第3章　基于对象的设计

通过第2章的学习，你已经对好的软件设计有了一定的认识，与好的设计相辅相成的还有对象思想，下面就要介绍有关对象的概念。只是在代码中使用对象的程序员与真正掌握面向对象程序设计的程序员之间是有差别的，区别就在于他们的对象以何种方式彼此相关，以及对象与程序的总体设计之间存在怎样的关联。

本章先从过程性程序设计过渡到面向对象程序设计。也许你用对象已经很多年了，即便如此，可能还希望通过阅读本章，对如何考虑对象获得一些全新的思想。在讨论对象之间的各种不同关系时，我们还会指出程序员在构建面向对象程序时常常遭遇的陷阱。你还将了解到抽象原则与对象有什么关系。

3.1　面向对象的世界观

从过程性（C风格）代码设计过渡到面向对象代码设计时，要记住最重要的一点：面向对象程序设计（object-oriented programming，OOP）只是以另一种思路来考虑程序中发生了什么。程序员在充分理解对象是什么之前，通常会先陷到OOP的新语法和新术语中不能自拔，这种情况屡见不鲜。本章只对编写代码做简单的介绍，而把重点放在概念和思想上。有关C++对象的具体语法，请参见第8章、第9章和第10章。

3.1.1　我是在以过程性思维思考吗

过程性语言（如C）把代码划分为小的代码块，（理想情况下）每个代码块都完成一个任务。如果没有C中的过程，所有代码都会堆到main()函数里。这样的代码很难阅读，你的同事也会对此大加光火，而且这还算是好的，这种代码可能还存在更严重的危害。

所有代码是否都放在main()中，或者是否划分到较小规模的代码块中（并有描述性命名和注释），计算机对此并不关心。过程是一种抽象，这种抽象能为作为程序员以及阅读和维护代码的人提供帮助。这个概念围绕着有关程序的一个基本问题展开，即这个程序要做什么？如果直接回答这个问题（比如用英语），你就是在用过程性思维考虑问题。例如，通过回答这个问题，你可能开始设计一个股票选择程序：首先，程序从Internet上得到股票的股价。然后，基于特定的标准对此数据进行排序。接下来，对已排序的数据完成分析。最后，输出一组股票买入和卖出建议信息。开始编写代码时，你可能会把心里想的这个模型直接转换成一系列C函数：retrieveQuotes()、sortQuotes()、analyzeQuotes()和outputRecommendations()。

> 虽然C把过程称为“函数”，但是C并不是函数式语言。“函数式”（functional）一词与过程性恰好相反，它是指诸如Lisp之类的语言，这些函数式语言使用了一种完全不同的抽象。

如果你的程序会严格地遵循特定的步骤运作，过程性方法往往就很合适。不过，在当前的大型应用中，事件很少会作为一个线性序列按顺序发生。通常用户可以在任何时刻完成任何命令。过程性思维对

于数据如何表示也未做任何考虑。在前面的例子中，并没有讨论股票股价到底是什么。

如果你正是基于这样一种过程性思维方式来编写程序，也不必担心。一旦你认识到 OOP 只是另外一种思维方式，即以一种更灵活的思路来考虑软件，就会很自然地采纳这种思想。

3.1.2 面向对象思想

过程性方法的基础是问这样一个问题："这个程序要做什么?"，与此不同，面向对象方法则问了另一个问题："我要为哪些实际对象建模?"。OOP 所基于的思想是：不应将程序划分为任务，而应当划分为物理对象模型。尽管乍看上去有些抽象，不过如果从物理对象的类、组件、属性和行为等方面来考虑，这个概念就会很清楚了。

类

类（class）有助于将对象与其定义相区别。可以考虑桔子来作为例子。在谈到一般意义上的桔子时，是指它是生长在树上的一种好吃的水果，这与某个特定的桔子是不同的（比如现在我桌子上的这个桔子，它的汁溅到了我的键盘上）。

在回答"什么是桔子"这个问题时，你所说的是称为桔子的这一类东西。所有桔子都是水果。所有桔子都生长在树上。所有桔子都是桔类的一个品种。所有桔子都有某种特定的口味。类就是一个封装，其中封装了定义一类对象的有关内容。

在描述一个特定的桔子时，你所指的是一个对象。所有对象都属于一个特定的类。因为桌子上的对象是一个桔子，我知道它属于桔子类。因此，我能肯定这是一个生长在树上的水果。我还能进一步指出，这是一个一般品种的桔子，口味非常好。对象是类的一个实例（instance），这个特定实例的特性会与同一个类的其他实例有所区别。

再来看一个更具体的例子，还是考虑前面的股票选择应用的例子。在 OOP 中，"股价"是一个类，因为它定义了构成股价的抽象概念。特定的股价，如"当前 Microsoft 的股价"就是一个对象，因为这是该类的一个特定实例。

如果你原先有 C 背景，可能认为类和对象与 C 中的类型和变量有相似之处。实际上，在第 8 章中，我们将会看到，类的语法与 C 的 struct 的语法确实很类似。对象的语法与 C 风格变量的语法也相当接近。

组件

如果考虑复杂的实际对象，如飞机，就能很容易地看到，它由多个更小的组件（component）所组成。这包括机身、操纵装置、起落装置、发动机和其他一些部件。在过程性程序设计中，把复杂任务分解为较小过程可谓是一个基本环节。与此相仿，将对象考虑为由更小组件所组成，这在 OOP 中同样堪称基石。

组件实际上也是类，只不过更小一些，而且更为特定。好的面向对象程序可能有一个 Airplane 类，不过，如果完全由这个类描述飞机，那么这个类可能很大。与此不同，Airplane 类处理了多个较小的、更可管理的组件。每个组件可能又有自己的子组件。例如，起落装置是飞机的一个组件，轮子又是起落装置的一个组件。

属性

属性（propertiy）是不同对象有所区别的部分。再来看前面的 Orange 类，应该记得所有桔子都定义为属于某个品种，而且有一种特定的口味。这两个特性就是属性。所有桔子都有相同的属性，只不过值不同而已。我的桔子的口味"非常好"，你的桔子可能"很难吃"。

还可以在类层次上考虑属性。前面已经提到过，所有桔子都是水果，而且都生长在树上。这些就是这类水果（水果类）的属性，而桔子的特定品种则由特定水果对象确定。类属性由类的所有成员共享，

而对象属性出现在类的所有对象中，但是不同对象的对象属性有不同的值。

在股票选择例子中，股价就有多个对象属性，包括公司名、股票代码、当前价格和其他统计信息。

属性就是描述对象的一些特性。属性回答了这样一个问题：是什么使得这个对象与众不同？

行为

行为（behavior）回答了以下问题："这个对象会做什么？"或者是"我能对这个对象做些什么？"。在桔子这个例子中，它本身做不了多少事情，不过我们可以对它做一些工作。首先，桔子可以吃，这就是一个行为。与属性一样，可以在类层次或对象层次考虑行为。所有桔子都可以用同样的方式吃掉。不过，在其他某种行为上可能会存在差异，如沿着斜坡向下滚，非常圆的桔子和比较扁的桔子相比，在这个行为上就会反映出不同来。

