附录 B 对比 C++和 Java

“作为一名 C++程序员，我们早已掌握了面向对象程序设计的基本概念，而且 Java 的语法无疑是非常熟悉的。事实上，Java 本来就是从 C++衍生出来的。”

然而，C++和 Java 之间仍存在一些显著的差异。可以这样说，这些差异代表着技术的极大进步。一旦我们弄清楚了这些差异，就会理解为什么说 Java 是一种优秀的程序设计语言。本附录将引导大家认识用于区分 Java 和 C++的一些重要特征。

(1) 最大的障碍在于速度：解释过的 Java 要比 C 的执行速度慢上约 20 倍。无论什么都不能阻止 Java 语言进行编译。写作本书的时候，刚刚出现了一些准实时编译器，它们能显著加快速度。当然，我们完全有理由认为会出现适用于更多流行平台的纯固有编译器，但假若没有那些编译器，由于速度的限制，必须有些问题是 Java 不能解决的。

(2) 和 C++一样，Java 也提供了两种类型的注释。

(3) 所有东西都必须置入一个类。不存在全局函数或者全局数据。如果想获得与全局函数等价的功能，可考虑将 static 方法和 static 数据置入一个类里。注意没有象结构、枚举或者联合这一类的东西，**一切只有“类”（Class）！**

(4) 所有方法都是在类的主体定义的。所以用 C++的眼光看，似乎所有函数都已嵌入，但实情并非如何（嵌入的问题在后面讲述）。

(5) 在 Java 中，类定义采取几乎和 C++一样的形式。但没有标志结束的分号。没有 class foo 这种形式的类声明，只有类定义。

```
class aType()
```

```
void aMethod() {/*  方法主体  */}
}
```

(6) Java 中没有作用域范围运算符“::”。Java 利用点号做所有的事情，但可以不用考虑它，因为只能在一个类里定义元素。即使那些方法定义，也必须在一个类的内部，所以根本没有必要指定作用域的范围。我们注意到的一项差异是对 static 方法的调用：使用 ClassName.methodName()。除此以外，package（包）的名字是用点号建立的，并能用 import 关键字实现 C++的“#include”的一部分功能。例如下面这个语句：

```
import java.awt.*;
```

（#include 并不直接映射成 import，但在使用时有类似的感觉。）

(7) 与 C++类似，Java 含有一系列“主类型”（Primitive type），以实现更有效率的访问。在 Java 中，这些类型包括 boolean，char，byte，short，int，long，float 以及 double。所有主类型的大小都是固有的，且与具体的机器无关（考虑到移植的问题）。这肯定会对性能造成一定的影响，具体取决于不同的机器。对类型的检查和要求在 Java 里变得更苛刻。例如：

■条件表达式只能是 boolean（布尔）类型，不可使用整数。

■必须使用象 X+Y 这样的一个表达式的结果；不能仅仅用“X+Y”来实现“副作用”。

(8) char（字符）类型使用国际通用的 **16 位 Unicode** 字符集，所以能自动表达大多数国家的字符。

(9) 静态引用的字串会自动转换成 String 对象。和 C 及 C++不同，没有独立的静态字符数组字串可供使用。

(10) Java 增添了三个右移位运算符“>>>”，具有与“逻辑”右移位运算符类似的功用，可在最末尾插入零值。“>>”则会在移位的同时插入符号位（即“算术”移位）。

(11) 尽管表面上类似，但与 C++相比，Java 数组采用的是一个颇为不同的结构，并具有独特的行为。有一个只读的 length 成员，通过它可知道数组有多大。而且一旦超过数组边界，运行期检查会自动丢弃一个异常。所有数组都是在内存“堆”里创建的，我们可将一个数组分配给另一个（只是简单地复制数组句柄）。数组标识符属于第一级对象，它的所有方法通常都适用于其他所有对象。

(12) 对于所有不属于主类型的对象，都只能通过 new 命令创建。和 C++不同，Java 没有相应的命令可以“在堆栈上”创建不属于主类型的对象。所有主类型都只能在堆栈上创建，同时不使用 new 命令。所有主要的类都有自己的“封装（器）”类，所以能够通过 new 创建等价的、以内存“堆”为基础的对象（主类型数组是一个例外：它们可象 C++那样通过集合初始化进行分配，或者使用 new）。

