

3 行为模式

3.1 多路通信

单客户机很容易利用单个提取（我们已经看见了被利用的变更提取任务模式，它是一个普通的任务）。然而，在特殊场合下，一个客户机将可能用到多道提取。此外，客户机可能不会提前知道有多少乃至哪个提取将会被用到。观察者（observer）、黑板（blackboard）、调解器（mediator）模式可以提供这样一种通信机会。

3.1.1 观察者

假如有一个麻省理工学院所有学生成绩的数据库，并且 6.170 的教员希望浏览选了 6.170 这门课的学生的成绩。他们可能会写一个 SpreadsheetView 类来显示来自数据库中的信息。（我们将假设观察者隐藏了 6.170 学生的信息——例如，为了重新拟定，它需要这个信息——但是它是否这样做并不是这里讨论的一个重点部分。）显示情况可能会如下图所示：

	PS1	PS2	PS3
B. Bitdiddle	30	85	80
A. Hacker	95	90	85
A. Turing	90	100	95

假设用于在成绩数据库和数据库视图窗口之间通信的代码采用了如下的接口：

```
interface GradeDBViewer {  
  
    void update(String course,String name,String assignment,int grade);  
  
}
```

当新的成绩信息有效时，（例如，一个新课题的成绩被评出并且将输入数据库，或者一个课题的成绩发生变动需要更改旧的成绩信息），成绩数据库将把那个信息通知给视图窗口。让我们假设 B.Bitdiddle 已经查询了课题 1 的成绩重新评定情况，而且重新评定的成绩确实揭示了成绩的错误：Ben's 的成绩应该要变成 30。数据库代码必须在某处调用 SpreadsheetView.Update，设想它是按照如下的方式完成的：

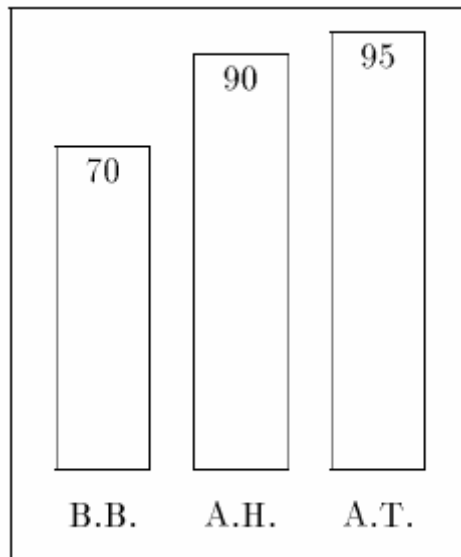
```
SpreadsheetView ssv = new SpreadsheetView();  
ssv.update("6.170", "B.Bitdiddle", "PS1", 30);
```

（短时间内，这段程序显示的精确的数值而不是关于 update 自变量的各种变量。）

于是 spreadsheet 视图窗口将重新显示如下：

	PS1	PS2	PS3
B. Bitdiddle	30	85	80
A. Hacker	95	90	85
A. Turing	90	100	95

学生们接着可能想把他们的平均成绩显示成方块图表，并以如下的视图实现：



支持这样的视图窗口除了需要 spreadsheet 视图窗口之外还要修改数据库代码：

```
SpreadsheetView ssv = new SpreadsheetView();
BargraphView bgv = new BargraphView();
```

同样的，增加一个饼图，或者删除一些视图，还需要对数据库代码程序进行更多的修改。面向对象程序设计（不提及好的程序设计准则）应该给硬编码更改提供帮助：代码应该不需要编辑和重新编译客户机或者设备而被重新利用。

观察器模式在这种情况下可以完成此目标。数据库可以支持一系列当状态被改变时会被通知的观察器，而不是去更新视图的硬代码。

```
Vector observers = new Vector();
...
for (int i=0; i<observers.size(); i++) {
    GradeDBViewer v = (GradeDBViewer) observers[i];
    v.update("6.170", "B.Bitdiddle", "PS1", 30);
}
```

为了初始化观察器的向量，数据库将提供两种额外的方法，登记——增加一个观察器，移除——移除一个观察器。

```
void register(GradeDBViewer observer) {
    observers.add(observer);
```

```
}  
Boolean remove(GradeDBViewer observer) {  
    Return observers.remove(observer);  
}
```

