Java 深度历险

目　录

导读.....1
第一章.....3
第二章.....14
第三章.....44
第四章.....61
第五章.....70
第六章.....97
附录 A.....110

导读

我的第一份「正式」工作就是在升阳电脑(Sun Microsystems)台湾分公司的教育训练中心从事讲师工作。在这之前，我除了当个管理学院的研究生之外，闲暇之余还会到处兼职当顾问、接接 Project 写程式糊口。这本书很多最初始的概念都源自于当时所服务的公司里头工程师所遇到的难题，或是自己使用 Java 撰写应用程式的时候所遇到百思不解的问题。

承蒙洪志鹏总经理愿意收留我这样的人，到了 Sun 工作之后，利用内部网路所看到的世界，竟然是一个我从没想过的天堂。我猜这是很多人愿意一开始就到大型软体公司上班的最大原因。在Sun待了一阵子你就发现，Sun跟Microsoft是不一样的公司，由于两家企业所发迹的领域不太相同，导致 Sun 在设计软体的时候比较重视架构，尤其是应用在企业领域上的软体架构，您可以从两个地方看出来，第一个是 Java 的认证考试中，最高级的认证叫做 SCJA(架构师认证)，第二个是市面上关于 Java 的书籍，排除入门书之后，许多重要的书籍都和 Design Pattern 有关系，比方说 J2EE Design Pattern、EJB Design Pattern，关于系统的书籍就少的可怜，讲 JVM 的书屈指可数，我手边讲述 Java 系统的书籍，我认为是好书的也只有两本。微软则刚好相反，微软重视系统更甚于架构，您可以发现.NET 刚出来，就有一堆讲述 CLR(Common Language Runtime)或.NET 内部运作的书籍，而讲述架构的书籍就少的多了。

很可惜的是，我本身就是一个系统的爱好者。最近这一年唯一让我觉得拍案叫奇的好书，就只有 Jeffery Richter 所撰写的 Applied Microsoft.NET Framework Programming，我就是喜欢 Applied Microsoft.NET Framework Programming 这种书籍的读者。看完 Jeffery Richter 的书之后，最大的悔恨就是 Java 出现了将近 7 年，可是却没有任何一本类似的书籍让我们更了解 JDK，或是 Java 的运作原理。这本书就是在这种悔恨之下形成的，虽然不是很厚，但是里面的每一篇都是我这几年来研究 Java 运作的心得。所以这里要感谢 Run!PC 的技术编辑吕志敏先生，没有他每个月定期的催稿，我就没有办法定期地整理我的心得，并将他们发表在杂志上，更没有办法在获得很多读者的回应之后，重新整理这些内容成册，并适当地加以修订和整合。我希望喜欢系统的朋友

读完这本书之后，不再有跟我一样的悔恨，而且对 Java 这个将近 10 年前就有的设计更具信心。

在 2002 年 JavaTwo 时，我用『数学家、物理学家、以及工程师的差别』来介绍自己。数学家的心情套用在我的身上，就知道我是个非常懒惰的人，你可能会以为我手边有 JDK 的原始码，所以我就会先花功夫去读懂 JDK 的原始码，然后把结果告诉你。事实上并非如此，撰写这本书的时候，尤其是第五章『package 与 import 机制』，因为我懒得去看 Java 编译器的原始码，所以我采用的方式是归纳法。我先设计了很多其奇怪怪的 use case，然后对 Java 编译器进行测试，利用所产生的结果来想像 Java 编译器的运作方式，最后归纳出属于我自己对 Java 编译器运作方式的『理论』。最后我花一点点时间来阅读原始码，发现竟然和我的推论不谋而合，所以这一章大概是我自以为最有成就感的一章。这种写作方式也让我发现，现代的科学家在探究我们所身处的世界时所使用的方法，竟然是如此地让人兴奋☺。

记得有一次讲完一场研讨会之后，有个听众来问我:「王森先生，我们公司使用 Borland JBuilder 开发应用程式，可是遇到很多问题，比方说常常出现 Class Not Found 的错误讯息」 。原来，JBuilder 虽然是一套 RAD 开发工具，但是他却不像其他如 Visual Basic、Delphi、Borland C++ Builder 一般好上手，因为它有其进入门槛。而这个进入门槛，来自于存在于 JBuilder 底部的 Java 2 SDK。如果对 Java 2 SDK 没有深入的了解，想要平顺地驾驭更高阶的开发工具绝对是一件很难的事情。我希望这本书带您跨过这个门槛。

这本书并未对 Java 程式语言做讨论，因为仿间这类的好书已经非常多了。本书着眼于其他 Java 书即从来没有提到的议题。期望带给您真正对 Java 的『深度历险』，也希望众多 Java 爱好者对这本书的回应，可以支持我整理出更多有趣的议题。

要感谢的人很多，很多我都在我先前的著作中提及了，我在心里默默的感激他们。如果您认为这本书带给您很多新的知识，请您也跟我一同感谢这些我生命中不可或缺的人们，以及催生这本书的所有工作伙伴。

Sun Microsystems

教育训练服务 技术顾问

王森 moli.wang@gmail.com

第一章

深入 Java 2 SDK
你越讨厌的事情，就越容易遇上
前言
Sun Microsystems 所发表的 Java 开发工具– Java 2 SDK ，永远都是 Java 初学者最早接触到的开发工具。一般人习惯称这套工具叫作 JDK(Java Development Kit)。
图:Java 版本及开发工具的演进
OAK Java 1.0 Java 1.0 Java 1.1 Java 1.1 Java 1.2 Java 1.2 Java 1.3 Java 1.3
Embedded Embedded
Java Java
Personal Personal
Java 1.0 Java 1.0
Personal Personal
Java 1.1 Java 1.1
Personal Personal
Java 1.2 Java 1.2
Java Platform(JDK) Java 2 Platform(Java 2 SDK)
Java 1.4 Java 1.4

JDK 是在 Java 2 Platform 之前的 Java Platform 所使用的开发工具名称(在本文中有时候会写 JDK，有时候会写 Java 2 SDK，但指的将是同一种东西)，我记得曾经有人戏称它是 Java Developer Killer 的缩写，除了挖苦 Sun 所制作的开发工具没有微软设计的开发工具要来的方便之外，其实还说明了另外一件事，就是 Java 从 1995 年发表后到现在，即使每一版的 JDK 都会附上为数庞大的官方说明文件，从来没有任何一份文件或一本书籍详细说明这套官方开发工具的特性。
没有文件或书籍来描述 JDK 的特性并不代表这些特性不重要。毕竟，任何最新的标准类别函式库，或是最新版本的虚拟机器，一定都会伴随着最新版的 JDK 所释出。就算您想跳过 JDK，直接使用如 Borland JBuilder 或 Forte for Java 这类的高级开发工具，JDK 仍然如影随形。以 Borland JBuilder 来说，当您将 Borland JBuilder 安装完成之后，在 JBuilder 的所在目录下也会内含一套 Java 2 2 SDK，如下图所示:
图:Borland JBuilder 内附的 Java 2 SDK

因此，我们可以得知，不管您如何地讨厌 JDK，只要想开发 Java 相关的应用程式，您就无法逃离 JDK 的掌握。事实上，不管您开发的是 J2SE、J2EE、J2ME、
甚至 Java Card 的应用程式，除了需要各种版本对应的开发套件之外，一定需要 JDK 的辅助。
图:各种版本的 Java 应用程式，都需要 Java 2 SDK 的辅助 3
J2SE(Java SDK)

J2EE SDK J2ME SDK Java Card
SDK

那么，JDK 到底是什么东西? 从技术的观点来说，因为高阶开发工具都是架设在 JDK 上头，因此高阶开发工具的行为或是引发的错误讯息都是根源自 JDK。为了更正确地掌控高阶的 Java 开发工具，所以我们必须了解 JDK 的特性
和组成。从求知的观点来看，Java 程式设计师每天输入无数的 java xxx.java 与 java xxx，到底 Java 程式是如何运作的? 在我们看不到的底层，到底发生了什么事情? 如果我们可以清楚地得知所有的来龙去脉，将会让我们更了解这套开发工具。
上述两个观点，都是本章所希望告诉您的。让我们开始深入了解 JDK 吧!
执行 java.exe 时所发生的怪事
当您在使用 JDK 时，您是否曾经发现执行 java.exe 的时候，会有底下一些奇怪的现象:
如果您安装的是 Java 2 SDK 1.3.x :
当您安装完 Java 2 SDK 1.3.x 之后，如果从未修改您电脑里头的任何设定，就直接进到命令提示字元下，执行 java.exe，就会出现底下画面: 4

萤幕上会告诉您，有一个–hotspot 选项可供您使用。
但是，此时如果我们输入指令:
path=c:\jdk1.3.1\bin
(注:假设笔者的 Java 2 SDK 1.3.x 安装于 c:\jdk1.3.1 底下)
然后重新执行 java.exe，则萤幕上的输出如下:
5
这一次，萤幕上的输出告诉我们有–hotspot、-server、以及-classic 可供选择。这真是奇怪的事情，原本根本没有出现在萤幕上的–hotspot、-server、以及-classic 选项，在我们使用 path 这个系统命令来改变执行档的搜寻路径之后，竟然出现了!
如果您安装的是 Java 2 SDK JDK 1.4.x :
当您安装完 Java 2 SDK 1.4.x 之后，如果从未修改您电脑里头的任何设定，就直接进到命令提示字元下，执行 java.exe，就会出现底下画面:

画面上告诉您，java.exe 有几个选项，包括–client、-server 、-hotspot(与-client 意义相同，但是已不建议使用)。这几个选项可以用来调整执行 Java 程式时所使用的 Java 虚拟机器。
所以此时如果您输入
java –version
的时候，萤幕输出如下:
6
而如果输入:
java –server -version
并执行之，萤幕上会出现错误讯息:

Error: no `server' JVM at `C:\Program Files\Java\j2re1.4.0-rc\bin\server\jvm.dll`.
意思是说，它在特定位置找不到名为 server 的虚拟机器。也就是虽然萤幕上有列出此选项，可是却无法使用。
但是，此时如果我们输入指令:
path=d:\j2sdk1.4.0\bin
(注:假设笔者的 JDK 1.4.x 安装于 d:\j2sdk1.4.0 底下)
然后重新执行:
java –server
将不再出现错误讯息:

如果此时您输入
java –server -version
则萤幕的输出如下:

真奇怪，原本无法运作的-server 选项，在我们使用 path 这个系统命令来改变执行档的搜寻路径之后，竟然可以运作了!
这真是令人匪夷所思的事情。 7
如此怪异的现象，在 JDK 1.3.x 上最容易看出来(JDK 1.4.x 上其实也是一样奇怪，因为无法使用的选项以及没有出现在我们眼中的选项，基本上是一样的。只不过如果没有特别去尝试，就很难看得出来，搞不好您还会认为眼不见为净，出现一些无法使用的选项反倒混淆使用者):
上述这么奇特的现象，该做如何解释? 我们有办法预测其行为吗? 再者，在您过去使用 Java 2 SDK 来开发 Java 应用程式的生涯之中，是否也曾发生过底下的怪事:
很可能您才刚刚灌好某个版本的 Java 2 SDK 或刚刚移除某个版本的 Java 2 SDK，但是之后却在执行 java.exe 时，萤幕上竟然告诉您:

在您经验中，萤幕上的 1.4 和 1.3 可能会换成其他版本号码，总之就是版本不合所引发的问题。
因此接下来，我们将开始探讨这种”灵异现象”的成因，并追踪其来龙去脉。最后，当您下次再遇到这种情况时，您将可以很容易地找出问题的解决方法。
JDK、JRE、JVM 之间的关系
要探讨上述的奇怪现象，必须先弄清楚 JDK(Java 开发套件)、JRE(Java 执行环境，Java Runtime Environment)、还有 JVM(Java 虚拟机器，Java Virtual Machine)三者之间的关系。
当您想要从 http://www.javasoft.com 下载 JDK 来开发程式的时候，或许您曾经疑惑过，官方网站上会出现 JDK 与 JRE 两种套件，然而一般初学者不清
楚 JDK 和 JRE 有什么不同，所以产生疑惑。这两者一定有所不同，否则 Sun Microsystems 不可能将两者区分开来。关于 JDK 与 JRE 两者间的关联，我们可
以用底下图片来描述:
图:JDK 1.3.x 与 JRE 1.3.x 的关系 8

JDK
JDK 1.3.x JRE 1.3.x JDK 1.3.x JRE 1.3.x
Java 程式
开发工具
(javac.exe,jar.exe 等)
JRE

(位于 JDK 安装目錄下

的\jre 子目錄)

JRE
(位于
Program Files\JavaSoft 下

的子目錄)

JRE
(位于
Program Files\
JavaSoft 下

的子目錄)

JDK 1.4.x 和 JDK 1.3.x 之间有些许的不同，如下图所示:
图:JDK 1.4.x 与 JRE 1.4.x 的关系
JDK
JDK 1.4.x JRE 1.4.x JDK 1.4.x JRE 1.4.x
Java 程式
开发工具
(javac.exe,jar.exe 等)
JRE

(位于 JDK 安装目錄下

的\jre 子目錄)

JRE
(位于
Program Files\JavaSoft 下

的子目錄)

JRE
(位于
Program Files\
Java 下

的子目錄)

从上图可以得知，如果您装了 JDK，那么您的电脑底下一定会有两套 JRE、一套位于<jdk 安装目录>\jre 底下(JDK 1.3.x 与 1.4.x 皆是如此)，另外一套位于 C:\Program File\JavaSoft 底下(JDK 1.4.x 则是放在 C:\Program File\Java 底下)。
注意! 9
如果您安装的是 JDK 1.3.x，那么您没有选择的余地，安装程式一定会同时安装两套 JRE 在您的电脑上。
可是如果是安装 JDK 1.4.x 的话，您可以在安装 JDK 时选择是否安装位于 C:\Program File\Java 目录下的 JRE，但是<jdk 安装目录>\jre 底下的这套 JRE 一样必须安装，没有选择的余地。下图是 JDK 1.4.x 的安装画面，您可以看到 JRE 是否安装可以由您自己决定，不过它指的 JRE 是位于 C:\Program File\Java 目录下的 JRE ，千万别跟<jdk 安装目录>\jre 底下那套 JRE 混淆了。

您的电脑上安装了 JDK 1.4.x 之后，档案系统中的档案分布如下图:
图:安装 JDK 1.4.x 之后，系统中的档案分布情形(1) 10

与
图:安装 JDK 1.4.x 之后，系统中的档案分布情形(2)

不晓得大家有没有发现，两个目录下所出现的档案和子目录根本一模一样(其实 11
有很小很小的差别，但是对我们的讨论没有任何影响，因此本文中当作两者没有任何差别)。
如果您只从网路下载了 JRE 而非 JDK，那么就只会在 C:\Program File\JavaSoft 底下(JRE 1.4.x 则是在 C:\Program File\Java 底下)安装唯一的一套 JRE。
接下来您一定会问，那么 JRE 是做什么用的? 笔者用下图描述 JRE 的功能:
图:JRE 与 PC 的类比
JRE
JRE JRE 与 与 PC PC

Java 類別函式库

(.class)
Java 虚拟机器
(jvm.dll)
原生函式库
(.dll)
辅助函式库
(.dll)
PC
作业系统

CPU
Win32 API
(.dll)

从上图您可以知道，JRE 的地位就像一台 PC 一样，我们撰写好的 Win32 应用程式需要作业系统帮我们执行，同样地，我们所撰写的 Java 程式也必须要 JRE 才能帮我们执行。所以。当您装完 JDK 之后，如果分别在硬碟的不同地方安装了两套 JRE，那么您就可以想像您的电脑有两台虚拟的 Java PC，都具有执行 Java 应用程式的功能。所以 Java 虚拟机器只是 JRE 里头的其中一个成员
而已，以更技术的角度来说，Java 虚拟机器只是 JRE 里头的一个动态联结函式
库罢了。
所以我们可以说，只要您的使用者电脑之中安装了 JRE，就可以正确地执行 Java 应用程式(Windows XP 宣称不支援 Java，就是因为微软不再于 Windows XP 中随机附上自己版本的 JRE，不过，就算微软不附自己的 JRE，Sun 还是提
供可以支援 Windows XP 的 JRE，JDK 1.4.x/JRE 1.4.x 就完全支援 Windows XP)。
接下来您一定会问，那么 Sun 为何要让 JDK 的安装程式同时安装两套几乎完全相同的 JRE 呢? 花费额外的硬碟空间在不同的目录下安装两个一模一样的东西，颇有浪费之嫌，不是吗? 其实真正的原因是因为- JDK 里面也附上了很多用 Java 所撰写开发工具(例如 javac.exe、jar.exe 等)，而且他们都放置在 12 <jdk 安装目录>\lib\tools.jar 这个档案之中。
什么? Java 的编译器 javac.exe 也是用 Java 撰写的， 真的吗? javac.exe 明明就是一个执行档，用 Java 撰写的程式应该是.class 档才对。为了证明这件事情，底下我们做个小实验，证明我所言不假:
首先，请先在命令列模式底下执行底下指令 javac.exe :

然后，请到<jdk 安装目录>\lib\ 底下把 tools.jar 改名成 tools1.jar，然后重新执行 javac.exe，您将看到萤幕输出如下:

这个意思是说，您输入 javac.exe 和输入
java -classpath d:\j2sdk1.4.0\lib\tools.jar com.sun.tools.javac.Main
会得到相同的结果。请把 tools1.jar 改回成 tools.jar，然后重新执行指令: 13

您会看到执行结果和输入 javac.exe 一模一样。从这里我们可以证明 javac.exe 只是一个包装器(wrapper)，而制作目的是为了让开发者免于输入太长的指令。
包装器的概念如下图所示:
图:包装器的概念
包装器
(javac.exe)
JRE
+

Tools.jar

1.执行

2.找到
3.启动
java -classpath d:\j2sdk1.4.0\lib\tools.jar com.sun.tools.javac.Main

其实包装器的概念您也可以在很多地方看到，比方说当您用 J2ME 开发 Palm 应用程式的时候，工具都会帮我们打包 JAR 档，然后用一个 PRC 档的外壳罩住，让 Java 程式看起来像是一个原生(native)的应用程式。如果您开发.NET 应用程式，那么.NET 开发工具所造出的执行档也是一个包装器的概念。不管如何，包
14
装器的设计是为了让使用者更方便地使用您的应用程式。
除了 javac.exe 之外，JDK 很多内附的开发工具也都是采用包装器的概念。
请试着开启<jdk 安装目录>\bin 目录，您将发现底下的执行档大小都很小，不大于 29 KB:
图:bin 目录下的执行档

从这里我们可以推得一个结论，就是: JDK 里面的工具几乎是用 Java 所撰写的，
所以 JDK 本身就是 Java 应用程式，因此要使用 JDK 附的工具来开发 Java 程式，也必须要自行附一套 JRE 才行，这就是<jdk 安装目录>\jre 底下需要一套 JRE 的原因。而位于 Program File\底下的那套 JRE 就是拿来执行我们自己所撰写的 Java 应用程式。不过，两套中任何一套 JRE 都可以拿来执行我们所撰写的
Java 应用程式，可是 JDK 内附的开发工具在预设使用包装器(.exe)来启动的情形下，都会自己去选用<jdk 安装目录>\jre 底下那套 JRE。
但是问题也就出在这里，在同一台电脑之中安装如此多的 JRE(如果您还安装了 Borland JBuilder 或 TogetherJ 这些完全使用 Java 开发的软体，那么您的电脑里头就会有更多套的 JRE)。一台电脑之中许多套 JRE，再加上作业系统本
身 PATH 环境变数的特性，就会造成最开头所示范的怪异现象:同样都是执行 java.exe，但是执行结果却不相同。
您所执行的是哪一个 java.exe ?
既然您的电脑里头至少有两套 JRE，那么谁来决定用哪一套 JRE 呢? 这个重责大任就落在 java.exe 的身上。
当我们在命令列输入
java XXX 15
的时候，java.exe 的工作就是找到合适的 JRE 来执行类别档。 java.exe 依照底下逻辑来寻找 JRE:
1. 自己的目录下有没有 JRE 目录。 (这个部分这样说并不是非常精确，原因请详见 JDK 原始码，这此不特别说明)
2. 父目录底下 JRE 子目录。
3. 查询 Windows

Registry(HKEY_LOCAL_MACHINE\Software\JavaSoft\Java Runtime Environment\)。
所以，java.exe 的执行结果和您电脑里面哪一个 java.exe 被执行，然后哪一套 JRE 被拿来执行 Java 应用程式有莫大的关系。
请您先使用 Windows 的搜寻功能，看看您的电脑里头到底有多少 java.exe? 笔者的电脑上一经搜寻，找到了好几个 java.exe，如下图:
图:电脑里的 java.exe

所以您可以断定，如果指令为:

您执行到的 java.exe 一定是 d:\j2sdk1.4.0\bin 底下的 java.exe，所以 java.exe 选到的是父目录(d:\j2sdk1.4.0)底下的 JRE(d:\j2sdk1.4.0\jre)。
请开启 d:\j2sdk1.4.0\jre\bin 这个目录，您会看到 client 和 server 两个目录，里面都会分别看到 jvm.dll，这就是一般我们所谓的 Java 虚拟机器之所在。
图:真正的 Java 虚拟机器之所在(hotspot client) 16

图:真正的 Java 虚拟机器之所在(hotspot server)

(注: 如果您设定 path=d:\jdk1.3.1\bin ，那么就应该开启 d:\jdk1.3.1\jre\bin，您将看到 classic、hotspot、与 server 三个子目录，里头都拥有 jvm.dll)
当您输入的指令为 java –server 时，就会使 java.exe 自动选择 d:\j2sdk1.4.0\jre\bin\server 这个目录底下的 JVM；而输入 java –classic 时，就会使 java.exe 自动选择 d:\jdk1.3.1\jre\bin\classic 这个目录底下的 JVM。从这里我们就可以解释之前为何设定了 path=c:\jdk1.3.1\bin 之后，会在画面中突然出现两个之前不曾出现的选项，也可以说明为何设定了 path=d:\j2sdk1.4.0\bin 之后，java –server 的指令会从原本的错误讯息变成可以正常使用。
至于为何没有设定 PATH 环境变数前，会出现错误讯息，这是因为系统预设的 PATH 环境变数内容如下: 17

所以 java.exe 在 c:\winnt\system32 底下找不到 JRE 目录，在 c:\winnt 也找不到 JRE 目录的情况下，根据下一个逻辑，就是去查询 Windows Registry，如下图所示:
图:Windows Registry
18

因为在 Windows Registry 之中查询到 JRE 所在位置为 C:\Programm Foles\Java\j2re1.4.0-rc 。开启该目录下的 bin 子目录却只有看到 client 子目录，却没有看到 server 子目录:

这就是之所以出现错误讯息:

的原因，因为 java.exe 根本无法在 JRE 所在目录下的 bin 子目录之中找到名为 server 的子目录。同样的道理，请您自行验证使用 JDK 1.3.x 时萤幕上执行 java.exe 的结果之成因，您将发现 Registry 之中所设定 1.3.x 版的 JRE 之路径
下的 bin 子目录，一样只有 client 目录。
其实，在 java.exe 找到 JRE 之后，还有一个验证版本的程序，也就是 java.exe 和 JRE 两者的版本要一致才能继续执行。所以假如阴错阳差，您执行 19
到 1.3.x 版 JDK 附的 java.exe，而此 java.exe 却找到版本号码为 1.4.x 的 JRE，就会发生底下结果:

上面林林总总解释了那么多，只想告诉大家，当您在开发 Java 程式或是执行 Java 程式的时候，一定要记得两件事:
1. 那一个 java.exe 被执行。
2. java.exe 找到哪一套 JRE。
只要这两件事都确定了，就知道问题发生的来龙去脉，也可以很容易地解决很多貌似灵异的怪问题。
常见的错误
前面已经拿 JRE 与 PC 做了个比喻，JRE 对 Java 应用程式来说，就是一台独立的电脑，而您的电脑上会有好几套不同的 JRE，因此对 Java 应用程式来说，就是有许多不同的「虚拟电脑」 。我们可以想像一种情况:您有两台电脑，一台是
PC，一台是 Notebook，如果您在 Notebook 上安装了 DirectX 函式库，那么 PC 上仍然无法执行需要 DirectX 的游戏，您一定会在 PC 上重新安装 DirectX。再举个例子，您在 PC 上设定好作业环境(如日期设定、密码设定、安全设定)之
后，您一定不会期待 Notebook 上也有这些设定，您必须自己动手设定您的 Notebook 才行。这个道理很简单- 因为这两台电脑根本就是两个不同独立的个体。
对 Java 应用程式来说，每个 JRE 都是独立不相干的个体。举凡函式库、安全设定等与特定 JRE 相关联的特性，如果您设定的是在 A 处的 JRE，但是执行
时 Java 应用程式却是在 B 处的 JRE 之中执行，那么您的设定就会完全没有作用。
所以，您必须清楚地知道哪一个 java.exe 被执行，以及 java.exe 到底选用到哪个 JRE，这些都是非常重要的细节。底下我们示范一个一般人几乎不太去
注意的小细节:
学习 Java 的朋友，一定会发现书上会希望您设定 PATH 环境变数，一旦 PATH 环境变数指向<jdk 安装目录>\bin\ 底下，您就可以在任何目录下执行 java.exe。如果您使用的是 Windows NT/2000/XP 作业系统，设定方式如下: 20

如上图所示，上面是设定使用者变数，下面是设定系统变数。通常一般人会设定使用者变数如下:

注意，假设笔者的 JDK 1.3.1 安装在 D 磁碟机，所以会在使用者变数的最开头加
上 d:\jdk1.3.1\bin。设定完成之后，就开始了您的 Java 程式设计学习之旅。 21
不过有一天，您学习的越来越深入时，您的生活会开始和 JRE 产生关联。
比方说您会学习不同的 Java 函式库，函式库的说明文件会希望您把这些函式库(通常是一堆 JAR 档)放到<JRE 所在目录>\lib\ext 底下，通常您会选择放到<jdk 安装目录>\jre\lib\ext 底下:

此外，您也有可能会学习 Java 的安全机制，与 Java 相关的安全设定档都放置于<JRE 所在目录>\lib\security 底下，改变这个目录下的档案设定，会可以改变 JVM 在安全上的设定，通常您也会选择修改<jdk 安装目录>\jre\lib\security 目录底下的档案:

接下来，当您执行 Java 程式时，就会发现您的安全设定根本不会发生作用；而叫用了其他 Java 函式库的程式，在编译阶段没有问题，可是却无法执行(萤幕 22
上会显示 ClassNotFoundException)。
一般初学者无法理解问题的发生原因，而学习 Java 已久的老鸟也可能不清楚这个问题之所以产生的真正原因。之所以会发生此怪异情形，我们必须从命令提示模式下来看:

就我们之前对环境变数的设定来说，在命令提示模式下看起来如此(系统变数在前，使用者变数在后)，我们前面还曾经要大家去搜寻您的电脑中到底有几个 java.exe(还记得吗? C:\winnt\system32 与 d:\jdk1.3.1\bin 底下都曾出现 java.exe)，所以当您编译 Java 应用程式时，由于只有 d:\jdk1.3.1\bin 底下可以找到 javac.exe，而 javac.exe 又会自动使用 JDK 所在目录下的那一套 JRE，因此很自然您的函式库如果拷贝到<jdk 安装目录>\jre\lib\ext 底下，很自然地在编译时 JVM 会找到函式库，所以编译不会发生任何问题。但是执行时就有差了，因为 PATH 环境变数的关系，所以您打 java XXX 的时候，会优先执
行 c:\winnt\system32 底下的那个 java.exe，也因此很自然地会去使用 c:\program files 目录下的那一套 JRE，因此要让您的 Java 应用程式可以正常执行，您还必须将那些函式库拷贝到 c:\program files 目录下的那一套 JRE 的 lib\ext 目录之下。 Java 安全设定之所以没有作用，也是基于一样的道理，如果以上述的情境来说，您就必须修改 c:\program files 目录下的那一套 JRE 的 lib\security 目录中的设定档才行。
如果您是每次都在命令提示模式下手动设定 PATH 环境变数如下:

就不会发生上述问题，因为不管是 javac.exe 或 java.exe，都会执行 JDK 所在目录下的，也因此找到的都是 JDK 安装目录下的那套 JRE，自然就没有上述问题。当然，我们可以类推，如果您设定的方式为: 23

就不会发生奇怪的问题，而如果您设定的方式为

则会发生上述的奇怪问题。

总结

在本文中，笔者为大家介绍 JDK、JRE、以及 JVM 三者之间的关系。依照 24 这三者的关系，笔者为大家解释了很多学习 Java 时遇到的灵异现象。只要知道事情的来龙去脉，灵异就不再是灵异，而且我们还可以去玩弄这些运作机制。

在看本章之后，如果要证明您已经了解本章内容，请容许笔者出个小题目帮您做个测试。还记得底下这个错误讯息吗?

