

Java 安全性编程实例

徐迎晓 编 著

清华大学出版社

内 容 提 要

本书以大量精简的实例介绍了 Java 安全性编程方面的概念和技术，全书共计 9 章。

经过精心设计，每个小节的实例着重说明一个问题，又相互贯穿和联系。涉及 Java 加密和解密，反编译和反反编译，对类、成员变量、方法的攻击和保护，消息摘要，消息验证码，数字签名，口令保护，数字证书和证书链的生成、签发、检验和维护，SSL 和 HTTPS 客户及服务器程序、基于代码位置和签发者的授权，签名 Java Applet，基于身份的验证和授权（JAAS）等。

全书的实例极为精简，只保留了能够说明问题的代码，而又可真正运行，便于 Java 入门者轻松掌握安全性方面繁杂的概念。适合于适合于初步了解 Java 语法的学习者，也适合于安全技术的入门及高校 Java 教学参考。

。

前言

随着 Internet 的发展，安全性已经引起越来越大的关注。Java 自其诞生起就将安全性作为主要考虑因素之一，随着 Java 的发展，更多的安全机制加入到 Java 中，在 Java 2 SDK 1.4 中更是集成了 JCE、JSSE、JAAS 等 Java 安全扩展平台。这些安全机制是开发基于企业级 Java 2 应用平台（J2EE）上安全的应用程序的基础。

很多安全性方面的书籍涉及大量难以理解的理论和概念，而缺乏浅显易懂的实例，使初次进入安全领域的学习者望而生畏。本书以实例为基础，从实例导入基本概念和理论，引导读者逐步进入安全领域。

本书以功能和实例为导向，每个小节一个实例完成一个小的功能和知识点。各个实例经过精心简化，只保留最关键部分，因此本书的程序往往数行便可实现关键功能。读者可以方便地进行扩充和加上自己的代码。

各个小节的实例像一块块积木，既有独立性又相互间关联，读者可以方便地利用这一块块“积木”搭建出大型的应用。

本书共分 9 章，主要内容如下：

第一章

解决的主要问题

运行本书的程序需要哪些软件？

主要内容

介绍本书所使用的主要软件及其安装和配置

第二章

解决的主要问题——内容的安全性

数据在网上传递怎么样防止被黑客窃听到？

硬盘上的文件中有敏感数据，如何防止被黑客看到？

主要内容

本章解决的是数据内容的安全性，介绍 Java 的加密和解密技术。学完该章可以通过 Java 编程对各种数据进行各种形式的加密。密码学也是安全机制的基础。

第三章

解决的主要问题——和源代码相关的安全性

编写好的程序给用户后，用户如果能反编译出源代码怎么办？

定义类、成员变量、方法时如何防止恶意或无意的攻击？

主要内容

本章解决的是和源代码相关的保护。包括源代码、类、成员变量、方法的保护。通过常用的反编译工具加强对源代码保护的认识，使用混淆器和加密等方式对源代码作了初步保护。同时演示了编写程序时如何考虑攻击者对类、成员变量、方法等方面的攻击。

第四章

解决的主要问题——确定数据的完整性和所有者

网上下载了一个程序，如何确定它确实是某某公司开发的？

如何确定黑客没有将程序修改过？

某公司或人发来一个文件，后来他不承认发过这个文件怎么办？

主要内容

第四章起开始介绍和身份认证相关的技术。包括身份确定性、不可篡改性、不可否认性等，该章介绍的消息摘要和签名技术可解决这些问题。

第五章、第六章

解决的主要问题——数字化身份的凭证

实际应用中如何方便地使用摘要和签名技术？

如何确定某个签名确实是某个人或机构的？

主要内容

第五章和第六章介绍基于摘要和签名技术的数字证书。这是 Java 安全中确定身份的主要技术。其中第五章介绍了数字证书的创建、签发、验证和维护等，第六章介绍了多个证书组成的证书链（CertPath）的创建和验证。

第七章

解决的主要问题——数据安全传输，服务器和用户身份的确定

客户机和服务器之间的通信如何自动进行加密传输？

客户机和服务器之间的通信如何相互确定身份？

浏览器访问一个站点，如何确定这个站点不是黑客的服务器？

主要内容

本章介绍使用加密技术和证书机制的一个实际应用，基于 SSL 和 HTTPS 的编程。学完本章可以编写自己的 SSL 和 HTTPS 客户及服务器程序。

第八章

解决的主要问题——基于代码来源的程序的安全运行

网上下载了一个程序，运行时会不会删除我的文件，或将某些文件泄漏给黑客？

编写了一个 Java Applet，如何让其能访问硬盘上的文件？

主要内容

本章介绍基于代码来源的程序的安全运行，可以基于运行时代码在哪个 URL、或代码是谁签名的限制其可以访问哪些用户资源。还介绍了定义自己的权限以及签名 Java Applet。

第九章

解决的主要问题——身份验证和基于执行者身份的程序的安全运行

程序需要用户输入账号和口令到数据库登录，但以后可能需要改为智能卡验证。

程序需要访问某个用户资源，但只有用户以某些特殊身份登录时才需要该权限。

主要内容

本章介绍 Java 验证和授权服务（JAAS），可以方便地更换验证模块，并实现基于身份的授权。

本书实例以帮助读者入门为目的，因此为了易于理解作了很多简化，如很多实例中口令以字符串保存，各个实例的异常处理都作了简化。因此，在编写实际应用系统时不可照搬。

本书作者是 SUN 认证讲师，多年从事 Java 的培训与研究。对于本书的各种意见和建议请 E-mail 至 xyx@shu.edu.cn，作者提供网站 <http://javabook.126.com> 供读者交流。

徐迎晓

二〇〇三年一月于

上海大学

目 录

第 1 章运行环境设置.....	10
1.1 J2SE的安装和设置	10
1.1.1 下载J2SE	10
1.1.2 安装J2SE	10
1.1.3 设置J2SE	12
2.1.4 J2SE的主要工具	13
1.2 反编译器的安装.....	14
1.3 混淆器的安装.....	16
第 2 章数据内容的保护—— 加密和解密	17
2.1 一个简单的加密和解密程序—— 凯撒密码	17
2.2 对称密钥的生成和保存.....	19
2.2.1 对称密钥的生成及以对象序列化方式保存	19
2.2.2 以字节保存对称密钥	20
2.3 使用对称密钥进行加密和解密	22
2.3.1 使用对称密钥进行加密	22
2.3.2 使用对称密钥进行解密	24
2.4 基于口令的加密和解密	26
2.4.1 基于口令的加密	27
2.4.2 基于口令的解密	29
2.5 针对流的加密和解密	31
2.5.1 针对输入流的解密和解密	32
2.5.2 针对输出流的解密和解密	34
2.6 加密方式的设定.....	36
2.6.1 使用CBC方式的加密	36
2.6.2 使用CBC方式的解密	38
2.7 生成非对称加密的公钥和私钥	40
2.8 使用RSA算法进行加密和解密	42
2.8.1 使用RSA公钥进行加密	42
2.8.2 使用RSA私钥进行解密	44
2.9 使用密钥协定创建共享密钥	47
2.9.1 创建DH公钥和私钥	47
2.9.2 创建共享密钥	50
第 3 章Java源代码和类、变量 及方法的保护	53
3.1 Java反编译及混淆器的使用	53
3.2 从网络资源加载字节码文件	57
3.3 以任意方式加载字节码文件	61
3.4 加载加密的字节码文件	63
3.5 加载当前目录下的加密字节码文件	66
3.6 Java类、成员变量和方法的保护	69
3.6.1 类的保护	69
3.6.2 成员变量和方法的保护	73
3.6.3 使用校验器	74

3.6.4 Reference类型私有成员变量的保护	76
3.6.5 保护常量	79
第4章 数据完整性和所有者的确认——消息摘要和签名	82
4.1 使用消息摘要验证数据未被篡改	82
4.1.1 计算消息摘要	83
4.1.2 基于输入流的消息摘要	84
4.1.3 输入流中指定内容的消息摘要	86
4.1.4 基于输入流的消息摘要	88
4.2 使用消息验证码	90
4.3 使用数字签名确定数据的来源	92
4.3.1 使用私钥进行数字签名	93
4.3.2 使用公钥验证数字签名	95
4.4 使用消息摘要保存口令	97
4.4.1 使用消息摘要保存口令	98
4.4.2 使用消息摘要验证口令	99
4.4.3 攻击消息摘要保存的口令	101
4.4.4 使用加盐技术防范字典式攻击	104
4.4.5 验证加盐的口令	107
第5章 数字化身份的确定——数字证书	111
5.1 数字证书的创建	111
5.1.1 使用默认的密钥库和算法创建数字证书	111
5.1.2 使用别名	113
5.1.3 使用指定的算法和密钥库和有效期	114
5.1.4 使用非交互模式	115
5.2 数字证书的显示	116
5.2.1 使用Keytool直接从密钥库显示条目信息	116
5.2.2 使用Keytool直接从密钥库显示证书详细信息	117
5.2.3 使用Keytool将数字证书导出到文件	118
5.2.4 使用Keytool从文件中显示证书	119
5.2.5 在Windows中从文件显示证书	120
5.2.6 Java程序从证书文件读取证书	121
5.2.7 Java程序从密钥库直接读取证书	123
5.2.8 Java程序显示证书指定信息（全名/公钥/签名等）	125
5.3 密钥库的维护	128
5.3.1 使用Keytool删除指定条目	128
5.3.2 使用Keytool修改指定条目的口令	129
5.3.3 Java程序列出密钥库所有条目	130
5.3.4 Java程序修改密钥库口令	131
5.3.5 Java程序修改密钥库条目的口令及添加条目	133
5.3.6 Java程序检验别名及删除条目	135
5.4 数字证书的签发	136
5.4.1 确定CA的权威性——安装CA的证书	136
5.4.2 验证CA的权威性——显示CA的证书	140
5.4.3 Java程序签发数字证书	141

5.4.4 数字证书签名后的发布	147
5.5 数字证书的检验	149
5.5.1 Java程序验证数字证书的有效期	149
5.5.2 使用Windows查看证书路径验证证书的签名	151
5.5.3 Windows中卸载证书	152
5.5.4 Java程序使用CA公钥验证已签名的证书	156
第6章 数字化身份 ——CertPath证书链	159
6.1 密钥库中创建并保存证书链的几种方法	159
6.1.1 使用Keytool将已签名的数字证书导入密钥库	159
6.1.2 使用Java程序将已签名的数字证书导入密钥库	163
6.2 几种获取CertPath证书链的方法	165
6.2.1 根据证书文件生成CertPath类型的对象	166
6.2.2 从密钥库读取证书链生成CertPath类型的对象	168
6.2.3 从HTTPS服务器获取证书链	170
6.3 CertPath对象的证书显示和保存	177
6.3.1 显示CertPath中的证书	177
6.3.2 保存CertPath中的证书	179
6.4 验证CertPath证书链	181
6.4.1 验证主体和签发者	182
6.4.2 验证签名	184
6.4.3 CertPathValidator类基于TrustAnchor验证证书链	186
6.4.4 CertPathValidator类基于密钥库验证证书链	190
6.5 使用CertStore对象保存和提取证书	194
6.5.1 创建CertStore对象	194
6.5.2 定义证书的选择标准	198
6.5.3 从CertStore中提取证书	201
6.6 证书的吊销	203
6.6.1 查看证书吊销清单常规信息	203
6.6.2 查看清单中被吊销的证书	209
6.6.3 从CertStore对象中提取已吊销的证书	211
第7章 数据的安全传输和身份验证 ——SSL和HTTPS编程	216
7.1 最简单的SSL通信	216
7.1.1 最简单的SSL服务器	216
7.1.2 最简单的SSL客户程序	218
7.1.3 进一步设置信任关系	221
7.1.4 设置默认信任密钥库	222
7.1.5 通过KeyStore对象选择密钥库	225
7.2 进一步的SSL客户和服务器的例子	227
7.2.1 设计通信规则	227
7.2.2 查看对方的证书等连接信息	232
7.3 HTTPS客户及服务器程序	236
7.3.1 最简单的HTTPS服务器程序	236
7.3.2 最简单的HTTPS客户程序	243
7.3.3 基于Socket的HTTPS客户程序	246

7.3.4 传输实际文件.....	248
7.4 基于证书的客户身份验证.....	252
7.4.1 最简单的验证客户身份的HTTPS服务器程序.....	252
7.4.2 编写客户程序连结需客户验证的HTTPS服务器.....	255
第8章 程序运行的安全性——基于代码来源的授权.....	257
8.1 安全管理器的使用.....	257
8.1.1 使用默认的安全管理器限制应用程序.....	257
8.1.2 编写自己的安全管理器.....	260
8.1.3 在程序中设置安全管理器.....	263
8.2 使用策略文件基于代码位置进行授权.....	264
8.2.1 允许所有代码具有所有权限.....	264
8.2.2 允许所有代码具有特定的权限.....	269
8.2.3 许所有代码具有多种不同权限.....	271
8.2.4 针对指定目录中的代码的授权.....	273
8.2.5 针对从网络下载的代码的授权.....	278
8.3 使用策略文件基于代码的所有者进行授权.....	282
8.3.1 编程者对代码进行签名.....	282
8.3.2 用户检验已签名的代码.....	283
8.3.3 针对签名者进行授权.....	285
8.4 定义特权代码.....	288
8.4.1 不同代码之间的调用和授权.....	288
8.4.2 使用doPrivileged()方法定义特权代码.....	293
8.4.3 使用匿名类定义特权代码.....	297
8.5 权限的操作及定义自己的权限.....	301
8.5.1 策略文件权限的检测.....	302
8.5.2 最简单的权限定义.....	305
8.5.3 使用签名的权限.....	309
8.6 Applet的安全运行.....	311
8.6.1 使用AppletViewer运行的Java Applet.....	311
8.6.2 浏览器中使用Java Plug-in运行Java Applet.....	314
8.6.3 浏览器基于策略文件运行Java Applet.....	320
8.6.4 浏览器运行RSA签名的Java Applet.....	322
8.6.5 Java Plug-in的证书管理.....	327
8.6.6 使用usePolicy权限加强RSA签名Applet的安全控制.....	329
第9章 程序运行的安全性——基于用户身份的验证和授权（JAAS）.....	332
9.1 最简单的身份验证.....	332
9.1.1 最简单的登录.....	332
9.1.2 更换登录模块修改验证方式.....	336
9.1.3 更换回调处理器修改登录界面.....	338
9.1.4 使用非交互式验证.....	339
9.2 编写自己的登录模块.....	342
9.2.1 简单的登录模块.....	342
9.2.2 完整的登录模块模板.....	350
9.2.3 使用模板编写自己的密钥库登录模块.....	358

9.3 使用堆叠式登录.....	366
9.3.1 堆叠式登录及各个登录模块的相互关系	366
9.3.2 堆叠登录模块之间的信息共享	370
9.4 编写自己的回调处理器.....	386
9.4.1 最简单的回调处理器	386
9.4.2 图形界面口令输入的安全性	392
9.4.3 文本界面口令输入的安全性	394
9.4.4 更加安全的文本界面口令输入方式	396
9.5 基于身份的授权.....	399
9.5.1 使用策略文件的基于身份授权	399
9.5.2 使用编程方式的基于身份授权	405
9.5.3 比较doAsPrivileged()和doAs()	409

第 1 章运行环境设置

本章重点:

本章将介绍本书所使用的软件的安装和设置。其中最主要的软件是 J2SDK1.4，安装好该软件后，将可以运行本书绝大部分程序。个别章节中使用的反编译器、混淆器等也在本章作了介绍。

本章主要内容:

- 安装和配置 J2SDK1.4 环境
- 安装反编译器
- 安装混淆器

1.1 J2SE 的安装和设置

J2SE 的全称是 Java™ 2 Software Development Kit, Standard Edition (Java 2 SDK, Standard Edition)。本节介绍其下载、安装、设置以及其中本书所使用的主要工具。

1.1.1 下载 J2SE

访问 <http://java.sun.com>，在出现的主页中显示了 Java 2 平台的三种不同版本：Enterprise Edition(J2EE)、Standard Edition(J2SE)和Micro Edition(J2ME)。本节使用的是J2SE。

单击“Standard Edition(J2SE)”链接，在出现的页面中单击“J2SE Downloads”链接，进而选择版本。由于自 1.4.0 版本开始集成了各种安全机制和相关的 API，因此请选择 1.4.0 或以上的版本。本书以 1.4.0 版本为例。

双击 1.4.0 版本的链接，在出现的下载窗口（<http://java.sun.com/j2se/1.4/download.html>）中选择“Windows (all languages, including English)”一行、SDK 一列中的“download”链接，最随后出现的协议窗口中单击“Accept”按钮，出现最后的下载窗口，单击“Download j2sdk-1_4_0_03-windows-i586.exe”，则提示保存目录并开始下载软件。该软件大小为 37,118,676 bytes。

在上述过程中同时可下载 J2SE 的文档。在文档中包含了多个教程和 API 文档。

也可在一些FTP搜索引擎如 <http://bingle.pku.edu.cn>中输入“j2sdk-1_4”关键字搜索一些国内的下载站点。

1.1.2 安装 J2SE

J2SE 安装过程如下：

(1) 开始安装

双击下载的 J2SE 安装程序，在出现的初始安装界面中单击“Next”按钮，出现许可协议窗口，选择“Yes”接受协议。

(2) 选择安装目录

接着选择安装目录。不同版本的 J2SE 默认安装目录不同，对于 j2sdk-1_4_0_03-windows-i586.exe，默认安装目录是 c:\j2sdk1.4.0_03，不妨单击“Browse”按钮，重新设置安装目录为 c:\j2sdk1.4.0。这样，本书叙述中对不同的改进版本就可以统一使用 c:\j2sdk1.4.0 代表 J2SE 的安装目录。继续单击“Next”按钮。

(3) 选择安装的组件

在接下来的窗口中选择欲安装的组件（如图 1-1），如果硬盘空间足够的话不妨安装所有组件。其中“Program Files”一项是必选的。“Java Source”组件提供了组成Java平台的所有类的源代码。在本书有些章节中会对利用部分源代码来理解Java各种文档和教程中某些比较含糊的地方。

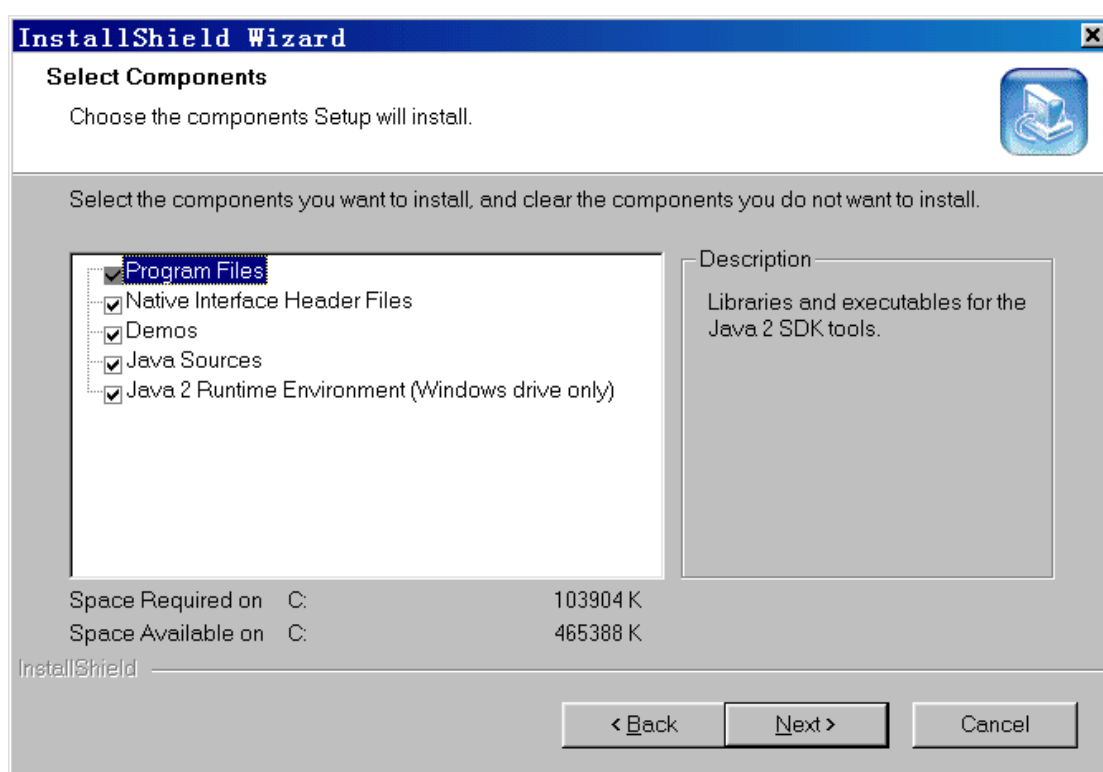


图 1-1 选择所安装的组件

(4) 选择使用 Java Plug-in 的浏览器

在图 1-1的窗口中单击“Next”按钮，出现图 1-2所示的窗口。在 8.6 节中将在浏览器中使用Java Plug-in来运行Java Applet，这里可选择所使用的浏览器类型。

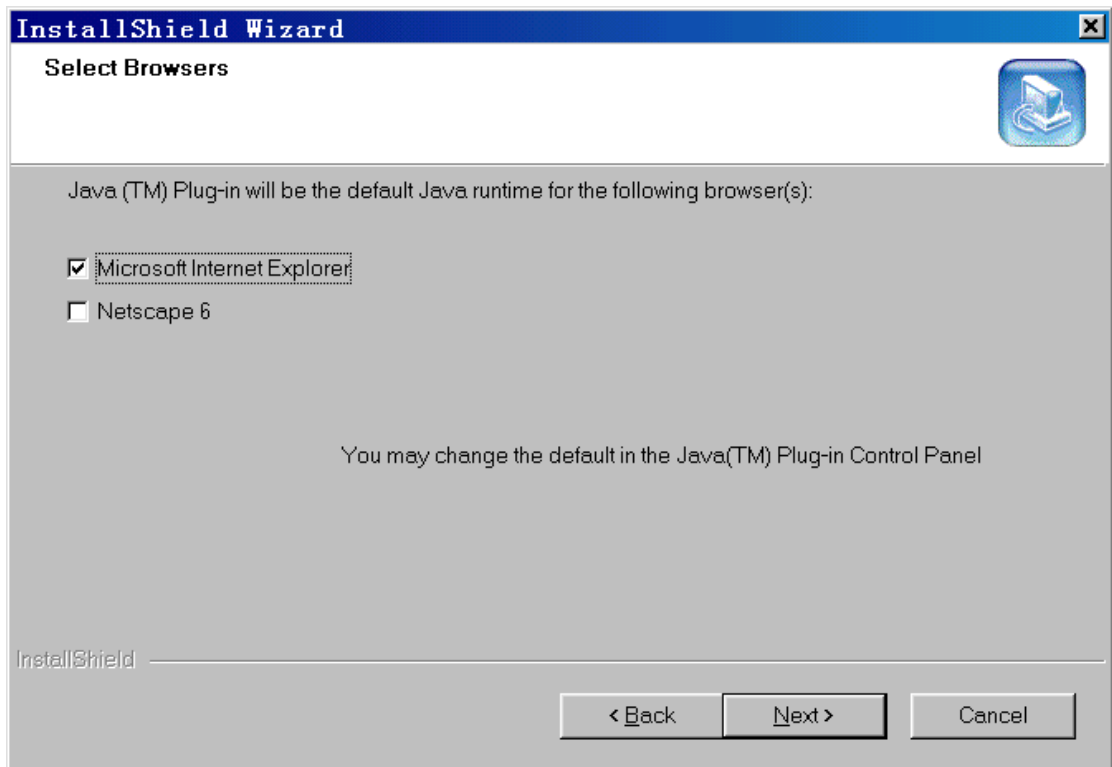


图 1-2 选择使用 Java Plug-in 的浏览器

(5) 结束安装

继续单击图 1-2 中的“Next”按钮将开始实际的安装过程，最后单击“Finish”按钮结束安装。

1.1.3 设置 J2SE

对 J2SE 的设置主要是设置环境变量，以方便使用安装目录下 bin 子目录中的各种工具。由于 J2SE 的编译、运行多在 DOS 窗口进行，为了能在任何目录中使用 c:\j2sdk1.4.0\bin 目录下的工具，可在 Windows 9X 操作系统 C:盘根目录的 autoexec.bat 中加入如下一行：

```
set path=c:\j2sdk1.4.0\bin;%path%
```

则以后每次打开 DOS 窗口时，会自动将 c:\j2sdk1.4.0\bin 目录加入搜索路径（第一次设置时需重新启动计算机才生效）。在 DOS 中执行一个程序时，如果当前目录没有该程序，会自动到 c:\j2sdk1.4.0\bin 等目录查找。

如果使用的是 Windows 2000 或 Windows XP，可打开“控制面板”，双击其中的“系统”，在系统特性窗口单击“高级”选项卡，进而单击“环境变量”按钮。在出现的环境变量窗口中，在“系统变量”中选择“Path”，单击“编辑”按钮，在弹出的“编辑系统变量”的窗口中变量值最后加上“; c:\j2sdk1.4.0\bin”。如图 1-3 所示。

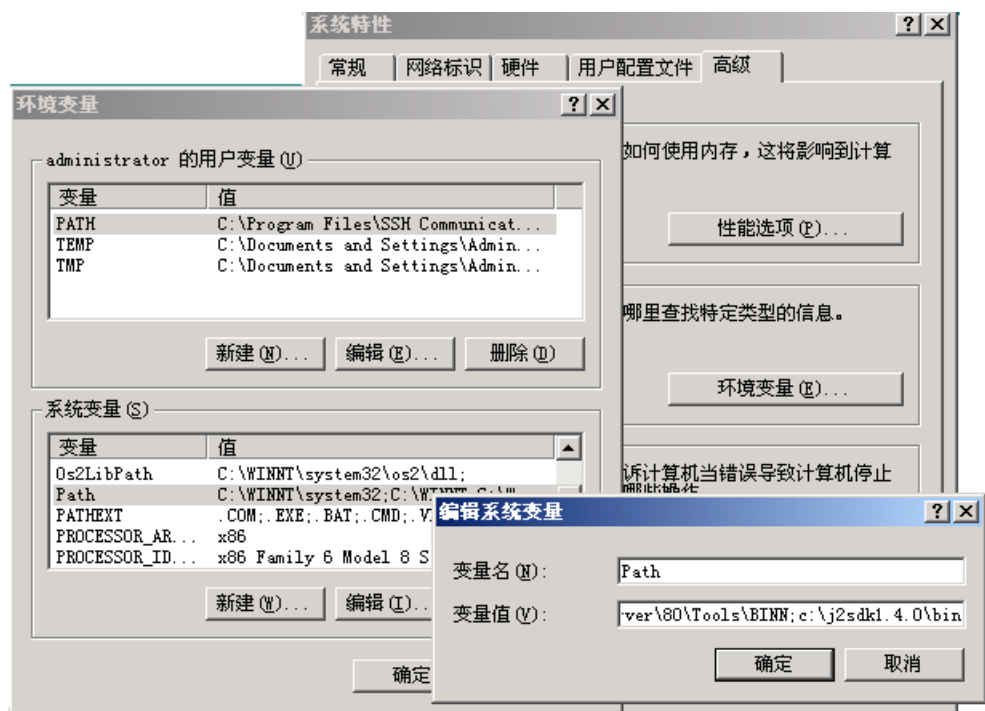


图 1-3 设置系统变量

单击多次“确定”后，无需重新启动计算机，则每次打开 DOS 窗口时将自动将 c:\j2sdk1.4.0\bin 目录加入搜索路径。

2.1.4 J2SE 的主要工具

Java 2 SDK 的主要工具安装在 c:\j2sdk1.4.0\bin 目录，这些工具主要功能如下，本书主要使用了其中的 javac、java、jar、keytool、policytool、Jarsigner、HtmlConverter 和 appletviewer 工具。

基本工具：

- javac** Java 编程语言的编译器。本书各章的程序都是在 DOS 窗口中通过执行“javac 文件名”来编译 Java 程序的。文件名必须以.java 为后缀，编译以后生成.class 为后缀的字节码文件。
- java** 用于执行 Java 应用程序。本书各章的程序大都通过在 DOS 窗口输入“java 字节码文件名称”来运行 javac 编译好的程序。输入命令时，字节码文件名称的后缀不输入。
- javadoc** 用于生成 API 文档。在编写程序时将注释语句写在“/**”和“*/”之间，则其内容便可被 javadoc 识别，执行“javadoc *.java”，自动生成 API 文档。
- appletviewer** 没有 Web 浏览器时可用来运行和调试 Java Applet 程序，本书 8.6 节使用了该工具。
- jar** 管理 jar 文件。本书多次使用该工具将 Java 程序打包成为一个文件，并进而进行进一步的处理。
- jdb** Java 调试器
- javah** C 头文件和存根的生成器，用于编写本地文件。
- javap** 类分解器。可显示字节码文件的包、标记为 public 及 protected 的变量和方法。

法等信息。
extcheck 检测 jar 文件的版本冲突

RMI 工具:

rmic 生成远程对象的架构和存根。执行后可根据给定的字节码文件 XX.class 可生成 XX_Stub.class 和 XX_Skel.class 文件部署在 RMI 系统中。
rmiregistry 提供远程对象的注册服务。RMI 客户程序可通过该服务找到远程对象。
rmid 启动激活系统后台程序。
serialver 返回类的 serialVersionUID

国际化工具:

native2ascii 将本地编码的文本转换为 Unicode 编码

安全工具

keytool 管理密钥库和证书。本书自第 5 章起大量使用该工具。
Jarsigner 对 jar 文件进行签名, 并验证 jar 文件的签名。本书第 8 章使用它为 jar 文件签名, 实现基于代码所有者的授权。
policytool 管理策略文件的图形界面工具。本书第 8 章开始使用它进行各种授权操作

Java IDL and RMI-IIOP 工具

tnameserv 提供访问名字服务
idlj 根据给定的 IDL 文件生成 Java 绑定, 使 Java 程序可以使用 CORBA 功能
orbd 在 CORBA 环境中使客户透明地定位和执行服务器上 persistent 对象
servertool 应用程序编写者注册、取消注册、启动、关闭 persistent 服务器的命令行工具。

Java Plug-in 工具

unregbean 用于取消 Java Bean 组件的注册
HtmlConverter 修改调用 Applet 的 HTML 网页, 将其中的<applet>标记按照一定格式转换为<Object>标记, 以便让浏览器使用 Java Plug-in 运行 Java Applet 程序。本书第 8 章使用了该工具进行转换。

1.2 反编译器的安装

JAD 是最著名的反编译器, 其主页为 <http://www.geocities.com/SiliconValley/Bridge/8617/jad.html>, 在各个网上各个软件下载站点也多可下载。

JAD 是命令行工具, 本书使用的是 Jad v1.5.7d。JAD 不需要安装, 只要将可执行文件从其 WinZip 格式的软件包中解开即可。不妨将其解压缩到 c:\windows 目录, 这样, 在 DOS 窗口任意目录都可以使用该工具。JAD 的 WinZip 格式软件包中还包括一个 readme.txt 文件。其部分用法如下:

(1) 最简单的用法

JAD 的最简单用法是直接输入“Jad 字节码文件名称”，如

```
Jad example.class
```

或

```
Jad example
```

则将 example.class 反编译，反编译得到的源代码保存为 example.jad。如果 example.jad 已经存在，会提示是否覆盖。使用命令选项-o 可跳过提示。

(2) 使用通配符

JAD 支持通配符，如可输入：

```
Jad *.class
```

则将当前目录所有字节码文件进行反编译。

(3) 指定输出形式

如果希望反编译结果以.java 为后缀，可输入

```
jad -sjava example.class
```

但此时要注意不要覆盖了原有的源代码，尤其加上-o 选项后，覆盖原有文件不会出现提示，更应小心。

类似地，可以指定任意后缀。

使用-p 选项可以将反编译结果输出到屏幕，如

```
jad -p example.class
```

进一步可将其重定向到任意文件名，如

```
jad -p example.class >my.java
```

使用“-d 目录名”选项可指定反编译后的输出文件的目录，若没有目录会自动创建。如

```
jad -o -dtest -sjava *.class
```

或

```
jad -o -d test -s java *.class
```

将当前目录所有字节码文件反编译到 test 目录。

(4) 设置输出内容

使用-a 选项则可以使用 JVM 字节码注释输出结果，得到类似如下的输出

```
public static void main(String args[])
    throws Exception
{
    System.setProperty("javax.net.ssl.keyStore", "mykeystore");
//    0    0:ldc1          #2    <String "javax.net.ssl.keyStore">
//    1    2:ldc1          #3    <String "mykeystore">
//    2    4:invokestatic #4    <Method String System.setProperty(String, String)>
//    3    7:pop
```

使用-b 选项可以输出冗余的大括号（缺省不加大括号）。如 if 语句中只有一条语句也加大括号，如

```
if(a){
    b();
}
```

使用-f 选项则对所有类、成员变量和方法使用全称，如

```
public static void main(java.lang.String args[])
    throws java.lang.Exception
```

```

{
    java.lang.System.setProperty("javax.net.ssl.keyStore",
        "mykeystore");
    java.lang.System.setProperty("javax.net.ssl.keyStorePassword",
        "wshr.ut"
    );
    javax.net.ssl.SSLServerSocketFactory sslserversocketfactory =
        (javax.net.ssl.SSLServerSocketFactory)
            javax.net.ssl.SSLServerSocketFactory.getDefault();
    java.net.ServerSocket serversocket =
        sslserversocketfactory.createServerSocket(5432);
    java.lang.System.out.println("Waiting for connection...");
}

```

使用-stat 选项可以最后统计类、方法、成员变量等的数量。

1.3 混淆器的安装

本书使用的混淆器为 Marvin Obfuscator 1.2b, 可在 <http://www.drjava.de/obfuscator/> 下载, 将其直接解压缩到某个目录即可。

该混淆器使用 Java 开发, 提供了一个 jar 文件。为了便于运行程序, Marvin Obfuscator 1.2b 提供了一个批处理文件 obfuscate.bat。其内容如下:

```

@echo off
set JAVAILIB=c:\java\jdk1.3\jre\lib\rt.jar
java -cp marvinobfuscator.jar;%JAVAILIB% drjava.marvin.Obfuscate %1 %2 %3 %4
%5 %6 %7 %8 %9

```

该批处理针对的是 JDK1.3, 对于本书的环境需要将 JAVAILIB 环境变量作些修改, 将新增的类库加入 JAVAILIB, 如可改为:

```
set JAVAILIB=C:\j2sdk1.4.0\jre\lib\rt.jar;C:\j2sdk1.4.0\jre\lib\jce.jar
```

混淆器使用时的配置放在 marvinobfuscator1_2b 安装目录的 dummyproject 子目录下的 config.txt 文件中。需要混淆的类要先用 jar 打包, 如打包为 My.jar, 然后将需要 config.txt 开头几行的 "somelibrary.jar" 改为需要进行混淆操作的打包文件名称 "My.jar"。此外, 需要将 config.txt 中 "mainClasses=({"myapp.Main"})" 改为 My.jar 文件中包含 main() 方法的字节码文件, 如改为 "mainClasses=({"xx.class"})"。然后将该 config.txt 文件保存在 My.jar 所在目录中。

以后就可以执行

```
obfuscate    目标目录    混淆以后的文件名
```

进行混淆操作了。

第 2 章数据内容的保护——

加密和解密

本章重点:

加密是保护重要数据以及程序之间进行秘密通信的重要方法,随着加密的广泛应用,已发展成为一门单独的学科:密码学(Cryptography)。密码学这一单词来自希腊语 Kryptus(隐藏)和 Gráphein(写),可见通过加密将信息隐藏起来是密码学的重要内容。Java 中提供了常用的加密和解密算法,本章将介绍如何使用这些已有的算法。

本章主要内容:

- 通过编写凯撒密码了解加密和解密的基本过程
- 创建对称密钥,使用对称密钥进行加密和解密
- 创建非对称密钥,使用非对称密钥进行加密和解密
- 使用密钥协定分发密钥

2.1 一个简单的加密和解密程序——凯撒密码

本节将介绍一个简单的加密和解密程序,通过本实例,读者将了解加密和解密内部的过程和相关概念,消除对加密和解密的神秘感。

★ 实例说明

凯撒密码是罗马扩张时期朱利斯·凯撒(Julius Caesar)创造的,用于加密通过信使传递的作战命令。它将字母表中的字母移动一定位置而实现加密。例如如果向右移动 2 位,则字母 A 将变为 C,字母 B 将变为 D,...,字母 X 变成 Z,字母 Y 则变为 A,字母 Z 变为 B。因此,假如有个明文字符串“Hello”用这种方法加密的话,将变为密文:“Jgnnq”。而如果要解密,则只要将字母向相反方向移动同样位数即可。如密文“Jgnnq”每个字母左移两位变为“Hello”。这里,移动的位数“2”是加密和解密所用的密钥。

本实例通过 Java 实现了这一过程,由此可以了解与加密和解密相关的概念。

★ 编程思路

首先获取要加密的内容以及密钥,凯撒密码的密钥即字符移动的位数。由于凯撒密码器的移位是针对字符的,因此需要将待加密的内容中每个字符取出,然后针对每个字符分别加以移位。主要步骤如下:

- (1) 读取要加密的字符串、密钥

```
String s=args[0];
```

```
int key=Integer.parseInt(args[1]);
```

分析:作为示例,程序中通过命令行参数传入要加密的字符串。凯撒密码器的密钥比较简单,只是移动的位数,这里不妨通过命令行参数传入。由于移动的位数为整数,因此使用 Integer.parseInt()方法进行了转换。

- (2) 取出字符串中每个字符

```
for(int i=0;i<s.length();i++){
    char c=s.charAt(i);
```

分析：这里使用字符串类的 charAt () 方法取出每个字符，分别加以移位。

(3) 对每个字符进行移位

```
c+=key%26;
    if(c<'a') c+=26;
    if(c>'z') c-=26;
```

分析：由于字母表中共 26 个字符，因此移位前先将移动的位数(key)和 26 取模。由于 Java 中字符和整型可自动转换，因此将字符加上一个正整数即代表在字母表中右移多少位。如果移动的位数是负值，则代表在字母表中左移多少位。

尽管在移动之前已经将移动的位数和 26 取了模，但通过这种方式实现右移或左移仍可能发生超界。如字母 x 右移 4 位应该是字母 b，但将字母 x 增加 4 后超出 26 个字母的范围。因此移位后使用两个 if 语句判断一下，如果向左超界 (c<'a') 则增加 26；向右超界 (c>'z') 则减去 26。

此外由于大写字母和小写字母判断是否超界的依据不同，程序中将字符分为大写和小写分别处理。

★ 代码与分析

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

    String s=args[0];
    int key=Integer.parseInt(args[1]);
    String es="";
    for(int i=0;i<s.length();i++)
        { char c=s.charAt(i);
            if(c>='a' && c<='z') // 是小写字母
            { c+=key%26; //移动 key%26 位
              if(c<'a') c+=26; //向左超界
              if(c>'z') c-=26; //向右超界
            }
            else if(c>='A' && c<='Z') // 是大写字母
            { c+=key%26;
              if(c<'A') c+=26;
              if(c>'Z') c-=26;
            }
            es+=c;
        }
    System.out.println(es);
}
```

该程序既可用于加密又可用于解密。只要执行：

```
java Caesar 明文（要加密的字符串） 密钥（移动的位数）
```

即可加密。

在密钥前面加上负号，将运行

```
java Caesar 明文（要加密的字符串） -密钥（移动的位数）
```

即可解密。

如为了加密字符串“Hello World!”，可随意取一个密钥如 4，运行：

```
java Caesar "Hello World!" 4
```

将输出“Lipps Asvph!”。这里“Hello World!”是明文，“Lipps Asvph!”是密文。

如果密钥大于 26，程序中移位前会和 26 取模而将其调整到 26 以下。因此运行：

```
java Caesar "Hello World!" 30
```

同样将输出“Lipps Asvph!”。

为了将密文“Lipps Asvph!”解密，需要知道加密该密文所用的密钥 4，这样，执行：

```
java Caesar "Lipps Asvph!" -4
```

将得到明文“Hello World!”。

如果密钥和加密时所用的不同，则解密时将得到无意义的输出，如运行

```
java Caesar "Lipps Asvph!" -3
```

程序将输出“Ifmmp Xpsme!”。这样，只有知道密钥才能得到原来的密文。

2.2 对称密钥的生成和保存

上节的凯撒密码是很脆弱的，密钥总共只有 26 个，攻击者得到密文后即使不知道密钥，也可一个一个地试过去，最多试 26 次就可以得到明文。

现代密码算法的过程要复杂得多，其中一类和凯撒密码类似，加密和解密使用相同的密钥，称为对称密钥算法；另一类则在加密时使用一种密钥，在解密时使用另一种密钥，称为非对称密钥算法。这些算法的密钥也不再是简单的整数，而是很长的二进制数。这样，一个 56 位的密钥有 2^{56} （即 72,057,594,037,927,936）个不同的可能取值，这需要耗费超级计算机约一天的时间尝试每一个密钥。当密钥长度达到 128 位，则密钥数量达到 2^{128} 个，需要的时间增加到 2^{72} 倍，约 1.29×10^{19} 年。

Java 中已经提供了常用的加密算法，我们不需要了解算法的细节而可以直接使用这些算法实现加密。各种算法所用的密钥各有不同，本节将学习 Java 中创建对称密钥的方法。在随后的章节中介绍如何利用这些密钥进行加密和解密。

2.2.1 对称密钥的生成及以对象序列化方式保存

★ 实例说明

本实例给出 Java 中创建对称密钥的步骤，并通过对象序列化方式保存在文件中。

★ 编程思路：

（1） 获取密钥生成器

```
KeyGenerator kg=KeyGenerator.getInstance("DESede");
```

分析：Java 中 KeyGenerator 类中提供了创建对称密钥的方法。Java 中的类一般使用 new 操作符通过构造器创建对象，但 KeyGenerator 类不是这样，它预定义了一个静态方法 getInstance（），通过它获得 KeyGenerator 类型的对象。这种类成为工厂类或工厂。

方法 getInstance（）的参数为字符串类型，指定加密算法的名称。可以是

“Blowfish”、“DES”、“DESede”、“HmacMD5”或“HmacSHA1”等。这些算法都可以实现加密，这里我们不关心这些算法的细节，只要知道其使用上的特点即可。其中“DES”是目前最常用的对称加密算法，但安全性较差。针对DES安全性的改进产生了能满足当前安全需要的TripleDES算法，即“DESede”。“Blowfish”的密钥长度可达448位，安全性很好。“AES”是一种替代DES算法的新算法，可提供很好的安全性。

(2) 初始化密钥生成器

```
kg.init(168);
```

分析：该步骤一般指定密钥的长度。如果该步骤省略的话，会根据算法自动使用默认的密钥长度。指定长度时，若第一步密钥生成器使用的是“DES”算法，则密钥长度必须是56位；若是“DESede”，则可以是112或168位，其中112位有效；若是“AES”，可以是128，192或256位；若是“Blowfish”，则可以是32至448之间可以被8整除的数；“HmacMD5”和“HmacSHA1”默认的密钥长度都是64个字节。

(3) 生成密钥

```
SecretKey k=kg.generateKey();
```

分析：使用第一步获得的KeyGenerator类型的对象中generateKey()方法可以获得密钥。其类型为SecretKey类型，可用于以后的加密和解密。

(4) 通过对象序列化方式将密钥保存在文件中

```
FileOutputStream f=new FileOutputStream("key1.dat");
ObjectOutputStream b=new ObjectOutputStream(f);
b.writeObject(k);
```

分析：ObjectOutputStream类中提供的writeObject方法可以将对象序列化，以流的方式进行处理。这里将文件输出流作为参数传递给ObjectOutputStream类的构造器，这样创建好的密钥将保存在文件key1.data中。

★代码与分析：

```
import java.io.*;
import javax.crypto.*;
public class Skey_DES{
    public static void main(String args[])
        throws Exception{
        KeyGenerator kg=KeyGenerator.getInstance("DESede");
        kg.init(168);
        SecretKey k=kg.generateKey();
        FileOutputStream f=new FileOutputStream("key1.dat");
        ObjectOutputStream b=new ObjectOutputStream(f);
        b.writeObject(k);
    }
}
```

运行java Skey_DES，在当前目录下将生成文件key1.dat，其中包含的密钥可以用于使用Triple DES算法的加密和解密。

2.2.2 以字节保存对称密钥

★ 实例说明

2.2.1 小节的实例将密钥通过对象序列化方式保存在文件中，在文件中保存的是对象，

本实例以另一种方式保存在文件中，即以字节保存在文件中。

★ 编程思路：

Java 中所有的密钥类都有一个 `getEncoded()` 方法，通过它可以从密钥对象中获取主要编码格式，其返回值是字节数组。其主要步骤为：

(1) 获取密钥

```
FileInputStream f=new FileInputStream("key1.dat");
ObjectInputStream b=new ObjectInputStream(f);
Key k=(Key)b.readObject();
```

分析：该步骤与 2.2.1 小节的第 4 步是相对应的，2.2.1 小节的第 4 步将密钥对象以对象流的方式存入文件，而这一步则将文件中保存的对象读取出来以便使用。首先创建文件输入流，然后将其作为参数传递给对象输入流，最后执行对象输入流的 `readObject()` 方法读取密钥对象。由于 `readObject()` 返回的是 `Object` 类型，因此需要强制转换成 `Key` 类型。

这里使用的是已有的密钥，也可以不使用这里的三行代码，而使用 2.1.1 小节中的前三步的代码生成新的密钥再继续下面的步骤。

(2) 获取主要编码格式

```
byte[] kb=k.getEncoded();
```

分析：执行 `SecretKey` 类型的对象 `k` 的 `getEncoded()` 方法，返回的编码放在 `byte` 类型的数组中。

(3) 保存密钥编码格式

```
FileOutputStream f2=new FileOutputStream("keykb1.dat");
f2.write(kb);
```

分析：先创建文件输出流对象，在其参数中指定文件名，如 `keykb1.dat`。然后执行文件输出流的 `write()` 方法将第 2 步中得到的字节数组中的内容写入文件。

★代码与分析：

```
import java.io.*;
import java.security.*;
public class Skey_kb{
    public static void main(String args[]) throws Exception{
        FileInputStream f=new FileInputStream("key1.dat");
        ObjectInputStream b=new ObjectInputStream(f);
        Key k=(Key)b.readObject();
        byte[] kb=k.getEncoded();
        FileOutputStream f2=new FileOutputStream("keykb1.dat");
        f2.write(kb);
        // 打印密钥编码中的内容
        for(int i=0;i<kb.length;i++){
            System.out.print(kb[i]+" ");
        }
    }
}
```

程序中在保存了密钥编码后，又使用循环语句将字节数组中的内容打印出来。这样可以较为直观地看到密钥编码的内容。

★运行程序

输入 `java Skey_kb` 运行程序，在程序的当前目录中将产生文件名为 `keykb1.dat` 的文件，屏幕输出如下：

```
11,-105,-119,50,4,-105,16,38,-14,-111,21,-95,70,-15,76,-74,67,-88,59,-71,55,-125,104,42,
```

此即程序中创建的密钥的编码内容，如果用文本编辑器打开 `keykb1.dat`，看到的不是上面的数字而是类似下面的字符：

```
棄 21 ?&驊1 謁禡??僑*
```

这是因为 `keykb1.dat` 是一个二进制文件，存放的是任意二进制数。

读者运行时肯定结果和上面会有所不同，实际上 2.2.1 小节的程序每次运行时生成的密钥都不会相同，这就保证了密钥的唯一性。作为对称密钥，只要保证若加密某段文字用的是某个密钥，则解密这段密文时用同样的密钥即可。

2.3 使用对称密钥进行加密和解密

在 2.2 节学习了如何创建对称密钥，本节介绍如何使用创建好的对称密钥进行加密和解密。

2.3.1 使用对称密钥进行加密

★ 实例说明

本实例的输入是 2.2.1 小节中生成并以对象方式保存在文件 `key1.dat` 中的密钥，以及需要加密的一段最简单的字符串 `"Hello World!"`，使用密钥对 `"Hello World!"` 进行加密，加密后的信息保存在文件中。在此基础上读者可以举一反三加密各种信息。

★ 编程思路：

首先要从文件中获取已经生成的密钥，然后考虑如何使用密钥进行加密。这涉及到各种算法。Java 中已经提供了常用的加密算法，我们执行 Java 中 `Cipher` 类的各个方法就可以完成加密过程，其主要步骤为：

- (1) 从文件中获取密钥

```
FileInputStream f=new FileInputStream("key1.dat");
ObjectInputStream b=new ObjectInputStream(f);
Key k=(Key)b.readObject();
```

分析：该步骤与 2.2.2 小节的第 1 步相同。

(2) 创建密码器 (Cipher 对象)

```
Cipher cp=Cipher.getInstance("DESede");
```

分析: 和 2.2.1 小节的第 1 步中介绍的 KeyGenerator 类一样, Cipher 类是一个工厂类,它不是通过 new 方法创建对象,而是通过其中预定义的一个静态方法 getInstance () 获取 Cipher 对象。

getInstance () 方法的参数是一个字符串,该字符串给出 Cipher 对象应该执行哪些操作,因此把传入的字符串称为转换 (transformation)。通常通过它指定加密算法或解密所用的算法的名字,如本例的"DESede"。此外还可以同时指定反馈模式及填充方案等,如"DESede/ECB/PKCS5Padding"。反馈模式及填充方案的概念和用途将在后面的章节介绍。

(3) 初始化密码器

```
cp.init(Cipher.ENCRYPT_MODE, k);
```

分析: 该步骤执行 Cipher 对象的 init () 方法对 Cipher 对象进行初始化。该方法包括两个参数,第一个参数指定密码器准备进行加密还是解密,若传入 Cipher.ENCRYPT_MODE 则进入加密模式。第二个参数则传入加密或解密所使用的密钥,即第 1 步从文件中读取的密钥对象 k。

(4) 获取等待加密的明文

```
String s="Hello World!";  
byte ptext[]=s.getBytes("UTF8");
```

分析: Cipher 对象所作的操作是针对字节数组的,因此需要将要加密的内容转换成字节数组。本例中要加密的是一个字符串 s,可以使用字符串的 getBytes () 方法获得对应的字节数组。getBytes () 方法中必须使用参数"UTF8"指定...,否则...

(5) 执行加密

```
byte ctext[]=cp.doFinal(ptext);
```

分析: 执行 Cipher 对象的 doFinal () 方法,该方法的参数中传入待加密的明文,从而按照前面几步设置的算法及各种模式对所传入的明文进行加密操作,该方法返回加密的结果。

(6) 处理加密结果

```
FileOutputStream f2=new FileOutputStream("SEnc.dat");  
f2.write(ctext);
```

分析: 第 5 步得到的加密结果是字节数组,对其可作各种处理,如在网上传递、保存在文件中等。这里将其保存在文件 SEnc.dat 中。

★代码与分析:

```
import java.io.*;  
import java.security.*;  
import javax.crypto.*;  
public class SEnc{  
    public static void main(String args[]) throws Exception{  
        String s="Hello World!";  
        FileInputStream f=new FileInputStream("key1.dat");  
        ObjectInputStream b=new ObjectInputStream(f);  
        Key k=(Key)b.readObject( );
```

```

Cipher cp=Cipher.getInstance("DESede");
cp.init(Cipher.ENCRYPT_MODE, k);
byte ptext[]=s.getBytes("UTF8");
for(int i=0;i<ptext.length;i++){
    System.out.print(ptext[i]+",");
}
System.out.println("");
byte ctext[]=cp.doFinal(ptext);
for(int i=0;i<ctext.length;i++){
    System.out.print(ctext[i] +",");
}
FileOutputStream f2=new FileOutputStream("SEnc.dat");
f2.write(ctext);
}
}

```

程序中使用两个循环语句将字节数组加密前后加密后的内容打印出来，可作为对比。

★运行程序

当前目录下必须有 2.2.1 小节中生成的密钥文件 key1.dat，输入 java SEnc 运行程序，在程序的当前目录中将产生文件名为 SEnc.dat 的文件，屏幕输出如下：

```

72,101,108,108,111,32,87,111,114,108,100,33,
-57,119,0,-45,-9,23,37,-56,-60,-34,-99,105,99,113,-17,76,

```

其中第一行为字符串"Hello World!"的字节数组编码方式，第二行为加密后的内容，第二行的内容会随着密钥的不同而不同。

第一行的内容没有加过密，任何人若得到第一行数据，只要将其用二进制方式写入文本文件，用文本编辑器打开文件就可以看到对应的字符串“Hello World!”。而第二行的内容由于是加密过的，没有密钥的人即使得到第二行的内容也无法知道其内容。

密文同时保存在 SEnc.dat 文件中，将其提供给需要的人时，需要同时提供加密时使用的密钥(key1.dat，或 keykb1.dat)，这样收到 SEnc.dat 中密文的人才能够解密文件中的内容。

2.3.2 使用对称密钥进行解密

★ 实例说明

有了 2.3.1 小节加密后的密文 SEnc.dat，以及加密时所使用的密钥 key1.dat 或 keykb1.dat，本实例对 SEnc.dat 中的密文进行解密，得到明文。

★ 编程思路：

首先要从文件中获取加密时使用的密钥，然后考虑如何使用密钥进行解密。其主要步骤为：

(1) 获取密文

```

FileInputStream f=new FileInputStream("SEnc.dat");
int num=f.available();

```



```
byte[] ctext=new byte[num];
f.read(ctext);
```

分析：密文存放在文件 SEnc.dat 中，由于解密是针对字节数组进行操作的，因此要先将密文从文件中读入字节数组。首先创建文件输入流，然后使用文件输入流的 available() 方法判断密文将占用多少字节，从而创建相应大小的字节数组 ctext，最后使用文件输入流的 read() 方法一次性读入数组 ctext。

如果不考虑通用性，也可将要加密的内容直接在程序中向数组赋值。如可将 2.3.1 小节的第二行输出的密文用如下语句直接赋值：

```
byte ctext[] = {-57,119,0,-45,-9,23,37,-56,-60,-34,-99,105,99,113,-17,76};
```

该句可替代上面的四条语句，只是通用性差了，只能加密这一条密文。

(2) 获取密钥

```
FileInputStream f2=new FileInputStream("keykb1.dat");
int num2=f2.available();
byte[] keykb=new byte[num2];
f2.read(keykb);
SecretKeySpec k=new SecretKeySpec(keykb,"DESede");
```

分析：获取可以和 2.3.1 小节第 1 步一样直接获取密钥，本实例使用另外一种方式获取密钥，即使用 2.2.2 小节以字节方式保存在文件 keykb1.dat 中的密钥。

首先要将 keykb1.dat 中的内容读入字节数组 keykb，这里使用了和第 1 步类似的四条语句。如果不考虑通用性，也可以将 2.2.2 小节输出的信息如下直接赋值：

```
byte[] keykb = {11, -105, -119, 50, 4, -105, 16, 38, -14, -111, 21, -95,
70, -15, 76, -74, 67, -88, 59, -71, 55, -125, 104, 42};
```

最后，使用将其作为参数传递给 SecretKeySpec 类的构造器而生成密钥。SecretKeySpec 类的构造器中第 2 个参数则指定加密算法。由于 keykb1.dat 中的密钥原来使用的是 DESede 算法，因此这里仍旧使用字符串“DESede”作为参数。

(3) 创建密码器 (Cipher 对象)

```
Cipher cp=Cipher.getInstance("DESede");
```

分析：该步骤同 2.3.1 小节的第 2 步。

(4) 初始化密码器

```
cp.init(Cipher.DECRYPT_MODE, k);
```

分析：该步骤和 2.3.1 的第 3 步类似，对 Cipher 对象进行初始化。该方法包括两个参数，第一个参数传入 Cipher.DECRYPT_MODE 进入解密模式，第二个参数则传入解密所使用的密钥。

(5) 执行解密

```
byte[] ptext=cp.doFinal(ctext);
```

分析：该步骤和 2.3.1 的第 5 步类似，执行 Cipher 对象的 doFinal() 方法，该方法的参数中传入密文，从而按照前面几步设置的算法及各种模式对所传入的密文进行解

密操作，该方法返回解密的结果。

★代码与分析:

```
import java.io.*;
import java.security.*;
import javax.crypto.*;
import javax.crypto.spec.*;

public class SDec{
    public static void main(String args[]) throws Exception{
        // 获取密文
        FileInputStream f=new FileInputStream("SEnc.dat");
        int num=f.available();
        byte[ ] ctext=new byte[num];
        f.read(ctext);
        // 获取密钥
        FileInputStream f2=new FileInputStream("keykb1.dat");
        int num2=f2.available();
        byte[ ] keykb=new byte[num2];
        f2.read(keykb);
        SecretKeySpec k=new SecretKeySpec(keykb,"DESede");
        // 解密
        Cipher cp=Cipher.getInstance("DESede");
        cp.init(Cipher.DECRYPT_MODE, k);
        byte [ ]ptext=cp.doFinal(ctext);
        // 显示明文
        String p=new String(ptext,"UTF8");
        System.out.println(p);
    }
}
```

程序中最后将明文生成字符串加以显示。

★运行程序

当前目录下必须有 2.2.2 小节中生成的密钥文件 keykb1.dat，以及 2.3.1 小节的密文文件 SEnc.dat。

输入 `java SDec` 运行程序，将输出明文字符串“Hello World!”。

2.4 基于口令的加密和解密

使用对称密钥加密时密钥都很长，如 2.2.2 小节的密钥对应的字节序列为

“11,-105,-119,50,4,-105,16,38,-14,-111,21,-95,70,-15,76,-74,67,-88,59,-71,55,-125,104,42”，很难记住。一种做法是像 2.2.2 小节那样把它保存在文件中，需要时读取文件，其缺点容易被窃取，携带也不方便；另一种做法是将其打印出来，需要时对照打印出的内容手工一个一个输入，但由于密钥很长，输入很麻烦。

在实际使用中，更常见的是基于口令的加密。加密时输入口令，口令可以由使用者自己确定一个容易记忆的。解密时只有输入同样的口令才能够得到明文。本节通过两个最简单的例子说明其基本用法。

2.4.1 基于口令的加密

★ 实例说明

本实例通过使用口令加密的一段最简单的字符串"Hello World!"，加密后的信息保存在文件中。在此基础上读者可以举一反三加密各种信息。

★ 编程思路：

和 2.3 节一样，基于口令的加密也是使用 Java 的 Cipher 类，只是在加密算法中使用基于口令的加密算法。此外，加密时所用的密钥是根据给定的口令生成的。为了增加破解的难度，PBE 还使用一个随机数（称为盐）和口令组合起来加密文件。此外还进行重复计算（迭代）。编程的主要步骤如下：

(1) 读取口令

```
char[] passwd=args[0].toCharArray();
PBEKeySpec pbks=new PBEKeySpec(passwd);
```

分析：本实例通过命令行参数读取口令。为了后面步骤可以由口令生成密钥，需要将口令保存在类 PBEKeySpec 中，类 PBEKeySpec 的构造器传入的参数是字符数组，所以使用了字符串的 toCharArray() 方法生成字符数组。

(2) 由口令生成密钥

```
SecretKeyFactory kf=SecretKeyFactory.getInstance("PBEWithMD5AndDES");
SecretKey k=kf.generateSecret(pbks);
```

分析：生成密钥可通过 SecretKeyFactory 类的 generateSecret() 方法实现，只要将存有口令的 PBEKeySpec 对象作为参数传递给 generateSecret() 方法即可。SecretKeyFactory 类是一个工厂类，通过预定义的一个静态方法 getInstance() 获取 SecretKeyFactory 对象。

getInstance() 方法的参数是一个字符串，指定口令加密算法，如 PBEWithMD5AndDES，PBEWithHmacSHA1AndDESede 等。JCE 中已经实现的是 PBEWithMD5AndDES。

(3) 生成随机数（盐）

```
byte[] salt=new byte[8];
Random r=new Random();
r.nextBytes(salt);
```

分析：对于 PBEWithMD5AndDES 算法，盐必须是 8 个元素的字节数组，因此创建数组 salt。Java 中 Random 类可以生成随机数，执行其 nextBytes() 方法，方法的参数为 salt，即可生成的随机数并将随机数赋值给 salt。

(4) 创建并初始化密码器

```
Cipher cp=Cipher.getInstance("PBEWithMD5AndDES");
```

```
PBEPParameterSpec ps=new PBEPParameterSpec(salt,1000);
cp.init(Cipher.ENCRYPT_MODE, k,ps);
```

分析: 和以前一样通过 `getInstance()` 方法获得密码器, 其中的参数使用基于口令的加密算法 “PBKDF2WithHmacSHA1”。但在执行 `init()` 初始化密码器时, 除了指定第 2 步生成的口令密钥外, 还需要指定基于口令加密的参数, 这些参数包括为了提高破解难度而添加的随机数 (盐), 以及进行迭代计算次数。只要将盐和迭代次数都作为参数传递给 `PBEPParameterSpec` 类的构造器即可。

(5) 获取明文, 执行加密

```
byte ptext[]=s.getBytes("UTF8");
byte ctext[]=cp.doFinal(ptext);
```

分析: 和以前一样将字符串转换为字节数组, 并执行密码器的 `doFinal()` 方法进行加密。加密结果保存在字节数组 `ctext` 中。

★代码与分析:

```
import java.io.*;
import java.util.*;
import java.security.*;
import javax.crypto.*;
import javax.crypto.spec.*;

public class PBEEnc{
    public static void main(String args[]) throws Exception{
        String s="Hello World!";
        char[] passwd=args[0].toCharArray( );
        PBESpec pbks=new PBESpec(passwd);
        SecretKeyFactory kf=
            SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
        SecretKey k=kf.generateSecret(pbks);
        byte[] salt=new byte[8];
        Random r=new Random( );
        r.nextBytes(salt);
        Cipher cp=Cipher.getInstance("PBKDF2WithHmacSHA1");
        PBESpec ps=new PBESpec(salt,1000);
        cp.init(Cipher.ENCRYPT_MODE, k,ps);
        byte ptext[]=s.getBytes("UTF8");
        byte ctext[]=cp.doFinal(ptext);
        // 将盐和加密结果合并在一起保存为密文
        FileOutputStream f=new FileOutputStream("PBEEnc.dat");
        f.write(salt);
        f.write(ctext);
        // 打印盐的值
        for(int i=0;i<salt.length;i++){
            System.out.print(salt[i] +",");
        }
        System.out.println("");
    }
}
```

```

        // 打印加密结果
        for(int i=0;i<ctext.length;i++){
            System.out.print(ctext[i] +",");
        }
    }
}

```

基于口令的加密的密文由两部分组成，一个是盐，一个是加密结果，两个值简单地合并起来即可，本程序中将其一起写入密文文件 `PBEEnc.dat`。程序最后将盐和加密结果打印出来。

★运行程序

输入 `java PBEEnc s7es1.886` 来运行程序，其中命令行参数 `s7es1.886` 为用户选择的用于加密的口令。将输出：

```

76,26,126,-117,12,-98,-112,95,
113,-56,-69,66,-101,-1,-12,-109,90,-85,-99,66,-80,-10,-84,-77,

```

其中第一行 8 个数字对应的是盐的值，第二行为加密结果。由于程序每次运行时使用的盐的值不同，因此即使程序运行时每次使用的口令相同，加密后的结果也不一样。

程序运行后当前目录下将创建一个文件 `PBEEnc.dat`，该文件中存放的是密文。其中前 8 个字节是盐，剩余部分是加密结果。

2.4.2 基于口令的解密

★ 实例说明

本实例的输入 2.4.1 小节的存放密文的文件 `PBEEnc.dat`，以及该文件的密文所使用的口令“`s7es1.886`”。本实例将演示如何使用该口令对密文解密。

★ 编程思路：

和加密时一样，基于口令的解密也是使用 Java 的 `Cipher` 类，只是初始化时传入的参数使用 `Cipher.DECRYPT_MODE`。此外，由于密文中既包含盐也包含加密结果，因此需要将这两部分分离出来。

此外，加密时所用的密钥是根据给定的口令生成的。为了增加破解的难度，PBE 还使用一个随机数（称为盐）和口令组合起来加密文件。此外还进行重复计算（迭代）。编程的主要步骤如下：

（6） 读取口令并生成密钥

```

char[] passwd=args[0].toCharArray();
PBEKeySpec pbks=new PBEKeySpec(passwd);
SecretKeyFactory kf=
    SecretKeyFactory.getInstance("PBEMD5AndDES");
SecretKey k=kf.generateSecret(pbks);

```

分析：该步骤和加密时完全相同。

(7) 获取随机数 (盐)

```
byte[] salt=new byte[8];
FileInputStream f=new FileInputStream("PBEEnc.dat");
f.read(salt);
```

分析: 由于盐的长度固定, 为 8 个字节, 因此定义大小为 8 的字节数组, 从文件 PBEEnc.dat 中读取盐, 存放在数组 salt 中。

(8) 获取加密结果

```
int num=f.available();
byte[] ctext=new byte[num];
f.read(ctext);
```

分析: 由于 PBEEnc.dat 中剩余部分为加密结果, 因此使用文件输入流的 available() 方法判断剩余字节的数量, 并创建相应大小的字节数组, 读入数据。

(9) 创建密码器, 执行解密

```
Cipher cp=Cipher.getInstance("PBEWithMD5AndDES");
PBEPParameterSpec ps=new PBEPParameterSpec(salt,1000);
cp.init(Cipher.DECRYPT_MODE, k,ps);
byte ptext[]=cp.doFinal(ctext);
```

分析: 该步骤和加密时类似, 只是初始化时使用的是 Cipher.DECRYPT_MODE, 执行 doFinal() 时传入的是以前的加密结果, 而返回的字节数组 ptext 即包含了解密后的文字。

★代码与分析:

```
import java.io.*;
import java.util.*;
import java.security.*;
import javax.crypto.*;
import javax.crypto.spec.*;

public class PBEDec{
    public static void main(String args[]) throws Exception{
        char[] passwd=args[0].toCharArray( );
        PBEKeySpec pbks=new PBEKeySpec(passwd);
        SecretKeyFactory kf=
            SecretKeyFactory.getInstance("PBEWithMD5AndDES");
        SecretKey k=kf.generateSecret(pbks);

        byte[] salt=new byte[8];
        FileInputStream f=new FileInputStream("PBEEnc.dat");
        f.read(salt);

        int num=f.available();
        byte[] ctext=new byte[num];
        f.read(ctext);

        Cipher cp=Cipher.getInstance("PBEWithMD5AndDES");
```

```

PBESpec ps=new PBESpec(salt,1000);
cp.init(Cipher.DECRYPT_MODE, k,ps);
byte ptext[]=cp.doFinal(ctext);
// 显示解密结果
for(int i=0;i<ptext.length;i++){
    System.out.print(ptext[i] +",");
}
System.out.println("");
// 以字符串格式显示解密结果
for(int i=0;i<ptext.length;i++){
    System.out.print((char) ptext[i]);
}
}
}

```

程序最后将解密后得到的字节数组 `ptext` 中的内容打印出来，为了使显示出的结果更加直观，最后将字节数组 `ptext` 中的内容转换字符进行显示。

★运行程序

输入 `java PBEDec s7es1.886` 来运行程序，其中命令行参数 `s7es1.886` 是解密所使用的口令，必须和加密时使用的口令一样。程序将输出：

```

72,101,108,108,111,32,87,111,114,108,100,33,
Hello World!

```

如果使用的口令不对，将无法解密。如输入 `java PBEDec s7es1.888` 运行程序，将显示如下异常信息：

```

Exception in thread "main" javax.crypto.BadPaddingException: Given final block not properly padded
    at com.sun.crypto.provider.DESCipher.engineDoFinal(DashoA6275)
    at com.sun.crypto.provider.DESCipher.engineDoFinal(DashoA6275)
    at com.sun.crypto.provider.PBEWithMD5AndDESCipher.engineDoFinal(DashoA6275)
    at javax.crypto.Cipher.doFinal(DashoA6275)
    at PBEDec.main(PBEDec.java:27)

```

2.5 针对流的加密和解密

2.2 和 2.3 节的加密和解密都是针对字节数组进行的，但实际编程中更常针对流进行加密，如对整个文件进行加密/解密或对网络通信进行加密/解密等。尽管我们可以先从流中读出字节然后进行加密/解密，但使用 Java 中针对流提供的专门的类更加方便。本节介绍其基本编程方法。

2.5.1 针对输入流的解密和加密

★ 实例说明

本实例以最简单的程序演示了针对输入流的加密和解密，将指定文件中的内容进行加密和解密。

★ 编程思路：

Java 中 `CipherInputStream` 提供了针对输入流的加密和解密，执行加密和解密的算法仍旧由以前使用的 `Cipher` 类担当，`CipherInputStream` 类的构造器中可以指定标准的输入流（如文件输入流）和密码器（`Cipher` 对象），当使用 `CipherInputStream` 类的 `read()` 方法从流中读取数据时，会自动将标准输入流中的内容使用密码器进行加密或解密再读出。其基本步骤如下：

(1) 生成密钥

```
FileInputStream f=new FileInputStream("key1.dat");
ObjectInputStream ob=new ObjectInputStream(f);
Key k=(Key)ob.readObject();
```

分析：这里和 2.3.1 小节一样从文件中读取以前保存的密钥，这样保证了本实例所用的密钥和以前相同，以便于对比加密结果。如果不需要作对比，也可以使用 2.2.1 小节的步骤生成新的密钥。

(2) 创建并初始化密码器

```
Cipher cp=Cipher.getInstance("DESede");
cp.init(Cipher.ENCRYPT_MODE, k);
```

分析：该步骤和以前相同，如果准备进行解密，则应将 `Cipher.ENCRYPT_MODE` 改为 `Cipher.DECRYPT_MODE`。

(3) 创建要加密或解密的输入流

```
FileInputStream in=new FileInputStream(args[0]);
```

分析：这里以加密文件为例，因此创建文件输入流，文件名由命令行参数传入。

(4) 创建 `CipherInputStream` 对象

```
CipherInputStream cin=new CipherInputStream(in, cp);
```

分析：将第 2 步创建的密码器和第 3 步创建的需要加密/解密的流作为参数传递给 `CipherInputStream` 对象。

(5) 读取输入流

```
while( (b=cin.read()) !=-1 ){
    System.out.print((byte)b+",");
}
```

分析：像使用基本的输入流一样使用 `read()` 方法从 `CipherInputStream` 流中读取数据，则在读取过程中会自动根据第 2 步密码器中的设置进行加密或解密。

★代码与分析：

```
import java.io.*;
import java.security.*;
import javax.crypto.*;
public class StreamIn{
    public static void main(String args[]) throws Exception{
```



```

        FileInputStream f=new FileInputStream("key1.dat");
        ObjectInputStream ob=new ObjectInputStream(f);
        Key k=(Key)ob.readObject( );
        Cipher cp=Cipher.getInstance("DESede");
        cp.init(Cipher.ENCRYPT_MODE, k);
        FileInputStream in=new FileInputStream(args[0]);
        CipherInputStream cin=new CipherInputStream(in, cp);
        int b=0;
        while( (b=cin.read()) !=-1 ){
            System.out.print((byte)b+",");
        }
    }
}

```

★运行程序

在当前目录下使用 Windows 中的记事本创建一个文本文件：**StreamIn1.txt**，在其中输入需要加密的字符串，可以输入多行。为了和以前的加密结果进行对比，不妨先只输入一行“Hello World!”。

输入 `java StreamIn StreamIn1.txt` 来运行程序，程序将输出加密以后的内容：

-57,119,0,-45,-9,23,37,-56,-60,-34,-99,105,99,113,-17,76,

该结果和 2.3.1 小节的运行结果相同。

注意，本实例和 2.3.1 小节的运行结果相同的前提是使用的密钥相同（都从 **key1.dat** 文件中读取），算法相同（都是 **DESede** 算法及默认的填充和模式）以及相同的加密内容（都是“Hello World!”）。如果在编辑 **StreamIn1.txt** 文件时在“Hello World!”后面加了回车或使用其他的文本编辑器（如使用 DOS 下的 **edit** 工具可能会在文件末尾自动加上一些隐藏字符），则结果可能会不同。

本实例将加密的结果打印了出来，也可以再创建一个文件输出流，将加密结果保存起来，其内容将和 2.3.1 小节的 **SEnc.dat** 相同。

将该实例稍作修改就可以对以文件形式保存的密文进行解密。如果将程序中的

```
cp.init(Cipher.ENCRYPT_MODE, k);
```

改为

```
cp.init(Cipher.DECRYPT_MODE, k);
```

则可进行解密操作。此时可将 2.3.1 小节输出的加密文件 **SEnc.dat** 拷贝到当前目录，运行 `java StreamIn SEnc.dat`，程序将输出解密结果：

72,101,108,108,111,32,87,111,114,108,100,33,

此即“Hello World!”字节数组编码方式。若进一步将该实例中的

```
System.out.print((byte)b+",");
```

改为

```
System.out.print((char)b);
```

则进行解密时将直接输出“Hello World!”。

2.5.2 针对输出流的解密和加密

★ 实例说明

本实例演示了针对输出流的加密和解密，将指定文件中的内容进行加密和解密，并把加密和解密的结果输入指定的另外一个文件。

★ 编程思路：

和输入流类似，Java 中 `CipherOutputStream` 提供了针对输出流的加密和解密。`CipherOutputStream` 类的构造器中可以指定标准的输出流（如文件输出流）和密码器（`Cipher` 对象），当使用 `CipherOutputStream` 类的 `write()` 方法进行输出时，会自动将 `write()` 方法参数中的内容使用密码器进行加密或解密后再写入标准输出流。其基本步骤如下：

（1） 生成密钥

```
FileInputStream f=new FileInputStream("key1.dat");
ObjectInputStream ob=new ObjectInputStream(f);
Key k=(Key)ob.readObject();
```

分析：该步骤和 2.5.1 小节的第 1 步一样。

（2） 创建并初始化密码器

```
Cipher cp=Cipher.getInstance("DESede");
if(args[0].equals("dec"))
    cp.init(Cipher.DECRYPT_MODE, k);
else cp.init(Cipher.ENCRYPT_MODE, k);
```

分析：该步骤和 2.5.1 小节的第 2 步一样，但为了使程序更具有通用性，这里不妨通过命令行参数确定密码器是加密模式还是解密模式。当第一个命令行参数为 `enc` 时，使用加密模式，否则为解密模式。

（3） 获取要加密或解密的内容

```
FileInputStream in=new FileInputStream(args[1]);
```

分析：要加密或解密的内容可以是各种形式，只要可以转换为整型或字节数组形式即可。如可以是一个字符串。本实例以加密文件为例，因此创建文件输入流，文件名由命令行的第 2 个参数传入。

（4） 获取加密或解密的输出以及 `CipherOutputStream` 对象

```
FileOutputStream out=new FileOutputStream(args[2]);
CipherOutputStream cout=new CipherOutputStream(out, cp);
```

分析：加密和解密的结果可以输出到各种输出流中，本实例将加密结果保存为文件，因此创建文件输出流。将其和第 3 步创建的密码器一起作为参数传递给 `CipherOutputStream` 对象。

（5） 写输出流

```
while( (b=in.read())!=-1){
    cout.write(b);
}
```

分析：像使用基本的输出流一样使用 `write()` 方法向 `CipherOutputStream` 流中写数据（数据为需要加密的明文，本实例从文件中使用 `read()` 方法从文件中读取明文），则在写之前 `CipherOutputStream` 流会自动按照其参数中的密码器设置先进行加密或解密操作，然后再写入其参数中的输出流中。本实例

★代码与分析:

```
import java.io.*;
import java.security.*;
import javax.crypto.*;

public class StreamOut{

    public static void main(String args[]) throws Exception{
        FileInputStream f=new FileInputStream("key1.dat");
        ObjectInputStream ob=new ObjectInputStream(f);
        Key k=(Key)ob.readObject( );
        Cipher cp=Cipher.getInstance("DESede");

        if(args[0].equals("dec"))
            cp.init(Cipher.DECRYPT_MODE, k);
        else
            cp.init(Cipher.ENCRYPT_MODE, k);
        FileInputStream in=new FileInputStream(args[1]);
        FileOutputStream out=new FileOutputStream(args[2]);
        CipherOutputStream cout=new CipherOutputStream(out, cp);
        int b=0;
        while( (b=in.read())!=-1){
            cout.write(b);
        }

        cout.close();
        out.close();
        in.close();
    }
}
```

★运行程序

仍旧使用 2.5.1 小节的文本文件: StreamIn1.txt 进行试验, 输入:

```
java StreamOut enc StreamIn1.txt mytest.txt
```

来运行程序, 则将把 StreamIn1.txt 中的内容加密成为文件 mytest.txt。

若进一步运行:

```
java StreamOut dec mytest.txt mytest2.txt
```

则将文件 mytest.txt 中的密文解密为文件 mytest2.txt。打开 mytest2.txt, 可以看到解密后的明文 “Hello World! ”。解密时必须有加密时所用的完全相同的密钥才能正常运行。

和 2.5.1 小节一样, 被加密的文件可以不止一行。

2.5.1 和 2.5.2 小节都使用了文件输入/输出流, 也可针对其他的流进行加密和解密。此外, 密码器也使用基于口令的加密和解密。

2.6 加密方式的设定

2.3.1 小节的程序加密的字符串如果是“Hello123Hello123Hello123Hello123”（每 8 个字符相同），则加密后的结果如下：

```
-46,-71,65,-43,48,105,-52,-13,  
-46,-71,65,-43,48,105,-52,-13,  
-46,-71,65,-43,48,105,-52,-13,  
-46,-71,65,-43,48,105,-52,-13,  
51,82,-102,-119,76,5,60,-114,
```

可以看出加密结果每 8 个字节出现相同，这是因为数据在进行加密时其实不是一个一个字节进行加密，也不是一次处理加密字节，而是每 8 个字节（64 位）作为一组进行加密，有些算法一次处理 16 个字节或更多。默认情况下，每组之间独立进行加密，因此相同的明文分组得到的加密结果也相同。

2.5.1 和 2.5.2 的例子使用密钥进行加密时，当文件 `StreamIn1.txt` 的内容为“Hello123Hello123Hello123Hello123”（每 8 个字符相同），也同样具有规律性。使用其他加密方式可以解决这一问题，本节将介绍 CBC 加密方式。

2.6.1 使用 CBC 方式的加密

★ 实例说明

本实例演示了使用 CBC 加密方式以及初始向量进行加密和解密编程步骤。

★ 编程思路：

对明文分组的不同处理方式形成了不同的加密方式，本章前面各节的程序中没有指定加密方式，默认的加密方式是 ECB（Electronic Code Book），它对每个明文分组独立进行处理。所以明文若 8 个字节一组相同的话（如本节开头的“Hello123Hello123Hello123Hello123”），加密出的结果也是 8 个字节一组相同的。

另一种加密方式称为 CBC（Cipher Block Chaining），它先加密第一个分组，然后使用得到的密文加密第二个分组，加密第二个分组得到的密文再加密第三个分组，……。这样，即使两个分组相同，得到的密文也不同。剩下的问题是如果两个密文的开头 8 个字节相同，按照这种加密方式，只要使用的密钥相同，则每条密文的开头 8 个字节也将相同。为此，CBC 使用一个 8 个字节的随机数（称为初始向量，IV）来加密第一个分组，其作用类似于基于口令加密中的盐。

因此，使用 CBC 方式首先要生成初始向量，然后在获取密码器对象时通过 `getInstance()` 方法的参数设定加密方式，在密码器初始化时传入初始向量。具体步骤如下：

（1）生成密钥

```
FileInputStream f=new FileInputStream("key1.dat");  
ObjectInputStream ob=new ObjectInputStream(f);  
Key k=(Key)ob.readObject();
```

分析：该步骤和以前一样。

（2）生成初始向量

```

byte[] rand=new byte[8];
Random r=new Random();
r.nextBytes(rand);
IvParameterSpec iv=new IvParameterSpec(rand);

```

分析：该步骤前三条语句和 2.4.1 小节的第 3 步一样，生成随机数，第 4 条语句则使用该随机数得到代表初始向量的 IvParameterSpec 对象。

(3) 获取密码器

```
Cipher cp=Cipher.getInstance("DESede/CBC/PKCS5Padding");
```

分析：在获取密码器时，通过 getInstance() 方法的参数指定加密方式，该参数“DESede/CBC/PKCS5Padding”由三部分组成。

第一部分“DESede”代表所用的加密算法。由于本实例仍旧使用了 2.2.1 小节生成的密钥，因此这里必须仍旧使用 DESede 算法。若 2.2.1 小节改为其他的算法，如“DES”、“Blowfish”等，则这里也必须相应改变。

第二部分“CBC”即加密模式，除了 CBC 外，还有 NONE、ECB、CFB、OFB 和 PCBC 等可以用。

第三部分为填充模式，明文在被 64 位一组分成明文分组时，最后一个分组可能不足 64 位，因此加密算法一般使用一定规则对最后一个分组进行填充。对称加密常用的填充方式称为“PKCS#5 padding”，其中的 PKCS 是 Public Key Cryptography Standard 的缩写。如果加密算法不进行填充（填充方式为 No padding），则要求明文长度必须是 64 的整数倍。在本章前面各节的程序中没有指定填充方式，默认的填充方式就是“PKCS#5 padding”，因此以前的语句 Cipher.getInstance("DESede") 和 Cipher.getInstance("DESede/ECB/PKCS5Padding") 是等价的。在本节的开头介绍加密字符串“Hello123Hello123Hello123Hello123”时，输出结果最后多出的“51, 82, -102, -119, 76, 5, 60, -114,”就是由于填充的结果（使用 PKCS#5 padding 时，即使明文长度是 8 字节的整数倍，也会再数据最后加上一个完整的填充块。

(4) 初始化密码器，并执行加密

```

cp.init(Cipher.ENCRYPT_MODE, k, iv);
byte ptext[]=s.getBytes("UTF8");
byte ctext[]=cp.doFinal(ptext);

```

分析：和前面的程序相比，在其参数中增加了一项初始化向量，即第 2 步得到的 iv。执行加密时同样使用 doFinal() 方法对字节数组进行加密。

★代码与分析：

```

import java.io.*;
import java.util.*;
import java.security.*;
import javax.crypto.*;
import javax.crypto.spec.*;

public class SEncCBC{

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

        String s="Hello123Hello123Hello123Hello123";

        //获取密钥

        FileInputStream f1=new FileInputStream("key1.dat");
        ObjectInputStream b=new ObjectInputStream(f1);
    }
}

```

```

        Key k=(Key)b.readObject( );
        // 生成初始化向量
        byte[] rand=new byte[8];
        Random r=new Random( );
        r.nextBytes(rand);
        IvParameterSpec iv=new IvParameterSpec(rand);
        //加密
        Cipher cp=Cipher.getInstance("DESede/CBC/PKCS5Padding");
        cp.init(Cipher.ENCRYPT_MODE, k, iv);
        byte ptext[]=s.getBytes("UTF8");
        byte ctext[]=cp.doFinal(ptext);
        //打印加密结果
        for(int i=0;i<ctext.length;i++){
            System.out.print(ctext[i] +",");
        }
        //保存加密结果
        FileOutputStream f2=new FileOutputStream("SEncCBC.dat");
        f2.write(rand);
        f2.write(ctext);
    }
}

```

为了方便看到加密结果，程序中通过循环打印出字节数组的内容。为了以后进行解密，程序中通过文件将初始化向量和加密结果保存在一起。

★运行程序

输入 java SEncCBC 运行程序，得到如下结果：

47,-79,65,-41,25,-70,-62,-55,3,10,-3,118,-12,100,-113,2,124,-66,-84,93,-74,8,17,64,-80,-82,2
9,126,-23,-102,6,-98,-85,-110,-64,10,-23,-82,-30,-80,

再运行一次，得到如下结果：

118,-63,110,81,21,-99,44,-17,29,59,-121,-27,80,40,-89,-37,74,-117,-110,52,33,54,85,85,94,1
21,-122,125,29,-39,11,-71,-80,-99,-50,0,22,-50,-72,-12,

可见明文有规律性时，密文并无规律性，而且相同的明文加密后的结果不同。

密文保存在文件“SEncCBC.dat”中，其中前 8 个字节为该密文对应的初始化向量。

2.6.2 使用 CBC 方式的解密

★ 实例说明

本实例演示了如何对 2.6.1 小节的密文进行解密。

★ 编程思路：

同样加密一样，先要获取加密时所用的初始向量。由于 2.6.1 小节将初始化向量保存在文件 SEncCBC.dat 的开头 8 个字节中，因此可直接使用文件输入流读取。进而读取密文和密钥，

最后在获取密码器对象时通过 `getInstance()` 方法的参数设定加密方式，在密码器初始化时传入初始向量。具体步骤如下：

(1) 获取初始向量

```
FileInputStream f=new FileInputStream("SEncCBC.dat");
byte[] rand=new byte[8];
f.read(rand);
IvParameterSpec iv=new IvParameterSpec(rand);
```

分析：使用文件输入流的 `read()` 方法从文件 `SEncCBC.dat` 中读取 8 个字节的对应初始向量的随机数，并用其创建 `IvParameterSpec` 对象。

(2) 获取密文和密钥

```
int num=f.available();
byte[] ctext=new byte[num];
f.read(ctext);
FileInputStream f2=new FileInputStream("key1.dat");
ObjectInputStream b=new ObjectInputStream(f2);
Key k=(Key)b.readObject();
```

分析：由于 `SEncCBC.dat` 中剩余部分为加密结果，因此使用文件输入流的 `available()` 方法判断剩余字节的数量，并创建相应大小的字节数组，读入数据。密钥必须和 2.6.1 小节所用的密钥相同。

(3) 获取并初始化密码器

```
Cipher cp=Cipher.getInstance("DESede/CBC/PKCS5Padding");
cp.init(Cipher.DECRYPT_MODE, k, iv);
byte[] ptext=cp.doFinal(ctext);
```

分析：该步骤和 2.6.1 小节相同，只是在初始化密码器时使用 `Cipher.DECRYPT_MODE`，表明进行解密。

★代码与分析：

```
import java.io.*;
import java.security.*;
import javax.crypto.*;
import javax.crypto.spec.*;

public class SDecCBC{
    public static void main(String args[]) throws Exception{
        // 获取初始向量
        FileInputStream f=new FileInputStream("SEncCBC.dat");
        byte[] rand=new byte[8];
        f.read(rand);
        IvParameterSpec iv=new IvParameterSpec(rand);
        //获取密文
        int num=f.available();
        byte[] ctext=new byte[num];
```

```

        f.read(ctext);
        //获取密钥
        FileInputStream f2=new FileInputStream("key1.dat");
        ObjectInputStream b=new ObjectInputStream(f2);
        Key k=(Key)b.readObject( );
        //获取密码器，执行加密
        Cipher cp=Cipher.getInstance("DESede/CBC/PKCS5Padding");
        cp.init(Cipher.DECRYPT_MODE, k, iv);
        byte []ptext=cp.doFinal(ctext);
        String p=new String(ptext,"UTF8");
        System.out.println(p);
    }
}

```

程序中最后将明文生成字符串加以显示。

★运行程序

输入 `java SDecCBC` 运行程序，得到如下结果：

Hello123Hello123Hello123Hello123

解密成功。

同样，对 2.5 节中的例子也可以类似地使用 CBC 加密方式。

2.7 生成非对称加密的公钥和私钥

本章前面几节的程序中，加密和解密使用的是同一个密钥，这种方式称为对称加密。使用对称密钥时，若 A 想让 B 向其秘密传送信息，A 必须先将密钥提供给 B，或者由 B 将密钥提供给 A。如果在传递密钥过程中密钥被窃取，则 A 和 B 之间的通信就不再安全了。

非对称加密解决了这一问题。它将加密的密钥和解密的密钥分开。A 事先生成一对密钥，一个用于加密，称为公钥（公钥），一个用于解密，称为私钥。由于产生这一对密钥的一些数学特性，公钥加密的信息只能用私钥解密。这样，A 只要将公钥对外公开，不论谁就可以使用这个公钥给 A 发送秘密信息了。A 接收到加密信息后可以用私钥打开。由于只需要传递公钥，而公钥只能加密不能解密，因此即使攻击者知道了公钥也无济于事。

本节以 RSA 算法为例介绍 Java 中如何生成公钥和私钥。

★ 实例说明

本实例演示了如何使用 Java 中定义好的类创建 RSA 公钥和私钥。

★ 编程思路：

Java 的 `KeyPairGenerator` 类提供了一些方法来创建密钥对以便用于非对称加密，密钥对创建好后封装在 `KeyPair` 类型的对象中，在 `KeyPair` 类中提供了获取公钥和私钥的方法。具体步

骤如下:

(1) 创建密钥对生成器

```
KeyPairGenerator kpg=KeyPairGenerator.getInstance("RSA");
```

分析: 密钥对生成器即 KeyPairGenerator 类型的对象, 和 2.2.1 小节的第 1 步中介绍的 KeyGenerator 类一样, KeyPairGenerator 类是一个工厂类, 它通过其中预定义的一个静态方法 getInstance () 获取 KeyPairGenerator 类型的对象。getInstance () 方法的参数是一个字符串, 指定非对称加密所使用的算法, 常用的有 RSA, DSA 等。

(2) 初始化密钥生成器

```
kpg.initialize(1024);
```

分析: 对于密钥长度。对于 RSA 算法, 这里指定的其实是 RSA 算法中所用的模的位数。可以在 512 到 2048 之间。

(3) 生成密钥对

```
KeyPair kp=kpg.genKeyPair();
```

分析: 使用 KeyPairGenerator 类的 genKeyPair () 方法生成密钥对, 其中包含了一对公钥和私钥的信息。

(4) 获取公钥和私钥

```
PublicKey pbkey=kp.getPublic();
```

```
PrivateKey prkey=kp.getPrivate();
```

分析: 使用 KeyPair 类的 getPublic () 和 getPrivate () 方法获得公钥和私钥对象。

★代码与分析:

```
import java.io.*;
import java.security.*;
import javax.crypto.*;
import javax.crypto.spec.*;

public class Skey_RSA{
    public static void main(String args[]) throws Exception{
        KeyPairGenerator kpg=KeyPairGenerator.getInstance("RSA");
        kpg.initialize(1024);
        KeyPair kp=kpg.genKeyPair();
        PublicKey pbkey=kp.getPublic();
        PrivateKey prkey=kp.getPrivate();
        // 保存公钥
        FileOutputStream f1=new FileOutputStream("Skey_RSA_pub.dat");
        ObjectOutputStream b1=new ObjectOutputStream(f1);
        b1.writeObject(pbkey);
        // 保存私钥
        FileOutputStream f2=new FileOutputStream("Skey_RSA_priv.dat");
        ObjectOutputStream b2=new ObjectOutputStream(f2);
        b2.writeObject(prkey);
    }
}
```

```
}
```

分析：本实例和 2.2.1 小节一样使用对象流将密钥保存在文件中，所不同的是加密所用的公钥和解密所用的私钥分开保存。将公钥对外公布，供其他人加密使用，而把私钥秘密保存，在需要解密时使用。

★运行程序

输入 `java Skey_RSA` 运行程序，当前目录下将生成两个文件：`Skey_RSA_pub.dat` 和 `Skey_RSA_priv.dat`，前者保存着公钥，后者保存着私钥。将文件 `Skey_RSA_pub.dat` 对外公布（如放在 Web 服务器上给大家下载，或者直接拷贝给所有需要的人），而 `Skey_RSA_priv.dat` 秘密保存。

2.8 使用 RSA 算法进行加密和解密

2.7 节的程序创建了 RSA 的公钥和密钥，本节使用公钥进行加密，然后使用私钥对加密的信息进行解密。

2.8.1 使用 RSA 公钥进行加密

★ 实例说明

本实例以加密一串最简单的字符串“Hello World!”为例，演示了如何使用 2.7 节生成的 RSA 公钥文件 `Skey_RSA_pub.dat` 进行加密。

★ 编程思路：

使用 RSA 公钥进行加密的代码和 2.3.1 小节使用 DESede 进行加密其实没什么大的区别，只是 Cipher 类的 `getInstance()` 方法的参数中应该指定使用 RSA。但由于 J2SDK1.4 中只实现了 RSA 密钥的创建，没有实现 RSA 算法，因此需要安装其他加密提供者软件才能直接使用 Cipher 类执行加密解密。其实有了 RSA 公钥和私钥后，自己编写程序从底层实现 RSA 算法也并不复杂。本实例给出简单的例子实现了 RSA 加密，使读者只使用 J2SDK1.4 便能直观地了解非对称加密算法。

RSA 算法是使用整数进行加密运算的，在 RSA 公钥中包含了两个信息：公钥对应的整数 e 和用于取模的整数 n 。对于明文数字 m ，计算密文的公式是： $m^e \bmod n$ 。因此，编程步骤如下：

（1） 获取公钥

```
FileInputStream f=new FileInputStream("Skey_RSA_pub.dat");
ObjectInputStream b=new ObjectInputStream(f);
RSAPublicKey pbk=(RSAPublicKey)b.readObject();
```

分析：从 2.7 节生成的公钥文件 `Skey_RSA_pub.dat` 中读取公钥，由于 2.7 节使用的是 RSA 算法，因此从文件读取公钥对象后强制转换为 `RSAPublicKey` 类型，以便后面读取 RSA 算法所需要的参数。

（2） 获取公钥的参数(e, n)

```
BigInteger e=pbk.getPublicExponent();
BigInteger n=pbk.getModulus();
```

分析: 使用 `RSAPublicKey` 类的 `getPublicExponent()` 和 `getModulus()` 方法可以分别获得公钥中 `e` 和 `n` 的值。由于密钥很长, 因此对应的整数值非常大, 无法使用一般的整型来存储, Java 中定义了 `BigInteger` 类来存储这类很大的整数并可进行各种运算。

(3) 获取明文整数(m)

```
String s="Hello World!";
byte ptext[]=s.getBytes("UTF8");
BigInteger m=new BigInteger(ptext);
```

分析: 明文是一个字符串, 为了用整数表达这个字符串, 先使用字符串的 `getBytes()` 方法将其转换为 `byte` 类型数组, 它其实是字符串中各个字符的二进制表达方式, 这一串二进制数转换为一个整数将非常大, 因此仍旧使用 `BigInteger` 类将这个二进制串转换为整型。

本实例中出于简化, 将整个字符串转换为一个整数。实际使用中, 应该对明文进行分组, 因为 RSA 算法要求整型数 `m` 的值必须小于 `n`。

(4) 执行计算

```
BigInteger c=m.modPow(e,n);
```

分析: 计算前面的公式: $m^e \bmod n$ 。`BigInteger` 类中已经提供了方法 `modPow()` 来执行这个计算。底数 `m` 执行这个方法, 方法 `modPow()` 的第一个参数即指数 `e`, 第二个参数即模 `n`。方法返回的结果即公式 $m^e \bmod n$ 的计算结果, 即密文。

★代码与分析:

```
import java.security.*;
import java.security.spec.*;
import javax.crypto.*;
import javax.crypto.spec.*;
import javax.crypto.interfaces.*;
import java.security.interfaces.*;
import java.math.*;
import java.io.*;

public class Enc_RSA{
    public static void main(String args[]) throws Exception{
        String s="Hello World!";
        // 获取公钥及参数 e,n
        FileInputStream f=new FileInputStream("Skey_RSA_pub.dat");
        ObjectInputStream b=new ObjectInputStream(f);
        RSAPublicKey pbk=(RSAPublicKey)b.readObject();
        BigInteger e=pbk.getPublicExponent();
        BigInteger n=pbk.getModulus();
        System.out.println("e= "+e);
        System.out.println("n= "+n);
        // 明文 m
        byte ptext[]=s.getBytes("UTF8");
        BigInteger m=new BigInteger(ptext);
        // 计算密文 c,打印
```

```

        BigInteger c=m.modPow(e,n);
        System.out.println("c= "+c);
        // 保存密文
        String cs=c.toString( );
        BufferedWriter out=
            new BufferedWriter(new OutputStreamWriter(
                new FileOutputStream("Enc_RSA.dat")));
        out.write(cs,0,cs.length( ));
        out.close( );
    }
}

```

程序最后将密文 c 打印出来，并以字符串形式保存在文件中。

★运行程序

输入 `java Enc_RSA` 运行程序，得到如下结果：

```

C:\java\CH2\Enc_RSA>java Enc_RSA
e= 65537
n= 12278026029167330255684085638332998034595810793059889266144708407656004675547
95494513994068004894113238783212203782558140726974693251862483680903933425972120
92885441337336889587786916466479793273008500769901072255068076061285938341180207
193099147886262148428123992861455851470831159153285863290922313423404207
c= 7817208477290738177732545166761801677594300637882594100507410994733307796185
96799697472380565136492799068789639118776480800477079303545255422168127523800784
61770907282147001717335465215074399112623105472212347575803155049577155483939409
45600948778550907949519664073118334208158642041604326438155965403385576
C:\java\CH2\Enc_RSA>_

```

其中显示了公钥中的参数以及加密的结果 c ，这些都是很大的整数， n 和 c 多达上百位。程序运行后密文 c 以字符串形式保存在文件 `Enc_RSA.dat` 中。

2.8.2 使用 RSA 私钥进行解密

★ 实例说明

本实例使用 2.7 节生成的私钥文件 `Skey_RSA_priv.dat`，对 2.8.1 小节生成的密文文件 `Enc_RSA.dat` 进行解密。

★ 编程思路：

和 2.8.1 小节类似，使用 RSA 私钥进行解密的代码也可以在 `Cipher` 类的 `getInstance()` 方法的参数中指定使用 `RSA`，使用解密模式进行解密。但需要安装其他加密提供者软件才能直接使用 `Cipher` 类执行加密解密。本实例给出简单的例子从底层实现 `RSA` 解密，以便只使用 `J2SDK1.4` 便能直观地了解非对称加密算法。

`RSA` 算法的解密和加密类似，在 `RSA` 私钥中包含了两个信息：私钥对应的整数 d 和用于取模的整数 n 。其中的 n 和加密时的 n 完全相同。对于密文数字 c ，计算明文的公式是： $c^d \bmod n$ ，之所以加密时由公式 $m^e \bmod n$ 得到的密文 c 通过这个公式计算一下就可以反过来得到原

来的明文 m ，有其本身的数学规律决定。从编程角度只需要知道这个结果就行了。编程步骤如下：

(1) 读取密文

```
BufferedReader in=
    new BufferedReader(new InputStreamReader(
        new FileInputStream("Enc_RSA.dat")));
String ctext=in.readLine();
BigInteger c=new BigInteger(ctext);
```

分析：从 2.8.1 小节生成的密文文件 Enc_RSA.dat 中读取密文，由于 2.8.1 小节保存的只是一行字符串，因此只要一条 readLine() 语句即可。由于这一行字符串表示的是一个很大的整型数，因此使用 BigInteger 类来表示这个整型数。

(2) 获取私钥

```
FileInputStream f=new FileInputStream("Skey_RSA_priv.dat");
ObjectInputStream b=new ObjectInputStream(f);
RSAPrivateKey prk=(RSAPrivateKey)b.readObject( );
```

分析：从 2.7 节生成的私钥文件 Skey_RSA_priv.dat 中读取公钥，由于 2.7 节使用的是 RSA 算法，因此从文件读取公钥对象后强制转换为 RSAPrivateKey 类型，以便后面读取 RSA 算法所需要的参数。

(3) 获取私钥的参数(d, n)

```
BigInteger d=prk.getPrivateExponent( );
BigInteger n=prk.getModulus( );
```

分析：使用 RSAPrivateKey 类的 getPrivateExponent() 和 getModulus() 方法可以分别获得公钥中 d 和 n 的值。

(4) 执行计算

```
BigInteger m=c.modPow(d,n);
```

分析：使用 BigInteger 的 modPow() 方法计算前面的公式： $c^d \bmod n$ 。方法返回的结果即公式 $c^d \bmod n$ 的计算结果，即明文对应的整型数 m。

(5) 计算明文整型数对应的字符串

```
byte[] mt=m.toByteArray();
for(int i=0;i<mt.length;i++){
    System.out.print((char) mt[i]);
}
```

分析：RSA 算法解密的结果 m 是一个很大的整数，为了计算出其对应的字符串的值，先使用 BigInteger 类的 toByteArray() 方法得到代表该整型数的字节数组，然后将数组中每个元素转换为字符，组成字符串。

★代码与分析：

```
import java.security.*;
import java.security.spec.*;
import javax.crypto.*;
import javax.crypto.spec.*;
import javax.crypto.interfaces.*;
import java.security.interfaces.*;
import java.math.*;
```

```

import java.io.*;

public class Dec_RSA{

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

        //读取密文
        BufferedReader in=
            new BufferedReader(new InputStreamReader(
                new FileInputStream("Enc_RSA.dat")));

        String ctext=in.readLine();

        BigInteger c=new BigInteger(ctext);

        //读取私钥
        FileInputStream f=new FileInputStream("Skey_RSA_priv.dat");
        ObjectInputStream b=new ObjectInputStream(f);
        RSAPrivateKey prk=(RSAPrivateKey)b.readObject( );
        BigInteger d=prk.getPrivateExponent();

        //获取私钥参数及解密
        BigInteger n=prk.getModulus();
        System.out.println("d= "+d);
        System.out.println("n= "+n);
        BigInteger m=c.modPow(d,n);

        //显示解密结果
        System.out.println("m= "+m);
        byte[] mt=m.toByteArray();
        System.out.println("PlainText is ");
        for(int i=0;i<mt.length;i++){
            System.out.print((char) mt[i]);
        }
    }
}

```

★运行程序

输入 java Dec_RSA 运行程序，得到如下结果：

```

C:\java\CH2\Dec_RSA>java Dec_RSA
d= 1165173185922506275725187332637262126376934947378681894225558073616060476968
80246056477212345642682435462152674093297236742198514916861243420512675824265625
24133170009455449567786642551051787441248599181114533154021791285659796179155748
943064802258579746445046827197361029871209773007013353718920416379669345
n= 12278026029167330255684085638332998034595810793059889266144708407656004675547
95494513994068004894113238783212203782558140726974693251862483680903933425972120
9288544133733688958778691646647979327300850076990107225506807606128593834118020
193099147886262148428123992861455851470831159153285863290922313423404207
m= 22405534230753928650781647905
PlainText is
Hello World!
C:\java\CH2\Dec_RSA>

```

其中显示了私钥中的参数以及解密的结果，其中 n 的值和 2.8.1 小节 n 的值完全相同，

整型的明文转换后显示出字符串“Hello World!”。

2.9 使用密钥协定创建共享密钥

非对称加密解决了密钥分发的难题，但其计算量比对称密钥大，因此一般并不使用非对称加密加密大量数据。常见的做法是：主要数据通过对称密钥加密，而使用非对称加密来分发对称密钥。将两者的优势结合了起来。

例如若 A 和 B 之间想秘密传送大量数据，一方（如 A）先创建公钥和私钥对，公钥对外公布，另一方（如 B）创建对称密钥，然后使用公钥加密对称密钥，传递给 A，A 收到后用私钥解密，得到对称密钥，以后 A 和 B 之间就可以使用对称密钥加密通信了。

除了这种方式以外，还可以使用密钥协定来交换对称密钥。执行密钥协定的标准算法是 DH 算法（Diffie-Hellman 算法），本节介绍在 Java 中如何使用 DH 算法来交换共享密钥。

2.9.1 创建 DH 公钥和私钥

★ 实例说明

DH 算法是建立在 DH 公钥和私钥的基础上的，A 需要和 B 共享密钥时，A 和 B 各自生成 DH 公钥和私钥，公钥对外公布而私钥各自秘密保存。本实例将介绍 Java 中如何创建并部署 DH 公钥和私钥，以便后面一小节利用它创建共享密钥。

★ 编程思路：

在 2.7 节中使用了 `KeyPairGenerator` 类创建 RSA 公钥和私钥，本节也一样，只是其参数中指定“DH”，此外在初始化时需要为 DH 指定特定的参数。具体步骤如下：

（1）生成 DH 参数

```
DHParameterSpec DHP=  
    new DHParameterSpec(skip1024Modulus,skip1024Base);
```

分析：和 RSA 算法类似，DH 算法涉及到一些指数和取模运算，DH 参数指定 A、B 双方在创建 DH 密钥时所公用的基数和模，Java 中 `DHParameterSpec` 类可以定义 DH 参数，其构造器的第一个参数指定模，第二个参数指定基数。模和基数的取值在 Internet 协议简单密钥管理（SKIP）标准中已经标准化，在安装 J2SDK1.4 后，计算机 C 盘中 `C:\j2sdk-1.4.0-doc\docs\guide\security\jce\JCERefGuide.html` 文件也包含了密钥长度为 1024 的 DH 密钥中模和基数的定义，可以直接拷贝下来使用，在 `JCERefGuide.html` 文件中查找“1024 bit Diffie-Hellman modulus”注释语句，将其下的 `skip1024ModulusBytes[]` 数组以及 `BigInteger` 类型的 `skip1024Modulus` 和 `skip1024Base` 变量拷贝下来即可。在本小节的“代码与分析”中也给出了完整的代码。此外 DH 密钥长度也可以是 512 或 2048 位。

（2）创建密钥对生成器

```
KeyPairGenerator kpg= KeyPairGenerator.getInstance("DH");
```

分析：密钥对生成器即 `KeyPairGenerator` 类型的对象，和 2.7 节一样，通过其中预定义的一个静态方法 `getInstance()` 获取 `KeyPairGenerator` 类型的对象。`getInstance()` 方法的参数指定为“DH”。

（3）初始化密钥生成器

```
kpg.initialize(DHP);
```

分析: 初始化时使用的参数即第 1 步中生成的参数。

(4) 生成密钥对, 获取公钥和私钥

```
KeyPair kp=kpg.genKeyPair();
```

```
PublicKey pbk=kp.getPublic();
```

```
PrivateKey prk=kp.getPrivate();
```

分析: 和 2.7 节一样, 使用使用 KeyPairGenerator 类的 genKeyPair() 方法生成密钥对, 进而使用密钥对的 getPublic() 和 getPrivate() 获取公钥和私钥。

★代码与分析:

```
import java.io.*;
import java.math.*;
import java.security.*;
import java.security.spec.*;
import javax.crypto.*;
import javax.crypto.spec.*;
import javax.crypto.interfaces.*;

public class Key_DH{
    //三个静态变量的定义从
    // C:\jdk-1.4.0-doc\docs\guide\security\jce\JCERefGuide.html
    // 拷贝而来
    // The 1024 bit Diffie-Hellman modulus values used by SKIP
    private static final byte skip1024ModulusBytes[] = {
        (byte)0xF4, (byte)0x88, (byte)0xFD, (byte)0x58,
        (byte)0x4E, (byte)0x49, (byte)0xDB, (byte)0xCD,
        (byte)0x20, (byte)0xB4, (byte)0x9D, (byte)0xE4,
        (byte)0x91, (byte)0x07, (byte)0x36, (byte)0x6B,
        (byte)0x33, (byte)0x6C, (byte)0x38, (byte)0x0D,
        (byte)0x45, (byte)0x1D, (byte)0x0F, (byte)0x7C,
        (byte)0x88, (byte)0xB3, (byte)0x1C, (byte)0x7C,
        (byte)0x5B, (byte)0x2D, (byte)0x8E, (byte)0xF6,
        (byte)0xF3, (byte)0xC9, (byte)0x23, (byte)0xC0,
        (byte)0x43, (byte)0xF0, (byte)0xA5, (byte)0x5B,
        (byte)0x18, (byte)0x8D, (byte)0x8E, (byte)0xBB,
        (byte)0x55, (byte)0x8C, (byte)0xB8, (byte)0x5D,
        (byte)0x38, (byte)0xD3, (byte)0x34, (byte)0xFD,
        (byte)0x7C, (byte)0x17, (byte)0x57, (byte)0x43,
        (byte)0xA3, (byte)0x1D, (byte)0x18, (byte)0x6C,
        (byte)0xDE, (byte)0x33, (byte)0x21, (byte)0x2C,
        (byte)0xB5, (byte)0x2A, (byte)0xFF, (byte)0x3C,
        (byte)0xE1, (byte)0xB1, (byte)0x29, (byte)0x40,
        (byte)0x18, (byte)0x11, (byte)0x8D, (byte)0x7C,
        (byte)0x84, (byte)0xA7, (byte)0x0A, (byte)0x72,
```



```

        (byte)0xD6, (byte)0x86, (byte)0xC4, (byte)0x03,
        (byte)0x19, (byte)0xC8, (byte)0x07, (byte)0x29,
        (byte)0x7A, (byte)0xCA, (byte)0x95, (byte)0x0C,
        (byte)0xD9, (byte)0x96, (byte)0x9F, (byte)0xAB,
        (byte)0xD0, (byte)0x0A, (byte)0x50, (byte)0x9B,
        (byte)0x02, (byte)0x46, (byte)0xD3, (byte)0x08,
        (byte)0x3D, (byte)0x66, (byte)0xA4, (byte)0x5D,
        (byte)0x41, (byte)0x9F, (byte)0x9C, (byte)0x7C,
        (byte)0xBD, (byte)0x89, (byte)0x4B, (byte)0x22,
        (byte)0x19, (byte)0x26, (byte)0xBA, (byte)0xAB,
        (byte)0xA2, (byte)0x5E, (byte)0xC3, (byte)0x55,
        (byte)0xE9, (byte)0x2F, (byte)0x78, (byte)0xC7
    };
    // The SKIP 1024 bit modulus
    private static final BigInteger skip1024Modulus
        = new BigInteger(1, skip1024ModulusBytes);
    // The base used with the SKIP 1024 bit modulus
    private static final BigInteger skip1024Base = BigInteger.valueOf(2);
    public static void main(String args[ ]) throws Exception{
        DHParameterSpec DHP=
            new DHParameterSpec(skip1024Modulus,skip1024Base);

        KeyPairGenerator kpg= KeyPairGenerator.getInstance("DH");
        kpg.initialize(DHP);
        KeyPair kp=kpg.genKeyPair();

        PublicKey pbk=kp.getPublic();
        PrivateKey prk=kp.getPrivate();
        // 保存公钥
        FileOutputStream f1=new FileOutputStream(args[0]);
        ObjectOutputStream b1=new ObjectOutputStream(f1);
        b1.writeObject(pbk);
        // 保存私钥
        FileOutputStream f2=new FileOutputStream(args[1]);
        ObjectOutputStream b2=new ObjectOutputStream(f2);
        b2.writeObject(prk);
    }
}

```

程序最后将公钥和私钥以对象流的形式保存在文件中，文件名通过命令行参数指定，第一个命令行参数对应的文件保存公钥，第二个命令行参数对应的文件保存私钥。

★运行程序

建立两个目录 A 和 B，模拟需要秘密通信的 A、B 双方，由于 DH 算法需要 A 和 B 各

自生成 DH 公钥和私钥，因此在这两个目录下都拷贝编译后文件 Key_DH。

首先由 A 创建自己的公钥和私钥，即在 A 目录下输入“java Key_DH Apub.dat Apri.dat”运行程序，这时在目录 A 下将产生文件 Apub.dat 和 Apri.dat，前者保存着 A 的公钥，后者保存着 A 的私钥。

然后由 B 创建自己的公钥和私钥，即在 B 目录下输入“java Key_DH Bpub.dat Bpri.dat”运行程序，这时在目录 B 下将产生文件 Bpub.dat 和 Bpri.dat，前者保存着 B 的公钥，后者保存着 B 的私钥。

最后发布公钥，A 将 Apub.dat 拷贝到 B 目录，B 将 Bpub.dat 拷贝到 A 的目录。

这样，A、B 双方的 DH 公钥和私钥已经创建并部署完毕。

2.9.2 创建共享密钥

★ 实例说明

DH 算法中，A 可以用自己的密钥和 B 的公钥按照一定方法生成一个密钥，B 也可以用自己的密钥和 A 的公钥按照一定方法生成一个密钥，由于一些数学规律，这两个密钥完全相同。这样，A 和 B 间就有了一个共同的密钥可以用于各种加密。本实例介绍 Java 中在上一小节的基础上如何利用 DH 公钥和私钥各自创建共享密钥。

★ 编程思路：

Java 中 KeyAgreement 类实现了密钥协定，它使用 init() 方法传入自己的私钥，使用 doPhase() 方法传入对方的公钥，进而可以使用 generateSecret() 方法生成共享的信息具体步骤如下：

- (1) 读取自己的 DH 私钥和对方的 DH 公钥

```
FileInputStream f1=new FileInputStream(args[0]);
ObjectInputStream b1=new ObjectInputStream(f1);
PublicKey pbk=(PublicKey)b1.readObject();
FileInputStream f2=new FileInputStream(args[1]);
ObjectInputStream b2=new ObjectInputStream(f2);
PrivateKey prk=(PrivateKey)b2.readObject();
```

分析：和 2.3.1 小节类似，从文件中获取密钥。只是分为公钥和私钥两个文件，通过命令行参数传入公钥和私钥文件名，第一个命令行参数为对方的公钥文件名，第二个命令行参数为自己的私钥文件名。

- (2) 创建密钥协定对象

```
KeyAgreement ka=KeyAgreement.getInstance("DH");
```

分析：密钥协定对象即 KeyAgreement 类型的对象，和 KeyPairGenerator 类类似，KeyAgreement 类是一个工厂类，通过其中预定义的一个静态方法 getInstance() 获取 KeyAgreement 类型的对象。getInstance() 方法的参数指定为“DH”。

- (3) 初始化密钥协定对象

```
ka.init(prk);
```

分析：执行密钥协定对象的 init() 方法，传入第 1 步获得的自己的私钥，它在第 1 步中通过第 2 个命令行参数提供。

- (4) 执行密钥协定

```
ka.doPhase(pbk,true);
```

分析：执行密钥协定对象的 doPhase () 方法，其第一个参数中传入对方的公钥。在本实例中，只有 A、B 两方需要共享密钥，因此对方只有一个，因此第二个参数设置为 true。如果有 A、B、C 三方需要共享密钥，则对方有两个，doPhase () 方法要写两次，每次在第 1 个参数中传入一个公钥，第 2 个参数最初设置为 false，最后一次设置为 true。例如 C 方应该执行 ka.doPhase(pbk_of_A,false); ka.doPhase(pbk_of_B,true);。一次类推，可以用密钥协定实现多方共享一个密钥。

(5) 生成共享信息

```
byte[] sb=ka.generateSecret();
```

分析：执行密钥协定对象的 generateSecret () 方法，返回字节类型的数组。A、B 双方得到的该数组的内容完全相同，用它创建密钥也各方完全相同。如可使用 SecretKeySpec k=new SecretKeySpec(sb,"DESede");创建密钥。

★代码与分析：

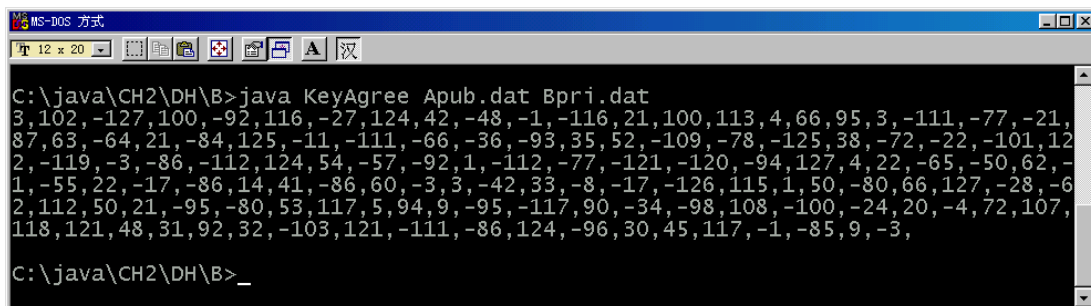
```
import java.io.*;
import java.math.*;
import java.security.*;
import java.security.spec.*;
import javax.crypto.*;
import javax.crypto.spec.*;
import javax.crypto.interfaces.*;

public class KeyAgree{
    public static void main(String args[] ) throws Exception{
        // 读取对方的 DH 公钥
        FileInputStream f1=new FileInputStream(args[0]);
        ObjectInputStream b1=new ObjectInputStream(f1);
        PublicKey pbk=(PublicKey)b1.readObject( );
        //读取自己的 DH 私钥
        FileInputStream f2=new FileInputStream(args[1]);
        ObjectInputStream b2=new ObjectInputStream(f2);
        PrivateKey prk=(PrivateKey)b2.readObject( );
        // 执行密钥协定
        KeyAgreement ka=KeyAgreement.getInstance("DH");
        ka.init(prk);
        ka.doPhase(pbk,true);
        //生成共享信息
        byte[] sb=ka.generateSecret();
        for(int i=0;i<sb.length;i++){
            System.out.print(sb[i]+",");
        }
        SecretKeySpec k=new SecretKeySpec(sb,"DESede");
    }
}
```

程序最后将共享信息打印了出来，以便直观地对比 A 和 B 得到的信息是否相同。

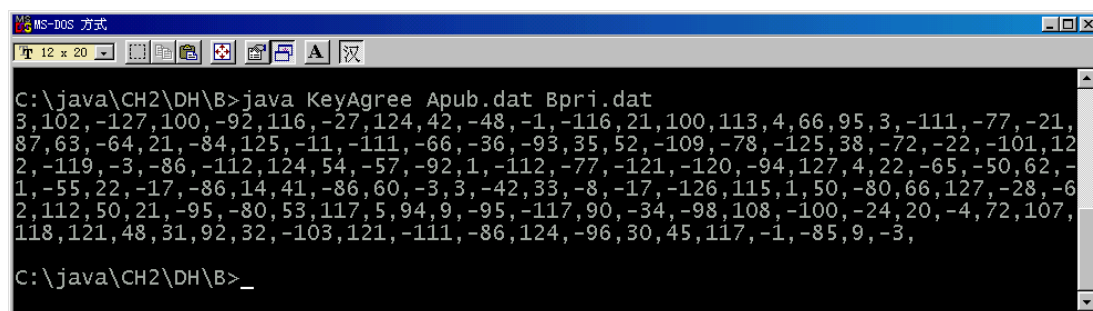
★运行程序

将程序 KeyAgree 编译后分别拷贝在 A 和 B 两个目录，首先在 A 目录输入“java KeyAgree Bpub.dat Apri.dat”运行程序，它使用文件 Bpub.dat 中对方的公钥和文件 Apri.dat 中自己的私钥创建了一段共享的字节数组。程序运行结果如下：



