中国互动出版网 www.china-pub.com | 网上书店 独家提供样章 HZ BOOKS

# 第1章 对象导论

“我们之所以将自然界分解，组织成各种概念，并按其含义分类，主要是因为我们是整个口语交流社会共同遵守的协定的参与者，这个协定以语言的形式固定下来……除非赞成这个协定中规定的有关语言信息的组织和分类，否则我们根本无法交谈。”

——Benjamin Lee Whorf (1897～1941)

计算机革命起源于机器，因此，编程语言的产生也始于对机器的模仿。

但是，计算机并非只是机器那么简单。计算机是头脑延伸的工具（就像Steve Jobs常喜欢说的“头脑的自行车”一样），同时还是一种不同类型的表达媒体。因此，这种工具看起来已经越来越不像机器，而更像我们头脑的一部分，以及一种如写作、绘画、雕刻、动画、电影等一样的表达形式。面向对象程序设计（Object-oriented Programming, OOP）便是这种以计算机作为表达媒体的大趋势中的组成部分。

本章将向读者介绍包括开发方法概述在内的OOP的基本概念。本章，乃至本书中，都假设读者已经具备了某些编程经验（当然不一定是C的）。如果读者认为在阅读本书之前还需要在程序设计方面多做些准备，那么就应该去研读可以从www.MindView.net网站上下载的《C编程思想》（Thinking in C）的多媒体资料。

本章介绍的是背景性的和补充性的材料。许多人在没有了解面向对象程序设计的全貌之前，感觉无法轻松自在地从事此类编程。因此，此处将引入许多概念，以期帮助读者扎实地了解OOP。然而，还有些人可能在看到具体结构之前，无法了解面向对象程序设计的全貌，这些人如果没有代码在手，就会陷于困境并最终迷失方向。如果你属于后面这个群体，并且渴望尽快获取Java语言的细节，那么可以先越过本章——在此处越过本章并不会妨碍你编写程序和学习语言。但是，你最终还是要回到本章来补充所学知识，这样才能够了解到对象的重要性，以及怎样使用对象进行设计。

23

## 1.1 抽象过程

所有编程语言都提供抽象机制。可以认为，人们所能够解决的问题的复杂性直接取决于抽象的类型和质量。所谓的“类型”是指“所抽象的是什么？”汇编语言是对底层机器的轻微抽象。接着出现的许多所谓“命令式”语言（如FORTRAN、BASIC、C等）都是对汇编语言的抽象。这些语言在汇编语言基础上有了大幅的改进，但是它们所作的主要抽象仍要求在解决问题时要基于计算机的结构，而不是基于所要解决的问题的结构来考虑。程序员必须建立起在机器模型（位于“解空间”内，这是你对问题建模的地方，例如计算机）和实际待解问题的模型（位于“问题空间”内，这是问题存在的地方，例如一项业务）之间的关联。建立这种映射是费力的，而且这不属于编程语言所固有的功能，这使得程序难以编写，并且维护代价高昂，同时也产生了作为副产物的整个“编程方法”行业。

另一种对机器建模的方式就是只针对待解问题建模。早期的编程语言，如LISP和APL，都选择考虑世界的某些特定视图（分别对应于“所有问题最终都是列表”或者“所有问题都是算法形式的”）。PROLOG则将所有问题都转换成决策链。此外还产生了基于约束条件编程的语言

中国互动出版网 www.china-pub.com | 网上书店 独家提供样章 HZ BOOKS



和专门通过对图形符号操作来实现编程的语言（后者被证明限制性过强）。这些方式对于它们所要解决的特定类型的问题都是不错的解决方案，但是一旦超出其特定领域，它们就力不从心了。

24

面向对象方式通过向程序员提供表示问题空间中的元素的工具而更进了一步。这种表示方式非常通用，使得程序员不会受限于任何特定类型的问题。我们将问题空间中的元素及其在解空间中的表示称为"对象"。（你还需要一些无法类比为问题空间元素的对象。）这种思想的实质是：程序可以通过添加新类型的对象使自身适用于某个特定问题。因此，当你在阅读描述解决方案的代码的同时，也是在阅读问题的表述。相比以前我们所使用的语言[⊖]，这是一种更灵活和更强有力的语言抽象。所以，OOP允许根据问题来描述问题，而不是根据运行解决方案的计算机来描述问题。但是它仍然与计算机有联系：每个对象看起来都有点像一台微型计算机——它具有状态，还具有操作，用户可以要求对象执行这些操作。如果要对现实世界中的对象作类比，那么说它们都具有特性和行为似乎不错。

Alan Kay曾经总结了第一个成功的面向对象语言、同时也是Java所基于的语言之一的Smalltalk的五个基本特性，这些特性表现了一种纯粹的面向对象程序设计方式：

1) 万物皆为对象。将对象视为奇特的变量，它可以存储数据，除此之外，你还可以要求它在自身上执行操作。理论上讲，你可以抽取待求解问题的任何概念化构件（狗、建筑物、服务等），将其表示为程序中的对象。

2) 程序是对象的集合，它们通过发送消息来告知彼此所要做的。要想请求一个对象，就必须对该对象发送一条消息。更具体地说，可以把消息想像为对某个特定对象的方法的调用请求。

25

3) 每个对象都有自己的由其他对象所构成的存储。换句话说，可以通过创建包含现有对象的包的方式来创建新类型的对象。因此，可以在程序中构建复杂的体系，同时将其复杂性隐藏在对象的简单性背后。

4) 每个对象都拥有其类型。按照通用的说法，"每个对象都是某个类（class）的一个实例（instance）"，这里"类"就是"类型"的同义词。每个类最重要的区别于其他类的特性就是"可以发送什么样的消息给它"。

5) 某一特定类型的所有对象都可以接收同样的消息。这是一句意味深长的表述，你在稍后便会看到。因为"圆形"类型的对象同时也是"几何形"类型的对象，所以一个"圆形"对象必定能够接受发送给"几何形"对象的消息。这意味着可以编写与"几何形"交互并自动处理所有与几何形性质相关的事物的代码。这种可替代性（substitutability）是OOP中最强有力的概念之一。

Booch 对对象提出了一个更加简洁的描述：对象具有状态、行为和标识。这意味着每一个对象都可以拥有内部数据（它们给出了该对象的状态）和方法（它们产生行为），并且每一个对象都可以唯一地与其他对象区分开来，具体说来，就是每一个对象在内存中都有一个唯一的地址[⊜]。

## 1.2 每个对象都有一个接口

亚里士多德大概是第一个深入研究类型（type）的哲学家，他曾提出过鱼类和鸟类这样的概念。所有的对象都是唯一的，但同时也是具有相同的特性和行为的对象所归属的类的一部分。

⊖ 某些编程语言的设计者认为面向对象编程本身不足以轻松地解决所有编程问题，所以他们提倡将不同的方式结合到多聚合编程语言（multipleparadigm programming language）中。读者可以查阅Timothy Budd的《Multipleparadigm Programming in Leda》一书（Addison-Wesley 1995）。

⊜ 这确实显得有一点过于受限了，因为对象可以存在于不同的机器和地址空间中，它们还可以被存储在硬盘上。在这些情况下，对象的标识就必须由内存地址之外的某些东西来确定了。

中国互动出版网 www.china-pub.com | 网上书店 独家提供样章 HZ BOOKS



这种思想被直接应用于第一个面向对象语言Simula-67，它在程序中使用基本关键字**class**来引入新的类型。

Simula，就像其名字一样，是为了开发诸如经典的“银行出纳员问题”（bank teller problem）这样的仿真程序而创建的。在银行出纳员问题中，有出纳、客户、账户、交易和货币单位等许多“对象”。在程序执行期间具有不同的状态而其他方面都相似的对象会被分组到对象的类中，这就是关键字**class**的由来。创建抽象数据类型（类）是面向对象程序设计的基本概念之一。抽象数据类型的运行方式与内置（built-in）类型几乎完全一致：你可以创建某一类型的变量（按照面向对象的说法，称其为对象或实例），然后操作这些变量（称其为发送消息或请求；发送消息，对象就知道要做什么）。每个类的成员或元素都具有某种共性：每个账户都有结余金额，每个出纳都可以处理存款请求等。同时，每个成员都有其自身的状态：每个账户都有不同的结余金额，每个出纳都有自己的姓名。因此，出纳、客户、账户、交易等都可以在计算机程序中被表示成唯一的实体。这些实体就是对象，每一个对象都属于定义了特性和行为的某个特定的类。

26

所以，尽管我们在面向对象程序设计中实际上进行的是创建新的数据类型，但事实上所有的面向对象程序设计语言都使用class这个关键词来表示数据类型。当看到类型一词时，可将其作为类来考虑，反之亦然⊖。

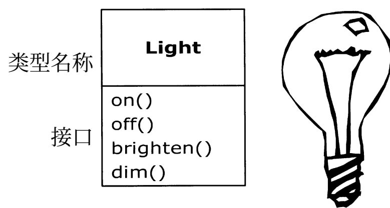
因为类描述了具有相同特性（数据元素）和行为（功能）的对象集合，所以一个类实际上就是一个数据类型，例如所有浮点型数字具有相同的特性和行为集合。二者的差异在于，程序员通过定义类来适应问题，而不再被迫只能使用现有的用来表示机器中的存储单元的数据类型。可以根据需求，通过添加新的数据类型来扩展编程语言。编程系统欣然接受新的类，并且像对待内置类型一样地照管它们和进行类型检查。

