China-pub.com
下载

第1章　计算机与网络安全基础

确保计算机安全的三条黄金法则：不要拥有计算机；不要打开计算机；不要使用计算机。

——Robert(Bob)T. Morris

安全，就是确保不发生有危害的事情。这个简洁的阐述很容易使人迷惑。实际上，安全有非常复杂的解释，对它们进行剖析有助于理解安全的真正含义。

通常情况下，某些经验性法比较适用于安全这个概念。首先，安全总是和效用有关。要想不发生危险，最好就是什么事都不做。打个比方，存放在车库里的车不可能发生交通事故，但是，汽车不跑路并不是我们的意图。我们真正的目的是保障有利的事情发生，而避免有危害的事情。

其次，安全与危险是共存的。例如，前门加锁的有效性很大程度上取决于你所要防卫的窃贼类型。对于小贼，有一定的防卫意义，而对工具齐全、熟于此道的老贼则是毫无意义的。

第三，必须从整个系统的角度去考虑安全问题。系统的安全程度由系统的最薄弱环节决定。也就是说，只保证前门安全是不够的，狡猾的小偷会从所有可能疏于防备的地方潜入房子，尤其是远离装有牢固的锁的那些地方。

第四，实施安全的措施必须简单。试想，如果你每次进门需花 30分钟的艰辛去打开一个复杂的锁，你也许会不锁门。

第五，安全的实现必须注意性能价格比。例如，如果你加锁的费用比所防护的内容的价值还昂贵，那是毫无意义的。由于每个人对价值的认定是不同的，事情就变得很复杂。

最后一点但也是很重要的一点就是，安全措施必须尽可能简单。正如专家所说，越复杂的系统就容易出错。使安全系统既简单又可靠是我们的宗旨。

通过对本书的学习，你将会体会到这些符合计算机安全的经验法则。

1.1　密码学与计算机安全

在讨论特定话题之前，应先明确密码学(cryptography)与计算机安全(computer security)这两个课题间的差别。密码学是一种以秘密的方式编码信息，使只有特定的接收者才可以访问被编码的信息的方法。经过一段很长的时间，密码术的应用有了很大的发展，从古老的凯撒密码到第二次世界大战时广泛应用的密码机，一直发展到用计算机软硬件实现的现代密码系统。

直到1960年，分时用户计算机操作系统，如剑桥大学早期的计算机系统 [80]和麻省理工大学的Multics系统[69]初次问世，计算机安全才首次成为研究热点。但是自那以后，除了 70年代中期，计算机安全技术一直未受到重视 [3、32、36、75]，而且在这段时期内，计算机安全大部分是基于军事上的需求。到了 90年代， Internet和电子商务得到广泛应用，尤其是 Java技术的发展才使计算机安全在商业上广为应用并成为主流技术。

安全机制常常得益于密码学的应用，如基于网络的用户登录协议。不过，它也并不是必须依赖于密码学的应用，例如UNIX系统中对文件的访问控制。

反过来，密码学也依赖于系统的安全性。密码算法常常用软件或硬件实现，它们能否正

China-pub.com 下载

常运作关键取决于是否有一个安全的系统。比如，如果系统缺乏访问控制的安全性，攻击者可以修改密码系统的软件算法。可见，安全性的缺乏会直接影响密码术的效用。

1.2 危险和防护

在有关计算机安全的文献中，危险或攻击通常分为三类：

(1) 秘密攻击(secrecy attack)。攻击者试图窃取口令、医疗记录、电子邮件 (E-mail)日志、薪水册等机密数据。其攻击手段很多，如贿赂安全卫士、寻求系统的安全漏洞和密码算法的不足之处等。

(2) 完备性攻击(integrity attack)。攻击者企图非法更改部分系统。例如，某位银行雇员为了把顾客的资金存入自己的个人帐户，修改帐户系统从而危及整个交易系统安全性 [61]。又如，某位大学生侵入学校管理系统提高他的考试分数，导致危及整个数据安全。这类攻击者常常会为了隐藏“脚印”而删除系统日志。

(3) 可得性攻击(availability attack)。攻击者企图中断系统正常运作，通常也称这种攻击为拒绝服务攻击(denial-of-service attack)。如用大量的IP包“轰炸”机器，使机器与网络的其余部分隔离。这类攻击者通过破坏计算机控制系统引起通信流量阻塞。

这三类攻击是息息相关的。也就是说，某一类攻击的技术和后果经常作为另一类攻击的辅助手段。比如，一个攻击者通过秘密攻击获知口令，这样就有权访问这个系统，然后修改系统资源，最终完成拒绝服务攻击。当遭受攻击时，系统会出错，但大多数系统由于缺乏安全机制仍认为是安全的，例如，有时系统冲突导致在公开的可读目录下倾倒机密的内核信息。⊖

同样地，这些攻击的防护机制之间也是紧密相关的。一般来讲，防护有一种或多种目的：防止攻击、检测是否遭受攻击或恢复系统。正如本章下面讲解的，一种防护机制并不是万能的。

为了保护秘密数据，可以把数据存储到不易让攻击者发现的地方，或者是安装严密的访问控制程序拒绝非法访问；也可以用加密技术加密数据使攻击者不能访问到真正数据，除非他能破译加密系统或窃取到密钥，通常破译加密系统是极其困难的。当然，可以同时采用多种防护机制。应注意到：对于秘密数据，大多数重要技术是防止遭受攻击，因为检测数据是否丢失非常困难，而恢复数据几乎是不可能的。

为了保护数据的完备性，可以再次利用前面讲述的任何一种或所有机制。对于数据的完备性，检测是否遭受攻击较为简单，恢复数据也是可能的。比如，对于文件 x，可以用著名的单向散列函数f计算其散列值，并分开存储 $f(x)$，当x被修改为x'时，可以计算$f(x)$值，与$f(x')$作比较，如果两者不一致，就表明数据的完备性已被破坏。

当然如果对应的$f(x)$也被破坏，检测也许是不可能的。如果存储 $f(x)$的地方不够安全，可以利用带有密钥的单向散列函数，并把 $f(x,k)$与x存储在一起。如果k的保密性很好，攻击者很难于修改x和散列值[22, 52]。

为了数据的完备性被破坏后，能够恢复数据，可以提前备份数据并在安全的地方存储备份数据[61]。在安全的网络中，也可以用更复杂的分布式计算技术来备份数据 [34、64、73、77]。

防护可得性攻击更为复杂。这是因为它不能应用通常的防护和检测技术。这里，计算机

⊖ 当然，可以从其他角度理解攻击。例如，对于个人资料 (生日和美国社会安全号)不定期、非法收集以及分布的保密性受到公众的普遍关注。

China-pub.com
下载

安全符合容错域计算。一些有益的研究成果在相关研究领域 (有时被称作可依赖系统(dependable system))可以查到。为了进一步理解这方面的技术，可以查询 [12、24、65、73]中的论文和所引用的文献。

1.3 外围防护

由于有许多潜在的薄弱环节和无穷尽的攻击，而每一种攻击又可能包含多种攻击技术。确保整个系统安全很困难，尤其是系统包括多个通过网络连接的主机时，更令人望而却步。由于系统的安全性是由它的最薄弱环节决定，因此，安全范围必须是整个系统性的。例如，在某个大企业的内部网络中，含有许多不同品牌和型号的机器，而这些机器的操作系统和应用程序又是千差万别。它们再通过不同供应商提供的不同性能和容量的网络设备和路由器连接起来。在这样不同种类、不断变化的环境中，检测整个系统和确保所有部件的安全是相当费时和复杂的。

面对这样一个混杂的场景，公司内部可以从心理和生理上比较容易地找到各部件。通常将它们划分为两大阵营Us和Them，Us包括公司拥有的、被操作的、被相关企业信任的机器，而Them则包括所有其他不友善的或不被信任的机器。一旦界限被化定，问题就在于如何使 Them在安全范围之外，而Us在安全范围之内。这种情况就被称为外围防护(perimeter defense)。

构筑外围防护的一种简便方法是不要把 Us和Them连接起来。的确，一些军事和商业实体拥有自己的内部网络，它们并不和更广域的网络 Internet相连接。为了与外部连接，他们或许有一些独立的终端或机器，但是这些特殊用途的机器通常被守护并将之与内部网络隔离。

如果整个系统内部包含有一些分布在不同地理位置的机器，通过租赁线或共享网络可以连接各站点从而组成专用网络。如果各站点间必须通过网络相互通信，在每两个不同站点之间必须运用加密技术去组成虚拟专用网 (VPN)。在图1-1的虚拟模型中，四所院校通过 Internet连接，三个站点(MIT、UTAustin和UCLA)形成(VPN)，就可以使它们之间的信息流量自动地与Stanford的窃听行为相隔绝。

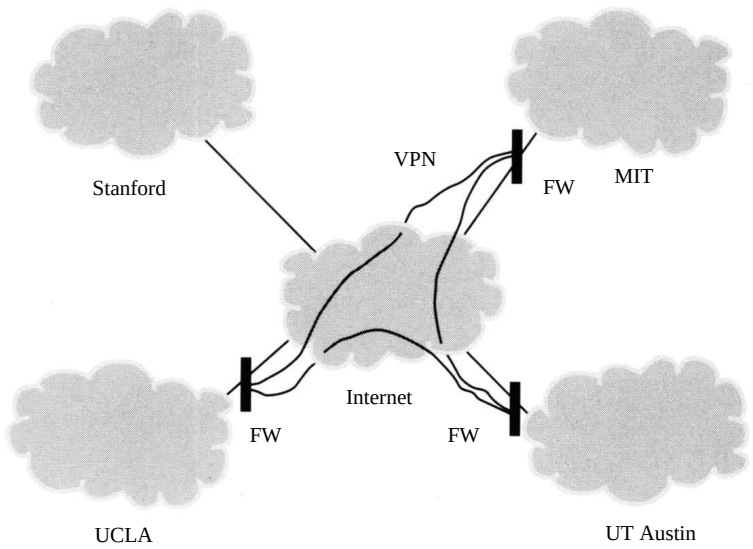


图1-1 虚拟专用网

China-pub.com 下载

但是，这样与外界隔离的方法并不理想。例如，随着人们通过 Internet 与外界交流需求的不断增加，E-mail已经成为Internet上的“杀手”。万维网(WWW)使Internet 更为流行，漫游网络寻求有用信息对提高生产率大有裨益(当然指的是正当的、合法的)，这些趋势正驱使先前封闭的企事业单位有选择地开放它们的边界控制。在构筑更为有用的外围防护方面，防火墙(fire wall)扮演了一个重要角色。

1.3.1 防火墙

各种防火墙的形式和尺寸千差万别[6]。但总的来说，防火墙是位于专用网络与公共网络之间的一种机器，它对网络流量起着筛选的作用，符合安全策略的特定通信量在某一方向上可以通过。安全策略可以非常简单也可以非常复杂，这通常取决于信源地和信宿地、采用的通信协议和涉及的应用程序等因素。防火墙还可以为信息分配路由，甚至在让信息通过之前对其进行某种操作。作为代理服务器，它还能加密信息。有加密技术的防火墙可以被用来构成 VPN。

用防火墙作为外围防护是一种有效的安全方案。防火墙作为中心控制点，易于实现和更新公司的安全策略。但是，也存在一些不足之处：第一，防火墙不能滤除和阻止所有的网络灾难。像HTTP协议这样的信息可以巧妙地通过防火墙。通常，防火墙与移动代码作用是相反的。因为前者试图阻塞或减少流入信息，包括那些后者想获得的信息。其次，防火墙目前已经成为大企业通信的瓶颈。再者，当把防火墙作为代理服务器时，桌面上许多应用程序必须重新编写。不过，对于新的应用程序，这个问题并不重要，这主要因为应用程序内嵌有代理支持。

1.3.2 仅仅使用外围防护的不足之处

作为一套安全方案，只用外围防护是不够的。这主要有以下几方面的原因：

定位和保护所有外围站点是十分困难的，比如，直接用电话线建立连接能有效穿透外围防护[61]。此外，由于企业允许雇员远程或在家中工作，必须严密监视和充分防护内部网络的远距离站点，这常常是不切实际的。

即使在企业内部，由于每件事或每个人并不是完全信任的，就需要对其加以控制。危害性很大的攻击通常发生于内部。由于内部入侵者与外部入侵者相比，有更为有利的条件，所以造成的损失更大。比如，会计部门必须加以防护，使只有被批准的雇员方可发出买单，而专利部门必须隔离开来防止信息泄露给竞争对手。

本章的其余部分回顾了在组织边界的外围和交界处大有用途的安全模型与技术。

1.4 访问控制与安全模型

安全模型(security model)是人们对访问被保护数据加以控制的方法一种抽象。像防火墙一样，安全模型也有多种形式和尺寸，对于不同的应用程序和应用环境，安全模型的要求有很大不同。安全模型分类方法很多，主要包括：

- MAC和DAC模型。
- 数据和信息安全模型。
- 静态(static)和动态(dynamic)模型。

China-pub.com
下载

1.4.1 MAC 和DAC模型

一类称为强制性访问控制(Mandatory Access Control，MAC)。在MAC模型中，系统内的实体可以是主题(大致相当于用户、过程、机器等概念)或者对象(大致与控制目标概念相同，例如文件、数据记录)。每一个实体属于一个秘密级别。这样的等级形成一个具有支配关系的框架。例如，“非密”、“保密”、“秘密”、“最高机密”就构成图1-2所示的支配关系。

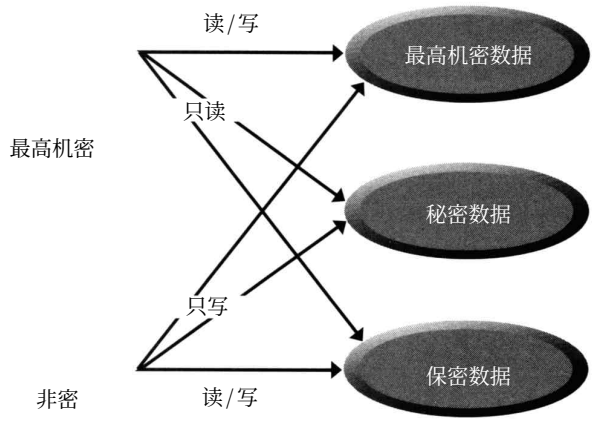


图1-2 MAC 安全模型

Bell和LaPadula[3]的书中示范了满足多种安全级别需要的MAC 模型，它是Multics系统[69]安全的数学模型。在Bell-LaPdula模型中，仅当主题对某一对象具有支配权时，它对该对象享有读许可权；仅当主题被某一对象所支配时，它对该对象享有写许可权。这被称为读下 (read-down)和写下(write-down)，更准确一点可称为不读上(no read-up)和不写下(no write-up)。应注意到，在这个模型中，只有当两个实体处在同一级别上时，才可以进行双向通信或通过被信任中间体进行通信。