```
MS-DOS 方式
C:\java\CH2\DH\B>java KeyAgree Apub.dat Bpri.dat
3,102,-127,100,-92,116,-27,124,42,-48,-1,-116,21,100,113,4,66,95,3,-111,-77,-21,
87,63,-64,21,-84,125,-11,-111,-66,-36,-93,35,52,-109,-78,-125,38,-72,-22,-101,12
2,-119,-3,-86,-112,124,54,-57,-92,1,-112,-77,-121,-120,-94,127,4,22,-65,-50,62,-
1,-55,22,-17,-86,14,41,-86,60,-3,3,-42,33,-8,-17,-126,115,1,50,-80,66,127,-28,-6
2,112,50,21,-95,-80,53,117,5,94,9,-95,-117,90,-34,-98,108,-100,-24,20,-4,72,107,
118,121,48,31,92,32,-103,121,-111,-86,124,-96,30,45,117,-1,-85,9,-3,
C:\java\CH2\DH\B>_
```

然后在 B 目录输入“java KeyAgree Apub.dat Bpri.dat”运行程序，它使用文件 Apub.dat 中对方的公钥和文件 Bpri.dat 中自己的私钥创建了一段共享的字节数组。程序运行结果如下：



```
MS-DOS 方式
C:\java\CH2\DH\B>java KeyAgree Apub.dat Bpri.dat
3,102,-127,100,-92,116,-27,124,42,-48,-1,-116,21,100,113,4,66,95,3,-111,-77,-21,
87,63,-64,21,-84,125,-11,-111,-66,-36,-93,35,52,-109,-78,-125,38,-72,-22,-101,12
2,-119,-3,-86,-112,124,54,-57,-92,1,-112,-77,-121,-120,-94,127,4,22,-65,-50,62,-
1,-55,22,-17,-86,14,41,-86,60,-3,3,-42,33,-8,-17,-126,115,1,50,-80,66,127,-28,-6
2,112,50,21,-95,-80,53,117,5,94,9,-95,-117,90,-34,-98,108,-100,-24,20,-4,72,107,
118,121,48,31,92,32,-103,121,-111,-86,124,-96,30,45,117,-1,-85,9,-3,
C:\java\CH2\DH\B>_
```

可以看到 A 和 B 运行后得到的字节数组内容完全相同，因此使用它创建的密钥也将相同。由于 DH 算法内在的数学规律，A 和 B 在运行时只使用了对方可以公开的信息（公钥），而各自独立地得到了相同的密钥，完成了密钥分发工作。

本章介绍了对称加密和非对称加密的基本用法，并演示了使用密钥协定进行密钥分发。读者在掌握了其原理后，可以举一反三，互相组合，满足不同应用。出于实例的简洁，本章的例子都使用文件来交换信息，读者也可以使用其它方式，如在密钥协定中可以使用 Socket 等多种方式在程序间传递信息。

第3章 Java 源代码和类、变量 及方法的保护

本章重点:

第2章的Java加密和解密除了直接保障了Java所处理的数据的私密性之外,作为一种基本的工具,Java加密/解密及其相关的技术还提供了更多的保护功能。后续章节中的各种层次上的保护功能都或多或少地与加密/解密技术相关。本章介绍Java程序自身的保护。

Java程序自身的保护涉及到两个方面,一是Java源代码的保护,主要对付各种反编译软件;二是Java中的类、变量、方法的保护。

本章主要内容:

- 使用混淆器加大反编译的难度
- 使用URLClassLoader从网络加载字节码文件
- 使用加密机制加密字节码文件,并定义ClassLoader子类加载加密的字节码文件
- 演示子类、错误的访问控制修饰符、未打开校验器、有问题的常量定义可能带来的安全问题

3.1 Java 反编译及混淆器的使用

Java程序的执行文件是文件名以“.class”为后缀的字节码文件,字节码文件虽然看起来杂乱无章,但其实使用一些反编译工具可以很方便地看到其源代码。在本书第一章提供了一些反编译工具,这些工具使用都很方便,通过命令行形式的命令或直接通过菜单打开“.class”为后缀的文件(有的甚至只需双击字节码文件)就可以看到源代码。

如果辛辛苦苦编制的程序就这么容易地被别人看到源代码,在很多情况下总是令人沮丧的,尤其对于商业软件更是这样。使用混淆器是一种常用的应对措施,它将字节码文件中的符号信息打乱,这样反编译软件得到的源代码将难以解读。本节通过一个实例介绍其基本用法。

★ 实例说明

本节使用2.2.2小节的程序Skey_kb.classa为例,先进行反编译,再使用混淆器对其进行混淆,并用反编译工具对混淆前后的效果作了比较。

★ 编程思路

Java虚拟机比计算机的微处理器要简单,而且文档齐全,因此反编译也比较容易。混淆器常采用的一些措施是:将类、变量、方法、包的名字改为无意义的字符串;使用非法字符串替换符号;添加一些东西使反编译软件崩溃;添加一些无关的指令或永远执行不到的指令等。这样对字节码文件作了修改后,程序从虚拟机的角度来看和以前仍然一样,但从破解者的角度来看则程序变得更难理解了。

由于现在各种反编译和混淆器的软件很多,因而我们不需要自己编程,直接使用这些软

件即可。

★ 代码与分析

本实例所使用的代码和 2.2.2 小节的代码一样。在此基础上进行反编译和混淆的演示。

```
import java.io.*;
import java.security.*;
public class Skey_kb{
    public static void main(String args[]) throws Exception{
        FileInputStream f=new FileInputStream("key1.dat");
        ObjectInputStream b=new ObjectInputStream(f);
        Key k=(Key)b.readObject( );
        byte[ ] kb=k.getEncoded( );
        FileOutputStream f2=new FileOutputStream("keykb1.dat");
        f2.write(kb);
        // 打印密钥编码中的内容
        for(int i=0;i<kb.length;i++){
            System.out.print(kb[i]+",");
        }
    }
}
```

★运行程序

(5) 按照普通方式运行程序

将 2.2.2 小节的程序放在一个单独的目录，如 c:\ch3\obf 目录。将该程序所用到的密钥文件（即 2.2.1 小节生成的 skey1.dat 文件）也放在该目录，输入“javac Skey_kb.java”编译程序，输入“java Skey_kb”运行程序。

则和 2.2.2 小节一样，程序输出：

```
11, -105, -119, 50, 4, -105, 16, 38, -14, -111, 21, -95, 70, -15, 76, -74, 67, -88, 59, -7
1, 55, -125, 104, 42,
```

读者运行时一般得到的是其他结果，这是由于密钥文件不同而造成的。

(6) 使用反编译软件获取源代码

本书第一章中给出了几种常用的反编译软件，这里不妨以命令行形式的 JAD 软件为例，这里使用的是 Jad v1.5.7d。

创建目录 c:\ch3\obf\jad，将字节码文件 Skey_kb.class 拷贝到该目录，尝试对其进行反编译。在 c:\ch3\obf\jad 目录下输入“jad Skey_kb”反编译程序，程序提示：

```
C:\CH3\obf\jad>jad Skey_kb
Parsing Skey_kb... Generating Skey_kb.jad
```

这样，得到反编译后的文件：Skey_kb.jad。该文件内容如下：

```
// Decompiled by Jad v1.5.7d. Copyright 2000 Pavel Kouznetsov.
// Jad home page:
// http://www.geocities.com/SiliconValley/Bridge/8617/jad.html
// Decompiler options: packimports(3)
// Source File Name:   Skey_kb.java

import java.io.*;
```

```

import java.security.Key;
public class Skey_kb
{
    public Skey_kb()
    {
    }
    public static void main(String args[])
        throws Exception
    {
        FileInputStream fileinputstream =
            new FileInputStream("key1.dat");
        ObjectInputStream objectinputstream =
            new ObjectInputStream(fileinputstream);
        Key key = (Key)objectinputstream.readObject();
        byte abyte0[] = key.getEncoded();
        FileOutputStream fileoutputstream =
            new FileOutputStream("keykb1.dat");
        fileoutputstream.write(abyte0);
        for(int i = 0; i < abyte0.length; i++)
            System.out.print(abyte0[i] + ",");
    }
}

```

从中可以看出,使用 JAD 反编译器我们只使用字节码文件而成功地反编译出了原文件。尽管变量名以及构造器等略有变化,但基本逻辑和源程序相同。

(7) 检验反编译软件得到的源代码

将反编译得到的 Skey_kb.jad 更名为 Skey_kb.java,重新编译,并将密钥文件 skey1.dat 拷贝到 c:\ch3\obf\jad 目录,输入“java Skey_kb”运行程序。同样输出:

```

11, -105, -119, 50, 4, -105, 16, 38, -14, -111, 21, -95, 70, -15, 76, -74, 67, -88, 59, -7
1, 55, -125, 104, 42,

```

说明反编译后的程序和原程序运行效果完全相同。

(8) 使用混淆器

下面开始对 c:\ch3\obf 目录的字节码文件 Skey_kb.class 进行混淆处理,以应对反编译软件。

在进行混淆之前,先将字节码文件用 J2SDK 自带的 jar 工具打包。即在 c:\ch3\obf 目录执行:

```
jar cvf My.jar Skey_kb.class
```

将字节码文件打包在 My.jar 文件中。

混淆器的软件有很多,这里不妨以第一章中安装的 marvinobfuscator1-2b 为例。在存放 marvinobfuscator1-2b 的目录中用文本编辑器打开 obfuscate.bat 文件,将其中的“set JAVA_LIB=c:\java\jdk1.3\jre\lib\rt.jar”一行根据机器中安装的 JDK 版本进行修改,如对于 J2SDK1.4,修改为:

```

set JAVA_LIB=C:\j2sdk1.4.0\jre\lib\rt.jar;C:\j2sdk1.4.0\jre\lib\jce.jar

```

打开 marvinobfuscator1-2b 安装目录的 dummyproject 子目录下的 config.txt 文

件,将开头几行的"somelibrary.jar"改为需要进行混淆操作的打包文件名称“My.jar”,
“mainClasses=(("myapp.Main"))”改为包含main()方法的Skey_kb.class文件,即
改为“mainClasses=(("Skey_kb.class"))”。然后将该config.txt文件保存在My.jar
所在的c:\ch3\obf目录。

接下去就可以开始进行混淆操作了。在marvinobfuscator1.2b的安装目录执行

```
obfuscate c:\ch3\obf mynew.jar
```

则自动对c:\ch3\obf目录下config.txt中设置的My.jar中的字节码文件进行混
淆操作,混淆操作的结果放在当前目录的mynew.jar文件中。屏幕输出如下:

The Marvin Obfuscator 1.2b, (c) 2000-2001 by Dr. Java (www.drjava.de)

Pass 1

Pass 2

1 entries written to jar file, total size=976, processing time: 720 ms

Saved mynew.jar (976 bytes)

(9) 检验混淆器的效果

上一步得到了混淆后的文件mynew.jar,将其放在某个目录如c:\ch3\obf\obf中,执行jar xvf
mynew.jar,则得到打包在其中的混淆后的字节码文件Skey_kb.class,将密钥文件key1.dat
也拷贝到c:\ch3\obf\obf目录,输入“java Skey_kb”运行程序,屏幕输出:

11,-105,-119,50,4,-105,16,38,-14,-111,21,-95,70,-15,76,-74,67,-88,59,-71,55,-125,104,42,
和第1步运行结果完全相同。说明混淆操作后字节码文件和以前运行起来完全一样。

在c:\ch3\obf\obf目录输入“jad Skey_kb”进行反编译操作,得到反编译后的源代码
Skey_kb.jad,其内容如下:

```
// Decompiled by Jad v1.5.7d. Copyright 2000 Pavel Kouznetsov.
// Jad home page:
// http://www.geocities.com/SiliconValley/Bridge/8617/jad.html
// Decompiler options: packimports(3)
// Source File Name:

import java.io.*;
import java.security.Key;

public class Skey_kb
{
    public Skey_kb()
    {
    }

    public static void main(String args[])
    {
        FileInputStream fileinputstream = new FileInputStream(ma);
        ObjectInputStream objectinputstream =
            new ObjectInputStream(fileinputstream);
        Key key = (Key)objectinputstream.readObject();
        byte abyte0[] = key.getEncoded();
        FileOutputStream fileoutputstream = new FileOutputStream(na);
        fileoutputstream.write(abyte0);
    }
}
```



```

        for(int i = 0; i < abyte0.length; i++)
            System.out.print(abyte0[i] + oa);
    }
    private static String a(String s)
    {
        int i = s.length();
        char ac[] = new char[i];
        for(int j = 0; j < i; j++)
            ac[j] = (char)(s.charAt(j) ^ 0xffff5aca);
        return new String(ac);
    }

    private static String ma =
        "\u5AA1\u5AAF\u5AB3\u5AFB\u5AE4\u5AAE\u5AAB\u5ABE";
    private static String na =
        "\u5AA1\u5AAF\u5AB3\u5AA1\u5AA8\u5AFB"
        "\u5AE4\u5AAE\u5AAB\u5ABE";
    private static String oa = "\u5AE6";
    public static
    {
        ma = a(ma);
        na = a(na);
        oa = a(oa);
    }
}

```

尽管通过反编译工具仍旧得到了源代码,但是和混淆前相比,这个源代码很难阅读,反编译后的代码中多了一些方法,文件名等信息也被打乱了。如果将文件名后缀改为.java,编译改文件会发现无法直接编译通过。

3.2 从网络资源加载字节码文件

3.1 节尽管对文件进行了混淆处理,但魔高一尺、道高一丈,反编译器和混淆器之间的争斗永无止境,没有绝对的成功者,因此从其他角度进一步保护 Java 源代码很有必要。本节介绍如何将字节码文件放在网上动态下载,使得字节码文件较难找到,从而增加反编译的难度。

★ 实例说明

本实例给出的程序 MyURL.java 在运行时,部分字节码文件可以不放在本地,而是在需要时从网络上下载。

★ 编程思路:

Java 程序在执行时,使用类加载器将所需要的字节码文件加载到 Java 虚拟机,这一加载过程由 ClassLoader 类完成。由于 Java 中已经实现了 ClassLoader 类,因此这一过程是自动完成的。如果有一些特殊需要,则可以重写 ClassLoader 类,对于从网络通过 URL 加载字节码文件,则可以在程序中使用 Java 提供的 URLClassLoader 类来完成。

URLClassLoader 类的基本用法是通过一个 URL 类型的数组告诉 URLClassLoader 类型的对象从什么地方加载类, 然后使用 loadClass() 方法从给定的 URL 中加载字节码文件, 获取其中的方法并执行之。其基本步骤如下:

(1) 创建 URL 类型数组

```
URL myurl[] = {  
    new URL("file:///C:/CH3/ClassLoader/web/"),  
    new URL("http://www.shu.edu.cn/~xyx/test/jvenc/")  
};
```

分析: 在 URL 类型的数组中给出存放字节码文件的 url 地址, 如 “http://...”、“ftp://...”等, 如果字节码文件就存放在硬盘的某个目录下, 也可以使用 “file:///...” 形式的 URL。地址的最后以 “/” 结尾, 表明 url 给出的是一个目录, 字节码文件存放在该目录下。也可以将字节码文件使用 J2SDK 提供的 jar 工具打包成一个 .jar 文件, 直接通过 url 指定具体的 .jar 文件。以后程序寻找字节码文件将根据该数组中的 url 顺序依次寻找字节码文件, 直至找到为止。

本实例中将字节码文件放在了 C:\CH3\ClassLoader\web 目录下, 或者放在 http://www.shu.edu.cn/~xyx/test/jvenc 的 Web 服务器上。

(2) 创建 URLClassLoader 对象

```
URLClassLoader x = new URLClassLoader (myurl);
```

分析: 将第 1 步中创建的 URL 类型数组传递给 URLClassLoader 的构造器。

(3) 使用 URLClassLoader 对象加载字节码文件

```
Class c = x.loadClass ("TestURL");
```

分析: 执行 URLClassLoader 类的 loadClass() 方法, 在其参数中指定字节码文件的名称。如本实例中加载的字节码文件为 “TestURL.class”。该方法返回的是 Class 类型的对象, 不妨使用变量名 c 来代表该对象, 通过它可以使用加载的字节码文件创建对象、执行其中的方法。

这里加载的字节码文件 TestURL.class 是放在网络中的, TestURL 的源代码见本节的 “代码与分析”。在 TestURL.java 的代码中我们定义了一个静态方法 main(String args[])、一个带参数的方法 tt(String s, int i) 和一个不带参数的方法 tt(), 以便演示各种情况下如何执行这些方法。以下每个步骤演示了一种情况, 可以根据需要选用。

(4) 执行静态方法

```
Class getArg1[] = { (new String[1]).getClass() };  
Method m = c.getMethod( "main", getArg1 );  
String[] my1={"arg1 passed","arg2 passed"};  
Object myarg1[] = {my1};  
m.invoke( null, myarg1 );
```

分析: 该段代码获取上一步所加载的字节码文件 TestURL.class 中的静态方法 main(String args[]), 并执行之。

首先要通过方法名称和参数找到要执行的方法 main(String args[]) 方法, 第 3 步得到的 Class 对象 c 的 getMethod() 方法可以完成这一操作。方法 getMethod() 的第 1 个参数是方法的名称, 第二个参数是方法的参数类型, 由此可以在对象 c 中进行匹配,

找到所需要的方法。

其中参数类型匹配使用的是 Class 类型的数组。由于类 TestURL 中 main() 方法的参数只有一个, 因此在本步骤的代码中, Class 类型的数组 getArg1 中只定义了一个元素。由于类 TestURL 的 main() 方法的参数是字符串类型的数组, 在 getArg1 的元素中使用 new String[1] 创建一个字符串数组, 并执行其 getClass() 方法获得 Class 类型的对象, 以便放在 Class 类型的数组中进行匹配。

通过 getMethod() 方法获取到 Method 类型的对象 m 后, 就可以执行其 invoke() 方法来执行变量 m 所代表的方法了。invoke() 的第一个参数是对象的名字, 由于 main() 是静态方法, 因此可以不创建对象而执行, 这时第一个参数也可以使用 null。第二个参数将传入变量 m 所代表的方法的参数。这个参数是一个 Object 类型的数组, 由于本实例中 m 所代表的方法 main() 只有一个参数, 因此这里传入的 Object 类型的数组只有一个元素: 字符串数组 my1。这样, 字符串数组 my1 将传入 main() 方法的参数 args。

(5) 执行非静态、无参数的方法

```
Object ob = c.newInstance();
Class arg2[] = { };
Method m2 = c.getMethod("tt", arg2);
m2.invoke( ob, null);
```

分析: 该段代码获取第 3 步所加载的字节码文件 TestURL.class 中的不带参数的方法 tt(), 并执行之。

由于方法不是静态的, 因此必须先创建对象再执行。可以使用 Class 对象 c 的 newInstance() 方法来创建 TestURL 类型的对象。然后和第 4 步一样通过 getMethod() 方法找到 TestURL 的不带参数的方法 tt(), 返回值 m2 就代表了这个方法 tt()。由于方法 tt() 没有参数, 所以定义了一个空的 Class 类型的数组 arg2 和其匹配。

方法 tt() 找到后, 同样执行 invoke() 方法来运行 m2 所代表的方法 tt()。其中第一个参数就是使用 newInstance() 创建的 TestURL 类型的对象, 由于方法 tt() 没有参数, 因此 invoke() 方法的第二个参数为 null。

(6) 执行非静态、有参数的方法

```
Class arg3[] = {(new String()).getClass(), int.class};
Method m3 = c.getMethod("tt", arg3);
Object myarg2[] = {"Arg1", new Integer(100)};
m3.invoke( ob, myarg2);
```

分析: 该段代码获取第 3 步所加载的字节码文件 TestURL.class 中的带参数的方法 tt(String s, int i), 并执行之。

由于方法不是静态的, 所以第 5 步创建 TestURL 类型的对象的语句: Object ob = c.newInstance() 仍然需要。使用 getMethod() 方法寻找方法 tt(String s, int i) 时, 由于 tt() 有两个参数, 因此需要创建有两个元素的 Class 类型的数组 arg3。方法 tt() 的第一个元素是字符串类型, 因此 arg3 的第一个元素为 new String().getClass(), 方法 tt() 的第二个元素是基本数据类型 int, 无法用 getClass() 方法获取 Class 类型对象, 可以使用 int.class 代表 int 类型的类, 同样, 其他的基本数据类型也可以类似地使用。

方法 `tt()` 找到后, 同样执行代表该方法的 `Method` 对象 `m3` 的 `invoke()` 方法来运行方法 `tt()`。执行 `invoke()` 时同样第一个参数传入 `TestURL` 类型的对象, 第二个参数传入 `Object` 类型的数组 `myarg2` 作为方法 `tt()` 的参数。由于方法 `tt()` 的第一个参数是字符串, 第二个参数是基本类型 `int`, 因此数组 `myarg2` 中第一个元素是一个字符串, 第二个元素是 `Integer` 类型的对象。

★代码与分析:

使用 `URLClassLoader` 加载网上的字节码文件的完整代码如下:

```
import java.net.*;
import java.lang.reflect.*;
public class MyURL{
    static public void main( String args[] ) throws Exception {
        URL myurl[] = {
            new URL("file:///C:/CH3/ClassLoader/web/"),
            new URL ("http://www.shu.edu.cn/~xyx/test/jvenc/")
        };
        URLClassLoader x = new URLClassLoader (myurl);
        Class c = x.loadClass ("TestURL");
        // 执行main(String args[])
        Class getArg1[] = { (new String[1]).getClass() };
        Method m = c.getMethod( "main", getArg1 );
        String[] my1={"arg1 passed","arg2 passed"};
        Object myarg1[] = {my1};
        m.invoke( null, myarg1 );

        // 执行tt( )
        Object ob = c.newInstance();
        Class arg2[] = { };
        Method m2 = c.getMethod("tt", arg2 );
        m2.invoke( ob, null);

        // 执行main(String s, int i)
        Class arg3[] = {(new String()).getClass(), int.class};
        Method m3 = c.getMethod("tt", arg3 );
        Object myarg2[] = {"Arg1",new Integer(100)};
        m3.invoke( ob,myarg2);
    }
}
```

该段代码所加载的 `TestURL` 类的代码如下:

```
import java.io.*;
public class TestURL{
    static public void main( String args[] ) throws Exception {
        System.out.println("From Main");
        System.out.println(args[0]);
        System.out.println(args[1]);
    }
}
```

```

    }

    public void tt( ) throws Exception{
        System.out.println("From tt without args");
        byte b[]="How are you!".getBytes("UTF8");
        FileOutputStream f=new FileOutputStream("ssss.txt");
        f.write(b);
    }

    public void tt(String s, int i){
        System.out.println("Fron tt with args");
        System.out.println(s);
        System.out.println(i);
    }
}

```

在 TestURL 的 main()语句中，使用打印语句将命令行参数的 args[0]和 args[1]打印出来，以便演示数组参数传递的效果。在不带参数的方法 tt()中，演示了从 URL 加载的字节码文件仍旧可以进行文件操作。在带参数的方法 tt()中，将传入的参数打印出来，演示了字符串和基本类型的参数传递。

★运行程序

在 C:\CH3\ClassLoader\web 目录中保存 TestURL 编译后的字节码文件，然后在任一目录下运行 java MyURL，屏幕输出如下：

```

From Main
arg1 passed
arg2 passed
From tt without args
Fron tt with args
Arg1
100

```

其中，第 1 至 3 行为执行 main()方法的输出，第 4 行为执行不带参数的方法 tt()的输出，随即在当前目录下将创建一个文本文件“ssss.txt”，其内容为“How are you!”。最后三行为执行带参数的方法 tt()的输出。

如果将 TestURL.class 放在地址为 <http://www.shu.edu.cn/~xyx/test/jvenc/> 的 Web 服务器上，运行结果也相同。程序运行时会自动根据程序 MyURL 中的 myurl 数组的设置，依次在各个 URL 寻找 TestURL.class。这个功能除了提高反编译难度外，还有助于提高程序的可靠性，即使用户不小心将 C:\CH3\ClassLoader\web 目录中的 TestURL.class 删除了，程序也会在运行时自动从网上下载。

3.3 以任意方式加载字节码文件

3.2 节使用 URLClassLoader 类将字节码文件放在网上，在执行时动态加载，提高了反编译的

难度。但 `URLClassLoader` 类要求字节码文件存放的位置必须能够以 “http://...”、“ftp://...”、“file:///...” 等 URL 形式来访问，因而有一定的局限。有时希望从数据库、自己定义的协议或其他非常规的地方加载字节码文件，这时可以定义 `ClassLoader` 类的子类来实现。本节给出了其编程方法。

★ 实例说明

本节给出一个程序，其运行的字节码文件可以从任意地方获得。

★ 编程思路：

通过定义 `ClassLoader` 类的子类，以后使用这个子类就可以改变类加载器在加载类时的操作。一般情况下，在子类中只需要重写 `findClass(String name)` 方法即可，其中的参数 `name` 是字节码文件的名字。编程的基本步骤如下：

- (4) 根据参数 `name`，将字节码文件从任意位置读入字节数组

```
FileInputStream f=  
    new FileInputStream("C:\\ch3\\classLoader\\web\\"+name+".class");  
int num=f.available();  
byte[] classData=new byte[num];  
f.read(classData);
```

分析：本示例程序从 `c:\ch3\classloader\web` 目录利用文件输入流读取字节码文件的内容，文件名为 `name` 加上 `.class` 后缀，读取的内容保存在字节数组 `classData` 中。读者也可以改从数据库或由程序动态生成等方式将所需的字节码文件读入字节数组。

- (5) 根据字节数组生成 `Class` 类型的对象

```
Class x=defineClass(name, classData, 0, classData.length);
```

分析：父类 `ClassLoader` 中的 `defineClass()` 方法可以利用字节数组中的内容生成 `Class` 类型的对象。

- (6) 返回 `Class` 类型的对象

```
return x;
```

分析：将从字节码文件生成的 `Class` 类型对象返回。

这样，以后程序使用这里定义好的子类时，可以执行和 3.1 小节类似执行 `loadClass()` 方法来调用某个类。这时将自动执行 `ClassLoader` 类预定的一系列操作，当自动执行到 `defineClass()` 方法时，将使用这里定义的步骤取得字节码文件并生成 `Class` 类型的对象。相关的调用程序见本节的“代码与分析”。

★代码与分析：

本节定义的 `ClassLoader` 的子类完整代码如下：

```
import java.io.*;  
import java.lang.reflect.*;  
public class MyClassLoader extends ClassLoader  
{  
    public Class findClass( String name){  
        byte[] classData=null;  
        try{  
            FileInputStream f=  
                new FileInputStream("C:\\CH3\\ClassLoader\\web\\"+name+".class");  
            int num=f.available();  
            byte[] classData=new byte[num];  
            f.read(classData);  
            Class x=defineClass(name, classData, 0, classData.length);  
            return x;  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```

        name+".class");
        int num=f.available();
        classData=new byte[num];
        f.read(classData);
    } catch(IOException e){ }
    Class x=defineClass(name, classData, 0, classData.length);
    return x;
}
}

```

使用这个 `ClassLoader` 子类从其指定地方加载类的程序示例如下：

```

import java.lang.reflect.*;
class MyURL2{
    static public void main( String args[] ) throws Exception {
        MyClassLoader x=new MyClassLoader();
        Class c = x.loadClass(args[0]);
        Class getArg1[] = { (new String[1]).getClass() };
        Method m = c.getMethod( "main", getArg1 );
        String[] my1={"arg1 passed","arg2 passed"};
        Object myarg1[] = {my1};
        m.invoke( null, myarg1 );
    }
}

```

该程序的内容和 3.2 节的 2,3,4 步骤相同，只是将 `URLClassLoader` 改为了我们自己定义的类 `MyClassLoader`。在类 `MyClassLoader` 中可以用任意方式获取字节码文件。本程序执行 `loadClass()` 方法时，传入的字节码文件名使用 `args[0]`，这样程序可以通过命令行参数指定不同的字节码文件名称。

除了执行 `main()` 方法外，也可以和 3.2 节中的程序一样执行其他的方法。

★运行程序

将本实例的两个程序在同一个目录编译后，输入 “`java MyURL2 TestURL`”，则将从 `c:\ch3\classloader\web` 目录加载 `TestURL.class` 字节码文件，屏幕输出：

```

From Main
arg1 passed
arg2 passed

```

3.4 加载加密的字节码文件

3.2 和 3.3 节中将字节码文件保存在其他地方，这样为反编译增加了一些难度。进一步，可以使用第 2 章的加密技术对字节码文件进行加密，在加载时再动态进行解密操作。

★ 实例说明

本节给出一个程序，其运行的字节码文件事先经过加密，并可以从任意地方获得。

★ 编程思路：

将 3.3 节的代码中读取字节码文件部分稍作修改, 增加解密部分即可实现此功能。这里不妨使用 2.5.2 小节的程序进行加密操作, 这样程序中解密时只要对应编程即可。因此, 只要将 3.3 节的程序 `MyClassLoader.java` 的 `try...catch` 语句之间的内容按照如下步骤修改即可。

- (1) 获取加密所用的密钥, 并初始化密码器

```
FileInputStream fkey=new FileInputStream("key1.dat");
ObjectInputStream ob=new ObjectInputStream(fkey);
Key k=(Key)ob.readObject();
Cipher cp=Cipher.getInstance("DESede");
cp.init(Cipher.DECRYPT_MODE, k);
```

分析: 该步骤和 2.5.2 小节的第 1, 2 步相同。由于这里只需要解密操作, 因此初始化密码器时使用 `Cipher.DECRYPT_MODE` 作为参数。

- (2) 根据参数 `name`, 获取已加密的字节码文件

```
FileInputStream in=
    new FileInputStream("c:\\ch3\\enc\\web\\"+name+".class");
```

分析: 该步骤将 2.5.2 小节的第 3 步和 3.4 节的第 1 步结合了起来。以后从该流中读取 `c:\\ch3\\enc\\web\\` 目录中的加密字节码。此外, 这里也可以从 Web 服务器等位置读取加密的字节码文件。

- (3) 获取解密的输出流以及 `CipherOutputStream` 对象

```
ByteArrayOutputStream out=new ByteArrayOutputStream();
CipherOutputStream cout=new CipherOutputStream(out, cp);
```

分析: 该步骤和 2.5.2 小节的第 4 步类似。由于这里解密后的内容需要放在字节数组中以便后面生成 `Class` 类型的对象, 因此使用 `ByteArrayOutputStream` 类作为解密后内容的输出流。然后将其作为参数传递给 `CipherOutputStream` 输出流。这样, 以后向 `CipherOutputStream` 输出流写入的字节将自动通过密码器 `cp` 进行解密, 并写入 `ByteArrayOutputStream` 字节数组输出流。

- (4) 写输出流

```
while( (b=in.read())!=-1){
    cout.write(b);
}
```

分析: 该步骤和 2.5.2 小节的第 5 步相同。用 `in.read()` 从加密字节码文件读取内容, 用 `cout.write()` 自动解密输出到字节数组输出流中。

- (5) 获取解密后的字节数组

```
classData=out.toByteArray();
```

分析: 使用字节数组输出流的 `toByteArray()` 方法获得其中的明文字节数组, 其内容即为解密后的字节码文件。

★代码与分析:

本节定义的包含解密过程的 `ClassLoader` 子类完整代码如下:

```
import java.io.*;
import java.lang.reflect.*;
import java.security.*;
import javax.crypto.*;
```



```

public class MyClassLoader2 extends ClassLoader
{
    public Class findClass( String name){
        byte[ ] classData=null;
        try{
            FileInputStream fkey=new FileInputStream("key1.dat");
            ObjectInputStream ob=new ObjectInputStream(fkey);
            Key k=(Key)ob.readObject( );
            Cipher cp=Cipher.getInstance("DESede");
            cp.init(Cipher.DECRYPT_MODE, k);
            FileInputStream in=
                new FileInputStream("c:\\ch3\\enc\\web\\"+name+".class");
            ByteArrayOutputStream out=new ByteArrayOutputStream( );
            CipherOutputStream cout=new CipherOutputStream(out, cp);
            int b=0;
            while( (b=in.read())!=-1){
                cout.write(b);
            }
            cout.close();
            classData=out.toByteArray( );
        } catch(Exception e){ }

        Class x=defineClass(name, classData, 0, classData.length);
        return x;
    }
}

```

使用这个 `ClassLoader` 子类从其指定地方加载类的程序和 3.3 节类似，只是类加载器使用的是本节的 `MyClassLoader2` 类：

```

class MyURL3{
    static public void main( String args[] ) throws Exception {
        MyClassLoader2 x=new MyClassLoader2();
        Class c = x.loadClass(args[0]);
        Class getArg1[] = { (new String[1]).getClass() };
        Method m = c.getMethod( "main", getArg1 );
        String[] my1={"arg1 passed","arg2 passed"};
        Object myarg1[] = {my1};
        m.invoke( null, myarg1 );
    }
}

```

★运行程序

将 3.2 节和 3.3 节使用的 `c:\ch3\classloader\web` 目录下的 `TestURL.class` 字节码文件拷贝到一个新的目录如 `C:\CH3\Enc\Web` 目录。将 2.5.2 小节所用的密钥文件 `key1.dat` 和加密程序 `streamOut.class` 也拷贝到该目录。输入“`java StreamOut enc TestURL.class`”

my.class”运行程序,则将 TestURL.class 加密为 my.class。删除 TestURL.class,将 my.class 改名为 TestURL.class。则此时 TestURL.class 为加密后的字节码文件。该文件由于已经加密过,因而无法直接运行,必须通过本节的程序在加载时进行解密。

在任一目录下编译本例中的程序,并在当前目录拷贝加密时所用的密钥文件 key1.dat。在当前目录没有 TestURL.class 的情况下,输入“java MyURL3 TestURL”运行程序,则屏幕和 3.3 节一样输出:

From Main

arg1 passed

arg2 passed

表明 TestURL 已经被解密并加载进入 Java 虚拟机而运行。

3.5 加载当前目录下的加密字节码文件

3.2、3.3 节和 3.4 节中如果所需要的字节码文件在当前目录下已经存在,则程序执行时会用优先采用默认的方法从当前目录下加载字节码文件,这样 findClass()就执行不到了。在 3.2 和 3.3 小节,这不会带来什么问题,但对于 3.4 节使用加过密的字节码文件,则 findClass()方法未执行将导致使用默认方法加载的字节码文件,由于默认的加载过程没有解密过程,因此程序无法执行。

而使用加密的字节码文件时,很多场合又需要加密的字节码文件放在当前目录,本节给出其编程方法。

★ 实例说明

本节修改了 3.4 节的程序,使得加密的字节码文件放在当前目录下也可正常运行。

★ 编程思路:

由于问题是由于 ClassLoader 类的 findClass()方法执行方式引起的,因此需要重写 findClass()方法,把默认的从当前目录下载字节码文件的操作改到解密操作之后即可。这样,只要在 3.4 小节的程序中增加一个方法 loadClass()即可。该方法的编程步骤为:

- (1) 检查类是否已经加载过

```
c = findLoadedClass(name);
```

```
if (c != null) return(c);
```

分析: Java 中加载一个类时会同时加载一系列其他相关的类,每次加载都会执行一次 loadClass()方法,因此需要使用检查一下是否已经加载过,若加载过,则直接返回。这一步是必须的。ClassLoader 类已经定义了 findLoadedClass()方法完成这一功能。

- (2) 若未加载过,则读取指定地方的字节码文件,并解密

```
c=findClass(name);
```

分析: 这里可以从各种地方如当前目录、指定的目录、URL、数据库等读取字节码文件,并可对其作各种处理,如进行解密等。这里不妨使用 3.4 节程序中已经重写过的 findClass()方法来完成这个功能。

- (3) 若从指定地方读取失败,则使用默认的方式读取字节码文件

```
if (c==null){
```

```
c = findSystemClass (name);
```

分析:ClassLoader 类默认的加载字节码的方法定义在 findSystemClass()方法中,

它在本地文件系统中寻找其参数中指定的字节码文件。这主要用于加载基本 Java 库。

(4) 必要时加载所用道的相关的类

```
if (resolve && c != null)
    resolveClass( c );
```

分析: 若所加载的类还用到了其他类, 方法 `loadClass()` 的第二个参数 `resolve` 会自动设置为 `true`, 这时该步骤将执行 `resolveClass()`, 继续加载相关的类。

★代码与分析:

本实例的代码是在 3.4 节的代码上增加了一个覆盖的方法 `loadClass()`, 其完整代码如下, 其中修改的部分以粗体标明。

此外在 `findClass()` 方法中, 本节的程序将 3.4 节指定的目录 “`c:\ch3\enc\web\`”+`name`+“.class” 改为当前目录 “`name`+“.class””, 这样将加密的字节码文件放在当前目录, 更符合使用习惯。

```
import java.io.*;
import java.lang.reflect.*;
import java.security.*;
import javax.crypto.*;

public class MyClassLoader3 extends ClassLoader
{
    //覆盖 loadClass() 方法
    public Class loadClass( String name, boolean resolve )
        throws ClassNotFoundException {
        byte[] classData=null;
        Class c = null;
        try {
            c = findLoadedClass(name);
            if (c != null) return(c);
            try {
                c=findClass(name);
            } catch( Exception fnfe ) {
            }
            if (c==null) {
                c = findSystemClass (name);
            }
            if (resolve && c != null)
                resolveClass( c );
            return(c);
        } catch( Exception e ) {
            throw new ClassNotFoundException( e.toString());
        }
    }
    //以下代码和 3.5 节相同。
```

```

public Class findClass( String name) {
    byte[ ] classData=null;
    try{
        FileInputStream fkey=new FileInputStream("key1.dat");
        ObjectInputStream ob=new ObjectInputStream(fkey);
        Key k=(Key)ob.readObject( );
        Cipher cp=Cipher.getInstance("DESede");
        cp.init(Cipher.DECRYPT_MODE, k);
        FileInputStream in=
            new FileInputStream(name+".class");
        ByteArrayOutputStream out=new ByteArrayOutputStream( );
        CipherOutputStream cout=new CipherOutputStream(out, cp);
        int b=0;
        while( (b=in.read())!=-1) {
            cout.write(b);
        }
        cout.close();
        classData=out.toByteArray( );
    } catch(Exception e) { }

    Class x=defineClass(name, classData, 0, classData.length);
    return x;
}
}

```

使用这个 `ClassLoader3` 的程序和 3.4 节类似，只是类加载器使用的是本节的 `MyClassLoader3` 类：

```

class MyURL4 {
    static public void main( String args[] ) throws Exception {
        MyClassLoader3 x=new MyClassLoader3();
        Class c = x.loadClass(args[0]);
        Class getArg1[] = { (new String[1]).getClass() };
        Method m = c.getMethod( "main", getArg1 );
        String[] my1={"arg1 passed", "arg2 passed"};
        Object myarg1[] = {my1};
        m.invoke( null, myarg1 );
    }
}

```

★运行程序

将本节的程序编译后保存在一个空的目录如 `c:\ch3\enc`，同时将 3.4 节使用的加密后的字节码文件 `TestURL.class` 和密钥文件 `key1.dat` 也放在该目录，输入“`java MyURL4 TestURL`”运行程序，和 3.4 节一样输出：

```
From Main
arg1 passed
arg2 passed
```

表明 TestURL 已经被解密并加载进入 Java 虚拟机而运行。

3.6 Java 类、成员变量和方法的保护

3.6.1 类的保护

★ 实例说明

当一个类定义好后，黑客有可能通过定义该类的子类、并以自己的子类替换原有的类来干扰系统。子类看起来和原来的类一样，但所做的事情可能完全不同，这很可能给系统带来破坏或造成泄密。如 Java 中的 String 类，对于 Java 编译器和解释器都很重要，因此它就被设计成不可有子类。

本实例定义了一个类 MyDate1，在类 Test1 构成的系统中使用了 MyDate1 类。MyDate1 类在该系统中非常重要，本实例演示了攻击者如何通过定义 MyDate1 类的子类来攻击该系统 Test1，以及编写类 MyDate1 如何防范。

★ 编程思路

Java 中定义类时，只要在类的声明中加上关键字 final，则其他程序员将不可以定义该类的子类。

本实例演示系统（Test1.java）的功能是加载命令行参数指定的类，并自动执行其无参数的 getMyDate() 方法。所加载的类的 getMyDate() 方法必须返回一个 MyDate 类型的对象。系统执行 MyDate 对象的 getYear() 方法获取当前的年份，系统的根据当前年份作各种重要操作。作为演示，这里不妨判断若年份大于 2050，则打印 “Do critical things here! ”，对应于一些机密或对系统来说很重要的操作。否则打印 “Do normal things here!”。

因此类 MyDate 在系统中是个重要的类，必须保证其方法 getYear() 方法获取的确实是当前时间。getYear() 方法中可能包括大量代码获得当前可靠的时间，如可以通过加密方式连接某个时间服务器。。这里为了简化程序，假定获得的时间是 2003，简单地将其赋值给变量 year 作演示用，并通过方法名称返回。

```
public int getYear() {
    // ....
    //以加密方式连接时间服务器，获取时间
    int year=2003;
    return(year);
}
```

系统 Test1 中的主要步骤如下：

- (1) 获取系统所使用的类加载器

```
ClassLoader cl=ClassLoader.getSystemClassLoader();
```

分析：执行 ClassLoader 类的静态方法 getSystemClassLoader() 得到启动程序时所使

用的类加载器。

- (2) 使用类加载器加载命令行参数指定的类

```
Class c= cl.loadClass(args[0]);
```

分析: 执行上一步所获得的类加载器的 loadClass() 方法, 加载命令行参数指定的类。

- (3) 创建对象

```
Object ob = c.newInstance();
```

分析: 使用 Class 对象 c 的 newInstance() 方法来创建上一步加载的类的对象

- (4) 获取无参数的 getMyDate() 方法

```
Class arg2[] = { };
```

```
Method m2 = c.getMethod("getMyDate", arg2 );
```

分析: 通过 Class 对象 c 的 getMethod() 方法找到上一步对象中不带参数的方法 tt()。

- (5) 执行 getMyDate() 方法

```
Object o=m2.invoke( ob, null);
```

分析: 执行上一步获取的 Method 对象的 invoke() 方法来运行其所代表的方法 getMyDate()。其中第一个参数就是第 3 步使用 newInstance() 创建的对象, 由于方法 tt() 没有参数, 因此 invoke() 方法的第二个参数为 null。执行结果返回 Object 类型的对象。

- (6) 处理返回值

```
MyDate d=(MyDate) o;
```

```
d.getYear()
```

分析: 将上一步执行结果强制转换为 MyDate 类型, 根据系统需要对其作各种处理, 如获取当前年份等。针对 d.getYear() 的执行结果可能作各种操作, 其中可能包括一些关键或机密操作。

由 MyDate 和 Test1 两个类组成的系统交给用户后, 用户就可以按照系统的要求编写包含 getMyDate() 方法的类, 作为参数提供给系统处理。本实例给出两个例子, 一个是 Normal.java, 它正常使用系统, 在程序中创建 MyDate 对象, 并在 getMyDate() 方法中返回该对象。另一个是 Attacker.java, 它通过编写 MyDate 的子类 MyDate2 对系统进行攻击。

★代码与分析:

本实例被攻击的系统 Test1.java 中包含两个类: Test1 和 MyDate。

```
import java.util.*;
import java.io.*;
import java.lang.reflect.*;

class Test1{
    public static void main(String args[] ) throws Exception{
        //获取ClassLoader并加载命令行参数指定的类
        ClassLoader cl=ClassLoader.getSystemClassLoader();
        Class c= cl.loadClass(args[0]);
        //创建命令行参数指定的类的对象
        Object ob = c.newInstance();
```

```

        //创建空的参数对象
        Class arg2[] = { };
        //获取命令行参数指定的类的getMyDate( )方法
        Method m2 = c.getMethod("getMyDate", arg2 );
        //执行getMyDate( )方法
        Object o=m2.invoke( ob, null);
        //处理方法返回结果
        MyDate d=(MyDate) o;
        if(d.getYear( ) >2050){
            System.out.println("Do critical things here!");
        }
        else{
            System.out.println("Do normal things here!");
        }
    }
}

class MyDate{
    public int year, month, day;
    public int getYear(){
        // ...
        //以加密方式连接时间服务器，获取时间
        int year=2003;
        return(year);
    }
}

```

本实例正常使用系统Test1的程序Normal. java内容如下：

```

class Normal {
    MyDate d=new MyDate();
    public MyDate getMyDate(){
        return(d);
    }
}

```

本实例攻击系统Test1的程序Attacker. java内容如下：

```

class Attacker {
    MyDate2 d=new MyDate2();
    public MyDate getMyDate(){
        MyDate t=d;
        return(t);
    }
}

class MyDate2 extends MyDate{
    public int getYear(){

```

```

        return(3000);
    }
}

```

★运行程序

本实例工作在C:\java\Ch3\class\final目录，输入

```
javac Test1.java
```

编译程序。然后用户开始正常系统该系统：编写Normal.java程序，输入

```
javac Normal.java
```

编译用户程序，最后输入

```
java Test1 Normal
```

运行程序。程序输出：

“Do normal things here! ”

由于用户使用系统提供的MyDate类，当前年份没有超过2050,因此这里进行了当前年份许可的操作。下面演示攻击者如何在年份尚未超过2050就欺骗系统执行一些机密操作。

攻击者编写本小节“代码与分析”部分给出的Attacker.java程序，它使用的是攻击者自己编写的MyDate类的子类MyDate2，该类中和MyDate类一样定义了int getYear()方法（重写父类方法），重写后的方法它没有像父类MyDate一样通过加密方式连接时间服务器来获取当前标准的时间，而是直接返回攻击者欺骗系统所需要的时间值3000。这样，将欺骗系统从事一些关键或机密操作，达到扰乱系统或获取机密信息的目的。

输入

```
javac Attacker.java
```

编译程序，输入

```
java Test1 Attacker
```

运行程序，程序将显示：

Do critical things here!

可见系统已经不是按照系统编写者预想的那样运行。如果系统编写者在编写类 MyDate 时考虑到它对于系统的重要性，将其定义为 final 类型：

```

final class MyDate{
    public int year, month, day;
    public int getYear(){
        // ...
        //以加密方式连接时间服务器，获取时间
        int year=2003;
        return(year);
    }
}

```

则攻击者就无法通过定义 MyDate 类的子类来攻击系统了。如果 MyDate 类中有很多方法，只有少部分方法若被子类重写则可能引发系统的安全问题，则可只将这些方法定义为 final，而不必将整个类定义为 final。

当然，攻击者在可以接触到系统代码时可能会修改系统代码，将 final 定义去掉，这可以通过代码签名等技术来解决。

3.6.2 成员变量和方法的保护

★ 实例说明

在类的成员变量和方法前面可以加上各种修饰符，如 `public`, `protected`, `private` 或缺省类型，不同的修饰符表明不同的可被访问的范围。

本实例定义了一个类 `MyDate3`，其中错误定义了成员变量的访问控制范围，本实例给出其可能造成的后果。

★ 编程思路

不同的权限如 表 3-1所示。

	同一个类中可访问	同一个包中可访问	子类中可访问	全局可访问
<code>public</code>	是	是	是	是
<code>protected</code>	是	是	是	
默认	是	是		
<code>Private</code>	是			

表 3-1 访问控制范围

本实例定义的代码 `MyDate3.java` 中，定义了设置日期的方法 `setDay(int d)`和获取日期的方法 `getDay()`。在 `setDay()`方法中使用了大量判断条件判断日期是否合法（如对于大月是否在 1 至 31 之间、小月是否在 1 至 30 之间，2 月份根据是否是闰年判断其是否在 1 至 29 或 28 之间），但其成员变量定义成了 `public` 类型，因此其他程序员可以任意修改其成员变量。从而可能引起各种意想不到的情况。

★代码与分析:

本实例编写的有问题的代码 `MyDate3.java` 如下:

```
class MyDate3{
    public int year, month, day;
    public void setDay(int d){
        //各种合法性检测
        //.....
        day=d;
    }
    public int getDay( ){
        return(day);
    }
}
```

为了简化程序，这里跳过合法性检测中大量繁琐的代码。该程序给其他程序员使用后，其他程序员可能这样使用程序:

```
public class TestVar{
    public static void main(String args[] ){
        MyDate3 d=new MyDate3( );
    }
}
```

```

        d.setDay(10);
        d.day=100;
        // 执行各种操作
        //.....
        int day=d.getDay();
        //使用day做各种事情
        //...
        System.out.println(day);
    }
}

```

★运行程序

本实例工作在C:\java\Ch3\class\var目录，输入

```
javac MyDate3.java
```

```
javac TestVar.java
```

编译程序。输入

```
java TestVar
```

运行程序，将显示当前日期是 100 号，而实际上任何一天绝不可能是 100 号，因此若程序使用该日期做一些重要操作，有可能会引起意想不到的结果。

如果在定义类 MyDate3 时考虑到其成员变量 year, month, day 如果被任意访问可能带来意想不到的结果，希望其他程序员使用该类时只能通过类 MyDate3 的方法进行，而定义 MyDate3 的方法时实现作了各种检测和处理。这时可将成员变量定义为 private。如本实例中若变量 day 的定义前面若加上 private，则 TestVar.java 中就无法通过 d.day=100 来访问成员变量了，否则编译时将出错。

同样，有些方法只是在一个类的内部使用，也可以在方法的定义前面加上 private 修饰符。

3.6.3 使用校验器

★ 实例说明

3.6.2 小节通过加上合适的访问控制修饰符保证成员变量和方法的安全，但它是在编译时确定的。本实例演示了如何在运行时强制使用访问控制修饰符的规则。

★ 编程思路

本实例先编写了程序 TestVeri.java，其中定义了一个类 MyDate4，该类使用 3.6.2 小节错误的方法将其成员变量定义为 public 类型了，在类 TestVeri 演示了可直接访问其成员变量。

本实例然后将其更正，但只重新编译了修改后的代码。通过本实例将发现，修改后的 MyDate4.java 程序虽然在 day 等成员变量定义中加上了 private 修饰符，但在类 TestVeri 中仍旧可以直接访问其成员变量。本实例通过在运行时加上 -verify 选项打开校验器解决了这一问题。

★ 代码与分析:

本实例错误的代码 TestVeri.java 中定义类 TestVeri 和类 MyDate4 如下:

```

public class TestVeri{
    public static void main(String args[] ){
        MyDate4 d=new MyDate4( );
        d.setDay(10);
        d.day=100;
        // 执行各种操作
        //.....
        int day=d.getDay();
        //使用day做各种事情
        //...
        System.out.println(day);
    }
}

class MyDate4{
    public int year, month, day;
    public void setDay(int d){
        //各种合法性检测
        //.....
        day=d;
    }
    public int getDay( ){
        return(day);
    }
}

```

本实例加上访问控制修饰符后的 MyDate4.java 程序如下：

```

class MyDate4{
    private int year, month, day;
    public void setDay(int d){
        //各种合法性检测
        //.....
        day=d;
    }
    public int getDay( ){
        return(day);
    }
}

```

★运行程序

本实例工作在C:\java\Ch3\class\verir目录，输入
javac TestVeri.java

编译程序。输入

```
java TestVeri
```

运行程序，将显示当前日期是 100 号。

然后编写 MyDate4.java 程序将类 MyDate4 的成员变量定义为 private 类型。输入

```
javac MyDate4.java
```

编译程序，然后再输入

```
java TestVeri
```

运行程序，将发现程序仍旧输出“100”。

这是因为编译时未重新编译 TestVeri，因而编译时未能校验出问题。在运行时，Java 的字节码验证器也会检测访问控制修饰符的规则。但对于 Java 应用程序，字节码验证器默认情况下是关闭的（如果是 Java Applet，则默认是开启的）。可通过 Java 命令选项-verify 打开字节码验证。如输入

```
java -verify TestVeri
```

运行程序，则程序将显示如下出错信息：

```
Exception in thread "main" java.lang.IllegalAccessError: try to access field
MyDate4.day from class TestVeri
    at TestVeri.main(TestVeri.java:5)
```

3.6.4 Reference 类型私有成员变量的保护

★ 实例说明

3.6.2 小节中，如果定义为 private 的成员变量是 Reference 类型（即不是八种基本数据类型），则在编程时若不小心仍可能产生安全问题。本实例给出一种场景。

★ 编程思路

如果一个类的成员变量是私有的，但是是 Reference 类型，而在该类的公共方法中又返回了该变量，则其他人可以通过该公共方法得到私有成员变量，进而直接访问和修改该成员变量，使得 private 修饰符的保护作用丧失。

如本实例的程序 MyDate5 中，如果年、月、日是分别保存在整型数组 date 三个元素中的，程序员希望其他人使用时只能通过方法 setDay() 来修改日期，getDay() 来或取当前日期，因而将该数组定义为 private。此时成员变量 date 是安全的。但若类 MyDate5 中又定义了一个方法 getDate()，它通过方法名称返回成员变量 date，则就不安全了，如：

```
class MyDate5{
    private int [] date=new int[3];    // year, month, day;
    public int[] getDate(){
        return(date);
    }
}
```

此时，其他程序（如本实例中的 TestRef.java）只要通过 getDate() 方法就可以获得成员变量 date，然后就可以直接修改数组里面的内容。

一种解决方法是，如果类的公共方法中需要返回私有成员变量，则使用 clone() 方法或其他方式新创建一个对象。如：

```
int [] x=(int []) date.clone();
```

新的对象 `x` 和私有成员变量 `date` 所指向的对象内容完全相同，但是是两个对象。这样，即使其他程序试图通过返回值修改对象内容，所修改的对象也不是私有成员变量所指向的对象，从而实现了保护。本实例 `MyDate6.java` 和 `TestRef2.java` 演示了其效果。

如果成员变量不是 `Reference` 类型，或虽然是 `Reference` 类型，但无法修改对象中内容（如 `String` 类型，字符串已经生成就无法修改字符串中的内容，除非生成新的字符串），则不存在本实例中的问题。。

★ 代码与分析:

本实例错误的代码 `MyDate5.java` 完整内容如下:

```
class MyDate5{
    private int [ ] date=new int[3];    // year, month, day;
    public void setDay(int d){
        //各种合法性检测
        //.....
        date[2]=d;
    }
    public int getDay( ){
        return(date[2]);
    }
    public int[ ] getDate(){
        return(date);
    }
}
```

本实例攻击该代码的程序 `TestRef.java` 完整内容如下:

```
public class TestRef{
    public static void main(String args[ ] ){
        MyDate5 d=new MyDate5( );
        d.setDay(10);
        int day=d.getDay();
        System.out.println(day);
        //跳过getDay( )方法直接修改私有成员变量
        int x[ ]=d.getDate( );
        x[2]=100;
        //查看被修改的效果
        day=d.getDay();
        System.out.println(day);
    }
}
```

对修改 `MyDate5.java` 作了改进后，新的程序 `MyDate6.java` 内容如下:

```
class MyDate6{
    private int [ ] date=new int[3];    // year, month, day;
```

```

    public void setDay(int d){
        //各种合法性检测
        //.....
        date[2]=d;
    }
    public int getDay( ){
        return(date[2]);
    }
    public int[ ] getDate(){
        int [ ] x=(int [ ]) date.clone();
        return(x);
    }
}

```

程序TestRef2.java使用和TestRef类似的方法试图直接访问类MyDate6的成员变量，其代码如下：

```

public class TestRef2{
    public static void main(String args[] ){
        MyDate6 d=new MyDate6( );
        d.setDay(10);
        int day=d.getDay();
        System.out.println(day);
        //跳过getDay( )方法直接修改私有成员变量
        int x[ ]=d.getDate( );
        x[2]=100;
        day=d.getDay();
        System.out.println(day);
    }
}

```

★运行程序

本实例工作在C:\java\Ch3\class\reference目录，输入

```

javac MyDate5.java
javac TestRef.java

```

编译程序。输入

```
java TestRef
```

运行程序，将输出如下结果：

```

10
100

```

可见程序开始通过类 MyDate5 提供的方法 setDay()将私有成员变量中表示日期的部分修改成为 10，然后跳过 setDay()方法而通过 getDate()方法得到成员变量，并进而直接修改了类 MyDate5 的私有成员变量，使得 setDay()方法中的种种合法性检测不再起作用。这样

会给程序带来各种意想不到的后果。

输入

```
javac MyDate6.java
javac TestRef2.java
```

编译改进以后的程序，输入

```
java TestRef2
```

运行程序，将输出如下结果：

```
10
10
```

可见，即使 TestRef2 中修改了返回对象的内容，也不影响类 MyDate6 的成员变量。

3.6.5 保护常量

★ 实例说明

本实例演示 Java 中的常量可能被修改的情况。

★ 编程思路

Java 中使用 final 修饰符代表常量，但如果一个常量是 Reference 类型，常量只是代表所指向的对象不变，而对象的内容仍可能被修改。

一种解决方法是，如果常量是 Reference 类型，常量所指向的对象内容可变，而程序中不希望对象内容可变，则先定义一个类，将原有的对象作为其私有成员变量，并且不提供修改该成员变量的方法。

★ 代码与分析：

本实例代码 TestConst.java 中定义了两个类，类 TestConst 用于测试两种常量，类 NewConst 创建的对象是按照本实例中的方法定义的不可修改的常量。其代码如下：

```
public class TestConst{
    //有问题的定义
    final int x[ ]={1,2,3};
    //无问题的定义
    final NewConst y=new NewConst();
    public static void main(String args[] ){
        TestConst t=new TestConst( );
        //执行go( )方法试图修改常量x和y中的内容
        t.go( );
        //查看常量x的内容
        int[ ] x1=t.x;
        System.out.println("The code with problem");
        for(int i=0;i<x1.length;i++){
            System.out.println(x1[i]);
        }
        //查看常量y的内容
        NewConst x2=t.y;
        System.out.println("The code without problem");
        for(int i=0;i<x2.getConst().length;i++){
```

```

        System.out.println(x2.getConst()[i]);
    }

}

void go(){
    //试图修改常量x和y中的内容
    x[0]=20;
    y.getConst()[0]=20;
}
}

class NewConst{
    private final int x[]={1,2,3};
    public final int[] getConst(){
        int [] s=(int []) x.clone();
        return(s);
    }
}

```

★运行程序

本实例工作在C:\java\Ch3\class\const目录，输入

```
javac TestConst.java
```

编译程序。输入

```
java TestConst
```

运行程序，将输出如下结果：

```

The code with problem
20
2
3
The code without problem
1
2
3

```

可见类 TestConst 中，常量 x 的内容被修改了，而常量 y 的内容没有被修改。

本章使用混淆器来解决反编译问题，使用类加载器动态加载加密的字节码文件，编程时使用各种修饰符来防止由于子类、直接访问成员变量或方法等带来的安全问题。从而在源代码、类加载、类和成员变量及方法的使用等角度上提高 Java 程序自身的安全性。

各种安全性的基础是用户对程序员和程序的信任，编程者对程序的安全性考虑得再多，如果攻击者冒充某个组织将恶意的程序提供给用户，或向用户提供已经泄密的密钥，或修改用户正在使用的程序，则编程者事先的总总措施将毫无用处。在后面的章节中将陆续介绍如何确定数据的所有者、如何确定某个人或机构的数字化身份、如何使用这些数字化身份来建

立通信双方的信任关系以及根据数字化身份进行授权。

第 4 章 数据完整性和所有者的确认

——消息摘要和签名

本章重点：

当数据（可能是一个文件，一个程序，一段文字等）从 A 传递到 B 时，通过第 2 和 3 章的加密可以保证只有拥有密钥者（B）才可读取这段信息，实现了安全性编程中内容的保密性要求。但安全性编程还要求具有不可篡改性（B 接收到数据后需要确认数据在传输过程中是否被别人修改过，发生纠纷时 A 需要检查 B 是否修改过原始数据）、身份的确定性（B 要能够确认数据确实是 A 发来的）以及不可否认性（发生纠纷时，B 能够证明数据确实是由 A 发来的）。

通过消息摘要、消息验证码、数字签名和数字证书等技术可以实现这些功能，Java 支持这些技术，本章将介绍 Java 中如何使用消息摘要、消息验证码和数字签名，同时给出消息摘要在口令验证中的应用。在下一章则将介绍数字证书。

本章主要内容：

- 通过消息摘要验证数据是否被篡改
- 通过消息验证码证数据是否被篡改
- 通过数字签名验证数据的身份
- 使用消息摘要保存和验证口令
- 字典式攻击和加盐技术

4.1 使用消息摘要验证数据未被篡改

消息摘要是对原始数据按照一定算法进行计算得到的计算结果，它主要用于检验原始数据是否被修改过。例如对于字符串，我们可以简单地将各个字符的 ASCII 码的值累加起来作为其消息摘要，这样，字符“Hello World!”的消息摘要是： $72+101+108+108+111+32+87+111+114+108+100+33=1085$ 。这样，如果接收者对收到的字符串作同样计算发现计算结果不是 1085，则可以确信收到的字符串在传输过程中被篡改了。

从这个简单的例子可以看出消息摘要和加密不同，从加密的结果可以得到原始数据，而从消息摘要中不可能得到原始数据。消息摘要长度比原始数据短得多（所以称为原始数据的“摘要”），实际使用中，原始数据不管多长，消息摘要一般是固定的 16 或 20 个字节长。

实际使用中消息摘要有许多成熟的算法，这些算法不仅处理效率高，而且不同的原始数据计算出相同的消息摘要的概率极其低，因此消息摘要可以看作原始数据的指纹，指纹不同则原始数据不同。本节介绍 Java 中如何使用这些成熟的消息摘要算法。

4.1.1 计算消息摘要

★ 实例说明

本实例使用最简洁的编程步骤计算出指定字符串的消息摘要。

★ 编程思路：

java.security 包中的 MessageDigest 类提供了计算消息摘要的方法，首先生成对象，执行其 update() 方法可以将原始数据传递给该对象，然后执行其 digest() 方法即可得到消息摘要。具体步骤如下：

(1) 生成 MessageDigest 对象

```
MessageDigest m=MessageDigest.getInstance("MD5");
```

分析：和 2.2.1 小节的 KeyGenerator 类一样。MessageDigest 类也是一个工厂类，其构造器是受保护的，不允许直接使用 new MessageDigest() 来创建对象，而必须通过其静态方法 getInstance() 生成 MessageDigest 对象。其中传入的参数指定计算消息摘要所使用的算法，常用的有 "MD5", "SHA" 等。若对 MD5 算法的细节感兴趣可参考 <http://www.ietf.org/rfc/rfc1321.txt>。

(2) 传入需要计算的字符串

```
m.update(x.getBytes("UTF8"));
```

分析：x 为需要计算的字符串，update 传入的参数是字节类型或字节类型数组，对于字符串，需要先使用 getBytes() 方法生成字符串数组。

(3) 计算消息摘要

```
byte s[] = m.digest();
```

分析：执行 MessageDigest 对象的 digest() 方法完成计算，计算的结果通过字节类型的数组返回。

(4) 处理计算结果

必要的话可以使用如下代码将计算结果 s 转换为字符串。

```
String result="";
for (int i=0; i<s.length; i++){
    result+=Integer.toHexString((0x000000ff & s[i]) | 0xffffffff).substring(6);
}
```

★ 代码与分析：

完整程序如下：

```
import java.security.*;

public class DigestPass{
    public static void main(String args[]) throws Exception{
        String x=args[0];
        MessageDigest m=MessageDigest.getInstance("MD5");
        m.update(x.getBytes("UTF8"));
        byte s[] = m.digest();
        String result="";
        for (int i=0; i<s.length; i++){
            result+=Integer.toHexString((0x000000ff & s[i]) |
                                         0xffffffff).substring(6);
        }
    }
}
```

```

    }
    System.out.println(result);
}
}

```

★运行程序

输入 `java DigestCalc abc` 来运行程序，其中命令行参数 `abc` 是原始数据，屏幕输出计算后的消息摘要：900150983cd24fb0d6963f7d28e17f72。

根据 <http://www.ietf.org/rfc/rfc1321.txt>，可测试以下字符串及输出结果：

输入字符串	程序输出
""	d41d8cd98f00b204e9800998ecf8427e
"a"	0cc175b9c0f1b6a831c399e269772661
"abc"	900150983cd24fb0d6963f7d28e17f72
"message digest"	f96b697d7cb7938d525a2f31aaf161d0
"abcdefghijklmnopqrstuvwxyz"	c3fcd3d76192e4007dfb496cca67e13b
"ABCDEFGHJKLMNOPQRSTUVWXYZ XYZabcdefghijklmnopqrstuvwxyz0123456789"	d174ab98d277d9f5a5611c2c9f419d9f
"1234567890123456789012345678901234567890 1234567890123456789012345678901234567890"	57edf4a22be3c955ac49da2e2107b67a

如果 A 欲向 B 发送信息：“I have got your \$800”，A 可输入“`java DigestCalc "I have got your $800"`”来运行程序，将得到消息摘要：“d9c17e68da7ee9b24e8929f150f56fe9”，A 将消息摘要和原始数据都发送给 B。如果 B 收到数据后原始数据已经被篡改成：“I have got your \$400”，B 可以类似地用自己的程序计算其消息摘要（消息摘要算法是公开的），如输入“`java DigestCalc "I have got your $400"`”来运行程序，将得到消息摘要：“62069826e27c7e0b60a044e412f66b2b”，发现 A 发来的消息摘要不同，从而知道数据已经被篡改。

4.1.2 基于输入流的消息摘要

4.1.1 小节给出了计算字符串的消息摘要的编程方法，实际使用中经常要对流（如文件流）计算消息摘要，这时虽然可以从流中读出所有字节然后计算，但是使用 `DigestInputStream` 类更加方便。

★ 实例说明

本实例使用 `DigestInputStream` 对象计算文件输入流的消息摘要，它可以在一边读入数据一边将数据传递给 `MessageDigest` 对象以计算消息摘要。

★ 编程思路

Java 中 `DigestInputStream` 类可以在读取输入流的同时将所读的字节传递给 `MessageDigest` 对象计算消息摘要，编程步骤如下：

（1）生成 `MessageDigest` 对象

```
MessageDigest m=MessageDigest.getInstance("MD5");
```

分析：和 4.1.1 小节第 1 步一样，其中传入的参数指定计算消息摘要所使用的算法，

常用的有"MD5", "SHA"等。

(2) 生成需要计算的输入流

```
FileInputStream fin=new FileInputStream(args[0]);
```

分析: 本实例针对文件输入流计算消息摘要, 因此这里先创建文件输入流。文件名称不妨从命令行参数传入。

(3) 生成 DigestInputStream 对象

```
DigestInputStream din=new DigestInputStream(fin,m);
```

分析: DigestInputStream 类的构造器传入两个参数, 第一个是所要计算的输入流, 即第 2 步得到的 fin, 第二个是第 1 步生成的 MessageDigest 对象。

(4) 从 DigestInputStream 流中读取数据

```
while(din.read() != -1);
```

分析: 该步骤和读取一般的输入流的方法一样, 实际读取的是上一步创建 DigestInputStream 对象时从构造器传入的输入流。一般的输入流在该 while 循环中应该对读取的字节进行处理, 对于 DigestInputStream, 读取过程中所读的字节除了通过 read() 方法返回外, 将传递给上一步穿入的 MessageDigest 对象, 因此这里的 while 循环体可以为空。

(5) 计算消息摘要

```
byte s[] =m.digest();
```

分析: 执行 MessageDigest 对象的 digest() 方法完成计算, 计算的结果通过字节类型的数组返回。

★代码与分析:

完整程序如下:

```
import java.security.*;
import java.io.*;
public class DigestInput {
    public static void main(String args[]) throws Exception {
        MessageDigest m=MessageDigest.getInstance("MD5");
        FileInputStream fin=new FileInputStream(args[0]);
        DigestInputStream din=new DigestInputStream(fin,m);
        while(din.read() != -1);
        byte s[] =m.digest();
        String result="";
        for (int i=0; i<s.length; i++) {
            result+=Integer.toHexString((0x000000ff & s[i]) |
            0xffffffff00).substring(6);
        }
        System.out.println(result);
    }
}
```

程序最后和 4.1.1 小节的程序一样将消息摘要转换为字符串打印出来。

★运行程序

可以计算该字节码文件自身的消息摘要: 输入 “java DigestInput

DigestInput.class”来运行程序，输出如下：

```
1ae0bb2f0c1bcb983060800730154b43
```

也可以计算运行 Java 程序的 java.exe 的消息摘要，如果 J2SDK 是安装在 c:\j2sdk1.4.0 目录下，可输入“java DigestInput c:\j2sdk1.4.0\bin\java.exe”来运行程序，输出如下：

```
6dcabd700656987230089b3c262b0249
```

可见不管原始数据多长，按照 MD5 算法计算出的消息摘要长度是相同的。此外，如果你的 java.exe 与我的完全相同的话，则计算出的消息摘要结果也必然是“6dcabd700656987230089b3c262b0249”，否则说明我们用的不是同一个 java.exe。

4.1.3 输入流中指定内容的消息摘要

4.1.2 小节中对给定的文件总是计算所有文件内容的消息摘要，有时只需要文件中指定内容的消息摘要，本小节给出一个实例。

★ 实例说明

本实例使用 DigestInputStream 类计算文件输入流中第一次出现“\$”以后的内容的消息摘要。

★ 编程思路

Java 中 DigestInputStream 类在读取输入流时可以通过方法 on() 随时控制是否将所读的字节传递给 MessageDigest 对象计算消息摘要，这样可以按照所需要的条件关闭或打开消息摘要功能。其编程方法和 4.1.2 小节类似，只要在 while 循环中根据所读内容调用 on() 方法即可。

- (1) 生成 MessageDigest 和 DigestInputStream 对象

```
MessageDigest m=MessageDigest.getInstance("MD5");  
FileInputStream fin=new FileInputStream(args[0]);  
DigestInputStream din=new DigestInputStream(fin,m);
```

分析：该步骤和 4.1.2 小节的 1 至 3 步相同。

- (2) 先关闭消息摘要功能

```
din.on(false);
```

分析：din 为上一步得到的 DigestInputStream 对象，在其 on() 方法中传入 false 作为参数，则以后通过 din 从输入流读取字节时将不会把读到的字节传递给 MessageDigest 对象计算消息摘要。

- (3) 从 DigestInputStream 流中读取数据

```
int b;  
while ( (b = din.read()) != -1){  
}
```

分析：该步骤和读取一般的输入流的方法一样。由于要根据所读到的字节控制是否开启消息摘要功能，因此将 read() 方法返回的内容传递给整型变量 b。

- (4) 若读到的内容为“\$”，则开启消息摘要功能

```
if(b=='$'){  
    din.on(true);  
}
```

分析：该段代码放在上一步 while 语句的循环体中，当读到的内容是“\$”时，则执行 DigestInputStream 对象的 on() 方法，传入 true 作为参数。这样上一步以后再通过 read() 方法读出的字节将自动传递给 MessageDigest 对象计算消息摘要。

(5) 计算消息摘要

```
byte s[] = m.digest();
```

分析：执行 MessageDigest 对象的 digest() 方法完成计算，计算的结果通过字节类型的数组返回。

★代码与分析：

完整程序如下：

```
import java.security.*;
import java.io.*;
public class DigestInputLine{
    public static void main(String args[]) throws Exception{
        MessageDigest m=MessageDigest.getInstance("MD5");
        FileInputStream fin=new FileInputStream(args[0]);
        DigestInputStream din=new DigestInputStream(fin,m);
        din.on(false);
        int b;
        while ( (b = din.read()) != -1){
            if(b=='$'){
                din.on(true);
            }
        }
        byte s[] = m.digest();
        String result="";
        for (int i=0; i<s.length; i++){
            result+=Integer.toHexString((0x000000ff & s[i]) |
                0xffffffff).substring(6);
        }
        System.out.println(result);
    }
}
```

程序最后和 4.1.1 小节的程序一样将消息摘要转换为字符串打印出来。

★运行程序

在当前目录存放三个文本文件，1.txt，2.txt 和 3.txt，内容分别为：

文件 1.txt:

I'll lend u \$200

文件 2.txt:

As for many reasons,

I won't lend u \$200

文件 3.txt:

As for many reasons,

I won't lend u \$100

则输入“java DigestInputStream 1.txt”运行程序，得到的结果为：

91f23d7175d3b3c2ea1ae301528f53c2

输入“java DigestInputStream 2.txt”运行程序，得到的结果同样为：

91f23d7175d3b3c2ea1ae301528f53c2

输入“java DigestInputStream 3.txt”运行程序，得到的结果则为：

b8a4f2f99c387b80cda72f6b43079b8b

可见程序只计算第一次出现“\$”符号以后的内容。

4.1.4 基于输入流的消息摘要

4.1.2 小节给出了基于输入流的消息摘要，本小节介绍基于输出流的消息摘要。

★ 实例说明

本实例从键盘读入数据，然后使用 `DigestOutputStream` 对象将数据写入文件输出流，同时计算其消息摘要。

★ 编程思路

Java 中 `DigestOutputStream` 类可以在向输出流写数据的同时将所写的字节传递给 `MessageDigest` 对象以便计算消息摘要，编程步骤如下：

(1) 生成 `MessageDigest` 对象

```
MessageDigest m=MessageDigest.getInstance("MD5");
```

分析：和 4.1.2 小节第 1 步一样，其中传入的参数指定计算消息摘要所使用的算法，常用的有“MD5”，“SHA”等。

(2) 生成需要的输出流

```
FileOutputStream fout=new FileOutputStream(args[0]);
```

分析：本实例以文件输出流为例，文件名称不妨从命令行参数传入。

(3) 生成 `DigestOutputStream` 对象

```
DigestOutputStream dout=new DigestOutputStream(fout,m);
```

分析：`DigestOutputStream` 类的构造器传入两个参数，第一个是所要处理的输入流，即第 2 步得到的 `fout`，第二个是第 1 步生成的 `MessageDigest` 对象。

(4) 向 `DigestOutputStream` 流中写数据

```
int b;
while ((b = System.in.read()) != -1) {
    dout.write(b);
}
```

分析：该步骤和一般的输出流用法类似，使用 `DigestOutputStream` 的 `write()` 方法写数据，这里一次写一个字节，也可以一次将一个字节类型数组中的内容写入 `DigestOutputStream` 流。在执行 `write()` 方法时，相应的数据实际上写入了上一步所

传入的文件输出流,同时传递给上一步穿入的 MessageDigest 对象。和 4.1.3 小节一样,可以使用 on() 方法传入 true 或 false 的值控制是否将 write() 方法中的数据传递给 MessageDigest 对象。

所写入的数据可以通过各种方式得到,这里使用 System.in.read() 从键盘读入数据。

(5) 关闭 DigestOutputStream 流

```
dout.close();
```

分析: 和一般的输出流一样使用 close() 方法关闭流。

(6) 计算消息摘要

```
byte s[] = m.digest();
```

分析: 执行 MessageDigest 对象的 digest() 方法完成计算,计算的结果通过字节类型的数组返回。

★代码与分析:

完整程序如下:

```
import java.security.*;
import java.io.*;
public class DigestOutput {
    public static void main(String args[]) throws Exception {
        MessageDigest m = MessageDigest.getInstance("MD5");
        FileOutputStream fout = new FileOutputStream(args[0]);
        DigestOutputStream dout = new DigestOutputStream(fout, m);
        int b;
        while ((b = System.in.read()) != -1) {
            dout.write(b);
        }
        dout.close();
        byte s[] = m.digest();
        String result = "";
        for (int i = 0; i < s.length; i++) {
            result += Integer.toHexString((0x000000ff & s[i]) |
                0xffffffff).substring(6);
        }
        System.out.println(result);
    }
}
```

程序最后和 4.1.2 小节的程序一样将消息摘要转换为字符串打印出来。

★运行程序

输入 java DigestOutput tmp.txt 运行程序,然后通过键盘输入几行文本,最后同时按下 Ctrl 和 Z 键结束输入,这时屏幕上将显示所输入文本的消息摘要,如:

```
java DigestOutput tmp.txt
```

```
Hi
How a you!
This is a test!
9dd3424b1b9f4cdb7f8bb028362011e5
```

打开文件 tmp.txt，将看到键盘输入的内容已经写入了文件 tmp.txt。

4.2 使用消息验证码

根据 4.1 节的内容，当 A 将数据传递给 B 时，可以同时将对应的消息摘要传递给 B。B 收到后可以用消息摘要验证数据在传输过程中是否被篡改过。但这样做的前提是 A 传递给 B 的消息摘要正确无误。如果攻击者在修改原始数据的同时重新计算一下消息摘要，然后将 A 传递给 B 的消息摘要替换掉，则 B 通过消息摘要就无法验证出原始数据是否被修改过了。

消息验证码可以解决这一问题。使用消息验证码的前提是 A 和 B 双方有一个共同的密钥，这样 A 可以将消息摘要加密发送给 B，防止消息摘要被篡改。由于使用了共同的密钥，接收者可以在一定程度上验证发送者的身份：一定是和自己拥有共同的密钥的人。所以称为“验证码”。本章介绍其编程方法。

★ 实例说明

本实例使用 2.2 节得到的密钥计算一段字符串的消息验证码，并用于验证字符串是否被篡改过。

★ 编程思路：

javax.crypto 包中的 Mac 类提供了计算消息验证码的方法。首先生成密钥对象和 Mac 类型的对象，Mac 对象的 init() 方法传入密钥，执行其 update() 方法可以将原始数据传递给 Mac 对象，然后执行其 doFinal() 方法即可得到消息验证码。具体步骤如下：

(1) 生成密钥对象

```
byte [] kb={11,-105,-119,50,4,-105,16,38,-14,-111,21,-95,70,
            -15,76,-74,67,-88,59,-71,55,-125,104,42};
```

```
SecretKeySpec k=new SecretKeySpec(kb,"HMACSHA1");
```

分析：这里使用 2.2 节得到的密钥。可以和 2.3.1 小节中的第 1 步那样从文件 key1.dat 中直接读取密钥对象，也可以像 2.3.2 小节中的第 2 步那样从文件 keykb1.dat 中读取密钥的字节，然后生成密钥对象。这里为简便起见直接将文件 keykb1.dat 中的内容赋值给字节数组 kb，然后使用它生成密钥对象。密钥算法名称为“HMACSHA1”。

(2) 生成 Mac 对象

```
Mac m=Mac.getInstance("HmacMD5")
```

分析：Mac 类也是一个工厂类，通过其静态方法 getInstance() 生成 MessageDigest 对象。其中传入的参数指定计算消息验证码所使用的算法，常用的有“HmacMD5”和“HmacSHA1”等

(3) 传入需要计算的字符串

```
m.update(x.getBytes("UTF8"));
```

分析: x 为需要计算的字符串, update 传入的参数是字节类型或字节类型数组, 对于字符串, 需要先使用 getBytes() 方法生成字符串数组。

(4) 计算消息验证码

```
byte s[] = m.doFinal();
```

分析: 执行 Mac 对象的 doFinal() 方法完成计算, 计算的结果通过字节类型的数组返回。

(5) 处理计算结果

必要的话可以使用如下代码将计算结果 s 转换为字符串。

```
String result="";
for (int i=0; i<s.length; i++){
    result+=Integer.toHexString((0x000000ff & s[i]) | 0xffffffff).substring(6);
}
```

★代码与分析:

完整程序如下:

```
import java.io.*;
import java.security.*;
import javax.crypto.*;
import javax.crypto.spec.*;

public class MyMac {
    public static void main(String args[]) throws Exception {
        //获取密钥
        byte [] kb={11,-105,-119,50,4,-105,16,38,-14,-111,
                    21,-95,70,-15,76,-74,67,-88,59,-71,55,-125,104,42};
        SecretKeySpec k=new SecretKeySpec(kb,"HMACSHA1");
        //获取 Mac 对象
        Mac m=Mac.getInstance("HmacMD5");
        m.init(k);
        String x=args[0];
        m.update(x.getBytes("UTF8"));
        byte s[] =m.doFinal();
        String result="";
        for (int i=0; i<s.length; i++) {
            result+=Integer.toHexString((0x000000ff & s[i]) |
                                         0xffffffff).substring(6);
        }
        System.out.println(result);
    }
}
```

★运行程序

输入 java MyMac "How are you!" 来运行程序, 其中命令行参数 “How are you!” 是原始

数据，屏幕输出计算后的消息摘要验证码：e0973b3fb96da6010b5f59f81194e3e9。

如果 A 欲向 B 发送信息：“I have got your \$800”，A 可输入“java MyMac "I have got your \$800"”来运行程序，将得到消息验证码：“10e431a267e586a43affb575e7a7c974”。A 将消息验证码和原始数据都发送给 B。

原始数据和消息验证码在传输过程中都受到了攻击，攻击者将原始数据篡改成：“I have got your \$400”，则 B 只要使用同样的密钥来计算消息验证码（消息验证码的算法是公开的），如输入“java MyMac "I have got your \$400"”来运行程序，将得到消息验证码：“a4a53ffec37332a3542653e0904e2391”，发现和 A 发来的消息验证码不同，从而知道数据已经被篡改。

如果攻击者想把消息验证码也替换掉，尽管攻击者知道消息验证码的算法，但是由于攻击者没有 A 和 B 共有的密钥：“11,-105,-119,50,4,-105,16,38,-14,-111,21,-95,70,-15,76,-74,67,-88, 59,-71,55,-125,104,42”，因而无法计算出正确的值“a4a53ffec37332a3542653e0904e2391”，因此将无法得逞。正是这一点，使得消息验证码更加安全。

4.3 使用数字签名确定数据的来源

使用消息摘要和消息验证码保证了数据未经篡改，但接收者尚无法确定数据是否确实是某个人发来的。尽管消息验证码可以确定数据是某个拥有同样密钥的人发来的，但这要求双方具有共享的密钥，当数据要提供给一组用户、这一组用户都需要确定数据的来源时，消息验证码就不方便了。

数字签名可以解决这一问题。消息验证码的基础是基于公钥和私钥的非对称加密，发送者使用私钥加密消息摘要（签名），接收者使用公钥解密消息摘要以验证签名是否是某个人。这和 2.8 节中的用法正好相反：2.8 节中使用公钥进行加密，只有拥有私钥者才可以解密。由 2.7 和 2.8 节可知，私钥和公钥是成对的，私钥由拥有者秘密保存，对应的公钥则完全公开。因此每个人都可以用公钥尝试能否解密，若可以解密，则这个消息摘要必然是对应的私钥加密的。由于私钥只有加密者才拥有，因此如果接收者用某个公钥解密了某个消息摘要，就可以确定这段消息摘要必然是对应的私钥持有者发来的。

可见私钥就像一个人的笔迹或印章，是每个人独有的，同时又是人人可以检验的。使用私钥加密消息摘要，就像在文件上签名或盖章，确认了数据的身份。这里之所以不直接对原始数据加密而是对消息摘要加密，是因为非对称算法一般计算速度较慢，这样加密很长的原始数据较耗时；而消息摘要既简短，又足以代表原始数据。同时无论原始数据多长，消息摘要的长度都固定。

本节先介绍 Java 中如何用自己的私钥进行数字签名，然后介绍接收者如何用发送者提供的公钥验证数字签名。

4.3.1 使用私钥进行数字签名

★ 实例说明

本实例使用 2.7 节得到的私钥文件 Skey_RSA_priv.dat 对文件 msg.dat 中的信息进行签名。签名将保存在文件 sign.dat 中。

★ 编程思路：

javax. security 包中的 Signature 类提供了进行数字签名的方法。Signature 对象的 initSign() 方法传入私钥, 执行其 update() 方法可以将原始数据传递给 Signature 对象, 然后执行其 sign() 方法即可得到消息验证码。具体步骤如下:

(1) 获取要签名的数据

```
FileInputStream f=new FileInputStream("msg.dat");
int num=f.available();
byte[ ] data=new byte[num];
f.read(data);
```

分析: 不妨将需要签名的数据放在 msg.dat 文件中, 通过文件输入流将其读入字节类型数组 data 中。

(2) 获取私钥

```
FileInputStream f2=new FileInputStream("Skey_RSA_priv.dat");
ObjectInputStream b=new ObjectInputStream(f2);
RSAPrivateKey prk=(RSAPrivateKey)b.readObject( );
```

分析: 这里使用 2.7 节生成的私钥文件 Skey_RSA_priv.dat, 通过文件输入流读入私钥存放在 RSAPrivateKey 类型的变量 prk 中。

(3) 获取 Signature 对象

```
Signature s=Signature.getInstance("MD5WithRSA");
```

分析: Signature 类是工厂类, 需要使用 getInstance() 方法获取对象, 方法的参数指定签名所用的算法, 参数中包含了计算消息摘要所用的算法和加密消息摘要所用的算法。如 “SHA1withRSA”、“MD5withDSA”、“SHA15withDSA” 等。

(4) 用私钥初始化 Signature 对象

```
s.initSign(prk);
```

分析: 使用 Signature 对象的 initSign() 方法初始化 Signature 对象, 其参数为第 2 步得到的私钥。这样, 以后可以用这个私钥加密消息摘要。

(5) 传入要签名的数据

```
s.update(data);
```

分析: 执行 Signature 对象的 update() 方法, 其参数是第 1 步获得的需要签名的数据。

(6) 执行签名

```
byte[ ] signeddata=s.sign( );
```

分析: 使用 Signature 对象的 sign() 方法, 将自动使用前几步的设置进行计算, 计算的结果以字节数组的类型通过方法返回。

★代码与分析:

完整程序如下:

```
import java.io.*;
import java.security.*;
```

```

import java.security.spec.*;
import java.security.interfaces.*;
import javax.crypto.*;
import javax.crypto.spec.*;
import javax.crypto.interfaces.*;

public class Sign{
    public static void main(String args[ ]) throws Exception{
        //获取要签名的数据，放在 data 数组
        FileInputStream f=new FileInputStream("msg.dat");
        int num=f.available();
        byte[ ] data=new byte[num];
        f.read(data);
        //获取私钥
        FileInputStream f2=
            new FileInputStream("Skey_RSA_priv.dat");
        ObjectInputStream b=new ObjectInputStream(f2);
        RSAPrivateKey prk=(RSAPrivateKey)b.readObject( );
        Signature s=Signature.getInstance("MD5WithRSA");
        s.initSign(prk);
        s.update(data);
        System.out.println("");
        byte[ ] signeddata=s.sign( );
        // 打印签名
        for(int i=0; i<data.length; i++) {
            System.out.print(signeddata[i]+", ");
        }
        //保存签名
        FileOutputStream f3=new FileOutputStream("Sign.dat");
        f3.write(signeddata);
    }
}

```

程序最后将签名结果在屏幕上显示，并保存在文件 Sign.dat 中。

★运行程序

在当前目录中存放三个文件：本小节的程序：Sign.class、秘密保存的私钥 Skey_RSA_priv.dat 和要签名的文件：msg.dat。msg.dat 中不妨输入一段内容：

I have got your \$800

输入 java sign 来运行程序，则得到如下结果：

49,-7,-48,-119,-14,68,-65,-27,24,-22,-128,54,-30,39,120,-99,56,92,14,21,85,106,

这个就是文件 msg.dat 签名的结果，它同时保存在文件 Sign.dat 中。

当发送者做完这些后，可以将 msg.dat 和 Sign.dat 同时提供给需要的人。提供时发送者可以放心地将文件通过 Internet 让接收者下载，或 E-mail 给接收者，甚至拷贝在软盘上由其

他人转交接收者。

4.3.2 使用公钥验证数字签名

当接收者接收到发送者发来的文件 msg.dat 及其签名 Sign.dat 后，可以对进行验证。其前提是接收者拥有发送者的公钥。本节介绍 Java 中如何验证数字签名。

★ 实例说明

本实例使用 4.3.1 小节所使用的私钥对应的公钥，即 2.7 节得到的公钥文件 Skey_RSA_pub.dat 对收到的文件 msg.dat 及其签名文件 Sign.dat 进行验证。确保 msg.dat 未被修改过，并且确实是发送者发来的。

★ 编程思路：

javax. security 包中的 Signature 类除了用于签名外，还可用于验证数字签名。Signature 对象的 initVerify () 方法传入公钥，执行其 verify () 方法用其参数中的签名信息验证原始数据。具体步骤如下：

(1) 获取要签名的数据

```
FileInputStream f=new FileInputStream("msg.dat");
int num=f.available();
byte[ ] data=new byte[num];
f.read(data);
```

分析：和 4.3.1 小节一样，从 msg.dat 文件读取需要验证的数据，存放在字节数组 data 中。

(2) 获取签名

```
FileInputStream f2=new FileInputStream("Sign.dat");
int num2=f2.available();
byte[ ] signeddata=new byte[num2];
f2.read(signeddata);
```

分析：从 Sign.dat 文件中读取数字签名，存放在字节数组 signeddata 中。

(3) 读取公钥

```
FileInputStream f3=new FileInputStream("Skey_RSA_pub.dat");
ObjectInputStream b=new ObjectInputStream(f3);
RSAPublicKey pbk=(RSAPublicKey)b.readObject();
```

分析：这里使用 2.7 节生成的公钥文件 Skey_RSA_pub.dat，通过文件输入流读入公钥存放在 RSAPublicKey 类型的变量 pbk 中。

(4) 获取 Signature 对象

```
Signature s=Signature.getInstance("MD5WithRSA");
```

分析：和 4.3.1 小节一样使用静态方法 getInstance () 方法获取 Signature 对象，算法使用和 4.3.1 小节相同的“MD5WithRSA”算法。

(5) 用公钥初始化 Signature 对象

```
s.initVerify(pbk);
```

分析：使用 Signature 对象的 initVerify () 方法初始化 Signature 对象，其参

数为第 3 步得到的公钥。这样，以后可以用这个公钥解密消息摘要。

(6) 传入要签名的数据

```
s.update(data);
```

分析：执行 Signature 对象的 update() 方法，其参数是第 1 步获得的需要签名的数据。

(7) 检验签名

```
s.verify(signeddata);
```

分析：使用 Signature 对象的 verify() 方法，将自动使用前几步的设置进行计算。如果验证通过，则返回 true，否则返回 false。

★代码与分析：

完整程序如下：

```
import java.io.*;
import java.security.*;
import java.security.spec.*;
import java.security.interfaces.*;
import javax.crypto.*;
import javax.crypto.spec.*;
import javax.crypto.interfaces.*;

public class CheckSign{
    public static void main(String args[ ]) throws Exception{
        //获取数据，放在 data 数组
        FileInputStream f=new FileInputStream("msg.dat");
        int num=f.available();
        byte[ ] data=new byte[num];
        f.read(data);
        //读签名
        FileInputStream f2=new FileInputStream("Sign.dat");
        int num2=f2.available();
        byte[ ] signeddata=new byte[num2];
        f2.read(signeddata);
        //读公钥
        FileInputStream f3=new FileInputStream("Skey_RSA-pub.dat");
        ObjectInputStream b=new ObjectInputStream(f3);
        RSAPublicKey pbk=(RSAPublicKey)b.readObject();
        //获取对象
        Signature s=Signature.getInstance("MD5WithRSA");
        //初始化
        s.initVerify(pbk);
        //传入原始数据
        s.update(data);
        boolean ok=false;
        try{
            //用签名验证原始数据
```



```

        ok= s.verify(signeddata);
        System.out.println(ok);
    }
    catch(SignatureException e) { System.out.println(e);}

    System.out.println("Check Over");
}
}

```

★运行程序

在当前目录中事先有两个文件：本小节的程序：CheckSign.class 和公开获得的发送者 A 的公钥 Skey_RSA_pub.dat（2.7 小节得到的公钥文件）。

然后某个人拿来两个文件：存放原始数据的待检验的文件 msg.dat 及其数字签名 Sign.dat，说是 A 发来的文件。接收者开始检验，输入 `java CheckSign` 来运行程序，则得到如下结果：

```

true
Check Over

```

表明签名验证通过，该文件确实是 A 发来的。

假如文件 msg.dat 及其数字签名 Sign.dat 在传递给接收者时被做了手脚，如我们可以对 msg.dat 或 Sign.dat 作任意修改以模仿攻击者做的手脚，这时接收者再输入 `java CheckSign` 来运行程序，则得到如下结果：

```

false
Check Over

```

说明 msg.dat 已经不是发送者发来的原始内容了。

如果攻击者想修改 msg.dat 而让接收者检查不出来，则只有重新计算 Sign.dat。而 Sign.dat 只有知道发送者 A 的私钥才能正确计算出，所以攻击者无计可施。这样，数据的身份可以唯一确定，无法仿冒。

下面我们再看一下数字签名如何实现不可否认性。若接收者拥有了 msg.dat 和相应的签名文件 Sign.dat，以后发送者 A 不承认 msg.dat 中的内容，则接收者可以让仲裁者使用 A 对外公开的公钥文件 Skey_RSA_pub.dat 运行一下“`java CheckSign`”来检验 msg.dat 和 Sign.dat，若显示“true”，则仲裁者可以确信发送者 A 确实承认过 msg.dat 中的内容。因为只有 A 才拥有公钥 Skey_RSA_pub.dat 对应的私钥，其他人都无法由 msg.dat 计算出能通过验证的签名文件 Sign.dat。反过来，如果 A 已经没有文件 msg.dat 的原件了，A 怀疑接收者出示的 msg.dat 是否做过手脚，也可以运行“`java CheckSign`”来检验一下，因为即使接收者对 msg.dat 做了手脚，接收者也无法计算出新的能通过验证的签名文件 Sign.dat。

4.4 使用消息摘要保存口令

程序中经常需要验证用户输入的口令是否正确，如果将正确的用户口令直接存放在程

序、文件或数据库中，则很容易被黑客窃取到，这时可以只保存口令的消息摘要。

4.4.1 使用消息摘要保存口令

★ 实例说明

在 4.1 节中介绍了消息摘要的计算，本节的实例将介绍如何在程序中将口令的消息摘要保存在文件中，以便以后验证用。

本实例中，运行“java SetPass 账号 口令”，将把账号以明文方式保存在文件 passwd.txt 中，而把口令的消息摘要保存在 passwd.txt 中。

★ 编程思路：

作为示例，为程序的简洁不妨通过命令行参数穿入账号和口令，然后按照 4.1.1 小节的方法计算消息摘要，最后将消息摘要

- (1) 读入帐号口令

```
String name=args[0];  
String passwd=args[1];
```

分析：这里为了简便而通过命令行读入帐号和口令，实际程序中可以制作图形界面供用户输入。

- (2) 生成 MessageDigest 对象

```
MessageDigest m=MessageDigest.getInstance("MD5");
```

分析：执行 MessageDigest 类的静态方法 getInstance() 生成 MessageDigest 对象。其中传入的参数指定计算消息摘要所使用的算法。

- (3) 传入需要计算的字节数组

```
m.update(passwd.getBytes("UTF8" ));
```

分析：passwd 为需要计算的口令，使用 getBytes() 方法生成字符串数组，传入 MessageDigest 对象的 update() 方法。

- (4) 计算消息摘要

```
byte s[] =m.digest();
```

分析：执行 MessageDigest 对象的 digest() 方法完成计算，计算的结果通过字节类型的数组返回。

- (5) 在文件中或数据库中保存帐号和口令的消息摘要

```
PrintWriter out= new PrintWriter(new FileOutputStream("passwd.txt"));  
out.println(name);  
out.println(result);  
out.close();
```

分析：这里将帐号和口令消息摘要报存在 passwd.txt 文件中，更好的做法是将其保存在数据库中。

★代码与分析：

本实例完整代码如下：

```
import java.io.*;  
import java.security.*;  
public class SetPass{
```

```

public static void main(String args[ ]) throws Exception{
    String name=args[0];
    String passwd=args[1];

    MessageDigest m=MessageDigest.getInstance("MD5");
    m.update(passwd.getBytes("UTF8"));
    byte s[ ]=m.digest( );
    String result="";
    for (int i=0; i<s.length; i++){
        result+=Integer.toHexString((0x000000ff & s[i]) |
                                    0xffffffff00).substring(6);
    }

    PrintWriter out= new PrintWriter(
        new FileOutputStream("passwd.txt"));
    out.println(name);
    out.println(result);
    out.close();
}
}

```

★运行程序

程序运行在 C:\java\ch4\password 目录，在命令行中输入

```
javac SetPass.java
```

编译程序，输入

```
java SetPass xyx akwi
```

运行程序，则以“xyx”为账号，“akwi”为口令，在文件 passwd.txt 中将保存如下信息：

```
xyx
4e4452f998059e3e574c696a489aac82
```

根据 MD5 消息摘要算法，只知道“4e4452f998059e3e574c696a489aac82”是无法推测出原有口令“akwi”的。因此，黑客即使得到了 passwd.txt 文件，仍旧无法知道原有口令是什么。

4.4.2 使用消息摘要验证口令

★ 实例说明

在 4.4.1 小节中将口令的消息摘要保存在文件中，本节的实例将介绍如何使用所保存的消息摘要验证用户输入的口令是否正确。

本实例中，运行“java CheckPass 账号 口令”，若账号和口令都和保存在 passwd.txt 中的相同，则提示“OK”，否则提示“Wrong password”。

★ 编程思路：

根据用户输入口令计算消息摘要，根据用户输入的账号在 4.3.1 小节保存口令的文件中找到预先保存的正确的口令的消息摘要。比较两个消息摘要是否相等。若不相等则说明输入

的口令不正确。其编程步骤如下：

- (1) 根据用户输入的账号读取文件中对应的口令的消息摘要

```
String name="", passwd="";
BufferedReader in = new BufferedReader(new FileReader("passwd.txt"));
while ((name = in.readLine()) != null) {
    passwd=in.readLine();
    if (name.equals(args[0])){
        break;
    }
}
```

分析：不妨将第一个命令行参数 args[0] 的值作为用户输入的账号。在 4.4.1 小节中，第一行保存的是帐号，第二行保存的是账号对应的口令的消息摘要。如果有多个账号和口令，则可以如该程序的方法依次读取账号/口令摘要，直到所读取的帐号和命令行参数指定的账号相同，则退出读取。

- (2) 计算用户输入的口令的消息摘要

```
MessageDigest m=MessageDigest.getInstance("MD5");
m.update(args[1].getBytes("UTF8" ));
byte s[] =m.digest();
```

分析：不妨将第二个命令行参数 args[1] 的值作为用户输入的口令。使用 4.4.1 小节中相同的步骤进行计算其消息摘要。

- (3) 比较用户输入的口令的消息摘要和文件中保存的口令摘要是否一致

```
if(name.equals(args[0])&&result.equals(passwd)){
    System.out.println("OK");
}
else{
    System.out.println("Wrong password");
}
```

分析：当账号和口令摘要都和文件中保存的一致，则验证通过。

★代码与分析：

本实例完整代码如下：

```
import java.io.*;
import java.security.*;
public class CheckPass{
    public static void main(String args[ ]) throws Exception{
        /* 读取保存的口令摘要 */
        String name="";
        String passwd="";
        BufferedReader in = new BufferedReader(
            new FileReader("passwd.txt"));
        while ((name = in.readLine()) != null) {
            passwd=in.readLine();
```

```

        if (name.equals(args[0])){
            break;
        }
    }

    /* 生成用户输入的口令摘要 */
    MessageDigest m=MessageDigest.getInstance("MD5");
    m.update(args[1].getBytes("UTF8"));
    byte s[] =m.digest( );
    String result="";
    for (int i=0; i<s.length; i++){
        result+=Integer.toHexString((0x000000ff & s[i]) |
            0xffffffff00).substring(6);
    }

    /* 检验口令摘要是否匹配 */
    if(name.equals(args[0])&&result.equals(passwd)){
        System.out.println("OK");
    }
    else{
        System.out.println("Wrong password");
    }
}
}
}

```

★运行程序

程序运行在 C:\java\ch4\password 目录，在命令行中输入

```
javac CheckPass.java
```

编译程序，输入

```
java CheckPass xyx akwi
```

运行程序，程序输出“OK”，表明帐号和口令正确。输入

```
java CheckPass xyx qwert
```

提示“Wrong password”，可见可以正确进行验证。

4.4.3 攻击消息摘要保存的口令

★ 实例说明

4.4.1 小节使用消息摘要保存口令的较为安全的机理是，攻击者即使通过攻击得到了口令文件，例如知道了 xyx 的口令摘要是 4e4452f998059e3e574c696a489aac82，也难以通过该值反推出口令的值。因而无法登录系统。

但是当口令比较短时，攻击者很容易通过字典式攻击由口令的消息摘要反推出原有口令的值。本实例给出一个例子。

★ 编程思路:

使用字典式攻击的思路是:实现计算好各种长度的字符组合所得到的字符串的消息摘要的,将其保存在文件(称为字典)中。这虽然要花很多时间,但只需要做一次。以后如果攻击者得到了某人的口令消息摘要,则不需要进行耗时的计算,直接和字典中的值相匹配,即可知道用户的口令。

其编程步骤可以如下

(1) 生成字符串组合

```
for(int i1='a';i1<'z';i1++){  
    System.out.println("Now Processing"+(char)i1);  
    for(int i2='a';i2<'z';i2++){  
        for(int i3='a';i3<'z';i3++){  
            for(int i4='a';i4<'z';i4++){  
                char[] ch={(char)i1,(char)i2,(char)i3,(char)i4};  
                String passwd=new String(ch);
```

分析:这里不妨使用四重for循环生成四个字符的所有组合,为简化程序,不妨只考虑口令为小写字符a到z的情况。实际使用时,需考虑各种常用字符,并需计算从1个字符到多个字符的各种组合。

(2) 计算消息摘要

```
m.update(passwd.getBytes("UTF8"));  
byte s[] =m.digest( );  
String result="";  
for (int i=0; i<s.length; i++){  
    result+=Integer.toHexString((0x000000ff & s[i]) |  
                                0xffffffff00).substring(6);  
}
```

分析:使用4.4.1小节中相同的步骤计算字符组合的消息摘要。

(3) 保存字典

```
PrintWriter out= new PrintWriter(  
    new FileOutputStream("dict.txt"));  
out.print(passwd+" ");  
out.println(result);
```

分析:将字母组合和消息摘要的对应关系写入字典文件。

字典文件生成后,如果知道了一个消息摘要,只要编写程序查找包含该消息摘要的一行即可。可通过字符串的indexOf()方法查看是否包含给定的消息摘要:

```
if (md.indexOf(args[0])!=-1){  
    System.out.println(md);  
    break;  
}
```

★代码与分析:

本实例完整代码如下:

```
import java.io.*;
```

```

import java.security.*;
public class AttackPass{
    public static void main(String args[ ]) throws Exception{
        MessageDigest m=MessageDigest.getInstance("MD5");
        PrintWriter out= new PrintWriter(
            new FileOutputStream("dict.txt"));

        for(int i1='a';i1<'z';i1++){
            System.out.println("Now Processing"+(char)i1);
            for(int i2='a';i2<'z';i2++)
            for(int i3='a';i3<'z';i3++)
            for(int i4='a';i4<'z';i4++){
                char[ ] ch={(char)i1,(char)i2,(char)i3,(char)i4};
                String passwd=new String(ch);
                m.update(passwd.getBytes("UTF8"));
                byte s[ ]=m.digest( );
                String result="";
                for (int i=0; i<s.length; i++){
                    result+=Integer.toHexString((0x000000ff & s[i]) |
                        0xffffffff00).substring(6);
                }
                out.print(passwd+"  ");
                out.println(result);
            }
        }
        out.close();
    }
}

```

根据已知的消息摘要值查找字典的代码如下：

```

import java.io.*;
import java.security.*;
public class DoAttack{
    public static void main(String args[ ]) throws Exception{
        String md;
        BufferedReader in = new BufferedReader(
            new FileReader("dict.txt"));
        while ((md = in.readLine( )) != null) {
            if (md.indexOf(args[0])!=-1){
                System.out.println(md);
                break;
            }
        }
        in.close();
    }
}

```

```
    }  
}
```

★运行程序

程序运行在 C:\java\ch4\password 目录，在命令行中输入

```
javac AttackPass.java  
javac DoAttack.java
```

编译程序，输入

```
java AttackPass
```

运行程序，则在不长的时间内就完成了所有四个字符的组合，生成的字典保存在 dict.txt 文件中。如果要生成 5 个字符、6 个字符、…的组合，则所需时间将成指数级增长，但只要口令长度不长，机器速度足够快，哪怕需要耗时十几年，由于生成字典只需做一次，一旦足够长度的字符组合的字典生成好了，以后就可以一劳永逸地迅速破解所有使用 4.4.1 小节机制的系统。本实例生成的字典 dict.txt 只针对 4 个字符长度的小写字母，因而速度较快。其部分内容如下所示：

```
aafe    519704dcefc42669c7afbf64a81c647  
aaff    b82bf3c70e89fd848b9e3f2785ebfecc  
aafg    ec02a3166c4d9e4dbd7a925e5a363cb4  
aafh    9e1dc4b41cadd812256d7983eed97ac6  
aafi    214eedbe6eec0d2aa91e487e82a4a939  
aafj    0afff9a7e8a30e96c524407dfa6fe5f4  
aafk    ec8f1121bd879cc498d1c2fdc991a1e6
```

如果攻击者得到了 4.4.1 小节所生成的口令文件 pass.txt，知道了 xyx 的口令消息摘要为“4e4452f998059e3e574c696a489aac82”，则可以运行如下程序来通过字典获取用户 xyx 的口令值：

```
java DoAttack 4e4452f998059e3e574c696a489aac82
```

程序输出

```
akwi    4e4452f998059e3e574c696a489aac82
```

这个查找过程瞬间就可以完成，可见 4.4.1 小节用户使用的口令已经被破解。

4.4.4 使用加盐技术防范字典式攻击

★ 实例说明

4.4.1 小节的口令被轻松攻击的主要原因在于口令过短。如果口令很长，则计算所有组合的消息摘要可能要成百上千年，这将大大加大生成字典的难度。

不过口令很长也给用户带来不便，因此用户使用的口令长度总是有限的。加盐技术即可在有限的口令长度基础上增加攻击者生成字典的难度。

★ 编程思路：

加盐技术的基本原理是，在用户输入的口令前面加上一串随机数（称为盐），然后将随机数和口令组合在一起计算消息摘要。最后将随机数（盐）和消息摘要一起保存。

其基本步骤如下：

(1) 读入帐号口令


```
String name=args[0];
String passwd=args[1];
```

分析: 这里为了简便而通过命令行读入帐号和口令, 实际程序中可以制作图形界面供用户输入。

(2) 生成随机数 (盐)

```
Random rand=new Random();
byte[] salt=new byte[12];
rand.nextBytes(salt);
```

分析: 创建字节数组 salt。使用 Java 中 Random 类生成随机数, 执行 Random 类的 nextBytes() 方法, 方法的参数为 salt, 即可生成的随机数并将随机数赋值给 salt。

(3) 生成 MessageDigest 对象

```
MessageDigest m=MessageDigest.getInstance("MD5");
```

分析: 执行 MessageDigest 类的静态方法 getInstance() 生成 MessageDigest 对象。其中传入的参数指定计算消息摘要所使用的算法。

(4) 传入盐和需要计算的字节数组

```
m.update(salt);
m.update(passwd.getBytes("UTF8"));
```

分析: 将第 2 步的盐和第 1 步的口令分别传递给 MessageDigest 对象的 update() 方法。

(5) 计算消息摘要

```
byte[] s=m.digest();
```

分析: 执行 MessageDigest 对象的 digest() 方法完成计算, 计算的结果通过字节类型的数组返回。

(6) 在文件中或数据库中保存帐号和口令的消息摘要

```
PrintWriter out= new PrintWriter(new FileOutputStream("passwdsalt.txt"));
out.println(name);
for (int i=0; i<salt.length; i++){
    out.print(salt[i]+",");
}
out.println("");
out.println(result);
```

分析: 这里将帐号、盐和口令消息摘要报存在 passwd.txt 文件中。对于盐, 这里将数组中各个 byte 值以数字保存在文件中, 各个数字之间以逗号隔开, 这样比较直观, 实际使用时可直接将字节数组以二进制保存。

如果攻击者得到了随机数 (盐) 和消息摘要, 虽然他也可以将各种长度的字符组合和用户所使用的盐合并起来计算消息摘要, 但是这个盐只在这一个口令中有效, 其他口令使用的是其他随机数, 因此攻击者对每个口令都要进行一次 4.4.3 小节中运行 AttackPass 计算字典的计算量, 而不像 4.4.3 小节那样耗时计算一次, 以后所有口令都就可以使用 DoAttack 快速进行匹配。因此如果用户口令在合理的长度内, 攻击者的计算量将非常巨大。

如果攻击者对加盐的口令也想像 4.4.3 小节那样先生成字典然后进行攻击, 则生成字典的计算量也将比 4.4.3 小节呈指数级增长直至可以认为不可能。如果口令有 12 位, 在加盐之

前，攻击者生成字典需要计算到长度为 12 个字符的组合。如果在口令前加了 12 位盐，则攻击者需要计算到长度为 24 个字符的组合，才能一劳永逸地通过简单匹配来获取消息摘要对应的口令。

★代码与分析:

本实例完整代码如下:

```
import java.util.*;
import java.io.*;
import java.security.*;
public class SetPassSalt{
    public static void main(String args[ ]) throws Exception{
        //读入账号口令
        String name=args[0];
        String passwd=args[1];
        //生成盐
        Random rand=new Random();
        byte[ ] salt=new byte[12];
        rand.nextBytes(salt);
        //计算消息摘要
        MessageDigest m=MessageDigest.getInstance("MD5");
        m.update(salt);
        m.update(passwd.getBytes("UTF8"));
        byte s[ ]=m.digest( );
        String result="";
        for (int i=0; i<s.length; i++){
            result+=Integer.toHexString((0x000000ff & s[i]) |
                                         0xffffffff00).substring(6);
        }
        //保存账号、盐和消息摘要
        PrintWriter out= new PrintWriter(
            new FileOutputStream("passwdsalt.txt"));

        out.println(name);
        for (int i=0; i<salt.length; i++){
            out.print(salt[i]+",");
        }
        out.println("");

        out.println(result);
        out.close();
    }
}
```

★运行程序

程序运行在 C:\java\ch4\password 目录，在命令行中输入

```
javac SetPassSalt.java
```

编译程序，输入

```
java SetPassSalt xyx akwi
```

运行程序，则将账号xyx和盐及口令的消息摘要保存在passwdsalt.txt文件中，打开该文件可以发现其内容如下：

```
xyx
67,45,-101,90,69,-31,100,-7,-71,110,-88,-99,
ada08d0495ca044cf0919b695544b7f6
```

再次输入

```
java SetPassSalt xyx akwi
```

打开 passwdsalt.txt 文件可以发现其内容如下

```
xyx
-80,-18,-116,-43,-108,-109,-54,73,1,-109,74,-82,
0c9c9bf284663373f630e297a0328c95
```

可见每次使用的盐都不一样不同，这样，同一个口令的计算出的消息摘要也不一样。攻击者得到 passwdsalt.txt 文件后，如果像 4.4.3 小节那样先生成字典，即使计算出 1 到 20 个字符长度的所有字符组合，也只能攻击 8 个字符长度的口令，如果用户口令长度超过 8 个，则字典将无效。而如果不预先生成字典进行攻击，则攻击者每次都必须先取出 passwdsalt.txt 文件中口令的盐的值，然后重复进行 4.4.3 小节的计算，而不是只需要计算一次，这样，如果一次计算需要耗时半年，而用户不到半年如一个月就修改一次口令，则攻击者将无法得逞。

4.4.5 验证加盐的口令

★ 实例说明

本实例演示如何验证 4.4.4 小节中加盐的口令。

★ 编程思路：

为了验证加盐的口令，需根据用户输入的账号在 4.4.4 小节保存口令的文件 passwdsalt.txt 中找到预先保存的、与该账号对应的盐和消息摘要。然后使用口令文件 passwdsalt.txt 中的盐和用户输入口令组合在一起计算消息摘要。比较两个消息摘要是否相等。若不相等则说明输入的口令不正确。其编程步骤如下：

- (1) 根据用户输入的账号读取对应的盐和消息摘要

```
BufferedReader in = new BufferedReader(new FileReader("passwdsalt.txt"));
while ((name = in.readLine()) != null) {
    salts=in.readLine( );
    passwd=in.readLine( );
```

```

        if (name.equals(args[0])){
            break;
        }
    }
}

```

分析：不妨将第一个命令行参数 args[0] 的值作为用户输入的账号。在 4.4.4 小节中，第一行保存的是帐号，第二行保存的是盐，第三行保存的是账号对应的口令的消息摘要。因此顺序读取账号、盐和口令摘要，直到所读取的帐号和命令行参数指定的账号相同。

(2) 将盐值转换为 byte 数组

```

String salttmp[ ]=salts.split(",");
byte salt[ ]=new byte[salttmp.length];
for (int i=0; i<salt.length; i++){
    salt[i]=Byte.parseByte(salttmp[i]);
}

```

分析：为了直观，口令文件 passsalt.txt 中保存的盐的值是以数字保存的，各个数字之间以逗号隔开。上一步读取的盐的字符串即是以逗号隔开的一串数字。这里使用字符串的 split() 方法以参数中的字符串（逗号）为分隔符，将字符串分解开来，存放在字符串数组中。然后使用 Byte 类的 parseByte() 方法将数组中各个字符形式的数字串转换成 byte 类型。

(3) 计算用户输入的口令的消息摘要

```

MessageDigest m=MessageDigest.getInstance("MD5");
m.update(salt);
m.update(args[1].getBytes("UTF8"));
byte s[ ]=m.digest( );

```

分析：不妨将第二个命令行参数 args[1] 的值作为用户输入的口令，加上上一步得到的盐，使用 4.4.1 小节中相同的步骤进行计算其消息摘要。

(4) 比较用户输入的口令的消息摘要和文件中保存的口令摘要是否一致

```

if(name.equals(args[0])&&result.equals(passwd)){
    System.out.println("OK");
}
else{
    System.out.println("Wrong password");
}

```

分析：当账号和口令摘要都和文件中保存的一致，则验证通过。

★代码与分析：

本实例完整代码如下：

```

import java.io.*;
import java.security.*;
public class CheckPassSalt{
    public static void main(String args[ ]) throws Exception{

```

```

/* 读取保存的盐和口令摘要 */
String name="";
String passwd="";
String salts="";
BufferedReader in = new BufferedReader(new
    FileReader("passwdsalt.txt"));
while ((name = in.readLine( )) != null) {
    salts=in.readLine( );
    passwd=in.readLine( );
    if (name.equals(args[0])){
        break;
    }
}
String salttmp[ ]=salts.split(",");
byte salt[ ]=new byte[salttmp.length];

for (int i=0; i<salt.length; i++){
    salt[i]=Byte.parseByte(salttmp[i]);
}
/* 生成用户输入的口令摘要 */
MessageDigest m=MessageDigest.getInstance("MD5");
m.update(salt);
m.update(args[1].getBytes("UTF8"));
byte s[ ]=m.digest( );
String result="";
for (int i=0; i<s.length; i++){
    result+=Integer.toHexString((0x000000ff & s[i]) |
        0xffffffff00).substring(6);
}
/* 检验口令摘要是否匹配 */
if(name.equals(args[0])&&result.equals(passwd)){
    System.out.println("OK");
}
else{
    System.out.println("Wrong password");
}
}
}

```

★运行程序

程序运行在 C:\java\ch4\password 目录，在命令行中输入

