了 TCPConnection的状态改变接口。每一个 TCPState操作都以一个 TCPConnection实例作为一个参数,从而让 TCPState可以访问 TCPConnection中的数据 and 改变连接的状态。

```

class TCPState {
public:
    virtual void Transmit(TCPConnection*, TCPOctetStream*);
    virtual void ActiveOpen(TCPConnection*);
    virtual void PassiveOpen(TCPConnection*);
    virtual void Close(TCPConnection*);
    virtual void Synchronize(TCPConnection*);
    virtual void Acknowledge(TCPConnection*);
    virtual void Send(TCPConnection*);
protected:
    void ChangeState(TCPConnection*, TCPState*);
};

```

TCPConnection将所有与状态相关的请求委托给它的 TCPState实例\_state。TCPConnection还提供了一个操作用于将这个变量设为一个新的 TCPState。TCPConnection的构造器将该状态对象初始化为 TCPClosed状态(在后面定义)。

```

TCPConnection::TCPConnection () {
    _state = TCPClosed::Instance();
}

void TCPConnection::ChangeState (TCPState* s) {
    _state = s;
}

void TCPConnection::ActiveOpen () {
    _state->ActiveOpen(this);
}

void TCPConnection::PassiveOpen () {
    _state->PassiveOpen(this);
}

void TCPConnection::Close () {
    _state->Close(this);
}

```

```
void TCPConnection::Acknowledge () {
    _state->Acknowledge(this);
}
```

```
void TCPConnection::Synchronize () {
    _state->Synchronize(this);
}
```

TCPState为所有委托给它的请求实现缺省的行为。它也可以调用 ChangeState操作来改变 TCPConnection的状态。TCPState被定义为 TCPConnection的友元，从而给了它访问这一操作的特权。

```
void TCPState::Transmit (TCPConnection*, TCPOctetStream*) { }
void TCPState::ActiveOpen (TCPConnection*) { }
void TCPState::PassiveOpen (TCPConnection*) { }
void TCPState::Close (TCPConnection*) { }
void TCPState::Synchronize (TCPConnection*) { }
```

```
void TCPState::ChangeState (TCPConnection* t, TCPState* s) {
    t->ChangeState(s);
}
```

TCPState的子类实现与状态有关的行为。一个 TCP连接可处于多种状态：已建立、监听、已关闭等等，对每一个状态都有一个 TCPState的子类。我们将详细讨论三个子类：TCPEstablished、TCPListen和TCPClosed。

```
class TCPEstablished : public TCPState {
public:
    static TCPState* Instance();

    virtual void Transmit(TCPConnection*, TCPOctetStream*);
    virtual void Close(TCPConnection*);
};
```

```
class TCPListen : public TCPState {
public:
    static TCPState* Instance();

    virtual void Send(TCPConnection*);
    // ...
};
```

```
class TCPClosed : public TCPState {
public:
    static TCPState* Instance();

    virtual void ActiveOpen(TCPConnection*);
    virtual void PassiveOpen(TCPConnection*);
    // ...
};
```

TCPState的子类没有局部状态，因此它们可以被共享，并且每个子类只需一个实例。每个 TCPState子类的唯一实例由静态的 Instance操作<sup>⊖</sup>得到。

每一个TCPState子类为该状态下的合法请求实现与特定状态相关的行为：

```
void TCPClosed::ActiveOpen (TCPConnection* t) {
    // send SYN, receive SYN, ACK, etc.

    ChangeState(t, TCPEstablished::Instance());
}
```

⊖ 这使得每一个TCPState子类成为一个 Singleton (参见 Singleton)。

```

}

void TCPClosed::PassiveOpen (TCPConnection* t) {
    ChangeState(t, TCPListen::Instance());
}

void TCPEstablished::Close (TCPConnection* t) {
    // send FIN, receive ACK of FIN

    ChangeState(t, TCPListen::Instance());
}

void TCPEstablished::Transmit (
    TCPConnection* t, TCPOctetStream* o
) {
    t->ProcessOctet(o);
}

void TCPListen::Send (TCPConnection* t) {
    // send SYN, receive SYN, ACK, etc.

    ChangeState(t, TCPEstablished::Instance());
}

```

在完成与状态相关的工作后，这些操作调用 `ChangeState` 操作来改变 `TCPConnection` 的状态。`TCPConnection` 本身对 TCP 连接协议一无所知；是由 `TCPState` 子类来定义 TCP 中的每一个状态转换和动作。

### 11. 已知应用

Johnson 和 Zweig [JZ91] 描述了 State 模式以及它在 TCP 连接协议上的应用。

大多数流行的交互式绘图程序提供了以直接操纵的方式进行工作的“工具”。例如，一个画直线的工具可以让用户通过点击和拖动来创建一条新的直线；一个选择工具可以让用户选择某个图形对象。通常有许多这样的工具放在一个选项板供用户选择。用户认为这一活动是选择一个工具并使用它，但实际上编辑器的行为随当前的工具而变：当一个绘制工具被激活时，我们创建图形对象；当选择工具被激活时，我们选择图形对象；等等。我们可以使用 State 模式来根据当前的工具改变编辑器的行为。

我们可定义一个抽象的 `Tool` 类，再从这个类派生出一些子类，实现与特定工具相关的行为。图形编辑器维护一个当前 `Tool` 对象并将请求委托给它。当用户选择一个新的工具时，就将这个工具对象换成新的，从而使得图形编辑器的行为相应地发生改变。

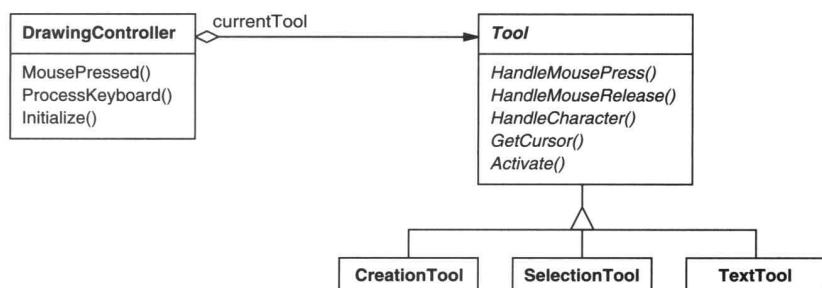
`HotDraw` [Joh92] 和 `Unidraw` [VL90] 中的绘图编辑器框架都使用了这一技术。它使得客户可以很容易地定义新类型的工具。在 `HotDraw` 中，`DrawingController` 类将请求转发给当前的 `Tool` 对象。在 `Unidraw` 中，相应的类是 `Viewer` 和 `Tool`。下页上图简要描述了 `Tool` 和 `DrawingController` 的接口。

Coplien 的 Envelope-Letter idiom [Cop92] 与 State 模式也有关。Envelope-Letter 是一种在运行时改变一个对象的类的技术。State 模式更为特殊，它着重于如何处理那些行为随状态变化而变化的对象。

### 12. 相关模式

Flyweight 模式 (4.6) 解释了何时以及怎样共享状态对象。

状态对象通常是 Singleton (3.5)。



## 5.9 STRATEGY(策略)——对象行为型模式

### 1. 意图

定义一系列的算法,把它们一个个封装起来,并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化。

### 2. 别名

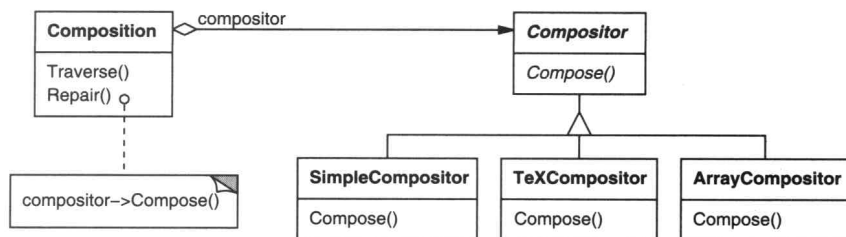
政策 (Policy)

### 3. 动机

有许多算法可对一个正文流进行分行。将这些算法硬编进使用它们的类中是不可取的,其原因如下:

- 需要换行功能的客户程序如果直接包含换行算法代码的话将会变得复杂,这使得客户程序庞大并且难以维护,尤其当其需要支持多种换行算法时问题会更加严重。
- 不同的时候需要不同的算法,我们不想支持我们并不使用的换行算法。
- 当换行功能是客户程序的一个难以分割的成分时,增加新的换行算法或改变现有算法将十分困难。

我们可以定义一些类来封装不同的换行算法,从而避免这些问题。一个以这种方法封装的算法称为一个策略(strategy),如下图所示。



假设一个 **Composition** 类负责维护和更新一个正文浏览程序中显示的正文换行。换行策略不是 **Composition** 类实现的,而是由抽象的 **Compositor** 类的子类各自独立地实现的。

