致读者：

我从 2002 年 7 月开始翻译这本书，当时还是第二版。但是翻完前言和介绍部分后，chinapub 就登出广告，说要出版侯捷的译本。于是我中止了翻译，等着侯先生的作品。

我是第一时间买的 这本书，但是我失望了。比起第一版，我终于能看懂这本书了，但是相比我的预期，它还是差一点。所以当 Bruce Eckel 在他的网站上公开本书的第三版的时候，我决定把它翻译出来。

说说容易，做做难。一本 1000 多页的书不是那么容易翻的。期间我也曾打过退堂鼓，但最终还是全部翻译出来了。从今年的两月初起，到 7 月底，我几乎放弃了所有的业余时间，全身心地投入本书的翻译之中。应该说，这项工作的难度超出了我的想像。

首先，读一本书和翻译一本书完全是两码事。英语与中文是两种不同的语言，用英语说得很畅的句子，翻成中文之后就完全破了相。有时我得花好几分钟，用中文重述一句我能用几秒钟读懂的句子。更何况作为读者，一两句话没搞懂，并不影响你理解整本书，但对译者来说，这就不一样了。

其次，这是一本讲英语的人写给讲英语的人的书，所以同很多要照顾非英语读者的技术文档不同，它在用词，句式方面非常随意。英语读者会很欣赏这一点，但是对外国读者来说，这就是负担了。

再有，Bruce Eckel 这样的大牛人，写了 1000 多页，如果都让你读懂，他岂不是太没面子？所以，书里还有一些很有“禅意”的句子。比如那句著名的“The genesis of the computer revolution was in a machine. The genesis of our programming languages thus tends to look like that machine.”我就一直没吃准该怎么翻译。我想大概没人能吃准，说不定 Bruce 要的就是这个效果。

这是一本公认的名著，作者在技术上的造诣无可挑剔。而作为译者，我的编程能力差了很多。再加上上面讲的这些原因，使得我不得不格外的谨慎。当我重读初稿的时候，我发现需要修改的地方实在太多了。因此，我不能现在就公开全部译稿，我只能公开已经修改过的部分。不过这不是最终的版本，我还会继续修订的。

本来，我准备到 10 月份，等我修改完前 7 章之后再公开。但是，我发现我又有点要放弃了，因此我决定给自己一点压力，现在就公开。以后，我将修改完一章就公开一章，请关注 www.wgqqh.com/shhgs/tij.html。

如果你觉得好，请给告诉我，你的鼓励是我工作的动力；如果你觉得不好，那就更应该告诉我了，我会参考你的意见作修改的。我希望能通过这种方法，译出一本配得上原著的书。

shhgs

2003 年 9 月 8 日

1: 对象简介

“我们剖析事物的本质，从中形成概念，并根据需要赋予它重要性。这一切很大程度上是源于，我们所使用的语言已经在其形式中包含了一套为它的使用者所完全接受的规范，而我们正属于接受这一规范的那群人...如果我们拒不接受语言在数据的组织与分类方面的原则的话，那我们根本就没法说话。” Benjamin Lee Whorf (1897-1941)

计算机革命的推动力在于机器。于是机器的发展也左右了编程语言发展。

然而真正被用作延伸智力的工具的计算机 (就像 Steve Jobs 喜欢说的，“思想的自行车”) 并不很多，它们更多的是被用来表达思想。所以结果就是，它们看上去已经不那么像机器了，而更像我们思想的一部分。就像文字，绘画，雕塑，动画以及电影一样，它已经成为一种新的媒体。而面向对象的编程(Object-oriented programming 缩写是 OOP)正是这一运动的一部分。

本章会向你介绍 OOP 的基本概念和开发的方法。本章，以及本书都假设你有过程语言的编程经验，虽然不一定是 C 的。如果你决定在继续本书之前，要在编程和 C 的语法上作更为充分的准备，你可以去看看本书封底所附的，名为 *Foundations for Java* 的练习 CD。

本章是一个背景介绍和补充材料。很多读者会觉得，在没理解大背景之前就去编写面向对象的程序，会有点不舒服。因此这里会对很多概念做个介绍，并且让你对 OOP 的概况有个清楚的了解。然而，有些人会认为在见到具体的工作机制之前没必要了解整个大背景；这些人在还没有看到代码之前就会被搞昏的。如果你属于后者，希望尽快进入这个语言的细节，没关系，跳过这章就是了——跳过这章不会妨碍你写程序或者学语言。但是最终你还是要回来补课的，因为只有这样你才会理解为什么对象很重要，而且该如何把它们用到设计中去。

抽象的过程

所有的编程语言都提供抽象。甚至可以这么说，你能解决的问题的复杂程度直接与抽象的种类与质量相关。我所说的“种类”是指，“你到底抽象了些什么”。汇编语言是对计算机的抽象，随后许多所谓的“命令”语言(诸如 FORTRAN, BASIC 和 C)是对汇编语言的抽象。这些语言较汇编语言有了巨大的进步，但这仍然是一种初级的抽象，仍然要求你从计算机的角度，而不是从待解决的问题的角度来思考。程序员必须在机器模型(在“解决空间”，也就是你要对那个问题建模的地方，比如在计算机上)与待解决的问题的模型(在“问题空间”，也就是有问题要解决的地方)之间建立关联。处理这种映射所带来的压力，以及编程语言对此无能为力的现

实，造成了程序难写以及维护代价高昂的后果。此外它还带来一个副产品，就是所谓的“编程方法”的行业。

取代对机器建模的另一个途径是对你要解决的问题建模。早期语言，诸如 LISP 和 APL 选择了“以某种特殊的视角来观察世界”的方法(分别是“归根到底，所有问题都是列表”或“所有问题都是算法”)。PROLOG 则将所有问题都转换成一系列决策。用这类语言编程，成了在约束条件下的编程，而编程也仅限于操控图像符号。(后来证明这种做法的局限性太大了。)它们的设计是针对某些很特殊的问题的。对于那些问题，这些方法都干得不错，但是一旦步出这一领域，它们就显得太滑稽了。

面向对象的方法则更进了一步，它为程序员提供了能在问题空间表述各种元素的工具。这种表述是非常通用的，这样程序员就不会被限制在某类特殊问题上了。我们将问题空间的元素同它在解决空间的表述称为“对象”。(此外还需要一些在问题空间并无对照的对象。)这一思想的要点是，通过往程序里面添加新的对象，可以让它适用于问题的各种变例。于是当你阅读代码的时候，它也在向你讲述它要解决的问题。这比我们曾经见过的语言抽象更为灵活也更为强大。[2] 由此 OOP 允许你以问题的角度，而不是以要解决问题的计算机的角度来描述问题。不过它与计算机还是有联系:每个对象看上去都有点像计算机——它有状态，有可以让你来执行的操作。但看上去这种对真实世界的模拟还不算太糟——在真实世界中对象也都有自己的特点和行为。

Alan Kay 总结了 Smalltalk 的五项基本特征。这是最早获得成功的面向对象的编程语言，也是 Java 所依赖的基础之一。这些特征代表了纯的面向对象的编程方法:

1. **万物皆对象。**将对象想成一种特殊的变量；它存储数据，而且还可以让你“提要求”，命令它进行某些操作。从理论上讲，你可以把所有待解决的问题中的概念性组件(狗，建筑，服务等)都表示成程序里的对象。
2. **程序就是一组相互之间传递消息，告诉对方该干些什么的对象**。你只要向那个对象“发一个消息”，就能向它提出要求。更确切的说，你可以这样认为，消息是调用专属某个对象的方法的请求。
3. **每个对象都利用别的对象来组建它自己的记忆**。换言之，你通过将已有的对象打成一个包，来创建新的对象。由此，你可以将程序的复杂性，隐藏在对象的简单性之下。
4. **对象都有类型**。说这句话的意思是，任何对象都是某个类的实例(*instance of a class*)，而这里的“类(class)”就是“类型(type)”的意思。用以区分类的最突出的特点就是“你能传给它什么消息？”
5. **所有属于同一类型的对象能接受相同的消息**。以后你就会知道，实际上这是“定义”而不是特点。一个“circle”型的对象也是一个“shape”型的对象，所以可以保证 circle 能接受 shape 的消息。也就是说，你写给 shape 的代码能自动交由任何符合 shape 描述的东西处理。这种互换性(*substitutability*)是 OOP 最强大的功能之一。

Booch 还给对象下了个更为简洁的定义：

对象有状态，行为和标识。

这就是说，对象可以有内部数据(这给了它状态)，有方法(因而产生了行为)，以及每个对象都能同其它对象区分开来——具体而言，每个对象在内存里都有唯一的地址。[3]

对象有接口

或许亚里士多德是第一个认真研究类型(*type*)这个概念的人；他提到过“鱼类和鸟类”这种概念。对象，虽然都是独一无二的，但同时也是某种有着相同特征和行为的对象的类的一员。这个概念直接以 **class** 关键词的形式为 Simula-67——第一种面向对象的语言所采纳。而 **class** 也为程序引入了一种全新的类型。

Simula，就像它的名字所说的，是为了开发诸如经典的“银行出纳问题”而创建的。在这个问题中，你有很多出纳员、客户、帐户、交易以及钱——总之，很多“对象”。那些除了程序执行时状态会有所不同，其它都完全相同的对象，会被集中起来，统称为“对象的类(classes of objects)”。由此得出了 **class** 关键词。在面向对象的编程中，创建抽象的数据类型(class)是一项基本概念。抽象数据类型的工作方式同内置数据类型的几乎没什么不同：你可以创建这个类型的变量(在 OOP 的术语中，这被称为对象 *object* 或实例 *instance*)，然后操纵这些变量(称为送消息 *sending messages* 或请求 *request*，你传给对象一个消息，由它来决定该作些什么。)每个类的每个成员(元素 element)都有某些共性：每个帐户都有余额，某个出纳都能受理存款，等等。同时，每个成员还都有自己的状态。每个帐户的余额是不一样的，每个出纳都有自己的名字等等。因此，出纳、客户、帐户、交易等等，它们每个都在计算机程序中代表一个唯一的实体。这个实体便是对象，而每个对象都属于某个类，而这个类会定义它的特征与行为。

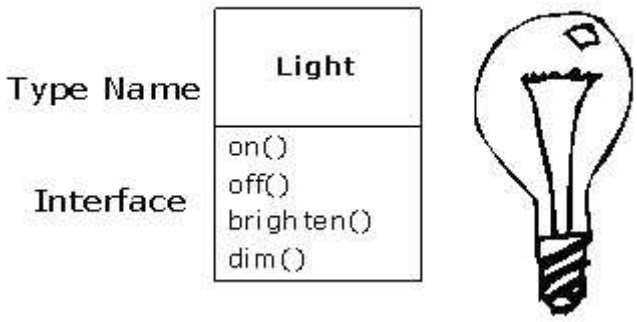
因此，虽然面向对象的编程就是创建一些新的数据类型，但几乎所有的 OOP 语言都选择“class”这个关键词。所以，当你看到“type”的时候，应该想到它就是“class”，反之亦然。[4]

由于类描述的是一组具有相同特征(数据元素)和行为(功能)的对象，所以类实际上是一种数据类型。因为，就拿浮点数举例，它也有它自己的特征和行为。类同数据类型的区别在于，类是程序员为解决特定问题而定做的，而数据类型是为了表示机器的存储单元而设计的，它是现成的，是程序员无可奈何时的选择。你可以根据需要添加新的数据类型，并以此来扩充编程语言。编程系统欢迎这些新的类，它会在管理和类型检查方面给予这些类与内置类完全相同的待遇。