请试着「刻意地」造出此错误讯息。如果您成功了，那么您已经开始进入玩弄 JDK、JRE、以及 JVM 的层次了。

第二章

深入类别载入器

如果阳春白雪真的是曲高和寡，
我希望有一天我能用下里巴人的方式
来引导他们学会阳春白雪。

前言

程式设计师在开发应用程式的时候，常常被老板耳提面命，要求写出来的程式要有弹性，容易扩充，甚至要求要能”Plug and Play”，相信不少程式设计师听到这些要求就非常头痛。在本文之中，笔者把这种需求称做对于『动态性』的需求。有了动态性，我们的应用程式就可以在不用全盘重新编译的情况下更新系统，或者在不用停止主程式运作的情况下(尤其是您的系统必须 24 小时运转，一
停止就会造成巨大损失时)，除去系统中原有的 bug，或者是增加原本不具备的新功能。

一般来说，常见的程式语言先天上并不具有动态性的本质，如 C、C++本身就不具备动态性。因此，为了让这些本身不具有动态性的程式语言具有某种程度的动态性，就必须依赖底层的作业系统提供一些机制来实现动态性，Windows 作业系统底下的动态联结函式库(Dynamic Linking Library)和 Unix 底下的共享物件(Share Object)这是这样的例子。但是，要运用这些底层作业系统所提供的机制，程式设计师必须多费一些功夫来撰写额外的程式码(例如 Windows 平台上需要使用 LoadLibrary()与 GetProcAddress()两个 Win32 API 来完成动态性的需求)，这些额外撰写的程式码也会因为作业平台的不同而不同，毕竟这些额外的程式码与程式本身的运作逻辑甚少关联，所以维护起来虽不算复杂，但仍有其难度。

相对来说，Java 是一个本质上就具有『动态性』的程式语言。在一般使用 Java 程式语言来开发应用程式的工程师眼中，很少有机会能够察觉 Java 因为具备了动态性之后所带来的优点和特性，甚至根本不曾利用过这个 Java 先天就具有的特性。这不是我们的错，而是因为这个动态的本质被巧妙地隐藏起来，使得使用 Java 的程式设计师在不知不觉中用到了动态性而不自知。

程式语言本质上就具备动态性的优点在于，程式设计师不需要撰写额外的程式码，所以比较没有跨平台的问题。而缺点则是底层的黑箱帮我们搞定了一大堆的琐碎工作，一旦程式设计师想要”御驾亲征”，自己完全主控动态性的时候，就必须了解更多细部的运作机制。

本章的主角是类别载入器。在 Java 中，讲到暗中帮助程式设计师，让使用 Java 所撰写的程式具备动态性的『黑手』，非类别载入器莫属。笔者将带大家
深入类别载入器的运作机制，让诸位读者了解类别载入器如何达成动态性。了解 2
动态性之所以能够顺利运行的来龙去脉之后，我们还要看看如何运用动态性做出一些巧妙且实际的功能。

为何要自己全盘掌控动态性?

学习 Java 的朋友一定都知道，Java 是一种天生就具有动态连结能力的技

术。 Java 把每个类别的宣告、介面的宣告，在编译器处理之后，全部变成一个个小的执行单位(类别档, .class)，一旦我们指定一个具有 public static void main(String args[])方法的类别作为起点开始运作之后，Java 虚拟机器会找出所有在执行时期需要的执行单位，并将他们载入记忆体之中，彼此互相交互运作。尽管本质上是一堆类别档，但是在记忆体之中，变成了一个”逻辑上”为一体的 Java 应用程式。所以，严格的来说，每个类别档对 Java 虚拟机器来说，都是一个独立的动态联结函式库，只不过它的副档名不是.dll 或.so，而是.class 罢了。因为这种特性，所以我们可以在不重新编译其他 Java 程式码的情况下，只修改有问题的执行单位，并放入档案系统之中，等到下次该 Java 虚拟机器重新启动时，这个逻辑上的 Java 应用程式就会因为载入了新修改的.class 档，自己的功能也做了更新。这是一个最基本的动态性功能。

但是，如果您用过支援 JSP/Servlet 的高档 Web Server(或称 Web Container)，或是高档的 Application Server 里的 EJB Container，他们一定会提供一个名为 Hot Deployment 的功能，这个功能的意思是说，您可以在 Web Server 不关闭的情况下，放入已经编译好的新 servlet 以取代旧的 servlet，下一次的 HTTP request 时，就会自动释放旧的 servlet 所代表的类别档，而重新载入新的 servlet 所代表的类别档，同理，如果主角换成 EJB Container，Hot Deployment 可以让元件部署者不用关闭 Application Server，就能够将旧的 EJB(也是一堆类别档的集合)换成新版的 EJB。

Hot Deployment 的功能并非每家厂商都能提供，只有还算高档的产品才有。接下来眼尖的朋友就会问了:「慢着，前面说新的类别必须要等到”下次该 Java 虚拟机器重新启动时”才会重新载入，可是有些 Web Server 或 EJB Container 本身就是用 Java 所撰写，他们也是在 Java 虚拟机器上面执行，那么，他们如何在 Java 虚拟机器不重新启动的情况下具有 Hot Deployment 的功能? 好吧!就算可以做到读取新版本的功能好了，在不关闭 Java 虚拟机器的情况下，类别所占用的记忆体将无法释放，那么记忆体里头肯定充满了同一个类别的新旧版本，如果 Hot Deployment 的次数太多，记忆体不会爆掉吗?」 。这是一个非常好的问题，也是本章之所以存在的理由。

了解了类别载入器的来龙去脉，您将可以让您的程式具有强大的动态性-在 Java 虚拟机器不重新启动的情况下做出具有载入最新类别档的功能;不关闭 Java 虚拟机器的情况下，释放类别所占用的记忆体，让记忆体不会因为充满了同一个类别的多个版本而面临记忆体不足的窘境。

我们在不知不觉中用到动态性 3

假设我们撰写了三个 Java 类别，分别是 Main、A、以及 B。他们的程式码分别如下所示:

档案:A.java

```
public class A
{
 public void print()
 {
  System.out.println("Using Class A") ;
 }
}
```

档案:B.java

```
public class B
{
 public void print()
 {
  System.out.println("Using Class B") ;
 }
}
```

档案:Main.java

```
public class Main
{
  public static void main(String args[])
 {
    A a1 = new A() ;
  a1.print() ;
    B b1 = new B() ;
  b1.print() ;
 }
}
```

这三个档案编译好之后，所在目录的内容如下图所示: 4

接着，我们执行指令:
java –verbose:class Main
萤幕上会产生一长串的输出结果，我们撷取最后的部分呈现出来，如下图所示:

萤幕上的输出告诉我们，类别载入器(class loader)在背景偷偷的运作，除了将 Java 程式运作时所需要的基础类别函式库(又叫核心类别函式库， Core Classes)载入记忆体(位于<JRE 所在位置>\lib\rt.jar 之中)，我们的程式所用到的类别 Main.class、A.class、以及 B.class 也都偷偷被类别载入器载入了记忆体之中。类别载入器的功用，就是把类别从静态的硬碟里(.class 档)，复制一份放到记忆体之中，并做一些初始化的工作，让这个类别”活起来”，其他人就能够使用它的功能。

有关于基础类别函式库，在第一章的时候曾经提到，java.exe 是利用几个基本原则来寻找 Java Runtime Environment(JRE)，然后把类别档(.class)直接转交给 JRE 执行之后，java.exe 就功成身退。类别载入器也是构成 JRE 的其中 5
一个重要成员，所以最后类别载入器就会自动从所在之 JRE 目录底下的 \lib\rt.jar 载入基础类别函式库。所以在上图里，一定是因为 java.exe 定位到 c:\j2sdk1.4.0\jre，所以才会有此输出结果。因此，如果 java.exe 定位到其他的 JRE，输出结果就会有稍许的不同，如下图所示:

上图就是 java.exe 定位到位于 C:\Program Files\Java\j2re1.4.0 这个 JRE

时，所输出的结果，各位可以看到，类别载入器改由从 C:\Program Files\Java\j2re1.4.0\lib\rt.jar 之中载入。

预先载入与依需求载入

上述的萤幕输出还透露了一个讯息，就是- 我们自己所撰写的类别(A.class 与 B.class)只会在"用到"的时候才载入(Main.class 是起始类别，所以一定比 A.class 和 B.class 优先载入)，而不是像基础类别函式库(位于 rt.jar 之中的类别)一次一股脑地全部载入记忆体之中，所以萤幕上才会输出先载入了 A.class，然后印出"Using Class A"，再印出载入了 B.class 的讯息，然后再印出"Using Class B"。

像基础类别函式库这样的载入方法我们叫做预先载入(pre-loading)，这是因为基础类别函式库里头的类别大多是 Java 程式执行时所必备的类别，所以为了不要老是做浪费时间的 I/O 动作(读取档案系统，然后将类别档载入记忆体之中)，预先载入这些类别会让 Java 应用程式在执行时速度稍微快一些。相对来说，我们自己所撰写的类别之载入方式，叫做依需求载入(load-on-demand)，也就是 Java 程式真正用到该类别的时候，才真的把类别档从档案系统之中载入记忆体。

看到上述的说明，大家就会立刻有了结论:『只要参考到特定类别，类别载入器就会自动帮我们载入类别。』这个结论对吗? 我们来试试修改后的 6 Main.java:

档案:Main.java

```
public class Main
{
  public static void main(String args[])
 {
    A a1 = new A() ;

  B b1 ;

 }
}
```

执行

```
java –verbose:class Main
```

之后，萤幕上的输出如下:

这个输出告诉我们，只有单独宣告(如: B 类别)而已，是不会促使类别载入器帮我们载入类别的，只有实体化指令(new XXX())才会让类别载入器帮我们载入该类别。

依需求载入的方式，可以让执行时期所占用的记忆体变小，这是因为我们的 Java 程式可能是由数以百计的类别所构成，但是不是每一种类别都会在执行的时候使用，因此依需求载入的方式，可以让需要的类别载入记忆体，而不需要的类别不会被载入。这种机制这在记忆体不多的装置上特别有用，因为 Java 最初就是为了嵌入式系统而设计，嵌入式装置拥有的记忆体通常很小，所以 Java 采这种设计方式有其历史因素。底下就是一个利用依需求载入功能来减少记忆体用量的例子:

档案:Office.java

```
public class Office
{
  public static void main(String args[])
 {
  if(args[0].equals("Word")) 7
  {
   Word w = new Word() ;
   w.print() ;
  }else if(args[0].equals("Excel"))
  {
   Excel e = new Excel() ;
   e.print() ;
  }
 }
}
```

档案:Word.java

```
public class Word
{
 public void print()
 {
  System.out.println("Using Word") ;
 }
}
```

档案:Excel.java

```
public class Excel
{
 public void print()
 {
  System.out.println("Using Excel") ;
 }
}
```

假设您有一个主程式叫 Office，而 Office 又有两个主要的应用程式 Word 及 Excel，两个应用程式执行时都需要占用很大的记忆体空间，而一般很少人同时用到这两个应用程式，这时利用上述 Office.java 的程式写法，就可以省去很多记忆体。当我们执行指令:

java Office Word

时，程式就会载入 Word.class，如下图:

而执行指令:

java Office Excel

时，程式就会载入 Excel.class，如下图:

依需求载入的优点是节省记忆体，但是仍有其缺点。举例来说，当程式第一次用到该类别的时候，系统就必须花一些额外的时间来载入该类别，使得整体执行效能受到影响，尤其是由数以万计的类别所构成的 Java 程式。可是往后需要用到该类别时，由于类别在初次载入之后就会被永远存放在记忆体之中，直到 Java 虚拟机器关闭，所以不再需要花费额外的时间来载入。
总的来说，就弹性上和速度上的考量，如此的设计所带来的优点(弹性和省记忆体)远超过额外载入时间的花费(只有第一次用到时) ，因此依需求载入的设计
是明智的选择。
■让 Java 程式具有动态性的两种方法
Java 本质上具有的动态性，带来了记忆体的节省、程式的弹性、以及一些额外的载入时间。既然谈到了动态性， 『那么，上述的程式是"有弹性"的写法吗? 』一些写程式的老手一定会这样问。严格来说，上述的程式只达到了一点点的弹性，就是让使用者可以在执行的时候选择载入哪个类别。接下来，笔者将示范让程式具有更多弹性的做法。
要让程式具有弹性，就必须利用 Java 所提供的动态性来完成。 Java 提供两种方法来达成动态性。一种是隐式的(implicit)，另一种是显式的(explicit)。这两种方式底层用到的机制完全相同，差异只有在程式设计师所使用的程式码有所不同，隐式的方法可以让您在不知不觉的情况下就使用，而显式的方法必须加入一些额外的程式码。您可以把这两种方法和 Win32 应用程式呼叫动态联结函

式库(Dynamic Linking Library)时的两种方法(implicit 与 explicit)来类比，意思几乎相同。
隐式的(implicit)方法我们已经谈过了，也就是当程式设计师用到 new 这个 Java 关键字时，会让类别载入器依需求载入您所需要的类别，这种方式使用了隐式的(implicit)方法，笔者在前面提到『一般使用 Java 程式语言来开发应用程式的工程师眼中，很少有机会能够察觉 Java 因为具备了动态性之后所带来的优点和特性，甚至根本不曾利用过这个 Java 先天就具有的特性。这不是我们的错，而是因为这个动态的本质被巧妙地隐藏起来，使得使用 Java 的程式设计师在不知不觉中用到了动态性而不自知』 ，就是因为如此。但是，隐式的(implicit)方法仍有其限制，无法达成更多的弹性，遇到这种情况，我们就必须动用显式的(explicit)方法来完成。
显式的方法，又分成两种方式，一种是藉由 java.lang.Class 里的 forName()方法，另一种则是藉由 java.lang.ClassLoader 里的 loadClass()方法。您可以任意选用其中一种方法。
隐式的 隐式的
使用 new new 关键字
Java 所提供的动态性
Java Java 提供的动态性 提供的动态性

显式的 显式的
使用 Class Class 的
forName forName 方法
使用 ClassLoader ClassLoader 的
loadClass loadClass 方法

用显式的方法来达成动态性:使用 Class.forName()
方法

使用显式的方法来达成动态性，意味着我们要自己动手处理类别载入时的细节部分。处理细节部分虽然需要撰写一些额外的程式码，但是可以让程式变的更具弹性，我们以上面这个 Office.java、Word.java、以及 Excel.java 的例子来说，这个程式虽然很有弹性(可以让执行程式的人在执行时期决定要载入哪个类
10
别) ，但是，如果我们新增了 Access.java 和 PowerPoint.java 这两个新类别时，Office.java 里的主程式就必须增加两个 if … else 的回圈。身为一个强调程式维护性的工程师，接下来要问的一定是:「那么，有没有更好的方法，可以在不修改主程式的情况下增加主程式的功能?」 有的，使用显式的方法所达成的动态性，可以增加程式的弹性，并达成我们不希望修改主程式的需求。程式码如下所示:

档案:Assemblyjava

```
public interface Assembly
{
 public void start() ;
}
```

档案:Office.java

```
public class Office
{
  public static void main(String args[]) throws Exception
 {
    Class c = Class.forName(args[0]) ;
    Object o = c.newInstance() ;
    Assembly a = (Assembly) o ;
  a.start() ;
 }
}
```

档案:Word.java

```
public class Word implements Assembly
{
 public void start()
 {
```

```
  System.out.println("Word starts") ;
 }
}
```

档案:Excel.java

```
public class Excel implements Assembly
{
 public void start() 11
 {
  System.out.println("Excel starts") ;
 }
}
```

如此一来，我们的主程式 Office.java 只要编译之后，往后只要叫用:
java Office Word 或 java Office Excel
就可以动态载入我们需要的类别，如下图所示:

除此之外，输入
java Office Access
的时候，虽然会出现错误讯息(因为我们还没有完成 Access.java)，如下图所示:

但是，一旦往后我们完成了 Access(Access.java) 这个类别，或是 PowerPoint(PowerPoint.java)，只要他们都实作了 Assembly 这个介面，我们

就可以在不修改主程式(Office.java)的情况下，新增主程式的功能:
档案:Access.java

```
public class Access implements Assembly
{
 public void start()
 {
  System.out.println("Access starts") ;
 }
}
```

档案:PowerPoint.java

```
public class PowerPoint implements Assembly
{
 public void start()
 {
  System.out.println("PowerPoint starts") ;
 }
}
```

如果您用过 JDBC 撰写资料库程式，使用 Class.forName()来动态载入类别的功能，正是 JDBC 里头用来动态载入 JDBC 驱动程式(JDBC Software driver)的方式。
注意: 请仔细端看加入–verbose:class 之后的萤幕输出，您会看到 Assembly.class 也被系统载入了。在此您可以发现，interface 如同 class 一般，会由编译器产生一个独立的类别档(.class)，当类别载入器载入类别时，如果发现该类别继承了其他类别，或是实作了其他介面，就会先载入代表该介面的类别档，也会载入其父类别的类别档，如果父类别也有其父类别，也会一并优先载入。换句话说，类别载入器会依继承体系最上层的类别往下依序载入，直到所有的祖先类别都载入了，才轮到自己载入。举例来说，如果有个类别 C 继承了类别 B、实作了介面 I，而 B 类别又继承自 A 类别，那么载入的顺序如下图:

如果您亲自搜寻 Java 2 SDK 说明档内部对于 Class 这个类别的说明，您可以发现其实有两个 forName()方法，一个是只有一个参数的(就是之前程式之中所使用的):
public static Class forName(String className) 13
另外一个是需要三个参数的:
public static Class forName(String name, boolean initialize,
    ClassLoader loader)
这两个方法，最后都是连接到原生方法 forName0()，其宣告如下:
private static native Class forName0(String name, boolean initialize,
ClassLoader loader) throws ClassNotFoundException;
只有一个参数的 forName()方法，最后叫用的是:
forName0(className, true, ClassLoader.getCallerClassLoader());
而具有三个参数的 forName()方法，最后叫用的是:
forName0(name, initialize, loader);

其中，关于名为 initialize 这个参数的用法，请看范例程式:
档案:Office.java

```
public class Office
{
  public static void main(String args[]) throws Exception
 {
    Class c = Class.forName(args[0],true,null) ;
    Object o = c.newInstance() ;
    Assembly a = (Assembly) o ;
  a.start() ;
 }
}
```

执行时输入
java Office Word
萤幕上的输出为:

之所以产生画面上的错误讯息，是因为我们在类别载入器的部分给的参数是 null，导致系统不知道用哪个类别载入器来载入类别档，因而发生错误。如果把程式修改如下:

档案:Office.java

```
public class Office
{ 14
  public static void main(String args[]) throws Exception
 {
    Office off = new Office() ;
    Class c = Class.forName(args[0],true,off.getClass().getClassLoader()) ;
    Object o = c.newInstance() ;
    Assembly a = (Assembly) o ;
  a.start() ;
 }
}
```

输出就正常了:

从这里我们可以知道，forName()方法的第三个参数，是用来指定载入类别的类别载入器的，只有一个参数的 forName()方法，由于在内部使用了 ClassLoader.getCallerClassLoader()来取得载入呼叫他的类别所使用的类别载入器，和我们自己写的程式有相同的效用。（注意，ClassLoader.getCallerClassLoader()是一个 private 的方法，所以我们无法自行叫用，因此必须要自己产生一个 Office 类别的实体，再去取得载入 Office 类别时所使用的类别载入器)。

那么 forName()的第二个参数的效用为何？我们修改主程式如下:

档案:Office.java

```
public class Office
{
  public static void main(String args[]) throws Exception
 {
    Office off = new Office() ;
  System.out.println("类别准备载入") ;
    Class c = Class.forName(args[0],true,off.getClass().getClassLoader()) ;
         System.out.println("类别准备实体化") ;
    Object o = c.newInstance() ;
Object o2 = c.newInstance() ;
 }
}
```

而 Word.java 之中则加入静态初始化区块: 15

档案:Word.java

```
public class Word implements Assembly
{
```

```
 static
 {
  System.out.println("Word static initialization") ;
 }
 public void start()
 {
  System.out.println("Word starts") ;
 }
}
```

执行:
java Office Word
时的结果如下:

如果 forName()方法的第二个参数给的是 false:
档案:Office.java

```
public class Office
{
  public static void main(String args[]) throws Exception
 {
    Office off = new Office() ;
  System.out.println("类别准备载入") ;
    Class c = Class.forName(args[0],false,off.getClass().getClassLoader()) ;
         System.out.println("类别准备实体化") ;
    Object o = c.newInstance() ;
         Object o2 = c.newInstance() ;
 }
}
```

则输出变成: 16

使用 true 和 false 会造成不同的输出结果，您看出端倪了吗? 过去在很多 Java 的书本上提到静态初始化区块(static initialization block)时，都会说「静态初始化区块是在类别第一次载入的时候才会被呼叫那仅仅一次。」 可是从上面的输出，却发现即使类别被载入了，其静态初始化区块也没有被呼叫，而是在第一次叫用 newInstance()方法时，静态初始化区块才真正被叫用。所以严格来说，应该改成「静态初始化区块是在类别第一次被实体化的时候才会被呼叫那仅仅一次。」
所以，我们得到以下结论:不管您使用的是 new 来产生某类别的实体、或是使用只有一个参数的 forName()方法，内部都隐含了”载入类别+呼叫静态初始化区块”的动作。而使用具有三个参数的 forName()方法时，如果第二个参数给定的是 false，那么就只会命令类别载入器载入该类别，但不会叫用其静态初始化区块，只有等到整个程式第一次实体化某个类别时，静态初始化区块才会被叫用。
■用显式的方法来达成动态性:直接使用类别载入器
要直接使用类别载入器帮我们载入类别，首先必须先取得指向类别载入器的

参考才行，而要取得指向类别载入器的参考之前，必须先取得某物件所属的类别。相信在大家的认知里，都有个基本的概念，就是:「类别是一个样版，而物件就是根据这个样版所产生出来的实体」 。在 Java 之中，每个类别最后的老祖
宗都是 Object，而 Object 里有一个名为 getClass()的方法，就是用来取得某特定实体所属类别的参考，这个参考，指向的是一个名为 Class 类别(Class.class) 的实体，您无法自行产生一个 Class 类别的实体，因为它的建构式被宣告成 private，这个 Class 类别的实体是在类别档(.class)第一次载入记忆体时就建立的，往后您在程式中产生任何该类别的实体，这些实体的内部都会有一个栏位记录着这个 Class 类别的所在位置。概念上如下图所示: 17

類别实体与類别实体与 Class.class Class.class

Class.class Class.class 的实体 的实体
为档案系统中某为档案系统中某 X.class X.class

在记忆体中的代理人在记忆体中的代理人

X.class 的实体
Class.class Class.class 的实体 的实体
为档案系统中某为档案系统中某 Y.class Y.class

在记忆体中的代理人在记忆体中的代理人

X.class 的实体
Y.class 的实体

基本上，我们可以把每个 Class 类别的实体，当作是某个类别在记忆体中的代理人。每次我们需要查询该类别的资料(如其中的 field、method 等)时，就可以请这个实体帮我们代劳。事实上，Java 的 Reflection 机制，就大量地利用 Class 类别。去深入 Class 类别的原始码，我们可以发现 Class 类别的定义中大多数的方法都是原生方法(native method)。

在 Java 之中，每个类别都是由某个类别载入器(ClassLoader 的实体)来载入，因此，Class 类别的实体中，都会有栏位记录着载入它的 ClassLoader 的实体(注意:如果该栏位是 null，并不代表它不是由类别载入器所载入，而是代表这个类别由靴带式载入器(bootstrap loader,也有人称 root loader)所载入，只不过因为这个载入器并不是用 Java 所写成，所以逻辑上没有实体)。其概念如下图所示: 18

類别实体類别实体, ,Class.class, Class.class,与与 ClassLoader ClassLoader

Class.class Class.class 的实体 的实体
((X.class) X.class)
X.class 的实体
Class.class Class.class 的实体 的实体
((Y.class) Y.class)
X.class 的实体
Y.class 的实体

Z.class 的实体
Z.class 的实体
Z.class 的实体 Class.class Class.class 的实体的实体
((Z.class) Z.class)
ClassLoader
实体
ClassLoader
实体

从上图我们可以得之，系统里同时存在多个 ClassLoader 的实体，而且一个类别载入器不限于只能载入一个类别，类别载入器可以载入多个类别。所以，只要取得 Class 类别实体的参考，就可以利用其 getClassLoader()方法篮取得载入该类别之类别载入器的参考。 getClassLoader()方法最后会呼叫原生方法 getClassLoader0()，其宣告如下:

```
private native ClassLoader getClassLoader0();
```

最后，取得了 ClassLoader 的实体，我们就可以叫用其 loadClass()方法帮我们载入我们想要的类别。因此，我们把程式码修改如下:

档案:Office.java

```
public class Office
{
  public static void main(String args[]) throws Exception
 {
    Office off = new Office() ;
  System.out.println("类别准备载入") ;
    ClassLoader loader = off.getClass().getClassLoader() ;
    Class c = loader.loadClass(args[0]) ;
              System.out.println("类别准备实体化") ;
    Object o = c.newInstance() ;
  Object o2 = c.newInstance() ;
 }
}
```

执行 19

```
java Office Word
```

之后，结果如下:

从这里我们也可以看出，直接使用 ClassLoader 类别的 loadClass()方法来载入类别，只会把类别载入记忆体，并不会叫用该类别的静态初始化区块，而必须等到第一次实体化该类别时，该类别的静态初始化区块才会被叫用。这种情形与使用 Class 类别的 forName()方法时，第二个参数传入 false 几乎是相同的结果。

上述的程式其实还有另外一种写法如下所示:

档案:Office.java

```
public class Office
{
  public static void main(String args[]) throws Exception
```

```
{
  Class cb = Office.class ;
 System.out.println("类别准备载入") ;
  ClassLoader loader = cb.getClassLoader() ;
   Class c = loader.loadClass(args[0]) ;
 System.out.println("类别准备实体化") ;
   Object o = c.newInstance() ;
 Object o2 = c.newInstance() ;
}
}
```

直接在程式里头使用 Office.class，就是直接取得某特定实体所属类别的参考，用起来比起产生 Office 的实体，再用 getClass()取出，这个方法方便的多，也较省记忆体。

自己建立类别载入器来载入类别

在此之前，当我们谈到使用类别载入器来载入类别时，都是使用既有的类别载入器来帮我们载入我们所指定的类别。那么，我们可以自己产生类别载入器来帮我们载入类别吗 ？ 答案是肯定的。利用 Java 本身提供的 java.net.URLClassLoader 类别就可以做到，范例如下:

档案:Office.java

```
import java.net.* ; 20
public class Office
{
 public static void main(String args[]) throws Exception
{
   URL u = new URL("file:/d:/my/lib/") ;
   URLClassLoader ucl = new URLClassLoader(new URL[]{ u }) ;
   Class c = ucl.loadClass(args[0]) ;
   Assembly asm = (Assembly) c.newInstance() ;
 asm.start() ;
}
}
```

在这个范例中，我们自己产生 java.net.URLClassLoader 的实体来帮我们载入我们所需要的类别。但是载入前，我们必须告诉 URLClassLoader 去哪个地方寻找我们所指定的类别才行，所以我们必须给它一个 URL 类别所构成的阵列，代表我们希望它去搜寻的所有位置。URL 可以指向网际网路上的任何位置，也可以指向我们电脑里的档案系统(包含 JAR 档)。在上述范例中，我们希望 URLClassLoader 到 d:\my\lib\这个目录下去寻找我们需要的类别，所以指定的 URL 为”file:/d:/my/lib/”。其实，如果我们请求的位置是主要类别(有 public static void main(String arga[])方法的那个类别)的相对目录，我们可以在 URL 的地方只写”file:lib/”，代表相对于目前的目录。

我们再把程式修改如下:

档案:Office.java

```
import java.net.* ;
public class Office
```

```
{
  public static void main(String args[]) throws Exception
 {
    URL u = new URL("file:/d:/my/lib/") ;
    URLClassLoader ucl = new URLClassLoader(new URL[]{ u }) ;
    Class c = ucl.loadClass(args[0]) ;
    Assembly asm = (Assembly) c.newInstance() ;
  asm.start() ;

    URL u1 = new URL("file:/d:/my/lib/") ;
    URLClassLoader ucl1 = new URLClassLoader(new URL[]{ u1 }) ;
    Class c1 = ucl1.loadClass(args[0]) ;
    Assembly asm1 = (Assembly) c1.newInstance() ;
  asm1.start() ; 21
 }
}
```

您将发现萤幕输出如下:

您可以发现，同样一个类别，却被不同的 URLClassLoader 分别载入，而且分别初始化一次。也就是说，在一个虚拟机器之中，相同的类别被载入了两次。

注意!

此范例中，我们的目录结构如下图:

类别被哪个类别载入器载入?

我们将上述的程式码稍作修改，修改后的程式码如下:

档案:Office.java

```
import java.net.* ;
public class Office
{
  public static void main(String args[]) throws Exception 22
 {
    URL u = new URL("file:/d:/my/lib/") ;
    URLClassLoader ucl = new URLClassLoader(new URL[]{ u }) ;
    Class c = ucl.loadClass(args[0]) ;
    Assembly asm = (Assembly) c.newInstance() ;
  asm.start() ;

    URL u1 = new URL("file:/d:/my/lib/") ;
    URLClassLoader ucl1 = new URLClassLoader(new URL[]{ u1 }) ;
    Class c1 = ucl1.loadClass(args[0]) ;
    Assembly asm1 = (Assembly) c1.newInstance() ;
  asm1.start() ;
```

```
  System.out.println(Office.class.getClassLoader()) ;
  System.out.println(u.getClass().getClassLoader()) ;
  System.out.println(ucl.getClass().getClassLoader()) ;
  System.out.println(c.getClassLoader()) ;
  System.out.println(asm.getClass().getClassLoader()) ;

  System.out.println(u1.getClass().getClassLoader()) ;
  System.out.println(ucl1.getClass().getClassLoader()) ;
  System.out.println(c1.getClassLoader()) ;
  System.out.println(asm1.getClass().getClassLoader()) ;
 }
}
```

执行后输出结果如下图:

从输出中我们可以得知，Office.class 由 AppClassLoader(又称做 System 23 Loader ，系统载入器)所载入，URL.class 与 URLClassLoader.class 由 Bootstrap Loader 所载入(注意:输出 null 并非代表不是由类别载入器所载入。在 Java 之中，所有的类别都必须由类别载入器载入才行，只不过 Bootstrap Loader 并非由 Java 所撰写而成，而是由 C++实作而成，因此以 Java 的观点来看，逻辑上并没有 Bootstrap Loader 的类别实体)。而 Word.class 分别由两个不同的 URLClassLoader 实体载入。至于 Assembly.class ，本身应该是由 AppClassLoader 载入，但是由于多型(Polymorphism)的关系，所指向的类别实体(Word.class)由特定的载入器所载入，导致列印在萤幕上的内容是其所参考的类别实体之类别载入器。 Interface 这种型态本身无法直接使用 new 来产生实体，所以在执行 getClassLoader()的时候，叫用的一定是所参考的类别实体的 getClassLoader()，要知道 Interface 本身由哪个类别载入器载入，您必须使用底下程式码:

```
Assembly.class.getClassLoader()
```

如果把这行程式加在上述程式之中，您会发现其输出结果为

这个输出告诉您，Assembly.class 是由 AppClassLoader 载入。

一切都是由 Bootstrap Loader 开始: 类别载入器的阶层体系

注意!

