第 1 章 对象入门

“为什么面向对象的编程会在软件开发领域造成如此震憾的影响？”

面向对象编程（OOP）具有多方面的吸引力。对管理人员，它实现了更快和更廉价的开发与维护过程。对分析与设计人员，建模处理变得更加简单，能生成清晰、易于维护的设计方案。对程序员，对象模型显得如此高雅和浅显。此外，面向对象工具以及库的巨大威力使编程成为一项更使人愉悦的任务。每个人都可从中获益，至少表面如此。

如果说它有缺点，那就是掌握它需付出的代价。思考对象的时候，需要采用形象思维，而不是程序化的思维。与程序化设计相比，对象的设计过程更具挑战性——特别是在尝试创建可重复使用（可再生）的对象时。过去，那些初涉面向对象编程领域的人都必须进行一项令人痛苦的选择：

(1) 选择一种诸如 Smalltalk 的语言，“出师”前必须掌握一个巨型的库。

(2) 选择几乎根本没有库的 C++（注释①），然后深入学习这种语言，直至能自行编写对象库。

①：幸运的是，这一情况已有明显改观。现在有第三方库以及标准的 C++库供选用。

事实上，很难很好地设计出对象——从而很难设计好任何东西。因此，只有数量相当少的“专家”能设计出最好的对象，然后让其他人享用。对于成功的 OOP 语言，它们不仅集成了这种语言的语法以及一个编译程序（编译器），而且还有一个成功的开发环境，其中包含设计优良、易于使用的库。所以，大多数程序员的首要任务就是用现有的对象解决自己的应用问题。本章的目标就是向大家揭示出面向对象编程的概念，并证明它有多么简单。

本章将向大家解释 Java 的多项设计思想，并从概念上解释面向对象的程序设计。但要注意在阅读完本章后，并不能立即编写出全功能的 Java 程序。所有详细的说明和示例会在本书的其他章节慢慢道来。

1.1 抽象的进步

所有编程语言的最终目的都是提供一种“抽象”方法。一种较有争议的说法是：解决问题的复杂程度直接取决于抽象的种类及质量。这儿的“种类”是指准备对什么进行“抽象”？汇编语言是对基础机器的少量抽象。后来的许多“命令式”语言（如 FORTRAN，BASIC 和 C）是对汇编语言的一种抽象。与汇编语言相比，这些语言已有了长足的进步，但它们的抽象原理依然要求我们着重考虑计算机的结构，而非考虑问题本身的结构。在机器模型（位于“方案空间”）与实际解决的问题模型（位于“问题空间”）之间，程序员必须建立起一种联系。这个过程要求人们付出较大的精力，而且由于它脱离了编程语言本身的范围，造成程序代码很难编写，而且要花较大的代价进行维护。由此造成的副作用便是一门完善的“编程方法”学科。

为机器建模的另一个方法是为要解决的问题制作模型。对一些早期语言来说，如 LISP 和 APL，它们的做法是“从不同的角度观察世界”——“所有问题都归纳为列表”或“所有问题都归纳为算法”。PROLOG 则将所有问题都归纳为决策链。对于这些语言，我们认为它们一部分是面向基于“强制”的编程，另一部分则是专为处理图形符号设计的。每种方法都有自己特殊的用途，适合解决某一类的问题。但只要超出了它们力所能及的范围，就会显得非常笨拙。

面向对象的程序设计在此基础上则跨出了一大步，程序员可利用一些工具表达问题空间内的元素。由于这种表达非常普遍，所以不必受限于特定类型的问题。我们将问题空间中的元素以及它们在方案空间的表示物称作“对象”（Object）。当然，还有一些在问题空间没有对应体的其他对象。通过添加新的对象类型，程序可进行灵活的调整，以便与特定的问题配合。所以在阅读方案的描述代码时，会读到对问题进行表达的话语。与我们以前见过的相比，这无疑是一种更加灵活、更加强大的语言抽象方法。总之，OOP 允许我们根据问题来描述问题，而不是根据方案。然而，仍有一个联系途径回到计算机。每个对象都类似一台小计算机；它们有自己的状态，而且可要求它们进行特定的操作。与现实世界的“对象”或者“物体”相比，编程“对象”与它们也存在共通的地方：它们都有自己的特征和行为。

Alan Kay 总结了 Smalltalk 的五大基本特征。这是第一种成功的面向对象程序设计语言，也是 Java 的基础语言。通过这些特征，我们可理解“纯粹”的面向对象程序设计方法是什么样的：

(1) 所有东西都是对象。可将对象想象成一种新型变量；它保存着数据，但可要求它对自身进行操作。理论上讲，可从要解决的问题身上提出所有概念性的组件，然后在程序中将其表达为一个对象。

(2) 程序是一大堆对象的组合；通过消息传递，各对象知道自己该做些什么。为了向对象发出请求，需向那个对象“发送一条消息”。更具体地讲，可将消息想象为一个调用请求，它调用的是从属于目标对象的一个子例程或函数。

(3) 每个对象都有自己的存储空间，可容纳其他对象。或者说，通过封装现

有对象，可制作出新型对象。所以，尽管对象的概念非常简单，但在程序中却可达到任意高的复杂程度。

(4) 每个对象都有一种类型。根据语法，每个对象都是某个“类”的一个“实例”。其中，“类”（Class）是“类型”（Type）的同义词。一个类最重要的特征就是“能将什么消息发给它？”。

(5) 同一类所有对象都能接收相同的消息。这实际是别有含义的一种说法，大家不久便能理解。由于类型为“圆”（Circle）的一个对象也属于类型为“形状”（Shape）的一个对象，所以一个圆完全能接收形状消息。这意味着可让程序代码统一指挥“形状”，令其自动控制所有符合“形状”描述的对象，其中自然包括“圆”。这一特性称为对象的“可替换性”，是 OOP 最重要的概念之一。

一些语言设计者认为面向对象的程序设计本身并不足以方便解决所有形式的程序问题，提倡将不同的方法组合成“多形程序设计语言”（注释②）。

②：参见 Timothy Budd 编著的《Multiparadigm Programming in Leda》，Addison-Wesley 1995 年出版。

1.2 对象的接口

亚里士多德或许是认真研究“类型”概念的第一人，他曾谈及“鱼类和鸟类”的问题。在世界首例面向对象语言 Simula-67 中，第一次用到了这样的一个概念：

所有对象——尽管各有特色——都属于某一系列对象的一部分，这些对象具有通用的特征和行为。在 Simula-67 中，首次用到了 class 这个关键字，它为程序引入了一个全新的类型（clas 和 type 通常可互换使用；注释③）。

③：有些人进行了进一步的区分，他们强调“类型”决定了接口，而“类”是那个接口的一种特殊实现方式。

Simula 是一个很好的例子。正如这个名字所暗示的，它的作用是“模拟”（Simulate）象“银行出纳员”这样的经典问题。在这个例子里，我们有一系列出纳员、客户、帐号以及交易等。每类成员（元素）都具有一些通用的特征：每个帐号都有一定的余额；每名出纳都能接收客户的存款；等等。与此同时，每个成员都有自己的状态；每个帐号都有不同的余额；每名出纳都有一个名字。所以在计算机程序中，能用独一无二的实体分别表示出纳员、客户、帐号以及交易。这个实体便是“对象”，而且每个对象都隶属一个特定的“类”，那个类具有自己的通用特征与行为。

因此，在面向对象的程序设计中，尽管我们真正要做的是新建各种各样的数据“类型”（Type），但几乎所有面向对象的程序设计语言都采用了“class”关键字。当您看到“type”这个字的时候，请同时想到“class”；反之亦然。

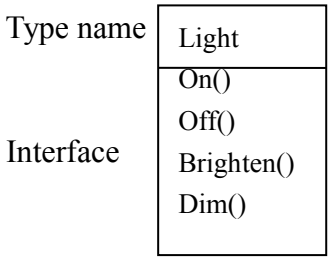
建好一个类后，可根据情况生成许多对象。随后，可将那些对象作为要解决问题中存在的元素进行处理。事实上，当我们进行面向对象的程序设计时，面临的最大一项挑战性就是：如何在“问题空间”（问题实际存在的地方）的元素与“方案空间”（对实际问题进行建模的地方，如计算机）的元素之间建立理想的“一对一”对应或映射关系。

如何利用对象完成真正有用的工作呢？必须有一种办法能向对象发出请求，

令其做一些实际的事情，比如完成一次交易、在屏幕上画一些东西或者打开一个开关等等。每个对象仅能接受特定的请求。我们向对象发出的请求是通过它的“接口”（Interface）定义的，对象的“类型”或“类”则规定了它的接口形式。“类型”与“接口”的等价或对应关系是面向对象程序设计的基础。

下面让我们以电灯泡为例：

29 页图




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

在这个例子中，类型 / 类的名称是 Light，可向 Light 对象发出的请求包括包括打开（on）、关闭（off）、变得更明亮（brighten）或者变得更暗淡（dim）。通过简单地声明一个名字（lt），我们为 Light 对象创建了一个“句柄”。然后用 new 关键字新建类型为 Light 的一个对象。再用等号将其赋给句柄。为了向对象发送一条消息，我们列出句柄名（lt），再用一个句点符号（.）把它同消息名称（on）连接起来。从中可以看出，使用一些预先定义好的类时，我们在程序里采用的代码是非常简单和直观的。

1.3 实现方案的隐藏

为方便后面的讨论，让我们先对这一领域的从业人员作一下分类。从根本上说，大致有两方面的人员涉足面向对象的编程：“类创建者”（创建新数据类型的人）以及“客户程序员”（在自己的应用程序中采用现成数据类型的人；注释④）。对客户程序员来讲，最主要的目标就是收集一个充斥着各种类的编程“工具箱”，以便快速开发符合自己要求的应用。而对类创建者来说，他们的目标则是从头构建一个类，只向客户程序员开放有必要开放的东西（接口），其他所有细节都隐藏起来。为什么要这样做？隐藏之后，客户程序员就不能接触和改变那些细节，所以原创者不用担心自己的作品会受到非法修改，可确保它们不会对其他人造成影响。

④：感谢我的朋友 Scott Meyers，是他帮我起了这个名字。

“接口”（Interface）规定了可对一个特定的对象发出哪些请求。然而，必须在某个地方存在着一些代码，以便满足这些请求。这些代码与那些隐藏起来的数据便叫作“隐藏的实现”。站在程式化程序编写（Procedural Programming）的角度，整个问题并不显得复杂。一种类型含有与每种可能的请求关联起来的函数。一旦向对象发出一个特定的请求，就会调用那个函数。我们通常将这个过程总结为向对象“发送一条消息”（提出一个请求）。对象的职责就是决定如何对这条消

息作出反应（执行相应的代码）。

对于任何关系，重要一点是让牵连到的所有成员都遵守相同的规则。创建一个库时，相当于同客户程序员建立了一种关系。对方也是程序员，但他们的目标是组合出一个特定的应用（程序），或者用您的库构建一个更大的库。

若任何人都能使用一个类的所有成员，那么客户程序员可对那个类做任何事情，没有办法强制他们遵守任何约束。即便非常不愿客户程序员直接操作类内包含的一些成员，但倘若未进行访问控制，就没有办法阻止这一情况的发生——所有东西都会暴露无遗。