面向对象的方法并不限于创建模拟场景。不论你承认与否，任何程序都在模拟你所设计的系统。用了 OOP 技术，你就能很容易地将一大堆问题简化成一个简单的解决方案。

一旦创建了类，你就能创建任意多的那个类的对象，然后把它们当作待解决的问题中的元素来进行操控。实际上，面向对象的编程的挑战之一就是，如何在问题空间的元素与解决空间的对象之间建立一种一对一的映射。

但是怎样让那些对象来为你干活呢？必需要有办法向对象发请求，这样它才能工作。比如完成一次交易，在屏幕上画些东西或者按一下开关之类的。而且每个对象只能满足某些请求。你能向对象发送的请求是由其接口(*interface*)所定义的，而决定接口的则是对象的类型。可以用电灯泡来作一个简单的例子：




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

接口只管你能向这个对象发什么请求。还必须要有能满足这一请求的代码。而这些代码，以及隐藏着的数据，组成了实现(*implementation*)。从过程编程的角度来讲，事情没那么复杂。类中的方法都是与各种请求相关联的，因此当你向这个对象提出请求的时候，就会调用这个方法。这个过程通常被称为，你向一个对象“传了一个消息(提了个请求)”，而这个对象会决定该如何处置这个消息(执行代码)。

这里，这个类型/类的名字是 **Light**，而这个具体的 **Light** 对象的名字是 **lt**。你向 **Light** 对象提出的请求有开灯，关灯，让它更亮一点或是暗点。通过定义 **Light** 类型的“reference”(**lt**)，和用 **new** 请求一个新对象，你创建了一个 **Light** 对象。发送消息的时候，你要用点号(dot/period)将对象的名字和那个消息连起来。如果编程的时候用到的都是已有的类，那么这点知识已经够用了。

上面的图表遵循了 *Unified Modeling Language*(UML)的格式。每个类都由一个方框来表示，这个类的名字写在方框的顶部，中间是你要关心的

数据元素，而方法(*method*，指属于这个对象的函数。它接受你传给这个对象的消息)则列在方框的底部。通常 UML 的设计图只会显示类的名字以及 public 方法，所以中间部分是不显示的。如果你只对类的名字感兴趣，那么最下面的那部分也可以不显示。

对象会提供服务

当你开发一个程序或者分析一个程序的设计时，理解对象的最佳的方式是把它们当作“服务的提供者”。程序本身会为用户提供服务，而它通过使用其它对象所提供的服务来完成这个工作。你的任务是制作(或者在更理想的情况下，从现有的代码库中找出)一组能为解决问题提供最佳服务的对象。

这么做的第一步是问“如果我可以像变魔术那样把东西从帽子里拿出来，我该拿出些什么东西，哪些对象能立即帮我解决问题？”举例来说，假设你要创建一个簿记程序。可能你会想应该有一些保存预设的输入界面的对象，一组进行簿记计算的对象，以及一个能在各种打印机上打印支票和发票的对象。有些对象或许已经有了，但是那些还没有的应该是什么样的呢？它们应该提供哪种服务，还有它们要完成任务的话，又该用哪些对象呢？如果你不断分析下去，最终你会发现，不是“那个对象写起来很容易”就是“那个对象已经有了。”这是将问题分解成一组对象的一个合理的方法。

将对象视作服务的提供者还有一个额外的优点：能提高对象的内聚性(cohesion)。内聚性高是高质量的软件设计一个基本要求：就是说软件的各种组件(比如对象，也可以是方法或类库)应该能很好的“组装在一起”。设计对象时常犯的一个错误就是，往对象里塞了太多的功能。举例来说，设计支票打印模块的时候，你也许会决定设计一个能通晓所有排格式和打印工作细节的对象。很快你就会发现这个任务太艰巨了，或许应该用三个或是更多对象来完成这个工作。第一个对象应该是支票格式的目录册，通过查询这个目录册可以获得该如何打印支票的信息。第二个对象，或是一组对象，应该是能分辨各种打印机的通用的打印接口(但是对簿记，它一窍不通——这个模块应该买来而不是自己写)。以及使用上述两个对象所提供的服务的，能最终完成任务的第三个对象。由此每个对象都提供一组互补的功能。在一个良好的面向对象的设计中，每个对象都应该只作一件事，并且作好一件事，而不是去作太多的事情。就像这里看到的，这样不仅能发现哪些对象应该买(打印机接口对象)，而且能设计出今后能复用的对象(支票格式的目录册)。

将对象视作服务的提供者还是一种很了不起的简化工具。它不仅在设计过程中有用，而且还能帮助别人理解你的代码或者复用这个对象——如果他们认同这个对象所提供的服务的话。将对象视作服务的提供者能使对象更容易地被用于设计。

隐藏实现

将程序员分成类的创建者(*class creator* 那些创建新的数据类型的人) 和客户程序员[5] (*client programmer* 那些使用这些类编程的人)能帮助我们更好地理解这个问题。客户程序员的目的是收集各种类，以便能快速开发应用程序。而类的创建者的目的是创建一些这样的类，它们只透露一些客户程序员必须知道的东西，其它部分则被完全隐藏了。为什么？因为隐藏之后，客户程序员就不能访问了，也就是说类的创建者们可以根据需要修改隐藏部分而不用担心会对其他人造成影响。通常隐藏起来的都是这个类最脆弱的，很容易被粗心大意或不知情的客户程序员给弄坏的内脏部分，所以把实现隐藏起来能减少程序的 bug。

隐藏实现，这一概念的重要性无论如何强调都不会过分。无论是那种关系，有一个为各方都尊重的边界是非常重要的。创建一个类库之后，你就与客户程序员建立了某种关系。他们也是程序员，他们用你的类库来组装一个应用程序，也可能是一个更大的类库。如果所有人都能看到类的所有成员，那么客户程序员就能对这个类作任何事，因此也谈不上什么强制性的规则了。尽管你真的不想让客户程序员直接操控类中的某些成员，但是如果没有访问控制的话，你也没办法禁止他去这么做。这个类只能赤裸裸地面对整个世界。

所以要控制访问权限的首要原因就是，禁止那些客户程序员去碰他们不该动的东西——就是那些数据类型内部运作所必须的东西。只允许他们接触解决问题所必需的接口。实际上这也是提供给用户的一种服务，因为由此他们能很方便的看出哪些东西对他们很重要，哪些跟他们没关系。

第二个原因是允许类库设计人员能在不打搅客户程序员的情况下修改类的内部工作方式。比如，刚开始设计这个类的时候时，为了降低开发难度，你用了一种很简单的方法，但随后你发觉应该重写这段代码，让跑得快一些。如果这个类的接口与实现分得很清楚，并且保护得很好，那么做起来就会很方便。

Java 用了三个明确的关键词来设置类中的边界：public，private，和 protected。它们的用法和意思都相当明了。这些访问控制符表示谁能使用由它定义的东西。public 表示后面跟着的东西谁都能用。而 private 关键词则表示，除非是类的创建者用这个类的方法进行访问，否则没人能访问到这个元素。private 是竖在你与客户程序员之间的一堵墙。那些要访问 private 成员的人，会在编译的时候得到一个错误。protected 关键词很像 private，它们的区别在于，继承类能够访问 protected 成员，但是不能访问 private 成员。继承问题过一会再介绍。

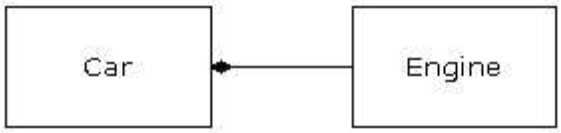
Java 还有个“缺省”的访问权限，如果你没用上面三个的话，那就是指它了。通常把它称为 package 访问权限。因为同属这个 package 的类

可以访问这个 package 中其它类的“缺省权限”的成员，但是出了这个 package，它们就都是 private 的了。

复用实现

一旦类创建完成并且测试通过之后，它就应该能(很好地)表示一段有用的代码。但实际上代码复用并不像我们希望的那样容易；要设计一个复用性良好的对象，需要经验和远见。代码复用是 OOP 语言最显著的优点之一。

复用代码最简单的方式是直接用这个类来创建对象，但是你也可以将那个类的对象放到一个新的类中。我们把它称为“创建一个成员对象”。为了能让那个新的类提供你所设计的功能，它可以由任意多个，任意类型的对象，以任意形式组合在一起。由于你是用已有的类来合成新的类，因此这一概念被称为合成(*composition*，如果这个对象是动态合成的，通常把它称为聚合 *aggregation*)。通常将合成称为“有(has-a)”关系，就像“轿车有引擎”。



(这个讲解轿车的 UML 关系图，用实心菱形表示合成关系。通常如果要让我表示这个关系的话会更简单：就是一根直线，没有菱形。[6])

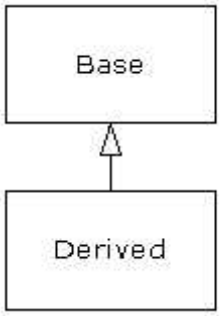
合成具有极大的灵活性。新类里面的成员对象通常都是 private 的，因此使用这些类的客户程序员们是无法访问这些对象的。这点能让你在不影响已有的客户代码的前提下，修改这些类。你还可以在运行时修改这些成员对象，并以此动态地改变程序地运行。下面要讲的继承就没有这种灵活性，如果你用继承创建一个类，编译器肯定会加上某些编译时的限制的。

由于继承在 OOP 中是如此重要，因此会被反复强调，以至于新手们会认为，应该尽量使用继承。其实一味的使用继承会导致很奇怪也很复杂的设计。相反，在创建新类的时候，你应该优先考虑使用合成，这么做会更简单也更灵活。如果这么做的话，整个设计也会变得更加井井有条。等你有了一点经验之后，你就会自然而然的知道继承应该用在哪里了。

继承：复用接口

对象这个概念本身就是一件很有力的工具。它能让你依照概念把数据与功能结合在一起，这样你就不用站在运行程序的计算机的角度上，而是站在解决问题的角度上来写代码了。这个概念作为程序的基本单位，在编程语言中用 **class** 这个关键词表示。

然而，要是我们费尽心机编写了一个类之后，又不得不再写去一个功能类似的全新的类的话，那就很惨了。要是能将已有的类克隆一份，然后在克隆的类上修修补补，那就好了。而这正是继承(*inheritance*)的功效。只是区别在于，如果原先那个类(称为基类 *base class*，父类 *parent class* 和 超类 *superclass*)作过了修改，那么这个“克隆”(称为衍生类，派生类 *derived class* 或者继承类 *inherited class* 或者子类 *subclass, child class*)也会随之发生变化。

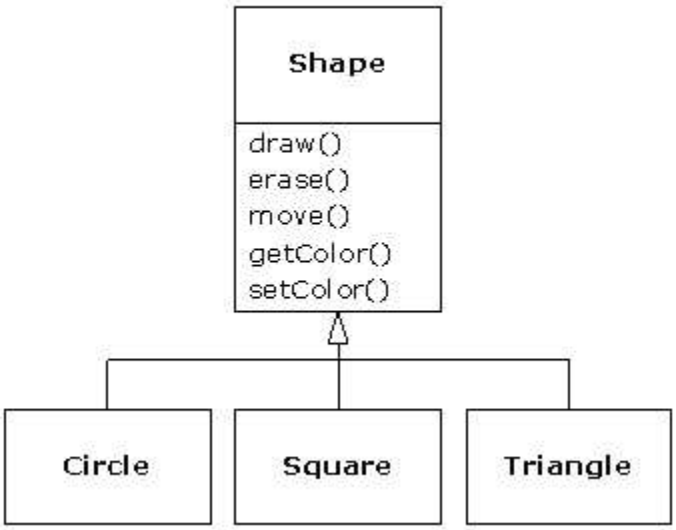


(在 UML 关系图中，箭头从派生类指向基类。你会发现，通常会有不止一个派生类。)