非Mac模型被叫做自由访问控制模型(Discretionary Access Control，DAC)。UNIX安全模型类似于DAC模型，文件的所有者通过设置文件许可权位决定谁有权对其进行访问。但在UNIX系统中读文件的人可以复制文件并供其他人阅读，而MAC模型不允许这样的自由决定。

1.4.2 对数据和信息的访问

到目前为止，对访问控制的讨论集中在指定对数据的明确访问的模型上，例如，直接读取存储在文件系统中的内容。然而，信息可以隐含地被传送，特别地，通信双方可以通过所谓的隐藏信道(covert channel，它于公开信道(overt channel)是相对的)进行通信。比如，双方共享同一磁盘分区，一方占用了整个分区空间，当另一方由于缺乏可用空间而导致建立新文件失败时，会发现分区空间已经被占用。填充或未填充磁盘空间，通过发送一个“1”或“0”告之对方。

在Lampson关于限制问题的论文[43]中首先研究了这种情况。Lampson论述了对一个应用程序加以限制的困难，使应用程序不能直接或通过发送信息影响外部世界。

这种攻击类型有多大效果与对敌人渗入的担心程度和内部攻击引起的严重损失是密切相关的。内部入侵的模式源远流长，“特洛伊木马”(从特洛伊的灭亡故事中衍生而来)在计算机安全领域指的是为破坏系统而植入计算机的任何程序(重要的计算机安全会议IEEE Symposium

China-pub.com 下载

on Security and Privacy已经采用“特洛伊木马”作为会议记录的封面)。最近，美国几位高级政府官员和雇员泄露国家机密给外国机构而构成犯罪的案件就采用了这种渗入方式。

关于限制问题的早期研究成果是基于信息流而非数据访问的安全模型。在这个领域，由Goguen和Meseguer提出的模型[18、19]是大多数理论工作的基础。同时，对实际系统中的隐藏信道也作了研究工作。例如，DEC的一个研究小组构造了一个通信双方共享同一磁盘的例子，一方可以有选择地读取一个文件或另一个文件，这时，如果另一方试图读第三个文件，会引起可检测到的延迟。这个延迟是由磁盘臂的运动造成的。两个不同的延迟值可分别表示为“1”、“0”。这些实际研究出的延迟值在很大程度上决定着隐藏信道的容量、可用性和危害。例如，磁盘臂隐藏信道比填充磁盘分区隐藏信道的速度快许多。

注意到隐藏信道的实际意义是难以衡量的。首先，噪声总是存在的。比如说，在磁盘臂隐藏信道中，第三方独立访问磁盘上的各种文件，会明显压缩隐藏信道的带宽。不过，对于非常机密的材料，如密钥，一个低速隐藏信道易于泄露这些秘密。第二，当一方植入特洛伊木马程序时可以利用隐藏信道。当这样的渗透发生时，利用其他通信形式就成为可能。

此外，对隐藏信道的防护仅在系统内部有效。例如，虽然不允许内部人员把信号传送到外界，但并不能防止内部人员的窃密行为。不过，一些组织，尤其是在美国政府中，比如海军研究实验室的研究人员正在开发一种称为Pump的系统，目的在于通过隐藏信道发送信息不泄漏或只有有限泄漏。JDK1.2并没有全面强调隐藏信道。

1.4.3 静态和动态模型

乍一看来，安全策略似乎是静态的，比如，一位雇员可以读或者不可以读文件A，此外没有第三种选择。实际上，安全策略是动态的，是随时间变化的。当某位雇员在公司内部调换部门时，可以获得先前并不具有的对某一文件的访问许可权。在MAC模型中，数据的机密级别和人员的许可权也是可以改变的。数据的机密级别可以降级和升级，同样人员的许可权也可以降级和升级。

下面几个著名的安全模型证明了这种动态机制。第一种称为高水印模型[44]，其中的数据机密级别随人员许可权而变化。

第二种是中国墙模型[8]。该模型实用性较强，尤其是咨询公司和金融机构可以利用它来避免利益冲突。例如，石油业的顾问可以向公司A或B进行咨询，A、B均为他的客户。这样，这位咨询人员有权访问A或B的资料，但是一旦他访问了A的资料，由于利益冲突，对B的资料的访问将会被拒绝。中国墙模型正是试图表示这种真实的策略。

第三种动态模型源于金融业的Clark Wilson一体化模型[11]，它可以满足金融交易的安全需要。比如，对某种限量货币进行交易，必须要有两位不同的人同时签名，并且签名顺序必须严格。这种模型是最先被广泛采用的安全模型，它明确表明金融业安全需求已经超过军事和政府机构的需求。这类模型主要属于MAC模型。

1.4.4 关于使用安全模型的几点考虑

安全模型用途很广，如驱动和分析计算机系统的设计，或形成系统操作的基础，但是它在使用中会产生许多有趣的问题，对这些安全问题目前已有一定程度的研究。

首先是所用模型的可决定性(decidablity)。即当给定一个实际系统的安全模型或对系统的

China-pub.com
下载

安全性有特殊要求和特殊条件 (如某位雇员不允许访问文件 A)时，能够根据实际运行情况分析系统是否安全。在通常情况下，系统是否安全是不可判决的 (参见文献[32])。后来，为了解决这个问题，对模型作了一些限制，问题就可以解决了。在许多模型中，为判决系统是否安全的计算复杂度仍属于Np-难题[72]。

第二是模型的不可能性或不可实现性。一个实际系统是相当庞大的，把它作为一个整体来规定和划分，这就是可组成性 (composability)。满足一套安全特性的各种部件以某种特定方式互相连接，那么由这些部件所构成的整个系统就会自动符合另一套安全特性 [48、57]。在不远的将来，就可以组建一个安全的、具有可组成性的系统。

第三，安全机制必须改进传统系统，至少安全地与传统系统连接起来，这就要求传统系统必须具有安全互用性。安全互用性就是每一个传统系统的安全特性在它最初的定义下必须是可保持的。不过，即便在非常简单的模型中判定一个特殊互用性是否安全通常也是一个 Np-难题[27]。

最后，安全并不仅仅意味着机密。系统的完备性也是至关重要的。一个早期的完备模型 [44]是Bell-LaPadula机密模型的双倍。有些人也把完备性看作可靠性和准确性的一方面，这样在容错域可以得到相关的解决手段。

1.5 密码系统的使用

密码系统就是信息的编码和解码。密码分析 (cryptanalysis)是密码学的反过程，是一种解码或在不获知密钥的条件下对编码信息的破译方法。密码学(cryptology)指的是整个密码研究领域。

安全性和密码术是两个彼此相关但又互有差别的领域。许多人经常混淆这两者。在某种意义上它们是正交的，每一方都有自己独立的用途。虽然它们是互相渗透的。比如，到目前为止讨论的所有安全模型都不必使用密码术。密码技术可以增强保密性和完备性，与计算机安全相比它是一种抽象研究领域。不过，现代密码术大部分存在于计算机和通信系统中，访问控制在保护对密钥的访问时非常有用。实际上，入侵一个密码系统就是设法损害密钥存储的设备。

最常使用的密码概念包括：

- 单向散列函数(one-way hash function) 。
- 对称密码(symmetric ciphers)。
- 非对称密码(asymmetric ciphers)。

下面三个小节对它们进行依次讲解。

应注意到：在密码术中，根据Claude Shannon理论，几乎没有一个密码系统是绝对安全的，在某种意义上，只要攻击者有足够的知识水平和计算能力就可以破译密码系统。唯一例外的是一次一密乱码本(one-time pad)系统，它与明文的长度一样而且只被使用一次。在收发双方有以安全方式互换密钥时，该系统是可行的。

目前，关于这一个论题讲解最深入的参考书目是《Handbook of Applied Cryptography》[51]，对于那些不想深入研究密码术背景和相关论题的人，《Applied Cryptography》 [74] (由机械工业出版社引进出版，中译本名为《应用密码学》)一书更为合适。

1.5.1 单向散列函数

单向散列函数是一种重要的构造块，它有助于获得数据的完备性。在存储和转换过程中

China-pub.com 下载

这些函数可以用来保护数据。

根据Knuth[38]所述，散列的思想是由IBM 的两个开发小组人员首先提出的。我所能检索到的关于单向函数的概念最早是在1968年由Wilkes[80]提出的，他论述了Needham⊖ 提出的剑桥分时计算机系统中所用的单向函数。

单向散列函数的概念可以追溯到许多年以前，众多研究者 Merkle[52]、Naor和Yung[58]、Damgs ard[13]都提出一些定义方法。Meyer和Schilling[54]、Merkle[53]、Robin[64]、Rivest[67]和其他研究者还提出了一些实用的设计方法。

单向散列函数引入了许多术语，一些是可以相互替代的，另一些是为了强调不同的假设前提，如单向散列函数、自由碰撞函数、手印函数、修正检测码和消息鉴别码。

单向散列函数是正向计算简单、反向计算，而且很难找到两个不同的输入值对应于同一个输出值的一种函数。这些特性允许下面的完备性防护。假定你在磁盘上存储文件，并假想它不会被篡改，把文件内容作为函数输入，可以计算散列函数值，它通常比文件长度短许多。以后，可以把文件的当前内容作为输入计算其散列函数值，如果新旧函数值不一致，就表明文件很可能已经被篡改。在这种意义下，单向散列函数在文件内容与函数值之间充当一种非假造连接。见图1-3单向散列函数示意图。

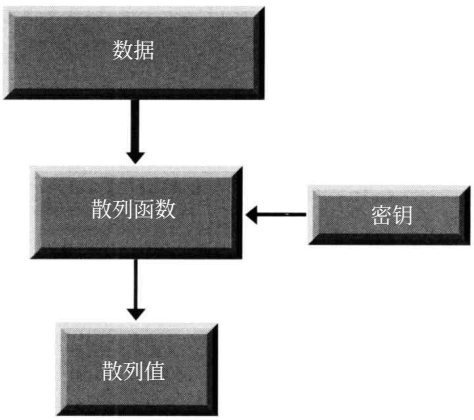


图1-3 单向散列函数

通常，可以将密钥与单向散列函数的输入组合在一起，这样，在未获悉密钥的情况下，不能正确计算或预测散列值。所以说，单向散列函数就不仅是文件内容与散列值之间的一种不可伪造链，而且还是密钥与散列函数值间的一个不可伪造链。

1.5.2 对称密码

对称密码就是在密钥的作用下，将输入(明文(plaintext))转换成输出(密文(ciphertext))，使只有拥有密钥的实体可以将密文转换成明文(密码分析学除外)(见图1-4)。

对称密码有很长的历史，在早期的凯撒系统[39]中首次被采用，从那以后，它们被广泛应用，例如，数据加密标准(DES)[62]和国际数据加密算法(IDEA)中都采用了对称密码。

⊖ Roger Needham 后来回忆说，这种思想是1967年在剑桥的鹰酒吧首次提出，但由于酒吧里人来人往，没有人能记起是谁提出的。

China-pub.com
下载

对称密码也称为秘密密钥密码(secret-key cipher)。在使用对称密码的系统中，通信双方必须共享同一密钥，而且，不同的通信双方间的密钥又必须不同。这必然在密钥管理和分配方面引起许多问题。当通信群体人数增加时，在理论上，所需的密钥是成指数增长的。

对称密码可以以不同模式被操作(如各种反馈模式)，也可以堆叠在一起改善整个系统的加密程度(如三重DES)。

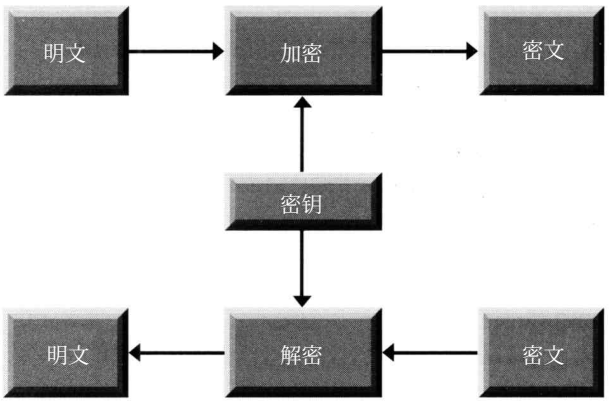


图1-4 对称密码

1.5.3 非对称密码

非对称密码类似与对称密码，不过，它必须采用一对密钥而不是一个密钥。其中一个密钥叫公钥(public key)，用来加密明文，另一个密钥叫私钥(private key)，用来破译密文。在给定私钥条件下，很容易推断出公钥，反之很难。这种性质使人们可以通过公共信道互换公钥的方法进行私人通信。在对称密码中，人们必须通过专有信道共享密钥。著名的非对称密码系统有Diffie-Hellman[14]和RSA[68]，它们通常可用来为对称密码系统加密和互换密钥。

非对称密码的另一个显著特性是：加密和解密是可逆的(reversible)。也就是说，一个人可以用私钥进行解密操作把明文变成密文，也可以用公钥进行加密操作把密文变成明文。在这种情况下，公钥是公开的，不需要机密保护。此外，由于只有私钥的拥有者可以产生密文，密文可以作为明文的数字签名(digital signature)。公钥的拥有者可以鉴别签名。RSA也许是最广泛用做数字签名的非对称密码系统。另一种系统，数字签名算法(DSA)，已被美国政府规定为国家标准，只具有数字签名功能，而不能用于加密系统。

一方为了证实另一方是否是公钥的真正所有者，可以通过被证明方为验证提供证书。公钥证书(public-key certificate)是一种来自另一方的数字签字，声明另一方的公钥(及其他信息)有一些特殊值。有时，可能有一系列证书，每一个证书含有一个需待证实的公钥，第一个证书叫根证书(root certificate)，没有其他公钥验证它。由于根证书自身含有证实自己的公钥，所以，通常又称为自签证书(self-signed certificate)。本书后几章将更深入地讨论证书。

1.6 鉴别

另一个安全问题就是鉴别(authentication)。它是对用户身份(或代表用户的机器操作)的证实。在分时计算机系统中，为了获知用户登录的身份，就出现了用户身份的鉴别问题。正如

China-pub.com 下载

前面提到的大多数安全模型是对某些用户是准许的，而某些用户是被拒绝的，鉴别对增强访问控制策略是至关重要的。

随着网络计算机系统的出现，鉴别显得越来越重要。网络是共享的和大众化的，获知另一端连接的用户的身份，就像用户需知将要连接的系统的身份一样同等重要。

目前，存在许多鉴别协议，但这些经过专家潜心研究的鉴别协议大多数都有微小的缺陷，这就导致鉴别技术成为一个重要的研究课题。

鉴别的基本方法就是首先让另一端的用户提供名字和口令，然后与系统记录比较。因为监控网络通信的任何一个人都可以获知口令并加以利用，这种方法易被窃听和受到猜测攻击。后来，对这种基本方法进行了一些改进，如一次性口令[40]、IETF标准所说的OTP[31]。但这些改进都很有限，因为一个人只可以携带有限个一次性口令。