股票选择例子提供了一些更为实际的行为，应该记得，在采用过程性思维考虑时，对于程序要分析股价这一点，我们确定这要作为程序的函数之一。用OOP思想来考虑时，我们可能决定股价对象可以自行分析！这样一来，分析就成为了股价对象的一个行为。

在面向对象程序设计中，会从过程中移出大量功能代码，并且转入到对象中去。通过构建有特定行为的对象，并定义对象的交互方式，OOP提供了一种更强大的机制，可以将代码与它所操作的数据相关联。

汇总

有了上述概念，可以再来审视前面的股票选择程序，并以一种面向对象的方式重新进行设计。

如前所述，由"股价"这个类起步就很合适。不过，为了得到一个股价表，程序需要有一组股价的概念，这通常称为集合（collection）。因此，更好的设计可能应该有一个表示"股价集合"的类，这个类由表示单个"股价"的较小组件所组成。

再来看属性，集合类至少要有一个属性，即所接收的实际股价表。另外，它可能还有其他的属性，如最近一次接收股价的准确日期和时间，以及所得到的股价个数。至于行为，"股价集合"要能够与一个服务器会话，得到股价，并提供一个有序的股价表。这就是"获取股价"行为。

股价类可以有前面所述的属性：公司名、股票代码、当前价格等等。同样地，前面还指出了，股价类要有一个分析行为。你还可以考虑其他行为，如买入和卖出股票等。

画一些图来表示组件之间的关系往往很有用。图3-1用多条线来表示一个"股价集合"包含有多个"股价"对象。

图　3-1

要以直观的方式表示类，还有一种有用的方法，这就是在脑海中建立程序的对象表示，同时用表列出类的属性和行为（如表3-1所示）。

表　3-1

类	相关组件	属　性	行　为
Orange（桔子）	无	颜色 口味	吃 滚 抛

（续）

类	相关组件	属　性	行　为
Collection of Stock Quotes（股价集合）	由单个股价对象组成	单个股价 时间戳 股价个数	获取股价 根据不同标准对股价排序
Stock Quote（股价）	无（目前还没有）	公司名 股票代码 当前价格等等	分析 买入 卖出

3.1.3　身处对象世界中

程序员从过程性思维转而采用面向对象思路时，通常会有所顿悟，发现需要将属性和行为结合放入对象中。有些程序员会再次查看他们以前所做的程序设计，并将某些部分重新编写为对象。另外一些程序员可能会把原来的所有代码全盘丢掉，重新启动项目来建立一个完全面向对象的应用。

要基于对象来开发软件，对此主要有两种方法。对某些人来说，对象只是对数据和功能的一个很好的封装。这些程序员会在他们的程序中多处使用对象，以使代码更加可读，更易于维护。采用这种方法的程序员将独立的代码块取出，而代之以对象，就像是外科医生植入一个起搏器一样。就其本身而言，这种方法并没有不当的地方。这些人把对象视作一种工具，在许多情况下这个工具都能带来好处。一个程序中的某些部分确实“感觉像是一个对象”，股价便是如此。这些部分可以独立出来，而且可以按真实世界中的说法来描述。

另外一些程序员则会完全采纳 OOP 范式，把所有一切都转变成对象。在他们看来，有些对象对应于真实世界中的事物，比如一个桔子或一个股价，而另外一些对象则封装了更为抽象的概念，如分拣排序器或者完成撤销工作的 undo 对象。理想的方法可能介于这两个极端之间。你的第一个面向对象程序很可能只是一个传统的过程性程序，其中散布了一些对象。你也可能会整个地从头开始，把所有东西都建立为对象，从表示 int 的类到表示主应用的类，一切都纳入到类（对象）中。不过，经过一段时间之后，可能会得到一个合适的折衷方案。

过度对象化

不要让你的开发小组成员把每一个细微琐碎的地方都转成对象，他们会为此苦不堪言，这与设计一个创造性的面向对象系统有着严格的界线。弗洛伊德曾经说过，有时变量只是一个变量而已。这句话的含义再清楚不过了。

也许你正在着手设计另一个可能畅销的连珠（Tic-Tac-Toe）游戏。对此，你打算完全遵循 OOP 方法，所以你坐下来，喝着咖啡，在一个笔记本上大致勾画出类和对象。在这样一个游戏中，通常有一个对象统管游戏全程，这个对象能够检测出谁是赢家。要表示游戏棋盘，可能会考虑采用一个 Grid 对象，由它记录每一步做的记号以及记号的位置。实际上，这个网格的组件可以是 Piece 对象，每个 Piece 对象表示一个 X 或一个 O。

别着急，先退一步看！这个设计打算专门有一个类来表示一个 X 或一个 O。这就可能存在着过度对象化。毕竟，拿一个 char 表示 X 或 O 不也很好吗？更好的做法是，为什么 Grid 不直接使用一个二维的枚举类型数组呢？Piece 对象是不是只会让代码更复杂呢？表 3-2 表示了原先提出的棋子类。

表　3-2

类	相关组件	属　性	行　为
Piece	无	X 或 O	无

这个表没有多少内容，这就充分体现出，将棋子设计为一个完整的对象可能粒度过细了。

另一方面，提前做打算的程序员可能会争辩说，Piece作为一个类尽管现在看来有些太“瘦”，但把它做成对象有利于以后的扩展，而且这也没有什么真正的损失。也许以后这要成为一个图形应用，而Piece类若能支持绘制行为可能很有用。还可以增加一些属性，如Piece的颜色，Piece最近是否移动过等等。

显然，对此并没有绝对正确的答案。重点在于，当你设计应用时应当考虑到这些问题。要记住，对象的存在是为了帮助程序员管理他们的代码。如果使用对象只是为了使代码显得“更面向对象”，就肯定存在问题了。

过于一般的对象

前面指出的是将本来不应当是对象的东西建立为对象，与此相比，也许更糟糕的是存在过于一般的对象。所有学习OOP的学生都会从“桔子”之类的例子开始入手，这些东西确实是对象，这一点毋庸置疑。在实际的编码中，对象则可能相当抽象。许多OOP程序都有一个“应用对象”，不过应用并不是你在实际世界中看得到摸得着的东西。但是，把应用表示为一个对象可能很有用，因为应用本身有一些特定的属性和行为。

所谓过于一般的对象是指，对象根本没有表示任何特定的东西。程序员可能想建立一个灵活的或者可重用的对象，但是最后得到的可能只是一个莫名其妙的对象。例如，假设有一个组织和显示媒体的程序，它可以建立照片目录、对电子音乐集进行组织，还可以用作个人杂志。如果采用过于一般的方法，会把所有这些东西都认为是“媒体”对象，并建立一个类来满足所有媒体格式。它可能有一个名为“data”（数据）的属性，其中包含图像、歌曲或杂志某项内容的原始二进制数据，具体是什么取决于媒体的类型。它还可能有一个名为“perform”（完成）的行为，这个行为会适当地绘制图像、播放歌曲，或者打开杂志的某项内容以供编辑。