(13) Java 中不必进行提前声明。若想在定义前使用一个类或方法，只需直接使用它即可——编译器会保证使用恰当的定义。所以和在 C++中不同，我们不会碰到任何涉及提前引用的问题。

(14) Java 没有预处理机。若想使用另一个库里的类，只需使用 import 命令，并指定库名即可。不存在类似于预处理机的宏。

(15) Java 用包代替了命名空间。由于将所有东西都置入一个类，而且由于采用了一种名为“封装”的机制，它能针对类名进行类似于命名空间分解的操作，

所以命名的问题不再进入我们的考虑之列。数据包也会在单独一个库名下收集库的组件。我们只需简单地“import”（导入）一个包，剩下的工作会由编译器自动完成。

(16) 被定义成类成员的对象句柄会自动初始化成 null。对基本类数据成员的初始化在 Java 里得到了可靠的保障。若不明确地进行初始化，它们就会得到一个默认值（零或等价的值）。可对它们进行明确的初始化（显式初始化）：要么在类内定义它们，要么在构建器中定义。采用的语法比 C++的语法更容易理解，而且对于 static 和非 static 成员来说都是固定不变的。我们不必从外部定义 static 成员的存储方式，这和 C++是不同的。

(17) 在 Java 里，没有象 C 和 C++那样的指针。用 new 创建一个对象的时候，会获得一个引用（本书一直将其称作“句柄”）。例如：

```
String s = new String("howdy");
```

然而，C++引用在创建时必须进行初始化，而且不可重定义到一个不同的位置。但 Java 引用并不一定局限于创建时的位置。它们可根据情况任意定义，这便消除了对指针的部分需求。在 C 和 C++里大量采用指针的另一个原因是为了能指向任意一个内存位置（这同时会使它们变得不安全，也是 Java 不提供这一支持的原因）。指针通常被看作在基本变量数组中四处移动的一种有效手段。Java 允许我们以更安全的形式达到相同的目标。解决指针问题的终极方法是“固有方法”（已在附录 A 讨论）。将指针传递给方法时，通常不会带来太大的问题，因为此时没有全局函数，只有类。而且我们可传递对对象的引用。Java 语言最开始声称自己“完全不采用指针！”但随着许多程序员都质问没有指针如何工作？于是后来又声明“采用受到限制的指针”。大家可自行判断它是否“真”的是一个指针。但不管在何种情况下，都不存在指针“算术”。

(18) Java 提供了与 C++类似的“构建器”（Constructor）。如果不自己定义一个，就会获得一个默认构建器。而如果定义了一个非默认的构建器，就不会为我们自动定义默认构建器。这和 C++是一样的。注意没有复制构建器，因为所有自变量都是按引用传递的。

(19) Java 中没有“破坏器”（Destructor）。变量不存在“作用域”的问题。一个对象的“存在时间”是由对象的存在时间决定的，并非由垃圾收集器决定。有个 finalize()方法是每一个类的成员，它在某种程度上类似于 C++的“破坏器”。但 finalize()是由垃圾收集器调用的，而且只负责释放“资源”（如打开的文件、套接字、端口、URL 等等）。如需在一个特定的地点做某样事情，必须创建一个特殊的方法，并调用它，不能依赖 finalize()。而在另一方面，C++中的所有对象都会（或者说“应该”）破坏，但并非 Java 中的所有对象都会被当作“垃圾”收集掉。由于 Java 不支持破坏器的概念，所以在必要的时候，必须谨慎地创建一个清除方法。而且针对类内的基础类以及成员对象，需要明确调用所有清除方法。

(20) Java 具有方法“过载”机制，它的工作原理与 C++函数的过载几乎是完全相同的。

(21) Java 不支持默认自变量。

(22) Java 中没有 goto。它采取的无条件跳转机制是“break 标签”或者“continue 标准”，用于跳出当前的多重嵌套循环。

(23) Java 采用了一种单根式的分级结构，因此所有对象都是从根类 Object 统一继承的。而在 C++中，我们可在任何地方启动一个新的继承树，所以最后往往看到包含了大量树的“一片森林”。在 Java 中，我们无论如何都只有一个分级

