

# Working Classes

第 6 章

## 可以工作的类

[cc2e.com/0665](http://cc2e.com/0665) 内容

- 6.1 类的基础：抽象数据类型（ADTs）：第 126 页
- 6.2 良好的类接口：第 133 页
- 6.3 有关设计和实现的问题：第 143 页
- 6.4 创建类的原因：第 152 页
- 6.5 与具体编程语言相关的问题：第 156 页
- 6.6 超越类：包：第 156 页

### 相关章节

- 软件构建中的设计：第 5 章
- 软件架构：第 3.5 节
- 高质量的子程序：第 7 章
- 伪代码编程过程：第 9 章
- 重构：第 24 章

在计算时代的早期，程序员基于语句思考编程问题。到了 20 世纪七八十年代，程序员开始基于子程序去思考编程。进入 21 世纪，程序员以类为基础思考编程问题。



KEY POINT

类是由一组数据和子程序构成的集合，这些数据和子程序共同拥有一组内聚的、明确定义的职责。类也可以只是由一组子程序构成的集合，这些子程序提供一组内聚的服务，哪怕其中并未涉及共用的数据。成为高效程序员的一个关键就在于，当你开发程序任一部分的代码时，都能安全地忽视程序中尽可能多的其余部分。而类就是实现这一目标的首要工具。

本章将就如何创建高质量的类提供一些精辟的建议。如果你是刚刚开始接触面向对象的概念，那会觉得本章的内容比较难懂。所以请一定先阅读第 5 章“软件构建中的设计”，然后再阅读第 6.1 节“类的基础：抽象数据类型（ADTs）”。之后，你就应该可以比较轻松地阅读剩余各节了。如果你已经对类的基础知识比较熟悉，那么可以略读第 6.1 节后深入阅读第 6.2 节关于类接口的论述。另外，在本章最后“更多资源”中还包含对其他一些初级读物、高级读物以及与特定编程语言相关的资料介绍。

## 6.1 Class Foundations: Abstract Data Types (ADTs) 类的基础：抽象数据类型（ADTs）

抽象数据类型（ADT, abstract data type）是指一些数据以及对这些数据所进行的操作的集合。这些操作既向程序的其余部分描述了这些数据是怎么样的，也允许程序的其余部分改变这些数据。“抽象数据类型”概念中“数据”一词的用法有些随意。一个 ADT 可能是一个图形窗体以及所有能影响该窗体的操作；也可以是一个文件以及对这个文件进行的操作；或者是一张保险费率表以及相关操作等。

**交叉参考** 首先考虑 ADT，而后才考虑类，这是一个“‘深入一种语言去编程’而不是‘在一种语言上编程’”的例子。请参阅第 4.3 节“你在技术浪潮中的位置”以及第 34.4 节“以所用语言编程，但思路不受其约束”。

要想理解面向对象编程，首先要理解 ADT。不懂 ADT 的程序员开发出来的类只是名义上的“类”而已——实际上这种“类”只不过就是把一些稍有点儿关系的数据和子程序堆在一起。然而在理解 ADT 之后，程序员就能写出在一开始很容易实现、日后也易于修改的类来。

传统的编程教科书在讲到抽象数据类型时，总会用一些数学中的事情打岔。这些书往往会像这么写：“你可以把抽象数据类型想成一个定义有一组操作的数学模型。”这种书会给人一种感觉，好像你从不会真正用到抽象数据类型似的——除非拿它来催眠。

把抽象数据类型解释得这么空洞是完全丢了重点。抽象数据类型可以让你像在现实世界中一样操作实体，而不必在低层的实现上摆弄实体，这多令人兴奋啊。你不用再向链表中插入一个节点了，而是可以在电子表格中添加一个数据单元格，或向一组窗体类型中添加一个新类型，或给火车模型加挂一节车厢。深入挖掘能在问题领域工作（而非在底层实现领域工作）的能量吧！

### Example of the Need for an ADT 需要用到 ADT 的例子

为了展开讨论，这里先举一个例子，看看 ADT 在什么情况下会非常有用。有了例子之后我们将继续深入细节探讨。

假设你正在写一个程序，它能用不同的字体、字号和文字属性（如粗体、斜体等）来控制显示在屏幕上的文本。程序的一部分功能是控制文本的字体。如果你用一个 ADT，你就能有捆绑在相关数据上的一组操作字体的子程序——有关的数据包括字体名称、字号和文字属性等。这些子程序和数据集合为一体，就是一个 ADT。

如果不使用 ADT，你就只能用一种拼凑的方法来操纵字体了。举例来说，如果你要把字体大小改为 12 磅（point），即高度碰巧为 16 个像素（pixel），你就要写类似这样的代码：

```
currentFont.size = 16
```

如果你已经开发了一套子程序库，那么代码可能会稍微好看一些：

```
currentFont.size = PointsToPixels(12)
```

或者你还可以给该属性起一个更特定的名字，比如说：

```
currentFont.sizeOnPixels = PointsToPixels(12)
```

但你不能同时使用 `currentFont.sizeInPixels` 和 `currentFont.sizeInPoints`，因为如果同时使用这两项数据成员，`currentFont` 就无从判断到底该用哪一个了。而且，如果你在程序的很多地方都需要修改字体的大小，那么这类语句就会散布在整个程序之中。

如果你需要把字体设为粗体，你或许会写出下面的语句，这里用到了一个按位 or 运算符和一个 16 进制常量 `0x02`：

```
currentFont.attribute = CurrentFont.attribute or 0x02
```

如果你够幸运的话，也可能代码会比这样还要干净些。但使用拼凑方法的话，你能得到的最好结果也就是写成这样：

```
currentFont.attribute = CurrentFont.attribute or BOLD
```

或者是这样：

```
currentFont.bold = True
```

就修改字体大小而言，这些做法都存在一个限制，即要求调用方代码直接控制数据成员，这无疑限制了 `currentFont` 的使用。

如果你这么编写程序的话，程序中的很多地方就会充斥着类似的代码。

## Benefits of Using ADTs

### 使用 ADT 的益处

问题并不在于拼凑法是种不好的编程习惯。而是说你可以采用一种更好的编程方法来替代这种方法，从而获得下面这些好处。

**可以隐藏实现细节** 把关于字体数据类型的信息隐藏起来，意味着如果数据类型发生改变，你只需在一处修改而不会影响到整个程序。例如，除非你把实现细节隐藏在一个 ADT 中，否则当你需要把字体类型从粗体的第一种表示变成第二种表示时，就不可避免地要更改程序中所有设置粗体字体的语句，而不能仅在一处进行修改。把信息隐藏起来能保护程序的其余部分不受影响。即使你想把内存里存储的数据改为在外存里存储，或者你想把所有操作字体的子程序用另一种语言重写，也都不会影响程序的其余部分。

**改动不会影响到整个程序** 如果想让字体更丰富，而且能支持更多操作（例如变成小型大写字母、变成上标、添加删除线等）时，你只需在程序的一处进行修改即可。这一改动也不会影响到程序的其余部分。

**让接口能提供更多信息** 像 `currentFont.size = 16` 这样的语句是不够明确的，因为此处 16 的单位既可能是像素也可能是磅。语句所处的上下文环境并不能告诉你到底是哪一种单位。把所有相似的操作都集中到一个 ADT 里，就可以让你基于磅数或像素数来定义整个接口，或者把二者明确地区分开，从而有助于避免混淆。

**更容易提高性能** 如果你想提高操作字体时的性能，就可以重新编写出一些更好的子程序，而不使用来回修改整个程序。

**让程序的正确性更显而易见** 验证像 `currentFont.attribute = currentFont.attribute or 0x02` 这样的语句是否正确是很枯燥的，你可以替换成像 `currentFont.SetBoldOn()` 这样的语句，验证它是否正确就会更容易一些。对于前者，你可能会写错结构体或数据项的名字，或者用错运算符（用了 `and` 而不是 `or`），也可能会写错数值（写成了 `0x20` 而不是 `0x02`）。但对于后者，在调用 `currentFont.SetBoldOn()` 时，唯一可能出错的地方就是写错方法（成员函数）名字，因此识别它是否正确就更容易一些。

**程序更具自我说明性** 你可以改进像 `currentFont.attribute or 0x02` 这样的语句——把 `0x02` 换成 `BOLD` 或“`0x02` 所代表的具体含义”，但无论怎样修改，其可读性都不如 `currentFont.SetBoldOn()` 这条语句。



Woodfield、Dunsmore 和 Shen 曾做过这样一项研究，他们让一些计算机科学专业的研究生和高年级本科生回答关于两个程序的问题：第一个程序按功能分解为 8 个子程序，而第二个程序分解为抽象数据类型中的 8 个子程序（1981）。结果，按那些使用抽象数据类型程序的学生的得分比使用按功能划分的程序的学生高出超过 30%。

**无须在程序内到处传递数据** 在刚才那个例子里，你必须直接修改 `currentFont` 的值，或把它传给每一个要操作字体的子程序。如果你使用了抽象数据类型，那么就不用再在程序里到处传递 `currentFont` 了，也无须把它变成全局数据。ADT 中可以用一个结构体来保存 `currentFont` 的数据，而只有 ADT 里的子程序才能直接访问这些数据。ADT 之外的子程序则不必再关心这些数据。

**你可以像在现实世界中那样操作实体，而不用在底层实现上操作它** 你可以定义一些针对字体的操作，这样，程序的绝大部分就能完全以“真实世界中的字体”这个概念来操作，而不再用数组访问、结构体定义、`True` 与 `False` 等这些底层的实现概念了。

这样一来，为了定义一个抽象数据类型，你只需定义一些用来控制字体的子程序——多半就像这样：

```
currentFont.SetSizeInPoints(sizeInPoints)
currentFont.SetSizeInPixels(sizeInPixels)
currentFont.SetBoldOn()
currentFont.SetBoldOff()
currentFont.SetItalicOn()
currentFont.SetItalicOff()
currentFont.GetTypeFace(faceName)
```



KEY POINT

这些子程序里的代码可能很短——很可能就像你此前看到的那个用拼凑法控制字体时所写的代码。这里的区别在于，你已经把对字体的操作都隔离到一组子程序里了。这样就为需要操作字体的其他部分程序提供了更好的抽象层，同时它也可以在针对字体的操作发生变化时提供一层保护。