**Compositor** 各个子类实现不同的换行策略:

- **SimpleCompositor** 实现一个简单的策略,它一次决定一个换行位置。
- **TeXCompositor** 实现查找换行位置的 TEX 算法。这个策略尽量全局地优化换行,也就是,一次处理一段文字的换行。
- **ArrayCompositor** 实现一个策略,该策略使得每一行都含有一个固定数目的项。例如,用

于对一系列的图标进行分行。

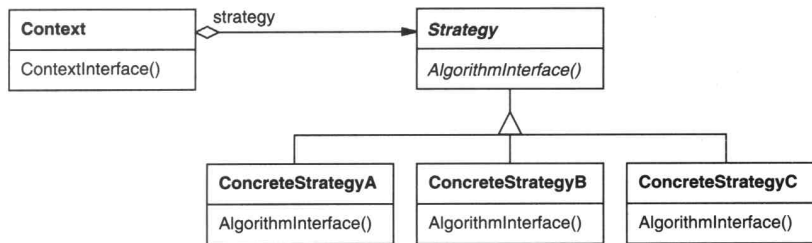
Composition维护对Compositor对象的一个引用。一旦Composition重新格式化它的正文，它就将这个职责转发给它的Compositor对象。Composition的客户指定应该使用哪一种Compositor的方式是直接将它想要的Compositor装入Composition中。

#### 4. 适用性

当存在以下情况时使用Strategy模式

- 许多相关的类仅仅是行为有异。“策略”提供了一种用多个行为中的一个行为来配置一个类的方法。
- 需要使用一个算法的不同变体。例如，你可能会定义一些反映不同的空间/时间权衡的算法。当这些变体实现为一个算法的类层次时 [HO87], 可以使用策略模式。
- 算法使用客户不应该知道的数据。可使用策略模式以避免暴露复杂的、与算法相关的数据结构。
- 一个类定义了多种行为，并且这些行为在这个类的操作中以多个条件语句的形式出现。将相关的条件分支移入它们各自的Strategy类中以代替这些条件语句。

#### 5. 结构



#### 6. 参与者

- Strategy(策略，如Compositor)
  - 定义所有支持的算法的公共接口。Context使用这个接口来调用某ConcreteStrategy定义的算法。
- ConcreteStrategy(具体策略，如SimpleCompositor, TeXCompositor, ArrayCompositor)
  - 以Strategy接口实现某具体算法。
- Context(上下文，如Composition)
  - 用一个ConcreteStrategy对象来配置。
  - 维护一个对Strategy对象的引用。
  - 可定义一个接口来让Strategy访问它的数据。

#### 7. 协作

- Strategy和Context相互作用以实现选定的算法。当算法被调用时，Context可以将该算法所需要的所有数据都传递给该Strategy。或者，Context可以将自身作为一个参数传递给Strategy操作。这就让Strategy在需要时可以回调Context。
- Context将它的客户的请求转发给它的Strategy。客户通常创建并传递一个ConcreteStrategy对象给该Context；这样，客户仅与Context交互。通常有一系列的ConcreteStrategy类可供客户从中选择。



## 8. 效果

Strategy模式有下面的一些优点和缺点：

1) 相关算法系列 Strategy类层次为Context定义了一系列的可供重用的算法或行为。继承有助于析取出这些算法中的公共功能。

2) 一个替代继承的方法 继承提供了另一种支持多种算法或行为的方法。你可以直接生成一个Context类的子类，从而给它以不同的行为。但这会将行为硬行编制到 Context中，而将算法的实现与 Context的实现混合起来，从而使Context难以理解、难以维护和难以扩展，而且还不能动态地改变算法。最后你得到一堆相关的类，它们之间的唯一差别是它们所使用的算法或行为。将算法封装在独立的Strategy类中使得你可以独立于其Context改变它，使它易于切换、易于理解、易于扩展。

3) 消除了一些条件语句 Strategy模式提供了用条件语句选择所需的行为以外的另一种选择。当不同的行为堆砌在一个类中时，很难避免使用条件语句来选择合适的行为。将行为封装在一个个独立的Strategy类中消除了这些条件语句。

例如，不用Strategy，正文换行的代码可能是象下面这样

```
void Composition::Repair () {
    switch (_breakingStrategy) {
        case SimpleStrategy:
            ComposeWithSimpleCompositor();
            break;
        case TeXStrategy:
            ComposeWithTeXCompositor();
            break;
        // ...
    }
    // merge results with existing composition, if necessary
}
```

Strategy模式将换行的任务委托给一个Strategy对象从而消除了这些case语句：

```
void Composition::Repair () {
    _compositor->Compose();
    // merge results with existing composition, if necessary
}
```

含有许多条件语句的代码通常意味着需要使用 Strategy模式。

4) 实现的选择 Strategy模式可以提供相同行为的不同实现。客户可以根据不同时间 / 空间权衡取舍要求从不同策略中进行选择。

5) 客户必须了解不同的Strategy 本模式有一个潜在的缺点，就是一个客户要选择一个合适的Strategy就必须知道这些Strategy到底有何不同。此时可能不得不向客户暴露具体的实现问题。因此仅当这些不同行为变体与客户相关的行为时，才需要使用Strategy模式。

6) Strategy和Context之间的通信开销 无论各个ConcreteStrategy实现的算法是简单还是复杂，它们都共享Strategy定义的接口。因此很可能某些 ConcreteStrategy不会都用到所有通过这个接口传递给它们的信息；简单的 ConcreteStrategy可能不使用其中的任何信息！这就意味着有时Context会创建和初始化一些永远不会用到的参数。如果存在这样问题，那么将需要在Strategy和Context之间更进行紧密的耦合。

7) 增加了对象的数目 Strategy增加了一个应用中的对象的数目。有时你可以将 Strategy实现为可供各Context共享的无状态的对象来减少这一开销。任何其余的状态都由 Context维护。



Context在每一次对Strategy对象的请求中都将这个状态传递过去。共享的Strategy不应在各次调用之间维护状态。Flyweight(4.6)模式更详细地描述了这一方法。

### 9. 实现

考虑下面的实现问题：

1) 定义Strategy和Context接口 Strategy和Context接口必须使得ConcreteStrategy能够有效的访问它所需要的Context中的任何数据，反之亦然。一种办法是让Context将数据放在参数中传递给Strategy操作——也就是说，将数据发送给Strategy。这使得Strategy和Context解耦。但另一方面，Context可能发送一些Strategy不需要的数据。

另一种办法是让Context将自身作为一个参数传递给Strategy，该Strategy再显式地向该Context请求数据。或者，Strategy可以存储对它的Context的一个引用，这样根本不再需要传递任何东西。这两种情况下，Strategy都可以请求到它所需要的数据。但现在Context必须对它的定义一个更为精细的接口，这将Strategy和Context更紧密地耦合在一起。

2) 将Strategy作为模板参数 在C++中，可利用模板机制用一个Strategy来配置一个类。然而这种技术仅当下面条件满足时才可以使用 (1) 可以在编译时选择Strategy (2) 它不需在运行时改变。在这种情况下，要被配置的类（如，Context）被定义为以一个Strategy类作为一个参数的模板类：

```
template <class AStrategy>
class Context {
    void Operation() { theStrategy.DoAlgorithm(); }
    // ...
private:
    AStrategy theStrategy;
};
```

当它被例化时该类用一个Strategy类来配置：

```
class MyStrategy {
public:
    void DoAlgorithm();
};

Context<MyStrategy> aContext;
```

使用模板不再需要定义给Strategy定义接口的抽象类。把Strategy作为一个模板参数也使得可以将一个Strategy和它的Context静态地绑定在一起，从而提高效率。

3) 使Strategy对象成为可选的 如果即使在不使用额外的Strategy对象的情况下，Context也还有意义的话，那么它还可以被简化。Context在访问某Strategy前先检查它是否存在，如果有，那么就使用它；如果没有，那么Context执行缺省的行为。这种方法的好处是客户根本不需要处理Strategy对象，除非它们不喜欢缺省的行为。

### 10. 代码示例

我们将给出动机一节例子的高层代码，这些代码基于InterViews[LCI+92]中的Composition和Compositor类的实现。

Composition类维护一个Component实例的集合，它们代表一个文档中的正文和图形元素。Composition使用一个封装了某种分行策略的Compositor子类实例将Component对象编排成行。每一个Component都有相应的正常大小、可伸展性和可收缩性。可伸展性定义了该Component可以增长到超出正常大小的程度；可收缩性定义了它可以收缩的程度。Composition将这些值

传递给一个 Compositor，它使用这些值来决定换行的最佳位置。

```
class Composition {
public:
    Composition(Compositor*);
    void Repair();
private:
    Compositor* _compositor;
    Component* _components;    // the list of components
    int _componentCount;       // the number of components
    int _lineWidth;            // the Composition's line width
    int* _lineBreaks;          // the position of linebreaks
                                // in components
    int _lineCount;            // the number of lines
};
```