这种基本方法是一种基于询问和响应的方法。当然，它们也能扩展到借助密钥分布函数来实现。不同的实体只能共享特有的密钥分配中心产生的钥匙，密钥分配中心可以动态地生成密钥。Needhan-Schroeder协议[59]是网络鉴别方面最早的协议实体A、B把密钥分配中心作为被信任的第三方来建立短期安全会话。这个协议是Kerberos系统(MIT Athena项目的一部分)的基础，后来被DCE、IETF标准所采纳[56]、[60]。

协议设计存在许多不安全之处。像Needham-Schroeder协议在许多方面逐渐暴露出缺陷[9、26]。对安全协议的攻击包括重放攻击(replay attack)和交叉攻击(interleaving attack)。一个攻击者窃听和记录网络信息并重新利用它们(有时需要巧妙地做一些修改)可以攻破安全体系。但这些又不易被协议的设计者发现。于是就有许多正式或非正式的协议分析技术被提出并加以利用。文献[9、15、26、50、55]中包括最新的模型检测工具型应用程序。

为应用而设计的认证协议是一个特别重要的问题，包括人们能记住的用户口令。由于这些口令通常是从相当小的范围内选出的(如字典中的词组)，它们很容易被查找和推演出来。1989年之前，由安全研究团体出版的所有认证协议都面临口令易于猜测这一问题。攻击者只要监控网络流量并设法获得认证协议的运行记录，就可以猜测每位用户的口令并验证猜测是否正确。所有工作都是离线的和不可察觉的。1989年后期，解决这个问题的技术方案[21、47]开始出现，包括EKE和A-EKE[4、5]。智能卡和其他基于硬件的安全设备有利于防止用户猜测口令。

1.7 移动代码

移动代码(mobile code)并不是一个新概念，理论上说，能引起远程系统不同动作的任何代码都可以被看作移动代码。移动代码是整个分布式计算领域正常工作的前提。它包括域名服务(DNS)数据信息、远端程序调用(RPC)的远端命令和UNIX系统远端外壳的执行脚本。本小节主要讨论可执行脚本，这种移动代码之所以非常流行，一是因为它有利于合理分配客户机/服务器上的计算负荷，二是因为它可以降低对网络带宽的要求。

PostScript文件就属于这种代码。只有在显示和查看PostScript文件时其文件内容才被执行。像微软Word文档中的宏一样，只有在文档被读时，宏才被解释执行。又例如Lisp，通过Emacs(一种功能强大的文本编辑器)人们可以阅读E-Mail，Emacs系统解释Lisp程序就像它自己所有的程序一样。当查看内部Emacs时，在E-Mail信息中嵌入的Lisp程序段才被激活。Activex控件和Java小应用程序都是这样的例子。

活动内容并没有带来新的危害，相反，它有助于暴露普通安全机制的不充分性。例如，

China-pub.com
下载

当利用移动代码来更新DNS时，其界面范围相当小，使得系统更安全。但是，在调用 Activex控件时，其界面范围就扩展成整个Win32 API界面，那么整个API的安全漏洞都可能被发现。

移动代码的不断增加引起两种反应。一方面，人们尽力增强系统安全性以便较好地控制和利用移动代码有利的一面，另一方面，人们恐惧移动代码不利的一面，想把它阻止在外围。后者阻碍了时代发展，这是因为移动代码和活动内容流通途径很多 (E-mail)，而滤掉每一个E-mail信息和部分消息是不可能的。Java技术的一个重要设计目标就是使Java成为移动代码的安全平台。

1.8 Java安全性的适用范围

前面各小节在相当广的范围内，描述了目前系统安全性方面的知识。从防火墙到访问控制，从加密到鉴别，Java安全是一种非常重要的谜。这主要因为Java技术是与平台无关的，并且是为Internet和Web编写移动代码、可执行代码的最好工具。

目前Java得到普遍应用，如在金融机构、在线电子商务软件和其他关键应用程序的部件中都采用了Java。所有这些表明，Java平台必须实现自己的诺言：Java平台是安全的Internet编程平台。⊖

Java平台既可被看作客户端应用程序 (如在浏览器内部运行Java)、服务器端应用程序 (用Java编写的服务器软件)，也可被看作是一种操作系统 (在MS-DOS或裸机上直接运行的JavaOS)。不同的用途或许需要不同甚至相互冲突的安全特征。JDK1.2内含公共函数，同时有充分的界面，可以很方便地处理特殊需求。

当Java技术在操作系统内部(例如，Solaris2.6或MS-Windows)得到利用时，它的出现不会改变底层系统的安全特性。例如，在Solaris系统中，Java虚拟机(JVM)的实例只能访问正在运行JVM的用户资源。不过，如果整个应用程序界面只限于Java平台，整个系统的安全性能会得到改善。如果所有的应用程序均采用Java编写，那么在Ms-Windows平台上的许多安全问题就会消失(实际上是隐藏在Java界面的底层而不易被发现)。

在本章最后，我想强调Java平台的安全特性并不局限在JDK1.2中所讨论的，JDK的后续版本无疑会继续丰富安全特性。此外，一整套标准Java界面已经或正在设计，其中包括加密、安全套接层(SSL)、用户鉴别等，这样，Java不仅是而且实际就是许多种安全谜语。正如SSL和浏览器把密码术带到大众市场一样，总之，Java技术在使计算机安全技术成为主流技术中扮演了重要角色。

⊖ 应该澄清JavaScipt并不是以Java为基础，两者只是在名字上相似而已。JavaScipt不具有Java所包含的全面的安全性考虑和安全机制。

China-pub.com
下载

参 考 文 献

[1] *The Java Extensions Framework*. JDK 1.2 documentation, Sun Microsystems, Inc., 1998. `http://java.sun.com/products/jdk/1.2/docs/guide/extensions`.

[2] K. Arnold and J. Gosling. *The Java Programming Language*, Second Edition. Addison-Wesley, Reading, Mass., 1998.

[3] D. E. Bell and L. J. LaPadula. Secure Computer Systems: A Mathematical Model. *Journal of Computer Security*, 4(2–3):239–263, 1996. A modern reprint of the same titled technical report, ESD-TR-73-278, Vol. 2, The MITRE Corporation, Bedford, Mass., 1973.

[4] S. Bellovin and M. Merritt. Encrypted Key Exchange: Password-Based Protocols Secure Against Dictionary Attacks. *Proceedings of the IEEE Symposium on Research in Security and Privacy*, 72–84, Oakland, Calif., May 1992.

[5] S. Bellovin and M. Merritt. Augmented Encrypted Key Exchange: A Password-Based Protocol Secure Against Dictionary Attacks and Password File Compromise. *Proceedings of the 1st ACM Conference on Computer and Communications Security*, 244–50, Fairfax, Va., November 1993.

[6] S. M. Bellovin and W. R. Cheswick. Network Firewalls. *IEEE Communications*, 50–57, September 1994.

[7] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuchynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, Safety, and Performance in the SPIN Operating System. *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 251–266, Copper Mountain Resort, Colo., December 1995. Published as *ACM Operating System Review* 29(5):251–266, December 1995.

[8] D. F. C. Brewer and M. J. Nash. The Chinese Wall Security Policy. *Proceedings of the IEEE Symposium on Security and Privacy*, 206–214, Oakland, Calif., April 1989.

[9] M. Burrows, M. Abadi, and R. M. Needham. A Logic for Authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, February 1990.

[10] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and Protection in a Single-Address-Space Operating System. *ACM Transactions on Computer Systems*, 12(4):271–307, November 1994.

[11] D. D. Clark and D. R. Wilson. A Comparison of Commercial and Military Computer Security Policies. *Proceedings of the IEEE Symposium on Security and Privacy*, 184–194, Oakland, Calif., April 1987.

[12] F. Cristian. Understanding Fault-Tolerant Distributed Systems. *Communications of the ACM*, 34(2):57–78, February 1991.

[13] I. B. Damgšard. Design Principles for Hash Functions. *Advances in Cryptology*: Proceedings of Crypto '89, Vol. 435 of Lecture Notes in Computer Science, 416–427. Springer-Verlag, New York, October 1989.

[14] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT–22(6):644–65, November 1976.

China-pub.com 下载

[15] D. Dolev and A. C. Yao. On the Security of Public Key Protocols. *IEEE Transactions on Information Theory*, IT–29(2):198–208, March 1983.
[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, Reading, Mass., 1995.
[17] M. Gasser. *Building a Secure Computer System*. Van Nostrand Reinhold Co., New York, 1988.
[18] J. A. Goguen and J. Meseguer. Security Policies and Security Models. *Proceedings of the IEEE Symposium on Security and Privacy*, 11–20, Oakland, Calif., April 1982.
[19] J. A. Goguen and J. Meseguer. Unwinding and Inference Control. *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, Calif., April 1984.
[20] T. C. Goldstein. The Gateway Security Model in the Java Electronic Commerce Framework. *Proceedings of Financial Cryptography 97*, 291–304, Anguilla, British Virgin Island, February 1997.
[21] L. Gong, T. M. A. Lomas, R. M. Needham, and J. H. Saltzer. Protecting Poorly Chosen Secrets from Guessing Attacks. *IEEE Journal on Selected Areas in Communications*, 11(5):648–656, June 1993.
[22] L. Gong. Collisionful Keyed Hash Functions with Selectable Collisions. *Information Processing Letters*, 55(3):167–170, August 1995.
[23] L. Gong. Java Security: Present and Near Future. *IEEE Micro*, 17(3):14–19, May/June 1997.
[24] L. Gong, P. Lincoln, and J. Rushby. Byzantine Agreement with Authentication: Observations and Applications in Tolerating Hybrid Faults. *Proceedings of the 5th IFIP Working Conference on Dependable Computing for Critical Applications*, 79–90, Urbana-Champaign, Ill., September 1995.
[25] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2. *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, 103–112, Monterey, Calif., December 1997.
[26] L. Gong, R. Needham, and R. Yahalom. Reasoning about Belief in Cryptographic Protocols. *Proceedings of the IEEE Symposium on Research in Security and Privacy*, 234–248, Oakland, Calif., May 1990.
[27] L. Gong and X. Qian. Computational Issues of Secure Interoperation. *IEEE Transactions on Software Engineering*, 22(1):43–52, January 1996.
[28] L. Gong and R. Schemers. Implementing Protection Domains in the Java Development Kit 1.2. *Proceedings of the Internet Society Symposium on Network and Distributed System Security*, 125–134, San Diego, Calif., March 1998.
[29] L. Gong. New Security Architectural Directions for Java (Extended Abstract). *Proceedings of IEEE COMPCON*, 97–102, San Jose, Calif., February 1997.
[30] J. Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, Reading, Mass., August 1996.
[31] N. Haller, C. Metz, P. Nesser, and M. Straw. A One-Time Password System. Request for Comments (RFC) 2289, Internet Engineering Task Force, February 1998.
[32] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in Operating Systems. *Communications of the ACM*, 19(8):461–471, August 1976.
[33] C. Hawblitzel, C-C Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing Multiple Protection Domains in Java. *Proceedings of the USENIX Annual Technical Conference*, New Orleans, La., June 1998.

China-pub.com
下载

[34] M. P. Herlihy and J. D. Tygar. How to Make Replicated Data Secure. Advances in Cryptology. *Proceedings of Crypto '87*, Vol. 293 of Lecture Notes in Computer Science, 379–391. Springer-Verlag, New York, 1987.

[35] R. Housley, W. Ford, T. Polk, and D. Solo. Internet X.509 Public Key Infrastructure Certificate and CRL Profile. Request for Comments (RFC) 2459, Internet Engineering Task Force, January 1999.

[36] A. K. Jones. *Protection in Programmed Systems*. Ph.D. dissertation, Carnegie-Mellon University, Pittsburgh, Penn., June 1973.

[37] D. E. Knuth. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, Revised Edition. Addison-Wesley, Reading, Mass., 1969.

[38] D. E. Knuth. *The Art of Computer Programming, Vol. 3: Searching and Sorting*. Addison-Wesley, Reading, Mass., 1973.

[39] A. G. Konheim. *Cryptography: A Primer*. John Wiley & Sons, Inc., New York, 1981.

[40] L. Lamport. Password Authentication with Insecure Communication. *Communications of the ACM*, 24(11):770–772, November 1981.

[41] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in Distributed Systems: Theory and Practice. *ACM Transactions on Computer Systems*, 10(4):265–310, November 1992.

[42] B. W. Lampson. Protection. *Proceedings of the 5th Princeton Symposium on Information Sciences and Systems*, Princeton University, March 1971. Reprinted in *ACM Operating Systems Review*, 8(1):18–24, January 1974.

[43] B. W. Lampson. A Note on the Confinement Problem. *Communications of the ACM*, 16(10):613–615, October 1973.

[44] C. E. Landwehr. Formal Models for Computer Security. *ACM Computing Survey*, 13(3):247– 278, September 1981.

[45] S. Liang and G. Bracha. Dynamic Class Loading in the Java Virtual Machine. *Proceedings of the ACM Conference on Object Oriented Programming Systems, Languages, and Applications*, 36–44, Vancouver, British Columbia, October 1998.

[46] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, Mass., 1997.

[47] T. M. A. Lomas, L. Gong, J. H. Saltzer, and R. M. Needham. Reducing Risks from Poorly Chosen Keys. *Proceedings of the 12th ACM Symposium on Operating System Principles,* Litchfield Park, Ariz. Published in *ACM Operating Systems Review*, 23(5):14–18, December 1989.

[48] D. McCullough. A Hookup Theorem for Multilevel Security. *IEEE Transactions on Software Engineering*, 16(6):563–568, June 1990.

[49] G. McGraw and E. W. Felten. *Java Security: Hostile Applets, Holes, and Antidotes*. John Wiley & Sons, New York, 1997.

[50] C. Meadows. Using Narrowing in the Analysis of Key Management Protocols. *Proceedings of the IEEE Symposium on Security and Privacy*, 138–147, Oakland, Calif., May 1989.

[51] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, New York, 1997.

[52] R. C. Merkle. *Secrecy, Authentication, and Public Key Systems*. PhD thesis, Stanford University, UMI Research Press, Mich., 1982. Revised from 1979 thesis.

[53] R. C. Merkle. A Fast Software One-Way Hash Function. *Journal of Cryptology*, 3(1):43–58, 1990.