## More Examples of ADTs

### 更多的 ADT 示例

假设你开发了一套软件来控制一个核反应堆的冷却系统。你可以为这个冷却系统规定如下一些操作，从而将其视作一个抽象数据类型：

```
coolingSystem.GetTemperature()
coolingSystem.SetCirculationRate(rate)
coolingSystem.OpenValve(valveNumber)
coolingSystem.CloseValve(valveNumber)
```

实现上述各操作的代码由具体环境决定。程序的其余部分可以用这些函数来操纵冷却系统，无须为数据结构的实现、限制及变化等内部细节而操心。

下面再举一些抽象数据类型以及它们可能提供的操作：

<b>巡航控制</b>	<b>搅拌机</b>	<b>油罐</b>
设置速度	开启	填充油罐
获取当前设置	关闭	排空油罐
恢复之前的速度	设置速度	获取油罐容积
解散	启动“即时粉碎器”	获取油罐状态
	停止“即时粉碎器”	
<b>列表</b>		<b>堆栈</b>
初始化列表	<b>灯光</b>	初始化堆栈
向列表中插入条目	开启	向堆栈中推入条目
从列表中删除条目	关闭	从堆栈中弹出条目
读取列表中的下一个条目		读取栈顶条目

<b>帮助屏幕</b>	<b>菜单</b>	<b>文件</b>
添加帮助项	开始新的菜单	打开文件
删除帮助项	删除菜单	读取文件
设置当前帮助项	添加菜单项	写入文件
显示帮助屏幕	删除菜单项	设置当前文件位置
关闭帮助显示	激活菜单项	关闭文件
显示帮助索引	禁用菜单项	
返回前一屏幕	显示菜单	<b>电梯</b>
	隐藏菜单	到上一层
<b>指针</b>	获取菜单选项	到下一层
获取新分配内存的指针		到指定层
用现有指针释放内存		报告当前楼层
更改已分配内存的大小		回到底层

通过研究这些例子，你可以得出一些指导建议，下面就来说明这些指导建议：

**把常见的底层数据类型创建为 ADT 并使用这些 ADT，而不再使用底层数据类型** 大多数关于 ADT 的论述中都会关注于把常见的底层数据类型表示为 ADT。从前面的例子中可以看到，堆栈、列表、队列以及几乎所有常见的底层数据类型都可以用 ADT 来表示。

你可能会问：“这个堆栈、列表或队列又是代表什么呢？”如果堆栈代表的是一组员工，就该把它看做是一些员工而不是堆栈；如果列表代表的是一个出场演员名单，就该把它看做是出场演员名单而不是列表；如果队列代表的是电子表格中的一组单元格，就该把它看做是一组单元格而不是一个一般的队列。也就是说，要尽可能选择最高的抽象层次。

**把像文件这样的常用对象当成 ADT** 大部分编程语言中都包含有一些抽象数据类型，你可能对它们已经比较熟悉了，而只是可能并未将其视作 ADT。文件操作是个很好的例子。在向磁盘写入内容时，操作系统负责把读/写磁头定位到磁盘上的特定物理位置，如果扇区的空间用完了，还要重新分配新扇区，并负责解释那些神秘的错误代码。操作系统提供了第一层次的抽象以及在该层次上的 ADT。高层语言则提供了第二层次的抽象以及在这一更高层次上的 ADT。高级语言可以让你无须纠缠于调用操作系统 API 以及管理数据缓冲区等繁琐细节，从而让你可以把一块磁盘空间视作一个“文件”。

你可以采用类似的做法对 ADT 进行分层。如果你想在某一层次用 ADT 来提供数据结构的操作（比如说在堆栈中压入和弹出数据），没问题。而你也可以在这一抽象层次之上再创建一个针对现实世界中的问题的抽象层次。

简单的事物也可当做 ADT 为了证明抽象数据类型的实用价值，你不一定非要使用庞杂的数据类型。在前面的一组例子中，有一盏只支持两种操作（开启、关闭）的灯。你可能会觉得把简单的“开”、“关”操作放到单独的子程序中有些浪费功夫，不过即使这样简单的操作也可以通过使用 ADT 而获益。把灯和与之相关的操作放到一个 ADT 里，可以提高代码的自我说明能力，让代码更易于修改，还能把改动可能引起的后果封闭在 TurnLightOn() 和 TurnLightOff() 两个子程序内，并减少了需要到处传递的数据的项数。

不要让 ADT 依赖于其存储介质 假设你有一张保险费率表，它太大了，因此只能保存到磁盘上。你可能想把它称做一个“费率文件”然后编出类似 RateFile.Read() 这样的访问器子程序（access routine）。然而当你把它称做一个“文件”时，已经暴露了过多的数据信息。一旦对程序进行修改，把这张表存到内存中而不是磁盘上，把它当做文件的那些代码将变成不正确，而且产生误导并使人迷惑。因此，请尽量让类和访问器子程序的名字与存储数据的方式无关，并只提及抽象数据类型本身，比如说“保险费率表”。这样一来，前面这个类和访问器子程序的名字就可能是 rateTable.Read()，或更简单的 rates.Read()。

## Handling Multiple Instances of Data with ADTs in Non-Object-Oriented Environments

### 在非面向对象环境中用 ADT 处理多份数据实例

面向对象的编程语言能自动支持对同一 ADT 的多份实例的处理。如果你只是在面向对象的环境中工作，那你根本就不用自己操心处理多个实例的实现细节了，恭喜你（你可以直接去读下一节“ADT 和类”）！

如果你是在像 C 语言这样的非面向对象的环境中工作，你就必须自己手工实现支持处理多个实例的技术。一般来说，这就意味着你要为 ADT 添加一些用来创建和删除实例的服务操作，同时需要重新设计 ADT 的其他服务操作，使其能够支持多个实例。

前面字体那个 ADT 原来只是提供这些操作：

```
currentFont.SetSize(sizeInPoints)
currentFont.SetBoldOn()
currentFont.SetBoldOff()
currentFont.SetItalicOn()
currentFont.SetItalicOff()
currentFont.SetTypeFace(faceName)
```

在非面向对象的环境里，这些操作不能附着在某个类上，因此很可能要写成：

```
SetCurrentFontSize( sizeInPoints )
SetCurrentFontBoldOn()
SetCurrentFontBoldOff()
SetCurrentFontItalicOn()
SetCurrentFontItalicOff()
SetCurrentFontTypeFace( faceName )
```

如果你想一次使用更多的字体，那么就需要增加一些服务操作来创建和删除字体的实例了，比如说这样：

```
CreateFont( fontId )
DeleteFont( fontId )
SetFont( fontId )
```

这里引入了一个 `fontId` 变量，这是用来在创建和使用多个字体实例时分别控制每个实例的一种处理方法。对于其他操作，你可以采用下列三种方法之一对 ADT 的接口进行处理：

- 做法 1：每次使用 ADT 服务子程序时都明确地指明实例。在这种情况下没有“当前字体”的概念。你把 `fontId` 传给每个用来操作字体的子程序。Font ADT 的服务子程序负责跟踪所有底层的数据，而调用方代码只需使用不同的 `fontId` 即可区分多份实例。这种方法需要为每个 Font 子程序都加上一个 `fontId` 参数。
- 做法 2：明确地向 ADT 服务子程序提供所要用到的数据。采用这种方法时，你要在调用 ADT 服务的子程序里声明一个该 ADT 所要用到的数据。换句话说，你要声明一个 `Font` 数据类型，并把它传给 ADT 中的每一个服务子程序。你在设计时必须要让 ADT 的每个服务子程序在被调用时都使用这个传入的 `Font` 数据类型。用这种方法时，调用方代码无须使用 `fontId`，因为它总是自己跟踪字体数据。（虽然从 `Font` 数据类型即可直接取得所有数据，但你仍然应该仅通过 ADT 的服务子程序来访问它。这称为保持结构体“封闭”。）

这种方法的优点是，ADT 中的服务子程序不需要根据 `fontId` 来查询字体的信息。而它的缺点则是向程序的其余部分暴露了字体内部的数据，从而增加了调用方代码可能利用 ADT 内部实现细节的可能性，而这些细节本应该隐藏在 ADT 的内部。

- 做法 3：使用隐含实例（需要倍加小心）。设计一个新的服务子程序，通过调用它来让某一个特定的字体实例成为当前实例——比如说 `SetFont( fontId )`。一旦设置了当前字体，其他所有服务子程序在被调用时都会使用这个当前字体。用这种方法也无须为其他服务子程序添加 `fontId` 参数。对于简单的应用程序而言，这么做可以让使用多个实例更为顺畅。然而对于复杂的应用程序来说，这种在系统范围内对状态的依赖性就

意味着，你必须在用到字体操作的所有代码中跟踪当前的字体实例。这样一来，复杂度有可能会急剧增长，对于任何规模的应用程序来说，还有一些更好的替代方案。

在抽象数据类型的内部，你还可以选择更多处理多个实例的方法；但在抽象数据类型的外部，如果你使用非面向对象的编程语言，能选择的方法也就是这些了。

## ADTs and Classes

### ADT 和类

抽象数据类型构成了“类/class”这一概念的基础。在支持类的编程语言里，你可以把每个抽象数据类型用它自己的类实现。类还涉及到继承和多态这两个额外的概念。因此，考虑类的一种方式，就是把它看做是抽象数据类型再加上继承和多态两个概念。

## 6.2 Good Class Interfaces 良好的类接口

创建高质量的类，第一步，可能也是最重要的一步，就是创建一个好的接口。这也包括了创建一个可以通过接口来展现的合理的抽象，并确保细节仍被隐藏在抽象背后。

### Good Abstraction

#### 好的抽象

正如第 5.3 节“形成一致的抽象”中所述，抽象是一种以简化的形式来看待复杂操作的能力。类的接口为隐藏在其后的具体实现提供了一种抽象。类的接口应能提供一组明显相关的子程序。

**交叉参考** 本书中的各种语言的代码示例都是用一种风格相似的

编码约定来格式化的。关于这种约定的细节（以及关于多种编码风格的论述）请参见第 11.4 节中的“混合语言编程的注意事项”。

你可以有一个实现雇员（Employee）这一实体的类。其中可能包含雇员的姓名、地址、电话号码等数据，以及一些用来初始化并使用雇员的服务子程序。看上去可能是这样的：