```
javac CheckPassSalt.java
```

编译程序，输入

```
java CheckPassSalt xyx akwi
```

运行程序，程序输出“OK”，表明帐号和口令正确。输入

```
java CheckPassSalt xyx qwert
```

提示“Wrong password”，可见可以正确进行验证。

本章介绍了认证机制的几个基本技术：消息摘要、消息验证码和数字签名，并给出了消息摘要和口令验证中的应用。

数字签名验证的实际上是公钥对应的私钥持有者认可了某个数据。但这个私钥持有者到底是谁则不一定可靠。攻击者有可能自己生成一个私钥和公钥，对外宣称该公钥是属于 A。这样，接收者将被误导。因此，验证数字签名所使用的公钥必须是从可靠的途径得到，如通过信誉很好的报纸等，此外下一章的数字证书也将从计算机的角度解决这一问题。

第 5 章 数字化身份的确定

——数字证书

本章重点:

第 4 章使用数字签名来确定数据的身份,保证了身份的确定性以及不可篡改性和不可否认性。使用数字签名的前提是接收数据者能够确性验证签名时所使用的公钥确实是某个人的,数字证书可以解决这一问题。

本章主要内容:

- 使用 Keytool 工具创建数字证书
- 使用 Keytool 工具和 Java 程序读取并显示数字证书
- 使用 Keytool 工具和 Java 程序维护密钥库
- 使用 Java 程序签发数字证书
- 对单个数字证书作初步检验

5.1 数字证书的创建

数字证书的主要功能是保存公钥和某个人或机构的对应关系,本节介绍几种数字证书的创建方法。它们都使用了 keytool 工具的-genkey 参数。

5.1.1 使用默认的密钥库和算法创建数字证书

★ 实例说明

本实例使用 J2SDK 提供的 keytool 工具用默认的密钥库和算法创建几个数字证书。

★ 运行程序

keytool 程序运行时加上命令行参数-genkey 即可。

在命令行中输入“keytool -genkey”将自动使用默认算法生成公钥和私钥,并以交互方式获得公钥持有者的信息。其交互过程如下,其中带下划线的字符为用户键盘输入的内容,其他为系统提示的内容。

```
C:\>keytool -genkey
输入 keystore 密码: 123456
您的名字与姓氏是什么?
[Unknown]: Xu Yingxiaol
您的组织单位名称是什么?
[Unknown]: Network Center
```

您的组织名称是什么？

[Unknown]: Shanghai University

您所在的城市或区域名称是什么？

[Unknown]: ZB

您所在的州或省份名称是什么？

[Unknown]: Shanghai

该单位的两字母国家代码是什么

[Unknown]: CN



CN=Xu Yingxiao1, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai,
C=CN 正确吗？

[否]: 是

输入<mykey>的主密码

(如果和 keystore 密码相同, 按回车): abcdefg

C:\>

如果使用中文操作系统, 上述操作中输入的“是”不能用英文“yes”代替。如果没有 DOS 下的中文输入系统的话, 可以在 Windows 的“记事本”中输入一个中文字符“是”, 然后点击 DOS 窗口左上角的  图标, 选择“编辑/粘贴”菜单, 或直接点击窗口中的  工具将中文字符“是”粘贴到 DOS 窗口, 如图 5-1 所示。

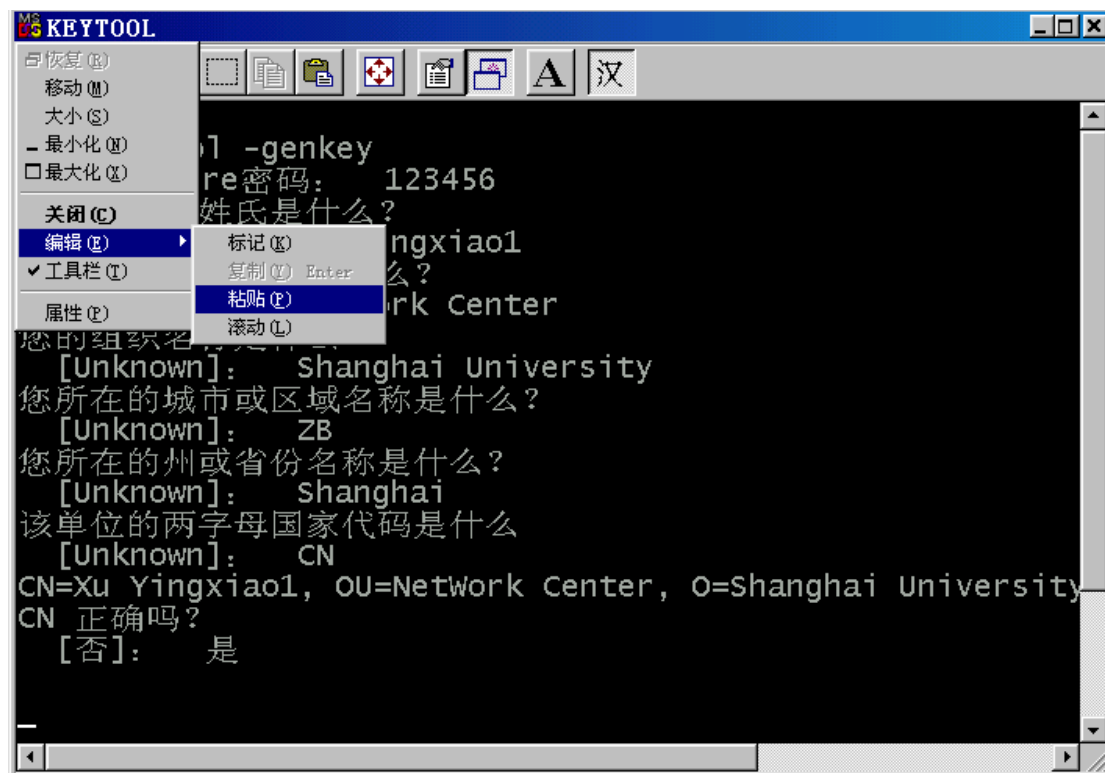


图 5-1 将中文字符粘贴到 DOS 窗口

以上操作将生成一个公钥和一个私钥, 这里并未指定使用何算法, 将使用默认的 DSA

算法。

同时上述操作将创建一个数字证书，证书中包含了新生成的公钥和一个名字为“CN=Xu Yingxiao1, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai, C=CN”的主体（人或机构）的对应关系。其中“CN=Xu Yingxiao1, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai, C=CN”是 X.509 格式的全名，包含了主体的国家、州、城市、机构、单位和名字。这样，这个证书将证明相应的公钥是这个人或机构所拥有的。

以上生成的公钥、私钥和证书都保存在用户的主目录中创建一个默认的文件“.keystore”中。如果使用的是 Windows 98 操作系统，用户的主目录是 c:\windows，在该目录下可以找到“.keystore”文件。如果是 Windows 2000 系统，用户主目录是 c:\Documents and Settings\用户名。

由于“.keystore”中包含了私钥，所以是一个需要保密的文件，因此上述操作提示为该文件设置一个密码：“输入 keystore 密码”，这里因为是第一次使用该密钥库，因此输入的密码“123456”将成为该默认的密钥库的密码（实际使用时应该设置复杂的口令）。以后再使用这个密钥库时必须提供这个口令才可以使用。

以上操作最后还提示“输入<mykey>的主密码”，这里“mykey”是默认的别名，使用该名字可以在密钥库“.keystore”中找到对应的公钥、私钥和证书。此处输入的密码是对应于该别名的私钥的密码，密钥库中每个别名可以使用不同的密码加以保护。

5.1.2 使用别名

密钥库中可以存放多个条目（公钥/私钥对和证书），它们在密钥库中以别名(alias)区分。5.1.1 小节在使用 keytool 工具时没有指定别名，因此系统使用了默认的别名 mykey。如果再次运行“keytool -genkey”，则系统将提示“keytool 错误： java.lang.Exception: 没有创建键值对，别名 <mykey> 已经存在”，因此当密钥库中有多个公钥/私钥对和证书时，应该使用别名。

★ 实例说明

本实例使用 J2SDK 提供的 keytool 工具用在默认的密钥库中利用别名增加多个证书。

★ 运行程序

keytool 程序运行时加上命令行参数 -alias 即可。

在命令行中输入“keytool -genkey -alias xuyingxiao2”将自动使用默认的算法生成别名为 xuyingxiao2 的公钥和私钥，并以交互方式获得公钥持有者的信息。其交互过程如下：

```
C:\>keytool -genkey -alias xuyingxiao2
```

```
输入 keystore 密码: 123456
```

```
您的名字与姓氏是什么？
```

```
[Unknown]: Xu Yingxiao2
```

```
您的组织单位名称是什么？
```

```
[Unknown]: Network Center
```

```
您的组织名称是什么？
```

```
[Unknown]: SHU
```

```
您所在的城市或区域名称是什么？
```

```
[Unknown]: ZB
```

```
您所在的州或省份名称是什么？
```

```
[Unknown]: SH
```

该单位的两字母国家代码是什么

[Unknown]: CN

CN=Xu Yingxiao2, OU=Network Center, O=SHU, L=ZB, ST=SH, C=CN 正确吗?

[否]: 是

输入<xuyingxiao2>的主密码

(如果和 keystore 密码相同, 按回车):

其中“输入 keystore 密码:”后面输入的内容必须和 5.1.1 小节相同的密码, 否则将无法访问密钥库, 并提示如下错误: “keytool 错误: java.io.IOException: Keystore was tampered with, or password was incorrect”, 这是因为 5.1.1 小节已经为默认的密钥库设置了该密码, 以后使用该密钥库都必须提供该密码。

在“输入<xuyingxiao2>的主密码”的提示后这里不妨直接按“回车键”, 这样该私钥将使用和密钥库相同的密码“123456”来保护。

以上操作将在用户主目录的“.keystore”文件(如对于 Windows 98 用户是 c:\windows\keystore)中增加一对公钥和私钥(DSA 算法), 同时增加一个数字证书, 证书中包含了新生成的公钥和一个名字为“CN=Xu Yingxiao2, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai, C=CN”的主体(人或机构)的对应关系。

5.1.3 使用指定的算法和密钥库和有效期

5.1.1 和 5.1.2 小节中使用的是默认的算法和密钥库, 本节介绍如何自己指定算法和密钥库。

★ 实例说明

本实例使用 J2SDK 提供的 keytool 工具用 RSA 算法和在指定的密钥库 mykeystore 中创建公钥/私钥对和证书。

★ 运行程序

keytool 的 -keyalg 参数可以指定密钥的算法, 如果需要指定密钥的长度, 可以再加上 -keysize 参数。密钥长度默认为 1024 位, 使用 DSA 算法时, 密钥长度必须在 512 到 1024 之间, 并且是 64 的整数倍。

Keytool 的 -keystore 参数可以指定密钥库的名称。密钥库其实是存放密钥和证书的文件, 密钥库对应的文件如果不存在自动创建。

-validity 参数可以指定所创建的证书有效期是多少天。

如在命令行中输入“keytool -genkey -alias mytest -keyalg RSA -keysize 1024 -keystore mykeystore -validity 4000”将使用 RSA 算法生成 1024 位的公钥/私钥对及整数, 密钥长度为 1024 位, 证书有效期为 4000 天。使用的密钥库为 mykeystore 文件。

```
C:\java\ch5>keytool -genkey -alias mytest -keyalg RSA -keysize 1024 -keystore mykeystore -validity 4000
```

输入 keystore 密码: wshr.ut

您的名字与姓氏是什么?

[Unknown]: Xu Yingxiao

您的组织单位名称是什么？

[Unknown]: Network Center

您的组织名称是什么？

[Unknown]: Shanghai University

您所在的城市或区域名称是什么？

[Unknown]: ZB

您所在的州或省份名称是什么？

[Unknown]: Shanghai

该单位的两字母国家代码是什么

[Unknown]: CN

CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai, C=CN
正确吗？

[否]: 是

输入<mytest>的主密码

(如果和 keystore 密码相同, 按回车):

C: \java\ch5>

由于当前目录下没有 `mykeystore` 文件, 因此以上操作将在当前目录建立文件名为 `mykeystore` 的文件, 并提示输入一个密码加以保护: “输入 keystore 密码: ”。因为这里使用的密钥库和 5.1.1 小节及 5.1.2 小节不是同一个文件, 因此这里输入的密码和 5.1.1 小节及 5.1.2 小节没有必要一致, 这里不妨设置为 “wshr.ut”。这样, 以后再使用这个密钥库文件时必须提供该密码。

对其中的 “输入<mytest>的主密码”, 这里不妨直接按回车键, 这样 `mykeysotre` 文件中的 `mytest` 条目将使用和密钥库相同的密码: “wshr.ut”。

5.1.4 使用非交互模式

★ 实例说明

前面各小节都是通过屏幕交互方式输入证书拥有者的信息, 本实例使用 J2SDK 提供的 `keytool` 工具直接在命令行参数中指定所有的信息来创建公钥/私钥对和证书。

★ 运行程序

前面各小节交互输入的内容主要有:

密钥库的密码: 这可以用命令行参数 `-storepass` 来指定。

别名条目的主密码: 这可以用命令行参数 `-storepass` 来指定。

证书拥有者的信息: 这可以用命令行参数 `-dname` 来指定。该参数的值是一个字符串, 其格式是: “CN=XX, OU= XX, O= XX, L= XX, ST= XX, C= XX”, 其中 CN, OU,O,L,ST,C 分别代表以前各小节交互性输入的名字与姓氏(Common Name)、组织单位名称(Organization Unit)、组织名称 ((Organization)、城市或区域名称 (Locality)、州或省份名称 (State)、国家代码 (Country)。

如可以在命令行输入如下内容来向密钥库 `mykeystore` 添加条目。

```
keytool -genkey -dname "CN=tmp, OU=NC, O=Shanghai University, L=ZB, ST=Shanghai, C=CN" -alias tmp -keyalg RSA -keystore mykeystore -keypass wshr.ut -storepass wshr.ut -validity 1000
```

该命令使用的密钥库和 5.1.3 小节一样，是 `mykeystore` 文件，因此这里的 `-storepass` 参数必须和 5.1.3 小节一样使用 “`wshr.ut`”，否则无法访问密钥库。条目的别名不妨使用 `tmp`，条目的密码通过 `-keypass` 参数指定，这里不妨仍旧使用 “`wshr.ut`”，也可以任意设定。

以上命令必须在命令行中一行输完，有些操作系统对命令行中一行输入的字符有限制，这时可以将以上命令放在一个批处理文件中（必须在一行中，中间不能换行）。如将该命令用 Windows 的“记事本”编辑，以文件名 `5.1.4.bat` 保存，则运行 `5.1.4.bat` 将自动完成所有操作。

又如执行下面的命令将在当前目录生成一个密钥库文件 `lfkeystore`，密钥库的密码是 `wshr.ut`，其中存放的证书别名为 `lf`。有效期为 3500 天。证书中包含的是需要公开的信息：一个主体 “`CN=Liu Fang, OU=Packaging, O=Shanghai University, L=ZB, ST=Shanghai, C=CN`” 拥有某个 RSA 公钥。公钥对应的私钥也保存在密钥库 `lfkeystore` 中，并用密码 `wshr.ut` 加以保护。

```
keytool -genkey -dname "CN=Liu Fang, OU=Packaging, O=Shanghai University, L=ZB, ST=Shanghai, C=CN" -alias lf -keyalg RSA -keysize 1024 -keystore lfkeystore -keypass wshr.ut -storepass wshr.ut -validity 3500
```

将该命令用 Windows 的“记事本”编辑，命令一行输完，以文件名 `5.1.4_2.bat` 保存，则运行 `5.1.4_2.bat` 将自动完成所有操作。

5.2 数字证书的显示

5.1 节用各种方式创建了多个数字证书，本节使用各种方式显示这些数字证书的信息，它们有的使用 `keytool` 工具的 `-list` 参数，有的直接通过 Java 编程来实现。

5.2.1 使用 Keytool 直接从密钥库显示条目信息

★ 实例说明

本实例使用 J2SDK 提供的 `keytool` 工具直接从密钥库中显示证书信息。

★ 运行程序

`keytool` 的命令行参数 `-list` 可以显示密钥库中的证书信息，如输入：

```
keytool -list
```

则显示默认的密钥库中的证书信息。如下：

```
C:\java\ch5>keytool -list
```

```
输入 keystore 密码: 123456
```

```
Keystore 类型: jks
```

Keystore 提供者: SUN

您的 keystore 包含 2 输入

```
xuyingxiao2, 2002-11-22, keyEntry,  
认证指纹 (MD5): 65: C9: FD: 8C: 82: C7: 36: E1: 7C: D9: AD: 9A: 34: 25: 5C: 71  
mykey, 2002-11-22, keyEntry,  
认证指纹 (MD5): BE: F1: 9F: 45: 5F: 4E: 02: FF: 94: 83: 39: 73: E1: F5: 59: 9C
```

程序开始要求输入密钥库的密码, 在 5.1.1 小节我们已经为默认密钥库设置了密码“123456”, 因此这里必须输入相同的密码才能使用密钥库。

在 5.1.1 和 5.1.2 小节我们已向默认的密钥库中添加了两个条目: **mykey** 和 **xuyingxiao2**, 在此处的输出信息中可以看到这两个条目的名称、创建日期、条目类型 (**keyEntry**, 密钥条目) 以及认证指纹。认证指纹其实是该条目的消息摘要。

如果进一步使用 **-alias** 参数则可以显示指定的条目的信息, 如:

```
C:\java\ch5>keytool -list -alias xuyingxiao2  
输入 keystore 密码: 123456  
xuyingxiao2, 2002-11-22, keyEntry,  
认证指纹 (MD5): 65: C9: FD: 8C: 82: C7: 36: E1: 7C: D9: AD: 9A: 34: 25: 5C: 71
```

如果进一步使用 **-keystore** 参数则可以显示指定的密钥库中的证书信息, 如:

```
C:\java\ch5>keytool -list -keystore mykeystore  
输入 keystore 密码: wshr.ut
```

Keystore 类型: jks
Keystore 提供者: SUN

您的 keystore 包含 2 输入

```
mytest, 2002-12-5, keyEntry,  
认证指纹 (MD5): B2: DC: 75: CD: 60: B7: 1E: 7A: 97: EE: E8: A4: 31: D6: 26: C6  
tmp, 2002-12-5, keyEntry,  
认证指纹 (MD5): 5C: FA: ED: 8E: AE: 30: 1B: 2B: CF: 39: ED: 4D: 6F: 94: E1: 6B
```

5.2.2 使用 Keytool 直接从密钥库显示证书详细信息

★ 实例说明

本实例使用 J2SDK 提供的 **keytool** 工具直接从密钥库中显示证书的详细信息。

★ 运行程序

5.2.1 的各个命令加上 **-v** 参数可以显示证书的详细信息, 如:

```
C:\java\ch5>keytool -list -v -keystore lfkeystore -alias lf  
别名名称: lf  
创建日期: 2002-12-5
```

输入类型: KeyEntry

认证链长度: 1

认证 [1]:

Owner: CN=Liu Fang, OU=Packaging, O=Shanghai University, L=ZB, ST=Shanghai,

发照者: CN=Liu Fang, OU=Packaging, O=Shanghai University, L=ZB, ST=Shanghai
CN

序号: 3deec441

有效期间: Thu Dec 05 11:13:05 CST 2002 至: Thu Jul 05 11:13:05 CST 2012

认证指纹:

MD5: 55: 73: 8D: 16: 05: E1: F8: 5F: F8: 25: C7: 29: C3: D6: 48: 67

SHA1: 3F: 75: 6A: DC: E7: 7B: 32: 64: C5: 99: 1E: CC: 9B: 9E: 77: 88: 59: 21: C2: 33

其中包含了发照者(签发者)、序号、有效期、MD5 和 SHA1 认证指纹等额外信息, 其含义在本章后续内容中将涉及。

5.2.3 使用 Keytool 将数字证书导出到文件

★ 实例说明

本实例使用 J2SDK 提供的 keytool 工具将指定的证书从密钥库导出为编码过和没编码过两种格式的文件。

★ 运行程序

使用 keytool 的 -export 参数可以将别名指定的证书导出到文件, 文件名通过 -file 参数指定。如输入如下命令:

```
C:\java\ch5>keytool -export -alias xuyingxiao2 -file xuyingxiao2.cer
```

输入 keystore 密码: 123456

保存在文件中的认证 <xuyingxiao2.cer>

则将默认密钥库中的 xuyingxiao2 条目对应的证书导出到文件 xuyingxiao2.cer 中。由于命令行中没有用 storepass 给出密码, 因此屏幕提示输入 keystore 密码。由于证书中不包含私钥, 因此不需要条目的主密码。

该操作完成后将在当前目录中创建 xuyingxiao2.cer 文件, 该文件即是默认密钥库中的 xuyingxiao2 条目对应的证书, 它包含了公钥和主体的对应关系, 内容也可以公开。

输入如下命令则可以指定密钥库:

```
C:\java\ch5>keytool -export -alias lf -file lf.cer -keystore lfkeystore  
-storepass wshr.ut
```

保存在文件中的认证 <lf.cer>

该操作完成后将在当前目录中创建 lf.cer 文件。

如果用文本编辑器打开 xuyingxiao2.cer 或 lf.cer, 将会发现它是二进制文件, 有些内容无法显示, 这不利于公布证书。在导出证书时加上 -rfc 参数则可以使用一种可打印的编码格式来保存证书。如:

```
C:\java\ch5>keytool -export -alias mytest -file mytest.cer -keystore  
mykeystore -storepass wshr.ut -rfc
```

保存在文件中的认证 <mytest.cer>

```
-----BEGIN CERTIFICATE-----
MIICZjCCAc8CBD3uEMwDQYJKoZIhvcNAQEEBQAwEjELMAkGA1UEBhMC004xETAPBgNUBAGTCFNo
YW5naGFpMQswCQYDUQAHewJaQjEcmBoGA1UEChMTU2hhbmdoYWkgUW5pdMUyc210eTEXMBUGA1UE
CxM0TmU0d29yayBDZW50ZXIxFDASBgNUBAMTC1h1IF1pbmd4aWFuMBA4XDTAyMTIwNTAyNTYwM1oX
DEZMTExNzAyNTYwM1owejELMAkGA1UEBhMC004xETAPBgNUBAGTCFNoYW5naGFpMQswCQYDUQAH
EwJaQjEcmBoGA1UEChMTU2hhbmdoYWkgUW5pdMUyc210eTEXMBUGA1UECzM0TmU0d29yayBDZW50
ZXIxFDASBgNUBAMTC1h1IF1pbmd4aWFuMIGfMA0GCsqGSIB3DQEBAQUAA4GNADCBiQKBgQdqahNN
3XUQ/CS3R3XBs7EaXHEzDQg+zflwCBWnNH1W1DgYXMAfJFIyUpt0sPb3C1e0DFUn+fr/ZG8ZGG
HRFtrzlms7CFv8Z10vU+SnDJGzn3WmoE/YeSuqSUhNIyc0rOMUC3sANYH0GPKv0K6vLv3YLZbKw5
5rnJB6YUoGy6wIDAQABMA0GCsqGSIB3DQEBBAUA4GBAL7U8zz+UxY03P6gHHZxrzHz0ww2Lh0y
H4eztB2Cu0q73101kLyoz0JFYe09Gd99q0XyShnBaxk09+z0xhpAn6lrjEnaziUn2ciRZ9sza0eW
cNaRaSRD1YFsnCwdTUAjAWUytif7G/OPShYLMelRGUJQx3Bibuykdgm+InwmAEft
-----END CERTIFICATE-----
```

MD5: B2: DC: 75: CD: 60: B7: 1E: 7A: 97: EE: E8: A4: 31: D6: 26: C6
SHA1: 32: E5: 89: 16: 7E: 25: 7F: 86: 16: 94: 34: 36: 95: 44: D7: CF: 14: C8: F2: 1E

5.2.5 在 Windows 中从文件显示证书

★ 实例说明

本实例在 Windows 中直接显示 5.2.3 小节导出的证书文件。

★ 运行程序

5.2.3 小节导出的证书文件中，只要文件名以 .cer 为后缀，Windows 操作系统就可以直接识别。如在 Windows 中双击 lf.cer 图标，将出现图 5-3 所示证书窗口。其中包含了证书的所有者、颁发者、有效期等信息，这些信息和 5.2.4 小节使用 keytool 显示出的信息一致。

由于该证书是用自己的私钥对该证书进行数字签名的，即自己给自己签发的证书，因此窗口中显示警告信息：“该证书发行机构根证书没受信任”。在后续章节中将介绍证书的签发问题。



图 5-3 证书的常规信息

点击图 5-3 的“详细资料”，可以看到证书的版本、序号、签名算法、颁发者、有效期、主题（即全名）、公钥算法、拇指算法、拇指等信息。其中的拇指即认证指纹，和 5.2.4 小节显示的 SHA1 认证指纹相同。如图 5-4 所示。

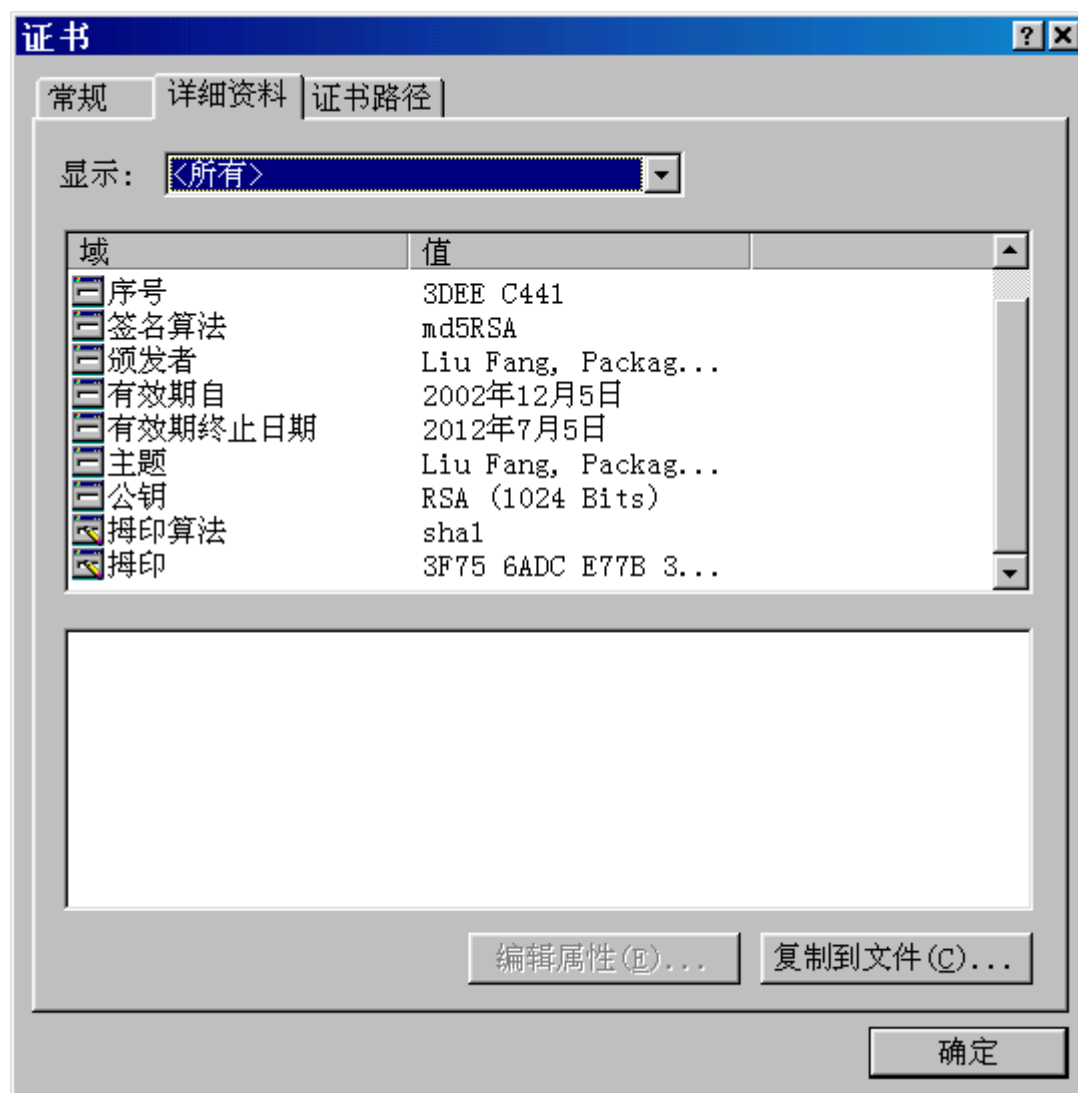


图 5-4 证书的详细信息

同样，点击编码过的证书文件如 mytest.cer 可以看到类似信息。

5.2.6 Java 程序从证书文件读取证书

前面各小节都使用了 keytool 工具来显示证书，本小节开始涉及编程部分，通过自己编写的 Java 程序来访问密钥库。

★ 实例说明

本实例使用 5.2.3 小节得到的证书文件 mytest.cer、lf.cer 等，演示了如何编程读取证书的信息。

★ 编程思路:

在 java.security.cert 包中有 Certificate 类代表证书, 使用其 toString() 方法可以得到它所代表的证书的所有信息。为了得到 Certificate 类型的对象, 可以使用 java.security.cert 包中的 CertificateFactory 类, 它的 generateCertificate() 方法可以从文件输入流生成 Certificate 类型的对象。具体步骤如下:

(10) 获取 CertificateFactory 类型的对象

```
CertificateFactory cf=CertificateFactory.getInstance("X.509");
```

分析: CertificateFactory 类是一个工厂类, 必须通过 getInstance() 方法生成对象, 其参数指定证书的类型, 这里使用 "X.509", 它是一个广泛使用的数字证书标准。

(11) 获取证书文件输入流

```
FileInputStream in=new FileInputStream(args[0]);
```

分析: 不妨从命令行参数读取 5.2.3 小节的证书文件, 创建文件输入流。

(12) 生成 Certificate 类型的对象

```
Certificate c=cf.generateCertificate(in);
```

分析: 执行第 1 步得到的 CertificateFactory 类型的对象的 generateCertificate() 方法, 以第 2 步的文件输入流作为其参数。

(13) 显示证书内容

```
String s=c.toString();
```

分析: 执行 toString() 方法, 可以将其写入文件, 或在屏幕上显示出来。

★代码与分析:

完整代码如下:

```
import java.io.*;
import java.security.cert.*;

public class PrintCert{
    public static void main(String args[ ]) throws Exception{
        CertificateFactory cf=CertificateFactory.getInstance("X.509");
        FileInputStream in=new FileInputStream(args[0]);
        Certificate c=cf.generateCertificate(in);
        in.close();
        String s=c.toString();
        // 显示证书
        FileOutputStream fout=new FileOutputStream("tmp.txt");
        BufferedWriter out=new BufferedWriter(new OutputStreamWriter(fout));
        out.write(s,0,s.length());
        out.close();
    }
}
```

程序最后创建文件输出流, 将读取到的证书信息保存在文件 tmp.txt 中。

★运行程序

输入 `java PrintCert mytest.cer` 来运行程序，将创建文件 `tmp.txt`，其内容如下：

```
[
[
  Version: V1
  Subject: CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB,
ST=Shanghai, C=CN
  Signature Algorithm: MD5withRSA, OID = 1.2.840.113549.1.1.4

  Key:  com.sun.net.ssl.internal.ssl.JSA-RSAPublicKey@d99a4d
  Validity: [From: Thu Dec 05 10:56:03 CST 2002,
              To: Sun Nov 17 10:56:03 CST 2013]
  Issuer: CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB,
ST=Shanghai, C=CN
  SerialNumber: [    3deec043 ]

]
  Algorithm: [MD5withRSA]
  Signature:
0000: BE D5 F3 3C FE 53 16 0E   DC FE A0 1C 7C F1 AF 31   ...<.S.....1
0010: F3 3B 0C 36 2E 1D 32 1F   87 B3 B4 1D 82 BB 4A BB   .;.6..2.....J.
0020: DE 5D 35 90 BC A8 CF 42   45 61 ED 3D 19 DF 7D AB   .]5....BEa.=....
0030: 45 F2 4A 19 C1 6B 19 0E   F7 EC CE C6 1A 40 9F A9   E.J..k.....@..
0040: 6B 8C 49 DA CC 85 67 D9   C8 91 67 DB 33 6B 47 96   k.I...g...g.3kG.
0050: 70 D6 91 69 24 43 D5 81   6C 9D C5 9D 4D 40 23 01   p..i$C..l...M@#.
0060: 65 72 B6 27 FB 1B F3 8F   4A 16 0B 31 E2 EB 19 42   er.'....J..l...B
0070: 50 C7 70 62 6E FC A4 76   03 3E 22 7C 26 00 47 ED   P.pbn..v.>".&.G.

]
```

从中同样可以看到该证书的版本、主体、签名算法、公钥、有效期、签名、序号和签名等信息。从中还可以看出签名者和主体完全相同，即该证书是用自己对应的私钥进行数字签名的，签名的算法是“MD5withRSA”，签名的结果也以十六进制和二进制显示了出来。

5.2.7 Java 程序从密钥库直接读取证书

★ 实例说明

5.2.6 小节的程序依赖于用 `keytool` 先将数字证书导出，本实例通过自己编写的 Java 程序来直接访问密钥库读取证书的信息。

★ 编程思路：

在 `java.security` 包中的 `KeyStore` 类代表密钥库，使用其 `load()` 方法可以从密钥库文件输入流中加载密钥库，使用其 `getCertificate()` 方法可以从密钥库中提取证书。具体步骤如下：

(1) 创建密钥库的文件输入流

```
FileInputStream in=new FileInputStream(name);
```

分析：其中 name 是字符串类型的参数，即密钥库文件的文件名。

(2) 创建 KeyStore 对象

```
KeyStore ks=KeyStore.getInstance("JKS");
```

分析：KeyStore 类是工厂类，必须用 getInstance() 方法生成对象。其参数指定密钥库的类型。一般是“JKS”，如果创建密钥库时使用了其他类型的密钥库，这里应指定对应的类型。注意 KeyStore 类的名称中字母 K 和 S 均为大写。

(3) 加载密钥库

```
ks.load(in,pass.toCharArray());
```

分析：执行上 1 步创建的 KeyStore 对象的 load() 方法加载密钥库。其第一个参数是第 1 步得到的文件输入流，第二个参数是创建密钥库时设置的口令。对于 5.1.1 小节的缺省密钥库，当时设置的口令为“123456”，5.1.3 小节的密钥库 mykeystore，当时设置的口令为“wshr.ut”。

(4) 获取密钥库中的证书

```
Certificate c=ks.getCertificate(alias);
```

分析：执行第 2 步得到的 KeyStore 对象的 getCertificate() 方法，可以获得密钥库中的证书，得到 Certificate 对象。以后，便可以像 5.2.6 小节一样使用该证书了。

★代码与分析：

完整代码如下：

```
import java.io.*;
import java.security.*;
import java.security.cert.Certificate;
import java.security.cert.CertificateFactory;
public class PrintCert2{
    public static void main(String args[ ]) throws Exception{
        String pass="wshr.ut";
        String alias="mytest";
        String name="mykeystore";
        FileInputStream in=new FileInputStream(name);
        KeyStore ks=KeyStore.getInstance("JKS");
        ks.load(in,pass.toCharArray());
        Certificate c=ks.getCertificate(alias);
        in.close();
        System.out.println(c.toString());
    }
}
```

在程序的 import 语句中使用了“import java.security.cert.Certificate;”而不是“import java.security.cert.*”。这是因为 J2SDK1.4 中在 java.security 和 java.security.cert 包中都有 Certificate 类，由于本示例程序既用到了 java.security 包又用到了 java.security.cert 包，因此

若使用“import java.security.cert.*”，则编译器将无法确定使用的是哪个 Certificate 类，编译时将报错。

java.security 包中的 Certificate 类是为了和以前版本的 JDK 兼容而保留的，已经不建议使用。编写新的程序是应该使用 java.security.cert 包中的 Certificate 类。如果在 import 语句中使用的是“import java.security.cert.*”，则在程序中应该使用 java.security.cert.Certificate 来使用该类型。如本实例中若将

```
import java.security.cert.Certificate;
import java.security.cert.CertificateFactory;
```

改为：

```
import java.security.cert.*;
```

则程序中的

```
Certificate c=ks.getCertificate(alias);
```

应该改为：

```
java.security.cert.Certificate c=ks.getCertificate(alias);
```

本实例中使用的密钥库是 5.1.3 小节创建的 mykeystore 文件，在 5.1.3 小节为其设置了口令“wshr.ut”，并添加了一个条目：mytest，本实例准备读取 mytest 条目对应的证书。因此，在程序开头提供了这三条信息。此外程序中只是在加载密钥库时用到了口令，在提取证书时并不需要 mytest 的口令。

如果准备提取 5.1.1 小节的默认的密钥库中 mykey 条目，则可以将程序中

```
String pass="wshr.ut";
String alias="mytest";
String name="mykeystore";
```

改为：

```
String pass="123456";
String alias="mykey";
String userhome=System.getProperty("user.home");
String name=userhome+File.separator+".keystore";
```

其中“123456”是 5.1.1 小节为默认的密钥库设置的密码，“mykey”是 5.1.1 小节添加的条目默认的别名，也可以使用 5.1.2 小节中的别名 xuyingxiao2。由于不同操作系统中默认密钥库的存放位置和文件分隔符都可能不同，因此程序通过 System.getProperty("user.home") 语句获取用户主目录，通过 File.separator 获取文件分隔符，最后和文件名“.keystore”共同组成密钥库文件的完整路径。对于 Windows 98 操作系统，通过这种方式获得的 name 的值是：“c:\windows\keystore”。在 5.1.1 小节为 mykey 条目设置的口令“abcdef”在程序中并不需要，因为该口令主要是保护密钥库中的私钥。

★运行程序

输入 java PrintCert2 运行程序，程序显示 mykeystore 密钥库中的 mytest 条目对应的证书。其输出结果很多，可以使用“java PrintCert2> tmp2.txt”将输出结果重定向到文件 tmp2.txt 中，可以看到 tmp2.txt 中的内容和 5.2.6 小节得到的内容完全一样。

5.2.8 Java 程序显示证书指定信息（全名/公钥/签名等）

★ 实例说明

5.2.6 和 5.2.7 小节在得到证书后都是将其用 toString() 方法打印出所有内容，本小节介绍如何从证书中只提取所需要的信息。包括版本、序列号、主体的全名、签发者的全名、有

效期、签名算法、签名和公钥等。其中公钥和全名（人或机构）是证书中包含的最重要的信息。

★ 编程思路：

无论使用 5.2.6 小节还是 5.2.7 小节的方法获得证书对象后，都可以使用其方法获得各种信息。假设证书对象可通过变量 t 访问，可以按照如下方法获取其指定信息

(1) 获取证书

```
CertificateFactory cf=CertificateFactory.getInstance("X.509");
FileInputStream in=new FileInputStream(args[0]);
java.security.cert.Certificate c=cf.generateCertificate(in);
in.close();
```

分析：这里不妨从 5.2.4 小节中导出的文件中读取 Certificate 对象。

(2) 将证书转换为 X509 类型

```
X509Certificate t=(X509Certificate) c;
```

分析：X509Certificate 是 Certificate 类的子类，Keytool 工具生成的证书是符合 X509 标准的，X509Certificate 类中提供了更多的方法可以获取 X509 证书的相关信息。以下几个步骤给出了几个方法，可以根据需要选用。

(3) 获取版本号

```
t.getVersion()
```

分析：getVersion() 方法返回整型数代表证书符合 X509 标准的哪个版本。

(4) 获取序列号

```
t.getSerialNumber().toString(16)
```

分析：每个证书在创建时都会分配一个唯一的序列号，用 getSerialNumber() 方法可以获取，它返回的 BigInteger 类型的对象，通过其方法 toString(16) 可以将其转换为 16 进制的字符串。

(5) 获取主体和签发者的全名

```
t.getSubjectDN()
```

```
t.getIssuerDN()
```

分析：证书中包含的主要信息是公钥和主体（人或机构）的对应关系，该主体的全名可通过 getSubjectDN() 方法获得，它返回 Principal 类型的对象，可直接转换为“CN=XX, OU=XX, O= XX, L= XX, ST= XX, C= XX”类型的字符串。

证书的信息由自己或另外的机构签发，签发者的全名类似地通过 t.getIssuerDN() 方法获得。

(6) 获取证书的有效期

```
t.getIssuerDN()
```

```
t.getNotBefore()
```

分析：getIssuerDN() 和 getNotBefore() 方法分别获取证书的有效期起始日期和有效期截至日期。它们返回的是 Date() 类型的对象。

(7) 获取证书的签名算法

```
t.getSigAlgName()
```

分析：自己或其他机构签发该证书是用签发者的私钥对该证书进行数字签名来实施的，数字签名所使用的算法名称可以通过 getSigAlgName() 方法获得。其返回值是字符串类型。

(8) 获取证书的签名

```
byte[] sig=t.getSignature();
```

```
new BigInteger(sig).toString(16)
```

分析：数字签名的结果可以用 `getSignature()` 方法获得，该方法返回值是 `byte` 类型的数组，数字签名一般可用 16 进制来表示，因此使用 `BigInteger` 类将 `byte` 类型的数组转换为 `BigInteger` 类型，既而用 `BigInteger` 类的 `toString(16)` 方法将其转换为 16 进制字符串。

(9) 获取证书的公钥

```
t.getPublicKey()
```

```
byte[] pkenc=pk.getEncoded()
```

分析：证书中包含的主要信息是公钥和主体（人或机构）的对应关系。主体已在第 5 步获得，而公钥则可以使用 `getPublicKey()` 方法获得。它返回 `PublicKey` 类型的对象，可以用于验证签名、显示等。这里不妨将其编码打印出来。

★代码与分析：

完整代码如下：

```
import java.io.*;
import java.security.*;
import java.security.cert.*;
import java.util.*;
import java.math.*;

public class ShowCertInfo{
    public static void main(String args[] ) throws Exception{
        CertificateFactory cf=CertificateFactory.getInstance("X.509");
        FileInputStream in=new FileInputStream(args[0]);
        java.security.cert.Certificate c=cf.generateCertificate(in);
        in.close();
        X509Certificate t=(X509Certificate) c;
        System.out.println("版本号 "+t.getVersion());
        System.out.println("序列号 "+t.getSerialNumber().toString(16));
        System.out.println("全名 "+t.getSubjectDN());
        System.out.println("签发者全名\n"+t.getIssuerDN());
        System.out.println("有效期起始日 "+t.getNotBefore());
        System.out.println("有效期截至日 "+t.getNotAfter());
        System.out.println("签名算法 "+t.getSigAlgName());
        byte[] sig=t.getSignature();
        System.out.println("签名\n"+new BigInteger(sig).toString(16));
        PublicKey pk=t.getPublicKey();
        byte[] pkenc=pk.getEncoded();
        System.out.println("公钥");
        for(int i=0;i<pkenc.length;i++){
            System.out.print(pkenc[i]+" ");
        }
    }
}
```

★运行程序

输入 `java ShowCertInfo mytest.cer > tt.txt` 运行程序，显示 5.2.3 小节导出的证书 `mytest.cer`，并将显示结果重定向到文件 `tt.txt` 中，`tt.txt` 中得到的输出如下：

```
版本号 1
序列号 3deec043
全名 CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB,
ST=Shanghai, C=CN
签发者全名
CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai,
C=CN
有效期起始日 Thu Dec 05 10:56:03 CST 2002
有效期截至日 Sun Nov 17 10:56:03 CST 2013
签名算法 MD5withRSA
签名
-412a0cc301ace9f123015fe3830e50ce0cc4f3c9d1e2cde0784c4be27d44b54421a2ca6f43
5730bdba9e12c2e6208254ba0db5e63e94e6f108133139e5bf60569473b625337a9826376e9
824cc94b8698f296e96dbbc2a7e93623a62b2bfdcfe9a8d49d804e40c70b5e9f4ce1d14e6bd
af388f9d91035b89fcc1dd83d9ffb813
公钥
48, -127, -97, 48, 13, 6, 9, 42, -122, 72, -122, -9, 13, 1, 1, 1, 5, 0, 3, -127, -115, 0, 48, -127
, -119, 2, -127, -127, 0, -22, 106, 19, 77, -35, 117, 16, -4, 36, -73, 71, 117, -63, -77, -79, 2
6, 92, 113, 51, 13, 8, 62, -51, -7, 93, 88, 32, 112, 53, -29, 71, -43, 109, 67, -127, -123, -52,
105, -14, 69, 35, 37, 79, -73, 75, 15, 111, 112, -91, 122, -128, -59, 82, 127, -97, -81, -10, 7
0, -15, -111, -122, 29, 17, 109, -81, 57, 102, -77, -80, -123, -65, -58, 117, 58, -11, 126, 74
, 112, -55, 27, 57, -9, 90, 106, 4, -3, -121, -110, -70, -92, -108, -124, -46, 50, 112, -22, -5
0, 49, 64, -73, -80, 3, 88, 31, 65, -113, -110, -13, -92, -22, -14, -17, -35, -126, -39, 108, -
84, 57, -26, -71, -55, 7, -90, 21, -96, 108, -80, -21, 2, 3, 1, 0, 1,
```

该结果和以前各种方法显示的信息一致。

5.3 密钥库的维护

5.1 和 5.2 节使用 `keytool` 在密钥库中创建公钥/私钥对，将公钥的持有者信息以数字证书的形式保存在密钥库中，并分别使用 `keytool` 和 `Java` 程序来读取数字证书的信息。本节介绍如何对密钥库中的这些证书和密钥进行删除、修改等维护。

5.3.1 使用 `Keytool` 删除指定条目

★ 实例说明

本实例使用 J2SDK 提供的 `keytool` 工具从密钥库中删除指定的条目。

★运行程序

`keytool` 的命令行参数 `-delete` 可以删除密钥库中的条目。进行删除操作之前先在密钥库中添加一些条目供试验。也可将 `mykeystore` 文件做个备份，以便试验完后恢复原来的密钥。类似 5.1.4 小节的做法，在批处理文件 5.3.1.bat 中输入如下命令（必须输入在一行中）：

```
keytool -genkey -dname "CN=tmp1, OU=tmp, O= tmp, L= tmp, ST= tmp, C= tmp " -alias  
tmp1 -keyalg RSA -keystore mykeystore -keypass abcdefg -storepass wshr.ut
```

执行 5.3.1.bat 批处理文件，将在密钥库 `mykeystore` 中创建一个临时条目 `tmp1`。在 `mykeystore` 文件所在目录中执行：

```
keytool -list -keystore mykeystore -storepass wshr.ut
```

将显示：

```
Keystore 类型: jks  
Keystore 提供者: SUN
```

您的 keystore 包含 3 输入

```
mytest, 2002-12-5, keyEntry,  
认证指纹 (MD5): B2: DC: 75: CD: 60: B7: 1E: 7A: 97: EE: E8: A4: 31: D6: 26: C6  
tmp, 2002-12-5, keyEntry,  
认证指纹 (MD5): 5C: FA: ED: 8E: AE: 30: 1B: 2B: CF: 39: ED: 4D: 6F: 94: E1: 6B  
tmp1, 2002-12-5, keyEntry,  
认证指纹 (MD5): 7B: 0F: 2B: C3: 68: 20: 5C: C6: 34: F3: 90: 10: 1C: 0E: 66: 28
```

从中可以看出密钥库 `mykeystore` 中共有三个条目。

下面开始删除该条目，如输入：

```
keytool -delete -alias tmp1 -keystore mykeystore
```

在屏幕提示“输入 keystore 密码”时输入密钥库的密码“`wshr.ut`”。这样，刚才执行 2.bat 添加的条目 `tmp1` 将被删除。执行

```
keytool -list -keystore mykeystore -storepass wshr.ut
```

将看到密钥库 `mykeystore` 中只剩下两个条目。

5.3.2 使用 Keytool 修改指定条目的口令

★ 实例说明

本实例使用 J2SDK 提供的 `keytool` 工具修改密钥库中指定条目的口令。

★运行程序

`keytool` 的命令行参数 `-keypassword` 可以修改密钥库中指定条目的口令。进行删除操作之前和 5.3.1 小节一样执行 2.bat 创建一个临时的条目 `tmp1`。在 5.3.1 小节中可以看到该条目在创建时设置的口令为“`abcdefg`”，现在我们准备把它改成“`123456`”。

其操作过程如下：

```
C:\java\ch5>keytool -keypasswd -alias tmp1 -keystore mykeystore  
输入 keystore 密码: wshr.ut
```

输入<tmp1>的主密码 abcdefg

新 <tmp1> 的主密码: 123456

重新输入新 <tmp1> 的主密码: 123456

这时, tmp1 条目的口令就改成了 123456, 以后如果要从密钥库 mykeystore 中提取 tmp1 条目对应的私钥时就必须提供该口令。

以上交互式操作也可全部通过命令行参数指定, 例如如果想把密码再由 123456 改为 asdfgh, 则可以在命令行输入如下命令:

```
keytool -keypasswd -alias tmp1 -keypass 123456 -new asdfgh -storepass wshr.ut
-keystore mykeystore
```

5.3.3 Java 程序列出密钥库所有条目

★ 实例说明

本实例使用 Java 程序列出密钥库 mykeystore 中所有条目的别名。

★ 编程思路:

java.security 包中的 KeyStore 类提供的 aliases() 方法可以列出 KeyStore 类所代表的密钥库中的所有条目。

具体步骤如下:

- (1) 获取密钥库 mykeystorede 的 KeyStore 对象, 并加载密钥库

```
FileInputStream in=new FileInputStream("mykeystore");
KeyStore ks=KeyStore.getInstance("JKS");
ks.load(in, "wshr.ut".toCharArray());
```

分析: 该步骤和 5.2.7 小节第 1-3 步相同。其中 “wshr.ut” 是密钥库的密码。

- (2) 执行 KeyStore 对象的 aliases() 方法

```
Enumeration e=ks.aliases();
```

分析: 该步骤和 5.2.7 小节第 1-3 步相同。其中 “wshr.ut” 是密钥库的密码。该方法返回枚举类型的对象, 其中包含了密钥库中所有条目的别名。

- (3) 处理枚举对象

```
while( e.hasMoreElements() ) {
    System.out.println(e.nextElement());
}
```

分析: 枚举对象的 hasMoreElements() 方法可以判断其中是否还有元素, nextElement() 方法可以从枚举对象中取出元素。由于该枚举对象中包含的是密钥库中的各个条目的别名名称, 因此可以通过打印语句将名称打印出来。

★代码与分析:

完整代码如下:

```
import java.util.*;
import java.io.*;
import java.security.*;
public class ShowAlias{
    public static void main(String args[ ]) throws Exception{
```

```

        String pass="wshr.ut";
        String name="mykeystore";
        FileInputStream in=new FileInputStream(name);
        KeyStore ks=KeyStore.getInstance("JKS");
        ks.load(in,pass.toCharArray());
        Enumeration e=ks.aliases( );
        while( e.hasMoreElements()) {
            System.out.println(e.nextElement());
        }
    }
}

```

★运行程序

输入“java ShowAlias”运行程序，将显示所有条目的别名：

```

mytest
tmp
tmp1

```

5.3.4 Java 程序修改密钥库口令

★ 实例说明

本实例使用 Java 程序修改密钥库 mykeystore 的口令。

★ 编程思路：

Java 程序修改密钥库口令实际上是创建一个用新的密钥库，并设置新的口令。java.security 包中的 KeyStore 类提供了 store()方法，它可以将 KeyStore 类中的信息写入该方法参数中指定的新文件，并将口令修改为该方法参数中设置的新口令。

具体步骤如下：

- (1) 获取密钥库 mykeystorede 的 KeyStore 对象，并加载密钥库

```

String name="mykeystore";
char[ ] oldpass=args[0].toCharArray();
FileInputStream in=new FileInputStream(name);
KeyStore ks=KeyStore.getInstance("JKS");
ks.load(in,oldpass);

```

分析：该步骤和 5.3.3 小节相同。这里，我们通过第一个命令行参数读入密钥库的原有口令，以方便后面恢复口令。

- (2) 创建新密钥库输出流

```

FileOutputStream output=new FileOutputStream(name);

```

分析：新的密钥库可以是一个新的文件，也可以使用原有的“mykeystore”文件。这里我们使用原有的文件，这样程序运行起来更像是修改原有的口令。

- (3) 执行 KeyStore 类的 store()方法

```

ks.store(output,newpass);

```

分析：其中 ks 是第 1 步得到的 KeyStore 类型的对象，store() 方法 第一个参数是第 2 步指定的文件输出流，第二个参数是为密钥库设置的新口令。

★代码与分析：

完整代码如下：

```
import java.io.*;
import java.security.*;
public class SetStorePass{
    public static void main(String args[ ]) throws Exception{
        char[ ] oldpass=args[0].toCharArray();
        char[ ] newpass=args[1].toCharArray();
        String name="mykeystore";
        FileInputStream in=new FileInputStream(name);
        KeyStore ks=KeyStore.getInstance("JKS");
        ks.load(in,oldpass);
        in.close();
        FileOutputStream output=new FileOutputStream(name);
        ks.store(output,newpass);
        output.close();
    }
}
```

★运行程序

运行程序之前可先输入如下命令将 mykeystore 密钥库文件备份到文件 mykeystore.bat 中：

```
C:\java\ch5>copy mykeystore mykeystore.bak
1 file(s) copied
```

输入“java SetStorePass wshr.ut mynewpass”运行程序，将密钥库 mykeystore 的口令将由原来的 wshr.ut 修改为 mynewpass，这时再使用 keytool 查看该密钥库时，应该输入新口令：mynewpass，否则将无法访问密钥库。如：

```
C:\java\ch5>keytool -list -keystore mykeystore
```

输入 keystore 密码： wshr.ut

keytool 错误： java.io.IOException: Keystore was tampered with, or password was incorrect

```
C:\java\ch5>keytool -list -keystore mykeystore
```

输入 keystore 密码： mynewpass

Keystore 类型： jks

Keystore 提供者： SUN

您的 keystore 包含 3 输入

mytest, 2002-12-5, keyEntry,

认证指纹 (MD5): B2: DC: 75: CD: 60: B7: 1E: 7A: 97: EE: E8: A4: 31: D6: 26: C6

tmp, 2002-12-5, keyEntry,

```
认证指纹 (MD5): 5C: FA: ED: 8E: AE: 30: 1B: 2B: CF: 39: ED: 4D: 6F: 94: E1: 6B  
tmp1, 2002-12-5, keyEntry,  
认证指纹 (MD5): 51: F7: 46: AE: 20: 87: BF: 3F: 03: B9: 54: 56: AF: CC: 09: F5
```

从中可以看出密钥库内容仍旧保持不变。再输入“java SetStorePass mynewpass wshr.ut”运行程序，则 mykeystore 的口令恢复到 wshr.ut。可以使用 4.1.2 小节的 DigestInput 程序检验 mykeystore 在口令恢复后和最初的文件是否相同。如：

```
C:\java\ch5>java DigestInput mykeystore  
f48d2ec0d5da98bde2d390374d1bd3c2
```

```
C:\java\ch5>java DigestInput mykeystore.bak  
f48d2ec0d5da98bde2d390374d1bd3c2
```

可见，密钥库文件 mykeystore 和修改口令前备份过的 mykeystore.bak 文件的消息摘要相同，两个文件完全相同。

5.3.5 Java 程序修改密钥库条目的口令及添加条目

★ 实例说明

本实例使用 Java 程序修改密钥库 mykeystore 中指定条目的口令。同时演示了 Java 程序从密钥库中提取证书、私钥以及增加条目等操作。

★ 编程思路：

Java 程序修改密钥库指定条目的口令，实际上将密钥库中该条目别名对应的证书、私钥提取出来，重新写入密钥库。重新写入时使用相同的别名，口令则重新设置。

具体步骤如下：

(1) 读取相关参数

```
String name="mykeystore";  
String alias=args[0];  
char[] storepass="wshr.ut".toCharArray();  
char[] oldkeypass=args[1].toCharArray();  
char[] newkeypass=args[2].toCharArray();
```

分析：这里不妨通过第一个命令行参数读入别名的名称，第二个命令行参数读入该别名条目的原有口令，第三个命令行参数读入为该别名条目设置的新口令。其中 wshr.ut 为密钥库的口令。

(2) 获取密钥库 mykeystore 的 KeyStore 对象，并加载密钥库

```
FileInputStream in=new FileInputStream(name);  
KeyStore ks=KeyStore.getInstance("JKS");  
ks.load(in,storepass);
```

分析：该步骤和 5.3.2 小节类似。

(3) 获取别名对应的条目的证书链

```
Certificate[] cchain=ks.getCertificateChain(alias);
```

分析：执行 KeyStore 对象的 getCertificateChain () 方法，获取其参数对应的条目的证书链。

(4) 读取别名对应的条目的私钥

```
PrivateKey pk=(PrivateKey)ks.getKey(alias,oldkeypass);
```

分析: 执行 KeyStore 对象的 getKey() 方法, 获取其参数对应的条目的私钥, 保护私钥的口令也通过方法的参数传入。

(5) 向密钥库中添加条目

```
ks.setKeyEntry(alias,pk,newkeypass,cchain);
```

分析: 执行 KeyStore 对象的 setKeyEntry() 方法, 方法的第一个参数指定所添加条目的别名, 这里别名使用欲修改的条目的别名, 这样将覆盖欲修改的条目。如果使用新的别名, 则会增加一个条目。第二个参数为该条目的私钥, 第三个参数为设置的新口令, 第四个参数是对应于该私钥的公钥的证书链。

(6) 将 KeyStore 对象内容写入新文件

```
FileOutputStream output=new FileOutputStream("333");
```

```
ks.store(output,storepass);
```

分析: 执行 KeyStore 类的 store() 方法, 将修改后的 KeyStore 保存在新的文件中。这里不妨使用文件名 “333”。也可以像 5.3.4 小节一样使用原有的文件名 “mykeystore”

★代码与分析:

完整代码如下:

```
import java.io.*;
import java.security.*;
import java.security.cert.Certificate;
public class SetKeyPass{
    public static void main(String args[ ]) throws Exception{
        String name="mykeystore";
        String alias=args[0];
        char[ ] storepass="wshr.ut".toCharArray();
        char[ ] oldkeypass=args[1].toCharArray();
        char[ ] newkeypass=args[2].toCharArray();

        FileInputStream in=new FileInputStream(name);
        KeyStore ks=KeyStore.getInstance("JKS");
        ks.load(in,storepass);
        Certificate[ ] cchain=ks.getCertificateChain(alias);
        PrivateKey pk=(PrivateKey)ks.getKey(alias,oldkeypass);
        ks.setKeyEntry(alias,pk,newkeypass,cchain);
        in.close();
        FileOutputStream output=new FileOutputStream("333");
        ks.store(output,storepass);
        output.close();
    }
}
```

★运行程序

输入 “java SetKeyPass mytest wshr.ut newpass” 运行程序, 将把密钥库 mykeystore

的别名为 mytest 的条目的口令由 “wshr.ut” 改为 “newpass”，并重新保存到文件 333 中。然后我们可以使用 5.3.2 小节的方法将口令再改一次，从中可以看到 mytest 条目的口令（主密码）确实已经是程序中设置的 “newpass” 了：

```
C:\java\ch5>keytool -keypasswd -alias mytest -keystore 333
```

输入 keystore 密码: wshr.ut

输入<mytest>的主密码 newpass

新 <mytest> 的主密码: 123456

重新输入新 <mytest> 的主密码: 123456

5.3.6 Java 程序检验别名及删除条目

★ 实例说明

本实例使用 Java 程序检验某个别名是否在密钥库中，若在，则在密钥库中删除该条目。

★ 编程思路：

KeyStore 类提供的 containsAlias () 方法可以判断参数中指定的别名条目是否在密钥库中，deleteEntry () 方法可以删除方法参数中指定的别名条目，

具体步骤如下：

- (1) 获取密钥库 KeyStore 类型的对象

```
FileInputStream in=new FileInputStream(name);
```

```
KeyStore ks=KeyStore.getInstance("JKS");
```

```
ks.load(in,pass.toCharArray());
```

分析：该步骤和 5.3.3 小节相同。

- (2) 检验别名条目是否在密钥库中

```
ks.containsAlias(args[0])
```

分析：这里不妨从命令行参数读取别名字符串，将其作为参数传递给 KeyStore 对象的 containsAlias () 方法。若字符串指定的别名在 KeyStore 对象对应的密钥库中存在，则返回 true；否则返回 false。

- (3) 删除别名对应的条目

```
ks.deleteEntry(args[0])
```

分析：将从命令行参数读取的别名字符串作为参数传递给 KeyStore 对象的 deleteEntry () 方法。

- (4) 重新写入

```
ks.deleteEntry(args[0])
```

分析：将从命令行参数读取的别名字符串作为参数传递给 KeyStore 对象的 deleteEntry () 方法。

- (5) 将 KeyStore 对象内容写入新文件

```
FileOutputStream output=new FileOutputStream(name);
```

```
ks.store(output,pass.toCharArray());
```

分析：和 5.3.4 小节一样，执行 KeyStore 类的 store () 方法，将修改后的 KeyStore 重新保存在 mykeystore 文件中。也可以像 5.3.5 小节一样换个文件名保存新的密钥库。

★运行程序

我们先输入“java DeleteAlias Hi”运行程序，试图删除一个不存在的条目“hi”，然后输入“java DeleteAlias tmp1”运行程序，将 5.3.2 小节执行 5.3.1.bat 得到的临时条目 tmp1 删除，最后使用 keytool 查看删除的效果。操作过程及效果如下：

```
C:\java\ch5>java DeleteAlias Hi
```

```
Alias not exist
```

```
C:\java\ch5>java DeleteAlias tmp1
```

```
Alias tmp1 deleted
```

```
C:\java\ch5>keytool -list -keystore mykeystore
```

```
输入 keystore 密码:  wshr.ut
```

```
Keystore 类型:  jks
```

```
Keystore 提供者:  SUN
```

```
您的 keystore 包含 2 输入
```

```
mytest, 2002-12-5, keyEntry,
```

```
认证指纹 (MD5):  B2: DC: 75: CD: 60: B7: 1E: 7A: 97: EE: E8: A4: 31: D6: 26: C6
```

```
tmp, 2002-12-5, keyEntry,
```

```
认证指纹 (MD5):  5C: FA: ED: 8E: AE: 30: 1B: 2B: CF: 39: ED: 4D: 6F: 94: E1: 6B
```

5.4 数字证书的签发

从 5.1 节可以看出，任何人都可以很方便地创建数字证书，宣称某个公钥是某个人或机构所拥有的。这样，当用户收到某个证书后，如何确定这个证书宣称的内容到底是真的还是假的呢？

和现实生活中一样，要有权威的机构检查证书中内容的真实性，然后再签发证书（在证书上盖章）。在计算机的世界中，这个盖章的过程就是数字签名，即权威机构用自己的私钥对证书进行数字签名。

这种权威机构已经有很多，如 Verisign, Thawte 等，这些机构称为 CA（Certification Authorities），这些 CA 的公钥已经以证书的形式包含在许多操作系统中。

CA 检查别人的数字证书，确定可靠后使用自己的私钥为证书签名。用户收到这样的证书后，用相应 CA 的公钥进行检验，若检验通过，说明证书是可靠的。这是因为根据数字签名的原理，其他人伪造的签名这样的签名将无法通过验证。本书附录中介绍了如何将 5.1 节创建的证书交给一个著名的 CA（Verisign）签发，以增强其可信性。本章使用 Java 程序实现 CA，对其他证书进行签名，这样便于理解签发证书的实质。

5.4.1 确定 CA 的权威性——安装 CA 的证书

本章假定你自己是 CA，全名是“CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai, C=CN”，简称“Xu Yingxiao”。你在计算机中的身份以 5.2.3

小节得到的证书 mytest.cer 来代表，它是在第 5.1.3 小节创建的，随证书一起创建的还有和证书中的公钥相对应的私钥，全部保存在密钥库 mykeystore 中，别名为 mytest。

该证书的文本如图 5-2 所示。可以通过电子邮件、网页或盖有公章的正式文件确定如图 5-2 所示的文件 mytest.cer 是权威的，在部门或单位内部是值得信任的，并可将其认证指纹（即消息摘要，拇指印）公布。在 5.2.4 小节中我们已经看到其认证指纹为：MD5：B2:DC:75:CD:60:B7:1E:7A:97:EE:E8:A4:31:D6:26:C6。

用户得到 CA 自身的证书后，可以将证书安装在计算机操作系统中，以便计算机自动检验其他证书是否值得信任。

★ 实例说明

本实例将代表 CA “Xu Yingxiao” 的证书文件 mytest.cer 安装在用户的机器中，以便在计算机中确立 mytest.cer 证书的权威性。

★ 运行程序

双击 5.2.3 小节导出的证书文件 mytest.cer，出现图 5-5 所示的证书窗口：

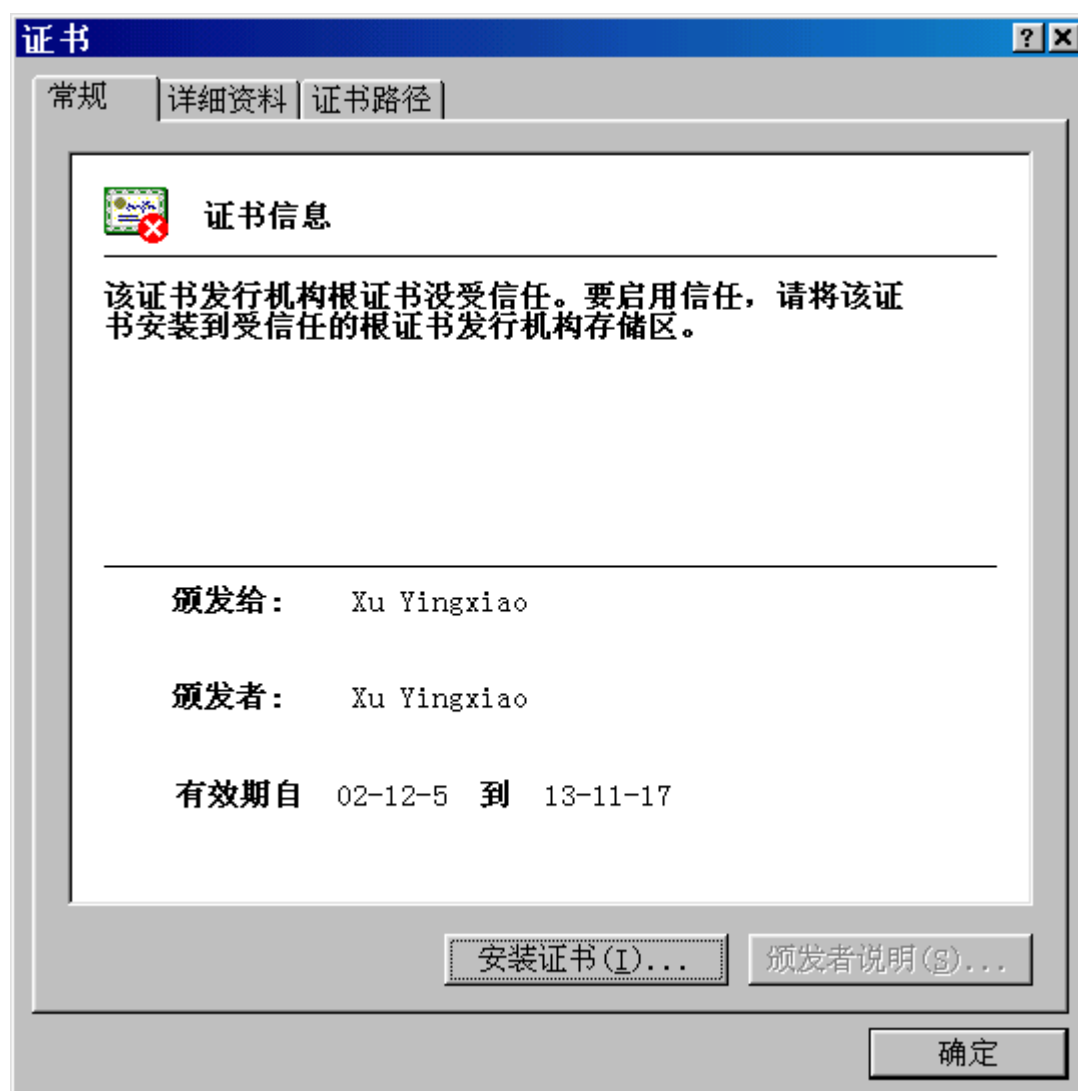


图 5-5 CA 的证书

点击窗口中的“安装证书”按钮，出现图 5-6 所示的证书管理器导入向导窗口。

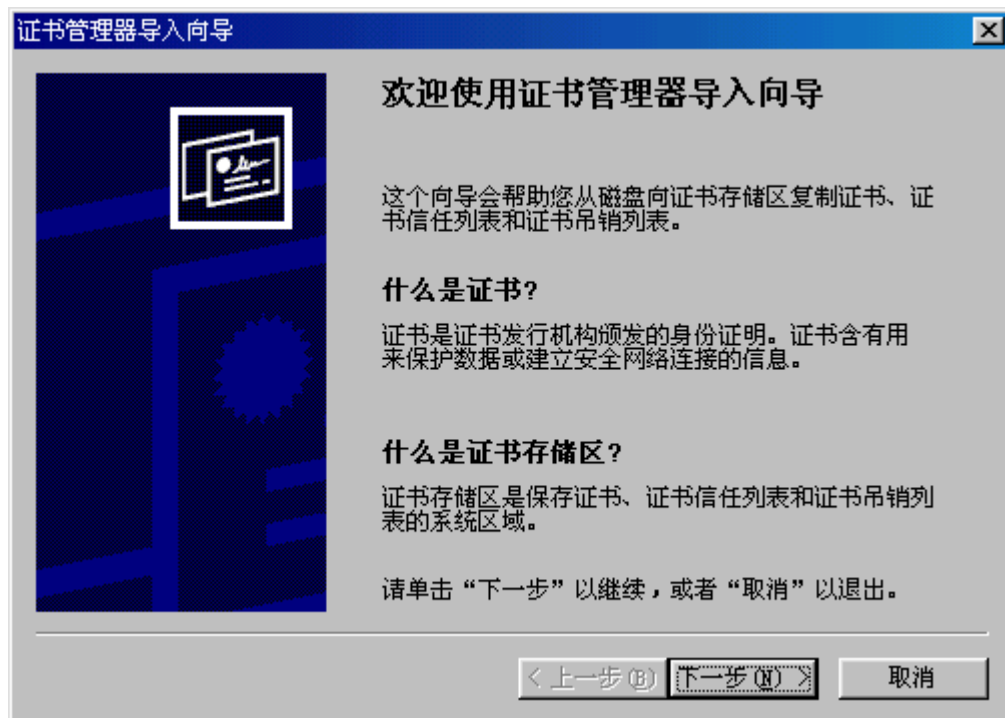


图 5-6 证书管理器导入向导窗口

点击其中的“下一步”按钮，出现图 5-7 所示的选定证书存储区的窗口。

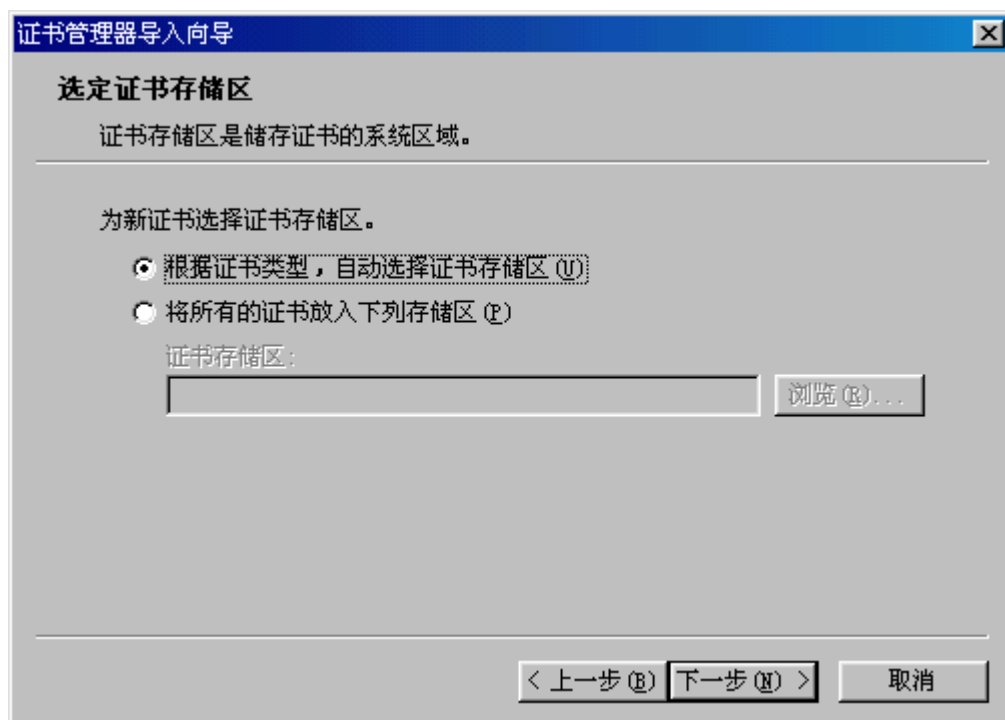


图 5-7 选定证书存储区的窗口。

不妨使用该窗口的默认选择：“根据证书类型，自动选择证书存储区”，继续点击其中的“下一步”按钮，出现图 5-8 所示的完成证书管理器导入向导的提示。

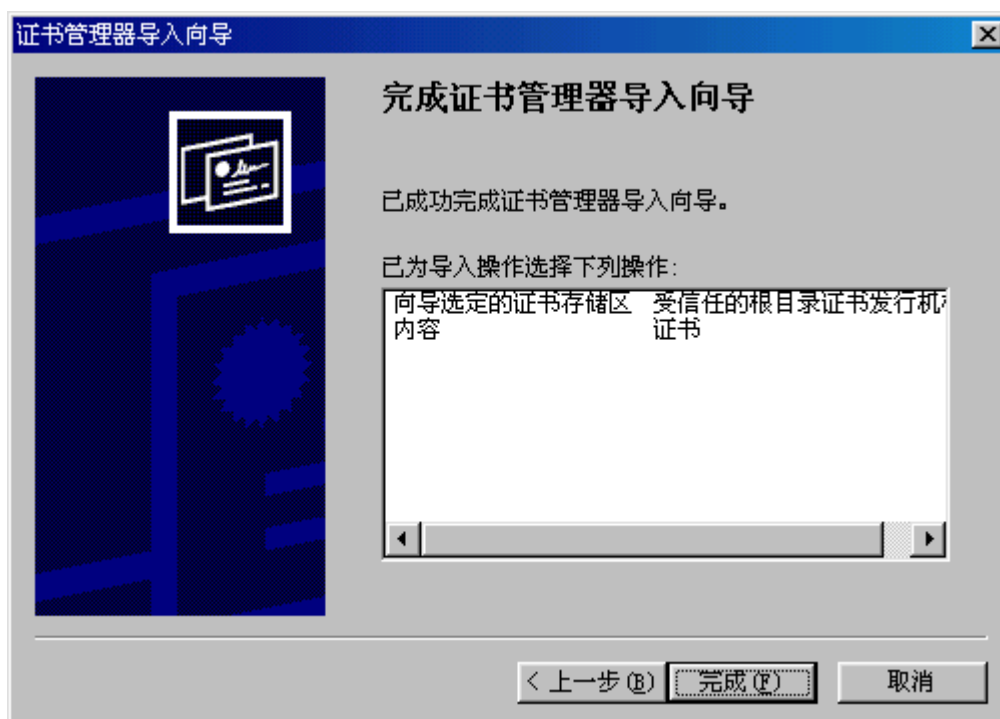


图 5-8 完成证书管理器导入向导的提示

该窗口显示证书将被存储到“受信任的根目录证书发行机构”区域，这样，以后操作系统将自动信任由 mytest.cer 证书签发的其他证书。点击其中的“完成”按钮，出现最后的确认窗口，如图 5-9 所示。



图 5-9 最后核实证书的窗口

如果用户怀疑该证书是否正确，可以再核实一遍该窗口中显示的拇印，确认后点击其中的“是”按钮，最后提示导入成功。

以上操作主要是针对我们自己创建的 CA 的证书：mytest.cer，如果使用著名的 CA 如 Verisign，则本小节的操作就不需要了，因为 Verisign 等著名的 CA 的证书已经存储在常用的操作系统中了。

此外，本小节的操作有一定的风险性，必须妥善保存 mytest.cer 对应的私钥。因此这里只是用于编程实验，实验结束最好使用 5.5.3 小节的方法撤销本小节的操作。

5.4.2 验证 CA 的权威性——显示 CA 的证书

实例说明

本实例验证 5.4.1 将代表 CA “Xu Yingxiao” 的证书 mytest.cer 安装在用户的机器中，以便在计算机中确立 mytest.cer 证书的权威性。

★运行程序

双击 5.2.3 小节导出的证书 mytest.cer，出现图 5-10 所示的证书窗口。



图 5-10 受信任的证书窗口

和图 5-5 相比，证书窗口中原先显示的“该证书发行机构根证书没受信任”警告已经消失了，取而代之的是该证书的用途。点击该窗口中的“证书路径”标签，出现图 5-11 的证书路径窗口，其中显示了该证书是正确的。

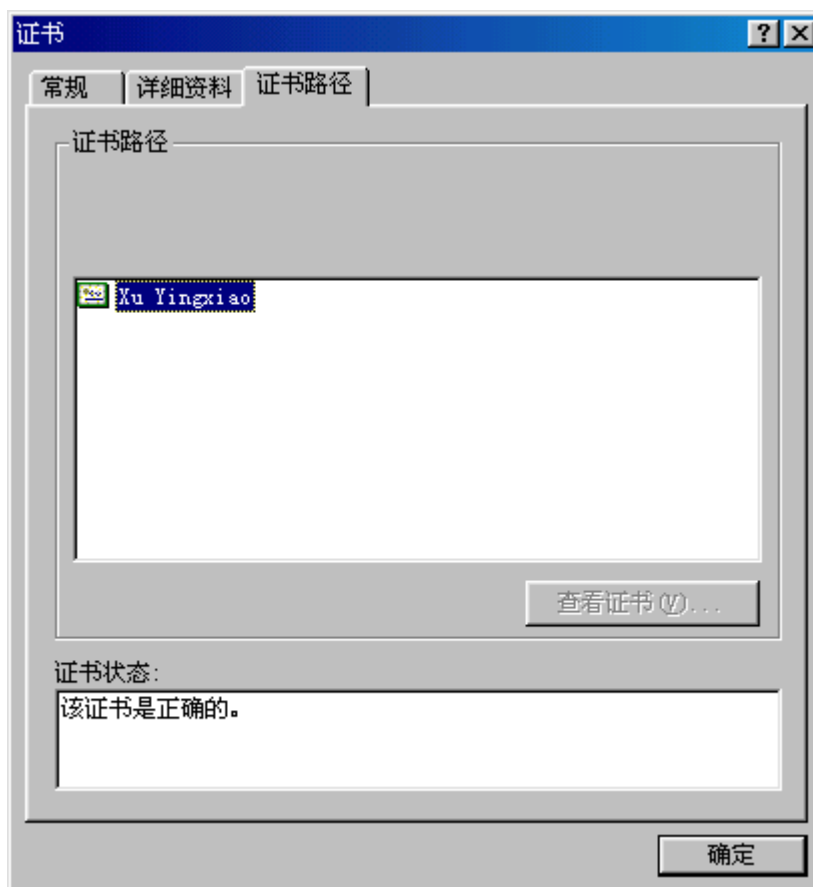


图 5-11 证书路径窗口

5.4.3 Java 程序签发数字证书

实例说明

本实例使用 5.4.1 小节确定的 CA “Xu Yingxiao” 对 5.1.3 小节创建的证书 “Liu Fang”（证书文件为 5.2.3 小节的 lf.cer 文件）进行签发，该实例使我们对 Verisign 等 CA 是如何签发证书的有一个实际的了解。

★ 编程思路：

CA 签发数字证书应该使用自己的私钥，CA 自身的证书中并不包含私钥信息，因此需从密钥库 mykeystore 中提取。此外，由于被签发的证书还需要知道 CA 的名字，这可以从 CA 的证书中获得。

签发证书实际上是创建了一个新的证书，本实例使用 J2SDK 内部使用的 sun.security.x509 包中的 X509CertImpl 类来创建新的证书，该类的构造器中传入有关新的证书各种信息，主要信息来自被签发的 lf.cer，只是对某些必须修改的信息如序列号、有效期、签发者等进行重新设置。最后使用 X509CertImpl 类的 sign() 方法用 CA 的私钥进行签名。可以打印新的证书的信息，也可以将其保存在密钥库中。

具体步骤如下：

- (1) 从密钥库读取 CA 的证书

```
FileInputStream in=new FileInputStream(name);  
KeyStore ks=KeyStore.getInstance("JKS");  
ks.load(in,storepass);
```

```
java.security.cert.Certificate c1=ks.getCertificate(alias);
```

分析：这里 name 的值为 “mykeystore”，alias 的值为 “mytest”。和 5.2.7 小节第 1-4 步相同。

(2) 从密钥库读取 CA 的私钥

```
PrivateKey caprk=(PrivateKey)ks.getKey(alias,cakeypass);
```

分析：该步骤和 5.3.5 小节第 4 步类似，执行 KeyStore 对象的 getKey() 方法，获取其参数对应的条目的私钥，保护私钥的口令也通过方法的参数传入。该口令必须和创建证书时所输入的“主密码”相同。所获得的私钥用于后面的签名。

(3) 从 CA 的证书中提取签发者信息

```
byte[] encod1=c1.getEncoded();
X509CertImpl cimp1=new X509CertImpl(encod1);
X509CertInfo cinfo1=(X509CertInfo)cimp1.get(X509CertImpl.NAME+
    "."+X509CertImpl.INFO);
X500Name issuer=(X500Name)cinfo1.get(X509CertInfo.SUBJECT+
    "."+CertificateIssuerName.DN_NAME);
```

分析：首先提取 CA 的证书的编码，然后用该编码创建 X509CertImpl 类型的对象，通过该对象的 get() 方法获取 X509CertInfo 类型的对象，该对象封装了证书的全部内容，最后通过该对象的 get() 方法获得 X500Name 类型的签发者信息。这些类在 J2SDK1.4 的 API 文档中并无介绍，但可以直接使用。

(4) 获取待签发的证书

```
CertificateFactory cf=CertificateFactory.getInstance("X.509");
FileInputStream in2=new FileInputStream(args[0]);
java.security.cert.Certificate c2=cf.generateCertificate(in2);
```

分析：待签发的证书可以像 5.2.7 小节那样从密钥库中读取，也可以像 5.2.6 小节那样从导出的证书文件读取。这里不妨采用后面一种方法，其编程步骤和 5.2.6 小节相同。证书文件的名称不妨从命令行参数读入，该证书可以使用在 5.1.4 小节中创建、在 5.2.3 小节导出的文件 lf.cer。

(5) 从待签发的证书提取证书信息

```
byte[] encod2=c2.getEncoded();
X509CertImpl cimp2=new X509CertImpl(encod2);
X509CertInfo cinfo2=(X509CertInfo)cimp2.get(
    X509CertImpl.NAME+"."+X509CertImpl.INFO);
```

分析：新证书的主要信息来自待签发的证书，待签发的证书中这些信息主要封装在 X509CertInfo 对象中，所以和第 3 步类似，先提取待签发者的证书编码，然后创建 X509CertImpl 类型的对象，最后通过该对象的 get() 方法获取 X509CertInfo 类型的对象。

以后就可以使用该对象创建新的证书了，再创建新证书之前，还需要使用其 set() 方法对其中部分信息作一些必要的修改。

(6) 设置新证书有效期

```
Date begindate =new Date();
```

```
//30000 day
Date enddate =new Date(begindate.getTime()+3000*24*60*60*1000L);
CertificateValidity cv=new CertificateValidity(begindate,enddate);
cinfo2.set(X509CertInfo.VALIDITY,cv);
```

分析：新证书的开始生效时间不妨从签发之时开始，因此首先使用 new Date() 获取当前时间。新证书截止日期不能超过 CA，作为测试，这里不妨设置截止日期为 3000 天以后，因此使用 new Date() 再创建一个日期对象，其参数传入长整型的值，即在原先日期的基础上增加 3000 天的时间（毫秒数）。

最后通过这两个日期创建 CertificateValidity 类型的对象，并把它作为参数传递给上一步得到的 X509CertInfo 对象的 set() 方法以设置有效期。

(7) 设置新证书序列号

```
int sn=(int)(begindate.getTime()/1000);
CertificateSerialNumber csn=new CertificateSerialNumber(sn);
cinfo2.set(X509CertInfo.SERIAL_NUMBER,csn);
```

分析：每个证书有一个唯一的序列号，这里不妨以当前的时间（以秒为单位）为序列号，创建 CertificateSerialNumber 对象，并作为参数传递给 X509CertInfo 对象的 set() 方法以设置序列号。

(8) 设置新证书签发者

```
cinfo2.set(X509CertInfo.ISSUER+"."+CertificateIssuerName.DN_NAME,issuer);
```

分析：执行 X509CertInfo 对象的 set() 方法设置签发者，传入的参数即第 3 步得到的签发者信息。

(9) 设置新证书签名算法信息

```
AlgorithmId algorithm =
    new AlgorithmId(AlgorithmId.md5WithRSAEncryption_oid);
info2.set(CertificateAlgorithmId.NAME+"."
    +CertificateAlgorithmId.ALGORITHM, algorithm);
```

分析：首先生成 AlgorithmId 类型的对象，在其构造器中指定 CA 签名该证书所使用的的算法为 md5WithRSA，然后将其作为参数传递给 X509CertInfo 对象的 set() 方法以设置签名算法信息。

(10) 创建证书并使用 CA 的私钥对其签名

```
X509CertImpl newcert=new X509CertImpl(cinfo2);
newcert.sign(caprk,"MD5WithRSA");
```

分析：X509CertImpl 类是 X509 证书的底层实现，将第 5 步得到的待签发的证书信息（部分信息已经在 7~9 步作了修改）传递给它构造器，将得到新的证书，执行其 sign() 方法，将使用 CA 的私钥对证书进行数字签名，第一个参数即第 1 步获得的 CA 的私钥，第二个参数即签名所用的算法。这样，就得到了经过 CA 签名的证书。

(11) 将新证书存入密钥库

```
ks.setCertificateEntry("If_signed", newcert);
FileOutputStream out=new FileOutputStream("newstore");
ks.store(out,"newpass".toCharArray());
```

分析：和 5.3.5 小节第 6 步类似，使用 KeyStore 对象的 store() 方法将 KeyStore 对象中的内容写入密钥库文件。store() 方法的第一个参数指定密钥库文件的文件输出流，这里不妨以“newstore”作为新的密钥库文件名，也可以使用原有的密钥库“mykeystore”覆盖原有的密钥库。第二个参数为密钥库文件设置保护口令，这里不妨以“newpass”作为口令。

在执行 KeyStore 对象的 store() 方法之前，需要将上一步得到的签过名的新证书写入 KeyStore 对象，这里使用了 KeyStore 对象的 setCertificateEntry() 方法，其第一个参数设置了新证书在密钥库中的别名，第二个参数传入上一步得到的证书。也可像 5.3.5 小节第 6 步类似的方法，使用 KeyStore 对象的 setKeyEntry() 方法，这时应该将被签名证书 lf.cer 对应的私钥先从 mykeystore 中提取出来，然后传递给 setKeyEntry() 方法。并用新证书构造数组传递给 setKeyEntry() 方法。如：

```
PrivateKey prk=(PrivateKey)ks.getKey("lf","wshr.ut".toCharArray());
java.security.cert.Certificate[] cchain={newcert};
ks.setKeyEntry("signed-1f",prk,"newpass".toCharArray(),cchain);
```

★代码与分析：

完整代码如下：

```
import java.io.*;
import java.security.*;
import java.security.cert.*;
import java.util.*;
import java.math.*;
import sun.security.x509.*;

public class SignCert{

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

        char[] storepass="wshr.ut".toCharArray( );
        char[] cakeypass="wshr.ut".toCharArray( );
        String alias="mytest";
        String name="mykeystore";
        // Cert of CA-----c1
        FileInputStream in=new FileInputStream(name);
        KeyStore ks=KeyStore.getInstance("JKS");
        ks.load(in,storepass);
        java.security.cert.Certificate c1=ks.getCertificate(alias);
        PrivateKey caprk=(PrivateKey)ks.getKey(alias,cakeypass);
        in.close();
        //得到签发者
        byte[] encod1=c1.getEncoded();
        X509CertImpl cimpl=new X509CertImpl(encod1);
        X509CertInfo cinfo1=(X509CertInfo)cimpl.get(X509CertImpl.NAME+
            ". "+X509CertImpl.INFO);
        X500Name issuer=(X500Name)cinfo1.get(X509CertInfo.SUBJECT+
            ". "+CertificateIssuerName.DN_NAME);
        // Cert of lf-----c2
```



```

CertificateFactory cf=CertificateFactory.getInstance("X.509");
FileInputStream in2=new FileInputStream(args[0]);
java.security.cert.Certificate c2=cf.generateCertificate(in2);
in2.close();
byte[] encod2=c2.getEncoded();
X509CertImpl cimp2=new X509CertImpl(encod2);
X509CertInfo cinfo2=(X509CertInfo)cimp2.get(
X509CertImpl.NAME+"."+X509CertImpl.INFO);
//设置新证书有效期
Date begindate =new Date();
//60 day
Date enddate =new Date(begindate.getTime()+3000*24*60*60*1000L);
CertificateValidity cv=new CertificateValidity(begindate,enddate);
cinfo2.set(X509CertInfo.VALIDITY,cv);
//设置新证书序列号
int sn=(int)(begindate.getTime()/1000);
CertificateSerialNumber csnew=new CertificateSerialNumber(sn);
cinfo2.set(X509CertInfo.SERIAL_NUMBER,csnew);
//设置新证书签发者
cinfo2.set(X509CertInfo.ISSUER+"."+
CertificateIssuerName.DN_NAME,issuer);
//设置新证书算法
AlgorithmId algorithm =
new AlgorithmId(AlgorithmId.md5WithRSAEncryption_oid);
cinfo2.set(CertificateAlgorithmId.NAME+
"."+CertificateAlgorithmId.ALGORITHM, algorithm);
// 创建证书
X509CertImpl newcert=new X509CertImpl(cinfo2);
// 签名
newcert.sign(caprk,"MD5WithRSA");
System.out.println(newcert);
// 存入密钥库
ks.setCertificateEntry("lf_signed", newcert) ;
/*
    PrivateKey prk=(PrivateKey)ks.getKey("lf",
        "wshr.ut".toCharArray( ));
    java.security.cert.Certificate[] cchain={newcert};
    ks.setKeyEntry("lf_signed",prk,
        "newpass".toCharArray(),cchain);
*/
FileOutputStream out=new FileOutputStream("newstore");
ks.store(out,"newpass".toCharArray());
out.close();
}

```

```
}
```

程序中添加了打印语句将新创建的证书的相关信息在屏幕上打印出来。

★运行程序

在当前目录下存放 5.1.3 小节创建密钥库 `mykeystore`，其中包含了 CA 的证书。当前目录下同时有 5.2.3 小节导出的证书 `lf.cer`，签发之前假定我们已经核实过 `lf.cer` 中包含的信息，确认无误，接下来我们开始用 CA 对该证书进行签名。

输入“`java SignCert lf.cer >1.txt`”运行程序，则程序将从密钥库中取出 CA 的私钥对 `lf.cer` 证书进行签名，输出结果已重定向到文件 `1.txt` 中，打开 `1.txt` 文件，可以看到如下有关新的证书的信息。

```
[
[
  Version: V1
  Subject: CN=Liu Fang, OU=Packaging, O=Shanghai University, L=ZB, ST=Shanghai,
C=CN
  Signature Algorithm: MD5withRSA, OID = 1.2.840.113549.1.1.4

  Key:   com.sun.net.ssl.internal.ssl.JSA-RSAPublicKey@ac2f9c
  Validity: [From: Thu Dec 05 12:04:35 CST 2002,
             To: Mon Feb 21 12:04:35 CST 2011]
  Issuer: CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB,
ST=Shanghai, C=CN
  SerialNumber: [    3deed053 ]

]
  Algorithm: [MD5withRSA]
  Signature:
0000: D2 3F 52 38 62 BF ED 59    D0 E5 B1 83 E3 4C 56 C9    .?R8b..Y....LV.
0010: 9C 8F C8 37 13 35 31 2F    36 F7 A0 9E CD 04 2C 58    ...7.51/6.....,X
0020: 72 DE 0C B6 46 F9 AF CD    96 E3 2D CF 70 9E 1A E5    r...F.....-.p...
0030: 9A B3 D9 12 97 EA 7C 97    4A F9 E6 8B 93 52 C4 42    .....J....R.B
0040: 13 6F EC 43 FD 30 ED B2    19 92 13 FD 0B DA A6 8C    .o.C.0.....
0050: 9B 3F 08 62 A9 9F 4B 23    CD A8 A0 CB BE 60 09 85    .?.b..K#.....\..
0060: E4 EC 3C 5E D7 CE BC 44    E7 F5 43 0B 01 EA 93 A3    ..<^...D..C.....
0070: CB EA 83 B3 BF 2F B4 2E    83 12 54 A4 55 AE E2 5C    ...../....T.U..\

]
```

从中可以看出签发者已经由原先的“CN=Liu Fang, OU=Packaging, O=Shanghai University, L=ZB, ST=Shanghai, C=CN”（自己给自己签名），变为了：“CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai, C=CN”（CA 的签名）。有效期也设置成了 3000 天。

新的证书同时由密钥保存在了密钥库 `newstore` 中，输入如下命令可以查看密钥库中的

新证书。

```
C:\java\ch5\Sign>keytool -list -keystore newstore
```

```
输入 keystore 密码:  newpass
```

```
输入 keystore 密码:  newpass
```

```
Keystore 类型:  jks
```

```
Keystore 提供者:  SUN
```

```
您的 keystore 包含 3 输入
```

```
mytest, 2002-12-5, keyEntry,
```

```
认证指纹 (MD5):  B2: DC: 75: CD: 60: B7: 1E: 7A: 97: EE: E8: A4: 31: D6: 26: C6
```

```
lf_signed, 2002-12-5, trustedCertEntry,
```

```
认证指纹 (MD5):  D3: 7E: C0: 72: 5D: 41: 46: CA: 7A: 8E: 85: 21: 1B: DA: 89: 0F
```

```
tmp, 2002-12-5, keyEntry,
```

```
认证指纹 (MD5):  5C: FA: ED: 8E: AE: 30: 1B: 2B: CF: 39: ED: 4D: 6F: 94: E1: 6B
```

其中 lf_signed 即程序中签发 lf.cer 生成的新证书，注意其类型为 “trustedCertEntry” 而不是 “keyEntry”。这是因为程序中使用的是 KeyStore 对象的 setCertificateEntry () 方法保存证书，它只将证书导入了密钥库，而没有导入对应的私钥。如果采用本小节第 11 步的分析中的方法，使用 KeyStore 对象的 setKeyEntry () 方法，则这里显示的 lf_signed 就和其他两个条目一样是 “keyEntry” 了。

5.4.4 数字证书签名后的发布

★ 实例说明

5.4.3 小节将签名后的数字证书保存在密钥库中，本实例介绍 CA 对某个人或机构的证书进行签名后，如何将签名后的证书提交给对方。

★ 运行程序

CA 对数字证书签名后可以将其导出到文件。类似 5.2.3 小节，在命令行中一行输入如下命令：

```
keytool -export -alias lf_signed -keystore newstore -storepass newpass -rfc -file lf_signed.cer
```

这样 5.4.3 小节的签名后的数字证书将被导出到文件 lf_signed.cer，可以将该文件 E-mail 给对方，也可以用 Windows 的记事本打开该证书文件，将其编码内容通过 E-mail 或 Web 等方式发布。其他人只要将其文本粘贴下来，保存到文件名以 “.cer” 为后缀的文本文件中即可。该证书内容如下所示：

```

-----BEGIN CERTIFICATE-----
MIICXjCCAccCBD3u0FMwDQYJKoZIhvcNAQEEBQAweljELMAkGA1UEBhMCQ04xETAPBgNUBAGTCFNo
YW5naGFpMQswCQYDUQQHwJaQjEcMBoGA1UEChMTU2hhbmdoYWkgUW5pdMUyc210eTEsMBAGA1UEC
xM0TmU0d29yayBDZW50ZXIxFDASBgNUBAMTC1h1IF1pbmd4aWwFvMB4XDAtYMTIwNTA0MDQzNUoX
DTEyMDIyMTA0MDQzNUowcjELMAkGA1UEBhMCQ04xETAPBgNUBAGTCFNoYW5naGFpMQswCQYDUQQH
EwJaQjEcMBoGA1UEChMTU2hhbmdoYWkgUW5pdMUyc210eTEsMBAGA1UECjQAwgYkCgYEAzFnaW5nMREw
DwYDUQDEwhMaXUgRmFuZzCBnzANBgkqhkiG9w0BAQEFAA0BjQAwgYkCgYEAzFnaW5nMREwDwYDUQ
NUZpoUgCS0bj1xGYtIKDXG6MaGLJstcfMxBipdrvo15r+0wgz8T290eTL0AS97mFz4zNr3jgYFSU
JFpUdQyMarMMwKODN+8c2X9NQ8UasHCvUwR14pHTy5T0W2Fzrb9/78FY+/wcDGftSKAUH1d9L9M
SI8CAwEAATANBgkqhkiG9w0BAQQFAA0BgQDSP1I4Yr/tWdD1sYPjTFbJnI/INxM1MS8296CezQQs
WHLedLZG+a/NluMtz3CeGuWas9kS1+p810r55ouTUsRCE2/sQ/0w7bIZkhP9C9qmjJs/CGKpn0sj
zaigy75gCYXk7Dxe1868R0f1QwsB6p0jy+qDs78vtC6DE1SkUa7iXA==
-----END CERTIFICATE-----

```

在 Windows 中双击该文件，将出现图 5-12 所示的证书窗口。

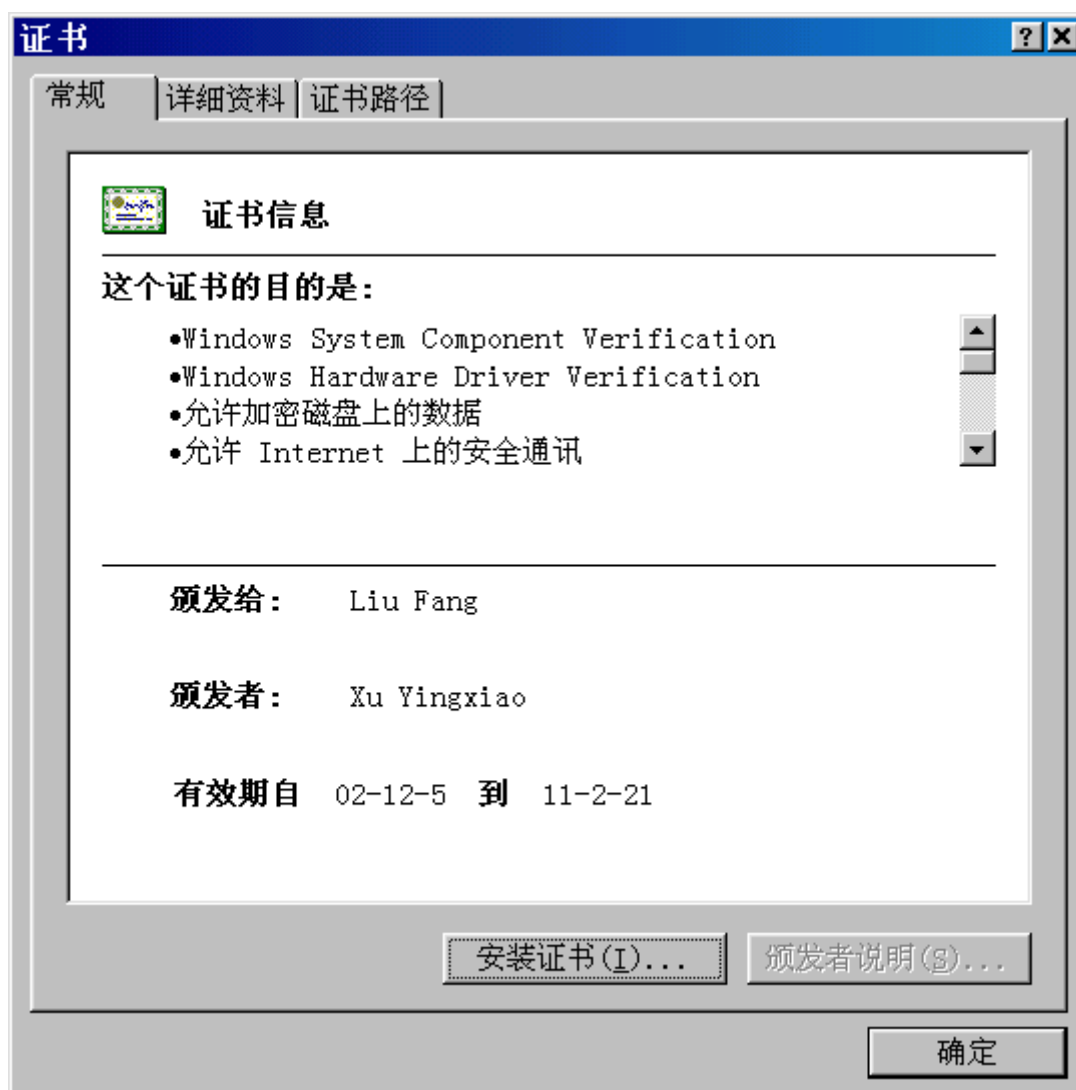


图 5-12 签名后的证书窗口

和图 5-3 相比，它不再有“该证书发行机构根证书没受信任”的警告，这是因为在 5.4.1 小节中我们已经在机器中安装了其发行机构的证书：mytest.cer。此外，图 5-12 中还可以看出其显示的颁发者已经是 Xu Yingxiao，而不是图 5-3 所示的自己给自己签名（自签名证书）。

实际上，本书附录中介绍的将证书交给 Verisign 等 CA 签发后得到的也是类似这里得到

的 If_signed 文件的证书。

得到该证书后,可以如 5.5.1 和 5.5.2 小节那样验证和显示签名后的证书,也可以如 5.5.3 小节那样将签名后的证书导入密钥库。

5.5 数字证书的检验

第 5.1 节我们创建了自签名的数字证书,第 5.4 节创建了通过我们自己的 CA 签名的数字证书,在附录中通过 Verisign 等 CA 签名的证书,本节介绍如何验证这些证书是否有效。

5.5.1 Java 程序验证数字证书的有效期

★ 实例说明

本实例使用 5.2.3 小节得到的证书文件 mytest.cer 和 lf.cer 以及 5.4.4 小节得到的证书文件 If_signed,演示了如何检验证书在某个日期是否有效。

★ 编程思路:

使用 5.2.6 或 5.2.7 小节得到的 X509Certificate 类型的对象可以方便地检验证书在某个日期是否有效,只要执行其 checkValidity()方法,方法的参数中传入日期。若已经过期,则程序会生成 CertificateExpiredException 异常,若尚未开始生效,则生成 CertificateNotYetValid 异常。

具体步骤如下:

- (1) 获取 X509Certificate 类型的对象

```
CertificateFactory cf=CertificateFactory.getInstance("X.509");
FileInputStream in=new FileInputStream(args[0]);
java.security.cert.Certificate c=cf.generateCertificate(in);
X509Certificate t=(X509Certificate) c;
```

分析: 和 5.2.6 小节一样,从命令行参数读入需要验证的证书文件。也可以像 5.2.7 小节那样从密钥库中直接读取证书。

- (2) 获取日期

```
Calendar cld=Calendar.getInstance();
int year=Integer.parseInt(args[1]);
int month=Integer.parseInt(args[2])-1;
int day=Integer.parseInt(args[3]);
cld.set(year,month,day);
Date d=cld.getTime();
```

分析: 我们的目的是验证证书在某个日期是否有效,因此不妨从命令行读入年月日,由此生成 Date()对象。由于 Date 类的很多设置年月日的方法已经不提倡使用,因此改用 Calendar 类,Calendar 类也是一个工厂类,通过 getInstance()方法获得对象,然后使用 set()方法设置时间,最后通过其 getTime()方法获得 Date()对象,

由于 Calendar 类的 set()方法参数是整数,因此对命令行参数读入的年月日字符串使用 Integer.parseInt()方法转换为整型数。由于 Calendar 类的 set()方法设置月份时从 0 开始,0 代表 1 月,11 代表 12 月,因此命令行读入的月份要减去 1。

- (3) 检验证书

```
t.checkValidity(d);
```

分析: 执行第 1 步得到的 X509Certificate 对象的 checkValidity() 方法, 方法参数传入第 2 步得到的 Date 对象。

(4) 处理 CertificateExpiredException 异常

```
catch(CertificateExpiredException e){
    System.out.println("Expired");
    System.out.println(e.getMessage());
}
```

分析: 第 3 步若生成 CertificateExpiredException 异常, 表明证书在指定的日期已经过期, 可以在 catch 语句中作相关处理。这里简单地打印一句 “Expired”, 并显示相关的异常信息。

(5) 处理 CertificateNotYetValidException 异常

```
catch(CertificateNotYetValidException e){
    System.out.println("Too early");
    System.out.println(e.getMessage());
}
```

分析: 第 3 步若生成 CertificateNotYetValidException 异常, 表明证书在指定的日期尚未开始生效, 可以在 catch 语句中作相关处理。这里简单地打印一句 “Too early”, 并显示相关的异常信息。

★ 代码与分析

完整代码如下:

```
import java.io.*;
import java.security.*;
import java.security.cert.*;
import java.util.*;
public class CheckCertValid{
    public static void main(String args[ ]) throws Exception{
        // X509Certificate 对象
        CertificateFactory cf=CertificateFactory.getInstance("X.509");
        FileInputStream in=new FileInputStream(args[0]);
        java.security.cert.Certificate c=cf.generateCertificate(in);
        in.close();
        X509Certificate t=(X509Certificate) c;
        //读取日期
        Calendar cld=Calendar.getInstance();
        int year=Integer.parseInt(args[1]);
        int month=Integer.parseInt(args[2])-1; // as 0 is Jan, 11
        int day=Integer.parseInt(args[3]);
        cld.set(year,month,day);
        Date d=cld.getTime();
        System.out.println(d);
        //检验有效期
        try{
            t.checkValidity(d);
```

```

        System.out.println("OK");
    } catch (CertificateExpiredException e) {    //过期
        System.out.println("Expired");
        System.out.println(e.getMessage());
    }
    catch (CertificateNotYetValidException e) {    //尚未生效
        System.out.println("Too early");
        System.out.println(e.getMessage());
    }
}
}
}

```

★运行程序

在当前目录下保存要检验的证书,如 5.2.3 小节得到的证书文件 mytest.cer 和 lf.cer, 5.4.4 小节得到的证书文件 lf_signed mytest.cer 等。

如 mytest.cer 有效期是 2002 年 12 月 5 日至 2013 年 11 月 17 日,则可输入如下几个命令测试程序。输入“java CheckCertValid mytest.cer 2002 12 4”运行程序检测证书 mytest.cer 在 2002 年 12 月 4 日是否有效,屏幕输出如下:

```

Wed Dec 04 13:26:41 CST 2002
Too early
NotBefore: Thu Dec 05 10:56:03 CST 2002

```

输入“java CheckCertValid mytest.cer 2003 6 16”运行程序检测证书 mytest.cer 在 2003 年 6 月 16 日是否有效,屏幕输出如下:

```

Mon Jun 16 13:27:42 CST 2003
OK

```

输入“java CheckCertValid mytest.cer 2013 11 18”运行程序检测证书 mytest.cer 在 2013 年 11 月 19 日是否有效,屏幕输出如下:

```

Mon Nov 18 13:28:16 CST 2013
Expired
NotAfter: Sun Nov 17 10:56:03 CST 2013

```

5.5.2 使用 Windows 查看证书路径验证证书的签名

★ 实例说明

本实例使用 5.4.4 小节得到的经过 CA 签名的证书文件 lf_signed, 演示了在 Windows 中查看该证书是否值得信任。

★运行程序

直接在 Windows 中用鼠标双击证书文件 lf_signed, 出现 5.4.4 小节图 5-12 所示的证书窗口, 由于在 5.4.1 小节中已经将签发者“CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai, C=CN”的证书安装在 Windows 中了。因此图 5-12 检验后没有警告信息, 表明该证书通过了验证。

进一步可在图 5-12 中点击“证书路径”标签, 将出现图 5-13 所示的证书路径窗口, 从中可以看出该证书“Liu Fang”是由“Xu Yingxiao”签发的。Windows 自动检验证书的签

名，并在“证书状态”下面显示“该证书是正确的”。如果点击图 5-13 证书路径中的“Xu Yingxiao”，则图 5-13 窗口中的“查看证书”按钮将被激活，点击后将显示签发者“Xu Yingxiao”的证书。

如果我们再使用密钥库 lfkeystore 中的证书“Liu Fang”（lf.cer）对应的私钥给其他证书签名，则证书路径将更长。

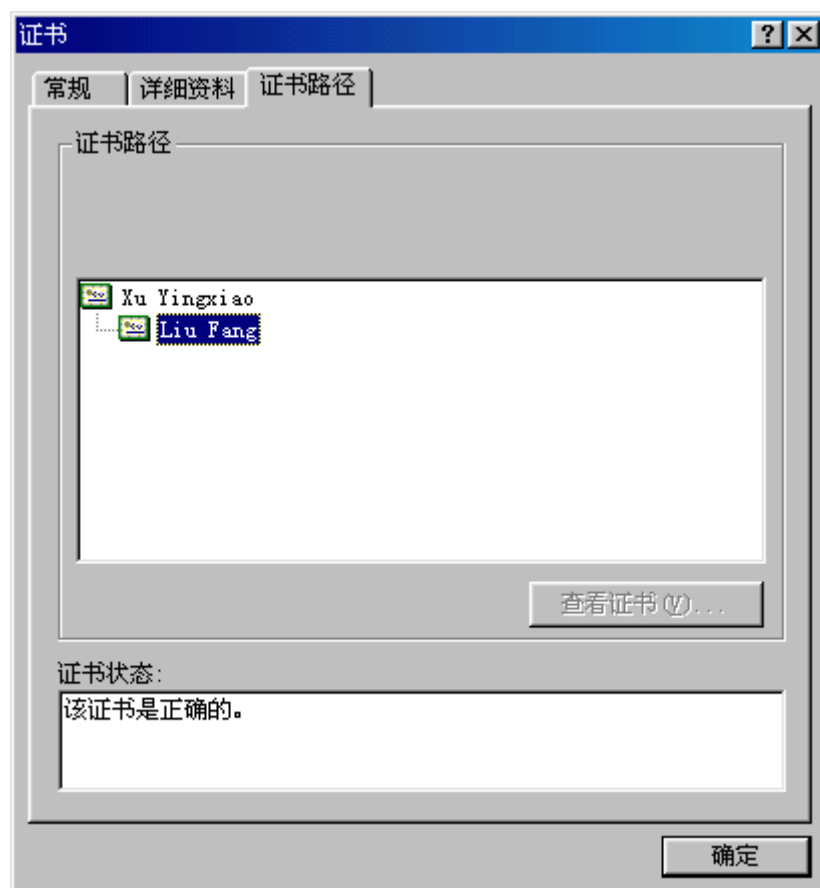


图 5-13 签名后证书的证书路径

5.5.3 Windows 中卸载证书

★ 实例说明

本实例将 5.4.1 安装的证书“Xu Yingxiao”卸载，然后再查看由“Xu Yingxiao”签名的证书。

★ 运行程序

Windows 中启动 Internet Explorer 浏览器，选择“工具/选项”菜单，出现图 5-14 所示的 Internet 选项窗口，点击其中的内容标签，出现图 5-15 所示的 Internet 选项内容设置窗口。点击其中的“证书”按钮，出现图 5-16 所示的证书管理器窗口。这里，显示了所有安装的证书。在 5.4.1 小节安装的证书“Xu Yingxiao”被自动安装在“受信任的根目录证书发行机构”中，点击该标签，出现图 5-17 的窗口，在其中选中“Xu Yingxiao”证书，点击删除，随后出现两次提示，选择“是”确认删除，则 5.4.1 小节的操作中添加的证书将被删除。Windows

不再信任 CA “Xu Yingxiao” 的证书，也就不再信任 “Xu Yingxiao” 所签发的证书。此时像 5.5.2 小节那样点击 “If_sign” 将显示 “由于信息不足，不能验证该证书”，如图 5-18 所示。

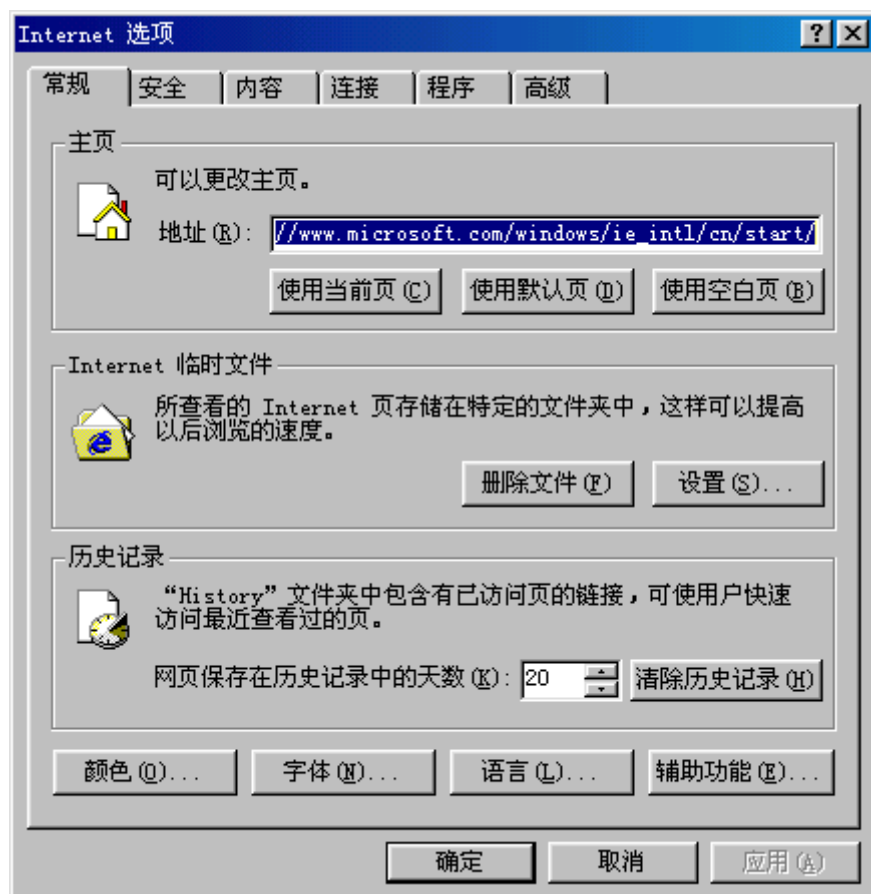


图 5-14 Internet 选项窗口



图 5-15 Internet 选项内容设置窗口

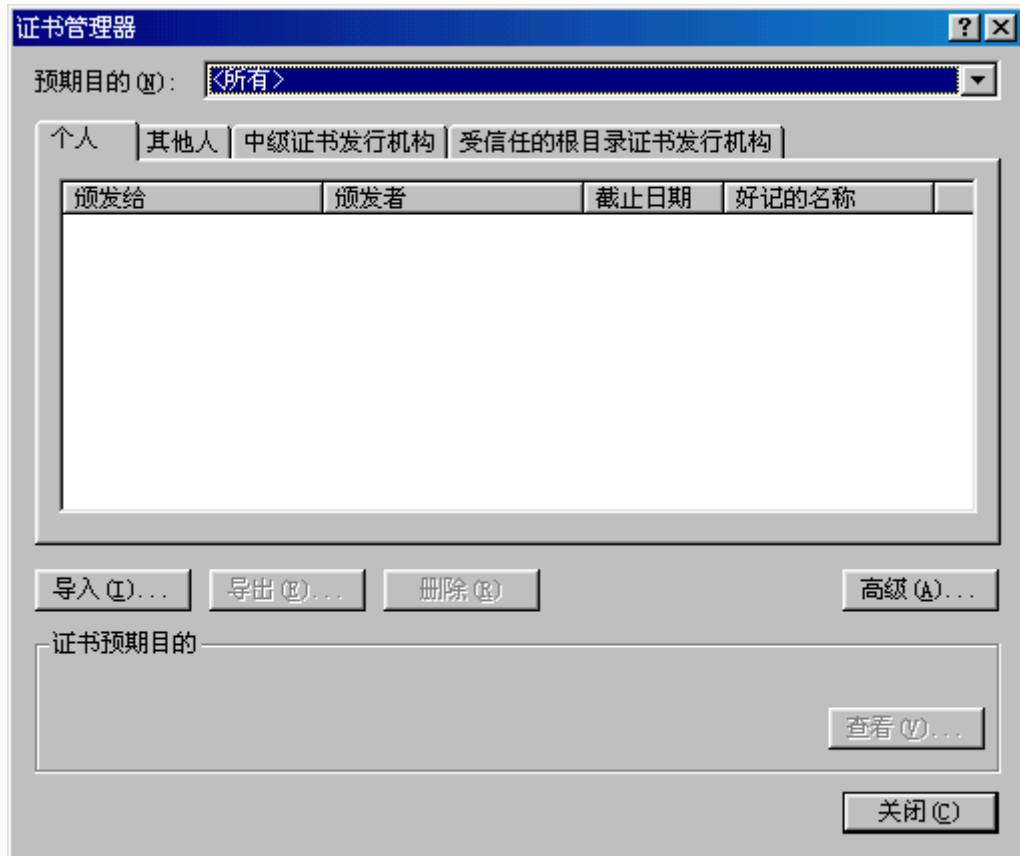


图 5-16 证书管理器窗口

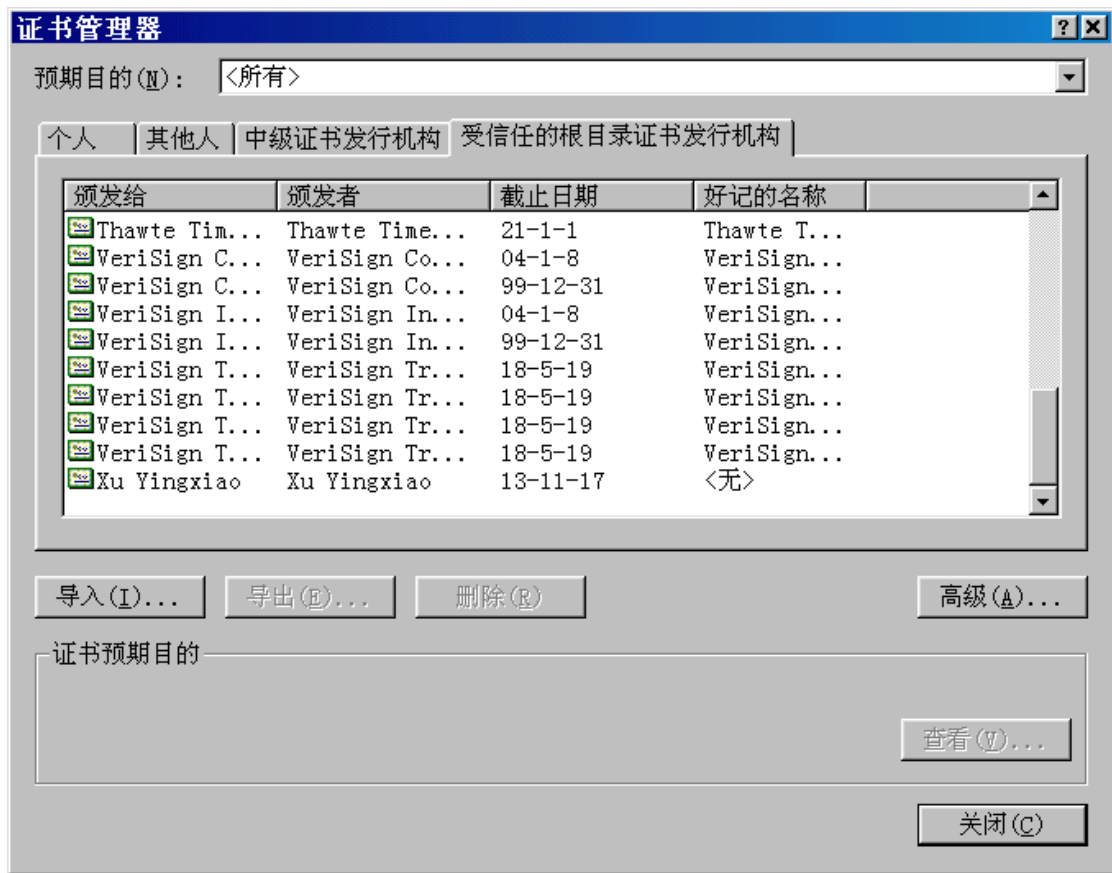


图 5-17 受信任的根目录证书发行机构

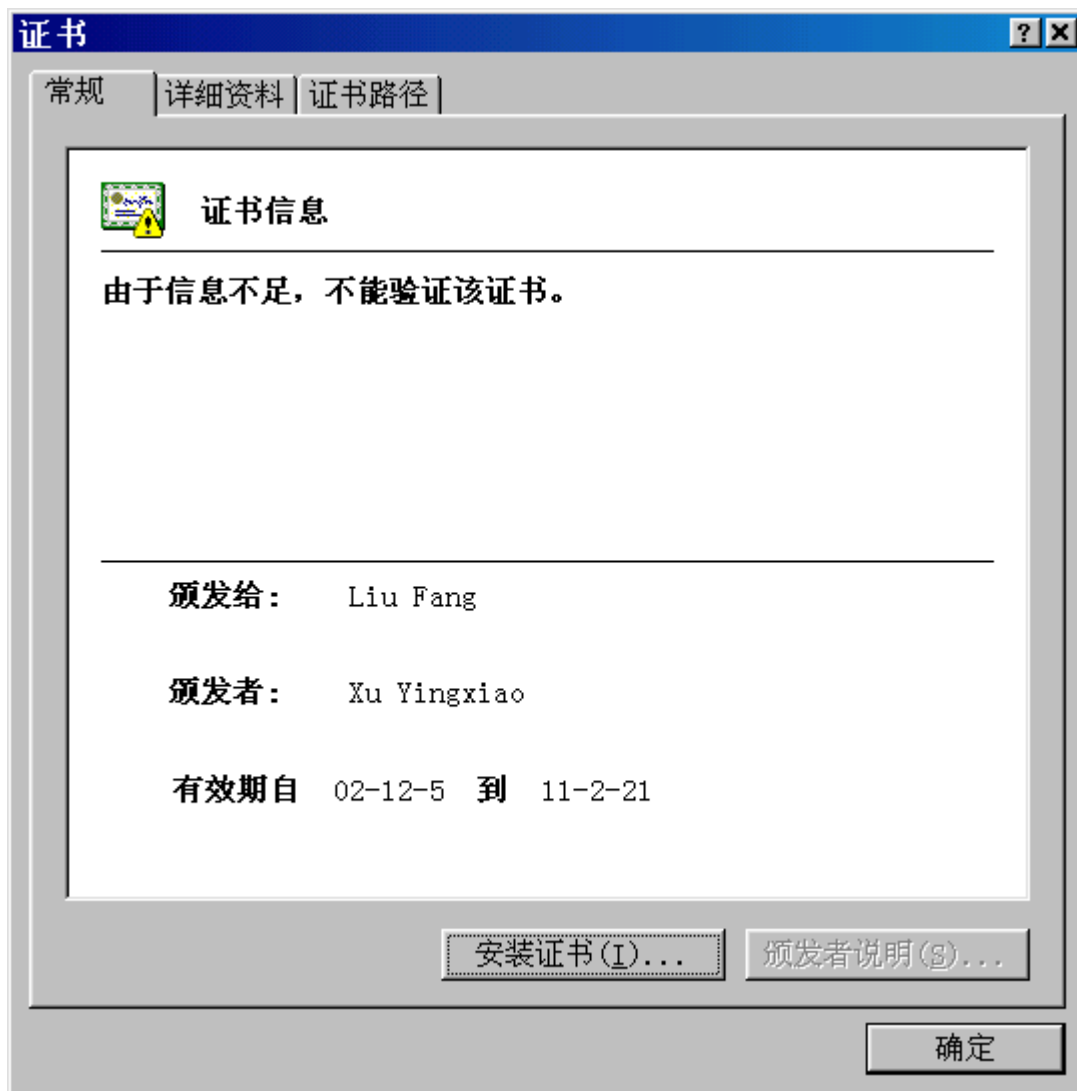


图 5-18 不再受信任的已签名证书

5.5.4 Java 程序使用 CA 公钥验证已签名的证书

5.5.2 小节是通过 Windows 程序自动验证证书是否合法。本小节通过 Java 程序验证某个证书是否确实是某个 CA 签发的。

★ 实例说明

本实例使用 CA “Xu Yingxiao” 的证书 mytest.cer，检验某个证书文件 lf_signed.cer，看它是否确实是 CA “Xu Yingxiao” 签发的。

★ 编程思路：

首先读取 CA 的证书 mytest.cer，取得其公钥，然后读取待检验的证书 lf_signed.cer，获得其证书对象后，执行证书对象的 Verify() 方法进行验证，

具体步骤如下：

- (1) 获取 CA 的证书