结构。尽管这表面上看似乎造成了限制，但由于我们知道每个对象肯定至少有一个 Object 接口，所以往往能获得更强大的能力。C++目前似乎是唯一没有强制单根结构的唯一一种 OO 语言。

(24) Java 没有模板或者参数化类型的其他形式。它提供了一系列集合：Vector（向量），Stack（堆栈）以及 Hashtable（散列表），用于容纳 Object 引用。利用这些集合，我们的一系列要求可得到满足。但这些集合并非是为实现象 C++"标准模板库"（STL）那样的快速调用而设计的。Java 1.2 中的新集合显得更加完整，但仍不具备正宗模板那样的高效率使用手段。

(25) "垃圾收集"意味着在 Java 中出现内存漏洞的情况会少得多，但也并非完全不可能（若调用一个用于分配存储空间的固有方法，垃圾收集器就不能对其进行跟踪监视）。然而，内存漏洞和资源漏洞多是由于编写不当的 finalize()造成的，或是由于在已分配的一个块尾释放一种资源造成的（"破坏器"在此时显得特别方便）。垃圾收集器是在 C++基础上的一种极大进步，使许多编程问题消弥于无形之中。但对少数几个垃圾收集器力有不逮的问题，它却是不大适合的。但垃圾收集器的大量优点也使这一处缺点显得微不足道。

(26) Java 内建了对多线程的支持。利用一个特殊的 Thread 类，我们可通过继承创建一个新线程（放弃了 run()方法）。若将 **synchronized**（同步）关键字作为方法的一个类型限制符使用，相互排斥现象会在对象这一级发生。在任何给定的时间，只有一个线程能使用一个对象的 synchronized 方法。在另一方面，一个 synchronized 方法进入以后，它首先会"锁定"对象，防止其他任何 synchronized 方法再使用那个对象。只有退出了这个方法，才会将对象"解锁"。在线程之间，我们仍然要负责实现更复杂的同步机制，方法是创建自己的"监视器"类。递归的 synchronized 方法可以正常运作。若线程的优先等级相同，则时间的"分片"不能得到保证。

(27) 我们不是象 C++那样控制声明代码块，而是将访问限定符（public，private 和 protected）置入每个类成员的定义里。若未规定一个"显式"（明确的）限定符，就会默认为"友好的"（friendly）。这意味着同一个包里的其他元素也可以访问它（相当于它们都成为 C++的"friends"——朋友），但不可由包外的任何元素访问。类——以及类内的每个方法——都有一个访问限定符，决定它是否能在文件的外部"可见"。private 关键字通常很少在 Java 中使用，因为与排斥同一个包内其他类的访问相比，"友好的"访问通常更加有用。然而，在多线程的环境中，对 private 的恰当运用是非常重要的。Java 的 protected 关键字意味着"可由继承者访问，亦可由包内其他元素访问"。注意 Java 没有与 C++的 protected 关键字等价的元素，后者意味着"只能由继承者访问"（以前可用"private protected"实现这个目的，但这一对关键字的组合已被取消了）。

(28) 嵌套的类。在 C++中，对类进行嵌套有助于隐藏名称，并便于代码的组织（但 C++的"命名空间"已使名称的隐藏显得多余）。Java 的"封装"或"打包"概念等价于 C++的命名空间，所以不再是一个问题。Java 1.1 引入了"内部类"的概念，它秘密保持指向外部类的一个句柄——创建内部类对象的时候需要用到。这意味着内部类对象也许能访问外部类对象的成员，毋需任何条件——就好象那些成员直接隶属于内部类对象一样。这样便为回调问题提供了一个更优秀的方案——C++是用指向成员的指针解决的。