**C++示例：展现良好抽象的类接口**

```
class Employee {
public:
    // public constructors and destructors
    Employee();
    Employee(
        FullName name,
        String address,
        String workPhone,
        String homePhone,
        TaxId taxIdNumber,
        JobClassification jobClass
    );
    virtual ~Employee();
    // public routines
}
```

```

    FullName GetName() const;
    String GetAddress() const;
    String GetWorkPhone() const;
    String GetHomePhone() const;
    TaxId GetTaxIdNumber() const;
    JobClassification GetJobClassification() const;
    ...
private:
    ...
};

```

在类的内部还可能会有支持这些服务的其他子程序和数据，但类的使用者并不需要了解它们。类接口的抽象能力非常有价值，因为接口中的每个子程序都在朝着这个一致的目标而工作。

一个没有经过良好抽象的类可能会包含有大量混杂的函数，就像下面这个例子一样：



#### C++示例：展现不良抽象的类接口

```

class Program {
public:
    ...
    // public routines
    void InitializeCommandStack();
    void PushCommand( Command command );
    Command PopCommand();
    void ShutdownCommandStack();
    void InitializeReportFormatting();
    void FormatReport( Report report );
    void PrintReport( Report report );
    void InitializeGlobalData();
    void ShutdownGlobalData();
    ...
private:
    ...
};

```

假设有这么一个类，其中有很多个子程序，有用来操作命令栈的，有用来格式化报表的，有用来打印报表的，还有用来初始化全局数据的。在命令栈、报表和全局数据之间很难看出什么联系。类的接口不能展现出一种一致的抽象，因此它的内聚性就很弱。应该把这些子程序重新组织到几个职能更专一的类里去，在这些类的接口中提供更好的抽象。

如果这些子程序是一个叫做 `Program` 类的一部分，那么可以这样来修改它，以提供一种一致的抽象：

**C++示例：能更好展现抽象的类接口**

```
class Program {
public:
    ...
    // public routines
    void InitializeUserInterface();
    void ShutDownUserInterface();
    void InitializeReports();
    void ShutDownReports();
    ...
private:
    ...
};
```

在清理这一接口时，把原有的一些子程序转移到其他更合适的类里面，而把另一些转为 `InitializeUserInterface()` 和其他子程序中使用的私用子程序。

这种对类的抽象进行评估的方法是基于类所具有的公用（public）子程序所构成的集合——即类的接口。即使类的整体表现一种良好的抽象，类内部的子程序也未必就能个个表现出良好的抽象，也同样要把它们设计得可以表现出很好的抽象。你可以在第 7.2 节“在子程序层上设计”里获得相关的指导建议。

为了追求设计优秀，这里给出一些创建类的抽象接口的指导建议：

**类的接口应该展现一致的抽象层次** 在考虑类的时候有一种很好的方法，就是把类看做一种用来实现抽象数据类型（ADT，见第 6.1 节）的机制。每一个类应该实现一个 ADT，并且仅实现这个 ADT。如果你发现某个类实现了不止一个 ADT，或者你不能确定究竟它实现了何种 ADT，你就应该把这个类重新组织为一个或多个定义更加明确的 ADT。

在下面这个例子中，类的接口不够协调，因为它的抽象层次不一致：

**C++示例：混合了不同层次抽象的类接口**

```
class EmployeeCensus: public ListContainer {
public:
    ...
    // public routines
    void AddEmployee( Employee employee );
    void RemoveEmployee( Employee employee );
    ...
    Employee NextItemInList();
    Employee FirstItem();
    Employee LastItem();
    ...
private:
    ...
};
```

这些子程序的抽象在  
“雇员”这一层次上。

这些子程序的抽象在  
“列表”这一层次上。

这个类展现了两个 ADT: Employee 和 ListContainer。出现这种混合的抽象，通常是源于程序员使用容器类或其他类库来实现内部逻辑，但却没有把“使用类库”这一事实隐藏起来。请自问一下，是否应该把使用容器类这一事实也归入到抽象之中？这通常都是属于应该对程序其余部分隐藏起来的实现细节，就像下面这样：

#### C++示例：有着一致抽象层次的类接口

```
class EmployeeCensus {
public:
    ...
    // public routines
    void AddEmployee( Employee employee );
    void RemoveEmployee( Employee employee );
    Employee NextEmployee();
    Employee FirstEmployee();
    Employee LastEmployee();
    ...
private:
    ListContainer m_EmployeeList;
    ...
};
```

所有这些子程序的抽象现在都是在“雇员”这一层次上了。

使用 ListContainer 库这一实现细节现在已经被隐藏起来了。

有的程序员可能会认为从 ListContainer 继承更方便，因为它支持多态，可以传递给以 ListContainer 对象为参数的外部查询函数或排序函数来使用。然而这一观点却经不起对“继承”合理性的主要测试：“继承体现了‘是一个……(is a)’关系吗？”如果从 ListContainer 中继承，就意味着 EmployeeCensus “是一个” ListContainer，这显然不对。如果 EmployeeCensus 对象的抽象是它能够被搜索或排序，这些功能就应该被明确而一致地包含在类的接口之中。

如果把类的公用子程序看做是潜水艇上用来防止进水的气锁阀 (air lock)，那么类中不一致的公用子程序就相当于是漏水的仪表盘。这些漏水的仪表盘可能不会让水像打开气锁阀那样迅速进入，但只要有足够的时间，它们还是能让潜艇沉没。实际上，这就是混杂抽象层次的后果。在修改程序时，混杂的抽象层次会让程序越来越难以理解，整个程序也会逐步堕落直到变得无法维护。



KEY POINT

**一定要理解类所实现的抽象是什么** 一些类非常相像，你必须非常仔细地理解类的接口应该捕捉的抽象到底是哪一个。我曾经开发过这样一个程序，用户可以用表格的形式来编辑信息。我们想用一个简单的栅格 (grid) 控件，但它却不能给数据输入单元格换颜色，因此我们决定用一个能提供这一功能的电子表格 (spreadsheet) 控件。

电子表格控件要比栅格控件复杂得多，它提供了 150 个子程序，而栅格控件只有 15 个。由于我们的目标是使用一个栅格控件而不是电子表格控件，因此我们让一位程序员写一个包裹类（wrapper class），隐藏起“把电子表格控件用做栅格控件”这一事实。这位程序员强烈抱怨，认为这样做是在毫无必要地增加成本，是官僚作风，然后就走了。几天以后，他带来了写好的包裹类，而这个类竟然忠实地把电子表格控件所拥有的全部 150 个子程序都暴露出来了！

这并不是我们想要的。我们要的是一个栅格控件的接口，这个接口封装了“背后实际是在用一个更为复杂的电子表格控件”的事实。那位程序员应该只暴露那 15 个栅格控件的子程序，再加上第 16 个支持设置单元格颜色的子程序。他把全部 150 个子程序都暴露出来，也就意味着一旦想要修改底层实现细节，我们就得支持 150 个公用子程序。这位程序员没有实现我们所需要的封装，也给他自己带来了大量无谓的工作。

根据具体情况的不同，正确的抽象可能是一个电子表格控件，也可能是一个栅格控件。当你不得不在两个相似的抽象之间做出选择时，请确保你的选择是正确的。

**提供成对的服务** 大多数操作都有和其相应的、相等的以及相反的操作。如果有一个操作用来把灯打开，那很可能也需要另一个操作来把灯关闭。如果有一个操作用来向列表中添加项目，那很可能也需要另一个操作来从列表中删除项目。如果有一个操作用来激活菜单项，那很可能也需要另一个操作来屏蔽菜单项。在设计一个类的时候，要检查每一个公用子程序，决定是否需要另一个与其互补的操作。不要盲目地创建相反操作，但你一定要考虑，看看是否需要它。

**把不相关的信息转移到其他类中** 有时你会发现，某个类中一半子程序使用着该类的一半数据，而另一半子程序则使用另一半数据。这时你其实已经把两个类混在一起使用了，把它们拆开吧！

**尽可能让接口可编程，而不是表达语义** 每个接口都由一个可编程（programmatic）的部分和一个语义（semantic）部分组成。可编程的部分由接口中的数据类型和其他属性构成，编译器能强制性地要求它们（在编译时检查错误）。而语义部分则由“本接口将会被怎样使用”的假定组成，而这些是无法通过编译器来强制实施的。语义接口中包含的考虑比如“RoutineA 必须在 RoutineB 之前被调用”或“如果 dataMember 未经初始化就传给 RoutineA 的话，将会导致 RoutineA 崩溃”。语义接口应通过注释说明，但要尽可能让接口不依赖于这些说明。一个接口中任何无法通过编译器强制实施的部分，就是一个可能被误用的部分。要想办法把语义接口的元素转换为编程接口的元素，比如说用 Asserts（断言）或其他的技术。

**交叉参考** 关于如何在修改代码时保持代码质量的建议，请参见第24章“重构”。

**谨防在修改时破坏接口的抽象** 在对类进行修改和扩展的过程中，你常常会发现额外所需的一些功能。这些功能并不十分适应于原有的类接口，可看上去却也很难用另一种方法来实现。举例来说，你可能会发现 Employee 类演变成了下面这个样子：



#### C++语言示例：在维护时被破坏的类接口

```
class Employee {
public:
    ...
    // public routines
    FullName GetName() const;
    Address GetAddress() const;
    PhoneNumber GetWorkPhone() const;

    ...
    bool IsJobClassificationValid( JobClassification jobClass );
    bool IsZipCodeValid( Address address );
    bool IsPhoneNumberValid( PhoneNumber phoneNumber );
    SqlDataReader GetQueryToCreateNewEmployee() const;
    SqlDataReader GetQueryToModifyEmployee() const;
    SqlDataReader GetQueryToRetrieveEmployee() const;
    ...
private:
    ...
};
```

前面代码示例中的清晰抽象，现在已经变成了由一些零散功能组成的大杂烩。在雇工和检查邮政编码、电话号码或职位的子程序之间并不存在什么逻辑上的关联，那些暴露 SQL 语句查询细节的子程序所处的抽象层次比 Employee 类也要低得多，它们都破坏了 Employee 类的抽象。