```
CertificateFactory cf=CertificateFactory.getInstance("X.509");
```

```
FileInputStream in1=new FileInputStream(args[0]);
java.security.cert.Certificate cac=cf.generateCertificate(in1);
```

分析：不妨和 5.2.4 一样从文件读入 CA “Xu Yingxiao” 的证书，文件名不妨从命令行参数传入。

(2) 获取待检验的证书

```
FileInputStream in2=new FileInputStream(args[1]);
java.security.cert.Certificate lfc=cf.generateCertificate(in2);
```

分析：和上一步类似，从不妨和 5.2.4 一样从文件读入待检验的证书，文件名不妨从命令行参数传入。

(3) 获取 CA 的公钥

```
PublicKey pbk=cac.getPublicKey();
```

分析：使用 CA 证书对象的 getPublicKey() 方法获得 CA 的公钥，用于证书检验。

(4) 检验证书

```
lfc.verify(pbk);
pass=true;
```

分析：执行被检验证书对象的 verify() 方法，其参数传入第 3 步获得的公钥。如果该证书确实是由该公钥签名的，将正常运行，可以执行到 pass=true 一句，否则将产生异常对象。

(5) 处理异常对象

```
catch(Exception e){
    pass=false;
}
```

分析：主要有四类异常，NoSuchAlgorithmException，InvalidKeyException，NoSuchProviderException，SignatureException 和 CertificateException 等，详见 API 文档，可以分不同异常分别处理，也可统一给 pass 变量赋值 false。

★代码与分析：

完整代码如下：

```
import java.io.*;
import java.security.*;
import java.security.cert.*;
public class CheckCertSign{
    public static void main(String args[ ]) throws Exception{
        String cacert=args[0];
        String lfcert=args[1];
        //CA 的证书
        CertificateFactory cf=CertificateFactory.getInstance("X.509");
        FileInputStream in1=new FileInputStream(cacert);
        java.security.cert.Certificate cac=cf.generateCertificate(in1);
        in1.close();
        //待检验的证书
        FileInputStream in2=new FileInputStream(lfcert);
        java.security.cert.Certificate lfc=cf.generateCertificate(in2);
        in2.close();
        PublicKey pbk=cac.getPublicKey();
```

```

        boolean pass=false;
        try{
            lfc.verify(pbk);
            pass=true;
        }
        catch(Exception e){
            pass=false;
            System.out.println(e);
        }
        if(pass){
            System.out.println("The Certificate is signed by the CA Xu Yingxiao");
        }
        else{
            System.out.println("!!!not signed by the CA Xu Yingxiao");
        }
    }
}

```

★运行程序

输入 `java CheckCertSign mytest.cer lf_signed.cer` 运行程序，将检验 `lf_signed.cer` 是否确实由 `mytest.cer` 对应的 CA 所签发，程序输出如下：

```

C:\java\ch5\check>java CheckCertSign mytest.cer lf_signed.cer
The Certificate is signed by the CA Xu Yingxiao

```

输入 `java CheckCertSign mytest.cer mytest.cer` 运行程序，将检验 `mytest.cer` 是否确实由 `mytest.cer` 对应的 CA 所签发（自签名），程序输出如下：

```

The Certificate is signed by the CA Xu Yingxiao

```

输入 `java CheckCertSign mytest.cer lf.cer` 运行程序，将检验 `lf.cer` 是否确实由 `mytest.cer` 对应的 CA 所签发，程序输出如下：

```

java.security.SignatureException: Signature does not match.
!!!The Certificate is not signed by the CA Xu Yingxiao

```

同样，如果输入如下命令再创建一个名称相同的证书：

```

keytool -genkey -dname "CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB,
ST=Shanghai, C=CN" -alias mytest -keyalg RSA -keysize 1024 -keystore hackerstore -keypass
wshr.ut -storepass wshr.ut

```

使用该的证书重复本章的步骤冒充 CA “Xu Yingxiao” 给其他证书签名，尽管签名后的证书中显示出来仍是 “CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai, C=CN” 签名的，但使用本小节的方法将可以发现它无法通过 CA “Xu Yingxiao” 公钥的验证。因为只有真正的 CA “Xu Yingxiao” 对应的私钥签发的证书才可通过本小节的验证。

本章介绍了数字证书的概念、创建、读取、签发及初步验证等。验证数字证书证在编程中用得比较多，下一章介绍和验证数字证书的相关的证书链及其验证。

第 6 章 数字化身份

——CertPath 证书链

本章重点:

在 5.5 节的编程中检验了单个证书是否是某个 CA 签名, 实际使用中经常是 A 的证书是由机构 B 的私钥签发的, 而机构 B 自身的证书是由机构 C 的私钥签发的, 机构 C 的证书又是由机构 D 的私钥签发的。这样组成了四个证书组成的证书链: A-B-C-D。若要检验证书 A 是否值得信任, 只要检验者信任 B,C,D 中的任一个证书即可。因此, 假如检验者信任某个证书 X (X 可能是 B, C 或 D), 则检验者检验证书 A 是否值得信任时, 应使用 X 的公钥沿着 A 的证书链逐个检验, 如使用 B 的公钥必须能验证 A, 若 B 的公钥和 A 的公钥相同, 则通过验证, 否则再检验 C, 依此类推, 直至最后一个 CA。

证书链本质上是一组按顺序排列的数字证书, 又称为认证链、证书路径。证书链证明了证书的合法性, 本章介绍如何检验证书链是否合法。证书链合法, 则就可以相信证书中所宣称的某个主体拥有某个公钥了。本章将介绍针对证书链的各种操作。

本章主要内容:

- 使用 Keytool 工具和 Java 程序创建并保存证书链
- 从证书文件、密钥库或 HTTPS 服务器获取代表证书链的 CertPath 对象
- 显示和保存证书链中的证书
- 验证证书链
- 使用 CertStore 来保存和提取证书
- 吊销证书

6.1 密钥库中创建并保存证书链的几种方法

在密钥库中应该能够保存证书链, 便于证书的验证。本节先介绍使用 keytool 导入证书构成证书链, 然后讨论 Java 程序如何构建证书链, 以及如何从密钥库中读取证书链。

6.1.1 使用 Keytool 将已签名的数字证书导入密钥库

★ 实例说明

本实例使用 5.4.4 小节得到的由 CA “Xu Yingxiao” 签发的证书文件 lf_signed.cer, 将其导入用户 “Liu Fang” 的密钥库, 证书 lf_signed.cer 和签发者的公钥 mytest.cer 组成证书链。得到新的密钥库 lfkeystore2。

如果我们是将证书交给 Verisign 等 CA 签发, 则得到签发后的证书文件后也可以和本小节一样进行处理。

★ 运行程序

本实例的场景是: 用户 “Liu Fang” 有一个密钥库 lfkeystore, 该密钥库中包含了用户 “Liu

Fang”的证书，在 5.2.3 小节已将其导出为 lf.cer 文件，在 5.4.3 和 5.4.4 小节中已经将证书交给 CA “Xu Yingxiao”签名。CA “Xu Yingxiao”的密钥库是 mykeystore，该密钥库中包含的 CA “Xu Yingxiao”的证书，在 5.2.3 小节已将其导出为 mytest.cer 文件。CA “Xu Yingxiao”用自己的私钥对 lf.cer 签名后得到了签名后的证书 lf_signed.cer。于是 CA “Xu Yingxiao”将 lf_signed.cer 文件和自己的证书 mytest.cer 一起发布给用户“Liu Fang”。现在，用户“Liu Fang”准备把 lf_signed.cer 导入自己的密钥库 lfkeystore。

在将第 5 章的文件 lf_signed.cer 和 lfkeystore 和 mytest.cer 拷贝到当前目录（如 c:\java\ch6\CertChain），操作之前先将 lfkeystore 文件备份一下（如备份为 lfkeystore.bak.before.6.1.1）。

Keytool 工具提供的 -import 参数可以将数字证书导入密钥库 lfkeystore，但如果我们直接导入将出现如下出错提示：

```
C:\Java\ch6\CertChain>keytool -import -alias lf -keystore lfkeystore -file lf_signed.cer
```

输入 keystore 密码: wshr.ut

keytool 错误: java.lang.Exception: 无法从回复中建立链接

这是因为导入的 lf_signed.cer 是使用 CA “Xu Yingxiao”签名的，而在密钥库 lfkeystore 中并没有 CA 的证书，因此无法建立证书链。所以，我们首先需要将 CA 的证书 mytest.cer 导入：

```
C:\Java\ch6\CertChain>keytool -import -alias CAmytest -keystore lfkeystore -file mytest.cer
```

输入 keystore 密码: wshr.ut

Owner: CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai, C=CN

发照者: CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai, C=CN

序号: 3deec043

有效期间: Thu Dec 05 10:56:03 CST 2002 至: Sun Nov 17 10:56:03 CST 2013

认证指纹:

MD5: B2: DC: 75: CD: 60: B7: 1E: 7A: 97: EE: E8: A4: 31: D6: 26: C6

SHA1: 32: E5: 89: 16: 7E: 25: 7F: 86: 16: 94: 34: 36: 95: 44: D7: CF: 14: C8: F2: 1E

信任这个认证? [否]: 是

认证已添加至 keystore 中

和 5.1.1 小节一样，这里的“是”可以从 Windows 窗口中粘贴到 DOS 窗口。该操作在密钥库 lfkeystore 中将增加一个条目“camytest”，其中包含了 CA “Xu Yingxiao”的证书。使用 keytool 可以显示一下增加后的条目：

```
C:\Java\ch6\CertChain>keytool -list -keystore lfkeystore
```

输入 keystore 密码: wshr.ut

Keystore 类型: jks

Keystore 提供者: SUN

您的 keystore 包含 2 输入

lf, 2002-12-5, keyEntry,

认证指纹 (MD5): 55: 73: 8D: 16: 05: E1: F8: 5F: F8: 25: C7: 29: C3: D6: 48: 67


```
camytest, 2002-12-5, trustedCertEntry,  
认证指纹 (MD5): B2: DC: 75: CD: 60: B7: 1E: 7A: 97: EE: E8: A4: 31: D6: 26: C6
```

这里, camytest 条目显示的是 “trustedCertEntry”, 而不是 “keyEntry”。这是因为我们在导入 mytest.cer 时只是导入了证书, 并没有导入其对应的私钥。

下面, 就可以导入签名后的证书 lf_signed.cer 了, 操作如下:

```
C: \Java\ch6\CertChain>keytool -import -alias lf -keystore lfkeystore -file  
lf_signed.cer
```

```
输入 keystore 密码: wshr.ut
```

```
认证回复已安装在 keystore 中
```

由于导入 lf_signed.cer 时使用的别名 “lf” 在密钥库 lfkeystore 中原先已经存在, 并且和证书 lf_signed.cer 中的公钥相匹配, 因此导入的证书将以 “keyEntry” 类型的条目而不是 “trustedCertEntry” 类型的条目保存, 如:

```
C: \Java\ch6\CertChain>keytool -list -keystore lfkeystore
```

```
输入 keystore 密码: wshr.ut
```

```
Keystore 类型: jks
```

```
Keystore 提供者: SUN
```

```
您的 keystore 包含 2 输入
```

```
lf, 2002-12-5, keyEntry,
```

```
认证指纹 (MD5): D3: 7E: C0: 72: 5D: 41: 46: CA: 7A: 8E: 85: 21: 1B: DA: 89: 0F
```

```
camytest, 2002-12-5, trustedCertEntry,
```

```
认证指纹 (MD5): B2: DC: 75: CD: 60: B7: 1E: 7A: 97: EE: E8: A4: 31: D6: 26: C6
```

事实上, 密钥库中原先自签名的证书已经被新的由 CA “Xu Yingxiao” 签名的证书替换了。

如果在命令中加上 -v 参数, 可以看到密钥库 lfkeystore 的详细信息, 如执行如下命令显示 lfkeystore 的详细信息, 并将结果输出到文件 1.txt 中:

```
keytool -list -v -keystore lfkeystore -storepass wshr.ut >1.txt
```

则文件 1.txt 的内容如下:

```
Keystore 类型: jks
```

```
Keystore 提供者: SUN
```

```
您的 keystore 包含 2 输入
```

```
别名名称: lf
```

```
创建日期: 2002-12-5
```

```
输入类型: KeyEntry
```

```
认证链长度: 2
```

```
认证 [1]:
```

```
Owner: CN=Liu Fang, OU=Packaging, O=Shanghai University, L=ZB, ST=Shanghai, C=CN
```

```
发照者: CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai,
```

C=CN

序号: 3deed053

有效期间: Thu Dec 05 12:04:35 CST 2002 至: Mon Feb 21 12:04:35 CST 2011

认证指纹:

MD5: D3: 7E: C0: 72: 5D: 41: 46: CA: 7A: 8E: 85: 21: 1B: DA: 89: 0F

SHA1: C8: 85: 45: B7: A8: 37: 1F: 23: DE: A3: C1: DF: A1: B4: 83: C3: B9: F1: B7: FA

认证 [2]:

Owner: CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai, C=CN

发照者: CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai,

C=CN

序号: 3deec043

有效期间: Thu Dec 05 10:56:03 CST 2002 至: Sun Nov 17 10:56:03 CST 2013

认证指纹:

MD5: B2: DC: 75: CD: 60: B7: 1E: 7A: 97: EE: E8: A4: 31: D6: 26: C6

SHA1: 32: E5: 89: 16: 7E: 25: 7F: 86: 16: 94: 34: 36: 95: 44: D7: CF: 14: C8: F2: 1E

别名名称: camytest

创建日期: 2002-12-5

输入类型: trustedCertEntry

Owner: CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai, C=CN

发照者: CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai,

C=CN

序号: 3deec043

有效期间: Thu Dec 05 10:56:03 CST 2002 至: Sun Nov 17 10:56:03 CST 2013

认证指纹:

MD5: B2: DC: 75: CD: 60: B7: 1E: 7A: 97: EE: E8: A4: 31: D6: 26: C6

SHA1: 32: E5: 89: 16: 7E: 25: 7F: 86: 16: 94: 34: 36: 95: 44: D7: CF: 14: C8: F2: 1E

从中可以看出别名 lf 的认证链长度是 2, 包含了两个证书, 第一个是 Owner: CN=Liu Fang, OU=Packaging, O=Shanghai University, L=ZB, ST=Shanghai, C=CN 的证书, 第二个是 Owner: CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai, C=CN 的证书。

不妨将导入后的 lfkeystore 文件备份为 lfkeystore2。

6.1.2 使用 Java 程序将已签名的数字证书导入密钥库

★ 实例说明

本实例的功能和 6.1.1 小节一样，将 5.4.4 小节得到的由 CA “Xu Yingxiao” 签发的证书文件 lf_signed.cer 导入用户 “Liu Fang” 密钥库。只是本实例全部通过 Java 程序来实现该功能。

★ 编程思路：

首先读取 CA 的证书 mytest.cer 和用户收到的签名后的证书 lf_signed.cer，使用这两个证书组成证书链，然后从用户的密钥库读取私钥，最后执行 KeyStore 对象的 setKeyEntry() 方法将私钥和证书一起写入密钥库，并使用 store() 方法保存为文件即可。

具体步骤如下：

(1) 获取相关参数

```
String cacert="mytest.cer";
String lfcert="lf_signed.cer";
String lfstore="lfkeystore";
char[] lfstorepass="wshr.ut".toCharArray();
char[] lfkeypass="wshr.ut".toCharArray();
```

分析： cacert 为 CA “Xu Yingxiao” 的证书文件名称，lfcert 为用户 “Liu Fang” 的证书文件名称，lfstore 为 lfcert 为用户 “Liu Fang” 的密钥库文件名称。lfstorepass 是用户 “Liu Fang” 的密钥库文件的保护口令，lfkeypass 是用户 “Liu Fang” 的密钥库文件中 lf 条目的口令，我们要导入的是该条目对应的证书签名后的证书。这里也可以看出，对于 CA，用户只需要知道其公钥即可，CA 的私钥和密钥库不需要提供给被签发者。

(2) 从 CA 的证书文件获取 X509Certificate 类型的对象

```
CertificateFactory cf=CertificateFactory.getInstance("X.509");
FileInputStream in1=new FileInputStream(cacert);
java.security.cert.Certificate cac=cf.generateCertificate(in1);
```

分析： 和 5.2.6 小节一样，其中 cacert 是 CA 的证书文件名称。

(3) 从签名后的证书文件获取 X509Certificate 类型的对象

```
FileInputStream in2=new FileInputStream(lfcert);
java.security.cert.Certificate lfc=cf.generateCertificate(in2);
```

分析： 和上一步一样，只是证书文件使用 CA 签发后发来的证书文件 lf_signed.cer。

(4) 生成证书链

```
java.security.cert.Certificate[] cchain={lfc,cac};
```

分析： 证书链由签名的证书和签发者（CA）的证书组成，如果签发者本身的证书又由上一级 CA 签发，则应把上一级的 CA 的证书放入证书链，以次类推。这里使用 Certificate 类型的数组构成证书链，将第 2 和 3 步得到的证书作为数组的元素。

(5) 获取签名证书对应的私钥

```
FileInputStream in3=new FileInputStream(lfstore);
```

```

        KeyStore ks=KeyStore.getInstance("JKS");
        ks.load(in3,lfstorepass);
        PrivateKey prk=(PrivateKey)ks.getKey("lf",lfkeypass);

```

分析：欲将签名后的证书导入密钥库，需要知道证书中公钥对应的私钥。签名证书 lf_signed.cer 是通过将密钥库 lfkeystore 中的 lf 条目对应的证书提交给 CA 签名而得到的，因此其对应的私钥应从密钥库 lfkeystore 中的 lf 条目读取。该步骤和 5.3.5 小节类似，先获取 KeyStore 对象，然后执行其 getKey() 方法读取对应的私钥。

(6) 导入证书

```

        ks.setKeyEntry("lf_signed",prk,lfstorepass,cchain);

```

分析：执行 KeyStore 对象的 setKeyEntry() 方法，方法的第一个参数为新的条目起一个别名，第二个参数为该条目对应的私钥，即第 5 步得到的私钥，第三个参数为该条目设置一个保护口令，可以和 lf 条目的口令相同，也可以另设一个口令。最后一个参数即第 4 步创建的证书链。

★代码与分析：

完整代码如下：

```

import java.io.*;
import java.security.*;
import java.security.cert.*;
import java.util.*;
import java.math.*;
import sun.security.x509.*;

public class ImportCert{
    public static void main(String args[ ]) throws Exception{
        //参数
        String cacert="mytest.cer";
        String lfcert="lf_signed.cer";
        String lfstore="lfkeystore";
        char[] lfstorepass="wshr.ut".toCharArray( );
        char[] lfkeypass="wshr.ut".toCharArray( );
        //CA "Xu Yingxiao"的证书
        CertificateFactory cf=CertificateFactory.getInstance("X.509");
        FileInputStream in1=new FileInputStream(cacert);
        java.security.cert.Certificate cac=cf.generateCertificate(in1);
        in1.close();
        //用户"Liu Fang"的签名证书
        FileInputStream in2=new FileInputStream(lfcert);
        java.security.cert.Certificate lfc=cf.generateCertificate(in2);
        in2.close();
        //证书链
        java.security.cert.Certificate[] cchain={lfc,cac};
        //用户的密钥库
        FileInputStream in3=new FileInputStream(lfstore);

```

```

        KeyStore ks=KeyStore.getInstance("JKS");
        ks.load(in3,lfstorepass);
        PrivateKey prk=(PrivateKey)ks.getKey("lf",lfkeypass);
        //导入证书
        ks.setKeyEntry("lf_signed",prk,lfstorepass,cchain);
        //保存密钥库
        FileOutputStream out4=new FileOutputStream("lfnewstore");
        ks.store(out4,"newpass".toCharArray());
        out4.close();
    }
}

```

★运行程序

运行程序之前先将 6.1.1 小节备份的文件 lfkeystore.bak.before.6.1.1 重新拷贝到 lfkeystore，以恢复 6.1.1 小节对 lfkeystore 的修改，然后输入“java ImportCert”运行程序，将得到一个新的密钥库文件 lfnewstore，使用 keytool 可以查看其内容。

```
C:\java\ch6\CertChain>Keytool -list -keystore lfnewstore -storepass newpass
```

```

Keystore 类型:  jks
Keystore 提供者:  SUN

```

您的 keystore 包含 2 输入

```

lf, 2002-12-5, keyEntry,
认证指纹 (MD5):  55: 73: 8D: 16: 05: E1: F8: 5F: F8: 25: C7: 29: C3: D6: 48: 67
lf_signed, 2002-12-5, keyEntry,
认证指纹 (MD5):  D3: 7E: C0: 72: 5D: 41: 46: CA: 7A: 8E: 85: 21: 1B: DA: 89: 0F

```

该证书中 lf_signed 条目对应的就是执行“java ImportCert”导入的已签名证书。和 6.1.1 小节一样，加上 -v 参数后可以看到更详细的信息。如输入

```
keytool -list -v -keystore lfnewstore -storepass newpass >2.txt
```

将会发现文件 2.txt 中别名 lf_signed 的证书链的长度为 2，其中包含了两个证书。

6.2 几种获取 CertPath 证书链的方法

6.1 节将证书链保存在密钥库中，本节介绍如何得到代表证书链的 CertPath 类型的对象。Java 中为了处理证书链定义了一系列类，称为 Java™ Certification Path API，其中 CertPath 类代表证书链。

6.2.1 根据证书文件生成 CertPath 类型的对象

★ 实例说明

本实例使用第 5 章得到的证书文件 lf_signed.cer 和 mytest.cer 生成 CertPath 类型的对象，证书 lf_signed.cer 是由证书 mytest.cer 对应的私钥签发的。

★ 编程思路：

Java 可以从存放证书的列表中生成 CertPath 类型的对象，因此可以将证书文件放在列表中而创建 CertPath 类型的对象。具体步骤如下：

- (1) 获取 CertificateFactory 类型的对象

```
CertificateFactory cf = CertificateFactory.getInstance("X.509");
```

分析: CertificateFactory 类是一个工厂类, 和 5.2.6 小节一样, 使用其 getInstance() 方法获得该对象。以后由它从证书文件中读取证书。

- (2) 创建列表对象

```
List mylist = new ArrayList();
```

分析: 不妨使用 ArrayList 数组列表对象, 以后各个证书存放在该列表中, 以便生成 CertPath 对象。

- (3) 读取证书文件, 存放入列表对象

```
FileInputStream in=new FileInputStream(args[i]);  
Certificate c=cf.generateCertificate(in);  
mylist.add(c);
```

分析: 不妨从命令行读入所有证书文件的名字, 和 5.2.6 小节一样分别创建其文件输入流, 通过第 1 步得到的 CertificateFactory 对象的 generateCertificate() 方法创建证书对象, 最后将其添加到第 2 步得到的列表对象中。因为证书链中的证书是有顺序关系的, 因此添加时应该按照证书签名关系的顺序添加。

- (4) 创建 CertPath 类型的对象

```
CertPath cp = cf.generateCertPath(mylist);
```

分析: 执行第 1 步得到的 CertificateFactory 对象的 generateCertPath() 方法, 参数中传入第 2、3 步得到的存放证书的列表对象即可。

★代码与分析:

完整代码如下:

```
import java.io.*;  
import java.util.*;  
import java.security.cert.*;  
public class GetCertPathCert{  
    public static void main(String args[ ]) throws Exception{  
        CertificateFactory cf = CertificateFactory.getInstance("X.509");  
        List mylist = new ArrayList();  
        for(int i=0;i<args.length;i++){  
            FileInputStream in=new FileInputStream(args[i]);  
            Certificate c=cf.generateCertificate(in);  
            mylist.add(c);  
        }  
        CertPath cp = cf.generateCertPath(mylist);  
        System.out.println(cp);  
    }  
}
```

```
}
}
```

获得 CertPath 对象后，可以将其中的所有证书打印出来。

★运行程序

将 5.4.3 小节得到的证书文件 lf_signed.cer 和 5.2.3 小节得到的证书文件 mytest.cer 拷贝到当前目录（如从 c:\java\ch5\check 目录拷贝到 c:\java\ch6\certpath 目录）。

输入 “java GetCertPathCert lf_signed.cer mytest.cer >1.txt ” 运行程序，则可以生成证书链，并打印出其内容，屏幕显示重定向到文件 1.txt。程序运行后其中保存的证书链的内容如下：

```
X.509 Cert Path: length = 2.
[
=====Certificate 1 start.
[
[
Version: V1
Subject: CN=Liu Fang, OU=Packaging, O=Shanghai University, L=ZB, ST=Shanghai, C=CN
Signature Algorithm: MD5withRSA, OID = 1.2.840.113549.1.1.4

Key: com.sun.net.ssl.internal.ssl.JSA-RSAPublicKey@7a84e4
Validity: [From: Thu Dec 05 12:04:35 CST 2002,
          To: Mon Feb 21 12:04:35 CST 2011]
Issuer: CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai, C=CN
SerialNumber: [ 3deed053 ]

]
Algorithm: [MD5withRSA]
Signature:
0000: D2 3F 52 38 62 BF ED 59 D0 E5 B1 83 E3 4C 56 C9 .?R8b..Y.....LV.
0010: 9C 8F C8 37 13 35 31 2F 36 F7 A0 9E CD 04 2C 58 ...7.51/6.....,X
0020: 72 DE 0C B6 46 F9 AF CD 96 E3 2D CF 70 9E 1A E5 r...F.....-.p...
0030: 9A B3 D9 12 97 EA 7C 97 4A F9 E6 8B 93 52 C4 42 .....J....R.B
0040: 13 6F EC 43 FD 30 ED B2 19 92 13 FD 0B DA A6 8C .o.C.0.....
0050: 9B 3F 08 62 A9 9F 4B 23 CD A8 A0 CB BE 60 09 85 .?.b..K#.....\..
0060: E4 EC 3C 5E D7 CE BC 44 E7 F5 43 0B 01 EA 93 A3 ..<^...D..C.....
0070: CB EA 83 B3 BF 2F B4 2E 83 12 54 A4 55 AE E2 5C ...../....T.U..\

]
=====Certificate 1 end.

=====Certificate 2 start.
```

```

[
[
Version: V1
Subject: CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai,
C=CN
Signature Algorithm: MD5withRSA, OID = 1.2.840.113549.1.1.4

Key: com.sun.net.ssl.internal.ssl.JSA-RSAPublicKey@c39a2d
Validity: [From: Thu Dec 05 10:56:03 CST 2002,
          To: Sun Nov 17 10:56:03 CST 2013]
Issuer: CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai, C=CN
SerialNumber: [ 3deec043 ]

]
Algorithm: [MD5withRSA]
Signature:
0000: BE D5 F3 3C FE 53 16 0E DC FE A0 1C 7C F1 AF 31 ...<.S.....1
0010: F3 3B 0C 36 2E 1D 32 1F 87 B3 B4 1D 82 BB 4A BB .;.6..2.....J.
0020: DE 5D 35 90 BC A8 CF 42 45 61 ED 3D 19 DF 7D AB .]5....BEa.=....
0030: 45 F2 4A 19 C1 6B 19 0E F7 EC CE C6 1A 40 9F A9 E.J..k.....@..
0040: 6B 8C 49 DA CC 85 67 D9 C8 91 67 DB 33 6B 47 96 k.I...g...g.3kG.
0050: 70 D6 91 69 24 43 D5 81 6C 9D C5 9D 4D 40 23 01 p..i$C..l...M@#.
0060: 65 72 B6 27 FB 1B F3 8F 4A 16 0B 31 E2 EB 19 42 er.'....J..1...B
0070: 50 C7 70 62 6E FC A4 76 03 3E 22 7C 26 00 47 ED P.pbn..v.>".&.G.

]
=====Certificate 2 end.

]

```

其中显示了证书链的长度为 2，即证书链中有两个证书，这两个证书的详细信息也显示了出来。

6.2.2 从密钥库读取证书链生成 CertPath 类型的对象

★ 实例说明

本实例使用 6.1.1 或 6.1.2 小节得到的密钥库 `lfkeystore2` 或 `lfnewstore`，读取其中保存的证书链，并生成 `CertPath` 对象。

★ 编程思路：

和以前一样得到代表密钥库的 `KeyStore` 对象，执行其 `getCertificateChian()` 方法获得指定条目的证书链，保存在数组之中，通过数组获得证书对象的列表，从而可以进一步创建

CertPath 对象。

具体步骤如下：

- (1) 获取相关参数

```
String storename=args[0];  
char[ ] storepass=args[1].toCharArray();  
String alias=args[2];
```

分析：不妨从命令行参数输入所要读取的密钥库名称、口令以及要读取哪个条目的证书链。

- (2) 获取 KeyStore 对象并加载密钥库

```
KeyStore ks = KeyStore.getInstance("JKS");  
ks.load(new FileInputStream(storename),storepass);
```

分析：和 5.2.6 一样通过 getInstance() 方法得到 KeyStore 对象，通过 load() 方法加载参数中指定的密钥库文件。

- (3) 从密钥库读取数组形式的证书链

```
java.security.cert.Certificate[] cchain =ks.getCertificateChain(alias);
```

分析：执行 KeyStore 对象的 getCertificateChain() 方法读取参数中指定条目的证书链，方法返回值是 Certificate 类型的数组，其中包含了证书链中的所有证书

- (4) 生成列表对象

```
List mylist = new ArrayList();  
for(int i=0;i<cchain.length;i++){  
    mylist.add(cchain[i]);  
}
```

分析：不妨使用 ArrayList 列表对象，将第 3 步从访问数组得到的数组形式的证书链中的各个证书添加到列表对象中。

- (5) 创建 CertPath 类型的对象

```
CertificateFactory cf = CertificateFactory.getInstance("X.509");  
CertPath cp = cf.generateCertPath(mylist);
```

分析：获取 CertificateFactory 对象，执行其 generateCertPath() 方法，参数中传入第 4 步得到的证书列表即可。

★代码与分析：

完整代码如下：

```
import java.io.*;  
import java.io.*;  
import java.security.*;  
import java.security.cert.*;  
  
public class GetCertPathtKs{  
  
    public static void main(String args[ ]) throws Exception{  
        String storename=args[0];  
        char[ ] storepass=args[1].toCharArray();  
        String alias=args[2];  
        KeyStore ks = KeyStore.getInstance("JKS");
```

```

ks.load(new FileInputStream(storename),storepass);
java.security.cert.Certificate[] cchain
    =ks.getCertificateChain(alias);
List mylist = new ArrayList();
for(int i=0;i<cchain.length;i++){
    mylist.add(cchain[i]);
}
CertificateFactory cf = CertificateFactory.getInstance("X.509");
CertPath cp = cf.generateCertPath(mylist);
System.out.println(cp);
}
}
}

```

★运行程序

将 c:\java\ch6\certchain 目录下 lfkeystore2 和 lfnewstore 拷贝到当前目录，输入“java GetCertPathKs lfkeystore2 wshr.ut lf >2.txt”运行程序，则可以显示 6.1.1 小节得到的密钥库 lfkeystore2 中 lf 条目对应的证书链，屏幕输出重定向到文件 2.txt。其内容和 6.2.1 小节的输出结果相同。

同样，输入“java GetCertPathKs lfnewstore newpass lf_signed >3.txt”运行程序，则可以显示 6.1.2 小节得到的密钥库 lfnewstore 中 lf_signed 条目对应的证书链，屏幕输出重定向到文件 3.txt。其内容和 6.2.1 小节的输出结果也相同。

6.2.3 从 HTTPS 服务器获取证书链

★ 实例说明

数字证书在网络上各种 HTTPS 服务器上用得很多，这些服务器对应的是“https://...”格式的网址，本实例介绍如何从这些服务器上获取证书链。

★ 编程思路：

SSLSession 类的 getPeerCertificates() 方法可以获得所连接的 HTTPS 服务器所使用的证书链，为了得到 SSLSession 类，可执行 SSLSocket 类的 getSession() 方法，而为了得到 SSLSocket 类，可执行 SSLSocketFactory 类的 createSocket() 方法。

具体步骤如下：

- (1) 创建 SSLServerSocketFactory 类型的对象

```
SSLSocketFactory factory = HttpsURLConnection.getDefaultSSLSocketFactory();
```

分析：执行 HttpsURLConnection 类的静态方法 getDefaultSSLSocketFactory()，获得 SSLServerSocketFactory 类型的对象。

- (2) 创建 SSLSocket 类型的对象

```
SSLSocket socket = (SSLSocket)factory.createSocket(hostname, port);
```

分析：执行上一步得到的 SSLServerSocketFactory 对象的 createSocket() 方法获得 SSLSocket 类型的对象，方法的第一个参数中指定 HTTPS 服务器的地址，第二个参数指定端口号，HTTPS 服务器一般使用 443 作为端口号。

(3) 和 HTTPS 服务器建立连接

```
socket.startHandshake();
```

分析: 执行 SSLSocket 类的 startHandshake() 方法, 则程序将和 HTTPS 服务器建立连接, 通过该连接可以得到证书链的信息。

(4) 获取连接的会话

```
SSLSession session=socket.getSession();
```

分析: 执行 SSLSocket 类的 getSession() 方法, 得到 SSLSession 类型的对象。

(5) 获取证书

```
java.security.cert.Certificate[] servercerts =  
    session.getPeerCertificates();
```

分析: 执行 SSLSession 类的 getPeerCertificates() 方法, 该程序和 HTTPS 服务器连接时服务器所使用的证书, 它返回的是 Certificate 类型的数组。。

(6) 将证书数组中内容放入列表

```
List mylist = new ArrayList();  
for(int i=0;i<servercerts.length;i++){  
    mylist.add(servercerts[i]);  
}
```

分析: 从数组中读取各个证书, 加入 ArrayList() 列表中。

(7) 由列表对象创建 CertPath 类型的对象。

```
CertificateFactory cf = CertificateFactory.getInstance("X.509");  
CertPath cp = cf.generateCertPath(mylist);
```

分析: 获取 CertificateFactory 对象, 执行其 generateCertPath() 方法, 参数中传入第 6 步得到的证书列表即可。

★代码与分析:

完整代码如下:

```
import java.io.*;  
import java.util.*;  
import java.security.cert.*;  
import javax.net.ssl.*;  
  
public class GetCertPathHttps {  
    public static void main(String args[ ])throws Exception {  
        int port = 443;  
        String hostname =args[0];  
        SSLSocketFactory factory =  
            HttpsURLConnection.getDefaultSSLSocketFactory();  
        SSLSocket socket = (SSLSocket)factory.createSocket(hostname, port);  
        // Connect to the server  
        socket.startHandshake();  
        SSLSession session=socket.getSession();  
        // Retrieve the server's certificate chain  
        java.security.cert.Certificate[] servercerts =
```

```

        session.getPeerCertificates();

List mylist = new ArrayList();
for(int i=0;i<servercerts.length;i++){
    mylist.add(servercerts[i]);
}
CertificateFactory cf = CertificateFactory.getInstance("X.509");
CertPath cp = cf.generateCertPath(mylist);
System.out.println(cp);
FileOutputStream f=new FileOutputStream("CertPath.dat");
ObjectOutputStream b=new ObjectOutputStream(f);
b.writeObject(cp);
}
}

```

程序最后将创建的 CertPath 对象以对象流的方式保存在文件 CertPath.dat 中，以便后面使用。

★运行程序

找到一些 https 开头的网址，如：

```

https://www.verisign.com/
https://happiness.dhs.org/
https://www.microsoft.com
https://www.sun.com
https://www.ftc.gov
https://intranet.ied.edu.hk
https://proxy.autistici.org
https://digitalid.verisign.com

```

在计算机已联网、可以访问这些站点的前提下，输入“java GetCertPathHttps HTTPS 服务器地址”运行程序，将得到相应服务器所使用的证书链。如输入“java GetCertPathHttps www.ftc.gov >ftc.txt”，可得到 www.ftc.gov 服务器上所使用的证书链，证书链对应的对象保存在文件 CertPath.dat 中，证书链的内容打印出来后这里重新定向到 ftc.txt 文件。该文件内容如下：

```

X.509 Cert Path: length = 3.
[
=====Certificate 1 start.
[
[
Version: V3
Subject: CN=www.ftc.gov, OU=Terms of use at www.verisign.com/rpa (c)00, O=Federal
Trade Commission, L=Washington, ST=District of Columbia, C=US
Signature Algorithm: MD5withRSA, OID = 1.2.840.113549.1.1.4

Key:  com.sun.net.ssl.internal.ssl.JSA-RSAPublicKey@6fa474

```

Validity: [From: Thu Jan 24 08:00:00 CST 2002,
To: Thu Feb 13 07:59:59 CST 2003]
Issuer: OU=www.verisign.com/CPS Incorp.by Ref. LIABILITY LTD.(c)97 VeriSign,
OU=VeriSign International Server CA - Class 3, OU="VeriSign, Inc.", O=VeriSign Trust Network
SerialNumber: [3a3249a2 dae41a4b e98009c2 84528a7d]

Certificate Extensions: 8

[1]: ObjectId: 2.16.840.1.113733.1.6.15 Criticality=false

Extension unknown: DER encoded OCTET string =

0000: 04 0B 16 09 30 30 33 32 35 37 37 39 37003257797

[2]: ObjectId: 1.3.6.1.5.5.7.1.1 Criticality=false

Extension unknown: DER encoded OCTET string =

0000: 04 28 30 26 30 24 06 08 2B 06 01 05 05 07 30 01 .(0&0\$.+.....0.
0010: 86 18 68 74 74 70 3A 2F 2F 6F 63 73 70 2E 76 65 ..http://ocsp.ve
0020: 72 69 73 69 67 6E 2E 63 6F 6D risign.com

[3]: ObjectId: 2.16.840.1.113730.1.1 Criticality=false

NetscapeCertType [

SSL server

]

[4]: ObjectId: 2.5.29.31 Criticality=false

Extension unknown: DER encoded OCTET string =

0000: 04 3F 30 3D 30 3B A0 39 A0 37 86 35 68 74 74 70 .?0=0;.9.7.5http
0010: 3A 2F 2F 63 72 6C 2E 76 65 72 69 73 69 67 6E 2E ://crl.verisign.
0020: 63 6F 6D 2F 43 6C 61 73 73 33 49 6E 74 65 72 6E com/Class3Intern
0030: 61 74 69 6F 6E 61 6C 53 65 72 76 65 72 2E 63 72 ationalServer.cr
0040: 6C 1

[5]: ObjectId: 2.16.840.1.113733.1.6.7 Criticality=false

Extension unknown: DER encoded OCTET string =

0000: 04 22 16 20 37 34 66 32 65 65 63 62 34 31 61 65 .". 74f2eecb41ae
0010: 31 64 65 33 63 62 34 37 39 31 64 32 63 32 62 30 1de3cb4791d2c2b0
0020: 30 38 36 61 086a

[6]: ObjectId: 2.5.29.37 Criticality=false

ExtendedKeyUsages [

[2.16.840.1.113730.4.1, 1.3.6.1.5.5.7.3.1, 1.3.6.1.5.5.7.3.2]]

```
[7]: ObjectId: 2.5.29.32 Criticality=false
CertificatePolicies [
  [CertificatePolicyId: [2.16.840.1.113733.1.7.23.3]
[PolicyQualifierInfo: [
  qualifierID: 1.3.6.1.5.5.7.2.1
  qualifier: 0000: 16 1C 68 74 74 70 73 3A 2F 2F 77 77 77 2E 76 65 ..https://www.ve
0010: 72 69 73 69 67 6E 2E 63 6F 6D 2F 72 70 61 risign.com/rpa

]] ]
]
```

```
[8]: ObjectId: 2.5.29.19 Criticality=false
BasicConstraints: [
CA: false
PathLen: undefined
]
```

```
]
Algorithm: [MD5withRSA]
Signature:
0000: 45 4D F8 FA F7 A8 84 DE 0E 20 A2 02 56 C2 AE BB EM..... ..V...
0010: 15 CF A5 4A 96 3A FB E0 18 05 D9 A6 DF 90 E2 63 ...J.:.....c
0020: C0 07 E3 3F BC EC 96 1F 03 E5 95 E5 FD B4 F0 49 ...?......I
0030: 8E 2F 9C 58 EB F4 CC F4 F0 66 16 54 12 5D 73 7C ./X.....f.T.]s.
0040: 18 29 3F 7F 57 23 C5 C3 A5 D4 B9 3C BB 17 46 FC .) ?.W#.....<..F.
0050: EC C3 1E F3 BB 39 D9 EC E0 58 32 3A 21 00 39 0C .....9...X2:!.9.
0060: C1 BB C1 5A FB FB B4 8D 4A D4 A0 A6 73 67 E4 3F ...Z....J...sg.?
0070: C2 BB 7E 5B CD 9C B2 88 04 C8 08 D1 02 CB 3E FA ... [. ....>.

]
```

=====Certificate 1 end.

=====Certificate 2 start.

```
[
[
Version: V3
Subject: OU=www.verisign.com/CPS Incorp.by Ref. LIABILITY LTD.(c)97
VeriSign, OU=VeriSign International Server CA - Class 3, OU="VeriSign, Inc.",
O=VeriSign Trust Network
Signature Algorithm: SHA1withRSA, OID = 1.2.840.113549.1.1.5

Key: com.sun.net.ssl.internal.ssl.JSA-RSAPublicKey@5b0afd
Validity: [From: Thu Apr 17 08:00:00 CST 1997,
```

```

To: Tue Oct 25 07:59:59 CST 2011]
Issuer: OU=Class 3 Public Primary Certification Authority, O="VeriSign, Inc.", C=US
SerialNumber: [ 78ee48de 185b2071 c9c9c3b5 1d7bddc1 ]

Certificate Extensions: 6
[1]: ObjectId: 2.16.840.1.113730.1.1 Criticality=false
NetscapeCertType [
    SSL CA
    S/MIME CA
]

[2]: ObjectId: 2.5.29.31 Criticality=false
Extension unknown: DER encoded OCTET string =
0000: 04 2D 30 2B 30 29 A0 27 A0 25 86 23 68 74 74 70 .-0+0).'.%.#http
0010: 3A 2F 2F 63 72 6C 2E 76 65 72 69 73 69 67 6E 2E ://crl.verisign.
0020: 63 6F 6D 2F 70 63 61 33 2D 67 32 2E 63 72 6C com/pca3-g2.crl

[3]: ObjectId: 2.5.29.15 Criticality=false
KeyUsage [
    Key-CertSign
    Crl-Sign
]

[4]: ObjectId: 2.5.29.37 Criticality=false
ExtendedKeyUsages [
    1.3.6.1.5.5.7.3.1, 1.3.6.1.5.5.7.3.2, 2.16.840.1.113730.4.1,
    2.16.840.1.113733.1.8.1]]

[5]: ObjectId: 2.5.29.32 Criticality=false
CertificatePolicies [
    [CertificatePolicyId: [2.16.840.1.113733.1.7.1.1]
    [PolicyQualifierInfo: [
        qualifierID: 1.3.6.1.5.5.7.2.1
        qualifier: 0000: 16 1C 68 74 74 70 73 3A 2F 2F 77 77 77 2E 76 65 ..https://www.ve
0010: 72 69 73 69 67 6E 2E 63 6F 6D 2F 43 50 53 risign.com/CPS

]] ]

[6]: ObjectId: 2.5.29.19 Criticality=false
BasicConstraints: [
    CA: true
    PathLen: 0

```

]

]

Algorithm: [SHA1withRSA]

Signature:

```
0000: 23 5D EE A6 24 05 FD 76 D3 6A 1A D6 BA 46 06 AA #]..$.v.j...F..
0010: 6A 0F 03 90 66 B2 B0 A6 C2 9E C9 1E A3 55 53 AF j...f.....US.
0020: 3E 45 FD DC 8C 27 DD 53 38 09 BB 7C 4B 2B BA 95 >E...'.S8...K+..
0030: 4A FE 70 4E 1B 69 D6 3C F7 4F 07 C5 F2 17 5A 4C J.pN.i.<.0...ZL
0040: A2 8F AC 0B 8A 06 DB B9 D4 6B C5 1D 58 DA 17 52 .....k..X..R
0050: E3 21 F1 D2 D7 5A D5 E5 AB 59 7B 21 7A 86 6A D4 .!...Z...Y.!z.j.
0060: FE 17 11 3A 53 0D 9C 60 A0 4A D9 5E E4 1D 0C 29 ...:S..^J.^...)
0070: AA 13 07 65 86 1F BF B4 C9 82 53 9C 2C 02 8F 23 ...e.....S,...#
```

]

=====Certificate 2 end.

=====Certificate 3 start.

[

[

Version: V1

Subject: OU=Class 3 Public Primary Certification Authority, O="VeriSign, Inc.", C=US

Signature Algorithm: MD2withRSA, OID = 1.2.840.113549.1.1.2

Key: com.sun.net.ssl.internal.ssl.JSA-RSAPublicKey@b64435

Validity: [From: Mon Jan 29 08:00:00 CST 1996,

To: Sat Jan 01 07:59:59 CST 2000]

Issuer: OU=Class 3 Public Primary Certification Authority, O="VeriSign, Inc.", C=US

SerialNumber: [02a10000 01]

]

Algorithm: [MD2withRSA]

Signature:

```
0000: 75 66 6C 3E D1 CD 81 DB B5 F8 2F 36 51 B6 F7 42 ufl>...../6Q..B
0010: BC CD 42 AF DC 0E FA 15 6C F8 67 93 57 3A EB B6 ..B.....l.g.W:...
0020: 92 E8 B6 01 CA 8C B7 8E 43 B4 49 65 F9 3E EE BD .....C.Ie.>..
0030: 75 46 2E C9 FC 25 5D A8 C7 2F 8B 9B 8F 68 CF B4 uF...%].../...h..
0040: 9C 97 18 C0 4D EF 1F D9 AF 82 B3 E6 64 B8 84 5C ....M.....d.\
0050: 8A 9A 07 52 43 61 FB 74 9E 5B 3A 36 FC 4C B2 FC ...RCa.t.[:6.L..
0060: 1A 3F 15 2E A5 5B 3C 1B 90 EC 88 29 E4 59 16 F9 .?...[<....).Y..
0070: CE 07 AD EC E9 DD DA D2 31 8A 4F D6 D8 EF 17 8D .....1.0.....
```

]

=====Certificate 3 end.

1

其中开头显示的“X.509 Cert Path: length = 3.”表明该证书链的长度为 3，即有三个证书。接下来是这三个证书的详细信息：

第一个证书为“CN=www.ftc.gov, OU=Terms of use at www.verisign.com/rpa (c)00, O=Federal Trade Commission, L=Washington, ST=District of Columbia, C=US”，它是 X.509 V3 版本，有效期为 2002 年 1 月 24 日至 2003 年 2 月 13 日。

签发该证书的是第二个证书：“OU=www.verisign.com/CPS Incorp.by Ref. LIABILITY LTD.(c)97 VeriSign, OU=VeriSign International Server CA - Class 3, OU="VeriSign, Inc.", O=VeriSign Trust Network”，它也是 X.509 V3 版本，有效期为 1997 年 4 月 17 日至 2011 年 10 月 25 日。

签发第二个证书的是第三个证书：“OU=Class 3 Public Primary Certification Authority, O="VeriSign, Inc.", C=US”，它是 X.509 V1 版本，有效期为 1996 年 1 月 29 日至 2000 年 1 月 1 日。

该证书链有些问题，第二个证书的有效期比其签发者（第三个证书）的有效期长。不过这并不影响本章的实例的运行。

6.3 CertPath 对象的证书显示和保存

本节介绍如何提取 6.2 节中得到的 CertPath 对象中的证书，并分别加以显示和保存。

6.3.1 显示 CertPath 中的证书

★ 实例说明

6.2 节得到了 CertPath 对象，本实例介绍如何提取该对象中包含的证书。

★ 编程思路：

执行 CertPath 对象的 getCertificates() 方法获得所有证书的列表，进而可以按照第 5 章的各种操作处理证书。

具体步骤如下：

- (1) 获取 CertPath 类型的对象

```
CertPath cp = cf.generateCertPath(mylist);
```

分析：使用 6.2.1、6.2.2 或 6.2.3 中的方法皆可。本实例不妨使用 6.2.1 小节中的方法为例。

- (2) 从 CertPath 对象的获取证书列表

```
List cplist=cp.getCertificates()
```

分析：执行 CertPath 对象的 getCertificates() 方法，返回值是列表类型。

- (3) 处理列表

```
Object[] o=cplist.toArray();
for(int i=0; i<o.length;i++){
    X509Certificate c=(X509Certificate) o[i];
```

```
}
```

分析: 访问列表中的各个元素可以得到证书链中的各个证书, 数组中的证书按照被签署的证书在前的位置排列, 可以对其进行各种转换、显示、检验等操作。

访问列表有多种方法, 这里不妨将其转换成 Object 类型的数组, 然后依次访问数组中的元素。

★代码与分析:

完整代码如下:

```
import java.io.*;
import java.util.*;
import java.security.cert.*;

public class ShowCertPath{
    public static void main(String args[ ]) throws Exception{
        CertificateFactory cf = CertificateFactory.getInstance("X.509");
        List mylist = new ArrayList();
        for(int i=0;i<args.length;i++){
            FileInputStream in=new FileInputStream(args[i]);
            Certificate c=cf.generateCertificate(in);
            mylist.add(c);
        }
        CertPath cp = cf.generateCertPath(mylist);
        List cplist=cp.getCertificates( );
        Object[ ] o=cplist.toArray();
        for(int i=0; i<o.length;i++){
            X509Certificate c=(X509Certificate) o[i];
            System.out.println(c.getSubjectDN( ));
            System.out.println("Owns PublicKey:");
            byte[ ] pbk=c.getPublicKey( ).getEncoded( );
            for(int j=0;j<pbk.length;j++){
                System.out.print(pbk[j]+",");
            }
            System.out.println("\nIssued by "+c.getIssuerDN( ));
            System.out.println("-----");
        }
    }
}
```

这里将证书链中的各个证书对象转换为 X509Certificate 类型, 打印出其中的主要信息: XXX 拥有 XXX 公钥, XXX 签署(证明)了该信息。 进一步还可以验证其签名的真假。

★运行程序

输入 “java ShowCertPath lf_signed.cer mytest.cer >xxx.txt ” 运行程序, 则可以显示 6.1.1 小节得到的密钥库 lfkeystore2 中 lf 条目对应的证书链, 屏幕输出重定向到文件 xxx.txt。其内容如下:

CN=Liu Fang, OU=Packaging, O=Shanghai University, L=ZB, ST=Shanghai, C=CN

Owns PublicKey:

48,-127,-97,48,13,6,9,42,-122,72,-122,-9,13,1,1,1,5,0,3,-127,-115,0,48,-127,-119,2,-127,-127,0,-84,-112,16,-77,-7,4,35,-41,66,-60,-46,126,53,86,105,-95,72,2,72,-26,-29,-41,17,-104,-76,-126,-125,92,110,-116,104,98,-55,-78,-41,31,51,16,72,-91,-38,-17,-94,94,107,-8,-20,32,-49,-60,-10,-12,-25,-109,44,-32,18,-9,-71,-123,-49,-116,-51,-81,120,-32,96,84,-108,36,90,85,117,12,-116,106,-77,12,-64,-93,-125,55,-17,28,-39,127,77,67,-59,90,-80,112,-81,-67,104,17,-41,-118,71,79,46,83,-47,109,-123,-50,-74,-3,-1,-65,5,99,-17,-16,112,49,-97,-75,34,-128,84,121,93,-12,-65,76,72,-113,2,3,1,0,1,

Issued by CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai, C=CN

CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai, C=CN

Owns PublicKey:

48,-127,-97,48,13,6,9,42,-122,72,-122,-9,13,1,1,1,5,0,3,-127,-115,0,48,-127,-119,2,-127,-127,0,-22,106,19,77,-35,117,16,-4,36,-73,71,117,-63,-77,-79,26,92,113,51,13,8,62,-51,-7,93,88,32,112,53,-29,71,-43,109,67,-127,-123,-52,105,-14,69,35,37,79,-73,75,15,111,112,-91,122,-128,-59,82,127,-97,-81,-10,70,-15,-111,-122,29,17,109,-81,57,102,-77,-80,-123,-65,-58,117,58,-11,126,74,112,-55,27,57,-9,90,106,4,-3,-121,-110,-70,-92,-108,-124,-46,50,112,-22,-50,49,64,-73,-80,3,88,31,65,-113,-110,-13,-92,-22,-14,-17,-35,-126,-39,108,-84,57,-26,-71,-55,7,-90,21,-96,108,-80,-21,2,3,1,0,1,

Issued by CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai, C=CN

6.3.2 保存 CertPath 中的证书

★ 实例说明

本实例如何将 CertPath 对象中的证书保存在密钥库并进而导出到文件中。

★ 编程思路:

不妨以 6.2.3 小节保存在 CertPath.dat 文件中的 CertPath 对象为例,从文件中读出该对象后,按照 6.3.1 小节的方法将其中的证书一个个取出来,然后使用 KeyStore 对象将这些证书保存到密钥库中。

具体步骤如下:

- (1) 获取 CertPath 类型的对象

```
FileInputStream f=new FileInputStream("CertPath.dat");  
ObjectInputStream b=new ObjectInputStream(f);  
CertPath cp=(CertPath)b.readObject();
```

分析: 使用 6.2.1、6.2.2 或 6.2.3 中的方法皆可。本实例不妨使用 6.2.3 小节中保存的 CertPath 对象。通过 ObjectInputStream 的 readObject() 方法读出,并强制转换为 CertPath 类型。

- (2) 创建密钥库

```
KeyStore ks=KeyStore.getInstance("JKS");
```

```
ks.load(null,null);
```

分析：使用 KeyStore 类的静态方法 getInstance() 获取 KeyStore 对象，然后执行其 load() 方法对密钥库进行初始化。由于这里是创建新的密钥库，而不是使用已有的用 keytool 工具创建的密钥库，因此 load() 方法的参数中文件名和口令都使用 null。

(3) 获得 CertPath 中的证书数组

```
List cplist=cp.getCertificates();  
Object[] o=cplist.toArray();
```

分析：和 6.3.1 小节一样，执行 CertPath 对象的 getCertificates() 方法获得证书列表，进而得到包含 CertPath 中所有证书的数组。

(4) 将 CertPath 中的证书写入密钥库

```
for(int i=0; i<o.length;i++){  
    X509Certificate c=(X509Certificate) o[i];  
    ks.setCertificateEntry("my"+i,c);  
}
```

分析：执行 KeyStore 对象的 setCertificateEntry() 方法将证书写入密钥库。该方法第二个参数是证书，第一个参数是为证书在密钥库中设置的别名，为了使每个证书别名不同，将变量 i 与字符串“my”组合生成别名，这样，证书链中的各个证书将分别以别名 my0, my1, my2, ... 保存。

(5) 保存密钥库

```
FileOutputStream output=new FileOutputStream("MyCertPathStore");  
ks.store(output,"mypass".toCharArray());
```

分析：执行 KeyStore 对象的 store() 方法保存密钥库，store() 方法第一个参数传入密钥输入流，不妨保存在“MyCertPathStore”文件中，第二个参数为该密钥文件设置一个保护口令。这里不妨设置为 mypass。

★代码与分析：

完整代码如下：

```
import java.io.*;  
import java.util.*;  
import java.security.cert.*;  
import java.security.*;  
import java.security.cert.X509Certificate;  
  
public class StoreCert {  
    public static void main(String args[] )throws Exception {  
        FileInputStream f=new FileInputStream("CertPath.dat");  
        ObjectInputStream b=new ObjectInputStream(f);  
        CertPath cp=(CertPath)b.readObject();  
        KeyStore ks=KeyStore.getInstance("JKS");  
        ks.load(null,null);  
        List cplist=cp.getCertificates();  
        Object[] o=cplist.toArray();
```

```

        for(int i=0; i<o.length;i++){
            X509Certificate c=(X509Certificate) o[i];
            ks.setCertificateEntry("my"+i,c);
        }
        FileOutputStream output=new FileOutputStream("MyCertPathStore");
        ks.store(output,"mypass".toCharArray( ));
        output.close();
    }
}

```

★运行程序

当前目录下存放 6.2.3 小节得到的“CertPath.dat”文件，输入“java StoreCert”运行程序，当前目录下将创建密钥库 MyCertPathStore，可以用 keytool 工具查看其中的内容。

```
C:\java\ch6\CertPath>keytool -list -keystore MyCertPathStore
```

输入 keystore 密码: mypass

Keystore 类型: jks

Keystore 提供者: SUN

您的 keystore 包含 3 输入

```

my2, 2002-12-6, trustedCertEntry,
认证指纹 (MD5): AC: 46: 90: 6D: F9: 38: 74: ED: 31: D4: C4: DD: ED: 59: 70: E4
my1, 2002-12-6, trustedCertEntry,
认证指纹 (MD5): 81: C8: 88: 53: 0A: FC: AD: 91: 6F: BE: 71: D9: 41: 7B: F1: 0C
my0, 2002-12-6, trustedCertEntry,
认证指纹 (MD5): 69: 95: 1F: 25: 74: 80: EB: 23: 4B: 33: 16: D3: 3C: 87: 04: 77

```

由于 CerPath 中存放的证书，不包括私钥，因此往密钥库存放证书时使用的是 setCertificateEntry() 而不是 setKeyEntry()，因此这里看到的条目类型是 trustedCertEntry，而不是 keyEntry。

进一步可以将密钥库中的证书导出到文件，只要分别执行一下命令即可：

```

keytool -export -file my0.cer -alias my0 -keystore MyCertPathStore -storepass mypass
keytool -export -file my1.cer -alias my1 -keystore MyCertPathStore -storepass mypass
keytool -export -file my2.cer -alias my2 -keystore MyCertPathStore -storepass mypass

```

6.4 验证 CertPath 证书链

本节介绍如何验证 6.2 及 6.3 小节中的证书链。

6.4.1 验证主体和签发者

★ 实例说明

本实例验证证书链中各个证书的签发者和证书链中后一个证书的主体名称是否匹配。

★ 编程思路：

得到 CertPath 对象对应的证书数组，使用 getIssuerDN() 方法和 getSubjectDN() 方法得到证书的签发者和主体，然后遍历数组，检查第 i 个证书的签发者和第 i+1 个证书的主体是否相等。

具体步骤如下：

(1) 获取证书数组

```
CertificateFactory cf = CertificateFactory.getInstance("X.509");
for (int i=0;i<args.length;i++){
    FileInputStream in=new FileInputStream(args[i]);
    certpath[i]=cf.generateCertificate(in);
}
```

分析：可以和 6.3.1 小节一样从 CertPath 对象的获取证书列表，进而得到证书数组。这里为程序简洁起见，不妨直接从命令行参数读取证书文件，创建证书数组。

(2) 遍历证书数组

```
for(int i=0;i<certpath.length-1;i++){
    Principal issuer = ((X509Certificate)certpath[i]).getIssuerDN();
    Principal subject =((X509Certificate)certpath[i+1]).getSubjectDN();
}
```

分析：使用循环语句，读取第 i 个证书的签发者和第 i+1 个证书的主体。

(3) 比较

```
if(! issuer.equals(subject) ){
    pass=false;
    reason=...
    break;
}
```

分析：若第 i 个证书的签发者和第 i+1 个证书的主体不相同，则表明没有通过主体/签发者检验，证书链实际上构不成一条链，可显示第几个证书出了问题，退出检验。若整个循环中第 i 个证书的签发者和第 i+1 个证书的主体都相同，则通过检验。

★代码与分析：

完整代码如下：

```
import java.io.*;
import java.security.*;
import java.security.cert.*;
import java.security.cert.Certificate;
import java.util.*;

public class ValiSubIssu{
    public static void main(String args[ ]) throws Exception{
        Certificate[] certpath=new Certificate[args.length];
    }
}
```

```

CertificateFactory cf = CertificateFactory.getInstance("X.509");

for (int i=0;i<args.length;i++){
    FileInputStream in=new FileInputStream(args[i]);
    certpath[i]=cf.generateCertificate(in);
}
boolean pass=true;
String reason="";
for(int i=0;i<certpath.length-1;i++){
    Principal issuer = ((X509Certificate)certpath[i]).
        getIssuerDN();
    Principal subject =((X509Certificate)certpath[i+1]).
        getSubjectDN();
    if(! issuer.equals(subject) ){
        pass=false;
        reason="in "+i+"\n";
        reason+="issuer is " +issuer+"\n";
        reason+="But in "+(i+1)+"\n";
        reason+="subject is "+subject+"\n";
        break;
    }
}
if(pass){
    System.out.println("OK");
}
else{
    System.out.println("Wrong \n"+reason);
}
}
}

```

★运行程序

当前目录下存放 6.3.2 小节得到的证书文件 my0.cer, my1.cer 和 my2.cer, 或 6.2.1 小节所使用的证书文件 lf_signed.cer 和 mytest.cer。

输入 “java ValiSubIssu my0.cer my1.cer my2.cer” 运行程序, 将验证 my0.cer、my1.cer 和 my2.cer 三个证书组成的证书链, 显示 “subject/issuer verification OK”。

输入 “java ValiSubIssu my0.cer my2.cer my1.cer” 运行程序, 则显示 “Wrong”, 并给出出错的原因:

```

subject/issuer verification Wrong
in 0
    issuer is OU=www.verisign.com/CPS Incorp.by Ref. LIABILITY LTD.(c)97 VeriSign,
OU=VeriSign International Server CA - Class 3, OU="VeriSign, Inc.", O=VeriSign Trust Network
But in 1

```

```
subject is OU=Class 3 Public Primary Certification Authority, O="VeriSign, Inc.", C=US"
```

输入“java ValiSubIssu lf_signed.cer mytest.cer”运行程序，将验证 lf_signed.cer 和 mytest.cer 两个证书组成的证书链，显示“subject/issuer verification OK”。

输入“java ValiSubIssu mytest.cer lf_signed.cer”运行程序，则显示“Wrong”，并给出出错的原因：

```
subject/issuer verification Wrong
in 0
issuer is CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB,
ST=Shanghai, C=CN
But in 1
subject is CN=Liu Fang, OU=Packaging, O=Shanghai University, L=ZB, ST=Shanghai, C=CN
```

6.4.2 验证签名

★ 实例说明

本实例验证证书链中各个证书的签名是否正确

★ 编程思路：

得到 CertPath 对象对应的证书数组，使用 getIssuerDN() 方法和 getSubjectDN() 方法得到证书的签发者和主体，然后遍历数组，检查第 i 个证书的签发者和第 i+1 个证书的主体是否相等。

具体步骤如下：

(1) 获取待验证的证书数组

```
for (i=0;i<args.length-1;i++){
    FileInputStream in=new FileInputStream(args[i]);
    certpath[i]=cf.generateCertificate(in);
}
```

分析：可以和 6.3.1 小节一样从 CertPath 对象的获取证书列表，进而得到证书数组。这里为程序简洁起见，不妨和 6.4.1 小节一样直接从命令行参数读取证书文件，创建证书数组。由于验证签名时需要一个最信任的证书，本实例准备从命令行参数传入一个证书作为信任的根证书，因此这里读取证书数组时循环条件设置为 i<args.length-1，而不是 6.4.1 小节中的 i<args.length。

(2) 获取根证书

```
FileInputStream in=new FileInputStream(args[i]);
Certificate trust=cf.generateCertificate(in);
```

分析：第 1 步的循环结束后，变量 i 的值为 args.length-1，因此此时 args[i] 对应的是命令行最后一个参数，不妨通过它指定的文件读取根证书。

(3) 遍历证书数组，读取公钥

```
for(i=0;i<certpath.length;i++)
    if(i==certpath.length-1)
        PublicKey pbk =trust.getPublicKey();
```


else

PublicKey pbk = certpath[i+1].getPublicKey();

分析：使用循环语句，由于证书链中第 i 个证书是使用第 i+1 个证书的公钥来验证的，因此读取第 i+1 个证书的公钥。由于最后一个证书是使用根证书的公钥来验证的，因此当 i 的值为最后一个时，读取的是第 2 步得到的根证书的公钥。

(4) 验证证书的签名

certpath[i].verify(pbk);

分析：和 5.5.4 小节一样，执行被检证书对象的 verify() 方法进行验证。如果验证失败，将产生异常。

★代码与分析：

完整代码如下：

```
import java.io.*;
import java.security.*;
import java.security.cert.*;
import java.security.cert.Certificate;
import java.util.*;

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

        Certificate[] certpath=new Certificate[args.length-1];
        CertificateFactory cf = CertificateFactory.getInstance("X.509");
        int i;
        for (i=0;i<args.length-1;i++){
            FileInputStream in=new FileInputStream(args[i]);
            certpath[i]=cf.generateCertificate(in);
        }
        // RootCert
        FileInputStream in=new FileInputStream(args[i]);
        Certificate trust=cf.generateCertificate(in);

        boolean pass=false;
        String reason="";
        for(i=0;i<certpath.length;i++){
            try{
                PublicKey pbk;
                if(i==certpath.length-1){
                    pbk =trust.getPublicKey();
                }
                else{
                    pbk = certpath[i+1].getPublicKey();
                }
                certpath[i].verify(pbk);
            }
```

```

        pass=true;
    }
    catch (Exception e){
        pass=false;
        reason+=i+" "+e.toString( );
        break;
    }
}

if(pass){
    System.out.println("signature verification OK");
}
else{
    System.out.println("signature verification failed in "
        +reason);
}
}
}

```

★运行程序

当前目录下存放 6.3.2 小节得到的证书文件 my0.cer, my1.cer 和 my2.cer, 或 6.2.1 小节所使用的证书文件 lf_signed.cer 和 mytest.cer。

输入 “java ValiSign my0.cer my1.cer my2.cer” 运行程序, 将验证 my0.cer 和 my1.cer 两个证书组成的证书链, 根证书是 my2.cer, 运行后显示 “signature verification OK”。

输入 “java ValiSign my0.cer my1.cer my2.cer my2.cer” 运行程序, 将验证 “my0.cer、my1.cer 和 my2.cer” 三个证书组成的证书链, 根证书是 my2.cer, 运行后显示 “signature verification OK”。

输入 “java ValiSign my1.cer my2.cer my0.cer my2.cer” 运行程序, 由于第 2 个证书 my2.cer 不是 my0.cer 对应的私钥签发的, 因此将显示出错信息:

```
signature verification failed in 1 java.security.SignatureException: Signature does not
match.
```

它表明第 2 个证书验证出错。

输入 “java ValiSign lf_signed.cer mytest.cer mytest.cer” 运行程序, 将验证 lf_signed.cer 和 mytest.cer 两个证书组成的证书链, 根证书是 mytest.cer, 运行后显示 “signature verification OK”。

6.4.3 CertPathValidator 类基于 TrustAnchor 验证证书链

★ 实例说明

证书链的实际验证要比 6.4.1 和 6.4.2 小节复杂得多, 本节介绍如何使用 CertPathValidator 类直接对 CertPath 类型的对象进行验证, 它使用 TrustAnchor 对象设置最信任哪个 CA。

★ 编程思路:

CertPathValidator 类中的 validate() 方法可以使用现成的 PKIX certification path 验证算法直接验证 CertPath 类型的对象。方法的第一个参数传入要验证的 CertPath 对象，第二个参数传入 PKIXParameters 类型的对象，它提供了验证时所使用的参数。

为了得到 PKIXParameters 类型的对象，必须指定最信任哪些 CA。

具体步骤如下：

(1) 获取待验证的 CertPath 对象

```
List mylist = new ArrayList();
for (i=0;i<args.length-1;i++){
    FileInputStream in=new FileInputStream(args[i]);
    Certificate c=cf.generateCertificate(in);
    mylist.add(c);
}
CertPath cp = cf.generateCertPath(mylist);
```

分析：可以使用 6.2 节的各种方法得到 CertPath 对象，这里不妨从命令行参数读入证书组放入证书列表中，进而创建 CertPath 对象，由于命令行参数最后一项准备传入最新任的 CA 的证书，因此读取证书创建 CertPath 的循环中循环条件设置为 `i<args.length-1`，而不是 `i<args.length`。如果最信任的证书有两个，则这里应将循环条件设置为 `i<args.length-2`，以次类推。

(2) 读取最信任的 CA 的证书

```
FileInputStream in=new FileInputStream(args[i]);
Certificate trust=cf.generateCertificate(in);
```

分析：第 1 步的循环结束后，变量 `i` 的值为 `args.length-1`，因此此时 `args[i]` 对应的是命令行最后一个参数，不妨通过它指定的文件读取最信任的 CA 的证书。

如果最信任的证书有多个，可以类似地再读取 `args[i+1]`、`args[i+2]` 对应的证书。

(3) 创建 TrustAnchor 对象

```
TrustAnchor anchor = new TrustAnchor( (X509Certificate)trust,null);
```

分析：TrustAnchor 对象代表最信任的 CA，其构造器的第一个参数传入相应的证书，即第 2 步从最后一个命令行参数传入的证书。第二个参数这里不妨设置为 `null`。

(4) 创建和设置 PKIXParameters 对象

```
PKIXParameters params =
    new PKIXParameters(Collections.singleton(anchor));
params.setRevocationEnabled(false);
```

分析：PKIXParameters 对象的构造器中传入的是 TrustAnchor 对象的集合，这里不妨只使用一个 TrustAnchor 对象，使用静态方法 `Collection.singleton()` 创建一个只有一个元素的集合，集合中只包含上一步创建的 TrustAnchor 对象。

如果有多个信任的证书，可以将多个 TrustAnchor 对象放在集合中传入 PKIXParameters 类的构造器。

为了简化程序，不妨执行所得到的 PKIXParameters 对象的 `setRevocationEnabled`，传入一个参数 `false`，这样将不检查证书是否已被吊销。

(5) 创建 CertPathValidator 对象

```
CertPathValidator cpv = CertPathValidator.getInstance("PKIX");
```

分析： 执行 CertPathValidator 类的静态方法 getInstance() 获得其实例，方法的参数指定验证时所使用的算法，这里使用“PKIX”算法。

(6) 执行验证

```
PKIXCertPathValidatorResult result =  
    (PKIXCertPathValidatorResult) cpv.validate(cp, params);
```

分析： 执行 CertPathValidator 类的方法 validate()，方法的第一个参数传入第一步得到的 CertPath 对象，第二个参数传入第四步得到的 PKIXParameters 对象。如果验证通过，将得到 PKIXCertPathValidatorResult 类型的对象，它代表验证成功后的结果。如果验证失败，将生成异常，可通过 try...catch...语句进行处理，执行异常对象的 getIndex() 方法可以得到是因为第几个证书造成了验证失败（从 0 开始计），执行异常对象的 getMessage() 方法可以获得验证失败的原因。

★代码与分析：

完整代码如下：

```
import java.io.*;  
import java.security.*;  
import java.security.cert.*;  
import java.security.cert.Certificate;  
import java.util.*;  
  
public class ValidateCP{  
    public static void main(String args[ ]) throws Exception{  
        CertificateFactory cf = CertificateFactory.getInstance("X.509");  
        int i;  
        List mylist = new ArrayList();  
        for (i=0;i<args.length-1;i++){  
            FileInputStream in=new FileInputStream(args[i]);  
            Certificate c=cf.generateCertificate(in);  
            mylist.add(c);  
        }  
        CertPath cp = cf.generateCertPath(mylist);  
        FileInputStream in=new FileInputStream(args[i]);  
        Certificate trust=cf.generateCertificate(in);  
        // Create TrustAnchor  
        TrustAnchor anchor = new TrustAnchor( (X509Certificate)trust,null);  
        // Set the PKIX parameters  
        PKIXParameters params =  
            new PKIXParameters(Collections.singleton(anchor));  
        params.setRevocationEnabled(false);  
        CertPathValidator cpv = CertPathValidator.getInstance("PKIX");  
        try {  
            PKIXCertPathValidatorResult result =
```

```

        (PKIXCertPathValidatorResult) cpv.validate(cp, params);
        System.out.println(result);
        System.out.println("Validation OK");
    } catch (CertPathValidatorException cpve) {
        System.out.println("Validation failure, cert["
            + cpve.getIndex() + "] : " + cpve.getMessage());
    }
}
}
}

```

★运行程序

当前目录下存放 6.3.2 小节得到的证书文件 my0.cer, my1.cer 和 my2.cer, 或 6.2.1 小节所使用的证书文件 lf_signed.cer 和 mytest.cer。

输入“java ValidateCP my0.cer my1.cer my2.cer”运行程序, 将验证 my0.cer 和 my1.cer 组成的证书链, 信任的 CA 使用的证书是 my2.cer。屏幕输出如下验证结果:

```

PKIXCertPathValidatorResult: [
    Trust Anchor: [
        Trusted CA cert: [
            [
                Version: V1
                Subject: OU=Class 3 Public Primary Certification Authority, O="VeriSign, Inc.",
                C=US
                Signature Algorithm: MD2withRSA, OID = 1.2.840.113549.1.1.2

                Key: com.sun.net.ssl.internal.ssl.JSA-RSAPublicKey@52544e
                Validity: [From: Mon Jan 29 08:00:00 CST 1996,
                    To: Sat Jan 01 07:59:59 CST 2000]
                Issuer: OU=Class 3 Public Primary Certification Authority, O="VeriSign, Inc.",
                C=US
                SerialNumber: [ 02a10000 01]

            ]
            Algorithm: [MD2withRSA]
            Signature:
            0000: 75 66 6C 3E D1 CD 81 DB B5 F8 2F 36 51 B6 F7 42 ufl>...../6Q..B
            0010: BC CD 42 AF DC 0E FA 15 6C F8 67 93 57 3A EB B6 ..B.....l.g.W:..
            0020: 92 E8 B6 01 CA 8C B7 8E 43 B4 49 65 F9 3E EE BD .....C.Ie.>..
            0030: 75 46 2E C9 FC 25 5D A8 C7 2F 8B 9B 8F 68 CF B4 uF...%]../...h..
            0040: 9C 97 18 C0 4D EF 1F D9 AF 82 B3 E6 64 B8 84 5C ....M.....d..\
            0050: 8A 9A 07 52 43 61 FB 74 9E 5B 3A 36 FC 4C B2 FC ...RCa.t.[:6.L..
            0060: 1A 3F 15 2E A5 5B 3C 1B 90 EC 88 29 E4 59 16 F9 ?....[<....).Y..
            0070: CE 07 AD EC E9 DD DA D2 31 8A 4F D6 D8 EF 17 8D .....1.0.....

        ]
    ]
]

```

```

Policy Tree: null
Subject Public Key: com.sun.net.ssl.internal.ssl.JSA-RSAPublicKey@77158a
]
Validation OK

```

其中包含了所信任的证书等信息。

输入“java ValidateCP lf_signed.cer mytest.cer”运行程序，同样可以通过验证。

6.4.4 CertPathValidator 类基于密钥库验证证书链

★ 实例说明

6.4.3 小节从命令行中输入了最信任的 CA 的证书，创建 TrustAnchor 对象传递给 PKIXParameters 类的构造器供验证用。本实例介绍如何从密钥库传入最信任的 CA 的证书。

★ 编程思路：

和 6.4.3 小节一样使用 CertPathValidator 类的 validate() 方法验证 CertPath 类型的对象。方法的第一个参数传入要验证的 CertPath 对象，第二个参数传入 PKIXParameters 类型的对象，它提供了验证时所使用的参数。只是在创建 PKIXParameters 对象时不再传入 TrustAnchor 对象，而是传入密钥库。

具体步骤如下：

(1) 获取待验证的 CertPath 对象

```

List mylist = new ArrayList();
for (i=0;i<args.length-2;i++){
    FileInputStream in=new FileInputStream(args[i]);
    Certificate c=cf.generateCertificate(in);
    mylist.add(c);
}
CertPath cp = cf.generateCertPath(mylist);

```

分析：和 6.4.3 小节类似，不妨从命令行参数读入证书组放入证书列表中，进而创建 CertPath 对象，由于使用密钥库需要密钥库文件名称及密钥库的保护密码，因此准备在命令行参数最后两项传入这两个值，因此读取证书创建 CertPath 的循环中循环条件设置为 i<args.length-2，而不是 6.4.3 小节的 i<args.length-1。

(2) 获取密钥库

```

FileInputStream kin=new FileInputStream(args[i]);
KeyStore ks=KeyStore.getInstance("JKS");
ks.load(kin,args[i+1].toCharArray());

```

分析：第 1 步的循环结束后，变量 i 的值为 args.length-2，因此此时 args[i] 对应的是命令行倒数第二个参数，传入密钥库的名称，args[i+1] 是最后一个参数，传入密钥库的保护密码。

(3) 创建和设置 PKIXParameters 对象

```
PKIXParameters params = new PKIXParameters(ks);
params.setRevocationEnabled(false);
```

分析：这里 PKIXParameters 对象的构造器中传入的不再是 TrustAnchor 对象的集合，而是上一步得到的密钥库 Keystore 对象。

和 6.4.3 小节一样，为了简化程序，不妨执行所得到的 PKIXParameters 对象的 setRevocationEnabled，传入一个参数 false，这样将不检查证书是否已被吊销。

(4) 创建 CertPathValidator 对象

```
CertPathValidator cpv = CertPathValidator.getInstance("PKIX");
```

分析：执行 CertPathValidator 类的静态方法 getInstance() 获得其实例，方法的参数指定验证时所使用的算法，这里使用“PKIX”算法。

(5) 执行验证

```
PKIXCertPathValidatorResult result =
    (PKIXCertPathValidatorResult) cpv.validate(cp, params);
```

分析：和 6.4.3 小节一样执行 CertPathValidator 类的方法 validate() 进行验证，方法返回 PKIXCertPathValidatorResult 类型的对象。

(6) 显示和验证相关的信息

```
PublicKey pbk=result.getPublicKey();
byte[] pkenc=pbk.getEncoded();
TrustAnchor anc=result.getTrustAnchor();
X509Certificate xc=anc.getTrustedCert();
```

分析：6.4.3 小节程序最后直接将验证结果转变成字符串显示出来。这里不妨显示一些具体的信息，如被验证者的公钥、验证时从密钥库提取到的最信任的 CA 的证书等。执行上一步 PKIXCertPathValidatorResult 对象的 getPublicKey() 方法可以得到被验证者的公钥，执行其 getTrustAnchor() 方法可以得到从密钥库提取的 TrustAnchor，进而可以执行 getTrustedCert() 方法获得验证时所使用的最信任的 CA 的证书。

★代码与分析：

完整代码如下：

```
import java.io.*;
import java.io.*;
import java.math.*;
import java.security.*;
import java.security.cert.*;
import java.security.cert.Certificate;
import java.util.*;

public class ValidateCPKs{
    public static void main(String args[] ) throws Exception{
        CertificateFactory cf = CertificateFactory.getInstance("X.509");
        int i;
```

```

        List mylist = new ArrayList();
        for (i=0;i<args.length-2;i++){
FileInputStream in=new FileInputStream(args[i]);
            Certificate c=cf.generateCertificate(in);
            mylist.add(c);
        }
        CertPath cp = cf.generateCertPath(mylist);

        FileInputStream kin=new FileInputStream(args[i]);
        KeyStore ks=KeyStore.getInstance("JKS");
        ks.load(kin,args[i+1].toCharArray());

        // Set the PKIX parameters
        PKIXParameters params = new PKIXParameters(ks);
        params.setRevocationEnabled(false);

        CertPathValidator cpv = CertPathValidator.getInstance("PKIX");
        try {
            PKIXCertPathValidatorResult result =
                (PKIXCertPathValidatorResult) cpv.validate(cp, params);

            PublicKey pbk=result.getPublicKey();
            byte[] pkenc=pbk.getEncoded();
            System.out.println("公钥");
            BigInteger pk=new BigInteger(pkenc);
            System.out.println(pk.toString(16));

            TrustAnchor anc=result.getTrustAnchor();
            X509Certificate xc=anc.getTrustedCert();
            System.out.println(xc.getSubjectDN());
            System.out.println(xc.getIssuerDN());
            System.out.println("Validation OK");
        } catch (CertPathValidatorException cpve) {
            System.out.println("Validation failure, cert["
                + cpve.getIndex() + "]" + ":" + cpve.getMessage());
        }
    }
}

```

这里不妨将所得到的最信任的 CA 的证书的主体和签发者打印出来。

★运行程序

当前目录下存放 6.3.2 小节得到的证书文件 my0.cer, my1.cer 和 my2.cer, 或 6.2.1 小节

所使用的证书文件 `lf_signed.cer` 和 `mytest.cer`。同时存放密钥库 `mycertpathstore` 和 `lfkeystore2`。前者是 6.3.2 小节创建的，保护密码是 `mypass`，其中包含 `my2.cer` 对应的证书。后者是 6.1.1 小节创建的，保护密码是 `wshr.ut`，其中包含 `mytest.cer` 对应的证书。

输入 “`java ValidateCPKs my0.cer my1.cer mycertpathstore mypass`” 运行程序，将验证 `my0.cer` 和 `my1.cer` 组成的证书链，信任的 CA 使用的证书从密钥库 `mycertpathstore` 中提取。屏幕输出如下验证结果：

```
C:\java\ch6\CertPath>java ValidateCPKs my0.cer my1.cer mycertpathstore mypass
公钥
30819d300d06092a864886f70d010101050003818b0030818702818100cd7a67e3f8b782c6bd470f
f2b10e4e0cb9c3997fa2395a6f64ab7b04bfc73d998fd0f7a1b2a39738a6ea56e7e39c21f2587bd1
5324a21b73d3ef643a4fc92e52364ddf7060516f30bcbcf69e03228a1aa43d7479f52284e51d100
d21b9f205d711884d1f7fc4efe14c81c06f9b842eee1d65812f075b6992e2fc1f68522e6bd020103

OU=Class 3 Public Primary Certification Authority, O="VeriSign, Inc.", C=US
OU=Class 3 Public Primary Certification Authority, O="VeriSign, Inc.", C=US
Validation OK
```

其中可以看到验证通过，从密钥库中提取的最信任的 CA 的证书是 “`OU=Class 3 Public Primary Certification Authority, O="VeriSign, Inc.", C=US`” 自签名证书。

类似地，输入 “`java ValidateCPKs lf_signed.cer lfkeystore2 wshr.ut`” 运行程序，将验证 `lf_signed.cer` 一个证书组成的证书链，信任的 CA 使用的证书从密钥库 `lfkeystore2` 中提取。屏幕输出如下验证结果：

```
C:\java\ch6\CertPath>java ValidateCPKs lf_signed.cer lfkeystore2 wshr.ut
公钥
30819f300d06092a864886f70d010101050003818d0030818902818100ac9010b3f90423d742c4d2
7e355669a1480248e6e3d71198b482835c6e8c6862c9b2d71f331048a5daefa25e6bf8ec20cfc4f6
f4e7932ce012f7b985cf8ccdaf78e0605494245a55750c8c6ab30cc0a38337ef1cd97f4d43c55ab0
70afb6d6811d78a474f2e53d16d85ceb6fdffbf0563eff070319fb5228054795df4bf4c488f020301
0001
CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai, C=CN
CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai, C=CN
Validation OK
```

其中可以看到验证通过，从密钥库中提取的最信任的 CA 的证书是 “`CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai, C=CN`” 自签名证书。

如果输入 “`java ValidateCPKs my0.cer my1.cer lfkeystore2 wshr.ut`” 运行程序，则程序显示验证失败：

```
Validation failure, cert[1] :subject/issuer name chaining check failed"
```

在验证主体和签发者时就没有通过。

此外，如果输入 “`java ValidateCPKs my0.cer my1.cer my2.cer lfkeystore2 wshr.ut`” 运行程序，将显示：

```
Validation failure, cert[2] :basic constraints check failed: this is not a CA
certificate
```

这是因为 `my2.cer` 是 V1 版本的证书，该版本中缺乏相关的约束属性。

6.5 使用 CertStore 对象保存和提取证书

创建证书链需要使用一系列证书,验证证书链也需要一系列信任的证书或已吊销证书的列表。尽管这些证书可以存储在密钥库中,但密钥库通常用于存放私钥和受信任的证书,大量非信任的证书或已吊销证书的清单一般通过 CertStore 对象来访问。本节介绍 CertStore 对象的使用。

6.5.1 创建 CertStore 对象

★ 实例说明

本实例通过已有的证书创建 CertStore 对象。

★ 编程思路:

CertStore 类通过静态方法 `getInstance()` 创建对象,该方法最简单的用法有两个参数,第一个参数通过字符串指定 CertStore 类型,即证书的存储类型,可以是“LDAP”或“Collection”字符串。前者将证书保存在 LDAP 目录中,后者将证书保存在集合中,如通过 SSL 协议的协商过程得到的证书或通过签名的 E-mail 得到的证书。第二个参数即 CertStore 初始化参数,不同的 CertStore 类型使用的参数不同。如对于 LDAP 类型,可使用 `LDAPCertStoreParameters` 类,该类的构造器中传入存放证书的 LDAP 服务器的服务器名称、端口等信息。对于 Collection 类型,可使用 `CollectionCertStoreParameters` 类,该类的构造器中传入保存有证书的集合对象。

本实例不妨以 Collection 类型的 CertStore 为例,具体步骤如下:

(14) 得到存放证书的集合对象

```
CertificateFactory cf = CertificateFactory.getInstance("X.509");
List mylist = new ArrayList();
for(int i=0;i<args.length;i++){
    FileInputStream in=new FileInputStream(args[i]);
    Certificate c=cf.generateCertificate(in);
    mylist.add(c);
}
```

分析: 可以通过密钥库、证书文件、SSL 协议等各种方式获得各种证书,所有这些证书都可以存放在一个集合对象中。和 6.2.1 小节类似,这里只不妨通过命令行参数传入证书文件,构造存放证书的集合对象。在集合对象中可以存放大量的各种证书。

(15) 设置 CertStore 参数

```
CertStoreParameters cparam=new CollectionCertStoreParameters(mylist);
```

分析: 使用 `CollectionCertStoreParameters` 类,将上一步得到的存放证书的集合对象作为构造器的参数。其中 `CertStoreParameters` 是接口类型, `CollectionCertStoreParameters` 类实现了该接口。此外 SUN 还提供了 `LDAPCertStoreParameters` 类实现了该接口。其他提供者还可能提供其他类型的实现。

(16) 创建 CertStore 对象

```
CertStore cs=CertStore.getInstance("Collection",cparam);
```

分析: 执行 CertStore 类的静态方法 getInstance(), 传入 "Collection" 字符串和上一步设置的参数。这样, 就得到了 CertStore 对象。

★代码与分析:

完整程序如下:

```
import java.io.*;
import java.security.*;
import java.security.cert.*;
import java.security.cert.Certificate;

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

        CertificateFactory cf = CertificateFactory.getInstance("X.509");
        List mylist = new ArrayList();
        for(int i=0;i<args.length;i++){
            FileInputStream in=new FileInputStream(args[i]);
            Certificate c=cf.generateCertificate(in);
            mylist.add(c);
        }
        CertStoreParameters cparam=
            new CollectionCertStoreParameters(mylist);
        CertStore cs=CertStore.getInstance("Collection",cparam);
        System.out.println(cs.getCertStoreParameters());
        System.out.println(cs.getProvider());
        System.out.println(cs.getType());
    }
}
```

程序最后通过 CertStore 对象的几个方法演示了所创建的 CertStore 对象。其中 getCertStoreParameters()方法可以得到创建 CertStore 对象时所用的参数, 包含了所有的证书信息。GetProvider()方法得到 CertStore 提供者的信息, getType()方法可以得到 CertStore 的类型。

★运行程序

在当前目录中存放 6.3.2 小节得到的 my0.cer、my1.cer、my2.cer 以及第 5 章得到的 If_signed.cer 和 mytest.cer。

输入 "java CStore my1.cer mytest.cer" 来运行程序, 程序将输出如下 CertStore 信息:

```
CollectionCertStoreParameters: [
  collection: [[
    [
      Version: V3
      Subject: OU=www.verisign.com/CPS Incorp.by Ref. LIABILITY LTD.(c)97 VeriSign,
      OU=VeriSign International Server CA - Class 3, OU="VeriSign, Inc.", O=VeriSign Trust
      Network
```

Signature Algorithm: SHA1withRSA, OID = 1.2.840.113549.1.1.5

Key: com.sun.net.ssl.internal.ssl.JSA-RSAPublicKey@decdec

Validity: [From: Thu Apr 17 08:00:00 CST 1997,
To: Tue Oct 25 07:59:59 CST 2011]

Issuer: OU=Class 3 Public Primary Certification Authority, O="VeriSign, Inc.", C=US

SerialNumber: [78ee48de 185b2071 c9c9c3b5 1d7bddc1]

Certificate Extensions: 6

[1]: ObjectId: 2.16.840.1.113730.1.1 Criticality=false

NetscapeCertType [

SSL CA

S/MIME CA

]

[2]: ObjectId: 2.5.29.31 Criticality=false

Extension unknown: DER encoded OCTET string =

0000: 04 2D 30 2B 30 29 A0 27 A0 25 86 23 68 74 74 70 .-0+0).'.%.#http

0010: 3A 2F 2F 63 72 6C 2E 76 65 72 69 73 69 67 6E 2E ://crl.verisign.

0020: 63 6F 6D 2F 70 63 61 33 2D 67 32 2E 63 72 6C com/pca3-g2.crl

[3]: ObjectId: 2.5.29.15 Criticality=false

KeyUsage [

Key-CertSign

Crl-Sign

]

[4]: ObjectId: 2.5.29.37 Criticality=false

ExtendedKeyUsages [

[1.3.6.1.5.5.7.3.1, 1.3.6.1.5.5.7.3.2, 2.16.840.1.113730.4.1,
2.16.840.1.113733.1.8.1]]

[5]: ObjectId: 2.5.29.32 Criticality=false

CertificatePolicies [

[CertificatePolicyId: [2.16.840.1.113733.1.7.1.1]

[PolicyQualifierInfo: [

qualifierID: 1.3.6.1.5.5.7.2.1

qualifier: 0000: 16 1C 68 74 74 70 73 3A 2F 2F 77 77 77 2E 76 65 ..https://www.ve

0010: 72 69 73 69 67 6E 2E 63 6F 6D 2F 43 50 53 risign.com/CPS

]]]

]

```

[6]: ObjectId: 2.5.29.19 Criticality=false
BasicConstraints: [
CA: true
PathLen: 0
]

]

Algorithm: [SHA1withRSA]
Signature:
0000: 23 5D EE A6 24 05 FD 76 D3 6A 1A D6 BA 46 06 AA #]..$.v.j...F..
0010: 6A 0F 03 90 66 B2 B0 A6 C2 9E C9 1E A3 55 53 AF j...f.....US.
0020: 3E 45 FD DC 8C 27 DD 53 38 09 BB 7C 4B 2B BA 95 >E...'.S8...K+..
0030: 4A FE 70 4E 1B 69 D6 3C F7 4F 07 C5 F2 17 5A 4C J.pN.i.<.0....ZL
0040: A2 8F AC 0B 8A 06 DB B9 D4 6B C5 1D 58 DA 17 52 .....k..X..R
0050: E3 21 F1 D2 D7 5A D5 E5 AB 59 7B 21 7A 86 6A D4 .!...Z...Y.!z.j.
0060: FE 17 11 3A 53 0D 9C 60 A0 4A D9 5E E4 1D 0C 29 ...:S..\.J.^...)
0070: AA 13 07 65 86 1F BF B4 C9 82 53 9C 2C 02 8F 23 ...e.....S.,..#

], [
[
Version: V1
Subject: CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai,
C=CN
Signature Algorithm: MD5withRSA, OID = 1.2.840.113549.1.1.4

Key: com.sun.net.ssl.internal.ssl.JSA_RSAPublicKey@415de6
Validity: [From: Thu Dec 05 10:56:03 CST 2002,
To: Sun Nov 17 10:56:03 CST 2013]
Issuer: CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai,
C=CN
SerialNumber: [ 3deec043 ]

]

Algorithm: [MD5withRSA]
Signature:
0000: BE D5 F3 3C FE 53 16 0E DC FE A0 1C 7C F1 AF 31 ...<.S.....1
0010: F3 3B 0C 36 2E 1D 32 1F 87 B3 B4 1D 82 BB 4A BB .;.6..2.....J.
0020: DE 5D 35 90 BC A8 CF 42 45 61 ED 3D 19 DF 7D AB .]5....BEa.=...
0030: 45 F2 4A 19 C1 6B 19 0E F7 EC CE C6 1A 40 9F A9 E.J..k.....@..
0040: 6B 8C 49 DA CC 85 67 D9 C8 91 67 DB 33 6B 47 96 k.I...g...g.3kG.
0050: 70 D6 91 69 24 43 D5 81 6C 9D C5 9D 4D 40 23 01 p..i$C..1...M@#.
0060: 65 72 B6 27 FB 1B F3 8F 4A 16 0B 31 E2 EB 19 42 er.'/....J..1...B
0070: 50 C7 70 62 6E FC A4 76 03 3E 22 7C 26 00 47 ED P.pbn..v.>".&.G.

```

```

]]
]
SUN version 1.2
Collection

```

前面大段内容是集合对象中各个证书的详细信息，后面两行分别是 CertStore 提供者的信息和 CertStore 类型：“Collection”。

在第一个证书 “OU=www.verisign.com/CPS Incorpor. by Ref. LIABILITY LTD.(c)97 VeriSign, OU=VeriSign International Server CA - Class 3, OU="VeriSign, Inc.", O=VeriSign Trust Network” 中可以看到其版本为 V3，因而有多个扩展项。

类似地，可以输入 “java Cstore lf_signed.cer mytest.cer my0.cer my1.cer my2.cer” 运行程序。

6.5.2 定义证书的选择标准

★ 实例说明

CertStore 对应的集合或 LDAP 数据库中可能有大量证书，可以按照各种规则从中选择满足要求的证书。本实例介绍如何制定选择证书的标准，并判断某个证书是否符合相应的标准。

★ 编程思路：

java.security.cert 包的 X509CertSelector 类可用于按照一定规则选择 X509Certificates 类型的证书，尤其常用于从 CertStore 对象中提取证书。该类的提供了一系列方法用于定义规则，同时提供 match() 方法来判断方法参数中的证书是不是满足这些规则。具体编程步骤可如下：

(1) 创建 X509CertSelector 对象

```
X509CertSelector selec=new X509CertSelector();
```

分析：直接使用不带参数的构造器创建对象。

(2) 设置规则

```

selec.setIssuer("CN=Xu Yingxiao,OU=Network Center,"+
                "O=Shanghai University,L=ZB,ST=Shanghai,C=CN");
selec.setCertificateValid(d);
BigInteger sn=new BigInteger("1039056963");
selec.setSerialNumber(sn);
selec.setSubject("CN=Xu Yingxiao,OU=Network Center,"+
                "O=Shanghai University,L=ZB,ST=Shanghai,C=CN");

```

分析：X509CertSelector 对象的各种方法可以按照各种条件来设置规则，如 setIssuer() 方法指定证书必须是由方法参数中指定的某个主体签发的，setCertificateValid() 指定证书必须是在方法参数中指定的日期仍有效的，setSerialNumber() 指定证书的序列号必须是方法参数中指定的数值，setSubject() 方法指定证书的主体必须是方法参数中所指定的。更多的方法可参考 API 文档。

注意: setIssuer() 和 setSubject() 方法的参数中, OU=、O=、L=、ST=、C=等字符串和前面的逗号之间不可有空格, 否则运行将出错。

(3) 检验证书是否满足规则

```
CertificateFactory cf = CertificateFactory.getInstance("X.509");
FileInputStream in=new FileInputStream(args[0]);
Certificate c=cf.generateCertificate(in);
if(selec.match(c)){
    System.out.println("Matched 1");
} else{
    System.out.println("not Matched 1");
}
```

分析: 这里不妨从命令行参数读入证书, 执行第一步得到的 X509CertSelector 对象的 match() 方法可以检验证书是否满足第二步设置的所有规则。只要有一个规则不匹配, 则 match() 方法的返回值就是 false。

★代码与分析:

完整的代码如下:

```
import java.util.*;
import java.io.*;
import java.math.*;
import java.security.*;
import java.security.cert.*;
import java.security.cert.Certificate;

public class DefineSelector{

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

        X509CertSelector selec=new X509CertSelector();

        //从命令行读取证书
        CertificateFactory cf = CertificateFactory.getInstance("X.509");
        FileInputStream in=new FileInputStream(args[0]);
        Certificate c=cf.generateCertificate(in);

        //检验是否满足规则
        if(selec.match(c)){
            System.out.println("Matched 1");
        } else{
            System.out.println("not Matched 1");
        }

        //增加一个规则,注意逗号后面不可有空格
        selec.setIssuer("CN=Xu Yingxiao,OU=Network Center, "+
            "O=Shanghai University,L=ZB,ST=Shanghai,C=CN");

        //检验是否满足规则
        if(selec.match(c)){
            System.out.println("Matched 2");
        } else{
```

```

        System.out.println("not Matched 2");
    }
    //读取日期值
    Calendar cld=Calendar.getInstance();
    int year=Integer.parseInt(args[1]);
    int month=Integer.parseInt(args[2])-1; // as 0 is Jan, 11
    int day=Integer.parseInt(args[3]);
    cld.set(year,month,day);
    Date d=cld.getTime();
    //增加一个规则
    selec.setCertificateValid(d);
    //检验是否满足规则
    if(selec.match(c)){
        System.out.println("Matched 3");
    } else{
        System.out.println("not Matched 3");
    }
    //增加一个规则
    BigInteger sn=new BigInteger("1039056963");
    selec.setSerialNumber(sn);
    //检验是否满足规则
    if(selec.match(c)){
        System.out.println("Matched 4");
    } else{
        System.out.println("not Matched 4");
    }
}
}
}

```

程序中首先不设置规则，检测证书是否满足某个规则，此时对于所有证书应该都可以匹配，因此“Matched 1”总是可以打印出来。然后增加了一个条件：签发者必须是“CN=Xu Yingxiao,OU=Network Center,O=Shanghai University,L=ZB,ST=Shanghai,C=CN”，如果满足这个条件，则打印“Matched 2”，再通过 Calendar 类读取用户设置的一个日期，如果证书既是“CN=Xu Yingxiao,OU=Network Center,O=Shanghai University,L=ZB,ST=Shanghai,C=CN”签发的，又在设置的日期仍有效，则打印“Matched 3”，最后设置一个序列号：“1039056963”，则只有证书既是“CN=Xu Yingxiao,OU=Network Center,O=Shanghai University,L=ZB,ST=Shanghai,C=CN”签发的，又在设置的日期仍有效，序列号又为“1039056963”（十六进制为：3DEEC043），才打印“Matched 4”

★运行程序

在当前目录中存放 6.3.2 小节得到的 my0.cer 以及第 5 章得到的 lf_signed.cer 和 mytest.cer。

输入“java DefineSelector mytest.cer 2013 1 1”运行程序，由于证书 mytest.cer 是“CN=Xu Yingxiao,OU=Network Center,O=Shanghai University,L=ZB,ST=Shanghai,C=CN”

自签名的，且有效期是 2002 年 12 月 5 日至 2013 年 11 月 17 日，序列号为“3DEEC043”，三个条件全满足，因此程序将输出：

```
Matched 1
Matched 2
Matched 3
Matched 4
```

输入“java DefineSelector lf_signed.cer 2004 1 1”运行程序，由于证书 lf_signed.cer 是“CN=Xu Yingxiao,OU=Network Center,O=Shanghai University,L=ZB,ST=Shanghai,C=CN”签发的，有效期是 2002 年 12 月 5 日至 2011 年 2 月 21 日，但序列号为“3DEE053”，故只有前面两个条件满足，因此程序输出：

```
Matched 1
Matched 2
Matched 3
not Matched 4
```

输入“java DefineSelector mytest.cer 1999 1 1”运行程序，由于在指定的日期无效，因此程序输出：

```
Matched 1
Matched 2
not Matched 3
not Matched 4
```

输入“java DefineSelector my0.cer 2003 1 1”运行程序，由于证书 my0.cer 不是“CN=Xu Yingxiao,OU=Network Center,O=Shanghai University,L=ZB,ST=Shanghai,C=CN”签发的，因此程序输出：

```
Matched 1
Not Matched 2
not Matched 3
not Matched 4
```

6.5.3 从 CertStore 中提取证书

★ 实例说明

本实例的从 6.5.1 小节得到的 CertStore 存储的大量证书中，按照 6.5.2 小节定义的规则提取所需要的证书。

★ 编程思路：

CertStore 对象的 getCertificates() 方法可以从 CertStore 中提取证书，凡是满足方法参数中指定的规则的证书都将被提取出来。

(1) 获得 CertStore 对象

```
CertStoreParameters cparam=
    new CollectionCertStoreParameters(mylist);
CertStore cs=CertStore.getInstance("Collection",cparam);
```

分析：和 6.5.1 小节一样，不妨从 ArrayList 中创建 CertStore 对象。

(2) 定义提取规则

```
X509CertSelector selec=new X509CertSelector();
selec.setIssuer("CN=Xu Yingxiao,OU=Network Center,"+
               "O=Shanghai University,L=ZB,ST=Shanghai,C=CN");
```

分析: 和 6.5.2 小节一样可以定义各种从 CertStore 对象中提取证书的规则。注意 setIssuer() 方法参数中逗号后面不可有空格

(3) 提取证书

```
Set clct=(Set) cs.getCertificates(selec);
```

分析: 执行 CertStore 对象的 getCertificates() 方法, 方法参数中传入第二步定义的规则, 方法返回 Collection 类型的对象, 这里可以将其强制转换为 Set 类型, 该集合中包含了 CertStore 中所有满足条件的证书。

(4) 处理证书

```
Object o[]=clct.toArray();
for(int i=0;i<o.length;i++){
    X509Certificate ct=(X509Certificate)o[i];
    System.out.println("Certificate "+i+" ");
    System.out.println(ct.getSubjectDN());
}
```

分析: 不妨执行上一步得到的 Set 对象的 toArray() 方法得到包含所有证书的数组, 然后通过数组访问各个证书。这里不妨将各个证书的主体打印出来。

★代码与分析:

完整代码如下:

```
import java.io.*;
import java.util.*;
import java.security.*;
import java.security.cert.*;
import java.security.cert.Certificate;
public class MySelector{
    public static void main(String args[ ]) throws Exception{
        //创建 CertStore 对象
        CertificateFactory cf = CertificateFactory.getInstance("X.509");
        List mylist = new ArrayList();
        for(int i=0;i<args.length;i++){
            FileInputStream in=new FileInputStream(args[i]);
            Certificate c=cf.generateCertificate(in);
            mylist.add(c);
        }
        CertStoreParameters cparam=
            new CollectionCertStoreParameters(mylist);
        CertStore cs=CertStore.getInstance("Collection",cparam);
        //设置规则, ,注意逗号后面不可有空格
        X509CertSelector selec=new X509CertSelector();
        selec.setIssuer("CN=Xu Yingxiao,OU=Network Center,"+
```

```

        "O=Shanghai University,L=ZB,ST=Shanghai,C=CN");
//提取证书
Set clct=(Set) cs.getCertificates(selec);
Object o[]=clct.toArray();
for(int i=0;i<o.length;i++){
    X509Certificate ct=(X509Certificate)o[i];
    System.out.println("Certificate "+i+" ");
    System.out.println(ct.getSubjectDN());
}
}
}
}

```

★运行程序

输入“java MySelector mytest.cer my0.cer my1.cer my2.cer lf_signed.cer”运行程序，将创建保存有证书 mytest.cer、my0.cer、my1.cer、my2.cer 和 lf_signed.cer 的 CertStore 对象，然后从中提取所有由“CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai, C=CN”签名的证书。程序输出如下：

```

Certificate 0
CN=Liu Fang, OU=Packaging, O=Shanghai University, L=ZB, ST=Shanghai, C=CN
Certificate 1
CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai, C=CN

```

可见提取到了两个证书。

6.6 证书的吊销

CA 在签发了某个证书后，可能因种种原因在证书尚未过期之前就需要吊销对证书的签发。比如 CA 可能发现被签发者不再值得信任（如私钥持有者跳槽了），或签发者可能不小心丢失了私钥而主动要求吊销私钥对应的证书。因此 CA 会周期性地公布已吊销证书的清单 (Certificate Revocation List, CRL)，本节介绍如何获取这些清单并使用 Java 程序进行查看。

6.6.1 查看证书吊销清单常规信息

★ 实例说明

本实例通过 Internet 下载证书吊销清单，并使用 Windows 和 Java 程序查看证书吊销清单。

★ 编程思路：

证书吊销清单（证书吊销列表）可以从 CA 的主页上下载，它一般是一个文件名以“.crl”为后缀的文件。通过它生成文件输入流传地给 CertificateFactory 类的 generateCRL() 方法，则可以创建对应的 CRL 对象。具体编程步骤如下：

- (1) 获得 CertificateFactory 对象

```
CertificateFactory cf = CertificateFactory.getInstance("X.509");
```

分析: 执行 CertificateFactory 类的 getInstance() 方法。

(2) 读取清单文件

```
FileInputStream in = new FileInputStream(args[0]);
```

分析: 不妨从命令行参数指定所下载的证书吊销清单文件名。

(3) 创建 X509CRL 对象

```
X509CRL crl =  
(X509CRL)cf.generateCRL(in);
```

分析: 执行 CertificateFactory 类的 generateCRL () 方法, 参数中传入上一步读入的清单文件, 方法返回 CRL 类型, 这里不妨将其强制转换为 X509CRL 类型, 则该对象就代表了证书吊销清单。

(4) 查看证书吊销清单

```
crl.getType()  
crl.getVersion()  
crl.getIssuerDN().getName()  
crl.getSigAlgName()  
crl.getThisUpdate()  
crl.getNextUpdate()
```

分析: 执行 X509CRL 对象的各个方法可以查看证书吊销清单的详细信息, 如 getType() 方法得到 CRL 的类型, getVersion() 方法得到 CRL 的版本, getIssuerDN() 方法从 CRL 得到签发者的名称, getSigAlgName() 方法得到 CRL 签名算法的名称, getThisUpdate() 方法得到 CRL 本次更新日期, 即生效日期, getNextUpdate() 方法得到下一次更新日期。

★代码与分析:

完整代码如下:

```
import java.io.*;  
import java.util.*;  
import java.security.cert.*;  
  
public class ShowCRLInfo{  
    public static void main(String [] args) throws Exception {  
        CertificateFactory cf = CertificateFactory.getInstance("X.509");  
        FileInputStream in = new FileInputStream(args[0]);  
        X509CRL crl =  
            (X509CRL)cf.generateCRL(in);  
        System.out.println("---CRL---");  
        System.out.println("type = " +crl.getType( ));  
        System.out.println("version = " + crl.getVersion( ));  
        System.out.println("issuer = "+crl.getIssuerDN().getName( ));  
        System.out.println("signing algorithm = "+crl.getSigAlgName( ));  
        System.out.println("this update = " + crl.getThisUpdate( ));  
    }  
}
```