当需要一个新的布局时，Composition 让它的 Compositor 决定在何处换行。Composition 传递给 Compositor 三个数组，它们定义各 Component 的正常大小、可伸展性和可收缩性。它还传递 Component 的数目、线的宽度以及一个数组，让 Compositor 来填充每次换行的位置。Compositor 返回计算得到的换行数目。

Compositor 接口使得 Composition 可传递给 Compositor 所有它需要的信息。此处是一个“将数据传给 Strategy”的例子：

```
class Compositor {
public:
    virtual int Compose(
        Coord natural[], Coord stretch[], Coord shrink[],
        int componentCount, int lineWidth, int breaks[]
    ) = 0;
protected:
    Compositor();
};
```

注意 Compositor 是一个抽象类，而其具体子类定义特定的换行策略。

Composition 在它的 Repair 操作中调用它的 Compositor。Repair 首先用每一个 Component 的正常大小、可伸展性和可收缩性初始化数组（为简单起见略去细节）。然后它调用 Compositor 得到换行位置并最终据以对 Component 进行布局（也省略了）：

```
void Composition::Repair () {
    Coord* natural;
    Coord* stretchability;
    Coord* shrinkability;
    int componentCount;
    int* breaks;

    // prepare the arrays with the desired component sizes
    // ...

    // determine where the breaks are:
    int breakCount;
    breakCount = _compositor->Compose(
        natural, stretchability, shrinkability,
        componentCount, _lineWidth, breaks
    );

    // lay out components according to breaks
    // ...
}
```

现在我们来查看各 Compositor 子类。SimpleCompositor 一次检查一行 Component，并决定在

那儿换行：

```
class SimpleCompositor : public Compositor {
public:
    SimpleCompositor();

    virtual int Compose(
        Coord natural[], Coord stretch[], Coord shrink[],
        int componentCount, int lineWidth, int breaks[]
    );
    // ...
};
```

TeXCompositor使用一个更为全局的策略。它每次检查一个段落 ( paragraph ), 并同时考虑到各Component的大小和伸展性。它也通过压缩 Component之间的空白以尽量给该段落一个均匀的“色彩”。

```
class TeXCompositor : public Compositor {
public:
    TeXCompositor();

    virtual int Compose(
        Coord natural[], Coord stretch[], Coord shrink[],
        int componentCount, int lineWidth, int breaks[]
    );
    // ...
};
```

ArrayCompositor用规则的间距将构件分割成行。

```
class ArrayCompositor : public Compositor {
public:
    ArrayCompositor(int interval);

    virtual int Compose(
        Coord natural[], Coord stretch[], Coord shrink[],
        int componentCount, int lineWidth, int breaks[]
    );
    // ...
};
```

这些类并未都使用所有传递给 Compose的信息。SimpleCompositor忽略Component的伸展性，仅考虑它们的正常大小；TeXCompositor使用所有传递给它的信息；而ArrayCompositor忽略所有的信息。

实例化Composition时需把想要使用的Compositor传递给它：

```
Composition* quick = new Composition(new SimpleCompositor);
Composition* slick = new Composition(new TeXCompositor);
Composition* iconic = new Composition(new ArrayCompositor(100));
```

Compositor的接口须经仔细设计，以支持子类可能实现的所有排版算法。你不希望在生成一个新的子类不得不修改这个接口，因为这需要修改其它已有的子类。一般来说，Strategy和Context的接口决定了该模式能在多大程度上达到既定目的。

#### 11. 已知应用

ET++[WGM88]和InterViews都使用Strategy来封装不同的换行算法。

在用于编译器代码优化的RTL系统[JML92]中，Strategy定义了不同的寄存器分配方案 ( RegisterAllocator ) 和指令集调度策略 ( RISCscheduler , CISCscheduler )。这就为在不同的目标机器结构上实现优化程序提供了所需的灵活性。

ET++SwapsManager计算引擎框架为不同的金融设备 [EG92]计算价格。它的关键抽象是 Instrument (设备) 和 YieldCurve (受益率曲线)。不同的设备实现为不同的 Instrument子类。YieldCurve计算贴现因子 (discount factors) 表示将来的现金流的值。这两个类都将一些行为委托给 Strategy对象。该框架提供了一系列的 ConcreteStrategy类用于生成现金流, 记值交换, 以及计算贴现因子。可以用不同的 ConcreteStrategy对象配置 Instrument和YieldCurve以创建新的计算引擎。这种方法支持混合和匹配现有的 Strategy实现, 也支持定义新的 Strategy实现。

Booch构件[BV90]将Strategy用作模板参数。Booch集合类支持三种不同的存储分配策略: 管理的 (从一个存储池中分配), 控制的 (分配/去配有锁保护), 以及无管理的 (正常的存储分配器)。在一个集合类实例化时, 将这些 Strategy作为模板参数传递给它。例如, 一个使用无管理策略的 UnboundedCollection实例化为 UnboundedCollection MyItemType\*, Unmanaged。

RApp是一个集成电路布局系统 [GA89, AG90]。RApp必须对连接电路中各子系统的线路进行布局和布线。RApp中的布线算法定义为一个抽象 Router类的子类。Router是一个 Strategy类。

Borland的ObjectWindows[Bor94]在对话框中使用Strategy来保证用户输入合法的数据。例如, 数字必须在一定范围, 并且一个数值输入域应只接受数字。验证一个字符串是正确的可能需要对某个表进行一次查找。

ObjectWindows使用Validator对象来封装验证策略。Validator是Strategy对象的例子。数据输入域将验证策略委托给一个可选的 Validator对象。如果需要验证时, 客户给域加上一个验证器 (一个可选策略的例子)。当该对话框关闭时, 输入域让它们的验证器验证数据。该类库为常用情况提供了一些验证器, 例如数字的 RangeValidator。可以通过继承 Validator类很容易的定义新的与客户相关的验证策略。

## 12. 相关模式

Flyweight (4.6): Strategy对象经常是很好的轻量级对象。

## 5.10 TEMPLATE METHOD(模板方法)——类行为型模式

### 1. 意图

定义一个操作中的算法的骨架, 而将一些步骤延迟到子类中。TemplateMethod使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

### 2. 动机

考虑一个提供 Application和Document类的应用框架。Application类负责打开一个已有的以外部形式存储的文档, 如一个文件。一旦一个文档中的信息从该文件中读出后, 它就由一个Document对象表示。

用框架构建的应用可以通过继承 Application和Document来满足特定的需求。例如, 一个绘图应用定义 DrawApplication和DrawDocument子类; 一个电子表格应用定义 SpreadsheetApplication和SpreadsheetDocument子类, 如下页图所示。

抽象的Application类在它的OpenDocument操作中定义了打开和读取一个文档的算法:

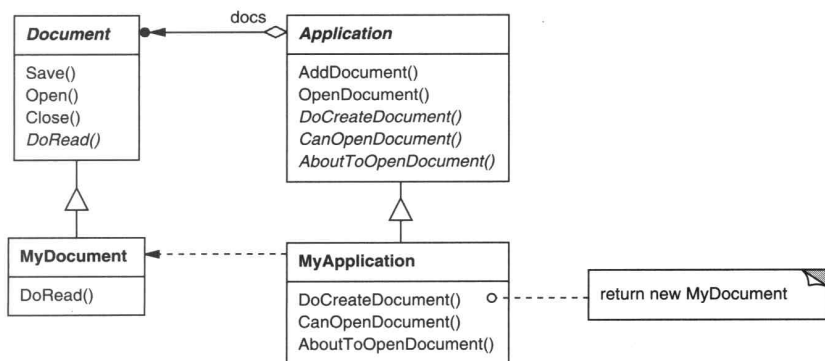
```
void Application::OpenDocument (const char* name) {  
    if (!CanOpenDocument(name)) {  
        // cannot handle this document  
        return;  
    }  
}
```

```

Document* doc = DoCreateDocument();

if (doc) {
    _docs->AddDocument(doc);
    AboutToOpenDocument(doc);
    doc->Open();
    doc->DoRead();
}
}

```



OpenDocument定义了一个打开文档的每一个主要步骤。它检查该文档是否能被打开，创建与应用相关的 Document对象，将它加到它的文档集合中，并且从一个文件中读取该 Document。

我们称OpenDocument为一个模板方法(template method)。一个模板方法用一些抽象的操作定义一个算法，而子类将重定义这些操作以提供具体的行为。Application的子类将定义检查一个文档是否能够被打开(CanOpenDocument)和创建文档(DoCreateDocument)的具体算法步骤。Document子类将定义读取文档(DoRead)的算法步骤。如果需要，模板方法也可定义一个操作(AboutToOpenDocument)让Application子类知道该文档何时将被打开。