**不要添加与接口抽象不一致的公用成员** 每次你向类的接口中添加子程序时，问问“这个子程序与现有接口所提供的抽象一致吗？”如果发现不一致，就要换另一种方法来进行修改，以便能够保持抽象的完整性。

**同时考虑抽象性和内聚性** 抽象性和内聚性这两个概念之间的关系非常紧密——一个呈现出很好的抽象的类接口通常也有很高的内聚性。而具有很强内聚性的类往往也会呈现为很好的抽象，尽管这种关系并不如前者那么强。

我发现，关注类的接口所表现出来的抽象，比关注类的内聚性更有助于深入地理解类的设计。如果你发现某个类的内聚性很弱，也不知道该怎么改，那就换一种方法，问问你自己这个类是否表现为一致的抽象。

## Good Encapsulation 良好的封装

**交叉参考** 关于  
封装的更多内容  
请参见第 5.3 节  
中的“封装实现  
细节”。

正如第 5.3 节中所论述的，封装是一个比抽象更强的概念。抽象通过提供一个可以让你忽略实现细节的模型来管理复杂度，而封装则强制阻止你看到细节——即便你想这么做。

这两个概念之所以相关，是因为没有封装时，抽象往往很容易被打破。依我的经验，要么就是封装与抽象两者皆有，要么就是两者皆失。除此之外没有其他可能。

设计精良的模块  
和设计糟糕的模块  
的唯一最大区别，  
就是对其他模块  
隐藏本模块  
内部数据和其他  
实现细节的程度。  
—Joshua Bloch

**尽可能地限制类和成员的可访问性** 让可访问性 (accessibility) 尽可能低是促成封装的原则之一。当你在犹豫某个子程序的可访问性应该设为公用 (public)、私用 (private) 抑或受保护 (protected) 时，经验之举是应该采用最严格且可行的访问级别 (Meyers 1998, Bloch 2001)。我认为这是一个很好的指导建议，但我认为还有更重要的建议，即考虑“采用哪种方式能最好地保护接口抽象的完整性？”如果暴露一个子程序不会让抽象变得不一致的话，这么做就很可能是可行的。如果你不确定，那么多隐藏通常比少隐藏要好。

**不要公开暴露成员数据** 暴露成员数据会破坏封装性，从而限制你对这个抽象的控制能力。正如 Arthur Riel 所指出的，一个 Point 类如果暴露了下面这些成员的话：

```
float x;  
float y;  
float z;
```

它就破坏了封装性，因为调用方代码可以自由地使用 Point 类里面的数据，而 Point 类却甚至连这些数据什么时候被改动过都不知道 (Riel 1996)。然而，如果 Point 类暴露的是这些方法的话：

```
float GetX();  
float GetY();  
float GetZ();  
void SetX(float x);  
void SetY(float y);  
void SetZ(float z);
```

那它还是封装完好的。你无法得知底层实现用的是不是 float x、y、z，也不会知道 Point 是不是把这些数据保存为 double 然后再转换成 float，也不可能知道 Point 是不是把它们保存在月亮上，然后再从外层空间中的卫星上把它们找回来。

**避免把私用的实现细节放入类的接口中** 做到真正的封装以后，程序员们是根本看不到任何实现细节的。无论是在字面上还是在喻意上，它们都被隐藏了起来。然而，包括 C++ 在内的一些流行编程语言却从语言结构上要求程序员在类的

接口中透露实现细节。下面就是一个例子：

#### C++示例：暴露了类内部的实现细节

```
class Employee {
public:
    ...
    Employee(
        FullName name,
        String address,
        String workPhone,
        String homePhone,
        TaxId taxIdNumber,
        JobClassification jobClass
    );
    ...
    FullName GetName() const;
    String GetAddress() const;
    ...
private:
    String m_Name;
    String m_Address;
    int m_jobClass;
    ...
};
```

这里暴露了实现  
细节。

把 `private` 段的声明放到类的头文件中，看上去似乎只是小小地违背了原则，但它实际是在鼓励程序员们查阅实现细节。在这个例子中，客户代码本意是要使用 `Address` 类型来表示地址信息，但头文件中却把“地址信息用 `String` 来保存”的这一实现细节暴露了出来。

Scott Meyers 在《*Effective C++*》一书第 2 版中的第 34 条里介绍了可以解决这个问题的一个惯用技法（Meyers 1998）。他建议你把类的接口与类的实现隔离开，并在类的声明中包含一个指针，让该指针指向类的实现，但不能包含任何其他实现细节。

#### C++示例：隐藏类的实现细节

```
class Employee {
public:
    ...
    Employee( ... );
    ...
    FullName GetName() const;
    String GetAddress() const;
    ...
private:
    EmployeeImplementation *m_implementation;
};
```

这样就把实现细节  
隐藏在指针之后了。

现在你就可以把实现细节放到 `EmployeeImplementation` 类里了，这个类只对 `Employee` 类可见，而对使用 `Employee` 类的代码来说则是不可见的。

如果你已经在项目里写了很多没有采用这种方法的代码，你可能会觉得把大量的现有代码改成使用这种方法是不值得的。但是当你读到那些暴露了其实现细节的代码时，你就应该顶住诱惑，不要到类接口的私用部分去寻找关于实现细节的线索。

**不要对类的使用者做出任何假设** 类的设计和实现应该符合在类的接口中所隐含的契约。它不应该对接口会被如何使用或不会被如何使用做出任何假设——除非在接口中有过明确说明。像下面这样一段注释就显示出这个类过多地假定了它的使用者：

- 请把 `x`, `y` 和 `z` 初始化为 1.0，因为如果把它们
- 初始化为 0.0 的话，`DerivedClass` 就会崩溃。

**避免使用友元类（friend class）** 有些场合下，比如说 State 模式中，按照正确的方式使用友元类会有助于管理复杂度（Gamma et al. 1995）。但在一般情况下友元类会破坏封装，因为它让你在同一时刻需要考虑更多的代码量，从而增加了复杂度。

**不要因为一个子程序里仅使用公用子程序，就把它归入公开接口** 一个子程序仅仅使用公用的子程序这一事实并不是十分重要的考虑要素。相反，应该问的问题是，把这个子程序暴露给外界后，接口所展示的抽象是否还是一致的。

**让阅读代码比编写代码更方便** 阅读代码的次数要比编写代码多得多，即使在开发的初期也是如此。因此，为了让编写代码更方便而降低代码的可读性是非常不经济的。尤其是在创建类的接口时，即使某个子程序与接口的抽象不很相配，有时人们也往往把这个子程序加到接口里，从而让正开发的这个类的某处调用代码能更方便地使用它。然而，这段子程序的添加正是代码走下坡路的开始，所以还是不要走出这一步为好。

**要格外警惕从语义上破坏封装性** 我曾认为，只要学会避免语法错误，就能稳操胜券。然而我很快就发现，学会避免语法错误仅仅是个开始，接踵而来的是无以计数的编码错误，而其中大多数错误都比语法错误更难于诊断和更正。

—P.J. Plauger

比较起来，语义上的封装性和语法上的封装性二者的难度相差无几。从语法的角度说，要想避免窥探另一个类的内部实现细节，只要把它内部的子程序和数据都声明为 `private` 就可以了，这是相对容易办到的。然而，要想达到语义上的

封装性就完全是另一码事儿了。下面是一些类的调用方代码从语义上破坏其封装性的例子。

- 不去调用 A 类的 `InitializeOperations()` 子程序，因为你知道 A 类的 `PerformFirstOperation()` 子程序会自动调用它。
- 不在调用 `employee.Retrieve(database)` 之前去调用 `database.Connect()` 子程序，因为你知道在未建立数据库连接的时候 `employee.Retrieve()` 会去连接数据库的。
- 不去调用 A 类的 `Terminate()` 子程序，因为你知道 A 类的 `PerformFinalOperation()` 子程序已经调过它了。
- 即便在 `ObjectA` 离开作用域之后，你仍去使用由 `ObjectA` 创建的、指向 `ObjectB` 的指针或引用，因为你知道 `ObjectA` 把 `ObjectB` 放置在静态存储空间中了，因此 `ObjectB` 肯定还可以用。
- 使用 `ClassB.MAXIMUM_ELEMENTS` 而不用 `ClassA.MAXIMUM_ELEMENTS`，因为你知道它们两个的值是相等的。



上面这些例子的问题都在于，它们让调用方代码不是依赖于类的公开接口，而是依赖于类的私用实现。每当你发现自己是通过查看类的内部实现来得知该如何使用这个类的时候，你就不是在针对接口编程了，而是在透过接口针对内部实现编程了。如果你透过接口来编程的话，封装性就被破坏了，而一旦封装性开始遭到破坏，抽象能力也就快遭殃了。

如果仅仅根据类的接口文档还是无法得知如何使用一个类的话，正确的做法不是拉出这个类的源代码，从中查看其内部实现。这是个好的初衷，但却是个错误的决断。正确的做法应该是去联系类的作者，告诉他“我不知道该怎么用这个类。”而对于类的作者来说，正确的做法不是面对面地告诉你答案，而是从代码库中 `check out`（签出）类的接口文件，修改类的接口文档，再把文件 `check in`（签入）回去，然后告诉你“看看现在你知道该怎么用它了。”你希望让这一次对话出现在接口代码里，这样就能留下来让以后的程序员也能看到。你不希望让这一次对话只存在于自己的脑海里，这样会给使用该类的调用方代码烙下语义上的微妙依赖性。你也不想让这一次对话只在个人之间进行，这样只能让你的代码获益，而对其他人没有好处。

**留意过于紧密的耦合关系** “耦合 (coupling)”是指两个类之间关联的紧密程度。通常，这种关联越松 (loose) 越好。根据这一概念可以得出以下一些指导建议：

- 尽可能地限制类和成员的可访问性。
- 避免友元类，因为它们之间是紧密耦合的。

- 在基类中把数据声明为 `private` 而不是 `protected`, 以降低派生类和基类之间耦合的程度。
- 避免在类的公开接口中暴露成员数据。
- 要对从语义上破坏封装性保持警惕。
- 察觉“Demeter (得墨忒耳) 法则”(见本章第 6.3 节)。