China-pub.com 下载

[54] C. H. Meyer and M. Schilling. Secure Program Load with Modification Detection Code. *Proceedings of the 5th Worldwide Congress on Computer and Communication Security and Protection–SECURICOM 88*, 111–130, Paris, 1988.

[55] J. K. Millen, S. C. Clark, and S. B. Freedman. The Interrogator: Protocol Security Analysis. *IEEE Transactions on Software Engineering*, SE-13(2):274–288, February 1987.

[56] S. P. Miller, C. Neuman, J. I. Schiller, and J. H. Saltzer. Kerberos Authentication and Authorization System. Project Athena Technical Plan Section E.2.1, Massachusetts Institute of Technology, October 1988.

[57] M. Moriconi, X. Qian, R. A. Riemenschneider, and L. Gong. Secure Software Architectures. *Proceedings of the IEEE Symposium on Security and Privacy*, 84–93, Oakland, Calif., May 1997.

[58] M. Naor and M. Yung. Universal One-Way Hash Functions and Their Cryptographic Applications. *Proceedings of the 21th Annual ACM Symposium on Theory of Computing*, 33–43, Seattle, Washington, May 1989.

[59] R. M. Needham and M. D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Communications of the ACM,* 21(12):993–999, December 1978.

[60] B. C. Neuman and T. Ts'o. Kerberos: An Authentication Service for Computer Networks. *IEEE Communications*, 32(9):33–38, September 1994.

[61] P. G. Neumann. *Computer-Related Risks*. Addison-Wesley, Reading, Mass., 1995.

[62] U.S. National Bureau of Standards. *Data Encryption Standard*, January 1977. U.S. Federal Information Processing Standards Publication, FIPS PUB 46.

[63] U.S. General Accounting Office. Information Security: Computer Attacks at Department of Defense Pose Increasing Risks. Technical Report GAO/AIMD-96-84, Washington, D. C., May 1996.

[64] M. O. Rabin. Fingerprinting by Random Polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, Cambridge, MA, 1981.

[65] M. Reiter. Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart. *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, 68–80, Fairfax, Va., November 1994.

[66] S. Ritchie. Systems Programming in Java. *IEEE Micro*, 17(3):30–35, May/June 1997.

[67] R. L. Rivest. The MD5 Message-Digest Algorithm. Request for Comments (RFC) 1321, Internet Engineering Task Force, April 1992.

[68] R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[69] J. H. Saltzer. Protection and the Control of Information Sharing in Multics. *Communications of the ACM*, 17(7):388–402, July 1974.

[70] J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[71] V. Samar and C. Lai. Making Login Services Independent from Authentication Technologies. *Proceedings of the SunSoft Developer's Conference*, March 1996.

China-pub.com
下载

[72] R. S. Sandhu. The Typed Access Matrix Model. *Proceedings of the IEEE Symposium on Research in Security and Privacy*, 122–136, Oakland, Calif., May 1992.

[73] F. B. Schneider. Implementing Fault-Tolerant Services Using the State-Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[74] B. Schneier. *Applied Cryptography*. John Wiley & Sons, New York, 1994.

[75] M. D. Schroeder. *Cooperation of Mutually Suspicious Subsystems in a Computer Utility*. Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, Mass., September 1972.

[76] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, 213–227, Seattle, Wash., October 1996. Published as *ACM Operating Systems Review*, 30, special winter issue, 1996.

[77] A. Shamir. How to Share a Secret. *Communications of the ACM*, 22(11):612–613, November 1979.

[78] L. van Doorn, M. Abadi, M. Burrows, and E. Wobber. Secure Network Objects. *Proceedings of the IEEE Symposium in Security and Privacy*, 211–221, Oakland, Calif., May 1996.

[79] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible Security Architectures for Java. *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, 116–128, Saint-Malo, France, October 1997.

[80] M. V. Wilkes. *Time-Sharing Computer Systems*. MacDonald, London, 1968.

[81] W. A. Wulf, R. Levin, and S. P. Harbison. *HYDRA/C.mmp—An Experimental Computer System*, McGraw-Hill, New York, 1981.

[82] F. Yellin. Low Level Security in Java. *Proceedings of the 4th International World Wide Web Conference*, Boston, Mass., December 1995.

China-pub.com 下载

China-pub.com
下载

第8章 发 展 方 向

任何发展都不是完美无缺的，一个问题的解决将使我们面对另外的问题

——马丁·路德

随着JDK 1.2和JCE1.2的发布，Java的安全问题已进入一个令人激动的新阶段。但这并不意味着问题的结束，而是象征着一个新阶段的开始，因为还有很多的问题亟待解决，本章将涉及Java安全技术的一些发展方向。

8.1 安全管理

众所周知，任何不好用的东西都不能在实际中得到应用，安全问题也不例外。在 JDK 1.2中，SunMicrosystemsJava安全系统研发小组制定了简便性的目标，因此我们只设计了为完成一定工作所必须的最少的 API。我们将各功能分配到各个类当中去，使得每一个类在逻辑上完备和易于理解。而且在保证它们的正确性和清晰性的前提下，我们为每一个类和方法选择了简单的名字。

使安全结构能用于现实生活的另一个重要特性是安全管理。如前所述， JDK本身并不是一个最终用户产品，所以它并不是捆绑安全管理软件最好的地方。相反，系统经销商和应用开发者可以为他们的顾客创建和表示正确的界面 (包括GUI)。

然而，Java插件，一个Java运行环境下的浏览器插件，正越来越多的被终端用户直接用于在浏览器中升级JVM。因此，Java插件作为一个直接从JDK衍生出来的产品，可以用更多的管理特性来改善。例如，它可以使用浏览器中存在的相同的证书库，因而无需为它建立专门的证书库。

同样，当插件在Internet上遇到以前未知的applet和应用程序时，为了使其能按照所设计地运行，我们需要一种解决方法来授予 applet所需要的特权。对于这个配置问题，一个解决办法就是为开发者设计一种方法来表示所需要的许可权，以及设计一种方法来帮助浏览器使用者理解这些许可权的意义和含义。

安全管理并不仅仅针对最终用户。通常应用程序开发者并不希望只成为编写安全应用程序的安全问题专家。例如，他们可以获益于某种界面，通过该界面他们可以打一个电话得到安全服务而不必要去了解其中的细节问题，如解码算法、密钥大小和协议类型。如此简单安全的API不但可以提高最终应用程序的安全质量，而且可以大大的提高生产率。

8.2 JDK特性的改善

从顾客的反馈来看，有几个安全特性在 JDK的一些未来的版本中是值得投资开发的，其中一个就是资源消耗管理。在某些情况下，这实现起来相对容易，例如限制一个应用程序同时可以弹出的窗口数目。但是在其他一些情况下就要困难的多，例如在实时操作中限定内存、CPU或文件系统的使用而同时又希望有好的性能。

另一个值得开发的安全特性就是类加载器的设计，它们在安全方面的含义仍是很微妙的，

China-pub.com 下载

并且当前对ClassLoader类的规定方法仍有待改进。除了URLClassLoader以外，只有当系统安全策略被配置成允许的时候，Applets和应用程序才可以创建类加载器。如此严格的限制也许会阻碍某些应用程序的开发，在这个问题上还需要做很多的研究工作。

第三个安全特性是即时撤销性，借此一项已赋予的特权可以在安全策略改变后马上被撤销。目前，新策略只有在它的内容被刷新之后而且只有对新开始的应用程序和applet才生效。实现即时撤销需要特权系统用Policy对象(如果许可权可以动态地改变它们的内容，那么也许是每一个Permission对象)来注册自己(作为一个接收者)。那么Policy对象必须能够保证通知系统是否改变它的安全策略。

另一个与安全策略有关的特性是安全策略的组成。目前，每一种安全管理都使用某一种形式的安全策略。如果想要实施某种新的策略，它可能是两种已有策略的组合，则需要使用新的SecurityManager。理想的情况是，安全策略可以被方便地构成，而不需要对现有的安全管理类完全重写。

JDK1.2介绍了一种保守而坚固的访问控制算法，可以阻止一些编程的错误进入安全黑洞。除此之外，还有一些其他的方法可用来促进这种工作。例如，在一定意义上，再次调用doPrivileged原语可以赋予一段代码的所有许可权。在某些情况下，一个应用程序希望仅仅只能启用它被赋予的许可权。这种选择性的启用进一步减少了编程错误带来的安全障碍。我们可以期待丰富这种原语，使之能够增加另外的参数来规定许可权的启用，可能是Permission，PermissionCollection或者Permissions等等类型。

另一种减少安全依赖性的方法是划分系统域。为了方便起见，系统域可以被看作是所有系统代码的单一大集合。然而，为了给予更好的保护，系统代码可以在多系统域内运行，在这里每一个域保护一定形式的资源而且被给予一定的权利。例如，如果文件系统代码和网络系统代码在不同的域内运行，前者对网络资源没有权利，而后者对文件系统资源没有权利，那么一个系统域内的错误或者安全方面的缺陷所造成的风险和后果就可能被局限在这个域的范围之内。

此外，目前保护域是作为类加载的结果而清晰地创建的。提供清楚的原语来创建一个新的域可能是有用的。通常情况下，域支持继承性，一个子域会自动的继承父域的安全特性，除了在某些情况下，父域进一步明确地限制或者扩大它的子域。

最后，我们需要一种一致的处理非类内容的方法。当applet或者应用程序运行签名的内容(类或其他资源)时，JAR和Manifest在代码签名的规定上允许一个非常灵活的形式。同一个档案中的类可以是未签名的、单密钥或多密钥签名的。档案中的其他资源，例如声音素材和图形图像，也可以是签名过的或者未签名的。然而，我们并不清楚如果档案中的任何一个类被签名，是否图像和声音素材也需要同样的密钥签名。如果图像和声音素材被不同的密钥签名，它们能够被放入相同的appletviewer(或浏览页)或者被发往不同的浏览者吗？这样的问题并不容易回答。任何回答都需要满足在平台和产品之间的一致性以使之最为有效。现在的方法是处理所有的图像和声音素材而不管它们是否被签名。一旦有了一致的意见，这个暂时的解决方法应该被改进。

8.3 Java鉴别和许可服务

当Java技术不只用于构建单一的桌面系统，而且也用于一个完全成熟的分布式系统时，

China-pub.com
下载

我们必须处理分布式系统安全——这个全新领域的问题(就像在第1章提到过的)。例如，在受到敌意的网络攻击的情况下就需要提供使RMI安全可靠的附加机制。对于Sun公司最近发布的基于Java的简单连接数字设备的技术Jini来说，如果环境包含了共存但潜在相互敌意的部分，那么服务登记和定位就必须被安全可靠的管理。一个高层的完整的服务必须是可靠的，就像电子商务的交易。另外，可以利用许多低层的安全协议，例如网络安全协议Kerberos和IPv6。这个领域如此巨大，在这个短小的章节中无法详述，但是对所有这些问题来说最关键的基础是对用户进行鉴别和根据这个信息来实施访问控制。

回想一下JDK1.2运用安全策略来决定对运行代码赋予访问许可权，而且这种决定依赖于代码的特性，例如代码从何而来、是否被数字签署以及被谁签署。这样一种代码中心型的访问控制并不常见。在成熟的操作系统中最常见的传统安全度量，是用户中心型的，它们以谁运行程序为基础申请控制，而不是以哪个程序被运行为基础。但是代码中心的访问控制被大大看好，因为用户基本上保持着不变的身份在网上冲浪而且运行可执行内容(例如，Java编写的动态代码)。而另一方面，用户可能更信任某一段动态编码，而且可能希望以更多的特权来运行这段代码。因此以代码中心方式来控制动态代码的安全性是很自然的。

然而，Java正越来越广泛地用于多用户环境中。例如，一个公共网络终端，一个企业应用程序(如SunMicrosystem的工资管理工具)，或者一个处理租借成千上万Palm Pilot的服务器。所有的这些都必须和不同的用户打交道，无论是同时的还是顺序的。而且必须根据用户的身份赋予不同的许可权。

Java鉴别和许可服务(JAAS)是为鉴别用户和分配许可权而设计的一套标准程序界面。利用JAAS和JDK1.2，一个应用程序可以提供代码中心的访问控制、用户中心的访问控制或者是两者的结合。JAAS也为支持交叉域许可和running-on-behalf-of形式授权的一般机制奠定了基础。

数十年来鉴别问题已成为安全研究中的主要问题。然而，Java环境提出了特殊的挑战。JAAS是为如下需要而设计的。

- 可延展性。Java程序界面要求鉴别和许可能够很容易地加以扩展。这是一个不太常见但是非常基础的要求。
- 可插入性。不同的系统可以很容易地把它们新的或者现存的鉴别包含到JAAS框架中去。
- 兼容性。JDK1.2中基于代码的访问控制结构和JAAS中新的基于用户的访问控制机制可以独立地共存,而且可以完美地结合起来以实现复杂的安全策略。

几个现存的标准，如普通安全服务应用程序界面(GSS-API)和简单鉴别与安全层应用程序界面(SASL)，都支持鉴别。SASL构建了为基于连接的协议而提供鉴别支持的框架，因此它适用于执行网络鉴别的应用程序。如同JAAS一样，SASL也是模块化构造。GSS进程，如Kerberos或简单公钥进程(SPKM)，可以在SASL框架下插入。另一方面，JAAS也对独立非连接性环境下的鉴别提供支持。因此JAAS和SASL/GSS互为补充，为鉴别提供本地的和基于网络的支持。

依赖于Kerberos(或者其他服务)来鉴别的环境，是这些结构可以共存的情况之一。当最初登录到Kerberos(为获得用户的Kerberos Ticket Granting标签)和底层的操作系统时，JAAS登录模块可以被插入登录应用程序中来鉴别用户。通过安装一个Kerberos登录模块，用户可以不必再执行额外的步骤，例如再执行Kinit命令来得到标签。当用户执行客户端应用程序，以通

China-pub.com 下载

过网络到达某些使用 Kerberos协议的服务器来进行鉴别的的时侯，这些应用程序可以使用SASL，它基本上可以插入合适的Kerberos机制来执行真正的鉴别。

8.3.1 主题与主体

用户常常依赖于计算服务的辅助来执行工作，而计算服务通常依赖于用户名来鉴别它的用户。然而，用户对每一个服务所用的名字可能不尽相同，而且事实上甚至可能对每一个服务都有不同的名字。此外，一个服务也可能是其他服务的用户，因此也可能有多重的名字。JAAS用主题(subject)来代表一个计算服务的所有用户。因此，用户和计算服务都代表了主题。同样，用主体(principal)代表与一个主题有关的名字。因为主题可能有多个名字 (一般对每个相互作用的服务都有一个名字)，基本上一个主题包含一组主体。