观察器模式允许客户机代码(它负责管理数据库和视图窗口)选择哪个观察器是动态的,且观察器即使在运行时间里也可以自由的增加和删除。

这个讨论掩盖了许多的细节。例如,客户机可能存储所有它关心的信息(可能是所有6170个成绩信息,或者可能是一部分学生的成绩信息,或者仅仅是用 DatabaseActivityViewer 更新数据库的次数),复制一部分数据库信息,或者当客户机需要的时候再访问数据库。一个相关的设计决定就是当一个更新发生时,数据库是否向客户机发送了所有潜在的相关信息,(这是被动 push 结构)或者数据库只是告诉客户机“一个更新已经发生”(这是主动 pull 结构)。主动 Pull 结构强制客户机主动请求获得信息,这可能会导致需要更多的通信信息,但一般地却使得被转移的数据变少了。

3.1.2 黑板

黑板模式推广了观察器模式,它既允许有多个数据源,又允许有多个视图窗口。它也有完全地分离发生器和接收器的作用。

黑板是一个能被所有进程读写的信息仓库。只要有其他进程对一个事件的发生有兴趣,该进程将很负责并且很专业地把这个事件的公告添加到黑板上。其他进程可以读取黑板上的信息。一个典型的例子,它们将不理睬与它们无关的大部分内容,但是它们对与自己相关的那部分信息会做出反应。向黑板寄出公告的每个进程并不知道是否会有,有一个或者多个进程会关注它的公告。

黑板一般情况下在它们的公告执行很详细的结构,但是一个便于进程间互操作的易于理解的信息格式还是需要的。一些黑板提供筛选服务,所以客户机不会看到所有的公告,而仅仅是它们中特殊的一类;其他黑板则自动地给已经提出申请的客户机发送它感兴趣的公告(它实际是主动 pull 结构)。

一个普通的广告牌(可以是物理似的,也可以是电子似的)就是黑板系统的一个例子。另一个黑板的例子是麻省理工学院的“西风”(zephyr)信息服务。

Listov 的课本把这种模式称为“白板”而不是“黑板”。前者的名字似乎更加时髦直观,但后者是标准的计算机科学专业术语,它已经被用了几十年,并且将更快的被6170外的人所认识。第一个较大的黑板系统是 Hearsay-II 语音识别系统,实现与1971~1976年。

3.1.3 调解器

调解器模式观察器和黑板的中间媒介。它可以分离发生器和接收器的信息,但不能分离对它们的控制。黑板通信是异步的,而调节器通信是同步的。在所有接收器收到信息之前调节器不把控制权返回给发生器。

3.2 组合

可查阅 4.2 组合 (composite) 模式部分内容

这个章节讨论遍历组合和/或在组合的所有子域中执行其他操作。我们的目标是能支持很多不同的操作，并且能够在不同的子域内执行它们。因为执行的操作和被操作的组合对象的类型都影响着实现的结果，所以决定如何分解这个问题就会有困难。

我们考虑一个抽象语法树的例子，或者简称 AST，它是一个计算机程序的（语法的）表示。例如，二进制加法运算+可以用 PlusOp 对象来表示：

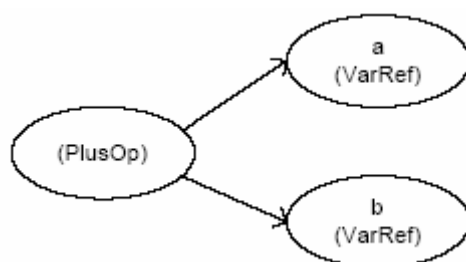
```
class PlusOp extends Expression {  
    Expression leftExp  
    Expression rightExp;  
    ...  
}
```

可变参数，赋值操作 (a=b)，和条件运算符 (a? b: c) 则是另一种类型的运算符：

```
class VarRef extends Expression {  
    String varname;  
    ...  
}  
class AssignOp extends Expression {  
    VarRef lvalue;           // 左手边; “a” 在 “a=b” 里  
    Expression rvalue;      // 右手边; “b” 在 “a=b” 里  
}  
class CondExpr extends Expression {  
    Expression condition;  
    Expression thenExpr;  
    Expression elseExpr;  
    ...  
}
```