类不仅仅在描述一组施加在对象之上的限制；它还会同其它类发生关系。两个类可以有一些共通的特征和行为，可能其中有一个比另一个特征更多，能处理的消息也更多(或者处理的方式不同)。通过基类和派生类的概念，继承表达了这种相似性。基类保存的是所有继承自它的类的共有特征和行为。创建基类是为了表达，你对系统中某些对象的核心思想的理解。而衍生类则是从基类派生出来的，它所表达的是各种实现这个核心思想的方式。

举例来说，有一个对垃圾分类的垃圾回收机。基类是“垃圾(trash)”，每件垃圾都有重量，价值等等，而且可以切碎，融化或者分解。由它派生出的一些具体的垃圾品种会有一些额外的特征(瓶子有颜色)或是行为(铝罐可以压缩，铁罐可以磁化)。此外有些行为会不同(纸张的价值由它的品种和状况决定)。通过继承，你能用类构建一个表述你要解决的这个问题的类系(type hierarchy)。

第二个例子是经典的“shape”案例。或许它能用于计算机辅助设计系统或是用来模拟游戏场景。基类是“shape”，每种形状都有大小，颜色，位置等属性。从它那里继承下来的具体的形状——圆，矩形，三角形等——都有额外的特征和行为。比如某些形状可以翻转。有些行为会不同，比如计算面积的方法。整个类系体现了各种形状之间的相同和不同点。

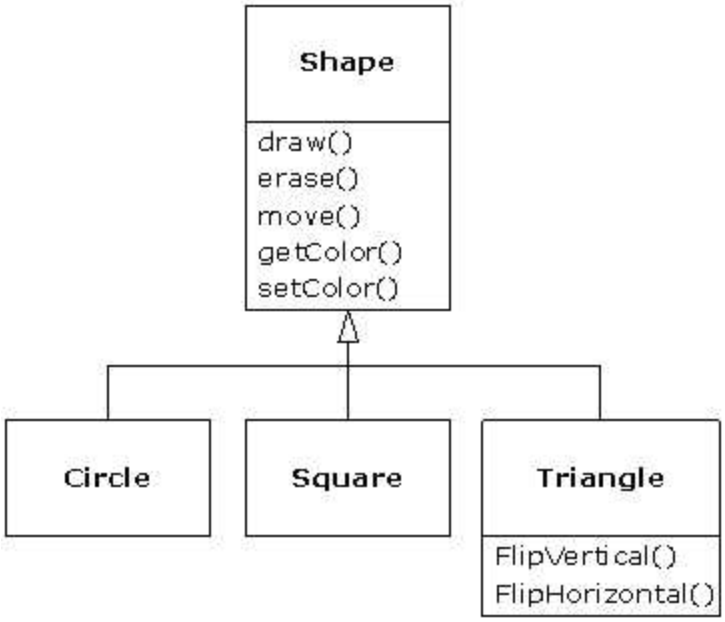


将描述问题之用的术语用于解决问题也是大有裨益的，因为这样你就不再需要那些在讲述问题和解决方案之间的中间模型了。对于对象，类层次结构是最主要的模型，由此你从用文字描述真实世界中的系统进到了用代码描述这个系统。实际上，大家在用 OOP 设计时会遇到的难题之一是，它从头到尾太简单了。习惯于用复杂方法解决问题的人可能会一开始就被这种简单给难住了。

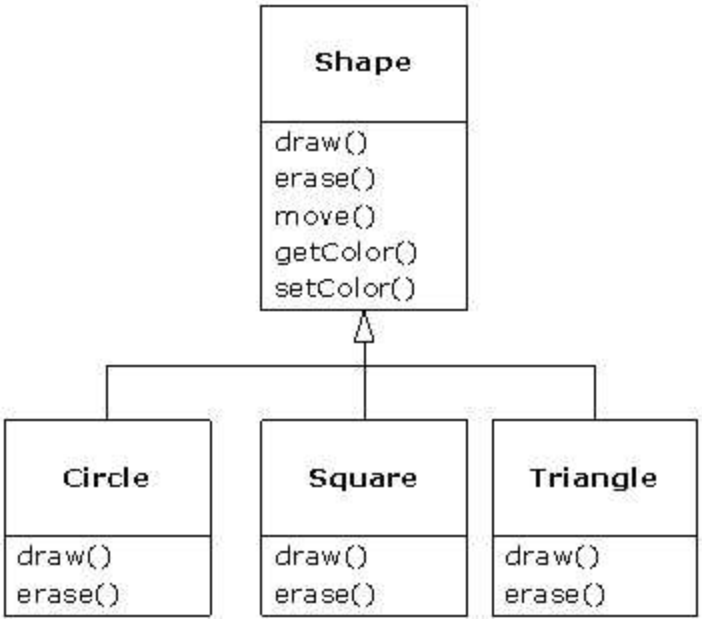
当你继承了一个类时，你也创建了一个新的类。这个新的类不仅包含了已有类的所有成员(尽管 **private** 成员已经隐藏起来了，是不能访问的)，更重要的是它复制了基类的接口。于是所有能够传给基类的消息也都可以传给派生类。由于我们根据能它能接受什么消息来判断这是什么类，因此这就意味着派生类“和基类是属于同一类型的”。在上面的例子里，“圆也是一种形状”。这种由继承而产生的类的相等性是一扇能让你理解 OOP 意义的门。

既然基类和派生类具有相同的基本接口，那么这个接口的背后就必须跟着实现。也就是当对象收到某个消息的时候，它必须能执行一些代码。如果你只是继承了一个类，其他什么都不做，那么基类里的方法会直接带进派生类。也就是说派生类的对象不但与基类的对象的类型相同而且行为也一样。这可没什么吸引力。

你有两种办法来区分新的派生出来的类和那个原来的基类。第一种方法很简单：直接往派生类里加新的方法。这些新的方法不属于基类的接口。也就是说由于基类不能完成所有的工作，你必须加上更多方法。这种简单原始的继承方法常常是解决问题的完美方案。然而你还得仔细看看基类是不是也需要这些方法。对于 OOP 的设计，这种反复发现的过程是很常见的。



尽管有些时候继承会隐含(特别是在 Java，其 **extends** 关键词专门用于继承)你准备往接口里加进一些新的方法，但这并不是必须的。第二个，也是更重要的区分方法是在新的类中修改基类方法的行为。这被称为覆写(*override*)那个方法。

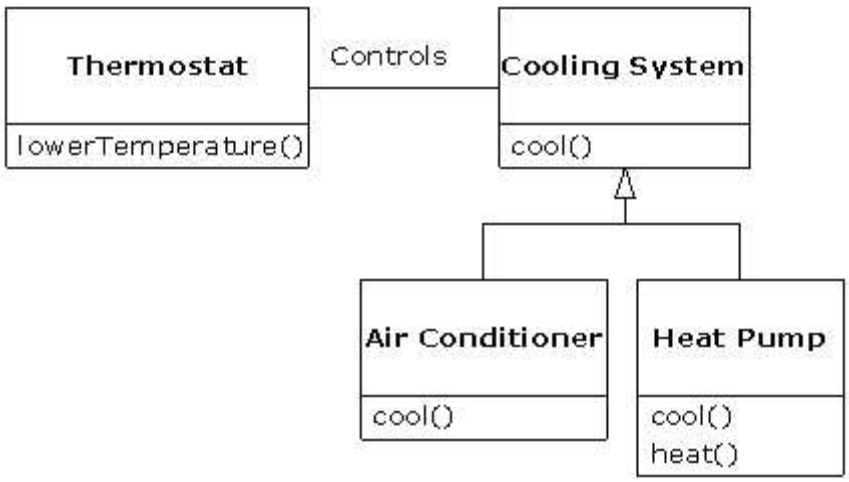


要覆写一个方法，你只要直接在派生类里重新定义那个方法就行了。你的意思是“我这里要用的是同一个接口里的方法，不过我要让它为新的类做一些不同的事。”

“是”与“像”的关系

继承能引发如下的争论；继承是不是应该“只覆写”基类中的方法(不能加入基类中没有的方法)？也就是说，因为接口相同，所以派生类“也就是”基类。结果是你能用派生类的对象直接替换基类的对象。你可以把这种情况想成完全替换(*pure substitution*)，通常这被称为替换原则(*substitution principle*)。从某种意义上讲，这是使用继承的理想方法。通常我们把基类同派生类的这种关系称作是(*is-a*)关系，因为我们会说“圆是一种形状”。有一种测试继承的方法，就是问自己能不能说这两个类之间存在“是”关系。

有时你不得不往派生类里加上一些新的接口元素，这样派生类的接口就得到了扩展，而派生类也因此变成了一个新的类。基类还是可以被新的类所替代，但是这种替换是不完全的，因为新的方法无法通过基类的接口访问。这可以被称为“像(*is-like-a*)”关系(我的术语)。新类有旧类的接口，但是还有它自己的方法，所以实际上你不能说它们完全相同。举空调为例。假设你屋子已经有了一个制冷控制系统；也就是说它有一个制冷的接口。假设空调坏了，于是你把它换成了一个既能制冷也能制热的热泵。热泵“像”一个空调，但是它的功能更多。由于房子的设计只支持制冷，因此它只能同新对象的制冷部分通信。新对象的接口已经扩展过了，而现有的系统对此一无所知，它只知道原先的接口。



当然看到这个设计图之后，一切就变得很清除了，基类“cooling system”没有包括制热，因此还不够通用。把制热再加进去之后，替换原则就能生效了。当然，这个图只是对真实设计所会发生的情况举个例子。

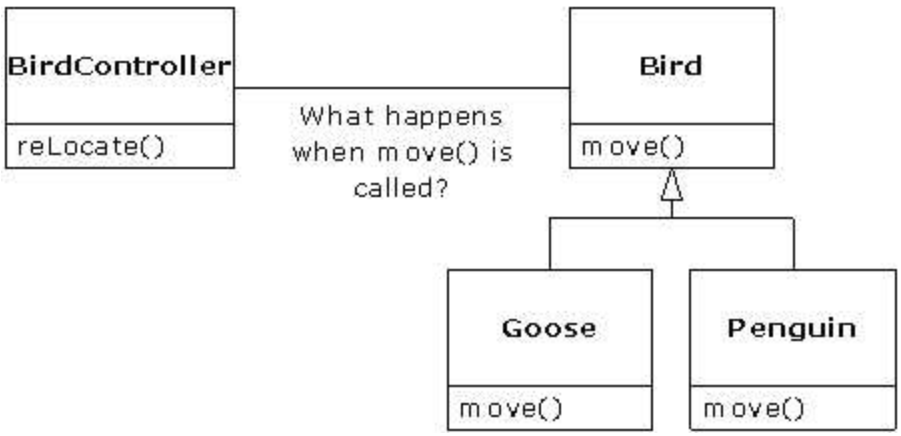
当你看到替换原则时，你会认为这种方法(完全替换 pure substitution)应该是你的不二之选。实际上如果这种设计能行得通的话，这个方法确实很好。但有时你也会发现，往派生类的接口里加新的方法也一样可以做得井井有条。仔细观察之后，该用哪种，不该用哪种，其实是很明显的。

可凭借多态性相互替换的对象

处理类系的时候，通常你用不着把它当作某个具体的类型，只要把它当作基类对象就可以了。由此你可以写出不依赖某个具体类型的代码。以 shape 为例，它的方法是处理泛型 shape 的，所以不用去管它是圆，矩形，三角或是什么还没定义的形状。所有的 shape 都可以被画出，擦除，移除，所以可以把这些消息传给 shape 对象；它们才不担心对象是如何处理这些消息的。