耦合性与抽象和封装性有着非常密切的联系。紧密的耦合性总是发生在抽象不严谨或封装性遭到破坏的时候。如果一个类提供了一套不完整的服务, 其他的子程序就可能要去直接读写该类的内部数据。这样一来就把类给拆开了, 把它从一个黑盒子变成了一个玻璃盒子, 从而事实上消除了类的封装性。

## 6.3 Design and Implementation Issues 有关设计和实现的问题

给类定义合理的接口, 对于创建高质量程序起到了关键作用。然而, 类内部的设计和实现也同样重要。这一节就来论述关于包含、继承、成员函数和数据成员、类之间的耦合性、构造函数、值对象与引用对象等的问题。

### Containment (“has a” Relationships)

#### 包含 (“有一个……” 的关系)



KEY POINT

包含是一个非常简单的概念, 它表示一个类含有一个基本数据元素或对象。与包含相比, 关于继承的论述要多得多, 这是因为继承需要更多的技巧, 而且更容易出错, 而不是因为继承要比包含更好。包含才是面向对象编程中的主力技术。

**通过包含来实现“有一个/has a”的关系** 可以把包含想成是“有一个”关系。比如说, 一名雇员“有一个”姓名、“有一个”电话号码、“有一个”税收 ID 等。通常, 你可以让姓名、电话号码和税收 ID 成为 `Employee` 类的数据成员, 从而建立这种关系。

**在不得已时通过 private 继承来实现“有一个”的关系** 在某些情况下, 你会发现根本无法用把一个对象当做另一对象的成员的办法来实现包含关系。一些专家建议此时可采用 `private` 继承自所要包含的对象的办法 (Meyers 1998、Sutter 2000)。这么做的主要原因是要让外层的包含类能够访问内层被包含类的 `protected` 成员函数与数据成员。然而在实践中, 这种做法会在派生类与基类之间形成一种过于紧密的关系, 从而破坏了封装性。而且, 这种做法也往往会造成一些设计上的错误, 而这些错误是可以用“`private` 继承”之外的其他方法解决的。

**警惕有超过约 7 个数据成员的类** 研究表明, 人们在做其他事情时能记住的离散项目的个数是  $7 \pm 2$  (Miller 1956)。如果一个类包含有超过约 7 个数据成员,

请考虑要不要把它分解为几个更小的类 (Riel 1996)。如果数据成员都是整型或字符串这种简单数据类型，你可以按  $7 \pm 2$  的上限来考虑；反之，如果数据成员都是复杂对象的话，就应按  $7 \pm 2$  的下限来考虑了。

## Inheritance (“is a” Relationships)

### 继承 (“是一个……” 关系)

继承的概念是说一个类是另一个类的一种特化 (specialization)。继承的目的在于，通过“定义能为两个或更多个派生类提供共有元素的基类”的方式写出更精简的代码。其中的共有元素可以是子程序接口、内部实现、数据成员或数据类型等。继承能把这些共有的元素集中在一个基类中，从而有助于避免在多处出现重复的代码和数据。

当决定使用继承时，你必须要做如下几项决策。

- 对于每一个成员函数而言，它应该对派生类可见吗？它应该有默认的实现吗？这一默认的实现能被覆盖 (override) 吗？
- 对于每一个数据成员而言（包括变量、具名常量、枚举等），它应该对派生类可见吗？

下面就来详细解释如何考虑这些事项。

用C++进行面向对象编程时的一个最重要的法则就是：`public`继承代表的是“是一个”的关系。请把这一法则印在脑中。

—Scott Meyers

**用 public 继承来实现 “是一个……” 的关系** 当程序员决定通过继承一个现有类的方式创建一个新类时，他是在表明这个新的类是现有类的一个更为特殊的版本。基类既对派生类将会做什么设定了预期，也对派生类能怎么运作提出了限制 (Meyers 1998)。

如果派生类不准备完全遵守由基类定义的同一个接口契约，继承就不是正确的实现技术了。请考虑换用包含的方式，或者对继承体系的上层做修改。

**要么使用继承并进行详细说明，要么就不要用它** 继承给程序增加了复杂度，因此它是一种危险的技术。正如 Java 专家 Joshua Bloch 所说，“要么使用继承并进行详细说明，要么就不要用它。”如果某个类并未设计为可被继承，就应该把它的成员定义成 `non-virtual` (C++)、`final` (Java) 或 `non-overridable` (Microsoft Visual Basic)，这样你就无法继承它了。

**遵循 Liskov 替换原则 (Liskov Substitution Principle, LSP)** Barbara Liskov 在一篇面向对象编程的开创性论文中提出，除非派生类真的“是一个”更特殊的基类，否则不应该从基类继承 (Liskov 1988)。Andy Hunt 和 Dave Thomas 把 LSP 总结为：“派生类必须能通过基类的接口而被使用，且使用者无须了解两者之间的差异。”(Hunt and Thomas 2000)。

换句话说，对于基类中定义的所有子程序，用在它的任何一个派生类中时的含义都应该是相同的。

如果你有一个 `Account` 基类以及 `CheckingAccount`、`SavingsAccount`、`AutoLoanAccount` 三个派生类，那么程序员应该能调用这三个 `Account` 派生类中从 `Account` 继承而来的任何一个子程序，而无须关心到底用的是 `Account` 的哪一个派生类的对象。

如果程序遵循 Liskov 替换原则，继承就能成为降低复杂度的一个强大工具，因为它能让程序员关注于对象的一般特性而不必担心细节。如果程序员必须要不断地思考不同派生类的实现在语义上的差异，继承就只会增加复杂度了。假如说程序员必须要记得：“如果我调用的是 `CheckingAccount` 或 `SavingsAccount` 中的 `InterestRate()` 方法的话，它返回的是银行应付给消费者的利息；但如果我调用的是 `AutoLoanAccount` 中的 `InterestRate()` 方法就必须记得变号，因为它返回的是消费者要向银行支付的利息。”根据 LSP，在这个例子中 `AutoLoanAccount` 就不应该从 `Account` 继承而来，因为它的 `InterestRate()` 方法的语义同基类中 `InterestRate()` 方法的语义是不同的。

**确保只继承需要继承的部分** 派生类可以继承成员函数的接口和/或实现。表 6-1 显示了子程序可以被实现和覆盖（override）的几种形式。

表 6-1 继承而来的子程序的几种形式

	可覆盖的	不可覆盖的
提供默认实现	可覆盖的子程序	不可覆盖的子程序
未提供默认实现	抽象且可覆盖的子程序	不会用到(一个未经定义但又不让覆盖的子程序是没有意义的)

正如此表所示，继承而来的子程序有三种基本情况。

- 抽象且可覆盖的子程序是指派生类只继承了该子程序的接口，但不继承其实现。
- 可覆盖的子程序是指派生类继承了该子程序的接口及其默认实现，并且可以覆盖该默认实现。
- 不可覆盖的子程序是指派生类继承了该子程序的接口及其默认实现，但不能覆盖该默认实现。

当你选择通过继承的方式来实现一个新的类时，请针对每一个子程序仔细考虑你所希望的继承方式。仅仅是因为要继承接口所以才继承实现，或仅仅是因为要继承实现所以才继承接口，这两类情况都值得注意。如果你只是想使用一个类的实现而不是接口，那么就应该采用包含方式，而不该用继承。

**不要“覆盖”一个不可覆盖的成员函数** C++和Java两种语言都允许程序员“覆盖”那些不可覆盖的成员函数。如果一个成员函数在基类中是私用(private)的话，其派生类可以创建一个同名的成员函数。对于阅读派生类代码的程序员来说，这个函数是令人困惑的，因为它看上去似乎应该是多态的，但事实上却非如此，只是同名而已。换种方法来说，本指导建议就是“派生类中的成员函数不要与基类中不可覆盖的成员函数的重名。”

**把共用的接口、数据及操作放到继承树中尽可能高的位置** 接口、数据和操作在继承体系中的位置越高，派生类使用它们的时候就越容易。多高就算太高了呢？根据抽象性来决定吧。如果你发现把一个子程序移到更高的层次后会破坏该层对象的抽象性，就该停手了。

**只有一个实例的类是值得怀疑的** 只需要一个实例，这可能表明设计中把对象和类混为一谈了。考虑一下能否只创建一个新的对象而不是一个新的类。派生类中的差异能否用数据而不是新的类来表达呢？单件(Singleton)模式则是本条指导方针的一个特例。

**只有一个派生类的基类也值得怀疑** 每当我看到只有一个派生类的基类时，我就怀疑某个程序员又在进行“提前设计”了——也就是试图去预测未来的需要，而又常常没有真正了解未来到底需要什么。为未来要做的工作着手进行准备的最好方法，并不是去创建几层额外的、“没准以后哪天就能用得上的”基类，而是让眼下的工作成果尽可能地清晰、简单、直截了当。也就是说，不要创建任何并非绝对必要的继承结构。

**派生后覆盖了某个子程序，但在其中没做任何操作，这种情况也值得怀疑** 这通常表明基类的设计中有错误。举例来说，假设你有一个Cat(猫)类，它有一个Scratch()(抓)成员函数，可是最终你发现有些猫的爪尖儿没了，不能抓了。你可能想从Cat类派生一个叫ScratchlessCat(不能抓的猫)的类，然后覆盖Scratch()方法让它什么都不做。但这种做法有这么几个问题。

- 它修改了Cat类的接口所表达的语义，因此破坏了Cat类所代表的抽象(即接口契约)。
- 当你从它进一步派生出其他派生类时，采用这一做法会迅速失控。如果你又发现有只猫没有尾巴该怎么办？或者有只猫不捉老鼠呢？再或者有只猫不喝牛奶？最终你会派生出一堆类似ScratchlessTaillessMicelessMilkylessCat(不能抓、没尾巴、不捉老鼠、不喝牛奶的猫)这样的派生类来。

- 采用这种做法一段时间后，代码会逐渐变得混乱而难以维护，因为基类的接口和行为几乎无法让人理解其派生类的行为。

修正这一问题的位置不是在派生类，而是在最初的 Cat 类中。应该创建一个 Claw（爪子）类并让 Cat 类包含它。问题的根源在于做了所有猫都能抓的假设，因此应该从源头上解决问题，而不是到发现问题的地方修补。