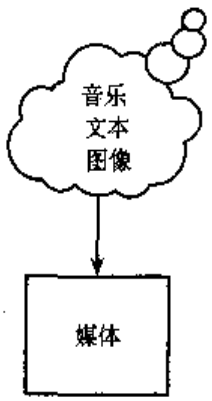

图 3-2

从这个类的属性和行为的名字上就能暗示出它可能太过一般了。“数据”一词本身没有太多的实际含义，之所以必须使用这样一个一般性的词，原因在于这个类做了过分扩展，要用于三个完全不同的用途。类似地，“完成”行为也会在三种不同的情况下做出截然不同的事情。最后要说明的是，这个设计为什么会过于一般，这是因为“媒体”并不是一个特定的对象。用户界面中没有这个对象，在实际生活中也找不到这个对象，甚至在程序员看来它也不是一个对象。要看一个类是不是太过一般了，有一个主要的线索，即是否将程序员脑海中的多种思想全都纠集在一起而成为一个对象，如图3-2所示。

3.1.4 对象关系

作为一个程序员，肯定会遇到这样一些情况：多个类有一些共同的特性，或者至少看上去这些类相互之间存在某种关联。例如，尽管在一个数字目录程序中创建一个“媒体”对象来表示图像、音乐和文本的做法太过一般，但是这些对象确实有一些共同的特性。你可能希望这些对象都能记录最后一次修改时的日期和时间，或者希望它们都支持一种删除行为。

面向对象语言为处理对象之间的这些关系提供了大量机制。难点在于要理解关系到底是什么。主要有两种对象关系，一种是*has-a*关系，还有一种是*is-a*关系。

has-a关系

参与一个has-a或聚集（aggregation）关系的对象都遵循以下模式，即A有一个B，或者A包含一个B。在这种关系中，可以把一个对象看作是另一个对象的一部分。如前面所定义的，组件一般就表示一种

has-a 关系，因为组件描述了构成其他对象的对象。

在实际中，动物园和一只猴子之间的关系就是这种 has-a 关系。可以说动物园有一只猴子，或者动物园中包含一个猴子。如果要用代码模拟动物园，可能有动物园对象，它会有一个猴子组件。

通常，考虑用户界面的情况将有助于理解对象关系。这是因为，尽管并非所有 UI（用户界面）都采用 OOP 实现（不过目前来讲，大多数 UI 都是如此），但是屏幕上的可视化元素都非常适于实现为对象。在 UI 领域中，可以打个比方，窗口包含一个按钮，这就是一个 has-a 关系。按钮和窗口完全是两个单独的对象，但是它们显然存在某种形式的关联。由于按钮在窗口内部，我们可以说窗口有一个按钮。

图 3-3 显示了真实世界和用户界面中的一些 has-a 关系。

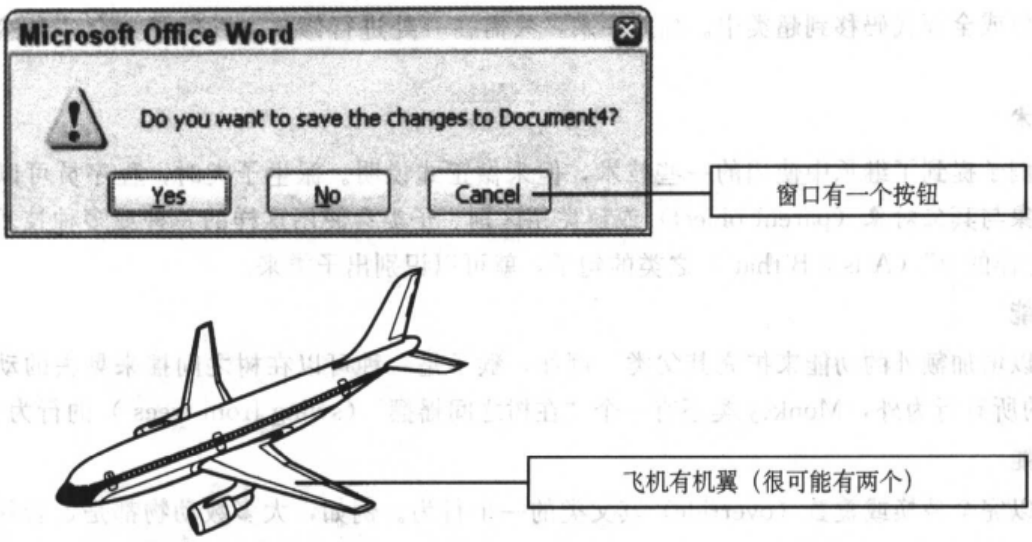

图　3-3

is-a 关系（继承）

is-a 关系是面向对象程序设计中的一个相当基本的概念，它有很多名字，包括派生（subclassing）、扩展（extending）和继承（inheriting）。真实世界的对象有属性和行为，类正是对这一事实进行建模。这些对象可能以层次体系加以组织，继承则是对这一事实进行建模。这些层次体系就指示出了 is-a 关系。

基本说来，继承遵循以下模式：A 是一个 B，或者 A 实际上很像 B，这一点可能很复杂。还是考虑简单的情况，再来看前面的动物园，不过假设动物园里除了猴子之外还有一些其他的动物。单凭这一句话就已经建立了一个关系，猴子是一个动物。类似地，长颈鹿是一个动物，袋鼠是一个动物，企鹅是一个动物。如果你意识到，猴子、长颈鹿、袋鼠和企鹅都有某些共同的东西，就能发现继承的神奇之处了。这些共同性正是动物的一般特性。

对于程序员来说，继承的含义是指，可以定义一个 Animal 类，其中封装每种动物都有的属性（体态大小、生活地区、食物等等）和行为（移动、吃、睡觉）。特定的动物（如猴子）则作为 Animal 的子类，因为猴子包含动物的所有特性（要记住，猴子就是一个动物再加上另外一些突出的特性，正是这些特性使猴子与其他动物有所不同）。图 3-4 显示了动物的一个继承图。箭头指示了 is-a 关系的方向。

猴子和长颈鹿是不同类型的动物，与此类似，用户界面通常也有类型不同的按钮。例如，复选框就是一种按钮。假设按钮是一个可以单击、可以完成某个动作的 UI 元素，Checkbox 类则扩展了 Button 类，并增加了状态，即复选框是否被选中。

将类加入 is-a 关系时，这样做的一个目标是将共同的功能置于超类（superclass）中，所谓超类就是其他类所扩展的类。如果发现所有子类都有一些很相似的代码，或者几乎完全相同的代码，就可以考虑