面向对象方法并不是仅局限于构建仿真程序。无论你是否赞成以下观点，即任何程序都是你所设计的系统的一种仿真，面向对象技术的应用确实可以将大量的问题很容易地降解为一个简单的解决方案。

一旦类被建立，就可以随心所欲地创建类的任意个对象，然后去操作它们，就像它们是存在于你的待求解问题中的元素一样。事实上，面向对象程序设计的挑战之一，就是在问题空间的元素和解空间的对象之间创建一对一的映射。

27

但是，怎样才能获得有用的对象呢？必须有某种方式产生对对象的请求，使对象完成各种任务，如完成一笔交易、在屏幕上画图、打开开关等等。每个对象都只能满足某些请求，这些请求由对象的接口（interface）所定义，决定接口的便是类型。以电灯泡为例来做一个简单的比喻（如右图所示）：




```
Light lt = new Light();
lt.on();
```

接口确定了对某一特定对象所能发出的请求。但是，在程序中必须有满足这些请求的代码。这些代码与隐藏的数据一起构成了实现。从过程型编程的观点来看，这并不太复杂。在类型中，每一个可能的请求都有一个方法与之相关联，当向对象发送请求时，与之相关联的方法就会被调用。此过程通常被概括为：向某个对象“发送消息”（产生请求），这个对象便知道此消息的目的，然后执行对应的程序代码。

上例中，类型/类的名称是**Light**，特定的**Light**对象的名称是**lt**，可以向**Light**对象发出的请求是：打开它、关闭它、将它调亮、将它调暗。你以下列方式创建了一个Light对象：定义这个

⊖ 有些人对此会区别对待，他们认为：类型决定了接口，而类是该接口的一个特定实现。

中国互动出版网 www.china-pub.com | 网上书店 独家提供样章 HZ BOOKS



对象的“引用”(lt)，然后调用**new**方法来创建该类型的新对象。为了向对象发送消息，需要声明对象的名称，并以圆点符号连接一个消息请求。从预定义类的用户观点来看，这些差不多就是用对象来进行设计的全部。 28

前面的图是UML（Unified Modelling Language，统一建模语言）形式的图，每个类都用一个方框表示，类名在方框的顶部，你所关心的任何数据成员都描述在方框的中间部分，方法（隶属于此对象的、用来接收你发给此对象的消息的函数）在方框的底部。通常，只有类名和公共方法被示于UML设计图中，因此，方框的中部就像本例一样并未给出。如果只对类型感兴趣，那么方框的底部甚至也不需要给出。

## 1.3 每个对象都提供服务

当正在试图开发或理解一个程序设计时，最好的方法之一就是将对象想像为“服务提供者”。程序本身将向用户提供服务，它将通过调用其他对象提供的服务来实现这一目的。你的目标就是去创建（或者最好是在现有代码库中寻找）能够提供理想的服务来解决问题的一系列对象。

着手从事这件事的一种方式就是问一下自己：“如果我可以将问题从表象中抽取出来，那么什么样的对象可以马上解决我的问题呢？”例如，假设你正在创建一个簿记系统，那么可以想像，系统应该具有某些包括了预定义的簿记输入屏幕的对象，一个执行簿记计算的对象集合，以及一个处理在不同的打印机上打印支票和开发票的对象。也许上述对象中的某些已经存在了，但是对于那些并不存在的对象，它们看起来像什么样子？它们能够提供哪些服务？它们需要哪些对象才能履行它们的义务？如果持续这样做，那么最终你会说“那个对象看起来很简单，可以坐下来写代码了”，或者会说“我肯定那个对象已经存在了”。这是将问题分解为对象集合的一种合理方式。

将对象看作是服务提供者还有一个附带的好处：它有助于提高对象的内聚性。高内聚是软件设计的基本质量要求之一：这意味着一个软件构件（例如一个对象，当然它也有可能是指一个方法或一个对象库）的各个方面“组合”得很好。人们在设计对象时所面临的一个问题是，将过多的功能都塞在一个对象中。例如，在检查打印模式的模块中，你可以这样设计一个对象，让它了解所有的格式和打印技术。你可能会发现，这些功能对于一个对象来说太多了，你需要的是三个甚至更多个对象，其中，一个对象可以是所有可能的支票排版的目录，它可以被用来查询有关如何打印一张支票的信息；另一个对象（或对象集合）可以是一个通用的打印接口，它知道有关所有不同类型的打印机的信息（但是不包含任何有关簿记的内容，它更应该是一个需要购买而不是自己编写的对象）；第三个对象通过调用另外两个对象的服务来完成打印任务。这样，每个对象都有一个它所能提供服务的内聚的集合。在良好的面向对象设计中，每个对象都可以很好地完成一项任务，但是它并不试图做更多的事。就像在这里看到的，不仅允许通过购买获得某些对象（打印机接口对象），而且还可以创建能够在别处复用的新对象（支票排版目录对象）。 29

将对象作为服务提供者看待是一件伟大的简化工具，这不仅在设计过程中非常有用，而且当其他人试图理解你的代码或重用某个对象时，如果他们看出了这个对象所能提供的服务的价值，它会使调整对象以适应其设计的过程变得简单得多。

## 1.4 被隐藏的具体实现

将程序开发人员按照角色分为类创建者（那些创建新数据类型的程序员）和客户端程序员[⊖]

⊖ 关于这个术语的表述，我感谢我的朋友Scott Meyers。

中国互动出版网 www.china-pub.com | 网上书店 独家提供样章 HZ BOOKS



（那些在其应用中使用数据类型的类消费者）是大有裨益的。客户端程序员的目标是收集各种用来实现快速应用开发的类。类创建者的目标是构建类，这种类只向客户端程序员暴露必需的部分，而隐藏其他部分。为什么要这样呢？因为如果加以隐藏，那么客户端程序员将不能够访问它，这意味着类创建者可以任意修改被隐藏的部分，而不用担心对其他任何人造成影响。被隐藏的部分通常代表对象内部脆弱的部分，它们很容易被粗心的或不知内情的客户端程序员所毁坏，因此将实现隐藏起来可以减少程序bug。

在任何相互关系中，具有关系所涉及的各方都遵守的边界是十分重要的事情。当创建一个类库时，就建立了与客户端程序员之间的关系，他们同样也是程序员，但是他们是使用你的类库来构建应用、或者构建更大的类库的程序员。如果所有的类成员对任何人都是可用的，那么客户端程序员就可以对类做任何事情，而不受任何约束。即使你希望客户端程序员不要直接操作你的类中的某些成员，但是如果没有任何访问控制，将无法阻止此事发生。所有东西都将赤裸裸地暴露于世人面前。

30

因此，访问控制的第一个存在原因就是让客户端程序员无法触及他们不应该触及的部分——这些部分对数据类型的内部操作来说是必需的，但并不是用户解决特定问题所需的接口的一部分。这对客户端程序员来说其实是一项服务，因为他们可以很容易地看出哪些东西对他们来说很重要，而哪些东西可以忽略。

访问控制的第二个存在原因就是允许库设计者可以改变类内部的工作方式而不用担心会影响到客户端程序员。例如，你可能为了减轻开发任务而以某种简单的方式实现了某个特定类，但稍后发现你必须改写它才能使其运行得更快。如果接口和实现可以清晰地分离并得以保护，那么你就可以轻而易举地完成这项工作。

Java用三个关键字在类的内部设定边界：**public**、**private**、**protected**。这些访问指定词（access specifier）决定了紧跟其后被定义的东西可以被谁使用。**public**表示紧随其后的元素对任何人都是可用的，而**private**这个关键字表示除类型创建者和类型的内部方法之外的任何人都不能访问的元素。**private**就像你与客户端程序员之间的一堵砖墙，如果有人试图访问**private**成员，就会在编译时得到错误信息。**protected**关键字与**private**作用相当，差别仅在于继承的类可以访问**protected**成员，但是不能访问**private**成员。稍后将会对继承进行介绍。

Java还有一种默认的访问权限，当没有使用前面提到的任何访问指定词时，它将发挥作用。这种权限通常被称为包访问权限，因为在这种权限下，类可以访问在同一个包（库构件）中的其他类的成员，但是在包之外，这些成员如同指定了**private**一样。

31

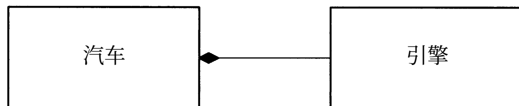
## 1.5 复用具体实现

一旦类被创建并被测试完，那么它就应该（在理想情况下）代表一个有用的代码单元。事实证明，这种复用性并不容易达到我们所希望的那种程度，产生一个可复用的对象设计需要丰富的经验和敏锐的洞察力。但是一旦你有了这样的设计，它就可供复用。代码复用是面向对象程序设计语言所提供的最了不起的优点之一。

最简单地复用某个类的方式就是直接使用该类的一个对象，此外也可以将那个类的一个对象置于某个新的类中。我们称其为“创建一个成员对象”。新的类可以由任意数量、任意类型的其他对象以任意可以实现新的类中想要的功能的方式所组成。因为是在使用现有的类合成新的类，所以这种概念被称为组合（composition），如果组合是动态发生的，那么它通常被称为聚合（aggregation）。组合经常被视为“has-a”（拥有）关系，就像我们常说的“汽车拥有引擎”一样。