(29) 由于存在前面介绍的那种内部类，所以 Java 里没有指向成员的指针。

(30) Java 不存在"嵌入"（inline）方法。Java 编译器也许会自行决定嵌入一

个方法，但我们对此没有更多的控制权力。在 Java 中，可为一个方法使用 final 关键字，从而“建议”进行嵌入操作。然而，嵌入函数对于 C++的编译器来说也只是一种建议。

(31) Java 中的继承具有与 C++相同的效果，但采用的语法不同。Java 用 extends 关键字标志从一个基础类的继承，并用 super 关键字指出准备在基础类中调用的方法，它与我们当前所在的方法具有相同的名字（然而，Java 中的 super 关键字只允许我们访问父类的方法——亦即分级结构的上一级）。通过在 C++中设定基础类的作用域，我们可访问位于分级结构较深处的方法。亦可用 super 关键字调用基础类构建器。正如早先指出的那样，所有类最终都会从 Object 里自动继承。和 C++不同，不存在明确的构建器初始化列表。但编译器会强迫我们在构建器主体的开头进行全部的基础类初始化，而且不允许我们在主体的后面部分进行这一工作。通过组合运用自动初始化以及来自未初始化对象句柄的异常，成员的初始化可得到有效的保证。

1045 页程序

(32) Java 中的继承不会改变基础类成员的保护级别。我们不能在 Java 中指定 public，private 或者 protected 继承，这一点与 C++是相同的。此外，在衍生类中的优先方法不能减少对基础类方法的访问。例如，假设一个成员在基础类中属于 public，而我们用另一个方法代替了它，那么用于替换的方法也必须属于 public（编译器会自动检查）。

(33) Java 提供了一个 interface 关键字，它的作用是创建抽象基础类的一个等价物。在其中填充抽象方法，且没有数据成员。这样一来，对于仅仅设计成一个接口的东西，以及对于用 extends 关键字在现有功能基础上的扩展，两者之间便产生了一个明显的差异。不值得用 abstract 关键字产生一种类似的效果，因为我们不能创建属于那个类的一个对象。一个 abstract（抽象）类可包含抽象方法（尽管并不要求在它里面包含什么东西），但它也能包含用于具体实现的代码。因此，它被限制成一个单一的继承。通过与接口联合使用，这一方案避免了对类似于 C++虚拟基础类那样的一些机制的需要。

为创建可进行“例示”（即创建一个实例）的一个 interface（接口）的版本，需使用 implements 关键字。它的语法类似于继承的语法，如下所示：

1046 页程序

(34) Java 中没有 virtual 关键字，因为所有非 static 方法都肯定会用到动态绑定。在 Java 中，程序员不必自行决定是否使用动态绑定。C++之所以采用了 virtual，是由于我们对性能进行调整的时候，可通过将其省略，从而获得执行效率的少量提升（或者换句话说：“如果不用，就没必要为它付出代价”）。virtual 经常会造成一定程度的混淆，而且获得令人不快的结果。final 关键字为性能的调整规定了一些范围——它向编译器指出这种方法不能被取代，所以它的范围可能被静态约束（而且成为嵌入状态，所以使用 C++非 virtual 调用的等价方式）。这些优化工作是由编译器完成的。

(35) Java 不提供多重继承机制（MI），至少不象 C++那样做。与 protected 类似，MI 表面上是一个很不错的主意，但只有真正面对一个特定的设计问题时，

才知道自己需要它。由于 Java 使用的是“单根”分级结构，所以只有在极少的场合才需要用到 MI。interface 关键字会帮助我们自动完成多个接口的合并工作。

(36) 运行期的类型标识功能与 C++极为相似。例如，为获得与句柄 X 有关的信息，可使用下述代码：

X.getClass().getName();

为进行一个“类型安全”的紧缩造型，可使用：

derived d = (derived)base;

这与旧式风格的 C 造型是一样的。编译器会自动调用动态造型机制，不要求使用额外的语法。尽管它并不象 C++的“new casts”那样具有易于定位造型的优点，但 Java 会检查使用情况，并丢弃那些“异常”，所以它不会象 C++那样允许坏造型的存在。