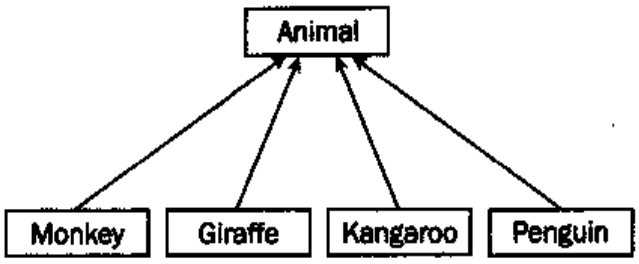

图　3-4

将这样的部分或全部代码移到超类中。如此一来，只需在一处进行修改，以后子类会“自然”地得到共享的功能。

继承技术

前面的例子提到了继承中使用的一些技术，但未做正式说明。派生子类时，程序员可以采用多种方法使一个对象与其父对象（parent object）或超类相区别。子类会使用这样的一种或多种技术，而且通过“A是一个怎样的B”（A is a B that ）之类的句子，就可以识别出子类来。

增加功能

子类可以增加额外的功能来扩充其父类。例如，猴子是一种可以在树之间摇来晃去的动物。除了拥有Animal的所有行为外，Monkey类还有一个“在树之间摇摆”（swing from trees ）的行为。

替换功能

子类可以完全替换或覆盖（override）其父类的一个行为。例如，大多数动物都是走着移动，所以你可能为Animal类提供了一个模拟走路的move行为。如果是这样，要知道，袋鼠就是跳着移动（而非走着行进）的动物。Animal超类的所有其他属性和行为对袋鼠仍然适用，Kangaroo子类只是会改变move行为的完成方式。当然，如果你发现需要把超类中的所有功能都替换掉，这就可能说明，在这种情况下派生子类根本不是一种合适的做法。

增加属性

子类除了拥有从超类继承得到的属性外，还可以增加新的属性。企鹅不仅有动物的所有属性，还有一个喙大小（beak size）的属性。

替换属性

C++提供了一种覆盖属性的方法，这与覆盖行为的方法很类似。不过，一般不应当这样做。重要的是不要把下面两个概念搞混了，即替换一个属性是一回事，而子类有不同的属性值又是另一回事。例如，所有动物都有一个diet属性，这说明它们都要吃东西。猴子爱吃香蕉（bananas），而企鹅要吃鱼（fish），不过无论是猴子还是企鹅都没有替换diet属性，只是为该属性所赋的值存在差别。

多态与代码重用

多态（Polymorphism）概念是指，可以交替地使用遵循一组标准属性和行为的对象（译者注：这句话可以理解为：程序中会交替出现不同的对象，而且这些对象对于标准属性和行为可以有不同的实现）。类定义相当于对象和与之交互的代码（即使用对象的代码）之间的一个合约。根据定义，任何Monkey对象都必须支持Monkey类的属性和行为。这个概念还可以延伸到超类。由于所有猴子都是动物，所有Monkey对象还要支持Animal类的属性和行为。

多态是面向对象程序设计中的一大妙处，因为它充分利用了继承的精神。在对动物园的模拟中，我们可以编写程序循环处理动物园中的所有动物，让每个动物移动一次。由于所有动物都是Animal类的一

员，它们都知道如何移动。有些动物覆盖了移动行为，这也是设计中最“酷”的地方，我们的代码只是告诉每个动物要移动，而无需知道（也不必关心）这是一种什么动物。每个动物都会按照它所知道的方式去移动。

除了多态外，派生子类还有另一个理由。通常，这样做只是为了充分利用既有的代码。例如，如果需要一个类来播放带回音效果的音乐，而你的同事已经写过一个类，可以不带任何效果地播放音乐，你就能扩展现有的类，并加入新的功能。is-a 关系仍然适用（回音特效音乐播放器也是一个音乐播放器，只是增加了回音效果而已），不过，你可能不希望交替地使用这些类。你最后要得到的是两个单独的类，并用于程序中完全不同的部分（或者可能甚至在完全不同的程序中使用），它们之所以存在关联，只是想避免重复做同样的工作。

has-a 与 is-a 间的清晰界线

在真实世界中，很容易区别对象之间的 has-a 和 is-a 关系。没有人会说桔子有一个水果，正常的说法是：桔子是一种水果（is a）。在代码中，有时事情则并非这么一目了然。

下面考虑一个表示散列表的假想的类。散列表是一种数据结构，可以高效地建立键与值的映射。例如，一家保险公司可能使用一个 Hashtable 类将成员 ID 映射至成员名，这样给定一个 ID 时，就能很容易地找到相应的成员名。成员 ID 是键（key），而成员名就是值（value）。

在一个标准的散列表实现中，每个键都有一个值。如果 ID 14534 映射至成员名“Kleper, Scott”，它就不能将这个 ID 又映射至成员名“Kleper, Marni”。在大多数实现中，如果想要为一个已经有值的键增加另一个值，前一个值就会丢掉。换句话说，如果 ID 14534 映射至“Kleper, Scott”，而你又想把 ID 14534 分配给“Kleper, Marni”，那么 Scott 的保险就没有了，下面显示了对假想散列表 enter()行为的两个调用，以及这两个调用之后所得到的散列表内容。hash. enter 有些类似于 C++的对象语法。可以认为这表示“使用 hash 对象的 enter 行为。”

```
hash. enter (14534, "Kleper, Scott");
```

键	值
14534	"Kleper, Scott" [字符串]

```
hash. enter (14534, "Kleper, Marni");
```

键	值
14534	"Kleper, Marni" [字符串]

假设有一个类似于散列表的数据结构，不过对应一个给定键允许有多个值，不难想像出这样一个数据结构的使用。在保险例子中，如果一个家庭对应一个 ID，这样就可能有多个名字（一个家庭中的多个成员）对应同一个 ID。由于这种数据结构与散列表很相似，最好能以某种方式利用散列表的功能。散列表对应一个键只能有一个值，但是这个值可以是任何东西。值可以不是一个字符串，而是一个集合（如数组或列表），其中包含对应键的多个值。每次为一个既有的 ID 增加一个新成员时，只需把新的成员名增加到集合中。由以下调用序列可以了解这种做法。

```
Collection collection;                  // Make a new collection.
collection.insert("Kleper, Scott"); // Add a new element to the collection.
hash.enter(14534, collection);          // Enter the collection into the table.
```

键	值
14534	{"Kleper, Scott"} [集合]

```
Collection collection = hash.get(14534);// Retrieve the existing collection.
collection.insert("Kleper, Marni");     // Add a new element to the collection.
hash.enter(14534, collection);          // Replace the collection with the updated one.
```

键	值
14534	{"Kleper, Scott"," Kleper, Marni"}[集合]

如果使用集合而不是字符串，这会很繁琐，而且需要大量重复性的代码。更好的做法是把这种多值功能包装在一个单独的类中，可以称之为 MultiHash。MultiHash 类与 Hashtable 的工作类似，不过其底层实现中，将每个值存储为一个字符串集合，而不是单一的一个串。显然，MultiHash 与 Hashtable 存在某种关联，因为它还是在使用一个散列表存储数据。所不明确的是，这要构成一种 is-a 关系还是一种 has-a 关系。