中国互动出版网 www.china-pub.com | 网上书店 独家提供样章 HZ BOOKS






（上面这张UML图用实心菱形表明了组合关系。我通常采用最简单的形式：仅仅用一条没有菱形的线来表示关联⊖）。

组合带来了极大的灵活性。新类的成员对象通常都被声明为**private**，使得使用新类的客户端程序员不能访问它们。这也使得你可以在不干扰现有客户端代码的情况下，修改这些成员。也可以在运行时修改这些成员对象，以实现动态修改程序的行为。下面将要讨论的继承并不具备这样的灵活性，因为编译器必须对通过继承而创建的类施加编译时的限制。

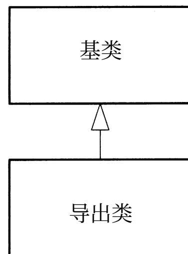
32

由于继承在面向对象程序设计中如此重要，所以它经常被高度强调，于是程序员新手就会有这样的印象：处处都应该使用继承。这会导致难以使用并过分复杂的设计。实际上，在建立新类时，应该首先考虑组合，因为它更加简单灵活。如果采用这种方式，设计会变得更加清晰。一旦有了一些经验之后，便能够看出必须使用继承的场合了。

## 1.6 继承

对象这种观念，本身就是十分方便的工具，使得你可以通过概念将数据和功能封装到一起，因此可以对问题空间的观念给出恰当的表示，而不用受制于必须使用底层机器语言。这些概念用关键字**class**来表示，它们形成了编程语言中的基本单位。

遗憾的是，这样做还是有很多麻烦：在创建了一个类之后，即使另一个新类与其具有相似的功能，你还是得重新创建一个新类。如果我们能够以现有的类为基础，复制它，然后通过添加和修改这个副本来创建新类那就要好多了。通过继承便可以达到这样的效果，不过也有例外，当源类（被称为基类、超类或父类）发生变动时，被修改的“副本”（被称为导出类、继承类或子类）也会反映出这些变动（如右图所示）。




（这张UML图中的箭头从导出类指向基类，就像稍后你会看到的，通常会存在一个以上的导出类。）

类型不仅仅只是描述了作用于一个对象集合上的约束条件，同时还有与其他类型之间的关系。两个类型可以有相同的特性和行为，但是其中一个类型可能比另一个含有更多的特性，并且可以处理更多的消息（或以不同的方式来处理消息）。继承使用基类型和导出类型的概念表示了这种类型之间的相似性。一个基类型包含其所有导出类型所共享的特性和行为。可以创建一个基类型来表示系统中某些对象的核心概念，从基类型中导出其他类型，来表示此核心可以被实现的各种不同方式。

33

以垃圾回收机为例，它用来归类散落的垃圾。“垃圾”是基类型，每一件垃圾都有重量、价值等特性，可以被切碎、熔化或分解。在此基础上，可以通过添加额外的特性（例如瓶子有颜色）或行为（例如铝罐可以被压碎，铁罐可以被磁化）导出更具体的垃圾类型。此外，某些行为可能不同（例如纸的价值取决于其类型和状态）。可以通过使用继承来构建一个类型层次结构，以此来表示待求解的某种类型的问题。

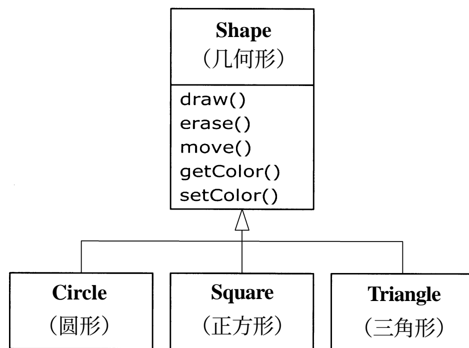
第二个例子是经典的几何形的例子，这在计算机辅助设计系统或游戏仿真系统中可能被用到。基类是几何形，每一个几何形都具有尺寸、颜色、位置等，同时每一个几何形都可以被绘制、擦

⊖ 通常对于大多数图来说，这样表示已经足够了，你并不需要关心所使用的是聚合还是组合。

中国互动出版网 www.china-pub.com | 网上书店 独家提供样章 HZ BOOKS



除、移动和着色等。在此基础上，可以导出（继承出）具体的几何形状——圆形、正方形、三角形等——每一种都具有额外的特性和行为，例如某些形状可以被翻转。某些行为可能并不相同，例如计算几何形状的面积。类型层次结构同时体现了几何形状之间的相似性和差异性（如右上图所示）。

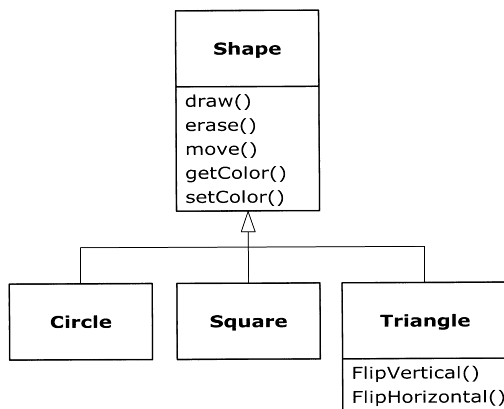



34

以同样的术语将解决方案转换成问题是大有裨益的，因为不需要在问题描述和解决方案描述之间建立许多中间模型。通过使用对象，类型层次结构成为了主要模型，因此，可以直接从真实世界中对系统的描述过渡到用代码对系统进行描述。事实上，对使用面向对象设计的人们来说，困难之一是从开始到结束过于简单。对于训练有素、善于寻找复杂的解决方案的头脑来说，可能会在一开始被这种简单性给难倒。

当继承现有类型时，也就创造了新的类型。这个新的类型不仅包括现有类型的所有成员（尽管**private**成员被隐藏了起来，并且不可访问），而且更重要的是它复制了基类的接口。也就是说，所有可以发送给基类对象的消息同时也可以发送给导出类对象。由于通过发送给类的消息的类型可知类的类型，所以这也就意味着导出类与基类具有相同的类型。在前面的例子中，“一个圆形也就是一个几何形”。通过继承而产生的类型等价性是理解面向对象程序设计方法内涵的重要门槛。

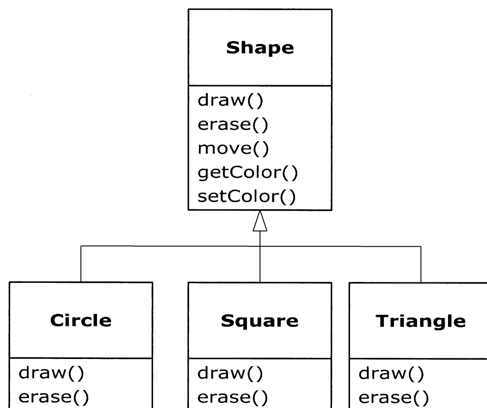
由于基类和导出类具有相同的基础接口，所以伴随此接口的必定有某些具体实现。也就是说，当对象接收到特定消息时，必须有某些代码去执行。如果只是简单地继承一个类而并不做其他任何事，那么在基类接口中的方法将会直接继承到导出类中。这意味着导出类的对象不仅与基类拥有相同的类型，而且还拥有相同的行为，这样做没有什么特别意义。




35

有两种方法可以使基类与导出类产生差异。第一种方法非常直接：直接在导出类中添加新方法。这些新方法并不是基类接口的一部分。这意味着基类不能直接满足你的所有需求，因此必需添加更多的方法。这种对继承简单而基本的使用方式，有时对问题来说确实是一种完美的解决方式。但是，应该仔细考虑是否存在基类也需要这些额外方法的可能性。这种设计的发现与迭代过程在面向对象程序设计中会经常发生（如右中图所示）。

虽然继承有时可能意味着在接口中添加新方法（尤其是在以**extends**关键字表示继承的Java中），但并非总需如此。第二种也是更重要的一种使导出类和基类之间产生差异的方法是改变现有基类的方法的行为，这被称之为覆盖（overriding）那个方法（如右下图所示）。




36

要想覆盖某个方法，可以直接在导出类中创

中国互动出版网 www.china-pub.com | 网上书店 独家提供样章 HZ BOOKS

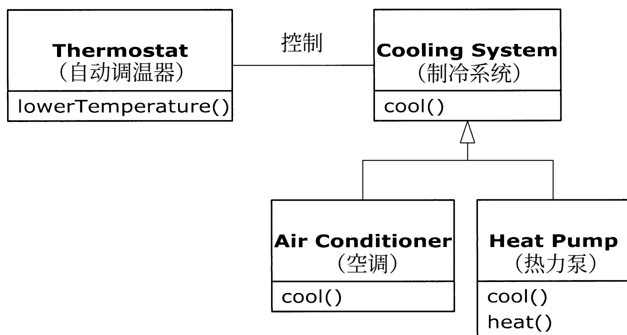


建该方法的新定义即可。你可以说 :“ 此时，我正在使用相同的接口方法，但是我想在新类型中做些不同的事情。”

### 1.6.1 “ 是一个 ” 与 “ 像是一个 ” 关系

对于继承可能会引发某种争论 ：继承应该只覆盖基类的方法（而并不添加在基类中没有的新方法）吗？如果这样做，就意味着导出类和基类是完全相同的类型，因为它们具有完全相同的接口。结果可以用一个导出类对象来完全替代一个基类对象。这可以被视为纯粹替代，通常称之为替代原则。在某种意义上，这是一种处理继承的理想方式。我们经常将这种情况下的基类与导出类之间的关系称为is-a（是一个）关系，因为可以说 “ 一个圆形就是一个几何形状 ”。判断是否继承，就是要确定是否可以用is-a来描述类之间的关系，并使之具有实际意义。