```

        System.out.println("next update = " + crl.getNextUpdate( ));
        in.close();
    }
}

```

★运行程序

先从Internet上下载CA公布的证书吊销清单，作为本小节程序的试验，不妨使用“CRL download”作为关键字随便找些CA的证书吊销清单测试一下。如可从http://www.ecommercepki.com/crl/download_revocationlist.htm下载“ecpki.crl”文件，或从<http://crl.verisign.com>下载大量的吊销清单，在本书配套光盘中从该站点下载了如下几个文件：NewClass2Individual.crl、Class3Commercial.crl和GatewayCA.crl。从<http://pki.physics.auth.gr/hellasgrid-ca/CRL/>可以下载V1或V2版本的吊销清单，在本书光盘中下载了hellasgrid-v2.crl文件。

在Windows中可直接双击这些文件的图标来查看，如双击“ecpki.crl”文件可以得到图6-1所示的窗口，其中显示了各种相关的信息。该文件是12月14日下载的，从图中可见其第二天就将更新。



图 6-1 证书吊销列表常规信息

单击其中的“吊销列表”标签，可看到该列表中宣布的已吊销的证书的序列号及其吊销日期。如图6-2。



图 6-2 吊销的证书清单

同样，双击 NewClass2Individual.crl 文件可以看到图 6-3 所示的窗口，

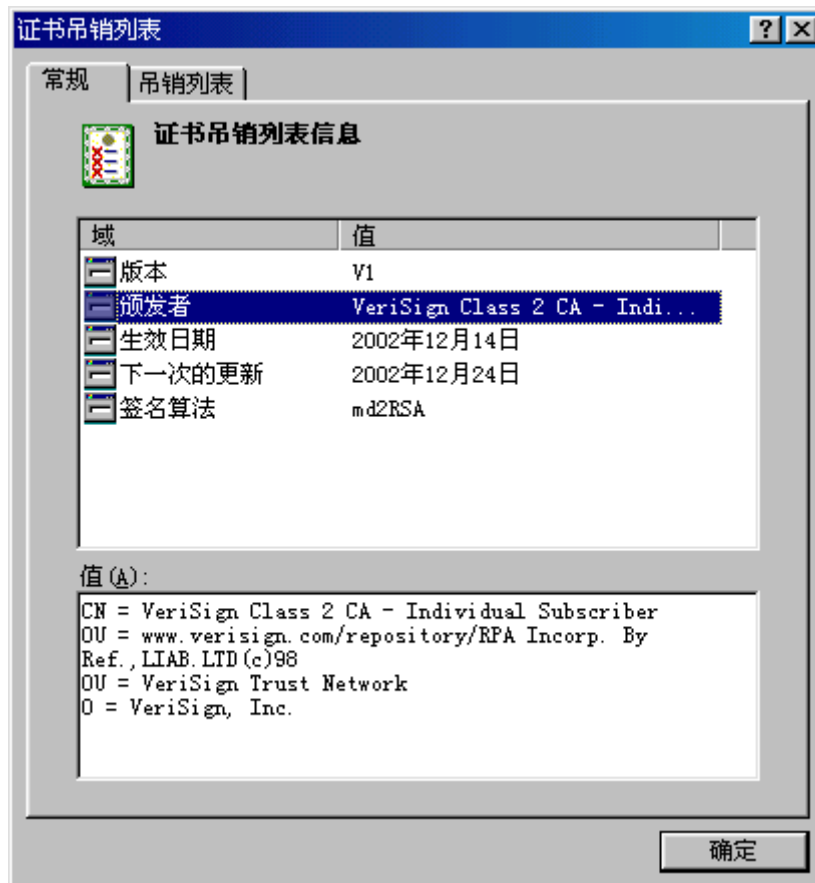


图 6-3 证书吊销列表常规信息

和图 6-1 相比，其中的信息要少一些，这是由于其版本为 V1，因而没有扩展项。该清单更新周期要长一些。类似地，单击其中的“吊销列表”标签，可看到该列表中宣布的已吊销的证书的序列号及其吊销日期。如图 6-4。

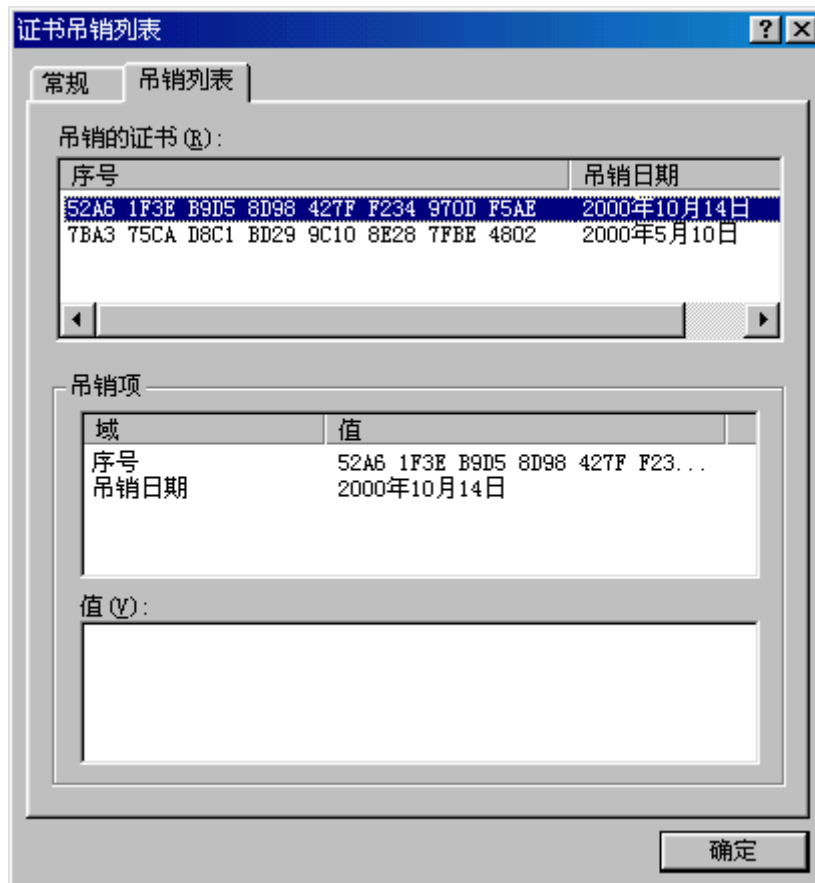


图 6-4 吊销的证书清单

下面我们可以使用本实例的 Java 程序获得同样的信息，输入“java ShowCRLInfo ecpci.crl”运行程序，将查看 ecpci.crl 的信息，输出如下：

```
---CRL---
type = X.509
version = 2
issuer = CN=E-Commerce PKI CA, OU=WiSeKey Affiliate CA, O=E-Commerce PKI, C=CH
signing algorithm = SHA1withRSA
signing OID = 1.2.840.113549.1.1.5
this update = Sat Dec 14 20:01:51 CST 2002
next update = Sun Dec 15 08:31:51 CST 2002
```

这些信息和图 6-1 中显示的信息一致。类似地，输入“java ShowCRLInfo NewClass2Individual.crl”，将显示 NewClass2Individual.crl 的信息，运行结果和图 6-3 一致，屏幕输出如下：

```
---CRL---
type = X.509
version = 1
issuer = CN=VeriSign Class 2 CA - Individual Subscriber,
OU="www.verisign.com/repository/RPA Incorp. By Ref., LIAB. LTD(c)98", OU=VeriSign Trust
Network, O="VeriS
```



```
ign, Inc."
signing algorithm = MD2withRSA
signing OID = 1.2.840.113549.1.1.2
this update = Sat Dec 14 18:07:22 CST 2002
next update = Tue Dec 24 18:07:22 CST 2002
```

从这两个输出同样可以看出，前一个清单（ecpki.crl）更新要快一些，而后一个下一次更新要 10 天以后。

6.6.2 查看清单中被吊销的证书

★ 实例说明

本实例通过 Java 程序查看证书吊销清单中被吊销的证书。

★ 编程思路：

和 6.6.1 小节一样得到代表证书吊销清单的 X509CRL 对象后，可以执行其 getRevokedCertificates() 方法获得其中的被吊销证书的信息，具体编程步骤如下：

(1) 获得 X509CRL 对象

```
CertificateFactory cf = CertificateFactory.getInstance("X.509");
FileInputStream in = new FileInputStream(args[0]);
X509CRL crl = (X509CRL)cf.generateCRL(in);
```

分析：和 6.6.1 小节一样，执行 CertificateFactory 类的 generateCRL() 方法，由命令行参数指定的证书吊销清单文件生成 X509CRL 对象

(2) 获得吊销清单中各个条目的集合

```
Set s = crl.getRevokedCertificates();
```

分析：执行上一步得到的 X509CRL 对象的 getRevokedCertificates() 方法，它返回的集合中存放的是 X509CRLEntry 类型的对象，该对象代表了证书吊销清单中的条目。

(3) 从集合中提取各个吊销清单条目

```
Iterator t=s.iterator();
while(t.hasNext()){
    X509CRLEntry entry = (X509CRLEntry)t.next();
```

分析：执行集合对象的 iterator() 方法得到迭代对象，执行其 next() 方法取出当前条目并强制转换为 X509CRLEntry 类型，执行其 hasNext() 方法查看是否还有未提取的条目，从而最终提取所有条目。

(4) 从吊销清单条目中提取对应的信息

```
entry.getSerialNumber()
entry.getRevocationDate()
entry.hasExtensions()
```

分析：执行上一步得到的 X509CRLEntry 对象的各个方法获得该条目对应的信息。如 getSerialNumber() 方法得到已吊销的证书的序列号，该方法返回的是 BigInteger 类型，不妨执行其 toString(16) 方法，以 16 进制显示序列号。此外还可以通过 getRevocationDate() 方法得到证书吊销的日期，通过 hasExtensions() 方法得到该条目是否有扩展项。

★代码与分析:

完整代码如下:

```
import java.io.*;
import java.util.*;
import java.security.cert.*;

public class ShowCRLEntries{
    public static void main(String [] args) throws Exception {
        CertificateFactory cf = CertificateFactory.getInstance("X.509");
        FileInputStream in = new FileInputStream(args[0]);
        X509CRL crl = (X509CRL)cf.generateCRL(in);
        Set s = crl.getRevokedCertificates();
        if (s != null && s.isEmpty() == false){
            Iterator t=s.iterator( );
            while(t.hasNext( )){
                X509CRLEntry entry = (X509CRLEntry)t.next();
                System.out.println("serial number = " +
                    entry.getSerialNumber());
                System.out.println("revocation date = " +
                    entry.getRevocationDate());
                System.out.println("extensions = " +
                    entry.hasExtensions());
            }
        }
        in.close();
    }
}
```

★运行程序

输入“java ShowCRLEntries NewClass2Individual.crl”运行程序，屏幕输出如下:

```
serial number = 7ba375cad8c1bd299c108e287fbe4802
revocation date = Wed May 10 09:02:27 CST 2000
extensions = false
serial number = 52a61f3eb9d58d98427ff234970df5ae
revocation date = Sat Oct 14 02:25:08 CST 2000
extensions = false
```

可见该列表中有两个吊销的证书（序列号为 7ba375cad8c1bd299c108e287fbe4802 和 52a61f3eb9d58d98427ff234970df5ae），吊销日期分别是 2000 年 5 月 10 日和 2000 年 10 月 14 日，该信息和图 6-4 的 Windows 中的显示一致。

输入“java ShowCRLEntries ecpi.crl”运行程序，屏幕将输出大量被吊销的证书，可将其重新定向到文件，部分输出如下:

```
serial number = 3df4ab53
revocation date = Tue Dec 10 22:47:06 CST 2002
extensions = true
```

```

serial number = 3b3881f2
revocation date = Wed Jun 27 00:45:22 CST 2001
extensions = true
serial number = 3bab5a3a
revocation date = Sat Sep 22 00:53:09 CST 2001
extensions = true
serial number = 3b545329
revocation date = Mon Oct 22 15:53:32 CST 2001
extensions = true

```

该信息和图 6-2 的信息一致。

6.6.3 从 CertStore 对象中提取已吊销的证书

★ 实例说明

和证书一样，大量的证书吊销清单也可以存储在 LDAP 目录、集合等中，通过 CertStore 对象来访问。本实例先创建这样的 CertStore 对象，然后使用 X509CRLSelector 类定义规则，从中提取符合条件的证书吊销清单。

★ 编程思路：

CertStore 对象的 getCRL() 方法可以从 CertStore 中提取证书吊销清单，凡是满足方法参数中指定的规则的证书都将被提取出来。具体步骤如下：

(1) 获得 X509CRL 的列表

```

CertificateFactory cf = CertificateFactory.getInstance("X.509");
List mylist = new ArrayList();
for(int i=0;i<args.length;i++){
    FileInputStream in=new FileInputStream(args[i]);
    X509CRL crl = (X509CRL)cf.generateCRL(in);
    mylist.add(crl);
}

```

分析：和 6.5.1 小节一样，不妨从集合对象 ArrayList 中创建 CertStore 对象。所不同的是，6.5.1 小节和 6.5.3 小节在 ArrayList 集合中存放的是 X509 证书，而本实例存放的是证书吊销清单（X509CRL 类型的对象）。

(2) 获得 CertStore 对象

```

CertStoreParameters cparam=new CollectionCertStoreParameters(mylist);
CertStore cs=CertStore.getInstance("Collection",cparam);

```

分析：由上一步的存放 X509CRL 对象的集合创建 CollectionCertStoreParameters 对象，进而创建 CertStore 对象。

(3) 定义提取规则

```

X509CRLSelector selec=new X509CRLSelector();
selec.addIssuerName("CN=E-Commerce PKI CA,"+
    "OU=WISeKey Affiliate CA,O=E-Commerce PKI,C=CH");

```

```
String name="CN=VeriSign Class 2 CA - Individual Subscriber,"
+"OU=www.verisign.com/repository/RPA Incorp."+
" By Ref.\,LIAB.LTD(c)98,"
+"OU=VeriSign Trust Network,"
+"O=VeriSign\, Inc.";
selec.addIssuerName(name);
```

分析: 和 6.5.3 小节类似, 定义各种从 CertStore 对象中提取证书的规则。但这里使用的是 X509CRLSelector 类, 相应的方法也不一样, 这里不妨使用 addIssuerName() 方法指定提取的证书吊销清单必须是某个主体签发的。addIssuerName() 方法可以执行多次, CertStore 对象中的 X509CRL 对象只要满足 addIssuerName() 中的一个条件即被提取出来。

注意, 如果根据 RFC2253 的规定, 主体的名字中各部分使用逗号 (,) 分隔, 本实例使用的 Class3Commercial.crl 文件是由 “CN = VeriSign Class 3 CA - Commercial Content/Software Publisher, OU = www.verisign.com/repository/RPA Incorp. by Ref., LIAB.LTD(c)98, OU = VeriSign Trust Network, O = VeriSign, Inc.” 签发的, 其中的 “by Ref., LIAB.LTD(c)98” 和 “VeriSign, Inc.” 自身包含逗号, 因此应该根据 RFC2253 的规定应使用 “\” 符号转义表示, 即表示成 “by Ref.\, LIAB.LTD(c)98” 和 “VeriSign\, Inc.”。又由于 Java 中在字符串中表示 “\” 需要使用 “\\”, 因此程序中表示成 “by Ref.\\, LIAB.LTD(c)98” 和 “VeriSign\\, Inc.”

(4) 提取证书

```
Set clct=(Set) cs.getCRLs(selec);
```

分析: 执行 CertStore 对象的 getCRLs() 方法, 方法参数中传入上一步定义的规则, 方法返回 Collection 类型的对象, 这里可以将其强制转换为 Set 类型, 该集合中包含了 CertStore 中所有满足条件的证书吊销清单 X509CRL 对象。

(5) 处理证书

```
o=clct.toArray();
System.out.println("Find "+o.length);
for(int i=0;i<o.length;i++){
    X509CRL crl = (X509CRL)o[i];
    System.out.println("issuer = "+crl.getIssuerDN().getName());
}
```

分析: 不妨执行上一步得到的 Set 对象的 toArray() 方法得到包含所有证书吊销清单的数组, 然后通过数组访问各个证书吊销清单。这里不妨将各个证书吊销清单的签发者打印出来。

★代码与分析:

完整代码如下, 该程序虽然看起来较长, 其实是类似的代码执行了三次, 以对比不同场合的执行效果。

```
import java.io.*;
import java.math.*;
import java.util.*;
import java.security.*;
import java.security.cert.*;
import java.security.cert.Certificate;
```

```

public class CRLSelector{
    public static void main(String args[ ]) throws Exception{
        //创建 CertStore 对象
        CertificateFactory cf = CertificateFactory.getInstance("X.509");
        List mylist = new ArrayList();
        for(int i=0;i<args.length;i++){
            FileInputStream in=new FileInputStream(args[i]);
            X509CRL crl = (X509CRL)cf.generateCRL(in);
            mylist.add(crl);
        }
        CertStoreParameters cparam=new CollectionCertStoreParameters(mylist);
        CertStore cs=CertStore.getInstance("Collection",cparam);
        //创建 Selector
        X509CRLSelector selec=new X509CRLSelector();
        //设置规则, 注意逗号后面不可有空格
        selec.addIssuerName("CN=E-Commerce PKI CA,"+
            "OU=WiSeKey Affiliate CA,O=E-Commerce PKI,C=CH");
        //提取证书吊销清单
        Set clct=(Set) cs.getCRLs(selec);
        Object o[]=clct.toArray();
        System.out.println("Find "+o.length);
        for(int i=0;i<o.length;i++){
            X509CRL crl = (X509CRL)o[i];
            System.out.println("issuer = "+crl.getIssuerDN().getName( ));
        }
        //设置规则, 注意转义符号
        String name="CN=VeriSign Class 2 CA - Individual Subscriber,"
            +"OU=www.verisign.com/repository/RPA Incorp."
            +" By Ref.\\,LIAB.LTD(c)98," //By 前面有个空格
            +"OU=VeriSign Trust Network,"
            +"O=VeriSign\\, Inc.";
        selec.addIssuerName(name);
        //提取证书吊销清单
        clct=(Set) cs.getCRLs(selec);
        o=clct.toArray();
        System.out.println("Find "+o.length);
        for(int i=0;i<o.length;i++){
            X509CRL crl = (X509CRL)o[i];
            System.out.println("issuer = "+crl.getIssuerDN().getName( ));
        }
        //设置规则, 注意转义符号
        selec.setIssuerNames(null);
        //提取证书吊销清单
    }
}

```

```

        clct=(Set) cs.getCRLs(selec);

        o=clct.toArray();
        System.out.println("Find "+o.length);
        for(int i=0;i<o.length;i++){
            X509CRL crl = (X509CRL)o[i];
            System.out.println("issuer = "+crl.getIssuerDN().getName());
        }
    }
}

```

程序中首先提取由“CN=E-Commerce PKI CA, OU=WiSeKey Affiliate CA, O=E-Commerce PKI, C=CH”签发的证书吊销清单, 然后提取由“CN=E-Commerce PKI CA, OU=WiSeKey Affiliate CA, O=E-Commerce PKI, C=CH”或者“CN=VeriSign Class 2 CA - Individual Subscriber, OU="www.verisign.com/repository/RPA Incorp. By Ref., LIAB.LTD(c)98, OU=VeriSign Trust Network, O="VeriSign, Inc.”签发的证书吊销清单, 最后将addIssuerName()的参数设置为 null, 即提取无论谁签发的证书吊销清单都提取出来。

★运行程序

输入“java CRLSelector ecpci.crl NewClass2Individual.crl Class3Commercial.crl”运行程序, 屏幕输出如下:

```

C:\java\ch6\Cstore>java      CRLSelector      ecpci.crl      NewClass2Individual.crl
Class3Commercial.crl
Find 1
issuer = CN=E-Commerce PKI CA, OU=WiSeKey Affiliate CA, O=E-Commerce PKI, C=CH
Find 2
issuer = CN=E-Commerce PKI CA, OU=WiSeKey Affiliate CA, O=E-Commerce PKI, C=CH
issuer = CN=VeriSign Class 2 CA - Individual Subscriber, OU="www.verisign.com/repository/RPA
Incorp. By Ref., LIAB.LTD(c)98", OU=VeriSign Trust Network, O="VeriSign, Inc. "
Find 3
issuer = CN=VeriSign Class 3 CA - Commercial Content/Software Publisher,
OU="www.verisign.com/repository/RPA Incorp. by Ref., LIAB.LTD(c)98", OU=VeriSign Trust
Network, O="VeriSign, Inc. "
issuer = CN=E-Commerce PKI CA, OU=WiSeKey Affiliate CA, O=E-Commerce PKI, C=CH
issuer = CN=VeriSign Class 2 CA - Individual Subscriber, OU="www.verisign.com/repository/RPA
Incorp. By Ref., LIAB.LTD(c)98", OU=VeriSign Trust Network, O="VeriSign, Inc. "

```

其中第一次提取到一个证书吊销清单, 第二次提取到两个证书吊销清单, 第三次则所有的证书吊销清单都提取了出来。这是因为 Class3Commercial.crl 既不是“CN=E-Commerce PKI CA, OU=WiSeKey Affiliate CA, O=E-Commerce PKI, C=CH”签发的, 也不是“CN=VeriSign Class 2 CA - Individual Subscriber, OU="www.verisign.com/repository/RPA Incorp. By Ref., LIAB.LTD(c)98, OU=VeriSign Trust Network, O="VeriSign, Inc.”签发的。

本章介绍了证书链的创建、提取、显示、保存、验证和吊销等。在下一章介绍的 **SSL** 和 **HTTPS** 中，将进一步使用证书来建立用户对服务器的信任和服务器对用户的信任。

第 7 章 数据的安全传输和身份验证

——SSL 和 HTTPS 编程

本章重点:

本章在前面几章介绍的加密和认证技术的基础上,介绍如何使用 SSL 协议加密 TCP/IP 数据流,并介绍基于 SSL 的、用于加密浏览器和 Web 服务器之间通信的 HTTPS 协议。

SSL 和 HTTPS 不仅可以加密通信,而且可以用于服务器和客户身份的验证。用户浏览器访问一个站点,需要确定这个站点确实是某个机构的(说不定黑客已经攻击了你使用的域名服务器,将你导向了黑客伪装的一个站点)。服务器在某些时候也可能需要确定用户是谁,以便决定是否向其提供某类信息。本章对此作了介绍。

本章主要内容:

- 编制 SSL 客户和服务程序
- 编制 HTTPS 客户和服务程序
- 设置服务器所使用的证书
- 设置客户程序信任的证书
- 设置客户程序所使用的证书
- 设置服务器信任的证书

7.1 最简单的 SSL 通信

SSL 编程使用客户机/服务器模式,二者之间的通信使用 SSL 协议进行加密。本节先通过最简单的程序介绍服务器和客户程序之间如何通过 SSL 进行加密通信。

7.1.1 最简单的 SSL 服务器

★ 实例说明

本实例编写了一个最简单的 SSL 服务器程序,它接受客户程序建立连接,并以加密方式向客户程序发送一串字符 Hi。

SSL 服务器程序运行时需要指定密钥库,以便向客户程序证明自己的身份。本实例演示了通过编程指定密钥库和通过 java 命令选项指定密钥库的两种运行方式。

★ 编程思路:

SSL 编程和基于 Socket 的编程类似,首先创建 ServerSocket 对象,传入端口号,然后执行 ServerSocket 对象的 accept()方法获取 Socket 类型的对象,并侦听端口以等待客户程序

和服务端连接。最后通过 `Socket` 类型的对象获得输入和输出流，通过输入和输出流和客户程序进行通信。`SSL` 编程和基于 `Socket` 的编程不同的地方在于其 `ServerSocket` 对象是通过一个特殊的对象：`SSLServerSocketFactory` 类型的对象创建的，这样以后的输入和输出流将自动按照 `SSL` 协议指定的方法交换密钥并对数据进行加密。此外，需要指定包含证书的密钥库，以便客户程序确定 `SSL` 服务器是否可靠。

具体步骤如下：

(7) 设置密钥库及口令

```
System.setProperty("javax.net.ssl.keyStore",
                    "mykeystore");
System.setProperty("javax.net.ssl.keyStorePassword",
                    "wshr.ut");
```

分析：通过 `System` 类的静态方法 `setProperty()` 可以设置系统参数。方法的第一个参数是系统参数的名称，第二个参数是为系统参数设置的值。作为 `SSL` 服务器程序，主要需要设置两个系统参数：`javax.net.ssl.keyStore` 指定密钥库的名称，`javax.net.ssl.keyStorePassword` 指定密钥库的密码。

这里不妨使用 5.1 节得到的密钥库 `mykeystore`，其密码为 `wshr.ut`。密钥库中必须存放私钥和证书，此外为私钥设置的密码应该和密钥库的密码相同。程序将自动从密钥库中提取证书。

(8) 创建 `SSLServerSocketFactory` 类型的对象

```
SSLServerSocketFactory ssf= (SSLServerSocketFactory)
                             SSLServerSocketFactory.getDefault();
```

分析：执行 `javax.net.ssl` 包中 `SSLServerSocketFactory` 类的静态方法 `getDefault()`，经过强制转换获得 `SSLServerSocketFactory` 类型的对象，后面将用它获取 `ServerSocket` 对象。

(9) 创建 `ServerSocket` 类型的对象

```
ServerSocket ss=ssf.createServerSocket(5432);
```

分析：执行上一步得到的 `SSLServerSocketFactory` 对象的 `createServerSocket()` 方法获得 `ServerSocket` 类型的对象，方法参数中指定一个整数作为端口号，其值一般在 `1~65535` 之间，其中 `1~1023` 一般用于知名的端口号或特定的 `UNIX` 服务，临时使用的端口号可取 `1024~65535` 之间的整数。

一台计算机上往往会运行不同的服务程序提供不同的服务，这些程序应使用不同的端口号，这样，当服务器收到客户程序发来的请求时，通过端口号确定哪个服务器程序与之通信。

(10) 等待客户程序连接

```
Socket s=ss.accept();
```

分析：执行上一步得到的 `ServerSocket` 对象的 `accept()` 方法，程序将在此处挂起，等待客户程序建立连接。该方法返回的 `Socket` 类型的对象可用于和客户程序之间的通信。

(11) 建立输出流

```
PrintStream out = new PrintStream(s.getOutputStream());
out.println("Hi");
```

分析：执行上一步得到的 `Socket` 对象的 `getOutputStream()` 方法可以得到输出流，通过该输出流发送的信息将加密传递给客户程序。这里不妨使用输出流创建 `PrintStream` 类型的对象，以便通过 `println()` 语句向客户程序打印字符串。

如果服务器程序同时需要处理客户程序发来的字符串，可以通过 Socket 对象的 `getInputStream()` 方法得到输入流，从输入流读取的信息即客户发来的信息。

★代码与分析:

完整代码如下:

```
import java.net.*;
import java.io.*;
import javax.net.ssl.*;

public class MySSLServer{
    public static void main(String args[ ]) throws Exception{
        System.setProperty("javax.net.ssl.keyStore",
                           "mykeystore");
        System.setProperty("javax.net.ssl.keyStorePassword",
                           "wshr.ut");
        SSLServerSocketFactory ssf=(SSLServerSocketFactory)
            SSLServerSocketFactory.getDefault();
        ServerSocket ss=ssf.createServerSocket(5432);
        System.out.println("Waiting for connection...");
        while(true) {
            Socket s=ss.accept();
            PrintStream out = new PrintStream(s.getOutputStream());
            out.println("Hi");
            out.close();
            s.close();
        }
    }
}
```

为了让程序接受到一个连接请求并发送完“Hi”后能继续接受其他客户程序建立连接，程序中将 `Socket s=ss.accept()` 及其输入/输出处理放在了一个 `while` 循环当中。

★运行程序

由于 SSL 协议需要通过数字证书向客户表明服务器是否值得信任，因此当前目录下必须有密钥库，本实例不妨使用 5.1 节得到的密钥库 `mykeystore`，将其拷贝到当前目录下，然后输入“`java MySSLServer`”运行程序，当等待一段时间完成初始化后，屏幕显示：“Waiting for connection...” ，此时开始等待客户程序的连接。

编程第 1 步设置系统参数也可以不在程序中指定，而是通过 `java` 命令选项来指定。例如如果省略了编程第 1 步，则可输入“`java -Djavax.net.ssl.keyStore=mykeystore -Djavax.net.ssl.keyStorePassword=wshr.ut MySSLServer`”来运行程序，这样程序本身更具有灵活性。

为了使客户程序能够顺利验证该服务器提供的证书，应该把 `mykeystore` 中所使用的证书或其签发者的证书提供给客户程序。这在下一小节“运行程序”部分将详细说明。

7.1.2 最简单的 SSL 客户程序

★ 实例说明

本实例编写了一个最简单的 SSL 客户程序,它和运行 7.1.1 小节程序的计算机建立连接,接受其发来的字符串并自动对其进行解密。本实例同时演示了通过程序指定密钥库和通过 java 命令选项指定密钥库的两种运行方式。

★ 编程思路:

SSL 客户端的编程也和基于 Socket 的客户端编程类似。首先得到 Socket 类型的对象,然后通过 Socket 类型的对象获得输入和输出流,通过输入和输出流和服务程序进行通信。和服务程序类似,SSL 客户端编程和基于 Socket 的客户端编程不同的地方在于其 Socket 对象是通过一个特殊的对象:SSLSocketFactory 类型的对象创建的。

具体步骤如下:

- (1) 设置客户程序信任的密钥库

```
System.setProperty("javax.net.ssl.trustStore",  
                    "clienttrust");
```

分析:客户端欲和 SSL 服务器通信,则必须信任 SSL 服务器程序所使用的数字证书。因此客户程序应该将所信任的证书放在一个密钥库中(本实例“运行程序”部分给出了如何创建这样的密钥库)。这里不妨假定客户程序信任的证书放在文件名为 clienttrust 的密钥库中。

通过 System 类的静态方法 setProperty() 可以设置系统参数 javax.net.ssl.trustStore,可以在程序中指定该文件名。由于 clienttrust 中存放的只是可以公开的证书,因此程序中不需要给出密钥库的密码。

- (2) 创建 SSLSocketFactory 类型的对象

```
SSLSocketFactory ssf= (SSLSocketFactory)  
    SSLSocketFactory.getDefault();
```

分析:执行 javax.net.ssl 包中 SSLSocketFactory 类的静态方法 getDefault(),经过强制转换获得 SSLSocketFactory 类型的对象,后面将用它获取 Socket 对象。

- (3) 创建 Socket 类型的对象,连接服务器程序

```
Socket s = ssf.createSocket("127.0.0.1", 5432);
```

分析:执行上一步得到的 SSLSocketFactory 对象的 createSocket() 方法和服务器指定端口建立连接。方法的第一个参数是字符串形式的服务器 IP 地址或域名,如果只有一台计算机,客户和服务程序都在同一台计算机上运行,则可以使用“127.0.0.1”作为服务器的 IP 地址,或“localhost”作为服务器的域名。第二个参数即 7.1.1 小节的服务器程序在第 3 步指定的端口号。

- (4) 建立输出流

```
BufferedReader in = new BufferedReader(  
    new InputStreamReader(s.getInputStream()));  
  
String x=in.readLine();
```

分析:执行上一步得到的 Socket 对象的 getInputStream() 方法可以得到输入流,通过该输入流读取服务器程序发送来的信息并自动解密。这里不妨使用输入流创建 BufferedReader 类型的对象,以便通过 readLine() 语句读取字符串。

如果客户程序同时需要向服务器程序发送信息,可以再通过 Socket 对象的 getOutputStream() 方法得到输出流,通过输出流发送的信息可以被服务器程序的输入流读取到。

★代码与分析:

完整代码如下:

```

import java.net.*;
import java.io.*;
import javax.net.ssl.*;
public class MySSLClient{
    public static void main(String args[ ]) throws Exception {
        System.setProperty("javax.net.ssl.trustStore",
            "clienttrust");
        SSLSocketFactory ssf=
            (SSLSocketFactory) SSLSocketFactory.getDefault( );
        Socket s = ssf.createSocket("127.0.0.1", 5432);
        BufferedReader in = new BufferedReader(
            new InputStreamReader(s.getInputStream( )));
        String x=in.readLine( );
        System.out.println(x);
        in.close( );
    }
}

```

★运行程序

服务器程序使用的密钥库是 5.1.3 小节得到的密钥库 **mykeystore**，假定已经用 5.2.3 小节的方法得到了证书文件 **mytest.cer**。将该文件存放在客户程序所在目录中，以便客户程序向服务器程序确认信任该证书。为了在程序中使用，需将该证书导入密钥库，操作如下：

```
C:\java\ch7\Client>keytool -import -alias mytest -file mytest.cer -keystore
clienttrust
```

输入 keystore 密码: 123456

Owner: CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai, C=CN

发照者: CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai, C=CN

序号: 3deec043

有效期间: Thu Dec 05 10:56:03 CST 2002 至: Sun Nov 17 10:56:03 CST 2013

认证指纹:

MD5: B2: DC: 75: CD: 60: B7: 1E: 7A: 97: EE: E8: A4: 31: D6: 26: C6

SHA1: 32: E5: 89: 16: 7E: 25: 7F: 86: 16: 94: 34: 36: 95: 44: D7: CF: 14: C8: F2: 1E

信任这个认证? [否]: 是

认证已添加至 keystore 中

该操作将证书 **my.cer** 导入密钥库 **clienttrust**。

运行程序之前检查一下 7.1.1 小节的程序是否已经运行，其 DOS 窗口停留在“Waiting for connection...”提示语句。客户程序可以在同一台计算机上再开设一个 DOS 窗口来运行，也可在另一台联网的计算机上运行，这时程序中的 IP 地址：127.0.0.1 应该改为运行 7.1.1 小节服务器程序所在计算机的实际 IP 地址。

在 DOS 窗口输入“java MySSLClient”运行客户程序，程序将显示服务器程序发来的

“Hi”。如果用抓包软件捕捉客户程序和服务器程序之间的通信，可以发现通信内容是以密文传递的。

和服务器程序一样，编程第 1 步设置系统参数也可以不在程序中指定，而是通过 java 命令选项来指定。例如如果省略了编程第 1 步，则可输入“java -Djavax.net.ssl.trustStore=clienttrust MySSLClient”来运行程序，这样程序本身更具有灵活性。

7.1.3 进一步设置信任关系

★ 实例说明

7.1.1 和 7.1.2 小节的例子中使用的密钥库 mykeystore 和证书 mytest.cer 是自签名的证书，本实例的服务器程序使用 6.1.1 小节得到的密钥库 lfkeystore2 中的证书“Liu Fang”，该证书是 CA “Xu Yingxiao”签发的，而客户程序不是直接信任信任证书“Liu Fang”，而是信任 CA “Xu Yingxiao”的证书。

本实例同时演示了通过 java 命令选项来指定密钥库及密码。

★ 编程思路：

服务器程序和 7.1.1 小节类似，只是密钥库使用 lfkeystore2 即可，密钥库的密码仍旧为 wshr.ut。服务器使用该密钥库中的证书“Liu Fang”向客户程序表明自己的身份。

7.1.2 小节的客户程序信任的证书为 mytest.cer，即 CA “Xu Yingxiao”的证书，由于 lfkeystore2 中的证书是 CA “Xu Yingxiao”签发的，因此客户程序即使没有直接信任“Liu Fang”的证书，只要信任 CA “Xu Yingxiao”的证书，则自动信任“Liu Fang”的证书。因此客户程序不需要作修改。

★ 代码与分析：

服务器程序完整代码如下：

```
import java.net.*;
import java.io.*;
import javax.net.ssl.*;

public class MySSLServer2 {
    public static void main(String args[ ]) throws Exception {
        SSLServerSocketFactory ssf=
            (SSLServerSocketFactory)
            SSLServerSocketFactory.getDefault();
        ServerSocket ss=ssf.createServerSocket(5432);
        System.out.println("Waiting for connection...");
        while(true) {
            Socket s=ss.accept();
            PrintStream out = new PrintStream(s.getOutputStream());
            out.println("Hi");
            out.close();
            s.close();
        }
    }
}
```

```
}
}
```

这里为了演示通过 java 命令选项来指定密钥库及密码，在程序中删除了 System.setProperty() 语句。

★运行程序

运行服务器程序的目录下存放 6.1.1 小节得到的密钥库文件 lfkeystore2，输入：“java -Djavax.net.ssl.keyStore=lfkeystore2 -Djavax.net.ssl.keyStorePassword=wshr.ut MySSLServer2” 运行程序，和 7.1.1 小节一样，显示 “Waiting for connection...” 提示，等待用户连接。

和 7.1.2 小节一样运行客户程序，尽管该程序没有直接信任服务器所使用的证书，但服务器的证书是客户所信任的证书签发的，因此程序可以正常运行。

7.1.4 设置默认信任密钥库

★ 实例说明

7.1.3 小节的服务器程序使用 CA “Xu Yingxiao” 签发的证书 “Liu Fang”，客户程序在运行时仍需要通过 System.setProperty() 方法或者 Java 命令选项设置客户程序信任什么证书。

本实例使用默认信任密钥库指定客户程序信任哪些证书。

★ 编程思路：

服务器程序使用 7.1.3 小节的程序，客户程序只要信任证书 “Liu Fang” 或者其签发者 CA “Xu Yingxiao” 的证书即可。

7.1.2 小节的客户程序通过 System.setProperty() 方法或者 Java 命令选项设置了系统参数：javax.net.ssl.trustStore，指定了客户程序信任哪些证书。

如果使用默认信任密钥库，则不需要在程序或 Java 命令中指定系统参数，因此只要将 7.1.2 小节的程序中 System.setProperty() 语句去掉即可。

Java 默认的信任密钥库是 C:\jdk1.4.0\jre\lib\security 目录下的 cacerts 文件，使用 J2SDK 提供的 keytool 工具可以将客户信任的证书导入该密钥库，则 Java 程序自动信任这些证书对应的 CA 签发的证书。

在默认信任密钥库中已经存有一些著名 CA 的证书，如果服务器程序所使用的证书是这些 CA 签发的，则不需要修改默认信任密钥库。

★代码与分析：

完整代码如下：

```
import java.net.*;
import java.io.*;
import javax.net.ssl.*;

public class MySSLClient2 {
    public static void main(String args[ ]) throws Exception {
        SSLSocketFactory ssf=
            (SSLSocketFactory) SSLSocketFactory.getDefault();
```

```

        Socket s = ssf.createSocket("127.0.0.1", 5432);
        BufferedReader in
            = new BufferedReader(
                new InputStreamReader(s.getInputStream()));
        String x=in.readLine();
        System.out.println(x);
        in.close();

    }
}

```

★运行程序

由于使用默认信任密钥库，因此运行程序时不需要 Java 命令选项，只要输入“java MySSLClient2”运行程序即可。运行之前先检查 7.1.3 小节的服务器程序确认已经在运行，并停留在“Waiting for connection...”提示等待客户程序连接。

MySSLClient2 运行将出现如下出错信息：

```

Exception in thread "main" javax.net.ssl.SSLHandshakeException: Couldn't find trusted
certificate

    at com.sun.net.ssl.internal.ssl.SSLSocketImpl.b(DashoA6275)
    at com.sun.net.ssl.internal.ssl.SSLSocketImpl.a(DashoA6275)
    at com.sun.net.ssl.internal.ssl.ClientHandshaker.a(DashoA6275)
    at com.sun.net.ssl.internal.ssl.ClientHandshaker.processMessage(DashoA62
75)

    at com.sun.net.ssl.internal.ssl.Handshaker.process_record(DashoA6275)
    at com.sun.net.ssl.internal.ssl.SSLSocketImpl.a(DashoA6275)
    at com.sun.net.ssl.internal.ssl.SSLSocketImpl.a(DashoA6275)
    at com.sun.net.ssl.internal.ssl.AppInputStream.read(DashoA6275)
    at java.io.InputStream.read(InputStream.java:88)
    at sun.nio.cs.StreamDecoder$ConverterSD.implRead(StreamDecoder.java:282)

    at sun.nio.cs.StreamDecoder.read(StreamDecoder.java:179)
    at java.io.InputStreamReader.read(InputStreamReader.java:167)
    at java.io.BufferedReader.fill(BufferedReader.java:136)
    at java.io.BufferedReader.readLine(BufferedReader.java:299)
    at java.io.BufferedReader.readLine(BufferedReader.java:362)
    at MySSLClient2.main(MySSLClient2.java:12)

```

客户程序既没有在程序中通过 `System.setProperty()` 方法、也没有在运行时通过 Java 命令选项指明客户程序信任哪个密钥库中的证书，因此运行时将使用默认信任密钥库 `C:\jdk1.4.0\jre\lib\security\cacerts` 检查客户程序是否信任服务器程序提供的证书。而在默认信任密钥库中既没有服务器提供的“Liu Fang”证书，也没有签发“Liu Fang”证书的 CA “Xu Yingxiao”的证书。所以程序抛出异常 `SSLHandshakeException`，提示无法找到信任的证书。

使用 **keytool** 工具可以看一下默认密钥库信任哪些证书:

```
C:\>keytool -list -keystore C:\j2sdk1.4.0\jre\lib\security\cacerts
```

输入 keystore 密码: changeit

您的 keystore 包含 10 输入

```
thawtepersonalfreemailca, 1999-2-13, trustedCertEntry,
认证指纹 (MD5): 1E: 74: C3: 86: 3C: 0C: 35: C5: 3E: C2: 7F: EF: 3C: AA: 3C: D9
thawtepersonalbasicca, 1999-2-13, trustedCertEntry,
认证指纹 (MD5): E6: 0B: D2: C9: CA: 2D: 88: DB: 1A: 71: 0E: 4B: 78: EB: 02: 41
verisignclass3ca, 1998-6-30, trustedCertEntry,
认证指纹 (MD5): 78: 2A: 02: DF: DB: 2E: 14: D5: A7: 5F: 0A: DF: B6: 8E: 9C: 5D
thawtepersonalpremiumca, 1999-2-13, trustedCertEntry,
认证指纹 (MD5): 3A: B2: DE: 22: 9A: 20: 93: 49: F9: ED: C8: D2: 8A: E7: 68: 0D
thawteserverca, 1999-2-13, trustedCertEntry,
认证指纹 (MD5): C5: 70: C4: A2: ED: 53: 78: 0C: C8: 10: 53: 81: 64: CB: D0: 1D
verisignclass4ca, 1998-6-30, trustedCertEntry,
认证指纹 (MD5): 1B: D1: AD: 17: 8B: 7F: 22: 13: 24: F5: 26: E2: 5D: 4E: B9: 10
verisignserverca, 1998-6-30, trustedCertEntry,
认证指纹 (MD5): 74: 7B: 82: 03: 43: F0: 00: 9E: 6B: B3: EC: 47: BF: 85: A5: 93
verisignclass1ca, 1998-6-30, trustedCertEntry,
认证指纹 (MD5): 51: 86: E8: 1F: BC: B1: C3: 71: B5: 18: 10: DB: 5F: DC: F6: 20
thawtepremiumserverca, 1999-2-13, trustedCertEntry,
认证指纹 (MD5): 06: 9F: 69: 79: 16: 66: 90: 02: 1B: 8C: 8C: A2: C3: 07: 6F: 3A
verisignclass2ca, 1998-6-30, trustedCertEntry,
认证指纹 (MD5): EC: 40: 7D: 2B: 76: 52: 67: 05: 2C: EA: F2: 3A: 4F: 65: F0: D8
```

默认密钥库的初始密码是 **changeit**, 需要时可以修改密码。和 7.1.2 小节的 **clienttrust** 密钥库类似, 其中存放的只有证书没有对应的私钥, 所以每个证书的名字后面显示的是 “**trustedCertEntry**” 而不是 **keyEntry**。另外 7.1.2 小节是通过 **System.setProperty()** 方法、或 Java 命令选项指定密钥库 **clienttrust** 的, 而默认密钥库不需要指定。

如果服务器程序使用的证书是这些证书对应的私钥所签发的, 则本实例的程序可以直接运行。7.1.3 小节服务器程序使用的证书是 CA “**Xu Yingxiao**” 签发的, 在默认密钥库中不存在其证书, 因此需要将 CA “**Xu Yingxiao**” 的证书导入该默认密钥库。这里可以使用 7.1.2 小节所使用的密钥文件 **mytest.cer**, 执行如下命令即可。执行命令前可将 **cacerts** 文件备份一下。

```
C:\java\ch7\Client>keytool -import -keystore C:\j2sdk1.4.0\jre\lib\security\cacerts
-file mytest.cer -alias mytest
```

输入 keystore 密码: changeit

Owner: CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai, C=CN

发行者: CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai, C=CN

序号: 3deec043

有效期间: Thu Dec 05 10:56:03 CST 2002 至: Sun Nov 17 10:56:03 CST 2013

认证指纹:

MD5: B2: DC: 75: CD: 60: B7: 1E: 7A: 97: EE: E8: A4: 31: D6: 26: C6

SHA1: 32: E5: 89: 16: 7E: 25: 7F: 86: 16: 94: 34: 36: 95: 44: D7: CF: 14: C8: F2: 1E

信任这个认证? [否]: 是

认证已添加至 keystore 中

这样, 只要是 mytest.cer 证书对应的私钥签发的证书都将自动被信任。如在 7.1.3 小节的服务器程序已经启动的前提下直接输入“java MySSLClient2”运行程序, 则程序将显示“Hi”。

试验完毕可将备份的 cacerts 恢复, 或执行 `keytool -delete -alias mytest -keystore C:\jdk1.4.0\jre\lib\security\cacerts -storepass changeit` 删除添加的证书。

7.1.5 通过 KeyStore 对象选择密钥库

★ 实例说明

除了通过 `System.setProperty()` 方法或者 Java 命令选项指定密钥库及其密码外, 还可以在程序中通过 `KeyStore` 对象指定密钥库及密码。

在前面各小节的例子中, 保护密钥库的密码和各个条目中保护私钥的密码必须相同。使用 `KeyStore` 对象指定密钥库及密码时, 两种密码可以不同。

★ 编程思路:

可对前面各个小节的例子中创建 `SSLServerSocketFactory` 类型对象的方法作些修改, 不再通过 `SSLServerSocketFactory` 类的静态方法 `getDefault()`, 而通过 `SSLContext` 类的 `getServerSocketFactory()` 方法获得 `SSLServerSocketFactory` 类型对象, 进而获得 `ServerSocket` 对象。

在 `SSLContext` 类的初始化过程中, 可以传入包含密钥库、密钥库口令、私钥口令等信息的 `KeyManagerFactory` 对象。具体编程步骤如下:

(1) 获取 `SSLContext` 对象

```
SSLContext context=SSLContext.getInstance("TLS");
```

分析: 通过 `SSLContext` 类的 `getInstance()` 方法获得 `SSLContext` 类型的对象, 方法的参数中指定协议类型, 可以是 SSL 或其低层的 TLS 等。该步骤得到的 `SSLContext` 对象实现了参数中指定的协议。

(2) 获取 `KeyManagerFactory` 对象

```
KeyManagerFactory kmf=KeyManagerFactory.getInstance("SunX509");
```

分析: 通过 `KeyManagerFactory` 类的 `getInstance()` 方法获得 `KeyManagerFactory` 类型的对象, 方法的参数中指定算法。该步骤得到的 `KeyManagerFactory` 对象实现了参数中指定的密钥管理算法。

(3) 获取 `KeyStore` 对象

```
FileInputStream fin=new FileInputStream(storename);
```

```
ks=KeyStore.getInstance("JKS");
```

```
ks.load(fin,storepass);
```

分析: 和 5.2.7 小节一样, 通过 `KeyStore` 类的静态方法 `getInstance()` 获得 `KeyStore` 对象, 执行其 `load()` 方法加载密钥库, 方法的参数指定密钥库的文件输入流和保护密钥库的密码。

(4) 初始化 `KeyManagerFactory` 对象

```
kmf.init(ks,keypass);
```

分析：执行第二步得到的 KeyManagerFactory 对象的 init() 方法，方法参数传入上一步得到的代表密钥库的 KeyStore 对象和提取其中的私钥所需要的密码。

(5) 初始化 SSLContext 对象

```
context.init(kmf.getKeyManagers(),null,null);
```

分析：执行第一步得到的 SSLContext 对象的 init() 方法，方法的第一个参数传入上一步得到的 KeyManagerFactory 对象，其他两个参数暂且设置为 null。在后面的内容中将进一步介绍。

(6) 创建 SSLServerSocketFactory 对象

```
SSLServerSocketFactory ssf= context.getServerSocketFactory();
```

分析：执行第一步得到的 SSLContext 对象的 getServerSocketFactory() 方法，它将创建 SSLServerSocketFactory 对象，可进而创建 ServerSocket 对象。

(7) 创建 ServerSocket 对象

```
ServerSocket ss=ssf.createServerSocket(5432);
```

分析：执行上一步得到的 ServerSocket 对象的 createServerSocket() 方法，它将创建 ServerSocket 对象，方法的参数指定端口号。从这里往后的编程就和 7.1.1、7.1.3 完全相同了。

★代码与分析：

完整代码如下：

```
import java.net.*;
import java.io.*;
import javax.net.ssl.*;
import java.security.*;

public class MySSLServerKs {
    public static void main(String args[] )throws Exception {
        SSLContext context;
        KeyManagerFactory kmf;
        KeyStore ks;
        char[] storepass="newpass".toCharArray();
        char[] keypass="wshr.ut".toCharArray();
        String storename="lfnewstore";
        context=SSLContext.getInstance("TLS");
        kmf=KeyManagerFactory.getInstance("SunX509");
        FileInputStream fin=new FileInputStream(storename);
        ks=KeyStore.getInstance("JKS");
        ks.load(fin, storepass);

        kmf.init(ks, keypass);
        context.init(kmf.getKeyManagers(), null, null);
        SSLServerSocketFactory ssf= context.getServerSocketFactory();

        ServerSocket ss=ssf.createServerSocket(5432);
```

```

        System.out.println("Waiting for connection...");
        while(true) {
            Socket s=ss.accept( );
            PrintStream out = new PrintStream(s.getOutputStream( ));
            out.println("Hi");
            out.close( );
            s.close( );
        }
    }
}

```

这里不妨使用 6.1.2 小节得到的密钥库 `lfnewstore`, 在 6.1.2 小节中, 该密钥库的保护口令设置为 `newpass`, 其中的私钥的保护口令为 `wshr.ut`。

★运行程序

在 6.1.2 小节中, 密钥库 `lfnewstore` 中有两个条目 `lf` 和 `lf_signed`, 如何在选择条目将在本章后面介绍, 这里为了简化, 不妨删除其中的 `lf` 条目。将原有 `lfnewstore` 备份后, 执行 `keytool -delete -alias lf -storepass newpass -keystore lfnewstore` 则 `lfnewstore` 中将只有一个条目: `lf_signed`, 该条目对应的证书“Liu Fang”是由 CA “Xu Yingxiao”签发的。

输入“`java MySSLServerKs`”运行程序, 屏幕出现“”提示后, 可再打开一个 DOS 窗口, 执行 7.1.2 或 7.1.3、7.1.4 的客户程序。

7.2 进一步的 SSL 客户和服务程序例子

本节在 7.1 节最简单的例子的基础上给出进一步的例子, 包括客户机/服务器的双向通信、查看对方的证书等。

7.2.1 设计通信规则

由 7.1 节的例子我们已经可以在客户机、服务器程序之间以加密方式交换信息, 剩下问题就是客户机和服务器程序在处理输入和输出时按照什么规则进行。这主要取决于具体应用的要求。

★ 实例说明

本实例使用一个简单的规则编写了客户机/服务器双向通信的程序, 服务器按照不同的客户请求发送不同文件到客户程序, 客户程序根据服务器发来的标题的不同按照不同方式保存文件。

★ 编程思路:

不妨制订如下简单的通信规则: 服务器收到客户发来的信息后, 如果发现是“.html”结尾, 则向客户程序先发送一串信息:“Sending HTML”, 再发送一串 HTML 文本, 发送完

毕后发送“Session Over”字符串结束会话。如果发现是“.gif”，则向客户程序先发送一串信息：“Sending GIF”，再发送一个图片文件。

客户机和服务器建立连接后，向服务器发送请求字符串，然后读取服务器反馈信息。若收到“Sending HTML”字符串则建立 HTML 为后缀的文件，若收到“Sending GIF”字符串则建立.gif 为后缀的文件，然后继续读取服务器反馈信息，将读取的内容存入文件。

这样，服务器程序开头部分和 7.1.1 小节一样，获取 Socket 类型的对象 s，其余部分的编程步骤为：

(1) 获取输出流

```
OutputStream outs=s.getOutputStream();
```

```
PrintStream out = new PrintStream(outs);
```

分析：执行 Socket 对象的 getOutputStream() 方法，得到 OutputStream 类型的对象，通过其 write() 方法可以向客户程序发送字节数组。不妨再利用 OutputStream 对象创建 PrintStream 类型的对象，通过其 println() 方法可以向客户程序发送字符串。

(2) 获取输入流

```
BufferedReader in = new BufferedReader(  
    new InputStreamReader(s.getInputStream()));
```

分析：执行 Socket 对象的 getInputStream() 方法，进而创建 BufferedReader 类型的对象，通过其 readln() 方法可以读取从客户程序发来的字符串。

(3) 读取客户发来的字符串

```
String line=in.readLine()
```

分析：执行输入流的 readLine() 方法。

(4) 判断客户发来的字符串

```
if (line.endsWith(".html")){  
    out.println("Sending HTML");  
    out.println(...);  
}  
else if(line.endsWith(".gif")){  
    out.println("Sending GIF");  
    out.println(...);  
}
```

分析：根据规则，检查字符串是以“.html”还是.gif 结尾，分别向客户程序发送不同的字符串和不同文件。完整的程序可以根据发来的字符串的不同从文件系统或网络中读取对应的文件提供给客户程序。本实例为简洁起见只向客户程序发送固定的内容。

客户程序开头部分和 7.1.2 小节一样，获取 Socket 类型的对象 s，其余部分的编程步骤为：

(1) 获取输出流

```
OutputStream outs=s.getOutputStream();
```

```
PrintStream out = new PrintStream(outs);
```

分析：和服务器程序一样。

(2) 获取输入流

```
InputStream ins = s.getInputStream();
BufferedReader in = new BufferedReader(
    new InputStreamReader(ins));
```

分析：和服务程序一样。

(3) 向服务器发送字符串

```
out.println(args[0]);
```

分析：执行输出流的 `println()` 方法。不妨从命令行参数读取相应的字符串，该字符串中可以指定要读取什么样的内容。只要客户程序和服务器程序编程之前约定好了字符串的格式和含义，字符串可以是任意格式。

(4) 接收服务器反馈信息

```
String x=in.readLine();
```

分析：执行输入流的 `readLine()` 方法。

(5) 根据服务器的反馈信息打开不同的文件输出流

```
if( x.equals("Sending HTML")){
    fouts=new FileOutputStream("result.html");
}
else if( x.equals("Sending GIF")){
    fouts=new FileOutputStream("result.gif");
}
```

分析：如果服务器发来“Sending HTML”，则创建文件名以.html 为后缀的文件；如果服务器发来“Sending GIF”，则创建文件名以.gif 为后缀的文件。

(6) 接收服务器进一步的反馈信息

```
while((kk=ins.read())!=-1){
    System.out.println(kk);
    fouts.write(kk);
}
```

分析：执行输入流的 `read()` 方法读取服务器发来的信息，将服务器发来的文件内容存入上一步对应的文件。

★代码与分析：

服务器完整代码如下：

```
import java.net.*;
import java.io.*;
import javax.net.ssl.*;

public class MySSLServerRule{
    public static void main(String args[ ]) throws Exception{
        byte[] x={
            (byte) 0x47, (byte) 0x49, (byte) 0x46, (byte) 0x38, (byte) 0x39,
            (byte) 0x61, (byte) 0x05, (byte) 0x00, (byte) 0x05, (byte) 0x00,
            (byte) 0x80, (byte) 0xff, (byte) 0x00, (byte) 0xff, (byte) 0xff,
            (byte) 0xff, (byte) 0x00, (byte) 0x00, (byte) 0x00, (byte) 0x2c,
            (byte) 0x00, (byte) 0x00, (byte) 0x00, (byte) 0x00, (byte) 0x05,
            (byte) 0x00, (byte) 0x05, (byte) 0x00, (byte) 0x40, (byte) 0x02,
            (byte) 0x07, (byte) 0x44, (byte) 0x0e, (byte) 0x86, (byte) 0xc7,
```

```

(byte) 0xed, (byte) 0x51, (byte) 0x00, (byte) 0x00, (byte) 0x3b
};
System.setProperty("javax.net.ssl.keyStore", "mykeystore");
System.setProperty("javax.net.ssl.keyStorePassword", "wshr.ut");
SSLServerSocketFactory ssf=
    (SSLServerSocketFactory)
        SSLServerSocketFactory.getDefault();
ServerSocket ss=ssf.createServerSocket(5432);
System.out.println("Waiting for connection...");
while(true) {
    Socket s=ss.accept();
    OutputStream outs=s.getOutputStream();
    PrintStream out = new PrintStream(outs);
    BufferedReader in = new BufferedReader(
        new InputStreamReader(s.getInputStream()));
    String line=in.readLine();
    System.out.println("Got "+line);
    if (line.endsWith(".html")) {
        System.out.println("Now Sending HTML");
        out.println("Sending HTML");
        out.println("<HTML><HEAD><TITLE>Test SSL</TITLE></HEAD>");
        out.println("<BODY><h1> This is a test</BODY>");
        out.println("</HTML>");
    }
    else if (line.endsWith(".gif")) {
        System.out.println("Now Sending GIF");
        out.println("Sending GIF");
        outs.write(x);
        out.println("");
    }
    out.close();
    s.close();
}

}
}

```

其中字节数组 `x` 中存放的是一个图片文件的十六进制代码，该图片很小，显示一个小圆点。如果程序稍改复杂点，可以根据客户程序发来的字符串直接从文件系统或网络中读取所需的 **HTML** 文件或图片。

客户程序完整代码如下：

```

import java.net.*;
import java.io.*;

```

```

import javax.net.ssl.*;

public class MySSLClientRule{
    public static void main(String args[ ]) throws Exception {
        FileOutputStream fouts=null;
        System.setProperty("javax.net.ssl.trustStore",
            "clienttrust");
        SSLSocketFactory ssf=
            (SSLSocketFactory) SSLSocketFactory.getDefault( );
        Socket s = ssf.createSocket("127.0.0.1", 5432);

        OutputStream outs=s.getOutputStream( );
        PrintStream out = new PrintStream(outs);
        InputStream ins = s.getInputStream( );
        BufferedReader in = new BufferedReader(
            new InputStreamReader(ins));

        out.println(args[0]);
        System.out.println("Sent");
        String x=in.readLine( );
        System.out.println(x);
        if( x.equals("Sending HTML")){
            fouts=new FileOutputStream("result.html");
        }
        else if( x.equals("Sending GIF")){
            fouts=new FileOutputStream("result.gif");
        }
        int kk;
        while((kk=ins.read())!=-1){
            fouts.write(kk);
        }
        in.close( );
        fouts.close();
    }
}

```

★运行程序

和 7.1 节的程序一样, 服务器程序使用密钥库 `mykeystore` 中的证书向客户程序证明自己, 客户程序使用密钥库 `clienttrust` 向服务器程序表明自己是否信任服务器程序提供的证书。

输入 “`java MySSLServerRule`” 启动服务器程序, 经过一段较长时间的初始化工作, 屏幕上显示 “Waiting for connection...”, 表明服务器准备就绪。

在计算机中再打开一个 DOS 窗口, 输入 “`java MySSLClientRule http://www/x.gif`” 运行程序, 这里通过命令行参数向服务器提供了字符串 “`http://www/x.gif`”, 要求获取该资源。由于我们的服务器程序作了许多简化, 只检查字符串的后缀, 因此这个字符串可以使用任意格式, 只要最后几个字符为 `.gif` 或 `.html` 即可。如果对服务器程序作些改进, 解析该字符串, 则可以实现更多功能。

经过一段较长时间的初始化工作，客户机和服务器开始了双向通信，服务器程序的 DOS 窗口显示：

```
Got http://www/x.html
```

```
Now Sending HTML
```

客户程序的 DOS 窗口显示：

```
Sent
```

```
Sending GIF
```

在运行客户程序的当前目录下将出现一个文件：**result.gif**，打开该文件将看到服务器程序发来的图片：一个圆点。

如果输入“**java MySSLClientRule http://www/test.html**”运行程序，则服务器的 DOS 窗口将显示：

```
Got http://www/test.html
```

```
Now Sending HTML
```

客户程序的 DOS 窗口显示：

```
Sent
```

```
Sending HTML
```

在运行客户程序的当前目录下将出现一个文件：**result.html**，打开该文件将看到服务器程序发来的网页。

和 7.1 小节的程序一样，可以删除程序中 **System.setProperty()** 代码，而通过 **java** 命令选项来指定系统参数。

7.2.2 查看对方的证书等连接信息

本章所有程序（不论是客户程序还是服务器程序）在得到 **Socket** 类型的对象后，都可以查看所连接的对方的证书。

★ 实例说明

本实例演示了如何修改 7.1 节的客户程序和服务器程序以查看对方的证书等连接信息。

★ 编程思路：

客户程序和服务器程序最终都是通过 **Socket** 对象得到输入/输出流而进行双向通信。通过 **Socket** 对象不仅可以得到输入/输出流，还可以得到 **SSLSession** 类型的对象，该对象描述了连接双方的关系，通过其方法可以获得连接双方的信息。

在得到 **Socket** 对象后，客户程序和服务器程序的编程方法相同，如下：

（1） 获取 **SSLSession** 对象

```
SSLSession session=((SSLSocket) s).getSession();
```

分析：先将 **Socket** 对象 **s** 强制转换为 **SSLSocket** 类型，再执行其 **getSession()** 方法，得到 **SSLSession** 对象。以后就可以根据需要有选择地执行该对象的各个方法，如下面各步骤。

（2） 获取对方在 **SSL** 协议握手阶段所使用的证书

```
Certificate[ ] cchain=session.getPeerCertificates();
```

分析：执行 **SSLSession** 对象的 **getPeerCertificates()** 方法可以获得对方在 **SSL** 协议握手阶段所使用的证书，如果对方使用的是证书链，将得到一组证书，存放在 **Certificate** 类型的数组中。对数组中的证书可以按照前面几章的方法进行显示、验证等。

(3) 获取自己在 SSL 协议握手阶段所使用的证书

```
Certificate[] cchain2=session.getLocalCertificates();
```

分析: 执行 SSLSession 对象的 getLocalCertificates() 方法可以获得自己在 SSL 协议握手阶段所使用的证书, 如果对方使用的是证书链, 将得到一组证书, 存放在 Certificate 类型的数组中。对数组中的证书可以按照前面几章的方法进行显示、验证等。

(4) 获取对方的主机名称

```
session.getPeerHost();
```

分析: 执行 SSLSession 对象的 getPeerHost() 方法可以获得对方的主机名称, 如果无法得到对方的主机名称, 则得到的是对方的 IP 地址。

(5) 获取 SSL 密码组名称

```
session.getCipherSuite()
```

分析: 执行 SSLSession 对象的 getCipherSuite() 方法可以获得该会话中所有连接所使用的 SSL 密码组 (cipher suite) 的名称。

(6) 获取会话所使用的协议

```
session.getProtocol()
```

分析: 执行 SSLSession 对象的 getProtocol() 方法可以获得该会话中所有连接所使用的协议名称。

(7) 获取会话标志符

```
session.getId()
```

分析: 每个会话有一个标识符, 执行 SSLSession 对象的 getId() 方法可以获得该会话标识符。它返回 byte 类型的数组。

(8) 获取会话创建时间

```
session.getCreationTime()
```

分析: 它返回一个长整型数, 表示会话创建时离格林威治标准时间 1970 年 1 月 1 日 0 时 0 分 0 秒相隔多少毫秒。

(9) 获取会话上次访问时间

```
session.getLastAccessedTime()
```

分析: 它返回一个长整型数, 表示上一次访问该会话时离格林威治标准时间 1970 年 1 月 1 日 0 时 0 分 0 秒相隔多少毫秒。这种访问是指会话级的访问, 获取这些时间可以用于会话的管理。

★代码与分析:

服务器程序和 7.1 节的各个服务器程序类似, 只是加上本小节增加的几步, 完整代码如下:

```
import java.net.*;
import java.math.*;
import java.io.*;
import javax.net.ssl.*;
import java.security.cert.*;
public class MySSLServerSession{
    public static void main(String args[]) throws Exception{
        System.setProperty("javax.net.ssl.keyStore","lfkeystore2");
```

```

        System.setProperty("javax.net.ssl.keyStorePassword", "wshr.ut");
        SSLServerSocketFactory ssf=(SSLServerSocketFactory)
            SSLServerSocketFactory.getDefault();
        ServerSocket ss=ssf.createServerSocket(5432);
        System.out.println("Waiting for connection...");
        while(true) {
            Socket s=ss.accept();
            SSLSession session=((SSLSocket) s).getSession();
            Certificate[ ] cchain2=session.getLocalCertificates();
            System.out.println("The Certificates used in local");
            for(int i=0; i<cchain2.length; i++) {
                System.out.println(((X509Certificate)cchain2[i]).
                    getSubjectDN());
            }

            System.out.println("Peer host is "+session.getPeerHost());
            System.out.println("Cipher is "+session.getCipherSuite());
            System.out.println("Protocol is "+session.getProtocol());
            System.out.println("ID is "+
                new BigInteger(session.getId()));
            System.out.println("Session created in "+
                session.getCreationTime());
            System.out.println("Session accessed in "+
                session.getLastAccessedTime());
            PrintStream out = new PrintStream(s.getOutputStream());
            out.println("Hi");
            out.close();
            s.close();
        }
    }
}

```

该服务器程序可以和 7.1 节的各个客户程序进行通信，也可以和本小节下面的客户程序通信。程序运行时将显示服务器程序实际使用的是哪个证书以及会话信息。对于证书，这里简单地将证书链中各个证书的名称打印出来。由于 7.1 节的各个客户程序只指定了信任哪些证书，并没有提供自己的证书给服务器程序，因此该服务器程序没有使用 `session.getPeerCertificates()` 获取对方在 SSL 协议握手阶段所使用的证书。对于字节数组类型的会话标志符，程序中使用 **BigInteger** 类将其转换成长整型再打印出来。

客户程序和 7.1 节的各个客户程序类似，只是加上本小节增加的几步，完整代码如下，它既可以和 7.1 节的各个服务器程序通信，也可和本小节的服务器程序通信。

```

import java.net.*;
import java.math.*;
import java.io.*;
import javax.net.ssl.*;
import java.security.cert.*;

```

```

public class MySSLClientSession{
    public static void main(String args[ ]) throws Exception {
        System.setProperty("javax.net.ssl.trustStore",
            "clienttrust");
        SSLSocketFactory ssf=
            (SSLSocketFactory) SSLSocketFactory.getDefault( );
        Socket s = ssf.createSocket("127.0.0.1", 5432);
        SSLSession session=((SSLSocket) s).getSession();
        Certificate[ ] cchain=session.getPeerCertificates();
        System.out.println("The Certificates used by peer");
        for(int i=0;i<cchain.length;i++){
            System.out.println( ((X509Certificate)cchain[i]).
                getSubjectDN( ) );
        }
        System.out.println("Peer host is "+session.getPeerHost());
        System.out.println("Cipher is "+session.getCipherSuite( ) );
        System.out.println("Protocol is "+session.getProtocol( ) );
        System.out.println("ID is "+new BigInteger(session.getId( ) ) );
        System.out.println("Session created in "+
            session.getCreationTime( ) );
        System.out.println("Session accessed in "+
            session.getLastAccessedTime( ) );
        BufferedReader in
            = new BufferedReader(new InputStreamReader(s.getInputStream( )));
        String x=in.readLine( );
        System.out.println(x);
        in.close( );
    }
}

```

程序运行时将显示所连接的服务器实际使用的是哪个证书以及会话信息。对于证书，这里简单地将证书链中各个证书的名称打印出来。由于该客户只指定了信任哪些证书，并没有提供自己的证书给服务器程序，因此该服务器程序没有使用 `session.getLocalCertificates()` 获取自己在 SSL 协议握手阶段所使用的证书。对于字节数组类型的会话标志符，程序中使用 `BigInteger` 类将其转换成长整型再打印出来。

★运行程序

输入“`java MySSLServerSession`”运行服务器程序，然后输入“`java MySSLClientSession`”运行本实例中的客户程序，则服务器程序的 DOS 窗口显示：

```

The Certificates used in local
CN=Liu Fang, OU=Packaging, O=Shanghai University, L=ZB, ST=Shanghai, C=CN
CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai, C=CN
Peer host is 127.0.0.1
Cipher is SSL-RSA-WITH-RC4-128-SHA

```

```
Protocol is TLSv1
ID is 28013371676272649684477370449682354285170795912842258606573317487778178237440
Session created in 1039073705950
Session accessed in 1039073706610
```

客户程序的 DOS 窗口显示：

```
The Certificates used by peer
CN=Liu Fang, OU=Packaging, O=Shanghai University, L=ZB, ST=Shanghai, C=CN
CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai, CN
Peer host is localhost
Cipher is SSL_RSA_WITH_RC4_128_SHA
Protocol is TLSv1
ID is 280133716762726496844773704496823542851707959128422586065733174877781782440
Session created in 1039073706000
Session accessed in 1039073706610
Hi
```

从中可以看出客户程序和服务器程序对同一个会话显示出的服务器所使用的证书、ID、创建时间、协议、加密器组等信息是一致的，而上次访问时间等则不同。

服务器程序或客户程序可以和 7.1 节中的各个服务器程序和客户程序替换，这样可以看到使用不同的证书的效果。

7.3 HTTPS 客户及服务器程序

7.2.1 小节设计了自己的通信规则进行双向通信，其实现有的各种网络流量都可以通过 SSL 进行加密，本章下面开始介绍使用 SSL 加密 HTTP 流量的 HTTPS。

7.3.1 最简单的 HTTPS 服务器程序

★ 实例说明

本实例给出最简单的 HTTPS 服务器程序的例子，它可以通过浏览器来访问，也可以通过本节后面的 HTTPS 客户程序来访问。

★ 编程思路：

服务器程序开头部分和 7.1.1 小节一样，获取 Socket 类型的对象 s，其中 HTTPS 使用的标准端口号是 443，因此服务器程序应该使用该端口号，这样浏览器就可以输入“https://服务器地址”来访问服务器了，如果服务器程序中的端口号不是使用 443，则浏览器访问时应通过“https://服务器地址:端口号”来访问服务器。

获得 Socket 类型对象后，就可以和以前一样得到输入/输出流，不同的只是输入/输出流的处理方式不一样，要按照 HTTP 协议规定的方式进行通信。

HTTP 协议基本的规则是：服务器先读取浏览器发来的数据（类似“GET /test.html HTTP/1.1”的字符串），如果是 GET 请求，则根据请求的内容，向浏览器发送 HTTP 版本、MIME 版本、数据类型、数据长度、一个空行以及数据内容等信息。浏览器只有接收到这样的信息，才认为 HTTP 协议执行正确，从而将后面发送的内容作为网页内容显示出来。如对

获取网页（HTML 文档）的请求，Web 服务器可以发送如下信息：

```
"HTTP/1.0 200 OK"
"MIME_version:1.0"
"Content_Type:text/html"
"Content_Length:"+c.length()
""
c
```

其中字符串 c 中包含的是网页的内容。

按照这样的流程，具体的编程步骤可以如下：

(1) 获取输出流

```
PrintStream out = new PrintStream(s.getOutputStream( ));
```

分析：执行 Socket 对象的 `getOutputStream()` 方法，得到 `OutputStream` 类型的对象，通过其 `write()` 方法可以向客户程序发送字节数组。不妨再利用 `OutputStream` 对象象创建 `PrintStream` 类型的对象，通过其 `println()` 方法可以向客户程序发送字符串。

(2) 获取输入流

```
BufferedReader in = new BufferedReader(
    new InputStreamReader(s.getInputStream( )));
```

分析：执行 Socket 对象的 `getInputStream()` 方法，进而创建 `BufferedReader` 类型的对象，通过其 `readln()` 方法可以读取从客户程序发来的字符串。

(3) 读取客户发来的字符串

```
while(( info=in.readLine())!=null){
    System.out.println("now got "+info);
    if(info.equals("")) break;
}
```

分析：浏览器要访问 Web 服务器之前会先发送用户浏览器版本、语言、要访问的内容等信息，因此服务器程序循环读取浏览器发来的信息，直到读到空字符串或读完为止。

(4) 向客户发送信息

```
out.println("HTTP/1.0 200 OK");
out.println("MIME_version:1.0");
out.println("Content_Type:text/html");
```

分析：根据 HTTP 协议的规定，Web 服务器收到浏览器发来的请求后，会通知浏览器 HTTP 版本、MIME 版本、数据类型等信息，通过输出流的打印语句将这些信息发送给浏览器或客户程序。

(5) 获取要传输的内容

```
i++;
String c="<html> <head></head><body> <h1> Hi, this is "
+i+"</h1></Body></html>";
```

分析：这里为简化程序，将要发送给浏览器的网页内容放在一个字符串中，该字符串中的网页显示“Hi, this is ”信息，末尾用一个变量显示一个数字，这个数字每次向浏览器发送信息时增加 1，这样造成动态效果。第一次发送网页时显示“Hi, this is 1”，第二次发送网页时则显示“Hi, this is 2”，字符串中添加了很多制作网页所用的 HTML 标记控制显示的格式。

实际使用中一般可根据浏览器发来的请求读取指定目录的文件发送给浏览器。除了网页外，也可以发送图片等内容。

(6) 发送网页

```
out.println("Content_Length:"+c.length());
out.println("");
out.println(c);
```

分析：先向浏览器发送网页的长度信息，然后将上一步得到的内容发送出去。

★代码与分析：

完整代码如下：

```
import java.net.*;
import java.io.*;
import javax.net.ssl.*;

public class MyHTTPSServer {
    public static void main(String args[ ]) {
        int i=0;
        try {
            SSLServerSocketFactory ssf= (SSLServerSocketFactory)
                SSLServerSocketFactory.getDefault( );
            ServerSocket ss=ssf.createServerSocket(443);
            System.out.println("Web Server OK ");
            while(true){
                Socket s=ss.accept( ); //等待请求
                PrintStream out = new PrintStream(s.getOutputStream( ));
                BufferedReader in = new BufferedReader(
                    new InputStreamReader(s.getInputStream( )));
                String info=null;
                while(( info=in.readLine())!=null){
                    System.out.println("now got "+info);
                    if(info.equals("")) break;
                }
                System.out.println("now go");
                out.println("HTTP/1.0 200 OK");
                out.println("MIME_version:1.0");
                out.println("Content_Type:text/html");
                i++;
                String c="<html> <head></head><body> <h1> Hi, this is "
                    +i+"</h1></Body></html>";
                out.println("Content_Length:"+c.length( ));
                out.println("");
                out.println(c);
                out.close( );
                s.close( );
                in.close( );
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```

    }
} catch (IOException e) {
    System.out.println(e);
}
}
}
}

```

★运行程序

如下运行程序，至屏幕上显示“Web Server OK”，开始可以接收浏览器的连接。

```

C:\java\ch7\Server>java -Djavax.net.ssl.keyStore=lfkeystore2
-Djavax.net.ssl.keyStorePassword=wshr.ut MyHTTPSServer

```

然后再另外一台联网的计算机，或在运行MyHTTPSServer程序的同一台计算机上打开浏览器，输入“https://服务器地址”，如“https://127.0.0.1”。则浏览器将出现图 7- 1所示的提示：

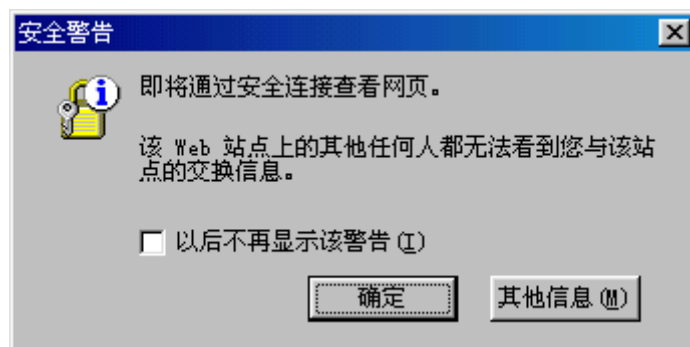


图 7- 1 浏览器访问 HTTPS 服务器的信息

该提示信息表明浏览器即将访问HTTPS服务器，浏览器和服务器之间传递的信息将加密处理。点击“确定”按钮后，浏览器将读取HTTPS服务器提供的证书，如图 7- 2所示，

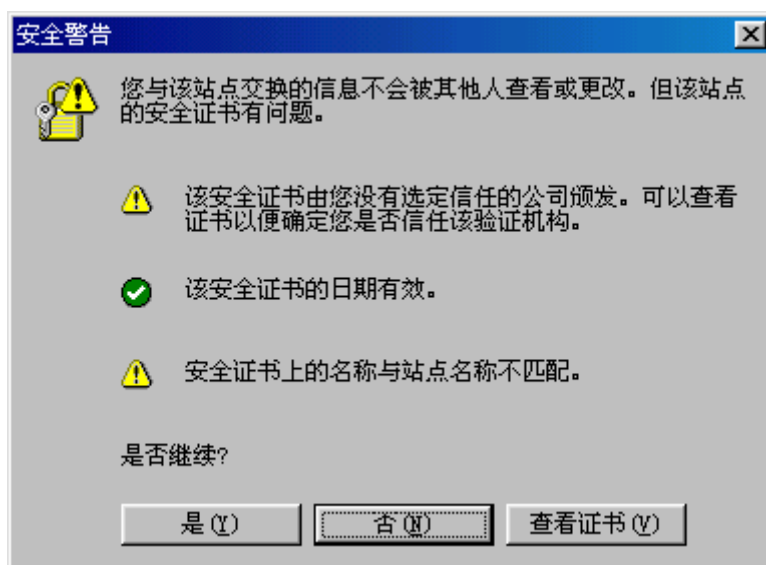


图 7-2 浏览器检查 HTTPS 服务器的证书

由于本实例HTTPS服务器使用的是密钥库Ifkeystore2 中的“Liu Fang”证书，在用户的浏览器中既没有信任该证书，也没有信任该证书的签发者CA “Xu Yingxiao”的证书，因此这里给出警告信息：“该安全证书由您没有选定信任的公司颁发”。如果按照 5.4.1 小节的方法安装CA “Xu Yingxiao”的证书mytest.cer或“Liu Fang”的证书If_signed.cer，或直接点击图 7-2提示中的“安装证书”按钮，在出现的图中类似 5.4.1 小节图 5-5 的窗口中点击“安装证书”，使得用户计算机信任该证书，则将不出现该提示。同样，如果HTTPS服务器使用的证书不是我们自己签发的，而是交给著名CA如Verisign签发（要付费），则浏览器也将自动信任HTTPS服务器的证书。

此外这里还检测证书的日期是否有效、证书上的名称与站点名称是否匹配等。图 7-2 中显示证书上的名称与站点名称不匹配，这是因为证书中的名称是“Liu Fang”，而访问该站点时我们使用了 127.0.0.1 来访问这个站点。如果创建证书时使用服务器的IP地址或者域名作为证书的名称，则该警告信息也将不再出现。

对于上面两个警告信息，点击“是”按钮，确定尽管有这两个警告，用户经过检查还是觉得浏览时信任该证书，于是可以看到网页内容，如图 7-3所示

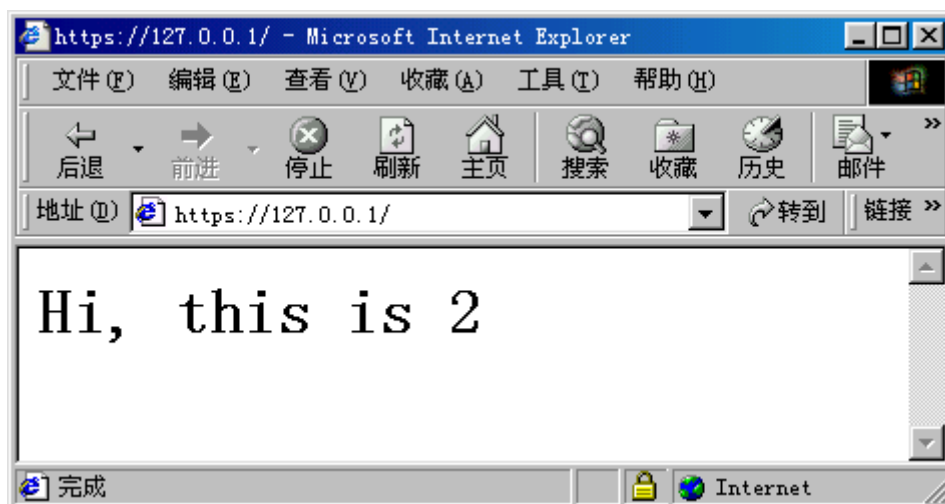


图 7-3 HTTPS 服务器访问结果

“Hi, this is 2”是由本实例的 HTTPS 服务器加密传送给浏览器的，如果点击浏览器“刷新”按钮，则继续出现“Hi, this is 3”，“Hi, this is 4”等内容。

Web 服务器上提示如下：

```
Web Server OK
now go
now got GET / HTTP/1.1
now got Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
application/vnd.ms-excel, application/msword, application/vnd.ms-powerpoint, */*
now got Accept-Language: zh-cn
now got Accept-Encoding: gzip, deflate
now got User-Agent: Mozilla/4.0 (compatible; MSIE 5.0; Windows 98; DigExt)
now got Host: 127.0.0.1
now got Connection: Keep-Alive
now got Accept-Language: zh-cn
now got Accept-Encoding: gzip, deflate
now got
now go
```

下面我们重新创建一个证书，使得证书中宣称的名称和服务器实际的名称一致。将如下命令保存在 c:\java\ch7\server 目录 comstore.bat 文件中，并执行该批处理文件。

```
keytool -genkey -dname "CN=www.my.com, OU=NC, O=Shanghai University, L=ZB,
ST=Shanghai, C=CN" -alias my -keyalg RSA -keystore mycomstore -keypass wshr.ut -storepass
wshr.ut -validity 1000
```

该命令将在当前目录下创建文件名为 mycomstore 的密钥库文件，其中条目 my 存放一个名称为 www.my.com 的证书。

此时，可以使用该证书运行服务器程序：

```
java -Djavax.net.ssl.keyStore=mycomstore -Djavax.net.ssl.keyStorePassword=wshr.ut
MyHTTPSServer
```

浏览器访问该服务器时，应该输入 <https://www.my.com>，这样就不会出现“证书上的名称与站点名称不匹配”的警告了。为了使用户能通过 www.my.com 的名称访问到该服务器，应该为该服务器的 IP 地址设置 www.my.com 的域名。小范围内使用时也可以不设置域名，而是在用户机器的 Hosts 文件中给服务器的 IP 地址设置“www.my.com”的主机名。

如本实例的服务器程序若运行在 IP 地址为 202.120.1.1 的计算机上，用户在另外一台联网的计算机 B 上通过浏览器访问该服务器，则假设计算机 B 安装的是 Windows 9X 操作系统，则用户可以在计算机 B 的 c:\windows 目录创建一个文本文件：hosts。该文件内容如下：

```
202.120.1.1 www.my.com
```

该段内容必须顶格写，前面不能有空格，则以后该用户访问 www.my.com 时将访问 IP 地址为 202.120.1.1 的计算机。

如果只有一台计算机，也可以使用 IP 地址 127.0.0.1 代表本台机器。如可以在 hosts 文件中加入如下内容：

127.0.0.1 www.my.com

这样，浏览器中输入：<https://www.my.com>，将出现图 7-4 所示的窗口。

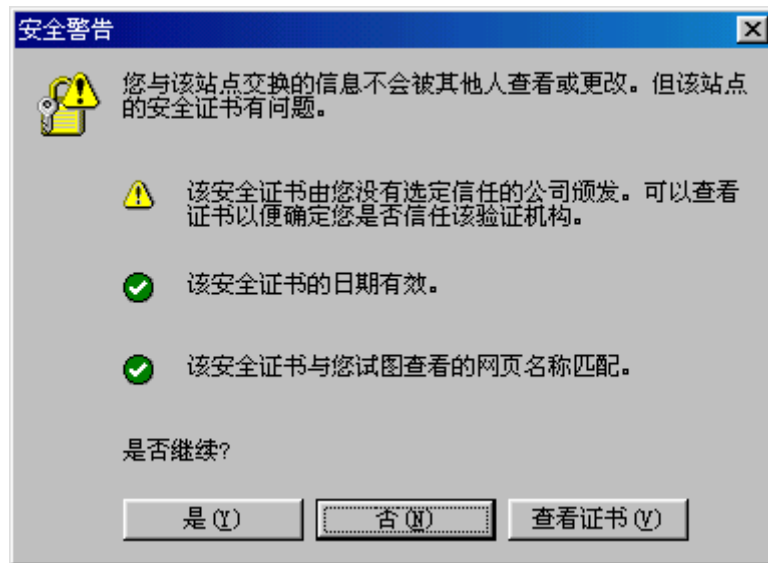


图 7-4 浏览器检查 HTTPS 服务器的证书

和图 7-2 相比，该窗口中已经不再有“证书上的名称与站点名称不匹配”的警告了。

下面我们按照使用 5.4.3 小节的程序签发 mycomstore 中的证书，并按照 6.1.1 小节的方法将签发后的证书导入到密钥库。

先建立一个临时目录 tmp，其中拷贝刚才创建的密钥库 mycomstore、5.4.3 小节的程序 SignCert、以及 CA “Xu Yingxiao” 的密钥库 mykeystore。

执行如下命令将 mycomstore 中的证书导出到文件：

```
keytool -export -alias my -file www.my.com.cer -keystore mycomstore
-storepass wshr.ut
```

执行如下命令将 mykeystore 中 CA “Xu Yingxiao” 的证书导出到文件：

```
keytool -export -alias mytest -file mytest.cer -keystore mykeystore
-storepass wshr.ut
```

执行“java SignCert www.my.com.cer”，则得到密钥库 newstore，密码为 newpass，签发后的证书保存在条目 lf_signed 中。执行如下命令将签发后的证书导出到文件 mycomsigned.cer：

```
keytool -export -file mycomsigned.cer -keystore newstore -storepass newpass
-alias lf_signed
```

再执行如下命令将 CA “Xu Yingxiao” 的证书 mytest.cer 导入到密钥库 mycomstore。

```
keytool -import -alias CAmymtest -keystore mycomstore -file mytest.cer
-storepass wshr.ut
```

最后执行

```
keytool -import -alias my -keystore mycomstore -file mycomsigned.cer
-storepass wshr.ut
```

导入签名后的证书。

不妨将 mycomstore 文件备份为 mycomstore2，拷贝到 c:\java\ch7\server 目录。这样，我们得到了又一个由 CA “Xu Yingxiao” 签名的证书，它的证书链以条目 my 保存在密钥库

mycomstore2 中。

同样，我们可以输入

```
java -Djavax.net.ssl.keyStore=mycomstore2 -Djavax.net.ssl.keyStorePassword=wshr.ut  
MyHTTPSServer
```

运行程序，如果用户的计算机上已经如 5.4.1 小节安装过 mytest.cer 或 mycomsigned.cer 证书，则输入 <https://www.my.com> 浏览时将直接看到网页，而不再出现图 7-2 或图 7-3 所示的警告窗口。

如果使用抓包软件捕捉浏览器和服务器的通信，可以发现网页的内容在网络中传递时是以密文传递的。

7.3.2 最简单的 HTTPS 客户程序

★ 实例说明

本实例使用 URL 类按照 HTTPS 方式和 7.3.1 小节的 HTTPS 服务器以及网上已有的支持 HTTPS 的 Web 服务器进行加密的通信。本实例的程序也可用于不加密的以 HTTP 方式访问的 Web 服务器。

★ 编程思路：

使用 java.net 包中的 URL 类可以根据“http://...”等形式的地址访问对应的 Web 站点上的网页、图片或其他 Internet 资源。首先生成 URL 类型的对象，然后通过其生成输入流，最后通过对输入流的操作获得所需要的资源。

具体步骤如下：

(1) 创建 URL 类型的对象

```
URL u = new URL(args[0]);
```

分析：使用 java.net 包中的 URL 类，其参数为字符串类型，代表所要访问的资源地址（如 <http://www.shu.edu.cn/~xyx>），这里不妨从命令行参数读入所要访问的地址。

(2) 获取输入流

```
InputStream in = u.openStream();
```

分析：执行 URL 对象的 openStream() 方法得到对应该 URL 的输入流，以后通过该输入流可以访问 URL 对应的资源。

(3) 处理输入流

```
BufferedReader f = new BufferedReader(new InputStreamReader(in));  
fileline = f.readLine();
```

分析：可以按照传统的输入流的各种使用方法从输入 URL 对应的输入流中读取相应的数据。如可利用它创建 BufferedReader 类型的对象，然后使用其 readLine() 方法一行一行读取网页内容。如果第 1 步中传入的是 HTTPS 开头的字符串，则程序内部将自动使用 HTTPS 协议和 Web 服务器进行通信，所有数据在网上传递时已经加密过，在程序内部和 Web 服务器内部自动解密。

★ 代码与分析：

完整代码如下：

```
import java.io.*;  
import java.net.*;
```

```

class HttpClient {
    public static void main(String args[ ]) throws IOException{
        String line;
        URL u = new URL(args[0]);
        InputStream in = u.openStream( );
        BufferedReader f= new BufferedReader(new InputStreamReader (in));
        while ((line = f.readLine( )) != null) {
            System.out.println(line+"\n");
        }
    }
}

```

本程序是针对 J2SDK1.4 版本的，如果使用的是老版本的 JDK，需要增加两行语句：

```

        System.setProperty("java.protocol.handler.pkgs",
            "com.sun.net.ssl.internal.www.protocol");
        Security.addProvider(new com.sun.net.ssl.internal.ssl.Provider());

```

才可以使用 HTTPS。

★运行程序

如果计算机已经联网，可以使用网上已有的 Web 服务器运行本程序。

首先测试不加密的方式，输入“java HttpClient <http://www.shu.edu.cn/~xyx> > my.html”运行程序，则可以访问<http://www.shu.edu.cn/~xyx>所对应的网页内容，这里将屏幕输出重定向到了网页my.html中，可以用浏览器打开网页查看网页内容。

同样，输入“java HttpClient <http://www.shu.edu.cn/~xyx/cindex.html> > my2.html”运行程序，则可以访问<http://www.shu.edu.cn/~xyx/cindex.html>所对应的网页内容。

如果使用抓包软件捕捉运行该 Java 程序的计算机的通信，可以发现网页的内容在网络中传递时是以明文传递的。

下面测试使用 HTTPS 的加密方式，这需要所连接的服务器支持 HTTPS。只要找一些 https 开头的网址即可，如“java HttpClient <https://intranet.ied.edu.hk> > uk.html”。

由于 intranet.ied.edu.hk 网站所使用的证书是客户机默认密钥库 C:\j2sdk1.4.0\jre\lib\security\cacerts 中的 CA 或其下级机构所签发的，因而可以通过验证，uk.html 文件中将得到 <https://intranet.ied.edu.hk> 网站的内容。

如果客户机不信任服务器所使用的证书，则无法通过 https 通信。如使用本实例连接 7.3.1 小节的 HTTPS 服务器，如果输入“java HttpClient <https://127.0.0.1>”连接 7.3.1 小节的服务器程序（如果 HttpClient 和 7.3.1 小节的程序不在同一台计算机上，应该把其中的 127.0.0.1 改为 7.3.1 小节程序所在计算机的实际 IP 地址或域名）。则程序出现如下错误提示：

```

Exception in thread "main" javax.net.ssl.SSLHandshakeException: Couldn't find trusted certificate

    at com.sun.net.ssl.internal.ssl.SSLSocketImpl.b(DashoA6275)
    at com.sun.net.ssl.internal.ssl.SSLSocketImpl.a(DashoA6275)
    at com.sun.net.ssl.internal.ssl.ClientHandshaker.a(DashoA6275)
    at com.sun.net.ssl.internal.ssl.ClientHandshaker.processMessage(DashoA6275)
    at com.sun.net.ssl.internal.ssl.Handshaker.process_record(DashoA6275)
    at com.sun.net.ssl.internal.ssl.SSLSocketImpl.a(DashoA6275)
    at com.sun.net.ssl.internal.ssl.SSLSocketImpl.a(DashoA6275)

```

```

at com.sun.net.ssl.internal.ssl.AppOutputStream.write(DashoA6275)
at java.io.OutputStream.write(OutputStream.java:58)
at com.sun.net.ssl.internal.ssl.SSLSocketImpl.startHandshake(DashoA6275)

at sun.net.www.protocol.https.HttpsClient.afterConnect(DashoA6275)
at sun.net.www.protocol.https.AbstractDelegateHttpsURLConnection.connect
(DashoA6275)
at sun.net.www.protocol.http.HttpURLConnection.getInputStream
(HttpURLConnection.java:556)
at sun.net.www.protocol.https.HttpsURLConnectionImpl.getInputStream
(DashoA6275)
at java.net.URL.openStream(URL.java:955)
at HttpsClient.main(HttpsClient.java:13)

```

这是因为 7.3.1 小节的 HTTPS 服务器所使用的证书（密钥库 `lfkeystore2` 中的“Liu Fang”证书或 `mycomstore2` 中的证书“www.my.com”）及其签发者都没有放在客户 Java 程序默认的信任密钥库 `C:\jdk1.4.0\jre\lib\security\cacerts` 文件中，因此客户程序不信任服务器所使用的证书。在 7.3.1 小节运行程序时，我们在浏览器中通过弹出的警告窗口手工确认信任服务器所使用的证书，或在 Windows 中安装证书表明浏览器信任哪些证书，而本实例则可以在运行时通过 Java 命令选项指定。

类似 7.1.2 小节，输入如下命令执行程序：

```
java -Djavax.net.ssl.trustStore=clienttrust HttpsClient https://127.0.0.1
```

该命令指定程序信任密钥库 `clienttrust` 中的证书，该密钥库是 7.1.2 小节创建的，包含了证书“Liu Fang”及“www.my.com”的签发者“Xu Yingxiao”的证书：`mytest.cer`。

此时不再出现找不到信任的证书的提示，而是出现如下出错提示：

```

C:\java\ch7\Client>java -Djavax.net.ssl.trustStore=clienttrust HttpsClient
https://127.0.0.1
Exception in thread "main" java.io.IOException: HTTPS hostname wrong:  should be
<127.0.0.1>

at sun.net.www.protocol.https.HttpsClient.b(DashoA6275)
at sun.net.www.protocol.https.HttpsClient.afterConnect(DashoA6275)
at sun.net.www.protocol.https.AbstractDelegateHttpsURLConnection.connect
(DashoA6275)
at
sun.net.www.protocol.http.HttpURLConnection.getInputStream(HttpURLConnection.java:556)
at
sun.net.www.protocol.https.HttpsURLConnectionImpl.getInputStream(DashoA6275)
at java.net.URL.openStream(URL.java:955)
at HttpsClient.main(HttpsClient.java:13)

```

这是因为客户程序是通过 127.0.0.1 来访问服务器程序的，而服务器程序的证书使用的不是这个名字。因此 7.3.1 小节中的服务器必须使用“java

-Djavax.net.ssl.keyStore=mycomstore2 -Djavax.net.ssl.keyStorePassword=wshr.ut
MyHTTPSServer”来运行，而本小节的程序必须使用“java -Djavax.net.ssl.trustStore=clienttrust
HttpsClient https://www.my.com”来运行。此时，屏幕输出如下：

```
C:\java\ch7\Client>java -Djavax.net.ssl.trustStore=clienttrust HttpsClient
https://www.my.com
<html> <head></head><body> <h1> Hi, this is 5</h1></Body></html>
```

此外，客户程序也可以像 7.1.4 小节使用默认信任密钥库或 7.1.2 小节使用 System.setProperty() 语句设置信任的证书。

如果使用抓包软件捕捉运行该 Java 程序的计算机的通信，可以发现网页的内容在网络中传递时是以加密方式传递的。

7.3.3 基于 Socket 的 HTTPS 客户程序

★ 实例说明

本实例从更低层使用 Socket 类和 7.3.1 小节以及 Internet 上其他的 HTTPS 服务器通信。

★ 编程思路：

和 7.1.2 及 7.2.1 小节一样，先获取 Socket 类型的对象 s，其中 HTTPS 使用的标准端口号是 443，因此连接服务器时应该使用该端口号，这样客户程序就可以和 7.3.1 小节的程序服务器程序中的端口号不是使用 443，则浏览器访问时应通过“https://服务器地址:端口号”来访问服务器。

获得 Socket 类型对象后，就可以和以前一样得到输入/输出流，不同的只是输入/输出流的处理方式不一样，要按照 HTTP 协议规定的方式进行通信。

具体的编程步骤可以如下：

(1) 获取输出流

```
OutputStream outs=s.getOutputStream();
```

```
PrintStream out = new PrintStream(outs);
```

分析：执行 Socket 对象的 getOutputStream() 方法，得到 OutputStream 类型的对象，进而得到 PrintStream 类型的对象，通过其 println() 方法可以向 HTTPS 服务器发送字符串。

(2) 获取输入流

```
InputStream ins = s.getInputStream();
```

```
BufferedReader in = new BufferedReader(
```

```
new InputStreamReader(ins));
```

分析：执行 Socket 对象的 getInputStream() 方法，得到 InputStream 类型的对象，进而得到 BufferedReader 类型的对象，通过其 readLine() 方法可以读取 HTTPS 服务器发来的字符串。

(3) 向服务器发送字符串

```
out.println("Hi,How are u!");
```

分析：执行输出流的 println() 方法。完整的 HTTP 协议中包含了一系列内容，这里不妨只发送一串。

(4) 接收服务器反馈信息

```

        while((line=in.readLine())!=null){
            System.out.println(line);
        }

```

分析：执行输入流的 readLine () 方法。

★代码与分析：

完整代码如下：

```

import java.net.*;
import java.io.*;
import javax.net.ssl.*;

public class HttpsSocketClient {
    public static void main(String args[ ])throws Exception {
        try {
            int port = 443;
            System.setProperty("javax.net.ssl.trustStore",
                               "clienttrust");
            String hostname = args[0];
            SSLSocketFactory ssf=
                (SSLSocketFactory) SSLSocketFactory.getDefault( );
            Socket s = ssf.createSocket(hostname, port);

            OutputStream outs=s.getOutputStream( );
            PrintStream out = new PrintStream(outs);
            InputStream ins = s.getInputStream( );
            BufferedReader in = new BufferedReader(
                new InputStreamReader(ins));

            out.println("Hi,How are u!");
            out.println("");
            String line=null;
            while((line=in.readLine())!=null){
                System.out.println(line);
            }

            in.close();
            out.close();
        } catch(IOException e) {
        }
    }
}

```

★运行程序

如果机器已经联网，可以使用网上已有的支持 HTTPS 的 Web 服务器测试程序。也可以连接 7.3.1 所示的服务器程序。执行“java HttpsSocketClient 127.0.0.1”，将得到如下 HTTPS 服务器反馈的信息：

```
HTTP/1.0 200 OK
MIME_version:1.0
Content_Type:text/html
Content_Length:65
```

```
<html> <head></head><body> <h1> Hi,  this is 1</h1></Body></html>
```

7.3.4 传输实际文件

★ 实例说明

7.3.1 小节的例子中，服务器程序为了简化，只向浏览器传输 Hi 信息。本实例修改了 7.3.1 小节最简单的 HTTPS 服务器程序，使其支持实际的文件传输。

★ 编程思路：

将 7.3.1 小节的程序 MyHTTPSServer 中向浏览器发送数据部分：

```
i++;
String c="<html> <head></head><body> <h1> Hi,  this is "
        +i+"</h1></Body></html>";
out.println("Content_Length:"+c.length());
out.println("");
out.println(c);
```

替换掉。

在浏览器中输入“http://127.0.0.1/xx/yy.html”等字符串，从 HTTPS 服务程序显示的信息可发现，服务器接收到的是“GET /xx/yy.html HTTP/1.1”字符串。因此，只要对该字符串进行解析，获取文件的相对目录和文件名，便可以将文件内容发送到用户浏览器。

HTTPS 服务器接收到的字符串中，文件相对路径和名称在第一个和第二个空格之间，因此，通过字符串的 indexOf()方法获得第一个和第二个空格的位置后，便可以通过字符串的 substring()方法提取出相对路径和文件名：

```
int sp1=request.indexOf(' ');
int sp2=request.indexOf(' ',sp1+1);
String filename=request.substring(sp1+2,sp2);
```

因为浏览器中输入的内容有时不带文件名，如输入 http://127.0.0.1 和 http://127.0.0.1/t/ 时，Web 服务器获得的字符串分别为“GET / HTTP/1.1”和“GET /t/ HTTP/1.1”，这样提取出的内容分别为空字符串和“t/”，因此这时应为其加上默认文件名：

```
if(filename.equals("") || filename.endsWith("/")){
    filename+="index.html";
}
```

最后通过文件输入流读取文件内容，将其发送到浏览器：

```
File fi=new File(filename);
InputStream fs=new FileInputStream(fi);
```



```

int n=fs.available( );
byte buf[ ]=new byte[1024];
    out.println("Content_Length:"+n);
    out.println("");
while ((n=fs.read(buf))>=0){
    out.write(buf,0,n);
}

```

进一步还可检测文件是否存在，不存在则发送出错信息。发送出错信息的代码如下：

```

String c="<html><head><title>Not Found</title></head><body>"
        +"<h1>Error 404-file not found</h1></body></html>";
outstream.println("HTTP/1.0 404 no found");
outstream.println("Content_Type:text/html");
outstream.println("Content_Length:"+c.length( ));

```

★代码与分析：

完整的程序如下：

```

import java.net.*;
import java.io.*;
import javax.net.ssl.*;

public class MyHTTPSServerFile {
    public static void main(String args[ ]) {
        int i=0;
        try {
            SSLServerSocketFactory ssf= (SSLServerSocketFactory)
                SSLServerSocketFactory.getDefault( );
            ServerSocket ss=ssf.createServerSocket(443);
            System.out.println("Web Server OK ");

            while(true){
                Socket s=ss.accept( ); //等待请求
                PrintStream out = new PrintStream(s.getOutputStream( ));
                BufferedReader in = new BufferedReader(
                    new InputStreamReader(s.getInputStream( )));
                String info=null;
                String request=null;
                while(( info=in.readLine())!=null){
                    if(info.indexOf("GET")!=-1){
                        //获取浏览器发来的get信息
                        request=info;
                    }
                    System.out.println("now got "+info);
                    if(info.equals("")) break;
                }
            }
        }
    }
}

```

```

System.out.println("now go");
System.out.println("now gotreq "+request);
if (request!=null){

    out.println("HTTP/1.0 200 OK");
    out.println("MIME_version:1.0");
    out.println("Content_Type:text/html");
    try{
        // 浏览器请求形如 GET /t/1.html HTTP/1.1
        // sp1, sp2为第一次和第二次出现空格的位置,
        // filename从浏览器请求中提取出文件路径和名称 如 t/1.html
        int sp1=request.indexOf(' ');
        int sp2=request.indexOf(' ',sp1+1);
        String filename=request.substring(sp1+2,sp2);
        // 若浏览器请求中无文件名, 则加上默认文件名index.html
        if(filename.equals("") || filename.endsWith("/")){
            filename+="index.html";
        }
        System.out.println("Sending "+filename);
        // 向浏览器发送文件
        File fi=new File(filename);
        InputStream fs=new FileInputStream(fi);
        int n=fs.available();
        byte buf[]=new byte[1024];
        out.println("Content_Length:"+n);
        out.println("");
        while ((n=fs.read(buf))>=0){
            out.write(buf,0,n);
        }
    } catch(Exception e){
        System.out.println(e);
    }
    out.close( );
    s.close( );
    in.close( );
} // end if
} // end while
} catch (IOException e) {
    System.out.println(e);
}
}
}

```

作为测试的网页index.html内容如下：

```
<HTML>
<HEAD><TITLE>Test</TITLE></Head>
<body>
A test for HTTPS
Click <a href=test/1.bat> here </a> for a bat file <p>
</body>
</HTML>
```

它提供了一个指向当前目录 test 子目录 1.bat 文件的链接。。

★运行程序

本实例服务器程序工作在 C:\java\ch7\Server 目录，将如下命令在一行中输入批处理文件 MyHTTPSServerFile.bat。

```
java -Djavax.net.ssl.keyStore=mycomstore2 -Djavax.net.ssl.keyStorePassword=wshr.ut
MyHTTPSServerFile
```

运行 MyHTTPSServerFile 批处理文件，当显示 “Web Server OK” 后启动完成。

用户可在同一台计算机上按照 7.3.1 小节的方法在 c:\windows 目录 hosts 文件中加上下一行：

127.0.0.1 www.my.com

也可以在另外一台计算机上按照 7.3.1 小节的方法在 c:\windows 目录 hosts 文件中加上下一行：

202.120.1.1 www.my.com

其中 202.120.1.1 应该替换为运行本实例程序的计算机。则用户可以在浏览器中输入 <https://www.my.com> 访问该Web服务器，如图 7-5所示。

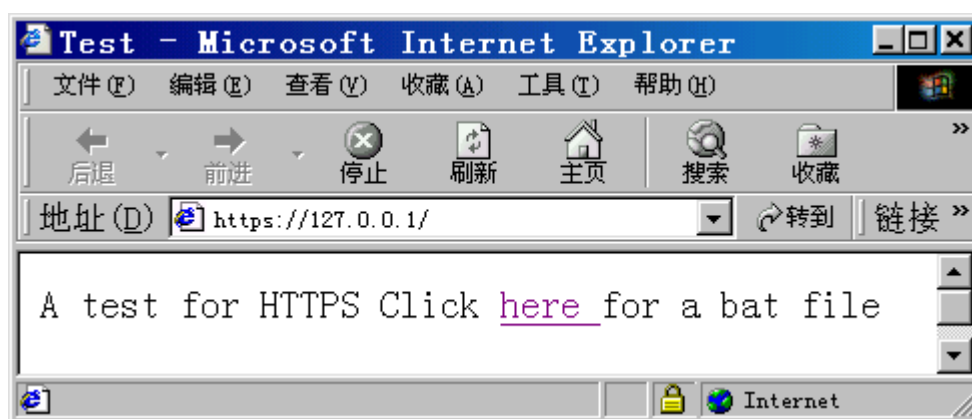


图 7-5 HTTPS 访问实际网页

单击其中的链接 “here”，可以浏览到 c:\java\ch7\server\test\1.bat 文件的内容。

7.4 基于证书的客户身份验证

7.4.1 最简单的验证客户身份的 HTTPS 服务器程序

★ 实例说明

本实例修改了 7.3.1 小节最简单的 HTTPS 服务器程序，使其支持客户身份的验证，它可以通过浏览器来访问，也可以通过本节后面的 HTTPS 客户程序来访问。

★ 编程思路：

程序和 7.3.1 小节类似，只是增加了一句执行 `ServerSocket` 对象的 `setNeedClientAuth(true)` 方法：

```
SSLServerSocket ss=(SSLServerSocket)ssf.createServerSocket(443);
ss.setNeedClientAuth(true);
```

★代码与分析：

完整代码如下：

```
import java.net.*;
import java.io.*;
import javax.net.ssl.*;

public class MyHTTPSServerAuth {

    public static void main(String args[ ]) {
        int i=0;
        try {
            SSLServerSocketFactory ssf= (SSLServerSocketFactory)
                SSLServerSocketFactory.getDefault( );

            SSLServerSocket ss=
                (SSLServerSocket)ssf.createServerSocket(443);

            //要求客户验证
            ss.setNeedClientAuth(true);
            System.out.println("Web Server OK ");

            while(true){
                Socket s=ss.accept( ); //等待请求
                PrintStream out = new PrintStream(s.getOutputStream( ));
                BufferedReader in = new BufferedReader(
                    new InputStreamReader(s.getInputStream( )));
                String info=null;
                while(( info=in.readLine())!=null){
                    System.out.println("now got "+info);
                    if(info.equals("")) break;
                }
            }
        }
    }
}
```

```

        System.out.println("now go");
        out.println("HTTP/1.0 200 OK");
        out.println("MIME_version:1.0");
        out.println("Content_Type:text/html");
        i++;
        String c="<html> <head></head><body> <h1> Hi, this is "
                +i+"</h1></Body></html>";
        out.println("Content_Length:"+c.length( ));
        out.println("");
        out.println(c);
        out.close( );
        s.close( );
        in.close( );
    }
} catch (IOException e) {
    System.out.println(e);
}
}
}
}

```

★运行程序

和7.3.1小节一样，输入：

```
java -Djavax.net.ssl.keyStore=lfkeystore2 -Djavax.net.ssl.keyStorePassword=wshr.ut
MyHTTPSServerAuth
```

或

```
java -Djavax.net.ssl.keyStore=mycomstore2
-Djavax.net.ssl.keyStorePassword=wshr.ut MyHTTPSServerAuth
```

运行程序，屏幕上出现“Web Server OK”，表明服务器已经正常启动。

由于本实例的服务器需要用户浏览器提供证书表明用户的身份，因此用户需首先在自己的计算机中安装代表用户的证书。可以如附录“申请数字标识（证书）”所示申请一个自己的证书，并如附录所示安装在自己的计算机中。

和7.3.1小节类似，在浏览器中输入<https://127.0.0.1>，在出现的图 7- 1所示的窗口中单击“确定”按钮。出现如图 7- 6所示窗口，该窗口中给出的是用户浏览器中已经安装的证书，选择证书后，单击“确定”按钮。出现图 7- 7所示的私钥容器提示信息，单击“确定”按钮后，如果HTTPS服务器信任用户浏览器提供的证书，则和7.3.1小节类似进入图 7- 8所示的警告窗口。继续单击“确定”按钮，最终浏览到本小节服务器发送的信息。

在本实例中，由于用户计算机上安装的证书是按照附录“申请数字标识（证书）”中的步骤从 Verisign 申请得到的，Verisign 的证书已经包括在 Java 默认的信任密钥库 C:\jdk1.4.0\jre\lib\security\cacerts文件中了。因此本实例的服务器程序信任用户浏览器提交的数字证书。

除了通过默认信任密钥库设置服务器信任的密钥外，还可以通过Java命令选项 -Djavax.net.ssl.trustStore进行设置。

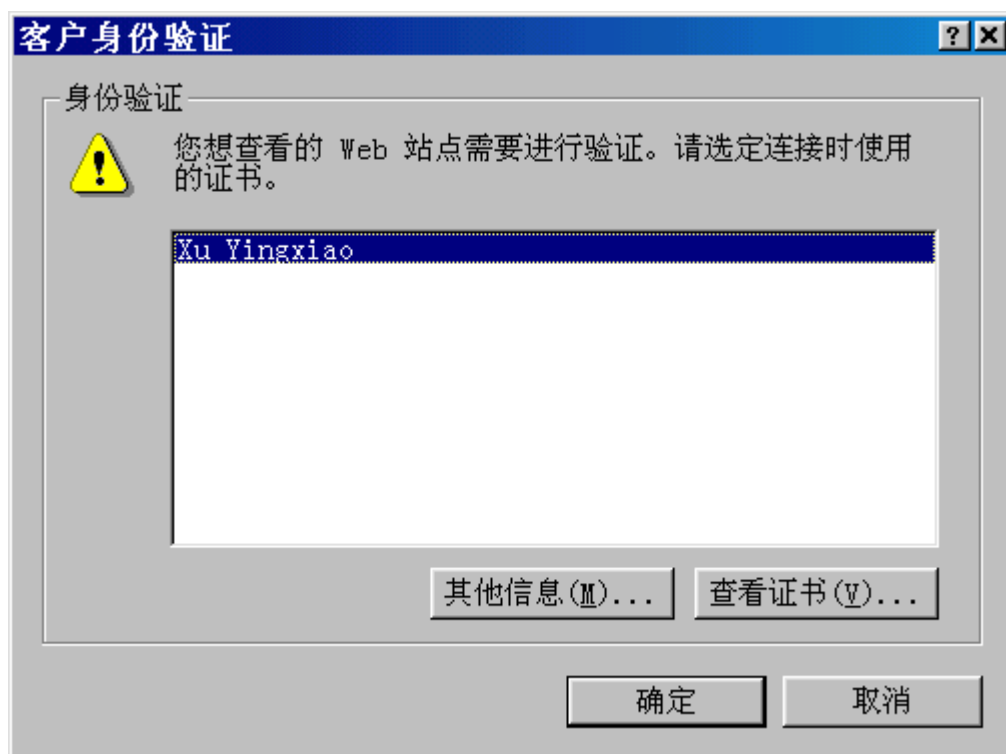


图 7-6 选择用户的证书



图 7-7 私钥容器提示应用程序在读取其内容

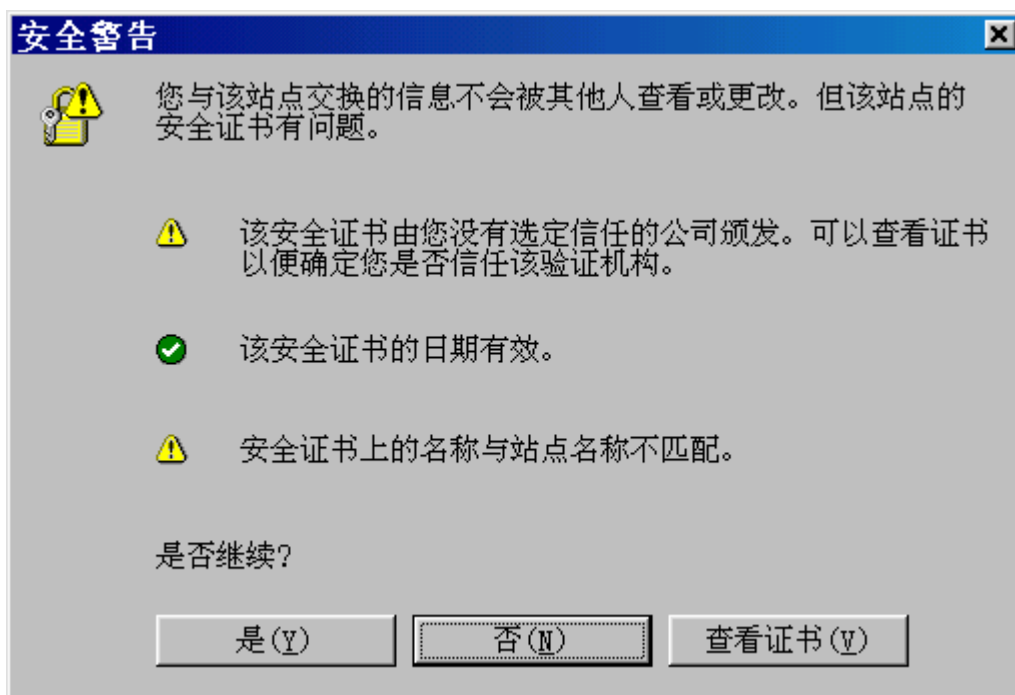


图 7-8 浏览器检查 HTTPS 服务器提供的证书

7.4.2 编写客户程序连结需客户验证的 HTTPS 服务器

★ 实例说明

本实例使用 7.3.2 和 7.3.3 小节的程序代替浏览器访问 7.4.1 小节的需要客户验证的服务器程序。

★ 编程思路：

在 7.3 节中，运行服务器程序时指定了 `-Djavax.net.ssl.keyStore` 选项，服务器从中提取证书向客户程序表明自己是谁。而运行客户程序时指定了 `-Djavax.net.ssl.trustStore` 选项，指定客户信任哪些证书，这样当其接收到服务器程序发来的证书后就可以判断是否相信服务器。

在本节中，除了 7.3 节的验证外，客户程序也需要 `-Djavax.net.ssl.keyStore` 选项向服务器表明自己是谁，类似地，服务器程序应如 7.4.1 小节那样使用默认信任密钥库，或使用 `-Djavax.net.ssl.trustStore` 选项指定服务器信任谁。

本小节服务器不妨使用密钥库 `mycomstore2`，该密钥库中 `my` 条目包含了证书 www.my.com，服务器使用它来向用户宣称自己是谁。该证书是由 CA "Xu Yingxiao" 签发的。

服务器信任的密钥库使用 `clienttrust`，其中包含了 CA "Xu Yingxiao" 的证书，这样，CA "Xu Yingxiao" 签发的所有证书都被服务器程序信任。可将 `c:\java\ch7\client` 目录中 `clienttrust` 文件拷贝到 `c:\java\ch7\server` 目录。

客户程序不妨使用密钥库 `lfkeystore2`，该密钥库中 `lf` 条目包含了 "Liu Fang" 的证书，客户程序使用它向服务器宣称自己是谁。该证书是由 CA "Xu Yingxiao" 签发的，由于服务器程序信任 CA "Xu Yingxiao" 签发的证书，因此将信任该证书。

客户程序使用的信任密钥库也使用 `clienttrust`，即信任 CA "Xu Yingxiao" 签发的证书。

★代码与分析：

本实例服务器程序使用 7.4.1 小节的 MyHTTPSServerAuth，客户程序使用 7.3.2 小节的 HttpClient 和 7.3.3 小节的 HttpsSocketClient 程序。

★运行程序

本实例服务器程序工作在 C:\java\ch7\Server 目录，将如下命令在一行中输入批处理文件 MyAuthServer.bat。

```
java -Djavax.net.ssl.keyStore=mycomstore2  
-Djavax.net.ssl.keyStorePassword=wshr.ut -Djavax.net.ssl.trustStore=clienttrust  
MyHTTPSServerAuth
```

执行 MyAuthServer，启动服务器程序，当屏幕提示“Web Server OK”后，服务器启动完毕。

本实例客户程序工作在 C:\java\ch7\Client 目录，使用 7.3.2 小节的程序 HttpClient，将如下命令在一行中输入批处理文件 MyAuthClient.bat。

```
java -Djavax.net.ssl.keyStore=lfkeystore2 -Djavax.net.ssl.trustStore=clienttrust  
-Djavax.net.ssl.keyStorePassword=wshr.ut HttpClient https://www.my.com
```

执行 MyAuthClient，则客户程序开始运行，并显示服务器程序发来信息：

```
<html> <head></head><body> <h1> Hi, this is 1</h1></Body></html>
```

如果使用 7.3.3 小节的 HttpsSocketClient，由于它在程序中已经指定了信任的密钥库 clienttrust，因此不需要指定 -Djavax.net.ssl.trustStore 选项。只要在批处理文件 MyAuthServer2.bat 中输入如下一行命令：

```
java -Djavax.net.ssl.keyStore=lfkeystore2 -Djavax.net.ssl.keyStorePassword=wshr.ut  
HttpsSocketClient 127.0.0.1
```

执行后 MyAuthServer2 将显示：

```
HTTP/1.0 200 OK  
MIME_version:1.0  
Content_Type:text/html  
Content_Length:65
```