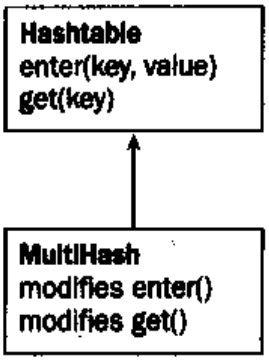

图 3-5

先来考虑 is-a 关系，假设 MultiHash 是 Hashtable 的子类。它必须对“向表中增加一项”这个行为进行覆盖，从而可以创建一个集合并增加新元素，或者是获取既有的集合并增加新元素。此外还要覆盖“获取值”的行为。例如，可以把对应一个给定键的所有值追加到一个串中。看上去这是一个很合理的设计。尽管它覆盖了超类的所有行为，但是由于子类中使用了原来的行为，因此也算利用了超类的行为。这种方法如图 3-5 所示。

下面再把它考虑为一个 has-a 关系。MultiHash 不再作为子类，它是一个单独的类，不过其中包含（contain）一个 Hashtable 对象。它可能有一个与 Hashtable 很类似的接口，不过无需完全相同。在底层实现中，用户向 MultiHash 增加内容时，所增加的内容实际上会包装在一个集合中，并放在 Hashtable 对象里。看上去似乎也很合理，如图 3-6 所示。

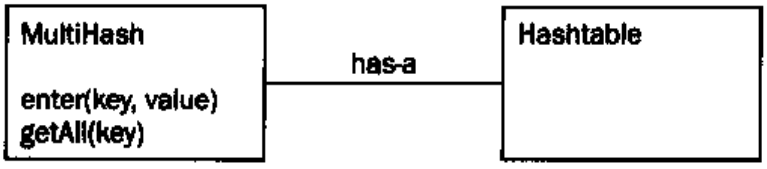

图 3-6

那么，哪一种解决方案才算正确呢？对此没有明确的答案，不过本书作者之一曾经写过一个 MultiHash 类（用于成品阶段），他就把这看作是一种 has-a 关系。其主要原因是，这样可以对所提供的接口进行修改，而不必操心维护散列表功能。例如，在图 3-6 中，get 行为修改为 getAll，以表明它要得到 MultiHash 中对应某个键的所有值（但不必修改 Hashtable）。另外，基于 has-a 关系，无需操心散列表带进来的所有功能（即不必了解这些功能的细节）。例如，如果散列表类支持一种行为，可以得到总共有多少个值，利用这一点，MultiHash 就能直接报告集合的个数，除非 MultiHash 有意要覆盖。

不过，有人可能会提出很充分的理由，认为 MultiHash 实际上就是带有一些新功能的 Hashtable，而

这应当是一个 is-a 关系。实际上这里的关键在于，在这两种关系之间，有时存在着一个清晰的界线，应当考虑类将如何使用，还要看看你所构建的类是否充分利用了另一个类的某些功能，或者实际上它就是另一个类，只不过增加或修改了某些功能而已。

针对 MultiHash 类可以采用的两种方法，表 3-3 列出了支持和反对这两种方法的理由。

表　3-3

	is-a	has-a
支持的理由	• 基本说来，MultiHash 与 Hashtable 相比，是存在不同特性的同一个抽象 • 它提供了与 Hashtable（几乎）相同的行为	• MultiHash 可以有任何有用的行为，而无需操心散列表有什么行为 • MultiHash 的实现可以修改为使用其他结构（不使用 Hashtable），而不会改变外部行为
反对的理由	• 根据定义，散列表对应每个键只有一个值。要说 MultiHash 是一个散列表，显然与上述定义相悖！MultiHash 完全覆盖了 Hashtable 的两个行为，从这一点上看，也强有力地表明这种设计存在问题 • Hashtable 的一些未知或不合适的属性或行为可能会“带入到” MultiHash 中	• 从某种意义上说，MultiHash 提供了新的行为，这也算是做了重复的工作 • Hashtable 另外的一些属性和行为可能很有用

not-a 关系

在考虑类存在哪一种类型的关系时，还要考虑这些类是否真的有关系。不要因为你对面向对象设计的狂热，而带来大量完全不必要的类/子类关系。

某些东西在真实世界中显然是相关的，但在编写代码时并不存在真正的关系，此时就会出现一个陷阱。在现实中，Mustang 是一种 Ford 汽车，但仅凭这一点并不表示，在编写一个汽车模拟程序时，Mustang 一定要作为 Ford 的一个子类。OO 层次体系需要对功能关系建模，而不是对人为的关系建模。图 3-7 显示的关系作为本体或者作为层次来讲是有意义的，但是在代码中可能并不表示有意义的关系。

要想避免没有必要的派生子类，最好的方法是先粗略地完成设计。对于每个类和子类，写出你打算在其中放置（实现）哪些属性和行为。如果发现一个类没有自己的特定属性或行为，或者一个类的所有属性和行为都会被其子类完全覆盖，就应该重新考虑设计了。

层次体系

就像类 A 可能是 B 的一个超类一样，B 也可能是 C 的一个超类。面向对象层次体系可以为诸如此类的多层关系建模。如果动物园中有更多的动物，模拟这样一个动物园时，可以把每种动物设计为一个公共 Animal 类的子类，如图 3-8 所示。

在编写上述各个子类时，可能会发现它们存在很多类似之处。如果出现这种情况，应当考虑将共同的地方放到一个公共的父类中。可以了解到，狮子（Lion）和豹子（Panther）有相同的移动方式，而且它们吃的东西也相同，这就表明可以为它们建立一个父类 BigCat。你还可以把 Animal 类进一步划分为包括水生动物（WaterAnimal）和有袋动物（Marsupial）。图 3-9 所示的层次设计更清楚地反映出这种共同性。

生物学家看到这个层次图时可能会不太满意，要知道，企鹅和海豚实际上不属于同一科。不过，由此可以很好地反映出，在编写代码时，对于真实世界中的关系和共享的功能关系，需要做适当的平衡。尽管两个东西在真实世界中没有太大的关联，但是在代码中它们之间可能存在着一种 not-a 关系，因为它们没有共享功能。也可以简单地把动物划分为哺乳动物和鱼类，但是这样一来，怎么在它们的超类中建