有时必须在导出类型中添加新的接口元素，这样也就扩展了接口。这个新的类型仍然可以替代基类，但是这种替代并不完美，因为基类无法访问新添加的方法。这种情况我们可以描述为is-like-a（像是一个）关系。新类型具有旧类型的接口，但是它还包含其他方法，所以不能说它们完全相同。以空调为例，假设房子里已经布线安装好了所有的冷气设备的控制器，也就是说，房子具备了让你控制冷气设备的接口。想像一下，如果空调坏了，你用一个既能制冷又能制热的热力泵替换了它，那么这个热力泵就is-like-a空调，但是它可以做更多的事。因为房子的控制系统被设计为只能控制冷气设备，所以它只能和新对象中的制冷部分进行通信。尽管新对象的接口已经被扩展了，但是现有系统除了原来接口之外，对其他东西一无所知。 37




当然，在看过这个设计之后，很显然会发现，制冷系统这个基类不够一般化，应该将其更名为 “ 温度控制系统 ”，使其可以包括制热功能，这样我们就可以套用替代原则了。这张图说明了在真实世界中进行设计时可能会发生的事情。

当你看到替代原则时，很容易会认为这种方式（纯粹替代）是唯一可行的方式，而且事实上，用这种方式设计是很好的。但是你会时常发现，同样显然的是你必须在导出类的接口中添加新方法。只要仔细审视，两种方法的使用场合应该是相当明显的。

## 1.7 伴随多态的可互换对象

在处理类型的层次结构时，经常想把一个对象不当作它所属的特定类型来对待，而是将其当作其基类的对象来对待。这使得人们可以编写出不依赖于特定类型的代码。在 “ 几何形 ” 的例子中，方法操作的都是泛化（generic）的形状，而不关心它们是圆形、正方形、三角形还是其他什么尚未定义的形状。所有的几何形状都可以被绘制、擦除和移动，所以这些方法都是直接对一个几何形对象发送消息 ；它们不用担心对象将如何处理消息。

38 这样的代码是不会受添加新类型影响的，而且添加新类型是扩展一个面向对象程序以便处

中国互动出版网 www.china-pub.com | 网上书店 独家提供样章 HZ BOOKS

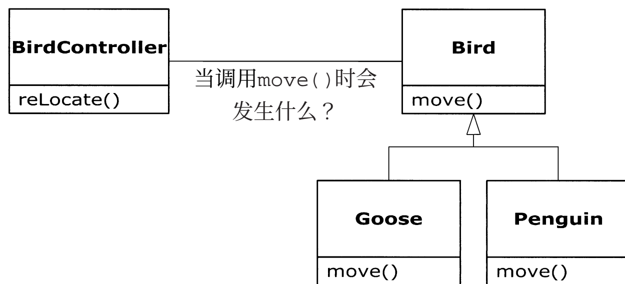


理新情况的最常用方式。例如，可以从“几何形”中导出一个新的子类型“五角形”，而并不需要修改处理泛化几何形状的方法。通过导出新的子类型而轻松扩展设计的能力是对改动进行封装的基本方式之一。这种能力可以极大地改善我们的设计，同时也降低软件维护的代价。

但是，在试图将导出类型的对象当作其泛化基类型对象来看待时（把圆形看作是几何形，把自行车看作是交通工具，把鸬鹚看作是鸟等等），仍然存在一个问题。如果某个方法要让泛化几何形状绘制自己、让泛化交通工具行驶，或者让泛化的鸟类移动，那么编译器在编译时是不可能知道应该执行哪一段代码的。这就是关键所在：当发送这样的消息时，程序员并不想知道哪一段代码将被执行；绘图方法可以被等同地应用于圆形、正方形、三角形，而对象会依据自身的具体类型来执行恰当的代码。

如果不需要知道哪一段代码会被执行，那么当添加新的子类型时，不需要更改调用它的方法，它就能够执行不同的代码。因此，编译器无法精确地了解哪一段代码将会被执行，那么它该怎么办呢？例如，在下面的图中，**BirdController**对象仅仅处理泛化的**Bird**对象，而不了解它们的确切类型。从**BirdController**的角度看，这么做非常方便，因为不需要编写特别的代码来判定要处理的**Bird**对象的确切类型或其行为。当**move()**方法被调用时，即便忽略**Bird**的具体类型，也会产生正确的行为（**Goose**（鹅）走、飞或游泳，**Penguin**（企鹅）走或游泳），那么，这是如何发生的呢？

39




这个问题的答案，也是面向对象程序设计的最重要的妙诀：编译器不可能产生传统意义上的函数调用。一个非面向对象编程的编译器产生的函数调用会引起所谓的前期绑定，这个术语你可能以前从未听说过，可能从未想过函数调用的其他方式。这么做意味着编译器将产生对一个具体函数名字的调用，而运行时将这个调用解析到将要被执行的代码的绝对地址。然而在OOP中，程序直到运行时才能够确定代码的地址，所以当消息发送到一个泛化对象时，必须采用其他的机制。

为了解决这个问题，面向对象程序设计语言使用了后期绑定的概念。当向对象发送消息时，被调用的代码直到运行时才能确定。编译器确保被调用方法的存在，并对调用参数和返回值执行类型检查（无法提供此类保证的语言被称为是弱类型的），但是并不知道将被执行的确切代码。

为了执行后期绑定，Java使用一小段特殊的代码来替代绝对地址调用。这段代码使用在对象中存储的信息来计算方法体的地址（这个过程将在第8章中详述）。这样，根据这一小段代码的内容，每一个对象都可以具有不同的行为表现。当向一个对象发送消息时，该对象就能够知道对这条消息应该做些什么。

在某些语言中，必须明确地声明希望某个方法具备后期绑定属性所带来的灵活性（C++是使用**virtual**关键字来实现的）。在这些语言中，方法在默认情况下不是动态绑定的。而在Java中，动态绑定是默认行为，不需要添加额外的关键字来实现多态。

40

中国互动出版网 www.china-pub.com | 网上书店 独家提供样章 HZ BOOKS



再来看看几何形状的例子。整个类族（其中所有的类都基于相同的一致接口）在本章前面已有图示。为了说明多态，我们要编写一段代码，它忽略类型的具体细节，仅仅和基类交互。这段代码和具体类型信息是分离的（decoupled），这样做使代码编写更为简单，也更易于理解。而且，如果通过继承机制添加一个新类型，例如**Hexagon**（六边形），所编写的代码对**Shape**（几何形）的新类型的处理与对已有类型的处理会同样出色。正因为如此，可以称这个程序是可扩展的。

如果用Java来编写一个方法（后面很快你就会学习如何编写）：

```
void doSomething(Shape shape) {
  shape.erase();
  // ...
  shape.draw();
}
```

这个方法可以与任何**Shape**对话，因此它是独立于任何它要绘制和擦除的对象的具体类型的。如果程序中其他部分用到了**doSomething()**方法：

```
Circle circle = new Circle();
Triangle triangle = new Triangle();
Line line = new Line();
doSomething(circle);
doSomething(triangle);
doSomething(line);
```

对**doSomething()**的调用会自动地正确处理，而不管对象的确切类型。

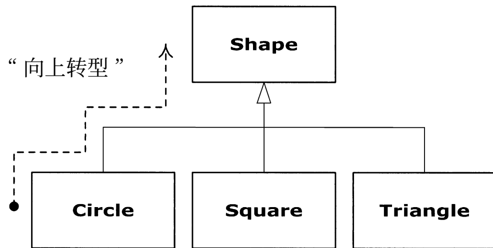
这是一个相当令人惊奇的诀窍。看看下面这行代码：

```
doSomething(circle);
```

当**Circle**被传入到预期接收**Shape**的方法中，究竟会发生什么。由于**Circle**可以被**doSomething()**看作是**Shape**，也就是说，**doSomething()**可以发送给**Shape**的任何消息，Circle都可以接收，那么，这么做是完全安全且合乎逻辑的。

41

把将导出类看做是它的基类的过程称为向上转型（upcasting）。转型（cast）这个名称的灵感来自于模型铸造的塑模动作；而向上（up）这个词来源于继承图的典型布局方式：通常基类在顶部，而导出类在其下部散开。因此，转型为一个基类就是在继承图中向上移动，即“向上转型”（如右图所示）。




一个面向对象程序肯定会在某处包含向上转型，因为这正是将自己从必须知道确切类型中解放出来的关键。让我们再看看**doSomething()**中的代码：

```
shape.erase();
// ...
shape.draw();
```

注意这些代码并不是说“如果是**Circle**，请这样做；如果是**Square**，请那样做……”。如果编写了那种检查**Shape**所有实际可能类型的代码，那么这段代码肯定是杂乱不堪的，而且在每次添加了Shape的新类型之后都要去修改这段代码。这里所要表达的意思仅仅是“你是一个**Shape**，我知道你可以**erase()**和**draw()**你自己，那么去做吧，但是要注意细节的正确性。”

**doSomething()**的代码给人印象深刻之处在于，不知何故，它总是做了该做的。调用**Circle**的**draw()**方法所执行的代码与调用**Square**或**Line**的**draw()**方法所执行的代码是不同的，而且当