通过使用抽象操作定义一个算法中的一些步骤，模板方法确定了它们的先后顺序，但它允许Application和Document子类改变这些具体步骤以满足它们各自的需求。

### 3. 适用性

模板方法应用于下列情况：

- 一次性实现一个算法的不变的部分，并将可变的行留留给子类来实现。
- 各子类中公共的行为应被提取出来并集中到一个公共父类中以避免代码重复。这是Opdyke和Johnson所描述过的“重分解以一般化”的一个很好的例子 [OJ93]。首先识别现有代码中的不同之处，并且将不同之处分离为新的操作。最后，用一个调用这些新的操作的模板方法来替换这些不同的代码。
- 控制子类扩展。模板方法只在特定点调用“hook”操作(参见效果一节)，这样就只允许在这些点进行扩展。

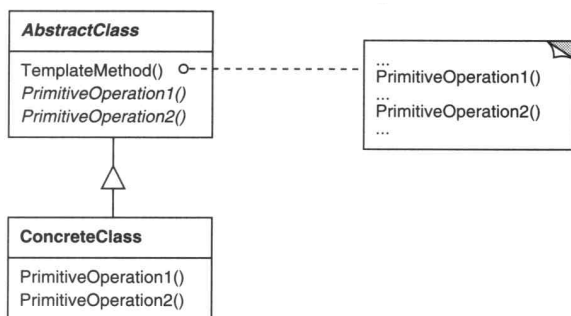
### 4. 结构(见下页图)

### 5. 参与者

- AbstractClass(抽象类，如Application)

— 定义抽象的原语操作(primitive operation)，具体的子类将重定义它们以实现一个算法

的各步骤。



— 实现一个模板方法,定义一个算法的骨架。该模板方法不仅调用原语操作,也调用定义在AbstractClass或其他对象中的操作。

• ConcreteClass (具体类,如MyApplication)

— 实现原语操作以完成算法中与特定子类相关的步骤。

## 6. 协作

• ConcreteClass靠AbstractClass来实现算法中不变的步骤。

## 7. 效果

模板方法是一种代码复用的基本技术。它们在类库中尤为重要,它们提取了类库中的公共行为。

模板方法导致一种反向的控制结构,这种结构有时被称为“好莱坞法则”,即“别找我们,我们找你”[Swe85]。这指的是一个父类调用一个子类的操作,而不是相反。

模板方法调用下列类型的操作:

- 具体的操作 (ConcreteClass或对客户类的操作)。
- 具体的AbstractClass的操作 (即,通常对子类有用的操作)。
- 原语操作 (即,抽象操作)。
- Factory Method (参见Factory Method (3.5))。
- 钩子操作 (hook operations),它提供了缺省的行为,子类可以在必要时进行扩展。一个钩子操作在缺省操作通常是一个空操作。

很重要的一点是模板方法应该指明哪些操作是钩子操作 (可以被重定义) 以及哪些是抽象操作 (必须被重定义)。要有效地重用抽象类,子类编写者必须明确了解哪些操作是设计为有待重定义的。

子类可以通过重定义父类的操作来扩展该操作的行为,其间可显式地调用父类操作。

```

void DerivedClass::Operation () {
    ParentClass::Operation();
    // DerivedClass extended behavior
}
  
```

不幸的是,人们很容易忘记去调用被继承的行为。我们可以将这样一个操作转换为一个模板方法,以使得父类可以对子类的扩展方式进行控制。也就是,在父类的模板方法中调用钩子操作。子类可以重定义这个钩子操作:

```

void ParentClass::Operation () {
    // ParentClass behavior
}
  
```



```
HookOperation();
}
```

ParentClass本身的HookOperation什么也不做：

```
void ParentClass::HookOperation () { }
```

子类重定义HookOperation以扩展它的行为：

```
void DerivedClass::HookOperation () {
    // derived class extension
}
```

## 8. 实现

有三个实现问题值得注意：

1) 使用C++访问控制 在C++中，一个模板方法调用的原语操作可以被定义为保护成员。这保证它们只被模板方法调用。必须重定义的原语操作须定义为纯虚函数。模板方法自身不需被重定义；因此可以将模板方法定义为一个非虚成员函数。

2) 尽量减少原语操作 定义模板方法的一个重要目的是尽量减少一个子类具体实现该算法时必须重定义的那些原语操作的数目。需要重定义的操作越多，客户程序就越冗长。

3) 命名约定 可以给应被重定义的那些操作的名字加上一个前缀以识别它们。例如，用于Macintosh应用的MacApp框架[App89]给模板方法加上前缀“Do-”，如“DoCreateDocument”，“DoRead”，等等。

## 9. 代码示例

下面的C++实例说明了一个父类如何强制其子类遵循一种不变的结构。这个例子来自于NeXT的AppKit[Add94]。考虑一个支持在屏幕上绘图的类View。一个视图在进入“焦点”(focus)状态时才设定合适的特定绘图状态(如颜色和字体)，因而只有成为“焦点”之后才能进行绘图。View类强制其子类遵循这个规则。

我们用Display模板方法来解决这个问题。View定义两个具体操作，SetFocus和ResetFocus，分别设定和清除绘图状态。View的DoDisplay钩子操作实施真正的绘图功能。Display在DoDisplay前调用SetFocus以设定绘图状态；Display此后调用ResetFocus以释放绘图状态。

```
void View::Display () {
    SetFocus();
    DoDisplay();
    ResetFocus();
}
```

为维持不变部分，View的客户通常调用Display，而View的子类通常重定义DoDisplay。

View本身的DoDisplay什么也不做：

```
void View::DoDisplay () { }
```

子类重定义它以增加它们的特定绘图行为：

```
void MyView::DoDisplay () {
    // render the view's contents
}
```

## 10. 已知应用

模板方法非常基本，它们几乎可以在任何一个抽象类中找到。Wirfs-Brock等人[WBW90,WBJ90]曾很好地概述和讨论了模板方法。



## 11. 相关模式

Factory Method模式（3.3）常被模板方法调用。在动机一节例子中，DoCreateDocument就是一个Factory Method，它由模板方法OpenDocument调用。

Strategy（5.9）：模板方法使用继承来改变算法的一部分。Strategy使用委托来改变整个算法。

## 5.11 VISITOR（访问者）——对象行为型模式

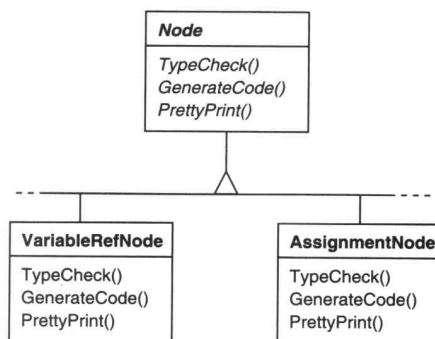
### 1. 意图

表示一个作用于某对象结构中的各元素的操作。它使你可以在不改变各元素的类的前提下定义作用于这些元素的新操作。

### 2. 动机

考虑一个编译器，它将源程序表示为一个抽象语法树。该编译器需在抽象语法树上实施某些操作以进行“静态语义”分析，例如检查是否所有的变量都已经被定义了。它也需要生成代码。因此它可能要定义许多操作以进行类型检查、代码优化、流程分析，检查变量是否在使用前被赋初值，等等。此外，还可使用抽象语法树进行优美格式打印、程序重构、code instrumentation以及对程序进行多种度量。

这些操作大多要求对不同的节点进行不同的处理。例如对代表赋值语句的结点的处理就不同于对代表变量或算术表达式的结点的处理。因此有用于赋值语句的类，有用于变量访问的类，还有用于算术表达式的类，等等。结点类的集合当然依赖于被编译的语言，但对于一个给定的语言其变化不大。



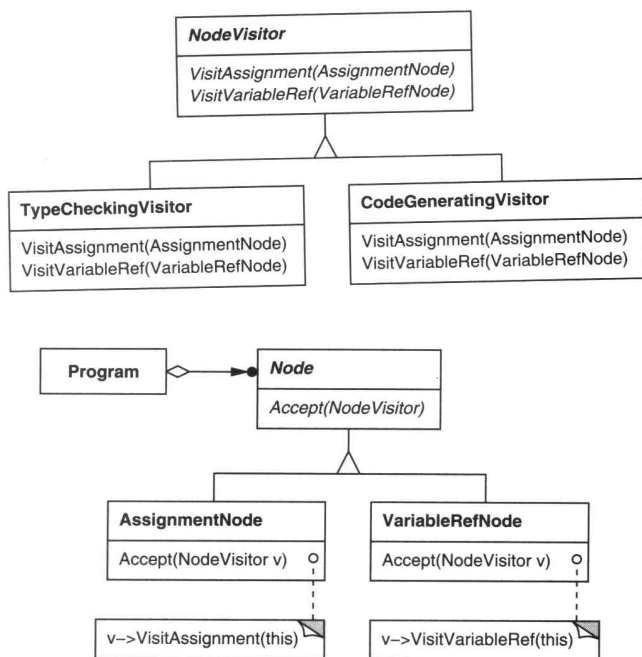
上面的框图显示了Node类层次的一部分。这里的问题是，将所有这些操作分散到各种结点类中会导致整个系统难以理解、难以维护和修改。将类型检查代码与优美格式打印代码或流程分析代码放在一起，将产生混乱。此外，增加新的操作通常需要重新编译所有这些类。如果可以独立地增加新的操作，并且使这些结点类独立于作用于其上的操作，将会更好一些。