**避免让继承体系过深** 面向对象的编程方法提供了大量可以用来管理复杂度的技术。然而每种强大的工具都有其危险之处，甚至有些面向对象技术还有增加——而不是降低——复杂度的趋势。

在《*Object-Oriented Design Heuristics*》(《面向对象设计的启发式方法》，1996)这本优秀著作中，Arthur Riel 建议把继承层次限制在最多 6 层之内。Arthur 是基于“神奇数字 7±2”这一理论得出这一建议的，但我仍觉得这样过于乐观了。依我的经验，大多数人在脑中同时应付超过 2 到 3 层继承时就有麻烦了。用那个“神奇数字 7±2”用来限制一个基类的派生类总数——而不是继承层次的层数——可能更为合适。

人们已经发现，过深的继承层次会显著导致错误率的增长 (Basili, Briand and Melo 1996)。每个曾经调试过复杂继承关系的人都应该知道个中原因。过深的继承层次增加了复杂度，而这恰恰与继承所应解决的问题相反。请牢牢记住首要的技术使命。请确保你在用继承来避免代码重复并使复杂度最小。

**尽量使用多态，避免大量的类型检查** 频繁重复出现的 case 语句有时是在暗示，采用继承可能是种更好的设计选择——尽管并不总是如此。下面就是一段迫切需要采用更为面向对象的方法的典型代码示例：

**C++示例：多半应该用多态来替代的case语句**

```
switch ( shape.type ) {
    case Shape_Circle:
        shape.DrawCircle();
        break;
    case Shape_Square:
        shape.DrawSquare();
        break;
    ...
}
```

在这个例子中，对 `shape.DrawCircle()` 和 `shape.DrawSquare()` 的调用应该用一个叫 `shape.Draw()` 的方法来替代，因为无论形状是圆还是方都可以调用这个方法来绘制。

另外，`case` 语句有时也用来把种类确实不同的对象或行为分开。下面就是一个在面向对象编程中合理采用 `case` 语句的例子：

#### C++示例：也许不该用多态来替代的case语句

```
switch ( ui.Command() ) {
    case Command_OpenFile:
        OpenFile();
        break;
    case Command_Print:
        Print();
        break;
    case Command_Save:
        Save();
        break;
    case Command_Exit:
        ShutDown();
        break;
    ...
}
```

此时也可以创建一个基类并派生一些派生类，再用多态的 `DoCommand()` 方法来实现每一种命令（就像 Command 模式的做法一样）。但在像这个例子一样简单的场合中，`DoCommand()` 意义实在不大，因此采用 `case` 语句才是更容易理解的方案。

**让所有数据都是 private（而非 protected）** 正如 Joshua Bloch 所言，“继承会破坏封装”（Bloch 2001）。当你从一个对象继承时，你就拥有了能够访问该对象中的 `protected` 子程序和 `protected` 数据的特权。如果派生类真的需要访问基类的属性，就应提供 `protected` 访问器函数（accessor function）。

### Multiple Inheritance

#### 多重继承

在C++的多重继承中有一个毋庸置疑的事实就是，它打开了一个潘多拉的盒子，里面是单继承所没有的复杂度。

—Scott Meyers

继承是一种强大的工具。就像用电锯取代手锯来伐木一样，当小心使用时，它非常有用，但在还没能了解应该注意的事项的人手中，它也会变得非常危险。

如果把继承比做是电锯，那么多重继承就是 20 世纪 50 年代的那种既没有防护罩、也不能自动停机的危险电锯。有时这种工具的确有用，但在大多数情况下，你最好还是把它放在仓库里为妙——至少在这儿它不会造成任何破坏。

虽然有些专家建议广泛使用多重继承（Meyer 1997），但以我的经验而言，多重继承的用途主要是定义“混合体（mixins）”，也就是一些能给对象增加一组属性的简单类。之所以称其为混合体，是因为它们可以把一些属性“混合”到派生类里面。“混合体”可以是形如 `Displayable`（可显示）、`Persistent`（持久化）、`Serializable`（可序列化）或 `Sortable`（可排序）这样的类。它们几乎总是抽象的，也不打算独立于其他对象而被单独实例化。

混合体需要使用多重继承，但只要所有的混合体之间保持完全独立，它们也不会导致典型的菱形继承（diamond-inheritance）问题。通过把一些属性夹（chunking）在一起，还能使设计方案更容易理解。程序员会更容易理解一个用了 `Displayable` 和 `Persistent` 混合体的对象——因为这样只需要实现两个属性即可——而较难理解一个需要实现 11 个更具体的子程序的对象。

Java 和 Visual Basic 语言也都认可混合体的价值，因为它们允许多重接口继承，但只能继承一个类的实现。而 C++ 则同时支持接口和实现的多重继承。程序员在决定使用多重继承之前，应该仔细地考虑其他替代方案，并谨慎地评估它可能对系统的复杂度和可理解性产生的影响。

### Why Are There So Many Rules for Inheritance 为什么有这么多关于继承的规则



KEY POINT

**交叉参考** 关于复杂度的更多内容，请参见第 5.2 节中的“软件的主要技术使命：管理复杂度”。

这一节给出了许多规则，它们能帮你远离与继承相关的麻烦。所有这些规则背后的潜台词都是在说，继承往往会让你和程序员的首要技术使命（即管理复杂度）背道而驰。从控制复杂度的角度说，你应该对继承持有非常歧视的态度。下面来总结一下何时可以使用继承，何时又该使用包含：

- 如果多个类共享数据而非行为，应该创建这些类可以包含的共用对象。
- 如果多个类共享行为而非数据，应该让它们从共同的基类继承而来，并在基类里定义共用的子程序。
- 如果多个类既共享数据也共享行为，应该让它们从一个共同的基类继承而来，并在基类里定义共用的数据和子程序。
- 当你想由基类控制接口时，使用继承；当你想自己控制接口时，使用包含。

## Member Functions and Data 成员函数和数据成员

**交叉参考** 关于子程序的总体论述, 请参见第7章“高质量的子程序”。

下面就有效地实现成员函数和数据成员给出一些指导建议。

**让类中子程序的数量尽可能少** 一份针对 C++ 程序的研究发现, 类里面的子程序的数量越多, 则出错率也就越高 (Basili, Briand, and Melo 1996)。然而, 也发现其他一些竞争因素产生的影响更显著, 包括过深的继承体系、在一个类中调用了大量的子程序, 以及类之间的强耦合等。请在保持子程序数量最少和其他这些因素之间评估利弊。

**禁止隐式地产生你不需要的成员函数和运算符** 有时你会发现应该禁止某些成员函数——比如说你想禁止赋值, 或不想让某个对象被构造。你可能会觉得, 既然编译器是自动生成这些运算符的, 你也就只能对它们放行。但是在这种情况下, 你完全可以通过把构造函数、赋值运算符或其他成员函数或运算符定义为 `private`, 从而禁止调用方代码访问它们 (把构造函数定义为 `private` 也是定义单件类 (singleton class) 时所用的标准技术, 本章后面还会讲到)。

**减少类所调用的不同子程序的数量** 一份研究发现, 类里面的错误数量与类所调用的子程序的总数是统计相关的 (Basili, Briand, and Melo 1996)。同一研究还发现, 类所用到的其他类的数量越高, 其出错率也往往会越高。这些概念有时也称为“扇入/fan in”。

**推荐阅读** 关于 Demeter 法则的更多内容, 推荐阅读《Pragmatic Programmer》(Hunt and Thomas 2000)、《Applied UML and Patterns》(Larman 2001) 以及《Fundamentals of Object-Oriented Design in UML》(Page-Jones 2000)。

**对其他类的子程序的间接调用要尽可能少** 直接的关联已经够危险了。而间接的关联——如 `account.ContactPerson().DaytimeContactInfo().PhoneNumber()` ——往往更加危险。研究人员就此总结出了一条“Demeter 法则” (Lieberherr and Holland 1989), 基本上就是说 A 对象可以任意调用它自己的所有子程序。如果 A 对象创建了一个 B 对象, 它也可以调用 B 对象的任何 (公用) 子程序, 但是它应该避免再调用由 B 对象所提供的对象中的子程序。在前面 `account` 这个例子中, 就是说 `account.ContactPerson()` 这一调用是合适的, 但 `account.ContactPerson().DaytimeContactInfo()` 则不合适。

这只不过是一种简化的解释。更多详细信息请参阅本章后面的“更多资源”一节。

**一般来说, 应尽量减小类和类之间相互合作的范围** 尽量让下面这几个数字最小:

- 所实例化的对象的种类
- 在被实例化对象上直接调用的不同子程序的数量
- 调用由其他对象返回的对象的子程序的数量

## Constructors

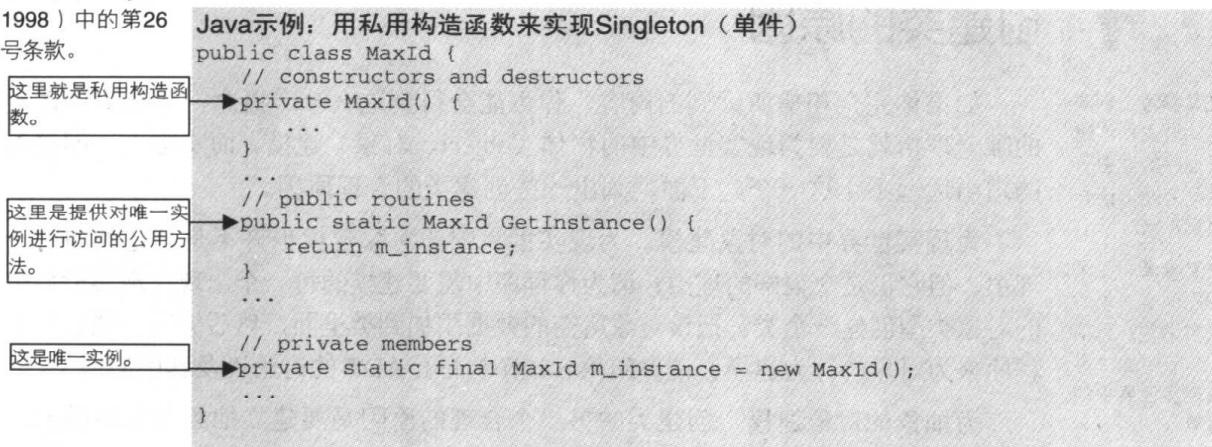
### 构造函数

接下来给出一些只适用于构造函数（constructor）的指导建议。针对构造函数的这些建议对于不同的语言（C++、Java 和 Visual Basic）都差不多。但对于析构函数（destructor）而言则略有不同，因此请查阅本章“更多资源”中列出的关于析构函数的资料。