中国互动出版网 www.china-pub.com | 网上书店 独家提供样章 HZ BOOKS



**draw()**消息被发送给一个匿名的**Shape**时，也会基于该**Shape**的实际类型产生正确的行为。这相当神奇，因为就像在前面提到的，当Java编译器在编译**doSomething()**的代码时，并不能确切知道**doSomething()**要处理的确切类型。所以通常会期望它的编译结果是调用基类**Shape**的**erase()**和**draw()**版本，而不是具体的**Circle**、**Square**或**Line**的相应版本。正是因为多态才使得事情总是能够被正确处理。编译器和运行系统会处理相关的细节，你需要马上知道的只是事情会发生，更重要的是怎样通过它来设计。当向一个对象发送消息时，即使涉及向上转型，该对象也知道要执行什么样的正确行为。 42

## 1.8 单根继承结构

在OOP中，自C++面世以来就已变得非常瞩目的一个问题就是，是否所有的类最终都继承自单一的基类。在Java中（事实上还包括除C++以外的所有OOP语言），答案是yes，这个终极基类的名字就是**Object**。事实证明，单根继承结构带来了很多好处。

在单根继承结构中的所有对象都具有一个共用接口，所以它们归根到底都是相同的基本类型。另一种（C++所提供的）结构是无法确保所有对象都属于同一个基本类型。从向后兼容的角度看，这么做能够更好地适应C模型，而且受限较少，但是当要进行完全的面向对象程序设计时，则必须构建自己的继承体系，使得它可以提供其他OOP语言内置的便利。并且在所获得的任何新类库中，总会用到一些不兼容的接口，需要花力气（有可能要通过多重继承）来使新接口融入你的设计之中。这么做来换取C++额外的灵活性是否值得呢？如果需要的话——如果在C上面投资巨大，这么做就很有价值。如果是刚刚从头开始，那么像Java这样的选择通常会有更高的生产率。

单根继承结构保证所有对象都具备某些功能。因此你知道，在你的系统中你可以在每个对象上执行某些基本操作。所有对象都可以很容易地在堆上创建，而参数传递也得到了极大的简化。

单根继承结构使垃圾回收器的实现变得容易得多，而垃圾回收器正是Java相对C++的重要改进之一。由于所有对象都保证具有其类型信息，因此不会因无法确定对象的类型而陷入僵局。这对于系统级操作（如异常处理）显得尤其重要，并且给编程带来了更大的灵活性。 43

## 1.9 容器

通常说来，如果不知道在解决某个特定问题时需要多少个对象，或者它们将存活多久，那么就不可能知道如何存储这些对象。如何才能知道需要多少空间来创建这些对象呢？答案是你不可能知道，因为这类信息只有在运行时才能获得。

对于面向对象设计中的大多数问题而言，这个问题的解决方案似乎过于轻率：创建另一种对象类型。这种新的对象类型持有对其他对象的引用。当然，你可以用在大多数语言中都有的数组类型来实现相同的功能。但是这个通常被称为容器（也称为集合，不过Java类库以不同的含义使用“集合”这个术语，所以本书将使用“容器”这个词）的新对象，在任何需要时都可扩充自己以容纳你置于其中的所有东西。因此不需要知道将来会把多少个对象置于容器中，只需要创建一个容器对象，然后让它处理所有细节。

幸运的是，好的OOP语言都有一组容器，它们作为开发包的一部分。在C++中，容器是标准C++类库的一部分，经常被称为标准模板类库（Standard Template Library, STL）。Object Pascal在其可视化构件库（Visual Component Library, VCL）中有容器；Smalltalk提供了一个非常完备的容器集；Java在其标准类库中也包含有大量的容器。在某些类库中，一两个通用容器足够满足所有的需要；但是在其他类库（例如Java）中，具有满足不同需要的各种类型的容器，例如

中国互动出版网 www.china-pub.com | 网上书店 独家提供样章 HZ BOOKS



List（用于存储序列），Map（也被称为关联数组，用来建立对象之间的关联），Set（每种对象类型只持有一个），以及诸如队列、树、堆栈等更多的构件。

从设计的观点来看，真正需要的只是一个可以被操作，从而解决问题的序列。如果单一类型的容器可以满足所有需要，那么就没有理由设计不同种类的序列了。然而还是需要对容器有所选择，这有两个原因。第一，不同容器提供了不同类型的接口和外部行为。堆栈相比于队列就具备不同的接口和行为，也不同于集合和列表的接口和行为。它们之中的某种容器提供的解决方案可能比其他容器要灵活得多。第二，不同的容器对于某些操作具有不同的效率。最好的例子就是两种List的比较：ArrayList和LinkedList。它们都是具有相同接口和外部行为的简单的序列，但是它们对某些操作所花费的代价却有天壤之别。在ArrayList中，随机访问元素是一个花费固定时间的操作；但是，对LinkedList来说，随机选取元素需要在列表中移动，这种代价是高昂的，访问越靠近表尾的元素，花费的时间越长。而另一方面，如果想在序列中间插入一个元素，LinkedList的开销却比ArrayList要小。上述操作以及其他操作的效率，依序列底层结构的不同而存在很大的差异。我们可以在一开始使用LinkedList构建程序，而在优化系统性能时改用ArrayList。接口List所带来的抽象，把在容器之间进行转换时对代码产生的影响降到最小限度。

44

### 1.9.1 参数化类型

在Java SE5出现之前，容器存储的对象都只具有Java中的通用类型：Object。单根继承结构意味着所有东西都是Object类型，所以可以存储Object的容器可以存储任何东西[⊖]。这使得容器很容易被复用。

要使用这样的容器，只需在其中置入对象引用，稍后还可以将它们取回。但是由于容器只存储Object，所以当将对象引用置入容器时，它必须被向上转型为Object，因此它会丢失其身份。当把它取回时，就获取了一个对Object对象的引用，而不是对置入时的那个类型的对象的引用。所以，怎样才能将它变回先前置入容器中时的具有实用接口的对象呢？

这里再度用到了转型，但这一次不是向继承结构的上层转型为一个更泛化的类型，而是向下转型为更具体的类型。这种转型的方式称为向下转型。我们知道，向上转型是安全的，例如Circle是一种Shape类型；但是不知道某个Object是Circle还是Shape，所以除非确切知道所要处理的对象的类型，否则向下转型几乎是不安全的。

然而向下转型并非彻底是危险的，因为如果向下转型为错误的类型，就会得到被称为异常的运行时错误，稍后会介绍什么是异常。尽管如此，当从容器中取出对象引用时，还是必须要以某种方式记住这些对象究竟是什么类型，这样才能执行正确的向下转型。

45

向下转型和运行时的检查需要额外的程序运行时间，也需要程序员付出更多的心血。那么创建这样的容器，它知道自己所保存的对象的类型，从而不需要向下转型以及消除犯错误的可能，这样不是更有意义吗？这种解决方案被称为参数化类型机制。参数化类型就是一个编译器可以自动定制作用于特定类型上的类。例如，通过使用参数化类型，编译器可以定制一个只接纳和取出Shape对象的容器。

Java SE5的重大变化之一就是增加了参数化类型，在Java中它称为范型。一对尖括号，中间包含类型信息，通过这些特征就可以识别对范型的使用。例如，可以用下面这样的语句来创建一个存储Shape的ArrayList：

```
ArrayList<Shape> shapes = new ArrayList<Shape>();
```

⊖ 它们不能持有基本类型，但是Java SE5的自动包装功能使得这项限制几乎不成什么问题了。有关这一点将在本书后面的章节中进行详细讨论。

中国互动出版网 www.china-pub.com | 网上书店 独家提供样章 HZ BOOKS



为了利用范型的优点，很多标准类库构件都已经进行了修改。就像我们将要看到的那样，范型对本书中的许多代码都产生了重要的影响。

## 1.10 对象的创建和生命期

在使用对象时，最关键的问题之一便是它们的生成和销毁方式。每个对象为了生存都需要资源，尤其是内存。当我们不再需要一个对象时，它必须被清理掉，使其占有的资源可以被释放和重用。在相对简单的编程情况下，怎样清理对象看起来似乎不是什么挑战：你创建了对象，根据需要使用它，然后它应该被销毁。然而，你很可能会遇到相对复杂的情况。

例如，假设你正在为某个机场设计空中交通管理系统（同样的模型在仓库货柜管理系统、录像带出租系统或宠物寄宿店也适用）。一开始问题似乎很简单：创建一个容器来保存所有的飞机，然后为每一架进入空中交通控制区域的飞机创建一个新的飞机对象，并将其置于容器中。对于清理工作，只需在飞机离开此区域时删除相关的飞机对象即可。

46

但是，可能还有别的系统记录着有关飞机的数据，也许这些数据不需要像主要控制功能那样立即引人注意。例如，它可能记录着所有飞离机场的小型飞机的飞行计划。因此你需要有第二个容器来存放小型飞机；无论何时，只要创建的是小型飞机对象，那么它同时也应该置入第二个容器内。然后某个后台进程在空闲时对第二个容器内的对象进行操作。

现在问题变得更困难了：怎样才能知道何时销毁这些对象呢？当处理完某个对象之后，系统某个其他部分可能还在处理它。在其他许多场合中也会遇到同样的问题，在必须明确删除对象的编程系统中（例如C++），此问题会变得十分复杂。