一个完整的表示可能也会有很多其他抽象语法树的节点类型，比如 AssignOp 用于赋值，运算符等等。

+ 的一个特殊用法，例如 a+b，可以在运行时表示成



一个编译器或者其他程序分析工具通过分析目标程序创建一个抽象语法树；在分析之

后，工具执行操作，例如：在抽象生成树上进行类型检查，打印，优化，生成代码。每个操作都互不相同，每个抽象语法树节点也互不相同。

这张表的每个空格将被填入一条不同的代码：

		对象	
		CondExpr	AssignOp
操作	类型检查		
	打印		

问题在于如何组织代码，使得所有类型检查代码能够聚集在一起（在执行期间处理 CondExprs 的必要的扩展代码）或者使得处理所有一类特殊运算符代码能够聚集在一起，但是要把处理特殊操作的代码分开。

（一个相关的问题就是应该如何选择并执行恰当的代码块，不管它被定位在哪里。Java 的方法调度机制选择一个过载方法的版本来唤起基于运行时间的接收器类型，这使得它能够基于不同的操作或者对象，但不在相同时间的基础上进行调度。）

解释程序和过程程序模式（访问者，一个精致的过程程序）在组合对象，例如抽象语法树上允许运算符的操作。解释程序把类似的对象集中在一起，并把类似的操作扩展区分开。而过程程序选择类似的操作集中，而把类似的对象区分开。这就意味着：

解释程序能容易的添加一个对象，但不便添加一个操作。

过程程序能轻松的添加一个操作，但不便添加一个对象。

“容易”和“困难”取决于多少个不同的类需要被修改。当一个解释程序类被用到时，添加一个对象需要创建一个新类，而添加一个新操作则需要修改每一个存在着的类。对于过程程序则正好相反。两种模式都必须为获得那些对象特性的所有对象拥有类。如上面关于 CondExpr 和 AssignOp 的程序代码例子。这里的问题时在哪里定位那些因为所有类而存在的操作的执行过程。下面的例子将阐明这个概念。

设计一个软件系统时你选择哪种方法取决于两个因素。第一，你把系统看成操作命令中心还是操作元中心？运算法则是中心还是对象是中心？（在一个面向对象系统中通常是对象是中心）第二，系统的什么部分最有可能改变？（一个程序语言的语法很少增加新的运算符而改变，但是一个程序分析器，例如编译器则经常扩展新的功能。）这些改变应该根据你设计模式的选择而灵活的运用。

3.2.1 解释器

解释器（interpreter）模式把一类特殊对象的所有操作都聚集起来。它利用为对象准备的预先存在类并给每个类都增加一种可能的操作。例如：

```
class Expression {
    ...
    Type typecheck();
    String prettyPrint();
}
```

```

...
class AssignOp extends Expression {
    ...
    Type typecheck() { ... }
    String prettyPrint() { ... }
}
class CondExpr extends Expression {
    ...
    Type typecheck() { ... }
    String prettyPrint() { ... }
}

```

3.2.2 过程

过程（procedural）模式把执行同一个特殊操作的所有代码集中起来。它为每个类创建一个操作；类为每个类型的操作元提供独立的方法。例如，类型检查代码可能看起来如下：

```

class Typecheck {
    ...
    // 类型检查 “a ? b: c”
    Type tcCondExpr(CondExpr e) {
        Type codeType = tcExpression(e.condition); // “a” 的类型
        Type thenType = tcExpression(e.thenExpr); // “b” 的类型
        Type elseType = tcExpression(e.elseExpr); // “c” 的类型
        // 布尔类型在其他地方定义
        if ((condType == BoolType) && (thenType == elseType)) {
            // 这个表达式是非常典型的，因为它的环境是布尔环境。
            // 然后赋予类型并且其他分支有相同的类型。
            // 整个表达式的类型就是分支的类型。
            return thenType;
        } else {
            return ErrorType; // 错误类型在别处定义
        }
    }
    // 类型检查 “a=b”
    Type tcAssignOp(AssignOp e) {
        ...
    }
}

```

过程程序模式能工作的很好，但是它也有一个坏的方面：`tcExpression` 的定义。它需要调用 `tcCondExpr` 或者 `tcAssignOp` 或者 `tcVarRef` 或者一些其他函数，它取决于在运行时运算符组成成分的类型。

```

class Typecheck {
    ...