如果可能，应该在所有的构造函数中初始化所有的数据成员 在所有的构造函数中初始化所有的数据成员是一个不难做到的防御式编程实践。

**推荐阅读** 在 C++ 中实现这一目的代码十分类似。详情请参阅《More Effective C++》( Meyers 1998 ) 中的第 26 号条款。

用私用（private）构造函数来强制实现单件属性（singleton property） 如果你想定义一个类，并需要强制规定它只能有唯一一个对象实例的话，可以把该类所有的构造函数都隐藏起来，然后对外提供一个 static 的 GetInstance() 子程序来访问该类的唯一实例。它的工作方式如下例所示：



仅在初始化 static 对象 m\_instance 时才会调用私用构造函数。用这种方法后，当你需要引用 MaxId 单件时就只需要简单地引用 MaxId.GetInstance() 即可。

优先采用深层复本（deep copies），除非论证可行，才采用浅层复本（shallow copies） 在设计副本对象的时候，你需要做出一项主要决策，即应为对象实现深拷贝（得到深层复本）还是浅拷贝（得到浅层复本）。对象的深层复本是对象成员数据逐项副制（member-wise copy）的结果；而其浅层复本则往往只是指向或引用同一个实际对象，当然，“深”和“浅”的具体含义可以有些出入。

实现浅层复本的动机一般是为了改善性能。尽管把大型的对象副制出多份副本从美学上看十分令人不快，但这样做很少会导致显著的性能损失。某几个对象可能会引起性能问题，但众所周知，程序员们很不擅长推测真正招致问题的代码（详见第 25 章“代码调整策略”）。

为了不确定的性能提高而增加复杂度是不妥的，因此，在面临选择实现深拷贝还是浅拷贝时，一种合理的方式便是优先实现深拷贝——除非能够论证浅拷贝更好。

深层复本在开发和维护方面都要比浅层复本简单。实现浅拷贝除了要用到两种方法都需要的代码之外，还要增加很多代码用于引用计数、确保安全地复制对象、安全地比较对象以及安全地删除对象等。而这些代码是很容易出错的，除非你有充分的理由，否则就应该避免它们。

如果你发现确实需要实现浅拷贝的话，Scott Meyers 写的《More Effective C++》（1996）一书的第 29 号调款就 C++ 中的这个问题进行了精辟的阐述。Martin Fowler 在《Refactoring》（《重构》，1999）一书中也论述了在深拷贝和浅拷贝之间相互转换的具体步骤（Fowler 把这两种复本对象分别称为引用对象（reference object）和值对象（value object））。

## 6.4 Reasons to Create a Class 创建类的原因

**交叉参考** 创建类的理由和创建子程序的理由有共同之处。请参见第 7.1 节。

**交叉参考** 关于识别现实对象的更多话题，见第 5.3 节中的“寻找现实世界中的对象”。

如果你完全相信所读到的内容，你可能会得到这么一个概念，即认为创建类的唯一理由就是要为现实世界中的物体（object，对象）建模。而实际上，创建类的理由远远不止这一个。下面就列出一些创建类的合理原因。

**为现实世界中的对象建模** 为现实世界中的对象建模也许不是创建类的唯一理由，但它仍是个很好的理由！请为你程序中需要建模的每一个出现在现实世界中的对象类型创建一个类。把该对象所需的数据添加到类里面，然后编写一些服务子程序来为对象的行为建模。请参阅第 6.1 节中关于 ADT 的讨论以及其中的例子。

**为抽象的对象建模** 创建类的另一个合理的原因除了要建立抽象对象的模型，所谓的抽象对象并不是一个现实世界中的具体对象，但它却能为另外一些具体的对象提供一种抽象。经典的 Shape（形状）对象就是一个很好的例子。Circle（圆）和 Square（正方形）都是真实存在的，但 Shape 则是对其他具体形状的一种抽象。

在程序设计中，抽象并不是像 Shape 一样现成就有的，因此我们必须努力工作以得出一些清晰的抽象。“从现实世界的实体中提炼出抽象的概念”这一过程是不确定的，不同的设计者会抽象出不同的共性（generalities）来。举例来说，假如我们并不了解诸如圆、正方形和三角形这样的几何形状，就可能会得出一些更不寻常的形状，比如说南瓜的形状、大头菜的形状、或是 Pontiac Aztek 似的形状。得出恰当的抽象对象是面向对象设计中的一项主要挑战。

**降低复杂度** 创建类的一个最重要的理由便是降低程序的复杂度。创建一个类来把信息隐藏起来，这样你就无须再去考虑它们。当然，当你写到这个类的时候还是要考虑这些信息的。但类写好后，你就应该能够忘掉这些细节，并能在无



须了解其内部工作原理的情况下使用这个类。其他那些创建类的原因——缩减代码空间、提高可维护性以及提高正确性——都是很好的，但一旦失去了类的抽象能力，那么复杂的应用程序对于我们的智力而言将是无法管理的了。

**隔离复杂度** 无论复杂度表现为何种形态——复杂的算法、大型数据集、或错综复杂的通讯协议等——都容易引发错误。一旦错误发生，只要它还在类的局部而未扩散到整个程序中，找到它就会比较容易。修正错误时引起的改动不会影响到其他代码，因为只有一个类需要修改，不会碰到其他代码。如果你找到了一种更好、更简单或更可靠的算法，而原有的算法已经用类隔离起来的话，就可以很容易地把它替换掉。在开发过程中，这样做可以让你更容易地尝试更多设计方案，保留最好的一种方案。

**隐藏实现细节** 想把实现细节隐藏起来的这种愿望本身便是创建类的一个绝佳理由，无论实现细节是像访问数据库那般复杂，还是像决定用数值还是字符串来存储某个特定数据成员那般寻常。

**限制变动的影响范围** 把容易变动的部分隔离开来，这样就能把变动所带来的影响限制在一个或少数几个类的范围内。把最容易变动的部分设计成最容易修改的。容易变动的部分有硬件依赖性、输入/输出、复杂数据类型、业务逻辑等。在第 5.3 节的“**隐藏秘密（信息隐藏）**”中介绍了几种常见的引起变化的根源。

**交叉参考** 关于使用全局数据的问题，请参见第 13.3 节“**全局数据**”。

**隐藏全局数据** 如果你需要用到全局数据，就可以把它的实现细节隐藏到某个类的接口背后。与直接使用全局数据相比，通过访问器子程序（access routine）来操控全局数据有很多好处。你可以改变数据结构而无须修改程序本身。你可以监视对这些数据的访问。“**使用访问器子程序**”的这条纪律还会促使你去思考有关数据是否就应该是全局的；经常你会豁然开朗地发现，“**全局数据**”原来只是对象的数据而已。

**让参数传递更顺畅** 如果你需要把一个参数在多个子程序之间传递，这有可能表明应该把这些子程序重构到一个类里，把这个参数当做对象数据来共享。实质上，让参数传递得更顺畅并不是目标，但把大量的数据到处传递是在暗示换一种类的组织方式可能会更好。

**交叉参考** 关于信息隐藏的更多详细内容，请参见第 5.3 节中的“**隐藏秘密（信息隐藏）**”。

**建立中心控制点** 在一个地方来控制一项任务是个好主意。控制可以表现为很多形式。了解一张表中记录的数目是一种形式；对文件、数据库连接、打印机等设备进行的控制又是一种。用一个类来读写数据库则是集中控制的又一种形式。如果需要把数据库转换为平坦的文件或者内存数据，有关改动也只会影响一个类。

集中控制这一概念和信息隐藏有些相似，但它具有独特的启发式功用，值得把它放到你的编程工具箱中。

**让代码更易于重用** 将代码放入精心分解（well-factored）的一组类中，比起把代码全部塞进某个更大的类里面，前者更容易在其他程序中重用。如果有一部分代码，它们只是在程序里的一个地方调用，只要它可以被理解为一个较大类的一部分，而且这部分代码可能会在其他程序中用到，就可以把它提出来形成一个单独的类。



**交叉参考** 关于实现最少所需功能的详情，参见第 24.2 节中的“程序中的一些代码似乎是在将来的某个时候才会用到的”。

美国 NASA 的软件工程实验室（Software Engineering Laboratory）曾经研究了 10 个积极追求代码重用的项目（McGarry, Waligora, and McDermott 1989）。研究结果表明，无论采用面向对象的设计方法还是以功能为导向的（functionally oriented）设计方法，在最初的项目中都没能太多地重用之前项目中的代码，因为之前的项目尚未形成充分的代码基础（code base）。然而到了后来，以功能为导向进行设计（functional design）的项目能重用之前项目中约 35% 的代码。而使用面向对象方法的项目则能重用之前项目中超过 70% 的代码。如果提前规划一下就能让你少写 70% 的代码，那当然要这样做了！

值得注意的是，NASA 这种创建可重用的类的方法并未涉及“为重用而设计”。NASA 在其项目结束时挑出了可供重用的备选代码。然后，他们进行了必要的工作来让这些代码可以重用，这些工作或被当做是主项目后期的一个特殊项目，或被当做是新项目的第一步。这种方法有助于避免“镀金”——增加一些并不实际需要的、但却会增加不必要的复杂度的功能。

**为程序族做计划** 如果你预计到某个程序会被修改，你可以把预计要被改动的部分放到单独的类里，同其他部分隔离开，这是个好主意。之后你就可以只修改这个类或用新的类来取代它，而不会影响到程序的其余部分了。仔细考虑整个程序族（family of programs）的可能情况，而不单是考虑单一程序的可能情况，这又是一种用于预先应对各种变化的强有力的方法（Parnas 1976）。

几年前我管理过一个团队，我们为客户开发一系列用于保险销售的程序。我们必须按照客户特定的保险费率、报价表格式等来定制每个程序。然而这些程序的很多部分都是相同的：用来输入潜在客户的信息的类、用来把信息存到客户数据库的类、用来查询费率的类、计算一个组的全部费率的类，等等。开发团队对程序的结构进行了规划，把每个能根据客户要求进行变化的部分都放到单独的类里面。按照开始的编程任务来计算的话，我们可能要花大约三个月的时间，但在有了新客户之后，我们仅仅需要为该客户开发出一些新类，然后让这些新类同其余代码一起工作。定制一套软件只用几天的工夫！