如果您参考其他资料，您将发现在讲解类别载入器时，资料中常常会提到「类别载入器的阶层体系」 (classloader hierarchy)。请注意，这里所说的继承体系，并非我们一般所指的类别阶层体系(即父类别与子类别构成的体系)。 「类别载入器的阶层体系」的意涵另有所指，请不要混淆了。笔者接下来将为您解释「类别载入器的阶层体系」所代表的真正意涵。

在前面我们曾经提过，Java 程式在编译之后会产生许多的执行单位(.class 档)，当我们执行主要类别时(有 public static void main(String arga[])方法的那个类别)，才由虚拟机器一一载入所有需要的执行单位，变成一个逻辑上为一

体的 Java 应用程式。因此接下来，我们将细部讨论这整个流程。

当我们在命令列输入 java xxx.class 的时候，java.exe 根据我们之前所提过的逻辑找到了 JRE(Java Runtime Environment)，接着找到位在 JRE 之中的 jvm.dll(真正的 Java 虚拟机器)，最后载入这个动态联结函式库，启动 Java 虚拟机器。这个动作的详细介绍请回头参阅第一章。

虚拟机器一启动，会先做一些初始化的动作，比方说抓取系统参数等。一旦初始化动作完成之后，就会产生第一个类别载入器，即所谓的 Bootstrap Loader，Bootstrap Loader 是由 C++所撰写而成(所以前面我们说，以 Java 24 的观点来看，逻辑上并不存在 Bootstrap Loader 的类别实体，所以在 Java 程式码里试图印出其内容的时候，我们会看到的输出为 null)，这个 Bootstrap Loader 所做的初始工作中，除了也做一些基本的初始化动作之外，最重要的就是载入定义在 sun.misc 命名空间底下的 Launcher.java 之中的 ExtClassLoader(因为是 inner class ，所以编译之后会变成 Launcher$ExtClassLoader.class)，并设定其 Parent 为 null，代表其父载入器为 Bootstrap Loader。然后 Bootstrap Loader 再要求载入定义于 sun.misc 命名空间底下的 Launcher.java 之中的 AppClassLoader(因为是 inner class，所以编译之后会变成 Launcher$AppClassLoader.class)，并设定其 Parent 为之前产生的 ExtClassLoader 实体。这里要请大家注意的是，Launcher$ExtClassLoader.class 与 Launcher$AppClassLoader.class 都是由 Bootstrap Loader 所载入，所以 Parent 和由哪个类别载入器载入没有关系。我们可以用下图来表示:

Bootstrap Loader
(c++) Extended Loader
(Java)
Parent
载入
System Loader
(Java)
Parent 载入

AppClassLoader 在 Sun 官方文件中常常又被称做系统载入器(System Loader)，但是在本文中为了避免混淆，所以还是称作 AppClassLoader。最后一个步骤，是由 AppClassLoader 负责载入我们在命令列之中所输入的 xxx.class(注意:实际上 xxx.class 很可能由 ExtClassLoader 或 Bootstrap Loader 载入，请参考底下「委派模型」一节)，然后开始一个 Java 应用程式的生命周期。上述整个流程如下图所示: 25

找到 找到 JRE JRE

命令列:java xxx.class

找到 找到 jvm.dll jvm.dll
载入
ExtClassLoader ExtClassLoader
产生
Bootstrp Loader Bootstrp Loader

启动 启动 JVM JVM

并进行初始化 并进行初始化

载入
AppClassLoader AppClassLoader
载入

这个由 Bootstrap Loader Î ExtClassLoader Î AppClassLoader，就是我们所谓「类别载入器的阶层体系」 。我们可以用底下的程式码证明这一点:
档案:test.java

```
public class test
{
  public static void main(String args[])
 {
  ClassLoader cl = test.class.getClassLoader() ;
  System.out.println(cl) ;
    ClassLoader cl1 = cl.getParent() ;
  System.out.println(cl1) ;
    ClassLoader cl2 = cl1.getParent() ;
  System.out.println(cl2) ;
 }
}
```

输出结果如下:

如果在上述程式中，如果您使用程式码:
cl.getClass().getClassLoader()及 cl1.getClass().getClassLoader()，您会发 26 现印出的都是 null，这代表它们都是由 Bootstrap Loader 所载入。这里也再次强调，类别载入器由谁载入(这句话有点诡异，类别载入器也要由类别载入器载入，这是因为除了 Bootstrap Loader 之外，其余的类别载入器皆是由 Java 撰写而成)，和它的 Parent 是谁没有关系，Parent 的存在只是为了某些特殊目的，这个目的我们将在稍后作解释。这三个主要载入器的关系如下图所示:
ExtClassLoader ExtClassLoader
Bootstrp Loader Bootstrp Loader
AppClassLoader AppClassLoader
getClassLoader( )
Parent
Parent
null

在此要请大家注意的是，AppClassLoader 和 ExtClassLoader 都是 URLClassLoader 的子类别。由于它们都是 URLClassLoader 的子类别，所以它们也应该有 URL 作为搜寻类别档的参考，由原始码中我们可以得知，AppClassLoader 所参考的 URL 是从系统参数 java.class.path 取出的字串所决定，而 java.class.path 则是由我们在执行 java.exe 时，利用–cp 或

-classpath 或 CLASSPATH 环境变数所决定。我们可以用底下程式码测试之:
档案:test.java

```
public class test
{
  public static void main(String args[])
 {
    String s = System.getProperty("java.class.path");
  System.out.println(s) ;
 }
}
```

输出结果如下: 27

从这个输出结果，我们可以看出，在预设情况下，AppClassLoader 的搜寻路径为”.”(目前所在目录)，如果使用-classpath 选项(与-cp 等效)，就可以改变 AppClassLoader 的搜寻路径，如果没有指定-classpath 选项，就会搜寻环境变数 CLASSPATH。如果同时有 CLASSPATH 的环境设定与-classpath 选项，则
以-classpath 选项的内容为主，CLASSPATH 的环境设定与-classpath 选项两者
的内容不会有加成的效果。
至于 ExtClassLoader 也有相同的情形，不过其搜寻路径是参考系统参数 java.ext.dirs。我们可以用底下程式码测试:
档案:test.java

```
public class test
{
  public static void main(String args[])
 {
    String s = System.getProperty("java.ext.dirs");
  System.out.println(s) ;
 }
}
```

输出结果如下: 28

输出结果告诉我们，系统参数 java.ext.dirs 的内容，会指向 java.exe 所选择的 JRE 所在位置下的\lib\ext 子目录。系统参数 java.ext.dirs 的内容可以在一开始下命列的时候来更改，如下:

最后一个类别载入器是 Bootstrap Loader，我们可以经由查询由系统参数 sun.boot.class.path 得知 Bootstrap Loader 用来搜寻类别的路径。请使用底下的程式码测试之:
档案:test.java

```
public class test
{
  public static void main(String args[])
```

```
{
   String s = System.getProperty("sun.boot.class.path");
  System.out.println(s) ;
 }
}
```
输出结果如下:

系统参数 sun.boot.class.path 的内容可以在一开始下命列的时候来更改，如下:
29

从这三个类别载入器的搜寻路径所参考的系统参数的名字中，其实还透漏了一个讯息。请回头看到 java.class.path 与 sun.boot.class.path，也就是说，AppClassLoader 与 Bootstrap Loader 会搜寻它们所指定的位置(或 JAR 档) ，如果找不到就找不到了，AppClassLoader 与 Bootstrap Loader 不会递回式地搜寻这些位置下的其他路径或其他没有被指定的 JAR 档。反观 ExtClassLoader，所参考的系统参数是 java.ext.dirs，意思是说，他会搜寻底下的所有 JAR 档以及 classes 目录，作为其搜寻路径(所以您会发现上面我们在测试的时候，如果加入-Dsun.boot.class.path=c:\winnt 选项时，程式的起始速度会慢了些，这是因为 c:\winnt 目录下的档案很多，必须花额外的时间来列举 JAR 档)。
在命令列下参数时，使用–classpath / -cp / 环境变数 CLASSPATH 来更改 AppClassLoader 的搜寻路径，或者用–Djava.ext.dirs 来改变 ExtClassLoader 的搜寻目录，两者都是有意义的。可是用–Dsun.boot.class.path 来改变 Bootstrap Loader 的搜寻路径是无效。这是因为 AppClassLoader 与 ExtClassLoader 都是各自参考这两个系统参数的内容而建立，当您在命令列下变更这两个系统参数之后， AppClassLoader 与 ExtClassLoader 在建立实体的时候会参考这两个系统参数，因而改变了它们搜寻类别档的路径;而系统参数 sun.boot.class.path 则是预设与 Bootstrap Loader 的搜寻路径相同，就算您更改该系统参与，与 Bootstrap Loader 完全无关。如果您手边有原始码，以 JDK 1.4.x 为例，请参考<原始码根目录>\hotspot\src\share\vm\runtime\os.cpp 这个档案里头的 os::set_boot_path 方法，您将看到程式码片段:

```
static const char classpathFormat[] =
 "%/lib/rt.jar:"
 "%/lib/i18n.jar:"
 "%/lib/sunrsasign.jar:"
 "%/lib/jsse.jar:"
 "%/lib/jce.jar:"
    "%/lib/charsets.jar:"
 "%/classes";
```

JDK 1.4.x 比 JDK 1.3.x 新增了一些核心类别函式库(例如 jsse.jar 与 jce.jar)，所以如果您测试时用的是 1.4.x 版的 JDK，sun.boot.class.path 内容应该如下: 30

更重要的是，AppClassLoader 与 ExtClassLoader 在整个虚拟机器之中只会存有一份，一旦建立了，其内部所参考的搜寻路径将不再改变，也就是说，即使我们在程式里利用 System.setProperty()来改变系统参数的内容，仍然无法更动 AppClassLoader 与 ExtClassLoader 的搜寻路径。因此，执行时期动态更改搜寻路径的设定是不可能的事情。如果因为特殊需求，有些类别的所在路径并非在一开始时就能决定，那么除了产生新的类别载入器来辅助我们载入所需的类别之外，没有其他方法了。

以下范例说明即使更改系统参数 java.class.path，仍然无法变更 AppClassLoader 载入类别时所使用的搜寻路径。

档案:test.java

```
public class test
{
  public static void main(String args[])
 {
    String old = System.getProperty("java.class.path");
  System.out.println(old) ;

  System.setProperty("java.class.path","d:\\test");
    String s = System.getProperty("java.class.path");
  System.out.println(s) ;

  testlib tl = new testlib() ;
  tl.print() ;
 }
}
```

档案:testlib.java

```
public class testlib
{
 public void print()
 {
  System.out.println("I get loaded") ;
 } 31
}
```

(注意: test.java 与 testlib.java 放在同一个目录下)

执行结果为:

因为我们并没有在 d:\test 这个路径底下放上 testlib.class，所以如果更改系统参数可以影响类别载入器的话，理论上应该无法正常地载入 testlib.class，可是萤幕输出确告诉我们 testlib.class 被成功地载入。这个范例足以证明 AppClassLoader 完全不受系统参数 java.class.path 的更改而产生影响，您一样可以试着测试改变 java.ext.dirs 来测试它对 ExtClassLoader 的影响。

委派模型

前面我们曾经提过「Bootstrap Loader 所做的初始工作中，除了也做一些基本的初始化动作之外，，最重要的就是载入定义在 sun.misc 命名空间底下的

Launcher.java 之中的 ExtClassLoader，并设定其 Parent 为 null，然后 Bootstrap Loader 再载入定义在 sun.misc 命名空间底下的 Launcher.java 之中的 AppClassLoader ，并设定其 Parent 为之前产生的 ExtClassLoader 实体。」这个动作产生了所谓的「类别载入器的阶层体系」 ，如下图所示:
ExtClassLoader ExtClassLoader
Bootstrp Loader Bootstrp Loader
AppClassLoader AppClassLoader
Parent
Parent
null

而之所以有阶层体系的存在，是为了实现委派模型。所谓的委派模型，用简单的话来讲，就是「类别载入器有载入类别的需求时，会先请示其 Parent 使用

其搜寻路径帮忙载入，如果 Parent 找不到，那么才由自己依照自己的搜寻路径搜寻类别」 ，请注意，这句话具有递回性。底下将用几个小测试来说明这句话的
意涵。测试前，我们将撰写两个类别: test.java 与 testlib.java，内容如下:
档案:test.java

```
public class test
{
  public static void main(String args[])
 {
  System.out.println(test.class.getClassLoader()) ;
  testlib tl = new testlib() ;
  tl.print() ;
 }
}
```

档案:testlib.java

```
public class testlib
{
 public void print()
 {
  System.out.println(this.getClass().getClassLoader()) ;
 }
}
```

这两个类别的作用就是印出载入它们的类别载入器是谁。将它们编译成 test.class 与 testlib.class 之后，我们再复制两份，分别至于<JRE 所在目录>\classes 底下(注意，您的系统下应该没有此目录，您必须自己建立)与<JRE 所在目录>\lib\ext\classes(注意，您的系统下应该没有此目录，您必须自己建立)底下，如下图所示: 33

而 d:\my 底下也保有一份: 34

然后我们切换到 d:\my 目录下，输入 java test 开始进行测试(注意，此测试指令和路径在底下的测试中将永远不会改变)，测试前，我们会先用表格表示目前 test.class 与 testlib.class 的分布情况，然后解说不厌其烦地解说执行结果的成因，有些执行结果会发生错误，我们也会说明如何解决(或者是根本无法解决)。
测试一:
<JRE 所在目录>\classes 底下
test.class
testlib.class
<JRE 所在目录>\lib\ext\classes 底下
test.class
testlib.class
d:\my 底下
test.class
testlib.class
输出结果如下图所示:

从输出我们可以看出，当 AppClassLoader 要载入 test.class 时，先请其 Parent，也就是 ExtClassLoader 来载入，而 ExtClassLoader 又请求其 Parent，即 Bootstrap Loader 来载入 test.class。由于<JRE 所在目录>\classes 目录为
35
Bootstrap Loader 的搜寻路径之一，所以 Bootstrap Loader 找到了 test.class，因此将它载入。接着在 test.class 之内有载入 testlib.class 的需求，由于 test.class 是由 Bootstrap Loader 所载入，所以 testlib.class 内定是由 Bootstrap Loader 根据其搜寻路径来寻找，因为 testlib.class 也位于 Bootstrap Loader 可以找到的路径下，所以也被载入了。
最后我们看到 test.class 与 testlib.class 都是由 Bootstrap Loader(null)载入。
测试二:
<JRE 所在目录>\classes 底下
test.class

<JRE 所在目录>\lib\ext\classes 底下
test.class
testlib.class
d:\my 底下
test.class
testlib.class
输出结果如下图所示:

从输出我们可以看出，当 AppClassLoader 要载入 test.class 时，先请其 Parent，也就是 ExtClassLoader 来载入，而 ExtClassLoader 又请求其 Parent，即 Bootstrap Loader 来载入 test.class。由于<JRE 所在目录>\classes 目录为
Bootstrap Loader 的搜寻路径之一，所以 Bootstrap Loader 找到了

test.class，因此将它载入。接着在 test.class 之内有载入 testlib.class 的需求，由于 test.class 是由 Bootstrap Loader 所载入，所以 testlib.class 内定是由 Bootstrap Loader 根据其搜寻路径来寻找，但是因为 Bootstrap Loader 根本找不到 testlib.class(被我们删除了)，而 Bootstrap Loader 又没有 Parent，所以无法载入 testlib.clss。
输出告诉我们，test.class 由 Bootstrap Loader 载入，且最后印出的讯息是 NoClassDefFoundError，代表无法载入 testlib.class。这个问题没有比较简单的方法能够解决，但是仍然可以透过较复杂的 Context Class Loader 来解决。这个技巧在本文中不讨论。
测试三: 36
<JRE 所在目录>\classes 底下

testlib.class
<JRE 所在目录>\lib\ext\classes 底下
test.class
testlib.class
d:\my 底下
test.class
testlib.class
输出结果如下图所示:

从输出我们可以看出，当 AppClassLoader 要载入 test.class 时，先请其 Parent，也就是 ExtClassLoader 来载入，而 ExtClassLoader 又请求其 Parent，即 Bootstrap Loader 来载入 test.class。但是 Bootstrap Loader 无法在其搜寻路径下找到 test.class(被我们删掉了)，所以 ExtClassLoader 只得自己搜寻。因此 ExtClassLoader 在其搜寻路径<JRE 所在目录>\lib\ext\classes 底下找到 test.class，因此将它载入。接着在 test.class 之内有载入 testlib.class 的需求，由于 test.class 是由 ExtClassLoader 所载入，所以 testlib.class 内定是由 ExtClassLoader 根据其搜寻路径来寻找，但是因为 ExtClassLoader 有 Parent，所以要先由 Bootstrap Loader 先帮忙寻找，testlib.class 位于 Bootstrap Loader 可以找到的路径下，所以被 Bootstrap Loader 载入了。最后我们看到 test.class 由 ExtClassLoader 载入，而 testlib.class 则是由 Bootstrap Loader(null)载入。
测试四:
<JRE 所在目录>\classes 底下

<JRE 所在目录>\lib\ext\classes 底下
test.class
testlib.class
d:\my 底下
test.class
testlib.class
输出结果如下图所示: 37

从输出我们可以看出，当 AppClassLoader 要载入 test.class 时，先请其 Parent，也就是 ExtClassLoader 来载入，而 ExtClassLoader 又请求其 Parent，即 Bootstrap Loader 来载入 test.class。但是 Bootstrap Loader 无法在其搜寻路径下找到 test.class(被我们删掉了)，所以 ExtClassLoader 只得自己搜寻。因此 ExtClassLoader 在其搜寻路径<JRE 所在目录>\lib\ext\classes 底下找到 test.class，因此将它载入。接着在 test.class 之内有载入 testlib.class 的需求，由于 test.class 是由 ExtClassLoader 所载入，所以 testlib.class 内定是由 ExtClassLoader 根据其搜寻路径来寻找，ExtClassLoader 一样要请求其 Parent 先试着载入，但是 Bootstrap Loader 根本找不到 testlib.class(被我们删除了)，所以只能由 ExtClassLoader 自己来，ExtClassLoader 在其搜寻路径<JRE 所在目录>\lib\ext\classes 底下找到 testlib.class，因此将它载入。
最后我们看到 test.class 与 testlib.clas 都是由 ExtClassLoader 载入。
测试五:
<JRE 所在目录>\classes 底下

<JRE 所在目录>\lib\ext\classes 底下
test.class

d:\my 底下
test.class
testlib.class
输出结果如下图所示:

从输出我们可以看出，当 AppClassLoader 要载入 test.class 时，先请其 Parent，也就是 ExtClassLoader 来载入，而 ExtClassLoader 又请求其 Parent，即 Bootstrap Loader 来载入 test.class。但是 Bootstrap Loader 无法在其搜寻路径下找到 test.class(被我们删掉了)，所以 ExtClassLoader 只得自己搜寻。因此 ExtClassLoader 在其搜寻路径<JRE 所在目录>\lib\ext\classes 底下找到 38 test.class，因此将它载入。接着在 test.class 之内有载入 testlib.class 的需求，由于 test.class 是由 ExtClassLoader 所载入，所以 testlib.class 内定是由 ExtClassLoader 根据其搜寻路径来寻找，ExtClassLoader 一样要请求其 Parent 先试着载入，但是 Bootstrap Loader 根本找不到 testlib.class(被我们删除了)，所以只能由 ExtClassLoader 自己来，ExtClassLoader 也无法在自己的搜寻路径中找到 testlib.class，所以产生错误讯息。
输出告诉我们，test.class 由 ExtClassLoader 载入，且最后印出的讯息是 NoClassDefFoundError，代表无法载入 testlib.class。要解决问题，我们必须让 ExtClassLoader 找的到 testlib.class 才行，所以请使用选项–Djava.ext.dirs=<路径名称> 来指定，请注意，ExtClassLoader 只会自动搜寻底下的 classes 子目录或是 JAR 档，其他的子目录或其他类型的档案一概不管。此外，这个错误亦可以透过 Context Class Loader 的技巧来解决。
测试六:
<JRE 所在目录>\classes 底下

<JRE 所在目录>\lib\ext\classes 底下

testlib.class
d:\my 底下
test.class
testlib.class
输出结果如下图所示:

从输出我们可以看出，当 AppClassLoader 要载入 test.class 时，先请其 Parent，也就是 ExtClassLoader 来载入，而 ExtClassLoader 又请求其 Parent，即 Bootstrap Loader 来载入 test.class。Bootstrap Loader 无法在其搜寻路径下找到 test.class(被我们删掉了)，所以转由 ExtClassLoader 来搜寻。ExtClassLoader 仍无法在其搜寻路径<JRE 所在目录>\lib\ext\classes 底下找到 test.class，最后只好由 AppClassLoader 载入它自己在搜寻路径底下找到的 test.class。接着在 test.class 之内有载入 testlib.class 的需求，由于 test.class 是由 AppClassLoader 所载入，所以 testlib.class 内定是由 AppClassLoader 根据其搜寻路径来寻找，AppClassLoader 一样要请求其 Parent 先试着载入，但是 Bootstrap Loader 根本找不到 testlib.class(被我们删除了)，所以回头转 39 由 ExtClassLoader 来搜寻，ExtClassLoader 在其搜寻路径<JRE 所在目录>\lib\ext\classes 底下找到 testlib.class，因此将它载入。
最后我们看到 test.class 由 AppClassLoader 载入，而 testlib.class 则是由 ExtClassLoader 载入。
测试七:
<JRE 所在目录>\classes 底下

<JRE 所在目录>\lib\ext\classes 底下

d:\my 底下
test.class
testlib.class
输出结果如下图所示:

从输出我们可以看出，当 AppClassLoader 要载入 test.class 时，先请其 Parent，也就是 ExtClassLoader 来载入，而 ExtClassLoader 又请求其 Parent，即 Bootstrap Loader 来载入 test.class。但是 Bootstrap Loader 无法在其搜寻路径下找到 test.class(被我们删掉了)， ExtClassLoader 只无法在其搜寻路径下找到 test.class(被我们删掉了)，最后只好由 AppClassLoader 载入它在自己搜寻路径底下找到的 test.class。接着在 test.class 之内有载入 testlib.class 的需求，由于 test.class 是由 AppClassLoader 所载入，所以 testlib.class 内定是由 AppClassLoader 根据其搜寻路径来寻找，但是 Bootstrap Loader 根

本找不到 testlib.class(被我们删除了) ， ExtClassLoader 也找不到 testlib.class(被我们删除了)，所以回头转由 AppClassloader 来搜寻，AppClassLoader 在其搜寻路径底下找到 testlib.class，因此将它载入。最后我们看到 test.class 和 testlib.class 都是由 AppClassLoader 载入。

类别载入体系

从以上的测试，我们可以知道一件事，就是:「类别载入器可以看到 Parent 所载入的所有类别，但是反过来并非除此」。所以在前面的测试二中，test.class 由 Bootstrap Loader 载入，且明明 AppClassLoader 可以找到，但是 Bootstrap Loader 就是看不到。如果您有用过 JDBC API 的经验，您就会开始感到奇怪。40 这是因为我们的 JDBC API 介面类别(属于核心类别函式库)都是由 Bootstrap Loader 或是 ExtClassLoader 来载入，可是 JDBC driver 常常是藉由 ExtClassLoader 或 AppClassLoader 来载入。假设 JDBC API 介面类别由 Bootstrap Loader 载入，而 JDBC driver 是藉由 ExtClassLoader 来载入，不就发生了与测试二相同的情形，那么 JDBC 是如何克服这个问题的呢? 其实不只是 JDBC，在 Java 领域中只要分成 API(Application Programming Interface，公开制定，会成为核心类别函式库(由 Bootstrap Loader 载入)或扩充类别函式库(由 ExtClassLoader 载入))与 SPI(Service Provide Interface，由特定厂商撰写，会成为扩充类别函式库(由 ExtClassLoader 载入)或应用程式(由 AppClassLoader 载入)的一部分)的函式库，都会遇到此问题，比方说 JDBC 或 JNDI 就是最好的例子。解决这个问题的方法是透过 Context Class Loader，不过本章不对此问题进行讨论。

如果您对整个类别载入的方式仍有所疑问，请容笔者重新解释一下之前程式

档案:Office.java

```
import java.net.* ;
public class Office
{
  public static void main(String args[]) throws Exception
 {
    URL u = new URL("file:/d:/my/lib/") ;
    URLClassLoader ucl = new URLClassLoader(new URL[]{ u }) ;
    Class c = ucl.loadClass(args[0]) ;
    Assembly asm = (Assembly) c.newInstance() ;
  asm.start() ;

    URL u1 = new URL("file:/d:/my/lib/") ;
    URLClassLoader ucl1 = new URLClassLoader(new URL[]{ u1 }) ;
    Class c1 = ucl1.loadClass(args[0]) ;
    Assembly asm1 = (Assembly) c1.newInstance() ;
  asm1.start() ;

  System.out.println(Office.class.getClassLoader()) ;
  System.out.println(u.getClass().getClassLoader()) ;
  System.out.println(ucl.getClass().getClassLoader()) ;
  System.out.println(c.getClassLoader()) ;
```

```
  System.out.println(asm.getClass().getClassLoader()) ;

  System.out.println(u1.getClass().getClassLoader()) ; 41
  System.out.println(ucl1.getClass().getClassLoader()) ;
  System.out.println(c1.getClassLoader()) ;
  System.out.println(asm1.getClass().getClassLoader()) ;

        System.out.println(Assembly.class.getClassLoader()) ;
 }
}
```

的执行结果。

在这个程式中，整个 Java 虚拟机器共产生五个类别载入器，如下图所示:

```
ExtClassLoader ExtClassLoader
Bootstrp Loader Bootstrp Loader
Parent null
AppClassLoader AppClassLoader
Parent
URLClassLoader URLClassLoader
Parent
URLClassLoader URLClassLoader
Parent
```

整个程序是这样子的: 一开始的时候，我们在命令列输入 java office Word，所以 AppClassLoader 必须负责载入 Office.class ，由于其 Parent(ExtClassLoader)与 Parent 的 Parent(Bootstrap Loader)都无法在其搜寻路径找到 Office.class ，所以最后是由 AppClassLoader 载入 Office.class。我们在 Office.class 之中，需要建立 URL 与 URLClassLoader 的类别实体，预设使用的类别载入器是当时所在的类别本身的类别载入器，也就是利用 ClassLoader.getCallerClassLoader()取得的类别载入器(注意，ClassLoader.getCallerClassLoader()是一个 private 的方法，所以我们无法自行叫用，这是 new 运算子本身隐函的呼叫机制中自行使用的)。接下来，AppClassLoader 仍然会去请求其 Parent(ExtClassLoader)与 Parent 的 Parent(Bootstrap Loader)来载入，其中，Bootstrap Loader 在<JRE 所在目录>\lib\rj.jar 之中找到 URL.class 与 URLClassLoader.class，所以理所当然由 Bootstrap Loader 负责载入。最后，我们请求 URLClassLoader 到相对路径下的 lib\子目录载入 Word.class，载入 Word.class 之前，必须先载入 42 Assembly.class，所以 URLClassLoader 会请求其 Parent(AppClassLoader)、Parent 的 Parent(ExtClassLoader)与 Parent 的 Parent 的 Parent(Bootstrap Loader)来载入，所以最后会由 AppClassLoader 载入 Assembly.class，而由 URLClassLoader 载入 Word.class。然后因为 Assembly.class 的实体(asm 与 asm1)参考到的事 Word.class 的实体(c 与 c1)，所以最后印出结果是 URLClassLoader 载入的，但是如果我们直接用 Assembly.getClassLoader()，就会显示出是 AppClassLoader 载入的。所以输出如下图所示:

但是，如果我们特意把 Word.class 放在 AppClassLoader 找的到的地方(例如与 office.class 放在同一个目录)，

就会产生底下结果: 43

这是因为 URLClassLoader 委派给 AppClassLoader 来载入类别时，AppClassLoader 在自己的搜寻路径下找到了 Word.class，所以输出的时候告诉我们 Word.class 皆是由 AppClassLoader 载入。

类别载入器的功用

从上面的种种测试和说明，我们了解了类别载入器和其载入机制是一个非常复杂的系统，那么为何要设计这么复杂的系统呢? 除了可以达到动态性之外，其实最重要的原因莫过于安全性。我们以下面这张图来说明:

ExtClassLoader ExtClassLoader
Bootstrp Loader Bootstrp Loader
Parent
AppClassLoader AppClassLoader
Parent
URLClassLoader URLClassLoader
Parent
URLClassLoader URLClassLoader
Parent