加入新的类型不会影响这些代码，而加入新的类是扩展面向对象的程序，使之处理新的情况的最普通的方法。比如，你可以从 shape 派生出一个叫 pentagon 的新子类，而不用去修改处理泛型的 shape 的方法。这种通过派生子类来扩展设计的简单方法是封装修改的基本方法之一。这在降低软件维护成本的同时大大增强了设计。

然而要把泛型的基类当作派生类来用(用 shape 来指 circle，vehicle 来指 bicycle，bird 来指 cormorant 鸬鹚时)还有个问题。如果某个方法要告诉泛型的 shape 来画它自己，泛型的 vehicle 转弯，或是泛型的 bird 移动，编译器编译的时候就没法知道应该执行那段代码了。这就是问题之所在——当消息发出之后，程序员不想知道哪段代码该执行；draw 方法能同等的作用于圆，矩形或是三角形，而对象则根据其自己的具体类型来执行适当的代码。如果你无需知道该执行那段代码，那么加入新的子类之后，你也无须改变要调用的方法就能让它执行不同的代码。但是如果编译器不知道该执行那段代码，那么它又作了些什么呢？比如，在下面的图中，**BirdController** 对象只控制泛型的 **Bird** 对象，根本不过问它的具体的类型。从 **BirdController** 的角度而言，这么做很方便，因为用不着写特别的代码来判断这到底是哪种 **Bird**，或者这只 **Bird** 有什么行为。所以在忽略 **Bird** 的具体类型的情况下调用 **move( )**的时候，这个调用会如何工作呢，又会发生些什么行为呢(**Goose** 鹅可以跑，飞或是游泳，而 **Penguin** 企鹅只能跑或是游泳)？



答案就是 OOP 最主要的魔法：编译器不用传统的方式进行函数调用。非 OOP 的编译器的做法被称为前绑定(*early binding*)。如果你从没想过这个问题的话，可能听也没听说过这个术语。它的意思是编译器会产生那个名字的函数的调用，而连接器负责将这个调用解析成须执行的代码的绝对

地址。在 OOP 中，不到运行的时候，程序没法确定代码的地址，所以向泛型对象发送一个消息的时候，就要用到一些特别的手段。

为了解决这个问题，OOP 语言是用了后绑定(*late binding*)的概念。当你向某个对象送了一个消息之后，不到运行时，系统不能确定到底该调用哪段代码。编译器只保证这个方法存在，并且检查参数和返回值的类型(不这么做的语言属于弱类型 *weakly typed*)，但是它并不知道具体执行的是哪段代码。

要进行后绑定，Java 用了一些特殊代码来代替绝对调用。代码用存储在对象中的信息来计算方法的地址(第 7 章会详细讲解这个过程)。因此每个对象的运行方式会根据这段特殊代码的内容而改变。当你向那个对象发送一个消息时，对象实际上知道该如何处置。

在有些语言中，你必须明确申明，某个方法要用到后绑定的灵活性(C++ 用 **virtual** 关键词)。在这些语言中，方法“不是”默认地动态绑定的。而动态绑定是 Java 的缺省行为，因此无需添加什么额外的关键词就能获得多态性。

还是用 shape 举例。前面已经图解了一个类系 (family class，所有的类都具有同一个接口)。要展示多态性，我们要单独写一段会忽略类型的具体信息，而只跟基类打交道的代码。这段代码会同与具体类型信息相分离，因此它写起来会比较简单，理解起来也比较容易。如果通过继承加入了一个新的类——比如 **Hexagon**，那么在这个新类上，你写的代码会同在已有的类上运行得一样好。因此，这个程序是可扩展的。

如果你用 Java 写了个方法(很快就会知道该如何写)：

```
void doStuff(Shape s) {
  s.erase();
  // ...
  s.draw();
}
```

这个方法可以同任何 **Shape** 交谈，所以它是同要 draw 和 erase 的具体类型相独立的。如果程序的其它部分用到了 **doStuff( )**方法：

```
Circle c = new Circle();
Triangle t = new Triangle();
Line l = new Line();
doStuff(c);
doStuff(t);
doStuff(l);
```

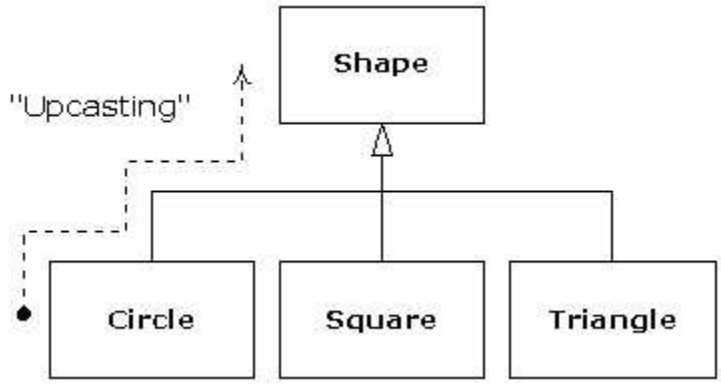
不论对象确切类型是什么，**doStuff( )**都能自动地正确工作。

这是个很精彩的手法。看看下面这行：

```
doStuff(c);
```

这里做的是把 **Circle** 传给了一个在等 **Shape** 的方法。但是由于 **Circle** 就是 **Shape**，所以它可以让 **doStuff( )**把它当作 **Shape** 来用。也就是说，**Circle** 可以接受任何由 **doStuff( )**传给 **Shape** 的消息。所以这么作是很安全的，也是很合逻辑的。

我们把这种将派生类当作它的基类来用的过程称为上传(*upcast*)。在英语里 *cast* 是指把东西浇注到模子里(译者注：其实 cast 在英语里也有扔，抛的意思)，而 *up* 是由继承关系图的布局得来得。通常基类总是放在顶部，派生类像扇子那样列在下面。于是把派生类传给它的基类在继承关系图里是向上，因此就有了：“upcasting”。



面向对象的程序到处都有类型转换(casting)，因为这么作之后，你就可以不必知道所处理的对象的具体类型了。看看 **doStuff( )**的代码：

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

要知道，它可没说“如果你是 **Circle**，必须这么作，如果你是个 **Square**，就那么做，等等。”要是你写那种程序的话，就得检查所有 **Shape** 的可能的具体类型了，那么这个程序就会变得一团糟，而且每次加上一种新的 **Shape**，你还得修改代码。这里，你只是说，“你是一个 **Shape** 对吧，好我知道你可以 **erase( )**和 **draw( )**，就这么做，照顾好自己。”

doStuff( )的代码里面最令人印象深刻的是，它会想方设法执行正确的代码。调用 **Circle** 的 **draw( )**所执行的代码会同调用 **Square** 或 **Line** 的不同，而当 **draw( )**消息传给一个泛型的 **Shape** 的时候，它会

根据其具体的类型作出正确的反映。这真是难以置信，因为我们前面已经讲过了，当 Java 编译器编译 **doStuff( )**的时候，它不知道要处理哪个具体的类型。所以，通常你会认为它会停在调用基类 **Shape** 的 erase( ) 和 draw( )，而不去管它到底是 **Circle**，**Square** 还是 **Line**。但是有了多态性，它就能正确地工作了。编译器和运行时系统会处理所有的细节；现在你要知道的就是它真的发生了，但是更重要的是，应该如何把它用到设计中去。当你向对象发送消息的时候，即便是上传之后，这个对象也会作出正确的反映。

abstract 基类和 interface

通常在有些设计方案中，你只想让基类去表示由其派生出的类的接口。也就是你不想让人创建一个基类的对象，而只是想把对象上传给它，以期能使用这个类的接口。这样你就需要使用 **abstract** 关键词来把这个类做成抽象类。编译器会阻止任何人创建 **abstract** 类的对象。这是个用来强化设计的工具。

abstract 关键词也可以用来表示这个方法还没有实现——它就像是一个声明“由这个类所派生出的所有的类都有这个方法，但是它的实现不在这里”的存根。**abstract** 方法只能存在于 **abstract** 类里。如果有个类继承了这个 **abstract** 类，那么它要么实现这个方法，要么也是一个 **abstract** 类。**abstract** 方法能让你将方法放到接口而不必为它写一些无意义的代码。

interface 关键词更是对 **abstract** 类的概念的深化，它不允许你实现任何方法。**interface** 是个很趁手也很常用的工具，因为它能彻底地将接口与实现分离开来。此外如果你愿意，还可以继承多个接口，因为要继承多个常规或是抽象类是不允许的。

对象的创建，使用和生命周期

从技术角度而言，OOP 只包括抽象的数据类型，继承和多态，但还有一些问题，它们的重要性不亚于上述的几点。这一节会讨论这些问题。

关于对象，最重要的问题就是，它是如何创建的，又是如何消亡的。用来表示对象的数据是存储在哪里的，又该怎样控制其生命周期？对于这些问题，有很多不同的思路。C++的做法强调效率是第一位的，所以它让程序员自己去选择。为了获得最快的运行速度，我们可以在写代码的时候决定将对象放到栈里(有时会被称为 *automatic* 或 *scoped* 的变量)或是静态的存储区域。这个方法把分配和释放存储空间的速度放在第一位，这一点在某些场合下是很有价值的。然而这么做的代价就是牺牲了灵活性，因为在写代码的时候你必须知道对象的确切数量，寿命以及类型。如果你要解决诸如计算机辅助设计，仓库管理，航空管制之类的较为一般化的问题，这就显得太过掣肘了。

第二种方法是在一个被称为堆的内存池里动态地创建对象。在这个方法下，只有到了运行时，你才知道对象的数量，寿命以及其确切的类型。它们都是在程序运行的一刹那决定的。如果你要一个新的对象，可以在要用到它的时候，在堆中直接创建一个。(在栈中分配存储空间通常只需一个汇编指令，把栈指针向下移就行了，想把指针指回来也只要一条指令。而堆的存储分配则取决于存储机制的设计。)这种动态的方法是建立在这样一个合理的假设的基础上的：对象都比较复杂，所以与创建对象相比，分配空间，释放空间这点开销不会对性能造成很大的影响。此外，更高的灵活性对于解决更为通用的编程问题也是很重要的。

Java 只使用第二种方法。[7]每次创建对象的时候，你都要用 **new** 关键词来创建一个那类对象的动态的实例。

然而还有一个问题，那就是对象的寿命。对于那些在栈里创建对象的语言而言，编译器会判断对象的寿命有多长，并且进行自动的清理。然而，要是对象创建在堆中，编译器就不知道它的寿命有多长了。在像 C++之类的语言里，你必须用代码来决定什么时候该清除对象，如果做错了的话，还会导致内存泄漏(对于 C++程序来说，这可是常见病多发病)。Java 有一种被称为垃圾回收器(*garbage collector*)的特性。它会自动发现那些已经没用的对象，并且把它给清除掉。垃圾回收器真的很方便，因为它能让你少写很多代码，能帮你跟踪很多问题。更重要的是，它提供了一种更高水平的保障，能防止内存泄漏这种阴险的问题(很多 C++的项目就是绊倒在它的脚下)。

Collection 和迭代器

如果你不知道解决某个问题需要多少对象，或者它们的寿命该有多长，那么你也就没法知道该如何存储这些对象了。你怎么知道需要多少空间来创建这些对象？你没法知道，因为这些东西不到运行的时候是没法知道的。

在面向对象的设计中，绝大多数的解决方案看起来都很贫：创建一个另一种类型的对象。这个新的对象会持有其它对象的 reference。当然你也可以使用绝大多数语言都有的数组。但是这种通常被称作容器(*container* 也被称为 *collection*，但是 Java 的类库赋予这个术语其它意义，所以本书使用“容器 container”)的新对象可以根据需要扩容，以便让你放进所有的东西。所以你无需知道要向容器里边放多少东西。你要做的只是创建一个容器对象，然后它会自己照料一切。