要实现上述两个目标，我们可以将每一个类中相关的操作包装在一个独立的对象（称为一个Visitor）中，并在遍历抽象语法树时将此对象传递给当前访问的元素。当一个元素“接受”该访问者时，该元素向访问者发送一个包含自身类信息的请求。该请求同时也将该元素本身作为一个参数。然后访问者将为该元素执行该操作——这一操作以前是在该元素的类中的。

例如，一个不使用访问者的编译器可能会通过在它的抽象语法树上调用TypeCheck操作对

一个过程进行类型检查。每一个结点将对调用它的成员的 TypeCheck 以实现自身的 TypeCheck (参见前面的类框图)。如果该编译器使用访问者对一个过程进行类型检查,那么它将会创建一个 TypeCheckingVisitor 对象,并以这个对象为一个参数在抽象语法树上调用 Accept 操作。每一个结点在实现 Accept 时将会回调访问者:一个赋值结点调用访问者的 VisitAssignment 操作,而一个变量引用将调用 VisitVariableReference。以前类 AssignmentNode 的 TypeCheck 操作现在成为 TypeCheckingVisitor 的 VisitAssignment 操作。

为使访问者不仅仅只做类型检查,我们需要所有抽象语法树的访问者有一个抽象的父类 NodeVisitor。NodeVisitor 必须为每一个结点类定义一个操作。一个需要计算程序度量的应用将定义 NodeVisitor 的新的子类,并且将不再需要在结点类中增加与特定应用相关的代码。Visitor 模式将每一个编译步骤的操作封装在一个与该步骤相关的 Visitor 中(参见下图)。



使用 Visitor 模式,必须定义两个类层次:一个对应于接受操作的元素(Node 层次)另一个对应于定义对元素的操作的访问者(NodeVisitor 层次)。给访问者类层次增加一个新的子类即可创建一个新的操作。只要该编译器接受的语法不改变(即不需要增加新的 Node 子类),我们就可以简单的定义新的 NodeVisitor 子类以增加新的功能。

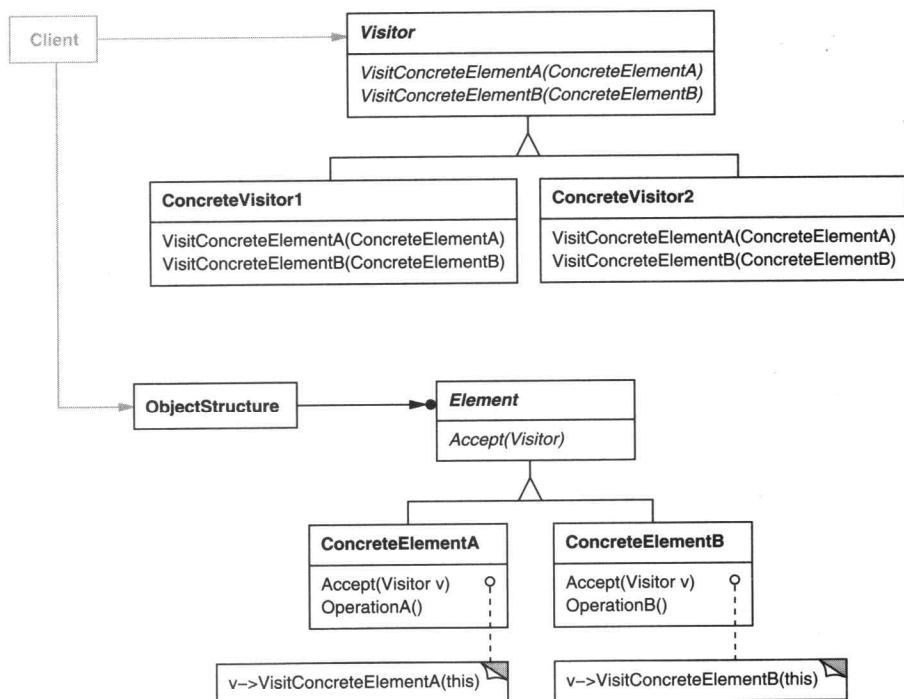
### 3. 适用性

在下列情况下使用 Visitor 模式:

- 一个对象结构包含很多类对象,它们有不同的接口,而你想对这些对象实施一些依赖于其具体类的操作。
- 需要对一个对象结构中的对象进行很多不同的并且不相关的操作,而你想避免让这些操作“污染”这些对象的类。Visitor 使得你可以将相关的操作集中起来定义在一个类中。当该对象结构被很多应用共享时,用 Visitor 模式让每个应用仅包含需要用到操作。
- 定义对象结构的类很少改变,但经常需要在此结构上定义新的操作。改变对象结构类需

要重定义对所有访问者的接口，这可能需要很大的代价。如果对象结构类经常改变，那么可能还是在这些类中定义这些操作较好。

#### 4. 结构



#### 5. 参与者

##### • Visitor（访问者，如NodeVisitor）

— 为该对象结构中 ConcreteElement 的每一个类声明一个 Visit 操作。该操作的名字和特征标识了发送 Visit 请求给该访问者的那个类。这使得访问者可以确定正被访问元素的具体类。这样访问者就可以通过该元素的特定接口直接访问它。

##### • ConcreteVisitor（具体访问者，如TypeCheckingVisitor）

— 实现每个由 Visitor 声明的操作。每个操作实现本算法的一部分，而该算法片断乃是对应于结构中对象的类。ConcreteVisitor 为该算法提供了上下文并存储它的局部状态。这一状态常常在遍历该结构的过程中累积结果。

##### • Element（元素，如Node）

— 定义一个 Accept 操作，它以一个访问者为参数。

##### • ConcreteElement（具体元素，如AssignmentNode，VariableRefNode）

— 实现 Accept 操作，该操作以一个访问者为参数。

##### • ObjectStructure（对象结构，如Program）

— 能枚举它的元素。

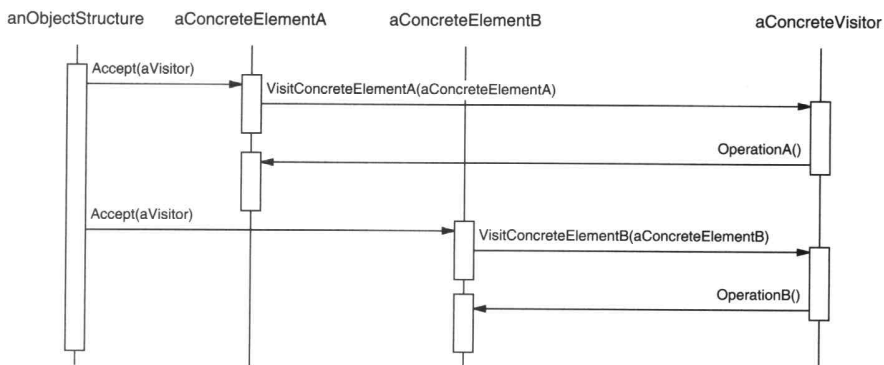
— 可以提供一个高层的接口以允许该访问者访问它的元素。

— 可以是一个复合（参见 Composite（4.3））或是一个集合，如一个列表或一个无序集合。

## 6. 协作

- 一个使用 Visitor模式的客户必须创建一个 ConcreteVisitor对象，然后遍历该对象结构，并用该访问者访问每一个元素。
- 当一个元素被访问时，它调用对应于它的类的 Visitor操作。如果必要，该元素将自身作为这个操作的一个参数以便该访问者访问它的状态。

下面的交互框图说明了一个对象结构、一个访问者和两个元素之间的协作。



## 7. 效果

下面是访问者模式的一些优缺点：

1) 访问者模式使得易于增加新的操作 访问者使得增加依赖于复杂对象结构的构件的操作变得容易了。仅需增加一个新的访问者即可在一个对象结构上定义一个新的操作。相反，如果每个功能都分散在多个类之上的话，定义新的操作时必须修改每一类。

2) 访问者集中相关的操作而分离无关的操作 相关的行为不是分布在定义该对象结构的各个类上，而是集中在一个访问者中。无关行为却被分别放在它们各自的访问者子类中。这就既简化了这些元素的类，也简化了在这些访问者中定义的算法。所有与它的算法相关的数据结构都可以被隐藏在访问者中。