对象的数据位于何处？怎样控制对象的生命周期？C++认为效率控制是最重要的议题，所以给程序员提供了选择的权力。为了追求最大的执行速度，对象的存储空间和生命周期可以在编写程序时确定，这可以通过将对象置于堆栈（它们有时被称为自动变量（automatic variable）或限域变量（scoped variable））或静态存储区域内来实现。这种方式将存储空间分配和释放置于优先考虑的位置，某些情况下这样控制非常有价值。但是，也牺牲了灵活性，因为必须在编写程序时知道对象确切的数量、生命周期和类型。如果试图解决更一般化的问题，例如计算机辅助设计、仓库管理或者空中交通控制，这种方式就显得过于受限了。

第二种方式是在被称为堆（heap）的内存池中动态地创建对象。在这种方式中，直到运行时才知道需要多少对象，它们的生命周期如何，以及它们的具体类型是什么。这些问题的答案只能在程序运行时相关代码被执行到的那一刻才能确定。如果需要一个新对象，可以在需要的时刻直接在堆中创建。因为存储空间是在运行时被动态管理的，所以需要大量的时间在堆中分配存储空间，这可能要远远大于在堆栈中创建存储空间的时间。在堆栈中创建存储空间和释放存储空间通常各需要一条汇编指令即可，分别对应将栈顶指针向下移动和将栈顶指针向上移动。创建堆存储空间的时间依赖于存储机制的设计。

47

动态方式有这样一个一般性的逻辑假设：对象趋向于变得复杂，所以查找和释放存储空间的开销不会对对象的创建造成重大冲击。动态方式所带来的更大的灵活性正是解决一般化编程问题的要点所在。

Java完全采用了动态内存分配方式[⊖]。每当想要创建新对象时，就要使用new关键字来构建此对象的动态实例。

还有一个议题，就是对象生命周期。对于允许在堆栈上创建对象的语言，编译器可以确定

⊖ 稍候你将看到，基本类型只是一种特例。

中国互动出版网 www.china-pub.com | 网上书店 独家提供样章 HZ BOOKS



对象存活的时间，并可以自动销毁它。然而，如果是在堆上创建对象，编译器就会对它的生命周期一无所知。在像C++这样的语言中，必须通过编程方式来确定何时销毁对象，这可能会因为不能正确处理而导致内存泄漏（这在C++程序中是常见的问题）。Java提供了被称为“垃圾回收器”的机制，它可以自动发现对象何时不再被使用，并继而销毁它。垃圾回收器非常有用，因为它减少了所必须考虑的议题和必须编写的代码。更重要的是，垃圾回收器提供了更高层的保障，可以避免暗藏的内存泄漏问题，这个问题已经使许多C++项目折戟沉沙。

Java的垃圾回收器被设计用来处理内存释放问题（尽管它不包括清理对象的其他方面）。垃圾回收器“知道”对象何时不再被使用，并自动释放对象占用的内存。这一点同所有对象都是继承自单根基类Object以及只能以一种方式创建对象（在堆上创建）这两个特性结合起来，使得用Java编程的过程较之用C++编程要简单得多，所要做出的决策和要克服的障碍也要少得多。 48

## 1.11 异常处理：处理错误

自从编程语言问世以来，错误处理就始终是最困难的问题之一。因为设计一个良好的错误处理机制非常困难，所以许多语言直接略去这个问题，将其交给程序库设计者处理，而这些设计者也只是提出了一些不彻底的方法，这些方法可用于许多很容易就可以绕过此问题的场合，而且其解决方式通常也只是忽略此问题。大多数错误处理机制的主要问题在于，它们都依赖于程序员自身的警惕性，这种警惕性来源于一种共同的约定，而不是编程语言所强制的。如果程序员不够警惕——通常是因为他们太忙，这些机制就很容易被忽视。

异常处理将错误处理直接置于编程语言中，有时甚至置于操作系统中。异常是一种对象，它从出错地点被“抛出”，并被专门设计用来处理特定类型错误的相应的异常处理器“捕获”。异常处理就像是与程序正常执行路径并行的、在错误发生时执行的另一条路径。因为它是另一条完全分离的执行路径，所以它不会干扰正常的执行代码。这往往使得代码编写变得简单，因为不需要被迫定期检查错误。此外，被抛出的异常不像方法返回的错误值和方法设置的用来表示错误条件的标志位那样可以被忽略。异常不能被忽略，所以它保证一定会在某处得到处理。最后需要指出的是：异常提供了一种从错误状况进行可靠恢复的途径。现在不再是只能退出程序，你可以经常进行校正，并恢复程序的执行，这些都有助于编写出更健壮的程序。

Java的异常处理在众多的编程语言中格外引人注目，因为Java一开始就内置了异常处理，而且强制你必须使用它。它是唯一可接受的错误报告方式。如果没有编写正确的处理异常的代码，那么就会得到一条编译时的出错消息。这种有保障的一致性有时会使得错误处理非常容易。

值得注意的是，异常处理不是面向对象的特征——尽管在面向对象语言中异常常被表示成为一个对象。异常处理在面向对象语言出现之前就已经存在了。 49

## 1.12 并发编程

在计算机编程中有一个基本概念，就是在同一时刻处理多个任务的思想。许多程序设计问题都要求，程序能够停下正在做的工作，转而处理某个其他问题，然后再返回主进程。有许多方法可以实现这个目的。最初，程序员们用所掌握的有关机器底层的知识来编写中断服务程序，主进程的挂起是通过硬件中断来触发的。尽管这么做可以解决问题，但是其难度太大，而且不能移植，所以使得将程序移植到新型号的机器上时，既费时又费力。

有时中断对于处理时间性强的任务是必需的，但是对于大量的其他问题，我们只是想把问题切分成多个可独立运行的部分（任务），从而提高程序的响应能力。在程序中，这些彼此独立运行的部分称之为线程，上述概念被称为“并发”。并发最常见的例子就是用户界面。通过使用

中国互动出版网 www.china-pub.com | 网上书店 独家提供样章 HZ BOOKS



任务，用户可以在撤下按钮后快速得到一个响应，而不用被迫等待到程序完成当前任务为止。

通常，线程只是一种为单一处理器分配执行时间的手段。但是如果操作系统支持多处理器，那么每个任务都可以被指派给不同的处理器，并且它们是在真正地并行执行。在语言级别上，多线程所带来的便利之一便是程序员不用再操心机器上是有多个处理器还是只有一个处理器。由于程序在逻辑上被分为线程，所以如果机器拥有多个处理器，那么程序不需要特殊调整也能执行得更快。

所有这些都使得并发看起来相当简单，但是有一个隐患：共享资源。如果有多个并行任务都要访问同一项资源，那么就会出问题。例如，两个进程不能同时向一台打印机发送信息。为了解决这个问题，可以共享的资源，例如打印机，必须在使用期间被锁定。因此，整个过程是：某个任务锁定某项资源，完成其任务，然后释放资源锁，使其他任务可以使用这项资源。

Java的并发是内置于语言中的，Java SE5已经增添了大量额外的库支持。 50

## 1.13 Java与Internet

如果Java仅仅只是众多的程序设计语言中的一种，你可能就会问：为什么它如此重要？为什么它促使计算机编程语言向前迈进了革命性的一步？如果从传统的程序设计观点看，问题的答案似乎不太明显。尽管Java对于解决传统的单机程序设计问题非常有用，但同样重要的是，它解决了在万维网（WWW）上的程序设计问题。

### 1.13.1 Web是什么

Web一词乍一看有点神秘，就像“网上冲浪”、“表现”、“主页”一样。回头审视它的真实面貌有助于对它的理解，但是要这么做就必须先理解客户/服务器系统，它是计算技术中另一个充满了诸多疑惑的话题。

1. 客户/服务器计算技术

客户/服务器系统的核心思想是：系统具有一个中央信息存储池（central repository of information），用来存储某种数据，它通常存在于数据库中，你可以根据需要将它分发给某些人员或机器集群。客户/服务器概念的关键在于信息存储池的位置集中于中央，这使得它可以被修改，并且这些修改将被传播给信息消费者。总之，信息存储池、用于分发信息的软件以及信息与软件所驻留的机器或机群被总称为服务器。驻留在用户机器上的软件与服务器进行通信，以获取信息、处理信息，然后将它们显示在被称为客户机的用户机器上。

客户/服务器计算技术的基本概念并不复杂。问题在于你只有单一的服务器，却要同时为多个客户服务。通常，这会涉及数据库管理系统，因此设计者把数据“均衡”分布于数据表中，以取得最优的使用效果。此外，系统通常允许客户在服务器中插入新的信息。这意味着必须保证一个客户插入的新数据不会覆盖另一个客户插入的新数据，也不会在将其添加到数据库的过程中丢失（这被称为事务处理）。如果客户端软件发生变化，那么它必须被重新编译、调试并安装到客户端机器上，事实证明这比想像的要更加复杂与费力。如果想支持多种不同类型的计算 51 机和操作系统，问题将更麻烦。最后还有一个最重要的性能问题：可能在任意时刻都有成百上千的客户向服务器发出请求，所以任何小的延迟都会产生重大影响。为了将延迟最小化，程序员努力减轻处理任务的负载，通常是分散给客户端机器处理，但有时也会使用所谓的中间件将负载分散给在服务器端的其他机器。（中间件也被用来提高可维护性。）