來自于 www.sun.com 來自于 www.xxx.com

这张图说明了两件事情，第一，假设我们利用 URLClassLoader 到网路上的任何地方下载了其他的类别，URLClassLoader 都不可能下载 AppClassLoader、ExtClassLoader、或者 Bootstrap Loader 可以找到的同名类别(指全名，套件名称+类别名称)，因此，蓄意破坏者根本没有机会植入有问题的程式码于我们的 44
电脑之中(除非蓄意破坏者能潜入您的电脑，置换掉您电脑内的类别档，但是这已经不是 Java 所涉及的安全问题了，而是作业系统本身的安全问题)。第二，类别载入器无法看到其他相同阶层之类别载入器所载入的类别，如上图所示，图中虚线索框起来的部分意指从 www.sun.com 下载程式码的类别载入器所能看到的类别。告诉我们从 www.sun.com 载入的类别，无法看到 www.xxx.com 载入的类别，这除了意味着不同的类别载入器可以载入完全相同的类别之外，也排除了误用或恶意使用别人程式码的机会。

总结

我们花了很长的篇幅，才探讨完 Java 里头类别载入器的特性，不知道您是否对类别载入器有了深入的认识呢? 类别载入器是 Java 平台上最神秘，也是最有趣的一个组件。透过类别载入器，除了可以达成程式的动态性之外，更能够做到无懈可击的安全性(附带一提，Java 的安全架构是由 Sun Microsystems 里头优秀的华裔科学家宫力(Gong Li)博士所设计，这真是华人的骄傲)。市面上许多提到安全性的书籍，或多或少都会提到类别载入器，大多数都会以特别一个章节来解说类别载入器，如果您对安全性的问题有兴趣，您可以参考 O'Reilly 所

出版的 Java Security 2/e 一书:
中文版网址:
http://www.oreilly.com.tw/chinese/java/java_security2.html

原文版网址:
http://www.oreilly.com/catalog/javasec2/

第三章

Java 与 Microsoft Office
如果你相信了他的经验，那么你的极限就会局限在他的经验之中。

简介

记得不久前，在网路上看到一位网友写着:「Java 把程式设计师和底层作业系统隔开，因而让不同平台上的程式可以用同一种方式开发。但是也因为如此，Java 让程式设计师和底层作业系统之间变得毫无关联，使得很多原本用 Visual Basic、Delphi 等原生开发工具很容易做到的事情，在 Java 中却难以做到。」其实，就基本的概念来说，这句话是正确的，也说明了 Java 在某些功能上的确力不从心。可是，这不就是 Java 之所以为 Java 的理由吗?
对实际参与 Java 应用程式开发的工程师来说，往往因为某些特殊因素(通常是客户和老板的要求)，需要完成一些与作业系统结合较紧密的程式，比方说要从 Java 应用程式里头去操作 Windows Registry、或是要让 Java 应用程式可以操作未公开格式的 Microsoft WORD 文件档。这些功能都是标准的 Java 类别函式库没有提供的，所以，除了到网路上搜寻是否有厂商提供类似功能的元件之外，工程师几乎一筹莫展。这个时候，JNI(Java Native Interface)就是最好的万灵丹。
如果要从 Java 应用程式里头去操作 Windows Registry，那么我们只要写个简单的转接器(adapter)，让 Java 叫用那些被包装好、用来操控 Windows Registry 的 Win32 API 即可，如下图所示: 转接器(Java)

动态聯结函式库(C++)

Java 程式(Java)
Win32 API
Windows Registry
JNI

然而，如果要让 Java 应用程式可以操作未公开格式的 Microsoft WORD 文件档，可就没有那么简单了，在无法得知档案格式的情况下，我们该如何解决呢? 于是，问题回到基本面:「既然只有 WORD 才能读取 WORD 文件档，那我
们就请 WORD 帮我们读不就好了吗?」，此时，就必须要动用到 OLE Automation 技术，方法如下图所示:

桥接用介面類別(Java)

动态聯结函式库(C++)

Java 程式(Java)
WORD Container
Component
WORD 文件

JNI
OLE
Automation

本文所要讨论的重点，就是如何在 Java 应用程式之中使用 Microsoft Office 之中所提供的软体元件。本文的程式码纯粹为了便于展示用，没有做过任何最佳化。如果您要让程式码变得更有扩充性，请使用适当的 Design Pattern 即可，程式基本的运作原理是不变的。

本章目的

本章将教大家从无到有制作一个「套表列印模组」 。此模组的功能说明如下:
功能缘起:
在一般公司行号的办公室当中，几乎是以 Microsoft Office 作为文件处理之首，而 Microsoft WORD 更可以制作出图文并茂的文件，这并非使用既有的列印元件(如:Borland C++Builder 提供的 Quick Report)就可以简单达到的事。因此，我们希望先用 Microsoft WORD 制作出图文并茂的文件样版，如下图所示:

然后，我们只要在 Java 程式之中提供一个字串阵列:
String context[] = { "test:恭喜您考取 SCJP" , "company:升阳" , "user:朱仲杰" } ;
当 Java 程式叫用了我们所撰写的套表列印模组之后，样版就会自动将出现『@test』的地方取代成「恭喜你考取 SCJP」 、 『@company』的地方取代成「升
阳」 、 『@user』的地方取代成「朱仲杰」 ，结果应如下图所示:
当然，当您的本文中真的需要使用”@”和”:”时，这样的设计会造成若干冲突，此时可能要动用到跳脱字元(escape character)的设计，请读者自行衡量。
在本文中，笔者将使用 C++制作一个可以呼叫 OLE Automation 的动态联结函式库，并从 Java 应用程式之中利用 JNI 技术呼叫此动态联结函式库，以达到前述我们所期望的功能。

基本技能

要了解本文之中的所有程式码，您必须具备下列几项技能:
1. Java 程式的撰写能力。
2. C++程式的撰写能力。
3. Visual Basic for Application(VBA)程式的撰写能力。
不过有鉴于一般的程式设计师通常只将自己的焦点放在其中一项技能，所以本文并不打算假设阅读本文的读者对这三项技能都非常熟悉。我的一位医师好友邱建勋曾经提醒我，好的老师一定要教学生拿钓竿，让他们自己具备捕鱼的能力，所以本文将会告诉大家，如果以后设计一个类似的功能模组时，要怎么样才能顺利完成。

即使如此，本文所用到的程式码仍有其进入门槛。如果有必要，请多参考本文所提到的所有相关资源。

架构

在整个范例程式之中，最重要的技术莫过于 JNI(Java Native Interface)，这是 Java 程式与 C++程式之所以可以协同运作的重要关键，所以我们必须了解 JNI 究竟是一种什么样的技术。

从上图中，读者可以发现，Java 程式都因为 Java 虚拟机器的关系，和底层完全地隔离开来。可以一旦使用 JNI 之后，Java 虚拟机器会开放一扇门户，和 Java 程式可以突破 Java 虚拟机器的限制，直接呼叫底层作业系统的原生函式库。

但是这么一来，会引发安全性的问题。因为 Java 的安全机制完全由 Java 执行环境之中的 Security Manager 来控制，并配合 Policy File 来设定控制权限。因此，如果要使用 JNI，一定要先考量安全性的议题，否则您的 Java 程式就在不知不觉中为破坏者开启了后门而不自觉。您必须在 Java 层级(也就是执行权限尚未穿越 Java 虚拟机器前)就完成安全性上的限制。关于此问题，请参考 Java Security 相关书籍。

接下来我们就要开始整个功能模组的建构程序。

功能模组建构程序一: 设计桥接用的介面类别

首先，我们必须制作一个可以供 Java 程式呼叫用的 Java 类别，这个类别的主要功能是作为桥接之用，除了作为 Java 与 C++之间的介面之外，也用来

帮助 Java 执行环境载入所需要的动态联结函式库，如下图所示: 桥接用介面類別(Java)

动态聯结函式库(C++)

Java 程式(Java)
回传值
载入
方法呼叫
方法呼叫 回传值

虚线部份表示:『来自于 Java 应用程式的方法呼叫，不一定需要有传回值回传(void) 』 。
桥接介面类别的原始码如下:
JavatoWord.java

```
/********************************************** ************
Java2Word.java
JNI 桥接介面物件
2002 Jan by 王森
moli@pchome.com.tw
********************************************** ************/
package com.sun.edu ;
public class JavatoWord {
 /*
  原生函式
  context ==> 所有文字内容所构成的阵列
        count = context.length
  */
  public native void NativeToWord(String[] context,int count) ;
 /*
    在使用原生函式之前，再次确认参数的正确性，尽量让错误的处理在 Java 之中完成。
    */
  public boolean ToWord(String[] context) {
  try
  {
   NativeToWord(context,context.length) ;
   return true ;
  }catch(Exception e)
  {
   return false ;
  }
 }
```

```
 static
 {
  //我们把实做转换至 WORD 的函式的模组做成 DLL(动态连结函式库)档,
  //取名叫 Transfer.dll,所以在这里要载入此 DLL
  System.loadLibrary("Transfer") ;
 }
}
```

此类别有两个特殊的地方:

1. NativeToWord 方法前面加上了 native 修饰字。
此修饰字的意思即是告诉编译器:"这个函式的实际内容并非以 Java 所撰写，而是以其他语言所撰写"。

2. 使用了静态初始化区块。
当 JavatoWord.class 被类别载入器(class loader)载入时，静态初始化区块就会被呼叫，并载入名为 Transfer.dll 的动态联结函式库。请注意，静态初始化区块在整个 Java 程式的生命周期之中，只有在被类别载入器载入的时候才被呼叫一次，往后不会再被叫用。

请注意，JavatoWord 这个类别属于 com.sun.edu 这个 package，所以一定要放置在相对路径 com\sun\edu\底下，否则编译的时候虽然不会出错，执行的时候会出现执行时期例外。相关内容请参考本书后面讨论 package 机制的章节，这里不再赘述。

功能模组建构程序二: 撰写使用桥接介面类别的 Java 程式

桥接用介面類別(Java)

Java 程式(Java)
方法呼叫

test.java

```
/********************************************** ********
test.java
JNI 测试物件
2002 Jan by 王森
moli@pchome.com.tw
********************************************** ********/
import com.sun.edu.* ;
class test
{

 //主程式开始
  public static void main(String args[])
 {
  //取得 JNI 桥接介面物件
    JavatoWord my = new JavatoWord() ;
```

```
  String context[] =
{ "test:恭喜您考取 SCJP" , "company:升阳" , "user:朱仲杰" } ;

  //开始进行转换工作
  if(my.ToWord(context))
  { System.out.println("呼叫成功") ;
  }else
  {
   System.out.println("传入阵列有误,可能是元素数量不正确") ;
  }
  }//end of main
}//end of class
```

■功能模组建构程序三: 产生编译动态联结函式库时所需要的C/C++引入档

请使用指令:

```
javah -classpath . com.sun.edu.JavatoWord
```

就可以在目录之中发现 javah.exe 帮我们在目录下产生了一个叫做 com_sun_edu_JavatoWord.h 的 C++引入档，其内容如下:

com_sun_edu_JavatoWord.h

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class com_sun_edu_JavatoWord */
#ifndef _Included_com_sun_edu_JavatoWord
 #define _Included_com_sun_edu_JavatoWord
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class: com_sun_edu_JavatoWord
 * Method: NativeToWord
 * Signature: ([Ljava/lang/String;I)V
 */
JNIEXPORT void JNICALL Java_com_sun_edu_JavatoWord_NativeToWord
    (JNIEnv *, jobject, jobjectArray, jint);

#ifdef __cplusplus
}
#endif
#endif
```

■功能模组建构程序四: 建立套表列印模组的基本架构

请先开启您的 Borland C++Builder，并开启一个新专案，如下图:

请在选择 DLL Wizard 之后，按下 OK 钮，接着会出现底下画面。

请直接按下 OK 钮即可。精灵会自动帮我们建构专案，并产生对应的动态联结函
式库程式码主干。
接着，请按下 Save All 钮，如下图:

请先建立一个目录，名为 Transfer，如下图:
进入 Transfer 目录，将主程式码档名储存为 Transfer.cpp，如下图:

同时也把专案档档名改储存为 Transfer.bpr，如下图:
接着，请选择 Project | Options 选项，我们要设定一些环境变数:

进入 Project Options 对话方块之后，请先选择 Directories and Conditionals 次页，将两个使用 JNI 时一定要使用的引入档之所在目录加进来，他们分别是 <jdk 安装目录>\include 与<jdk 安装目录>\include\win32，如下图:

(注意: 笔者将 JDK 安装在 d:\jdk1.3.1，请各位读者依自己电脑中 JDK 所在位置作调整。)

接着，请将 com_sun_edu_JavatoWord.h 放到专案所在的目录下，所有与该专案相关的档案如下图所示:

完成上述步骤之后，请将 Transfer.cpp 修改成底下的样子:
Transfer.cpp

```
#include <vcl.h>
#include <windows.h>
#include "com_sun_edu_JavatoWord.h"
#pragma hdrstop

#pragma argsused
int WINAPI DllEntryPoint(HINSTANCE hinst, unsigned long reason, void* lpReserved)
{
        return 1;
}

JNIEXPORT void JNICALL Java_com_sun_edu_JavatoWord_NativeToWord
  (JNIEnv *env, jobject obj, jobjectArray context, jint count)
{
        //读入样板档
```

```
        TOpenDialog* templatefile = new TOpenDialog(NULL) ;
        templatefile->Title = "请选择样板档" ;
        templatefile->Filter = "Word files (*.doc)|*.doc" ;
        AnsiString filename ;
        if(templatefile->Execute())
        {
              filename = templatefile->FileName ;
        } //检查样板档是否存在
        if(!FileExists(filename))
        {
              ShowMessage("您所输入的档案并不存在,请重新输入") ;
              delete templatefile ;
              return ;
        }
        delete templatefile ;
        ShowMessage("您选择的档名为" + filename) ;
}
```

完成后，请使用选单上的 Project | Make Transfer 建立动态联结函式库(.dll 档)。然后到 DOS 视窗，执行指令:

```
java –cp . –Djava.library.path=.\Transfer test
```

如果执行正常，萤幕输出应该如下图:

选择了某个档案之后，按下开启之后，萤幕上会出现:

DOS 视窗的输出为:

接下来，我们要测试从 Java Î C++时的参数传递是否正确，因此我们把程式码改成底下的样子: Transfer.cpp

```
#include <vcl.h>
#include <windows.h>
#include <iostream.h>
#include "com_sun_edu_JavatoWord.h"
#pragma hdrstop

#pragma argsused
int WINAPI DllEntryPoint(HINSTANCE hinst, unsigned long reason, void* lpReserved)
{
      return 1;
}

JNIEXPORT void JNICALL Java_com_sun_edu_JavatoWord_NativeToWord
   (JNIEnv *env, jobject obj, jobjectArray context, jint count)
```

```
{
//读入样板档
        TOpenDialog* templatefile = new TOpenDialog(NULL) ;
        templatefile->Title = "请选择样板档" ;
        templatefile->Filter = "Word files (*.doc)|*.doc" ;
        AnsiString filename ;
        if(templatefile->Execute())
        {
                filename = templatefile->FileName ;
        }
        //检查样板档是否存在
        if(!FileExists(filename))
        {
                ShowMessage("您所输入的档案并不存在,请重新输入") ;
                delete templatefile ;
                return ;
        }
        delete templatefile ;
        ShowMessage("您选择的档名为" + filename) ;

         cout << "接收到" << count << "  个字串" << endl;
                for(int i = 0 ; i < count ; i++)
        { //取出  context
                        jstring jsdata =
(jstring)(*env).GetObjectArrayElement(context,i) ;
                        wchar_t* csdata =
(wchar_t*)(*env).GetStringChars(jsdata,NULL) ;
                        int size = (*env).GetStringLength(jsdata) ;
                        WideString wtmp = WideString(csdata,size) ;
                        ShowMessage(wtmp) ;
        }
}
```

完成后，请使用选单上的 Project | Make Transfer 重新建立动态联结函式库(.dll 档)。然后到 DOS 视窗，执行指令:

```
 java –cp . –Djava.library.path=.\Transfer test
```

如果执行正常，除了输出选择的档案名称之外，萤幕还会额外输出:
等其他两个视窗(因为我们传递了三个参数)。
DOS 视窗的输出如下:

这么一来表示从 Java Î C++之间的参数传递没有任何问题。

■ 功能模组建构程序五: Visual Basic for Application

从 Java Î C++之间的参数传递没有问题了，接着我们要使用 C++来操控 VBA(Visual Basic for Application，也就是一般 Microsoft Office 术语之中常

说的”巨集”指令)，所使用的技术就是 OLE Automation。可是，VBA 指令怎么下呢？笔者所使用的方法步骤如下:
1.制作样版档
首先，先撰写我们所需要的 WORD 文件样版档如下:
2.录制巨集
请选择 Microsoft WORD 选单上的『工具|巨集|录制新巨集』，如下图所示:

在录制巨集对话方块中填入 test 为巨集名称，如下图所示:
按下确定钮之后，我们在 Microsoft WORD 里面所作的每一件事情，都会被录制成巨集，也就是 VBA 的程式码。
现在，请按下 Microsoft WORD 选单上的『编辑|取代』

在寻找/取代对话方块之中填入资料，如下图所示:
按下全部取代钮之后，系统会自动完成搜寻/取代的工作，并显示底下视窗:

同时，我们的样版档变成:

最后，请选择 Microsoft WORD 选单上的『工具|巨集|停止录制』，如下图所示:
就可以停止巨集的录制。
3.观看巨集
请选择 Microsoft WORD 选单上的『工具|巨集|巨集』，如下图所示:

在巨集对话方块之中，选择名为 test 的巨集，然后按下编辑，如下图所示:
系统会秀出刚刚我们所作的动作相对应的 VBA 程式码:
巨集 test 的 VBA 程式码

```
Sub test()
'
' test 巨集
' 巨集录制于 2001/12/28，录制者 moli
'
    Selection.Find.ClearFormatting
    Selection.Find.Replacement.ClearFormatting
    With Selection.Find
        .Text = "@company"
        .Replacement.Text = "升阳"
        .Forward = True
        .Wrap = wdFindContinue
        .Format = False
        .MatchCase = False
        .MatchWholeWord = False
        .MatchByte = True
        .MatchWildcards = False
```

```
        .MatchSoundsLike = False
        .MatchAllWordForms = False
    End With
    Selection.Find.Execute Replace:=wdReplaceAll
End Sub
```

同样地，如果是一个开启某个 WORD 文字档的 VBA 程式码，经过录制以后内容如下:

巨集 test1 的 VBA 程式码

```
Sub test1()
'
' test1 巨集
' 巨集录制于 2001/12/28，录制者 moli
'
    ChangeFileOpenDirectory "E:\Java2Word\"
    Documents.Open FileName:="test.doc", ConfirmConversions:=False,
ReadOnly:= _
        False, AddToRecentFiles:=False, PasswordDocument:="",
PasswordTemplate:= _
        "", Revert:=False, WritePasswordDocument:="",
WritePasswordTemplate:="", _
        Format:=wdOpenFormatAuto
End Sub
```

有了上述两组巨集指令之后，我们就可以用 C++撰写完整的套表列印功能模组了。

功能模组建构程序六: 完成套表列印功能模组

好了，我们已经完成了所有的前置工作，我们可以动手撰写套表列印功能模组的程式码。请启动您的 C++ Builder，并开启 Transfer.bpr 这个专案档。我们将 Transfer.cpp 的内容依以下步骤修改:

步骤一: 加入引入档

请先加入几个使用 COM 技术(OLE Automation 的基础架构)时必须用到的引入档，他们分别是:OleServer.hpp、utilcls.h、comobj.hpp。另外，要使用 Microsoft WORD 2000 所提供的软体元件，还必须要额外引入 vcl\word_2k.h(在安装 C++Builder 的时候，安装程式会根据您电脑上的 Office 版本产生对应的引入档，所以如果您编译 Transfer.cpp 时，编译器说找不到此档，那么请重新检视您电脑上 C++Builder 的安装程序，笔者已测试过 Professional 版与 Enterprise 版，一切正常)。修改后的 Transfer.cpp 如下:

Transfer.cpp

```
#include <vcl.h>
#include <OleServer.hpp>
#include <windows.h>
#include "com_sun_edu_JavatoWord.h"
#include <utilcls.h>
#include <comobj.hpp>
```

```
#include <iostream.h>
#include <vcl\word_2k.h> #pragma hdrstop

#pragma argsused
int WINAPI DllEntryPoint(HINSTANCE hinst, unsigned long reason, void*
lpReserved)
{
      return 1;
}

JNIEXPORT void JNICALL Java_com_sun_edu_JavatoWord_NativeToWord
   (JNIEnv *env, jobject obj, jobjectArray context, jint count)
{
      ...略...
}
```

步骤二: 加入使用 COM 技术时必须叫用的 API

根据规定，使用 COM 技术之前，必须呼叫 CoInitialize(NULL)，而离开之前，需要叫用 CoUninitialize()。因此程式码需要修改成底下的样子:

Transfer.cpp

```
...略...

JNIEXPORT void JNICALL Java_com_sun_edu_JavatoWord_NativeToWord
   (JNIEnv *env, jobject obj, jobjectArray context, jint count)
{
      //读入样板档
      TOpenDialog* templatefile = new TOpenDialog(NULL) ;
      templatefile->Title = "请选择样板档" ;
      templatefile->Filter = "Word files (*.doc)|*.doc" ;
      AnsiString filename ;
      if(templatefile->Execute())
      {
            filename = templatefile->FileName ;
      }
      //检查样板档是否存在
      if(!FileExists(filename))
      {
            ShowMessage("您所输入的档案并不存在,请重新输入") ; delete
templatefile ;
            return ;
      }
      delete templatefile ;
```

```
        CoInitialize(NULL) ;
        try
        {

        }catch(Exception& e)
        {
              ShowMessage(e.Message) ;
              CoUninitialize() ;
              return ;
        }
        CoUninitialize() ;
}
```

步骤三: 使用 WORD 来开启 WORD 文字档

根据我们所取得的 test1 巨集内容，我们可以得知要如何使用 WORD 来开启 WORD 文字档:

从巨集内容可以得知，需要使用 Documents 物件的 Open 功能来开启，因此 Transfer.cpp 需要修改如下:

Transfer.cpp

...略...

```
JNIEXPORT void JNICALL Java_com_sun_edu_JavatoWord_NativeToWord
    (JNIEnv *env, jobject obj, jobjectArray context, jint count)
{
        //读入样板档
        TOpenDialog* templatefile = new TOpenDialog(NULL) ;
templatefile->Title = "请选择样板档" ;
        templatefile->Filter = "Word files (*.doc)|*.doc" ;
        AnsiString filename ;
        if(templatefile->Execute())
        {
              filename = templatefile->FileName ;
        }
        //检查样板档是否存在
        if(!FileExists(filename))
        {
              ShowMessage("您所输入的档案并不存在,请重新输入") ;
              delete templatefile ;
              return ;
        }
        delete templatefile ;
```

```
CoInitialize(NULL) ;
try
{
     Variant Wordobj ;
    Variant Docsobj ;
    //产生 Word 物件
   Wordobj = CreateOleObject("Word.Application") ;
   cout << "建立 Word 物件成功" <<endl ;
   Wordobj.OlePropertySet("Visible",true) ;

   //开启 user 所指定的样版档
   Docsobj = Wordobj.OlePropertyGet("Documents") ;
   Docsobj.OleFunction("Open",filename) ;
}catch(Exception& e)
{
   ShowMessage(e.Message) ;
   CoUninitialize() ;
   return ;
}
CoUninitialize() ;
```

} 修改完成后，请使用选单上的 Project | Make Transfer 重新建立动态联结函式库(.dll 档)。然后到 DOS 视窗，执行指令:

```
java –cp . –Djava.library.path=.\Transfer test
```

如果执行正常，您将发现您在选择样版对话方块之中所指定的档名，已经由 WORD 帮我们开启。

在我们的程式之中，首先建立了一个名为 Word.Application 的物件，这个物件代表的是整个 Microsoft WORD，所以要在 C/C++之中利用 OLE Automation 来操控 WORD，第一个步骤一定要先产生此物件。同理，如果您要操控 Microsoft EXCEL，您也必须先产生 Excel.Application 物件。

有了 Word.Application 物件之后，由于预设的情况下 WORD 是隐藏的，我们必须让他显示在萤幕上，所以我们利用 OlePropertySet 函式，将 Application 物件(底下我们直接以 Application 物件来称呼 Word.Application)的 Visible 属性设为 true。有关 Visible 属性，WORD 内附的 Visual Basic 编辑器辅助说明如下:

(上图取自于 Microsoft WORD 内附 Visual Basic 编辑器的辅助说明， 版权属于 Microsoft 所有)

接下来，我们从 test1 巨集之中得知，我们必须使用 Documents 物件的 Open 方法来开启档案。于是，我们查询了 Microsoft WORD 中的物件关系图，发现 Documents 物件属于 Application 物件的直属物件，如下图所示

(上图取自于 Microsoft WORD 内附 Visual Basic 编辑器的辅助说明， 版权属于 Microsoft 所有)

所以我们可以直接利用 Application 物件来取得 Documents 物件，然后叫用其 Open 方法，并给予其欲开启的档案路径和名称即可。有关 Open 方法的描述，请参阅 Microsoft WORD 内附 Visual Basic 编辑器的辅助说明。
步骤四: 使用取代功能
根据我们所取得的 test 巨集内容，我们可以得知要如何使用 WORD 内建的文字取代功能:
根据物件关系图，我们可以知道 Selection 物件属于 Application 物件的直属物件，如下图所示:
(上图取自于 Microsoft WORD 内附 Visual Basic 编辑器的辅助说明， 版权属于 Microsoft 所有)
如果您使用 Microsoft WORD 内附 Visual Basic 编辑器的辅助说明，只要按下上图 Selection 右边的红色箭头，就可以依序找到与 Selection 物件有相依关系的 Find 物件您可以查询到属于 Find 物件的 ClearFormatting 方法与 Execute 方法。再按下 Find 物件右边的红色箭头，就可以依序找到与 Find 物件有相依关系的 Replacement 物件，最后就可以找到其 ClearFormatting 方法。因此，最后的程式码如下:
Transfer.cpp

...略...