立其共同性呢？

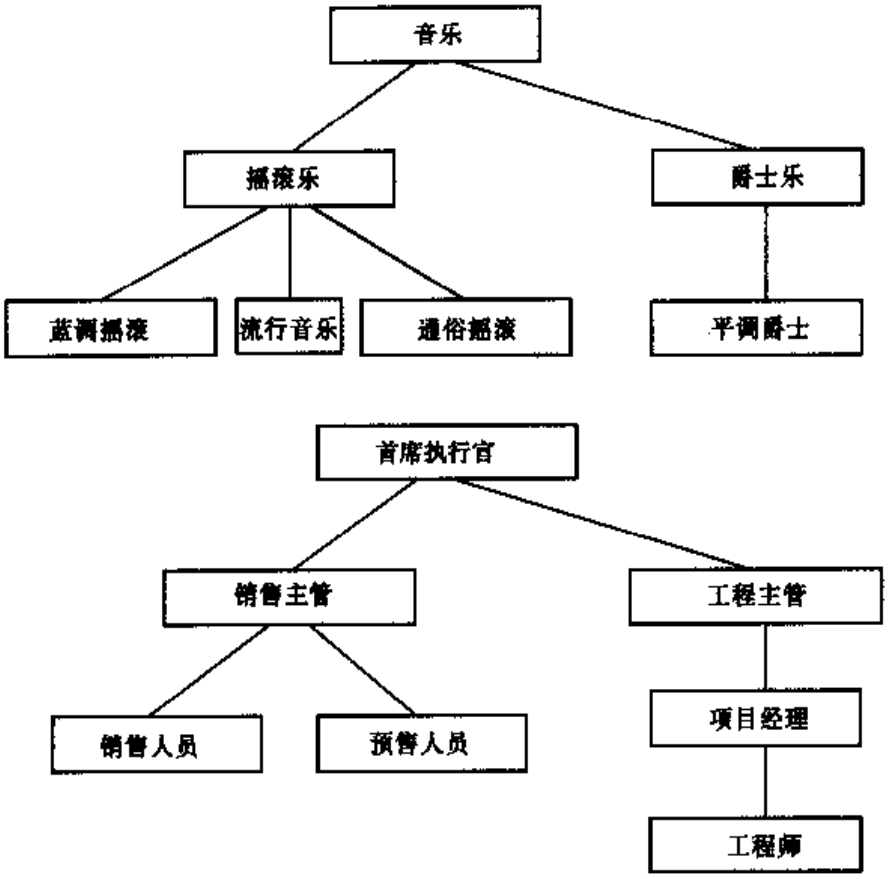

图 3-7

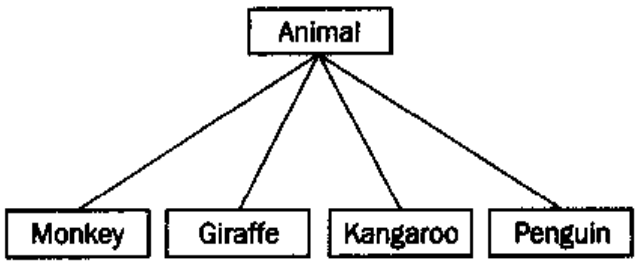

图 3-8

还有一点很重要，对于层次体系的组织可能还有其他的方法。前面的设计主要是按动物如何移动来组织。如果按动物的食物或高度来组织，所得到的层次体系就会又是一个样子。最后需要指出，如何使用类才是关键。这些需求可以指示出对象层次体系的设计。

好的面向对象层次体系要满足以下要求：

- 将类按有意义的功能关系加以组织。
- 将共同的功能放到超类中，从而支持代码重用。
- 避免子类过多地覆盖父类的功能。

多重继承

到目前为止，每个例子都只有一条继承链。换句话说，一个给定的类至多有一个直接父类。并不一

定必须如此。通过多重继承，类可以有多个超类（译者注：更确切的说，多重继承是指类可以有多个直接父类）。

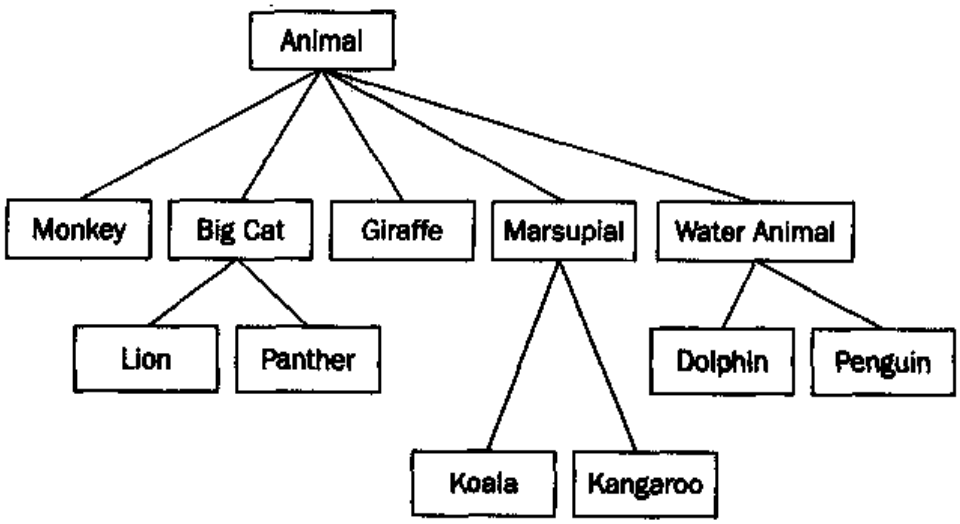

图　3-9

> 如果你认为动物在太多方面存在着差异，那么任何一个动物对象层次体系似乎都不太好，此时你可能就需要多重继承。利用多重继承，可以创建三个单独的层次体系，一个按大小组织，一个按食物组织，还有一个按移动方式组织。这样，每个动物都能选择其中的一个体系结构。

图 3-10 显示了一个多重继承设计。这里仍然有一个名为 Animal 的超类，它进一步按大小进行了划分。另外根据动物的食物建立了一个单独的层次体系，第三个层次体系则按动物的移动方式组织。这样，每一类动物都是这三个类（Mover、Eater 和 Animal）的子类，对此用不同颜色的线显示。

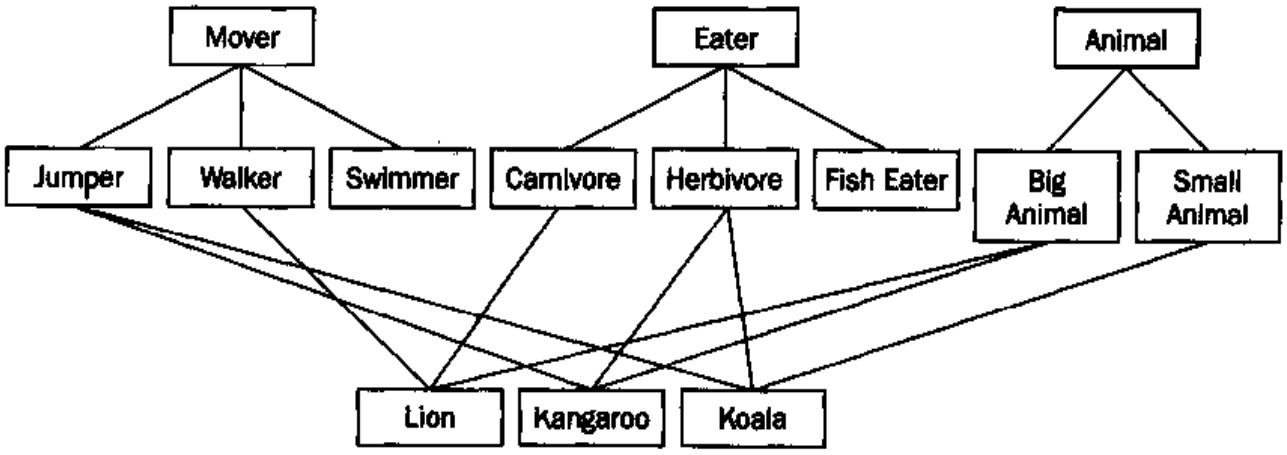

图　3-10

假设在用户界面中有一个图像，用户可以单击这个图像。这个对象看上去既是一个按钮，又是一个图像，因此其实现可以既派生 Image 类，又派生 Button 类，如图 3-11 所示。