有两方面的原因促使我们控制对成员的访问。第一个原因是防止程序员接触他们不该接触的东西——通常是内部数据类型的设计思想。若只是为了解决特定的问题，用户只需操作接口即可，毋需明白这些信息。我们向用户提供的实际是一种服务，因为他们很容易就可看出哪些对自己非常重要，以及哪些可忽略不计。

进行访问控制的第二个原因是允许库设计人员修改内部结构，不用担心它会对客户程序员造成什么影响。例如，我们最开始可能设计了一个形式简单的类，以便简化开发。以后又决定进行改写，使其更快地运行。若接口与实现方法早已隔离开，并分别受到保护，就可放心做到这一点，只要求用户重新链接一下即可。

Java 采用三个显式（明确）关键字以及一个隐式（暗示）关键字来设置类边界：public，private，protected 以及暗示性的 friendly。若未明确指定其他关键字，则默认为后者。这些关键字的使用和含义都是相当直观的，它们决定了谁能使用后续的定义内容。“public”（公共）意味着后续的定义任何人均可使用。而在另一方面，“private”（私有）意味着除您自己、类型的创建者以及那个类型的内部函数成员，其他任何人都不能访问后续的定义信息。private 在您与客户程序员之间竖起了一堵墙。若有人试图访问私有成员，就会得到一个编译期错误。“friendly”（友好的）涉及“包装”或“封装”（Package）的概念——即 Java 用来构建库的方法。若某样东西是“友好的”，意味着它只能在这个包装的范围内使用（所以这一访问级别有时也叫作“包装访问”）。“protected”（受保护的）与“private”相似，只是一个继承的类可访问受保护的成员，但不能访问私有成员。继承的问题不久就要谈到。

1.4 方案的重复使用

创建并测试好一个类后，它应（从理想的角度）代表一个有用的代码单位。但并不象许多人希望的那样，这种重复使用的能力并不容易实现；它要求较多的经验以及洞察力，这样才能设计出一个好的方案，才有可能重复使用。

许多人认为代码或设计方案的重复使用是面向对象的程序设计提供的最伟大的一种杠杆。

为重复使用一个类，最简单的办法是仅直接使用那个类的对象。但同时也能将那个类的一个对象置入一个新类。我们把这叫作“创建一个成员对象”。新类可由任意数量和类型的其他对象构成。无论如何，只要新类达到了设计要求即可。这个概念叫作“组织”——在现有类的基础上组织一个新类。有时，我们也将组织称作“包含”关系，比如“一辆车包含了一个变速箱”。

对象的组织具有极大的灵活性。新类的“成员对象”通常设为“私有”（Private），使用这个类的客户程序员不能访问它们。这样一来，我们可在不干扰客户代码的前提下，从容地修改那些成员。也可以在“运行期”更改成员，这进一步增大了

灵活性。后面要讲到的“继承”并不具备这种灵活性，因为编译器必须对通过继承创建的类加以限制。

由于继承的重要性，所以在面向对象的程序设计中，它经常被重点强调。作为新加入这一领域的程序员，或许早已先入为主地认为“继承应当随处可见”。沿这种思路产生的设计将是非常笨拙的，会大大增加程序的复杂程度。相反，新建类的时候，首先应考虑“组织”对象；这样做显得更加简单和灵活。利用对象的组织，我们的设计可保持清爽。一旦需要用到继承，就会明显意识到这一点。

1.5 继承：重新使用接口

就其本身来说，对象的概念可为我们带来极大的便利。它在概念上允许我们将各式各样数据和功能封装到一起。这样便可恰当表达“问题空间”的概念，不用刻意遵照基础机器的表达方式。在程序设计语言中，这些概念则反映为具体的数据类型（使用 class 关键字）。

我们费尽心思做出一种数据类型后，假如不得不又新建一种类型，令其实现大致相同的功能，那会是一件非常令人灰心的事情。但若能利用现成的数据类型，对其进行“克隆”，再根据情况进行添加和修改，情况就显得理想多了。“继承”正是针对这个目标而设计的。但继承并不完全等价于克隆。在继承过程中，若原始类（正式名称叫作基础类、超类或父类）发生了变化，修改过的“克隆”类（正式名称叫作继承类或者子类）也会反映出这种变化。在 Java 语言中，继承是通过 extends 关键字实现的

使用继承时，相当于创建了一个新类。这个新类不仅包含了现有类型的所有成员（尽管 private 成员被隐藏起来，且不能访问），但更重要的是，它复制了基础类的接口。也就是说，可向基础类的对象发送的所有消息亦可原样发给衍生类的对象。根据可以发送的消息，我们能知道类的类型。这意味着衍生类具有与基础类相同的类型！为真正理解面向对象程序设计的含义，首先必须认识到这种类型的等价关系。

由于基础类和衍生类具有相同的接口，所以那个接口必须进行特殊的设计。也就是说，对象接收到一条特定的消息后，必须有一个“方法”能够执行。若只是简单地继承一个类，并不做其他任何事情，来自基础类接口的方法就会直接照搬到衍生类。这意味着衍生类的对象不仅有相同的类型，也有同样的行为，这一后果通常是我们不愿见到的。

有两种做法可将新得的衍生类与原来的基础类区分开。第一种做法十分简单：为衍生类添加新函数（功能）。这些新函数并非基础类接口的一部分。进行这种处理时，一般都是意识到基础类不能满足我们的要求，所以需要添加更多的函数。这是一种最简单、最基本的继承用法，大多数时候都可完美地解决我们的问题。然而，事先还是要仔细调查自己的基础类是否真的需要这些额外的函数。

1.5.1 改善基础类

尽管 extends 关键字暗示着我们要为接口“扩展”新功能，但实情并非肯定如此。为区分我们的新类，第二个办法是改变基础类一个现有函数的行为。我们将其称作“改善”那个函数。

为改善一个函数，只需为衍生类的函数建立一个新定义即可。我们的目标是：“尽管使用的函数接口未变，但它的新版本具有不同的表现”。

1.5.2 等价与类似关系

针对继承可能会产生这样的一个争论：继承只能改善原基础类的函数吗？若答案是肯定的，则衍生类型就是与基础类完全相同的类型，因为都拥有完全相同的接口。这样造成的结果就是：我们完全能够将衍生类的一个对象换成基础类的一个对象！可将其想象成一种“纯替换”。在某种意义上，这是进行继承的一种理想方式。此时，我们通常认为基础类和衍生类之间存在一种“等价”关系——因为我们可以理直气壮地说：“圆就是一种几何形状”。为了对继承进行测试，一个办法就是看看自己是否能把它们套入这种“等价”关系中，看看是否有意义。

但在许多时候，我们必须为衍生类型加入新的接口元素。所以不仅扩展了接口，也创建了一种新类型。这种新类型仍可替换成基础类型，但这种替换并不是完美的，因为不可在基础类里访问新函数。我们将其称作“类似”关系；新类型拥有旧类型的接口，但也包含了其他函数，所以不能说它们是完全等价的。举个例子来说，让我们考虑一下制冷机的情况。假定我们的房间连好了用于制冷的各种控制器；也就是说，我们已拥有必要的“接口”来控制制冷。现在假设机器出了故障，我们把它换成一台新型的冷、热两用空调，冬天和夏天均可使用。冷、热空调“类似”制冷机，但能做更多的事情。由于我们的房间只安装了控制制冷的设备，所以它们只限于同新机器的制冷部分打交道。新机器的接口已得到了扩展，但现有的系统并不知道除原始接口以外的任何东西。

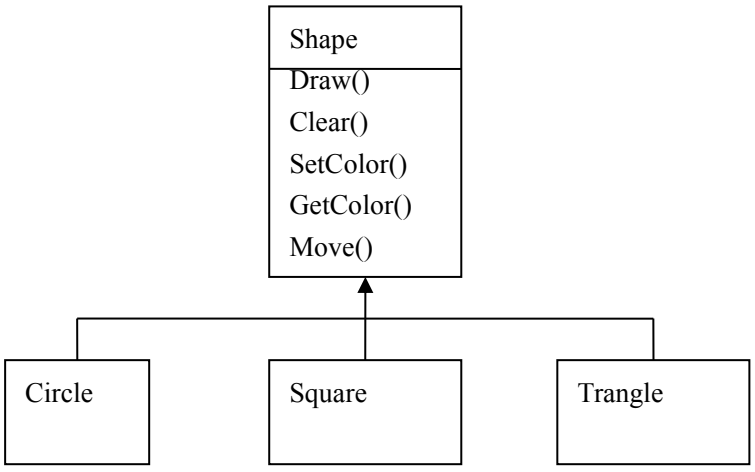
认识了等价与类似的区别后，再进行替换时就会有把握得多。尽管大多数时候“纯替换”已经足够，但您会发现在某些情况下，仍然有明显的理由需要在衍生类的基础上增添新功能。通过前面对这两种情况的讨论，相信大家已心中有数该如何做。

1.6 多形对象的互换使用

通常，继承最终会以创建一系列类收场，所有类都建立在统一的接口基础上。我们用一幅颠倒的树形图来阐明这一点（注释⑤）：

⑤：这儿采用了“统一记号法”，本书将主要采用这种方法。

35 页图



对这样的一系列类，我们要进行的一项重要处理就是将衍生类的对象当作基础类的一个对象对待。这一点是非常重要的，因为它意味着我们只需编写单一的

代码，令其忽略类型的特定细节，只与基础类打交道。这样一来，那些代码就可与类型信息分开。所以更易编写，也更易理解。此外，若通过继承增添了一种新类型，如“三角形”，那么我们为“几何形状”新类型编写的代码会象在旧类型里一样良好地工作。所以说程序具备了“扩展能力”，具有“扩展性”。

以上面的例子为基础，假设我们用 Java 写了这样一个函数：

35-36 页程序

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

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

这个函数可与任何“几何形状”（Shape）通信，所以完全独立于它要描绘（draw）和删除（erase）的任何特定类型的对象。如果我们在其他一些程序里使用 doStuff()函数：

36 页上程序

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

那么对 doStuff()的调用会自动良好地工作，无论对象的具体类型是什么。

这实际是一个非常有用的编程技巧。请考虑下面这行代码：

doStuff(c);

此时，一个 Circle（圆）句柄传递给一个本来期待 Shape（形状）句柄的函数。由于圆是一种几何形状，所以 doStuff()能正确地进行处理。也就是说，凡是 doStuff()能发给一个 Shape 的消息，Circle 也能接收。所以这样做是安全的，不会造成错误。

我们将这种把衍生类型当作它的基本类型处理的过程叫作“Upcasting”（上溯造型）。其中，“cast”（造型）是指根据一个现成的模型创建；而“Up”（向上）表明继承的方向是从“上面”来的——即基础类位于顶部，而衍生类在下方展开。所以，根据基础类进行造型就是一个从上面继承的过程，即“Upcasting”。

在面向对象的程序里，通常都要用到上溯造型技术。这是避免去调查准确类型的一个好办法。请看看 doStuff()里的代码：

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

注意它并未这样表达：“如果你是一个 Circle，就这样做；如果你是一个 Square，就那样做；等等”。若那样编写代码，就需检查一个 Shape 所有可能的类型，如圆、矩形等等。这显然是非常麻烦的，而且每次添加了一种新的 Shape 类型后，都要相应地进行修改。在这儿，我们只需说：“你是一种几何形状，我知道你能将自己删掉，即 erase()；请自己采取那个行动，并自己去控制所有的细节吧。”

1.6.1 动态绑定

在 doStuff()的代码里，最让人吃惊的是尽管我们没作出任何特殊指示，采取的操作也是完全正确和恰当的。我们知道，为 Circle 调用 draw()时执行的代码与为一个 Square 或 Line 调用 draw()时执行的代码是不同的。但在将 draw()消息发给一个匿名 Shape 时，根据 Shape 句柄当时连接的实际类型，会相应地采取正确的操作。这当然令人惊讶，因为当 Java 编译器为 doStuff()编译代码时，它并不知道自己要操作的准确类型是什么。尽管我们确实可以保证最终会为 Shape 调用 erase()，为 Shape 调用 draw()，但并不能保证为特定的 Circle，Square 或者 Line 调用什么。然而最后采取的操作同样是正确的，这是怎么做到的呢？

将一条消息发给对象时，如果并不知道对方的具体类型是什么，但采取的行动同样是正确的，这种情况就叫作“多形性”（Polymorphism）。对面向对象的程序设计语言来说，它们用以实现多形性的方法叫作“动态绑定”。编译器和运行期系统会负责对所有细节的控制；我们只需知道会发生什么事情，而且更重要的是，如何利用它帮助自己设计程序。

有些语言要求我们用一个特殊的关键字来允许动态绑定。在 C++中，这个关键字是 virtual。在 Java 中，我们则完全不必记住添加一个关键字，因为函数的动态绑定是自动进行的。所以在将一条消息发给对象时，我们完全可以肯定对象会采取正确的行动，即使其中涉及上溯造型之类的处理。

1.6.2 抽象的基础类和接口

设计程序时，我们经常都希望基础类只为自己的衍生类提供一个接口。也就是说，我们不想其他任何人实际创建基础类的一个对象，只对上溯造型成它，以便使用它们的接口。为达到这个目的，需要把那个类变成“抽象”的——使用 abstract 关键字。若有人试图创建抽象类的一个对象，编译器就会阻止他们。这种工具可有效强制实行一种特殊的设计。

亦可用 abstract 关键字描述一个尚未实现的方法——作为一个“根”使用，指出：“这是适用于从这个类继承的所有类型的一个接口函数，但目前尚没有对它进行任何形式的实现。”抽象方法也许只能在一个抽象类里创建。继承了一个类后，那个方法就必须实现，否则继承的类也会变成“抽象”类。通过创建一个抽象方法，我们可以将一个方法置入接口中，不必再为那个方法提供可能毫无意义的主体代码。

interface（接口）关键字将抽象类的概念更延伸了一步，它完全禁止了所有的函数定义。“接口”是一种相当有效和常用的工具。另外如果自己愿意，亦可

将多个接口都合并到一起（不能从多个普通 class 或 abstract class 中继承）。

1.7 对象的创建和存在时间

从技术角度说，OOP（面向对象程序设计）只是涉及抽象的数据类型、继承以及多形性，但另一些问题也可能显得非常重要。本节将就这些问题进行探讨。

最重要的问题之一是对象的创建及破坏方式。对象需要的数据位于哪儿，如何控制对象的“存在时间”呢？针对这个问题，解决的方案是各异其趣的。C++认为程序的执行效率是最重要的一个问题，所以它允许程序员作出选择。为获得最快的运行速度，存储以及存在时间可在编写程序时决定，只需将对象放置在堆栈（有时也叫作自动或定域变量）或者静态存储区域即可。这样便为存储空间的分配和释放提供了一个优先级。某些情况下，这种优先级的控制是非常有价值的。然而，我们同时也牺牲了灵活性，因为在编写程序时，必须知道对象的准确的数量、存在时间、以及类型。如果要解决的是一个较常规的问题，如计算机辅助设计、仓储管理或者空中交通控制，这一方法就显得太局限了。

第二个方法是在一个内存池中动态创建对象，该内存池亦叫“堆”或者“内存堆”。若采用这种方式，除非进入运行期，否则根本不知道到底需要多少个对象，也不知道它们的存在时间有多长，以及准确的类型是什么。这些参数都在程序正式运行时才决定的。若需一个新对象，只需在需要它的时候在内存堆里简单地创建它即可。由于存储空间的管理是运行期间动态进行的，所以在内存堆里分配存储空间的时间比在堆栈里创建的时间长得多（在堆栈里创建存储空间一般只需要一个简单的指令，将堆栈指针向下或向下移动即可）。由于动态创建方法使对象本来就倾向于复杂，所以查找存储空间以及释放它所需的额外开销不会为对象的创建造成明显的影响。除此以外，更大的灵活性对于常规编程问题的解决是至关重要的。

C++允许我们决定是在写程序时创建对象，还是在运行期间创建，这种控制方法更加灵活。大家或许认为既然它如此灵活，那么无论如何都应在内存堆里创建对象，而不是在堆栈中创建。但还要考虑另外一个问题，亦即对象的“存在时间”或者“生存时间”（Lifetime）。若在堆栈或者静态存储空间里创建一个对象，编译器会判断对象的持续时间有多长，到时会自动“破坏”或者“清除”它。程序员可用两种方法来破坏一个对象：用程序化的方式决定何时破坏对象，或者利用由运行环境提供的一种“垃圾收集器”特性，自动寻找那些不再使用的对象，并将其清除。当然，垃圾收集器显得方便得多，但要求所有应用程序都必须容忍垃圾收集器的存在，并能默许随垃圾收集带来的额外开销。但这并不符合 C++语言的设计宗旨，所以未能包括到 C++里。但 Java 确实提供了一个垃圾收集器（Smalltalk 也有这样的设计；尽管 Delphi 默认为没有垃圾收集器，但可选择安装；而 C++亦可使用一些由其他公司开发的垃圾收集产品）。

本节剩下的部分将讨论操纵对象时要考虑的另一些因素。

1.7.1 集合与继承器

针对一个特定问题的解决，如果事先不知道需要多少个对象，或者它们的持续时间有多长，那么也不知道如何保存那些对象。既然如此，怎样才能知道那些对象要求多少空间呢？事先上根本无法提前知道，除非进入运行期。

在面向对象的设计中，大多数问题的解决办法似乎都有些轻率——只是简单地创建另一种类型的对象。用于解决特定问题的新型对象容纳了指向其他对象的

句柄。当然，也可以用数组来做同样的事情，那是大多数语言都具有的一种功能。但不能只看到这一点。这种新对象通常叫作“集合”（亦叫作一个“容器”，但AWT 在不同的场合应用了这个术语，所以本书将一直沿用“集合”的称呼。在需要的时候，集合会自动扩充自己，以便适应我们在其中置入的任何东西。所以我们事先不必知道要在一个集合里容下多少东西。只需创建一个集合，以后的工作让它自己负责好了。

幸运的是，设计优良的 OOP 语言都配套提供了一系列集合。在 C++中，它们是以“标准模板库”（STL）的形式提供的。Object Pascal 用自己的“可视组件库”（VCL）提供集合。Smalltalk 提供了一套非常完整的集合。而 Java 也用自己的标准库提供了集合。在某些库中，一个常规集合便可满足人们的大多数要求；而在另一些库中（特别是 C++的库），则面向不同的需求提供了不同类型的集合。例如，可以用一个矢量统一对所有元素的访问方式；一个链接列表则用于保证所有元素的插入统一。所以我们能根据自己的需要选择适当的类型。其中包括集、队列、散列表、树、堆栈等等。

所有集合都提供了相应的读写功能。将某样东西置入集合时，采用的方式是十分明显的。有一个叫作“推”（Push）、“添加”（Add）或其他类似名字的函数用于做这件事情。但将数据从集合中取出的时候，方式却并不总是那么明显。如果是一个数组形式的实体，比如一个矢量（Vector），那么也许能用索引运算符或函数。但在许多情况下，这样做往往会无功而返。此外，单选定函数的功能是非常有限的。如果想对集合中的一系列元素进行操纵或比较，而不是仅仅面向一个，这时又该怎么办呢？

办法就是使用一个“继续器”（Iterator），它属于一种对象，负责选择集合内的元素，并把它们提供给继承器的用户。作为一个类，它也提供了一级抽象。利用这一级抽象，可将集合细节与用于访问那个集合的代码隔离开。通过继承器的作用，集合被抽象成一个简单的序列。继承器允许我们遍历那个序列，同时毋需关心基础结构是什么——换言之，不管它是一个矢量、一个链接列表、一个堆栈，还是其他什么东西。这样一来，我们就可以灵活地改变基础数据，不会对程序里的代码造成干扰。Java 最开始（在 1.0 和 1.1 版中）提供的是一个标准继承器，名为 Enumeration（枚举），为它的所有集合类提供服务。Java 1.2 新增一个更复杂的集合库，其中包含了一个名为 Iterator 的继承器，可以做比老式的 Enumeration 更多的事情。

从设计角度出发，我们需要的是一个全功能的序列。通过对它的操纵，应该能解决自己的问题。如果一种类型的序列即可满足我们的所有要求，那么完全没有必要再换用不同的类型。有两方面的原因促使我们需要对集合作出选择。首先，集合提供了不同的接口类型以及外部行为。堆栈的接口与行为与队列的不同，而队列的接口与行为又与一个集（Set）或列表的不同。利用这个特征，我们解决问题时便有更大的灵活性。

其次，不同的集合在进行特定操作时往往有不同的效率。最好的例子便是矢量（Vector）和列表（List）的区别。它们都属于简单的序列，拥有完全一致的接口和外部行为。但在执行一些特定的任务时，需要的开销却是完全不同的。对矢量内的元素进行的随机访问（存取）是一种常时操作；无论我们选择的选择是什么，需要的时间量都是相同的。但在一个链接列表中，若想到处移动，并随机挑选一个元素，就需付出“惨重”的代价。而且假设某个元素位于列表较远的地方，找到它所需的时间也会长许多。但在另一方面，如果想在序列中部插入一个

元素，用列表就比用矢量划算得多。这些以及其他操作都有不同的执行效率，具体取决于序列的基础结构是什么。在设计阶段，我们可以先从一个列表开始。最后调整性能的时候，再根据情况把它换成矢量。由于抽象是通过继承器进行的，所以能在两者方便地切换，对代码的影响则显得微不足道。

最后，记住集合只是一个用来放置对象的储藏所。如果那个储藏所能满足我们的所有需要，就完全没必要关心它具体是如何实现的（这是大多数类型对象的一个基本概念）。如果在一个编程环境中工作，它由于其他因素（比如在 Windows 下运行，或者由垃圾收集器带来了开销）产生了内在的开销，那么矢量和链接列表之间在系统开销上的差异就或许不是一个大问题。我们可能只需要一种类型的序列。甚至可以想象有一个“完美”的集合抽象，它能根据自己的使用方式自动改变基层的实现方式。

1.7.2 单根结构

在面向对象的程序设计中，由于 C++的引入而显得尤为突出的一个问题是：所有类最终是否都应从单独一个基础类继承。在 Java 中（与其他几乎所有 OOP 语言一样），对这个问题的答案都是肯定的，而且这个终级基础类的名字很简单，就是一个“Object”。这种“单根结构”具有许多方面的优点。

单根结构中的所有对象都有一个通用接口，所以它们最终都属于相同的类型。另一种方案（就象 C++那样）是我们不能保证所有东西都属于相同的基本类型。从向后兼容的角度看，这一方案可与 C 模型更好地配合，而且可以认为它的限制更少一些。但假期我们想进行纯粹的面向对象编程，那么必须构建自己的结构，以期获得与内建到其他 OOP 语言里的同样的便利。需添加我们要用到的各种新类库，还要使用另一些不兼容的接口。理所当然地，这也需要付出额外的精力使新接口与自己的设计方案配合（可能还需要多重继承）。为得到 C++额外的“灵活性”，付出这样的代价值得吗？当然，如果真的需要——如果早已是 C 专家，如果对 C 有难舍的情结——那么就真的很值得。但假如你是一名新手，首次接触这类设计，象 Java 那样的替换方案也许会更省事一些。

单根结构中的所有对象（比如所有 Java 对象）都可以保证拥有一些特定的功能。在自己的系统中，我们知道对每个对象都能进行一些基本操作。一个单根结构，加上所有对象都在内存堆中创建，可以极大简化参数的传递（这在 C++里是一个复杂的概念）。

利用单根结构，我们可以更方便地实现一个垃圾收集器。与此有关的必要支持可安装于基础类中，而垃圾收集器可将适当的消息发给系统内的任何对象。如果没有这种单根结构，而且系统通过一个句柄来操纵对象，那么实现垃圾收集器的途径会有很大的不同，而且会面临许多障碍。

由于运行期的类型信息肯定存在于所有对象中，所以永远不会遇到判断不出一个对象的类型的情况。这对系统级的操作来说显得特别重要，比如违例控制；而且也能在程序设计时获得更大的灵活性。

但大家也可能产生疑问，既然你把好处说得这么天花乱坠，为什么 C++没有采用单根结构呢？事实上，这是早期在效率与控制上权衡的一种结果。单根结构会带来程序设计上的一些限制。而且更重要的是，它加大了新程序与原有 C 代码兼容的难度。尽管这些限制仅在特定的场合会真的造成问题，但为了获得最大的灵活程度，C++最终决定放弃采用单根结构这一做法。而 Java 不存在上述的问题，它是全新设计的一种语言，不必与现有的语言保持所谓的“向后兼容”。所

以很自然地，与其他大多数面向对象的程序设计语言一样，单根结构在 Java 的设计方案中很快就落实下来。

1.7.3 集合库与方便使用集合

由于集合是我们经常都要用到的一种工具，所以一个集合库是十分必要的，它应该可以方便地重复使用。这样一来，我们就可以方便地取用各种集合，将其插入自己的程序。Java 提供了这样的一个库，尽管它在 Java 1.0 和 1.1 中都显得非常有限（Java 1.2 的集合库则无疑是一个杰作）。

1. 下溯造型与模板 / 通用性

为了使这些集合能够重复使用，或者“再生”，Java 提供了一种通用类型，以前曾把它叫作“Object”。单根结构意味着、所有东西归根结底都是一个对象”！所以容纳了 Object 的一个集合实际可以容纳任何东西。这使我们对它的重复使用变得非常简便。

为使用这样的一个集合，只需添加指向它的对象句柄即可，以后可以通过句柄重新使用对象。但由于集合只能容纳 Object，所以在我们向集合里添加对象句柄时，它会上溯造型成 Object，这样便丢失了它的身份或者标识信息。再次使用它的时候，会得到一个 Object 句柄，而非指向我们早先置入的那个类型的句柄。所以怎样才能归还它的本来面貌，调用早先置入集合的那个对象的有用接口呢？

在这里，我们再次用到了造型（Cast）。但这一次不是在分级结构中上溯造型成一种更“通用”的类型。而是下溯造型成一种更“特殊”的类型。这种造型方法叫作“下溯造型”（Downcasting）。举个例子来说，我们知道在上溯造型的时候，Circle（圆）属于 Shape（几何形状）的一种类型，所以上溯造型是安全的。但我们不知道一个 Object 到底是 Circle 还是 Shape，所以很难保证下溯造型的安全进行，除非确切地知道自己要操作的是什么。

但这也不是绝对危险的，因为假如下溯造型成错误的东西，会得到我们称为“违例”（Exception）的一种运行期错误。我们稍后即会对此进行解释。但在从一个集合提取对象句柄时，必须用某种方式准确地记住它们是什么，以保证下溯造型的正确进行。

下溯造型和运行期检查都要求花额外的时间来运行程序，而且程序员必须付出额外的精力。既然如此，我们能不能创建一个“智能”集合，令其知道自己容纳的类型呢？这样做可消除下溯造型的必要以及潜在的错误。答案是肯定的，我们可以采用“参数化类型”，它们是编译器能自动定制的类，可与特定的类型配合。例如，通过使用一个参数化集合，编译器可对那个集合进行定制，使其只接受 Shape，而且只提取 Shape。

参数化类型是 C++一个重要的组成部分，这部分是 C++没有单根结构的缘故。在 C++中，用于实现参数化类型的关键字是 template（模板）。Java 目前尚未提供参数化类型，因为由于使用的是单根结构，所以使用它显得有些笨拙。但这并不能保证以后的版本不会实现，因为“generic”这个词已被 Java“保留到将来实现”（在 Ada 语言中，“generic”被用来实现它的模板）。Java 采取的这种关键字保留机制其实经常让人摸不着头脑，很难断定以后会发生什么事情。

1.7.4 清除时的困境：由谁负责清除？

每个对象都要求资源才能“生存”，其中最令人注目的资源是内存。如果不

再需要使用一个对象，就必须将其清除，以便释放这些资源，以便其他对象使用。如果要解决的是非常简单的问题，如何清除对象这个问题并不显得很突出：我们创建对象，在需要的时候调用它，然后将其清除或者“破坏”。但在另一方面，我们平时遇到的问题往往要比这复杂得多。

举个例子来说，假设我们要设计一套系统，用它管理一个机场的空中交通（同样的模型也可能适于管理一个仓库的货柜、或者一套影带出租系统、或者宠物店的宠物房。这初看似乎十分简单：构造一个集合用来容纳飞机，然后创建一架新飞机，将其置入集合。对进入空中交通管制区的所有飞机都如此处理。至于清除，在一架飞机离开这个区域的时候把它简单地删去即可。

但事情并没有这么简单，可能还需要另一套系统来记录与飞机有关的数据。当然，和控制器的主要功能不同，这些数据的重要性可能一开始并不显露出来。例如，这条记录反映的可能是离开机场的所有小飞机的飞行计划。所以我们得到了由小飞机组成的另一个集合。一旦创建了一个飞机对象，如果它是一架小飞机，那么也必须把它置入这个集合。然后在系统空闲时期，需对这个集合中的对象进行一些后台处理。

问题现在显得更复杂了：如何才能知道什么时间删除对象呢？用完对象后，系统的其他某些部分可能仍然要发挥作用。同样的问题也会在其他大量场合出现，而且在程序设计系统中（如 C++），在用完一个对象之后必须明确地将其删除，所以问题会变得异常复杂（注释⑥）。

⑥：注意这一点只对内存堆里创建的对象成立（用 new 命令创建的）。但在另一方面，对这儿描述的问题以及其他所有常见的编程问题来说，都要求对象在内存堆里创建。

在 Java 中，垃圾收集器在设计时已考虑到了内存的释放问题（尽管这并不包括清除一个对象涉及到的其他方面）。垃圾收集器“知道”一个对象在什么时候不再使用，然后会自动释放那个对象占据的内存空间。采用这种方式，另外加上所有对象都从单个根类 Object 继承的事实，而且由于我们只能在内存堆中以一种方式创建对象，所以 Java 的编程要比 C++的编程简单得多。我们只需要作出少量的抉择，即可克服原先存在的大量障碍。

1. 垃圾收集器对效率及灵活性的影响

既然这是如此好的一种手段，为什么在 C++里没有得到充分的发挥呢？我们当然要为这种编程的方便性付出一定的代价，代价就是运行期的开销。正如早先提到的那样，在 C++中，我们可在堆栈中创建对象。在这种情况下，对象会得以自动清除（但不具有在运行期间随心所欲创建对象的灵活性）。在堆栈中创建对象是为对象分配存储空间最有效的一种方式，也是释放那些空间最有效的一种方式。在内存堆（Heap）中创建对象可能要付出昂贵得多的代价。如果总是从同一个基础类继承，并使所有函数调用都具有“同质多形”特征，那么也不可避免地需要付出一定的代价。但垃圾收集器是一种特殊的问题，因为我们永远不能确定它什么时候启动或者要花多长的时间。这意味着在 Java 程序执行期间，存在着一种不连贯的因素。所以在某些特殊的场合，我们必须避免用它——比如在一个程序的执行必须保持稳定、连贯的时候（通常把它们叫作“实时程序”，尽管并不是所有实时编程问题都要这方面的要求——注释⑦）。

⑦：根据本书一些技术性读者的反馈，有一个现成的实时 Java 系统（www.newmonics.com）确实能够保证垃圾收集器的效能。

C++语言的设计者曾经向 C 程序员发出请求（而且做得非常成功），不要希望在可以使用 C 的任何地方，向语言里加入可能对 C++的速度或使用造成影响的任何特性。这个目的达到了，但代价就是 C++的编程不可避免地复杂起来。Java 比 C++简单，但付出的代价是效率以及一定程度的灵活性。但对大多数程序设计问题来说，Java 无疑都应是我们的首选。

1.8 违例控制：解决错误

从最古老的程序设计语言开始，错误控制一直都是设计者们需要解决的一个大问题。由于很难设计出一套完美的错误控制方案，许多语言干脆将问题简单地忽略掉，将其转嫁给库设计人员。对大多数错误控制方案来说，最主要的一个问题是它们严重依赖程序员的警觉性，而不是依赖语言本身的强制标准。如果程序员不够警惕——若比较匆忙，这几乎是肯定会发生的——程序所依赖的错误控制方案便会失效。

“违例控制”将错误控制方案内置到程序设计语言中，有时甚至内建到操作系统内。这里的“违例”（Exception）属于一个特殊的对象，它会从产生错误的地方“扔”或“掷”出来。随后，这个违例会被设计用于控制特定类型错误的“违例控制器”捕获。在情况变得不对劲的时候，可能有几个违例控制器并行捕获对应的违例对象。由于采用的是独立的执行路径，所以不会干扰我们的常规执行代码。这样便使代码的编写变得更加简单，因为不必经常性强制检查代码。除此以外，“掷”出的一个违例不同于从函数返回的错误值，也不同于由函数设置的一个标志。那些错误值或标志的作用是指示一个错误状态，是可以忽略的。但违例不能被忽略，所以肯定能在某个地方得到处置。最后，利用违例能够可靠地从一个糟糕的环境中恢复。此时一般不需要退出，我们可以采取某些处理，恢复程序的正常执行。显然，这样编制出来的程序显得更加可靠。

Java 的违例控制机制与大多数程序设计语言都有所不同。因为在 Java 中，违例控制模块是从一开始就封装好的，所以必须使用它！如果没有自己写一些代码来正确地控制违例，就会得到一条编译期出错提示。这样可保证程序的连贯性，使错误控制变得更加容易。

注意违例控制并不属于一种面向对象的特性，尽管在面向对象的程序设计语言中，违例通常是用一个对象表示的。早在面向对象语言问世以前，违例控制就已经存在了。

1.9 多线程

在计算机编程中，一个基本的概念就是同时对多个任务加以控制。许多程序设计问题都要求程序能够停下手头的工作，改为处理其他一些问题，再返回主进程。可以通过多种途径达到这个目的。最开始的时候，那些拥有机器低级知识的程序员编写一些“中断服务例程”，主进程的暂停是通过硬件级的中断实现的。尽管这是一种有用的方法，但编出的程序很难移植，由此造成了另一类的代价高昂问题。

有些时候，中断对那些实时性很强的任务来说是很有必要的。但还存在其他

许多问题，它们只要求将问题划分进入独立运行的程序片断中，使整个程序能更迅速地响应用户的请求。在一个程序中，这些独立运行的片断叫作“线程”（Thread），利用它编程的概念就叫作“多线程处理”。多线程处理一个常见的例子就是用户界面。利用线程，用户可按下一个按钮，然后程序会立即作出响应，而不是让用户等待程序完成了当前任务以后才开始响应。

最开始，线程只是用于分配单个处理器的处理时间的一种工具。但假如操作系统本身支持多个处理器，那么每个线程都可分配给一个不同的处理器，真正进入“并行运算”状态。从程序设计语言的角度看，多线程操作最有价值的特性之一就是程序员不必关心到底使用了多少个处理器。程序在逻辑意义上被分割为数个线程；假如机器本身安装了多个处理器，那么程序会运行得更快，毋需作出任何特殊的调校。

根据前面的论述，大家可能感觉线程处理非常简单。但必须注意一个问题：共享资源！如果有多个线程同时运行，而且它们试图访问相同的资源，就会遇到一个问题。举个例子来说，两个进程不能将信息同时发送给一台打印机。为解决这个问题，对那些可共享的资源来说（比如打印机），它们在使用期间必须进入锁定状态。所以一个线程可将资源锁定，在完成了它的任务后，再解开（释放）这个锁，使其他线程可以接着使用同样的资源。

Java 的多线程机制已内建到语言中，这使一个可能较复杂的问题变得简单起来。对多线程处理的支持是在对象这一级支持的，所以一个执行线程可表达为一个对象。Java 也提供了有限的资源锁定方案。它能锁定任何对象占用的内存（内存实际是多种共享资源的一种），所以同一时间只能有一个线程使用特定的内存空间。为达到这个目的，需要使用 synchronized 关键字。其他类型的资源必须由程序员明确锁定，这通常要求程序员创建一个对象，用它代表一把锁，所有线程在访问那个资源时都必须检查这把锁。

1.10 永久性

创建一个对象后，只要我们需要，它就会一直存在下去。但在程序结束运行时，对象的“生存期”也会宣告结束。尽管这一现象表面上非常合理，但深入追究就会发现，假如在程序停止运行以后，对象也能继续存在，并能保留它的全部信息，那么在某些情况下将是一件非常有价值的事情。下次启动程序时，对象仍然在那里，里面保留的信息仍然是程序上一次运行时的那些信息。当然，可以将信息写入一个文件或者数据库，从而达到相同的效果。但尽管可将所有东西都看作一个对象，如果能将对象声明成“永久性”，并令其为我们照看其他所有细节，无疑也是一件相当方便的事情。

Java 1.1 提供了对“有限永久性”的支持，这意味着我们可将对象简单地保存到磁盘上，以后任何时间都可取回。之所以称它为“有限”的，是由于我们仍然需要明确发出调用，进行对象的保存和取回工作。这些工作不能自动进行。在 Java 未来的版本中，对“永久性”的支持有望更加全面。

1.11 Java 和因特网

既然 Java 不过另一种类型的程序设计语言，大家可能会奇怪它为什么值得如此重视，为什么还有这么多的人认为它是计算机程序设计的一个里程碑呢？如果您来自一个传统的程序设计背景，那么答案在刚开始的时候并不是很明显。Java 除了可解决传统的程序设计问题以外，还能解决 World Wide Web（万维网）上

的编程问题。

1.11.1 什么是 Web?

Web 这个词刚开始显得有些泛泛，似乎“冲浪”、“网上存在”以及“主页”等等都和它拉上了一些关系。甚至还有一种“Internet 综合症”的说法，对许多人狂热的上网行为提出了质疑。我们在这里有必要作一些深入的探讨，但在这之前，必须理解客户机 / 服务器系统的概念，这是充斥着许多令人迷惑的问题的又一个计算领域。

1. 客户机 / 服务器计算

客户机 / 服务器系统的基本思想是我们能在一个统一的地方集中存放信息资源。一般将数据集中保存在某个数据库中，根据其他人或者机器的请求将信息投递给对方。客户机 / 服务器概述的一个关键在于信息是“集中存放”的。所以我们能方便地更改信息，然后将修改过的信息发放给信息的消费者。将各种元素集中到一起，信息仓库、用于投递信息的软件以及信息及软件所在的那台机器，它们联合起来便叫作“服务器”（Server）。而对那些驻留在远程机器上的软件，它们需要与服务器通信，取回信息，进行适当的处理，然后在远程机器上显示出来，这些就叫作“客户”（Client）。

这样看来，客户机 / 服务器的基本概念并不复杂。这里要注意的一个主要问题是单个服务器需要同时向多个客户提供服务。在这一机制中，通常少不了一套数据库管理系统，使设计人员能将数据布局封装到表格中，以获得最优的使用。除此以外，系统经常允许客户将新信息插入一个服务器。这意味着必须确保客户的新数据不会与其他客户的新数据冲突，或者说需要保证那些数据在加入数据库的时候不会丢失（用数据库的术语来说，这叫作“事务处理”）。客户软件发生了改变之后，它们必须在客户机器上构建、调试以及安装。所有这些会使问题变得比我们一般想象的复杂得多。另外，对多种类型的计算机和操作系统的支持也是一个大问题。最后，性能的问题显得尤为重要：可能会有数百个客户同时向服务器发出请求。所以任何微小的延误都是不能忽视的。为尽可能缓解潜伏的问题，程序员需要谨慎地分散任务的处理负担。一般可以考虑让客户机负担部分处理任务，但有时亦可分派给服务器所在地的其他机器，那些机器亦叫作“中间件”（中间件也用于改进对系统的维护）。

所以在具体实现的时候，其他人发布信息这样一个简单的概念可能变得异常复杂。有时甚至会使人产生完全无从着手的感觉。客户机 / 服务器的概念在这时就可以大显身手了。事实上，大约有一半的程序设计活动都可以采用客户机 / 服务器的结构。这种系统可负责从处理订单及信用卡交易，一直到发布各类数据的方方面面的任务——股票市场、科学研究、政府运作等等。在过去，我们一般为单独的问题采取单独的解决方案；每次都要设计一套新方案。这些方案无论创建还是使用都比较困难，用户每次都要学习和适应新界面。客户机 / 服务器问题需要从根本上加以变革！

2. Web 是一个巨大的服务器

Web 实际就是一套规模巨大的客户机 / 服务器系统。但它的情况要复杂一些，因为所有服务器和客户都同时存在于单个网络上面。但我们没必要了解更进一步的细节，因为唯一要关心的就是一次建立同一个服务器的连接，并同它打交

道（即使可能要在全世界的范围内搜索正确的服务器）。

最开始的时候，这是一个简单的单向操作过程。我们向一个服务器发出请求，它向我们回传一个文件，由于本机的浏览器软件（亦即“客户”或“客户程序”）负责解释和格式化，并在我们面前的屏幕上正确地显示出来。但人们不久就不满足于只从一个服务器传递网页。他们希望获得完全的客户机 / 服务器能力，使客户（程序）也能反馈一些信息到服务器。比如希望对服务器上的数据库进行检索，向服务器添加新信息，或者下一份订单等等（这也提供了比以前的系统更高的安全要求）。在 Web 的发展过程中，我们可以很清晰地看出这些令人心喜的变化。

Web 浏览器的发展终于迈出了重要的一步：某个信息可在任何类型的计算机上显示出来，毋需任何改动。然而，浏览器仍然显得很原始，在用户迅速增多的要求面前显得有些力不从心。它们的交互能力不够强，而且对服务器和因特网都造成了一定程度的干扰。这是由于每次采取一些要求编程的操作时，必须将信息反馈回服务器，在服务器那一端进行处理。所以完全可能需要等待数秒乃至数分钟的时间才会发现自己刚才拼错了一个单词。由于浏览器只是一个纯粹的查看程序，所以连最简单的计算任务都不能进行（当然在另一方面，它也显得非常安全，因为不能在本机上面执行任何程序，避开了程序错误或者病毒的骚扰）。

为解决这个问题，人们采取了许多不同的方法。最开始的时候，人们对图形标准进行了改进，使浏览器能显示更好的动画和视频。为解决剩下的问题，唯一的办法就是在客户端（浏览器）内运行程序。这就叫作“客户端编程”，它是对传统的“服务器端编程”的一个非常重要的拓展。

1.11.2 客户端编程（注释⑧）

Web 最初采用的“服务器－浏览器”方案可提供交互式内容，但这种交互能力完全由服务器提供，为服务器和因特网带来了不小的负担。服务器一般为客户浏览器产生静态网页，由后者简单地解释并显示出来。基本 HTML 语言提供了简单的数据收集机制：文字输入框、复选框、单选钮、列表以及下拉列表等，另外还有一个按钮，只能由程序规定重新设置表单中的数据，以便回传给服务器。用户提交的信息通过所有 Web 服务器均能支持的“通用网关接口”（CGI）回传到服务器。包含在提交数据中的文字指示 CGI 该如何操作。最常见的行动是运行位于服务器的一个程序。那个程序一般保存在一个名为“cgi-bin”的目录中（按下 Web 页内的一个按钮时，请注意一下浏览器顶部的地址窗，经常都能发现“cgi-bin”的字样）。大多数语言都可用来编制这些程序，但其中最常见的是 Perl。这是由于 Perl 是专为文字的处理及解释而设计的，所以能在任何服务器上安装和使用，无论采用的处理器或操作系统是什么。

⑧：本节内容改编自某位作者的一篇文章。那篇文章最早出现在位于 www.mainspring.com 的 Mainspring 上。本节的采用已征得了对方的同意。

今天的许多 Web 站点都严格地建立在 CGI 的基础上，事实上几乎所有事情都可用 CGI 做到。唯一的问题就是响应时间。CGI 程序的响应取决于需要传送多少数据，以及服务器和因特网两方面的负担有多重（而且 CGI 程序的启动比较慢）。Web 的早期设计者并未预料到当初绰绰有余的带宽很快就变得不够用，这正是大量应用充斥网上造成的结果。例如，此时任何形式的动态图形显示都几乎不能连贯地显示，因为此时必须创建一个 GIF 文件，再将图形的每种变化从

服务器传递给客户。而且大家应该对输入表单上的数据校验有着深刻的体会。原来的方法是我们按下网页上的提交按钮（Submit）；数据回传给服务器；服务器启动一个 CGI 程序，检查用户输入是否有错；格式化一个 HTML 页，通知可能遇到的错误，并将这个页回传给我们；随后必须回到原先那个表单页，再输入一遍。这种方法不仅速度非常慢，也显得非常繁琐。

解决的办法就是客户端的程序设计。运行 Web 浏览器的大多数机器都拥有足够强的能力，可进行其他大量工作。与此同时，原始的静态 HTML 方法仍然可以采用，它会一直等到服务器送回下一个页。客户端编程意味着 Web 浏览器可获得更充分的利用，并可有效改善 Web 服务器的交互（互动）能力。

对客户端编程的讨论与常规编程问题的讨论并没有太大的区别。采用的参数肯定是相同的，只是运行的平台不同：Web 浏览器就象一个有限的操作系统。无论如何，我们仍然需要编程，仍然会在客户端编程中遇到大量问题，同时也有很多解决的方案。在本节剩下的部分里，我们将对这些问题进行一番概括，并介绍在客户端编程中采取的对策。

1. 插件

朝客户端编程迈进的时候，最重要的一个问题就是插件的设计。利用插件，程序员可以方便地为浏览器添加新功能，用户只需下载一些代码，把它们“插入”浏览器的适当位置即可。这些代码的作用是告诉浏览器“从现在开始，你可以进行这些新活动了”（仅需下载这些插入一次）。有些快速和功能强大的行为是通过插件添加到浏览器的。但插件的编写并不是一件简单的任务。在我们构建一个特定的站点时，可能并不希望涉及这方面的工作。对客户端程序设计来说，插件的价值在于它允许专业程序员设计出一种新的语言，并将那种语言添加到浏览器，同时不必经过浏览器原创者的许可。由此可以看出，插件实际是浏览器的一个“后门”，允许创建新的客户端程序设计语言（尽管并非所有语言都是作为插件实现的）。

2. 脚本编制语言

插件造成了脚本编制语言的爆炸性增长。通过这种脚本语言，可将用于自己客户端程序的源码直接插入 HTML 页，而对那种语言进行解释的插件会在 HTML 页显示的时候自动激活。脚本语言一般都倾向于尽量简化，易于理解。而且由于它们是从属于 HTML 页的一些简单正文，所以只需向服务器发出对那个页的一次请求，即可非常快地载入。缺点是我们的代码全部暴露在人们面前。另一方面，由于通常不用脚本编制语言做过份复杂的事情，所以这个问题暂且可以放在一边。

脚本语言真正面向的是特定类型问题的解决，其中主要涉及如何创建更丰富、更具有互动能力的图形用户界面（GUI）。然而，脚本语言也许能解决客户端编程中 80％的问题。你碰到的问题可能完全就在那 80％里面。而且由于脚本编制语言的宗旨是尽可能地简化与快速，所以在考虑其他更复杂的方案之前（如 Java 及 ActiveX），首先应想一下脚本语言是否可行。

目前讨论得最多的脚本编制语言包括 JavaScript（它与 Java 没有任何关系；之所以叫那个名字，完全是一种市场策略）、VBScript（同 Visual Basic 很相似）以及 Tcl/Tk（来源于流行的跨平台 GUI 构造语言）。当然还有其他许多语言，也有许多正在开发中。

JavaScript 也许是目常用的，它得到的支持也最全面。无论 NetscapeNavigator，Microsoft Internet Explorer，还是 Opera，目前都提供了对 JavaScript 的支持。除此以外，市面上讲述 JavaScript 的书籍也要比讲述其他语言的书多得多。有些工具还能利用 JavaScript 自动产生网页。当然，如果你已经有 Visual Basic 或者 Tcl/Tk 的深厚功底，当然用它们要简单得多，起码可以避免学习新语言的烦恼（解决 Web 方面的问题就已经够让人头痛了）。

3. Java

如果说一种脚本编制语言能解决 80％的客户端程序设计问题，那么剩下的 20％又该怎么办呢？它们属于一些高难度的问题吗？目前最流行的方案就是 Java。它不仅是一种功能强大、高度安全、可以跨平台使用以及国际通用的程序设计语言，也是一种具有旺盛生命力的语言。对 Java 的扩展是不断进行的，提供的语言特性和库能够很好地解决传统语言不能解决的问题，比如多线程操作、数据库访问、连网程序设计以及分布式计算等等。Java 通过“程序片”（Applet）巧妙地解决了客户端编程的问题。

程序片（或“小应用程序”）是一种非常小的程序，只能在 Web 浏览器中运行。作为 Web 页的一部分，程序片代码会自动下载回来（这和网页中的图片差不多）。激活程序片后，它会执行一个程序。程序片的一个优点体现在：通过程序片，一旦用户需要客户软件，软件就可从服务器自动下载回来。它们能自动取得客户软件的最新版本，不会出错，也没有重新安装的麻烦。由于 Java 的设计原理，程序员只需要创建程序的一个版本，那个程序能在几乎所有计算机以及安装了 Java 解释器的浏览器中运行。由于 Java 是一种全功能的编程语言，所以在向服务器发出一个请求之前，我们能先在客户端做完尽可能多的工作。例如，再也不必通过因特网传送一个请求表单，再由服务器确定其中是否存在一个拼写或者其他参数错误。大多数数据校验工作均可在客户端完成，没有必要坐在计算机前面焦急地等待服务器的响应。这样一来，不仅速度和响应的灵敏度得到了极大的提高，对网络和服务器造成的负担也可以明显减轻，这对保障因特网的畅通是至关重要的。

与脚本程序相比，Java 程序片的另一个优点是它采用编译好的形式，所以客户端看不到源码。当然在另一方面，反编译 Java 程序片也并不是件难事，而且代码的隐藏一般并不是个重要的问题。大家要注意另外两个重要的问题。正如本书以前会讲到的那样，编译好的 Java 程序片可能包含了许多模块，所以要多次“命中”（访问）服务器以便下载（在 Java 1.1 中，这个问题得到了有效的改善——利用 Java 压缩档，即 JAR 文件——它允许设计者将所有必要的模块都封装到一起，供用户统一下载）。在另一方面，脚本程序是作为 Web 页正文的一部分集成到 Web 页内的。这种程序一般都非常小，可有效减少对服务器的点击数。另一个因素是学习方面的问题。不管你平时听别人怎么说，Java 都不是一种十分容易便可学会的语言。如果你以前是一名 Visual Basic 程序员，那么转向 VBScript 会是一种最快捷的方案。由于 VBScript 可以解决大多数典型的客户机 / 服务器问题，所以一旦上手，就很难下定决心再去学习 Java。如果对脚本编制语言比较熟，那么在转向 Java 之前，建议先熟悉一下 JavaScript 或者 VBScript，因为它们可能已经能够满足你的需要，不必经历学习 Java 的艰苦过程。

4. ActiveX

在某种程度上，Java 的一个有力竞争对手应该是微软的 ActiveX，尽管它采用的是完全不同的一套实现机制。ActiveX 最早是一种纯 Windows 的方案。经过一家独立的专业协会的努力，ActiveX 现在已具备了跨平台使用的能力。实际上，ActiveX 的意思是“假如你的程序同它的工作环境正常连接，它就能进入 Web 页，并在支持 ActiveX 的浏览器中运行”（IE 固化了对 ActiveX 的支持，而 Netscape 需要一个插件）。所以，ActiveX 并没有限制我们使用一种特定的语言。比如，假设我们已经是一名有经验的 Windows 程序员，能熟练地使用象 C++、Visual Basic 或者 BorlandDelphi 那样的语言，就能几乎不加任何学习地创建出 ActiveX 组件。事实上，ActiveX 是在我们的 Web 页中使用“历史遗留”代码的最佳途径。

5. 安全

自动下载和通过因特网运行程序听起来就象是一个病毒制造者的梦想。在客户端的编程中，ActiveX 带来了最让人头痛的安全问题。点击一个 Web 站点的时候，可能会随同 HTML 网页传回任何数量的东西：GIF 文件、脚本代码、编译好的 Java 代码以及 ActiveX 组件。有些是无害的；GIF 文件不会对我们造成任何危害，而脚本编制语言通常在自己可做的事情上有着很大的限制。Java 也设计成在一个安全“沙箱”里在它的程序片中运行，这样可防止操作位于沙箱以外的磁盘或者内存区域。

ActiveX 是所有这些里面最让人担心的。用 ActiveX 编写程序就象编制 Windows 应用程序——可以做自己想做的任何事情。下载回一个 ActiveX 组件后，它完全可能对我们磁盘上的文件造成破坏。当然，对那些下载回来并不限于在 Web 浏览器内部运行的程序，它们同样也可能破坏我们的系统。从 BBS 下载回来的病毒一直是个大问题，但因特网的速度使得这个问题变得更加复杂。

目前解决的办法是“数字签名”，代码会得到权威机构的验证，显示出它的作者是谁。这一机制的基础是认为病毒之所以会传播，是由于它的编制者匿名的缘故。所以假如去掉了匿名的因素，所有设计者都不得不为它们的行为负责。这似乎是一个很好的主意，因为它使程序显得更加正规。但我对它能消除恶意因素持怀疑态度，因为假如一个程序便含有 Bug，那么同样会造成问题。

Java 通过“沙箱”来防止这些问题的发生。Java 解释器内嵌于我们本地的 Web 浏览器中，在程序片装载时会检查所有有嫌疑的指令。特别地，程序片根本没有权力将文件写进磁盘，或者删除文件（这是病毒最喜欢做的事情之一）。我们通常认为程序片是安全的。而且由于安全对于营建一套可靠的客户机 / 服务器系统至关重要，所以会给病毒留下漏洞的所有错误都能很快得到修复（浏览器软件实际需要强行遵守这些安全规则；而有些浏览器则允许我们选择不同的安全级别，防止对系统不同程度的访问）。

大家或许会怀疑这种限制是否会妨碍我们将文件写到本地磁盘。比如，我们有时需要构建一个本地数据库，或将数据保存下来，以便日后离线使用。最早的版本似乎每个人都能在线做任何敏感的事情，但这很快就变得非常不现实（尽管低价“互联网工具”有一天可能会满足大多数用户的需要）。解决的方案是“签了名的程序片”，它用公共密钥加密算法验证程序片确实来自它所声称的地方。当然在通过验证后，签了名的一个程序片仍然可以开始清除你的磁盘。但从理论上说，既然现在能够找到创建人“算帐”，他们一般不会干这种蠢事。Java 1.1 为数字签名提供了一个框架，在必要时，可让一个程序片“走”到沙箱的外面来。

数字签名遗漏了一个重要的问题，那就是人们在因特网上移动的速度。如下

载回一个错误百出的程序，而它很不幸地真的干了某些蠢事，需要多久的时间才能发觉这一点呢？这也许是几天，也可能几周之后。发现了之后，又如何追踪当初肇事的程序呢（以及它当时的责任有多大）？

6. 因特网和内联网

Web 是解决客户机 / 服务器问题的一种常用方案，所以最好能用相同的技术解决此类问题的一些“子集”，特别是公司内部的传统客户机 / 服务器问题。对于传统的客户机 / 服务器模式，我们面临的问题是拥有多种不同类型的客户计算机，而且很难安装新的客户软件。但通过 Web 浏览器和客户端编程，这两类问题都可得到很好的解决。若一个信息网络局限于一家特定的公司，那么在将 Web 技术应用于它之后，即可称其为“内联网”（Intranet），以示与国际性的“因特网”（Internet）有别。内联网提供了比因特网更大的安全级别，因为可以物理性地控制对公司内部服务器的使用。说到培训，一般只要人们理解了浏览器的常规概念，就可以非常轻松地掌握网页和程序片之间的差异，所以学习新型系统的开销会大幅度减少。

安全问题将我们引入客户端编程领域一个似乎是自动形成的分支。若程序是在因特网上运行，由于无从知晓它会在什么平台上运行，所以编程时要特别留意，防范可能出现的编程错误。需作一些跨平台处理，以及适当的安全防范，比如采用某种脚本语言或者 Java。

但假如在内联网中运行，面临的一些制约因素就会发生变化。全部机器均为 Intel/Windows 平台是件很平常的事情。在内联网中，需要对自己代码的质量负责。而且一旦发现错误，就可以马上改正。除此以外，可能已经有了一些“历史遗留”的代码，并用较传统的客户机 / 服务器方式使用那些代码。但在进行升级时，每次都要物理性地安装一道客户程序。浪费在升级安装上的时间是转移到浏览器的一项重要原因。使用了浏览器后，升级就变得易如反掌，而且整个过程是透明和自动进行的。如果真的是牵涉到这样的一个内联网中，最明智的方法是采用 ActiveX，而非试图采用一种新的语言来改写程序代码。

面临客户端编程问题令人困惑的一系列解决方案时，最好的方案是先做一次投资 / 回报分析。请总结出问题的全部制约因素，以及什么才是最快的方案。由于客户端程序设计仍然要编程，所以无论如何都该针对自己的特定情况采取最好的开发途径。这是准备面对程序开发中一些不可避免的问题时，我们可以作出的最佳姿态。

1.11.3 服务器端编程

我们的整个讨论都忽略了服务器端编程的问题。如果向服务器发出一个请求，会发生什么事情？大多数时候的请求都是很简单的一个“把这个文件发给我”。浏览器随后会按适当的形式解释这个文件：作为 HTML 页、一幅图、一个 Java 程序片、一个脚本程序等等。向服务器发出的较复杂的请求通常涉及到对一个数据库进行操作（事务处理）。其中最常见的就是发出一个数据库检索命令，得到结果后，服务器会把它格式化成 HTML 页，并作为结果传回来（当然，假如客户通过 Java 或者某种脚本语言具有了更高的智能，那么原始数据就能在客户端发送和格式化；这样做速度可以更快，也能减轻服务器的负担）。另外，有时需要在数据库中注册自己的名字（比如加入一个组时），或者向服务器发出一份订单，这就涉及到对那个数据库的修改。这类服务器请求必须通过服务器端的

一些代码进行，我们称其为“服务器端的编程”。在传统意义上，服务器端编程是用 Perl 和 CGI 脚本进行的，但更复杂的系统已经出现。其中包括基于 Java 的 Web 服务器，它允许我们用 Java 进行所有服务器端编程，写出的程序就叫作“小服务程序”（Servlet）。

1.11.4 一个独立的领域：应用程序

与 Java 有关的大多数争论都是与程序片有关的。Java 实际是一种常规用途的程序设计语言，可解决任何类型的问题，至少理论上如此。而且正如前面指出的，可以用更有效的方式来解决大多数客户机 / 服务器问题。如果将视线从程序片身上转开（同时放宽一些限制，比如禁止写盘等），就进入了常规用途的应用程序的广阔领域。这种应用程序可独立运行，毋需浏览器，就象普通的执行程序那样。在这儿，Java 的特色并不仅仅反应在它的移植能力，也反映在编程本身上。就象贯穿全书都会讲到的那样，Java 提供了许多有用的特性，使我们能在较短的时间里创建出比用从前的程序设计语言更健壮的程序。

但要注意任何东西都不是十全十美的，我们为此也要付出一些代价。其中最明显的是执行速度放慢了（尽管可对此进行多方面的调整）。和任何语言一样，Java 本身也存在一些限制，使得它不十分适合解决某些特殊的编程问题。但不管怎样，Java 都是一种正在快速发展的语言。随着每个新版本的发布，它变得越来越可爱，能充分解决的问题也变得越来越多。

1.12 分析和设计

面向对象的范式是思考程序设计时一种新的、而且全然不同的方式，许多人最开始都会在如何构造一个项目上皱起了眉头。事实上，我们可以作出一个“好”的设计，它能充分利用 OOP 提供的所有优点。

有关 OOP 分析与设计的书籍大多数都不尽如人意。其中的大多数书都充斥着莫名其妙的话语、笨拙的笔调以及许多听起来似乎很重要的声明（注释⑨）。我认为这种书最好压缩到一章左右的空间，至多写成一本非常薄的书。具有讽刺意味的是，那些特别专注于复杂事物管理的人往往在写一些浅显、明白的书上面大费周章！如果不能说得简单和直接，一定没多少人喜欢看这方面的内容。毕竟，OOP 的全部宗旨就是让软件开发的过程变得更加容易。尽管这可能影响了那些喜欢解决复杂问题的人的生计，但为什么不从一开始就把事情弄得简单些呢？因此，希望我能从开始就为大家打下一个良好的基础，尽可能用几个段落来说清楚分析与设计的问题。

⑨：最好的入门书仍然是 Grady Booch 的《Object-Oriented Design withApplications，第 2 版本》，Wiely & Sons 于 1996 年出版。这本书讲得很有深度，而且通俗易懂，尽管他的记号方法对大多数设计来说都显得不必要地复杂。

1.12.1 不要迷失

在整个开发过程中，最重要的事情就是：不要将自己迷失！但事实上这种事情很容易发生。大多数方法都设计用来解决最大范围内的问题。当然，也存在一些特别困难的项目，需要作者付出更为艰辛的努力，或者付出更大的代价。但是，大多数项目都是比较“常规”的，所以一般都能作出成功的分析与设计，而且只需用到推荐的一小部分方法。但无论多么有限，某些形式的处理总是有益的，这

可使整个项目的开发更加容易，总比直接了当开始编码好！

也就是说，假如你正在考察一种特殊的方法，其中包含了大量细节，并推荐了许多步骤和文档，那么仍然很难正确判断自己该在何时停止。时刻提醒自己注意以下几个问题：

(1) 对象是什么？（怎样将自己的项目分割成一系列单独的组件？）

(2) 它们的接口是什么？（需要将什么消息发给每一个对象？）

在确定了对象和它们的接口后，便可着手编写一个程序。出于对多方面原因的考虑，可能还需要比这更多的说明及文档，但要求掌握的资料绝对不能比这还少。

整个过程可划分为四个阶段，阶段 0 刚刚开始采用某些形式的结构。

1.12.2 阶段 0：拟出一个计划

第一步是决定在后面的过程中采取哪些步骤。这听起来似乎很简单（事实上，我们这儿说的一切都似乎很简单），但很常见的一种情况是：有些人甚至没有进入阶段 1，便忙忙慌慌地开始编写代码。如果你的计划本来就是“直接开始开始编码”，那样做当然也无可非议（若对自己要解决的问题已有很透彻的理解，便可考虑那样做）。但最低程度也应同意自己该有个计划。

在这个阶段，可能要决定一些必要的附加处理结构。但非常不幸，有些程序员写程序时喜欢随心所欲，他们认为“该完成的时候自然会完成”。这样做刚开始可能不会有什么问题，但我觉得假如能在整个过程中设置几个标志，或者“路标”，将更有益于你集中注意力。这恐怕比单纯地为了“完成工作”而工作好得多。至少，在达到了一个又一个的目标，经过了一个接一个的路标以后，可对自己的进度有清晰的把握，干劲也会相应地提高，不会产生“路遥漫漫无期”的感觉。

座我刚开始学习故事结构起（我想有一天能写本小说出来），就一直坚持这种做法，感觉就象简单地让文字“流”到纸上。在我写与计算机有关的东西时，发现结构要比小说简单得多，所以不需要考虑太多这方面的问题。但我仍然制订了整个写作的结构，使自己对要写什么做到心中有数。因此，即使你的计划就是直接开始写程序，仍然需要经历以下的阶段，同时向自己提出一些特定的问题。

1.12.3 阶段 1：要制作什么？

在上一代程序设计中（即“过程化或程序化设计”），这个阶段称为“建立需求分析和系统规格”。当然，那些操作今天已经不再需要了，或者至少改换了形式。大量令人头痛的文档资料已成为历史。但当时的初衷是好的。需求分析的意思是“建立一系列规则，根据它判断任务什么时候完成，以及客户怎样才能满意”。系统规格则表示“这里是一些具体的说明，让你知道程序需要做什么（而不是怎样做）才能满足要求”。需求分析实际就是你和客户之间的一份合约（即使客户就在本公司内部工作，或者是其他对象及系统）。系统规格是对所面临问题的最高级别的一种揭示，我们依据它判断任务是否完成，以及需要花多长的时间。由于这些都需要取得参与者的一致同意，所以我建议尽可能地简化它们——最好采用列表和基本图表的形式——以节省时间。可能还会面临另一些限制，需要把它们扩充成为更大的文档。

我们特别要注意将重点放在这一阶段的核心问题上，不要纠缠于细枝末节。这个核心问题就是：决定采用什么系统。对这个问题，最有价值的工具就是一个

名为“使用条件”的集合。对那些采用“假如……，系统该怎样做？”形式的问题，这便是最有说服力的回答。例如，“假如客户需要提取一张现金支票，但当时又没有这么多的现金储备，那么自动取款机该怎样反应？”对这个问题，“使用条件”可以指示自动取款机在那种“条件”下的正确操作。

应尽可能总结出自己系统的一套完整的“使用条件”或者“应用场合”。一旦完成这个工作，就相当于摸清了想让系统完成的核心任务。由于将重点放在“使用条件”上，一个很好的效果就是它们总能让你放精力放在最关键的东西上，并防止自己分心于对完成任务关系不大的其他事情上面。也就是说，只要掌握了一套完整的“使用条件”，就可以对自己的系统作出清晰的描述，并转移到下一个阶段。在这一阶段，也有可能无法完全掌握系统日后的各种应用场合，但这也没有关系。只要肯花时间，所有问题都会自然而然暴露出来。不要过份在意系统规格的“完美”，否则也容易产生挫败感和焦燥情绪。

在这一阶段，最好用几个简单的段落对自己的系统作出描述，然后围绕它们再进行扩充，添加一些“名词”和“动词”。“名词”自然成为对象，而“动词”自然成为要整合到对象接口中的“方法”。只要亲自试着做一做，就会发现这是多么有用的一个工具；有些时候，它能帮助你完成绝大多数的工作。

尽管仍处在初级阶段，但这时的一些日程安排也可能会非常管用。我们现在对自己要构建的东西应该有了一个较全面的认识，所以可能已经感觉到了它大概会花多长的时间来完成。此时要考虑多方面的因素：如果估计出一个较长的日程，那么公司也许决定不再继续下去；或者一名主管已经估算出了这个项目要花多长的时间，并会试着影响你的估计。但无论如何，最好从一开始就草拟出一份“诚实”的时间表，以后再进行一些暂时难以作出的决策。目前有许多技术可帮助我们计算出准确的日程安排（就象那些预测股票市场起落的技术），但通常最好的方法还是依赖自己的经验和直觉（不要忘记，直觉也要建立在经验上）。感觉一下大概需要花多长的时间，然后将这个时间加倍，再加上 10%。你的感觉可能是正确的；“也许”能在那个时间里完成。但“加倍”使那个时间更加充裕，“10%”的时间则用于进行最后的推敲和深化。但同时也要对此向上级主管作出适当的解释，无论对方有什么抱怨和修改，只要明确地告诉他们：这样的一个日程安排，只是我的一个估计！

1.12.4 阶段 2：如何构建？

在这一阶段，必须拿出一套设计方案，并解释其中包含的各类对象在外观上是什么样子，以及相互间是如何沟通的。此时可考虑采用一种特殊的图表工具：“统一建模语言”（UML）。请到 http://www.rational.com 去下载一份 UML 规格书。作为第 1 阶段中的描述工具，UML 也是很有帮助的。此外，还可用它在第 2 阶段中处理一些图表（如流程图）。当然并非一定要使用 UML，但它对你会很有帮助，特别是在希望描绘一张详尽的图表，让许多人在一起研究的时候。除 UML 外，还可选择对对象以及它们的接口进行文字化描述（就象我在《Thinking in C++》里说的那样，但这种方法非常原始，发挥的作用亦较有限。

我曾有一次非常成功的咨询经历，那时涉及到一小组人的初始设计。他们以前还没有构建过 OOP（面向对象程序设计）项目，将对象画在白板上面。我们谈到各对象相互间该如何沟通（通信），并删除了其中的一部分，以及替换了另一部分对象。这个小组（他们知道这个项目的目的是什么）实际上已经制订出了设计方案；他们自己“拥有”了设计，而不是让设计自然而然地显露出来。我在

那里做的事情就是对设计进行指导，提出一些适当的问题，尝试作出一些假设，并从小组中得到反馈，以便修改那些假设。这个过程中最美妙的事情就是整个小组并不是通过学习一些抽象的例子来进行面向对象的设计，而是通过实践一个真正的设计来掌握 OOP 的窍门，而那个设计正是他们当时手上的工作！

作出了对对象以及它们的接口的说明后，就完成了第 2 阶段的工作。当然，这些工作可能并不完全。有些工作可能要等到进入阶段 3 才能得知。但这已经足够了。我们真正需要关心的是最终找出所有的对象。能早些发现当然好，但 OOP 提供了足够完美的结构，以后再找出它们也不迟。

1.12.5 阶段 3：开始创建

读这本书的可能是程序员，现在进入的正是你可能最感兴趣的阶段。由于手头上有一个计划——无论它有多么简要，而且在正式编码前掌握了正确的设计结构，所以会发现接下去的工作比一开始就埋头写程序要简单得多。而这正是我们想达到的目的。让代码做到我们想做的事情，这是所有程序项目最终的目标。但切不要急功冒进，否则只有得不偿失。根据我的经验，最后先拿出一套较为全面的方案，使其尽可能设想周全，能满足尽可能多的要求。给我的感觉，编程更象一门艺术，不能只是作为技术活来看待。所有付出最终都会得到回报。作为真正的程序员，这并非可有可无的一种素质。全面的思考、周密的准备、良好的构造不仅使程序更易构建与调试，也使其更易理解和维护，而那正是一套软件赢利的必要条件。

构建好系统，并令其运行起来后，必须进行实际检验，以前做的那些需求分析和系统规格便可派上用场了。全面地考察自己的程序，确定提出的所有要求均已满足。现在一切似乎都该结束了？是吗？

1.12.6 阶段 4：校订

事实上，整个开发周期还没有结束，现在进入的是传统意义上称为“维护”的一个阶段。“维护”是一个比较暧昧的称呼，可用它表示从“保持它按设想的轨道运行”、“加入客户从前忘了声明的功能”或者更传统的“除掉暴露出来的一切臭虫”等等意思。所以大家对“维护”这个词产生了许多误解，有的人认为：凡是需要“维护”的东西，必定不是好的，或者是有缺陷的！因为这个词说明你实际构建的是一个非常“原始”的程序，以后需要频繁地作出改动、添加新的代码或者防止它的落后、退化等。因此，我们需要用一个更合理的词语来称呼以后需要继续的工作。

这个词便是“校订”。换言之，“你第一次做的东西并不完善，所以需为自己留下一个深入学习、认知的空间，再回过头去作一些改变”。对于要解决的问题，随着对它的学习和了解愈加深入，可能需要作出大量改动。进行这些工作的一个动力是随着不断的改革优化，终于能够从自己的努力中得到回报，无论这需要经历一个较短还是较长的时期。

什么时候才叫“达到理想的状态”呢？这并不仅仅意味着程序必须按要求的那样工作，并能适应各种指定的“使用条件”，它也意味着代码的内部结构应当尽善尽美。至少，我们应能感觉出整个结构都能良好地协调运作。没有笨拙的语法，没有臃肿的对象，也没有一些华而不实的东西。除此以外，必须保证程序结构有很强的生命力。由于多方面的原因，以后对程序的改动是必不可少。但必须确定改动能够方便和清楚地进行。这里没有花巧可言。不仅需要理解自己构建的

是什么，也要理解程序如何不断地进化。幸运的是，面向对象的程序设计语言特别适合进行这类连续作出的修改——由对象建立起来的边界可有效保证结构的整体性，并能防范对无关对象进行的无谓干扰、破坏。也可以对自己的程序作一些看似激烈的大变动，同时不会破坏程序的整体性，不会波及到其他代码。事实上，对“校订”的支持是 OOP 非常重要的一个特点。

通过校订，可创建出至少接近自己设想的东西。然后从整体上观察自己的作品，把它与自己的要求比较，看看还短缺什么。然后就可以从容地回过头去，对程序中不恰当的部分进行重新设计和重新实现（注释⑩）。在最终得到一套恰当的方案之前，可能需要解决一些不能回避的问题，或者至少解决问题的一个方面。而且一般要多“校订”几次才行（“设计范式”在这里可起到很大的帮助作用。有关它的讨论，请参考本书第 16 章）。

构建一套系统时，“校订”几乎是不可避免的。我们需要不断地对比自己的需求，了解系统是否自己实际所需要的。有时只有实际看到系统，才能意识到自己需要解决一个不同的问题。若认为这种形式的校订必然会发生，那么最好尽快拿出自己的第一个版本，检查它是否自己希望的，使自己的思想不断趋向成熟。

反复的“校订”同“递增开发”有关密不可分的关系。递增开发意味着先从系统的核心入手，将其作为一个框架实现，以后要在这个框架的基础上逐渐建立起系统剩余的部分。随后，将准备提供的各种功能（特性）一个接一个地加入其中。这里最考验技巧的是架设起一个能方便扩充所有目标特性的一个框架（对这个问题，大家可参考第 16 章的论述）。这样做的好处在于一旦令核心框架运作起来，要加入的每一项特性就象它自身内的一个小项目，而非大项目的一部分。此外，开发或维护阶段合成的新特性可以更方便地加入。OOP 之所以提供了对递增开发的支持，是由于假如程序设计得好，每一次递增都可以成为完善的对象或者对象组。

⑩：这有点类似“快速造型”。此时应着眼于建立一个简单、明了的版本，使自己能对系统有个清楚的把握。再把这个原型扔掉，并正式地构建一个。快速造型最麻烦的一种情况就是人们不将原型扔掉，而是直接在它的基础上建造。如果再加上程序化设计中“结构”的缺乏，就会导致一个混乱的系统，致使维护成本增加。

1.12.7 计划的回报

如果没有仔细拟定的设计图，当然不可能建起一所房子。如建立的是一所狗舍，尽管设计图可以不必那么详尽，但仍然需要一些草图，以做到心中有数。软件开发则完全不同，它的“设计图”（计划）必须详尽而完备。在很长的一段时间里，人们在他们的开发过程中并没有太多的结构，但那些大型项目很容易就会遭致失败。通过不断的摸索，人们掌握了数量众多的结构和详细资料。但它们的使用却使人提心吊胆在意——似乎需要把自己的大多数时间花在编写文档上，而没有多少时间来编程（经常如此）。我希望这里为大家讲述的一切能提供一条折衷的道路。需要采取一种最适合自己需要（以及习惯）的方法。不管制订出的计划有多么小，但与完全没有计划相比，一些形式的计划会极大改善你的项目。请记住：根据估计，没有计划的 50％以上的项目都会失败！

1.13 Java 还是 C++?

Java 特别象 C++；由此很自然地会得出一个结论：C++似乎会被 Java 取代。但我对这个逻辑存有一些疑问。无论如何，C++仍有一些特性是 Java 没有的。而且尽管已有大量保证，声称 Java 有一天会达到或超过 C++的速度。但这个突破迄今仍未实现（尽管 Java 的速度确实在稳步提高，但仍未达到 C++的速度）。此外，许多领域都存在为数众多的 C++爱好者，所以我并不认为那种语言很快就会被另一种语言替代（爱好者的力量是容忽视的。比如在我主持的一次“中／高级 Java 研讨会”上，Allen Holub 声称两种最常用的语言是 Rexx 和 COBOL）。

我感觉 Java 强大之处反映在与 C++稍有不同的领域。C++是一种绝对不会试图迎合某个模子的语言。特别是它的形式可以变化多端，以解决不同类型的问题。这主要反映在象 Microsoft Visual C++和 Borland C++ Builder（我最喜欢这个）那样的工具身上。它们将库、组件模型以及代码生成工具等合成到一起，以开发视窗化的末端用户应用（用于 Microsoft Windows 操作系统）。但在另一方面，Windows 开发人员最常用的是什么呢？是微软的 Visual Basic（VB）。当然，我们在这儿暂且不提 VB 的语法极易使人迷惑的事实——即使一个只有几页长度的程序，产生的代码也十分难于管理。从语言设计的角度看，尽管 VB 是那样成功和流行，但仍然存在不少的缺点。最好能够同时拥有 VB 那样的强大功能和易用性，同时不要产生难于管理的代码。而这正是 Java 最吸引人的地方：作为“下一代的 VB”。无论你听到这种主张后有什么感觉，请无论如何都仔细想一想：人们对 Java 做了大量的工作，使它能方便程序员解决应用级问题（如连网和跨平台 UI 等），所以它在本质上允许人们创建非常大型和灵活的代码主体。同时，考虑到 Java 还拥有我迄今为止尚未在其他任何一种语言里见到的最“健壮”的类型检查及错误控制系统，所以 Java 确实能大大提高我们的编程效率。这一点是勿庸置疑的！

但对于自己某个特定的项目，真的可以不假思索地将 C++换成 Java 吗？除了 Web 程序片，还有两个问题需要考虑。首先，假如要使用大量现有的库（这样肯定可以提高不少的效率），或者已经有了一个坚实的 C 或 C++代码库，那么换成 Java 后，反映会阻碍开发进度，而不是加快它的速度。但若想从头开始构建自己的所有代码，那么 Java 的简单易用就能有效地缩短开发时间。

最大的问题是速度。在原始的 Java 解释器中，解释过的 Java 会比 C 慢上 20 到 50 倍。尽管经过长时间的发展，这个速度有一定程度的提高，但和 C 比起来仍然很悬殊。计算机最注重的就是速度；假如在一台计算机上不能明显较快地干活，那么还不如用手做（有人建议在开发期间使用 Java，以缩短开发时间。然后用一个工具和支撑库将代码转换成 C++，这样可获得更快的执行速度）。

为使 Java 适用于大多数 Web 开发项目，关键在于速度上的改善。此时要用到人们称为“刚好及时”（Just-In Time，或 JIT）的编译器，甚至考虑更低级的代码编译器（写作本书时，也有两款问世）。当然，低级代码编译器会使编译好的程序不能跨平台执行，但同时也带来了速度上的提升。这个速度甚至接近 C 和 C++。而且 Java 中的程序交叉编译应当比 C 和 C++中简单得多（理论上只需重编译即可，但实际仍较难实现；其他语言也曾作出类似的保证）。

在本书附录，大家可找到与 Java／C++比较．对 Java 现状的观察以及编码规则有关的内容。