```
JNIEXPORT void JNICALL Java_com_sun_edu_JavatoWord_NativeToWord
   (JNIEnv *env, jobject obj, jobjectArray context, jint count)
{
      //读入样板档
      TOpenDialog* templatefile = new TOpenDialog(NULL) ;
      templatefile->Title = "请选择样板档" ;
      templatefile->Filter = "Word files (*.doc)|*.doc" ;
      AnsiString filename ;
      if(templatefile->Execute())
      {
            filename = templatefile->FileName ;
      }
      //检查样板档是否存在
      if(!FileExists(filename))
      {
            ShowMessage("您所输入的档案并不存在,请重新输入") ;
            delete templatefile ;
            return ;
      }
      delete templatefile ;
      CoInitialize(NULL) ;
      try
      {
            Variant Wordobj ;
```

```
            Variant Docsobj ;
            //产生 Word 物件
            Wordobj = CreateOleObject("Word.Application") ;
            cout << "建立 Word 物件成功" <<endl ;
            Wordobj.OlePropertySet("Visible",true) ;
            //开启 user 所指定的样版档
            Docsobj = Wordobj.OlePropertyGet("Documents") ;
            Docsobj.OleFunction("Open",filename) ;

            Variant ActDoc ;
            Variant Selection ;
            Variant Find ;
            Variant Replacement ;
            //取得目前作用的 document
            ActDoc = Wordobj.OlePropertyGet("ActiveDocument") ;
            //取得 Selection.Find.Replacement 物件
            Selection = Wordobj.OlePropertyGet("Selection") ;
            Find = Selection.OlePropertyGet("Find") ;
            Replacement = Find.OlePropertyGet("Replacement") ;
            //执行所有来自 Java 程式中,所请求的取代动作
            for(int i = 0 ; i < count ; i++)
            {
                  //取出 context 的内容
                  jstring jsdata =
(jstring)(*env).GetObjectArrayElement(context,i) ;
                  wchar_t* csdata =
(wchar_t*)(*env).GetStringChars(jsdata,NULL) ;
                  int size = (*env).GetStringLength(jsdata) ;
                  WideString wtmp = WideString(csdata,size) ;
                  int start = wtmp.Pos(":") ;
                  WideString text = wtmp.SubString(1,start-1) ;
                  text = "@" + text ;
                  WideString rtext =
wtmp.SubString(start+1,wtmp.Length() - start) ;

                  //开始取代动作
                  Find.OleFunction("ClearFormatting") ;
                  Replacement.OleFunction("ClearFormatting") ;
Find.OleFunction("Execute",text,true,false,false,
false,false,true,wdFindContinue,false,rtext,wdReplaceAll) ;
            }
      }catch(Exception& e)
      { ShowMessage(e.Message) ;
            CoUninitialize() ;
```

```
            return ;
        }
        CoUninitialize() ;
}
```

修改完成后，请使用选单上的 Project | Make Transfer 重新建立动态联结函式库(.dll 档)。然后到 DOS 视窗，执行指令:

```
java –cp . –Djava.library.path=.\Transfer test
```

如果执行正常，您将发现此模组已经完成了我们所期望的功能。

总结

作为一个占有率最高的的办公室软体套件，Microsoft Office 在功能上的确有其独到的地方。尤其是，Microsoft Office 里头有很多非常优秀、且可以重复使用的软体元件，几乎都被我们忽略了。其实，多多善用 Microsoft Office 的软体元件所提供的强大功能，可以减少程式设计师沉重的负担，例如 Excel 的强大计算功能与图表功能就是一个很好的例子，当然，使用了 JNI 之后，您的 Java 程式就失去了跨平台的特性。这是诸位读者必须要取舍的地方。

希望各位阅读过本章之后，将来有必要时，可以善加利用 Microsoft Office 既有的软体元件(其实不只只有 Microsoft Office，其他如 Microsoft Visio、Microsoft Internet Explorer、还有 AutoCAD，几乎只要可以用 VBA 来操控的软体，都有此功能)来加强自己开发的软体。

Java Native Interface 参考资源

利用 Java 配合 BCB 4.0 制作 CPU 特征侦测器
作者 : 王森
(网路版本: http://www.javatwo.net/experts/moli/publish/article/TC/BCB_CPU_Detector.htm)
游戏设计大师 NO.8 1999 年 11~12 月号

Essential Jni : Java Native Interface
作者: Rob Gordon, Robert Gordon, Alan McClellan
http://www.amazon.com/exec/obidos/ASIN/0136798950/o/qid=990362161/sr=8-1/ref=aps_sr_b_1_1/002-7278315-6856028

The Java Native Interface : Programmer's Guide and Specification
作者 : Sheng Liang

http://www.amazon.com/exec/obidos/ASIN/0201325772/o/qid=990362161/sr=8-2/ref=aps_sr_b_1_2/002-7278315-6856028

第四章

用 Visual Studio.net 来
操控 Java 虚拟机器
愚公移山的故事，是中国人自扫门前雪的最佳典范。
天神把山移到别人家门口，如果上天不买这家人的帐，
恐怕这个倒楣者和他的可怜后代都要一辈子在那里移山了。
就算上天又感动了，则又是另外一家人倒楣的故事。

前言

在前一章中，笔者带大家用 Borland C++ Builder ，配合 JNI 技术，使得 Java 程式可以存取 Microsoft Office 所提供的软体元件资源。其实熟悉 C++和 Windows Programming 的朋友一定知道，这样的工作并非只有 Borland C++ Builder 才能做到，用 Microsoft Visual C++一样可以完成相同的事情。
因此在本章中，我们将利用 Microsoft 所推出的 Visual Studio.NET 正式版，一样配合 JNI 技术，撰写可以操控 Java 虚拟机器，并且能够使用 Java 类别函式库的 C++程式码。
为何会有这个主题的出现呢? 这是因为只要您使用上一章的技巧，就可以善用 Java Î JNI Î C++ 的方式来扩充 Java 应用程式的能力。但是反过来呢? 有些已经用 Java 撰写好的类别函式库，再用 C++重新撰写划不划算? 如果可以在 C++中重复使用 Java 程式码，也是一件重复使用软体元件的典范。
.NET 和 Java 的战争似乎有水火不容的趋势，而 Microsoft 于在 2002 年 1 月发表 Visual Studio.NET 正式版，似乎正式地点燃了战火。但是在本章中，我们利用 Visual Studio.NET 来开发 C++程式，并借此操控 Java 2 SDK/JRE 1.3.x 版或 Java 2 SDK/JRE 1.4.x 版内含的 Java 虚拟机器(Java Virtual Machine)，当我们控制了 Java 虚拟机器，我们就能够在 C++之中重复使用已经用 Java 撰写完成的类别函式库。有了这样的经验，您将发现，两家子在某种程度上还是可以相处的非常愉快。
当然，要让.NET 和 Java 融合在一起运作，至少要让 C#撰写的程式码可以存取 Java 所撰写的程式码(managed code 存取 byte code)，或是让 Java 撰写的程式码可以使用 C#所撰写的程式码(byte code 存取 managed code)才算真正的融合在一起。这两个议题在本书中完全没有涉猎，笔者会在本书的下一版之中告诉各位该如何完成这两件事情。所以您几乎可以认定，本章的标题只是个幌子，但希望这是个两大阵营大和解时代来临的第一步。

简介

想像有一天，您自己用 Java 撰写了底下这样一个简单的工具函式库:
档案:MyToolkit.java
您会不会希望可以从 C++程式码中重复使用这段程式码呢? 本文的目的就就是教您如何利用 C++来操控 Java 虚拟机器，再利用 Java 虚拟机器来载入我们所需要的类别函式库，然后使用我们想重复利用的功能函式。其架构如下图所示:
Java Java 虚拟机器虚拟机器((jvm.dll) jvm.dll)

類別函式库(.class)

C/C++程式(.exe)

2.载入(JNI)
1.唤起 4.回传值(JNI)
3.方法呼叫(JNI)
整体程序概观 整体程序概观

如上图所示，整个程序的首要之先，就是先从用 C/C++所撰写的程式码之中，唤起 Java 虚拟机器。由于 Java 虚拟机器在 Windows 平台上是个动态联结函式库(.dll,即 dynamic linking library)，所以唤起程序的第一件事情，就是执行档把 jvm.dll 这个动态联结函式库载入记忆体，并和我们的执行档执行时所属的行程(process)连接起来(attach)，如此一来我们才可以利用 jvm.dll 所开放出来的函式来操控 Java 虚拟机器。这个复杂的程序有两种方法可以做到，第一种为 explicit 式，也就是在程式执行时期使用 Win32 API 里头的 LoadLibrary()与 GetProcAddress()来做，第二种为 implicit 式，也就是在编译时期就依靠标头档(.h)与符号表档(.lib)来解决外部参考的问题。由于第一种比较有弹性，但是

```
1 public class MyToolkit
2 {
3 public static void function1(String param)
4 {
5 System.out.println("You input:" + param) ;
6 }
7 }
```

8 比较不易除错，所以本文采用第二种方式。因此，只要在编译程式的时候让编译
器可以找到标头档(.h)与符号表档(.lib)两种档案即可。
当动态联结函式库接驳上我们的行程之后，我们就可以利用 JNI_CreateJavaVM()以在记忆体中建立一个 Java 虚拟机器的实体，然后我们就可以操控它来载入我们希望使用的类别函式库。当我们不需要这个 Java 虚拟
机器的时候，我们也要利用 JNI_DestroyJavaVM()来清除它。整个唤起 Java 虚拟机器的程序如下图所示:
Java Java 虚拟机器虚拟机器((jvm.dll) jvm.dll)
C/C++程式(.exe)
1.载入 jvm.dll 于

执行档所在

行程的记忆体之中

Java Java 虚拟机器唤起程序虚拟机器唤起程序
2.叫用
JNI_CreateJavaVM
3.叫用
JNI_DestroyJavaVM()

了解了整个架构之后，接下来我们就要开始进行程式码的撰写了。本文内容

将采 step by step 的步骤，让不熟悉工具操作的朋友也可以完成整个系统。

用 Visual Studio.NET 撰写主程式
用 Visual Studio.NET 撰写主程式的程序如下图所示: 开启空白解决方案(Solution)
用用 Visual Studio.NET Visual Studio.NET 撰写主程式之程序撰写主程式之程序
开启空白专案(Project)
加入 C++程式码档(Item)
撰写程式码

如图所示，首先我们必须先开启一个空白的 Solution。请选择主选单的 File / New 方/ Blank Solution，如下图所示:

接着，在 Name 的地方打入解决方案(Solution)的名字(本文取名为 JavaSolution)，并在 Location 处填入您希望您的解决方案所储存的位置，如下图所示:
完成之后按下 OK 即可。
有了 Solution，我们才能加入专案(Project)。请开启 Solution Explorer，在 Solution 的上方按下滑鼠右键以叫出内容选单，接着选择 Add / New Project，如下图所示。

在 Add New Project 对话方块中，请先在 Project Types 区点选 Visual C++ Projects，然后到 Templates 区选择 Win32 Project，最后请在 Name 处填入 Project 名称(本文使用 JNIProject)即可，如下图所示:
完成之后，请按下 OK 钮。
接下来，萤幕上会出现 Win32 应用程式精灵(Win32 Application Wizard)，请先选择左方的 Application Settings，然后在 Application type 处选择 Console application，并勾选 Additional options 中的 Empty project，如下图所示:

这是因为我们不希望 Visual Studio.NET 帮我们产生一堆乱七八糟的程式码，所以才设定成产生空白专案。
有了空白解决方案和空白专案之后，再来必须产生 C++程式码档(.cpp)，请开启 Solution Explorer，在 Project 的上方按下滑鼠右键以叫出内容选单，点选 Add / Add New Item，如下图所示:

出现 Add New Item 对话方块之后，请在右边的 Templates 处选择 C++ File(.cpp)，然后在 Name 处输入档名(本文使用 Main.cpp)，最后按下 OK 钮，就完成了 Visual Studio.NET 的专案建构程序，如下图所示。

请双击 Solution Explorer 中的 Main.cpp，然后撰写程式如下:

档案:Main.cpp
```
#include<iostream>
```

```
#include<jni.h>

using namespace std ;

int main()
{
 JavaVM* jvm ;
  JNIEnv* env ;
  JavaVMOption options[1] ;
 JavaVMInitArgs vmargs ;
  long status ;

  options[0].optionString = "-Djava.class.path=.";

  vmargs.version = JNI_VERSION_1_2;
  vmargs.options = options;
  vmargs.nOptions = 1;

  status = JNI_CreateJavaVM(&jvm,(void**)&env,&vmargs) ;
  if(status != JNI_OK)
 {
  cout << "Java 虚拟机器建立失败/不知名的错误" << endl ;
  cout << "错误代码  : " << status ;
   return 1 ;
 }
  cout << "Java 虚拟机器建立成功" ;

 jvm->DestroyJavaVM();
  return 0 ;

}
```

完成之后，请按下 Visual Studio.NET 选单中的 Build / Build Solution 来建造整个解决方案。一开始您会先遇到编译找不到标头档 jni.h 的错误讯息。要解决这个问题，请选择 Visual Studio.NET 选单中的 Tools / Options 以叫出 Options 对话方块，如下图所示:

开启 Options 对话方块之后，请先选择左方的 Projects 之下的 VC++ Directories。然后在 Show directories for 的地方点选 Include files，然后加入 <jdk 安装目录>\include 与<jdk 安装目录>\include\win32 这两个目录，由于笔者的 Java 2 SDK 1.3.1 装在 d:\jdk1.3.1 这个目录下，所以必须指到 d:\jdk1.3.1\include 以及 d:\jdk1.3.1\include\win32 (注意:jni.h 中参考到位于<jdk 安装目录>\include\win32 底下的 jni_md.h，所以必须加入<jdk 安装目录>\include\win32，否则会产生找不到 jni_md.h 的错误讯息)，如下图所示:

完成上述事项之后，请按下 OK 钮，然后重新建造整个解决方案。
由于我们使用使用动态联结函式库的方式是使用 implicit 的方式，因此当编译器完成程式的编译程序，接着要进行连结程序(linking)的时候，就会产生错误讯息，如下图所示:

为了解决这个问题，我们必须还要让联结器(linker)可以找到.lib 档才行。请选择 Visual Studio.NET 选单中的 Tools / Options 以叫出 Options 对话方块。开启 Options 对话方块之后，请先选择左方的 Projects 之下的 VC++ Directories。然后在 Show directories for 的地方点选 Library files，再加入<jdk 安装目录>\lib 这个子目录，由于笔者的 Java 2 SDK 1.3.1 装在 d:\jdk1.3.1 这个目录下，所以必须指到 d:\jdk1.3.1\lib 才行。
但是光是设定这里，只能告诉连结器去哪里找 lib，这是不够的，因为连结器还需要知道使用哪一个.lib 档来解决外部参考才行。所以请重新开启 Solution Provider，在 Project 的上方按下滑鼠右键以叫出内容选单，点选 Properties。叫出 JNIProject Property Pages 对话方块之后，请先点选左方 Linker 底下的 Input，然后在右边 Additional Dependences 的地方加入 jvm.lib，如下图所示:

完成上述事项之后，请按下 OK 钮，然后重新建造整个解决方案。如此就可以顺利地造出可执行档。
注意: 上述加入引入档(.h)和符号表档(.lib)参考路径的设定，其实也可以在 JNIProject
Property Pages 对话方块中的选项来设定。
加入引入档参考路径:
在 C/C++底下的 General 中的 Additional Include Directories 里。

加入符号表档参考路径:
在 Linker 底下的 General 中的 Additional Library Directories 里头:

这种设定方式和从 Visual Studio.NET 系统选单 Tools / Options 叫出 Options 对话方块来设定的差别在于，使用 Tools / Options 叫出 Options 对话方块来设定会影响往后所有的专案，而在 JNIProject Property Pages 的设定只会影响到目前这个专案(即:JNI Project)。

无法唤起 Java 虚拟机器

执行档建造完成之后，您一定会急着想要执行看看，一执行，就会产生错误讯息如下:

这是因为使用 implicit 的方式来连接动态联结函式库的程式，都会在一开始执行时找出动态联结函式库。前面我们说过，Java 虚拟机器被放置在 jvm.dll 里头，可是我们的执行档找遍了所有的设定路径(环境变数 PATH 的设定)，就是找不到 jvm.dll，因此产生了上述的错误讯息。那么您不禁要问:jvm.dll 到底藏在哪里?
如果您是一位经验丰富的 Windows Programmer，您一定会使用

Windows 的搜寻功能找寻 jvm.dll，您将可以在您的电脑中发现好几个 jvm.dll (随个人机器的不同而不同，但是如果您安装的 Java 2 SDK 1.3.1，则应该至少会发现 4 个左右，但是如果您安装的 Java 2 SDK 1.4.0 RC，则应该至少会发现 3 个左右，)。
此时问题来了，我们该使用哪一个 jvm.dll 呢? 每个大小都不同，也放在不同目录底下。如果您想用暴力法一个一个来 try and error，我们把每个 jvm.dll 依序拷贝到与我们的执行档同一个目录下，执行之后，虽然不会出现找不到 jvm.dll 的错误讯息了，可是您的命令列模式下永远出现底下错误讯息:
这下可好了，连 Java 虚拟机器都无法唤起，更不要说重复使用 Java 的类别函式库了。这个问题该怎么解决呢?
顺利唤起 Java 虚拟机器
请您回头复习我们在第一章所提到的概念。在第一章中，我们可以知道 java.exe 用何种方式找到 JRE，然后再选择所使用的 Java 虚拟机器(jvm.dll)。其实，jvm.dll 无法单独存在，当 jvm.dll 启动之后，会使用 explicit 的方式(即使用 Win32 API 之中的 LoadLibrary()与 GetProcAddress())来载入辅助用的动态联结函式库，而这些辅助用的动态联结函式库都必须位于 jvm.dll 所在目录的父目录之中。这也是为什么之前单单把 jvm.dll 拷贝到我们的执行档所在目录却依然发生错误的原因。那么，您是不是开始想着:”除了把 jvm.dll 拷贝到我们执行档的所在目录之外，也把那些位于<JRE 所在目录>\bin 底下的辅助用动态联结函式库都拷贝到执行档所在目录的父目录”呢? 这样做虽然可行，可是实在太麻烦!! 最简单的方式，就是让环境变数 PATH 指向 JRE 所在目录底下的 jvm.dll，
这样一来就不用如此大费周章。
假设我们要使用 Hotspot Client Virtual Machine，我们只要把 PATH 设定中加入<JRE 所在路径>\bin\hotspot 即可，笔者想用 JDK 底下的那组 JRE，而笔者的 JDK 又灌在 d:\jdk1.3.1\底下，所以要执行指令:
path=%path%;d:\jdk1.3.1\jre\bin\hotspot

注意:
如果您拷贝了 jvm.dll 到执行档所在目录，此时请记得删除它。因为 Windows 作业系统会以执行档所在目录所找到的动态联结函式库为优先。

设定完成之后，请重新执行我们的程式，您就可以看到 Java 虚拟机器被成功地唤起了，如下图所示:
如果您希望可以看到整个唤起 Java 虚拟机器的详细资料，请将程式修改如下:
档案:Main.cpp

```
#include<iostream>
#include<jni.h>

using namespace std ;

int main()
{
 JavaVM* jvm ;
```

```
 JNIEnv* env ;
 JavaVMOption options[2] ;
JavaVMInitArgs vmargs ;
 long status ;

 options[0].optionString = "-Djava.class.path=.";
 options[1].optionString = "-verbose:jni";

 vmargs.version = JNI_VERSION_1_2;
 vmargs.options = options;
 vmargs.nOptions = 2;


 status = JNI_CreateJavaVM(&jvm,(void**)&env,&vmargs) ;
 if(status != JNI_OK)
{
 cout << "Java 虚拟机器建立失败/不知名的错误" << endl ;
 cout << "错误代码 : " << status ;
  return 1 ;
}
 cout << "Java 虚拟机器建立成功" ;

jvm->DestroyJavaVM();
 return 0 ;
}
```

您的命令列视窗将输出许多 jvm.dll 连结辅助用动态联结函式库的相关讯息，如下图:

(注:使用–verbose:jni 时，可以加上–Xcheck:jni 以取得更多进阶资讯)

叫用 Java 类别函式库

最后，这些杂七杂八的问题都搞定了，我们可以开始叫用 Java 类别函式库，程式如下:

档案:Main.cpp

```
#include<iostream>
#include<jni.h>

using namespace std ;

int main()
{
 JavaVM* jvm ;
 JNIEnv* env ;
 JavaVMOption options[1] ;
```

```
 JavaVMInitArgs vmargs ; long status ;

  options[0].optionString = "-Djava.class.path=D:\\java_and_vsdotnet\\java";

  vmargs.version = JNI_VERSION_1_2;
  vmargs.options = options;
  vmargs.nOptions = 1;


  status = JNI_CreateJavaVM(&jvm,(void**)&env,&vmargs) ;
  if(status != JNI_OK)
 {
  cout << "Java 虚拟机器建立失败/不知名的错误" << endl ;
  cout << "错误代码 : " << status << endl ;
   return 1 ;
 }
  cout << "Java 虚拟机器建立成功" << endl;

  jclass toolclass ;
  toolclass = env->FindClass("MyToolkit") ;
  if(toolclass == NULL)
 {
  cout << "找不到 MyToolkit 类别" ;
   return 1 ;
 }

  jstring jstr = env->NewStringUTF("My first class!");


  jmethodID function ;
  function                                                                    =
env->GetStaticMethodID(toolclass,"function1","(Ljava/lang/String;)V") ;
  if(function == NULL)
 {
  cout << "找不到 function1 函式" ;
   return 1 ;
 }
  env->CallStaticVoidMethod(toolclass, function, jstr);

 jvm->DestroyJavaVM();
  return 0 ;

}
```

首先，请先将 java.class.path 指向 MyToolkit.class 的所在位置，如下图

所示:

其原因已经在前面的章节里提过。
再来，我们要请虚拟机器载入 MyToolkit.class，方法如下:

由于 MyToolkit.class 里面的方法为静态方法(static method)，因此不用产生实体物件就可以直接使用。
名为 function1 的方法需要一个字串做参数，所以我们要产生一个 UTF 字串，方
法如下:

将类别载入之后，我们还必须指引 Java 虚拟机器找到类别里的静态方法，方法如下:

GetStaticMethodID 的最后一个参数为函式的传回值和参数，如果您不晓得这个参数怎么下，请使用 UltraEdit 开启 MyToolkit.class 即可，如下图:
最后，万事具备，只欠东风，我们呼叫该方法为我们服务，方法如下:

如果执行顺利，萤幕会输出底下画面:

就这样，我们完成了我们的目的。
结论

在上一章中，笔者为大家说明了如何从 Java Î C++ ，而本章反过来说明如何从 C++ Î Java。相信大家都已经见识到，其实使用 Java 来开发程式是不寂寞的。希望往后 Java 的爱好者对 Java 更有信心。

第五章

package 与 import 机制
所谓的定律，也不过是目前尚未被人推翻的假设罢了。

前言

许多对 Java 程式设计有兴趣的初学者，多半到书店买一本 Java 程式设计的入门书开始他的 Java 程式设计之旅。首先，他可能会到 http://www.javasoft.com 下载最新版本的 Java 2 SDK，将它安装在自己的电脑上，然后依照书本的指示，用文字编辑器输入一个名为 HelloWorld.java 的 Java 应用程式原始码档。熟悉 Java 的朋友当然可以顺利通过编译并成功地执行 Java 应用程式。但是对初学者来说，当他键入:

javac HelloWorld.java

之后可能会编译成功，产生 HelloWorld.class，但是，接下来当他输入:

java HelloWorld

时，却可能发现 java.exe 跑出恼人的讯息:
Exception in thread "main" java.lang.NoClassDefFoundError:HelloWorld

如果是一位对命令列模式(console)没有太多概念的初学者，可能连编译这一步都无法跨过，编译时萤幕上会出现一大堆让人不知如何下手的的错误讯息。因为这个进入 Java 程式设计领域的重大门槛，不知浇熄了多少 Java 初学者的热情:连简单一个 Hello World 的 Java 程式，想看到其执行结果都不是一件简单的事情了，想当然书本里头五花八门的程式设计议题都不再有意义。这两个门槛不知道阻碍了多少人学习 Java 程式设计。
本章上半部分将假设您是一位第一次使用 Java 的朋友，所以内容的编排都是为了要打破这进入门槛，让大家可以快快乐乐地学习 Java 程式设计；而下半部分就稍微复杂了点，如果您希望深入了解运作细节，那么您可以继续看完后半部分，没兴趣的话略过也无妨。如果您曾试着使用 JBuilder 这种威力强大的开发工具，您一定也会发现如果对于 Java 的 package 与 import 机制没有妥善的了解，那么开发程式的时候一大堆的错误讯息一样也会搞的您心烦意乱，希望看完本篇之后，可以让您更轻松驾驭像 JBuilder 这种高阶开发工具。

初探 package 与 import 机制

进入 Java 程式设计领域的第一个门槛，就是要了解 Java 的 package 与 import 机制，每一位学习 Java 的人都必须对这个机制有切确的了解，底下我们将假设您刚装好 Java 2 SDK 于 d:\jdk1.3.0_01 目录之中(注意! package 与 import 机制不会因为您用了新版的 JDK 而改变，如果您用的是 JDK 1.4.x，甚至是以后更新版本的 Java 2 SDK，都会产生与本章相同的结果)，并没有修改任何的环境变数。接着我们在 D 磁碟机根目录底下新增一个名为 my 的空目录(d:\my)，并以这个空目录作为我们讨论的起始点。
讨论一:
首先，请您将目录切换到 d:\my 底下，请先试着在命令列中输入

java 或 javac

您的萤幕上可能会输出错误讯息如下:

这是因为系统不知道到哪里去找 java.exe 或 javac.exe 的缘故。所以您可以试着输入指令:

```
path d:\jdk1.3.0_01\bin
```

然后您再重新输入

```
java 或 javac
```

就会看到以下画面:

请注意，以上说明只是为了强调 path 环境变数的使用。事实上，几乎大多数版本的 JDK 都会于安装时主动在<Windows 安装目录>\system32 底下复制一份 java.exe，而<Windows 安装目录>\system32 通常又是 Windows 预设 path 环境变数中的其中一个路径，所以一般的情况下，都会发生可以执行 java.exe，却不能执行 javac.exe 的情形。这小小的差异会产生微妙的差别，如果您阅读过本书第一章，他们之间的差异对您来说已经非常清楚。

接下来，请您在 d:\my 目录下新增两个档案，分别是:

A.java

```
public class A
{
  public static void main(String[] args)
 {
  B b1=new B() ;
  b1.print() ;
 }
}
```

B.java

```
public class B
{
 public void print()
 {
  System.out.println("package test") ; }
}
```

接着请您在命令列输入:

```
javac A.java
```

如果您的程式输入无误，那么您就会在 d:\my 中看到产生了两个类别档，分别是 A.class 与 B.class。

请注意，javac.exe 是我们所谓的 Java 编译器，它具有类似 make 的能力。举例来说，我们的 A.java 中用到了 B 这个类别，所以编译器会自动帮您编译 B.java。反过来说，如果您输入的指令是 javac B.java，由于 B.java 中并没有用到类别 A，所以编译器并不会自动帮您编译 A.java，

编译成功之后，请输入指令:

```
java A
```

您会看萤幕上成功地输出

package test

在此推荐您一个非常好用的选项: -verbose 。您可以在 javac.exe 或 java.exe 中使用此选项。他可以让您更了解编译和执行过程中 JVM 所做的每件事情。如果您在编译的时候使用–verbose 选项，结果如下:

从这个选项，您可以发现，由于编译器采用了 JDK 所在目录底下那套 JRE 的缘
故，所以编译时使用<JRE 所在目录>\lib\rt.jar 里头已经编译好的核心类别函式库(第一章的时候我们提过 Java 编译器也是用 Java 写成的，您可以试着让 javac.exe 使用其他 JRE 所在路径之下的 rt.jar 来进行编译吗? 留给大家做个小测验)。
当您执行 Java 程式时，使用–verbose 选项的结果如下:

省略…

您可以发现，由于 java.exe 根据预设的逻辑而选择使用了 JDK 所在目录底下那套 JRE 来执行程式，所以执行时所载入的核心类别都是使用<JRE 所在目录>\lib\rt.jar 里头那些。

您一定觉得上述讨论一的内容很简单，您也可以做到，所以接下来我们要更进一步，探究更复杂的机制。

讨论二:
请先删除讨论一中所产生的类别档，接着修改您的两个原始码，使其内容为:
A.java

```
import edu.nctu.* ;
public class A
{
  public static void main(String[] args)
 {
  B b1=new B() ;
  b1.print() ; }
}
```

B.java