分发信息这个简单思想的复杂性实际上是有很多不同层次的，这使得整个问题可能看起来高深莫测。但是它仍然至关重要：算起来客户/服务器计算技术大概占了所有程序设计行为的一

中国互动出版网 www.china-pub.com | 网上书店 独家提供样章 HZ BOOKS



半，从制定订单、信用卡交易到包括股票市场、科学计算、政府、个人在内的任意类型的数据分发。过去我们所做的，都是针对某个问题发明一个单独的解决方案，所以每一次都要发明一个新的方案。这些方案难以开发且难以使用，而且用户对每一个方案都要学习新的接口。因此，整个客户/服务器问题需要彻底解决。

2. Web就是一台巨型服务器

Web实际上就是一个巨型客户/服务器系统，但稍微差一点，因为所有的服务器和客户机都同时共存于同一个网络中。你不需要了解这些，因为你所要关心的只是在某一时刻怎样连接到一台服务器上，并与之进行交互（即便你可能要满世界地查找你想要的服务器）。

最初只有一种很简单的单向过程：你对某个服务器产生一个请求，然后它返回给你一个文件，你的机器（也就是客户机）上的浏览器软件根据本地机器的格式来解读这个文件。但是很快人们就希望能够做得更多，而不仅仅是从服务器传递回页面。人们希望实现完整的客户/服务器能力，使得客户可以将信息反馈给服务器。例如，在服务器上进行数据库查找，将新信息添加到服务器以及下订单（这需要特殊的安全措施）。这些变革，正是我们在Web发展过程中所目睹的。

Web浏览器向前跨进了一大步，它包含了这样的概念：一段信息不经修改就可以在任意型号的计算机上显示。然而，最初的浏览器仍然相当原始，很快就因为加诸于其上的种种需要而陷入困境。浏览器并不具备显著的交互性，而且它趋向于使服务器和Internet阻塞，因为在任何时候，只要你需要完成通过编程来实现的任务，就必须将信息发回到服务器去处理。这使得即便是发现你的请求中的拼写错误也要花去数秒甚至是数分钟的时间。因为浏览器只是一个观察器，因此它甚至不能执行最简单的计算任务。（另一方面，它却是安全的，因为它在你的本地机器上不会执行任何程序，而这些程序有可能包含bug和病毒。）

52

为了解决这个问题，人们采用了各种不同的方法。首先，图形标准得到了增强，使得在浏览器中可以播放质量更好的动画和视频。剩下的问题通过引入在客户端浏览器中运行程序的能力就可以解决。这被称为“客户端编程”。

### 1.13.2 客户端编程

Web最初的“服务器-浏览器”设计是为了能够提供交互性的内容，但是其交互性完全由服务器提供。服务器产生静态页面，提供给只能解释并显示它们的客户端浏览器。基本的HTML（HyperText Markup Language，超文本标记语言）包含有简单的数据收集机制：文本输入框、复选框、单选框、列表和下拉式列表以及按钮——它只能被编程来实现复位表单上的数据或提交表单上的数据给服务器。这种提交动作通过所有的Web服务器都提供的通用网关接口（common gateway interface，CGI）传递。提交内容会告诉CGI应该如何处理它。最常见的动作就是运行一个在服务器中常被命名为“cgi-bin”的目录下的一个程序。（当点击了网页上的按钮时，如果观察浏览器窗口顶部的地址，有时可以看见“cgi-bin”的字样混迹在一串冗长和不知所云的字符中。）几乎所有的语言都可以用来编写这些程序，Perl已经成为最常见的选择，因为它被设计用来处理文本，并且是解释型语言，因此无论服务器的处理器和操作系统如何，它都适于安装。然而，Python（www.Python.org）已对其产生了重大的冲击，因为它更强大且更简单。

当今许多有影响力的网站完全构建于CGI之上的，实际上你几乎可以通过CGI做任何事。然而，构建于CGI程序之上的网站可能会迅速变得过于复杂而难以维护，并同时产生响应时间过长的问题。CGI程序的响应时间依赖于所必须发送的数据量的大小，以及服务器和Internet的负载。（此外，启动CGI程序也相当慢。）Web的最初设计者们并没有预见到网络带宽被人们开发的各种应用迅速耗尽。例如，任何形式的动态图形处理几乎都不可能连贯地执行，因为图形交互格式

53

中国互动出版网 www.china-pub.com | 网上书店 独家提供样章 HZ BOOKS



（graphic interchange format，GIF）的文件必须在服务器端创建每一个图形版本，并发送给客户端。再比如，你肯定经历过对Web输入表单进行数据验证的过程：你按下网页上的提交按钮；数据被发送回服务器；服务器启动一个CGI程序来检查、发现错误，并将错误组装为一个HTML页面，然后将这个页面发回给你；之后你必须回退一个页面，然后重新再试。这个过程不仅很慢，而且不太优雅。

问题的解决方法就是客户端编程。大多数运行Web浏览器的机器都是能够执行大型任务的强有力的引擎。在使用原始的静态HTML方式的情况下，它们只是闲在那里，等着服务器送来下一个页面。客户端编程意味着Web浏览器能用来执行任何它可以完成的工作，使得返回给用户的结果更加迅捷，而且使得你的网站更加具有交互性。

客户端编程的问题是：它与通常意义上的编程十分不同，参数几乎相同，而平台却不同。Web浏览器就像一个功能受限的操作系统。最终，你仍然必须编写程序，而且还得处理那些令人头晕眼花的成堆的问题，并以客户端编程的方式来产生解决方案。本节剩下的部分对客户端编程的问题和方法作一概述。

1. 插件

客户端编程所迈出的最重要的一步就是插件（plug-in）的开发。通过这种方式，程序员可以下载一段代码，并将其插入到浏览器中适当的位置，以此来为浏览器添加新功能。它告诉浏览器：从现在开始，你可以采取这个新行动了（只需要下载一次插件即可）。某些更快更强大的行为都是通过插件添加到服务器中的。但是编写插件并不是件轻松的事，也不是构建某特定网站的过程中所要做的事情。插件对于客户端编程的价值在于：它允许专家级的程序员不需经过浏览器生产厂商的许可，就可以开发某种语言扩展，并将它们添加到服务器中。因此，插件提供了一个“后门”，使得可以创建新的客户端编程语言（但是并不是所有的客户端编程语言都是以插件的形式实现的）。

54

2. 脚本语言

插件引发了浏览器脚本语言（scripting language）的开发。通过使用某种脚本语言，你可以将客户端程序的源代码直接嵌入到HTML页面中，解释这种语言的插件在HTML页面被显示时自动激活。脚本语言先天就相当易于理解，因为它们只是作为HTML页面一部分的简单文本，当服务器收到要获取该页面的请求时，它们可以被快速加载。此方法的缺点是代码会暴露给任何人去浏览（或窃取）。但是，通常不会使用脚本语言去做相当复杂的事情，所以这个缺点并不太严重。

如果你期望有一种脚本语言在Web浏览器不需要任何插件的情况下就可以得到支持，那它非JavaScript莫属（它与Java之间只存在表面上的相似，要想使用它，你必须在额外的学习曲线上攀爬。它之所以这样被命名只是因为想赶上Java潮流）。遗憾的是，大多数Web浏览器最初都是以彼此相异的方式来实现对JavaScript的支持的，这种差异甚至存在于同一种浏览器的不同版本之间。以ECMAScript的形式实现的JavaScript的标准化有助于此问题的解决，但是不同的浏览器为了跟上这一标准化趋势已经花费了相当长的时间（并且这种努力由于微软一直在推进它自己的VBScript形式的标准化日程而显得无所帮助，VBScript与JavaScript之间也存在着暧昧的相似性）。通常，你必须以JavaScript的某种最小公分母形式来编程，以使得你的程序可以在所有的浏览器上运行。JavaScript的错误处理的调试只能一团糟来形容。作为其使用艰难的证据，我们可以看到直到最近才有人创建了真正复杂的JavaScript脚本片段（Google在GMail），并且编写这样的脚本需要超然的奉献精神和超高的专业技巧。

这也表明，在Web浏览器内部使用的脚本语言实际上总是被用来解决特定类型的问题，主

中国互动出版网 www.china-pub.com | 网上书店 独家提供样章 HZ BOOKS



要是用来创建更丰富、更具有交互性的图形化用户界面（graphic user interface, GUI）。但是，脚本语言可以解决客户端编程中所遇到的百分之八十的问题。你的问题可能正好落在这百分之八十的范围之内，由于脚本语言提供了更容易、更快捷的开发方式，因此你应该在考虑诸如Java这样的更复杂的解决方案之前，先考虑脚本语言。