只有当主题成功地被一个服务鉴别时，主体才与某一个主题相联系。鉴别体现了一方鉴别另一方的身份的过程，而且必须在安全方式下执行；否则，一个犯罪者可以冒充别人毫无限制地访问一个系统。一般的，鉴别要求主题出示一定形式的证据来证明它的身份。这样的证据也许只是主题希望知道(例如密码或PIN)或者已有(例如指纹或者嗓音)的信息。

当主题试图鉴别于服务时，它通常在提供它的身份证据的同时也给出它的名字。如果鉴别成功，服务程序将会根据所给的名字为主题分配一个特定服务主体。这样，应用程序和服务可以仅索引与这个主题相关的主体来判定主题的身份。

8.3.2 证件

除了主体信息，一些服务也许希望将主题和其他与安全有关的特性和数据相联系。 JAAS把这样的与安全有关的一般特性叫做证件 (credential)。证件中可能包含将来其他服务鉴别主题的信息。Kerberos标签[60]就是这样的一个证件。证件也可能只包括或索引一些仅仅使主题执行某一活动的数据。例如密钥，就是一种可以使主题签名或加密数据的证件。

尽管Kerberos标签和密钥举例说明了一般形式的证件，证件还可以表示更大范围内的与安全有关的数据。代表用户运行的程序必须与它们所依赖的服务合作，以期在必需的且在它们相互作用中应被理解和承认的证件上达成一致。因此，一些证件是标准的或公认的，而另一些可能是专门面向应用程序和服务的。另外，证件实现不一定非要包含实际的与安全相关的数据，它们可以只是索引这些数据。当数据必须物理地驻留在单独的服务器或者硬件设施中时(如智能卡的私钥)就属于这种情况。

一个主题只有被服务成功的鉴别才能得到证件。在成功鉴别的基础上，服务程序建立起正确的证件而且将它与主题相联系。一旦一个主题有了证件，在主题上运行的应用程序就可以(如果有正确的许可权)访问和使用这些证件。 JAAS对于将证件授权给第三方没有强加任何限制。相反，它要么允许每一个证件实现规定它自己的授权协议 (就像Kerberos)，要么将授权决定权交给应用程序。

8.3.3 可插入和可堆叠的鉴别

传统的服务程序要求一个主题提供它的名字或者某种形式的证据来鉴别。证据的类型取决于特定服务的安全参数。 JAAS鉴别框架基于PAM[71]，因此支持这样一种结构，即允许系统管理员插入正确的鉴别服务来满足安全需要。这种结构也使应用程序能够与底层的鉴别程序

China-pub.com
下载

保持独立。因此，当新的鉴别服务程序出现或者当目前的服务过期时，系统管理员可以轻易地将它们插入，而不用修改或者重新编译现有的应用程序。这个流程见图 8-1。

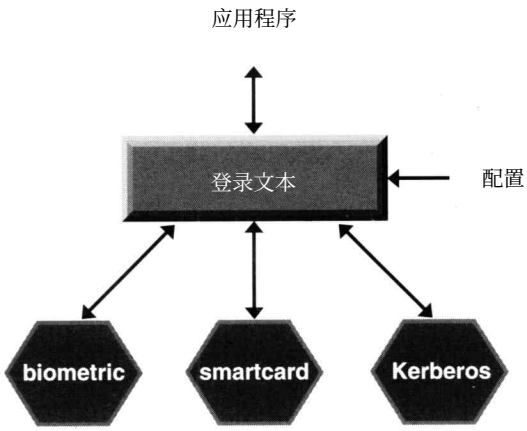


图8-1 可插入的鉴别

当试图鉴别一个主题时，一个应用程序调入鉴别框架，在 JAAS中该框架被定义为一个登录文本(login context)。这个登录文本将参考那些决定鉴别服务或者在这个应用程序中被插入的登录模块(login module)的配置。因为应用程序只和登录文本界面，所以它完全与配置过的登录模块相独立。

每个登录模块都以不同的方式来鉴别主题。例如，传统的基于口令的登录模块提示输入用户名并验证密码；智能卡登录模块告知主题将它的卡插入读卡器并验证 PIN；生物型的登录模块提示输入用户名并且验证主题的指纹。针对应用程序的安全要求，系统管理员可以配置正确的登录模块。事实上，系统管理员可以在一个应用程序中插入多个登录模块。这种形式的层叠配置见图8-2。

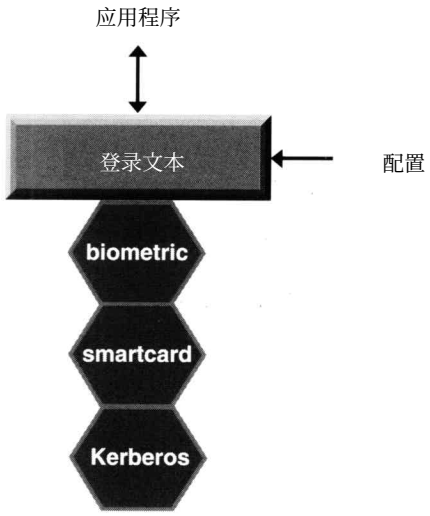


图8-2 可堆叠的鉴别

China-pub.com 下载

一个主题按照配置中规定的顺序被登录模块鉴别。一般的，不管前一个登录模块是否登录成功，主题继续被堆栈里的登录模块鉴别。这也有助于使黑客找不到错误源。配置中的其他参数可以打破这种规则，也可以规定必须通过哪个登录模块才能使整个鉴别通过。关于登录配置规定的具体细节将在本节后面叙述。

只有当所有必须的登录模块(由配置决定)登录成功之后，登录文本才会返回一个鉴别成功标志给调用程序。为保证这一点，登录文本用两个阶段执行这个鉴别步骤。两个阶段都必须成功地完成，登录文本才能返回标志总体鉴别成功的状态。

(1) 登录文本调用每一个配置过的登录模块，而且指引它验证主题的身份。如果所有必须的登录模块成功地通过这个阶段，登录文本就进入第二个阶段。

(2) 登录文本再次调用每一个配置过的登录模块，指引它正式的执行鉴别过程。在这个阶段中，每一个登录模块都会将主题与任何相关的含有用户名和主题的证件的主体联系起来。

因此，一旦总体鉴别过程完成，调用程序可以遍历一个主题的主体集合来得到它的各种身份，而且可以遍历一个主题的证件来访问补充数据。一些登录模块可能只和主题的证件 (而不是主体)相联系。例如，智能卡模块可以通过验证被提供的 PIN来鉴别主题，如果鉴别成功，它只将主体与索引到卡上的密钥的证件相连。在这种情况中，智能卡模块就并不将主题与主体相联。

不管哪一阶段失败了，登录文本都会调用每一个配置过的登录模块，且指示它们放弃所有的登录尝试。这时每一个登录模块都清除掉它所有的已与鉴别尝试相联系的相关状态。

在这两阶段的鉴别过程中，如果任一个登录模块鉴别失败，它并不休眠，也不重试鉴别过程。否则，试图鉴别的主体可以探测到是哪一个登录模块失败。调用程序有责任执行如下任务，如重试，或者在每一个模块都完成了一轮(包括两个阶段)的鉴别后选择是否重新执行它们。

在这个鉴别过程中，登录模块有机会和能力共享信息，不管它是不是依赖于安全要求。共享信息的一个原因是为了达到单次登录的目的。例如，堆叠的登录模块可以共享用户名和口令信息，用户只需要输入一次信息，就能够鉴别多个服务上。如果主体对每个服务有不同的用户名和口令，登录模块也可以相互合作将这个信息映射到相关的专门服务信息中去。因此，尽管主体只输入一个用户名和口令，从该信息中可以映射出主体在各个服务程序中的用户名和口令，这样使得主体能够相对容易地在多个服务中得到鉴别。

配置

JAAS登录配置规定登录模块在一定的应用程序下被插入。配置语法基于 PAM的定义。下面是一个例子。

```
Login {
   com.sun.security.auth.sampleLoginModule  REQUIRED debug=true;
   com.sun.security.auth.SolarisLoginModule  REQUIRED;
}
```

配置中每个项目通过一个应用程序名字被索引，如此例中的 Login，而且包含一系列为这个应用程序配置的登录模块。鉴别程序按照列出的顺序在这个队列中运行，而 REQUIRED这个标志值控制着总体行为。

以下是有效标志值的描述。

- REQUIRED。此登录模块必须成功。然而，不管它是否成功，鉴别仍按照登录模块队列里的顺序继续进行。即使某模块登录失败也必须继续，因为在该点的放弃将给潜在的攻

China-pub.com
下载

击者提供有用的信息，例如哪个模块失败了及其原因等。

- REQUISITE。此登录模块必须成功。如果它成功了，鉴别仍按照登录模块队列中的顺序继续进行。如果它失败了，控制权立刻返回应用程序 (鉴别不继续进行)。
- SUFFICIENT。此登录模块不要求必须成功。如果它成功了，控制权立刻返回应用程序 (鉴别不继续进行)。如果它失败了，鉴别按照登录模块队列中的顺序继续进行。
- OPTIONAL。此登录模块不要求必须成功。不管它成功或者失败，鉴别仍然按照登录模块队列中的顺序继续进行。

只有当所有的REQUIRED和REQUISITE登录模块成功时，总的鉴别过程才通过。如果对一个应用程序没有REQUIRED或REQUISITE登录模块被配置，那么至少一个 SUFFICIENT或者OPTIONAL登录模块必须通过。也可以用 LoginModule来定义选项以支持调试 /检测能力。要做到这一点，需要使用键 -值对debug=true。

8.3.4 回调

通过使用登录文本，应用程序和底层的登录模块保持独立。然而，登录模块可能需要在任何形式的应用程序中插入。不管登录模块插入到何种应用程序之下，它必须能够通过呼叫程序来收集信息和对主体显示信息。例如，一个需要用户名和口令信息的登录模块要有提示主体输入这些信息的能力，而不需要知道是否调用程序有 GUI。

登录文本通过允许应用程序规定一个回调 (callback)来解决这个问题，通过它底层的登录模块可以与主题相互作用。应用程序提供一个回调工具给登录文本，这样登录文本可以直接把它传给每个登录模块。所以登录模块可以调用回调来收集或者显示相关的信息。由应用程序实现的回调，本身就知道是创建一个图像窗口还是简单地使用一个标准输出流。回调的设计和使用见图8-3所示。

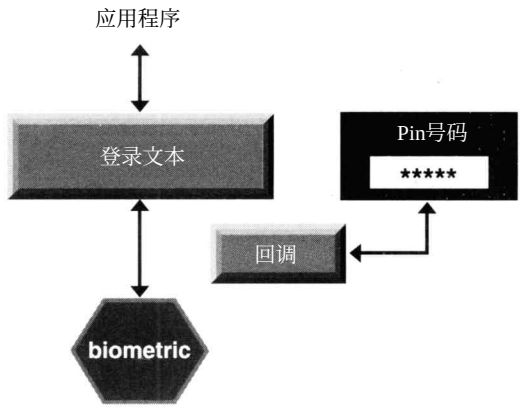


图8-3 回调的设计和使用

8.3.5 访问控制

当前的JDK1.2SecurityManager类实现实施基于原代码的访问控制，一种基于代码从哪里来和谁签署了编码的控制方式。 JAASSecurityManager实现使用基于主题的访问控制，一种基于谁运行这些代码的控制方式，从而提高了安全管理。只有通过基于原代码的核查和基于主

China-pub.com 下载

题的核查，JAAS SecurityManager才能对敏感的资源进行访问。

1. 许可权的赋予

鉴别是许可的基础[41]。尤其是一旦应用程序知道了主题的准确身份，它就可以确切的规定主题可以执行那些操作。因为主题只代表为用户保存相关信息的没有名字的容器，而主体代表这个主题被鉴别过的身份，所以赋予主题的一组许可权依赖于与这个主题相联系的主体，而不是主题本身。换句话说，许可权基于被鉴别的主体被赋予一个主题。被赋予的确切许可权由外部的访问控制策略来配置。

JDK1.2策略语句是为基于原码的访问控制规定的，JAAS语句是它的一个简单延伸，并且加上了赋予许可权语句中的主体项目。因此基于原代码的策略对一个原代码赋予许可权（一个URL以及签名者的别名），然而JAAS策略对主体赋予许可权(以主体类名和主体本身的名字来识别)。

```
grant Principal com. sun. security. auth. SolarisPrincipal "gong"  {
  permission java . lang. RuntimePermisssion. "queuePrintjob";
  permission java . io. FilePermission  "/home/gong", "read, write";
}
```

在这里，任何含有名字gong的相关的SolarisPrincipal主体被赋予打印任务等待的许可权，以及从gong的主目录中读写文件的许可权。

2. 实施以用户为中心的安全策略

在JAAS中，应用程序将主题与一系列操作相联系，以期得到这个主题的访问控制。当执行操作时，JAAS SecurityManager取得相关主题，并在允许它执行敏感操作和访问敏感资源之前核查它是否被赋予了必要的许可权。如果主题没有被赋予充足的许可权，SecurityManger就启动SecurityException。

运行时，某个操作可能要调用其他的操作，因此建立了一系列的嵌套操作。当这种情况发生时，每一个嵌套操作都被推入一个堆栈。当JAAS SecurityManger必须做出访问控制决定时，它访问堆栈并找到所有与堆栈中操作有关的主题。它核查并确定堆栈中的每一个主题都被赋予了必要的许可权。如果一个或者更多的主题没有必要的许可权，SecurityManger就启动SecurityException。结果，对整个堆栈的总许可权实际上等于对堆栈中每一个主题所赋予的许可权的交集。

在某些情况下，一个操作和它的相关的主题可能希望行使它自己的许可权，而且，为了得到全面的访问，希望不依赖于早先在堆栈中有相同许可权的操作。这种情况发生在当一个正在执行的操作与它相关的主题，查询另一个与不同主题相关的操作，以完成自身的一定任务时。被查询的操作当然具有必要的许可权来执行所需任务，但一般来说早先的那个操作并没有。如果还是使用早先规定的访问控制模式，访问可能被拒绝。为解决这个问题，被查询的操作，如果有许可权，则可以定义自身享有特权。当享有特权时，该操作不需要它的调用程序或请求者被赋予和它一样的特权，就可以对被赋予许可权的资源进行全面的访问。只有操作本身以及它有关的主题，需要有必要的许可权。

注意到如果一个享有特权的操作随后又调用一个非特权的操作，那么再一次的，访问操作权决定于有特权的和后来的无特权的操作这两者的主题的许可权的交集。JAAS中的特权操作是对JDK1.2中的doPrivileged界面的模仿。