```
<html> <head></head><body> <h1> Hi, this is 2</h1></Body></html>
```

本章介绍了基于 SSL 和 HTTPS 的数据加密传输，同时通过客户和服务器的数字证书，客户和服务器程序之间可以相互向对方表明自己的身份，实现程序之间的信任关系。

除了加密传输数据外，应用程序在运行过程中往往需要访问用户的本地资源，本书后面章节将介绍程序的安全运行。

第 8 章 程序运行的安全性

——基于代码来源的授权

本章重点：

当用户在计算机中执行 Java 程序时，很关心这个 Java 程序是否是安全的。不安全的程序在运行时可能泄漏一些信息（比如可能读取“我的文档”中的文件并通过网络秘密发送出去），也可能删除你的一些重要文件。

本章介绍一种解决方法：根据代码的来源（存放在什么位置或所有者是谁）来设置代码可以有那些权限。

本章主要内容：

- 使用安全管理器来运行程序
- 编写自己的安全管理器
- 根据代码的 URL 进行各种授权
- 使用 jarsigner 工具对代码进行签名
- 根据代码的签名者进行各种授权
- 多个代码相互调用中定义特权代码
- 定义自己的权限
- 对 Java Applet 进行签名
- 使用 Java Plug-in 运行 Java Applet
- 基于策略文件和 RSA 签名控制 Java Applet

8.1 安全管理器的使用

本节首先给出通过 Java 命令行选项指定默认安全管理器实现程序运行的安全的实例，然后给出如何定义自己的安全管理器满足特殊需要，最后给出程序中指定安全管理器的方法。

8.1.1 使用默认的安全管理器限制应用程序

★ 实例说明

本实例先给出一个简单的 Java 程序，在一定条件下它会偷偷读取用户目录 C:盘根目录 autoexec.bat 文件的内容。然后给出如果用户不希望自己的计算机被 Java 程序任意读写，如

何才能禁止 Java 程序进行所有这类的操作。

★ 编程思路:

本实例的程序只是一个简单的读取文件的程序，本节重点在如何运行程序。

该程序可能是一个大型的程序，其中可能某段代码包含了读取用户文件的操作。程序中 ShowFile 类的 go(String name) 方法中，通过参数中的文件名称生成 FileReader 类型的对象，继而生成 BufferedReader 类型的对象，通过 BufferedReader 对象的 readLine() 方法读取文件中的内容。

为了使本示例更有代表性，另外定义了一个类 RunShowFile 作为主程序，它从命令行参数读入要读取的文件的名称，然后创建 ShowFile 类，调用其 go() 方法显示指定的文件的内容。

此外主程序还做了一个陷阱：如果用户输入的文件名以 “.txt” 为后缀，则偷偷读取 c:\autoexec.bat 文件的内容，然后偷偷做各种事情（示例程序不妨偷偷做哪些事情忽略，只演示确实读取了 c:\autoexec.bat 文件的内容）。程序中使用的是 “c:\\autoexec.bat”，这是因为 Java 字符串中斜杠 “\” 代表转义，字符串中的斜杠要用双斜杠 “\\” 表示。

★ 代码与分析:

完整程序如下:

```
import java.io.*;

public class ShowFile{
    public String go(String name) throws IOException{
        String s;
        String content="";
        BufferedReader in;
        in = new BufferedReader(new FileReader(name));
        while ((s = in.readLine()) != null) {
            content+=s+"\n";
        }
        return content;
    }
}

public class RunShowFile{
    public static void main(String args[]) throws IOException{
        ShowFile t=new ShowFile();
        String s=t.go(args[0]);
        if(args[0].endsWith(".txt")){
            String s2=t.go("c:\\autoexec.bat");
        }
        // 使用 s2 做各种事情
        System.out.println(s);
        System.out.println("Over");
    }
}
```

这两个类分别放在两个文件中。

★运行程序

程序编写者输入“`javac *.java`”编译程序，当前目录下将出现两个文件：`ShowFile.class` 和 `RunShowFile.class`。然后将这两个文件提供给使用者，告诉使用者：运行 `RunShowFile.class`，则显示参数中指定的文件的内容。但程序编写者隐瞒了如果用户输入的文件名以“.txt”为后缀，则偷偷读取 `c:\autoexec.bat` 文件的内容的陷阱。

当用户得到 `ShowFile.class` 和 `RunShowFile.class` 文件后，在不考虑安全性问题时，可能只是输入：

```
java RunShowFile RunShowFile.java
```

则将显示当前目录下 `RunShowFile.java` 文件的内容。

或输入：

```
java RunShowFile C:\j2sdk1.4.0\README.txt
```

则将显示 `C:\j2sdk1.4.0\README.txt` 文件的内容。

用户也可以在当前目录建立一个.txt 为后缀的文件，如文件名可以使用“1.txt”，随便输一些内容以便进行测试。则输入

```
java RunShowFile 1.txt
```

将显示当前目录下 `1.txt` 文件的内容。

但用户并不知道，在其输入“`java RunShowFile C:\j2sdk1.4.0\README.txt`”或者“`java RunShowFile 1.txt`”时，用户机器上 `c:\autoexec.bat` 文件的内容已经被泄漏了。更严重的是，只要程序开发者愿意，程序其实可能在你不知不觉中任意修改、查看你的计算机系统中任何信息。如果这个程序是从一些不可靠的地方得来的，如从网上随便下载的，则这种安全问题可能更严重。

各种语言编写的程序都会面临着这一问题，尽管用户可以分析程序的源代码或进行各种监控，但当程序很大，这种工作是非常繁琐且不可靠的。而 Java 使用安全管理器较好地解决了这一问题。

只要在运行 Java 程序时指定一个 Java 命令行选项：`-Djava.security.manager`，则将使用 Java 缺省的安全管理器 `java.lang.SecurityManager` 来强制进行各种安全检查，如输入：

```
java -Djava.security.manager RunShowFile C:\j2sdk1.4.0\README.txt
```

运行程序，则将显示：

```
Exception in thread "main" java.security.AccessControlException: access denied (
  java.io.FilePermission C:\j2sdk1.4.0\README.txt read)
    at
  java.security.AccessControlContext.checkPermission(AccessControlContext.java:270)
    at java.security.AccessController.checkPermission(AccessController.java:401)
    at java.lang.SecurityManager.checkPermission(SecurityManager.java:542)
    at java.lang.SecurityManager.checkRead(SecurityManager.java:887)
    at java.io.FileInputStream.<init>(FileInputStream.java:100)
    at java.io.FileInputStream.<init>(FileInputStream.java:66)
    at java.io.FileReader.<init>(FileReader.java:39)
    at ShowFile.go(RunShowFile.java:7)
    at RunShowFile.main(RunShowFile.java:19)
```

其中第二行指出，程序对 `c:\j2sdk1.4.0\readme.txt` 文件没有读的许可（权限）。此时刚开始访问 `c:\j2sdk1.4.0\readme.txt`，尚未执行到访问 `c:\autoexec.bat` 的语句，就已经被禁止访问

了。

输入

```
java -Djava.security.manager RunShowFile 1.txt
```

运行程序，则将显示：

```
Exception in thread "main" java.security.AccessControlException: access denied (
java.io.FilePermission c:\autoexec.bat read)
    at java.security.AccessControlContext.checkPermission(AccessControlConte
xt.java:270)
    at java.security.AccessController.checkPermission(AccessController.java:
401)
    at java.lang.SecurityManager.checkPermission(SecurityManager.java:542)
    at java.lang.SecurityManager.checkRead(SecurityManager.java:887)
    at java.io.FileInputStream.<init>(FileInputStream.java:100)
    at java.io.FileInputStream.<init>(FileInputStream.java:66)
    at java.io.FileReader.<init>(FileReader.java:39)
    at ShowFile.go(RunShowFile.java:7)
    at RunShowFile.main(RunShowFile.java:21)
```

其中第二行指出，程序对 c:\autoexec.bat 文件没有读的权限。此时，访问 1.txt 文件已经通过，因为 1.txt 在当前目录。而程序偷偷访问的 c:\autoexec.bat 不在当前目录，因此被禁止访问。

只有输入

```
java -Djava.security.manager RunShowFile RunShowFile.java
```

因为 RunShowFile.java 在当前目录，因此可以正常运行，程序将显示 RunShowFile.java 文件的所有内容。

8.1.2 编写自己的安全管理器

★ 实例说明

上一节通过 Java 命令行选项 “-Djava.security.manager” 使用缺省的安全管理器 java.lang.SecurityManager 来强制进行各种安全检查，本实例先给出如何自己编写这样一个安全管理器，从中可以了解 Java 安全管理器的工作细节。

★ 编程思路：

仍旧使用 8.1.1 小节的程序，只是安全管理器使用我们自己定义的，而不是缺省的 java.lang.SecurityManager 类。

为了定义自己的安全管理器，只要定义 java.lang.SecurityManager 类的子类即可。java.lang.SecurityManager 类中有大量 checkXXX() 形式的方法，Java 类库中各个类的方法凡是执行敏感的操作时，都会先获取安全管理器，执行安全管理器相应的 checkXXX() 方法检验看是否有某种权限。

如 8.1.1 小节的程序使用了 FileReader 类，我们可以使用 WinZip 工具打开 J2SDK 安装目录 C:\j2sdk1.4.0 下的 src.zip 文件，该文件是 Java 类库的源代码，从中找到 FileReader.java 文件查看其源代码，会发现 FileReader 类的构造器中执行了 super(new FileInputStream(fileName))，继续打开 src.zip 文件中的 FileInputStream.java 文件，会发现其构造器中有如下代码：

```

public FileInputStream(File file) throws FileNotFoundException {
    String name = file.getPath();
    SecurityManager security = System.getSecurityManager();
    if (security != null) {
        security.checkRead(name);
    }
}

```

可见，在我们编写程序执行 `new FileReader(name)` 时，程序已经自动通过 `System.getSecurityManager()` 方法得到了安全管理器，并执行了其中的 `checkRead()` 方法检查是否有权限。如果检查没有通过，`checkRead()` 方法将抛出异常对象。

因此，我们编写自己的安全管理器时，只要将缺省的 `java.lang.SecurityManager` 类中满足不了自己需要的 `checkXXX()` 方法重写即可。

如对 8.1.1 小节的例子，默认的安全管理器不允许访问 `c:\autoeec.bat`，也不允许访问 `C:\j2sdk1.4.0\README.txt` 文件。如果希望能够读 C 所有的“.txt”和“.java”后缀的文件，则可以重写 `checkRead()` 方法，判断方法参数传入的文件名是否以“.txt”或“.java”后缀为后缀。由于程序在运行过程中还需要读其他类，因此同时要判断文件名是否以“.class”为后缀，以及文件是否在“C:\j2sdk1.4.0”或其子目录（如果是其他版本的 J2SDK，则应修改为相应的安装目录）。如果条件都不满足，则抛出 `SecurityException` 类型的异常对象，在创建 `SecurityException` 对象时可通过构造器的参数指定产生异常的原因。

★代码与分析:

完整程序如下:

```

import java.io.*;

public class MySecurityManager extends SecurityManager {
    public void checkRead(String file) {
        if ( !(file.endsWith(".txt"))
            && !(file.endsWith(".java"))
            && !(file.endsWith(".class"))
            && !(file.startsWith("C:\\j2sdk1.4.0")) ) {
            throw new SecurityException ("No Read Permission for : " + file);
        }
    }
}

```

★运行程序

运行程序时可在 `java` 命令行选项 `-Djava.security.manager` 后加上等于号“=”，将所编写的自己的安全管理器的类名赋值给 `java.security.manager` 即可。

如输入

`java -Djava.security.manager=MySecurityManager RunShowFile RunShowFile.java`
运行程序，则使用 `MySecurityManager` 类型的对象作为安全管理器，由于它允许读所有“.java”后缀的文件，因此程序和 8.1.1 小节一样可以正常显示 `RunShowFile.java` 文件的内容。

又如输入“`java -Djava.security.manager=MySecurityManager RunShowFile C:\j2sdk1.4.0\README.txt`”，则显示如下信息:

```
C:\java\ch8\ShowFile>java -Djava.security.manager=MySecurityManager RunShowFile
C:\j2sdk1.4.0\README.txt
Exception in thread "main" java.lang.SecurityException: No Read Permission for :
c:\autoexec.bat
    at MySecurityManager.checkRead(MySecurityManager.java:8)
    at java.io.FileInputStream.<init>(FileInputStream.java:100)
    at java.io.FileInputStream.<init>(FileInputStream.java:66)
    at java.io.FileReader.<init>(FileReader.java:39)
    at ShowFile.go(RunShowFile2.java:7)
    at RunShowFile.main(RunShowFile.java:21)
```

对比 8.1.1 小节的输出可以发现，程序不再提示对 c:\j2sdk1.4.0\readme.txt 文件没有读的权限，而是提示对 c:\autoexec.bat 没有读的权限。因为我们自己定义的 MySecurityManager 类在 checkRead() 方法中允许读 c:\j2sdk1.4.0\readme.txt 文件，对该文件名不抛出异常对象。

执行“java -Djava.security.manager=MySecurityManager RunShowFile 1.txt”也是同样的输出。

如果显示其他目录中的 java 程序，如按照 8.1.1 小节的方式输入

```
java -Djava.security.manager RunShowFile c:\java\ch2\Caesar.java
```

运行程序，则程序输出：

```
C:\java\ch8\ShowFile>java -Djava.security.manager RunShowFile
c:\java\ch2\Caesar.java
Exception in thread "main" java.security.AccessControlException: access denied (
java.io.FilePermission c:\java\ch2\Caesar.java read)
    at
java.security.AccessControlContext.checkPermission(AccessControlContext.java:270)
    at java.security.AccessController.checkPermission(AccessController.java:401)
    at java.lang.SecurityManager.checkPermission(SecurityManager.java:542)
    at java.lang.SecurityManager.checkRead(SecurityManager.java:887)
    at java.io.FileInputStream.<init>(FileInputStream.java:100)
    at java.io.FileInputStream.<init>(FileInputStream.java:66)
    at java.io.FileReader.<init>(FileReader.java:39)
    at ShowFile.go(RunShowFile2.java:7)
    at RunShowFile.main(RunShowFile.java:19)
```

其中显示对 c:\java\ch2\Caesar.java 没有读的权限，这是因为默认的安全管理器不允许任意访问。如果使用本节的安全管理器输入

```
java -Djava.security.manager=MySecurityManager RunShowFile c:\java\ch2\Caesar.java
```

运行程序，则程序将显示 c:\java\ch2\Caesar.java 文件的内容，这是因为我们自己定义的 MySecurityManager 类在 checkRead() 方法中对任何“.java”后缀的文件都不抛出异常对象。

8.1.3 在程序中设置安全管理器

★ 实例说明

8.1.1和 8.1.2 小节都是通过 Java 命令行选项“-Djava.security.manager”来指定安全管理器的，本实例在 Java 程序中直接指定安全管理器。

★ 编程思路：

通过 System 类的静态方法 setSecurityManager()可以在程序中设置安全管理器，其参数中传入的是安全管理器对象。可以执行 System.setSecurityManager(new java.lang.SecurityManager())使用默认的安全管理器，也可以执行 System.setSecurityManager(new MySecurityManager())使用 8.1.2 小节定义的自己的安全管理器。

★代码与分析：

完整程序如下：

```
import java.io.*;

public class RunShowFile2{

    public static void main(String args[]) throws IOException{

        System.setSecurityManager( new MySecurityManager( ));

        ShowFile t=new ShowFile();
        String s=t.go(args[0]);
        if(args[0].endsWith(".txt")){
            String s2=t.go("c:\\autoexec.bat");
        }
        // 使用 s2 做各种事情
        System.out.println(s);
        System.out.println("Over");
    }
}
```

★运行程序

直接输入

```
java RunShowFile2 C:\jdk1.4.0\README.txt
```

运行程序，即可使用 8.1.2 小节自己定义的安全管理器运行程序，和 8.1.2 小节一样，程序显示对“c:\autoexec.bat”没有读的权限。

直接输入

```
java RunShowFile2 c:\java\ch2\Caesar.java
```

运行程序，则程序将和 8.1.2 小节一样显示 c:\java\ch2\Caesar.java 文件的内容。

程序中安全管理器只能指定一个，如输入

```
java -Djava.security.manager RunShowFile2 c:\java\ch2\Caesar.java
```

运行程序，则程序显示如下出错信息：

```
C:\java\ch8\ShowFile>java -Djava.security.manager RunShowFile2 c:\java\ch2\Caesar.java
Exception in thread "main" java.security.AccessControlException: access denied (
java.lang.RuntimePermission createSecurityManager)
```

```
at
java.security.AccessControlContext.checkPermission(AccessControlContext.java:270)
at
java.security.AccessController.checkPermission(AccessController.java:401)
at java.lang.SecurityManager.checkPermission(SecurityManager.java:542)
at java.lang.SecurityManager.<init>(SecurityManager.java:298)
at MySecurityManager.<init>(MySecurityManager.java:2)
at RunShowFile2.main(RunShowFile2.java:19)
```

在执行 `System.setSecurityManager(new MySecurityManager())` 语句时发生不允许创建安全管理器的访问控制异常。

8.2 使用策略文件基于代码位置进行授权

默认安全管理器禁止了所有不安全的操作，尽管 8.1.2 小节定义的自己的安全管理器可以放开部分权限，但它需要使用者进行编程，而且如果需要同一个程序中不同的类具有不同权限时，编写自己的安全管理器将非常复杂。

因此自 JDK1.2 起开始使用比较简便的基于策略文件（又称为规则文件）的授权。策略文件是一个文本文件，可以手工编写，也可以使用 Java 安装目录 bin 子目录下的 `policytool` 工具自动生成。本节给出策略文件的创建和使用方法，并给出基于代码的位置进行各种授权的例子

8.2.1 允许所有代码具有所有权限

★ 实例说明

本实例演示了如何创建策略文件，允许所有的代码具有所有权限。该实例仅作演示用，因为实际使用时既然使用了安全管理器总是希望限制一些权限。

★ 编程思路

通过 `policytool` 工具编写策略文件，授予所有代码所有权限，然后在运行程序时通过 `java` 命令行选项 “`-Djava.security.policy`” 指定使用该策略文件。

编写策略文件的步骤如下：

（12） 启动策略工具

在命令行中输入 “`policytool`” 命令启动策略工具软件，出现 图 8-1 所示的图形界面，其中提示找不到规则文件，这是因为策略工具软件刚启动时会到用户主目录寻找是否有 “`.java.policy`” 策略文件。这里简单地单击 “确认” 按钮即可。



图 8-1 策略工具启动画面

(13) 添加规则项目

在图 8-1 的窗口中单击“添加规则项目”按钮，出现图 8-2 所示的添加规则项目窗口。

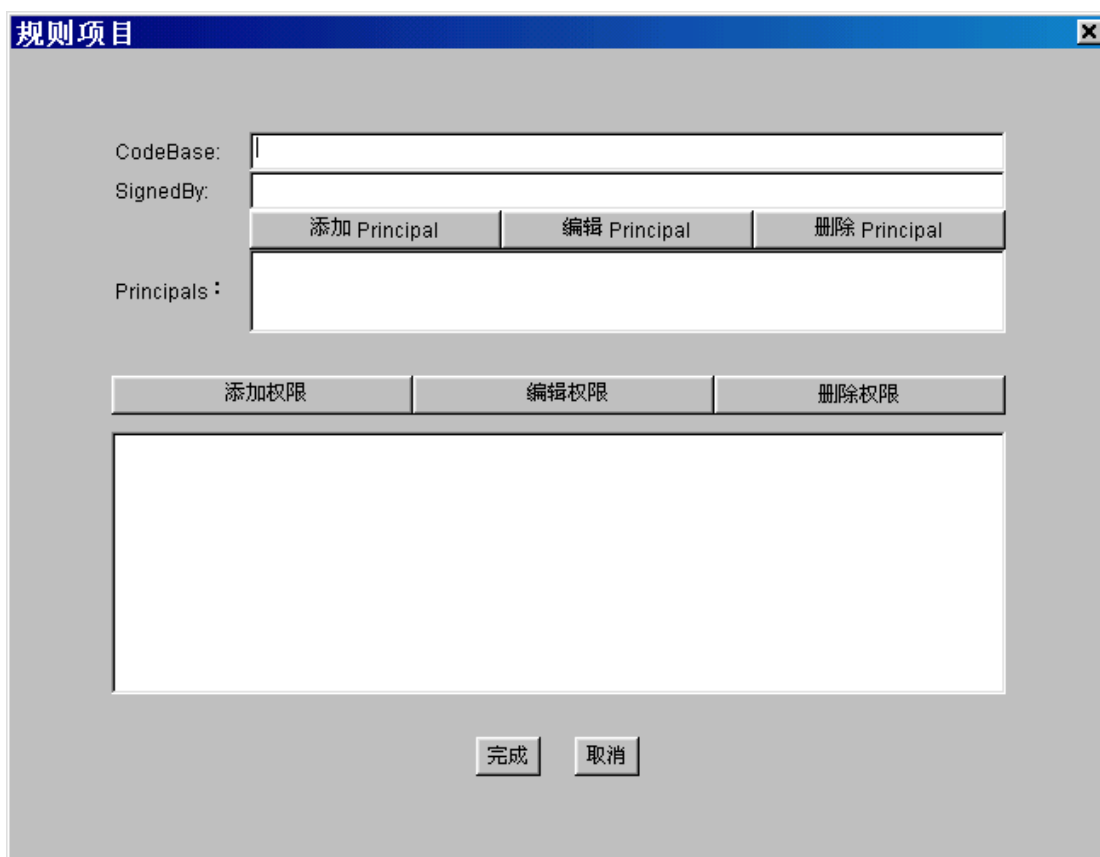


图 8-2 添加规则项目

(14) 添加权限

在图 8-2 的窗口中，“CodeBase”项和“SignedBy”项不填写，则添加的权限是针对所有代码的。单击“添加权限”按钮，出现图 8-3 所示的添加权限窗口。

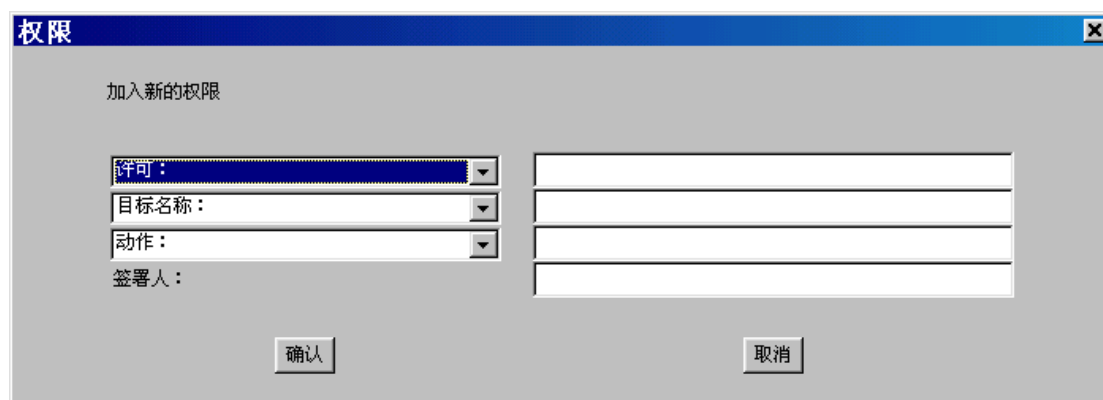


图 8-3 添加权限

(15) 设置权限

图 8-3 窗口是设置权限的主要窗口，其中“许可”一栏有大量选项，如图 8-4 所示。可以通过这些选项定义是否允许程序进行各种特定的操作。这里不妨选中“AllPermission”，则窗口中该项自动填上“java.security.AllPermission”。

作为简单的示例，其他栏目的作用将在后面介绍。

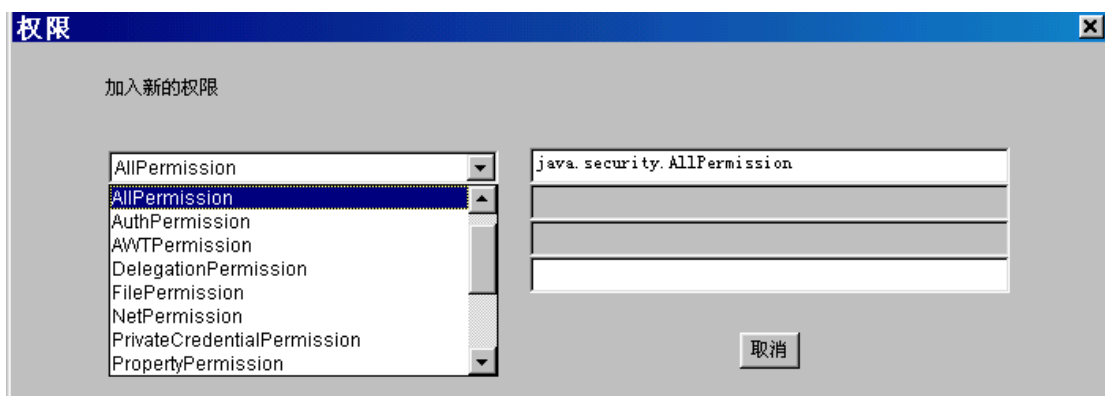


图 8-4 设置权限

(16) 保存设置

设置好权限后，单击“确定”，图 8-2的窗口变成 图 8-5所示，其中显示了已经添加的一个权限：“permission java.security.AllPermission”，如果需要更多的权限，还可以继续单击图 8-5窗口中的“添加权限”按钮。

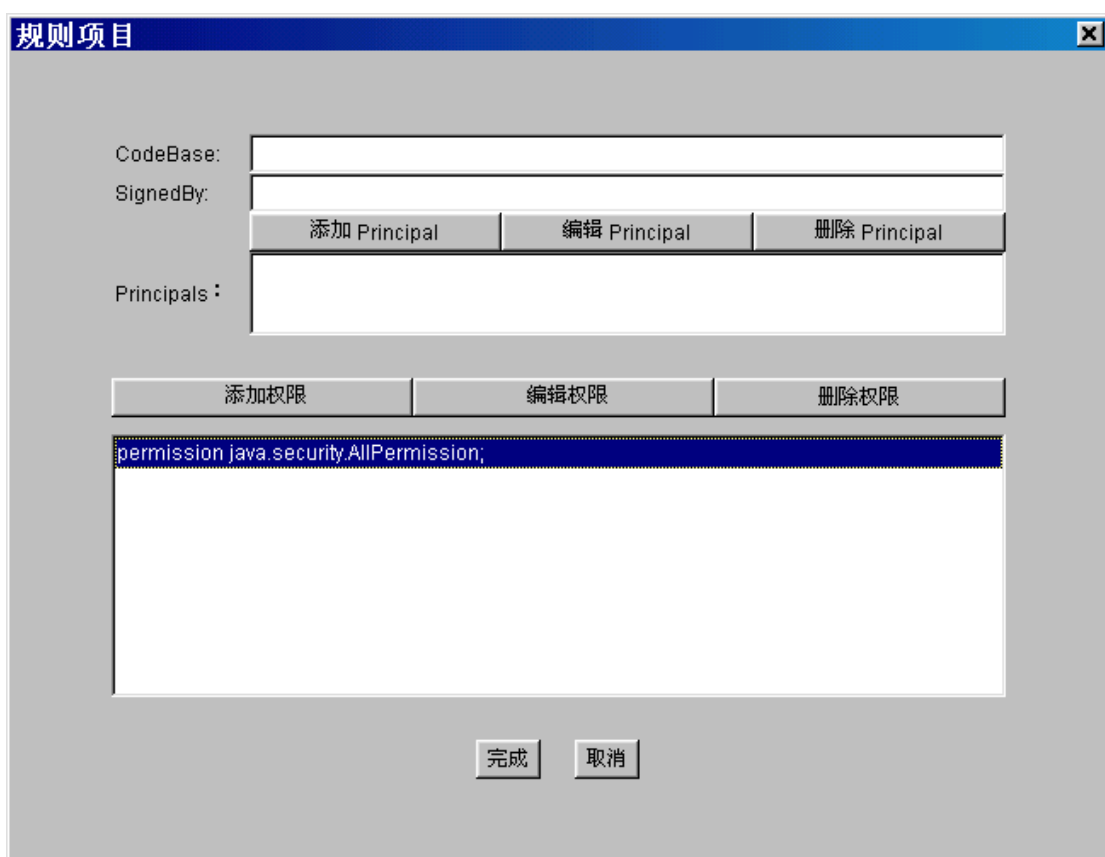


图 8-5 已经添加的权限

最后单击图 8-5窗口中的“完成”按钮，图 8-1的窗口变为 图 8-6所示的窗口。其中显示了刚创建的规则条目：“CodeBase <All>”，表明该项的设置是针对所有代码的。需要时可

选中该条目进行修改或删除。

最后选择“文件/保存”菜单，将设置保存为文件，文件名可以任意设置，不妨设置为：“myall_all.policy”，最后提示规则成功写至文件，单击“确定”按钮，并从“文件/退出”菜单退出规则工具。



图 8-6 针对所有代码的设置

★代码与分析:

本实例不妨仍旧使用 8.1.1 小节的程序。编程思路部分通过策略工具创建的策略文件 myall_all.policy 的内容如下:

```
/* AUTOMATICALLY GENERATED ON Fri Dec 20 11:12:57 CST 2002*/  
/* DO NOT EDIT */  
grant {  
    permission java.security.AllPermission;  
};
```

其中“/*”和“*/”之间的内容是策略工具自动添加的注释，grant{ }中的内容是设置的权限。这里我们允许程序具有所有的权限。

★运行程序

在 8.1.1 小节运行程序的命令中加上一个 java 命令行选项“-Djava.security.policy”即可

以在使用默认的安全管理器的安全设置基础上，使用 myall_all.policy 策略文件中放开的权限。如输入：

```
java -Djava.security.manager -Djava.security.policy=myall_all.policy RunShowFile C:\j2sdk1.4.0\README.txt
```

运行程序，则将不再出现 8.1.1 小节中的“Exception in thread "main" java.security.AccessControlException: access denied (java.io.FilePermission C:\j2sdk1.4.0\README.txt read)”出错信息，而是可以读取 C:\j2sdk1.4.0\README.txt 并将其内容显示出来。

同样，可输入

```
java -Djava.security.manager -Djava.security.policy=myall_all.policy RunShowFile 1.txt
```

运行程序。

8.2.2 允许所有代码具有特定的权限

★ 实例说明

本实例演示了如何创建策略文件，允许所有的代码都可以读取 C:\j2sdk1.4.0 目录下的文件。

★ 编程思路

本实例的场景是：8.1.1 小节的程序编写者向用户提供了两个字节码文件 ShowFile.class 和 RunShowFile.class，并告诉用户运行 RunShowFile 文件则可以显示命令行参数中指定的文件名。但 8.1.1 小节的程序编写者隐瞒了如果用户输入的文件名以“.txt”为后缀，则偷偷读取 c:\autoexec.bat 文件的内容。

当用户得到 ShowFile.class 和 RunShowFile.class 文件后，想使用它显示 C:\j2sdk1.4.0 目录下的所有文件，但又担心程序会不会读写硬盘中的其他文件。于是准备使用默认安全管理器，限制所有的不安全操作，同时编写策略文件放开对 C:\j2sdk1.4.0 目录的读的权限。

和 8.2.1 小节一样通过 policytool 工具编写策略文件，启动程序 授予所有代码所有权限，然后在运行程序时通过 java 命令行选项“-Djava.security.policy”指定使用该策略文件。

启动策略工具并单击“添加规则项目”，再单击“添加权限”进入图 8-3所示的添加权限窗口后，在“许可”栏目下选择“FilePermission”，则该栏目自动填上“java.io.FilePermission”，在目标名称后面填写“c:\j2sdk1.4.0*”，（使用“\”代表文件分隔符，其中“*”代表针对该目录中所有文件，但不包括其子目录中的文件，若使用“-”代替“*”则包括子目录），在“动作”栏目中选择“read”，则该栏目自动天上“read”。如图 8-7所示。

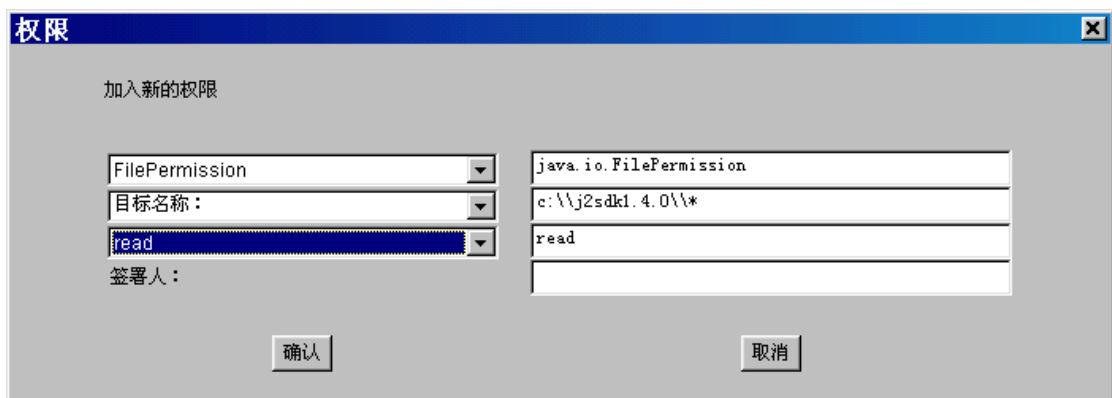


图 8-7 设置允许读特定文件

单击“确认”按钮，出现图 8-8所示的窗口，其中显示了已添加的针对c:\j2sdk1.4.0 目

录的读文件权限，继续单击“完成”，最后保存策略文件，不妨取文件名为“myall_cread.policy”。

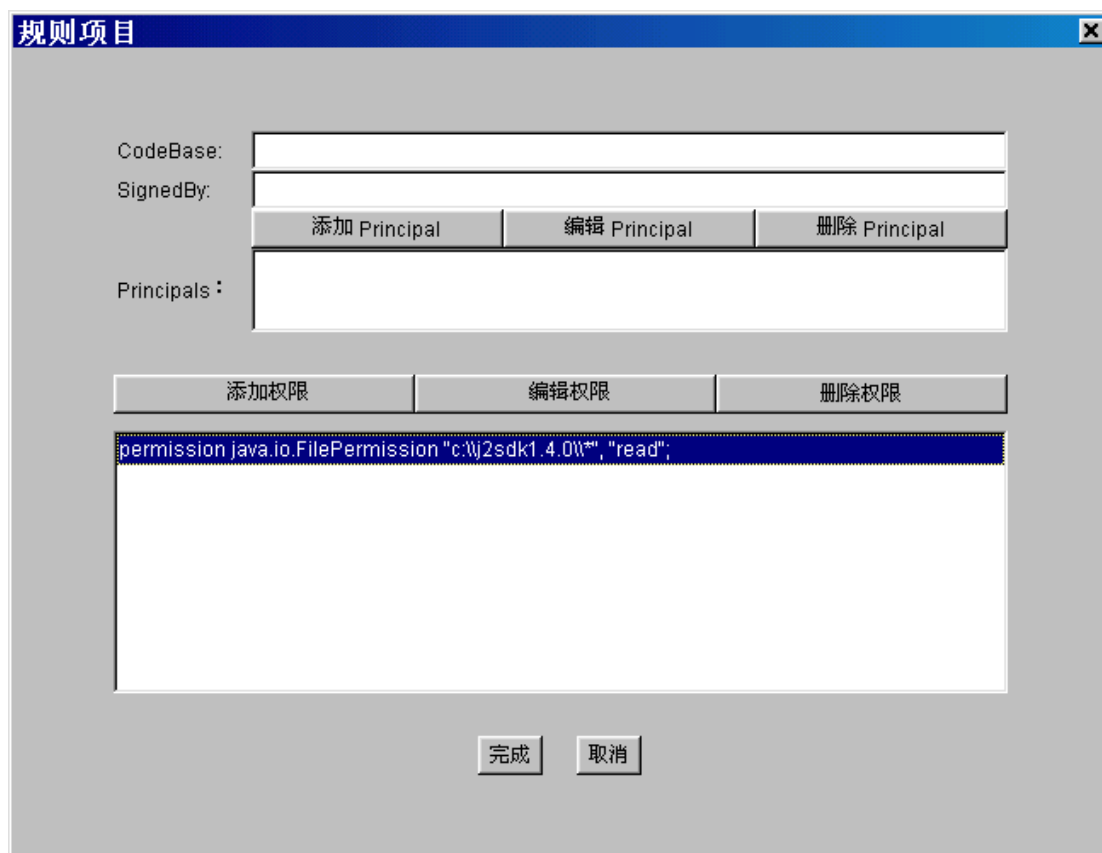


图 8-8 设置允许读特定文件

★代码与分析

本实例不妨仍旧使用 8.1.1 小节的程序。编程思路部分通过策略工具创建的策略文件 myall_cread.policy 的内容如下：

```
/* AUTOMATICALLY GENERATED ON Fri Dec 20 11:12:57 CST 2002 */  
/* DO NOT EDIT */  
grant {  
    permission java.security.AllPermission;  
};
```

★运行程序

同样使用 java 命令行选项“-Djava.security.policy”指定策略文件 myall_cread.policy。在同一行中输入：

```
java -Djava.security.manager -Djava.security.policy=myall_cread.policy RunShowFile  
C:\j2sdk1.4.0\COPYRIGHT
```

运行程序，则程序正常运行，显示 C:\j2sdk1.4.0 目录下 COPYRIGHT 文件的内容。因为尽管使用了默认的管理器限制了所有不安全操作，但通过策略文件 myall_cread.policy 另外指

定了允许读 C:\j2sdk1.4.0 目录下的文件。

如果输入

```
java -Djava.security.manager -Djava.security.policy=myall_cread.policy RunShowFile  
C:\j2sdk1.4.0\README.txt
```

运行程序，由于文件名以“.txt”为后缀，因而触发了编程者设置的陷阱，程序想偷偷读取 c:\autoexec.bat 文件。而在策略文件 myall_cread.policy 中没有授权程序这样做，因此安全检查没有通过，程序显示：

```
Exception in thread "main" java.security.AccessControlException: access denied  
(java.io.FilePermission c:\autoexec.bat read)
```

Java 安全系统成功地阻止了程序的非授权操作。

8.2.3 许所有代码具有多种不同权限

★ 实例说明

本实例演示了如何创建策略文件，允许所有的代码不仅可以读取 C:\j2sdk1.4.0 目录下的文件，而且可以读取 c:\目录下的“autoexec.bat”文件。

★ 编程思路

本实例的场景是 8.1.1 小节的程序编写者向用户提供了两个字节码文件，告诉用户运行 RunShowFile 文件则可以显示命令行参数中指定的文件名。同时程序确实需要读取 c:\autoexec.bat 文件的内容。

用户得到 ShowFile.class 和 RunShowFile.class 文件后，被告知程序还要读取 c:\autoexec.bat 文件的内容。用户觉得 c:\autoexec.bat 文件被该程序读到不会发生安全问题，决定允许程序读取该文件，同时用户想使用该程序显示 C:\j2sdk1.4.0 目录下的所有文件。

用户执行程序时担心程序会不会读写硬盘中的其他文件，于是准备使用默认安全管理器，限制所有的其他不安全操作，同时编写策略文件放开对 C:\j2sdk1.4.0 目录和 C:\autoexec.bat 文件的读的权限。

按照 8.2.2 小节的步骤通过 policytool 工具编写策略文件，在图 8-8 中，继续单击“添加权限”按钮，在出现的窗口中类似地增加对 c:\autoexec.bat 的读的权限，如图 8-9 所示。其中文件分隔符仍用双斜杠“\”表示。

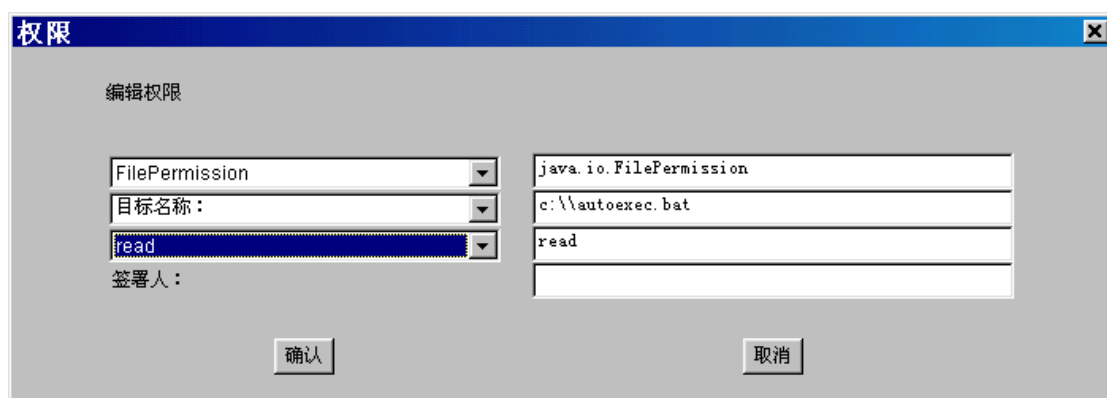


图 8-9 设置对 c:*.bat 的权限

单击“确认”按钮后，出现图 8-10 所示的窗口，其中显示了已添加的两个权限，需要时可分别选中进行修改或删除。最后单击“完成”按钮，并将其保存到文件中，文件名不妨

取 “myall_multi.policy”。

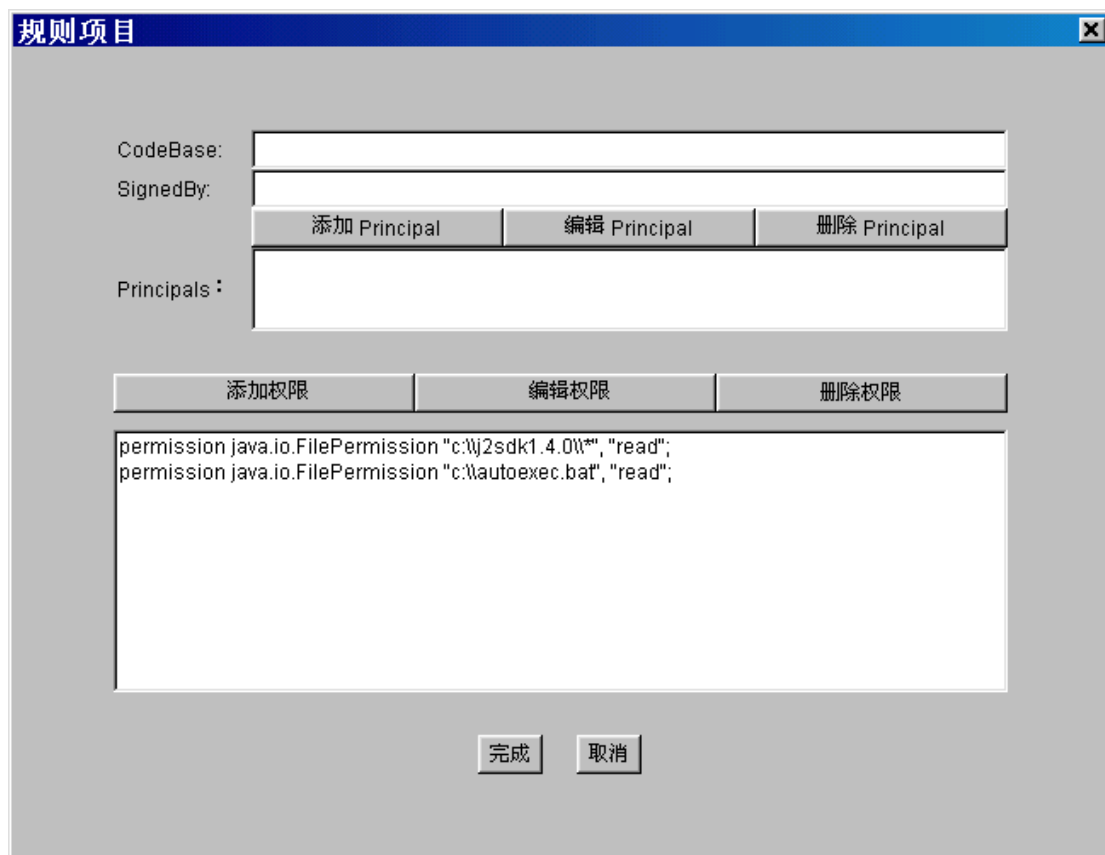


图 8-10 已添加的多个权限

★代码与分析

本实例不妨仍旧使用 8.1.1 小节的程序。编程思路部分通过策略工具创建的策略文件 myall_multi.policy 的内容如下：

```
/* AUTOMATICALLY GENERATED ON Fri Dec 20 14:21:20 CST 2002 */
/* DO NOT EDIT */

grant {
    permission java.io.FilePermission "c:\\j2sdk1.4.0\\*", "read";
    permission java.io.FilePermission "c:\\autoexec.bat", "read";
};
```

★运行程序

同样使用 java 命令行选项 “-Djava.security.policy” 指定策略文件 myall_multi.policy。在同一行中输入：

```
java -Djava.security.manager -Djava.security.policy=myall_multi.policy RunShowFile
C:\\j2sdk1.4.0\\README.txt
```


此时程序可正常显示 C:\j2sdk1.4.0\README.txt 文件的内容。

8.2.4 针对指定目录中的代码的授权

★ 实例说明

前面的例子中策略文件是针对所有代码的，本小节起开始介绍根据代码的位置设置不同的权限。本实例中，代码如果存放在 c:\java 目录或其子目录，则允许读取 C:\j2sdk1.4.0 目录（不包括子目录）下的文件，也可读取 c:\目录下的“autoexec.bat”文件，若存放在 d:\tt 目录中，则只允许读取 C:\j2sdk1.4.0 目录（包括子目录）下的文件，但不允许读取 c:\目录下的“autoexec.bat”文件和其他目录的文件。代码在其他目录中则禁止任何不安全的操作。

★ 编程思路

在添加规则项目时，可以在 CodeBase 项中填入该规则是针对哪个目录中的代码的。这里不妨以 8.2.3 小节中的规则文件为基础进行修改。其步骤如下：

(1) 打开原有文件

启动Policytool后，选择“文件/打开”菜单，选中 8.2.3 小节的myall_multi.policy文件，进入图 8-11所示的窗口。



图 8-11 打开已有的策略文件

(2) 设置规则条目适用的代码位置

双击图 8-11 的“CodeBase<ALL>”条目,进入和图 8-10一样的窗口,在其中“CodeBase”一栏中填上“file:/c:/java/-”。如图 8-12所示。这里, file代表代码所在的位置是本地硬盘, c:/java是代码所在目录, 通配符“-”代表该目录下所有代码(包括子目录)都使用该规则条目。也可以使用“file:/c:/java/*”排除子目录中的代码。

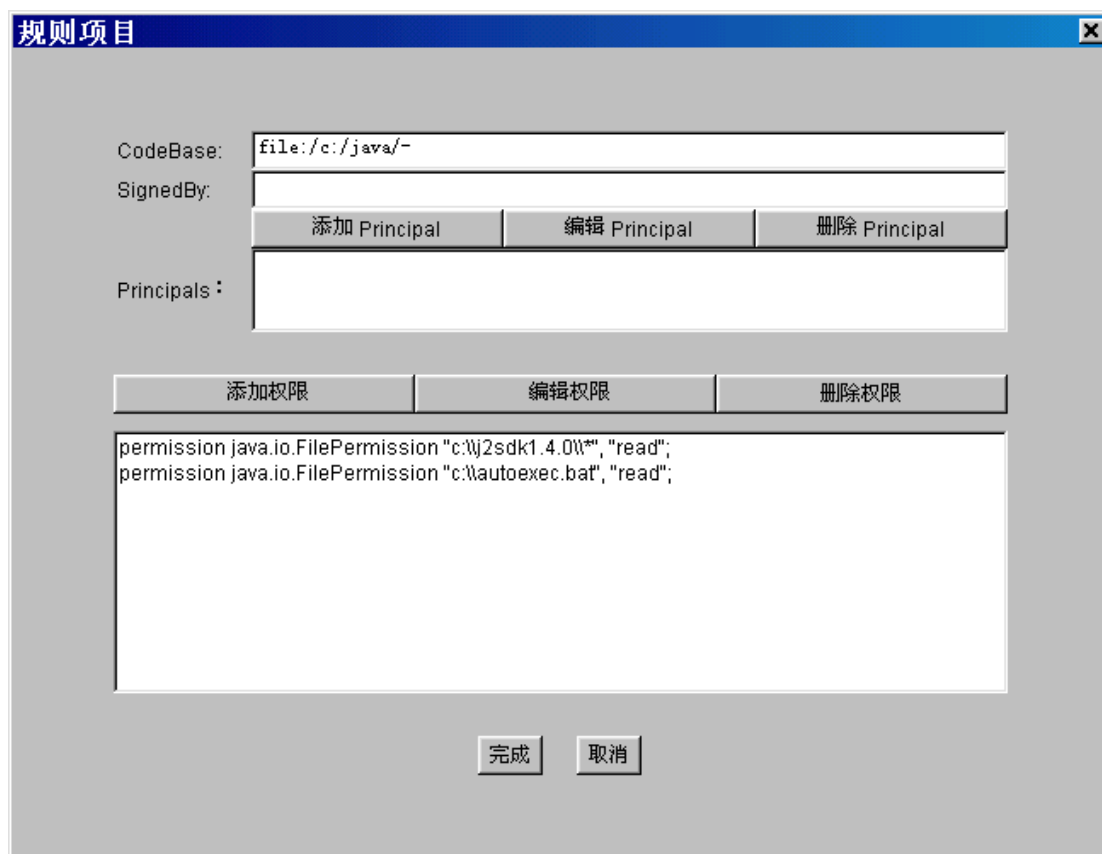


图 8-12 指定代码位置

单击“完成”按钮,出现图 8-13所示窗口,它和图 8-11的窗口类似,只是其中的条目名称“CodeBase<ALL>”已经变成了“codeBase "file:/c:/java/-"”。



图 8- 13 修改条目后的窗口

(3) 添加新的规则条目

单击图 8- 13 的“添加规则条目”按钮，继续添加规则，在“CodeBase”栏目下输入“file:/d:/tt/*”，并单击“添加权限”按钮，类似图 8-9 设置 d:\tt 目录中的代码只允许读取 C:\j2sdk1.4.0 目录及其子目录下的文件（在目标名称中填入“c:\j2sdk1.4.0\”）。最后，新添加的条目如图 8- 14 所示。

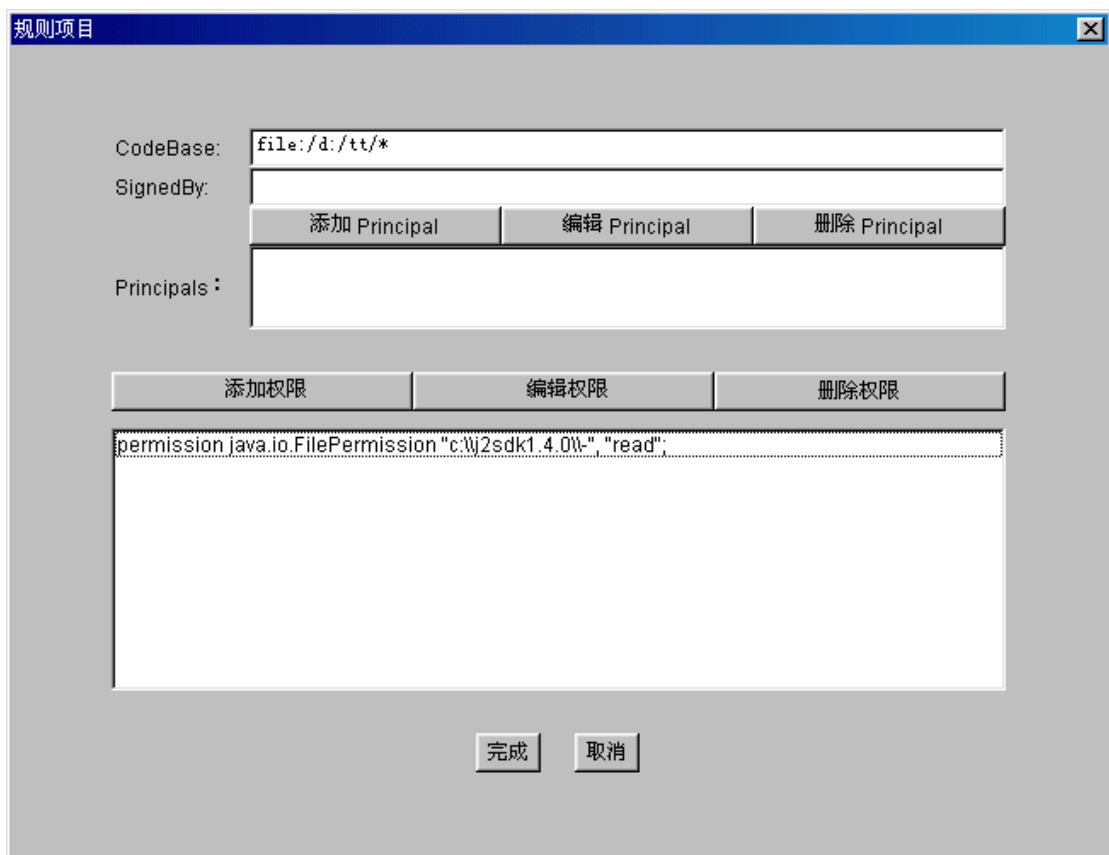


图 8-14 新的规则条目

单击“完成”按钮，图 8-13的窗口变成图 8-15所示。

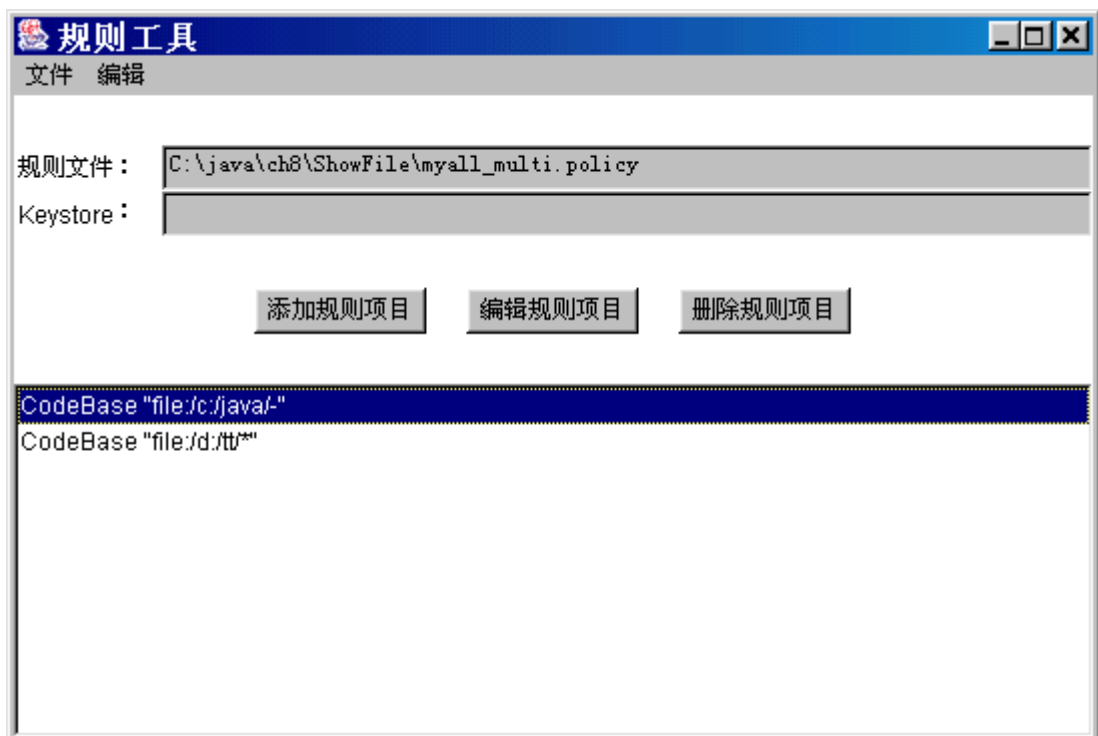


图 8-15 添加条目后的窗口

最后选择“文件/另存为”菜单，将修改后的策略文件保存为“mydir_multi.policy”文件。

★代码与分析

本实例不妨仍旧使用 8.1.1 小节的程序。编程思路部分通过策略工具创建的策略文件 mydir_multi.policy 的内容如下：

```
/* AUTOMATICALLY GENERATED ON Fri Dec 20 18:18:14 CST 2002 */
/* DO NOT EDIT */

grant codeBase "file:/c:/java/-" {
    permission java.io.FilePermission "c:\\j2sdk1.4.0\\*", "read";
    permission java.io.FilePermission "c:\\autoexec.bat", "read";
};

grant codeBase "file:/d:/tt/*" {
    permission java.io.FilePermission "c:\\j2sdk1.4.0\\-", "read";
};
```

其中有多条 grant 项，每个 grant 项对应一条规则条目，grant 后面使用 codeBase 指定该 grant 项是针对何种代码的。

★运行程序

同样使用 java 命令行选项“-Djava.security.policy”指定策略文件 mydir_multi.policy。在 C:\java\ch8\ShowFile 目录下输入：

```
java -Djava.security.manager -Djava.security.policy=mydir_multi.policy RunShowFile
C:\j2sdk1.4.0\README.txt
```

此时程序可正常显示 C:\j2sdk1.4.0\README.txt 文件的内容。

输入

```
java -Djava.security.manager -Djava.security.policy=mydir_multi.policy RunShowFile
C:\j2sdk1.4.0\jre\COPYRIGHT
```

则程序显示对 C:\j2sdk1.4.0\jre\ COPYRIGHT 文件没有读的权限。这是因为策略文件 mydir_multi.policy 中规定了 c:\java 及其子目录中的代码只能读 C:\j2sdk1.4.0 中（不包括子目录）的文件。

若将 RunShowfile.class、ShowFile.class 和 mydir_multi.policy 拷贝到 d:\tt 目录，则再执行

```
java -Djava.security.manager -Djava.security.policy=mydir_multi.policy RunShowFile
C:\j2sdk1.4.0\jre\COPYRIGHT
```

则可以正常显示 C:\j2sdk1.4.0\jre\COPYRIGHT 文件的内容。这是因为策略文件 mydir_multi.policy 中规定了 d:\tt 目录中的代码可以读 C:\j2sdk1.4.0 目录（包括子目录）中的文件。

但若在 d:\tt 目录下执行

```
java -Djava.security.manager -Djava.security.policy=mydir_multi.policy RunShowFile
C:\j2sdk1.4.0\README.txt
```

或

```
java -Djava.security.manager -Djava.security.policy=mydir_multi.policy RunShowFile  
C:\j2sdk1.4.0\jre\README.txt
```

则都显示对 c:\autoexec.bat 文件没有读的权限，这是因为命令行参数中的文件以“.txt”为后缀，程序 RunShowFile 中此时会读取 c:\autoexec.bat 文件，而策略文件 mydir_multi.policy 中对 d:\tt 目录下的代码只允许读 C:\j2sdk1.4.0 目录及其子目录中的文件。

8.2.5 针对从网络下载的代码的授权

★ 实例说明

使用 codeBase 不仅可以针对不同目录中的代码设置不同的权限，而且可以针对从网络下载的不同的代码授予不同的权限。本小节的实例对从 <http://127.0.0.1/> 站点上下载的代码允许读取 c:\j2sdk1.4.0 目录及子目录中所有文件，但不允许读取其他目录中的文件。

编程思路

在添加规则项目时，可以在 CodeBase 项中填入该规则是针对从哪个站点动态下载的代码的，和以前一样可以使用通配符“-”和“*”等。如可在 policytool 工具规则的设置中输入“<http://127.0.0.1/>”，并在添加的权限中允许 CodeBase 为“<http://127.0.0.1/>”代码读 c:\j2sdk1.4.0 目录及其子目录的文件。如图 8-16 所示。

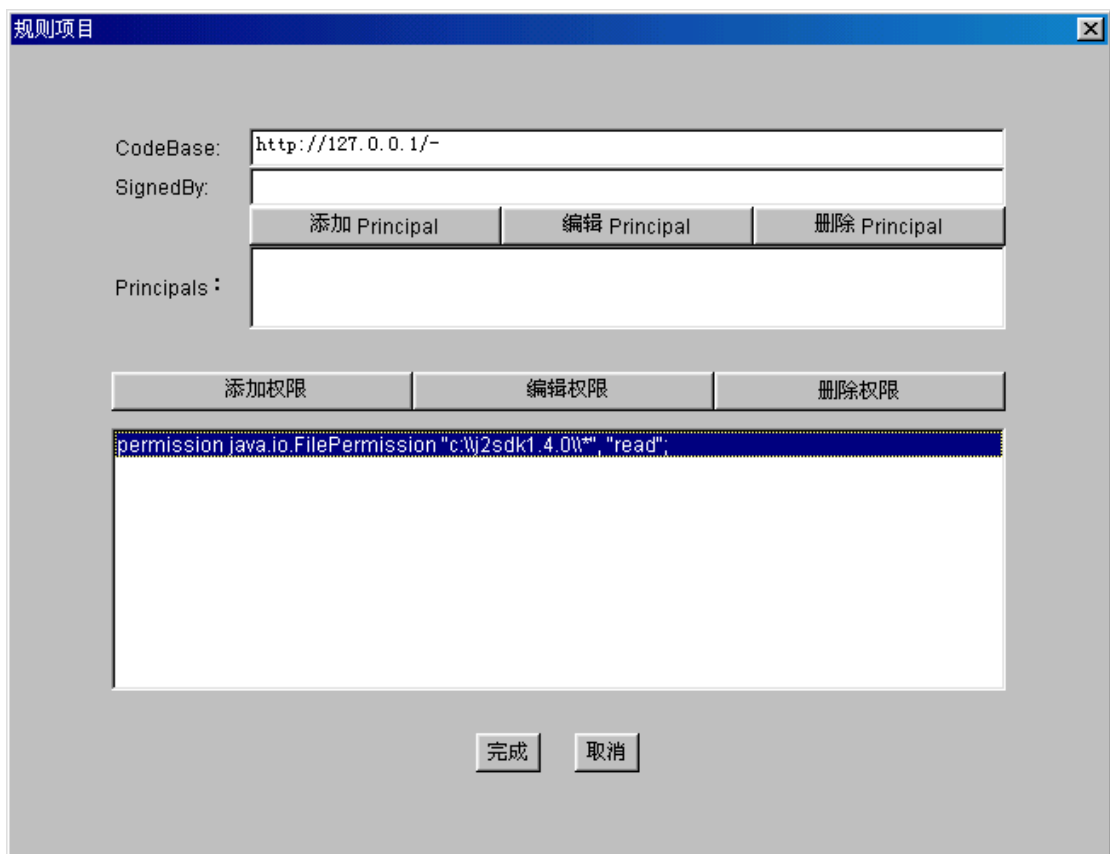


图 8-16 针对从网络下载的代码进行授权

此外，还需要授予 RunURLShowFile.class 的创建 ClassLoader、和 Web 服务器建立连接和接收数据以及对“c:\j2sdk1.4.0*”的读的权限。

★代码与分析

本实例所创建的策略文件如下:

```
/* AUTOMATICALLY GENERATED ON Sun Dec 22 15:01:53 CST 2002*/
/* DO NOT EDIT */

grant codeBase "http://127.0.0.1/-" {
    permission java.io.FilePermission "c:\\j2sdk1.4.0\\*", "read";
};

grant codeBase "file:/c:/java/ch8/url/pro/*" {
    permission java.lang.RuntimePermission "createClassLoader";
    permission java.net.SocketPermission "127.0.0.1", "accept, connect";
    permission java.io.FilePermission "c:\\j2sdk1.4.0\\*", "read";
};
```

在 8.1.1 小节编写了两个类: **ShowFile** 和 **RunShowFile**。本实例使用 **ShowFile** 类, 将其放在一个 Web 服务器上, 供远程下载执行。但对 **RunShowFile** 程序作一些修改, 使它从网络下载 **ShowFile.class** 并执行 **ShowFile.class**, 该程序如下:

```
import java.io.*;
import java.net.*;
import java.lang.reflect.*;

public class RunURLShowFile{
    public static void main(String args[]) throws Exception{
        URL myurl[] = {
            new URL ("http://127.0.0.1/")
        };
        //加载类, 创建对象。
        URLClassLoader x = new URLClassLoader (myurl);
        Class c = x.loadClass ("ShowFile");
        Object ob = c.newInstance();
        //获取所加载的类的 go(String)方法
        Class arg3[] = {(new String()).getClass()};
        Method m3 = c.getMethod("go", arg3 );
        //创建参数
        Object myarg2[] = {args[0]};
        //执行 go(String)(方法
        String s=(String) m3.invoke( ob,myarg2);
        System.out.println(s);
        System.out.println("Over");
    }
}
```

该程序所使用的相关技术见本书 3.2 节。它从 Web 服务器 127.0.0.1 下载 ShowFile.class 并执行之。这里使用的 Web 服务器和运行 RunURLShowFile 程序的计算机是同一台，如果使用其他的 Web 服务器，只要把“127.0.0.1”修改为所使用的 Web 服务器即可。如果 ShowFile.class 不是放在 Web 服务器提供 Web 服务的文档根目录上，则在创建 URL 对象时应带上目录名称。

没有 Web 服务器的读者也可以在一个空目录（如 C:\java\ch8\url\web 目录）下编译运行下面的程序充当 Web 服务器，并将 8.1.1 小节的 ShowFile.class 文件放在该目录供本小节的 RunShowFile 程序下载。

```
import java.net.*;
import java.io.*;

public class MyThreadWebServer2 {
    public static void main(String args[ ])    {
        try {
            ServerSocket ss=new ServerSocket(80);
            System.out.println("Web Server OK");
            while (true) {
                Socket s=ss.accept( );
                Process p=new Process(s);
                Thread t=new Thread(p);
                t.start( );
            }
        } catch (Exception e) {System.out.println(e);}
    }
}

class Process implements Runnable{
    Socket s;
    public Process (Socket s1) {
        s=s1;
    }
    public void run( )
    {
        try {
            PrintStream out = new PrintStream(s.getOutputStream( ));
            BufferedReader in
                = new BufferedReader(new
                    InputStreamReader(s.getInputStream( )));
            String info=in.readLine( );
            System.out.println("now got "+info);
            out.println("HTTP/1.0 200 OK");
            out.println("MIME_version:1.0");
            out.println("Content_Type:text/html");
            // 浏览器请求形如 GET /t/1.html HTTP/1.1
```



```

// sp1, sp2 为第一次和第二次出现空格的位置,
// filename 为从浏览器请求中提取出文件路径和名称 如 t/1.html
int sp1=info.indexOf(' ');
    int sp2=info.indexOf(' ',sp1+1);
String filename=info.substring(sp1+2,sp2);
// 若浏览器请求中无文件名, 则加上默认文件名 index.html
if(filename.equals("") || filename.endsWith("/"))
    filename+="index.html";
System.out.println("Sending "+filename);
// 向浏览器发送文件
File fi=new File(filename);
InputStream fs=new FileInputStream(fi);
int n=fs.available( ); //n 为文件的长度
byte buf[ ]=new byte[1024];
out.println("Content_Length:"+n);
out.println("");
while ((n=fs.read(buf))>=0){
    //一次从文件中读 1024 个 byte 放在 buf 数组中
    //read 的返回值 n 为实际读到的字节数
    out.write(buf,0,n); // 将读到的内容向浏览器输出
}
out.close( );
s.close( );
in.close( );
} catch (IOException e) {
    System.out.println("Exception:"+e);
}
}
}
}

```

★运行程序

先在 C:\java\ch8\url\web 目录执行“java MyThreadWebServer2”，启动 Web 服务器，如果使用的是网络中已有的 Web 服务器则可以省略这一步。

然后执行 RunURLShowFile 程序。如果不使用安全管理器，可输入

```
java RunURLShowFile C:\j2sdk1.4.0\COPYRIGHT
```

运行程序，此时程序可正常显示 C:\j2sdk1.4.0\COPYRIGHT 文件的内容。

使用策略安全管理器后，同样可以使用 java 命令行选项“-Djava.security.policy”指定策略文件 mynet_multi.policy。如在 C:\java\ch8\url\pro 目录下输入：

```
java -Djava.security.manager -Djava.security.policy=mynet_multi.policy
RunURLShowFile C:\j2sdk1.4.0\COPYRIGHT
```

运行程序，此时程序可正常显示 C:\j2sdk1.4.0\COPYRIGHT 文件的内容。

但若输入

```
java -Djava.security.manager -Djava.security.policy=mynet_multi.policy  
RunURLShowFile C:\autoexec.bat
```

运行程序，则显示对 c:\autoexec.bat 没有读的权限。

8.3 使用策略文件基于代码的所有者进行授权

上一节在运行程序时通过策略文件根据代码所在的不同位置允许其访问不同的资源，但实际使用时还经常需要判断代码的所有者，如用户可能不管程序保存在哪个目录，只要程序是甲提供的，就比较信任，而如果程序是由乙提供的，则不怎么信任，想作较多的限制。

本节给出编程者如何对代码进行签名，以及用户如何检验某个代码是否确实是某个人或机构编写的、如何对其进行授权。

8.3.1 编程者对代码进行签名

★ 实例说明

假定 8.1.1 小节的程序是编程者“Liu Fang”编写的，“Liu Fang”拥有 6.1.1 小节得到的密钥库 lfkeystore2 中别名 lf 对应的私钥。

本实例给出编程者“Liu Fang”如何用自己的私钥对自己编写的代码进行签名。

★ 编程思路

J2SDK 提供了 jarsigner 工具来对代码进行签名，签名前需先使用 J2SDK 提供的 jar 工具将所有的代码打包成“.jar”为后缀的文件，签名是针对该“.jar”为后缀的文件进行的。

★ 代码与分析：

本实例仍旧使用 8.1.1 小节的程序。

★ 运行程序

本实例的程序在 C:\java\ch8\sign\demo1\A 目录进行，该目录是编程者“Liu Fang”的工作目录，其中拷贝了编程者在 8.1.1 小节编写并编译后得到的代码：ShowFile.class 和 RunShowFile.class，同时拷贝了编程者“Liu Fang”在 6.1.1 小节得到的密钥库 lfkeystore2。

首先，编程者执行如下命令将自己的代码用 jar 工具打包成一个文件:showfile.jar。

```
C:\java\ch8\sign\demo1\A>jar cvf showfile.jar *.class
```

标明清单(manifest)

增加: RunShowFile.class(读入= 702) (写出= 440) (压缩了 37%)

增加: ShowFile.class(读入= 696) (写出= 432) (压缩了 37%)

然后，编程者使用 jarsigner 工具利用密钥库 lfkeystore2 中别名为 lf 的私钥对 showfile.jar 进行签名。

```
C:\java\ch8\sign\demo1\A>jarsigner -keystore lfkeystore2 showfile.jar lf
```

Enter Passphrase for keystore: wshr.ut

此时，showfile.jar 文件就已经包含了编程者“Liu Fang”签名的信息，用户得到 showfile.jar 文件后可以使用编程者“Liu Fang”的证书检验其签名。为了便于用户检验，编程者应该向用户提供自己的证书，可以通过 keytool 工具导出自己的证书提供给用户：

```
C:\java\ch8\sign\demo1\A>keytool -export -keystore lfkeystore2 -alias lf -file lf.cer
```

输入 keystore 密码: wshr.ut

保存在文件中的认证 <lf.cer>

这样, 编程者只要将 showfile.jar 和 lf.cer 文件一起提供给用户即可。

8.3.2 用户检验已签名的代码

★ 实例说明

本实例给出用户得到 8.3.1 小节中编程者“Liu Fang”提供的已签名代码 showfile.jar 和签名者的证书 lf.cer 文件后, 如何检验已签名的代码。

★ 编程思路

J2SDK 提供了 jarsigner 工具不仅可以用来对代码进行签名, 而且可以检验签名后的代码, 使用其命令行选项 -verify 即可。

★ 运行程序

本实例的程序在 C:\java\ch8\sign\demo1\B 目录进行, 该目录是用户的工作目录, 其中拷贝了编程者在 8.3.1 小节提供的 showfile.jar 和 lf.cer 两个文件。

(1) 检查签名和证书

为了检查 showfile.jar 确实是“Liu Fang”编写的, 可以通过 jarsigner 工具查看 showfile.jar 中各个文件签发者的证书。

```
C:\java\ch8\sign\demo1\B>jarsigner -verify -verbose -certs Showfile.jar
```

```
205 Mon Dec 23 21:28:34 CST 2002 META-INF/MANIFEST.MF
258 Mon Dec 23 21:28:34 CST 2002 META-INF/LF.SF
1591 Mon Dec 23 21:28:34 CST 2002 META-INF/LF.RSA
    0 Mon Dec 23 21:24:12 CST 2002 META-INF/
sm      702 Sun Dec 22 15:10:52 CST 2002 RunShowFile.class

X.509, CN=Liu Fang, OU=Packaging, O=Shanghai University, L=ZB, ST=Shanghai, C=CN
X.509, CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai,
C=CN

sm      696 Sun Dec 22 15:10:52 CST 2002 ShowFile.class

X.509, CN=Liu Fang, OU=Packaging, O=Shanghai University, L=ZB, ST=Shanghai, C=CN
X.509, CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai,
C=CN

s = signature was verified
m = entry is listed in manifest
k = at least one certificate was found in keystore
i = at least one certificate was found in identity scope
```

```
jar verified.
```

其中显示了 `showfile.jar` 文件中所包含的所有文件。文件开头几个 `META-INF` 目录中的文件分别是清单文件、签名文件和签名块文件。后面几个 “.class” 为后缀的文件是被打包在 `jar` 文件中的代码：`RunShowFile.class` 和 `ShowFile.class`。这两个文件名前面都显示了 `sm`，其中 `s` 代表该文件的签名已经验证通过，`m` 代表在清单文件中有该条目。

如果建立一个临时目录，如 `C:\java\ch8\sign\demo1\B\tmp`，拷贝 `showfile.jar`，在命令行中运行 `jar xvf showfile.jar`，则可以将这些文件解包，可查看其详细内容。

如清单文件 `Manifest.mf` 的内容为：

```
Manifest-Version: 1.0
Created-By: 1.4.0 (Sun Microsystems Inc.)

Name: RunShowFile.class
SHA1-Digest: 0GrV5QDQQ9SI6jfoUwLzdo+JfDA=

Name: ShowFile.class
SHA1-Digest: v71hEuyFYR6buayWQfZEaRg+Jko=
```

该文件的名字是固定的，文件开头两行是使用 `jarsigner` 工具签名前就有的，后面两行是 `jar` 文件中被打包的文件的名称、所使用的消息摘要算法及消息摘要的值。这里的消息摘要是指针对应的文件（如 `RunShowFile.class`）本身的二进制内容的。

签名文件 `Lf.sf` 的内容为：

```
Signature-Version: 1.0
Created-By: 1.4.0 (Sun Microsystems Inc.)
SHA1-Digest-Manifest: wSSB0phdT9ExGUNh1a8QkpbSc6E=

Name: RunShowFile.class
SHA1-Digest: isY25Lt5M7QazQjKXyxow5P3f2g=

Name: ShowFile.class
SHA1-Digest: iFUyhVMWpxAQ1V8e85TV0HgvSmg=
```

该文件的名字是根据签名时所用的别名自动生成的，如果别名的名称长度超过 8 个字符则只取前 8 个字符，如果别名名称不是字母、数字、下划线或连字符中的一种，则替换成下划线。

签名文件开头为针对清单文件的消息摘要，后面两段分别对应清单文件中的两项，是针对清单文件中的三段内容（如 `Name: RunShowFile.class`、`SHA1-Digest: 0GrV5QDQQ9SI6jfoUwLzdo+JfDA=`）的摘要。

签名块文件 `Lf.rsa` 存放对签名文件的签名，同时包含了用于验证签名的证书或证书链。

(2) 检验证书

为了检验证书是否是值得信任，必须指定用户信任哪些证书，如果用户信任 8.3.1 小节编程者 “Liu Fang” 提供的证书 `lf.cer`，或其签发者 CA “Xu Yingxiao” 的证书，则可以检验通过。

可以使用 7.1.2 小节得到的密钥库 `clienttrust`，该密钥库中包含了签发 `lf.cer` 的 “Xu

Yingxiao”的证书。也可使用如下命令将编程者“Liu Fang”的证书 lf.cer 导入密钥库：

```
C:\java\ch8\sign\demo1\B>keytool -import -file lf.cer -keystore clienttrustlf -alias lf
-storepass 123456
```

```
Owner: CN=Liu Fang, OU=Packaging, O=Shanghai University, L=ZB, ST=Shanghai, C=CN
```

```
发照者: CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai, C=CN
```

```
序号: 3deed053
```

```
有效期间: Thu Dec 05 12:04:35 CST 2002 至: Mon Feb 21 12:04:35 CST 2011
```

```
认证指纹:
```

```
MD5: D3: 7E: C0: 72: 5D: 41: 46: CA: 7A: 8E: 85: 21: 1B: DA: 89: 0F
```

```
SHA1: C8: 85: 45: B7: A8: 37: 1F: 23: DE: A3: C1: DF: A1: B4: 83: C3: B9: F1: B7: FA
```

```
信任这个认证? [否]: 是
```

```
认证已添加至 keystore 中
```

则用户在使用 jarsigner 工具检验某个 jar 文件是否确实值得信任时，可以通过命令行选项 -keystore 指定包含用户所信任的证书的密钥库，如 clienttrust 或 clienttrustlf。可输入如下命令：

```
jarsigner -verify -verbose -keystore clienttrust showfile.jar
```

```
或 jarsigner -verify -verbose -keystore clienttrustlf showfile.jar
```

屏幕输出都为：

```
205 Mon Dec 23 21:28:34 CST 2002 META-INF/MANIFEST.MF
258 Mon Dec 23 21:28:34 CST 2002 META-INF/LF.SF
1591 Mon Dec 23 21:28:34 CST 2002 META-INF/LF.RSA
0 Mon Dec 23 21:24:12 CST 2002 META-INF/
smk 702 Sun Dec 22 15:10:52 CST 2002 RunShowFile.class
smk 696 Sun Dec 22 15:10:52 CST 2002 ShowFile.class
```

```
s = signature was verified
```

```
m = entry is listed in manifest
```

```
k = at least one certificate was found in keystore
```

```
i = at least one certificate was found in identity scope
```

```
jar verified.
```

其中 RunShowFile.class 和 ShowFile.class 前面都标志了“smk”，其中 k 表示在密钥库中包含了对应于签发 showfile.jar 时所用私钥的证书。

8.3.3 针对签名者进行授权

★ 实例说明

8.3.1 小节中编程者“Liu Fang”将代码 showfile.jar 签名后提供给用户，8.3.2 小节将编

程者的证书 lf.cer 导入了密钥库 clienttrustlf，该密钥库的 lf 条目包含了编程者的证书，本实例介绍如何针对代码的签名者（所有者）进行授权。

★ 编程思路

使用 policytool 工具针对签名者进行授权时，添加规则和以前一样，但要指明所添加的规则是针对哪个代码拥有者的，即代码签发者的证书对应于哪个密钥库的哪个条目。具体步骤如下：

（1）指定密钥库

为了判断签名者，首先要指定包含签名者的证书的密钥库。在policytool工具的窗口中选择“编辑/更改KeyStore”菜单，如图 8- 17所示，可以出现图 8- 18所示窗口，在其中的“新KeyStore URL”栏目中可以输入“file:c:/java/ch8/sign/demo1/b/clienttrustlf”，指定密钥库的名称和路径。密钥库也可以放在网上用http方式来访问。在其中“新KeyStore类型”后面输入密钥库类型，密钥库clienttrustlf在创建时没有指定类型，因此是默认的类型“JKS”。

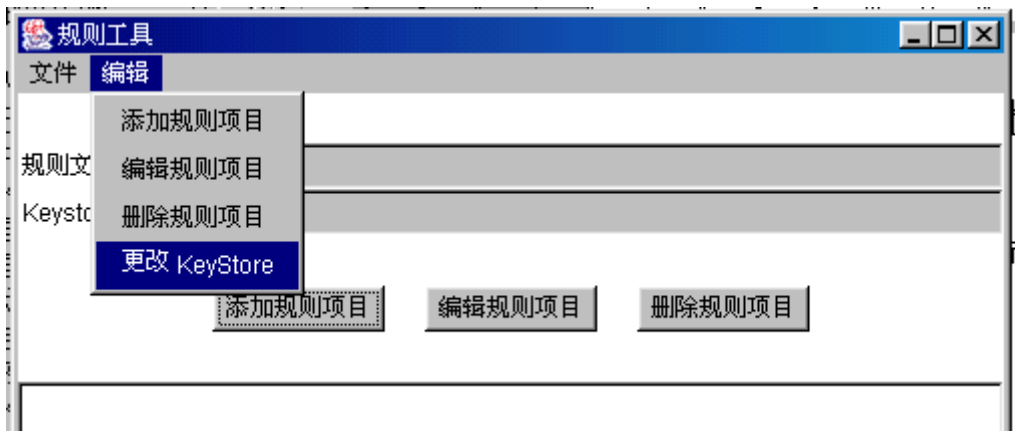


图 8- 17更改 KeyStore

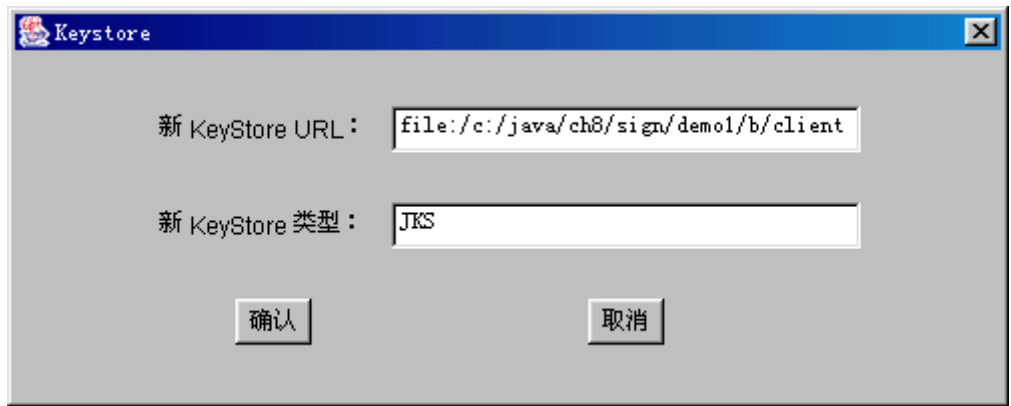


图 8- 18指定密钥库

（2）指定规则针对哪个签发者

图 8- 18的窗口中单击“确定”按钮后，回到图 8- 17窗口，单击“添加规则项目”按钮，出现图 8- 19所示窗口，在其中的“SignedBy”栏目后输入签发者的证书在上一步所设置的密钥库中的别名。在 8.3.2 小节将编程者“Liu Fang”的证书导入密钥库时使用的别名是“lf”，

因此这里可输入“lf”。如果代码有多个签发者，这里可以输入多个别名，之间用逗号隔开。

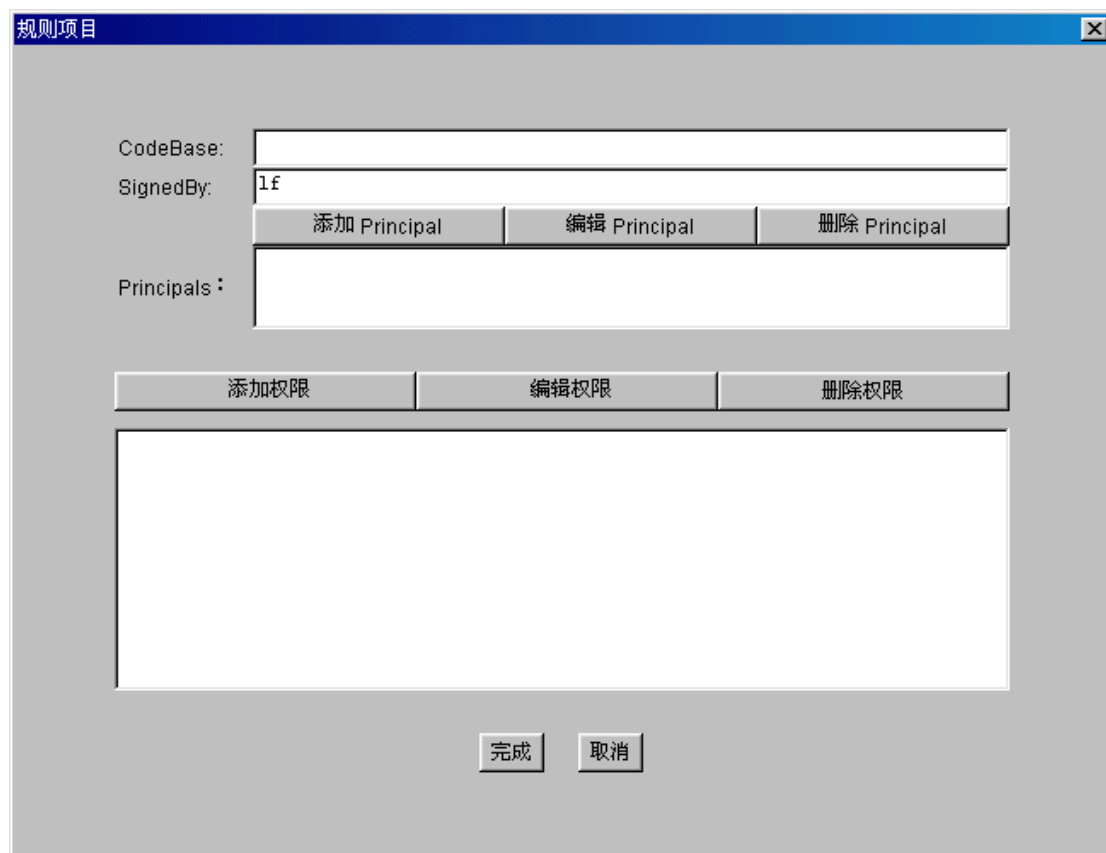


图 8-19 指定签名者

(3) 添加权限

单击图 8-19 窗口中“添加权限”按钮，类似 8.2 节授予各种权限。如可和 8.2.2 小节图 8-7 一样允许其读取 c:\jdk1.4.0 目录下所有文件。

最后不妨将其保存为 mysign_dir.policy 文件。

★代码与分析：

本实例生成的策略文件 mysign_dir.policy 内容如下：

```
/* AUTOMATICALLY GENERATED ON Tue Dec 24 21:48:49 CST 2002 */
/* DO NOT EDIT */

keystore "file:/c:/java/ch8/sign/demo1/b/clienttrustlf", "JKS";

grant signedBy "lf" {
    permission java.io.FilePermission "c:\\jdk1.4.0\\*", "read";
};
```

其中开头使用“keystore”指明所使用的密钥库，grant 后面使用“signedBy”指明该 grant 项是针对何种代码的。

★运行程序

本实例的程序在 C:\java\ch8\sign\demo1\B 目录进行, 该目录是用户的工作目录, 其中包含编程者在 8.3.1 小节提供的签名的代码 showfile.jar 文件和包含信任证书的密钥库 clienttrust 或 clienttrustlf。

(1) 直接运行程序

编程者可以直接运行程序:

```
java -classpath showfile.jar RunShowFile C:\j2sdk1.4.0\COPYRIGHT
```

此时没有使用安全管理器, 也没有检验 showfile.jar 是否确实是“Liu Fang”编写的。由于签名的代码是打包成 jar 文件提供的, 因此运行时需要通过命令行选项 -classpath 指定 jar 文件的名称和路径。

(2) 使用策略文件

将如下命令放入批处理文件, 如 1.bat, 执行后将可以正常显示 C:\j2sdk1.4.0\COPYRIGHT 文件的内容。

```
java -Djava.security.manager -Djava.security.policy=mysign-dir.policy  
-classpath showfile.jar RunShowFile C:\j2sdk1.4.0\COPYRIGHT
```

若将如下命令放入批处理文件, 如 2.bat, 则执行后将显示对 c:\autoexec.bat 没有读的权限。因为在策略文件中未作此授权。

```
java -Djava.security.manager -Djava.security.policy=mysign-dir.policy  
-classpath showfile.jar RunShowFile C:\j2sdk1.4.0\readme.txt
```

8.4 定义特权代码

在本章前面各节的例子中, RunShowFile.class 调用了 ShowFile.class, 这两个类在同一个包中, 同时进行授权。在编程时更常见的是不同的代码之间相互调用, 如可能甲编写了 ShowFile.java 程序, 并将其定义在 myp 包中, 提供给另一个编程者乙使用。而乙在编写 RunShowFile.java 程序时调用了甲提供的 myp 包中的 ShowFile 类。

用户对编程者甲和乙的信任程度往往不同, 因而可能某些权限只给甲而不给乙, 而另外一些权限可能只给乙而不给甲。此时这部分代码应该以特权方式运行, 本节介绍其编程方法。

8.4.1 不同代码之间的调用和授权

★ 实例说明

本实例中, 编程者“Liu Fang”编写了 ShowFile2.java 程序, 该代码需要临时访问 c:\autoexec.bat 文件。另一个编程者编写代码 RunShowFile2.java, 其中调用了编程者“Liu Fang”编写的类 ShowFile2。

本实例给出了几种不同的策略文件, 通过其运行效果可以看出定义特权代码的必要性。

★ 编程思路

本实例中代码 ShowFile2 需要临时访问 c:\autoexec.bat 文件, 因此需要对 c:\autoexec.bat 文件读的权限。代码 RunShowFile2 中创建 ShowFile2 对象, 执行其方法 go()。执行时通过代码 RunShowFile2 的命令行参数传入文件名, 进一步传递给 ShowFile2 对象的 go() 方法,

在 go()方法中读取文件内容。因此代码 ShowFile2 又需要对命令行参数指定的文件的读的权限

Java 中如果一个线程经过多个代码如 A,B,C,D,E 执行到某个步骤时需要某个权限,则该线程执行到该步骤所经过的所有代码(A,B,C,D,E)一般应该都具有相应的权限才可以正常运行。因此在本实例中尽管读取文件的具体代码都是在代码 ShowFile2 中执行的,但是用户对调用者 RunShowFile2 也必须授予相同权限才能够正常运行。

因此本实例设计了几种不同的策略文件,分别授予代码 ShowFile2 和 RunShowFile2 不同的权限,查看其运行效果。

★代码与分析:

本实例中编程者“Liu Fang”编写的代码如下:

```
package myp;

import java.security.*;
import java.io.*;

public class ShowFile2{

    public String go(String name) throws IOException{

        String s;
        String content="";
        BufferedReader in;

        in = new BufferedReader(new FileReader(name));
        while ((s = in.readLine( )) != null) {
            content+=s+"\n";
        }
        in = new BufferedReader(new FileReader("c:\\autoexec.bat"));
        while ((s = in.readLine( )) != null) {
            //...
            //临时用于特殊用途
        }
        return content;
    }
}
```

程序开头使用“package myp”将该类定义在 myp 包中。在其 go()方法中,除了读取方法参数中指定的文件外,还需要读取 c:\autoexec.bat 文件,临时用作某种特殊用途。此处没有将其定义为特权代码。

另一个编程者编写的调用 ShowFile2 类的代码如下:

```
import myp.*;
import java.io.*;

public class RunShowFile2{

    public static void main(String args[]) throws IOException{

        ShowFile2 t=new ShowFile2();
        String s=t.go(args[0]);
        System.out.println(s);
    }
}
```

```

        System.out.println("Over");
    }
}

```

程序开头使用“import myo.*”语句以使用编程者“Liu Fang”提供的 myp 包中的类，然后创建 ShowFile2 对象，执行其 go() 方法，方法返回值是命令行参数中指定的文件的内容。

本实例使用的几种策略文件如下：

8.4.1.1.policy 文件，两个代码授予相同权限：

```

/* AUTOMATICALLY GENERATED ON Wed Dec 25 17:35:33 CST 2002*/
/* DO NOT EDIT */

keystore "clienttrustlf", "JKS";

grant signedBy "lf" {
    permission java.io.FilePermission "c:\\autoexec.bat", "read";
    permission java.io.FilePermission "c:\\j2sdk1.4.0\\*", "read";
};

grant codeBase "file:/c:/java/ch8/sign/demo2/User/*" {
    permission java.io.FilePermission "c:\\autoexec.bat", "read";
    permission java.io.FilePermission "c:\\j2sdk1.4.0\\*", "read";
};

```

8.4.1.2.policy 文件，将 RunShowFile2 代码对 c:\autoexec.bat 文件读的权限去掉。

```

/* AUTOMATICALLY GENERATED ON Wed Dec 25 17:35:33 CST 2002*/
/* DO NOT EDIT */

keystore "clienttrustlf", "JKS";

grant signedBy "lf" {
    permission java.io.FilePermission "c:\\autoexec.bat", "read";
    permission java.io.FilePermission "c:\\j2sdk1.4.0\\*", "read";
};

grant codeBase "file:/c:/java/ch8/sign/demo2/User/*" {
    permission java.io.FilePermission "c:\\j2sdk1.4.0\\*", "read";
};

```

8.4.1.3.policy 文件，将 ShowFile2 代码对 c:\j2sdk1.4.0 目录下的文件读的权限去掉。

```

/* AUTOMATICALLY GENERATED ON Wed Dec 25 17:35:33 CST 2002*/
/* DO NOT EDIT */

keystore "clienttrustlf", "JKS";

```

```
grant signedBy "lf" {
    permission java.io.FilePermission "c:\\autoexec.bat", "read";
};

grant codeBase "file:/c:/java/ch8/sign/demo2/User/*" {
    permission java.io.FilePermission "c:\\autoexec.bat", "read";
    permission java.io.FilePermission "c:\\j2sdk1.4.0\\*", "read";
};
```

★运行程序

(1) 编程者“Liu Fang”的工作

假定编程者“Liu Fang”在 C:\java\ch8\sign\demo2\A 目录工作。其中包含了该编程者所编写的程序 ShowFile2.java 和编程者“Liu Fang”的密钥库 lfkeystore2，密钥库的条目 lf 中存放着编程者“Liu Fang”的私钥和数字证书。

输入

```
javac -d . ShowFile2.java
```

编译程序，编译后的代码将根据文件 ShowFile2.java 中的包的名字放在当前目录的 myp 子目录中。

然后输入

```
jar cvf showfile2.jar myp\*.class
```

将编程者“Liu Fang”的代码打包成 showfile2.jar 文件。

再输入

```
jarsigner -keystore lfkeystore2 -keypass wshr.ut -storepass wshr.ut showfile2.jar lf
```

使用密钥库 lfkeystore2 中 lf 条目对应的私钥签名 showfile2.jar 文件。这里将密码 wshr.ut 直接通过 jarsigner 命令行选项给出了，也可以和 8.3.1 小节一样以交互方式输入。

然后执行

```
keytool -export -keystore lfkeystore2 -alias lf -file lf.cer -storepass wshr.ut
```

将编程者的证书导出到文件 lf.cer。

最后将 showfile2.jar 和 lf.cer 文件提供给其他编程者（如编程者 B）使用，如可以拷贝到 C:\java\ch8\sign\demo2\B 目录。

(2) 编程者 B 的工作

假定编程者“Liu Fang”在 C:\java\ch8\sign\demo2\B 目录工作。其中包含了编程者“Liu Fang”提供给他的软件包 showfile2.jar，编程者 B 自己编写了 RunShowFile2.java 程序。

输入如下命令编译程序：

```
javac -classpath showfile2.jar;. RunShowFile2.java
```

其中 javac 命令行选项 -classpath 指定程序中所使用的软件包的路径，多个软件包用分号隔开。这里分号后面加上一个点号代表当前目录，这样 RunShowFile2.java 中如果用到一些非系统的类，将从 showfile2.jar 和当前目录中去寻找。

如果编程者 B 的程序也定义在某个包中，和编程者“Liu Fang”一样，也可以在编译时加上一个 javac 命令行参数 -d。

最后编程者 B 将自己的软件 RunShowFile2.class 和类库 showfile2.jar 以及类库签名者的证书文件 lf.cer 提供给用户。

如果需要，编程者 B 也可以和编程者“Liu Fang”一样对自己的代码打包并签名。

(3) 用户运行程序

假定编程者用户在 C:\java\ch8\sign\demo2\User 目录工作。其中包含了编程者 B 提供文件 RunShowFile2.class、showfile2.jar 和 lf.cer。编程者 B 声称 RunShowFile2.class 是自己编写的，showfile2.jar 是编程者“Liu Fang”签发的，lf.cer 是编程者“Liu Fang”的证书文件。

用户对编程者“Liu Fang”非常熟悉，用户也可能以前就有编程者“Liu Fang”的证书，或者通过其他可靠的方式得到了编程者“Liu Fang”的证书，或者编程者““Liu Fang”的证书”lf.cer 是用户信任的 CA 签发的。这样，和 8.3.2 小节类似，用户可以检验出 showfile2.jar 确实是值得信任的。

然后用户将 lf.cer 导入自己的密钥库，如

```
C:\java\ch8\sign\demo2\User> keytool -import -file lf.cer -keystore clienttrustlf  
-alias lf -storepass abcdefg
```

```
Owner: CN=Liu Fang, OU=Packaging, O=Shanghai University, L=ZB, ST=Shanghai, C=CN
```

```
发照者: CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai, C=CN  
序号: 3deed053
```

```
有效期间: Thu Dec 05 12:04:35 CST 2002 至: Mon Feb 21 12:04:35 CST 2011
```

```
认证指纹:
```

```
MD5: D3: 7E: C0: 72: 5D: 41: 46: CA: 7A: 8E: 85: 21: 1B: DA: 89: 0F
```

```
SHA1: C8: 85: 45: B7: A8: 37: 1F: 23: DE: A3: C1: DF: A1: B4: 83: C3: B9: F1: B7: FA
```

```
信任这个认证? [否]: 是
```

```
认证已添加至 keystore 中
```

用户创建 C:\java\ch8\sign\demo2\package 目录，将当前目录的 showfile2.jar 移动到该目录。这里之所以不将 showfile2.jar 放在当前目录，是因为如果将其放在当前目录，则策略文件针对 file:c:/java/ch8/sign/demo2/User/* 的授权就既包含 showfile2.jar 又包含 RunShowFile2.class 文件了。

将如下内容输入 1.bat 文件，

```
java -Djava.security.manager -Djava.security.policy=8.4.1.1.policy  
-classpath ../package/showfile2.jar;. RunShowFile2 C:\j2sdk1.4.0\readme.txt
```

执行后将可以正确显示 C:\j2sdk1.4.0\readme.txt 文件的内容。

将 8.4.1.1.policy 文件中任何一个权限删除，运行时都将显示缺少相应的权限。如执行如下命令：

```
java -Djava.security.manager -Djava.security.policy=8.4.1.2.policy  
-classpath ../package/showfile2.jar;. RunShowFile2 C:\j2sdk1.4.0\readme.txt
```

将显示对 c:\autoexec.bat 没有读的权限。

又如执行如下命令：

```
java -Djava.security.manager -Djava.security.policy=8.4.1.3.policy  
-classpath ../package/showfile2.jar;. RunShowFile2 C:\j2sdk1.4.0\readme.txt
```

将显示对 C:\j2sdk1.4.0\readme.txt 没有读的权限。

由以上运行可见，读取 c:\autoexec.bat 文件的操作虽然只在编程者“Liu Fang”代码

ShowFile2 中执行，但由于执行该操作时是通过编程者 B 的代码 RunShowFile2 调用的，因此两个代码都需要对 c:\autoexec.bat 文件的读的权限。

如果编程者“Liu Fang”考虑到自己的信任度大一些，用户可能只允许“Liu Fang”的代码读取 c:\autoexec.bat 文件，而不允许编程者 B 编写的代码读取 c:\autoexec.bat 文件，则应如下一小节所示将代码 ShowFile2 中读取 c:\autoexec.bat 文件的部分以特权方式运行。

8.4.2 使用 doPrivileged() 方法定义特权代码

★ 实例说明

本实例中，编程者“Liu Fang”编写了 ShowFile3.java 程序，该代码需要临时访问 c:\autoexec.bat 文件。另一个编程者编写代码 RunShowFile3.java，其中调用了编程者“Liu Fang”编写的类 ShowFile3。

编程者“Liu Fang”在编写 ShowFile3.java 程序时考虑到访问 c:\autoexec.bat 文件只是自己的临时需要，用户可能不允许其调用者访问 c:\autoexec.bat 文件，因此决定以特权方式运行访问 c:\autoexec.bat 文件。本实例给出其编程方法。

★ 编程思路

Java.Security 包中的 AccessController 类提供了静态方法 doPrivileged()，它可以以特权方式运行指定的代码。

方法 doPrivileged() 的参数是接口类型 java.security.PrivilegedAction，该接口只有一个方法需要实现：public Object run()。需要以特权方式运行的代码放在该方法中即可。

为了使调用者能够向以特权方式运行的代码传入文件名等参数，实现 java.security.PrivilegedAction 接口的类中定义了构造器来传入参数。

为了调用者能够获取特权代码的执行结果，实现 java.security.PrivilegedAction 接口的类中定义了获取执行结果的方法。

编写实现 java.security.PrivilegedAction 接口的类的具体步骤为：

(17) 编写构造器传入参数

```
public Mypriv(String fname) {  
    filename = fname;  
}
```

分析：其中 filename 为成员变量，特权代码可以直接访问 filename 变量。调用者从构造器中传入要特权读取的文件名，如“c:\autoexec.bat”，赋值给成员变量 filename 后，

(18) 实现 run() 方法

```
public Object run() {  
    //...  
    //特权代码，执行结果通过变量 value 返回  
    return value;  
}
```

分析：在 run() 方法中编写需要特权运行的代码，其中可利用构造器传入的参数 filename，如可通过 filename 创建文件输入流，读取指定的文件的。

(19) 返回特权代码执行结果

```
public String getValue() {
```

```
        return value;
    }
    分析: 返回上一步的变量 value。
```

编程者“Liu Fang”在编写了上面的代码后, 就可以修改 8.4.1 小节的 ShowFile2.java 程序, 将其中的

```
in = new BufferedReader(new FileReader("c:\\autoexec.bat"));
while ((s = in.readLine()) != null) {
    //...
    //临时用于特殊用途
}
```

替换为:

```
Mypriv mp=new Mypriv("c:\\autoexec.bat");
AccessController.doPrivileged(mp);
String sp = mp.getValue();
content+="-----privileged-----"+sp;
```

其中的 Mypriv 即实现 java.security.PrivilegedAction 接口的类, 创建其对象后, 将其作为参数传递给 java.security 包中 AccessController 类的静态方法 doPrivileged(), 则 Mypriv 对象的 run() 方法将自动执行。最后通过在 Mypriv 对象中定义的 getValue() 方法获取特权代码执行的结果。

★代码与分析:

本实例中编程者“Liu Fang”编写的实现 java.security.PrivilegedAction 接口的类完整代码如下:

```
package myp;
import java.security.*;
import java.io.*;
class Mypriv implements PrivilegedAction {
    private String filename;
    private String value;
    //通过构造器传入要特权读取的文件名称
    public Mypriv(String fname) {
        filename = fname;
    }
    //以特权方式执行的代码
    public Object run() {
        String s;
        try{
            BufferedReader in = new BufferedReader(
                new FileReader(filename));
            while ((s = in.readLine()) != null) {
                value+=s+"\n";
            }
        }
    }
}
```

```

        catch(IOException e){
        }
        return value;
    }
    //返回特权代码执行结果
    public String getValue() {
        return value;
    }
}

```

编程者“Liu Fang”修改 8.4.1 小节 ShowFile2.java 后，新的代码 ShowFile3.java 完整代码如下：

```

package myp;
import java.security.*;
import java.io.*;
public class ShowFile3{
    public String go(String name) throws IOException{
        String s;
        String content="";
        BufferedReader in;
        //正常执行
        in = new BufferedReader(new FileReader(name));
        while ((s = in.readLine( )) != null) {
            content+=s+"\n";
        }
        //特权执行
        Mypriv mp=new Mypriv("c:\\\\autoexec.bat");
        AccessController.doPrivileged(mp);
        String sp = mp.getValue();
        content+="-----privileged-----"+sp;
        return content;
    }
}

```

另一个编程者 B 编写的调用 ShowFile3 类的代码如下：

```

import myp.*;
import java.io.*;
public class RunShowFile3{
    public static void main(String args[]) throws IOException{
        ShowFile3 t=new ShowFile3();
        String s=t.go(args[0]);
        System.out.println(s);
        System.out.println("Over");
    }
}

```

```
}
```

用户运行程序所使用的策略文件 `mypri_dir.policy` 完整内容如下:

```
keystore "clienttrustlf", "JKS";
```

```
grant signedBy "lf" {  
    permission java.io.FilePermission "c:\\autoexec.bat", "read";  
    permission java.io.FilePermission "c:\\j2sdk1.4.0\\*", "read";  
};  
  
grant codeBase "file:/c:/java/ch8/sign/demo3/User/*" {  
    permission java.io.FilePermission "c:\\j2sdk1.4.0\\*", "read";  
};
```

其中, 用户对编程者 B 提供的 `RunShowFile3.class`, 只授予对 “`c:\j2sdk1.4.0\`” 目录下文件读的权限, 而没有授予对 `c:\autoexec.bat` 文件读的权限。

★运行程序

(1) 编程者 “Liu Fang” 的工作

假定编程者 “Liu Fang” 在 `C:\java\ch8\sign\demo3\A` 目录工作。其中包含了该编程者所编写的程序 `ShowFile3.java`、`Mypriv.java` 和编程者 “Liu Fang” 的密钥库 `lfkeystore2`, 密钥库的条目 `lf` 中存放着编程者 “Liu Fang” 的私钥和数字证书。

输入

```
javac -d . Mypriv.java  
javac -d . ShowFile3.java
```

编译程序, 编译后的代码将根据文件 `ShowFile2.java` 和 `Mypriv.java` 中的包的名字放在当前目录的 `myp` 子目录中。

然后输入

```
jar cvf showfile3.jar myp\*.class
```

将编程者 “Liu Fang” 的代码打包成 `showfile3.jar` 文件。

再输入

```
jarsigner -keystore lfkeystore2 -keypass wshr.ut -storepass wshr.ut showfile3.jar lf
```

使用密钥库 `lfkeystore2` 中 `lf` 条目对应的私钥签名 `showfile3.jar` 文件。这里将密码 `wshr.ut` 直接通过 `jarsigner` 命令行选项给出了, 也可以和 8.3.1 小节一样以交互方式输入。

然后执行

```
keytool -export -keystore lfkeystore2 -alias lf -file lf.cer -storepass wshr.ut
```

将编程者的证书导出到文件 `lf.cer`, 也可直接将 8.4.1 小节的 `lf.cer` 拷贝过来。

最后将 `showfile3.jar` 和 `lf.cer` 文件提供给其他编程者 (如编程者 B) 使用, 如可以拷贝到 `C:\java\ch8\sign\demo3\B` 目录。

(2) 编程者 B 的工作

假定编程者 “Liu Fang” 在 `C:\java\ch8\sign\demo3\B` 目录工作。其中包含了编程者 “Liu Fang” 提供给他的软件包 `showfile3.jar`, 编程者 B 自己编写了 `RunShowFile3.java` 程序。

和 8.4.1 小节一样输入如下命令编译程序:


```
javac -classpath showfile3.jar;. RunShowFile3.java
```

最后编程者 B 将自己的软件 RunShowFile3.class 和类库 showfile3.jar 以及类库签名者的证书文件 If.cer 提供给用户。

如果需要，编程者 B 也可以和编程者“Liu Fang”一样对自己的代码打包并签名。

(3) 用户运行程序

假定编程者用户在 C:\java\ch8\sign\demo3\User 目录工作。其中包含了编程者 B 提供文件 RunShowFile3.class、showfile3.jar 和 If.cer。编程者 B 声称 RunShowFile3.class 是自己编写的，showfile3.jar 是编程者“Liu Fang”签发的，If.cer 是编程者“Liu Fang”的证书文件。

用户创建 C:\java\ch8\sign\demo3\package 目录，将当前目录的 showfile3.jar 移动到该目录。

用户检验出 showfile3.jar 确实是值得信任的，于是使用 mypri_dir.policy 策略文件，允许其读取 c:\autoexec.bat 文件，但不允许编程者 B 的代码 RunShowFile3.class 读取 c:\autoexec.bat 文件。

和 8.4.1 小节一样，用户将 If.cer 导入自己的密钥库，也可直接拷贝 8.4.1 小节的密钥库 clienttrustlf。

将如下内容输入 1.bat 文件，

```
java -Djava.security.manager -Djava.security.policy=mpri_dir.policy
-classpath ../package/showfile3.jar;. RunShowFile3 C:\j2sdk1.4.0\readme.txt
```

执行后将可以正确显示 C:\j2sdk1.4.0\readme.txt 和 c:\autoexec.bat 文件的内容。

对比 8.4.1 小节的程序可见，使用特权代码，读取 c:\autoexec.bat 文件的操作只需要“Liu Fang”的代码有权限就行了，而不需要其调用者 B 的代码也有相同权限。

8.4.3 使用匿名类定义特权代码

★ 实例说明

本实例和 8.4.2 小节类似，编程者“Liu Fang”编写了代码 ShowFile4.java 程序，其中部分代码没有指定特权，部分代码指定了特权。然后另一个编程者编写代码 RunShowFile4.java，其中调用了编程者“Liu Fang”编写的类 ShowFile4。

但本实例中编程者“Liu Fang”在编写特权代码时没有另外定义一个类实现 java.security.PrivilegedAction 接口，而是通过匿名类实现。

★ 编程思路

使用匿名类的结构是：

```
AccessController.doPrivileged(
    new PrivilegedAction() {
        public Object run() {
            // ...
            //要执行的特权代码
            return null;
        }
    }
)
```

```
);
```

匿名类是内类的一种，不能直接访问外类 `ShowFile4` 中的局部变量（如存放特权代码准备访问的文件名的 `pname` 变量），除非其定义为 `final` 类型。为了能够传入文件名参数，可在执行 `AccessController.doPrivileged()` 方法的前一句定义 `final` 变量，将需要传给匿名类中特权代码的变量先赋值给 `final` 变量，如：

```
String pname; // 特权代码要读取的文件名称
...
final String filename=pname;
AccessController.doPrivileged(
    new PrivilegedAction() {
        public Object run() {
            // ...
            //要执行的特权代码，使用 filename
            return null;
        }
    }
);
```

由于 `final` 类型的变量一旦赋值便不可改变内容，为了使特权代码执行的结果返回给外类 `ShowFile4`，可以定义 `final` 类型的数组。因为尽管 `final` 类型的变量内容不可改变，但对于数组类型，只是数组对象（地址）不变，数组中的内容可变。

```
String pname; // 特权代码要读取的文件名称
...
final String result[] = {null};
final String filename=pname;
AccessController.doPrivileged(
    new PrivilegedAction() {
        public Object run() {
            // ...
            //要执行的特权代码，使用 filename
            //执行结果放在字符串 value 中
            result[0] = value; //传出执行结果
            return null;
        }
    }
);
// 可使用 result[0]做各种事情
```

另外一种做法是通过 `AccessController.doPrivileged()` 方法的返回值获取特权代码的执行结果。如：

```
String pname; // 特权代码要读取的文件名称
...
final String filename=pname;
```

```

String result = (String) AccessController.doPrivileged(
    new PrivilegedAction() {
        public Object run() {
            // ...
            //要执行的特权代码，使用 filename
            //执行结果放在字符串 value 中
            return value; //传出执行结果
        }
    }
);
//可使用 result 做各种事情

```

其中匿名类的 `run()` 方法中通过 `return` 语句返回执行结果，该结果将继续通过 `doPrivileged()` 方法返回，其类型是 `Object` 类型，因此需要根据实际返回的类型作强制转换，如转换为字符串。

★代码与分析:

编程者“Liu Fang”不再需要 8.4.2 小节的 `MyPriv` 程序，按照编程思路直接修改 8.4.2 小节 `ShowFile3.java` 程序，新的代码 `ShowFile4.java` 完整代码如下：

```

package myp;
import java.security.*;
import java.io.*;
public class ShowFile4{
    public String go(String name) throws IOException{
        String s;
        String pname="c:\\autoexec.bat"; //特权代码准备访问的文件
        String content="";
        BufferedReader in;
        in = new BufferedReader(new FileReader(name));
        while ((s = in.readLine()) != null) {
            content+=s+"\n";
        }
        //特权代码
        final String filename=pname;
        String result = (String) AccessController.doPrivileged(
            new PrivilegedAction() {
                public Object run() {
                    String value="";
                    String s;
                    try{
                        BufferedReader in = new BufferedReader(
                            new FileReader(filename));
                        while ((s = in.readLine()) != null) {
                            value+=s+"\n";
                        }
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                    return value;
                }
            }
        );
        return result;
    }
}

```

```

        }
    }
    catch(IOException e){
    }
    return value; //传出执行结果
}
}
);
content+=result;
return content;
}
}

```

这里传出特权代码执行结果使用了变成思路中的后一种方法，即通过 `AccessController.doPrivileged()` 方法的返回值获取特权代码的执行结果。

另一个编程者 B 编写的调用 `ShowFile4` 类的代码如下：

```

import myp.*;
import java.io.*;
public class RunShowFile4{
    public static void main(String args[]) throws IOException{
        ShowFile4 t=new ShowFile4();
        String s=t.go(args[0]);
        System.out.println(s);
        System.out.println("Over");
    }
}

```

用户的策略文件和 8.4.2 小节的 `mypri_dir.policy` 文件类似，它对编程者 B 提供的 `RunShowFile4.class`，只授予对“`c:\j2sdk1.4.0\`”目录下文件读的权限，而没有授予对 `c:\autoexec.bat` 文件读的权限。不妨使用文件名 `mypri_dir2.policy`，其内容如下：

```

keystore "clienttrustlf", "JKS";

grant signedBy "lf" {
    permission java.io.FilePermission "c:\\autoexec.bat", "read";
    permission java.io.FilePermission "c:\\j2sdk1.4.0\\*", "read";
};

grant codeBase "file:/c:/java/ch8/sign/demo4/User/*" {
    permission java.io.FilePermission "c:\\j2sdk1.4.0\\*", "read";
};

```

★运行程序

(1) 编程者“Liu Fang”的工作

假定编程者“Liu Fang”在 C:\java\ch8\sign\demo4\A 目录工作。其中包含了该编程者所编写的程序 ShowFile4.java 和编程者“Liu Fang”的密钥库 lfkeystore2，密钥库的条目 lf 中存放着编程者“Liu Fang”的私钥和数字证书。

输入

```
javac -d . ShowFile4.java
```

编译程序，编译后的代码将放在当前目录的 myp 子目录中。由于使用了匿名类，因此编译后 myp 子目录将有两个字节码文件：ShowFile4.class 和 ShowFile4\$1.class。输入：

```
jar cvf showfile4.jar myp\*.class
```

将编程者“Liu Fang”的代码打包成 showfile4.jar 文件。

再输入

```
jarsigner -keystore lfkeystore2 -keypass wshr.ut -storepass wshr.ut showfile4.jar lf
```

使用进行签名。

然后执行

```
keytool -export -keystore lfkeystore2 -alias lf -file lf.cer -storepass wshr.ut
```

将编程者的证书导出到文件 lf.cer，也可直接将 8.4.1 小节的 lf.cer 拷贝过来。

最后将 showfile4.jar 和 lf.cer 文件提供给其他编程者（如编程者 B）使用，如可以拷贝到 C:\java\ch8\sign\demo4\B 目录。

（2）编程者 B 的工作

假定编程者“Liu Fang”在 C:\java\ch8\sign\demo4\B 目录工作。其中包含了编程者“Liu Fang”提供给他的软件包 showfile4.jar，编程者 B 自己编写了 RunShowFile4.java 程序。

和 8.4.1 小节一样输入如下命令编译程序：

```
javac -classpath showfile4.jar;. RunShowFile4.java
```

最后编程者 B 将自己的软件 RunShowFile4.class 和类库 showfile3.jar 以及类库签名者的证书文件 lf.cer 提供给用户。

（3）用户运行程序

假定编程者用户在 C:\java\ch8\sign\demo4\User 目录工作。其中包含了编程者 B 提供文件 RunShowFile3.class 和 lf.cer。用户创建 C:\java\ch8\sign\demo4\package 目录，存放编程者 B 提供的、由编程者“Liu Fang”编写的 showfile4.jar 类库。

和 8.4.1 小节一样，用户将 lf.cer 导入自己的密钥库，也可直接拷贝 8.4.1 小节的密钥库 clienttrustlf。使用策略文件 mypri_dir2.policy，在批处理文件中输入如下命令运行程序：