幸运的是，一个好的 OOP 语言都跟着一套容器类。对于 C++，这部分就是标准 C++类库(Standard C++ Library)，有时它也被称为标准模板类库(Standard Template Library 缩写是 STL)。Object Pascal 在它的 Visual Component Library(VCL)中也有容器。Smalltalk 有一套非常完整的容器类。而 Java 的标准类库也有容器类。有些类库认为泛型容器已经能很好的满足所有的需求，而另一些(Java 就属于这类)则为不同的任务提供了不同的容器：好几种 **List** 类(以持有线性序列)，**Map** 类

(也称为关联性数组 *associative arrays*，将一个对象同另一个对象关联起来)，以及 **Set** 类(不持有两个相同的对象)。有些容器类还包括队列，树，栈等。

所有容器都能让你放东西进去，拿东西出来；通常有将元素放入容器的方法以及将元素取出来的方法。但是将元素取出来可能会有一些问题，因为如果一个方法每次只能选一个对象，那么它的功能太弱了。怎样才能操控或者比较容器中的一组元素，而不只是一个呢？

解决方法就是“迭代器(*iterator*)”。它也是一种对象，其功能就是在容器中选取一个元素，并把它交给迭代器的用户。作为一个类，它也提供了某种抽象。这种抽象可以将容器的细节与访问容器的代码分隔开来。通过迭代器，容器被抽象成了一个简单的线性序列。迭代器能让你在遍历这个线性序列的时候不用去关心它到底用了什么结构——就是说，你不用去管它到底是 **ArrayList**，**LinkedList**，**Stack** 还是其它什么东西。它能让你在不改动程序代码的前提下，修改其背后的具体数据结构。起初(在 1.0 和 1.1 版中)，Java 用一种叫 **Enumeration** 的标准迭代器来为所有容器类服务。Java 2 新增加了一个更为完整的容器类，其中的迭代器也换成了 **Iterator**。它的功能较老的 **Enumeration** 有了增强。

从设计的角度来说，你要的只是一个能够操控的，能用来解决问题的线性序列。如果一种线性序列就能满足你的全部要求，那么你就没理由再要其它的。有两个原因会促使你去挑选容器。首先，容器能提供不同的接口和外部行为。栈(stack)的接口和行为，就同队列(queue)的不同，而它们又与 **Set** 和 **List** 的不同。或许有一种容器能比其它容器提供更为灵活的解决方案。第二，不同的容器在进行同一种操作的时候，会有效率上的差异。最好的例子就是比较两种 **List**：**ArrayList** 和 **LinkedList**。两者都是简单的线性序列，都具有相同的接口和外部行为。但是在做某些操作的时候，它们的效率会显得天差地别。对于 **ArrayList**，随机访问是一种时间恒定的操作，不论你访问哪个元素，所需的时间是相同的。然而对于 **LinkedList**，随机访问和选取元素的代价会很大，但是另一方面，如果你要在这个序列中间插入元素的话，**LinkedList** 的效率会比 **ArrayList** 的高出许多。所有这些差异都是源于其背后所使用的数据结构。在设计阶段，你可以用 **LinkedList**，到了性能调整的阶段，就可以把它改成 **ArrayList**。由于用了 **List** 的基类和迭代器作了抽象，你就能以最小的代价来完成这种转换了。

单根继承体系

是否所有的类都应该归根结蒂继承同一个根类？自从 C++率先提出这个问题之后，它在 OOP 的诸多问题中变得更突出了。Java(实际上除了 C++所有其它 OOP 语言)的回答是，是的，而这个最终的基类就叫 **Object**。而且事实证明单根的继承体系有很多好处。

在单根继承体系中，所有对象都有共通的接口，所以它们最终都属于同一种基本类型。不然的话(也就是在 C++中)你就不知道两个对象是不是属于同一个基本类型了。从向后兼容的角度看，这样做能更好的适应 C 的模式，而且限制较少，但是当你真的要完全按照面向对象的范式进行编程的时候，为了获得其它 OOP 语言那样的便利，你就必须搞一套自己的类系。此外，你拿到的新类库还有可能会用到一些不兼容的接口。要把这些新的接口融入到设计之中，所需的工作量会非常大(可能会用到多重继承)。C++的这种额外的“灵活性”是否值得？如果你真的需要——假设你有很多 C 的代码——那它还是很有价值的。但是，如果你是从零开始的，那还是选择 Java 吧，它能提供更高的编程效率。

在单根继承体系中(比如 Java)，可以保证所有的对象都有某种功能。你知道，系统中的每个对象都能进行某些基本操作。单根继承体系，以及在堆中创建所有对象，这两者大大简化了参数的传递(在 C++里面，这可是一个很复杂的专题)。

单根继承体系还使得垃圾回收器的实现变得更为容易(Java 附带了一个，用起来很方便)。所有必要的支持都被作进了基类，于是垃圾回收器就能向系统里的每个对象发送适当的消息。如果没有单根继承体系，或者系统不是通过 reference 来操控对象的话，要实现垃圾回收器会很难。

由于所有对象都保证能提供运行时的类型信息，所以你不会碰到不能确定对象类型的情况。这一点对于异常处理这类系统级操作来说是非常重要的。此外，它还能让你在编程时获得更大的灵活性。

下传与模板/泛型

Java 的容器要能起作用，就得能持有一种万能的类型：**Object**。单根继承体系意味着所有东西都是 **Object**，所以持有 **Object** 的容器能持有任何东西。[8]这使得容器的复用变得很简单。

使用容器的时候，只要把对象的 reference 直接放进去，用的时候再请出来就是了。但是，由于容器只能持有 **Object**，所以当你往容器里边放 reference 的时候，它就被上传给了 **Object**，于是就把自己的身份给丢了。当你把它提出来之后，你得到的是 **Object** 的 reference，而不是那个你放进去的东西。所以如何才能把你放进去的东西原封不动的变回来，怎样才能用到这个对象放进容器时候的接口呢？

这里又要用到类型转换了，但这次不是顺着继承关系上传给一个更一般的类型。相反，你得沿着继承图向下传给一个更为具体的类型。这种方式称为下传(*downcast*)。对于上传，打个比方，你知道 **Circle** 是 **Shape** 型的，所以上传很安全，但是你不一定知道这个 **Object** 是不是一个 **Circle** 或 **Shape**，所以除非你对那些的类型有十足的把握，否则下传十有八九是不安全的。

然而，也不是说这样作就一定很危险。因为如果下传出错的话，你就会得到一个叫做异常(*exception*)的运行时错误。我们很快就会讲什么是异常。所以只懂如何从容器中提取出对象的 reference 还不够，你还必须记得它们是什么类型的，这样才能进行正确的下传。

下传所需的运行时检查会引起程序运行效率的降低，同时也加重了编程的负担。为什么不创造一种知道它所存取的对象的类型的容器呢？这样就能让下传变得多此一举，并且同时消除出错的隐患。解决方案就是被称为参数化类型(*parameterized type*)的机制。参数化类型是一种能够根据需要由编译器自动指派类型的类。例如，对于参数化类型的容器，编译器可以将它定制为只能存取 **Shape**。

参数化类型是 C++的重要特性，部分原因是因为它没有一个统一的继承体系。在 C++中，参数化类型是由"template"关键词实现。目前 Java 没有参数化类型，可能是因为设计师们认为，用单根继承体系可以做到这个效果——它们也不想想，这做起来有多滑稽。不过，Java 已经有了一个参数化类型的提案，其语法同 C++的 template 惊人的类似。所以我们可以期待下一版的 Java 会提供参数化类型的支持(其正式命名将是 *generics*)。

确保正确地清除

为了生存每个对象都需要资源，最明显的就是内存。当对象不再需要时，就应该将其清除，这样才能释放它所占用的资源，以供别人使用。在简单的编程情况下，如何清除对象并不是什么大问题：你创建对象，需要的时候就用，用完就清除。然而实际情况往往没有这么简单。

假设，你在为机场设计一个空中管制系统。(也可以是管理仓库里的板条箱，录像带租赁系统，甚至是关宠物的笼子。)初看起来这很简单：创建一个容器来保存飞机，然后每次有飞机进入这个空管区域的时候，创建一个新的飞机，把它放进容器里。清除的时候，只要飞机飞离了空管区域，直接把那个飞机给删了就是了。

但是，也许还要记录一些供其它系统使用的飞机数据；或许这种数据不像供主控程序用的那样急。可能这是一个所有从这个机场起飞的小飞机的清单。所以你需要第二个容器来保存小飞机，而且当你创建飞机对象的时候，如果它是小飞机，你还要把它放进这第二个容器中。然后，空下来的时候，后台进程就能处理这个容器中的对象了。

现在这个问题就复杂多了：你怎样才能知道什么时候该清除对象？可能这部分程序已经用完这个对象，而那部分程序还没有。很多情况下，很多编程系统中(比如 C++)都会遇到这个问题。它们都会让你在用完这个对象之后，明确地把它给删了。因此事情就变得复杂了。

Java 设计了一个垃圾回收器来处理释放内存的问题(仅此而已，不包括清除对象的其它方面)。垃圾回收器会“知道”这个对象是不是还有人要用，并且会自动释放那些已经“没用”的对象所占据的内存。这点(加上所有对象都是继承自单根的 **Object** 类，以及你只能用一种方法来创建对象——在堆中)使得用 Java 编程比用 C++简单了许多。你要作的决策和要解决的问题都会少得多。

垃圾回收器的效率与灵活性

如果这么做只有好处没坏处，那为什么 C++不弄一个？实际上所有能简化编程的东西都是有代价的，这个代价就是代码的执行效率。正如前面提到的，C++能在栈里创建对象，这时这些对象的清理是自动的(但是你也失去了在程序运行期间，按照需要创建对象的灵活性)。在栈里创建对象是分配和释放内存的最高效的方法。在堆里创建对象的代价就比较高了。此外，单根继承体系和多态性的支持都是要收“买路费”的。但是垃圾回收器最特殊的一点还在于，你永远都不知道它会在什么时候启动，会运行多长时间。也就是说 Java 程序的执行速度是不一致的，因此它不适宜用于某些特定的情况下，比如要求程序能一贯高效运行的环境。(这些程序通常称为实时程序，虽然它们所解决的问题并不都是那么的迫切。)

C++的设计人员极力迎合 C 的程序员(在这点上格外成功)，为了不让程序员们能有借口回过头去用 C，他们拒绝往 C++里添加进任何会影响速度，或是限制其用途的特性。这个目标实现了，但是代价是用 C++编程的时候复杂性会高出许多。Java 比 C++简单多了，但是在效率和适用性方面打了折扣。不过对于绝大多数编程问题，Java 仍然是优先的选择。

异常处理：与错误打交道

自从有编程语言的那一天起，错误处理就是其最难的问题之一。然而要设计出一个好的错误处理方案实在是太难了，所以许多语言干脆就不作考虑，他们把难题留给了类库设计人员。而类库设计人员，常常会搞出一个中途半端的解决方案，它能解决一些问题，但是却漏洞百出，所以通常他们也把这个问题给忽略了。绝大多数错误处理方案都有一个大问题，那就是它不是由语言强制实施的，而是要由程序员们去主动遵守某种约定。如果程序员们不理会——如果开发任务很紧的话，这是常有的事——这个方案就成了废纸一张。

异常处理通过同编程语言，有时甚至是操作系统直接对话，来解决“错误处理”这个问题。异常是一种能从错误发生的地方被“抛出”的对象，它会被那个能解决这类问题的“异常处理程序”所“捕获”。看上去，异常处理就像是程序在出错的时候所采用的一条平行的执行线路。由于不必时常去检查错误条件，编程变得更容易了。此外，异常还同那些为了标识错误条件而由方法返回或设定的值不同，这些值是可以忽略的，但异常却不行。所以异常肯定能在什么地方得到处理。最后，异常还提供了一种能可