China-pub.com
下载

8.3.6 JAAS工具

JAAS工具由分布在三个程序包中的大约25个类和界面组成。

- javax.security.auth。此包包含JAAS框架需要的基本类，包括Subject类和Credential界面。尽管主体也是一个基本概念，它们在JAAS中并没有相应的类或者界面。然而，JAAS使用JDK已经提供的主体界面java.security.Principal。此包也包括许可需要的基本类。尤其是，它包括一个SecurityManager类和一个访问控制Policy类。
- javax.security.auth.login。此包包含支持可插入鉴别的类。它包括一个LoginContext类，LoginModule界面、LoginConfiguration类和LoginException类(报告鉴别失败)。
- javax.security.auth.callback。此包包括不同的回调类，登录模块可以用它来与主题相互作用。它包括一个CallbackHandler类和Callback界面以及几个回调工具，例如NameCallback、PasswordCallback、TextInputCallback及TextOutputCallback。

除此之外，包中还包含一些为实施正确的安全核查而定义的Permission类和为实施正确的错误处理而定义的Exception类，这里不再详述。

8.4 小结

在本书中，我们带领你在Java的安全结构中进行旅行。这次旅行包括Java平台提供的多层次上的各种安全特性——语法级别上的，JVM内的，还有贯穿核心库及其外延的——当然还包括各种定制的应用程序和applet。你也看到了例如类型安全、对象级别的安全、安全策略、细微访问控制以及安全配置问题等。而且参观了Java的密码结构，其中包括了JDK中的数字签名和证书，以及单独发行的JCE。

JDK的发展已经经历了从最初的JDK1.0到JDK1.1，再到JDK1.2。与此独立的安全扩展也发展起来——最初是JCE，而后到JAAS，而且这条道路还在向前发展。

一些主要的Java技术趋势支撑着这个发展，理解这些有助于我们放眼Java的发展方向。

首先Java技术在工业上的应用不再单一地集中在客户方，而同样也应用在服务器方。与此新的焦点相伴而生的是日益增长的处理安全问题的需要，这就像一个企业的环境一样的平常。例如，处理单一用户拥有的一个JVM不再能满足要求，取而代之的，运行于一个JVM中的服务器应用程序可以代表多个用户。再如，敏感任务的应用程序必须容许错误，并且经受得住拒绝服务攻击。因此JVM内的资源控制变得更加重要。所以我们说，JCE与JAAS只是一个开始，在未来的应用程序中安全方面还有广阔的发展空间。

第二个趋势是世界正变得越来越小。Java技术被应用在信息和Internet设备中，如智能卡、个人数字助理(PDA)、蜂窝式电话、传呼机、网络可视电话、机顶盒等等。与JDK所针对的标准的桌面系统相比，这些设备有独特的需要。由于具有较小的外形和内存以及相对的动力不足，它们经常采用特殊的Java平台，如PersonalJava和EmbeddedJava。进一步来讲，这些设备特定的应用环境有更强的、而不是更弱的安全需要。例如，蜂窝电话机制造商可能允许电话服务供应公司，如电话最初的设备制造商(OEM)添加一些销售特性，但不希望任何组织改变电话的核心。

例如，当有第三方，如旅馆连锁店与电话公司开始合作提供一定的合作服务时，情况可能变得更加复杂。当一个升级的新的软件特性通过无绳连接自动从旅馆连锁上下载时，保证对升级软件的鉴别、无绳连接的完整性以及升级软件(和它的同属应用软件)与其他已安装的软

China-pub.com 下载

件之间的分离是非常关键的。

为保证例如PersonalJava和EmbeddedJava这样的平台有很好的安全性，一个主要的挑战是设计适用于小的指纹的安全特性。例如，为满足机顶盒等的客户安全需要，为 JDK1.2设计的细微访问控制已经被应用到 PersonalJava中。结果表明，整个的安全特性包，包括数字签名这个基本功能(特别是对数字证书的支持)过于庞大；例如， X.509明显的不是为目前大众市场中的小存储器的设备设计的。然而，我们可以设法达到这个目标，因为 JDK1.2中的访问控制机制是通过一个单一界面java.security.cert.Certificate与签名和证件功能相连的。在这两个机制之间没有其他的API关联。所以，密码特性被设置成可选项，当密码特性可用时访问控制机制可以正常地工作，反之则将所有的代码设为未签名的。

一个相关的趋势是世界正变得更加紧密相连。这也增加了对安全通讯、安全移动和无绳操作以及安全在线传输的需要。

最后一个趋势是围绕 Java技术的热点不再仅仅是平台和基本设施问题。越来越多进入市场的应用程序带来了安全设计的新问题，包括安全用户界面和策略管理。过去，安全技术更多的集中在平台上 (如操作系统和网络协议)，相对来说对应用程序注意较少 (大部分都在完全被信任的基础上运行)。因此在这个领域中仍需要大量的工作。

计算机安全技术经过大约三十年的发展，现在已日趋成熟，并成为主流技术，但是仍有大量的工作尚未完成，我们仍需面对许多的挑战和机遇。

China-pub.com
下载

第7章　编程密码学

安全是文明社会的主要伪装。

—— 萧伯纳

前面几章简要地谈到了密码学的一些基本概念，以及代码签名和使用证书等问题，它们都依赖于公钥加密系统，本章将深入到Java的内部,研究隐藏在API和其他工具中的Java密码学结构(JCA)。

JCA最早出现在JDK1.1中，只有有限的功能，包括为数字签名和消息摘要而设计的 API。为了增强和发展基于Java平台的密码学功能， JDK1.2，已经有意使JCA扩展为框架结构。更为一般的讲，现在的JCA包含所有JDK1.2中与密码有关的安全API。例如，新的支持X.509 v3证书的证书管理结构。而且也包括一些提供器结构，允许多种可相互作用的密码学实现，以及与之相关的规则说明。

从逻辑上讲， JCA包含定义在JDK1.2和JCE1.2中的密码API，它们为加密 ,密钥交换, MAC(消息鉴别代码) ,以及许多与加密相关的类提供了特性。而JDK和JCE 联合提供了一套与平台无关的API。现在， JCE作为JDK的一个扩展版本而单独发行，并与美国密码 出口法规保持一致。本章讨论的结构话题，包含JDK与JCE,但仅仅集中讨论JDK中与密码相关的界面。⊖

7.1　设计原则

JCA设计应遵循的两个原则：

(1) 算法的独立性和可扩展性。

(2) 实现的独立性和相互作用性。

JCA的目的是使API的用户建立密码的概念，例如数字签名和消息摘要，而不是去关心如何实现这些概念，或是用来实现这些概念的算法。与此同时， JCA提供了标准的API，使得开发者能够请求指定的算法和实现。

算法的独立性是通过定义密码服务类(service classe)来获得的，该类提供了密码算法功能，例如MessageDigest，Signature，KeyFactory类 。

使用基于提供器的结构，可得到实现的独立性。第 4章讨论了提供器的一般概念。在 JCA中，提供器代表密码服务提供器(Cryptographic Service Provider，CSP)实现一个或多个JCA密码服务的一个或一系列程序包，例如数字签名算法、消息摘要程序包、密钥转换服务等。一个应用程序也许仅需要一个实现特殊服务 (比如DSA签名算法)的特殊类型的对象(如Signature对象)，并接收来自一个已安装的提供器的实现。如果希望的话，还可以从某个指定的提供器中实现这个操作。当更快或更安全的版本可用时 ,对应用程序而言,提供器会变得更加清晰。

在JDK1.1中，包含一个与平台无关的实现 —— 产生随机数，此操作在一些机器上是无效的。除此之外，该实现有固定的代码，不能轻易设置。但在 JDK1.2中，可以轻松指定一个自

⊖ 关于JCE1.2的参考书正在酝酿当中。

China-pub.com 下载

制的服务，以利用特定操作平台 (例如Linux)上好的随机数资源。

考虑到API设计的一般性，从某种程度上说，即使是各种实现有不同的特性 ,也可得到实现的相互作用性。它们可以共同运行，例如，相互使用密钥，相互验证签名。例如，当某个密钥工厂实现被安装，对于同样的算法，由一个提供器产生的密钥可由另一个提供器使用 .同样，由一个提供器产生的签名可由其他提供器来验证。即使一个提供器在软件中运行，而另一个提供器在硬件中运行 ;或一个与平台无关，而另一个与指定的平台相关，这种相互作用性也可以得到实现。界面设计也可以得到扩展，因为可轻易添加与所支持的服务类相匹配的新算法。

图7-1描述了JCA结构的大体轮廓。遵循前面提出的两个设计原则， JCA为密码软件市场带来了巨大的利益。一方面，对于软件开发者来说，可以只关心一套 API，而不用理会采用何种算法，安装哪一种提供器程序包。另一方面，密码工具包，或是数据库可以在智能属性上 (例如已模块化的算法和技术)达到相互一致，从而使性能得以优化，在 JCAAPI中，可保持各种实现的完全相互作用性。

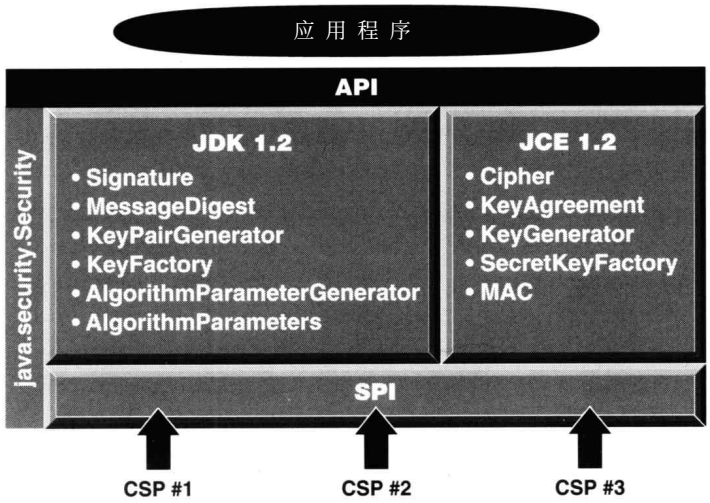


图7-1 JCA结构

7.2 密码服务和服务提供器

JCA介绍了CSP的概念。在JCA1.1中，提供器有一种操作，包含一个或多个签名算法、消息摘要算法、密钥产生算法。在JDK1.2中，又增加了五种服务：

- 密钥工厂。
- 密钥库创建与密钥管理。
- 算法参数管理。
- 算法参数产生。
- 证书工厂。

JDK1.2中也提供了一种随机数产生 (RNG)算法。

在JDK的典型安装中，安装了一个或者几个提供器程序包。用户可以通过静态或动态的方式，添加新的提供器。每一提供器以唯一的名称被引用。用户可以用不同的提供器来配置

China-pub.com
下载

自己的运行时间，并指定优先级次序。 JCA提供了一套API。允许用户查询安装了哪些提供器及它们所提供的服务，如果一个应用程序要求一个指定的提供器。那么从提供器中仅返回对象。如果指定的提供器不存在，那么只能使用缺省的提供器。

当多个提供器同时可用时，那么将建立一个优先级次序。这个次序也是查找请求的服务的次序。当最高级提供器中没有提供请求服务时，那么将依照优先级次序查询下一个提供器。

例如，假设在你的JVM环境中已安装了两个提供器： Provider1和Provider2，进一步假设，Provider1实现带有DSA SHA1和MD5算法，而Provider2实现带有DSA SHA1、RSA MD5 及MD5算法。

如果Provider1的优先级次序是 1(最高优先级)， Provider2的优先级次序是 2，那么有可能遇到下面几种情况：

(1) 如果你想实现MD5，两个提供器都提供这样的实现，由于 Provider1拥有最高优先级，因此最先被查找，所以返回 Provider1的实现。

(2) 如果你想实现RSA MD5 ，优先查找Provider1，但没有发现这样的实现，于是依序查找Provider2，实现被发现并返回。

(3) 如果你想实现 RSA SHA1，两个提供器都没有安装该实现，于是出现 NoSuchAlgorithmException。

Java运行时间环境Sun版本，提供一个缺省的提供器 SUN，其余的Java运行时间环境没有必要提供这个SUN提供器。SUN提供器程序包包含下列实现：

- NIST FIPS186中描述的数字签名算法(DSA)。
- MD5(RFC 1321)和SHA-1(NIST FIPS180-1)消息摘要算法。
- DSA 密钥对产生器能产生与DSA算法相匹配的公钥与私钥对。
- DSA 算法参数产生器。
- DSA 算法参数管理器。
- DSA 密钥工厂，能够在(不透明的)DSA私钥及公钥对象与基本的密钥材料之间，提供一种双向转换。
- 专有SHA1PRNG 伪随机数产生算法，该算法出自 IEEE P1363 标准建议书。
- X.509 证书和CRL的证书工厂。
- 专有密钥库类型JKS。

服务类以一种抽象方式定义密码服务，而且没有具体的实现，密码服务总是与一特定的算法相关联，它可以提供下列实现：

- 密码操作(如数字签名和消息摘要)。
- 密码操作所需的加密材料(密钥和参数)。
- 数据对象(密钥库和证书)，以一种安全方式压缩用于加密操作的密钥。

服务类的一个例子 —— Signature，提供对 DSA功能的访问。 DSA密钥工厂通过调用DSASignature对象中的initSign或initVerity方法，提供DSA公钥或私钥。

程序员可以请求并使用服务类的实例,执行相应的操作， JDK1.2中定义了下列服务类：

- MessageDigest:用来记录指定数据的消息摘要。
- Signature:签署数据并验证数字签名。
- KeyPairGenerator：产生与指定算法相匹配的公钥和私钥 s。

China-pub.com 下载

- KeyFactory：将类型为Key 的模糊密钥转换为清晰密钥。反之亦然。
- CertificateFactory:创建公钥证书和CRL。
- KeyStore：创建和管理密钥库 。
- Algorithm Parameter：管理特定算法的参数，包括参数的编码和解码。
- Algorithm Parameter Generator：产生与指定算法相匹配的一系列参数。
- SecureRandom: 产生随机或伪随机数。

在JCA语境中，产生器和工厂的不同点在于前者以新的内容创建一个对象，而后者是从已有材料(如编码)中创建对象。

服务类为指定类型的密码服务的功能提供了界面，这种服务独立于特定的加密算法。它定义了API，使应用程序能够访问指定的加密服务。对于一个或多个提供器而言，真正的实现是它们各自的算法。

服务类提供的应用程序界面由SPI(服务提供器界面)实现。也就是说，对于每一个服务类，都有一个抽象SPI类，由它来定义密码服务提供器必须实现的 SPI方法。

例如，Signature服务类提供访问DSA功能的服务，由SignatureSpi子类提供的真正 实现将针对指定类型的签名算法，如带有 DSA SHA1、 RSA SHA1或RSA MD5。