3) 增加新的 ConcreteElement类很困难 Visitor模式使得难以增加新的 Element的子类。每添加一个新的 ConcreteElement都要在 Visitor中添加一个新的抽象操作，并在每一个 ConcreteVisitor类中实现相应的操作。有时可以在 Visitor中提供一个缺省的实现，这一实现可以被大多数的 ConcreteVisitor继承，但这与其说是一个规律还不如说是一种例外。

所以在应用访问者模式时考虑关键的问题是系统的哪个部分会经常变化，是作用于对象结构上的算法呢还是构成该结构的各个对象的类。如果老是有新的 ConcreteElement类加入进来的话，Visitor类层次将变得难以维护。在这种情况下，直接在构成该结构的类中定义这些操作可能更容易一些。如果 Element类层次是稳定的，而你不断地增加操作或修改算法，访问者模式可以帮助你管理这些改动。

4) 通过类层次进行访问 一个迭代器（参见 Iterator（5.4））可以通过调用节点对象的特定操作来遍历整个对象结构，同时访问这些对象。但是迭代器不能对具有不同元素类型的对象结构进行操作。例如，定义在第5章的 Iterator接口只能访问类型为 Item的对象：

```
template <class Item>
class Iterator {
    // ...
```

```
Item CurrentItem() const;
};
```

这就意味着所有该迭代器能够访问的元素都有一个共同的父类 Item。

访问者没有这种限制。它可以访问不具有相同父类的对象。可以对一个 Visitor接口增加任何类型的对象。例如，在

```
class Visitor {
public:
    // ...
    void VisitMyType(MyType*);
    void VisitYourType(YourType*);
};
```

中，MyType和YourType可以完全无关，它们不必继承相同的父类。

5) 累积状态 当访问者访问对象结构中的每一个元素时，它可能会累积状态。如果没有访问者，这一状态将作为额外的参数传递给进行遍历的操作，或者定义为全局变量。

6) 破坏封装 访问者方法假定 ConcreteElement接口的功能足够强，足以让访问者进行它们的工作。结果是，该模式常常迫使你提供访问元素内部状态的公共操作，这可能会破坏它的封装性。

## 8. 实现

每一个对象结构将有一个相关的 Visitor类。这个抽象的访问者类为定义对象结构的每一个 ConcreteElement类声明一个 VisitConcreteElement操作。每一个 Visitor上的 Visit操作声明它的参数为一个特定的 ConcreteElement，以允许该 Visitor直接访问 ConcreteElement的接口。ConcreteVistor类重定义每一个 Visit操作，从而为相应的 ConcreteElement类实现与特定访问者相关的行为。

在C++中，Visitor类可以这样定义：

```
class Visitor {
public:
    virtual void VisitElementA(ElementA*);
    virtual void VisitElementB(ElementB*);

    // and so on for other concrete elements
protected:
    Visitor();
};
```

每个 ConcreteElement类实现一个 Accept操作，这个操作调用访问者中相应于本 ConcreteElement类的 Visit...的操作。这样最终得到调用的操作不仅依赖于该元素的类也依赖于访问者的类<sup>⊖</sup>。

具体元素声明为：

```
class Element {
public:
    virtual ~Element();
    virtual void Accept(Visitor&) = 0;
protected:
```

⊖ 因为这些操作所传递的参数各不相同，我们可以使用函数重载机制来给这些操作以相同的简单命名，例如 Visit。这样的重载有好处也有坏处。一方面，它强调了这样一个事实：每个操作涉及的是相同的分析，尽管它们使用不同的参数。另一方面，对阅读代码的人来说，可能在调用点正在进行些什么就不那么显而易见了。其实这最终取决于你认为函数重载机制究竟是好还是坏。

```

    Element();
};

class ElementA : public Element {
public:
    ElementA();
    virtual void Accept(Visitor& v) { v.VisitElementA(this); }
};

class ElementB : public Element {
public:
    ElementB();
    virtual void Accept(Visitor& v) { v.VisitElementB(this); }
};

```

一个CompositeElement类可能象这样实现 Accept :

```

class CompositeElement : public Element {
public:
    virtual void Accept(Visitor&);
private:
    List<Element*>* _children;
};

void CompositeElement::Accept (Visitor& v) {
    ListIterator<Element*> i(_children);

    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Accept(v);
    }
    v.VisitCompositeElement(this);
}

```

下面是当应用 Visitor模式时产生的其他两个实现问题：

1) 双分派 (Double-dispatch) 访问者模式允许你不改变类即可有效地增加其上的操作。为达到这一效果使用了一种称为双分派 (double-dispatch) 的技术。这是一种很著名的技术。事实上，一些编程语言甚至直接支持这一技术 (例如，CLOS)。而象C++和Smalltalk这样的语言支持单分派 (single-dispatch)。

在单分派语言中，到底由哪一种操作将来实现一个请求取决于两个方面：该请求的名和接收者的类型。例如，一个 GenerateCode请求将会调用的操作决定于你请求的结点对象的类型。在C++中，对一个 VariableRefNode实例调用 GenerateCode将调用 VariableRefNode::GenerateCode (它生成一个变量引用的代码)。而对一个 AssignmentNode调用 GenerateCode将调用 AssignmentNode::GenerateCode (它生成一个赋值操作的代码)。所以最终哪个操作得到执行依赖于请求和接收者的类型两个方面。

双分派意味着得到执行的操作决定于请求的种类和两个接收者的类型。Accept是一个 double-dispatch操作。它的含义决定于两个类型：Visitor的类型和Element的类型。双分派使得访问者可以对每一个类元的素请求不同的操作。<sup>⊖</sup>

这是Visitor模式的关键所在：得到执行的操作不仅决定于 Visitor的类型还决定于它访问的Element的类型。可以不将操作静态地绑定在 Element接口中，而将其安放在一个 Visitor中，并

⊖ 如果我们可以有双分派，那么为什么不可以是三分派或四分派，甚至是任意其他数目的分派呢？实际上，双分派仅仅是多分派 (multiple-dispatch) 的一个特例，在多分派中操作的选择基于任意数目的类型。(事实上CLOS支持多分派。) 在支持双分派或多分派的语言中，Visitor模式的就不那么必需了。



使用Accept在运行时进行绑定。扩展Element接口就等于定义一个新的Visitor子类而不是多个新的Element子类。

2) 谁负责遍历对象结构 一个访问者必须访问这个对象结构的每一个元素。问题是，它怎样做？我们可以将遍历的责任放到下面三个地方中的任意一个：对象结构中，访问者中，或一个独立的迭代器对象中（参见Iterator（5.4））。

通常由对象结构负责迭代。一个集合只需对它的元素进行迭代，并对每一个元素调用Accept操作。而一个复合通常让Accept操作遍历该元素的各子构件并对它们中的每一个递归地调用Accept。

另一个解决方案是使用一个迭代器来访问各个元素。在C++中，既可以使用内部迭代器也可以使用外部迭代器，到底用哪一个取决于哪一个可用和哪一个最有效。在Smalltalk中，通常使用一个内部迭代器，这个内部迭代器使用do: 和一个块。因为内部迭代器由对象结构实现，使用一个内部迭代器很大程度上就像是让对象结构负责迭代。主要区别在于一个内部迭代器不会产生双分派——它将以该元素为一个参数调用访问者的一个操作而不是以访问者为参数调用元素的一个操作。不过，如果访问者的操作仅简单地调用该元素的操作而无需递归的话，使用一个内部迭代器的Visitor模式很容易使用。

甚至可以将遍历算法放在访问者中，尽管这样将导致对每一个聚合ConcreteElement，在每一个ConcreteVisitor中都要复制遍历的代码。将该遍历策略放在访问者中的主要原因是想实现一个特别复杂的遍历，它依赖于对该对象结构的操作结果。我们将在代码示例一节给出这种情况的一个例子。

## 9. 代码示例

因为访问者通常与复合相关，我们将使用在Composite（4.3）代码示例一节中定义的Equipment类来说明Visitor模式。我们将使用Visitor定义一些用于计算材料存货清单和单件设备总花费的操作。Equipment类非常简单，实际上并不一定要使用Visitor。但我们可以从中很容易地看出实现该模式时会涉及的内容。

这里是Composite（4.3）中的Equipment类。我们给它添加一个Accept操作，使其可与一个访问者一起工作。

```
class Equipment {
public:
    virtual ~Equipment();

    const char* Name() { return _name; }

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();

    virtual void Accept(EquipmentVisitor&);
protected:
    Equipment(const char*);
private:
    const char* _name;
};
```