```
java -Djava.security.manager -Djava.security.policy=mypri_dir.policy  
-classpath ../package/showfile4.jar;. RunShowFile4 C:\j2sdk1.4.0\readme.txt
```

执行后将可以正确显示 C:\j2sdk1.4.0\readme.txt 和 c:\autoexec.bat 文件的内容。

8.5 权限的操作及定义自己的权限

本章前面各节的例子使用的都是已有的权限，如 java.io.FilePermission。当已有的权限满足不了要求时，可以定义自己的权限。本节介绍其编程方法，并介绍有关权限的各种操作。

8.5.1 策略文件权限的检测

★ 实例说明

本章前面各节的程序都是自动读取策略文件判断是否具有某种权限，本实例介绍当需要某种权限时，如何在程序中主动检测策略文件是否提供了某种权限。

★ 编程思路

检测策略文件中是否具有某种权限可以使用 `SecurityManager` 类的 `checkPermission()` 方法，也可以使用 `AccessController` 类的静态方法 `checkPermission()`。这两个方法都会从 Java 命令行选项 `-Djava.security.policy` 指定的策略文件或默认的策略文件中读取配置信息，计算是否具有某种权限。若有，则可正常向下运行；若没有，则抛出异常对象：`java.security.AccessControlException`。

两种方法其实是一样的，如果我们使用 WinZip 打开 `C:\j2sdk1.4.0\src.zip` 文件，查看 J2SDK 提供的 `SecurityManager` 类的源代码：`SecurityManager.java` 程序，会发现其中的 `checkPermission()` 方法是这样定义的：

```
public void checkPermission(Permission perm) {  
    java.security.AccessController.checkPermission(perm);  
}
```

因此，使用 `SecurityManager` 类的 `checkPermission()` 方法在本质上仍然使用的是 `AccessController` 类的静态方法 `checkPermission()`。所不同的是，使用 `SecurityManager` 类的 `checkPermission()` 方法时必须先获得 `SecurityManager` 对象，因此在运行程序时必须指定 Java 命令行选项 `-Djava.security.manager`。

具体编程步骤如下：

(1) 创建 `Permission` 对象

```
FilePermission fp=new FilePermission("c:\\autoexec.bat", "read");  
AWTPermission ap=new AWTPermission("accessClipboard");
```

分析：不同类型的权限使用不同名称，这些类的名称通常是 `XXXPermission` 形式，例如对于文件操作，权限类型为 `FilePermission`。对于图形界面操作，权限类型为 `AWTPermission`。

有些类型的权限如 `FilePermission` 的构造器使用两个参数，第一个参数代表目标，即该类型的权限是针对谁的，第二个参数代表动作，即允许对目标做什么操作。因此 `new FilePermission("c:\\autoexec.bat", "read")` 所创建的 `Permission` 对象代表对 `c:\\autoexec.bat` 文件的“读”的文件权限。在 `FilePermission` 类的 API 文档中提供了两个参数的具体含义、第二个参数可以取哪些值、分别代表什么含义等。

有些类型的权限如 `AWTPermission` 的构造器只使用一个参数，该参数代表目标，表明允许做什么操作。如 `AWTPermission` 类的 API 文档给出其构造器的参数可以取哪些值，分别代表什么含义。其中“`accessClipboard`”值代表允许读写 AWT 剪贴板。

(2) 获取安全管理器

```
SecurityManager sm = System.getSecurityManager();
```

分析：执行 `System` 类的静态方法 `getSecurityManager()`，如果在运行 Java 程序时使用 `-Djava.security.manager` 命令行选项指定了使用默认的安全管理器或自己定义的安全管理器，则将返回该安全管理器。否则将得到 `null` 值。

(3) 检测权限

```
if (sm != null) {  
    sm.checkPermission(fp);
```

```

        sm.checkPermission(ap);
    }

```

分析：如果程序在运行时指定了安全管理器（上一步骤返回值不是 null），则执行其 checkPermission() 方法检测策略文件中是否授予了第一步指定的权限。如果授予了，则程序可继续向下运行，否则抛出 java.security.AccessControlException 异常对象，可使用 try...catch...语句进行处理。

上面各步是使用 SecurityManager 类的 checkPermission() 方法进行检测，如果使用 AccessController 类的静态方法 checkPermission()，只要把第 2, 3 步替换为

```

AccessController.checkPermission(fp);
AccessController.checkPermission(ap);

```

★代码与分析：

本实例有两个程序，分别演示了两种不同的检测方法。

程序 SecMCheckPerm.java 使用 SecurityManager 类的 checkPermission() 方法进行检测，完整代码如下：

```

import java.io.*;
import java.io.*;
import java.security.*;
import java.awt.*;

public class SecMCheckPerm{
    public static void main(String args[]){
        try{
            SecurityManager sm = System.getSecurityManager();
            if (sm != null) {
                FilePermission fp=
                    new FilePermission("c:\\autoexec.bat", "read");
                sm.checkPermission(fp);
            }
            System.out.println("Has FilePermission to read c:\\autoexec.bat");
        }
        catch(AccessControlException e){
            System.out.println(e);
        }

        try{
            SecurityManager sm = System.getSecurityManager();
            if (sm != null) {
                AWTPermission ap=new AWTPermission("accessClipboard");
                sm.checkPermission(ap);
            }
            System.out.println("Has AWTPermission to access"+
                                "AWT Clipboard");
        }
    }
}

```

```

    }
    catch(AccessControlException e){
        System.out.println(e);
    }
}
}

```

程序 `AccessCCheckPerm.java` 使用 `AccessController` 类的静态方法 `checkPermission()` 进行检测，完整代码如下：

```

import java.io.*;
import java.security.*;
import java.awt.*;
public class AccessCCheckPerm{
    public static void main(String args[]){
        try{
            FilePermission fp=
                new FilePermission("c:\\autoexec.bat", "read");
            AccessController.checkPermission(fp);
            System.out.println("Has FilePermission to"+
                                "read c:\\autoexec.bat");
        }
        catch(AccessControlException e){
            System.out.println(e);
        }

        try{
            AWTPermission ap=new AWTPermission("accessClipboard");
            AccessController.checkPermission(ap);
            System.out.println("Has AWTPermission to access AWT Clipboard");
        }
        catch(AccessControlException e){
            System.out.println(e);
        }
    }
}

```

本实例使用两个策略文件分别演示不同的效果，其中 `CheckPer1.policy` 中分别授予 `c:\java\ch8` 下所有代码相应的权限，而 `CheckPer2.policy` 中授予的权限和需要检测的权限不同。

文件 `CheckPer1.policy` 的完整内容如下：

```

grant codeBase "file:/c:/java/ch8/-" {
    permission java.awt.AWTPermission "accessClipboard";
    permission java.io.FilePermission "c:\\autoexec.bat", "read,write";
};

```


文件 CheckPer2.policy 的完整内容如下:

```
grant codeBase "file:/c:/java/ch8/-" {  
    permission java.awt.AWTPermission "readDisplayPixels";  
    permission java.io.FilePermission "c:\\autoexec.bat", "delete,execute";  
};
```

★运行程序

输入

java -Djava.security.manager -Djava.security.policy=CheckPer1.policy SecMCheckPerm
或 java -Djava.security.manager -Djava.security.policy=CheckPer1.policy AccessCCheckPerm
运行程序, 将显示

```
Has FilePermission to read c:\autoexec.bat
```

```
Has AWTPermission to access AWT Clipboard
```

这是因为策略文件 CheckPer1.policy 授予了所检测的权限。

输入

java -Djava.security.manager -Djava.security.policy=CheckPer2.policy SecMCheckPerm
或 java -Djava.security.manager -Djava.security.policy=CheckPer2.policy AccessCCheckPerm
运行程序, 将显示

```
java.security.AccessControlException: access denied (java.io.FilePermiutoexec.bat  
read)
```

```
java.security.AccessControlException: access denied (java.awt.AWTPermissClipboard)
```

这是因为策略文件 CheckPer2.policy 没有授予所检测的权限。

如果运行时将 Java 命令行选项 -Djava.security.manager 去掉, 则输入

```
java -Djava.security.policy=CheckPer2.policy SecMCheckPerm
```

运行结果如下:

```
Has FilePermission to read c:\autoexec.bat
```

```
Has AWTPermission to access AWT Clipboard
```

这是因为没有如果 Java 命令行选项 -Djava.security.manager, 则 System.getSecurityManager() 语句返回的是 null, 因而程序将不再执行检测操作。

而如果输入

```
java -Djava.security.policy=CheckPer2.policy AccessCCheckPerm
```

则显示:

```
java.security.AccessControlException: access denied (java.io.FilePermiutoexec.bat  
read)
```

```
java.security.AccessControlException: access denied (java.awt.AWTPermissClipboard)
```

这是因为程序 AccessCCheckPerm.java 没有检测是否存在安全管理器, 总是执行 AccessController.checkPermission() 方法。

8.5.2 最简单的权限定义

★ 实例说明

8.5.1 小节的例子使用了一个 java.io.AWTPermission 类型的权限, 用户可以在策略文件中指定该类型的 accessClipboard、readDisplayPixels 等权限, 并可以在程序中检测是否具有

这些权限。

本实例使用最简单的方法定义了一个类型为 `my.test.RatePermission` 的权限, 用户同样可以策略文件中指定该类型的 `setR` 等权限, 并可以在程序中检测是否具有这些权限。

★ 编程思路

本实例的场景是, 编程者甲编写了一个类 `MyRate` 给编程者乙使用 (甲和乙可以是两个不同的人, 也可以是同一个人), 其中定义了一个私有的成员变量 `rate`。类 `MyRate` 中定义了方法 `getRate()` 用于获取成员变量 `rate` 的值, 定义了方法 `setRate()` 用于修改成员变量 `rate` 的值。编程者甲觉得成员变量 `rate` 的值一般不用修改, 如果编程者乙使用 `setRate()` 方法修改该值, 应该在用户的许可之下进行。

于是编程者甲定义了一个自己的权限, 名称为 `my.test.RatePermission`。在编写 `MyRate` 类的 `setRate()` 方法时, 按照 8.5.1 小节类似的方法先检查用户的策略文件中是否授予了 `my.test.RatePermission` 类型的权限。

编写自己的权限最简单的方法是扩展 `java.security` 包中的抽象类 `BasicPermission`, 最简单的例子只要定义构造器即可。其步骤如下:

(1) 扩展 `BasicPermission`

```
public final class RatePermission extends BasicPermission
```

分析: 这里给自己的权限起的名称为 `RatePermission`, 此外将其定义为 `final` 类型, 这样就不可以再定义 `RatePermission` 类的子类。

(2) 定义一个参数的构造器

```
public RatePermission(String name){
    super(name);
}
```

分析: 该构造器带一个字符串参数, 该字符串表示权限的名称。最简单的编程只需要在构造器中执行 `super()` 方法将参数传递给 `BasicPermission` 即可。

以后只要策略文件中加入 `permission my.test.RatePermission "xxx"` (其中 `my.test` 是包的名字), 则程序中使用 `AccessController.checkPermission(new RatePermission("xxx"))` 就不会产生 `AccessControlException` 类型的异常。

(3) 定义两个参数的构造器

```
public RatePermission(String name, String actions){
    super(name, actions);
}
```

分析: 该构造器带两个字符串参数, 该字符串表示权限的名称。第一个参数代表目标, 第二个参数代表动作。

在 `BasicPermission` 中第二个参数其实不使用, 第二个参数传入什么值没有影响。因此不管策略文件中设置的是 `permission my.test.RatePermission "xxx", "aaa"`, 还是 `permission my.test.RatePermission "xxx", "bbb"`, 执行 `AccessController.checkPermission(new RatePermission("xxx", "aaa"))` 和 `AccessController.checkPermission(new RatePermission("xxx", "bbb"))` 的效果都是一样的。

定义好自己的权限后, 编程者甲在编写 `MyRate` 类的 `setRate()` 方法时就可以先检测一下该权限是否已经配置在策略文件中了, 不妨使用如下语句:

```

SecurityManager sm = System.getSecurityManager();
if (sm != null) {
    sm.checkPermission(new RatePermission("setR"));
}

```

这里，创建 `RatePermission` 对象时传入的“setR”字符串是编程者甲为权限起的一个名称，编程者甲在将类交给编程者乙或用户时，应说明如果要修改 `rate` 变量的值，应该在策略文件中授予 `my.test.RatePermission` 类型的 `setR` 权限，并告知用户随便让别人修改 `rate` 变量的值可能带来的安全问题。

这样，其他编程者在使用编程者甲提供的软件包时，如果需要执行 `MyRate` 类的 `setRate()` 方法修改 `rate` 变量的值，则必须经过程序执行者的授权，如果程序执行者对其不信任，则可以拒绝其修改 `rate` 变量。

★代码与分析:

本实例中，编程者甲编写了两个程序，一个是自己的权限，定义在 `RatePermission.java` 程序中，另一个是使用该权限的类，定义在 `MyRate.java` 程序中。

程序 `RatePermission.java` 完整代码如下：

```

package my.test;
import java.security.*;
public final class RatePermission extends BasicPermission {
    public RatePermission(String name){
        super(name);
    }
    public RatePermission(String name, String actions){
        super(name, actions);
    }
}

```

该类定义在 `my.test` 包中，这样在程序和策略文件中可以使用 `my.test.RatePermission` 访问该权限类型。

程序 `RatePermission.java` 完整代码如下：

```

package my.test;
import java.io.*;
public class MyRate{
    private double rate=0.8;
    public void setRate(double r){
        SecurityManager sm = System.getSecurityManager();
        if (sm != null) {
            sm.checkPermission(new RatePermission("setR"));
        }
        rate=r;
    }

    public double getRate( ){
        return rate;
    }
}

```

```

    }
}

```

该类也定义在 `my.test` 包中，程序员甲可以将这两个类打包提供给其他程序员使用，并告知运行 `setRate()` 设置 `rate` 变量时需要在策略文件中授予 `my.test.RatePermission` 类型的 `setR` 权限。

程序员乙收到该软件包后，编写了程序 `TestRate.java`，这个程序可能是一个大型的程序，使用到了程序员甲提供的软件包，不妨提取其中关键几句如下：

```

import my.test.*;
import java.io.*;
import java.security.*;
public class TestRate{
    public static void main(String args[]){
        my.test.MyRate mr=new my.test.MyRate( );
        System.out.println(mr.getRate());
        try{
            mr.setRate(0.6);
        }
        catch(AccessControlException e){
            System.out.println(e);
        }
        System.out.println(mr.getRate());
    }
}

```

该段代码中，先创建 `MyRate` 类型的对象，然后执行其 `getRate()` 方法获取 `rate` 变量的值并打印出来。然后执行其 `setRate()` 方法修改 `rate` 变量的值，最后再执行其 `getRate()` 方法获取 `rate` 变量的值并打印出来。

用户使用的策略文件 `ownPer1.policy` 内容如下

```

grant codeBase "file:/c:/java/ch8/-" {
    permission my.test.RatePermission "setR";
};

```

它授予 `c:\java\ch8` 目录及子目录下所有代码具有 `RatePermission` 类型的 `setR` 权限。

另一个策略文件 `ownPer1.policy` 内容如下

```

grant codeBase "file:/c:/java/ch8/-" {
    permission java.io.FilePermission "c:\\autoexec.bat", "read,write";
};

```

它授予的权限和程序所要求的权限不一样。

★运行程序

本实例工作目录在 `C:\java\ch8\mypermission\OwnPerm` 目录。输入

```

javac -d . RatePermission.java
javac -d . MyRate.java

```

编译甲的程序，将在当前目录生成 my\test 子目录，存放编译后的字节码文件。

输入

```
javac TestRate.java
```

编译乙的程序，将在当前目录生成 TestRate.class 文件。

用户输入

```
java TestRate
```

运行程序，得到输出结果

```
0.8
```

```
0.6
```

可见变量 rate 的值由 0.8 改到了 0.6。这是因为用户没有指定使用安全管理器，因而应用程序具有所有权限。用户如果使用安全管理器，同时对程序 TestRate 不太信任，则可输入

```
java -Djava.security.manager -Djava.security.policy=ownPer2.policy TestRate
```

运行程序，则程序输出：

```
0.8
```

```
java.security.AccessControlException: access denied (my.test.RatePermission setR)
```

```
0.8
```

其中显示可见 my.test.RatePermission setR 权限，并可见变量 rate 的值没有被修改。而如果用户信任程序 TestRate，则可输入

```
java -Djava.security.manager -Djava.security.policy=ownPer1.policy TestRate
```

运行程序，则程序输出：

```
0.8
```

```
0.6
```

8.5.3 使用签名的权限

★ 实例说明

8.5.3 小节的一个问题是编程者乙可能修改变程者甲提供的软件包及其中定义的 my.test.RatePermission 权限，本实例使用签名解决了这一问题。

★ 编程思路

本实例使用类似 8.3.1 小节的方法对自己定义的权限进行签名，为了在策略文件中指定权限必须经过签名，应该在权限后面加上一句 signedBy "xxx"，其中 xxx 是策略文件所使用的密钥库的条目的名称。如

```
keystore "file:/c:/java/ch8/mypermission/SignedPerm/clienttrustlf", "JKS";  
grant codeBase "file:/c:/java/ch8/-" {  
    permission my.test.RatePermission "setR", signedBy "lf";  
}
```

如果使用 policytool 工具，可以在添加权限时在“签署人”栏目后面输入代表签发者的条目名称。如图 图 8- 20 所示。

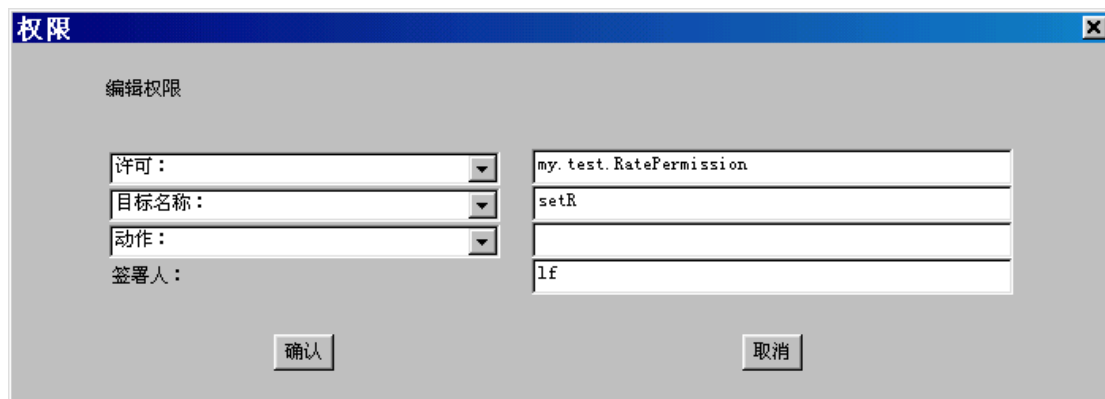


图 8-20 使用签名的权限

★代码与分析:

本实例使用 8.5.2 小节的程序，不同的是策略文件，不妨使用两个不同的策略文件加以对比，一个是 SignPer1.policy，它不要求权限必须签名。另一个是 SignPer2.policy，它要求 my.test.RatePermission "setR" 权限必须是 file:/c:/java/ch8/mypermission/SignedPerm/clienttrustlf 中 lf 条目对应的证书所对应的私钥签名的，即必须是 lfkeystore2 密钥库中 lf 条目对应的私钥签名的。

策略文件 SignPer1.policy 的完整内容如下：

```
grant codeBase "file:/c:/java/ch8/-" {
    permission my.test.RatePermission "setR";
};
```

策略文件 SignPer2.policy 的完整内容如下：

```
keystore "file:/c:/java/ch8/mypermission/SignedPerm/clienttrustlf", "JKS";
grant codeBase "file:/c:/java/ch8/-" {
    permission my.test.RatePermission "setR", signedBy "lf";
};
```

★运行程序

本实例工作目录在 C:\java\ch8\mypermission\SignedPerm 目录。其中拷贝了 8.5.2 小节 C:\java\ch8\mypermission\OwnPerm 目录中所有字节码文件以及 8.3.1 小节 C:\java\ch8\sign\demo1\A 目录中用于签名的密钥库 lfkeystore2 文件和 8.3.2 小节 C:\java\ch8\sign\demo1\B 目录用户信任的密钥库 clienttrustlf。

和 8.3.1 小节类似，首先编程者甲将自己的软件包打包为 MyRate.jar 文件：

```
C:\java\ch8\mypermission\SignedPerm>jar cvf MyRate.jar my\test\*.class
标明清单(manifest)
增加: my/test/MyRate.class(读入= 652) (写出= 409) (压缩了 37%)
增加: my/test/RatePermission.class(读入= 347) (写出= 231) (压缩了 33%)
```

然后编程者甲使用自己的密钥库 lfkeystore2 中 lf 条目进行签名：

```
C:\java\ch8\mypermission\SignedPerm>jarsigner -keystore lfkeystore2 MyRate.jar lf
Enter Passphrase for keystore: wshr.ut
```

此时用户如果输入

```
java -Djava.security.manager -Djava.security.policy=SignPer1.policy TestRate
```

运行程序，这时使用的是当前目录下 my\test 目录中的 RatePermission 类，它没有经过编程者甲签名，但由于策略文件 SignPer1.policy 中不管权限是否经过签名，因而程序正常输出 0.8 和 0.6。

但如果用户如果输入

```
java -Djava.security.manager -Djava.security.policy=SignPer2.policy TestRate
```

运行程序，这时使用的是当前目录下 my\test 目录中的 RatePermission 类，它没有经过编程者甲签名，而策略文件 SignPer2.policy 中要求 my.test.RatePermission "setR" 权限必须是 file:/c:/java/ch8/mypermission/SignedPerm/clienttrustlf 中 If 条目对应的证书所对应的私钥签名的，因而输出：

0.8

java.security.AccessControlException: access denied (my.test.RatePermission setR)

0.8

如果用户输入

```
java -classpath MyRate.jar;. -Djava.security.manager  
-Djava.security.policy=signPer2.policy TestRate
```

它增加了 -classpath 命令行选项，指定使用 MyRate.jar 中的 RatePermission 类。由于 MyRate.jar 已经经过签名，因而程序正常输出 0.8 和 0.6。

8.6 Applet 的安全运行

本章前面各节都是以 Java Application 为例介绍程序的安全运行，缺省情况下 Java Application 可以访问用户的任何资源，但通过指定 Java 命令行选项可以使 Java 程序运行在沙盒（SandBox）中，从而保护用户的安全。

对于 Java Applet，由于它可以在用户浏览网页时自动从 Internet 上下载到用户的机器上运行，因此缺省情况下 Java Applet 自动在沙盒中运行，沙盒中的程序有许多限制，如不能访问用户的文件系统、不能和程序原先所在机器以外的服务器通信等。这样缺省情况下即提供了足够的安全性。

Java Applet 既可以使用 AppletViewer 来运行，也可以在浏览器中运行。很多情况下可能希望 Java Applet 能在用户的许可下访问用户本机系统，本章给出在用户许可的前提下、针对不同运行方式的突破沙盒限制的方法。

8.6.1 使用 AppletViewer 运行的 Java Applet

★ 实例说明

本实例给出一个访问用户文件系统的 Java Applet 程序，演示了缺省情况下的安全性，并给出使用 AppletViewer 运行 Java Applet 时通过策略文件允许其访问用户文件的方法。

★ 编程思路

编写 Java Applet 时只要定义 java.applet 包中 Applet 类的子类即可，在网页中通过 <applet> 标记调用 Java Applet 程序，则浏览器打开网页时会自动创建对象，并根据一定的规则执行 Java Applet 的 init(), start(), stop(), paint() 等方法。编写 Java Applet 程序时可根据需要重写这些方法。

其中, `init()` 方法在浏览器刚创建 Applet 对象并加载进浏览器时执行, 以后不再执行, 一般放一些初始化操作, 本实例将在该方法中读取 `c:\autoexec.bat` 文件的内容, 并在 Applet 窗口中添加一个文本区域 (TextArea), 用于显示读取的 `c:\autoexec.bat` 文件的内容。

用户运行程序时窗口由不可见到可见或窗口被其他窗口遮盖住一部分然后其他窗口又移开时 (Exposure), 会自动执行 `paint()` 方法。本实例中在 `paint()` 方法中执行 TextArea 对象的 `setText()` 方法, 将 `init()` 方法中所读取的 `c:\autoexec.bat` 文件内容传递给 TextArea 对象显示。

最后, 编写网页文件 `1.html` 调用该 Java Applet 程序。网页文件使用 HTML 语法, 可以使用各种文本编辑工具编写, 其中根据 HTML 的规则加入各种标记。如整个文件放在 `<HTML>` 和 `</HTML>` 两个标记之间。在 `<HEAD>` 和 `</HEAD>` 之间是文档的头, 其中最常用的是放置 `<TITLE>` 和 `</TITLE>` 标记指定网页的标题, 它将在浏览器标题栏中显示出来。和 `<HEAD>` 和 `</HEAD>` 并列的是 `<BODY>` 和 `</BODY>` 标记, 这两个标记之间是网页需要显示的内容。这里, 显示了一串字符 “This is my test Applet to show file in local system”, 然后调用 Java Applet 程序。网页中调用 Applet 的格式是:

```
<applet archive=预加载的工具包 codebase=Applet 字节码文件所在位置
        code=Applet 字节码文件名 name=Applet 实例名 align=对齐模式
        vspace=上下间隔 hspace=左右间隔 width=宽度 height=高度 alt=字符串>
<param name=参数名 1    value=参数值 1>
<param name=参数名 2    value=参数值 2>
...
</applet>
```

其中,

- archive 中给出 Applet 用到的其他软件包, 如果有多个软件包则用逗号隔开。
- codebase 给出 Applet 字节码文件所在位置, 可以是绝对路径, 如 `http://xxx.xxx.xxx/xx/`, 也可以是相对 HTML 文档所在的路径, 如 `code/class`。
- code 给出字节码文件的名称。
- Name 为 Applet 的对象定义一个名称, 当一个网页中有多个 Applet 程序时, 可通过它相互进行交互。
- Align 指定 Applet 窗口和周围其他网页内容之间的对齐方式, 可以是 `left`, `right`, `top`, `texttop`, `middle`, `absmiddle`, `baseline`, `bottom` 或 `absbottom` 等值。
- Vspace 和 hspace 指定 Applet 窗口和周围其他网页内容之间上下和左右之间的间隔。

本实例中网页文件 `1.html` 简单地指定了 Applet 字节码文件的名称, 以及 Applet 在浏览器中运行时浏览器为其开设的窗口的宽度和高度。

最后, 为了使 Java Applet 可以突破沙盒的限制, 编写了类似 8.2.4 小节的策略文件, 允许 `c:\java\ch8` 目录下及其所有子目录下的程序读取 `c:\autoexec.bat` 文件。本实例的策略文件是基于代码的位置进行授权的, 也可以和 8.3 节一样对代码进行签名, 然后针对签名者进行授权。

使用 AppletViewer 运行 Java Applet 时只要通过命令行选项指定该策略文件, 就可以获得该策略文件中指定的权限。AppletViewer 的 `-J` 命令行选项后面可以带上 `-Djava.security.policy` 参数指定策略文件的名称。

★代码与分析:

本实例 Java Applet 的完整代码如下:


```

import java.io.*;
import java.awt.*;
import java.applet.*;
public class AppletShowFile extends Applet{
    String content="The content of file:";
    TextArea ta=new TextArea(10,80);
    public void init ( ) {
        String s;
        BufferedReader in;
        try{
            in = new BufferedReader(new FileReader("c:\\autoexec.bat"));
            while ((s = in.readLine( )) != null) {
                content+=s+"\n";
            }
        }
        catch(Exception e){
            System.err.println(e);
        }
        add(ta);
    }
    public void paint(Graphics g){
        ta.setText(content);
    }
}

```

本实例调用 Java Applet 程序的网页文件 1.html 内容如下:

```

<html>
<head>
<title> My test Applet </title>
</head>
<body>
This is my test Applet to show file in local system
<applet code=AppletShowFile.class Width=700 height=200>
</applet>
</body>
</html>

```

本实例所使用的策略文件 appletdir.policy 如下:

```

grant codeBase "file:/c:/java/ch8/-" {
    permission java.io.FilePermission "c:\\autoexec.bat", "read";
};

```

★运行程序

首先使用 AppletViewer 按照传统的方式运行 Java Applet 程序，在命令行中进入 C:\java\ch8\Applet 目录，输入如下命令

AppletViewer 1.html

尽管 AppletViewer 仍旧打开了一个图形窗口，窗口中有一个文本区域，但其中并没有 c:\autoexec.bat 文件的内容。DOS 窗口提示：

```
java.security.AccessControlException:    access    denied    (java.io.FilePermission
c:\autoexec.bat read)
```

这是因为 Applet 默认情况下即在沙盒中运行，没有权限访问 c:\autoexec.bat 文件。

如果用户确定允许程序访问 c:\autoexec.bat 文件，可以在运行时指定使用策略文件 appletdir.policy，输入如下命令即可：

AppletViewer -J-Djava.security.policy=appletdir.policy 1.html

注意“-J”和“-Djava.security.policy=appletdir.policy”之间不能有空格。程序运行后出现图 8-21 所示的窗口。从中可以看到 Java Applet 程序成功地读取了 c:\autoexec.bat 文件，并将其内容显示了出来。

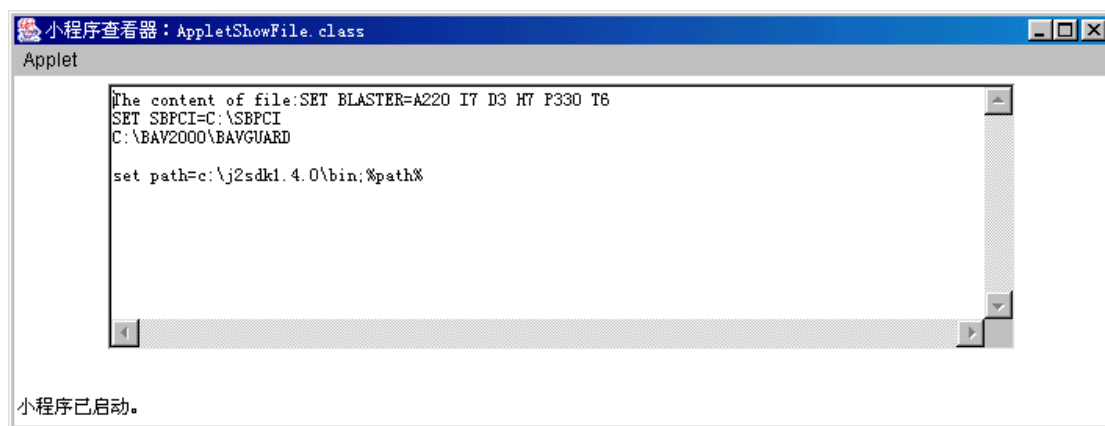


图 8-21 使用 AppletViewer 运行的 Java Applet 读取本地文件

AppletViewer 在运行程序只处理网页中 <applet> 标记的内容，而忽略其他网页内容，因而图 8-21 中没有显示网页中的字符串 “This is my test Applet to show file in local system”。

8.6.2 浏览器中使用 Java Plug-in 运行 Java Applet

★ 实例说明

Java Applet 更多地是通过浏览器来运行，本章后面各节将介绍通过浏览器运行 Java Applet 时如何突破沙盒的限制。由于本章后面各节是基于 Java 2 平台的，而老版本的浏览器本可能不支持 Java 2，本实例给出如何使 Java Applet 在 Java 2 中运行。

★ 编程思路

Java 2 中使用 Java Plug-in 增强浏览器的功能，各种浏览器缺省的 Java 运行环境可能不同，但使用 Java Plug-in 后，各种不同版本的浏览器都会自动在标准的 Java 2 环境中运行 Java Applet。

有两种方法可以让浏览器使用 Java Plug-in 运行 Java Applet。一种是在安装过 Java 2（如安装过 J2SDK1.4）的计算机上，可以直接在浏览器中设置使用 Java 2 运行 Java Applet。另一种是更加通用的做法，是修改调用 Applet 的 HTML 网页，将其中的<applet>标记按照一定格式转换为<Object>标记。

在 J2SDK 的安装目录 bin 子目录中提供了 HTMLConverter 工具可以自动完成网页的转换。转换后的网页用浏览器打开时，支持 Java 2 的浏览器不管是否设置了使用 Java 2 运行 Java Applet，都会在 Java 2 环境中运行 Java Applet。如果浏览器不支持 Java 2，则会自动下载并安装所需的文件，并且这种下载和安装只需要做一次。这样，大大方便了 Java Applet 的部署。

★代码与分析：

本实例 Java Applet 仍旧使用的 8.6.1 小节的 AppletShowFile.java 程序。

调用该 Java Applet 的 HTML 网页文件仍旧使用 8.6.1 小节的 1.html 文件，在本小节“运行程序”部分，使用 J2SDK 提供的 HTMLConverter 工具对其进行了转换，转换后的内容如下：

```
<html>
<head>
<title> My test Applet </title>
</head>
<body>
This is my test Applet to show file in local system
<!--"CONVERTED_APPLET"-->
<!-- HTML CONVERTER -->
<OBJECT
    classid="clsid:CAFEFAC-0014-0000-0000-ABCDEFEDCBA"
    WIDTH = 700 HEIGHT = 200

codebase="http://java.sun.com/products/plugin/autodl/jinstall-1_4_0-win.cab#
Version=1,4,0,0">
    <PARAM NAME = CODE VALUE = AppletShowFile.class >

    <PARAM NAME="type" VALUE="application/x-java-applet;jpi-version=1.4">
    <PARAM NAME="scriptable" VALUE="false">

    <COMMENT>
    <EMBED
        type="application/x-java-applet;jpi-version=1.4"
        CODE = AppletShowFile.class
        WIDTH = 700
        HEIGHT = 200
        scriptable=false

    pluginspage="http://java.sun.com/products/plugin/index.html#download">
    <NOEMBED>
```

```

        </NOEMBED>
    </EMBED>
</COMMENT>
</OBJECT>

<!--
<APPLET CODE = AppletShowFile.class WIDTH = 700 HEIGHT = 200>

</APPLET>
-->

<!--"END_CONVERTED_APPLET"-->

</body>
</html>

```

其中原有的<Applet>标记被注释掉了，改称了<Object>标记。当浏览器不支持 Java 2 时，会根据 <Object> 标记中的设置自动从 http://java.sun.com/products/plugin/autodl/jinstall-1_4_0-win.cab#Version=1,4,0,0 下载并安装。

★运行程序

首先演示使用第一种方式进行设置。启动IE浏览器后，选择“工具/Internet选项”菜单，进入Internet选项设置窗口。单击窗口中的“高级”标签，进入图 8-22所示的窗口，选中其中的“使用Java 2 v1.4.0 用于<applet>”选项（打勾号为已选中），单击“确定”按钮后关闭浏览器。

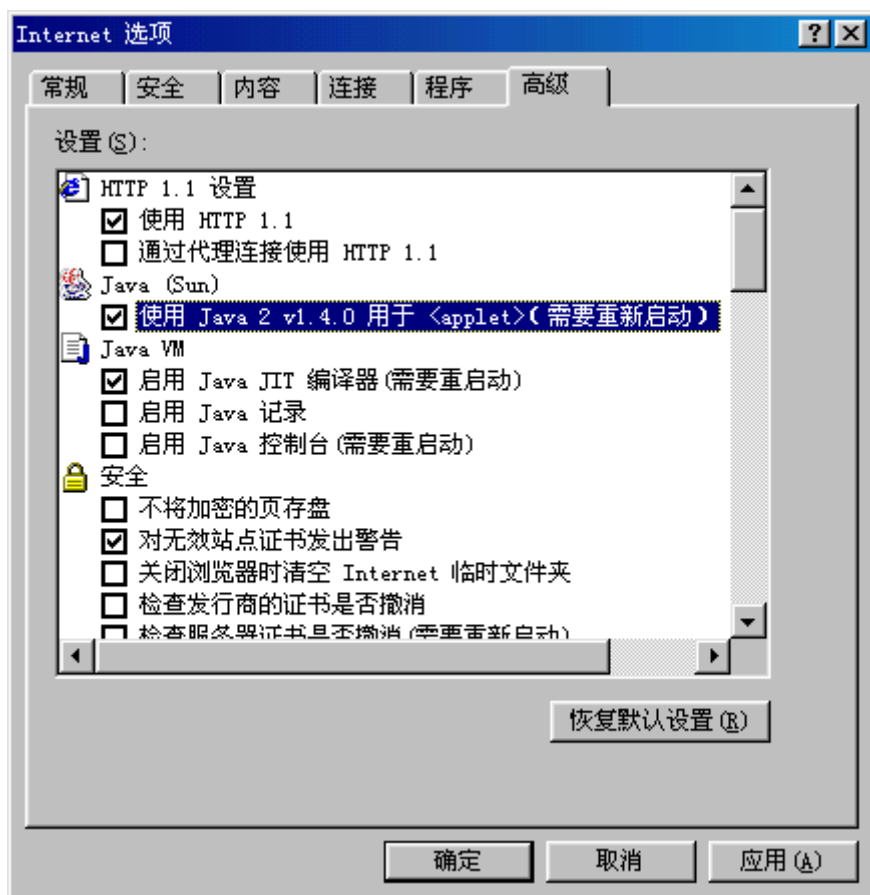


图 8-22 设置支持 Java Plug-in

以后浏览器将对所有Java Applet使用Java Plug-in来运行。如打开 1.html文件，将出现 图 8-23所示的界面。

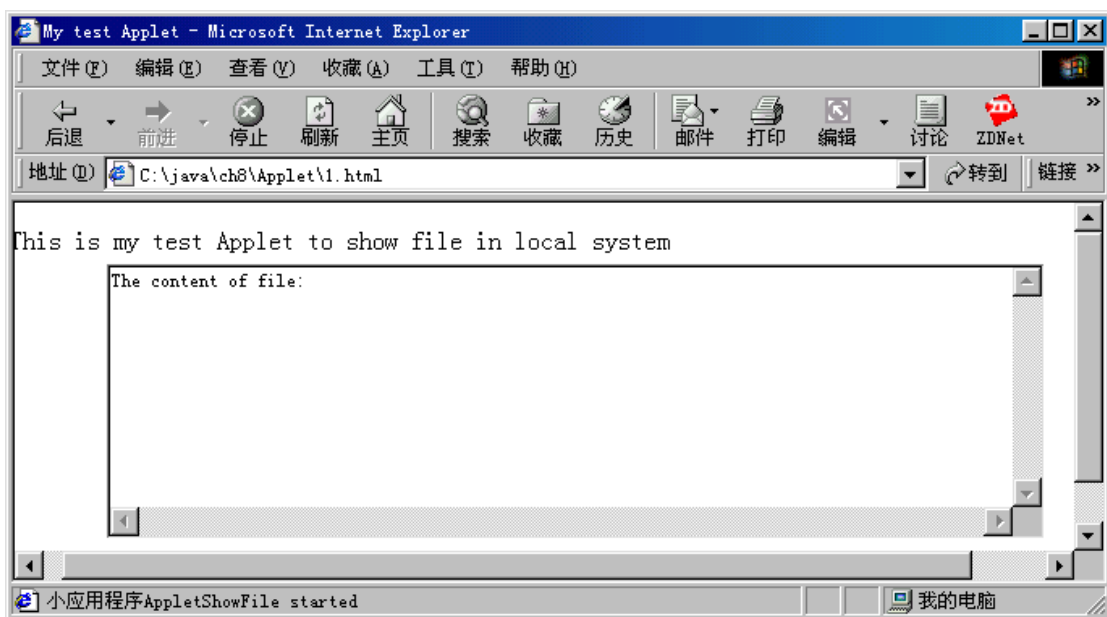


图 8-23 使用 Java Plug-in 运行的 Java Applet

图 8-23中，并没有显示c:\autoexec.bat文件中的内容，这是因为尚未进行授权。在后面各小节将介绍如何授权。

使用Java Plug-in运行Java Applet时，Windows下方工具栏的最右边将出现Java控制台图标，双击该图标将进入图 8-24所示的Java控制台窗口。

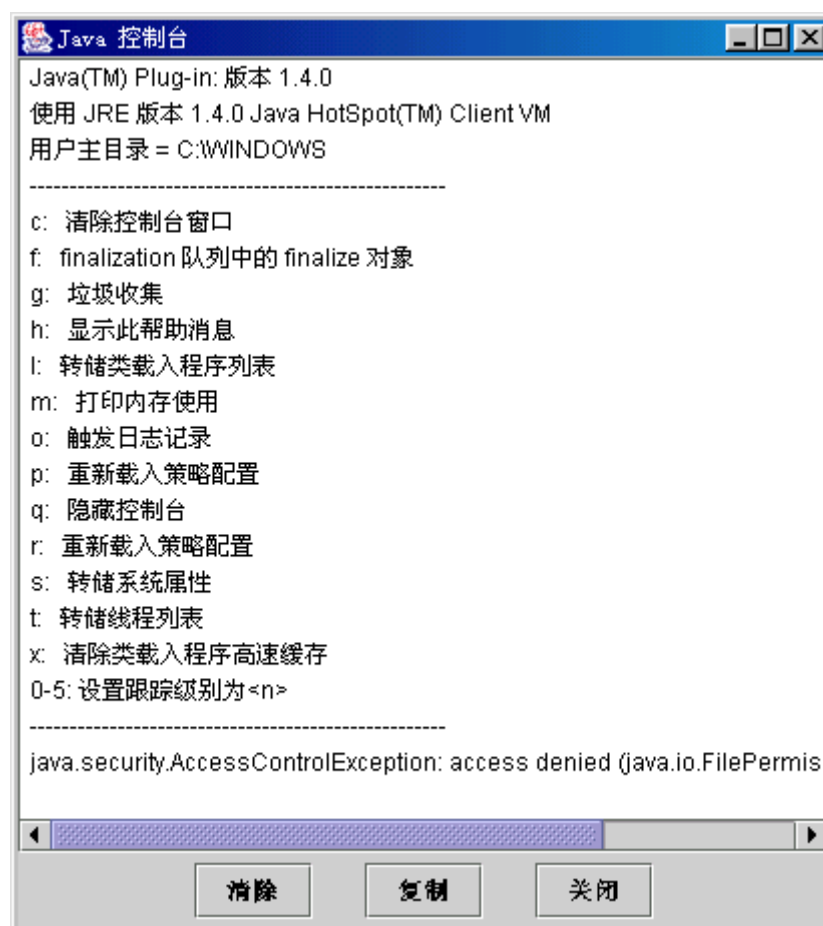


图 8-24 Java 控制台窗口

在该窗口中，显示了 Java Plug-in 的版本、用户主目录等信息。通过键盘可以进行清空窗口等各种操作。Java Applet 运行时的各种信息也记录在该窗口，如这里显示了对 c:\autoexec.bat 文件没有读的权限：“java.security.AccessControlException: access denied (java.io.FilePermission c:\autoexec.bat read)”。

下面演示使用第二种方式设置浏览器以 Java Plug-in 运行 Java Applet。在 C:\java\ch8\Applet 目录输入“HTMLConverter 1.html”，则 1.html 文件自动被转换成为本小节“代码与分析”部分的新的网页内容。原来的 1.html 文件自动备份在 C:\java\ch8\Applet_BAK 目录。

如果运行HTMLConverter时没有指定要处理的文件名，则将出现一个图形窗口，如图 8-25所示。

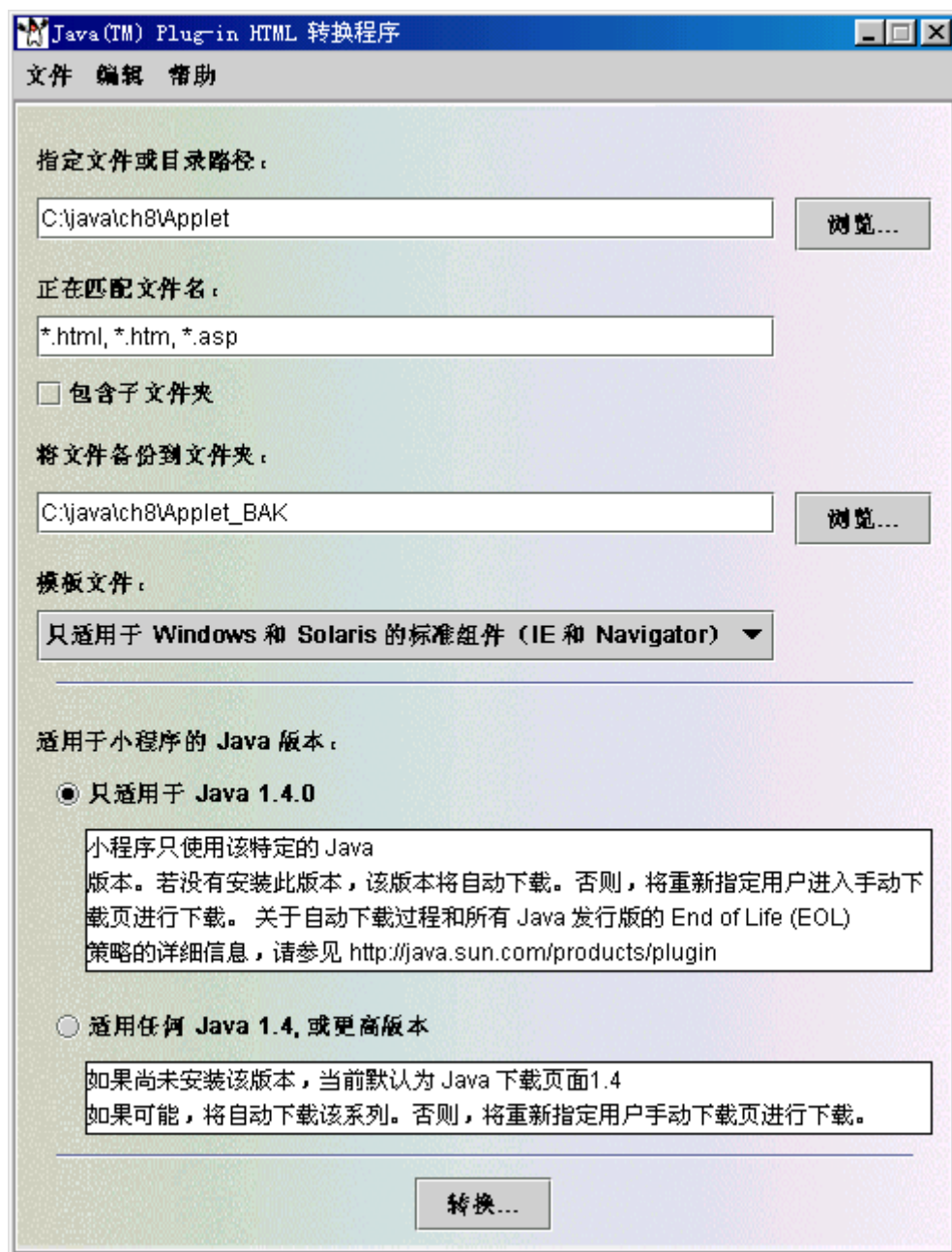


图 8- 25 HTMLConverter 窗口

该窗口中，可以将指定目录中所有指定后缀的文件（可以包括子目录）一起进行转换，原有文件可以备份到指定文件夹。在“模板文件”选项中，可以指定转换后的网页文件针对何种浏览器。最后可以单击“转换”按钮进行转换，将出现一个窗口显示转换的进度，并显示处理过的文件总数、找到的小应用程序总数和错误总数。

转换完毕后，不妨将转换后的 1.html 备份为 2.html，并将备份目录中原有的 1.html 拷贝回 C:\java\ch8\Applet 目录。

此时，打开 2.html，将和第一种方法一样出现图 8- 23所示的窗口，并可和图 8- 24一样显示Java控制台。

文件 2.html通用性很广，不管浏览器是否已经按照第一种图 8- 22的方法设置过“使用Java 2 v1.4.0 用于<applet>”都可以使用。即使用户的计算机上没有安装过J2SDK，也会在

用户同意的前提下自动下载并安装浏览器中Java 2 的运行环境。

8.6.3 浏览器基于策略文件运行 Java Applet

★ 实例说明

8.6.2 小节的 Java Applet 运行于 Java 2 环境，因而可以使用策略文件进行授权，在用户允许的前提下允许其访问用户本地资源。本实例给出浏览器中设置策略文件的方法。

★ 编程思路

8.6.1 小节使用 AppletViewer 命令行选项指定了 Applet 使用哪个策略文件，而使用浏览器时就无法指定命令行选项了，这时可将策略文件设置在默认的 Java 系统安全策略文件或默认的用户安全策略文件中。

对于浏览器，Windows 系统中默认的 Java 系统安全策略文件是 C:\Program Files\Java\j2re1.4.0\lib\security\java.policy 文件，如果浏览器中安装的是其他版本的 Java Plug-in，目录名称中 j2re1.4.0 会不同。该安全策略文件适用于所有用户，打开该文件，将 8.6.1 小节所使用的策略文件 appletdir.policy 中的内容加进去即可允许 c:\java\ch8 目录及其所有子目录中的 Java Applet 读取 c:\autoexec.bat 文件。

在 J2SDK 安装目录下也有内容一样的文件，如 C:\j2sdk1.4.0\jre\lib\security\java.policy，它主要用于 J2SDK 自带的工具如 java 或 AppletViewer 运行 Java 程序。而对于浏览器，使用的是 C:\Program Files\Java\j2re1.4.0\lib\security\java.policy 文件。

修改了 C:\Program Files\Java\j2re1.4.0\lib\security\java.policy 文件后，再使用 8.6.2 小节中任一种方法都可以和 8.6.1 小节一样正常显示 c:\autoexec.bat 文件的内容。

C:\Program Files\Java\j2re1.4.0\lib\security\java.policy 文件是针对所有用户的，如果只针对单个用户，也可由用户在自己主目录中的默认用户安全策略文件中进行授权。

对于 Windows 系统，默认的用户安全策略文件是 c:\windows\java.policy 文件（文件名前面有个点号），打开该文件（如果 c:\windows 尚没有 java.policy 文件，可用文本编辑器创建一个空文件，保存为 java.policy 文件名），将 8.6.1 小节所使用的策略文件 appletdir.policy 中的内容加进去即可。

★代码与分析：

本实例修改后的 C:\Program Files\Java\j2re1.4.0\lib\security\java.policy 文件如下所示：

```
// 8.6.3 added temporarily
grant codeBase "file:/c:/java/ch8/-" {
    permission java.io.FilePermission "c:\\autoexec.bat", "read";
};

// Standard extensions get all permissions by default

grant codeBase "file:${java.home}/lib/ext/*" {
    permission java.security.AllPermission;
};
```



```

// default permissions granted to all domains

grant {
    // Allows any thread to stop itself using the java.lang.Thread.stop()
    // method that takes no argument.
    // Note that this permission is granted by default only to remain
    // backwards compatible.
    // It is strongly recommended that you either remove this permission
    // from this policy file or further restrict it to code sources
    // that you specify, because Thread.stop() is potentially unsafe.
    // See "http://java.sun.com/notes" for more information.
    permission java.lang.RuntimePermission "stopThread";

    // allows anyone to listen on un-privileged ports
    permission java.net.SocketPermission "localhost:1024-", "listen";

    // "standard" properties that can be read by anyone

    permission java.util.PropertyPermission "java.version", "read";
    permission java.util.PropertyPermission "java.vendor", "read";
    permission java.util.PropertyPermission "java.vendor.url", "read";
    permission java.util.PropertyPermission "java.class.version", "read";
    permission java.util.PropertyPermission "os.name", "read";
    permission java.util.PropertyPermission "os.version", "read";
    permission java.util.PropertyPermission "os.arch", "read";
    permission java.util.PropertyPermission "file.separator", "read";
    permission java.util.PropertyPermission "path.separator", "read";
    permission java.util.PropertyPermission "line.separator", "read";

    permission java.util.PropertyPermission "java.specification.version",
"read";
    permission java.util.PropertyPermission "java.specification.vendor",
"read";
    permission java.util.PropertyPermission "java.specification.name",
"read";

    permission java.util.PropertyPermission
"java.vm.specification.version", "read";
    permission java.util.PropertyPermission "java.vm.specification.vendor",
"read";
    permission java.util.PropertyPermission "java.vm.specification.name",
"read";
    permission java.util.PropertyPermission "java.vm.version", "read";

```

```

    permission java.util.PropertyPermission "java.vm.vendor", "read";
    permission java.util.PropertyPermission "java.vm.name", "read";
};

```

该文件原先包括两项：第一项是针对 `file:${java.home}/lib/ext/*` 目录下所有代码的，允许 Java 扩展类具有所有权限，第二项是针对所有代码，允许其对各种 Java 属性有读的权限。本实例添加了一项针对 `file:c:/java/ch8/` 目录及其所有子目录下代码的，允许其读 `c:\autoexec.bat` 文件。

这里作为实例是按照代码的目录进行授权，实际使用时可针对某个站点或某个签名的代码进行授权。

★运行程序

按照“代码与分析”部分修改 `C:\Program Files\Java\j2re1.4.0\lib\security\java.policy` 文件，按照 8.6.2 的方式任何一种打开 1.html 或直接打开 2.html 文件，则出现图 8-26 所示窗口。

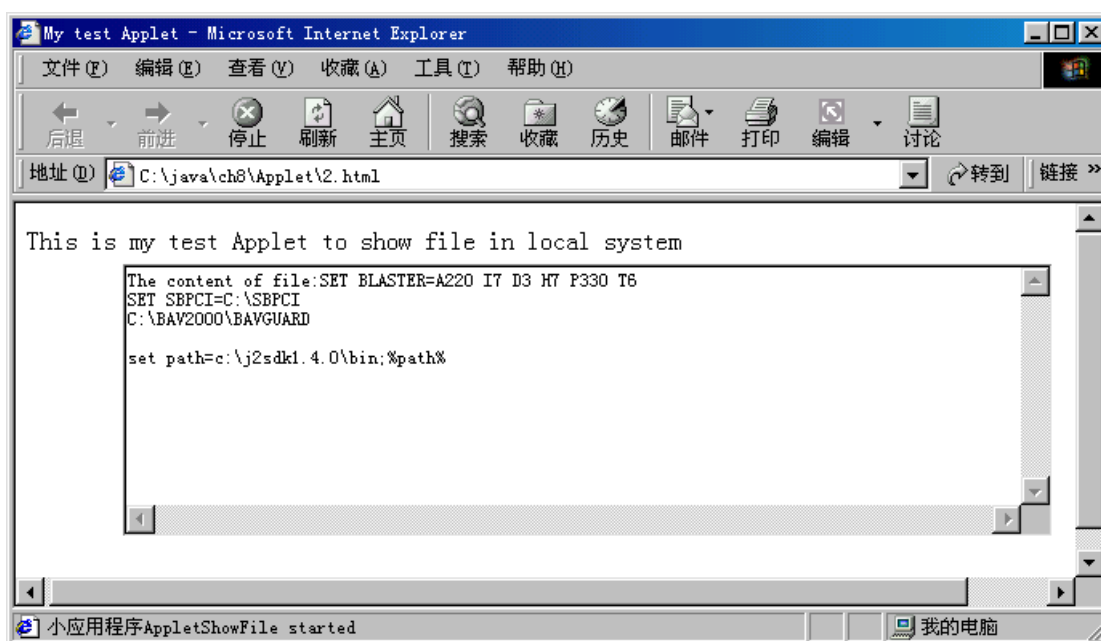


图 8-26 浏览器基于策略文件运行 Java Applet

将 `C:\Program Files\Java\j2re1.4.0\lib\security\java.policy` 文件恢复到修改之前的内容，将 `appletdir.policy` 文件的内容添加到 `c:\windows\java.policy` 文件（如果 `c:\windows` 目录没有 `java.policy` 文件则可在 DOS 中进入 `c:\windows` 目录直接执行“`copy C:\java\ch8\Applet\appletdir.policy java.policy`”）。

此时按照 8.6.2 的方式任何一种打开 1.html 或直接打开 2.html 文件，同样可出现图 8-26 所示窗口。试验完后可将 `c:\windows\java.policy` 文件恢复到修改之前的状态。

8.6.4 浏览器运行 RSA 签名的 Java Applet

★ 实例说明

8.6.3 小节基于策略文件使得浏览器中运行的 Java Applet 也可以访问用户的本机资源，

但是其部署起来比较麻烦，尽管 Java Plug-in 可以通过 8.6.2 小节的第二种方法自动安装，但策略文件却需要每个用户都对自己的或系统的策略文件进行修改。本实例使用 RSA 签名 Java Applet，使得用户只要在访问网页时选择信任签名者的证书即可自动授予 Java Applet 所有权限。

★ 编程思路

通过使用 8.3.1 小节类似的方法对 Java Applet 进行签名，表明签名者已经检查过该 Java Applet 的安全性，保证其没有安全问题并愿意承担所有的安全责任。浏览器中运行 RSA 签名过的 Java Applet 时将自动检验 Applet 签名是否正确，以保证程序没有被第三方篡改过，此外将检验证书是否合法，并弹出对话框，由用户选择是否信任签名者，只有用户信任签名者，该程序才可访问本机资源。

和 8.3.1 小节一样，签名后的 Java Applet 是打包成“.jar”为后缀的文件的，为了使浏览器能够找到该 jar 文件，应该在网页内容中通过<applet>标记的 archive 属性指定 jar 文件的位置和名称。

★代码与分析：

本实例的 Java Applet 程序仍旧使用 8.6.1 小节的 AppletShowFile.java 程序。

打包后的文件不妨使用 myapplet.jar 作为文件名，则调用 Java Applet 的网页文件 3.html 的内容为：

```
<html>
<head>
<title> My test Applet </title>
</head>
<body>
This is my test Applet to show file in local system
<applet code=AppletShowFile.class archive=myapplet.jar
        Width=700 height=200>

</applet>
</body>
</html>
```

类似 8.6.2 小节，经过 HTMLConverter 转换后，使用<object>标记的网页文件内容为：

```
<html>
<head>
<title> My test Applet </title>
</head>
<body>
This is my test Applet to show file in local system
<!--"CONVERTED_APPLET"-->
<!-- HTML CONVERTER -->
<OBJECT
    classid="clsid:CAFEEFAC-0014-0000-0000-ABCDEFEDCBA"
    WIDTH = 700 HEIGHT = 200

codebase="http://java.sun.com/products/plugin/autodl/jinstall-1_4_0-win.c
ab#Version=1,4,0,0">
```

```

        <PARAM NAME = CODE VALUE = AppletShowFile.class >
    <PARAM NAME = ARCHIVE VALUE = myapplet.jar >

    <PARAM
                                                NAME="type"
VALUE="application/x-java-applet;jpi-version=1.4">
        <PARAM NAME="scriptable" VALUE="false">

    <COMMENT>
    <EMBED
        type="application/x-java-applet;jpi-version=1.4"
        CODE = AppletShowFile.class
        ARCHIVE = myapplet.jar
        WIDTH = 700
        HEIGHT = 200
        scriptable=false

    pluginspage="http://java.sun.com/products/plugin/index.html#download">
        <NOEMBED>

    </NOEMBED>
    </EMBED>
    </COMMENT>
</OBJECT>

<!--
    <APPLET CODE = AppletShowFile.class ARCHIVE = myapplet.jar WIDTH = 700
HEIGHT = 200>

    </APPLET>
-->

    <!--"END_CONVERTED_APPLET"-->

</body>
</html>

```

★运行程序

和 8.3.1 小节类似，执行如下命令将 Java Applet 程序打包到 myapplet.jar 文件。

```
C:\java\ch8\Applet>jar cvf myapplet.jar *.class
```

标明清单(manifest)

增加: AppletShowFile.class (读入= 1193) (写出= 716) (压缩了 39%)

将密钥库 lfkeystore2 拷贝到当前目录，执行如下命令使用密钥库 lfkeystore2 中 lf 条目

对应的私钥签名 myapplet.jar 文件。

```
C:\java\ch8\Applet>jarsigner -keystore lfkeystore2 myapplet.jar lf
```

Enter Passphrase for keystore: wshr.ut

然后编写“代码与分析”部分所示的网页 3.html，执行“HTMLConverter 3.html”进行转换，不妨将转换后的 3.html 更名为 4.html，将 c:\java\ch8\applet_BAK 目录中备份的 3.html 拷贝回 c:\java\ch8\applet_BAK 目录。此时，3.html 是转换前的网页，4.html 是转换后的网页。

这样，就可以直接在浏览器中打开 4.html 文件，或按照 8.6.2 小节的方法先设置浏览器再打开 3.html 来运行签名后的 Java Applet 程序了。

浏览器载入 Java Applet 程序后，出现如图 8-27 所示的安全警告：



图 8-27 运行签名小应用程序的提示

该窗口提示将要安装并运行某个签发者签名的 Java Applet，由于用户的计算机中并不信任签发者“Liu Fang”的证书，因此出现了该安全警告信息，让用户选择是否认可。

图 8-27 窗口中，如果用户单击其中的“授予该会话”按钮，则 Applet 开始正常运行，并可根据需要访问用户的本机资源如读 c:\autoexec.bat 文件。以后如果再次运行该 Java Applet 时，仍会出现该页面。

如果单击“总是授权”按钮，则 Applet 同样开始正常运行，同时会把“Liu Fang”的证书安装到 Java Plug-in 信任的证书中。因此以后再运行该 Java Applet 程序时将不出现警告窗口而直接运行 Java Applet 程序。因此，如果只是用于测试，测试完后应按照 8.6.5 小节的方法删除证书。

如果用户想先检查一下签发者，可单击图 8-27 窗口中“查看证书”按钮，则出现图 8-28 窗口所示的签发者“Liu Fang”的证书及其签名。

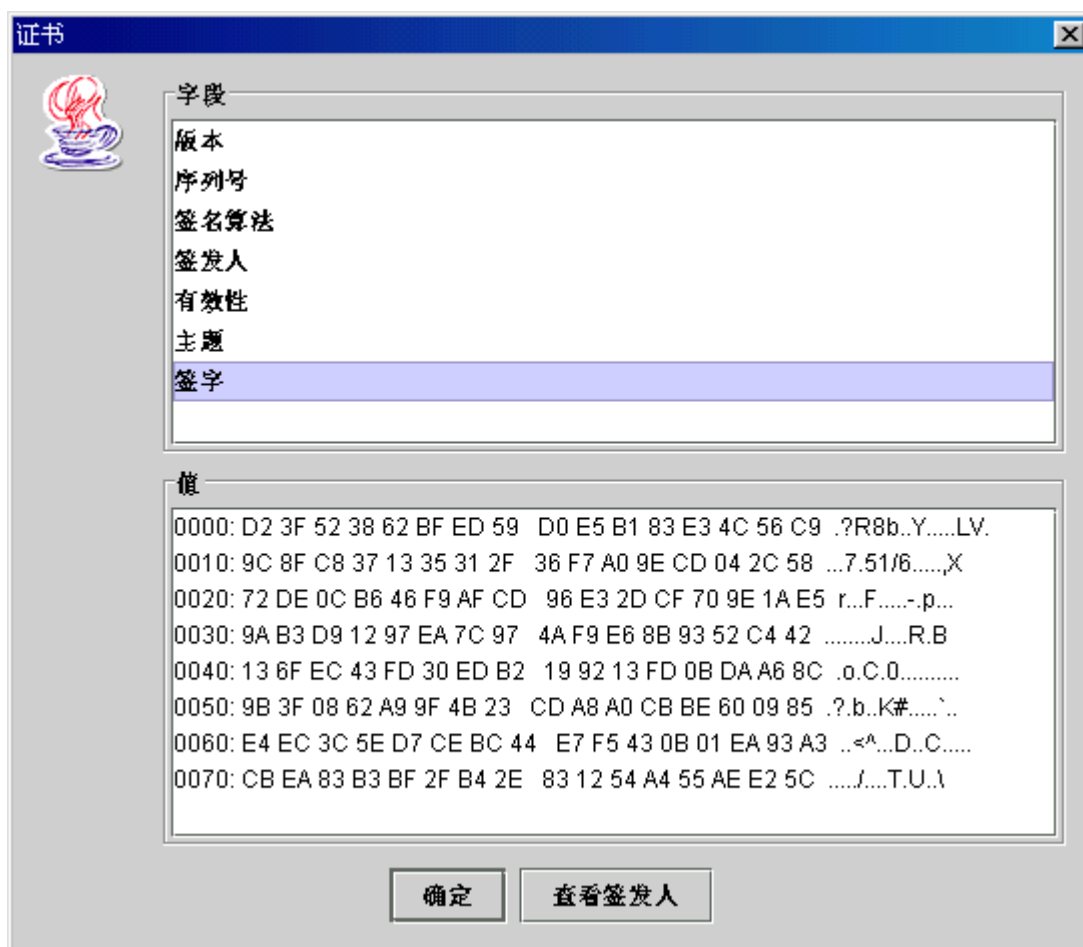


图 8- 28 Java Applet 签发者“Liu Fang”的证书

该窗口中单击各个字段可以查看证书的各种信息，如版本、序列号、签名算法、证书本身的签发者等，如单击“主题”字段，显示的是该证书的主体，如图 8- 29所示。

在图 8- 28或图 8- 29的窗口中进一步单击“查看签发人”按钮，可以进一步查看“Liu Fang”的证书“Xu Yingxiao”的证书。

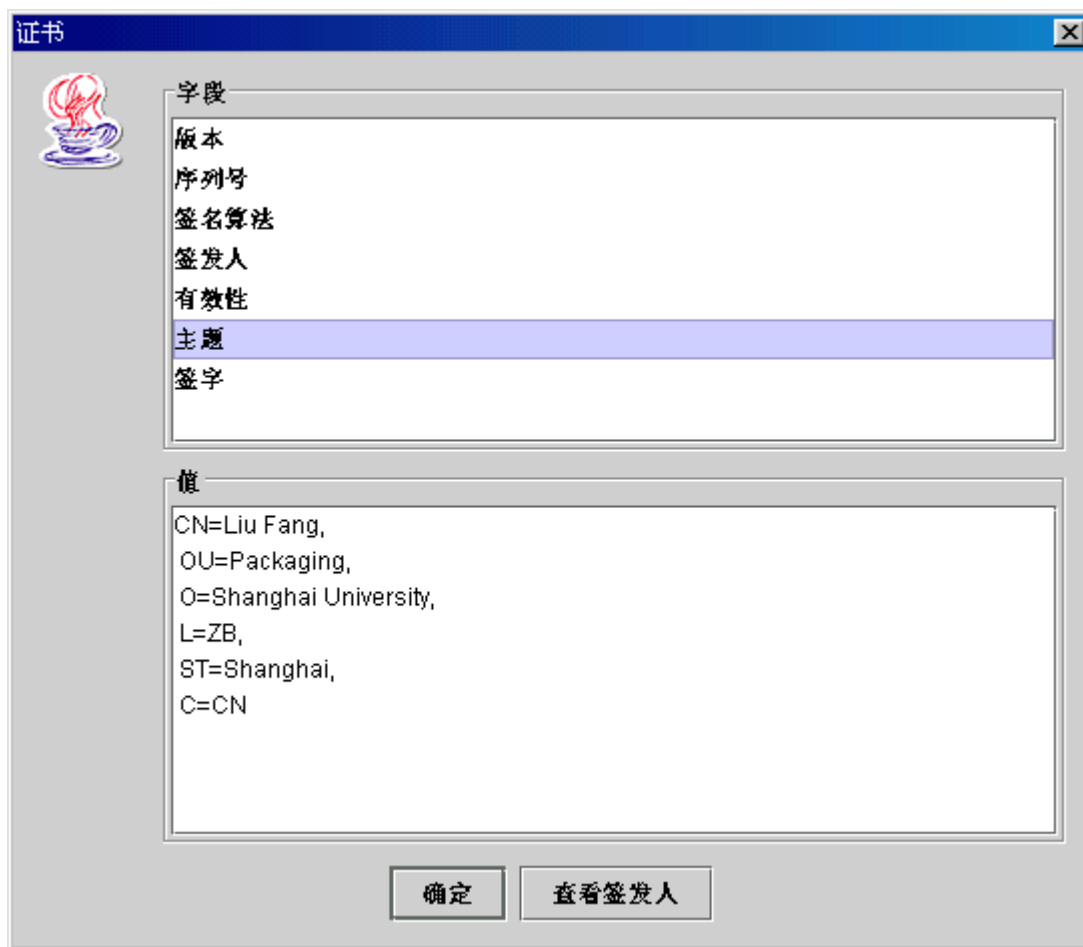


图 8- 29 证书的主体

8.6.5 Java Plug-in 的证书管理

★ 实例说明

本实例使用 Java Plug-in 的控制面板来设置 Java Plug-in 的证书

★ 运行程序

先不妨如 8.6.4 小节打开网页 4.html 运行 Java Applet 程序，在 图 8- 27 的窗口中选择“总是授权”按钮，将签名者的证书添加入 Java Plug-in 的证书列表，以后再打开网页 4.html 时就不会出现 图 8- 27 的窗口而直接运行了。

下面启动 Java Plug-in 控制面板来管理添加进来的证书。双击 Windows 控制面板中的“Java Plug-in 1.4.0”图标，将启动 图 8- 30 所示的 Java Plug-in 控制面板。其中可以设置 Java Plug-in 的各种选项。

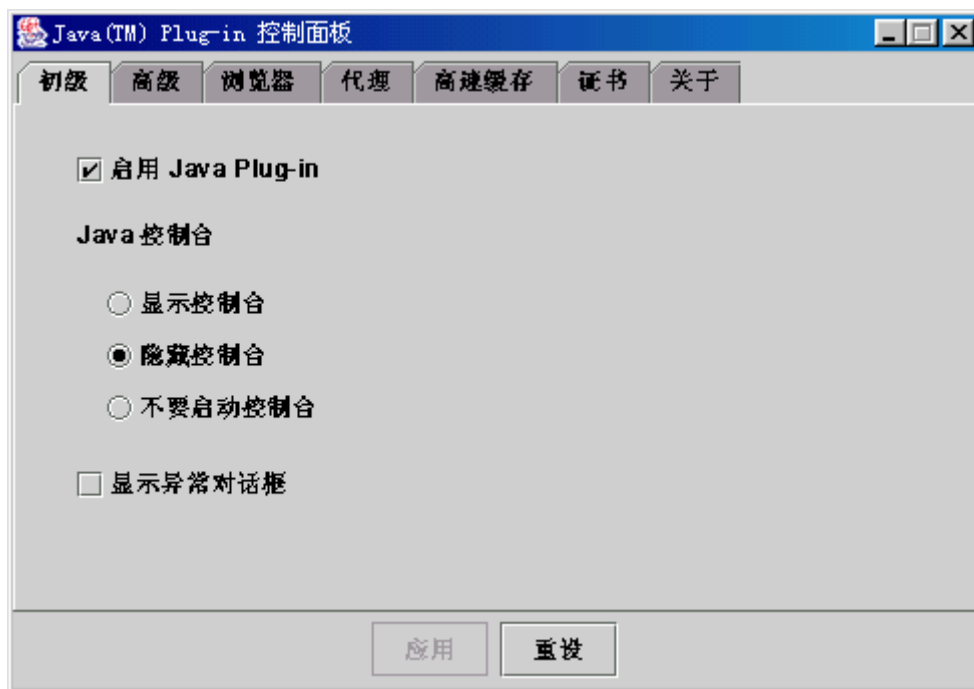


图 8-30 Java Plug-in 控制面板启动窗口

单击其中的“证书”标签可以管理Java Plug-in中的证书，如图 8-31所示。



图 8-31 Java Plug-in 控制面板证书窗口

图 8-31的窗口中，显示了由“Xu Yingxiao”签发的“Liu Fang”的证书。单击“详细信息”按钮可以查看证书的详细信息。该证书是在图 8-27的窗口中选择“总是授权”按钮时自动添加进来的，也可以单击图 8-31的窗口中的“导入”按钮将“Liu Fang”的证书文件导入，或选择“导出”按钮将Java Plug-in中的证书导出到文件。某个人或机构的证书只要证书出现在图 8-31的窗口中，Java Plug-in运行其签名的Java Applet时就不会出现图 8-27要求确认

的窗口而可以自动运行，并可访问用户的本机资源。

图 8-31 的窗口中共显示了四种证书。

第一种称为“带签名的小程序”，它表明用户信任这些证书签名的小程序。它对应于用户主目录（如对于 Windows9X 为 c:\windows）下 .java 子目录中 jpicerts140 文件（如果是其他版本的 Java Plug-in 则文件名中 140 相应跟着变化）。这些证书可以使用“导入”、“导出”、“删除”或“详细信息”按钮进行操作。

第二种称为“安全站点”，它们是用于安全站点的证书，对应于用户主目录（如对于 Windows9X 为 c:\windows）下 .java 子目录中 jpihttpscerts140 文件（如果是其他版本的 Java Plug-in 则文件名中 140 相应跟着变化）。这些证书可以使用“导入”、“导出”、“删除”或“详细信息”按钮进行操作。

第三种称为“签名者 CA”，带签名的小程序的签发者的证书可由它们签发。这些证书可以使用“详细信息”按钮查看证书内容。

第四种称为“安全站点 CA”，安全站点的签发者的证书可由它们签发。这些证书可以使用“详细信息”按钮查看证书内容。

这里不妨将“Liu Fang”证书删除，此时如 8.6.4 小节再次打开网页 4.html 运行 Java Applet 程序时，又会出现图 8-27 的窗口。如果在图 8-31 的窗口中单击“导入”按钮，在出现的选择文件窗口输入“*.cer”，导入 C:\java\ch8\sign\demo1\A 目录下的 lf.cer 证书，则以后再打开网页 4.html 时就不会出现图 8-27 的窗口而直接运行了。

8.6.6 使用 usePolicy 权限加强 RSA 签名 Applet 的安全控制

★ 实例说明

使用 RSA 签名的 Java Applet 尽管大大方便了 Java Applet 的部署，但是 RSA 签名的 JavaApplet 不再受策略文件限制而可以访问所有的用户资源，这样就难以单独针对不同资源分别进行授权控制。本实例使用 usePolicy 权限使得 Java Applet 同样受策略文件控制。

★ 编程思路

Java Plug-in 会首先检查默认的系统策略文件或用户策略文件中是否设置了 usePolicy 权限，由于默认的系统策略文件没有设置此权限，因此如果用户信任 RSA 签名的 Applet，则 Applet 可以访问所有用户资源。而用户如果在系统策略文件或用户主目录中默认的策略文件中加上 usePolicy 权限，则 Java Plug-in 会使用策略文件中定义的权限。

★ 代码与分析：

本实例在默认的系统策略文件 C:\Program Files\Java\j2re1.4.0\lib\security\java.policy 文件或默认的用户主目录策略文件 c:\windows\java.policy 文件加入如下代码

```
grant {  
    permission java.lang.RuntimePermission "usePolicy";  
    permission java.io.FilePermission "c:\\autoexec.bat", "read";  
};
```

以上代码对所有的 RSA 签名的代码都不允许其访问所有资源，而只允许其访问策略文件中规定的资源。

如果只针对“Liu Fang”签名的代码，则可使用如下代码：

```

keystore "file:/c:/java/ch8/sign/demo1/b/clienttrust1f", "JKS";

grant signedBy "lf" {
    permission java.lang.RuntimePermission "usePolicy";
    permission java.io.FilePermission "c:\\autoexec.bat", "read";
};

```

这些代码中，“permission java.lang.RuntimePermission "usePolicy";”和“permission java.io.FilePermission "c:\\autoexec.bat", "read";”可以根据需要任意组合。如可以在同一项“grant”或“grant signedBy "lf"”中，也可以分开在不同项中；可以在同一个文件，也可以一个在 C:\Program Files\Java\j2re1.4.0\lib\security\java.policy 文件，而另一个在 c:\windows\java.policy 文件中。

★运行程序

先在系统策略文件 C:\Program Files\Java\j2re1.4.0\lib\security\java.policy 文件或默认的用户主目录策略文件 c:\windows\java.policy 文件中只加入

```

grant {
    permission java.lang.RuntimePermission "usePolicy";
}

```

或

```

keystore "file:/c:/java/ch8/sign/demo1/b/clienttrust1f", "JKS";
grant signedBy "lf" {
    permission java.lang.RuntimePermission "usePolicy";
};

```

在浏览器中打开网页 4.html 运行 Java Applet 程序，将发现无法显示 c:\autoexec.bat 文件的内容。这是因为 Java Plug-in 在默认策略文件中发现了 usePolicy 权限，因而不出现图 8-27 所示的窗口，而直接使用策略文件中的设置。而此时策略文件中为授权该代码访问 c:\autoexec.bat 程序，因而产生异常。

然后关闭浏览器，在系统策略文件 C:\Program Files\Java\j2re1.4.0\lib\security\java.policy 文件或默认的用户主目录策略文件 c:\windows\java.policy 文件中加入：

```

permission java.io.FilePermission "c:\\autoexec.bat", "read";

```

则再次在浏览器中打开网页 4.html 时不提示图 8-27 所示的窗口而直接显示出 c:\autoexec.bat 文件的内容。

最后，将策略文件中

```

permission java.lang.RuntimePermission "usePolicy";

```

删除，而只保留

```

permission java.io.FilePermission "c:\\autoexec.bat", "read";

```

此时，该设置对网页 4.html 所执行的 Java Applet 不起作用，因为该 Java Applet 是 RSA 签名的，而 Java Plug-in 没有检测到默认策略文件中对其授予过 usePolicy 权限，因而出图 8-27 所示的窗口，当用户授权后将不再使用策略文件中的设置而直接赋予所有权限。

本章介绍了 Java 程序安全运行的第一种控制方式：通过代码的 URL 和签名设置代码具有何种权限。用户可在运行 Java 程序时指定安全管理器限制所有危险操作，然后通过策略

文件分别授予指定的权限。

在下一章将介绍另一种控制方式：对相同的或不同的代码，根据执行者的身份确定允许其作何种操作。

第 9 章 程序运行的安全性——

基于用户身份的验证和授权 (JAAS)

本章重点:

第 8 章介绍的安全运行主要判断代码的来源（位置或身份），进而控制哪些来源的代码可以访问用户的哪些资源，以保护代码执行者的安全性。

本章则介绍如何在程序中判断代码执行者的身份，从而判断哪些身份的用户可以访问哪些公共资源，以保护系统的安全。

相关的技术有很多，本章介绍 Java 验证和授权服务 (JAAS)。JAAS 在 J2SDK 1.3 版本中是一个可选的包，现在已经集成到 J2SDK 1.4 版本中了，可以直接使用。本章以 Java Application 为例介绍 JAAS，同样的编程方法可用于 Java Applet、Java Servlet 或 Java Bean 等。

本章主要内容:

- 使用 JAAS 提供的登录模块和回调处理器执行身份验证
- 以各种方式更换登录模块和回调处理器
- 根据需要编写自己的登录模块和回调处理器
- 通过设置登录模块的选项，实现各种堆叠式登录
- 在多个登录模块之间传递信息
- 通过策略文件以声明的方式基于身份授权
- 通过编程方式基于身份授权
- 正确使用 `doAsPrivileged()` 和 `doAs()` 方法

9.1 最简单的身份验证

本节首先给出通过 Java 命令行选项指定默认安全管理器实现程序运行的安全的实例，然后给出如何定义自己的安全管理器满足特殊需要，最后给出程序中指定安全管理器的方法。

9.1.1 最简单的登录

★ 实例说明

本实例演示了最简单的 JAAS 登录，它从给出对话框读取用户输入的信息，通过密钥库中的信息进行验证。

★ 编程思路:

身份验证涉及如何同用户交互以获取用户的信息(账号、口令、指纹等)、如何根据用户输入的信息验证用户等。和用户交互获取用户信息使用回调处理器,本实例中使用 JAAS 提供的 `com.sun.security.auth.callback` 包中的 `DialogCallbackHandler` 类;验证用户信息使用登录模块,本实例中使用 JAAS 提供的 `com.sun.security.auth.module` 包中的 `KeyStoreLoginModule` 类。这些类都已经集成在 J2SDK1.4 中。

JAAS 使用可插入式验证模块(PAM)的结构,开发者可以不管具体的验证方式而使用标准的接口进行开发,使用何种验证技术以及如何验证可交给系统管理员通过登录配置文件进行配置,而不需要修改应用程序的代码。

登录配置文件是一个普通的文本文件,包含了使用哪些登录模块,类似如下格式:

```
simp {  
    com.sun.security.auth.module.KeyStoreLoginModule required  
    keyStoreURL="file:C:/java/ch9/SimpLogin/mykeystore";  
};
```

在运行 Java 程序时可以通过命令行选项指定使用该配置文件中指定的方法进行验证。其中“simp”是配置条目的名称,一个配置文件中可以有多个条目。`com.sun.security.auth.module.KeyStoreLoginModule` 是登录模块类的全名,这里使用 JAAS 提供的 `KeyStoreLoginModule` 类,它采用基于密钥库的验证机制,通过 `keyStoreURL` 指定密钥库的位置。只有当用户输入的别名和密码和密钥库中的相吻合,验证才通过。

JAAS 的登录机制使用 `javax.security.auth.login` 包中的 `LoginContext` 类实现独立于底层验证技术的登录。在创建 `LoginContext` 对象时通过其参数指定使用哪个回调处理器和用户进行交互,以及如何获得登录模块。执行其 `login()` 方法则自动执行登录模块的登录操作,由登录模块调用回调处理器向用户询问账号、口令等相关信息,并进行验证。如果没有通过验证,将抛出 `LoginException` 异常。具体编程步骤如下:

(20) 创建和用户交互的回调处理器对象

```
DialogCallbackHandler handler=new DialogCallbackHandler();
```

分析:不妨使用 `com.sun.security.auth.callback` 包中的 `DialogCallbackHandler` 类,它使用 Swing 对话框向用户询问与验证相关的问题。针对不同的验证方式, `DialogCallbackHandler` 会出现不同的窗口。如对本节的使用密钥库的验证方式,它会要求用户输入别名、密钥库密码和私钥密码。

除了 `DialogCallbackHandler` 类外, `com.sun.security.auth.callback` 包还提供了 `TextCallbackHandler`,以文本方式和用户交互。

(21) 创建 LoginContext 对象

```
LoginContext c = new LoginContext("simp",handler);
```

分析: `LoginContext` 类的构造器有两个参数,第一个是登录配置文件中的条目名称,JAAS 会读取登录配置文件时会找到该条目,并读取其中的登录模块,进而执行这些登录模块。第二个参数是上一步创建的回调处理器,以后登录模块会通过它和用户进行交互。

(22) 执行登录操作

```
c.login();
```

分析:执行上一步得到的 `LoginContext` 对象的 `login()` 方法,该方法会自动执行登录模块的 `login()` 和 `commit()` 方法进行登录操作。而登录模块的 `login()` 方法会调用第 1 步的回调处理器读取用户输入的信息,并验证这些信息和数据库中的信息是否

匹配。

该步骤若验证失败则抛出 `LoginException` 异常。

(23) 处理登录结果

```
Subject s = c.getSubject();
System.out.println(s.getPrincipals());
```

分析：登录成功，则可以执行 `LoginContext` 对象的 `getSubject()` 方法获得代表登录者的主体，并可进一步执行其 `getPrincipals()` 方法获得其身份标志。如果登录不成功，则可以处理 `LoginException` 对象，可根据需要返回出错信息或重复登录。

★代码与分析：

本实例所使用的登录配置文件 `simp.config` 内容如下：

```
simp {
    com.sun.security.auth.module.KeyStoreLoginModule required
        keyStoreURL = "file:C:/java/ch9/SimpLogin/mykeystore";
};
```

使用 `LoginContext` 类实现登录的完整程序如下：

```
import com.sun.security.auth.callback.DialogCallbackHandler;
import javax.security.auth.*;
import javax.security.auth.login.*;

public class SimpLogin {
    public static void main(String[] args) throws Exception {
        //登录
        DialogCallbackHandler handler=new DialogCallbackHandler( );
        LoginContext c = new LoginContext("simp",handler);
        boolean pass;
        try {
            c.login();
            //登录成功
            pass=true;
        }
        catch (LoginException le) {
            //登录失败
            pass=false;
            System.err.println("Authentication failed:");
            System.err.println(" " + le.getMessage());
        }
        //显示登录结果
        if(!pass){
            System.out.println("Sorry");
        }
        else{
            System.out.println("Authentication succeeded!");
        }
    }
}
```

```

        Subject s = c.getSubject();
        System.out.println(s.getPrincipals());
    }
}
}

```

★运行程序

程序运行在 C:\java\ch9\SimpLogin 目录下，该目录中有编程者编写的 SimpLogin 程序。程序运行者使用 simp.config 登录配置文件，该配置文件中使用了 5.1.3 小节创建的密钥库 mykeystore，该密钥库拷贝在当前目录中，密码是 wshr.ut，有一个条目 mytest。

输入

```
java -Djava.security.auth.login.config==simp.config SimpLogin
```

运行程序，其中命令行选项-Djava.security.auth.login.config指定登录配置文件的名称。程序运行将出现图 9-1所示窗口：

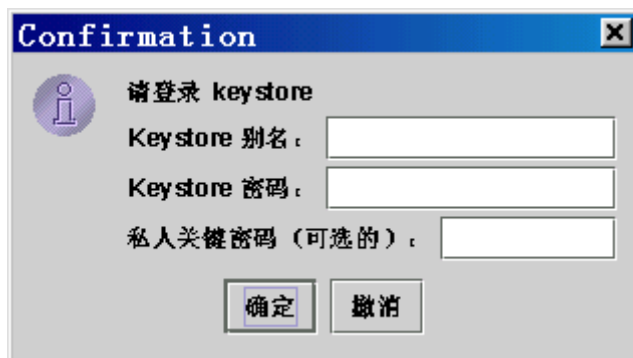


图 9-1 登录窗口

该窗口便是程序中创建 LoginContext 类时所传入的 com.sun.security.auth.callback.DialogCallbackHandler 类型的对象弹出的。在其中用户可以输入别名 mytest，密钥库密码 wshr.ut。由于在 5.1.3 小节创建该密钥库时 mytest 条目对应的私钥使用了和密钥库一样的密码，所以窗口中“私人关键密码”（即条目的主密码、保护私钥的密码）可以不填。

正确输入信息后，单击“确定”按钮，DOS 窗口将显示：

```
Authentication succeeded!
```

```
[CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai, C=CN]
```

可见，验证通过后程序可获取用户的身份。

如果输入的信息和密钥库中不匹配，如在密码一项中输入的不是“wshr.ut”，则提示验证失败：

```
Authentication failed:
```

```
Error initializing keystore: java.io.IOException: Keystore was tampered with,or
password was incorrect
```

```
Sorry
```

9.1.2 更换登录模块修改验证方式

★ 实例说明

本实例演示 JAAS 的可插入式特性，在不修改 9.1.1 小节的应用程序的基础上更换登录模块，

★ 编程思路：

JAAS 中更换登录模块只要修改配置文件即可，不需要修改应用程序。在 9.1.1 小节中，登录模块使用的是 `com.sun.security.auth.module` 包中的 `KeyStoreLoginModule` 类，在该包中还有很多其它类可直接使用。如：

- `JndiLoginModule` 类，向用户询问账号和口令，然后通过 JNDI 和保存在目录服务中的口令进行比较以进行验证。
- `Krb5LoginModule` 类使用 Kerberos 协议验证用户。
- `NTLoginModule` 类读取 NT 当前登录用户的身份标志信息。
- `UnixLoginModule` 类根据当前登录的用户获取 Unix 用户的身份标志信息。

需要时也可以自己编写登录模块。

此外，在 9.1.1 小节中使用 `KeyStoreLoginModule` 登录模块时使用了一个选项：`keyStoreURL` 指定了密钥库的位置，如果没有设置该选项则会使用 `java.home` 属性指定目录的“`.keystore`”文件。需要时还可以使用其它选项，不同模块可以使用的选项不同，查看 `KeyStoreLoginModule` 类的 API 文档，可以发现还可以使用以下选项：

- `KeyStoreType`
指定密钥库类型，如果不指定则使用 `KeyStore.getDefaultType()` 获得的缺省密钥库类型。
- `KeyStoreProvider`
指定密钥库提供者，如“`Sun`”。如果不指定则使用标准的搜寻顺序查找。
- `keyStoreAlias`
指定要登录到密钥库的哪个别名上。该选项没有默认值，如果创建 `LoginContext` 对象时没有指定回调处理器，即不和用户交互，则需要指定该选项。
- `KeyStorePasswordURL`
指定密钥库密码存在哪个 URL（如某个目录或 Web 站点的文件中）。该选项没有默认值，如果创建 `LoginContext` 对象时没有指定回调处理器，即不和用户交互，则需要指定该选项。
- `privateKeyPasswordURL`
`keyStoreAlias` 选项已经指定了别名，`privateKeyPasswordURL` 选项则指定密钥库中访问该别名对应的私钥应该使用何密码，和 `KeyStorePasswordURL` 一样，密码可以存在某个目录或 Web 站点的文件中。如果没有指定该选项，则使用 `privateKeyPasswordURL` 中同样的值。

本实例将修改登录策略文件，演示不同选项的效果以及 `UnixLoginModule` 的使用。

★代码与分析：

本实例所使用的 Java 程序和 9.1.1 小节相同。

登录配置文件 `keystore2.config` 增加了别名，则以后该登录模块执行回调处理器时将自动在别名中填上“`mytest`”，用户输入口令即可，该登录配置文件内容如下：

```
simp {  
com.sun.security.auth.module.KeyStoreLoginModule required
```



```

keyStoreURL ="file:C:/java/ch9/SimpLogin/mykeystore"
keyStoreAlias=mytest;
};
登录配置文件 Unix.config 使用 UnixLoginModule, 其内容如下:
simp {
    com.sun.security.auth.module.UnixLoginModule required;
};

```

★运行程序

程序运行在 C:\java\ch9\SimpLogin 目录下, 该目录中有 9.1.1 小节的 SimpLogin 程序, 以及新编写的 keystore2.config 和 Unix.config 配置文件。

输入

```
java -Djava.security.auth.login.config==keystore2.config SimpLogin
```

运行程序, 将出现如下对话框

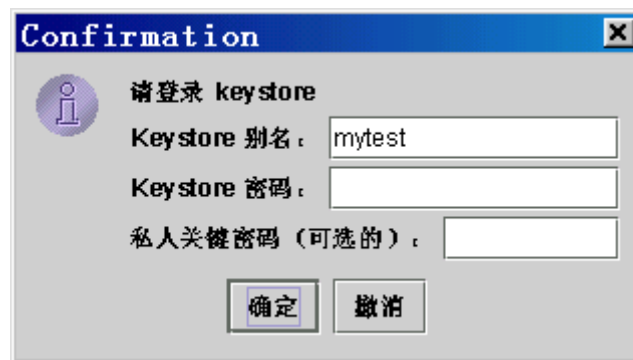


图 9-2 自动输入别名的登录窗口

其中密钥库的别名已经自动填写在上面, 输入正确的密码后将和 9.1.1 小节一样输出

Authentication succeeded!

[CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai, C=CN]

Unix.config 中使用的是 Unix 用户的信息 (如果没有 Unix 平台, 也可以使用 NTLoginModule 在 NT 平台上运行), 在一台 Unix 机器上以用户 xyx 登录 Unix 操作系统, 输入:

```
java -Djava.security.auth.login.config=Unix.config SimpLogin
```

运行程序, 将出现类似如下的输出:

Authentication succeeded!

[UnixPrincipal: xyx, UnixNumericUserPrincipal: 1005, UnixNumericGroupPrincipal [主群组]: 10]

它获取了当前用户的身份, 以后可以使用该身份进行各种操作。

如果换一个用户身份, 以 root 登录 Unix 平台, 输入:

```
java -Djava.security.auth.login.config=Unix.config SimpLogin
```

运行程序, 将出现类似如下的输出:

Authentication succeeded!

[UnixPrincipal: root, UnixNumericUserPrincipal: 0, UnixNumericGroupPrincipal [主群

组]: 1, UnixNumericGroupPrincipal [附加群组]: 0, UnixNumericGroupPrincipal [附加群组]: 2, UnixNumericGroupPrincipal [附加群组]: 3, UnixNumericGroupPrincipal [附加群组]: 4, UnixNumericGroupPrincipal [附加群组]: 5, UnixNumericGroupPrincipal [附加群组]: 6, UnixNumericGroupPrincipal [附加群组]: 7, UnixNumericGroupPrincipal [附加群组]: 8, UnixNumericGroupPrincipal [附加群组]: 9, UnixNumericGroupPrincipal [附加群组]: 12]

9.1.3 更换回调处理器修改登录界面

★ 实例说明

9.1.1 小节的程序使用了图形化的登录界面，本实例演示如何将其修改为文本的登录界面，本实例同时演示了非交互式登录。

★ 编程思路：

9.1.1 小节程序的图形化登录界面是由 `com.sun.security.auth.callback` 包中的 `DialogCallbackHandler` 类提供的，在 `com.sun.security.auth.callback` 包中还提供了 `TextCallbackHandler` 用于以文本方式交互登录。只要将 9.1.1 小节程序中

```
DialogCallbackHandler handler=new DialogCallbackHandler();
```

改为

```
TextCallbackHandler handler=new TextCallbackHandler();
```

并将程序开头的

```
import com.sun.security.auth.callback.DialogCallbackHandler;
```

改为

```
import com.sun.security.auth.callback.TextCallbackHandler;
```

即可。其完整代码见 `SimpLoginTXT.java`。

★代码与分析：

本实例所使用的 Java 程序如下：

```
import com.sun.security.auth.callback.TextCallbackHandler;
import javax.security.auth.*;
import javax.security.auth.login.*;

public class SimpLoginTXT {
    public static void main(String[] args) throws Exception {
        //登录
        TextCallbackHandler handler=new TextCallbackHandler();
        LoginContext c = new LoginContext("simp",handler);
        boolean pass;
        try {
            c.login();
            //登录成功
            pass=true;
        }
```

```

    }
    catch (LoginException le) {
        //登录失败
        pass=false;
        System.err.println("Authentication failed:");
        System.err.println("  " + le.getMessage());
    }
    //显示登录结果
    if(!pass){
        System.out.println("Sorry");
    }
    else{
        System.out.println("Authentication succeeded!");
        Subject s = c.getSubject();
        System.out.println(s.getPrincipals());
    }
}
}
}

```

★运行程序

程序运行在 C:\java\ch9\SimpLogin 目录,使用其中的 SimpLoginTXT 程序和 Simp.config 文件。

输入

```
C:\java\ch9\SimpLogin>java -Djava.security.auth.login.config==Simp.config SimpLoginTXT
```

请登录 keystore

Keystore 别名: mytest

Keystore 密码: wshr.ut

私人关键密码 (可选的):

0. OK [default]

1. Cancel

Enter a number: 0

Authentication succeeded!

[CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai, C=CN]

此时登录模块使用 TextCallbackHandler 会调处理器和用户交互,通过用户读取别名、密码等信息。

9.1.4 使用非交互式验证

★ 实例说明

在 9.1.2 小节中使用 UnixLoginModule 或 NTLoginModule 登录模块时并没有出现登录界面,而是直接读取操作系统当前登录用户的信息。对于 KeyStoreLoginModule 登录模块,提

供了足够的选项后也可以实现非交互式验证。

非交互式验证不需要回调处理器，本节介绍其编程方法，以及如何通过安全配置文件指定回调处理器。

★ 编程思路：

9.1.2 小节的程序中使用 `UnixLoginModule` 或 `NTLoginModule` 登录模块时，应用程序仍然指定了回调处理器，但并没有使用该回调处理器。其实，也可以使用不带回调处理器的 `LoginContext` 对象，只要将 9.1.1 小节的第 1，2 步合并为一个语句：

```
LoginContext c = new LoginContext("simp");
```

使用只有一个参数的构造器、不带回调处理器的登录可以不使用用户交互，直接从配置文件中获取用户的信息，如 9.1.2 小节所使用的 `Unix.config` 中的 `UnixLoginModule` 就不需要用户交互，直接从当前使用操作系统的用户中获取信息。此外 `KeyStoreLoginModule` 中指定了 `keyStoreURL`、`keyStoreAlias`、`KeyStorePasswordURL` 选项后也不需要用户交互而可直接获取别名和口令信息来和密钥库相匹配。

应用程序使用只有一个参数的 `LoginContext` 构造器时，如果希望使用交互方式进行验证，可以通过 `java.security` 文件中的 `auth.login.defaultCallbackHandler` 的 Java 安全属性来指定回调处理器。J2SDK 1.4 中，`java.security` 文件在 `C:\j2sdk1.4.0\jre\lib\security` 目录中，只要在其中加上一句：

`auth.login.defaultCallbackHandler=com.sun.security.auth.callback.DialogCallbackHandler`
或 `auth.login.defaultCallbackHandler=com.sun.security.auth.callback.TextCallbackHandler`
即可。

★代码与分析：

本实例所使用的 Java 程序如下：

```
import javax.security.auth.*;
import javax.security.auth.login.*;

public class SimpLogin2 {
    public static void main(String[] args) throws Exception {
        //登录
        LoginContext c = new LoginContext("simp");
        boolean pass;
        try {
            c.login();
            //登录成功
            pass=true;
        }
        catch (LoginException le) {
            //登录失败
            pass=false;
            System.err.println("Authentication failed:");
            System.err.println(" " + le.getMessage());
        }
        //显示登录结果
        if(!pass){
            System.out.println("Sorry");
        }
    }
}
```

```

        else{
            System.out.println("Authentication succeeded!");
            Subject s = c.getSubject();
            System.out.println(s.getPrincipals());
        }
    }
}

```

本实例使用的登录配置文件除了 9.1.1 和 9.1.2 小节的 `Simp.config` 和 `keystore2.config` 外，还编写了 `keystore3.config` 文件，为 `KeyStoreLoginModule` 提供更多的选项，以实现非交互式的登录，其内容如下：

```

simp {
    com.sun.security.auth.module.KeyStoreLoginModule required
        keyStoreURL = "file:C:/java/ch9/SimpLogin/mykeystore"
        keyStoreAlias=mytest
        keyStorePasswordURL = "file:C:/java/ch9/SimpLogin/passwd.txt"
        privateKeyPasswordURL = "file:C:/java/ch9/SimpLogin/passwd.txt";
};

```

其中口令存放在 `c:\java\ch9\SimpLogin\passwd.txt` 文件中，该文件只包含一串字符：`wshr.ut`，是密钥库的口令。

此外，在运行程序时会修改 `C:\j2sdk1.4.0\jre\lib\security\java.security` 文件，加上
`auth.login.defaultCallbackHandler=com.sun.security.auth.callback.DialogCallbackHandler`
 或

`auth.login.defaultCallbackHandler=com.sun.security.auth.callback.TextCallbackHandler`
 以观察不同的配置效果。

★运行程序

程序运行在 `C:\java\ch9\SimpLogin` 目录，使用其中的 `SimpLogin2.class` 和 `keystore3.config` 文件。

输入

```
java -Djava.security.auth.login.config==keystore3.config SimpLogin2
```

运行程序，程序不出现窗口而直接显示：

```

Authentication succeeded!
[CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai,
C=CN]

```

此时程序根据 `keystore3.config` 中的配置，从 `c:\java\ch9\SimpLogin\passwd.txt` 文件读取密钥库口令，访问 `c:\java\ch9\SimpLogin\mykeystore` 密钥库中的 `mytest` 条目，可以成功访问，因而验证通过。如果修改 `c:\java\ch9\SimpLogin\passwd.txt` 文件中提供的口令值，则无法通过验证。

此外，也可以和 9.1.2 小节类似，在 Unix 平台的计算机中输入

```
java -Djava.security.auth.login.config=Unix.config SimpLogin2
```

同样可以进行非交互式的登录。

如果要更改用户交互的方式，可以修改 `C:\j2sdk1.4.0\jre\lib\security\java.security` 文件，在最后一行加上一句：

`auth.login.defaultCallbackHandler=com.sun.security.auth.callback.DialogCallbackHandler`
此时若LoginContext的构造器中没有指定回调处理器，则将强制使用DialogCallbackHandler提供的图形化登录界面，无论输入

```
java -Djava.security.auth.login.config==keystore3.config SimpLogin2
java -Djava.security.auth.login.config==keystore2.config SimpLogin2
```

还是

```
java -Djava.security.auth.login.config==Simp.config SimpLogin2
```

都将出现图 9-1或图 9-2的登录界面。

如果在 `C:\j2sdk1.4.0\jre\lib\security\java.security` 文件最后一行加上的是

`auth.login.defaultCallbackHandler=com.sun.security.auth.callback.TextCallbackHandler`
则将使用 `com.sun.security.auth.callback` 包中所提供的文本登录界面。其交互过程和 9.1.3 小节相同。此时登录模块使用 TextCallbackHandler 会调处理器和用户交互,通过用户读取别名、密码等信息。

修改了 `C:\j2sdk1.4.0\jre\lib\security\java.security` 文件后若运行 9.1.1 小节的程序,则仍旧是图形登录界面,因为 9.1.1 小节中已经在程序中指定了回调处理器。

测试完毕可将 `C:\j2sdk1.4.0\jre\lib\security\java.security` 文件恢复到原来的内容。

9.2 编写自己的登录模块

在 9.1 节的例子中使用了 JAAS 提供的几个登录模块,当底层的验证机制需要发生变化时,可以修改登录配置文件指定不同的登录模块,当 JAAS 提供的登录模块满足不了要求时,可以编写自己的登录模块。自己的登录模块可以独立于应用程序、并可由其他程序员进行编写。

9.2.1 简单的登录模块

★ 实例说明

本实例提供了一个简化的登录模块,演示了登录模块的工作原理和编程方法。

★ 编程思路:

9.1.1 小节通过 LoginContext 对象实现了独立于底层验证技术的登录。LoginContext 对象通过 LoginModule 接口操作登录模块,自动读取并加载登录配置文件中指定的登录模块,创建登录模块的对象。登录模块对象创建后,会马上执行 initialize()方法进行各种初始化操作。

LoginContext 对象的执行 LoginContext 对象 login()方法进行登录时,其验证过程包括两步,首先会自动调用各个登录模块的 login()方法进行登录,若登录模块的 login()方法登录成功,则第二步为执行各个登录模块的 commit()方法更新主体。如果登录失败(登录模块的 login()方法或 commit()方法抛出 LoginException 异常),则执行登录模块的 abort()方法。因此只有两步验证都成功了,整个验证过程才成功。

如果执行 LoginContext 对象执行 logout()方法退出登录,则会执行登录模块的 logout()方法完成此操作。

因此,编写自己的登录模块只要编写类实现 LoginModule 接口中的 initialize()、login()、commit()、abort()和 logout()方法即可。

登录模块确认登录成功后，需要向代表用户的主体添加用户身份标志，以后进行授权时可以在策略文件中指定具有何种身份标志的用户拥有何权限。因此，在编写登录模块之前要先编写一个类实现 `Principal` 接口，类名不妨使用 `MyPrincipal`，该类中要实现以下方法 `getName()`、`hashCode()`、`toString()` 和 `equals()`。

这些 `Principal` 接口的方法中，大部分只需要一个 `return` 语句返回与该身份标志的名称相关的内容即可。只有 `equals()` 方法用于该 `Principal` 对象与参数中的对象是否相同。不妨规定当两个 `Principal` 对象的名字相同时即相同，这样可以如下定义 `equals()` 方法。

```
public boolean equals(Object obj) {
    if ((obj!=null)&& (obj instanceof MyPrincipal)) {
        MyPrincipal obj2 = (MyPrincipal)obj;
        if (name.equals(obj2.getName())) {
            return true;
        }
    }
    return false;
}
```

该段代码中首先判断 `equals()` 方法传入的对象是否是 `MyPrincipal` 类型，若是，则转换为 `MyPrincipal` 类型并比较其名称与执行 `equals()` 方法的对象名称是否相同，相同则返回 `true`，否则返回 `false`。

有了 `MyPrincipal` 类以后，可以如下编写 `LoginModule` 的各个方法。

- 实现 `initialize()` 方法

```
public void initialize(Subject subject, CallbackHandler callbackHandler,
                      Map sharedState, Map options) {
    this.subject = subject;
    this.callbackHandler = callbackHandler;
}
```

分析：该方法将参数中的各个对象传递给登录模块的成员变量。

在 9.1.1 小节的程序中，创建 `LoginContext` 对象使用了两个参数的构造器，分别传入配置文件中的条目名称和回调处理器。在需要时也可使用三个参数的构造器，由第三个参数传入 `Subject` 类型的对象，并在这里通过 `LoginModule` 对象的 `initialize()` 方法传入登录模块。如果在创建 `LoginContext` 对象时没有指定 `Subject` 对象，则执行 `LoginContext` 对象的 `login()` 方法时会自行创建一个。

`CallbackHandler` 对象也是由 `LoginContext` 对象的构造器传入。其他两个参数：`shareState` 主要用于不同登录模块之间共享信息，`options` 用于读取登录配置文件中为该模块设置的选项，这两个参数在本节中未使用到，将在后面小节中介绍。

- 实现 `login()` 方法

`login()` 方法是验证过程的第一步，它完成登录和验证的主要过程，使用回调处理器要求用户输入相关信息，然后按照一定方法验证用户输入的信息。不同的验证方法编程步骤也不同，本实例不妨使用最常用的账号、口令验证，由用户输入账号和口令，然后和正确的帐号和口令进行比较，若一致则验证通过。其编程步骤如下：

- (1) 检查回调处理器

```
if (callbackHandler == null){
    throw new LoginException("Error: no CallbackHandler available");
}
```

```
}
```

分析：检查 LoginContext 对象是否已经传入回调处理器。

(2) 创建充当登录模块和回调处理器之间桥梁的 Callback 对象

```
Callback[] callbacks = new Callback[2];  
callbacks[0] = new NameCallback("user name: ");  
callbacks[1] = new PasswordCallback("password: ", false);
```

分析：Callback 对象是登录模块和回调处理器之间桥梁，登录模块需要用户输入哪些数据通过不同的 Callback 对象传递给回调处理器。如需要用户输入用户名（账号）时，可传递 NameCallback 对象给回调处理器，当需要用户输入口令时，可传递 PasswordCallback 对象给回调处理器。这些 Callback 对象的构造器参数将用于用户提示。由于登录时一般需要用户输入多个信息，因此定义了 Callback 类型的数组保存这些 Callback 对象，一起传递给回调处理器。

Callback 是一个接口，除了 NameCallback 和 PasswordCallback 实现了该接口外，J2SDK 还提供了以下类实现了 Callback 接口，用于登录模块所需要的不同信息。

- ChoiceCallback, 让回调处理器提供用户一系列选项，让用户从中选择一个。其构造器中传入四个参数，第一个是字符串，要求回调处理器显示相应的提示信息，第二个是字符串数组，提供一系列选项，要求回调处理器显示这些选项，第三个是整数，表明默认选中第几个选项，第四个是 boolean 类型的 true 或 false，若为 true 则表明要求回调处理器允许多选。
- ConfirmationCallback 让回调处理器提示“YES/NO”，“OK/CANCEL”，“YES/NO/CANCEL”等信息，让用户进行确认。其具体用法见 9.2.3 小节。
- LanguageCallback, 让用户输入或从用户环境中获取语言等相关的信息以实现程序的本地化
- TextInputCallback, 让回调处理器提示用户输入文本。其构造器可使用两个字符串参数，第一个参数指定回调处理器在用户界面提示何信息，第二个参数设置默认的文本。
- TextOutputCallback, 让回调处理器显示相关的信息或警告、错误提示信息。其具体用法见 9.2.3 小节。

(3) 执行回调处理器

```
callbackHandler.handle(callbacks);
```

分析：将上一步的 Callback 类型的数组传递给回调处理器的 handle() 方法，回调处理器中会逐一检查 Callback 类型的数组中各个元素是 NameCallback 类型、还是 PasswordCallback 或其他类型，从而提供不同的界面让用户输入信息。用户输入的内容也将保存相应的 NameCallback 或 PasswordCallback 对象中。

(4) 从回调处理器返回结果中读取用户输入的信息

```
username = ((NameCallback)callbacks[0]).getName();  
char[] tmpPassword =  
    ((PasswordCallback)callbacks[1]).getPassword();
```

分析：在第 2 步中，Callback 类型的数组第一个元素为 NameCallback 类型，第二个元素 PasswordCallback 类型，故将第一个元素强制转换为 NameCallback 类型，并执行 getName() 方法获得用户输入的账号值，将第二个元素强制转换为 PasswordCallback 类型，并执行其 getPassword() 方法获得用户输入的口令值。

如果是 ChoiceCallback 类型的对象，可通过 ChoiceCallback 对象的

getSelectedIndexes() 方法得到字符串数组, 其中包含的是用户选择的是哪个或哪几个选项。

如果是 LanguageCallback 类型的对象, 可通过 LanguageCallback 对象的 getLocale() 方法得到 Local 类型的对象, 其中包含的是用户环境使用的是何种语言。

如果是 TextInputCallback 类型的对象, 可通过 TextInputCallback 对象的 getText() 方法从中提取用户输入了什么文本。

ConfirmationCallback 和 TextOutputCallback 类型的用法见 9.2.3 小节。

(5) 验证用户输入的信息

```
if (username.equals("testUser") && password.equals("testPassword")){
    succeeded=true;
    return true;
} else {
    succeeded = false;
    throw new FailedLoginException("Error name//password pair");
}
```

分析: 这里简单地比较一下上一步得到的账号是否是“testUser”、口令是否是“testPassword”。若是, 则返回 true, 验证通过, 否则抛出 javax.security.auth.login 包中的 FailedLoginException 异常对象。

实际使用中可以将账号和口令与数据库中的账号和口令进行比较, 也可将使用其它方式进行验证。

● 实现 commit() 方法

commit() 方法主要完成登录模块 login() 方法验证结束以后的第二阶段的验证, 当所有登录模块的 login() 方法执行成功后, 将分别执行每个登录模块的 commit() 方法。commit() 方法的操作主要将包括: 如果 login() 方法验证通过, 则使用相关信息 (如验证时的用户名) 创建用户身份标志, 添加到主体中, 随即清除不必要账号、口令等信息。只有当 login() 方法和 commit() 方法都成功完成, 整个验证过程才告成功。

其主要代码如下:

(1) 创建通过验证的用户名对应的身份标志

```
MyPrincipal userPrincipal = new MyPrincipal(username);
```

分析: 它使用本小节开头定义的实现 Principal 接口的类 MyPrincipal, 传入用户输入的、已通过验证的用户名。

(2) 检查登录模块主体中是否已经包含了该用户的身份标志

```
if( subject.getPrincipals().contains(userPrincipal)){
```

分析: 它使用 initialize() 方法中传入的 Subject 对象, 执行 getPrincipals() 方法获取主体当前已有的身份标志, 进而执行其 contains() 方法检查上一步创建的身份标志是否已经包含在其中。

(3) 若未包含则将该用户的身份标志添加入主体

```
subject.getPrincipals().add(userPrincipal);
```

分析: 执行 Subject 对象的 getPrincipals() 方法获取主体当前已有的身份标志, 进而执行其 add() 方法将上一步创建的身份标志添加进去。

- 实现 abort()方法

当整体验证没有通过，如登录模块的 login()方法失败，或登录模块的第一阶段 login()方法验证通过，但第二阶段 commit()方法验证产生异常，则调用 abort()方法退出验证过程，并清除状态信息（用户账号、口令、身份标志、主体中添加的身份标志信息等）

- 实现 logout()方法

退出登录，删除主体中的身份标志。

★代码与分析:

本实例应用程序仍旧使用 9.1.1 小节的 Java 程序 SimpLogin 和 9.1.3 小节的 SimpLoginTXT，所使用的登录配置文件 MyLM.config 如下：

```
simp {  
    MyLoginModule required;  
};
```

该登录配置文件使用了如下自己编写的登录模块代码 MyLoginModule:

```
import java.util.*;  
import java.io.IOException;  
import javax.security.auth.*;  
import javax.security.auth.callback.*;  
import javax.security.auth.login.*;  
import javax.security.auth.spi.*;  
  
public class MyLoginModule implements LoginModule {  
    private Subject subject;  
    private CallbackHandler callbackHandler;  
    private boolean succeeded = false;  
    private String username;  
    private String password;  
    private MyPrincipal userPrincipal;  
  
    public void initialize(Subject subject, CallbackHandler callbackHandler,  
        Map sharedState, Map options) {  
        this.subject = subject;  
        this.callbackHandler = callbackHandler;  
    }  
  
    public boolean login() throws LoginException {  
        if (callbackHandler == null){  
            throw new LoginException("Error: no CallbackHandler available");  
        }  
        Callback[] callbacks = new Callback[2];  
        callbacks[0] = new NameCallback("user name: ");  
        callbacks[1] = new PasswordCallback("password: ", false);
```

```

try {
    callbackHandler.handle(callbacks);
    username = ((NameCallback)callbacks[0]).getName();
    char[] tmpPassword =
        ((PasswordCallback)callbacks[1]).getPassword();
    password=new String(tmpPassword);
} catch (java.io.IOException ioe) {
    throw new LoginException(ioe.toString());
} catch (UnsupportedCallbackException uce) {
    throw new LoginException(uce.toString( ));
}

if (username.equals("testUser") && password.equals("testPassword")){
    succeeded=true;
    return true;
} else {
    succeeded = false;
    throw new FailedLoginException("Error name//password pair");
}
}

public boolean commit( ) throws LoginException {
    if (succeeded == false) {
        return false;
    }
    else {
        userPrincipal = new MyPrincipal(username);
        if (!subject.getPrincipals().contains(userPrincipal)){
            subject.getPrincipals().add(userPrincipal);
        }
        username=null;
        password=null;
        return true;
    }
}

public boolean abort( ) throws LoginException {
    if (succeeded == false) {
        username = null;
        userPrincipal = null;
        return false;
    } else {
        logout();
    }
    return true;
}

```

```

    }

    public boolean logout() throws LoginException {
        subject.getPrincipals().remove(userPrincipal);
        username=null;
        password=null;
        userPrincipal=null;
        succeeded = false;
        return true;
    }
}

```

该登录模块使用了如下实现Principal接口的代码MyPrincipal:

```

import java.io.Serializable;
import java.security.Principal;
public class MyPrincipal implements Principal, Serializable {
    private String name;
    public MyPrincipal(String n) {
        name = n;
    }
    public String getName( ) {
        return name;
    }
    public int hashCode( ) {
        return name.hashCode();
    }
    public String toString( ) {
        return getName();
    }
    public boolean equals(Object obj) {
        if ( (obj!=null)&& (obj instanceof MyPrincipal)) {
            MyPrincipal obj2 = (MyPrincipal)obj;
            if (name.equals(obj2.getName())) {
                return true;
            }
        }
        return false;
    }
}

```

★运行程序

程序运行在 C:\java\ch9\MyLoginModule 目录，其中拷贝了 9.1.1 小节编译得到的 SimpLogin.class 和 9.1.3 小节得到的 SimpLoginTXT.class 文件，同时存放本节编写的 MyPrincipal.java、MyLoginModule.java 和 MyLM.config 文件。

在 9.1.1 小节曾输入

```
java -Djava.security.auth.login.config==simp.config SimpLogin
```

运行SimpLogin程序，当时出现的是图 9-1的登录窗口，提示用户输入别名、密码等信息，现在不改变SimpLogin代码，输入

```
java -Djava.security.auth.login.config==MyLM.config SimpLogin
```

运行程序，则弹出图 9-3所示窗口：



图 9-3 自己编写的登录模块

该窗口是 MyLoginModule 中执行回调处理器的 handler()方法：

```
callbackHandler.handle(callbacks)
```

由回调处理器给出的，回调处理器根据 handle()方法中的参数，逐一查看参数中 Callback 数组的各个元素，发现第一个元素是 NameCallback 类型，构造器传入的参数是“user name”，因此给出一个输入框，提示输入用户名，这里不妨输入“testUser”；第二个元素经回调处理器检查发现是 PasswordCallback 类型，构造器传入的参数是“password”，因此给出一个输入框，提示输入密码，并将密码的回显设置为“*”。这里不妨输入“testPassword”作为密码。单击“确定”按钮后，DOS 窗口显示：

```
Authentication succeeded!
```

```
[testUser]
```

如果密码或用户输入有误，则显示：

```
Authentication failed:
```

```
Error name//password pair
```

```
Sorry
```

从这里可以看出，和 9.1.2 小节一样，修改登录配置文件即可使用自己编写的登录模块。

同样，对于 9.1.3 小节的使用文本登录界面的 SimpLoginTXT，也可以类似地使用本小节的登录模块，输入：

```
java -Djava.security.auth.login.config==MyLM.config SimpLoginTXT
```

运行程序，其交互过程如下：

```
C:\java\ch9\MyLoginModule>java -Djava.security.auth.login.config==MyLM.config  
SimpLoginTXT
```