```

```

Type tcExpression(Expression e) {
  if (e instanceof PlusOp) {
    return tcPlusOp((PlusOp)e);
  } else if (e instanceof VarRef) {
    return tcVarRef((VarRef)e);
  } else if (e instanceof AssignOp) {
    return tcAssignOp((AssignOp)e);
  } else if (e instanceof CondExpr) {
    return tc CondExpr ((CondExpr)e);
  } else ...
  ...
}
}

```

维护这些代码是单调而易于出错的，并且长的级联可能会使测试运行的很慢。尽管这段代码是不合需要的即使它只产生了一次，事实上，它将又一次出现在打印类和其他每一个操作类中。代码中的系统重复通常是需要被重新设计的信号，它可能用到一个设计模式。

我们已经知道 Java 构建自动选择哪个代码将被执行是建立在一类测试（方法调度）基础上的。在测试时，它像级联那样做同样的比较和选择，但是它不打乱代码，这样可能被认为是更有效率的。访问者模式利用了这种方法。

3.2.3 访问者

访问者（visitor）模式对一个分层的数据结构（例如来自组合模式的一个结果）进行深度优先遍历（也可以用其他类型的遍历代替深度优先遍历）编码。访问者模式取决于两个操作：代码（对象）接受了访问者，访问者访问了代码（对象）。根据概念，代码结构如下：

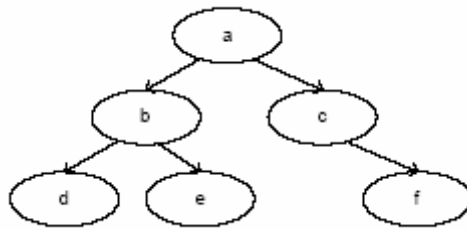
```

class Node {
  ...
  void accept(Visitor v) {
    for each child of this node {
      child.accept(v);
    }
    v.visit(this);
  }
}
class Visitor {
  ...
  void visit(Node n) {
    perform work on n
  }
}

```

accept 和 visit 方法一起工作，所以 n.accept(v)对来源于 n 的结构进行深度优先遍历，同时依次对结构的各组成部分执行用 v 表示的操作。

考虑一个组合有下列的结构：



起因于为一些访问者 v 存在的 $a.accept(v)$ 的调用顺序如下：

```
a.accept(v)
  b.accept(v)
    d.accept(v)
      v.visit(d)
    e.accept(v)
      v.visit(e)
    v.visit(b)
  c.accept(v)
    f.accept(v)
      v.visit(f)
    v.visit(c)
  v.visit(a)
```

执行真正工作的 `visit` 的调用顺序是：`d, e, b, f, c, a`；这是一个深度优先搜索。`Visit` 方法会计算节点的数目，或者执行类型检查，或是其他一些操作。访问者模式需要增加 `accept` 和 `visit` 方法；看看 Liskov 书中的一个例子，正如过程程序模式，访问者很容易增加操作（访问者）但是很难增加节点（它需要修改每一个存在的访问者）。

一个访问者很像一个 `iterator`：本质上说，一个数据结构的每个元素依次出现在 `visit` 方法中。它给了更多的机会，然而：一个访问者能够累加那些不可能单独取决于节点顺序的状态。不幸的是，上述的执行方法中，不提供任何一种方法来调用 `visit` 之间的互相通信。

对于这个问题有两种解决方法。书中建议把信息保存在一个独立的能被读写的数据结构中。（例如，堆栈）这能保证访问者和接收者不受影响，但却很难看出数据在调用中是如何流动的。

一种可以替代的方法就是把一些工作放在访问者里。

```
class Node {
  ...
  void accept(Visitor v) {
    v.visit(this);
  }
}
class Visitor {
```

```
...
void visit(Node n) {
    for each child of this node {
        child.accept(v);
    }
    perform work on n
}
}
```

这种解决方法有一些问题。首先，有很多的访问者，所以横向代码被重复了很多次而不是仅仅出现一次（因为只有一个接收器）。第二，接收器并不再真正做任何事情了。访问者本质上对它自己进行了一次深度优先搜索。这个解决方法确实有利于使得信息流更加清楚，在通常情况下，一个访问者节点取决于访问它的孩子节点的结果。

3.3. 状态

我们将不再详细讨论状态（state）模式，但你们可以为执行 `StreetNumberSet` 仔细的考虑考虑。