(37) Java 采取了不同的异常控制机制，因为此时已经不存在构建器。可添加一个 finally 从句，强制执行特定的语句，以便进行必要的清除工作。Java 中的所有异常都是从基础类 Throwable 里继承而来的，所以可确保我们得到的是一个通用接口。

1047 页程序

(38) Java 的异常规范比 C++的出色得多。丢弃一个错误的异常后，不是象 C++那样在运行期间调用一个函数，Java 异常规范是在编译期间检查并执行的。除此以外，被取代的方法必须遵守那一方法的基础类版本的异常规范：它们可丢弃指定的异常或者从那些异常衍生出来的其他异常。这样一来，我们最终得到的是更为“健壮”的异常控制代码。

(39) Java 具有方法过载的能力，但不允许运算符过载。String 类不能用+和+=运算符连接不同的字串，而且 String 表达式使用自动的类型转换，但那是一种特殊的内建情况。

(40) 通过事先的约定，C++中经常出现的 const 问题在 Java 里已得到了控制。我们只能传递指向对象的句柄，本地副本永远不会为我们自动生成。若希望使用类似 C++按值传递那样的技术，可调用 clone()，生成自变量的一个本地副本（尽管 clone()的设计依然尚显粗糙——参见第 12 章）。根本不存在被自动调用的副本构建器。为创建一个编译期的常数值，可象下面这样编码：

static final int SIZE = 255

static final int BSIZE = 8 * SIZE

(41) 由于安全方面的原因，“应用程序”的编程与“程序片”的编程之间存在着显著的差异。一个最明显的问题是程序片不允许我们进行磁盘的写操作，因为这样做会造成从远程站点下载的、不明来历的程序可能胡乱改写我们的磁盘。随着 Java 1.1 对数字签名技术的引用，这一情况已有所改观。根据数字签名，我们可确切知道一个程序片的全部作者，并验证他们是否已获得授权。Java 1.2 会进一步增强程序片的能力。

(42) 由于 Java 在某些场合可能显得限制太多，所以有时不愿用它执行象直接访问硬件这样的重要任务。Java 解决这个问题的方案是“固有方法”，允许我们调用由其他语言写成的函数（目前只支持 C 和 C++）。这样一来，我们就肯定能够解决与平台有关的问题（采用一种不可移植的形式，但那些代码随后会被隔离起来）。程序片不能调用固有方法，只有应用程序才可以。

(43) Java 提供对注释文档的内建支持，所以源码文件也可以包含它们自己的文档。通过一个单独的程序，这些文档信息可以提取出来，并重新格式化成 HTML。这无疑是文档管理及应用的极大进步。

(44) Java 包含了一些标准库，用于完成特定的任务。C++则依靠一些非标准的、由其他厂商提供的库。这些任务包括（或不久就要包括）：

■连网
■数据库连接（通过 JDBC）
■多线程
■分布式对象（通过 RMI 和 CORBA）
■压缩
■商贸

由于这些库简单易用，而且非常标准，所以能极大加快应用程序的开发速度。

(45) Java 1.1 包含了 Java Beans 标准，后者可创建在可视编程环境中使用的组件。由于遵守同样的标准，所以可视组件能够在所有厂商的开发环境中使用。由于我们并不依赖一家厂商的方案进行可视组件的设计，所以组件的选择余地会加大，并可提高组件的效能。除此之外，Java Beans 的设计非常简单，便于程序员理解；而那些由不同的厂商开发的专用组件框架则要求进行更深入的学习。

(46) 若访问 Java 句柄失败，就会丢弃一次异常。这种丢弃测试并不一定要正好在使用一个句柄之前进行。根据 Java 的设计规范，只是说异常必须以某种形式丢弃。许多 C++运行期系统也能丢弃那些由于指针错误造成的异常。

(47) Java 通常显得更为健壮，为此采取的手段如下：

■对象句柄初始化成 null（一个关键字）
■句柄肯定会得到检查，并在出错时丢弃异常
■所有数组访问都会得到检查，及时发现边界违例情况
■自动垃圾收集，防止出现内存漏洞
■明确、“傻瓜式”的异常控制机制
■为多线程提供了简单的语言支持
■对网络程序片进行字节码校验