各Equipment操作返回设备的属性，例如它的功耗和价格。对于特定种类的设备（如，底盘、发动机和平面板）子类适当地重定义这些操作。



如下所示，所有设备访问者的抽象父类对每一个设备子类都有一个虚函数。所有的虚函数的缺省行为都是什么也不做。

```
class EquipmentVisitor {
public:
    virtual ~EquipmentVisitor();

    virtual void VisitFloppyDisk(FloppyDisk*);
    virtual void VisitCard(Card*);
    virtual void VisitChassis(Chassis*);
    virtual void VisitBus(Bus*);

    // and so on for other concrete subclasses of Equipment
protected:
    EquipmentVisitor();
};
```

Equipment子类以基本相同的方式定义 Accept：调用EquipmentVisitor中的对应于接受Accept请求的类的操作，如：

```
void FloppyDisk::Accept (EquipmentVisitor& visitor) {
    visitor.VisitFloppyDisk(this);
}
```

包含其他设备的设备（尤其是在 Composite模式中CompositeEquipment的子类）实现Accept时，遍历其各个子构件并调用它们各自的 Accept操作，然后对自己调用 Visit操作。例如，Chassis::Accept可象如下这样遍历底盘中的所有部件：

```
void Chassis::Accept (EquipmentVisitor& visitor) {
    for (
        ListIterator<Equipment*> i(_parts);
        !i.IsDone();
        i.Next()
    ) {
        i.CurrentItem()->Accept(visitor);
    }
    visitor.VisitChassis(this);
}
```

EquipmentVisitor的子类在设备结构上定义了特定的算法。PricingVisitor计算该设备结构的价格。它计算所有的简单设备（如软盘）的实价以及所有复合设备（如底盘和公共汽车）打折后的价格。

```
class PricingVisitor : public EquipmentVisitor {
public:
    PricingVisitor();

    Currency& GetTotalPrice();

    virtual void VisitFloppyDisk(FloppyDisk*);
    virtual void VisitCard(Card*);
    virtual void VisitChassis(Chassis*);
    virtual void VisitBus(Bus*);
    // ...
private:
    Currency _total;
};

void PricingVisitor::VisitFloppyDisk (FloppyDisk* e) {
    _total += e->NetPrice();
}
```

```
void PricingVisitor::VisitChassis (Chassis* e) {
    _total += e->DiscountPrice();
}
```

PricingVisitor将计算设备结构中所有结点的总价格。注意 PricingVisitor在相应的成员函数中为一类设备选择合适的定价策略。此外，我们只需改变 PricingVisitor类即可改变一个设备结构的定价策略。

我们可以象这样定义一个计算存货清单的类：

```
class InventoryVisitor : public EquipmentVisitor {
public:
    InventoryVisitor();

    Inventory& GetInventory();

    virtual void VisitFloppyDisk(FloppyDisk*);
    virtual void VisitCard(Card*);
    virtual void VisitChassis(Chassis*);
    virtual void VisitBus(Bus*);
    // ...

private:
    Inventory _inventory;
};
```

InventoryVisitor为对象结构中的每一种类型的设备累计总和。InventoryVisitor使用一个Inventory类，Inventory类定义了一个接口用于增加设备（此处略去）。

```
void InventoryVisitor::VisitFloppyDisk (FloppyDisk* e) {
    _inventory.Accumulate(e);
}

void InventoryVisitor::VisitChassis (Chassis* e) {
    _inventory.Accumulate(e);
}
```

下面是如何在一个设备结构上使用InventoryVisitor：

```
Equipment* component;
InventoryVisitor visitor;

component->Accept(visitor);
cout << "Inventory "
    << component->Name()
    << visitor.GetInventory();
```

现在我们将说明如何用Visitor模式实现Interpreter模式中那个Smalltalk的例子（5.3）。像上面的例子一样，这个例子非常小，Visitor可能并不能带给我们很多好处，但是它很好地说明了如何使用这个模式。此外，它说明了一种情况，在此情况下迭代是访问者的职责。

该对象结构（正则表达式）由四个类组成，并且它们都有一个accept:方法，它以某访问者为一个参数。在类SequenceExpression中，accept:方法是：

```
accept: aVisitor
    ^ aVisitor visitSequence: self
```

在类RepeatExpression中，accept:方法发送visitRepeat消息；在类AlternationExpression中，它发送visitAlternation:消息；而在类LiteralExpression中，它发送visitLiteral:消息。

这四个类还必须有可供Vistor使用的访问函数。对于SequenceExpression这些函数是

expression1和expression2；对于AlternationExpression这些函数是alternative1和alternative2；对于RepeatExpression是repetition；而对于LiteralExpression则是component。

具体的访问者是REMatchingVisitor。因为它所需要的遍历算法是不规则的，因此由它自己负责进行遍历。其最大的不规则之处在于RepeatExpression要重复遍历它的构件。REMatchingVisitor类有一个实例变量inputState。它的各个方法除了将名字为inputState的参数替换为匹配的表达式结点以外，与Interpreter模式中表达式类的match:方法基本上是一样的。它们还是返回该表达式可以匹配的流的集合以标识当前状态。

```
visitSequence: sequenceExp
    inputState := sequenceExp expression1 accept: self.
    ^ sequenceExp expression2 accept: self.

visitRepeat: repeatExp
    | finalState |
    finalState := inputState copy.
    [inputState isEmpty]
        whileFalse:
            [inputState := repeatExp repetition accept: self.
             finalState addAll: inputState].
    ^ finalState

visitAlternation: alternateExp
    | finalState originalState |
    originalState := inputState.
    finalState := alternateExp alternative1 accept: self.
    inputState := originalState.
    finalState addAll: (alternateExp alternative2 accept: self).
    ^ finalState

visitLiteral: literalExp
    | finalState tStream |
    finalState := Set new.
    inputState
        do:
            [:stream | tStream := stream copy.
                (tStream nextAvailable:
                    literalExp components size
                ) = literalExp components
                ifTrue: [finalState add: tStream]
            ].
    ^ finalState
```

## 10. 已知应用

Smalltalk-80编译器有一个称为ProgramNodeEnumerator的Visitor类。它主要用于那些分析源代码的算法。它未被用于代码生成和优美格式打印，尽管它也可以做这些工作。

IRISInventor[Str93]是一个用于开发三维图形应用的工具包。Inventor将一个三维场景表示成一个结点的层次结构，每一个结点代表一个几何对象或其属性。诸如绘制一个场景或是映射一个输入事件之类的一些操作要求以不同的方式遍历这个层次结构。Inventor使用称为“action”的访问者来做到这一点。生成图像、事件处理、查询、填充和决定边界框等操作都有各自相应的访问者来处理。

为使增加新的结点更容易一些，Inventor为C++实现了一个双分派方案。该方案依赖于运行时刻的类型信息和一个二维表，在这个二维表中行代表访问者而列代表结点类。表格中存储绑定于访问者和结点类的函数指针。

Mark Linton 在X Consortium的Fresco Application Toolkit设计说明书中提出了术语“Visitor” [LP93]。

## 11. 相关模式

Composite (4.3)：访问者可以用于对一个由 Composite模式定义的对象结构进行操作。

Interpreter (5.3)：访问者可以用于解释。

## 5.12 行为模式的讨论

### 5.12.1 封装变化

封装变化是很多行为模式的主题。当一个程序的某个方面的特征经常发生改变时，这些模式就定义一个封装这个方面的对象。这样当该程序的其他部分依赖于这个方面时，它们都可以与此对象协作。这些模式通常定义一个抽象类来描述这些封装变化的对象，并且通常该模式依据这个对象<sup>①</sup>来命名。例如，

- 一个Strategy对象封装一个算法 (Strategy (5.9))。
- 一个State对象封装一个与状态相关的行为 (State (305))。
- 一个Mediator对象封装对象间的协议 (Mediator (5.5))。
- 一个Iterator对象封装访问和遍历一个聚集对象中的各个构件的方法 (Iterator (5.4))。

这些模式描述了程序中很可能会改变的方面。大多数模式有两种对象：封装该方面特征的新对象，和使用这些新的对象的已有对象。如果不使用这些模式的话，通常这些新对象的功能就会变成这些已有对象的难以分割的一部分。例如，一个Strategy的代码可能会被嵌入到其Context类中，而一个State的代码可能会在该状态的Context类中直接实现。

但不是所有的对象行为模式都象这样分割功能。例如，Chain of Responsibility (5.1) 可以处理任意数目的对象（即一个链），而所有这些对象可能已经存在于系统中了。

职责链说明了行为模式间的另一个不同点：并非所有的行为模式都定义类之间的静态通信关系。职责链提供在数目可变的对象间进行通信的机制。其他模式涉及到一些作为参数传递的对象。

### 5.12.2 对象作为参数

一些模式引入总是被用作参数的对象。例如 Visitor (5.11)。一个Visitor对象是一个多态的Accept操作的参数，这个操作作用于该Visitor对象访问的对象。虽然以前通常代替Visitor模式的方法是将Visitor代码分布在一些对象结构的类中，但visitor从来都不是它所访问的对象的一部分。