```
package edu.nctu ;
public class B
{
 public void print()
 {
  System.out.println("package test") ;
 }
```

}
接着请您在命令列输入:
javac A.java
如果您的程式输入无误，那么您会看到萤幕上出现了许多错误讯息，主要的错误在于
A.java:1: package edu.nctu does not exist

意思是说找不到 edu.nctu 这个 package。其他两个错误都是因为第一个错误所衍生。
可是不对呀，按照 Java 的语法，既然 B 类别属于 edu.nctu，所以我们在 B.java 之中一开始就使用
package edu.nctu ;
而 A 类别用到了 B 类别，所以我们也在 A.java 开头加上了
import edu.nctu.* ;
所以这段程式在理论上应该不会发生编译错误才是。
为了解决这个问题，请您在 d:\my 目录下建立一个名为 edu 的目录，在 edu 目录下再建立一个名为 nctu 的子目录，然后将 B.java 移至 d:\my\edu\nctu 目录下，请重新执行
javac –verbose A.java
如果操作无误，那么您的萤幕上会显示底下讯息:

注意，如果您在 d:\my 底下仍保有原本的 B.java，则会产生底下错误讯息。

从这个输出您可以发现，编译器总是先到 A.java 本身所在的路径中寻找 B.java，虽然编译器找到了 B.java，可是比对过其 package 宣告之后，认为它应该位于 edu\nctu 目录下，不该在此目录下，因此产生错误讯息。这个动作在本章后面会有更详细的解释。
不管如何，这次终于编译成功了!! 您会在 d:\my 目录下找到 A.class，也会在 d:\my\edu\nctu 目录下找到编译过的 B.class。编译成功之后，请输入指令:
java A
您会看到萤幕上输出
package test
程式顺利地执行了。接这着咱们做个测试，请将 d:\my\edu\nctu 目录下的 B.class 移除，重新执行
java A
萤幕上会输出错误讯息:

意思是说 java.exe 无法在 edu/nctu 这个目录下找到 B.class。完成了这个实验之后，请将刚刚移除的 B.class 还原，以便往后的讨论。
到此处，我们得到了两个重要的结论:
结论一:
如果您的类别属于某个 package，那么您就应该将它至于该 package 所对应的相对路径之下。举例来说，如果您有个类别叫做 C，属于 xyz.pqr.abc 套件，那么您就必须建立一个三层的目录 xyz\pqr\abr，然后将 C.java 或是 C.class

放置到这个目录下，才能让 javac.exe 或是 java.exe 顺利执行。
其实这里少说了一个重点，就是这个新建的目录应该从哪里开始? 一定要从 d:\my 底下开始建立吗? 请大家将这个问题保留在心里，我们将在底下的讨论之中为大家说明。

结论二:
当您使用 javac.exe 编译的时候，类别原始码的位置一定要根据结论一所说来放置，如果该原始码出现在不该出现的地方(如上述测试中 B.java 同时存在 d:\my 与 d:\my\edu\nctu 之下)，除了很容易造成混淆不清，而且有时候抓不出编译为何发生错误，因为 javac.exe 输出的错误讯息根本无法改善问题。

在我们做进一步测试讨论前，请再做一个测试，请将 d:\my\edu\nctu 目录下的 B.class 复制到 d:\my 目录中，执行指令:
java A
您应当还是会看到萤幕上输出
package test
但是此时如果您重新使用编译指令
javac A.java 重新编译 A 类别，萤幕上会出现
bad class file: .\B.class
class file contains wrong class: edu.nctu.B
的错误讯息。

最后，请您删掉 d:\my 底下刚刚复制过来的 B.class，也删除 d:\my\edu\nctu 目录中的 B.java ，也就是让整个系统中只剩下 d:\my\edu\nctu 目录中拥有 B.class，然后再使用指令
javac A.java
您一定会发现，除了可以通过编译之外，也可以顺利地执行。

在测试中，我们又得到了两个结论:
结论三:
编译时，如果程式里用到其他的类别，不需要该类别的原始码也一样能够通过编译。

结论四:
当您使用 javac.exe 编译程式却又没有该类别的原始码时，类别档放置的位置应该根据结论一所说的方式放置。如果类别档出现在不该出现的地方(如上述测试中 B.class 同时存在 d:\my 与 d:\my\edu\nctu 之下)，有很大的可能性会造成难以的编译错误。虽然上述的测试中，使用 java.exe 执行 Java 程式时，类别档乱放不会造成执行错误，但是还是建议您尽量不要这样做，除了没有意义之外，这种做法像是一颗不定时炸弹，随时都有可能造成您的困扰。

讨论三:
请先删除讨论二中所产生的所有类别档，接着请将 d:\my 里的 edu 目录连同子目录全部移到(不是复制喔!)d:\底下。然后执行

javac A.java
糟了，萤幕上又出现三个错误，意思是说找不到 edu.nctu 这个 package。其他两个错误都是因为第一个错误所衍生。
接着请执行修改过的指令
javac –classpath d:\ A.java
就可以编译成功，您可以在 d:\my 目录下找到 A.class，也可以在 d:\edu\nctu 目录下找到 B.class。
请执行指令
java A
又出现错误讯息了，意思是说 java.exe 无法在 edu\nctu 这个目录下找到 B.class。

聪明的你一定想到解决方法了，请执行修改过的指令
java –classpath d:\ A
可式萤幕上仍然出现错误讯息，但是这回 java.exe 告诉您他找不到 A.class，而不是找不到 B.class。

好吧，那再将指令改成
java –classpath d:\;. A
哈哈!这回终于成功地执行了。

到此，我们归纳出一个新的结论，但是在此之前请回过头看一次先前的结论一，再回来看新结论: 结论五:
结论一中我们提到，如果您有个类别叫做 C，属于 xyz.pqr.abc 套件，那么您就必须建立一个三层的目录 xyz\pqr\abr，然后将 C.java 或是 C.class 放置到这个目录下，才能让 javac.exe 或是 java.exe 顺利执行。但是这个新建的目录应该从哪里开始呢? 答案是:"可以建立在任何地方"。但是您必须告诉 java.exe 与 javac.exe 到哪里去找才行，告诉的方式就是利用它们的-classpath 选项。在此测试中，我们把 edu\nctu 这个目录移到 d:\之下，所以我们必须使用指令:
javac –classpath d:\ A.java
与
java –classpath d:\;. A
告诉 java.exe 与 javac.exe 说:"如果你要找寻类别档，请到-classpath 选项后面指定的路径去找，如果找不到就算了"。
不过这里又衍生出两个问题，首先第一个问题是，那么为什么之前的指令都不需要额外指定-classpath 选项呢?这是因为当您执行 java.exe 和 javac.exe 时，其实他已经自动帮您加了参数，所以不管是讨论一或讨论二里所使用的指令，其实执行时的样子如下:
javac –classpath . A.java
或
java –classpath . A
意思就是说，他们都把当时您所在的目录当作是-classpath 选项的预设参数。那么您可以发现，如果是 javac.exe 执行时，当时我们所在的路径是 d:\my，

所以很自然地，他会自动到 d:\my 底下的 edu\nctu 目录寻找需要的档案，java.exe 也是一样的道理。
但是，一旦我们搬移了 edu\nctu 目录之后，这个预设的参数使得 javac.exe 要寻找 d:\my\edu\nctu 底下的 B.java 或 B.class 来进行编译时，发生连 d:\my\edu\nctu 目录都找不到的情况(因为被我们移走了)，所以自然发生编译错误。因此我们必须使用在指令中加上–classpath d:\，让 javac.exe 到 d:\下寻找相对路径 d:\edu\nctu。执行 java.exe 时之所以也要在指令中加上–classpath d:\，也是一样的道理。
如果您仔细比较两个修改过的指令，您还是会发现些许的差异，于是引发了第二个问题:为什么 javac.exe 用的是–classpath d:\，而 java.exe 如果不用–classpath d:\;.而只用–classpath d:\却无法执行呢? 我们在第二章曾经提过，-classpath 是用来指引 AppClassLoader 到何处载入我们所需要的类别。在执行时期，主程式 A.class 也是一个类别，所以如果不是预设的情况(预设指向目前所在目录，即”.”)，我们一定要向 AppClassLoader 交代清楚所有相关类别档的所在位置。但是对编译器来说，A.java 就单纯是个档案，只要在目前目录之下，编译器就能找到，基本上和 AppClassLoader 毫无关联，需要在 -classpath 指定路径，只是为了在编译时期 AppClassLoader 可以帮我们载入 B.class 或指引编译器找到 B.java，所以必须这样指定。因此，对 javac.exe 来说，-classpath 有上述两种作用;然而对于 java.exe 来说，就只有传递给 AppClassLoader 一种作用而已。

那么您一定会问，如果每次编译会执行都要打那么长的指令，那岂不是非常麻烦，虽然 java.exe 也提供一个简化的选项–cp(功能同–classpath)，但是还是很麻烦。为了解决这个问题，所以设计了一个名为 CLASSPATH 的环境变数，这
个环境变数可以省掉您不少麻烦。
您可以在命令列打上
set CLASSPATH=d:\;.

然后您就可以像之前一样使用指令:
javac A.java
与
java A
完成编译和执行的工作。但是请注意，如果您设定了环境变数 CLASSPATH，而在使用 java.exe 或 javac.exe 的时候也使用了–classpath 选项，那么环境变数将视为无效，而改为以了–classpath 选项所指定的路径为准。请注意，环境变数
CLASSPATH 与–classpath 选项分别所指定的路径并不会有加成效果。

JAR 档与 ZIP 档的效用
如果上述讨论都没有问题了，接下来请利用 Winzip 之类的压缩工具将 B.java 封装到 edu.zip 之中，并将 edu.zip 放到 d:\之下。 edu.zip 内容如下图所示:

请执行

javac –classpath d:\edu.zip A.java

会出现错误讯息如下:

从这里您可以知道，原始码档放在压缩档之中是没有任何效果的。前面既然说过编译或执行都可以不需要原始码，所以改将 B.class 封在 edu.zip 之中，edu.zip 的内容变成如下所示:

请重新执行

javac –classpath d:\edu.zip A.java

您会发现可以通过编译。同时您也可以使用

java –classpath d:\edu.zip;. A

一样可以成功地执行。

到这里我们又归纳出一个新结论:

结论六:

使用 ZIP 档的效果和单纯的目录相同，如果您在–classpath 选项指定了目录，就

是告诉 java.exe 和 javac.exe 到该目录寻找类别档；如果您在–classpath 选项指定了 ZIP 档的档名，那么就是请 java.exe 和 javac.exe 到该压缩档中寻找类别档。请注意，即使是在压缩档中，但是该有的相对路径还是要有，否则 java.exe 和 javac.exe 仍然无法找到他们所需要的类别档。

当然，在环境变数 CLASSPATH 中也可以指定 ZIP 档作为其内容。

为何这此我们要提出 ZIP 这种格式的压缩档呢? 如果您常常使用别人所开发的套件，您一定会发现别人很喜欢使用 JAR 档(.jar)这种档案格式。其实 JAR 档就是 ZIP 档。为何大家喜欢使用 JAR 档的封装方式呢?这是因为一般别人所开

发的套件总是会附带许多类别档和资源档(例如图形档)，这么多档案全部放在目录下很容易让人感到杂乱不堪，如果将这些档案全部封装在一个压所档之中，不但可以减少档案大小，也可以让您的硬碟看起来更精简。这也是为何每次我们下载了某人所开发的套件时，如果只有一个 JAR 档，我们就必须在环境变数 CLASSPATH 或–classpath 选项中加上这个 JAR 档的档名，因为唯有如此，我们

才能让 java.exe 和 javac.exe 找到我们的程式中所使用到别人套件中类别的类别档，并成功执行。 (在第二章中，我们也曾经提到可以放到<JRE 所在目录\lib\ext 底下，请回头参阅第二章)

路径的顺序问题

在这个讨论中最后要提的是，CLASSPATH 或–classpath 选项中所指定的路径或 JAR 档(或 ZIP 档)的先后次序也是有影响的。

举例来说，我们分别把 B.java 放在 d:\edu\nctu 与 d:\my\edu\nctu 目录下，然后键入指令:

javac –classpath d:\ A.java

您将发现编译器编译的是 d:\edu\nctu 目录下的 B.java。我们如果将指令修改为:

javac –classpath .;d:\ A.java
您将发现编译器编译的是 d:\my\edu\nctu 目录下的 B.java。所以在指定环境变数 CLASSPATH 或–classpath 选项时，所给予的目录名称或 JAR 档名称之顺
序一定要谨慎。
之所以要特别提醒大家这个问题，是因为常常在写程式的时候不小心在两个地方都有相同的套件(这也是笔者本身的切身之痛)，一个比较旧，但指定 classpath 的时候该路径较前，导致虽然另外一个路径里有新版的程式，但是执行或编译时却总是跑出旧版程式的执行结果，让人丈二金刚摸不着脑袋。所以最后还是要提醒您特别注意正视这个问题。

讨论四:
请删除前面讨论中制作的所有 JAVA 档和类别档，并重新在 d:\my 目录下产生一个新档案 A.java，内容如下:

A.java
```
package edu.nctu.mot ;
public class A
{
  public static void main(String[] args)
 {
  System.out.println("I'm A") ;
 }
}
```
在此我们把类别 A 归属到 edu.nctu.mot 套件之下。
接着请在命令列输入指令:
javac A.java
咦? 您会发现竟然可以成功地编译，这是怎么一回事呢? 其实这里的 A.java 和前面的 B.java 是有些许不同的。虽然 A.java 属于 edu.nctu.mot 套 件，而 B.java 属于 edu.nctu 套件。但是之前我们编译 B.java 时候并非直接编译它，而是透过 javac.exe 类似 make 的机制间接编译 B.java(因为类别 A 中用到了类别 B)。可是在这里，我们直接编译 A.java，没有透过 make 机制，所以可以编译成功(意思就是说前面讨论中用到的 B.java，即使不放在该放置的目录中一样可以成功编译，只要它自己以及所用到的类别可以在环境变数 CLASSPATH 或–classpath 选项指定的位置找到即可)。
好，编译成功了，现在我们试着执行看看，请输入:
java A
相信您的萤幕上一定出现了许多错误。

接下来我们应当根据结论一所说，建立一个名为\edu\nctu\mot 的目录，并将 A.java 至于其中，并删除 d:\my 里面的 A.java 和 A.class。然后输入指令
javac A.java
萤幕上出现错误讯息

奇怪了，前面不是说 javac.exe 会内定从目前所在目录寻找类别的原始码呢? 这里怎么无法运作呢? 其实问题和前面一样，这是因为现在我们直接要求 javac.exe 编译 A.java，而非透过 make 机制间接编译，所以我们的指令无法让 javac.exe 清楚地知道我们的意图，所以我们必须更改指令内容，您会想到的指令可能有以下几种:

1 javac edu.nctu.mot.A.java

2 javac -classpath .\my\edu\nctu\mot A.java

3 javac -classpath d:\my\edu\nctu\mot A.java

4 javac d:\my\edu\nctu\mot\A.java

其中，第一种指令是对 Java 比较熟悉的人可能会想到的方法，意义我们会在稍后介绍。第二和第三种指令其实意思是一样的，只是一个用了相对路径，一个用了绝对路径。但是，真正能够成功编译 A.java 的却只有第四种指令。所以从这里可以得知，如果您的专案中充满了许多类别，请尽量利用 javac.exe 的 make 机制来自动帮您编译程式码，否则可是很辛苦的。

现在我们试着执行，请输入:

java A

java.exe 说他找不到类别 A。

我们必须更改指令内容，您会想到的指令可能有以下几种:

1 java edu.nctu.mot.A

2 java -classpath .\edu\nctu\mot A

3 java -classpath d:\my\edu\nctu\mot A

4 java d:\my\edu\nctu\mot\A

第二和第三种指令其实意思是一样的，只是一个用了相对路径，一个用了绝对路径，但是，真正能够执行的却只有第一种指令。这个指令的意思是告诉 java.exe 说:”请你根据环境变数 CLASSPATH 或–classpath 选项指定的位置执行相对路径

\edu\nctu\mot 之下的 A.class”，前面我们提过，如果您没有指定环境变数 CLASSPATH 或–classpath 选项，预设会用目前所在路径，所以第一种指令真正

执行时应该是:

java –classpath . edu.nctu.mot.A

因为当时我们所在的路径为 d:\my，所以很自然地 java.exe 会寻找 d:\my\edu\nctu\mot 目录中的 A.class 来执行。

根据结论五所说，如果你将指令改成

java –classpath d:\ edu.nctu.mot.A 那么就无法执行了，除非您在 d:\edu\nctu\mot 里也有 A.class。

以上做了那么多测试，归纳了这几点结论，最后请大家在利用 package 与

import 机制的时候务必做到下列几项工作:
1. 将 JAVA 档和类别档确实安置在其所归属之 package 所对应的相对路径下。
2. 不管使用 java.exe 或 javac.exe，最好明确指定–classpath 选项，如果怕麻烦，使用环境变数 CLASSPATH 可以省去您输入额外指令的时间。
请注意在他们所指定的路径或 JAR 档中是否存有 package 名称和类别名称相同的类别，因为名称上的冲突很容易引起混淆。
只要确实做到上述事项，就可以减少萤幕上大量与实际问题不相干的错误讯息，并确保您在 Java 程式设计的旅途中顺利航行。

在此要提醒大家，到此为止我们所举的例子都是使用 java.exe 和 jvac.exe 的测试结果，但是在实际生活中，只要和 Java 程式语言相关的工具，都会和 package 与 import 机制息息相关。比方说如果您开发 MIDlet，必定会用到 preverify.exe；如果要使用 JNI，您就会用到 javah.exe，使用这些工具时如果您没有遵循 package 与 import 机制的运作方式，可是无法得到您要的结果。

深入 package 与 import 机制

在本章前半个部分，笔者向大家说明了初学者进入 Java 程式设计领域常见的重大门槛 - package 与 import 机制。相信大家已经可以自行处理 Class Not Found 这一类 Exception 了。但是，前半个部分所使用的讨论方式只是工程上的结果，那么原理呢? 只要理解了这些错误之所以出现的理由，就可以知道编译之所以成功或错误的原因。因此，本章下半部要用理论的方式为大家说明 package 与 import 机制运作的秘密。
接下来的讨论都是假设你遵守「每一个类别都会根据其所属的 package 放在正确的相对路径上」 。从前半个部分的讨论我们可以知道，如果我们不把类别
放在正确的相对位置，则会产生许多与问题本身不相干的错误讯息，这这部分涉及到 Java 编译器自己本身对错误的处理方式，所以我们不加以讨论。

编译时期(Compile-time)的 Package 运作机制

在讲解编译时期的 Package 运作机制之前，首先我们要先制作一个完整的

范例，我们测试所使用的目录结构如下图所示: 目錄: src

目錄 : comtime 目錄 : outer

目錄 : com

目錄 : edu

档案 : CA.java
档案 : Main.java
档案 : CA.java
档案 : CA.java
档案 : CA.java

目錄 : com

档案 : CA.java

在我们的测试范例之中，我们总共有六个档案，他们的内容分别如下:

src\comtime\Main.java

```
public class Main
{
  public static void main(String args[])
 {
    CA ca = new CA() ;
 }
}
```

src\comtime\CA.java

```
public class CA
{
}
```

src\comtime\edu\CA.java

```
package edu;
public class CA
{
}
```

src\comtime\com\CA.java

```
package com; public class CA
{
}
```

src\outer\CA.java

```
package outer;
public class CA
{
}
```

src\com\CA.java

```
package com;
public class CA
{
}
```

接下来，我们将使用 src\comtime 作为我们测试的根目录(意即:直接放置在此目录之下的类别可以不需属于任何 package，而属于任何 package 的类别将可以以此目录做相对参考点，根据自己所属的 package 产生相对应的目录来放置自己)，因此请将提示符号切换到 src\comtime 目录，并执行

javac Main.java Î 指令(1)

然后你将发现不属于任何 package 的两个类别分别被编译成类别档:

接下来，请将 Main.java 的内容修改成:
src\comtime\Main.java
```
import com.*;
import edu.*;
public class Main
{ public static void main(String args[])
 {
   CA ca = new CA() ;
 }
}
```
重新执行
javac Main.java Î 指令(2)
你会发现与使用指令(1)的测试结果相同，不属于任何 package 的两个类别被编译成类别档:

从这两个简单的测试大家可以发现，不管你有没有使用 import 指令，存在目前目录下的类别都会被编译器优先采用，只要它不属于任何 package。这是因为编译器总是先假设您所输入的类别名称就是该类别的全名(不属于任何 package)，然后从-classpath 所指定的路径中搜寻属于该类别的.java 档或.class 档。从这里我们可以知道 default package 的角色非常特殊。
接下来，请将 src\comtime\CA.java 改名成 NU.java，代表它不再为我们所使用。
请重新执行
javac Main.java Î 指令(3)
你会发现底下错误讯息:

这个错误讯息说明了编译器产生疑惑，因为 com 与 edu 这两个 package 底下都有名为 CA 的类别，他不知道该用哪一个。解决的办法就是明确地告诉编译器我们要哪个 package 底下的 CA，解决方法有两种:
1. 在 import 处明确宣告，也就是把 Main.java 改成
src\comtime\Main.java import com.CA;
```
import edu.*;
public class Main
{
  public static void main(String args[])
 {
   CA ca = new CA() ;
 }
}
```
或者

2. 引用时详细指定该类别的全名(即套件名称.类别名称的组合)
src\comtime\Main.java
```
import com.*;
import edu.*;
public class Main
{
  public static void main(String args[])
 {
   com.CA ca = new com.CA() ;
 }
}
```

假设我们要使用的是 com.CA 这个类别，只要利用这两种方法的其中一种，编译器就可以明确地知道我们所希望使用的类别，所以当你选用其中一种方法来修改 Main.java 之后，你一定可以发现 src\com\CA.java 被编译成类别档了。

好的，接下来我们再将 Main.java 修改成
src\comtime\Main.java
```
import com.*;
import edu.*;
import outer.*;
public class Main
{
  public static void main(String args[])
 {
    CA ca = new CA() ;
 }
} 执行
```
javac Main.java Î 指令(4)
之后，你会看到其中一项错误讯息:

框住的地方告诉我们编译器找不到 outer 这个套件，为了解决这个问题，我们将指令改成
javac –classpath .. Main.java Î 指令(5)
结果萤幕上还是出现错误讯息:

这次编译器竟然告诉我们找不到 edu 这个 package，我们只好再将指令改成
javac –classpath ..;. Main.java Î 指令(6)
这次不再出现找不到 package 的错误讯息了，而是编译器再次告诉我们他不知道该采用哪一个 package 底下的 CA:

解决的方法我们在前面已经提过了。假设我们把 Main.java 改成
src\comtime\Main.java
```
import com.CA;
```

```
import edu.*; import outer.*;
public class Main
{
  public static void main(String args[])
 {
    CA ca = new CA() ;
 }
}
```

重新使用指令
javac –classpath ..;. Main.java Î 指令(7)
来编译那么你会发现什么结果呢?你将发现 src\com 底下的 CA.java 被编译了。
如果我们把指令改成:
javac –classpath .;.. Main.java Î 指令(8)
你将发现 src\comtime\com 底下的 CA.java 被编译了。

从上面的测试结果，揭露了 java 编译器的运作情形，当 java 编译器开始编译某个类别的原始码时，首先他会作一件事情，就是建构「类别路径参考表」，建构逻辑我们用下面几张图来表示:
图:使用指令(3)的时候，建构「类别路径参考表」的程序

類別路径參考表 步骤 1.

根据
选项 选项- -classpath classpath
或

环境变數 环境变數 CLASSPATH CLASSPATH

的内容建构類別路径參考表

.
结果.
因为没有指定
选项 选项- -classpath classpath
或

环境变數 环境变數 CLASSPATH CLASSPATH

所以预设情况下類別路径

參考表只有一笔记錄,即

目前的目錄(“.”)

请注意，环境变数 CLASSPATH 的内容会被选项-classpath 所覆盖，没有加成效果。也就是说，只有其中一个会被拿来当作类别路径参考表的内容。所以当我们使用指令(7)或指令(8)的时候，结果如下图所示: 图:使用指令(7)与使用指令(8)的时候， 「类别路径参考表」的内容

類別路径參考表

.

使用指令(7)的结果

1
.. 2

類別路径參考表

..

使用指令(8)的结果

1
. 2

当编译器将类别路径参考表建好之后，接着编译器要确定它可以利用类别参考表里的资料作为相对起始路径，找到所有用到的 package，其流程如下图所示: 图:使用「类别路径参考表」确认 package 之存在的流程。

類別路径參考表

D:\test
步骤 3.

根据類別路径參考表的内容为起始点,

比对其中一个路径下是否存在同名的

目錄或档案名称

1
步骤 2.
找出原始码之中所有使用
import packagename.*;
或
import packagename.classname

的指令,并将 packagename 之中的

“.”以“/”取代.
C:\p.jar 2

完成建构類別路径參考表的程序

以上面这张图来说，这是假设我们用的指令是:
javac –classpath d:\test;c:\p.jar test.java
所以使得类别路径参考表之中有两笔资料，接着，假设我们的 test.java 之中有 import A.B.* ; import C.D ;
那么编译器就会先看看 d:\test 目录底下使否存在有 A\B 目录，也就是说编译器会检查 d:\test\A\B 究竟是否存在，如果找不到，它会试着再找看看 c:\p.jar 之中是否存在 A\B 这个目录，如果都没找到，就会发出 package AB does not exist 的讯息。同理，编译器也会试着寻找是否存在 d:\test\C\D.class 或 d:\test\C\D.java(如果找不到就进入 c:\p.jar 里头找)，并确定他们所属的 package 名称无误，也确保他们为 public(如此才能让其他 package 的类别所存取，且档名一定与类别名称相同)，如果没找到就会发出 cannot resolve symbol 的错误讯息。

从这里我们回头看指令(4)，为何它会说 package outer does not exist?
因为指令中根本没有指定选项-classpath，我们也没有设定环境变数 CLASSPATH，所以类别路径参考表只有预设的".”，因此编译器根本无法在 src\comtime 之下找到 outer 这个子目录。同理，回头看看指令(5)，因为设定了选项-classpath，所以所以类别路径参考表只有"..”，对 src\comtime 来说，其父目录即 src，而编译器在 src 目录之下根本找不到 edu 这个子目录，所以自然发出 package edu does not exist 的错误讯息。所以要让编译器至少在建构类别路径参考表的过程没有问题，我们就必须使用指令(6)才行。

根据上述讨论，编译器会完成一张名为「类别参考表」与「相对类别参考表」的资料结构，如下图所示
图:使用指令(7)的时候，建构「类别参考表」的程序

類別參考表

步驟 1.
只要是明确使用
import packagename.classname

的格式,就会被加入類別參考表之中 com.CA

结果.
因为程式码内容为
import com.CA;
import edu.*
import outer.*

所以類別參考表只有一笔

记錄,而相对類別參考表

有兩笔记錄.

1

相对類别參考表

com.* 1
outer.* 1
步骤 2.
只要是使用
import packagename.*的格式,就

会被加入相对類别參考表之中

最后，编译器必须开始解析程式码里头所有的类别名称，其逻辑如下:
图:类别名称解析的程序
使否指定全名?
(即 packagename.classname)

根据類别路径參考表与

類别全名做配对组合

,看看是否在某類别路径下

找的到该類别?

如果该類别

不存在類别档,

则先编译该類别原始码

产生類别档.

(make 机制)
开始
Yes No
Yes No
产生
编译错误
设为 default package,

根据類别路径參考表下的设定,

看看这些路径下是否有

该類别档或其原始码?

Yes

如果该類別

不存在類別档,

则先编译该類別原始码

产生類別档.

(make 机制)
No
Package
解析

由上面这张图，我们可以解释指令(1)(2)的结果，同时也可以解释当我们使用指令(3)之后，为了解决了类别名称暧昧不明的情形，而我们使用程式码
com.CA ca = new com.CA() ;
之后的编译结果。但是要解释指令(7)(8)的结果，就必须用下图来说明:
图:package 解析的程序(接上图)

是否在類別參考表之中有同名的類別

有几个同名類別?

开始
Yes No
一个
产生编译错误

先编译第一个符合的類別,

如果该類別不存在類別档,

则先编译该類別原始码

产生類別档.

(make 机制)
一个以上
一个
产生编译错误
没有
或
一个以上

根据類別路径參考表与相对類別

參考表的配对组合,看看是否在某

類别路径下找的到该類别?

根据類别路径參考表与

類别參考表的配对组合,

看看是否在某類别路径下

找的到该類别?

从上图我们可以解释，可以解释当我们使用指令(3)之后，为了解决了类别名称暧昧不明的情形，而我们使用程式码
import com.CA ;
之后的编译结果。
当我们使用指令(7)的时候，由于类别参考表之中只有 com.CA 一个名为 CA 的类别，所以接下来编译器根据相对类别参考表的内容，先合成..\com\CA.class 与..\com\CA.java，所以我们才会看到 src\com 底下的 CA 被编译。同理，如果改用指令(8)，由于会合成.\com\CA.class 与.\com\CA.java， 所一才导致 src\comtime\com 底下的 CA.java 被编译。
上面这张图还可以解释指令(6)之所以发出错误讯息的原因，也同样地可以解释如果我们的原始码为:
src\comtime\Main.java
```
import com.CA;
import edu.CA;
import outer.*;
public class Main
{
  public static void main()
 {
    CA ca = new CA() ;
 }
}
```
编译器一样发出错误讯息

的原因。
从上面所讨论的 Java 编译器运作逻辑，亦对两个一般人常常有的误解做了解释:
1. import 和 C/C++里的 include 同意义。
2. 在 import 里使用万用字元，如 import xy* ，会让编译器效率减低。

其中，第一个误解也常让很多程式设计师觉得不解，为什么在 Java 之中无法使用

import java.*.*
这种有一个以上万用字元的语法? 由上面的讨论你可以知道，在 Java 里头，import 应该和 C/C++里的
namespace 等价才对，而非一般所说的 include。
假设你有两个类别，分别是 XA 与 XBC，你必须分别
import X.* ;
import X.B.* ;
才行，因为这两个指令会让编译器分别在类别路径参考表之中建立两笔资料，如此一来当编译器将类别名称与类别路径参考表里的资料作合成的时候，才能让你的程式:
C c1 = new C();
通过编译，如果你只有
import X.* ;
则编译器回发出错误讯息，说它无法解析类别 C。也就是说，类别路径参考表里的资料，只是单纯拿来合成，让编译器可以很快地定位到类别档或原始码的位置，并不会递回式地帮我们自动搜寻该路径底下其他的类别，所以不管在 import 处使用完整的类别名称或万用字元，即使原始码会使得类别路径参考表和相对类别参考表很大很大，但是 javac.exe 内部在处理这些资料表格时也是采用 Hash Table 的设计，所以对编译速度的影响几乎微乎其微。

好的，到这里为止，我们已经将整个编译机制前半部关于 package 与 import 机制的部分向大家解释清楚了，最后我们把整个编译程序之中与 package 与 import 机制相关的部分用一张完整的图做表示:
图:整个编译程序与 package 相关的步骤解析

類別名称解析程序

建构

類別路径參考表

与

類別參考表

建构成功
编译结束,
发出警告讯息
开始
建构失败

已存在该類別的

類別档

继续其他编译工作

该類別的

類别档

不存在 寻找该類别原始档

找不到

找到
编译该原始码
Make 机制,递回式编译

其实类别名称解析程序之中，还有一个重要的步骤，在图中并没有说明，就是当
编译器找到该类别的类别档或原始码之后，都会加以验证他们使否真的属于这个 package，如果有问题，编译器一样会发出错误讯息;但是如果并非透过 make(自动解析名称，自动编译所需类别)机制来编译程式，是不会做这项验证工作的。这也是为什么笔者再三强调每一个类别都要根据其所属的 package 放在正确的相对路径之下，否则会导致编译器产生许多难以理解的编译错误。
从这里我们也可以发现，如果我们不够透过 make 机制，而直接切到 src\com 底下直接打指令
javac CA.java
也可以编译成功，而且就算 CA.java 的内容为
src\com\CA.java
```
package test;
public class CA
{
}
```
也就是说所属 package 和所在路径不一致的情况下，一样可以编译成功，仔细回头看看之前所有的示意图，我们将可以发现这段程式码并不会引发任何 package 与 import 机制的相对应动作，所以可以编译成功。也因为如此，请大家尽量让整个应用程式所用到的类别都可以透过 make 机制来编译，否则一个一个手动编译不但辛苦，也会有潜在的错误发生。

Java 的动态连结本质
用过 Java 的朋友都知道，Java 会将每个 class 宣告编译成一个附档名为.class 的档案。不管你在同一个原始码(.java)之使用了几个类别宣告，他们都会一一编译成.class 档，即使是内部内别、匿名类别都一样，举例来说:
MultiClass.java
```
class A
{
   class C
   {
   }
}
```

```
class B
{
   public void test()
   { class D
      {
      }
   }
}
```
当你编译 MultiClass.java 之后，你会发现目录下出现了底下几个类别档，他们分别是:
```
 A.class
 A$C.class
    B.class
 B$1$D.class
```
```
A.class
MultiClass.java
Javac MultiClass.java
B.class A$C.class B$1$D.class
```

这种方式与传统的程式语言有很大的不同，一般的程式语言在最后都是将所有的程式码 static link 至同一个执行档(.exe)之中，如果要做到动态抽换功能模组的的话，就必须借助动态联结函式库(Windows 上为 DLL，UNIX 上为 share object)。但是，在 Java 之中，每一个类别所构成的类别档我们都能将它视为动态联结函式库。

好，假设程式码如下所示，

```
src\comtime\Main.java
import com.CA;
import edu.*;
import outer.*;
public class Main
{ public static void main(String args[])
 {
    CA ca = new CA() ;
 }
}
```