服务类的一个实例 —— API对象(和与之对应的SPI类) —— 通过调用服务类中的getInstance( )方法来创建该实例(作为一个私有域)被压缩对应SPI类和SPI对象的实例。所有API对象中的API方法最终都会被声明，并且它们的实现激活了被压缩的 SPI对象中相应的SPI方法(见图7-2)。

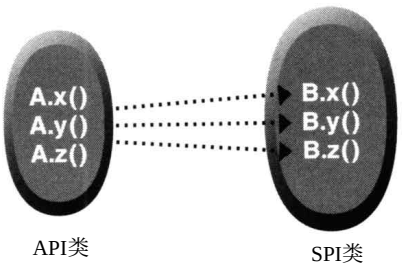


图7-2 服务类中的API和SPI

每个SPI类的名字与相应的服务类的名字相一致，例如，与 Signature服务类对应的SPI类的名称是Signature Spi。SPI类是抽象类。对于一个指定的算法来说，为了提供一个特殊类型服务的实现，提供器必须使相应的 SPI类成为子类，并为所有的抽象方法提供实现。

另一个服务类的例子是 MessageDigest类。它提供访问消息摘要算法。在 MessageDigest Spi子类中,它的实现实质上是各种消息摘要算法，如 SHA-1、MD5或MD2。

最后一个例子是 KeyFactory服务类，提供从模糊密钥到清晰密钥类型的转换。在KeyFactorySpi子类中，真正的实现是密钥的指定类型，如 DSA公钥和私钥。

JCA CSP提供各种密码服务实现，如 SUN。提供器也可以实现不可用的服务，例如基于RSA的签名算法或者MD2消息摘要算法。

安装和添加提供器

假如你是一个用户，为了得到一个合适的对象，如 Signature对象，你可以调用 Signature类中的getInstance方法,并指定一个签名算法 (如:带有DSA SHA1)和能提供所需实现的提供器

China-pub.com
下载

例如，政府机关可能需要接收政府验证的实现。

如果你想使用的提供器没有安装，即使在其他已安装的提供器中包含有所需的实现算法。那么也会出现NoSuchProviderException，所以必须安装所需要的提供器。这就涉及到了安装和配置提供器程序包的问题。可以采用以下两种方法安装提供器类：

- 将包含提供器类的ZIP和JAR文件放在CLASSPATH下。
- 将提供器JAR文件作为已安装和绑定的文件扩展。

下一步，可以用动态或静态的方法将提供器添加到已批准的提供器列表中。如果在静态条件下实现这一点，可以编辑Java安全属性文件，可以设置的一个属性是：

```
security.provider.n=masterClassName
```

这一设置定义了一个提供器，并指定了它的优先级次序 n。提供器优先级次序也就是查找请求的算法的次序。

masterClassName指定提供器的主类(master class)，这一点在提供器的资源文件中设置，此类总是Provider类的子类。它的构造器设置各种属性，在 JCA的API中,提供器通过属性查找请求的算法和其他设施。假设主类是COM.abcd.provider.Abcd，为了将Abcd的优先级设置为3，可以在安全属性文件中加入一行：

```
security.provider.3=COM.abcd.provider.Abcd
```

也可以在Security类中,通过调用addProvider或insertProviderAt方法，动态地注册提供器。注册类型也不是一成不变的，但只能由被赋予充足许可权的可信程序来完成。例如，下面的策略表明，只有从本地文件系统中目录 /home/sysadmin/下署名为JAR文件加载的代码，可以调用Security类中的方法，实现添加和删除提供器或者设置安全属性。 (通过使用用户密钥库中别名为sysadmin的公钥 ，来验证JAR文件的签名)。

```
grant signedBy "sysadmin", codeBase "file:/home/sysadmin/*" {
   permission java.security.SecurityPermission
                  "Security.insertProvider.*";
   permission java.security.SecurityPermission
                  "Security.removeProvider.*";
   permission java.security.SecurityPermission
                  "Security.setProperty.*";
};
```

7.3 密码类

本节将讨论JCA核心类的设计和用法。

7.3.1 java.security.Security

Security类管理已安装的提供器和安全方面的属性。它仅包含静态方法，因此不能用具体的例子来说明。

```
public static String getAlgorithmProperty(String algName,String propName)

public static int insertProviderAt(Provider provider, int position)
```

China-pub.com 下载

```
public static int addProvider(Provider provider)
public static void removeProvider(String name)
public static Provider[] getProviders()
public static Provider getProvider(String name)
public static String getProperty(String key)
public static void setProperty(String key, String datum)
```

getProviders()方法返回一个包含所有已安装提供器的数组，提供器在数组中的次序对应着它们的优先级次序。addProvider方法将一个提供器加到已安装提供器列表的结尾，并返回提供器的优先级位置值。如果这个提供器已经被安装，那么再次添加是无效的，返回值是－1。

insertProvider方法在搜索请求算法的提供器中指定一个位置，并插入新的提供器。(如果没有指定提供器)。如果这个提供器已存在，那么不能再次添加。如果给定的提供器安装在指定的位置，那么原来位于这个位置和优先级小于此位置的提供器依次下移，此方法返回值是所增加提供器的优先级。如果提供器已存在，那么返回值为－1。

removeProvider方法删除已命名的提供器，如果已命名的提供器没有安装，那么将返回缺省值。如果指定的提供器被删除，那么所有优先级低于该提供器的提供器依次上移。如果要改变某个已安装提供器的优先级位置，应当首先删除它，然后再将其插入到新的优先级位置。

7.3.2 java.security.Provider

每个Provider类实例都有自己的名字、版本号及字符串描述和它所提供的服务。可以通过调用下列方法来查询Provider实例:

```
public String getName( )
public double getVersion( )
public String getInfo( )
```

7.3.3 java.security.MessageDigest

MessageDigest类是一种提供密码安全消息摘要，如 SHA1 MD5功能的服务类。密码安全消息摘要采用任意长度的输入，然后产生一个固定长度的输出，称为一个摘要或散列。因为两个不同消息的散列值不可能相同，因此，从消息摘要中不可能得到生成它的输入，故消息摘要有时又被称为数据的“数字水印”。

为了计算消息摘要，首先应创建一个消息摘要实例，与所有的服务类一样，对于特定类型的消息摘要算法而言，可通过调用 MessageDigest类中的getInstance静态方法得到Messsage Digest对象:

```
public static MessageDigest getInstance(String algorithm)
```

算法名对大小写不敏感，例如，下列调用方法都是相等的：

```
MessageDigest.getInstance("SHA")
MessageDigest.getInstance("sha")
MessageDigest.getInstance("sHa")
```

调用者可以任意指定提供器名，但必须保证 请求算法的实现出自己命名的提供器：

```
public static MessageDigest getInstance(String algorithm,String provider)
```

调用getInstance方法，返回一个被初始化的消息摘要对象，因此不需要进一步初始化对象。

下一步，为计算一些数据的摘要，应当向被初始化的消息摘要对象提供数据。这一点可

China-pub.com
下载

通过调用下列已更新的方法来实现。

```
public void update(byte input)
public void update(byte[ ] input)
public void update(byte[ ] input,int offset,int len)
```

通过调用更新的方法提供数据后，可运用下列摘要方法计算摘要：

```
public byte[ ] digest( )
public byte[ ] digest(byte[ ],input )
public int digest(byte[ ] buf, int offset ,int len)
```

前两个方法返回已计算的摘要，第三个方法将计算得到的摘要存储在提供器的 buf缓存中，从offset开始,len是分配给摘要的字节数。上述方法返回存储在 buf中的真正字节数。调用以byte数组参数作为输入的摘要方法，等效于在无参数的分类方法调用之后再调用指定输入的public void updata(byte[] input)。

7.3.4 java.security.Signature

Signature类是一种提供密码数字签名算法，如 DSA SHA1和RSA MD5功能的服务类。密码安全性签名算法，采用任意长度数据和私钥作为输入，产生一个固定长度的字符串 —— 签名，签名具有下列属性：

(1) 当提供与产生签名的私钥相应的公钥时，就可以验证输入的可靠性和完整性。

(2) 签名和公钥不能揭露任何私钥信息。

Signature对象可以用来签署数据，通常也可以用来验证签名是否是相关数据的合法签名。Signature对象属于模式(modal)对象。也就是说,Signature对象总是处于某个特定状态,对它只能进行特定类型的操作。对象所处状态在它们各自的类中 (如Signature)以最终整型常量表示。通常一个签名对象有三种状态:

(1) UNINITIALIZED(未初始化)。

(2) SIGN(签名)。

(3) VERIFY(验证)。

为了签名和验证签名，首先创建 Signature实例。正如所有服务类一样 ,通过调用 Signature类中的getInstance 静态方法可以获得具有特定签名算法的 Signature对象。

```
public static Signature getInstance(String algorithm)
public static Signature getInstance(String algorithm,String provider)
```

Signature对象在创建时处于UNINITIALIZED状态，所以在使用前，必须先对其进行初始化。Signature类定义了两种初始化方法 initSign和initVerify，它们可以分别将 Signature对象的状态转换成SIGN和VERIFY状态。到底采用哪一种方法 ,主要取决于是想对 Signature对象进行签名操作还是验证操作。如要进行签名操作，需用产生签名的实体的私钥初始化对象，这可通过调用initSign方法实现：

```
public final void initSign(PrivateKey privateKey)
```

调用这种方法使Signature对象处于SIGN状态。

如要进行验证操作 ,需用产生签名的实体的公钥初始化对象，这可通过调用 initVerify方法实现：

```
public final void initVerify(PublicKey publicKey)
```

调用这种方法使Signature对象处于VERIFY状态。

China-pub.com 下载

如果Signature对象已经进行了签名初始化 (已经处于SIGN状态)，就可以将待签署的数据传递给对象，这可通过调用一次或多次update方法之一来实现：

```
public final void update(byte b)
public final void update(byte[] data)
public final void update(byte[] data,int off,int len)
```

对update方法的调用一直持续到所有待签署的数据全部传递给了 Signature对象。

为了产生签名，只需简单地调用 sign方法：

```
public final byte[] sign()
public final int sign(byte[] outbuf,int offset,int len)
```

第一种方法以byte数组形式返回签名。第二种方法返回的签名结果存储在设定的缓冲区outbuf中，offset表示缓冲区的初值，len表示outbuf的长度，这种方法返回的是实际存储的字节数。签名可被带有两个整型参数r、s的标准ASN.1序列编码。

调用sign方法可以重置Signature对象状态，使它返回到调用 initSign方法前的状态。也就是说Signature对象被重置，并且可以通过重新调用 updata和sign，用同一个私钥产生另一个不同的签名。相对地，可以重新调用 initSign指定不同私钥，或者重新调用 initVerify初始化Signature对象验证签名。

如果Signature对象已经进行了验证初始化 (已经处于VERIFY状态)，则可以通过向对象提供验证数据(这个过程与签名过程相反)来验证签名是否是相关数据的合法签名。这可以通过一次或多次调用update方法之一来实现：

```
public final void update(byte b)
public final void update(byte[] data)
public final void update(byte[] data,int off,int len)
```

对update方法的调用一直持续到向Signature对象提供了所有的验证数据为止。

可以通过调用verify方法来验证签名：

```
public final boolean verify(byte[] encodedSignature)
```

参数必须是包含签名的byte数组，该签名可被带有两个整型参数r、s的标准ASN.1序列编码。这是一种常用的标准编码。这与 Sign方法产生的结果相同。 verify方法返回一个boolean，以指示编码的签名是否是update方法所提供数据的合法签名。

调用verify方法可以重置Signature对象状态，使它返回到调用 initVerify方法前的状态。也就是说签名对象被重置，并且可以验证身份的另一个签名，该身份的公钥在调用 initVerify时指定。相对地，重新调用 initVerify以不同公钥初始化 Signature对象来验证不同实体的签名，或者重新调用initSign初始化Signature对象来产生签名。

7.3.5 算法参数

JCA可用来处理密码算法。不同的算法有很大的差异。每一种算法都有自己的参数 (密钥大小和被定义常数)。为了组织这些参数,每种算法都有一种算法参数规格说明,这些说明被分成一些小的类集。

算法参数规格说明是算法所用参数集的清晰 (transparent)表示。这意味着可以独立访问每一个参数，虽然相应的规格说明类中定义了 get方法(如,DSAParameterSpec定义了getp、getQ、getG方法,它们可以分别访问 p、q、g参数)。相反的，在模糊 (opaque)表示中,正如AlgorithmParameters类所提供的,不能直接访问参数域。不过可以得到与参数集有关的算法名

China-pub.com
下载

(调用getAlgorithm)和参数集的编码方法(调用getEncoded)。可以调用getParameterSpec方法,把AlgorithmParameters对象转换成清晰说明。

算法参数规格说明界面和类

下面讨论java.security和java.scurity.spec包中的算法参数规格说明界口和类：

- AlgorithmParameterSpec
- DSAParameterSpec
- AlgorithmParameters
- AlgorithmParameterGenerator

1. AlgorithmParameterSpec

对于密码参数的清晰说明，AlgorithmParameterSpec是一种基本界面,它不含方法或常数。它的唯一目的是对所有参数说明分组。所有参数说明必须实现这个界面。

2. DSAParameterSpec

这个类指定了DSA算法所用的参数集，用以实现AlgorithmParameter界面。它有三种方法:

```
public BigInteger getP()
public BigInteger getQ()
public BigInteger getG()
```

这三种方法分别返回DSA算法参数:主要参数p、次要参数q、基数g。

3. AlgorithmParameters

这种服务类提供密码参数的模糊表示。像所有服务类一样，通过调用 getInstance静态方法可以获得特定算法类的对象，并且可以有选择地指定提供器名，以确保算法参数实现来自命名的提供器。

```
public static AlgorithmParameters getInstance(String algorithm)
public static AlgorithmParameters getInstance(String algorithm,
                                  String provider)
```

一旦AlgorithmParameters 对象实例化后，必须使用合适的参数说明及编码调用 init进行初始化。

```
public void init(AlgorithmParameterSpec paramSpec)
public void init(byte[] params)
public void init(byte[] params,String format)
```

其中，params是含有编码参数的数组， format是解码格式名。在含有params参数,而没有format参数的init方法中，使用基本解码格式。如果 ASN.1参数说明存在，则基本的解码格式是ASN.1。注意:AlgorithmParameter对象只能进行一次初始化,即不能重用。

在AlgorithmParameters对象中,通过调用getEncoded方法可获得字节型参数编码：

```
public byte[] getEncoded()
```

其返回值是基本编码格式的参数。

为了获得指定编码格式参数,可以调用带有format参数的getEncoded方法：

```
public byte[] getEncoded(String format)
```