其他模式定义一些可作为令牌到处传递的对象，这些对象将在稍后被调用。Command (5.2) 和Memento (5.6) 都属于这一类。在Command中，令牌代表一个请求；而在Memento中，它代表在一个对象在某个特定时刻的内部状态。在这两种情况下，令牌都可以有一个复杂的内部表示，但客户并不会意识到这一点。但这里还有一些区别：在Command模式中多态

① 这个主题也贯穿于其他种类的模式。AbstractFactory(3.1)，Builder(3.2)和Prototype(3.4)都封装了关于对象是如何创建的信息。Decorator(4.4)封装了可以被加入一个对象的职责。Bridge(4.2)将一个抽象与它的实现分离，使它们可以各自独立的变化。

很重要，因为执行 Command 对象是一个多态的操作。相反，Memento 接口非常小，以至于备忘录只能作为一个值传递。因此它很可能根本不给它的客户提供任何多态操作。

### 5.12.3 通信应该被封装还是被分布

Mediator (5.5) 和 Observer (5.7) 是相互竞争的模式。它们之间的差别是，Observer 通过引入 Observer 和 Subject 对象来分布通信，而 Mediator 对象则封装了其他对象间的通信。

在 Observer 模式中，不存在封装一个约束的单个对象，而必须是由 Observer 和 Subject 对象相互协作来维护这个约束。通信模式由观察者和目标连接的方式决定：一个目标通常有多个观察者，并且有时一个目标的观察者也是另一个观察者的目标。Mediator 模式的目的是集中而不是分布。它将维护一个约束的职责直接放在一个中介者中。

我们发现生成可复用的 Observer 和 Subject 比生成可复用的 Mediator 容易一些。Observer 模式有利于 Observer 和 Subject 间的分割和松耦合，同时这将产生粒度更细，从而更易于复用的类。

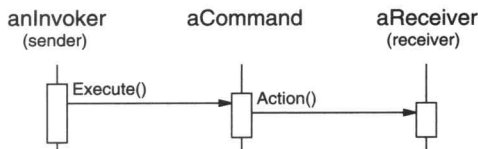
另一方面，相对于 Observer，Mediator 中的通信流更容易理解。观察者和目标通常在它们被创建后很快即被连接起来，并且很难看出此后它们在程序中是如何连接的。如果你了解 Observer 模式，你将知道观察者和目标间连接的方式是很重要的，并且你也知道寻找哪些连接。然而，Observer 模式引入的间接性仍然会使得一个系统难以理解。

Smalltalk 中的 Observer 可以用消息进行参数化以访问 Subject 的状态，因此与在 C++ 中的 Observer 相比，它们具有更大的可复用性。这使得 Smalltalk 中 Observer 比 Mediator 更具吸引力。因此一个 Smalltalk 程序员通常会使用 Observer 而一个 C++ 程序员则会使用 Mediator。

### 5.12.4 对发送者和接收者解耦

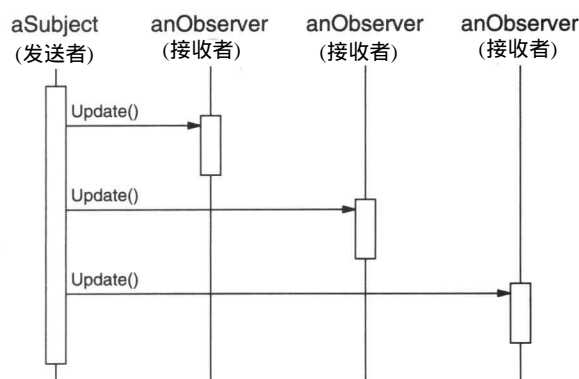
当合作的对象直接互相引用时，它们变得互相依赖，这可能会对一个系统的分层和重用性产生负面影响。命令、观察者、中介者，和职责链等模式都涉及如何对发送者和接收者解耦，但它们又各有不同的权衡考虑。

命令模式使用一个 Command 对象来定义一个发送者和一个接收者之间的绑定关系，从而支持解耦，如下图所示。



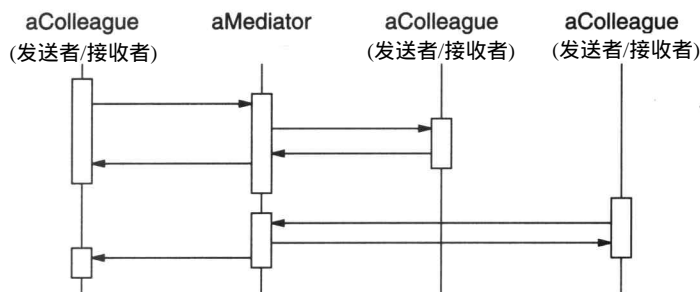
Command 对象提供了一个提交请求的简单接口（即 Execute 操作）。将发送者和接收者之间的连接定义在一个单独的对象使得该发送者可以与不同的接收者一起工作。这就将发送者与接收者解耦，使发送者更易于复用。此外，可以复用 Command 对象，用不同的发送者参数化一个接收者。虽然 Command 模式描述了避免使用生成子类的实现技术，名义上每一个发送者 - 接收者连接都需要一个子类。

观察者模式通过定义一个接口来通知目标中发生的改变，从而将发送者（目标）与接收者（观察者）解耦。Observer 定义了一个比 Command 更松的发送者 - 接收者绑定，因为一个目标可能有多个观察者，并且其数目可以在运行时变化，如下图所示。



观察者模式中的 Subject 和 Observer 接口是为了处理 Subject 的变化而设计的，因此当对象间有数据依赖时，最好用观察者模式来对它们进行解耦。

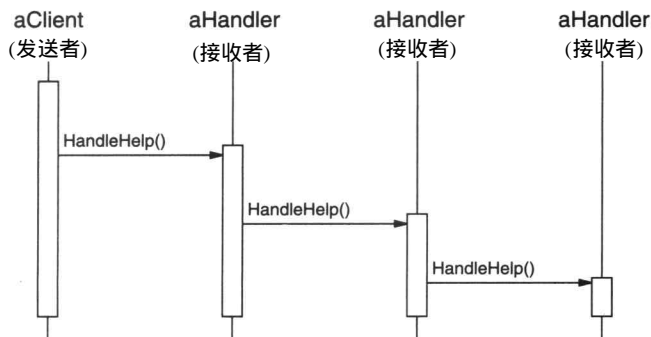
中介者模式让对象通过一个 Mediator 对象间接的互相引用，从而对它们解耦，如下图所示。



一个 Mediator 对象为各 Colleague 对象间的请求提供路由并集中它们的通信。因此各 Colleague 对象仅能通过 Mediator 接口相互交谈。因为这个接口是固定的，为增加灵活性 Mediator 可能不得不实现它自己的分发策略。可以用一定方式对请求编码并打包参数，使得 Colleague 对象可以请求的操作数目不限。

中介者模式可以减少一个系统中的子类生成，因为它将通信行为集中到一个类中而不是将其分布在各个子类中。然而，特别的分发策略通常会降低类型安全性。

最后，职责链模式通过沿一个潜在接收者链传递请求而将发送者与接收者解耦，如下图所示。



因为发送者和接收者之间的接口是固定的，职责链可能也需要一个定制的分发策略。因此它与 Mediator 一样存在类型安全的问题。如果职责链已经是系统结构的一部分，同时在链

上的多个对象中总有一个可以处理请求，那么职责链将是一个很好的将发送者和接收者解耦的方法。此外，因为链可以被简单的改变和扩展，从而该模式提供了更大的灵活性。

#### 5.12.5 总结

除了少数例外情况，各个行为设计模式之间是相互补充和相互加强的关系。例如，一个职责链中的类可能包括至少一个 Template Method(5.10)的应用。该模板方法可使用原语操作确定该对象是否应处理该请求并选择应转发的对象。职责链可以使用 Command模式将请求表示为对象。Interpreter(243)可以使用 State模式定义语法分析上下文。迭代器可以遍历一个聚合，而访问者可以对它的每一个元素进行一个操作。

行为模式也与能其他模式很好地协同工作。例如，一个使用 Composite (4.3) 模式的系统可以使用一个访问者对该复合的各成分进行一些操作。它可以使用职责链使得各成分可以通过它们的父类访问某些全局属性。它也可以使用 Decorater (4.4) 对该复合的某些部分的这些属性进行改写。它可以使用 Observer模式将一个对象结构与另一个对象结构联系起来，可以使用 State模式使得一个构件在状态改变时可以改变自身的行为。复合本身可以使用 Builder (3.2) 中的方法创建，并且它可以被系统中的其他部分当作一个 Prototype (3.4)。

设计良好的面向对象式系统通常有多个模式镶嵌在其中，但其设计者却未必使用这些术语进行思考。然而，在模式级别而不是在类或对象级别上的进行系统组装可以使更方便地获取同等的协同性。