靠地从错误状态下恢复的方法。相比退出程序，大多数情况下你都可以将状态重新设置正确，然后让程序恢复运行，这样就能编写出更为鲁棒的程序。

Java 的异常处理在所有编程语言中显得那样地突出，这是因为 Java 的异常处理是从底层开始的，而且还不能不用。如果你不编写正确地处理异常的代码的话，编译就通不过。这种有保障的一致性有时会使错误处理变得更简单。

值得注意的是，尽管在面向对象的语言中异常通常以对象的形式出现，但异常处理并不是一种面向对象的特性。异常处理在面向对象的语言问世之前就已经有了。

并发

计算机编程有一个基本思想，就是要让程序能同时处理多个任务。很多编程问题都要求程序能停下它正在做的事，去处理一些别的问题，然后再返回主进程。要做到这点有很多办法。起初，程序员们借助对于机器底层知识的了解，通过硬件中断来编写挂起主进程的中断服务的程序。尽管这么做效果很好，但是程序很难写而且也没法移植，因此要把这类程序弄到新的机器上，工期会很长，代价也很高。

有时为了处理紧急任务必须得使用中断，但是很多情况下，你会把问题分成单独运行的片断，这样整个程序就能反应得快些。这种在程序内部单独运行的片断被称为线程，而这一概念被通称为并发(concurrency)或多线程(multithreading)。多线程用途方面举得最多的一个例子就是用户界面。通过使用多线程，用户按完按钮之后就会很快得到回应，而不必干等着程序做完当前的任务。

通常，线程只是一种分配单个处理器时间的方法。但是如果操作系统支持多处理器，那么线程也可以被分配到不同的处理器上，这样它们就能真正地并行运行了。在编程语言级别上的提供多线程支持所带来的好处就是，程序员不用操心运行这个程序的机器有多少 CPU 了。程序被逻辑地划分成线程，如果机器上有多个处理器，无需任何调整，程序就能跑得更快。

这使得线程听起来很简单。但是这里有个陷阱：共享的资源。如果多个正在运行的线程都要访问同一项资源，那就有问题了。举例来说，两个进程不能同时往一台打印机上送任务。要解决这个问题，可共享的资源，比如打印机，在用的时候就必须上锁。所以线程会先锁上一个资源，完成任务之后再解锁，这样其它人才能接着使用这项资源。

Java 语言内置了多线程的支持，这使得这个复杂的课题变得简单了许多。多线程的支持是在对象级别，因此线程就被表示成对象。此外，Java 还有一定的资源锁定的功能。它能锁定任何对象的内存(毕竟这也是一种共享资源)，这样同一时刻就只有一个线程能够访问这些内存。这是

由 **synchronized** 关键词来做的。其它资源就得靠程序员自己来锁定了。通常可以创建一个表示这项资源的对象，然后让线程在访问资源之前先检查一下。

Persistence

对象创建之后，只要还用得着，它就一直还在。但是一旦程序运行结束，它就再也不能存在下去了。初看上去这是很符合逻辑的。但是在某些情况下，如果对象能永续生存，能在即使是程序不运行的时候也保存着信息，那将会非常有用。这样，当你再次运行程序的时候，对象还在那里，仍然保存着和它上次运行时相同的信息。当然，你能通过把信息写入文件或数据库来达到相似的效果，但是本着“万物皆对象”的思想，如果声明对象是 persistent 的，它就会为你自动照料一切，那岂不是相当方便吗？

Java 提供了“轻量级 persistence”的支持，也就是说你能很轻易的将对象保存到磁盘上，供今后恢复之用。之所以称其为“轻量级”是因为，在存储和提取过程中，你还必须明确地进行调用。轻量级的 persistence 既可以通过对象的序列化(*object serialization* 第 12 章讲)，也可以通过 Java 数据对象(*Java Data Objects* 简称 JDO，在 *Thinking in Enterprise Java* 讲)来实现。

Java 和 Internet

可能你会问，如果 Java 只是一种新的计算机编程语言的话(实际上这话也没错)，它为什么会那么重要，为什么会被拔高到“计算机编程领域的革命性的进步”，这个高度。如果你是从传统编程的立场上来看这个问题，也许答案还不是那么有说服力。尽管在解决传统的，孤立的编程问题方面，Java 也是很能干的，但是真正让它脱颖而出的，是因为它能解决在万维网上编程的问题。

Web 是什么?

刚开始的时候，Web 看上去很神秘，大家都在谈冲浪、在线、主页什么的。要把讲 Web 讲清楚，最好是退回来从头开始。但是这么做，先得理解客户/服务器系统。这是计算机技术的另一个领域，里面也有大把让人头晕的问题。

客户/服务器系统

客户机/服务器(client/server)系统的主要思想是，你有一个中央信息库(central repository of information)——通常是保存在数据库中一些信息——要根据需要，把它们分配给某些人或机器。客户机/服务器系统的关键在于，信息库会集中管理信息，因此信息的修改能够传播到用户那里。信息库，分发信息的软件，以及存储信息和软件的机器合起来称为服

务器。存储在远程机器上的软件会同这个服务器通讯，提取信息，处理信息，并且在远程机器上显示结果。这被称为客户。

这么看来，客户/服务器计算的基本概念没那么复杂。但是，当你试图用孤零零的一个服务器来为很多客户服务的时候，问题就来了。这个架构通常都会牵扯到数据库管理系统，所以为了优化应用，设计人员会去“平衡”数据的格式。此外，通常系统还允许客户往服务器里插入新的数据。这就意味着你必须保证一个客户的新数据不会和另一个客户的新数据搅在一起，以及数据不会在添加的过程中遗失(这被称为事务处理*transaction processing*)。当客户端的程序修改之后，还必须重新编译，调试并且安装到客户机上，这要比你相像的复杂昂贵得多。而且如果要支持多种机器或操作系统的话，事情会更麻烦。最后还有一个最重要的性能问题：服务器可能会同时响应成百上千个客户，所以再小的耽搁都是很要命的。为了把延时降到最低，程序员们尽量减轻服务器的负载，通常会把这些处理任务挪到客户端，不过有时也会移到所谓的中间件(*middleware*)的服务器上。(中间件也被用来增进系统的可维护性。)

分发数据这个简单的思想竟然会引出这么些复杂层次，而所有这些问题看起来都像是根本不可能解开的谜。但还有更重要的：大约有一半的开发项目都是基于客户/服务器架构的。它们包括像接收订单，信用卡交易以及分发各种各样的数据——股票市场的，科研的，政府部门的，只要你能叫的上名字的。以前我们的作法是为每个问题设计不同的解决方案，每次都发明一种新方法。这种项目开发起来难，用户用起来也不方便，它们必须适应新的界面。客户/服务器架构这个问题必须要能在总体上解决。

把 Web 当作巨型的服务器

Web 实际上就是一个巨型的客户/服务器系统。实际上还更差一点，因为所有的服务器和客户机是共存在同一个网络上的。不过这点你并不知道，因为你只关心是不是能连到那台服务器，并且对它进行操作(尽管你可能得先在什么地方找到那台服务器)。

最初这只是个单向过程。你向服务器提请求，它交给你一个文件，然后你用本地机上的浏览器(也就是客户)来解释这个文件并且为它重新排版。但没过多久，人们就不满足于仅仅从服务器收发文件了。他们需要完整的客户/服务器功能，所以客户也能向服务器发送信息了，比如查询服务器端的数据库，向服务器添加新的信息，或者下定单(这项任务所要求的安全性要比系统原先能提供的要高得多)。这些就是我们在 Web 的发展历程中亲眼目睹的变化。

Web 浏览器是一项巨大的进步：它的思想是要让同样的信息以通常的形式显示在所有的机器上。然而浏览器还是太原始了一些，而且也很快被加在它身上的任务给拖垮了。它的互动性不好，而且所有需要编程解决的任务都要交到服务器上去处理，所以经常会把服务器和 Internet 给堵了。有时可能会花几秒钟，甚至是几分钟，才会发现提交的请求里面有一个拼

写错误。由于浏览器只是用来显示，不能承担哪怕是最简单的计算任务。(另一方面这样也很安全，因为它不会在你的本地机上执行可能包含 bug 或病毒的程序。)

为了解决这个问题，人们用了很多办法。开始是升级图形标准，让览器器能显示效果更佳的动画和视频。但是有些问题，只能通过让客户端的浏览器运行程序来解决了。这被称为客户端编程。

客户端编程

Web 最初的服务器一浏览器设计提供了互动内容，但是这种互动性是完全建立在服务器之上的。服务器为客户端提供静态页面，而浏览器只是简单的解释页面，然后显示出来。*HyperText Markup Language* (HTML)包括了基本的数据采集的功能：输入框，复选框，单选按钮，列表，下拉式列表，以及只能用于清除表单或是把表单数据“提交(submit)”给服务器的按钮。提交上来的数据会交给 Web 服务器上的通用网关接口(*Common Gateway Interface* 缩写为 CGI)程序。这些文本会告诉 CGI 该作些什么。最常见的就是在服务器上运行一个程序，这个程序一般会放在“cgi-bin”目录中。(如果按完 Web 页面上的按钮之后，你仔细观察浏览器顶部的地址条的话，有时你就会在那些不知所云的东西中间看到"cgi-bin”。)大多数语言都可以写这些程序。Perl 是最常用的，因为它设计的目的就是为了处理和解释文本，所以不论服务器用的是那种处理器，或是那种操作系统，都能安装 Perl。但 Python(我的最爱——见 www.Python.org) 由于其功能强大简单易用，已经对 Perl 的霸主地位发起了挑战。

很多大型网站都是完全建立在 CGI 之上的，而且实际上你能用 CGI 来作几乎任何事情。但是建立在 CGI 之上的网站会很快变得过于复杂而难以维护，此外还有一个相应时间的问题。CGI 程序的相应时间取决于它要发送多少数据，以及服务器端和 Internet 的负载。(而且 CGI 程序本身就启动很慢。)最初设计 Web 的人没能预料到带宽会消耗在这类应用程序中。举例来说，动态图形实际上是不可能一致地运行的，所以服务器会把所有版本的 GIF (Graphics Interchange Format)文件都传到客户端。而且，毫无疑问你肯定作过像填写表格这种简单的事。当你按了发送按钮之后，数据会被送回到服务器，然后服务器启动 CGI 程序，但是发现有个错误，于是重新生成一个 HTML 告诉你出来错，再把这个页面传给你；然后你再重头来一遍。这样作不但慢，而且蠢。

解决方案就是客户端编程。绝大多数运行 Web 浏览器的机器都有强大的引擎，能作很多工作，而且在原先静态的 HTML 方法下，大多数时间，它们都是在傻等着服务器发送下一个页面。客户端编程意味着 Web 浏览器能物尽其用，结果是用户能更快同时也更互动地体验你的网站。

客户端编程所遇到的问题同通用编程的没有什么本质性的区别。参数几乎是相同的，只是平台有写差异；浏览器就是一个小型的操作系统。最后，

你还得写程序，还要解决一串让人眼花缭乱的问题，还要提供客户端编程的解决方案。本节的余下部分会简要介绍客户端编程的问题与方法。

插件

在客户端编程方面最重要的步骤之一就是开发插件(plug-in)。这是程序员就为浏览器提供新功能的办法。它要求用户从网上下载插件程序，然后把它集成到浏览器里。插件告诉浏览器“从今往后，你可以进行这些新的工作了。”(插件只需下载一次。)插件为浏览器添加了一些快速强大的功能，但是写一个插件可不是什么轻松的任务，也不应该是建网站应该作的事。插件对于客户端编程的价值在于，它能让专家级的程序员开发一种新的语言，并且不经开发商的同意，就把这种语言加进浏览器里。于是插件提供了一个能创建新的客户端编程语言的“后门”(尽管不是所有的语言都是通过插件来实现的)。