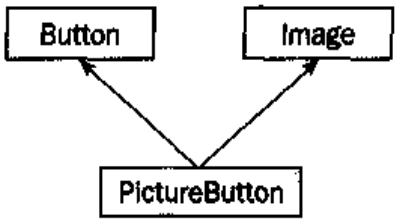

图 3-11

多重继承有什么坏处

许多程序员都不喜欢多重继承。C++对这种关系提供了显式支持，但是Java语言去掉了这个特性。对多重继承存在的批评，原因是多方面的。

首先，多重继承的可视化表示很复杂。如图3-10所示，即使是这样一个简单的类图，由于存在多个层次体系和交错的线，它也会变得非常复杂。建立类层次就是为了使程序员更容易理解代码之间的关系。基于多重继承，一个类可以有多个父类，这些父类之间并无关联。由于你的对象中有这么多类的“贡献”，你真的能搞清楚到底发生了什么吗？在真实世界中，我们往往不会认为对象有多个is-a关系。

其次，本来应该很清晰的层次结构可能会因为多重继承而变得混乱不堪。在动物例子中，如果转而采用一种多重继承方法，这说明Animal超类已经没多少意义了，因为描述动物的代码现在已经分别放到了三个单独的层次体系中。尽管图3-10中所示的设计显示了三个清晰的层次体系，但不难想像它们将成为怎样的一团乱麻。例如，如果发现所有Jumper不光都跳着走，而且吃的东西也一样，会怎么样呢？因为这三个层次体系都是独立的，我们只能再增加另一个子类，除此以外，再没有别的办法可以结合移动方式和食物这两个概念了。

第三，多重继承的实现很复杂。如果两个超类以不同方式实现了同一个行为会怎么样呢？能不能让这两个超类本身又作为另一个共同超类的子类呢？由于存在这样一些可能性，就会导致实现很复杂，因为在代码中建立这么错综复杂的关系不光对编写代码的人来说是个难题，对于阅读代码的人来说也同样很费劲。

其他一些语言可能去除了多重继承，其原因是：通常多重继承是可以避免的。通过重新考虑层次体系，或者使用第26章介绍的某些设计模式，不必引入多重继承就能准确地把握项目的设计。

混合类

混合（mix-in）类表示了类之间的另外一种关系。在C++中，混合类的实现语法与多重继承很像，不过其语义则截然不同！混合类回答了这样一个问题“这个类还能做什么？”，而且答案中往往是“能够……”。基于混合类，可以为一个类增加功能，而不必建立一个完整的is-a关系。

再来看动物园的例子，你可能想表达这么一种概念，有些动物“可以逗着玩”。也就是说，来动物园的游客可以和这样一些动物嬉戏，却不会被咬伤或打伤。你可能希望所有可以逗着玩的动物都支持一个行为“逗着玩”。由于可以逗着玩的动物并没有其他共同之处，所以你不想打破原先设计的层次体系，这么看来，Pettable就完全可以作为一个混合类。

混合类通常用在用户界面中。我们一般不会说一个PictureButton类同时是一个Image和一个Button，而会说这是一个可以单击的（Clickable）Image。桌面上的一个文件夹则是一个可拖放（Draggable）的Image。在软件开发中，我们造出了许多这种有意思的形容词（译者注：这是指Clickable、Draggable等等表示“可以做什么”的词汇，在英语中本来并没有这样的单词）。

混合类和超类之间存在着区别，这种区别并不是代码上的差别，更反映在你将如何考虑类。一般来说，混合类比多重继承更易于理解，因为混合类往往限制在某个范围内。Pettable混合类只是为所有既有

的类增加一个行为而已。Clickable 混合类可能只是增加了“鼠标按下”和“鼠标放开”行为。另外，混合类一般没有太大的层次体系，所以不存在功能的相互交错。

3.1.5　抽象

在第 2 章中，你已经了解了抽象的概念，这是指将具体实现与对实现的访问方法相分离。抽象是一个很好的思想，前面已经分析过为什么抽象很有意义。这也是面向对象设计的一个基本部分。

接口和实现

抽象的关键就是将接口（interface）与实现（implementation）有效地分离。所谓实现，是指为完成所需完成的任务而编写的代码。接口则是指其他人用什么方法来使用你的代码。在 C 中，如果头文件描述了你所写的库中的函数，这个头文件就是一个接口。在面向对象程序设计中，类的接口是一个由可公共访问的属性和行为所组成的集合。

确定对外的接口

在设计一个类时，会出现这样一个问题，即其他程序员如何与你的对象交互。在 C++ 中，类的各个属性和行为可以限定为 public、protected 或 private。public 是指其他代码可以访问这个属性或行为。protected 表示其他代码不能访问此属性或行为（译者注：但子类可以访问父类的 protected 属性或行为）。private 则是一个更严格的控制，它表示不仅一般的其他代码不能访问这些属性或行为，另外，即使是子类也不允许访问。

设计对外的接口实际上就是选择哪些属性和行为要置为 public。与其他程序员共同完成一个大型项目时，应当把设计对外接口看作是开发中的一个重要过程。

考虑受众

设计对外接口的第一步就是考虑你是为谁设计这个接口。使用这个接口的人（受众）是开发小组中的一个成员吗？你是不是只打算自己使用这个接口？这个接口会不会被你公司之外的某个程序员使用呢？会是一个顾客，还是一个下级承包人？这一步除了可以确定谁将利用这个接口，还会明确你的一些设计目标。

如果只是你自己使用这个接口，那么设计时就可能有更大的自由度。你在利用这个接口时，可以对其进行修改来满足自己的需求。不过，一定要谨记工程小组中的角色可能会发生变动，而且总有一天很可能其他人也会使用这个接口。

为其他内部程序员（即小组中的其他成员）设计接口则稍有不同。从某种意义上说，接口也就是你与这些程序员之间的一个合约。例如，如果你实现了程序的数据存储库组件，其他人要依赖于这个接口来支持某种操作。就需要明确小组中的其他人要使用你的类做些什么。他们需要版本控制吗？他们能存储哪些类型的数据？作为一个合约，你应当把接口看作是一种不太灵活的东西（译者注：即不太会改变）。如果在编写代码之前协商好了接口，而在代码写完之后你又决定修改接口，那么其他程序员肯定会对你提出抱怨。