```
user name: testUser
```

```
password: testPassword
```

```
Authentication succeeded!
```

```
[testUser]
```

如果账号或者密码不正确，则出现如下提示：

```
Authentication failed:
```

```
Error name//password pair
Sorry
```

9.2.2 完整的登录模块模板

★ 实例说明

本实例分析了 SUN 提供的完整的登录模块模板，演示了其使用方法。

★ 编程思路：

9.2.1 小节中简单的登录模块主要为了说明登录模块的机理和主要编程步骤，因而为了程序的简洁作了很多简化，本实例对 SUN 文档提供的完整的登录模块模板做了分析，分析了编程的思想。

(1) 使用包

该模板中，登录模块名称为 `SampleLoginModule`，使用的实现 `Principal` 接口的类是 `SamplePrincipal`。这两个类都定义在包 `sample.module` 中，便于打包成 `jar` 文件给程序使用者使用。在执行程序时，可通过 `Java` 命令选项 `-classpath` 指定该包。

(2) 使用 `debug` 等登录模块选项

模板中有一个 `boolean` 类型的 `debug` 变量，通过

```
debug = "true".equalsIgnoreCase((String)options.get("debug"));
```

语句查看登录配置文件中有没有指定 `debug` 选项，若指定了则 `debug` 变量赋值为 `true`。在登录模块的各个方法中检查该变量，若为 `true` 则显示各种调试信息。

这里，`options` 变量是在登录模块初始化执行 `public void initialize(Subject subject, CallbackHandler callbackHandler, Map sharedState, Map options)` 方法时从最后一个参数自动传入的。如果登录模块还有其他信息需要通过登录配置文件灵活地设置，同样可以使用 `options.get("选项名称")` 来读取。例如可使用如下语句：

```
String s=(String)options.get("xxx");
```

则可以通过登录配置文件中 `xxx` 选项来设置变量 `s` 的值。即登录配置文件可以这样设置：

```
Simp {
    登录模块名称 required debug=true
    xxx=...;
};
```

(3) 口令处理

在口令处理中，模板的代码考虑了字符串作为口令的不安全性，因而使用 `char` 数组保存口令，使用 `System.arraycopy()` 方法将读取的口令从临时的数组拷贝到正式保存口令的数组。在从 `Callback` 对象读取口令后，执行 `Callback` 对象的 `clearPassword()` 方法清空口令。

```
char[] tmpPassword = ((PasswordCallback)callbacks[1]).getPassword();
if (tmpPassword == null) {
    tmpPassword = new char[0];
}
password = new char[tmpPassword.length];
System.arraycopy(tmpPassword, 0,
    password, 0, tmpPassword.length);
((PasswordCallback)callbacks[1]).clearPassword();
```

在比较口令是否匹配时也全部按照字符逐个比较。在不再使用保存在 `char` 数组中的口令时及时将口令清空。清空时其中不是简单地将 `password` 赋值 `null`，而是先将 `password` 数

组每个元素填上空格字符，这样可以防止攻击者从内存中读取到口令的值。相关代码如下：

```
for (int i = 0; i < password.length; i++){
    password[i] = ' ';
}
password = null;
```

(4) abort()方法

在 abort() 方法中，区分了验证过程第一阶段执行登录模块的 login() 方法和第二阶段执行登录模块的 commit() 方法的各种不同执行结果。为了标记这种执行结果，相应地在登录模块的 login() 方法和 commit() 方法中分别使用变量 succeeded 和 commitSucceeded 代表 login() 和 commit() 方法的执行情况。如果登录模块的 login() 方法通过验证，则 succeeded 变量为 true。若 commit() 方法在向主体添加身份标志等操作中没有抛出异常，则 commitSucceeded 变量为 true。

abort() 方法中，如果 succeeded 变量为 false，则说明登录模块 login() 方法本身就没有通过验证，不需清空 login() 过程中输入的信息。此时验证的第二步是执行 abort() 而不是 commit() 方法，因此不会向主体添加身份标志，不需要清空身份标志信息。abort() 方法直接返回 false。相关代码如下：

```
if (succeeded == false) {
    return false;
}
```

abort() 方法中，如果 succeeded 变量为 true，即登录模块 login() 方法已通过验证，此时，验证的第二步是执行 commit() 方法，若 commit() 方法运行时产生异常（commitSucceeded 变量为 false），则整个验证过程仍旧失败，需要将第一步验证（登录模块 login() 方法）中输入的用户、口令信息清空，由于 commit() 方法发生异常之前可能已经创建了代表用户身份标志的 Principal 对象，因此应该将其也清空。相关代码如下：

```
else if (succeeded == true && commitSucceeded == false) {
    // login succeeded but overall authentication failed
    succeeded = false;
    username = null;
    if (password != null) {
        for (int i = 0; i < password.length; i++)
            password[i] = ' ';
        password = null;
    }
    userPrincipal = null;
}
```

abort() 方法中，如果 succeeded 变量和 commitSucceeded 变量都为 true，说明两步的验证都已通过，因此既需要清除账号、口令信息，也需要清除 Principal 对象和主体中已经添加的 Principal 对象。即执行登录模块的 logout 方法即可。

```
else {
    // overall authentication succeeded and commit succeeded,
    // but someone else's commit failed
    logout();
}
```

```
return true;
```

“代码与分析”部分给出了全部代码，该模板可以直接使用，也可以根据需要进行修改，例如如果需要用户输入的不只是账号和口令，则可以继续使用 `NameCallback`、`PasswordCallback`、`ChoiceCallback`、`ConfirmationCallback`、`LanguageCallback`、`TextInputCallback` 或 `TextOutputCallback` 类通过回调处理器用户进行交互，也可以自己定义类实现 `Callback` 接口通过回调处理器用户进行交互。

该模板简单地检测账号是否为“testUser”，密码是否为“testPassword”来验证用户输入的信息，也可以修改该验证方法，通过 `JDBC`、`JNDI`、`SSL` 或其他方式从后台的各种服务器查询账号、口令或其他信息进行各种验证。验证通过后，根据用户输入的账户或其他信息，可以从后台服务器得到用户更多的信息，这样一次验证可以创建多个相同或不同的 `Principal` 对象添加到主体中。如 9.1.2 小节使用的 JAAS 中 `UnixLoginModule` 登录模块中一次就添加了三个不同的 `Principal` 对象：`UnixPrincipal`、`UnixNumericUserPrincipal`、`UnixNumericGroupPrincipal`。用户以 `xyx` 身份登录 Unix 操作系统运行程序时，屏幕输出了这三个身份标志的值。

★代码与分析:

本实例应用程序仍旧使用 9.1.1 小节的 Java 程序 `SimpLogin` 和 9.1.3 小节的 `SimpLoginTXT`，所使用的登录配置文件 `sampleLM.config` 内容如下：

```
/** Login Configuration for the JAAS Sample Application */
Simp {
    sample.module.SampleLoginModule required debug=true;
};
```

该登录配置文件中增加了一个选项：`debug=true`。

该登录配置文件所使用的登录模块的代码如下：

```
package sample.module;

import java.util.*;
import java.io.IOException;
import javax.security.auth.*;
import javax.security.auth.callback.*;
import javax.security.auth.login.*;
import javax.security.auth.spi.*;
import sample.principal.SamplePrincipal;

public class SampleLoginModule implements LoginModule {

    // initial state
    private Subject subject;
    private CallbackHandler callbackHandler;
    private Map sharedState;
    private Map options;
```



```

// configurable option
private boolean debug = false;

// the authentication status
private boolean succeeded = false;
private boolean commitSucceeded = false;

// username and password
private String username;
private char[] password;

// testUser's SamplePrincipal
private SamplePrincipal userPrincipal;

public void initialize(Subject subject, CallbackHandler callbackHandler,
    Map sharedState, Map options) {

    this.subject = subject;
    this.callbackHandler = callbackHandler;
    this.sharedState = sharedState;
    this.options = options;

    // initialize any configured options
    debug = "true".equalsIgnoreCase((String)options.get("debug"));
}

public boolean login() throws LoginException {

    // prompt for a user name and password
    if (callbackHandler == null)
        throw new LoginException("Error: no CallbackHandler available " +
            "to garner authentication information from the user");

    Callback[] callbacks = new Callback[2];
    callbacks[0] = new NameCallback("user name: ");
    callbacks[1] = new PasswordCallback("password: ", false);

    try {
        callbackHandler.handle(callbacks);
        username = ((NameCallback)callbacks[0]).getName();
        char[] tmpPassword = ((PasswordCallback)callbacks[1]).getPassword();
        if (tmpPassword == null) {
            // treat a NULL password as an empty password
            tmpPassword = new char[0];
        }
    }
}

```

```

    }
    password = new char[tmpPassword.length];
    System.arraycopy(tmpPassword, 0,
        password, 0, tmpPassword.length);
    ((PasswordCallback)callbacks[1]).clearPassword();

} catch (java.io.IOException ioe) {
    throw new LoginException(ioe.toString());
} catch (UnsupportedCallbackException uce) {
    throw new LoginException("Error: " + uce.getCallback().toString() +
        " not available to garner authentication information " +
        "from the user");
}

// print debugging information
if (debug) {
    System.out.println("\t\t[SampleLoginModule] " +
        "user entered user name: " +
        username);
    System.out.print("\t\t[SampleLoginModule] " +
        "user entered password: ");
    for (int i = 0; i < password.length; i++)
        System.out.print(password[i]);
    System.out.println();
}

// verify the username/password
boolean usernameCorrect = false;
boolean passwordCorrect = false;
if (username.equals("testUser"))
    usernameCorrect = true;
if (usernameCorrect &&
    password.length == 12 &&
    password[0] == 't' &&
    password[1] == 'e' &&
    password[2] == 's' &&
    password[3] == 't' &&
    password[4] == 'P' &&
    password[5] == 'a' &&
    password[6] == 's' &&
    password[7] == 's' &&
    password[8] == 'w' &&
    password[9] == 'o' &&
    password[10] == 'r' &&

```

```

password[11] == 'd') {

    // authentication succeeded!!!
    passwordCorrect = true;
    if (debug)
        System.out.println("\t\t[SampleLoginModule] " +
            "authentication succeeded");
    succeeded = true;
    return true;
} else {
    // authentication failed -- clean out state
    if (debug)
        System.out.println("\t\t[SampleLoginModule] " +
            "authentication failed");
    succeeded = false;
    username = null;
    for (int i = 0; i < password.length; i++)
        password[i] = ' ';
    password = null;
    if (!usernameCorrect) {
        throw new FailedLoginException("User Name Incorrect");
    } else {
        throw new FailedLoginException("Password Incorrect");
    }
}
}

public boolean commit() throws LoginException {
    if (succeeded == false) {
        return false;
    } else {
        // add a Principal (authenticated identity)
        // to the Subject

        // assume the user we authenticated is the SamplePrincipal
        userPrincipal = new SamplePrincipal(username);
        if (!subject.getPrincipals().contains(userPrincipal))
            subject.getPrincipals().add(userPrincipal);

        if (debug) {
            System.out.println("\t\t[SampleLoginModule] " +
                "added SamplePrincipal to Subject");
        }
    }
}

```

```

        // in any case, clean out state
        username = null;
        for (int i = 0; i < password.length; i++)
            password[i] = ' ';
        password = null;

        commitSucceeded = true;
        return true;
    }
}

public boolean abort() throws LoginException {
    if (succeeded == false) {
        return false;
    } else if (succeeded == true && commitSucceeded == false) {
        // login succeeded but overall authentication failed
        succeeded = false;
        username = null;
        if (password != null) {
            for (int i = 0; i < password.length; i++)
                password[i] = ' ';
            password = null;
        }
        userPrincipal = null;
    } else {
        // overall authentication succeeded and commit succeeded,
        // but someone else's commit failed
        logout();
    }
    return true;
}

public boolean logout() throws LoginException {

    subject.getPrincipals().remove(userPrincipal);
    succeeded = false;
    succeeded = commitSucceeded;
    username = null;
    if (password != null) {
        for (int i = 0; i < password.length; i++)
            password[i] = ' ';
        password = null;
    }
    userPrincipal = null;
}

```

```

        return true;
    }
}

```

★运行程序

程序运行在 C:\java\ch9\SampleLoginModule 目录，其中拷贝了 9.1.1 小节编译得到的 SimpLogin.class 和 9.1.3 小节得到的 SimpLoginTXT.class 文件，同时存放本节的 SamplePrincipal.java、SampleLoginModule.java 和 sampleLM.config.config 文件。

输入

```
javac -d . SamplePrincipal.java
```

```
javac -d . SampleLoginModule.java
```

编译程序，将根据两个文件中包的名字在当前目录分别创建 sample\principal 和 sample\module 目录存放两个字节码文件。执行如下命令将 sample 目录中的文件打包成 sampleLM.jar 文件：

```
C:\java\ch9\SampleLoginModule>jar cvf sampleLM.jar sample
```

标明清单(manifest)

增加: sample/(读入= 0) (写出= 0) (存储了 0%)

增加: sample/principal/(读入= 0) (写出= 0) (存储了 0%)

增加: sample/principal/SamplePrincipal.class (读入= 1003) (写出= 603) (压缩了 39%)

增加: sample/module/(读入= 0) (写出= 0) (存储了 0%)

增加: sample/module/SampleLoginModule.class (读入= 4375) (写出= 2214) (压缩了 49%)

程序运行过程如下：

```
C:\java\ch9\SampleLoginModule>java -classpath sampleLM.jar;.
```

```
-Djava.security.auth.login.config==sampleLM.config SimpLoginTXT
```

```
user name: testUser
```

```
password: testPassword
```

```
[SampleLoginModule] user entered user name: testUser
```

```
[SampleLoginModule] user entered password: testPassword
```

```
[SampleLoginModule] authentication succeeded
```

```
[SampleLoginModule] added SamplePrincipal to Subject
```

```
Authentication succeeded!
```

```
[SamplePrincipal: testUser]
```

其中四行[SampleLoginModule]是调试信息，由于在登录配置文件 sampleLM.config 中增加了一条 debug 选项：debug=true，因而 SampleLoginModule.java 中

```
debug = "true".equalsIgnoreCase((String)options.get("debug"));
```

一条语句将给 debug 变量赋值 true，因而 SampleLoginModule.java 程序 “if(debug)” 中的语句将被执行到，打印出调试信息。

如果将登录配置文件 sampleLM.config 中 debug=true 一项去掉，则将不显示调试信息。

如果运行时输入的账号或口令不正确，将提示验证失败，如：

```
C:\java\ch9\SampleLoginModule>java -classpath sampleLM.jar;.
```

```
-Djava.security.auth.login.config==sampleLM.config SimpLoginTXT
```

```

user name: testUser
password: 123456
[SampleLoginModule] user entered user name: testUser
[SampleLoginModule] user entered password: 123456
[SampleLoginModule] authentication failed
Authentication failed:
    Password Incorrect
Sorry

```

如果输入

java -classpath sampleLM.jar;. -Djava.security.auth.login.config==sampleLM.config SimpLogin
运行程序，将使用图形界面登录。

9.2.3 使用模板编写自己的密钥库登录模块

★ 实例说明

在 9.1.1 小节已经使用过 JAAS 提供的 KeyStoreLoginModule 实现基于密钥库的登录验证，本实例使用 9.2.2 小节的模板编写了一个自己的密钥库登录模块。

★ 编程思路：

9.2.2 小节的模板中只读取了账号和口令信息进行验证，而对于密钥库，需要知道密钥库的名称和位置、条目的别名、密钥库的保护口令和条目的保护口令，因此，为了将 9.2.2 小节的模板改成基于密钥库的登录验证，首先需要修改 login() 方法中从用户处读取信息的部分。在验证用户输入的信息时，将不再是将用户输入的账号/口令和数据库中的账号和口令进行比较，而是判断通过用户输入的别名、密钥库的保护口令和条目的保护口令能否读取指定的密钥库。因此，9.2.1 小节中介绍的 login() 方法的编程步骤中，第 2 步创建充当登录模块和回调处理器之间桥梁的 Callback 对象、第 4 步从回调处理器返回结果中读取用户输入的信息、第 5 步验证用户输入的信息将需要修改。

- (1) 创建充当登录模块和回调处理器之间桥梁的 Callback 对象

```

String keyStoreURL= (String)options.get("keyStoreURL");
Callback[] callbacks = new Callback[5];
TextOutputCallback txtCallback = new TextOutputCallback(
    TextOutputCallback.INFORMATION,
    "请登录密钥库");
ConfirmationCallback confirmCallback = new ConfirmationCallback(
    ConfirmationCallback.INFORMATION,
    ConfirmationCallback.OK_CANCEL_OPTION,
    ConfirmationCallback.OK);
callbacks[0]= txtCallback;
callbacks[1] = new NameCallback("密钥库别名");
callbacks[2] = new PasswordCallback("密钥库保护口令 ", false);
callbacks[3] = new PasswordCallback("私钥保护口令 ", false);
callbacks[4] = confirmCallback;

```

分析：这里和用户交互时要读取和显示的信息比在模板中的多，因此定义了 5 个元素的 Callback 对象和用户交互。在不妨保留 9.2.2 小节 SUN 提供的模板中读取 username 和 password 的部分，将其当作条目名称和保护密钥库的口令，当然其提示内容要修改为“密钥库别名”和“密钥库保护口令”。此外增加了一个 PasswordCallback 对象让用户输入私钥保护口令。对于密钥库文件的名字和位置，不妨通过登录配置文件的 keyStoreURL 选项来指定。

本实例还使用了两个新的 Callback: TextOutputCallback 和 ConfirmationCallback。

TextOutputCallback 对象在登录窗口中显示一行参数中指定的字符。其构造器有两个参数，第一个参数指定显示的信息类型，如使用 TextOutputCallback.INFORMATION 表明显示的是一般提示信息，使用 TextOutputCallback.WARNING 表明显示的是警告信息，使用 TextOutputCallback.ERROR 表明显示的是出错信息。第二个参数是需要在用户登录窗口显示的字符串，这里显示“请登录密钥库”。

ConfirmationCallback 对象在登录窗口中显示几个选择按钮，其构造器有三个参数，第一个参数和 TextOutputCallback 类似，指定显示的信息类型，如使用 ConfirmationCallback.INFORMATION 表明显示的是一般提示信息，使用 ConfirmationCallback.WARNING 表明显示的是警告信息，使用 ConfirmationCallback.ERROR 表明显示的是出错信息。第二个参数指定选择按钮的选项类型，可以是 ConfirmationCallback.YES_NO_OPTION、ConfirmationCallback.OK_CANCEL_OPTION、ConfirmationCallback.YES_NO_CANCEL_OPTION 等，分别显示不同的“确定”、“撤销”、和“是”、“否”、“撤销”按钮组合，也可使用字符串数组，数组中每个字符串将显示一个按钮，并在按钮上显示该字符串。第三个参数指定默认的选项，可以是 ConfirmationCallback.OK、ConfirmationCallback.CANCEL、ConfirmationCallback.YES、ConfirmationCallback.NO 等。

(2) 从回调处理器返回结果中读取用户输入的信息

```
tmpPassword = ((PasswordCallback)callbacks[3]).getPassword();
if (tmpPassword == null) {
    tmpPassword = new char[0];
}
pkpassword = new char[tmpPassword.length];
System.arraycopy(tmpPassword, 0,
    pkpassword, 0, tmpPassword.length);
((PasswordCallback)callbacks[3]).clearPassword();
int confirmationResult = confirmCallback.getSelectedIndex();
```

分析：和 9.2.2 小节第三步一样，从 Callback 对象读取用户输入的各种信息，对于口令将其保存在 char 数组中。对于 ConfirmationCallback 对象，执行其 getSelectedIndex() 可以获得用户选择的是哪个按钮。

(3) 验证用户输入信息的部分

```
if (confirmationResult == ConfirmationCallback.CANCEL) {
    throw new LoginException("Login cancelled");
}
```

```

ks = KeyStore.getInstance("JKS");
InputStream in = new URL(keyStoreURL).openStream();
ks.load(in, password);
in.close();
Key privateKey = ks.getKey(username, pkpassword);
if (privateKey == null
    || !(privateKey instanceof PrivateKey)){
    throw new FailedLoginException(
        "Unable to recover key from keystore");
}

```

分析：首先判断用户是否选择了“取消”按钮，若是，则登录失败。否则先读取登录配置文件中“keyStoreURL”选项的值，和 9.2.2 小节第三步一样，从 Callback 对象读取用户输入的各种信息，对于口令将其保存在 char 数组中。对于 ConfirmationCallback 对象，执行其 getSelectedIndex() 可以获得用户选择的是哪个按钮。

若用户点击的是“确定”按钮，则和第 5 章类似，创建 KeyStore 对象，然后将用户输入的密钥库口令传入其 load() 方法加载密钥库，再将用户输入的别名和保护私钥的口令传入其 getKey() 方法读取私钥。如果这些操作都可正确进行，则表明用户输入的信息是正确的，用户可通过该步骤的验证。

9.2.2 小节的模板中使用自己定义的 SamplePrincipal 类向主体中添加身份标志信息，其实 javax.security.auth.x500 包中的 X500Principal 类也可以充当此功能，只是 X500Principal 类的构造器中传入的参数必须 X.500 格式的字符串。由于密钥库中的证书支持 X.500 格式的名字，可用于创建 X500Principal 类型的对象，因此这里直接从密钥库中读取证书，然后获取证书的 X.500 格式的名字，作为参数传递给 X500Principal 类型的对象，最后将其添加入主体。

```

cchain = ks.getCertificateChain(username);
X509Certificate certificate = (X509Certificate)cchain[0];
userPrincipal = new javax.security.auth.x500.X500Principal
    (certificate.getSubjectDN().getName());
subject.getPrincipals().add(userPrincipal);

```

★代码与分析：

本实例应用程序仍旧使用 9.1.1 小节的 Java 程序 SimpLogin 和 9.1.3 小节的 SimpLoginTXT，所使用的登录配置文件 MyKS.config 内容如下：

```

simp {
    KSLoginModule sufficient
    keyStoreURL = "file:C:/java/ch9/MyKSLoginModule/mykeystore";
};

```

该登录配置文件所使用的登录模块的代码如下：


```

import javax.security.auth.x500.X500Principal;
import java.io.*;
import java.net.*;
import java.security.*;
import java.security.cert.*;
import java.util.*;
import java.security.cert.Certificate;
import javax.security.auth.*;
import javax.security.auth.callback.*;
import javax.security.auth.login.*;
import javax.security.auth.spi.LoginModule;

public class KSLoginModule implements LoginModule {
    // initial state
    private Subject subject;
    private CallbackHandler callbackHandler;
    private Map sharedState;
    private Map options;

    // configurable option
    private boolean debug = false;

    // the authentication status
    private boolean succeeded = false;
    private boolean commitSucceeded = false;

    // username and password
    private String username;
    private char[] password;
    private char[] pkpassword;
    private Certificate[] cchain;
    private javax.security.auth.x500.X500Principal userPrincipal;
    private KeyStore ks;

    public void initialize(Subject subject, CallbackHandler callbackHandler,
        Map sharedState, Map options) {

        this.subject = subject;
        this.callbackHandler = callbackHandler;
        this.sharedState = sharedState;
        this.options = options;

        // initialize any configured options
        debug = "true".equalsIgnoreCase((String)options.get("debug"));

```

```

}

public boolean login() throws LoginException {

    // prompt for a user name and password
    if (callbackHandler == null)
        throw new LoginException("Error: no CallbackHandler available " +
            "to garner authentication information from the user");
    String keyStoreURL= (String)options.get("keyStoreURL");
    Callback[] callbacks = new Callback[5];
    TextOutputCallback txtCallback =
        new TextOutputCallback(
            TextOutputCallback.INFORMATION,
            "请登录密钥库");
    ConfirmationCallback confirmCallback =
        new ConfirmationCallback(
            ConfirmationCallback.INFORMATION,
            ConfirmationCallback.OK_CANCEL_OPTION,
            ConfirmationCallback.OK);

    callbacks[0]= txtCallback;
    callbacks[1] = new NameCallback("密钥库别名");
    callbacks[2] = new PasswordCallback("密钥库保护口令 ", false);
    callbacks[3] = new PasswordCallback("私钥保护口令 ", false);
    callbacks[4] = confirmCallback;

    try {

        callbackHandler.handle(callbacks);
        username = ((NameCallback)callbacks[1]).getName();
        char[] tmpPassword = ((PasswordCallback)callbacks[2]).getPassword();
        if (tmpPassword == null) {
            // treat a NULL password as an empty password
            tmpPassword = new char[0];
        }
        password = new char[tmpPassword.length];
        System.arraycopy(tmpPassword, 0,
            password, 0, tmpPassword.length);
        ((PasswordCallback)callbacks[2]).clearPassword();

        tmpPassword = ((PasswordCallback)callbacks[3]).getPassword();
        if (tmpPassword == null) {

```

```

        tmpPassword = new char[0];
    }
    pkpassword = new char[tmpPassword.length];
    System.arraycopy(tmpPassword, 0,
        pkpassword, 0, tmpPassword.length);
    ((PasswordCallback)callbacks[3]).clearPassword();

} catch (java.io.IOException ioe) {
    throw new LoginException(ioe.toString());
} catch (UnsupportedCallbackException uce) {
    throw new LoginException("Error: " + uce.getCallback().toString() +
        " not available to garner authentication information " +
        "from the user");
}

// print debugging information
if (debug) {
    System.out.println("\t\t[SampleLoginModule] " +
        "user entered user name: " +
        username);
    System.out.print("\t\t[SampleLoginModule] " +
        "user entered password: ");
    for (int i = 0; i < password.length; i++)
        System.out.print(password[i]);
    System.out.println();
}

int confirmationResult = confirmCallback.getSelectedIndex();

if (confirmationResult == ConfirmationCallback.CANCEL) {
    throw new LoginException("Login cancelled");
}

try {
    ks = KeyStore.getInstance("JKS");
    InputStream in = new URL(keyStoreURL).openStream();
    ks.load(in, password);
    in.close();
    Key privateKey =
        ks.getKey(username, pkpassword);
    if (privateKey == null
        || !(privateKey instanceof PrivateKey)) {
        throw new FailedLoginException(
            "Unable to recover key from keystore");
    }
}

```

```

        succeeded = true;
        return true;
    }
    catch (Exception e) {
        succeeded = false;
        throw new LoginException(
            "Error in login to Keystore " + e);
    }
}

public boolean commit() throws LoginException {
    if (succeeded == false) {
        return false;
    } else {
        // add a Principal (authenticated identity)
        // to the Subject
        try{
            cchain =
                ks.getCertificateChain(username);
            X509Certificate certificate = (X509Certificate)cchain[0];
            userPrincipal = new javax.security.auth.x500.X500Principal
                (certificate.getSubjectDN().getName());
        }catch(Exception e){
            throw new LoginException(e.toString());
        }
        subject.getPrincipals().add(userPrincipal);

        if (debug) {
            System.out.println("\t\t[SampleLoginModule] " +
                "added SamplePrincipal to Subject");
        }

        // in any case, clean out state
        username = null;
        for (int i = 0; i < password.length; i++)
            password[i] = ' ';
        password = null;

        commitSucceeded = true;
        return true;
    }
}

public boolean abort() throws LoginException {

```

```

        if (succeeded == false) {
            return false;
        } else if (succeeded == true && commitSucceeded == false) {
            // login succeeded but overall authentication failed
            succeeded = false;
            username = null;
            if (password != null) {
                for (int i = 0; i < password.length; i++)
                    password[i] = ' ';
                password = null;
            }
            userPrincipal = null;
        } else {
            // overall authentication succeeded and commit succeeded,
            // but someone else's commit failed
            logout();
        }
    }
    return true;
}

public boolean logout() throws LoginException {

    subject.getPrincipals().remove(userPrincipal);
    succeeded = false;
    succeeded = commitSucceeded;
    username = null;
    if (password != null) {
        for (int i = 0; i < password.length; i++)
            password[i] = ' ';
        password = null;
    }
    userPrincipal = null;
    return true;
}
}

```

★运行程序

程序运行在 C:\java\ch9\MyKSLoginModule 目录，其中拷贝了 9.1.1 小节编译得到的 SimpLogin.class、9.1.3 小节得到的 SimpLoginTXT.class 文件和 9.1.1 小节所使用的密钥库 mykeystore，同时存放本节的 KSLoginModule 程序。

输入 `java -Djava.security.auth.login.config==MyKS.config SimpLogin` 运行程序，出现图 9-4 所示窗口，可以看出它和本实例第一步 Callback 对象的对应关系。

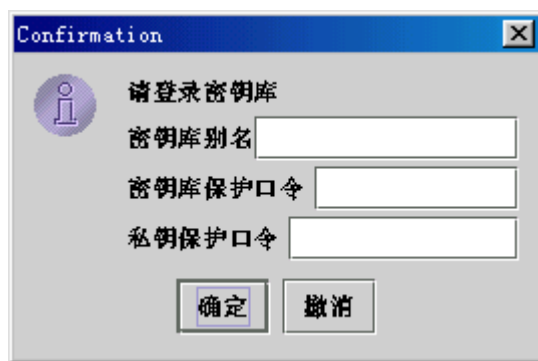


图 9-4 自己编写的密钥库登录模块

分别输入 mytest、wshr.ut、wshr.ut，可正确登录密钥库。（这里私钥保护口令不可不输入，如果在 KSLoginModule.java 中加一句判断：若私钥保护口令输入为空，则使用密钥库保护口令，则私钥保护口令就可以像 JAAS 提供的 KeyStoreLoginModule 类一样跳过私钥保护口令了。

如果输入 `java -Djava.security.auth.login.config==MyKS.config SimpLoginTXT` 运行程序，则可以以文本方式进行交互，其过程如下：

```
C:\java\ch9\MyKSLoginModule>java -Djava.security.auth.login.config==MyKS.config
SimpLoginTXT
请登录密钥库
密钥库别名 mytest
密钥库保护口令 wshr.ut
私钥保护口令 wshr.ut
0. OK [default]
1. Cancel
Enter a number: 0
Authentication succeeded!
[CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai, C=CN]
```

9.3 使用堆叠式登录

9.3.1 堆叠式登录及各个登录模块的相互关系

★ 实例说明

前面各个小节在配置文件中只使用了一个登录模块，用户登录后多只有一种身份。有时需要用户多次登录，如用户既作为 E-mail 用户、又作为 Web 用户登录，同时具有两种身份。堆叠式登录可解决这一问题，本实例在登录配置文件中使用了多个登录模块实现了堆叠式登录。

★ 编程思路：

一个登录配置文件中可以有多个条目，每个条目放在一对大括号中，一个条目中可以指定多个登录模块，实现堆叠式登录，JAAS 将按照顺序执行各个登录模块。每个条目的语法格式为：

```
<应用程序查阅该条目的名字> {  
    <登录模块名 1> <控制标记>  
        <登录模块 1 的可选项 1=可选项值>  
        <登录模块 1 的可选项 2=可选项值>  
        ..... ;  
    <登录模块名 2> <控制标记>  
        <登录模块 2 的可选项 1=可选项值>  
        <登录模块 2 的可选项 2=可选项值>  
        ..... ;  
    .....;  
};
```

登录模块的名字即 9.2.1 和 9.2.2 小节编写的类，或 JAAS 已经提供的 NTLoginModule、UnixLoginModule、KeyStoreLoginModule、JndiLoginModule 和 Krb5LoginModule 等。

在 9.1.2 和 9.1.4 小节已经使用过 KeyStoreLoginModule 登录模块的多个可选项，在 9.2.2 小节使用的 debug 也是一个可选项，登录模块中通过(String)options.get("debug")获得的登录配置文件中为 debug 可选项设定的值。其他可选项的值也可以类似地获取。

登录模块可以指定各种控制标记如 required、requisite、sufficient 或 optional 等，控制什么情况下整个验证过程算通过。这几个标记在两方面控制对应的登录模块，一是如果该模块没有通过验证，整个登录过程是否算通过；二是如果该模块如果没有通过验证，后面的其他模块是否还继续执行验证。

其中：

required 指定用户必须通过该登录模块的验证，否则总体验证算没有通过。在执行时不管用户有没有通过该模块的验证，其他模块的验证过程会继续向下进行。

Requisite 指定用户必须通过该登录模块的验证，否则总体验证算没有通过。在执行时如果用户通过该模块的验证，则继续向下执行其他登录模块，如果没有通过该模块的验证，则不再继续执行其他登录模块，验证过程马上终止，回到应用程序。

Sufficient 指定用户不一定非要通过该登录模块的验证。如果没有通过该模块的验证，则继续执行下面的登录模块，但如果该模块通过验证了，则验证过程已经“足够(sufficient)”了，下面的各个登录模块将不再执行，马上回到应用程序。因此，此时若在这之前、标记为 required 或 requisite 的所有模块都是验证通过的，则整个验证通过。

Optional 指定该模块是可选的，其登录成功与否不影响整个验证结果，不管其成功还是失败，验证过程将继续执行下面的各个模块。

按照上述规则，假如一个条目中有四个登录模块，顺序分别为登录模块 1、登录模块 2、登录模块 3、登录模块 4，则不同的控制标记组合结果如表 9-1 所示。

登录模块名称	控制标记	登录模块执行结果(pass/通过验证, fail/未通过验证)							
登录模块 1	required	pass	pass	pass	pass	fail	fail	fail	fail
登录模块 2	sufficient	pass	fail	fail	fail	pass	fail	fail	fail

登录模块 3	requisite	不再执行	pass	pass	fail	不再执行	pass	pass	fail
登录模块 4	optional	不再执行	pass	fail	不再执行	不再执行	pass	fail	不再执行
整个验证结果		pass	pass	pass	fail	fail	fail	fail	fail

表 9-1 登录模块不同控制标记的作用

★代码与分析:

本实例应用程序仍旧使用 9.1.1 小节的 Java 程序 SimpLogin 和 9.1.3 小节的 SimpLoginTXT。

本实例使用 9.2.2 小节的 sample.module.SampleLoginModule 登录模块以及 JAAS 提供的 com.sun.security.auth.module.KeyStoreLoginModule 登录模块。

此外，为 KeyStoreLoginModule 登录模块提供了两个密钥库：store1 和 store2，store1 中有条目 email 和 web，store2 中有条目 proxy。

使用以下登录配置文件查看不同的运行结果：

文件 stack1.config:

```
simp {

    sample.module.SampleLoginModule requisite;

    com.sun.security.auth.module.KeyStoreLoginModule optional
        keyStoreURL ="file:C:/java/ch9/multiLoginModule/store1"
        keyStoreAlias=Email;

    com.sun.security.auth.module.KeyStoreLoginModule sufficient
        keyStoreURL ="file:C:/java/ch9/multiLoginModule/store1"
        keyStoreAlias=web;

    com.sun.security.auth.module.KeyStoreLoginModule required
        keyStoreURL ="file:C:/java/ch9/multiLoginModule/store2"
        keyStoreAlias=proxy;
};
```

这里，将控制标记的排列和 表 9-1 不一样，以演示其他组合的结果。

★运行程序

程序运行在 C:\java\ch9\multiLoginModule 目录，其中拷贝了 9.1.1 小节编译得到的 SimpLogin.class 和 9.1.3 小节的 SimpLoginTXT.class。同时拷贝了 9.2.2 小节的 sampleLM.jar 文件，此外保存了本节的登录配置文件 stack1.config

输入如下命令输入批处理文件 1.bat，执行 1.bat 创建本小节使用的密钥库。

```
keytool -genkey -dname "CN=Email User, OU=ME, O=SouthEast University, L=GL, ST=Nanjing, C=CN" -alias email -keyalg RSA -keystore store1 -keypass kpass123 -storepass store1pass -validity 3000
```



```
keytool -genkey -dname "CN=Webmaster, OU=NC, O=Shanghai University, L=ZB, ST=Shanghai, C=CN" -alias web -keyalg RSA -keystore store1 -keypass kpass123 -storepass store1pass -validity 3000
```

```
keytool -genkey -dname "CN=Proxy User, OU=IC, O=Fudan University, L=ZB, ST=Shanghai, C=CN" -alias proxy -keyalg RSA -keystore store2 -keypass kpass123 -storepass store2pass -validity 3000
```

这些命令将在当前目录创建两个密钥库 store1 和 store2。其中，密钥库 store1 的保护口令为 store1pass，有两个条目 email 和 web，保护口令都为 kpass123。密钥库 store2 的保护口令为 store2pass，有一个条目 proxy，保护口令都为 kpass123。

输入

```
java -classpath sampleLM.jar;. -Djava.security.auth.login.config==stack1.config  
SimpLoginTXT
```

运行程序，其交互过程如下：

```
user name: testUser  
password: 123456  
Authentication failed:  
    Password Incorrect  
Sorry
```

由于在第一个登录模块进行验证时输入的密码不正确，因此 SampleLoginModule 登录模块验证没有通过。由于该模块的控制选项为 Requisite，因此验证不再继续，返回验证失败。

继续如下试验：

```
C:\java\ch9\multiLoginModule>java -classpath sampleLM.jar;. -Djava.security.auth.login.config==stack1.config SimpLoginTXT  
user name: testUser  
password: testPassword  
请登录 keystore  
Keystore 别名: [Email]  
Keystore 密码: store1pass  
私人关键密码 (可选的): kpass123  
0. OK [default]  
1. Cancel  
Enter a number: 0  
请登录 keystore  
Keystore 别名: [web]  
Keystore 密码: store1pass  
私人关键密码 (可选的): kpass123  
0. OK [default]  
1. Cancel  
Enter a number: 0  
Authentication succeeded!  
[SamplePrincipal: testUser, CN=Email User, OU=ME, O=SouthEast University,
```

L=GL, ST=Nanjing, C=CN, CN=Webmaster, OU=NC, O=Shanghai University, L=ZB, ST=Shanghai, C=CN]

这里三个登录模块都输入了正确的信息,验证通过。由于第三个登录模块的控制标记是 sufficient,因此不再进行第四个模块的验证,整个验证通过,最后得到的身份标志是

[SamplePrincipal: testUser, CN=Email User, OU=ME, O=SouthEast University, L=GL, ST=Nanjing, C=CN, CN=Webmaster, OU=NC, O=Shanghai University, L=ZB, ST=Shanghai, C=CN]

可见三个模块中的身份标记都已经加到主体中了。

继续如下试验:

```
C:\java\ch9\multiLoginModule>java -classpath sampleLM.jar;.
-Djava.security.auth.login.config==stack1.config SimpLoginTXT
user name: testUser
password: testPassword
请登录 keystore
Keystore 别名: [Email]
Keystore 密码:
私人关键密码 (可选的):
0. OK [default]
1. Cancel
Enter a number: 0
请登录 keystore
Keystore 别名: [web]
Keystore 密码: store1pass
私人关键密码 (可选的): kpass123
0. OK [default]
1. Cancel
Enter a number: 0
Authentication succeeded!
[SamplePrincipal: testUser, CN=Webmaster, OU=NC, O=Shanghai University, L=ZB,
ST=Shanghai, C=CN]
```

这里第二个登录模块中直接按回车而没有输入正确的信息,验证没通过。但由于该模块的控制命令是 optional,因此对整个验证是否通过没有影响,并继续向下执行第三个控制模块,由于第三个登录模块的控制标记是 sufficient,因此验证通过后不再进行第四个模块的验证。返回应用程序,整个验证通过,最后得到的身份标志是

[SamplePrincipal: testUser, CN=Webmaster, OU=NC, O=Shanghai University, L=ZB, ST=Shanghai, C=CN]

其中第二个验证模块的 web 身份标志不再出现在主体中。

9.3.2 堆叠登录模块之间的信息共享

★ 实例说明

9.3.1 小节在登录配置文件中使用了多个登录模块,其中每个登录模块都要输入一遍验

证信息，显得很麻烦。在很多时候，用户的多重身份可能具有相同的账号或口令，这时需要将一个登录模块中的信息（如用户输入的口令等）传递另外一个登录模块。

本实例给出如何在各个登录模块传递口令信息，这些是实现单点登录的基础。

★ 编程思路：

本节前面各个小节编写自己的登录模块时，在initialize(Subject subject, CallbackHandler callbackHandler, Map sharedState, Map options)方法中有四个参数，其中Map类型的sharedState参数可用于登录模块的信息传递（如传递口令）。

LoginContext类中定义了Map类型的成员变量state，在执行登录模块的初始化方法时会将该对象传递给登录模块，因此每个登录模块可通过该对象相互传递信息。在Map对象中可保存两个对象之间的映射关系，因此登录模块中只要执行Map类型的shareSate对象的put(XX,YY)方法，即可将需要传递的信息YY保存到shareSate对象中，其他登录模块只要使用shareSate对象的get(XX)方法即可获取保存在shareSate对象中的对应信息。

为了对登录模块之间的信息传递进行控制，可在登录配置文件中使用的登录模块选项。如可规定如果登录模块的savepass选项为字符串“true”，则将用户输入的口令与字符串“mypass”相关联，保存在shareSate对象中。如果登录模块的getpass选项为字符串“true”，则从shareState对象中读取字符串“mypass”对应的口令值。

因此，和前面各小节的程序相比，登录模块的login()方法主要的改动如下：

- (1) 从登录配置文件读取 savepass 选项的值

```
boolean savepass=false;
String sp = (String)options.get("savepass");
if( (sp!=null) && sp.equals("true")){
    savepass = true;
}
```

分析：使用 initialize(Subject subject, CallbackHandler callbackHandler, Map sharedState, Map options)方法的最后一个参数，读取登录配置文件中该登录模块 savepass 选项的值，若为 true，则将 boolean 类型的变量 savepass 设置为 true。

- (2) 从登录配置文件读取 getpass 选项的值

```
boolean getpass=false;
sp = (String)options.get("getpass");
if( (sp!=null) && sp.equals("true")){
    getpass = true;
}
```

分析：使用 initialize(Subject subject, CallbackHandler callbackHandler, Map sharedState, Map options)方法的最后一个参数，读取登录配置文件中该登录模块 getpass 选项的值，若为 true，则将 boolean 类型的变量 getpass 设置为 true。

- (3) 编写 Callback 对象和用户交互时，若选项 getPass 为 true，则不需要输入口令

```
if(getpass){
    callbacks[2] = new PasswordCallback(
        "密钥库保护口令(不必输入)", false);
}
else{
    callbacks[2] = new PasswordCallback(
        "密钥库保护口令 ", false);
}
```

分析: 在选项 `getPass` 不为 `true` 时, 仍旧使用原有的 `PasswordCallback` 提示用户输入口令。为了尽量少修改前面各个小节的代码, 这里在选项 `getPass` 为 `true` 时仍旧提示输入口令的界面, 只是在后面增加一个括号提示: “不必输入”, 实际使用时可以根本不出现输入口令的界面, 或者在登录失败时再出现口令提示。

(4) 读取口令值时, 若选项 `getPass` 为 `true`, 则从 `shareState` 对象读取

```
if(getpass){
    Object op= sharedState.get("mypass");
    tmpPassword=(char[ ])op;
}
```

分析: 由于本实例约定如果登录模块的 `savepass` 选项为字符串 “true”, 则将用户输入的口令与字符串 “mypass” 相关联, 保存在 `shareSate` 对象中。因此这里通过 `sharedState.get("mypass")` 获取所保存的口令值, 并将其转换为字符数组。

(5) 若选项 `getPass` 不为 `true`, 则从 `Callback` 对象中读取用户输入的口令值

```
else{
    tmpPassword = ((PasswordCallback)callbacks[2]).getPassword();
}
```

分析: 此时仍旧使用以前的方法读取口令值。

(6) 若选项 `savePass` 为 `true`, 则从 `Callback` 对象中读取用户输入的口令值时将其保存

```
if(savepass){
    sharedState.put("mypass",tmpPassword);
}
```

分析: 根据本小节的约定, 将用户输入的口令和字符串: “mypass” 相关联, 保存在 `shareState` 对象中。

以上是一个参考流程, 实际使用时可以根据其原理在各个登录模块之间交流各种信息。

★代码与分析:

本实例应用程序仍旧使用 9.1.1 小节的 Java 程序 `SimpLogin` 和 9.1.3 小节的 `SimpLoginTXT`。

本实例使用的登录配置文件如下:

文件 `share1.config`:

```
simp {
    ShareSampLoginModule required
        savepass=true;
    ShareKSLoginModule required
        keyStoreURL ="file:C:/java/ch9/ShareLoginModule/store1"
        getpass=true;
};
```

其中, `ShareSampLoginModule` 使用了选项 `savepass=true`, 将用户输入的口令保存在 `Map` 对象中。`ShareKSLoginModule` 使用了选项 `getpass=true`, 将前面登录模块保存的读取出来。

另一个登录配置文件 share2.config 如下:

```
simp {
    ShareKSLoginModule required
        keyStoreURL = "file:C:/java/ch9/ShareLoginModule/store1"
        savepass=true;
    ShareKSLoginModule required
        keyStoreURL = "file:C:/java/ch9/ShareLoginModule/store1"
        getpass=true;
};
```

这里使用的两个登录模块都是 ShareKSLoginModule, 第一个登录模块输入的口令保存, 第二个登录模块读取保存的口令。

登录模块文件 ShareKSLoginModule.java 是在 9.2.3 小节的代码 KSLoginModule.java 的基础上修改而来的, 其完整代码如下,

```
import javax.security.auth.x500.X500Principal;
import java.io.*;
import java.net.*;
import java.security.*;
import java.security.cert.*;
import java.util.*;
import java.security.cert.Certificate;
import javax.security.auth.*;
import javax.security.auth.callback.*;
import javax.security.auth.login.*;
import javax.security.auth.spi.LoginModule;

public class ShareKSLoginModule implements LoginModule {
    // initial state
    private Subject subject;
    private CallbackHandler callbackHandler;
    private Map sharedState;
    private Map options;

    // configurable option
    private boolean debug = false;

    // the authentication status
    private boolean succeeded = false;
    private boolean commitSucceeded = false;

    // username and password
    private String username;
```

```

private char[] password;
private char[] pkpassword;
private Certificate[] cchain;
private javax.security.auth.x500.X500Principal userPrincipal;
private KeyStore ks;

public void initialize(Subject subject, CallbackHandler callbackHandler,
    Map sharedState, Map options) {

    this.subject = subject;
    this.callbackHandler = callbackHandler;
    this.sharedState = sharedState;
    this.options = options;

    // initialize any configured options
    debug = "true".equalsIgnoreCase((String)options.get("debug"));
}

public boolean login( ) throws LoginException {
    boolean savepass=false;
    boolean getpass=false;
    char[ ] tmpPassword;
    String sp = (String)options.get("savepass");
    if( (sp!=null) && sp.equals("true")){
        savepass = true;
    }
    sp = (String)options.get("getpass");
    if( (sp!=null) && sp.equals("true")){
        getpass = true;
    }
    // prompt for a user name and password
    if (callbackHandler == null)
        throw new LoginException("Error: no CallbackHandler available " +
            "to garner authentication information from the user");
    String keyStoreURL= (String)options.get("keyStoreURL");
    Callback[] callbacks = new Callback[5];
    TextOutputCallback txtCallback =
        new TextOutputCallback(
            TextOutputCallback.INFORMATION,
            "请登录密钥库");
    ConfirmationCallback confirmCallback =
        new ConfirmationCallback(
            ConfirmationCallback.INFORMATION,

```

```

        ConfirmationCallback.OK_CANCEL_OPTION,
        ConfirmationCallback.OK);
callbacks[0]= txtCallback;
callbacks[1] = new NameCallback("密钥库别名");
if(getpass){
    callbacks[2] = new PasswordCallback(
        "密钥库保护口令(不必输入) ", false);
}
else{
    callbacks[2] = new PasswordCallback(
        "密钥库保护口令 ", false);
}
callbacks[3] = new PasswordCallback("私钥保护口令 ", false);
callbacks[4] = confirmCallback;

try {
    callbackHandler.handle(callbacks);
    username = ((NameCallback)callbacks[1]).getName( );
    if(getpass){
        Object op= sharedState.get("mypass");
        tmpPassword=(char[ ])op;
    }
    else{
        tmpPassword = ((PasswordCallback)callbacks[2]).getPassword();
        if(savepass){
            sharedState.put("mypass",tmpPassword);
        }
    }
    if (tmpPassword == null) {
        // treat a NULL password as an empty password
        tmpPassword = new char[0];
    }
    password = new char[tmpPassword.length];
    System.arraycopy(tmpPassword, 0,
        password, 0, tmpPassword.length);
    ((PasswordCallback)callbacks[2]).clearPassword();

    tmpPassword = ((PasswordCallback)callbacks[3]).getPassword();
    if (tmpPassword == null) {
        tmpPassword = new char[0];
    }
    pkpassword = new char[tmpPassword.length];
    System.arraycopy(tmpPassword, 0,

```

```

        pkpassword, 0, tmpPassword.length);
    ((PasswordCallback)callbacks[3]).clearPassword();

} catch (java.io.IOException ioe) {
    throw new LoginException(ioe.toString());
} catch (UnsupportedCallbackException uce) {
    throw new LoginException("Error: " + uce.getCallback().toString() +
        " not available to garner authentication information " +
        "from the user");
}

// print debugging information
if (debug) {
    System.out.println("\t\t[SampleLoginModule] " +
        "user entered user name: " +
        username);
    System.out.print("\t\t[SampleLoginModule] " +
        "user entered password: ");
    for (int i = 0; i < password.length; i++)
        System.out.print(password[i]);
    System.out.println();
}

int confirmationResult = confirmCallback.getSelectedIndex();

if (confirmationResult == ConfirmationCallback.CANCEL) {
    throw new LoginException("Login cancelled");
}
try {
    ks = KeyStore.getInstance("JKS");
    InputStream in = new URL(keyStoreURL).openStream();
    ks.load(in, password);
    in.close();
    Key privateKey =
        ks.getKey(username, pkpassword);
    if (privateKey == null
        || !(privateKey instanceof PrivateKey)) {
        throw new FailedLoginException(
            "Unable to recover key from keystore");
    }
    succeeded = true;
    return true;
}

```



```

        catch (Exception e) {
            succeeded = false;

            throw new LoginException(
                "Error in login to Keystore " + e);
        }
    }

    public boolean commit() throws LoginException {
        if (succeeded == false) {
            return false;
        } else {
            // add a Principal (authenticated identity)
            // to the Subject
            try{
                cchain =
                    ks.getCertificateChain(username);
                X509Certificate certificate = (X509Certificate)cchain[0];
                userPrincipal = new javax.security.auth.x500.X500Principal
                    (certificate.getSubjectDN().getName());
            }catch(Exception e){
                throw new LoginException(e.toString( ));
            }
            subject.getPrincipals( ).add(userPrincipal);

            if (debug) {
                System.out.println("\t\t[SampleLoginModule] " +
                    "added SamplePrincipal to Subject");
            }

            // in any case, clean out state
            username = null;
            for (int i = 0; i < password.length; i++)
                password[i] = ' ';
            password = null;

            commitSucceeded = true;
            return true;
        }
    }

    public boolean abort() throws LoginException {
        if (succeeded == false) {
            return false;
        }
    }

```

```

    } else if (succeeded == true && commitSucceeded == false) {
        // login succeeded but overall authentication failed
        succeeded = false;
        username = null;
        if (password != null) {
            for (int i = 0; i < password.length; i++)
                password[i] = ' ';
            password = null;
        }
        userPrincipal = null;
    } else {
        // overall authentication succeeded and commit succeeded,
        // but someone else's commit failed
        logout();
    }
    return true;
}

public boolean logout() throws LoginException {

    subject.getPrincipals().remove(userPrincipal);
    succeeded = false;
    succeeded = commitSucceeded;
    username = null;
    if (password != null) {
        for (int i = 0; i < password.length; i++)
            password[i] = ' ';
        password = null;
    }
    userPrincipal = null;
    return true;
}
}

```

登录模块文件 `ShareSampLoginModule.java` 是在 9.2.2 小节的代码 `SampleLoginModule.java` 的基础上修改来的。本实例假定 `ShareSampLoginModule` 进行验证时，用户 `testUser` 和密钥库 `store1` 的保护口令相同，因此 `ShareSampLoginModule` 的代码中口令若为 “store1pass” 则验证算成功。同时为了简化程序，使用 `javax.security.auth.x500` 包中的 `X500Principal` 类作为添加到 `Principal` 的用户身份标志，而不是定义自己的 `Principal`。`X500Principal` 类的构造器中传入的名字必须符合 X.500 格式，这里简单地在名字前面加上字符串 “CN=”。

其完整代码如下：

```
// package sample.module;
```

```

import java.util.*;
import java.io.IOException;
import javax.security.auth.*;
import javax.security.auth.callback.*;
import javax.security.auth.login.*;
import javax.security.auth.spi.*;
// import sample.principal.SamplePrincipal;

public class ShareSampLoginModule implements LoginModule {

    // initial state
    private Subject subject;
    private CallbackHandler callbackHandler;
    private Map sharedState;
    private Map options;

    // configurable option
    private boolean debug = false;

    // the authentication status
    private boolean succeeded = false;
    private boolean commitSucceeded = false;

    // username and password
    private String username;
    private char[] password;

    // testUser's SamplePrincipal
    private javax.security.auth.x500.X500Principal userPrincipal;

    public void initialize(Subject subject, CallbackHandler callbackHandler,
        Map sharedState, Map options) {

        this.subject = subject;
        this.callbackHandler = callbackHandler;
        this.sharedState = sharedState;
        this.options = options;

        // initialize any configured options
        debug = "true".equalsIgnoreCase((String)options.get("debug"));
    }

    public boolean login() throws LoginException {

```

```

boolean savepass=false;
boolean getpass=false;
char[ ] tmpPassword;
String sp = (String)options.get("savepass");
if( (sp!=null) && sp.equals("true")){
    savepass = true;
}
sp = (String)options.get("getpass");
if( (sp!=null) && sp.equals("true")){
    getpass = true;
}
// prompt for a user name and password
if (callbackHandler == null)
    throw new LoginException("Error: no CallbackHandler available " +
        "to garner authentication information from the user");

Callback[] callbacks = new Callback[2];
callbacks[0] = new NameCallback("user name: ");
if(getpass){
    callbacks[1] = new PasswordCallback("Password could be skipped: ", false);
}
else{
    callbacks[1] = new PasswordCallback("password: ", false);
}

try {
    callbackHandler.handle(callbacks);
    username = ((NameCallback)callbacks[0]).getName();
    if(getpass){
        Object op= sharedState.get("mypass");
        tmpPassword=(char[ ])op;
    }
    else{
        tmpPassword = ((PasswordCallback)callbacks[1]).getPassword();
        if(savepass){
            sharedState.put("mypass", tmpPassword);
        }
    }

    if (tmpPassword == null) {
        // treat a NULL password as an empty password
        tmpPassword = new char[0];
    }
    password = new char[tmpPassword.length];

```

```

        System.arraycopy(tmpPassword, 0,
            password, 0, tmpPassword.length);
        ((PasswordCallback)callbacks[1]).clearPassword();

    } catch (java.io.IOException ioe) {
        throw new LoginException(ioe.toString());
    } catch (UnsupportedCallbackException uce) {
        throw new LoginException("Error: " + uce.getCallback().toString() +
            " not available to garner authentication information " +
            "from the user");
    }

// print debugging information
if (debug) {
    System.out.println("\t\t[SampleLoginModule] " +
        "user entered user name: " +
        username);
    System.out.print("\t\t[SampleLoginModule] " +
        "user entered password: ");
    for (int i = 0; i < password.length; i++)
        System.out.print(password[i]);
    System.out.println();
}

// verify the username/password
boolean usernameCorrect = false;
boolean passwordCorrect = false;
if (username.equals("testUser"))
    usernameCorrect = true;
if (usernameCorrect &&
    password.length == 10 &&
    password[0] == 's' &&
    password[1] == 't' &&
    password[2] == 'o' &&
    password[3] == 'r' &&
    password[4] == 'e' &&
    password[5] == '1' &&
    password[6] == 'p' &&
    password[7] == 'a' &&
    password[8] == 's' &&
    password[9] == 's' ) {

    // authentication succeeded!!!
    passwordCorrect = true;
}

```

```

        if (debug)
            System.out.println("\t\t[SampleLoginModule] " +
                               "authentication succeeded");
        succeeded = true;
        return true;
    } else {
        // authentication failed -- clean out state
        if (debug)
            System.out.println("\t\t[SampleLoginModule] " +
                               "authentication failed");
        succeeded = false;
        username = null;
        for (int i = 0; i < password.length; i++)
            password[i] = ' ';
        password = null;
        if (!usernameCorrect) {
            throw new FailedLoginException("User Name Incorrect");
        } else {
            throw new FailedLoginException("Password Incorrect");
        }
    }
}

public boolean commit() throws LoginException {
    if (succeeded == false) {
        return false;
    } else {
        // add a Principal (authenticated identity)
        // to the Subject

        // assume the user we authenticated is the SamplePrincipal
        userPrincipal = new javax.security.auth.x500.X500Principal(
            "CN="+username);
        if (!subject.getPrincipals().contains(userPrincipal))
            subject.getPrincipals().add(userPrincipal);

        if (debug) {
            System.out.println("\t\t[SampleLoginModule] " +
                               "added SamplePrincipal to Subject");
        }

        // in any case, clean out state
        username = null;
        for (int i = 0; i < password.length; i++)

```

```

        password[i] = ' ';
        password = null;

        commitSucceeded = true;
        return true;
    }
}

public boolean abort() throws LoginException {
    if (succeeded == false) {
        return false;
    } else if (succeeded == true && commitSucceeded == false) {
        // login succeeded but overall authentication failed
        succeeded = false;
        username = null;
        if (password != null) {
            for (int i = 0; i < password.length; i++)
                password[i] = ' ';
            password = null;
        }
        userPrincipal = null;
    } else {
        // overall authentication succeeded and commit succeeded,
        // but someone else's commit failed
        logout();
    }
    return true;
}

public boolean logout() throws LoginException {

    subject.getPrincipals().remove(userPrincipal);
    succeeded = false;
    succeeded = commitSucceeded;
    username = null;
    if (password != null) {
        for (int i = 0; i < password.length; i++)
            password[i] = ' ';
        password = null;
    }
    userPrincipal = null;
    return true;
}
}

```

★运行程序

程序运行在 C:\java\ch9\ShareLoginModule 目录，其中拷贝了 9.1.1 小节编译得到的 SimpLogin.class、9.1.3 小节的 SimpLoginTXT.class 和 9.3.1 小节的密钥库 store1。此外保存了本节的登录配置文件 share1.config 和 share2.config、登录模块 ShareSampLoginModule.java 和 ShareKSLoginModule.java。

输入

```
javac ShareSampLoginModule.java
```

```
javac ShareKSLoginModule.java
```

编译程序，输入

```
java -Djava.security.auth.login.config==share1.config SimpLogin
```

运行程序，首先弹出 图 9-5所示的窗口。



图 9-5 第一个登录模块保存口令

输入用户名testUser和口令store1pass后，单击确定按钮，此时口令store1pass将保存在各个登录模块共享的Map对象中，随即弹出 图 9-6所示的窗口。

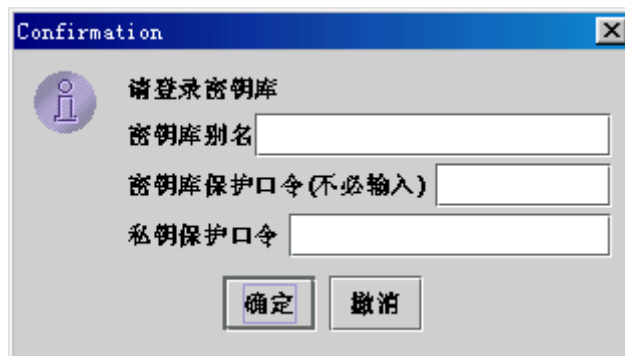


图 9-6第二个登录模块使用保存的口令

在该窗口中可见第二项输入密钥库保护口令中提示“不必输入”，在别名处输入 web，私钥保护口令处输入 kpass123，单击“确定”按钮，则 DOS 窗口提示：

```
Authentication succeeded!
```

```
[CN=testUser, CN=Webmaster, OU=NC, O=Shanghai University, L=ZB, ST=Shanghai, C=CN]
```

如果输入

```
java -Djava.security.auth.login.config==share2.config SimpLogin
```

运行程序，则首先弹出 图 9-7所示的窗口。



图 9-7 密钥库登录模块保存口令

输入别名web和密钥库保护口令store1pass及私钥保护口令kpass123后，单击确定按钮，此时密钥库保护口令store1pass将保存在各个登录模块共享的Map对象中，随即弹出 图 9-8 所示的窗口。

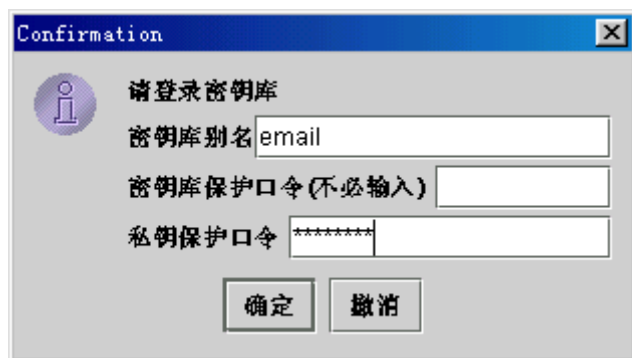


图 9-8 密钥库登录模块使用保存的口令

在该窗口中可见第二项输入密钥库保护口令中提示“不必输入”，在别名处输入 email，私钥保护口令处输入 kpass123，单击“确定”按钮，则 DOS 窗口提示：

Authentication succeeded!

[CN=Webmaster, OU=NC, O=Shanghai University, L=ZB, ST=Shanghai, C=CN,
CN=Email User, OU=ME, O=SouthEast University, L=GL, ST=Nanjing, C=CN]

可见用户以两种身份登录成功。

如果应用程序使用 9.1.3 小节基于文本交互的 SimpLoginTXT，其交互过程如下：

```
C:\java\ch9\ShareLoginModule> java -Djava.security.auth.login.config==share1.config
SimpLoginTXT
user name: testUser
password: store1pass
请登录密钥库
密钥库别名 web
密钥库保护口令(不必输入)
私钥保护口令 kpass123
0. OK [default]
1. Cancel
```

```

Enter a number: 0
Authentication succeeded!
[CN=testUser, CN=Webmaster, OU=NC, O=Shanghai University, L=ZB, ST=Shanghai, C=CN]

C:\java\ch9\ShareLoginModule> java -Djava.security.auth.login.config=share2.
config SimpLoginTXT
请登录密钥库
密钥库别名 web
密钥库保护口令 store1pass
私钥保护口令 kpass123
0. OK [default]
1. Cancel
Enter a number: 0
请登录密钥库
密钥库别名 email
密钥库保护口令 (不必输入)
私钥保护口令 kpass123
0. OK [default]
1. Cancel
Enter a number: 0
Authentication succeeded!
[CN=Webmaster, OU=NC, O=Shanghai University, L=ZB, ST=Shanghai, C=CN, CN=Email U
ser, OU=ME, O=SouthEast University, L=GL, ST=Nanjing, C=CN]

```

9.4 编写自己的回调处理器

在 9.1 节的例子中使用了 JAAS 提供的回调处理器 `com.sun.security.auth.callback.DialogCallbackHandler` 和 `com.sun.security.auth.callback.TextCallbackHandler` 类，当这些满足不了要求时，可以编写自己的回调处理器。由于用户需要的界面千差万别，因此编写自己的回调处理器比编写自己的登录模块更常用。

9.4.1 最简单的回调处理器

★ 实例说明

本实例提供了一个简化的登录模块，演示了登录模块的工作原理和编程方法。

★ 编程思路：

在 9.2 节中，登录模块创建回调处理器对象，然后创建各种 `Callback` 对象，如 `NameCallback`、`PasswordCallback`、`ChoiceCallback`、`ConfirmationCallback`、`LanguageCallback`、`TextInputCallback` 和 `TextOutputCallback` 等，将其组成 `Callback` 类型的数组，传递给回调处

理器的 `handle()` 方法，就可以出现用户交互界面。用户输入完成以后，登录模块就可以从 `Callback` 类型的数组中提取各个 `Callback` 对象，执行 `Callback` 对象的方法提取用户输入的内容了。

因此，编写回调处理器需要实现 `Callback` 接口，并实现其 `handle()` 方法。`handle()` 方法中根据传入的 `Callback` 类型的数组中各个元素，知道登录模块需要用户输入的是哪些信息，进而布置用户界面，用户输入完毕应将用户输入的内容保存到对应的 `Callback` 对象中。

故 `handle()` 方法的基本内容是遍历参数数组中的每个元素，判断其是 `NameCallback`、`PasswordCallback`、`ChoiceCallback`、`ConfirmationCallback`、`LanguageCallback`、`TextInputCallback` 和 `TextOutputCallback` 中的哪一种，继而将其强制转换为对应的类型，并执行相应的方法获取编写登录模块的程序员传递来的信息。其基本编程框架如下：

```
class XXX implements CallbackHandler{
    public void handle(Callback[] callbacks)
        throws IOException, UnsupportedCallbackException {
        for (int i = 0; i < callbacks.length; i++) {
            if (callbacks[i] instanceof NameCallback) {
                // 提示用户输入账号
            } else if (callbacks[i] instanceof PasswordCallback) {

                // 提示用户输入口令
            } else if (callbacks[i] instanceof ChoiceCallback) {
                // 提示用户从给出的选项选择一个
            } else if (callbacks[i] instanceof ConfirmationCallback) {
                // 让用户进行确认
            } else if (callbacks[i] instanceof LanguageCallback) {
                // 让用户输入语言等相关的信息
            } else if (callbacks[i] instanceof TextInputCallback) {
                // 提示用户输入文本
            } else if (callbacks[i] instanceof TextOutputCallback) {
                // 相关的信息或警告、错误提示信息
            } else {
                throw new UnsupportedCallbackException
                    (callbacks[i], "Unrecognized Callback");
            }
        } // end for
    } // end handle()
} // end class
```

需要时还可以定义自己的 `Callback` 对象。

其中，`handle()` 方法参数数组中的元素如果是 `NameCallback` 类型，其处理步骤为：

(1) 强制转换

```
NameCallback nameCallback = (NameCallback) callbacks[i];
```

分析：将 `handle()` 方法参数数组中属于 `NameCallback` 类型的元素强制转换成 `NameCallback` 类型。

(2) 显示提示信息

```
System.out.print( nameCallback.getPrompt()
    + "["+ nameCallback.getDefaultName()+"]:" );
```

分析：通过 NameCallback 对象 `getPrompt()` 方法获得登录模块中传来的提示信息，这里通过文本的界面显示出来，提示用户输入什么信息。也可制作各种图形界面作提示。

在登录配置文件中可能已经通过登录模块选项提供了缺省的名称，通过 NameCallback 对象的 `getDefaultName()` 方法可以获得该值，在提示用户输入信息时可提醒用户缺省值是多少。

(3) 读取用户输入

```
String username = new BufferedReader(
    new InputStreamReader(System.in)).readLine();
if(username==null || username.length() == 0){
    username= nameCallback.getDefaultName();
}
```

分析：这里为简单起见，不妨直接通过标准输入流读取一行用户输入的用户名，也可制作各种图形界面让用户输入。

如果用户没有输入，则使用 NameCallback 对象的 `getDefaultName()` 方法获取登录配置文件中为该项设置的缺省值。

(4) 保存用户输入

```
nameCallback.setName(username);
```

分析：用户输入的内容通过 NameCallback 对象的 `setName()` 方法保存在 Callback 对象中，供登录模块通过其 `getName()` 方法从中提取。

类似地，如果是 PasswordCallback 类型，可通过 PasswordCallback 对象 `getPrompt()` 方法获得登录模块中传来的提示信息，可通过图形或文本的界面显示出来提示用户输入什么信息。通过 PasswordCallback 对象的 `isEchoOn()` 方法获得登录模块是否要求口令回显，若得到的是 `false`，则用户输入时口令显示应该用 “*” 显示，以免用户输入时周围人看到口令内容。口令输入完毕，通过 PasswordCallback 对象的 `setPassword()` 方法保存在 Callback 对象中，供登录模块通过其 `getPassword()` 方法从中提取，登录模块使用完口令后还可以使用 PasswordCallback 对象的 `clearPassword()` 方法彻底清除 Callback 对象中的口令，以提高安全性。

如果是 ChoiceCallback 类型，则可通过 ChoiceCallback 对象的 `getPrompt()` 方法得到登录模块传来的提示字符串，可通过图形或文本的界面显示出来提示用户输入什么信息。可通过 ChoiceCallback 对象的 `getChoices()` 方法得到字符串数组，其中包含的是登录模块要求回调处理器显示的选项。可通过 ChoiceCallback 对象的 `getDefaultChoice()` 方法得到整型数，表明登录模块要求回调处理器默认选择第几项。显示的选项。可通过 ChoiceCallback 对象的 `allowMultipleSelections()` 方法得到 boolean 类型值，若为 `true` 则登录模块要求回调处理器允许多选。这些都可用于界面的布置。用户输入完毕后，将用户选择了第几项通过 ChoiceCallback 对象的 `setSelectedIndex()` 方法保存在 Callback 对象中。如果允许选择多项，则可先将用户选择了哪几项放在整型数组中，然后通过 ChoiceCallback 对象的 `setSelectedIndexs()` 方法保存在 Callback 对象中。

如果是 LanguageCallback 类型的对象，可先通过检测或用户输入创建 Local 类型的对象，传地给 LanguageCallback 对象的 `setLocale()` 方法，将用户的语言环境保存在 LanguageCallback 对象中传递给登录模块。

如果是 `TextInputCallback` 类型的对象，可通过 `TextInputCallback` 对象 `getPrompt()` 方法获得登录模块中传来的提示信息，可通过图形或文本的界面显示出来提示用户输入什么信息，通过 `TextInputCallback` 对象的 `getDefaultText()` 方法可得到登录模块要求默认使用何文本，这些可用于布置界面并提取用户输入内容。最后将用户输入的内容通过 `TextInputCallback` 对象的 `setText(String text)` 方法保存在 `Callback` 对象中，供登录模块通过其 `getText()` 方法从中提取。

如果是 `TextOutputCallback` 类型的对象，可以通过 `TextOutputCallback` 对象的 `getMessage()` 方法获得要显示什么信息，通过 `TextOutputCallback` 对象的 `getMessageType()` 方法获得信息是一般信息、还是警告或出错提示，进而安排用户界面，如使用标签在特定位置显示信息，或显示一个惊叹号，再跟上要显示的内容。

★代码与分析：

本实例编写的简单的回调处理器如下，它可以处理 `NameCallback`、`PasswordCallback`、`TextOutputCallback` 和 `ConfirmationCallback` 四类 `Callback` 对象，因此可以适用于本章各个应用程序和登录模块。为了简化程序，对于 `ConfirmationCallback` 类型，程序中不作处理，即不和用户交互而直接当作用户已确认。

回调处理器的完整代码如下：

```
import java.io.*;
import javax.security.auth.callback.*;

public class MyCallbackHandler implements CallbackHandler{
    public void handle(Callback[] callbacks)
        throws IOException, UnsupportedCallbackException {
        for (int i = 0; i < callbacks.length; i++) {
            if (callbacks[i] instanceof NameCallback) {
                //强制转换
                NameCallback nameCallback = (NameCallback)callbacks[i];
                // 显示提示信息
                System.out.print( nameCallback.getPrompt()
                    + "["+ nameCallback.getDefaultName()+"]:" );
                System.out.flush();
                //读取用户输入
                String username = new BufferedReader(
                    new InputStreamReader(System.in)).readLine();
                if(username==null || username.length() ==0){
                    username= nameCallback.getDefaultName();
                }
                //保存用户输入
                nameCallback.setName(username);
            } else if (callbacks[i] instanceof PasswordCallback) {
                PasswordCallback passwordCallback =
                    (PasswordCallback) callbacks[i];
                System.out.print( passwordCallback.getPrompt() + "? " );
                System.out.flush();
                String password = new BufferedReader(
```

```

        new InputStreamReader(System.in)).readLine();
        passwordCallback.setPassword(password.toCharArray());
        password = null;
    } else if (callbacks[i] instanceof TextOutputCallback) {
        TextOutputCallback textOutputCallback=
            (TextOutputCallback)callbacks[i];
        System.out.println(textOutputCallback.getMessage( ));
    } else if (callbacks[i] instanceof ConfirmationCallback) {
    }
    else {
        throw new UnsupportedOperationException(
            callbacks[i], "Unrecognized Callback");
    }
} // end for
} // end handle( )
} // end class

```

使用该回调处理器的应用程序代码根据 9.1.3 小节的程序修改而来，其完整代码如下：

```

import javax.security.auth.*;
import javax.security.auth.login.*;

public class SimpLoginMyCB {
    public static void main(String[] args) throws Exception {
        //登录
        MyCallbackHandler handler=new MyCallbackHandler( );
        LoginContext c = new LoginContext("simp",handler);
        boolean pass;
        try {
            c.login();
            //登录成功
            pass=true;
        }
        catch (LoginException le) {
            //登录失败
            pass=false;
            System.err.println("Authentication failed:");
            System.err.println(" " + le.getMessage());
        }
        //显示登录结果
        if(!pass){
            System.out.println("Sorry");
        }
        else{
            System.out.println("Authentication succeeded!");
        }
    }
}

```

```

        Subject s = c.getSubject();
        System.out.println(s.getPrincipals());
    }
}
}

```

本实例不妨仍旧使用 9.1.1 小节最简单的登录所使用的登录配置文件 **Simp.config** 和 9.1.2 小节所使用的登录配置文件 **keystore2.config**。

Simp.config 文件内容如下：

```

simp {
    com.sun.security.auth.module.KeyStoreLoginModule required
        keyStoreURL = "file:C:/java/ch9/SimpLogin/mykeystore";
};

```

keystore2.config 文件内容如下：

```

simp {
    com.sun.security.auth.module.KeyStoreLoginModule required
        keyStoreURL = "file:C:/java/ch9/SimpLogin/mykeystore"
        keyStoreAlias=mytest;
};

```

这两个文件都使用了 C:\java\ch9\SimpLogin 目录中的 mykeystore 密钥库。

★运行程序

程序运行在 C:\java\ch9\MyCallback 目录，输入 `javac *.java` 编译程序，使用 **Simp.config** 登录配置文件的运行过程如下：

```

C:\java\ch9\MyCallback> java -Djava.security.auth.login.config==Simp.config Sim
pLoginMyCB
请登录 keystore
Keystore 别名: [null]:mytest
Keystore 密码: ? wshr.ut
私人关键密码 (可选的): ?
Authentication succeeded!
[CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai, C=
CN]

```

使用 **keystore2.config** 登录配置文件的运行过程如下：

```

C:\java\ch9\MyCallback> java -Djava.security.auth.login.config==keystore2.conf
ig SimpLoginMyCB
请登录 keystore
Keystore 别名: [mytest]:
Keystore 密码: ? wshr.ut
私人关键密码 (可选的): ?
Authentication succeeded!
[CN=Xu Yingxiao, OU=Network Center, O=Shanghai University, L=ZB, ST=Shanghai, C=
CN]

```

这里, 在输入 **KeyStore** 别名时直接按了回车, 使用了登录配置文件中指定的缺省值。进一步可使用本章 9.2 节中自己定义各个登录模块进行试验。

9.4.2 图形界面口令输入的安全性

★ 实例说明

9.4.1 小节编写回调处理器时涉及界面的制作。9.4.1 小节口令输入时, 用户输入的口令值会回显在屏幕上, 此时若用户周围有人, 便很容易看到口令的内容。

本实例给出一个普通的应用程序, 演示了图形界面中如何控制口令的回显。

★ 编程思路:

Java Swing 提供了 **JPasswordField** 类用于输入一行口令, 通过其 **setEchoChar()** 方法可以设置回显的字符, 通过其 **getPassword()** 方法可以获得输入的口令值。

如果输入一般的文本, 不需要设置回显的字符, 则可以使用 **JTextField** 类。通过其 **getText()** 方法获得用户输入的文本。

本实例通过一个按钮指示输入完毕, 用户输入用户名和口令后单击“确定”按钮, 击发事件处理器进行处理, 本实例的处理简单地将用户输入内容在屏幕上显示出来。

进一步, 如果希望用户可以通过在输入区域按回车指示输入完毕, 可以使用 **JPasswordField** 类及 **JTextField** 类按照类似按钮的操作进行处理。如可在代码中增加一段:

```
passwordField.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        //获取输入的口令值
        JPasswordField input = (JPasswordField)e.getSource();
        char[] password = input.getPassword();
        //获取输入的用户名
        String name = textField.getText();
        System.out.println("Input name "+name);
        System.out.println("Input Password "+
            new String(password));
    }
});
```

★代码与分析:

本实例的完整代码如下:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class PasswordDemo {
    //口令和文本输入域
    JPasswordField passwordField = new JPasswordField(12);
    JTextField textField = new JTextField(12);
    public static void main(String[] argv) {
```



```

        PasswordDemo pd=new PasswordDemo();
        pd.go();
    }
    void go() {
        //设置口令回显字符为 x
        passwordField.setEchoChar('X');
        //创建各个标签、按钮
        final JFrame f = new JFrame("PasswordDemo");
        JLabel label1 = new JLabel("Input name: ");
        JLabel label2 = new JLabel("Input password: ");
        JButton button1 = new JButton("OK");
        JButton button2 = new JButton("Cancel");
        //鼠标单击“OK”按钮则执行该段代码
        button1.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                //获取输入的名字
                String name = textField.getText();
                //获取输入的口令值
                char[] password = passwordField.getPassword();
                System.out.println("Input name "+name);
                System.out.println("Input Password "+
                    new String(password));
            }
        });
        //使用网格布局安排各个按钮和输入域
        JPanel contentPane = new JPanel(new GridLayout(3,2));
        contentPane.add(label1);
        contentPane.add(textField);
        contentPane.add(label2);
        contentPane.add(passwordField);
        contentPane.add(button1);
        contentPane.add(button2);
        f.setContentPane(contentPane);
        //窗口事件，用于关闭窗口
        f.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) { System.exit(0); }
        });
        f.pack();
        f.setVisible(true);
    }
}

```

★运行程序

程序运行在C:\java\ch9\Password目录，输入“java PasswordDemo”运行程序，则出现图 9-9所示的窗口。

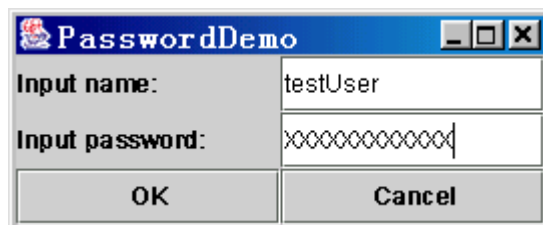


图 9-9 图形界面的口令回显

图中可见口令输入区域使用了预定的字符进行回显，输入完毕单击“确定”按钮，则DOS 窗口提示如下信息：

```
Input name testUser
Input Password testPassword
```

9.4.3 文本界面口令输入的安全性

★ 实例说明

9.4.2 小节使用图形界面很方便地实现了口令输入时回显指定字符，本实例给出按照文本方式交互时如何指定回显字符。

★ 编程思路：

文本界面输入信息多通过标准输入流 `System.in`，该输入流在输入时总是会将输入的内容原封不动地回显在屏幕上，本实例使用线程在后台随时清除回显的信息，尽管这样做并不完美，但还是能满足一定的需要。

为了清除回显信息，需要不停地在屏幕上打印一串字符以覆盖用户输入回显的字符。此外每次打印不能换行，但需将光标移动到最前面，因此可使用转义符号“`\r`”。如果原先屏幕上提示输入“`Input:`”，则清除回显信息时必须保留原有的提示信息“`Input:`”，因此应该打印“`\rInput:` ”，在冒号后面有两个空格，是为了清除用户可能回显的口令。此外，清除后应该将光标移动到紧靠冒号的后面，因此最后应该再加上两个转义符“`\b`”，将光标向前移动两个位置。最终，线程应该反复打印的字符串为：“`\rInput: \b\b`”（“`\b`”前面有两个空格）。

因此，如果字符串 `prompt` 中保存的是提示符，则线程应该反复执行如下打印语句进行刷新：

```
System.out.print("\r" + prompt + "  \b\b");
```

★代码与分析：

本实例定义的线程 `MaskingThread` 的完整代码如下：

```
public class MaskingThread extends Thread {
    private boolean stop = false;
```

```

private String prompt;
private int time;

public MaskingThread(String prompt, int time){
    //传入提示符和刷新闻隔时间
    this.prompt=prompt;
    this.time=time;
}

public void run() {
    while(!stop) {
        try {
            sleep(time);
        }catch (InterruptedException e) {
            e.printStackTrace();
        }
        if(!stop){
            //执行刷新
            // "\b"前面有两个空格
            System.out.print("\r" + prompt + "  \b\b");
            System.out.flush();
        }
    }
}

//停止线程
public void stopMasking() {
    this.stop = true;
}
}

```

在应用程序中使用 **MaskingThread** 时，只要先创建 **MaskingThread** 对象，在提示输入口令以后执行线程的 **start()** 方法启动线程，则可以开始读取键盘输入的操作，键盘输入时该行中回显的内容都将被线程刷新。键盘输入完毕，可执行线程的 **stopMasking()** 方法终止线程。其完整代码如下：

```

import java.io.*;
import java.util.*;

public class TXTPassword {

    public static void main(String argv[]) throws IOException {

        String prompt="Input:";
        //创建线程，传入提示符和刷新闻隔时间
        MaskingThread mask= new MaskingThread(prompt,1);
        //启动线程，在后台反复刷新
        mask.start();
        //提示用户输入
        System.out.print(prompt);
        //读取键盘输入
    }
}

```

```

        String password = new BufferedReader(
            new InputStreamReader(System.in)).readLine();
        //终止线程
        mask.stopMasking();
        System.out.println("The Passowrd inputed is: "+password);
    }
}

```

★运行程序

程序运行在 C:\java\ch9\Password 目录，输入 “java TXTPassword” 运行程序，则程序输出如下：

```

C:\java\ch9\Password>java TXTPassword
Input:
The Passowrd inputed is: TestPassword

```

在 “Input:” 提示符后面输入的口令在屏幕上不显示（其实是刚显示一个字符就被线程立刻清除）。

9.4.4 更加安全的文本界面口令输入方式

★ 实例说明

9.4.1 小节和 9.4.3 小节为了简化程序，直接将口令通过 `BufferedReader` 类的 `readLine()` 方法读入字符串。由于字符串保存的口令无法从内容中物理清除，因此不够安全，本实例提供从各种输入流读取口令的方法 `readPassword()`。

★ 编程思路：

通过输入流的 `read()` 方法每次读取一个字节，不妨以流结束标志 (-1) 或换行标志 ('\n') 来分隔口令。因此读取到这两种标志则不再继续读取输入流。

```

loop: while (true) {
    switch (c = in.read()) {
        case -1:
        case '\n':
            System.out.println("over____"+(int)c);
            break loop;
    }
}

```

如果读到光标移至行首的标志 "\r"，则三种情况。

一种是输入一串字符后，直接回车。此时输入流中的内容是以 "\r" 标志跟上一个换行标志 "\n"，读取口令完成，不需要继续读输入流中后面的内容。

另一种情况是，输入一串字符，光标移至行首后尚未换行就的结束了输入流。此时输入流中末尾的内容是以 "\r" 标志跟上一个流结束标志结束。不需要继续读输入流中后面的内容。

第三种是输入一串字符后，光标移至行首（不换行），对开头几个字符重新作了修改。

此时应取修改后的内容作为实际的口令值。此时"\r"标志后面的内容是其他字符。如 `System.out.print()` 语句打印一串内容: "123456\abc", 将屏幕输出重定向到文件中 (如果使用 Windows 中的记事本打开该文件, 会发现其中的内容类似: **123456■abc**)。将该文件作为输入流, 用本小节的 `readPassword()` 方法读到的口令则为: abc456。

```
case '\r':
    int c2 = in.read();
    if ((c2 != '\n') && (c2 != -1)) {
        if (!(in instanceof PushbackInputStream)) {
            in = new PushbackInputStream(in);
        }
        ((PushbackInputStream) in).unread(c2);
    } else{
        break loop;
    }
}
```

这里, 如果从输入流中读到"\r", 则试读其后面一个字符, 如果是前面两种情况, 则退出循环, 不再读取输入流。如果是前面的第三种情况, 则使用类 `PushbackInputStream`, 执行其 `unread()` 方法取消试读操作。这样下一次再使用 `in.read()` 读取字符时, 已试读过的内容会再次读一遍。

★代码与分析:

本实例使用 9.4.3 小节的 `MaskingThread` 程序演示了本实例提供的 `readPassword()` 方法。

```
import java.io.*;
import java.util.*;

public class CharPassword {
    public static void main(String argv[]) throws IOException {
        CharPassword cp=new CharPassword();
        String prompt="Enter Password:";
        //创建线程, 传入提示符和刷新间隔时间
        MaskingThread mask= new MaskingThread(prompt,1);
        //启动线程, 在后台反复刷新
        mask.start();
        //提示用户输入
        System.out.print(prompt);
        //读取键盘输入
        char [] passwd=cp.readPassword(System.in);
        //终止线程
        mask.stopMasking();
        //使用口令
        System.out.print("The Passowrd inputed is: ");
        for(int i=0;i<passwd.length;i++){
            System.out.print(passwd[i]);
        }
        //清除口令
        for(int i=0;i<passwd.length;i++){
```

```

        passwd[i]=' ';
    }
}

private char[] readPassword(InputStream in) throws IOException {

    char[] lineBuffer;
    char[] buf;
    int i;
    buf = lineBuffer = new char[128];
    int room = buf.length;
    int offset = 0;
    int c;

loop: while (true) {
    switch (c = in.read()) {
        case -1:
        case '\n':
            break loop;
        case '\r':
            int c2 = in.read();
            if ((c2 != '\n') && (c2 != -1)) {
                if (!(in instanceof PushbackInputStream)) {
                    in = new PushbackInputStream(in);
                }
                ((PushbackInputStream)in).unread(c2);
            } else
                break loop;

        default:
            if (--room < 0) {
                buf = new char[offset + 128];
                room = buf.length - offset - 1;
                System.arraycopy(lineBuffer, 0, buf, 0, offset);
                Arrays.fill(lineBuffer, ' ');
                lineBuffer = buf;
            }
            buf[offset++] = (char) c;
            break;
    }
}

    if (offset == 0) {
        return null;
    }
}

```

```

    }

    char[] ret = new char[offset];
    System.arraycopy(buf, 0, ret, 0, offset);
    Arrays.fill(buf, ' ');

    return ret;
}
}

```

★运行程序

程序运行在 C:\java\ch9\Password 目录，输入 “java CharPassword” 运行程序，其运行过程如下：

```

C:\java\ch9\Password>java CharPassword
Enter Password:
The Passowrd inputed is: wshr.ut

```

其中在输入时键盘输入内容不显示。

9.5 基于身份的授权

前面各节都只实现 JAAS 验证的功能，本章介绍如何根据用户登录后的身份确定用户可以做什么事情。

9.5.1 使用策略文件的基于身份授权

★ 实例说明

本实例在 9.1.2 小节的代码 `SimpLoginTXT.java` 的基础上，给出如何通过策略文件基于身份进行授权，指定某段代码只有某种身份才能执行。该方法又称为申明授权方式 (Declarative authorization)。

★ 编程思路：

本实例的场景是：类 `MyActionKS` 中有一段代码需要读取 `c:\autoexec.bat` 文件，编程者编写的类 `LoginDoAsKS` 需要调用该代码，但考虑到该文件比较重要，因此准备让用户或系统管理员配置谁可以执行该代码，如只允许用户 `Webmaster` 来执行类 `MyActionKS` 中的该段代码，对其他用户则不允许。

使用 `Subject` 对象的静态方法 `doAsPrivileged (Subject subject, PrivilegedAction action, AccessControlContext acc)` 可以将用户登录的身份 `subject` 和欲执行的代码 `action` 相关联，这样可以以身份 `subject` 执行参数 `action` 传入的代码（如 `MyActionKS` 类型的对象）的 `run()` 方法。其中最后一个参数这里可以指定为 `null`。

JAAS 的各个登录模块在通过验证时都会将对应的身份标志添加到代表用户的 `Subject` 对象中，在 `action` 参数传入的代码中可以获取用户的身份标志，并检测其权限。

为了使类 `MyActionKS` 中的代码能够被 `Subject.doAsPrivileged()` 以某种身份执行，类

MyActionKS 必须实现 PrivilegedAction 接口。该接口中需实现 run() 方法，需要执行的代码就放在该方法中。作为测试，不妨在其 run() 方法中使用输入流读取 c:\autoexec.bat 文件的内容。

```
in = new BufferedReader(new FileReader(filename));
while ((s = in.readLine()) != null) {
    content+=s+"\n";
}
```

在类 FileReader 的构造器中会自动执行权限的检测，此外也可按照 8.5.1 小节的方法使用 AccessController.checkPermission() 方法检测该段代码是否获得了某个权限。

应用程序 LoginDoAsPrvKS 中登录并执行 Subject.doAsPrivileged() 方法的步骤如下：

(1) 创建回调处理器

```
TextCallbackHandler handler=new TextCallbackHandler();
```

分析：这里不妨使用 JAAS 提供的使用 TextCallbackHandler() 文本交互界面，也可使用 DialogCallbackHandler 或 9.4 节自己的回调处理器提供用户界面。

(2) 执行登录验证

```
c = new LoginContext("simp",handler);
c.login();
```

分析：创建 LoginContext 对象，传入登录配置文件中的条目名称和上一步创建的回调处理器。执行 LoginContext 对象的 login() 方法，则将使用回调处理器和用户交互，获取用户输入的信息，并进行验证。验证通过后，各个登录模块会将对应的身份标志信息添加到 Subject 对象。

(3) 获取当前用户身份

```
Subject subj = c.getSubject();
```

分析：执行 LoginContext 对象的 getSubject() 方法，获取当前登录用户的主体（身份标志集合）。

(4) 创建欲以 subj 身份执行的代码

```
MyactionKS myact=new MyactionKS();
```

分析：MyactionKS 是本节开头定义的实现 PrivilegedAction 接口的类。

(5) 以 subj 身份执行代码

```
String filecontent=(String) Subject.doAsPrivileged(subj, myact,null);
```

分析：执行 Subject 类的静态方法 doAsPrivileged，第一个参数第三步获得的用户身份，第二个参数传入上一步创建的欲执行的代码，最后一个参数这里传入 null。

被执行的代码是 MyactionKS 中的 run() 方法，执行结果通过 MyactionKS 中的 run() 返回给 Subject.doAsPrivileged()，并最终通过 Subject.doAsPrivileged() 方法的名称返回。由于在 MyactionKS 中的 run() 方法中实际返回的是包含 c:\autoexec.bat 文件内容的字符串，所以这里将其强制转换为字符串类型。

Subject.doAsPrivileged(subj, myact,null) 方法在执行 myact 参数传入的 MyactionKS 对象的 run() 方法时，当执行到执行到读取文件的操作时，会自动检测登录上下文中所有代码是否有读对应文件的权限。因此需对其进行授权。

自 J2SDK1.4 开始，由于集成了 JAAS，Java 中的策略文件开始支持基于身份的授权，其语法格式为：

```
grant <signer(s) field>, <codeBase URL>
```



```

<Principal field(s)> {
    permission perm_class_name "target_name", "action";
    ....
    permission perm_class_name "target_name", "action";
};

```

因此，只要在第 8 章的策略文件中 grant 后面加上一个域：Principal 即可。最简单的定义如：

```

grant      Principal javax.security.auth.x500.X500Principal
          "CN=Webmaster,OU=NC,O=Shanghai University,L=ZB,ST=Shanghai,C=CN"
{
    permission java.io.FilePermission "c:\\-", "read";
};

```

则执行 Subject.doAsPrivileged(subj, myact,null)时，只要 Subject 类型的对象 subj 中有名字为 testUser 的 Principal，即用户登录的身份中包含 testUser，则参数 myact 传入的 MyactionKS 对象的 run() 方法在执行时就有权限读 c:盘所有文件。

除了使用 X500Principal，也可以使用 9.2.1 小节定义的 MyPrincipal，9.2.2 小节定义的 sample.principal.SamplePrincipal。如

```

grant      Principal MyPrincipal "testUser"{
    permission java.io.FilePermission "c:\\-", "read";
}

```

或

```

grant      Principal sample.principal.SamplePrincipal "testUser"{
    permission java.io.FilePermission "c:\\-", "read";
}

```

此外，为了能够进行登录验证，以及执行 doAsPrivileged() 等方法，策略文件中一般需要授予以下权限。

```

permission javax.security.auth.AuthPermission "createLoginContext";
permission javax.security.auth.AuthPermission "doAs";
permission javax.security.auth.AuthPermission "doAsPrivileged";
permission javax.security.auth.AuthPermission "modifyPrincipals";
permission javax.security.auth.AuthPermission "getSubject";

```

★代码与分析：

本实例定义的需要以特定身份执行的代码 MyactionKS 完整内容如下：

```

import java.security.*;
import java.io.*;

public class MyactionKS implements PrivilegedAction {
    public Object run() {
        String content="";
        String s;
        //本代码要读取的文件

```

```

        String filename="c:\\autoexec.bat";
        BufferedReader in;
        try{
            //读取文件
            in = new BufferedReader(new FileReader(filename));
            while ((s = in.readLine( )) != null) {
                content+=s+"\n";
            }
        }
        catch(IOException e){
            System.out.println(e);
        }
        //返回文件内容
        return content;
    }
}

```

本实例中执行代码MyactionKS的程序LoginDoAsPrvKS.java的完整内容如下:

```

import com.sun.security.auth.callback.TextCallbackHandler;
import javax.security.auth.*;
import javax.security.auth.login.*;
import java.security.*;
import java.util.*;
import java.io.*;
public class LoginDoAsPrvKS{
    public static void main(String[] args) {
        LoginContext c=null;
        //登录
        TextCallbackHandler handler=new TextCallbackHandler( );
        boolean pass;
        try {
            c = new LoginContext("simp",handler);
            c.login();
        }
        catch (LoginException le) {
            System.out.println("Authentication failed!");
            System.exit(1);
        }
        //获取Subject.doAsPrivileged( )方法的参数
        Subject subj = c.getSubject();
        MyactionKS myact=new MyactionKS( );
        try {
            //执行代码
            String filecontent=(String) Subject.doAsPrivileged(

```

```

subj, myact,null);

//输出执行结果
System.out.println("\n Below is result of run MyactionKS");
System.out.println(filecontent);
} catch (AccessControlException e) {
System.out.println(e);
}
}
}
}

```

本实例使用的策略文件 authX500.policy 如下:

```

grant {
    permission javax.security.auth.AuthPermission "createLoginContext";
    permission javax.security.auth.AuthPermission "doAs";
    permission javax.security.auth.AuthPermission "doAsPrivileged";
    permission javax.security.auth.AuthPermission "modifyPrincipals";
    permission javax.security.auth.AuthPermission "getSubject";
};

grant Principal javax.security.auth.x500.X500Principal
    "CN=Webmaster,OU=NC,O=Shanghai University,L=ZB,ST=Shanghai,C=CN" {
    permission java.io.FilePermission "c:\\\\-", "read";
};

```

这里为简单起见,几个 AuthPermission 权限授予所有程序。Principal 使用 J2SDK 提供的 X500Principal,因此后面的名称是 X.500 格式的"CN=..."。这样,用户以"CN=Webmaster,OU=NC,O=Shanghai University,L=ZB,ST=Shanghai,C=CN"身份登录,则将可以读 c:盘所有文件。

本实例使用的策略文件 authKS.policy 如下:

```

simp {
    com.sun.security.auth.module.KeyStoreLoginModule sufficient
        keyStoreURL ="file:C:/java/ch9/Authorization/store1";
};

```

其中使用了 9.3.1 小节的密钥库 store1。

★运行程序

程序运行在 C:\java\ch9\Authorization 目录,其中拷贝了 9.3.1 小节的密钥库 store1,并保存有本节的 authX500.policy、authKS.config、MyactionKS.java 和 LoginDoAsPrvKS.java 程序。

在 9.3.1 小节的密钥库 store1 中,条目 web 对应的是"CN=Webmaster, OU=NC, O=Shanghai University, L=ZB, ST=Shanghai, C=CN",条目 email 对应的是"CN=Email User, OU=ME, O=SouthEast University, L=GL, ST=Nanjing, C=CN"。

输入

```
javac MyactionKS.java
javac LoginDoAsPrvKS.java
```

编译程序。

将如下内容在一行中输入批处理文件 start1.bat,

```
java -Djava.security.manager
-Djava.security.policy==authX500.policy
-Djava.security.auth.login.config==authKS.config
LoginDoAsPrvKS
```

由于使用策略文件, 因此运行时需使用 Java 命令选项-Djava.security.manager, 并使用-Djava.security.policy==authX500.policy 指定策略文件 authX500.policy。此外和本章前面各节一样使用 Java 命令选项-Djava.security.auth.login.config 指定登录配置文件。

其执行过程如下:

```
C:\java\ch9\Authorization>start1

C:\java\ch9\Authorization>java -Djava.security.manager
-Djava.security.policy==authX500.policy
-Djava.security.auth.login.config==authKS.config LoginDoAsPrvKS
请登录 keystore
Keystore 别名: web
Keystore 密码: store1pass
私人关键密码 (可选的): kpass123
0. OK [default]
1. Cancel
Enter a number: 0

Below is result of run MyactionKS
SET BLASTER=A220 I7 D3 H7 P330 T6
SET SBPCI=C:\SBPCI
C:\BAV2000\BAVGUARD

set path=c:\j2sdk1.4.0\bin;%path%
```

可见, 当输入别名 web, 并且输入正确口令登录完成验证过程后, 可以读取 c:\autoexec.bat 文件的内容并显示出来。如果将策略文件中

```
grant Principal javax.security.auth.x500.X500Principal
"CN=Webmaster,OU=NC,O=Shanghai University,L=ZB,ST=Shanghai,C=CN"
```

一项去掉, 则将显示无权访问。此外, 如果以其他用户身份登录, 即使通过了验证, 也将无权访问 c:\autoexec.bat 文件, 如:

```
C:\java\ch9\Authorization>start1

C:\java\ch9\Authorization>java -Djava.security.manager
-Djava.security.policy==authX500.policy
-Djava.security.auth.login.config==authKS.config LoginDoAsPrvKS
```

```

请登录 keystore
Keystore 别名: email
Keystore 密码: storelpass
私人关键密码 (可选的): kpass123
0. OK [default]
1. Cancel
Enter a number: 0
java.security.AccessControlException:                access                denied
(java.io.FilePermission c:\autoexec.bat read)

```

该执行过程中，尽管登录验证成功，但以 email 别名登录时，登录模块加到 Subject 中的 Principal 名称是"CN=Email User, OU=ME, O=SouthEast University, L=GL, ST=Nanjing, C=CN"，在登录配置文件中没有授权该 Principal 读取 c:\autoexec.bat 文件，因此提示：

```
access denied (java.io.FilePermission c:\autoexec.bat read)
```

9.5.2 使用编程方式的基于身份授权

★ 实例说明

9.5.1 小节通过策略文件申明某种身份具有某种权限，本小节则在代码中直接判断已验证的身份是否满足某个条件，只有具备条件才可执行代码。这种授权方式称为编程授权方式 (Programmatic authorization)。

★ 编程思路：

本实例的场景和 9.5.1 小节一样，类 MyactionPro 中有一段代码需要读取 c:\autoexec.bat 文件，但编程者不准备让用户通过策略文件来授权谁可以执行该段代码，而是通过编程者指定的方式，如通过数据库等。

和 9.5.2 小节一样，通过 Subject 对象的静态方法 doAsPrivileged (Subject subject, PrivilegedAction action, AccessControlContext acc)可以将用户登录的身份 subject 和欲执行的代码 action 相关联。

同样，类 MyactionPro 必须实现 PrivilegedAction 接口。该接口中需实现 run()方法，需要执行的代码就放在该方法中。在 9.5.1 小节中在 run()方法中使用输入流读取 c:\autoexec.bat 文件的内容，文件输入流会自动使用安全管理器检查配置文件看有没有为 Subject 中的身份标志 Principal 授权。而在本小节中，则在 run()方法中通过编程方式检测权限，其步骤如下：

获取当前上下文，

- (1) 获取访问控制上下文

```
AccessControlContext context = AccessController.getContext();
```

分析: 使用 AccessController 类的 getContext() 方法获取当前的访问控制上下文，。

- (2) 获取包含用户身份的 Subject 对象

```
Subject subject = Subject.getSubject(context);
```

分析: 使用 Subject 类的 getContext() 方法获取当前访问控制上下文的 Subject 对象。即当前包含用户身份的主体。

- (3) 获取用户身份标志的集合

```
Set principals = subject.getPrincipals();
```

分析: 执行上一步得到的 subject 对象的 getPrincipals() 方法，得到包含当前

登录用户所有身份标志的集合。

(4) 遍历身份标志

```
Iterator iterator = principals.iterator();
while (iterator.hasNext()) {
    Principal principal = (Principal)iterator.next();
```

分析: 执行上一步集合对象的 `iterator()` 方法, 然后通过 `iterator()` 对象的 `next()` 方法遍历集合所有对象, 对每一个对象转换为 `Principal` 类型。

(5) 检测身份标志

```
String nameInSub= principal.getName( );
if (nameInSub.equals( name )) {
    System.out.println("Now run code permitted by "+name );
    // 进一步的操作.....
    return content;
}
}
throw new AccessControlException("Denied in MyactionPro");
```

分析: 对遍历得到的每一个身份标志, 通过 `Principal` 类的 `getName()` 方法获取身份标志的名称, 然后作各种处理, 如可以有一个数据库配置各个身份标志可以做什么, 这样可以使用 `nameInSub` 检索该数据库, 根据检索的值进行相应操作。

这里简单地将得到的名字和字符串 `name` 进行比较, 若当前通过验证的用户的身份标志集合包含 `name` 指定的名称时, 则打印一串信息 “Now run code permitted by ...”, 并可进一步执行其他操作, 如读取文件 `c:\autoexec.bat` 等。若集合中没有 `name` 指定的名称, 则抛出异常对象, 提示 “Denied in MyactionPro”。

调用类 `MyactionPro` 的应用程序和 9.5.1 小节的 `LoginDoAsPrvKS.java` 类似。

★代码与分析:

本实例定义的需要以特定身份执行的代码 `MyactionPro` 完整内容如下:

```
import javax.security.auth.*;
import javax.security.auth.login.*;
import java.security.*;
import java.util.*;
import java.io.*;

class MyactionPro implements PrivilegedAction {
    public Object run() {
        //获取当前登录上下文的主体
        AccessControlContext context = AccessController.getContext();
        Subject subject = Subject.getSubject(context );
        if (subject == null ) {
            throw new AccessControlException("Denied");
        }
        //获取身份标志集合
        Set principals = subject.getPrincipals();
```

```

        Iterator iterator = principals.iterator();
//允许的身份
        String name="CN=Webmaster,OU=NC,O="+
                "Shanghai University,L=ZB,ST=Shanghai,C=CN";
//遍历并检测当前主体的身份标志
        while (iterator.hasNext()) {
            Principal principal = (Principal)iterator.next();
            String nameInSub= principal.getName( );
            if (nameInSub.equals( name )) {
                System.out.println("Now run code permitted by "
                                   +name );
                //以符合条件的身份执行的读取文件的代码
                String content="";
                String s;
                String filename="c:\\autoexec.bat";
                try{
                    BufferedReader in = new BufferedReader(
                        new FileReader(filename));
                    while ((s = in.readLine( )) != null) {
                        content+=s+"\n";
                    }
                }
                catch(IOException e){
                    System.out.println(e);
                }
                return content;
            }
        }
        throw new AccessControlException("Denied in MyactionPro");
    }
}

```

本实例中执行代码MyactionPro的程序LoginDoAsPrvPro.java的完整内容如下:

```

import com.sun.security.auth.callback.TextCallbackHandler;
import javax.security.auth.*;
import javax.security.auth.login.*;
import java.security.*;
import java.util.*;
import java.io.*;
public class LoginDoAsPrvPro{
    public static void main(String[] args) {
        LoginContext c=null;
        //登录
    }
}

```

```

TextCallbackHandler handler=new TextCallbackHandler( );
boolean pass;
try {
    c = new LoginContext("simp",handler);
    c.login();
}
catch (LoginException le) {
    System.out.println("Authentication failed!");
    System.exit(1);
}
//获取Subject.doAsPrivileged( )方法的参数
Subject subj = c.getSubject();
MyactionPro myact=new MyactionPro( );
try {
    //执行代码
    String filecontent=(String) Subject.doAsPrivileged(
                                                subj, myact,null);

    //输出执行结果
    System.out.println("\n Below is result of run MyactionKS");
    System.out.println(filecontent);
} catch (AccessControlException e) {
    System.out.println(e);
}
}
}

```

该代码和 9.5.1 小节的 LoginDoAsPrvPro.java 类似，只是执行 doAsPrivileged() 时传入的是本小节的 MyactionPro 对象。

登录配置文件仍旧使用 9.5.1 小节的 authKS.config 文件。

★运行程序

程序运行在 C:\java\ch9\Authorization 目录，其中拷贝了 9.3.1 小节的密钥库 store1，并有上一小节编写的 authKS.config 登录配置文件，同时保存有本节的 MyactionPro.java 和 LoginDoAsPrvPro.java 程序。

由于本实例不使用策略文件进行授权，而使用编程方式进行授权，因此本实例不需要策略文件，运行时也不需要 Java 命令选项 -Djava.security.manager 和 -Djava.security.policy，只需要 -Djava.security.auth.login.config 指定登录配置文件即可。

其运行过程如下：

```

Java -Djava.security.auth.login.config==authKS.config LoginDoAsPrvPro
请登录 keystore
Keystore 别名: web
Keystore 密码: store1pass
私人关键密码（可选的): kpass123

```



```

0. OK [default]
1. Cancel
Enter a number: 0
Now      run      code      permitted      by      CN=Webmaster,OU=NC,O=Shanghai
University,L=ZB,ST=Shanghai,C=CN

```

```

Below is result of run MyactionKS
SET BLASTER=A220 I7 D3 H7 P330 T6
SET SBPCI=C:\SBPCI
C:\BAV2000\BAVGUARD

```

```
set path=c:\j2sdk1.4.0\bin;%path%
```

如果和 9.5.1 小节一样以 email 别名登录，则会显示

```
java.security.AccessControlException: Denied in MyactionPro
```

9.5.3 比较 doAsPrivileged() 和 doAs()

★ 实例说明

9.5.1 和 9.5.2 小节都是使用 Subject.doAsPrivileged() 方法来将某个 Subject 和要执行的代码相关联，本节介绍类 Subject 另一个静态方法 doAs()。

★ 编程思路：

使用 WinZip 软件打开 c:\j2sdk1.4.0\src.zip 文件查看 J2SDK 的源代码，打开其中的 Subject.java 文件，可以看到 Subject.doAsPrivileged() 方法和 Subject.doAs() 的区别仅在于获取访问控制上下文的方式不同。

Subject 类的源代码中，方法

```
public static Object doAs(final Subject subject,
    final java.security.PrivilegedAction action)
```

定义如下：

```

{
    java.lang.SecurityManager sm = System.getSecurityManager();
    if (sm != null) {
        sm.checkPermission(new AuthPermission("doAs"));
    }
    if (action == null)
        throw new NullPointerException
            (ResourcesMgr.getString("invalid null action provided"));

    // set up the new Subject-based AccessControlContext
    // for doPrivileged
    final AccessControlContext currentAcc = AccessController.getContext();
    // call doPrivileged and push this new context on the stack

```

```

        return java.security.AccessController.doPrivileged
            (action, createContext(subject, currentAcc));
    }

```

类 Subject 的源代码中，方法

```

public static Object doAsPrivileged(final Subject subject,
    final java.security.PrivilegedAction action,
    final java.security.AccessControlContext acc)

```

的定义中，只是将方法 doAs() 获取访问控制上下文的一句：

```

    final AccessControlContext currentAcc = AccessController.getContext( );

```

替换成了

```

    final AccessControlContext callerAcc =
        (acc == null ?
        new AccessControlContext(new ProtectionDomain[0]) :
        acc);

```

从以上可见，方法 doAs() 是通过 AccessController.getContext() 方法获取当前正在执行 doAs() 时的访问访问控制上下文。该访问控制上下文是在程序刚开始执行时就自动创建的，包含了 AccessControlContext 对象创建后到执行该步所经过的所有代码信息（包括代码从策略文件中所获得的权限、代码位置等信息）。

而 doAsPrivileged() 方法则检测 acc 是否为 null，若不为 null，则使用的是参数中传入的访问控制上下文 acc。若为 null（9.5.1 和 9.5.2 都传入了 null），则使用 new 操作符创建新的 AccessControlContext 对象。此时的 AccessControlContext 对象中并不包含以前的代码信息，只包含该步以后所经过的所有代码信息。

当 doAs() 或 doAsPrivileged() 方法中执行的代码（通过 action 参数传入）中通过安全管理器检测是否有某种权限时，只有登录上下文中所有代码都具有该权限，检测才可通过。

在 9.5.1 小节的程序 LoginDoAsPrvKS.java 和 9.5.2 小节的程序 LoginDoAsPrvPro.java 中，已经使用 doAsPrivileged() 方法分别执行了基于策略文件进行授权的代码（MyactionKS）和基于编程方式进行授权的代码（MyactionPro）。

本小节则使用 doAs() 方法分别执行 MyactionPro 和 MyactionKS，并对这几种组合进行比较。

代码 LoginDoAsPro.java 使用 doAs() 方法执行代码 MyactionPro，和 9.5.2 小节的程序相比，其主要变化如下：

```

MyactionPro myact=new MyactionPro( );
String filecontent=(String) Subject.doAs(subj, myact);

```

代码 LoginDoAsKS.java 使用 doAs() 方法执行代码 MyactionKS，和 9.5.1 小节的程序相比，其主要变化如下：

```

MyactionKS myact=new MyactionKS( );
String filecontent=(String) Subject.doAs(subj, myact);

```

★代码与分析:

本实例定义的代码 LoginDoAsPro.java 完整内容如下:

```
import com.sun.security.auth.callback.TextCallbackHandler;
import javax.security.auth.*;
import javax.security.auth.login.*;
import java.security.*;
import java.util.*;
import java.io.*;

public class LoginDoAsPro{
    public static void main(String[] args) {
        LoginContext c=null;
        //登录
        TextCallbackHandler handler=new TextCallbackHandler( );
        boolean pass;
        try {
            c = new LoginContext("simp",handler);
            c.login();
        }
        catch (LoginException le) {
            System.out.println("Authentication failed!");
            System.exit(1);
        }
        Subject subj = c.getSubject();
        MyactionPro myact=new MyactionPro( );
        try {
            String filecontent=(String) Subject.doAs(subj, myact);
            System.out.println("\n Below is result of run MyactionKS");
            System.out.println(filecontent);
        } catch (AccessControlException e) {
            System.out.println(e);
        }
    }
}
```

本实例定义的代码 LoginDoAsKSo.java 完整内容如下:

```
import com.sun.security.auth.callback.TextCallbackHandler;
import javax.security.auth.*;
import javax.security.auth.login.*;
import java.security.*;
import java.util.*;
import java.io.*;

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

```

LoginContext c=null;
//登录
TextCallbackHandler handler=new TextCallbackHandler( );
boolean pass;
try {
    c = new LoginContext("simp",handler);
    c.login();
}
catch (LoginException le) {
    System.out.println("Authentication failed!");
    System.exit(1);
}
Subject subj = c.getSubject();
MyactionKS myact=new MyactionKS( );
try {
    String filecontent=(String) Subject.doAs(subj, myact);
    System.out.println("\n Below is result of run MyactionKS");
    System.out.println(filecontent);
} catch (AccessControlException e) {
    System.out.println(e);
}
}
}

```

除了使用9.5.1和9.5.2小节的安全策略文件、登录配置文件外，本实例另外增加了一个安全策略文件authX500.jar.policy，其完整内容如下：

```

grant {
    permission javax.security.auth.AuthPermission "createLoginContext";
    permission javax.security.auth.AuthPermission "doAs";
    permission javax.security.auth.AuthPermission "doAsPrivileged";
    permission javax.security.auth.AuthPermission "modifyPrincipals";
    permission javax.security.auth.AuthPermission "getSubject";
};

grant Principal javax.security.auth.x500.X500Principal
    "CN=Webmaster,OU=NC,O=Shanghai
    University,L=ZB,ST=Shanghai,C=CN" {
    permission java.io.FilePermission "c:\\\\-", "read";
};

grant codebase "file:./My.jar"{
    permission java.io.FilePermission "c:\\\\-", "read";
};

```

该安全策略文件的详细分析见本实例的“运行程序”部分。

★运行程序

程序运行在 C:\java\ch9\Authorization 目录,其中包含了 9.5.1 和 9.5.2 小节所使用的密钥库 store1、登录配置文件 authKS.config 等。

输入

```
Java -Djava.security.auth.login.config==authKS.config LoginDoAsPro
运行程序,它以 doAs( )方法执行 MyactionPro 代码。通过 web 别名正确登录后可以正常显示 c:\autoexec.bat 文件:
```

```
请登录 keystore
Keystore 别名: web
Keystore 密码: store1pass
私人关键密码 (可选的): kpass123
0. OK [default]
1. Cancel
Enter a number: 0
Now run code permitted by CN=Webmaster,OU=NC,O=Shanghai
University,L=ZB,ST=Shanghai,C=CN

Below is result of run MyactionKS
SET BLASTER=A220 I7 D3 H7 P330 T6
.....
```

可见它和 9.5.2 小节的运行结果没有什么不同,这是因为本节的 LoginDoAsPro.java 和 9.5.2 小节的 LoginDoAsPrvPro.java 都不是通过策略文件和安全管理器来进行授权,因此尽管两个程序中执行 MyactionPro 时所使用的访问控制上下文不同,但并不影响权限的检测。

下面运行 LoginDoAsKS 程序,它以 doAs()方法运行 MyactionKS 代码。将下面的命令输入批处理文件 startAs.bat:

```
java -Djava.security.manager -Djava.security.policy==authX500.policy
-Djava.security.auth.login.config==authKS.config LoginDoAsKS
输入 startAs 运行程序,程序运行过程如下:
```

```
请登录 keystore
Keystore 别名: web
Keystore 密码: store1pass
私人关键密码 (可选的): kpass123
0. OK [default]
1. Cancel
Enter a number: 0
java.security.AccessControlException: access denied
(java.io.FilePermission c:\autoexec.bat read)
```

可见,9.5.1 小节 LoginDoAsPrvKS.java 程序使用 doAsPrivileged()方法执行 MyactionKS

可以正常读取 c:\autoexec.bat 文件，而本小节的 LoginDoAsKS.java 使用 doAs()方法时，则显示没有权限。

这是因为，本小节的 LoginDoAsKS.java 使用 doAs()方法时，访问控制上下文是在程序刚开始执行时就自动创建的，包含了 AccessControlContext 对象创建后到执行该步所经过的所有代码（包括 LoginDoAsKS 和 MyactionKS 等）是否有权限等的信息。而基于策略文件验证时必须这些代码都具有检测的权限，检测才能通过。由于策略文件 authX500.policy 中只对 Principal 为"CN=Webmaster,OU=NC,O=Shanghai University,L=ZB,ST=Shanghai,C=CN"的代码授予了 C:盘所有文件的读的权限，而在刚执行 LoginDoAsKS 代码时用户尚未登录，因此 LoginDoAsKS 代码没有获得读取文件的权限，最终程序抛出访问控制异常。

而 9.5.1 小节的 LoginDoAsPrvKS.java 程序使用 doAsPrivileged()方法执行 MyactionKS 则没有问题，这是因为 9.5.1 小节中向 doAsPrivileged()方法的第三个参数传入了 null，根据本小节编程思路中的分析，其访问控制上下文中只包含了 doAsPrivileged()中创建新登录上下文以后所经过的代码（如 MyactionKS 等），不包括代码 LoginDoAsKS。

因此，对于本小节的 LoginDoAsKS，必须对该策略文件也授权读取 c:\autoexec.bat 文件，才能正常运行。

为了单独对 LoginDoAsKS.class 进行授权，可将其打包到 jar 文件，输入：

```
jar cvf My.jar LoginDoAsKS.class
```

则将代码 LoginDoAsKS.class（注意大小写）打包到 My.jar 文件，在策略文件 authX500.policy 中加入一段针对 My.jar 文件的授权。

```
grant codebase "file:./My.jar"{  
    permission java.io.FilePermission "c:\\-", "read";  
};
```

不妨将新的策略文件以文件名 authX500jar.policy 另存，然后将下面的命令输入批处理文件 startAs2.bat：

```
java -classpath My.jar;. -Djava.security.manager  
-Djava.security.policy==authX500jar.policy  
-Djava.security.auth.login.config==authKS.config LoginDoAsKS
```

输入 startAs2 运行程序，则输入别名“web”、Keystore 密码“store1pass”和私钥保护密码“kpass123”后可正确显示 c:\autoexec.bat 文件的内容。

本章介绍了 JAAS，它使用可配置的登录模块实现了应用程序和验证机制的分离。应用程序编写者可以不管底层验证机制，而通过统一的方式通过执行登录上下文的 login()方法要求用户登录。登录模块和用户界面都可以由其他编程者完成。在此基础上，程序使用者可以按照用户的身份进行授权。

本章的基于身份的授权和第 8 章的基于代码来源的授权可以相结合，给用户多种控制方式。