脚本语言

插件导致了脚本语言的泛滥。你可以用脚本语言把客户端程序的源代码直接嵌进 HTML 页面里去，HTML 页面显示的时候，会自动激活这个插件去解释这些程序的。脚本语言应该比较简单，而且作为 HTML 页面的一部分，它们都是简单的文本，所以会随页面一起获取，而且装载也很快。不过代价就是代码只能给别人看了(或者说偷了)。总之，由于你没法用脚本语言做非常复杂的东西，所以它的难度不会太大。

这一点决定了 Web 浏览器所用的脚本语言实际上是用来解决某些具体的问题的，主要是创建更丰富的，更互动的用户界面的(GUI)。然而，脚本语言也许能解决百分之八十的客户端编程的问题。可能你要解决的问题正好落进这百分之八十的范围，而且由于脚本语言开发起来更简单也更快，所以你应该在选用 Java 或 ActiveX 这类难度更大的方案之前，先考虑一下脚本语言。

最常被提及的浏览器用的脚本语言有 JavaScript(跟 Java 没什么关系；它取这个名字只是想搭 Java 的顺风车)，VBScript(看上去很像 Visual BASIC)，以及 Tcl/Tk，这是一种很流行的跨平台的创建 GUI 的语言。此外还有一些，但是正在开发肯定还要多。

JavaScript 可能是人气最旺的。Netscape Navigator 同 Microsoft Internet Explorer (IE)都内置了这个语言。然而不幸的是这两个浏览器所支持的 JavaScript 在风格上有很大的不同(Mozilla 支持的 ECMAScript 或许有一天能成为普遍支持的标准。可以到 www.Mozilla.org 去下载这个浏览器)。此外讲 JavaScript 的书可能要比其它语言的都多，而且有些工具还能自动生成包括 JavaScript 程序的页面。但是如果你已经熟悉了 Visual BASIC 或 Tcl/Tk，你还是接着用吧，这要比从头学一种新的语言更有效率。(处理这些 Web 的问题就已经够你忙的了。)

Java

如果脚本语言能解决百分之八十的客户端编程问题，那剩下的百分之二十——那些“真正的硬骨头”该由谁来啃呢？Java 是解决这类问题的常用方法。Java 不仅仅是一个安全的、跨平台的、国际化的、功能强大的编程语言，而且它还在不断的扩展之中。通过补充语言特性和类库，Java 潇洒地解决了很多困扰传统编程语言的问题，像多线程，数据库访问，网络编程，以及分布式计算。Java 能通过 *applet* 和 *Java Web Start* 来进行客户端编程。

Applet 是一个只能运行在 Web 浏览器里的小程序。Applet 作为 Web 页面的一部分自动下载(就像图片一样自动下载)。Applet 被激活后就开始运行程序。这正是它的优点——它提供了一种能让服务器不早不晚，在客户需要的时候自动分发客户端软件的方法。客户会成功地获取最新版的客户端软件，而且还不用安装。由于 Java 的设计，程序员只需要创建一个简单的程序，就能让它自动地运行于任何机器，只要这台机器装上了内置有 Java 解释器的浏览器就行了。(可以有把握的说，绝大多数机器都有 Java。)由于 Java 是一种成熟的编程语言，所以不论是在提交给服务器之前还是之后，你都可以用它来干活。比如，你不应该为了检查日期有没有搞错或其它参数是不是出了错，而把一个表单通过 Internet 给发出去，相反，你应该让客户端的计算机检查完之后再发，这样就不用再等着服务器把图片也传回来了。这样不仅能节省时间，提高反应速度，而且服务器端的网络流量和负载也能降低，由此也能防止整个 Internet 慢下来。

Java applet 比脚本程序有个优点，这就是它是以编译过的字节代码的形式发送到客户端的，因此客户是不能得到源代码的。但是，另一方面反编译 Java applet 也不是一件太难的事。不过保护代码通常不是最要紧的事，还有两个更重要的因素。学过这本书后，你就会知道， Java applet 如果比较大的话，编译后需要更长时间才能下载。而脚本程序作为文本是直接植入 Web 页面中的(通常这样会变得更小一些，而且能减少对服务器的请求)。这对网站的反应速度很重要。还有一个就是非常重要的学习曲线。不论你听过些什么，Java 可不是一种学起来非常简单的语言。如果你是一个 Visual BASIC 的程序员，还是用 VBScript 吧。用它解决问题会更快(前提是你能把用户绑在 Windows 平台)。由于它能解决绝大多数常见的客户/服务器问题，可能你很难找到应该学 Java 的理由。如果你对某个脚本语言已经很有经验了，你应该在下决心学 Java 之前去看看 JavaScript 或 VBScript。可能它们能很方便的满足你的需求，而且你也能很快上手。

.NET 和 C#

就当前而言，Java applet 的主要竞争对手还是 Microsoft 的 ActiveX。不过它要求客户端运行 Windows。但是此后，Microsft 以**.Net** 平台和**C#**语言向 Java 发起了全方位的挑战。**.NET** 平台大致对应于 Java 虚拟

机和 Java 类库，而 **C#**则是依样画葫芦地模仿了 Java。这的确是 Microsoft 在编程语言和编程环境领域所作的最成功的工作了。当然，它们占了能看到 Java 在哪些地方做的好，哪里作得不成功的大便宜。因此虽然它们用的是自己的资源，但还是应该算建立在 Java 之上的。
自**.NET** 发布之后，Java 头一次有了真正的对手。如果一切正常的话，Sun 的 Java 设计者们也会认真研究 **C#**，分析程序员们为什么会离开，而且会以对 Java 作的实质性的改进来回应挑战的。

现在**.NET** 所面临的最大的弱点和问题还是，Microsoft 会不会允许它被“完整的”移植到其它平台上。它们声称没问题，而且 Mono 计划 (www.go-mono.com)还在 Linux 上部分实现了**.NET**。但是除非它能完全实现，而且 Microsoft 还决定不作小动作，否则要赌**.NET** 能成为跨平台的解决方案仍然是有风险的。

要知道更多关于**.NET** 和 **C#**，请参阅由 Larry O’Brien 和 Bruce Eckel 撰写的，由 Prentice Hall 在 2003 出版的 *Thinking in C#*。

安全性

能通过 Internet 自动下载并且运行程序，这听上去就像是写病毒的人做梦都想做的事。当你点击了 Web 站点，除了会自动下载 HTML 页面之外，还会下载很多其它东西：GIF 文件，脚本程序，编译过的 Java 代码，ActiveX 组件等等。有些是无害的；GIF 文件不会作任何坏事，脚本语言由于功能有限也做不了什么。同样 Java 设计的时候就规定，applet 只能在的安全性的“沙箱”之内运行，这样就能防止它向磁盘读写，或是访问沙箱之外的内存。

Microsoft 的 ActiveX 是另一个极端。用 ActiveX 编程和在 Windows 下编程没什么不同——你可以作任何你想做的事情。所以如果你点击了一个会下载 ActiveX 组件的页面，那么下载下来的组件就有可能会破坏磁盘上的文件。当然，你下载到机器上的，能在 Web 浏览器之外运行的程序，也能作同样的事情。长久以来，从 BBS (Bulletin Board Systems) 上下载下来的病毒已经成为一个老问题了，现在 Internet 又加重了这个问题。

看来解决方法是使用“电子签名”，但凭什么要检查程序是谁写的。这是基于这个考虑，之所以会有人写病毒，是因为他们是匿名的，所以如果你把这种匿名性给去掉了，那么每个人就都必须为他的所作所为负责了。这看上去像是个好主意，因为这样一来程序就能多干正事，而且我还觉得那些恶作剧也会少些。但是如果程序里面有一个没发现的会造成破坏的 bug 呢，这还是个问题。

Java 的做法是通过沙箱杜绝此类事件的发生。Web 浏览器里的 Java 解释器会在装载 applet 的时候查出所有不合格的指令。更重要的是，applet 既不能往磁盘上写文件，也不能删除文件(病毒破坏性的重要特征

之一)。通常 applet 被认为是安全的，而且由于事关客户/服务器的可靠性，Java 语言中的所有能被病毒利用的 bug 都会被尽快修复。(值得一提的是浏览器实际上强化了这种安全限制，有些浏览器允许你选用不同的安全级别来访问系统。)

你可能会怀疑这种禁止向本地磁盘写文件的做法是不是有点太过头了。因为，你可能会要创建一个本地数据库，或是保存数据供用户离线使用。最初的观点是，最终大家都会上线去做重要的事情，但很快这种看法就变得不切实际了(虽然，或许有一天低成本的“Internet 家用电器”能满足很大一部分用户的需要)。解决方案是使用“经过签名的 applet(signed applet)”，这就允许用户使用公钥加密算法去检验这个 applet 是不是真的是从它宣称的那个地方来的。经过签名的 applet 仍然可以破坏你的磁盘，但是重要的是你现在可以确定是谁写 applet 的了，因此它们不能干坏事。Java 提供了电子签名的框架，因此如果需要的话，你就能让 applet 走出限制它的沙箱。第 14 章里有一个例子就是讲如何签署一个 applet 的。

此外 Java Web Start 也是一个相对比较新的方法，它能很方便的分发无需在 Web 浏览器中运行的独立的程序。在浏览器里运行程序的客户端编程总有一些问题，而这项技术有望解决这些问题。Web Start 的程序既可以是签名的，也可以是每次都要得到用户许可才能运行那些可能对本地机造成破坏的程序的。第 14 章包括了一个 Java Web Start 的例子，并且作了讲解。

电子签名没有考虑到一个重要的问题，这就是人们上网的频度。如果你下载了一个有问题的程序，并且造成了影响，那么要多久你才会发现这个问题呢？可能是几天，也可能是几个星期。到了那个时候，你如何确定是哪个程序干得呢？电子签名又能帮上什么忙呢？

Internet 和 intranet

在解决客户/服务器编程问题方面，Web 是通用性最好的方案，所以用同一种技术去解决问题的某一部分，特别是公司内部的客户/服务器系统应该是顺理成章的。如果使用传统的客户/服务器的方法，你必须克服客户端计算机种类繁多以及安装新的客户端软件这两个难点，但是在 Web 方式下，这两者分别由 Web 浏览器和 Web 的客户端编程解决了。当 Web 技术运用于某个公司的内部信息网络的时候，它就被称为 intranet 了。由于你能够在公司的范围内，物理地控制对服务器的访问，因此 Intranet 的安全性比 Internet 的高了许多。通过培训，一旦员工掌握了浏览器的基本概念，就能很很容易地处理各种不同风格的网页和 applet 了，因此新系统的学习曲线也会比较平缓。

安全问题是客户端编程自然要考虑的问题。而如果程序是在 Internet 上运行的，那么你就不知道它会在哪个平台下运行。此外还必须格外小心，

不要散布有问题的代码。你要的是跨平台而且安全的东西，就像脚本语言或 Java。

如果已经有了 intranet，那么碰到的困难可能就有些不一样了。很可能你的机器是清一色的 Intel/Windows 架构的。如果是 intranet，那么你必须对自己写的代码负责，而且一旦发现了错误，也要由你自己去修改。此外，还可能在原先的客户/服务器系统中还用着一大堆老程序，更烦人的是每次升级还要亲自跑过去安装客户端软件。安装升级程序所浪费的时间可能是你转到浏览器方式的最主要的原因，因为这样一来升级就会在不知不觉中自动完成(Java Web Start 也是一个解决方案)。如果你碰到的是这样一个 intranet，最好还是不要用新的语言去重写程序，使用现有的代码库能帮你节省不少力气，这才是最明智的方法。