如果客户是一个外部顾客，你就要基于另外一组迥然不同的需求开始设计。理想情况下，可能会要求目标顾客指定你的接口要提供哪些功能。你不仅要考虑他们希望得到的特定功能，还要考虑他们将来可能希望得到哪些功能。接口中的用词应当与顾客熟悉的词相对应，而且在写文档时必须从看文档人的角度来考虑。在设计中，不应当出现玩笑话、代码名和程序员的行话。

考虑用途

编写接口的原因有很多。在实际编写代码前，甚至在确定要提供什么功能之前，需要理解接口的用途。

应用编程接口（API）

应用编程接口（Application Programming Interface，API）是一种外部可见的机制，利用API可以在另一个上下文中扩展一个产品，或者使用该产品的功能。如果说内部接口是一个合约，那么API就相当于铁定的法律。一旦有你公司之外的人使用你的API，他们肯定不希望API出现改动，除非你增加了新的功能能够对其提供帮助。由此说来，在让顾客使用你的API之前，一定要特别小心地规划API，并与顾客讨论交流。

在设计API时，主要是要权衡API的易用性和效率。由于使用接口的目标用户并不熟悉产品的内部工作原理，如何使用API的学习曲线应当是逐渐上升的（即不应过陡）。毕竟，你的公司之所以向顾客提供这个API，就是为了让顾客能够使用它。如果太难于使用，这个API就要算是一个失败。通常灵活性会与此背道而驰。你的产品可能有许多不同的用途，而且你希望顾客能够充分利用你所提供的所有功能。不过，你的产品也许能做很多工作，如果想让顾客都能用到的话，这样的API就会太过复杂了。

正如一则编程箴言所说的："好的API可以让容易依然容易，让困难的事情也变为可能。"也就是说，API应该有一个简单的学习曲线。通过API，大多数程序员想做的工作应该都能够做到。不过，API还支持更高级的使用，尽管一般情况下都要求API具有简单性，但是在极少情况下，出于某种折衷考虑，复杂的API也是可以接受的。

工具类或库

通常，你的任务就是开发某个特定的功能，以便在应用中别的地方加以使用。这可能是一个随机数库或者是一个日志记录类。在这些情况下，接口是比较容易确定的，因为你要提供大多数或者所有的功能，而最好不暴露有关实现的太多内容。通用性是一个需要考虑的重要问题。由于这个类或库是通用的，必须在设计时就考虑到一组可能的用例。

子系统接口

你可能要设计应用中两个主要子系统之间的接口，如访问数据库的机制。在这些情况下，将接口与实现相分离是至关重要的，这是因为，在实现完成之前，其他程序员很可能已经基于你的接口开始完成他们的实现了。在开发子系统时，首先要考虑这个子系统的一个主要用途是什么。一旦明确了子系统所要完成的主要任务，再来考虑它的特殊用途，以及应当如何提供给代码中的其他部分。要从总体角度考虑，而不要过分深陷于实现细节当中。

组件接口

你定义的大多数接口都可能比子系统接口或API要小。这些对象会在你所写的其他代码中使用。在这些情况下，主要存在这样一个陷阱，接口可能会不断扩大，以至于最后发展到无法控制的地步。尽管这些接口只是你自己使用，但是不要这样考虑这些接口，而要认为这些接口还可能为别人所用。与子系统接口一样，需要考虑每个类的一个主要用途，对于与这个用途无关的功能，对外提供这样一些功能时要务必谨慎。

为将来做考虑

在设计接口时，要考虑到将来的情况。是不是今后几年你都要采用这个设计？倘若如此，就可能需要提供一个插件架构来留出扩展的空间。你是否发现别人可能将你的接口另作他用（即并非原先所设计的用途）？如果是这样，就要与他们交流，对他们的用例有更充分的认识。还有一种做法是以后再重新编写接口，或者更糟糕的，发现需要新功能时就随时加上，根本没有任何计划，这样最后就会得到一个杂乱无章的接口。不过千万注意，如果过分强调通用性，则会落入另一个陷阱。如果不清楚将来会做何使用，就不要设计一个"万能"的日志记录类。

设计一个成功的抽象

要设计好的抽象，经验和反复是关键。只有经过多年编写和使用抽象的历练，才能设计出真正的好接口。你遇到其他抽象的时候，一定要记住其中哪些可行，哪些不可行。你上周使用 Windows 文件系统 API 时发现其中缺少了什么？如果是自己写网络包装器，与你的同事编写的相比，你会做哪些不同处理？一般来说，最早在纸上设计的接口往往不是最好的接口，所以一定要反复调整。让相关的人了解你的设计，并得到他们的反馈。也许你已经开始编写代码了，但也不要因此害怕修改抽象，即使这可能意味着其他程序员要做相应调整。他们应该能意识到，从长远来看，好的设计对每个人都有利。

与其他程序员交流你的（新）设计时，有时需要稍做一些解释。开发小组的其他人可能并未发现前一个设计（即修改前的设计）有问题，也有可能认为采用你的新方法的话，他们要做太多的工作。在这种情况下，一方面要坚持你的工作，另一方面还要在适当的地方采纳他们的思想。如果他们还存在异议，可以提供一个清楚的文档和一个示例代码，这往往就能把他们“征服”。

要当心单类（single-class）抽象。如果你写的代码层次过深，就要考虑可以另外建立哪些辅助类来“辅佐”主接口。例如，如果你提供了一个接口来完成一些数据处理，还可以考虑编写一个结果对象，由它提供一种简便方法来查看和解释结果。

在可能的情况下，应当将属性转变为行为。换句话说，不要让外部代码直接操纵类中的底层数据。一些粗心或草率的程序员可能会把一个“兔子”对象的高度设置成负数，你肯定不希望这样。所以正确的做法应当是，建立一个“设置高度”的行为，其中要完成必要的越界检查。

在此还要重申反复的意义，因为这是最重要的一点。要搜寻对设计的反馈，并做出反应，如果必要还要进行修改，并从错误中学习。

第 5 章介绍了设计接口和可重用代码的更多原则。

3.2　小结

在本章中，你对面向对象程序的设计有了一定的认识，却没有过多地涉及具体的代码。你所学到的概念几乎可以应用到任何一种面向对象语言中。其中一些概念对你来说可能只是复习，而另外一些则可能对你以前熟悉的概念做了全新的诠释，需要全新的理解。通过本章的学习，你也许会采用某些新方法来解决原有的问题，也可能会引用新的证据来支持你一直以来在开发小组中倡导的一些概念。即使在代码中没有用过对象，或者用得很少，但对于如何设计面向对象程序，你现在所掌握的可能比许多有经验的 C++程序员还要多。

学习对象之间的关系非常重要，这是因为对象之间如果有清晰的关系，将有利于代码复用，并能减少混乱，还不仅如此，你会在一个团队中工作，这也是一个原因。如果对象以合理的方式关联，这样的对象将更易于阅读和维护。设计程序时，你可能会以 3.1.4 节作为参考。

最后，你还学习了如何创建成功的抽象，并了解了两个最重要的设计考虑——谁来使用此设计，以及设计的用途。第 4 章将展开介绍抽象的开发，包括诸如代码重用、思想重用等主题，以及可供使用的一些库。