```
src\comtime\com\CA.java
package com;
public class CA
{
}
```
而各位执行了上述的指令(8) ，各位会看到编译之后产生了 src\ccomtime\Main.class 与 src\comtime\com\CA.class。接着我们执行指令:

java Main Î 指令(9)
程式将成功地执行完毕，萤幕上没有任何错误讯息，但也没有其他讯息，

于是我们将 CA.java 改成:
src\comtime\com\CA.java

```
package com;
public class CA
{
   public CA()
   {
      System.out.println(“New CA”) ;
   }
}
```

并手动切换到 src\comtime\com\CA.java 之下编译 CA.java，再重新执行指令(9)，结果如下图所示:
根据之前我们讨论的，我们可以保证 Main.java 并没有被重新编译，只有 CA.java 被我们重新编译而已，而如各位所见，有了新的输出结果，足以证明 Java 动态连结的本质。如果各位需要更细部的资讯，请尝试着使用:
java –verbose Main Î 指令(10)
这个部分输出非常多讯息，我们只取其中一个部分让大家看:

萤幕上会告诉我们系统究竟到哪里载入了什么类别，聪明的各位，也看出了这些输出所代表的意函吗?

执行时期(Run-time)的 Package 运作机制

根据上述所描述的 Java 动态连结本质，那么，大家一定很疑惑，究竟整个系统是根据什么样的原则来载入适当的类别档呢?
首先我们先来看看前面两个类别编译之后产生的类别档之内容:
src\comtime\Main.class 的内容:

src\comtime\com\CA.class 的内容
从这两个档案的内容我们可以知道，在类别档之中，所有对于特定类别的所有动作都被转换成该类别的全名，我们在 Main.java 之中所写的 import 资讯全部都不见了(也就是说，import 除了用来指引编译器解析出正确的类别名称之外，没有其他功能了)，Main.class 之中原来我们只有写 CA 的部分，编译之后变成 com/CA。同样的情况也出现在 CA.class 之中，我们可以发现编译器自动把 package 名称和类别名称做了结合，也组成了 com\CA 于类别档之中。
在执行时期时，仍然用到一个与编译器相同的程序，就是类别路径参考表的建构，而利用动态连结载入类别档的机制如下图所示
图:执行时期的动态连结相关步骤解析
建构

類別路径參考表

符合

执行错误,

发出 Exception 讯息
开始

不符合

根据類别档内的资讯,

与類别路径參考表的

资料合成類别档的

绝对路径.

检查该類别的

類别档内资讯,

是否符合相对路径资讯.

载入该類别

找到類别档 找不到

類别档

我们以指令(9)的例子来说明，当系统执行 Main.class 的时候，首先先建立类别路径参考表，因为我们没有指令选项-classpath 或环境变数 CLASSPATH，所以类别路径参考表之中只有一笔资料，即预设的”.”(目前的目录)，首先，它会先寻找 Main.class，因为 Main.class 就在目前的目录之下，所以系统合成”.\Main.class”，自然就找到该类别档了。接着系统要检查 Main.class 之内的资讯，由于我们是以 src\comtime 为起始点，所以 Main.class 可以不需要属于任何类别，所以符合相对路径资讯，因此开始备载入执行。
接着，Main.class 之中用到 com/CA，所以它会合成”.\com\CA.class”，因此系统也可以找到 CA.class，在 CA.class 之中有个资讯为 com/CA，所以符合 src\comtime\com 这个相对路径，因此也正常地载入执行。
所以，我们可以很大胆地预测，如果您用的指令是
java –classpath .. Main
其执行结果:

系统告诉我们它找不到 Main 这个类别，因为从类别路径参考表的资料来看，根本没有足够的路径供我们找到 Main.class。这样各位明白了。

检视

利用相同的逻辑，我们回头来解释一下，在本章前半部之中谈到 package 与 import 机制那个段落的最后一个范例，我们所使用的档案是:

A.java

```
package edu.nctu.mot ;
public class A
{
  public static void main(String[] args)
 {
  System.out.println("I'm A") ;
 }
}
```

编译用的指令有以下几种，我们分别解释当时其成功与失败之原因:

1 javac edu.nctu.mot.A.java

类别路径参考表中只有一个".”，以此为起始点，找不到名为 edu.nctu.mot.A.java 的档案。也就是说，在编译时期，不会自动把命令列之中所指定的原始档之中的".”转换成"/”。我们指定什么档案，编译器一律认为是单纯的档名。

2 javac -classpath .\my\edu\nctu\mot A.java

类别路径参考表中只有一个".\edu\nctu\mot”，以此为起始点，其实可以找到名为 A.java 的档案，可是问题是，A.java 并非透过 make 机制来编译，而是我们手动打指令编译，所以根本不会用到类别路径参考表。因此编译器根本找不到 A.java。

3 javac -classpath d:\my\edu\nctu\mot A.java

类别路径参考表中只有一个"d:\my\edu\nctu\mot”，以此为起始点，其实可以找到名为 A.java 的档案，可是问题是，A.java 并非透过 make 机制来编译，而是我们手动打指令编译，所以根本不会用到类别路径参考表。因此编译器根本找不到 A.java。

4 javac d:\my\edu\nctu\mot\A.java

在指定的路径之下可以找到该档案，可以编译成功。

执行用的指令有以下几种，我们分别解释当时其成功与失败之原因:

1 java edu.nctu.mot.A

类别路径参考表中只有一个".”，以此为起始点，系统会自动把".”转换成"/”，因此会找到".\edu\nctu\mot”底下的 A.class。

2 java -classpath .\edu\nctu\mot A

类别路径参考表中只有一个".\edu\nctu\mot”，以此为起始点，系统会 A.class，可是在检查类别档内部资讯的时候，发现 A.class 属于 edu.nctu.mot 这个 package ，所以应该放置在".\edu\nctu\mot\edu\nctu\mot”这个目录之下，因此出现错误

讯息。
3 java -classpath d:\my\edu\nctu\mot A
类别路径参考表中只有一个”d:\my\edu\nctu\mot”，以此为起始点，系统会 A.class，可是在检查类别档内部资讯的时候，发现 A.class 属于 edu.nctu.mot 这个 package ，所以应该放置在”.\edu\nctu\mot\edu\nctu\mot”这个目录之下，因此出现错误讯息。
4 java d:\my\edu\nctu\mot\A
类别路径参考表中只有一个”.”。但是系统根本无法接受这样子的命令。

结论

在本章上半部分，笔者先采用程式黑手的方式讨论整个 package 与 import 机制，说明一些初学 Java 的人常见的问题。不过毕竟是程式黑手的方式，有太多的使用案例会造成许多的错误讯息，我们并无法完全讨论到，因而让人有骚不到痒处的感觉。因此在本章下半部分，笔者采用由理论验证实际的方式，为大家说明整个 package 与 import 机制的来龙去脉，只要理解了原理，就能以不变应万变。希望本章的内容可以让您豁然开朗。

第六章

Ant
中庸是一种滑头的行为。

前言

在第五章的时候，笔者曾经提到，Java 编译器(javac.exe)本身具有类似 make 的能力。也就是说，Java 编译器可以根据原始码之中的宣告及型态资讯，配合 import 指令，就可以自动找出类别档或原始码档来进行编译工作。这个行为就是笔者所称的『make 机制』 。底下撷取自第二章的程式码就是一个说明 make 机制的绝佳范例:

档案: Office.java

```
public class Office
{
public static void main(String args[])
{
if(args[0].equals("Word"))
{
Word w = new Word() ;
w.print() ;
}else if(args[0].equals("Excel"))
{
Excel e = new Excel() ;
e.print() ;
}
}
}
```

Word.java
Excel.java

编译器先扫描 Office.java，发现程式中用到了 Word 与 Excel 这两个类别，所以会自动去寻找 Word.java 与 Excel.java 来进行编译。如果 Word.java 与 Excel.java 里用到其他类别，那么一样会递回式地自动寻找及编译。

虽然 Java 编译器本身有类似 make 的功能，但是仍有力有未逮之处。比方说第二章所提到的另外一个范例，Java 编译器本身的 make 能力就无法涵盖: 2

档案: Office.java

```
public class Office
{
public static void main(String args[]) throws
Exception
{
Class c = Class.forName(args[0]) ;
Object o = c.newInstance() ;
```