当你面对这一系列让人眼花缭乱的客户端编程的解决方案之时，最好的办法还是作一个性价比分析。考虑一下这个问题的约束条件以及最直接的解决方案。由于客户端编程仍然是编程，所以对于特定问题采用最快的开发方法总是无可厚非的。但是从长远考虑，为解决程序开发的问题而做准备，则是一种积极的态度。

服务器端编程

讲到现在我们还没讲服务器端的编程问题。当你像服务器提出请求的时候到底发生了些什么呢？绝大多数时候，这个请求只是简单的“把这个文件发给我。”然后浏览器用适当的方式解释这个文件：把它当作 HTML 页面，图像文件，Java applet，脚本程序等等。更复杂的请求通常会牵涉到数据库处理。最常见的情况是 Web 请求引发了一串复杂的数据库查询，然后服务器把查询结果重新组织成 HTML 文件，再把这个文件送还给浏览器。(当然，如果客户端的 Java 或脚本程序更智能的话，这些数据可以直接传输，然后再在客户端重新组装，这样不仅更快，而且对服务器端的负载也较轻。)或者，当你需要加入某个组或是下订单的时候，你就必须往数据库里写东西了，于是也要修改数据库的数据了。这些数据库的请求必须经由服务器端的程序处理，它们被统称为服务器端编程。通常服务器端程序是用 Perl，Python，C++之类的语言写的 CGI 程序。但是有些系统更为复杂。其中就包括基于 Java 的 Web 服务器，它能让你用 Java，通过编写被称为 servlet 程序的方式，来进行所有的服务器端的编程。Servlet 及其后继者 JSP，是让很多公司转向用 Java 开发网站的两个最主要的原因，特别是它们能解决与能力不同的浏览器打交道时所遇到的问题(这些内容会在 *Thinking in Enterprise Java* 中讲的)。

应用程序

很多关于 Java 的议论都是关于 applet 的。实际上 Java 也是一种通用的编程语言，其它语言能够解决的问题，Java 一样可以解决——至少是理论上如此。而且正如我们先前所指出的，它在解决绝大多数的客户/服务器问题的时候，效能更好。如果你放弃使用 applet(同时也解脱了诸如不

能向磁盘写东西的束缚)，编写通用程序的大门就向你完全敞开了，你可以编写跟普通程序一样能独立运行的，无需 Web 浏览器的程序。在这个领域，Java 的力量不仅体现在其可移植性，而且还在于其可编程性。正如你将在本书中看到的，相比以前的编程语言，Java 有比更多的许多能让你在段时间内创建健壮程序的特性。

要知道这是件利弊参半的事情。你用更慢的运行速度来换取这些改进(虽然在这方面还正做着一些很重要的工作——特别是最近几版 Java 里面的叫做“hotspot”的性能提升技术)。就像其它语言，Java 也有其固有的局限性，因此它可能不适于用来解决某些编程问题。然而 Java 还是一个正在快速发展中的语言，随着每一个新版本的发布，它在解决大型问题方面正变得越来越具吸引力。

Java 为什么能成功

Java 之所以能如此成功是因为它是针对当今程序员所面临的很多问题而设计的。Java 最重要的一个目标是为了提高编程的效率。效率的提高有多方面的因素，但是这个语言的设计目的就是为了超越在它之前语言，并且为程序员提供更大的便利。

系统能更易于表述和理解

根据问题设计出来的类能把问题讲得更清楚。这就是说，当你编程的时候，你是在用问题空间的术语 (“把铁环放到箱子里去”)，而不是用解决空间里的计算机的术语(“把这位设成 1 表述中继关闭了”)，来描述解决方法。你用了更高层次的思想来处理问题，于是能用一行代码完成更多的工作。

表达简单的另一个好处是便于维护， (如果报告是可信的话)维护阶段的成本在整个软件生命周期中，占了非常大的比例。如果程序易于理解，那么也易于维护。这也有助于创建和维护文档。

最大程度上利用类库

创建程序的最快的办法是使用已经写好的代码：类库。Java 的主要目标就是让类库更易于使用。这一点是通过将类库转化成新的数据类型(类)来做到的，这样引入一个类库就意味着往语言中加入了新的类型。由于 Java 的编译器会负责管理类库的使用——保证正确的初始化和清除，保证方法能被正确的调用——你要留意的是让类库去干什么，而不是应该怎么干。

错误处理

C 语言的错误处理可以说是一个臭名昭著并且常被忽略的问题——与其用它还不如双手合十，祈求上帝。如果你正在开发一个大型的，复杂的程序，那么最糟糕的莫过于程序里边藏着一个不为人知的错误。当它跳出来的时候，你什么线索也没有。Java 的异常处理是一种能保证让你注意到错误的机制，一旦发生了错误，它就会有所行动。

编写大项目

很多传统语言都有固有的程序长度和难度的限制。举例来说，如果要很快地凑出一个解决某些问题的方案的话，BASIC 会是一个很好的选择。但是如果程序有好几页长，或者问题的难度超出了这个语言通常能解决的限度，这个任务就不那么愉快了。语言不会告诉你，是它让你陷入失败的境地的，即使有你也不会理会的。你不会说，“我的 BASIC 程序太长了，我应该用 C 来重写一遍”。相反你会想方设法再塞几行代码进去，以期能增加一些功能。这样就招惹了更多的麻烦。

Java 的设计初衷就是要帮助你编写大型程序——也就是说它消除了小程序和大项目之间的复杂度的界限。如果要写“hello, world”之类的程序，当然用不着动用 OOP 的特性，但是这些特性随时都能用。而且不管是大项目还是小程序，在查找可能会产生 bug 的错误方面，编译器是一样的积极。

Java 还是 C++?

Java 看上去很像 C++，所以很自然有人会认为 Java 会取代 C++。但是我对这点表示怀疑。首先，C++仍然有一些 Java 还不具备的特性，而且虽然有很多保证说有一天 Java 会和 C++一样快，甚至更快，但时至今日，我们看到的都只是一些稳步的改进，而不是显著的突破。此外人们对 C++的兴趣并未消退，所以我认为 C++不会很快退出。看起来语言是不会很快消失的。

慢慢的我发觉，Java 的优势的体现在一些与 C++稍有不同的领域。C++是一种没什么固定模式的语言。当然，为了解决某些特定的问题，人们已经用很多方法对它作了改造。有些 C++工具将类库，组件模型，以及代码生成工具组合在一起，把它打造成专门用来开发供最终用户用的，窗口应用程序(为 Microsoft Windows)的工具。但是绝大多数 Windows 的开发人员都用什么呢？是 Microsoft 的 Visual BASIC (VB)。要知道，即使程序的长度只有几页，VB 的代码就已经变得无法管理了(而且它的语法也不尽合理)。因此，即便是像 VB 这样成功，而且流行的语言，也算不上是语言设计的好榜样。如果能有 VB 的功能和易用性，但是却不会像它那样产生无法管理的代码，那就太好了。我觉得这正是 Java 的闪光点：“下一个 VB[9]。”不管你是否对此感到惊讶，先看一看：Java 里边有那么多用来简化应用级问题的方案，比如网络和跨平台的 UI；同时它的语言设计又能让你编写很长，同时又不失灵活性的代

码。再加上类型检查和错误处理，这使得它与绝大多数语言相比，有了巨大的改进。因此，你能显著地提高你的编程效率。

如果你是从零开始开发的，那么与 C++相比，Java 的简单性能让它大大缩短开发周期——非正式的证据 (一些与我有联系的，已经换用 Java 的 C++的开发小组) 表明开发速度是用 C++的两倍。如果 Java 的性能不是问题，或者你能得到补偿，那么在纯粹的用时间换市场的问题上，要不用 Java 而选 C++就比较困难了。

最大的问题是性能。作为解释语言，Java 总是慢的，最早的 Java 解释器运行程序的时候要比 C 的慢 20 到 50 倍。随着时间的推移，这一点已经有了很大的改观(特别是最近几版 Java)，但是性能仍会是一个重要的问题。计算机就是要跑得快；如果用了计算机速度还没有显著的提高，那还不如手工去作。(我曾听说有人建议先用 Java，以缩短开发时间，然后，如果需要执行得更快得话，可以用工具和类库把代码翻译成 C++的。)

让 Java 能用于很多开发项目的关键是，它有了诸如"just-in-time" (JIT)编译器，Sun 自己的“hotspot”技术，甚至是本机代码的编译器这样的性能提升工具。当然，本机代码编译器会破坏 Java 引以为豪的跨平台运行能力，不过它们的运行速度会比较接近 C 和 C++的。而且跨平台编译 Java 程序应该比 C 和 C++的简单了许多。(从理论上讲重新编译一次就行了，不过其它语言以前也曾作过这种承诺。)

总结

本章的目的是要让你对 OOP 和 Java 的诸多要点，包括 OOP 为何如此与众不同，以及为什么 Java 又格外的特出，有一个初步的认识。

不是每个人都会喜欢 OOP 和 Java 的。你应该先评估一下你的需求，然后判断 Java 是不是能满足这些需求，或许你应该选用别的编程系统(包括你正在使用的)。这点很重要。如果你知道，在可预见的将来，你的需求将是很特殊的，而且 Java 无法满足，那么你应该庆幸你没选 Java(我特别推荐你去试试 Python；见 www.Python.org)。即便你最终选择 Java，你也应该了解一下还有什么别的选择，以及你为什么选择这个方向。

你知道过程语言是什么样的：不外乎数据定义和函数调用。要想知道程序在干什么，你必须得花功夫去跟踪函数的调用，以及其底层的意义，这样才能在心中创建一个模型。这就是为什么在用过程语言设计程序的时候，我们需要先建一个中间模型——因为只看源代码会比较吃力，因为它所用的术语，更多的是在为计算机，而不是为正在解决问题的程序员服务的。

由于 Java 往过程语言里面添加了很多新的概念，于是你会想当然地认为 Java 的 **main( )**会比 C 的复杂了许多。这里，你会很高兴也很意外地发

现：一个好的 Java 程序通常会比作同一件事的 C 程序简单许多，也更易于理解。你会看到定义了很多对象，而这些对象则对应着问题空间里的概念(而不是代表计算机的问题)，而发送给这些对象的消息也都代表着问题空间的行为。OOP 编程的惊喜之一就是，要理解程序很容易，读就是了。通常代码也短了很多，因为很多问题都通过使用已有的类库的方法解决了。

---

[2]有些语言设计者认为仅依靠面向对象的编程并不能很容易地解决编程问题，因而他们主张将各种方法结合进多范式编程语言。见由 Timothy Budd 撰写的(Addison-Wesley 1995)*Multiparadigm Programming in Leda*。

[3]这句话或许有点太过了。因为对象还能存在于另一台机器上以及不同的内存空间中，此外还能保存在硬盘上。在这种情况下，对象的身份就不能用内存地址，而必须要用别的方法来确定。

[4]有些人要把这两个概念区分开来，他们说 type 定义了接口，而 class 是这个接口的某个实现。

[5]对于这个术语，我和我的朋友 Scott Meyers 也有分歧。

[6]通常，对于绝大多数图表这么画已经可以了，你用不着去分清楚这是个聚合还是个合成。

[7]你后面要学到的 primitive 类型是特例。

[8]很遗憾 primitive 是个例外。本书会详细讨论这部分内容。

[9]实际上 Microsoft 已经说了 C#和 .NET“没那么快”。很多人问 VB 的程序员是不是应该转用其它语言，是 Java，C#还是 VB.NET。