55

3. Java

如果脚本语言可以解决客户端编程百分之八十的问题的话，那么剩下那百分之二十（那才是真正难啃的硬骨头）又该怎么办呢？Java是处理它们最流行的解决方案。Java不仅是一种功能强大的、安全的、跨平台的、国际化的编程语言，而且它还在不断地被扩展，以提供更多的语言功能和类库，能够优雅地处理在传统编程语言中很难解决的问题，例如并发、数据库访问、网络编程和分布式计算。Java是通过applet以及使用Java Web Start来进行客户端编程的。

applet是只在Web浏览器中运行的小程序，它是作为网页的一部分而自动下载的（就像网页中的图片被自动下载一样）。当applet被激活时，它便开始执行一个程序，这正是它优雅之处：它提供一种分发软件的方法，一旦用户需要客户端软件时，就自动从服务器把客户端软件分发给用户。用户获取最新版本的客户端软件时不会产生错误，而且也不需要很麻烦的重新安装过程。Java的这种设计方式，使得程序员只需创建单一的程序，而只要一台计算机有浏览器，且浏览器具有内置的Java解释器（大多数的机器都如此），那么这个程序就可以自动在这台计算机上运行。由于Java是一种成熟的编程语言，所以在提出对服务器的请求之前和之后，可以在客户端尽可能多地做些事情。例如，不必跨网络地发送一张请求表单来检查自己是否填写了错误的日期或其他参数，客户端计算机就可以快速地标出错误数据，而不用等待服务器作出标记并给你传回图片。这不仅立即就获得了高速度和快速的响应能力，而且也降低了网络流量和服务器负载，从而不会使整个Internet的速度都慢了下来。

4. 备选方案

老实说，Java applet没有达到当初它所吹嘘的境界。当Java首度出现时，似乎大家最欢欣鼓舞的莫过于applet了，因为它们最终将解决严峻的客户端可编程性问题，从而提高基于互联网的应用的可响应性，同时降低它们对带宽的需求。人们展望到了大量的可能性。

56

实际上，你可以发现在Web上确实存在一些非常灵巧的applet，但是压倒性的向applet的迁移却始终未发生。这其中最大的问题可能在于安装Java运行时环境（JRE）所必需的10MB带宽对于一般的用户来说过于恐怖了，而微软没有选择在IE（Internet Explorer）中包含JRE这一事实也许就此已经封杀了applet的命运。无论怎样，Java applet始终没有得到大规模应用。

尽管如此，applet和Java Web Start应用在某些情况下仍旧很有价值。无论何时，只要你想控制用户的机器，例如在一个公司的内部，使用这些技术来发布和更新客户端应用就显得非常恰当，并且这可以节省大量的时间、人力和财力，特别是你需要频繁地更新的时候。

在“图形化用户界面”一章中，我们将看到一种折中的新技术，Macromedia的Flex，它允许你创建基于Flash的与applet相当的应用。因为Flash Player在超过98%的Web浏览器上都可用（包含Windows, Linux和Mac操作系统上的浏览器），因此它被认为是事实上已被接受的标准。安装和更新Flash Player都十分快捷。ActionScript语言是基于ECMAScript的，因此我们对它应该很熟悉，但是Flex使得我们在编程时无需担心浏览器相关性，因此，它远比JavaScript要吸引人得多。对于客户端编程而言，这是一种值得考虑的备选方案。

5. .NET和C#

曾几何时，Java applet的主要竞争对手是微软的ActiveX——尽管它要求客户端必须运行Windows平台。从那以后，微软以.NET平台和C#编程语言的形式推出了与Java全面竞争的对

中国互动出版网 www.china-pub.com | 网上书店 独家提供样章 HZ BOOKS



手。.NET平台大致相当于Java虚拟机（JVM，即执行Java程序的软件平台）和Java类库，而C#毫无疑问与Java有类似之处。这当然是微软在编程语言与编程环境这块竞技场上所做出的最出色的成果。当然，他们有相当大的有利条件：他们可以看得到Java在什么方面做得好，在什么方面做得还不够好，然后基于此去构建，并要具备Java不具备的优点。这是自从Java出现以来，Java所碰到的真正的竞争。因此，Sun的Java设计者们已经认真仔细地去研究了C#，以及为什么程序员们可能会转而使用它，然后通过在Java SE5中对Java做出的重大改进而做出了回应。

57

目前，.NET主要受攻击的地方和人们所关心的最重要的问题就是，微软是否会允许将它完全地移植到其他平台上。微软宣称这么做没有问题，而且Mono项目（www.go-mono.com）已经有了一个在Linux上运行的.NET的部分实现；但是，在该实现完成及微软不会排斥其中的任何部分之前，.NET作为一种跨平台的解决方案仍旧是一场高风险的赌博。

6. Internet与Intranet

Web是最常用的解决客户/服务器问题的方案，因此，即便是解决这个问题的一个子集，特别是公司内部的典型的客户/服务器问题，也一样可以使用这项技术。如果采用传统的客户/服务器方式，可能会遇到客户端计算机有多种型号的问题，也可能会遇到安装新的客户端软件的麻烦，而它们都可以很方便地通过Web浏览器和客户端编程得以解决。当Web技术仅限用于特定公司的信息网络时，它就被称为Intranet（企业内部网）。Intranet比Internet提供了更高的安全性，因为可以物理地控制对公司内部服务器的访问。从培训的角度看，似乎一旦人们理解了浏览器的基本概念后，对他们来说，处理网页和applet的外观差异就会容易得多，因此对新型系统的学习曲线也就减缓了。

安全问题把我们带到了一个领域，这似乎是在客户端编程世界自动形成的。如果程序运行在Internet之上，那么就不可能知道它将运行在什么样的平台之上，因此，要格外小心，不要传播有bug的代码。你需要跨平台的、安全的语言，就像脚本语言和Java。

如果程序运行与Intranet上，那么可能会受到不同的限制。企业内所有的机器都采用Intel/Windows平台并不是什么稀奇的事。在Intranet上，你可以对你自己的代码质量负责，并且在发现bug之后可以修复它们。此外，你可能已经有了以前使用更传统的客户/服务器方式编写的遗留代码，因此，你必须在每一次升级时都要物理地重装客户端程序。在安装升级程序时所浪费的时间是迁移到浏览器方式上的最主要的原因，因为在浏览器方式下，升级是透明的、自动的（Java Web Start也是解决此问题的方式之一）。如果你身处这样的Intranet之中，那么最有意义的方式就是选择一条能够使用现有代码库的最短的捷径，而不是用一种新语言重新编写你的代码。

58

当面对各种令人眼花缭乱的解决客户端编程问题的方案时，最好的方法就是进行性价比分析。认真考虑问题的各种限制，然后思考哪种解决方案可以成为最短的捷径。既然客户端编程仍然需要编程，那么针对自己的特殊应用选取最快的开发方式总是最好的做法。为那些在程序开发中不可避免的问题提早作准备是一种积极的态度。

### 1.13.3 服务器端编程

前面的讨论忽略了服务器端编程的话题，它是Java已经取得巨大成功的因素之一。当提出对服务器的请求后，会发生什么呢？大部分时间，请求只是要求“给我发送一个文件”，之后浏览器会以某种适当的形式解释这个文件，例如将其作为HTML页面、图片、Java applet或脚本程序等来解释。

更复杂的对服务器的请求通常涉及数据库事务。常见的情形是复杂的数据库搜索请求，然后服务器将结果进行格式编排，使其成为一个HTML页面发回给客户端。（当然，如果客户端通

中国互动出版网 www.china-pub.com | 网上书店 独家提供样章 HZ BOOKS



过Java或脚本程序具备了更多的智能，那么服务器可以将原始的数据发回，然后在客户端进行格式编排，这样会更快，而且服务器的负载将更小。）另一种常见情形是，当你要加入一个团体或下订单时，可能想在数据库中注册自己的名字，这将涉及对数据库的修改。这些数据库请求必须通过服务器端的某些代码来处理，这就是所谓的服务器端编程。过去，服务器端编程都是通过使用Perl、Python、C++或其他某种语言编写CGI程序而实现的，但却造成了从此之后更加复杂的系统。其中就包括基于Java的Web服务器，它让你用Java编写被称为servlet的程序来实现服务器端编程。servlet及其衍生物JSP，是许多开发网站的公司迁移到Java上的两个主要的原因，尤其是因为它们消除了处理具有不同能力的浏览器时所遇到的问题。服务器端编程的话题在《企业Java编程思想》（Thinking in Enterprise Java）一书中有所论述。 59

## 1.14 总结

你知道过程型语言看起来像什么样子：数据定义和函数调用。想了解此类程序的含义，你必须忙上一阵，需要通读函数调用和低层概念，以在脑海里建立一个模型。这正是我们在设计过程式程序时，需要中间表示形式的原因。这些程序总是容易把人搞糊涂，因为它们使用的表示术语更加面向计算机而不是你要解决的问题。

因为OOP在你能够在过程型语言中找到的概念的基础上，又添加了许多新概念，所以你可能会很自然地假设：由此而产生的Java程序比等价的过程型程序要复杂得多。但是，你会感到很惊喜：编写良好的Java程序通常比过程型程序要简单得多，而且也易于理解得多。你看到的只是有关下面两部分内容的定义：用来表示问题空间概念的对象（而不是有关计算机表示方式的相关内容），以及发送给这些对象的用来表示在此空间内的行为的消息。面向对象程序设计带给人们的喜悦之一就是：对于设计良好的程序，通过阅读它就可以很容易地理解其代码。通常，其代码也会少很多，因为许多问题都可以通过重用现有的类库代码而得到解决。

OOP和Java也许并不适合所有的人。重要的是要正确评估自己的需求，并决定Java是否能够最好地满足这些需求，还是使用其他编程系统（包括你当前正在使用的）才是更好的选择。如果知道自己的需求在可预见的未来会变得非常特殊化，并且Java可能不能满足你的具体限制，那么就应该去考察其他的选择（我特别推荐读者看看Python，参见www.Python.org）。即使最终仍旧选择Java作为编程语言，至少也要理解还有哪些选项可供选择，并且对为什么选择这个方向要有清楚的认识。 60