如果format参数为空，正如其他getEncoded方法一样，使用基本编码格式㊀

㊀ 由SUN提供器提供的AlgorithmParameters缺省实现中，格式参数通常被忽略。

China-pub.com 下载

算法参数的清晰说明可以通过调用 AlgorithmParameters对象的getParameteSpec方法获得：

```
public AlgorithmParameterSpec getParameterSpec(Class paramSpec)
```

paramSpec指定将返回的参数的说明类。例如，DSAParameterSpec.class,将指明DSAParameterSpec的实例返回的参数。

4. AlgorithmParametersGenerator

这种服务类在创建AlgorithmParameter Generator实例后,可以产生指定算法的参数集。为了获得特定算法的AlgorithmParameterGenerator对象,可以调用AlgorithmParameterGenerator类中的getInstance 静态方法：

```
public static AlgorithmParameterGenerator getInstance(String algorithm)
public static AlgorithmParameterGenerator getInstance(String algorithm,
                                                       String provider)
```

AlgorithmParamerGenerator 对象可用下列两种方式之一进行初始化：

• 算法无关方式。

• 特定算法方式。

算法无关方式基于所有参数产生器共享随机数源和大小这一事实。虽然所有算法参数都用到大小这一概念,但是不同算法的大小含义不同,例如，DSA算法中,大小以位的形式表示基本模数。使用这种方法后,特定算法方式参数产生值缺省为一些标准值。

init方法使用这两个广泛应用的参数类型。另一种 init方法则只使用size参数，其random是系统缺省值。

```
public void init(int size,SecureRandom random);
public void init(int size)
```

在特定算法方式中，参数产生器对象必须采用特定算法语句进行初始化，该语句可由AlgorithmParameterSpec 对象中特定算法参数产生值表示。

```
public void init(AlgorithmParameterSpec genParamSpec,SecureRandom random)
public void init(AlgorithmParameterSpec genParamSpec)
```

在系统参数产生过程中，如 Diffie-Hellman方案中，参数产生值通常包含基本模数的大小和随机指数的大小，两者均以位的形式表示。 Diffie-Hellman的算法是JCE1.2的一部分,不在JDK范围之内。

一旦创建并初始化了AlgorithmParameterGenerator对象，则可以调用generateParameters方法产生算法参数：

```
public AlgorithmParameters generateParameters()
```

7.3.6 java.security.Key和java.security.spec.KeySpec

本节讨论下面的界面和子界面：

• Key

• PublicKey

• PrivateKey

• KeySpec

1. Key

Key是所有模糊密钥中最高层的界面，它定义了所有模糊密钥对象的共享功能。模糊密钥

China-pub.com
下载

具有以下三个特性：

- 算法 —— 密钥算法。
- 编码形式。
- 格式。

密钥算法指的是像DSA或RSA等与密钥有关的算法,它们结合RSA MD5和RSA SHA1等算法一起工作。密钥算法名称可通过调用 get方法获得：

```
public String getAlgorithm()
```

编码形式是密钥的外部编码形式 ,当JVM外部(如发送密钥到第三方时)需要密钥的标准表示时，要用到编码形式。密钥按照标准格式 (如X.509或PKCS#8)编码，可通过调用 get方法得到编码形式：

```
public byte[] getEncoded()
```

格式是经过编码后的密钥格式，可以通过调用 get方法获得：

```
public String getFormat()
```

密钥可以通过密钥产生器、证书、密钥说明 (使用KeyFactory)或KeyStore实现等访问管理密钥的密钥库来获得。利用 KeyFactory，可以用特定算法方式分析已编码密钥。同样，也可以用CertificateFactory分析证书。

2. PublicKey和PrivateKey

PublicKey和PrivateKey界面均是对Key界面的扩展。它们不提供方法，只为了实现类型安全和类型辨识。

密钥说明是密钥材料的清晰表示。如果密钥存储在硬件设备中，它的说明可能包含辨识密钥的信息。密钥的清晰意味着通过调用说明类中的 get方法可以分别访问每一个密钥材料值。例如：DSAPrivateKeySpec中的getX、getP、getQ、getG方法可以访问私钥x和用于计算密钥的DSA算法参数：主要参数p、次要参数q和基数g。

密钥可以用指定算法方式或算法无关方式编码格式 (如ASN.1)规定。如：DSA私钥可由参数x、p、q规定，也可以用DER编码规定。

3. KeySpec

KeySpec界面不提供方法和常量。它唯一的目的是为所有密钥说明提供类型安全。所有密钥说明必须实现这个界面。

- DSAPrivateKeySpec。此类实现KeySpec界面并借助相关参数规定了 DSA私钥。它提供下列方法返回私钥x，使用如下DSA算法参数：主要参数p、次要参数q和基数g。

```
public BigInteger getX()
public BigInteger getP()
public BigInteger getQ()
public BigInteger getG()
```

- DSAPublicKeySpec。此类实现KeySpec界面并借助相关参数规定了 DSA公钥。它提供下列方法返回公钥y，使用如下DSA算法参数：主要参数p、次要参数q和基数g。

```
public BigInteger getY()
public BigInteger getP()
public BigInteger getQ()
public BigInteger getG()
```

- RSAPrivateKeySpec。此类实现KeySpec界面并借助相关参数规定了 RSA私钥。它提供

China-pub.com 下载

下列方法返回组成RSA私钥的模数n和私有指数d。

```
public BigInteger getMdulus()
public BigInteger getPrivateExponent()
```

- RSAPrivateCrtKeySpec。此类是RSAPrivateKeySpec类的一种扩展，它以CRT信息值规定一个RSA私钥(如PKCS#1标准中所定义的)。除了从其子类RSAPrivateKeySpec中继承的方法外，还提供下列方法：

```
public BigInteger getPublicExponent()
public BigInteger getPrimeP()
public BigInteger getPrimeQ()
public BigInteger getPrimeExponentP()
public BigInteger getPrimeExponentQ()
public BigInteger getCrtCoefficient()
```

这些方法分别返回公共指数 e和CRT信息整数：模数n的主要因子p、n的主要因子q，指数d模(p-1)指数d模(q-1)和CRT系数模p。在逻辑上， RSA私钥只包含模数和私有指数，为了实现高效性，才引入CRT值。

- RSAPublicKeySpe。此类实现KeySpec界面并规定RSA公钥。它提供下列方法，其返回值为组成RSA公钥的模数n和公有指数e。

```
public BigInteger getModulus()
public BigInteger getPublicExponent()
```

- EncodedKeySpec。此类实现 KeySpec界面并以编码格式规定公钥或私钥。EncodedKeySpec类中的getEncoded、getFormat方法分别返回编码密钥和编码格式名。

```
public abstract byte[] getEncoded()
public abstract String getFormat()
```

- PKCS8EncodedKeySpec。此类是EncodedKeySpec类的子类，表示符合PKCS#8标准规定格式的私钥的DER编码。getEncoded方法返回符合PKCS#8标准经过编码的密钥字节，getFormat方法返回字符串PKCS#8。X509EncodedKeySpec类是EncodedKeySpec类的子类，表示符合X.509标准规定格式的私钥或公钥的DER编码。getEncoded返回符合X.509标准经过编码的密钥字节， getFormat方法返回字符串X.509。

7.3.7 java.security.KeyFactory和java.security.cert.CertificateFactory

本节回顾生成密钥和证书的工厂类。

1. KeyFactory

KeyFactory类是一种(基本密钥材料的清晰表示)模糊密钥(Key类型)和密钥说明转换的服务类。密钥工厂是双向的，也就是说，可以从给定的密钥材料建立模糊 Key对象，也可以从Key对象获知密钥材料。

对同一个密钥可能有多个兼容的密钥说明。例如， DSA公钥可由y、p、q和g说明，也可根据X.509标准由DER编码说明。密钥工厂可实现兼容的密钥说明间的相互转换，即密钥分析。例如，将X509EncodedKeySpec转换为DSAPublicKeySpec，表示将已编码密钥分析为部件。

通过调用 KeyFactory类中的 getInstance静态工厂方法可以获得特定类型密钥算法的KeyFacrory对象。

```
public static KeyFactory getInstance(String algorithm)
public static KeyFactory getInstance(String algorithm,String provider)
```

China-pub.com
下载

调用者通过指定提供器的名称，可保证密钥工厂的实现来自 KeyFactory的命名提供器。

在给定私钥或公钥的密钥说明中，调用 generatePublic或generatePrivate方法，可以分别获得模糊PublicKey或PrivateKey对象。

```
public PublicKey generatePublic(KeySpec Keyspec)
public PrivateKey generatePrivate(KeySpec Keyspec)
```

相反地，对于给定Key对象，调用getKeySpec方法可以获得相应的KeySpec对象。

```
public KeySpec getKeySpec(Key key, Class KeySpec)
```

KeySpec确定返回的密钥材料的说明类。例如，如果 KeySpec是DSAPublicKeySpec.class，就说明密钥材料将以DSAPublicKeySpec类的实例形式返回。

2. CertificateFactory

CertificateFactory类是一个定义证书工厂功能的服务类。它可以用来从编码中产生证书和CRL对象。X.509证书的证书工厂的返回值是证书或 CRL对象，它们分别是java.security.cert.X509Certificate和java.security.cert.X509CRL的实例。

- 对于证书或CRL类型，为了获得一个CertificateFactory对象，可在CertificateFactory类中调用getInstance静态工厂方法：

```
public static CertificateFactory getInstance(String type)
```

- 为了指定一个提供器，可使用 getInstance方法：

```
public static CertificateFactory getInstance(String type, String provider)
```

- 为了产生证书对象并且用从一个输入流中读取的数据对它初始化，可使用 generate Certificate方法：

```
public final Certificate generateCertificate(InputStream is)
```

- 为了返回一个从给定输入流中读取的证书的集合 (可能为空)，可使用generateCertificates方法：

```
public final Collection generateCertificates(InputStream is)
```

- 为了产生一个CRL对象并且用从输入流中读取数据对它初始化，可使用 generateCRL方法：

```
public final CRL generateCRL(InputStream is)
```

- 为了返回一个从给定输入流中读取的 CRL集合(可能为空)，可使用generateCRLs方法。

```
public final Collection generateCRLs(InputStream is)
```

7.3.8 KeyPair和KeyPairGenerator

KeyPair类是一个密钥对 (一个公钥和一个私钥)的持有者。它有两个公用的方法，分别用来返回公钥和私钥：

```
public PrivateKey getPrivate()
public PublicKey getPublic()
```

KeyPairGenerator类是一个用来产生公私钥对的服务类。产生过程可以是算法无关方式的，也可以有指定算法方法的，这取决于对象如何初始化。

所有的密钥对产生都开始于一个 KeyPairGenerator。这是一个有特定算法的密钥对产生器，它可以使用该算法创建一个公钥私钥对，该生成器也把指定算法的参数与每一个生成的公钥私钥对相关联。可使用下列工厂方法之一来创建一个 KeyPairGenerator：

China-pub.com 下载

```
public static KeyPairGenerator getInstance(String algorithm)
public static KeyPairGenerator getInstance(String algorithm, String provider)
```

密钥产生成器在生成密钥之前需要进行初始化。在多数情况下，算法无关方式的初始化就已经足够了。所有的密钥对产生器都包括两个概念：随机源和密钥大小。密钥大小对不同的算法有不同的解释。例如，在DSA算法中，大小指的是模数的长度。

initialize方法使用这两个广泛应用的参数类型，而另一种 initialize方法由于使用系统提供的随机源，所以只使用一个密钥大小参数。

```
public void initialize(int keysize)
public void initialize(int keysize, SecureRandom random)
```

既然当调用上面的算法无关的initialize方法时，没有指明其他的参数，那么提供器就必须确定与每一个密钥相关的指定算法参数。例如，如果算法是 DSA且模数大小(密钥大小)是512、768或1024，那么SUN提供器为p、q和g参数提供一组预先计算好的参数。如果模数大小不包含在这些值之中， SUN提供器就创建一组新的参数。其他的提供器有可能预先计算好多组参数，而不仅仅是这里提到的三个模数大小。其他的提供器也有可能根本就没有预先计算好的参数列表，而是每次创建新的参数组。

在某些情况下，需要一个指定算法的初始化，例如，当存在一组指定算法参数时 (例如DSA中所谓的“公用”参数)。两个initialize方法使用了一个AlgorithmParameterSpec参数。其中之一在它的随机源由系统提供的情况下，不使用 SecureRandom参数。

```
public void initialize(AlgorithmParameterSpec params)
public void initialize(AlgorithmParameterSpec params, SecureRandom random)
```

为了产生一个密钥对，从KeyPairGenerator中调用下列方法：

```
public KeyPair generateKeyPair()
```

多次调用generateKeyPair可产生不同的密钥对。

7.3.9 java.security.KeyStore

KeyStore类定义了访问和修改密钥库中信息的界面。第 4章讲述用来管理密钥和证书的密钥库，并且通过使用keytool来演示密钥库的用途。本节讨论KeyStore的API设计和实现。

KeyStore由keytool、jarsigner和policytool使用。当它处理策略文件时，也可由缺省的Policy实现来使用。JDK用户可以写出附加的、使用或扩展KeyStore的安全应用程序。

多种不同的实现都是有可能的，每一个实现针对一个特殊的密钥库类型。例如，一个实现可能提供连续的密钥库，而另一个实现可以使用智能卡。因此不同类型的密钥库实现并不意味着互相兼容。

KeyStore实现基于提供器。相应的抽象 KeystoreSpi类指明了SPI界面。Provider类通常是KeystoreSpi的子类。JDK包含一个缺省的密钥库实现，这个实现属于一个名为 JKS的专有密钥库类型(格式)。

KeyStore表示内存中的一个密钥和证书的集合，并且管理两种类型的输入项：密钥输入项和可信证书输入项。为了创建一个 KeyStore对象，调用KeyStore类中的getInstance静态工厂方法，并且可指定提供器名：

```
public static KeyStore getInstance(String type)
```

China-pub.com
下载


```
public static KeyStore getInstance(String type, String provider)
```

在使用一个KeyStore对象之前，实际的密钥库数据必须通过load方法加载入内存：

```
public final void load(InputStream stream, char[] password)
```

可选的password可以用来核查密钥库数据的完整性。如果没有提供口令，也就不会进行完整性核查。如果想建立一个空的密钥库，可以把空值作为 InputStream参数加入load方法。

Keystore中的每一个输入项有一个唯一的别名串鉴别，密钥库中提供的别名列表可通过以下方法获得：

```
public final Enumeration aliases()
```

下列方法确定由给定别名指定的输入项是一个密钥输入项还是一个可信证书输入项：

```
public final boolean isKeyEntry(String alias)
public final boolean isCertificateEntry(String alias)
```

下列方法对密钥库的内容进行操作：

```
public final void setCertificateEntry(String alias,
                                Certificate cert)
public final void setKeyEntry(String alias, byte[] key,
                                Certificate[] chain)
public final void setKeyEntry(String alias, