## 对象入门

### 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。

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。

### 1.2对象的接口

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：

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

访问控制原因：

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。

1. 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。
2. 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。

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

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。

**注意**：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

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。

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**

**等价关系：**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。在某种意义上，这是进行继承的一种理想方式。

**类似关系：**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。

### 1.6多形对象的互换使用

假设我们用Java 写了这样一个函数：

void doStuff(Shape s) {

s.erase();

// ...

s.draw();

}

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：

Circle c = new Circle();

Triangle t = new Triangle();

Line l = new Line();

doStuff(c);

doStuff(t);

doStuff(l);

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。

**上溯类型：**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。

**1.6.1动态绑定**

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。

**动态绑定与多形性：**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；

**1.6.2抽象的基础类和接口**

1. 抽象方法只能在一个抽象类里创建
2. 继承了一个抽象类后，那个方法就必须实现，否则继承的类也会变成“抽象”类。
3. 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。
4. 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

**1.7.1集合与继承器**

集合：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。

“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。

**矢量**（Vector）和**列表**（List）的区别：

**相同：**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。

**不同：**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。（**随机访问矢量强，插入列表强**）

**1.7.2单根结构**

所有类都从单独一个基础类继承。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**：

（1）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。

（2）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）。

（3）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。

（4）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。

**1.7.3集合库与方便适用集合**

（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。

**使用：**只需添加指向容纳了Object 的一个集合的对象句柄即可，以后可以通过句柄重新使用对象。但由于集合只能容纳Object，

**上溯造型：**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。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

**下溯造型：**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）。下溯造型中我们不知道一个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。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。

**缺点：**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。

**C++解决办法：**创建一个“智能”集合，令其知道自己容纳的类型，消除下溯造型的必要以及潜在的错误。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暂时没有解决办法。

**1.7.4清除时的困境：由谁负责清除？**

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

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）。

### 1.8违例控制：解决错误

**背景**：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。

**方法：**“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。

**优点：**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。

**不同：**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

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。

**多线程处理难点：资源共享。**

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多线程：**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

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。

### 1.11java和因特网

**1.1.1什么是web**

（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”。**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）。

1. 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在于单个网络上面。

**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。

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）。

这个问题的解决办法就是在客户端（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**

Web **最初**采用的“服务器－浏览器”方案可提供交互式内容，这种交互能力**完全由服务器提供**：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）。

**出现背景：**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。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。

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。

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）。

1. **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。

**优缺点：**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。

**目标：**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。

**（3）java**

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。

**描述：**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**优点**体现在：

**1.**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。

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。

**2.**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。

**3.**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。

**4.**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。

**（4）ActiveX**

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。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。

1. **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。

**目前解决办法：**

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的解决办法：**

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）。

1. 因特网和互联网

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。

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。