**把相关操作包装到一起** 即便你无法隐藏信息、共享数据或规划灵活性，你仍然可以把相关的操作合理地分组，比如分为三角函数、统计函数、字符串处理子程序、位操作子程序以及图形子程序，等等。类是把相关操作组合在一起的一种方法。除此之外，根据你所使用的编程语言不同，你还可以使用包（package）、命名空间（namespace）或头文件等方法。

**实现某种特定的重构** 第 24 章“重构”中所描述的很多特定的重构方法都会生成新的类，包括把一个类转换为两个、隐藏委托、去掉中间人以及引入扩展类等。为了能更好地实现本节所描述的任何一个目标，这些都是产生各种新类的动机。

## Classes to Avoid 应该避免的类

尽管通常情况下类是有用的，但你也可能会遇到一些麻烦。下面就是一些应该避免创建的类。

**避免创建万能类** (god class) 要避免创建什么都知道、什么都能干的万能类。如果一个类把工夫都花在用 `Get()` 方法和 `Set()` 方法向其他类索要数据（也就是说，深入到其他类的工作中并告诉它们该如何去做）的话，请考虑是否应该把这些功能组织到其他那些类中去，而不要放到万能类里（Riel 1996）。

**交叉参考** 这种类通常也叫结构体(structure)。关于结构体的更多内容，请参见第 13.1 节“结构体”。

**消除无关紧要的类** 如果一个类只包含数据但不包含行为的话，应该问问自己，它真的是一个类吗？同时应该考虑把这个类降级，让它的数据成员成为一个或多个其他类的属性。

**避免用动词命名的类** 只有行为而没有数据的类往往不是一个真正的类。请考虑把类似 `DatabaseInitialization` (数据库初始化) 或 `StringBuilder` (字符串构造器) 这样的类变成其他类的一个子程序。

## Summary of Reasons to Create a Class 总结：创建类的理由

下面总结一下创建类的合理原因：

- 对现实世界中的对象建模
- 对抽象对象建模
- 降低复杂度
- 隔离复杂度
- 隐藏实现细节
- 限制变化所影响的范围
- 隐藏全局数据

- 让参数传递更顺畅
- 创建中心控制点
- 让代码更易于重用
- 为程序族做计划
- 把相关操作放到一起
- 实现特定的重构

## 6.5 Language-Specific Issues 与具体编程语言相关的问题

不同编程语言在实现类的方法上有着很有意思的差别。请考虑一下如何在一个派生类中通过覆盖成员函数来实现多态。在 Java 中，所有的方法默认都是可以覆盖的，方法必须被定义成 `final` 才能阻止派生类对它进行覆盖。在 C++ 中，默认是不可以覆盖方法的，基类中的方法必须被定义成 `virtual` 才能被覆盖。而在 Visual Basic 中，基类中的子程序必须被定义为 `overridable`，而派生类中的子程序也必须要用 `overrides` 关键字。

下面列出跟类相关的，不同语言之间有着显著差异的一些地方：

- 在继承层次中被覆盖的构造函数和析构函数的行为
- 在异常处理时构造函数和析构函数的行为
- 默认构造函数（即无参数的构造函数）的重要性
- 析构函数或终结器（`finalizer`）的调用时机
- 和覆盖语言内置的运算符（包括赋值和等号）相关的知识
- 当对象被创建和销毁时，或当其被声明时，或者它所在的作用域退出时，处理内存的方式

关于这些事项的详细论述超出了本书的范围，不过在“更多资源”一节中提供了一些与特定语言相关的很好资源。

## 6.6 Beyond Classes: Packages 超越类：包

### 交叉参考

关于类和包的区别，请参见第5.2节中的“设计的层次”。

类是当前程序员们实现模块化（modularity）的最佳方式。不过模块化是个很庞大的话题，其影响范围要远远超出类。在过去几十年间，软件开发的进展在很

很大程度上要归功于我们在编程时进行工作的粒度的增长。首先是语句，这在当时算得上是自从机器指令以来迈进的一大步。接下来就是子程序，再后来则是类。

很显然，如果我们能有更好的工具来把对象聚合起来，我们就可能更好地朝着抽象和封装的目标迈进。Ada 语言早在十多年前就已经支持包（package）的概念了，现今 Java 语言也支持包了。如果你所用的编程语言不能直接支持包的概念，你也可以自行创建自己的包（的“可怜程序员版/poor-programmer's version”），并通过遵循下列编程标准来强制实施你的包：

- 用于区分“公用的类”和“某个包私用的类”的命名规则
- 为了区分每个类所属的包而制定的命名规则和/或代码组织规则（即项目结构）
- 规定什么包可以用其他什么包的规则，包括是否可以用继承和/或包含等

这些变通之法也是展示“在一种语言上编程”和“深入一种语言去编程”之间区别的好例子。关于这一节的更多信息，请参见第 34.4 节“以所用语言编程，但思路不受其约束”。

## CHECKLIST: Class Quality

### 核对表：类的质量

cc2e.com/0672

**交叉参考** 这是关于类的质量的考虑事项的核对表。关于创建类的步骤列表，请参见第 9 章 233 页中的“伪代码编程过程”。

#### 抽象数据类型

- 你是否把程序中的类都看做是抽象数据类型了？是否从这个角度评估它们的接口了？

#### 抽象

- 类是否有中心目的？
- 类的命名是否恰当？其名字是否表达了其中心目的？
- 类的接口是否展现了一致的抽象？
- 类的接口是否能让人清楚明白地知道该如何用它？
- 类的接口是否足够抽象，使你能不必顾虑它是如何实现其服务的？你能把类看做黑盒子吗？
- 类提供的服务是否足够完整，能让其他类无须动用其内部数据？
- 是否已从类中除去无关信息？
- 是否考虑过把类进一步分解为组件类？是否已尽可能将其分解？
- 在修改类时是否维持了其接口的完整性？

**封装**

- 是否把类的成员的可访问性降到最小？
- 是否避免暴露类中的数据成员？
- 在编程语言所许可的范围内，类是否已尽可能地对其他的类隐藏了自己的实现细节？
- 类是否避免对其使用者，包括其派生类会如何使用它做了假设？
- 类是否不依赖于其他类？它是松散耦合的吗？

**继承**

- 继承是否只用来建立“是一个/is a”的关系？也就是说，派生类是否遵循了 LSP（Liskov 替换原则）？
- 类的文档中是否记述了其继承策略？
- 派生类是否避免了“覆盖”不可覆盖的方法？
- 是否把公用的接口、数据和行为都放到尽可能高的继承层次中了？
- 继承层次是否很浅？
- 基类中所有的数据成员是否都被定义为 private 而非 protected 的了？

**跟实现相关的其他问题**

- 类中是否只有大约七个或更少的数据成员？
- 是否把类直接或间接调用其他类的子程序的数量减到最少了？
- 类是否只在绝对必要时才与其他的类相互协作？
- 是否在构造函数中初始化了所有的数据成员？
- 除非拥有经过测量的、创建浅层复本的理由，类是否都被设计为当作深层复本使用？

**与语言相关的问题**

- 你是否研究过所用编程语言里和类相关的各种特有问题？

## Additional Resources

## 更多资源

### Classes in General

#### 类，一般问题

[cc2e.com/0679](#)

Meyer, Bertrand. 《*Object-Oriented Software Construction*》(《面向对象软件构造》), 2d ed. New York, NY: Prentice Hall PTR, 1997. 这本书详细地讲解了抽象数据类型，并解释了它是如何构成类的基础的。第 14 至 16 章深入地讲解了继承。在第 15 章中, Meyer 提出了支持多重继承的正面论据。

Riel, Arthur J. 《*Object-Oriented Design Heuristics*》. Reading, MA: Addison-Wesley, 1996. 这本书就如何改善程序的设计给出了大量的建议，这些建议大多是从类的角度出发的。有几年时间我一直回避这本书，因为它看上去实在太庞大了。不过，这本书的主体部分只有约 200 页厚。Riel 的写作风格通俗易懂，令人赏心悦目。这本书内容集中，很有实用性。

### C++

#### C++

[cc2e.com/0686](#)

Meyers, Scott. 《*Effective C++: 50 Specific Ways to Improve Your Programs and Designs*》(《Effective C++, 第二版》), 2d ed. Reading, MA: Addison-Wesley, 1998.

Meyers, Scott, 1996, 《*More Effective C++: 35 New Ways to Improve Your Programs and Designs*》(《More Effective C++ 中文版》). Reading, MA: Addison-Wesley, 1996. Meyers 的这两本书都可以称得上是 C++ 程序员的圣典了。他的书在诙谐之中向我们传达了一位语言大师对于 C++ 细微之处的品味。

### Java

#### Java

[cc2e.com/0693](#)

Bloch, Joshua. 《*Effective Java Programming Language Guide*》. Boston, MA: Addison-Wesley, 2001. Bloch 在该书中给出了很多关于 Java 语言的实用建议，同时也介绍了一些更为常用的、合理的面向对象实践。

### Visual Basic

#### Visual Basic

[cc2e.com/0600](#)

下面这几本书都是与 Visual Basic 中类相关的不错的参考资料：

- Foxall, James. 《*Practical Standards for Microsoft Visual Basic .NET*》. Redmond, WA: Microsoft Press, 2003.
- Cornell, Gary, and Jonathan Morrison. 《*Programming VB .NET: A Guide for Experienced Programmers*》. Berkeley, CA: Apress, 2002.
- Barwell, Fred, et al. 《*Professional VB .NET*》, 2d ed. Wrox, 2002.

## Key Points

### 要点

- 类的接口应提供一致的抽象。很多问题都是由于违背该原则而引起的。
- 类的接口应隐藏一些信息——如某个系统接口、某项设计决策、或一些实现细节。
- 包含往往比继承更为可取——除非你要对“是一个/is a”的关系建模。
- 继承是一种有用的工具，但它却会增加复杂度，这有违于软件的首要技术使命——管理复杂度。
- 类是管理复杂度的首选工具。要在设计类时给予足够的关注，才能实现这一目标。