```
Assembly a = (Assembly) o ;
a.start() ;
}
}
```

Word.java Excel.java
Assembly.java

由于我们在程式中用了 Class.forName()方法，而且把其他的功能都抽象地用 Assembly 介面来表示，因此程式码之中找不到类别之间的相依性，使得您编译 Office.java 的时候，编译器只会自动编译 Assembly.java，却无法顺道编译 Word.java 及 Excel.java。因为上述的情况，使得 Java 编译器无法有效编译出所有程式执行时所需的类别档。因此需要有更高阶的管理工具来帮助我们。本章所介绍的管理工具就是 - Ant。

关于 Ant

许多学习 JSP/Servlet 程式设计的朋友，一定都曾经拜访过 Apache Jakarta 网站(http://jakarta.apache.org)下载 Tomcat。 Tomcat Web Server 是一套可以支援最新 JSP/Servlet 规格的网页伺服器，
图 1: Apache Jakarta 网站(http://jakarta.apache.org) 3

在下载 Tomcat 之余，您是否曾经观察过网页左方的子计划列表(Sub Projects)呢? 这么多子计划之中，有一个子计划的名称很有趣，功能也最吸引人，那就是 Ant。
图 2: Ant 计划的标志

Ant 是怎样的一个东西呢? 为什么这样一个简单的工具会在今年度获得由 Java World 这本线上杂志的读者票选，成为本年度最佳的开发工具之一?
Ant 是 Another Neato Tool 的缩写，是一位 Sun 内部的工程师，James Duncan Davidson 所开发出来的一套小工具。根据 James 本人所说，他之所以会想要制作 Ant，是因为他被一套历史悠久的开发工具– make 给搞烦了。

不管您是在 Windows 或 Unix 开发程式，只要选用 C/C++程式语言作为开发语言，那么十之八九都会选用 make 作为您的专案管理工具，如果您是一位 Open Source 的爱好者，或者您常常在 Linux 或 FreeBSD 底下工作，您最常使用的软体安装指令大概就是 make all 或 make clean。就算您使用如 Visual C++或 Borland C++Builder 这样高阶的开发工具，这些工具在底层仍然采用 make 来作为专案管理之用。正由于 make 是设计最初就是设计给 C/C++这类 4 程式语言所使用的专案管理工具，即使 make 能够支援各种程式语言的专案开发，但是却往往因为各种程式语言的特性，使得无法完全发挥特定语言的特色。以 Java 来说，Java 和 C/C++最明显的不同就是 C/C++会用到 include 来引入标头档，但是 Java 没有这样的机制(Java 里的 import 和 include 是不同的两种机制。 Java 的 package/import 指令和 C++里头的 namespace/using 指令类似，笔者已于本书第五章详细解释过其运作机制)。

很多程式语言本质上的差异都导致 make 无法善用 Java 的特性，因此当我们用 make 来管理 Java 专案时，虽然可以正常运作，但是在效能上令人诟

病。 James 就是遇到了这样的问题，最后，他为了一次解决所有讨厌的麻烦事，于是著手开发可以让他的职业生涯更轻松的工具，这就是 Ant 的由来。没想到 Ant 在释出之后，开始受到 Java 社群的欢迎，现在连.NET 上也有对应的工具 NAnt(http://nant.sourceforge.net/)，可见得 Ant 的确是一套相当成功的作品。
图 3: NAnt 计划的标志

本文之所以选择这套开发工具来进行讨论，是因为笔者长久观察使用 Java 的程式设计师之后，赫然发现，使用 Java 来开发程式的程式设计师，通常分成两大类，一类是使用如 Borland JBuilder 或 Sun 自己所推出的 Sun One Studio(以前叫做 Forte for Java)这类的高阶开发工具的程式设计师。这些人会选用高阶开发工具，通常是因为高阶的开发工具可以很快地帮我们产生程式码的主干，尤其在开发伺服器端应用程式时(如 EJB)，开发工具都可以很快地帮我们包装成一个包裹档，并直接帮工程师部署到伺服器上进行测试。高阶开发工具另一个吸引人的特点就是它可以帮我们有效率地管理整个软体专案，这也是很多人选择高阶开发工具的主因；另外一类则是属于骨灰级的工程师，他们喜欢拿 JDK 配上 UltraEdit 就直接开发程式，因为高阶开发工具虽然好用，可是会产生一堆杂七杂八且没有效率的程式码，所以这类的人宁可凡事自己动手来，虽然开发起来比较复杂，这些人到也自得其乐。
这两种类型的程式设计师非常地极端，不过并没有孰好孰坏的问题，只有个人喜好的问题。在这个流行新中间路线的年代，笔者希望找到一个好方法，可以在两者之间取得平衡点。找出这个平衡点的最佳方法就是使用 Ant。当我们在开发 Java 应用程式的时候，Ant 可以让我们利用 XML 里头阶层的概念来组织专案，Ant 也可以让我们不需要重复地输入一堆相同的指令(javac、java、jar 等指令)，如此一来，不管在专案的规划上，或程式的撰写及测试上，都可以节省大量的时间，尤其如果您是一个喜欢使用文字编辑器+ JDK 来开发 Java 程式 5 的程式设计师，那么 Ant 的出现，可以让您的生产力开始追上使用高阶开发工具的程式设计师。如果您只是想使用高阶开发工具的专案管理功能，却讨厌开发工具占据了庞大的硬碟空间与缓慢的启动速度，那么 Ant 更是上上之选，因为 Ant 是一个不占硬碟空间，而且执行效率奇佳的一套专案管理及开发工具。
关于 Ant 这套工具的使用， O'Reilly 已经出版了 Ant – The Definitive Guide 一书专门讨论，
图 4: Ant – The Definitive Guide 一书

http://www.oreilly.com/catalog/anttdg/
如果您有是一位 Java 程式设计师，常常埋首于大量的 Java 程式码之中，本书非常值得成为您书桌上的常客。这本书请到 Ant 的原设计人 James Duncan Davidson 写推荐序，您可以在推荐序看到 James 述说 Ant 的发展历史。
为了让本章不至于只有讨论 Ant 的功能面(这些在 Ant – The Definitive Guide 这本书中都可以找到)，因此本文将专注于描述如何使用 Ant 来开发一个可以直接在 Windows 档案总管上双击就能执行的 Java 图形使用者介面应用程式，让大家充分体会 Ant 的优异功能，以及 Ant 为 Java 程式设计师所带来的便利性。就算本文所制作的目地产物非您所需，本文所叙述的内容，多少也可以让您体会如何在您的 Java 专案当中使用 Ant。

基础工具的安装

笔者假设您已经具备 Java 程式的撰写能力，您至少应该使用过 javac.exe 来编译 Java 原始档、使用 java.exe 来执行 Java 应用程式、以及使用 jar.exe 将许多类别档与资源档包裹成 JAR 档或 WAR 档的经验。本文假设您安装了 JDK 1.3.1 于 d:\jdk1.3.1 这个目录底下，因此，所有的基础开发工具都位于 d:\jdk1.3.1\bin 目录之下。

名称 所在目录
JDK 1.3.1 d:\jdk1.3.1
JDK 1.3.1 内附工具 D:\jdk1.3.1\bin

Ant 的下载与安装 6

请至 Ant 官方网站(http://jakarta.apache.org/ant/)下载最新版的 Ant，

图 5: Ant 官方网站

在撰写本章时，最新的版本是 Ant 1.5Beta 2，但是为了稳定起见，笔者选用最稳定的新版本 Ant 1.4.1。您可以在 http://jakarta.apache.org/builds/jakarta-ant/release/v1.4.1/bin/找到 1.4.1 版。

图 6: Ant 1.4.1 版

7

请下载档案 jakarta-ant-1.4.1-bin.zip。

完成下载之后，只要将此压缩档解压缩至磁碟机之中即可。本文假设您将 Ant 1.4.1 解压缩至 d:\ 底下，所以 Ant 1.4.1 的所在目录为 d:\jakarta-ant-1.4.1。

名称 所在目录
Ant 1.4.1 d:\jakarta-ant-1.4.1
Ant 1.4.1 的执行档 d:\jakarta-ant-1.4.1\bin

图 7: Ant 1.4.1 的目录结构

完成安装程序之后，接下来我们要做一些小调整，让我们可以方便地使用 Ant。

首先，请开启位于 d:\jakarta-ant-1.4.1\bin 底下的 ant.bat(在 Windows 底下都是用这个指令来启动 Ant)。在最开头的地方加入两行指令:

set ANT_HOME=d:\jakarta-ant-1.4.1
set JAVA_HOME=d:\jdk1.3.1

图 8: d:\jakarta-ant-1.4.1\bin\ant.bat

这是因为 Ant 本身需要用到很多 JDK 内附的开发工具，因此我们必须设定 8 JAVA_HOME 这个环境变数。 Ant 也需要用到一些自己目录下的资讯，所以我们必须设定环境变数 ANT_HOME。

最后，为了让我们更方便地使用 Ant，请设定您的 PATH 环境变数，让 Ant 安装目录下的 bin 子目录成为其中一员:

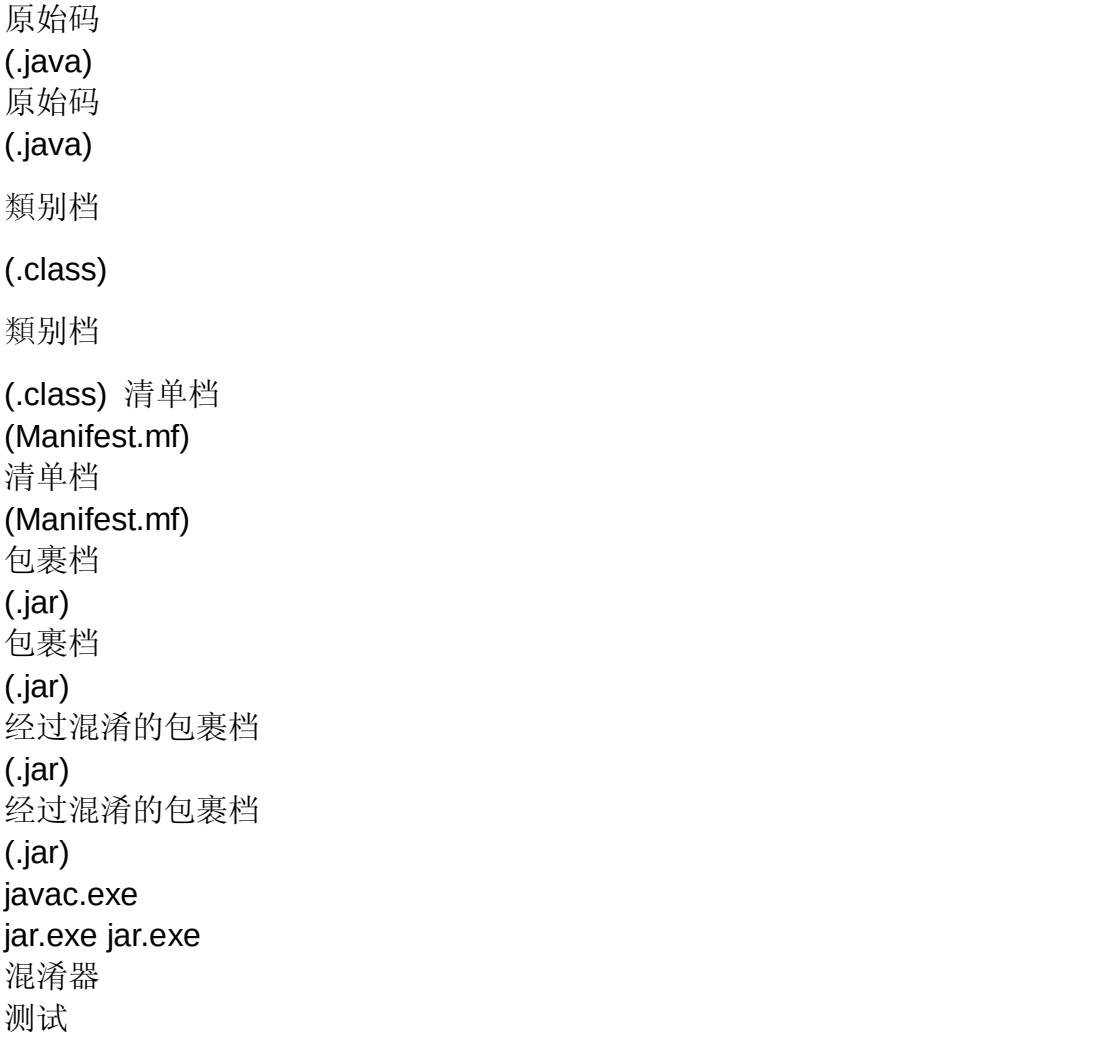
图 9: 环境变数设定

如此一来，当我们进入文字模式底下时，就可以在任何一个目录下执行 ant.bat。

Java 图形使用者介面应用程式开发流程

Java 图形使用者介面应用程式的开发流程如下图所示:

图 10: Java 图形使用者介面应用程式开发流程 9



上图所表示的是商用 Java 程式的开发流程，一开始我们必须撰写程式码(.java 档)，完成之后使用 Java 编译器(javac.exe)帮我们把原始程式转换成二进位的类别档(.class 档)。

为了使用者使用上的方便，我们习惯把二进位档及所有执行时所需的资源(图形、音效等)全部包装成单一一个 JAR 档。且为了让我们的图形使用者介面应

用程式可以直接在 Windows 档案总管上双击就能执行，我们必须撰写一个清单档(Manifest.mf)，清单档内含一些资讯，这些资讯可以引导 java.exe 自动启动我们的图形使用者介面应用程式。这里会利用 JDK 内附的 jar.exe 这个包装工具帮我们产生 JAR 档，并且将清单档塞入 jar 档内的特定位置中。

到了这里，其实已经完成一个的 Java 程式专案的所有开发程序了，可是，Java 的类别档具有容易反组译的性质，如果您不希望您的心血结晶被别人给偷

走，我们必须对写好的程式作混淆(obfuscate)的动作。经过混淆的 Java 程式，仍然可以正常执行，但是别人想要反组译就没那么容易了。混淆器还有个不错的副作用，就是经常会降低程式的大小，这是因为正常的类别档之中包含了太多名称资讯(函式名称、类别名称等)，经过混淆器混淆之后，由于混淆器通常会将我们原本的函式呼叫改成没有意义且比较短的名称，所以类别档经过混淆之后，几乎都会变小(仍有例外)。这对一般 Java 程式设计师来说，可能没太大意义，但如果您是一位撰写 Java 手机程式的程式设计师来说，尽力地缩小程式的大小，一定是您梦寐以求的功能。

完成混淆的工作之后，最后的包裹档(.jar)就可以拿来进行测试。

混淆器

在进入真正的 Java 程式专案开发之前，请容笔者在此向大家介绍几个笔者 10 曾经听过，或者常常接触到的混淆器，它们分别是:

名称 位址 性质

DashO http://www.preemptive.com/tools/clients.html 最贵，一般都是大公司采用。

ZKM http://www.zelix.com/ 可试用，最多同时混淆五个档案，商业版不便宜。

JBuilder http://www.borland.com 购买 JBuilder 时会内附

JAX http://www.alphaworks.ibm.com/tech/JAX/ 可试用，商业版价格中等

RetroGuard http://www.retrologic.com/ OpenSource

笔者没用过 DashO，所以无法对其作出个人评价。所以笔者仅用个人观点说明自己曾用过的产品。

如果就混淆的程度来说，ZKM 最好，JAX 中等，RetroGuard 最差，一分钱一分货，这是千古不变的道理。如果就功能性而言，ZKM 和 JAX 都不错，不过，JAX 是 IBM 所开发的产品，因此也继承了大部分 IBM 产品的最大特色，就是『功能超强，可是不易使用』，可能是笔者程度不好，光是看 JAX 的设定文件的第一页，头就痛了起来。不过，如果您勇于接受挑战，您可以在 Wireless Java 网站找到一篇介绍 JAX 混淆器与手机程式开发的文章『Obfuscating MIDP Applications with JAX』，请参考 http://wireless.java.sun.com/midp/ttips/jax/。而另外一篇类似的文章『Managing Wireless Builds with Ant』一文 (http://wireless.java.sun.com/midp/articles/ant/)，提及混淆器的时候，也是以 JAX 作例子。

图 11: JBuilder 内附的混淆器 11

但是就使用上的方便性来说，笔者还是最喜欢 RetroGuard，除了使用方便之外，最重要的是它是 Open Source 的软体，因此本文在开发流程中使用混淆器的部分，会以 RetroGuard 为例子说明。如果您对其他混淆器有兴趣，相信

读完本文之后，要拿其他产品替换 RetroGuard 不会有很大的问题才是。

下载及安装 RetroGuard

当您下载了 RetroGuard 并解开至磁碟机之后，其目录结构如下图所示:

图 12: RetroGuard 的目录结构 12

我们需要的只有 retroguard.jar 这个档案而已。 RetroGuard 的使用方法非常简单，如下图所示:

图 13: RetroGuard 的使用方式

RetroGuard 本身也是用 Java 撰写而成，所以启动时必须靠 java.exe 来启动。您只要将 classpath 选项指向 retroguard.jar，主类别名称 RetroGuard 即可。根据萤幕上的输出，我们可以知道，我们只要喂给 RetroGuard 一个 JAR 档，并指定输出档名，就可以完成混淆的工作。

底下笔者拿 MyGUI.jar 作为输入档当作例子。 MyGUI.jar 在混淆之前，内容如下:

图 14: 混淆前的 MyGUI.jar 13

于是我们使用底下指令进行混淆，

图 15: 混淆 MyGUI.jar 的指令

图 16: 混淆 MyGUI.jar 后所产生的 MyGUI-o.jar

从上面的测试我们可以看到，混淆器会帮我们变更类别档的档名与内部的函式名称等，这可以使得反编译之后的程式码不易阅读。 RetroGuard 在执行之后，会产生纪录档如下:

图 17: 混淆后 RetroGuard 所产生的纪录档 14

从上述输出您可以发现，RetroGuard 会混淆 JAR 档里头的每一个类别名称及内含的方法名称。但是，如果您基于某些特殊原因，使得您不希望 RetroGuard 混淆某类别的名称，那么我们必须藉由某种机制告诉 RetroGuard，不要对某些特定的类别进行混淆，以免影响程式的正常执行。以上面的例子来说，假设我们不希望 my.Main 及其内部属于 public 权限的方法被混淆，所以我们会撰写一个名为 script.rgs 的控制档，内容如下:

图 18: script.rgs

当我们写好控制档之后，重新执行指令:

图 19: 使用控制档来控制混淆过程所使用的指令 15

图 20: 经控制档控制之后的 MyGUI-o.jar

我们可以看出 my.Main 因为控制档的影响，所以其名称并没有被混淆。

关于控制档的内容以及 RetroGuard 更深入的使用方式，笔者不再详加说

明，请各位自行参考 RetroGuard 的说明文件。

双击后即可执行的 JAR 档

要怎样让合成后的 JAR 档可以双击之后就执行，其实设计了 Windows 内部的一些运作机制，我们可以用下图来表示:

图 21: 可双击后就执行的 JAR 档之运作机制

清单档
(Manifest.mf)
清单档
(Manifest.mf)
包裹档
(xyz.jar)
包裹档
(xyz.jar)
在档案总管上
双击 xyz.jar
取出 xyz.jar 内部的
meta-inf\Manifest.mf
取出 Manifest.mf 内部的
Main-Class: aa.bb 属性
Windows
Registry
Windows
Registry

转换指令

javaw –jar xyz.jar

转换指令

javaw –jar xyz.jar
查询 Registry 内部

对于 JAR 档的处理设定

转换指令

javaw –classpath xyz.jar aa.bb

转换指令

javaw –classpath xyz.jar aa.bb

从上图里可以发现，位于 meta-ing\Manifest.mf 内部的 Main-Class 属性对于

整个运作来说非常重要，不过，我们只要撰写好文字档，内含 Main-Class 属性，当我们使用 jar.exe 来合成最终的 JAR 档时，jar.exe 就会自动帮我们把这个文字档改名成 manifest.mf，并塞入 JAR 档内部的 meta-inf 目录之中。如果您想

在命令列模式下直接启动 JAR 档，则运作机制如下图所示:
图 22: 可在命令列用 java –jar 就执行的 JAR 档之运作机制
清单档
(Manifest.mf)
清单档
(Manifest.mf)
包裹档
(xyz.jar)
包裹档
(xyz.jar)

于命令列模式下输入

java –jar xyz.jar
取出 xyz.jar 内部的
meta-inf\Manifest.mf
取出 Manifest.mf 内部的
Main-Class: aa.bb 属性

转换指令为

Java –classpath xyz.jar aa.bb

转换指令为

Java –classpath xyz.jar aa.bb

您可以发现位于 meta-ing\Manifest.mf 内部的 Main-Class 属性一样扮演重要的角色。
在图 21 之中提到一个『查询 Registry 内部对于 JAR 的设定』的动作，这涉及 Windows 内部的运作方式。简单地说，您可以打开档案总管中的工具/资料夹选项，您可以看到底下视窗，请选择档案类型这个次页，并找到 JAR 档的设定: 17

选好 JAR 之后，您可以看到预设用来开启 JAR 的应用程式为 javaw.exe。请点选右下角的”进阶”，您会看到底下画面:

再按下右方的”编辑”，您可以看到 18

从这里您就可以发现，当您双击 JAR 档(xyz.jar)时，其实 Windows 内部帮您转换成指令:
C:\Program Files\JavaSoft\JRE\1.3.1_03\bin\javaw.exe -jar xyz.jar
这就是为何我们的 JAR 档能够自动执行的原因。好奇的您一定会问，那么 java.exe 和 javaw.exe 有什么不同。其实如果您观察两者的原始码，用的是一样的原始码，两者的差别在于 java.exe 的主程式进入点为 main()，而 javaw.exe 的主程式进入点为 winmain()，所以如果您撰写的是 Java 的视窗程式，最好使用 javaw.exe 来启动您的程式。如果改成使用 java.exe 也没太

大关系，但因为 java.exe 的主程式进入点是 main()，所以作业系统总是会帮您打开一个空白的 DOS 视窗，感觉上总是不太专业就是。但是如果要进行除错，改成使用 java.exe 倒是个不错的选择。

最后还要提醒您，java.,exe 与 javaw.exe 选用 JRE 的逻辑完全相同，而预设开启 JAR 档的 javaw.exe 位于 C:\Program Files\JavaSoft\JRE 底下，所以 javaw.exe 必定选用 C:\Program Files\JavaSoft\JRE 之下的那一套 JRE。因此别忘了第一章所提到的问题(类别函式库与安全设定的相关问题)。到此为止，我们已经具备了所有的基础技能，接下来将正式进入利用 Ant 来管理及开发 Java 程式的部分。

专案的目录结构

要使用 Ant 来开发和管理专案，最重要的就是必须先决定专案的目录结构。所谓的目录结构，就是所有专案相关档案的确切摆放位置，以及整个开发流程的产出物的摆放位置。有了明确的目录结构，才知道要如何设计 Ant 所使用建构档 build.xml，建构档的功能就是引导 Ant 帮我们建构整个专案。

就一个 Java 程式专案来说，我们希望所有的原始码都放置在 src 目录底下，而其他的资源档(如图片、文字档、声音档、影像档等)都放置在 res 目录之中。建构过程之中所产生的档案都放置在 build 目录底下，如果是类别档，就放在 build\classes 底下，最后产生的 JAR 档，就放置在 build\bin 目录之中。未精混淆器混淆的 JAR 档我们取名作”专案名称-unobfus.jar”(每个软体专案都会有不同的专案名称)，而经过混淆器混淆的 JAR 档，我们命名为” 专案名称.jar”。

最后，假设我们的软体专案包含了两个原始码档，一个是 Main.java，一个 19 是 MyHandler.java，而他们都属于 my 这个 package。

图 23: Main.java

图 24: MyHandler.java

我们也针对双击 JAR 即可执行的需求撰写了 Manifest.mf。

图 25: Manifest.mf

为了未来能够方便地使用混淆器，我们也把混淆器拷贝到专案的所在目录中，使用混淆器时必须要到的 script.rgs 也放置在专案所在目录里头。 20

图 26: script.rgs

因此最初的目录结构如下所示

图 27: 最初的专案目录结构

专案所在目錄 专案所在目錄

src 目錄 src 目錄

Manifest.mf Manifest.mf build.xml build.xml
build.properties build.properties

my 目錄 my 目錄

Main.java Main.java
MyHandler.java MyHandler.java
retroguard.jar retroguard.jar
script.rgs script.rgs

res 目錄 res 目錄

pic.jpg pic.jpg
21

我们希望可以让整个 Java 程式的建构序比较有弹性，将来不会因为某个目录名称或开发工具有所变动就必须进行修改，我以我们把一些常常会变动的参数部分独立放到 build.properties 这个档案之中，而建构的程序就放到 Ant 的标准建构档 build.xml 之中。然后会在 build.xml 引入 build.properties 的所有设定。
图:build.properties 22

图:build.xml 23

有了这些基础设定之后，我们就可以开始撰写整个程式建构程序。所有建构程序都完成之后，专案目录的结构将如下所示:
图 28: 完成整个建构程序之后的专案目录结构

专案所在目錄 专案所在目錄

src 目錄 src 目錄

res 目錄 res 目錄

Manifest.mf Manifest.mf build.xml build.xml
build.properties build.properties

my 目錄 my 目錄

Main.java Main.java
MyHandler.java MyHandler.java
retroguard.jar retroguard.jar
script.rgs script.rgs

build 目錄 build 目錄

bin 目錄 bin 目錄

未混淆的 jar 档 未混淆的 jar 档

classes 目錄 classes 目錄

已混淆的 jar 档 已混淆的 jar 档

设定参数档(build.properties) 与建构档 (build.xml)

底下我们将采取渐进的方式，实作建构 Java 程式的每个阶段。 24

编译时的 build.properties 与 build.xml

Java 程式建构程序的第一个步骤，就是编译，所以我们修改 build.xml 如下。

图:build.xml

完成修改之后，在目录下输入 ant compile 即可开始进行编译，由于我们在<project>组件中设定 default=”compile”，所以直接输入 ant 也可以，ant 会自动找到<target name=”compile”>组件，并依照里面的设定工作。

图:执行结果

注意! 如果您使用的 JDK 为 1.4.x 以上的版本，那么出现一些小问题，这是因为 RetroGuard 无法正确处理 JDK 1.4.x 以上版本所产生的类别档。为了解决这个问题，我们必须强制编译器产生旧版的类别档，所以必须用到 java 编译器的 target 选项。

请将 build.xml 修改如下: 25

包装时的 build.properties 与 build.xml

Java 程式建构程序的第二个步骤是将类别档与资源档包装在一个 JAR 档之中，所以我们修改 build.xml 如下:

图:build.xml

26

完成修改之后，在目录下输入 ant package 即可开始进行打包成 JAR 的工作，由于我们在<project>组件中设定 default=”package”，所以直接输入 ant 也可以，ant 会自动找到<target name=”package”>组件，并依照里面的设定工作。

图:执行结果

27

混淆时的 build.properties 与 build.xml

Java 程式建构程序的第三个步骤是混淆，所以我们修改 build.xml 如下。

图:build.xml

完成修改之后，在目录下输入 ant obfuscate 即可开始进行混淆的工作，由于我们在<project>组件中设定 default=”obfuscate”，所以直接输入 ant 也可以，ant 会自动找到<target name=”obfuscate”>组件，并依照里面的设定工作。

图:执行结果 28

一次搞定

所有的步骤要一一输入指令仍嫌太麻烦，我们希望 Ant 帮我们把所有建构步骤连同唤起模拟器一次搞定，所以请修改 build.xml 如下:

图:build.xml 29

完成之后，以后我们只要在目录下输入 ant 或 ant all 就可以轻松让 Ant 帮我们自动清除前一次的建构结果，然后从头到尾再帮我们建构整个专案。

图:执行结果 30

最后，请直接使用档案总管，在 MyGUI.jar 上双击滑鼠左键，就可以看到程式成功地执行了:

结论

在本文中，笔者为大家介绍了 Ant 这套轻便的开发工具，并将 Ant 导入 Java 31 图形使用者介面应用程式设计的流程之中，希望撰写 Java 应用程式的人可以更加轻松，把精力专注在程式本身，而非杂七杂八的建构工作。如果您从事其他类型 Java 程式的开发工作(Servlet、EJB 等)，那么相信藉由本文的介绍，能够大幅降低您的工作量。用过 Ant 的人都会大大赞赏 Ant 所带来的便利性，您说是吗?

附录 A

Java 2 SDK 版原始码概观

简介

据说资讯社会的演进有个 lifecycle，从早期每个人的办公桌上只有一部终端机的时代慢慢进步到 PC 时代，然后我们发现逐渐地潮流又走回 NC(Network Computer)，厂商鼓吹 thin client 的时代。对于原始码的看法亦是如此，从早期厂商在出售的机器内附上程式原始码供顾客自行修改，然后渐渐地程式码变成厂商独有而不愿公开的资产，最后又演变成大家鼓吹 Open Source、Free Software 的时代。姑且不论普罗大众是否真的有兴趣去 trace 附在您电脑之中的原始码，但是我想这就是开放原始法最可贵的地方– 大家都有机会一窥系统的内部行为，自己电脑上资料的安全以及程式的运作方式再也不会掌控在特定厂商的手上，有了原始码，我们在也不用怀疑是否某个厂商的作业系统或是应用程式会偷偷地在背后将您的资料回传到他们公司的伺服器中。

随着这股开放原始码的潮流，Java2 SDK 的原始码其实已经 release 出来很长一段时间了，但是采用的 License 为 Sun Community Source Code License(SCSL)，与一般我们讨论到 Open Source 时所会提到的 General Public License(GPL)有所不同。相信最近大家会发现，在相关文章中也开始提及将 Java SDK 改用 General Public License 释出的议题。当 2000 年 10 月 Sun 将 Star Office 的原始码以 General Public License 释出之后，接下来大家所期待的就是 Sun 将 Java SDK 的原始码以 General Public License 释出，如果此事成真，那么 Java 今后的发展将不是我们目前所能预测的，如果成功，那么 Java 会像 Linux 一样展翅高飞，如果没有成功，那么将可能像 Mozilla 计划一般雷声大雨点小。

先姑且不论哪个 License 比较 open，哪个比较 free，笔者撰写此系列文章的目的只有一个，就是挖掘出存在于 Java2 SDK 内部的运作机制。如果您希望了解一些我们常用的执行档(exe，例如 javac.exe、java.exe 等)、几个重要的动态连结函式库(dll，例如 jvm.dll、java.dll 等)，以及其他使用 Java 程式语言所撰写的套件(package，例如 java.io、java.util 等)，最终都需要从原始码之中一探究竟。

当然，trace 别人的程式码这种事情乍看之下有点像是程式黑手在做的事情，但是笔者深深觉得，我们的软体工业要站上国际舞台仍然需要很长一段时间。从大观点来看，系统分析、系统设计、系统测试、如何妥善地分配有限人力这些议题上，至少我们还没有办法举出哪些国内公司将这些工作做的十分出色，从 Java 2 SDK 的原始码结构中，我们可以看到开发一套 SDK(Software Development Kit)时采用的团队组织架构之缩影，原始码的位置怎么放? 那么多程式设计师写出来的程式如何整合在一起发挥纵效? 如何让文件的产生自动化? 我想我们一定可以从 Java 2 SDK 的原始码得到一些答案。从小观点来看，如何善用 Design Pattern? 如何写出有效率的程式码? 如何让程式设计师写出来的程式码具有一致性? 我们都知道 Java 内部的套件大量地使用了 Design Pattern，我们也知道 Java 因为虚拟机器的关系，所以程式码必须非常地有效率，相信这些问题也能够从 Java 2 SDK 的原始码获得答案。

大力推展 Open Source 的开发中国家也非常清楚，开放原始码将是开发中国家赶上已开发国家软体工业的终南捷径。期待个人的努力可以成为这个大革命中一颗不可或缺小螺丝钉。
在本附录中，笔者将为大家介绍 Java 2 SDK 原始码的轮廓，让每位读者都具有取得 Java 2 SDK 原始码并能够成功编译这些原始码的基本能力。废话不多说，我们就开始啰。 。

如何取得 Java2 SDK 原始码

要取得 Java2 SDK 的原始码，首先您必须要是 Java Developer Connection(JDC) 的一员，所以请您先到 http://developer.java.sun.com/developer/网站上注册，取得使用者 ID 和密码。如下图所示:

当您成功成为 Java Developer Connection(JDC)的一员之后，请您到 http://www.sun.com/software/communitysource/java2/，您将会在右手边看到下载 Java 2 SDK 原始码的选项:

现在提供 Java 2 SDK 1.2.2、1.3、1.3.1、以及 1.4.0 版本原始码的下载。请点选右边红色的”Download”。

请点选您希望下载的版本，并点选”Download”。您可以发现，在 Java 2 SDK 1.3.1 以上的版本，都已经可以直接从网路上观看原始码的编译方式，而之前的版本则必须等到下载之后，才能找到如何编译原始码。
假设您选择下载 1.3.0，则接下来的画面如下:

请在左方输入您在 Java Developer Connection(JDC)的使用者 ID 和密码，接下来则出现版权宣告画面:
如果您接受版权宣告的内容，请在 Accept 处选择 Yes，然后按下 Continue，就开始选择 Java 2 SDK 原始码的动作。如下图所示:

请选择您希望下载的版本，就开始下载程序。
如果您选择下载 Java 2 SDK 1.4.0 的原始码，则程序上稍有不同。由 Java 2 SDK 1.4.0 内附可以支援 Netscape 或 Mozilla 浏览器的 Plug-in，所以编译时必须有 Mozilla 的相关档案才能完整编译。因此，请您除了下载 Java 2 SDK 1.4.0 的原
始码之外，还必须额外下载 Mozilla Binaries。

底下我们同时列出两个下载的画面:

如上图所示，萤幕上会出现类似购物车的画面，不过请放心，不需要付钱☺。请点选右下角的”Check Out”即可。接下来一样会出现版权宣告画面，请一样在 Accept 处选择 Yes，然后按下”Continue”即可。
最后，会出现类似网路购物结帐的画面，按下右下角的”Place Order”就可以继续下载动作。

最后请选择您欲下载的档案:
如果您只需要 Java 2 SDK 1.4.0 的原始码，那么只要下载 j2sdk-1_4_0-src-scsl.zip 即可，而 j2sdk-sec-1_4_0-src-scsl.zip 是 Java Cryptography Extension 的原始码，不下载也不会影响编译程序。

Java 2 SDK 1.3.0 原始码的架构

当您解开 Java 2 SDK 1.3.0 版原始码的压缩档之后，您会看到如下图所示的目录结构:

◇ build 目录

放置了所有的 makefile 档，这些 makefile 为 Visual Studio 内附的 nmake.exe 所能解读的格式。通常 makefile 的档名有五种，分别是 Makefile、*.nmk、*.jmk(指定要编译的.java 档)、*.cmk(指定要编译的 .c 档)、*.mk。在 build\share 底下您还可以找到一些.java 档。

由于我们要编译的是 Windows 作业系统下的 Java SDK，所以 Makefile 的主档为 build\win32\Makefile，整个 Java 2 SDK 的编译流程都是根源于这个档案，在 build\win32\makefiles 目录之下您可以找到共用的 Makefile，通常附档名为.nmk。

build\win32 目录下的 makefile 都是用来建造和 Windows 作业系统相关的程式。而在 build\share 目录底下就多为.mk、.jmk、.cmk 档，这些档案通常用来建造和 Windows 作业系统无关的程式。

◇ ext 目录

这个目录底下放置了用来支援 Java 2 SDK 函式库的 makefile 与原始码，这些原始码皆使用 Java 程式语言所撰写。

ext\i18n 目录下放置了与国际化(internationalization)相关的函式库，这些编译过的档案最后都被放到 i18n.jar 里头。

ext\jpda 目录下放置了与 Java 平台除错器架构(Java Platform Debugger Architecture)相关的档案。

◇ src 目录

这个目录放置了所有建构 Java SDK 所需要用到的.c 档(以 C 程式语言撰写)以及.java 档(以 Java 程式语言撰写)。

jdk1.3-src
build
  ext
  src
LegalReadme.html
OriginalCode.txt
README.html
share
win32
share
win32

I18n
jpda
build
 src
build
 src src\win32 目录下都是用来建造和 Windows 作业系统相关的原始码。而在 src\share 目录底下就多为与 Windows 作业系统无关的原始码。因此聪明的读者们应该可以猜到大部分 Java 标准套件的原始码大多放在 src\share 目录之下。这些标准套件的原始码在最后都被加入到 src.jar 之中，而编译过的.class 档则被打散放入许多不同的 jar 档内。

◇ LegalReadme.html

Sun Community Source Code License 版权宣告

◇ OriginalCode.txt

Java2 SDK 原始码中所有档案的列表。

◇ README.html

关于 Java 2 SDK 原始码的说明，该内容大部分都已经被笔者吸收而整合到这篇附录之中。

从 Java 2 SDK 原始码的架构中，我们可以发现其档案的放置方法很独特，是采用 makefile 与原始码分开放置的做法，不同于一般我们所见(例如 Linux kernel 原始码)采 makefile 和原始码放在一起的做法。这两种做法各有利弊。以 Linux Kernel 原始码的做法，每次编译完成之后，您将看到原始码档案和编译过程中产生的中间档混杂在一起，感觉有点混乱。如果我们采用 Java 2 SDK 原始码的目录结构，每次编译出来的中间档和原始码就会分隔在不同的目录下，看起就非常赏心悦目，但是，这样放置的代价就是 makefile 里头对于路径的参照变得十分复杂，您将会在 Java 2 SDK 原始码的 makefile 中发现很多..\..\..\src 的叙述，看起来真是不舒服，如果我们要了解该执行档是由哪些档案构成的时候就很辛苦了。

Java 2 SDK 1.4.0 原始码的架构

当您解开 Java 2 SDK 1.4.0 原始码的压缩档之后，您会看到如下图所示的目录结构，与 1.3.0 版其实有蛮大的不同:

jdk1.4.0-src
 control
  hotspot
   j2se
LegalReadme.html
OriginalCode.txt
README.html
build.html
build-xxx.html
make
build
src

◇ control 目录

放置了建够整个 Java 2 SDK 里面每一个组成成分所共享的 makefile 档，而且附档名都是.gmk，所以这些共享的 makefile 与特定平台无关。而其他的 makefile 则藏身在 hotspot\build 目录底下，这个目录底下分别有编译 linux、solaris 以及 Windows 平台上之 Java 虚拟机器所需的 makefile。在 j2se\make 目录底下也放置着许多 makefile，这些 makefile 都是用来建够 Java 2 SDK 内的每一个组成分子，比方说 Java 类别函式库、Java 2 SDK 内附的开发工具等等。

◇ hotspot 目录

内含 Hotspot Client VM 与 Hotspot Server VM 的原始码以及 makefile。

◇ j2se 目录

内含 Java 2 SDK 内的每一个组成分子的原始码与 makefile。比方说 Java 类别函式库、Java 2 SDK 内附的开发工具等等。

◇ LegalReadme.html

Sun Community Source Code License 版权宣告

◇ OriginalCode.txt

Java2 SDK 原始码中所有档案的列表。

◇ README.html

关于 Java 2 SDK 原始码的说明，该内容大部分都已经被笔者吸收而整合到这篇附录之中。

◇ build.html 、 build-linux.html 、 build-solaris.html 、以及 build-win32.html

说明如何编译各个平台上 Java 2 SDK 的文件。

从上述说明我们可以看出，Java 2 SDK 1.4.0 的原始码结构比起 Java 2 SDK 1.3.0 的原始码结构要更清楚，要深入每一个原始码也因此更方便。底下我们以 Java 2 SDK 1.3.0 的原始码为例，说明如何完成其编译程序。如果您想编译 Java 2 SDK 1.4.0 版，请多参考 build-win32.html，里面对于编译程序讲的十分清楚。

如何编译 Java 2 SDK 1.3.0 原始码

当我们深入了解 Java 2 SDK 原始码之后，很多工程师必定想自己尝试修改原始码，然后进行编译，我想这是身为一个工程师最大的乐趣。当然，在我们还没有修改之前，我们也必须试着编译整个原始码，顺便测试一下原始码是否完整，因此接下来要介绍编译 Java 2 原始码的准备步骤和方法。

要顺利编译 Java2 SDK 原始码，您必须符合底下几种条件:

(1) 作业系统您必须在 Windows NT 4.0 或是 Windows 2000 上才能顺利编译 Java 2 原始码。一旦编译完成之后，所产生的执行档或是动态连结和式库将能够在所有的 Win32 作业平台上执行。

(2) 硬体设备

在 Java2 SDK 原始码内附的文件之中，建议的硬体设备为 Pentium 等级的处理器，配上至少 128 MB 的记忆体。在笔者的电脑上，Pentium III 450 配上 PC 100 128 MB 的记忆体，总共花了将近 2.5 个小时完成所有编译动作。如果算上编译时期遇到的一些错误以及恢复错误的时间，大概花了 3 个小时。

(3) 开发工具

我们必须准备几项工具，他们分别是:

z Microsoft Visual C++ version 6.0，并配上最新的 Service pack(或者 Visual Studio 97 配上 Service Pack 3)。

在 Java2 SDK 原始码之中，存在有许多与 Windows 平台相关的程式码，这些程式码都以 C 语言撰写而成，为了顺利编译这些原生程式码，我们必须要借助 Visual C++编译器以便产生能在 Windows 平台上执行的执行档(.exe)以及动态连结和式库(.dll)。

同时，Java2 SDK 原始码之中有许多 Makefile，这些 Makefile 是用来控制所有程式的编译动作。我们必须借助 Visual C++内附的 nmake.exe 这个工具以帮助我们解读这些 Makefile 并顺利编译出整个 Java2 SDK。

z MKS Toolkit version 5.2 以上的版本。

MKS Toolkit 是一套能够在 Windows 上模拟 UNIX 执行环境的工具，请到 http://www.mks.com 下载这套工具的 30 天试用版(正式版必须要付费)。或者如果您有订阅 MDSN Profession 以上的版本，您可以在光碟中找到 Microsoft Windows Services for UNIX 2.0 Add-On Pack(在 UNIX20_INSERV\3RDPARTY\MKS 目录底下，执行 x86 子目录中的 setup.exe 即可安装)。 30 天试用版的功能骑士和付费版本的功能一样，只不过当您在编译 Java 2 SDK 原始码的时候，只要用到了 MKS Toolkit 里的工具，就会出现一个广告画面罢了，应该不会影响您的心情才是。

z Java 2 SDK 1.3 的可执行版本。

相信读者们看到这个地方，一定会觉得很疑惑。奇怪，编译 Java 2 SDK 的原始码竟然也需要 Java 2 SDK 的协助? 不禁让人想起鸡生蛋蛋生鸡的问题，不是吗? 其实 Java 2 SDK 在整个编译过程中，扮演的是 bootstrap compiler(靴带式编译器)的角色。之所以需要 Java 2 SDK 的原因是因为大家在编译 Java 程式时所使用的 javac.exe 核心程式竟然也是用 Java 撰写而成的，由于 Java 2 SDK 的原始码有许多与平台无关、以 Java 程式语言所撰写的程式，因此在编译初期，我们必须借助过去旧版的 Java 2 SDK 或是 JDK 帮助我们产生新版的 Java 2 SDK，再利用新产生的新版 javac.exe 编译 Java 2 SDK 原始码内的其他.java 档。

当大家将所有必备的条件都准备妥当之后，接着我们要开始准备编译前的相关工作。底下的内容将有下列假设:

工具 安装路径

Windows NT/2000 系统所在目录 c:\winnt

Microsoft Visual C++ version 6.0 d:\Microsoft Visual Studio

Microsoft Visual Studio 97 d:\Program Files\DevStudio

MKS Toolkit c:\mskdemo\mksnt

Java2 SDK 1.3 的可执行版本 d:\jdk1.3

Java2 SDK 原始码 d:\jdk1.3-src

接下来我们要设定环境变数，为了方便起见，请大家新增一个叫做 env.bat 的批次档，这样以后要修改环境变数的时候就不必大费周章:
如果您使用的 C 编译器为 Microsoft Visual C++ version 6.0，那么 env.bat 的内容如下:

```
env.bat
set MSVCDir=d:\Microsoft Visual Studio\vc98
set MSDevDir=d:\Microsoft Visual Studio\Common\MSDev98
set WinDir=c:\winnt
set
include=%MSVCDir%\include;%MSVCDir%\mfc\include;%MSVCDir%\atl\inclu
de
set lib=%MSVCDir%\lib;%MSVCDir%\mfc\lib
set
PATH=c:\mskdemo\mksnt;%MSVCDir%\bin;%MSDevDir%\bin;%WindowsDir
%\system3
2;%PATH%
set ALT_BOOTDIR=d:\jdk1.3
set TMPDIR=d:\jdk1.3-src\tmp
```

如果您使用的 C 编译器为 Microsoft Visual Studio 97，那么 env.bat 的内容如下:

```
env.bat
set MSVCDir= d:\Program Files\DevStudio\VC
set MSDevDir= d:\Program Files\DevStudio\SharedIDE
set WinDir=c:\winnt
set
include=%MSVCDir%\include;%MSVCDir%\mfc\include;%MSVCDir%\atl\inclu
de set lib=%MSVCDir%\lib;%MSVCDir%\mfc\lib
set
PATH=c:\mskdemo\mksnt;%MSVCDir%\bin;%MSDevDir%\bin;%WindowsDir
%\system3
2;%PATH%
set ALT_BOOTDIR=d:\jdk1.3
set TMPDIR=d:\jdk1.3-src\tmp
```

任何一个环境变数设定有误，都会造成编译错误，请读者们务必小心。关于环境变数，底下有几件事情请读者一定要注意:

1. 对于 PATH 环境变数来说，MKS Toolkit 的路径一定要出现在 Microsoft Visual C++ version 6.0(或是 Microsoft Visual Studio 97)的所在路径之前，否则会造成编译错误。所以请不要将 PATH 设定为:

```
set
PATH=%MSVCDir%\bin;%MSDevDir%\bin;c:\mskdemo\mksnt;%WindowsDir
%\system3
2;%PATH%
```

(MKS Toolkit 出现在 Microsoft Visual C++ version 6.0(或是 Microsoft Visual Studio 97)的所在路径之后)
2. 请关闭 CLASSPATH 环境变数。如果您是在 Windows 2000 底下，请在”我的电脑”图示上按下滑鼠右键，选择”内容”，点选”进阶”次页，再按下”环境变数”按钮。将出现环境变数对话方块，如下图所示:

请寻找使用者变数中或是系统变数中任何名为 CLASSPATH 的环境变数删除，否则在编译 Java 2 SDK 原始码初期就会出现错误讯息。
3. 环境变数 ALT_BOOTDIR 是用来设定 bootstrap compiler(靴带式编译器)的所在位置，您只要设定为 Java SDK 的安装位置就可以了。

当上述准备工作全部完成之后，请开启”命令提示字元”视窗(就是 DOS 视窗啦!)，并将目录切换到 d:\jdk1.3-src\build\win32 底下，在命列上输入
nmake world
此时编译 Java2 SDK 原始码的过程正式展开，毫无疑问地，这将是一个漫长的过程。
在编译的过程之中，在笔者的电脑上出现了一些编译错误，在这里顺便向各位提及，顺便说明笔者的解决办法:
z 找不到 .properties 档:
在编译 Java2 SDK 原始码的时候常常出现找不到.properties 档的错误讯息，如果发现编译终止的原因是因为在个特定目录底下找不到某个.properties 档，那么请读者自行搜寻 jdk1.3-src\src 目录底下，一定能找到编译器所找不到的.properties 档，请将这个.properties 档复制到编译器期望该档案所在的目录底下即可解决。
z 找不到 .class 档:
在编译 Java2 SDK 原始码初期的时候常常出现找不到.class 档的错误讯息，笔者发现这些找不到的类别档都是放置在 jdk1.3-src\ext\i18n\src\share\sun\io\底下，之所以会找不到这些.class 档的缘故是因为根据 Makefile 里头的内容，jdk1.3-src\ext 目录里的原始码会较晚编译，而我们又关掉了 CLASSPATH 环境变数，导致在需要这些档案的时候，这些档案尚未从.java 档编译成.class 档，很自然地编译器就无法编译。要解决这个问题，请读者们先行到 jdk1.3-src\ext\i18n\src\share\sun\io\底下，输入 javac *.java 以产生所有需要的.class 档，然后将这些新产生的.class 档复制到 jdk1.3-src\build\win32\classes\sun\io 这个目录底下。这样一来，就再也不会遇到找不到.class 档的错误讯息了。
z Java2 SDK 原始码的内附文件中要我们设定环境变数 TMP，但是实际的测试结果，发现在这个路径似乎在编译过程中一点用处也没有，反倒是编译的时候有用到环境变数 TEMP 所指向的目录(在 Windows 2000 底下这个环境变数被设定为 %USERPROFILE%\Local Settings\Temp)，不知道是不是写说明档的人不小心漏打字。

当编译工作告一段落之后，您将可以在 jdk1.3-src\build\win32 目录下找到所有编译过的执行档以及.class 档，还有一些编译过程中产生的中间档。jdk1.3-src\build\win32\bin 目录下，您将会看到所有让 Java2 SDK 正常运作的执行档以及动态联结函式库。又由于我们是使用 make world 指令，如果您观察一下这个目录，您会发现每个档案都有最佳化版本与除错版本(ex: java.exe 与 java_g.exe，其他以此类推)。
jdk1.3-src\build\win32\lib 目录下将放置所有让 Java 程式能够顺利执行的套件函式库，例如 i18n.jar 等，在 jdk1.3-src\build\win32\lib\ext 目录下可以找到 iiimp.jar 等扩充套件还有一些资源档。
jdk1.3-src\build\win32\classes 目录下放置了最后会压缩成 rt.jar 的所有 .class 档(类别档)。

另外，除了输入
nmake world
指令之外，Makefile 之中定义了其他非 world 的编译选项，请参阅下表:
编译指令 作用
nmake all
建立 Java SDK 核心程式的除错版本以及最佳化版本
nmake ext 建立 Java SDK 扩充程式的除错版本以及最佳化版本
nmake world
(同 nmake world-all)
建立 all 与 ext 中所有的程式
nmake releasedocs 执行 javadoc.exe 以产生 Java SDK 说明文件
nmake alldocs 执行 javadoc.exe 以产生在 sun.*套件之下程式的所有说明文件
Nmake
world-clobber
删除前一次编译之后产生的所有档案和目录
如果您想要了解所有的选项，请您自行开启 jdk1.3-src\build\win32\Makefile 或是 jdk1.3-src\build\win32\makefiles\defs.nmk 这两个档案做更深入的研究。

接下来

有关 Java 2 SDK 原始码的概观介绍就到这里，相信大家都已经有了坚实的基础，如果您已经是一位非常有经验的工程师，相信您已经能够随意地修改出自己版本的 Java2 SDK 了!

网路资源

名称 URL
Java Developer
Connection
http://developer.java.sun.com/developer/
Java2 原始码下载 http://www.sun.com/software/communitysource/java2/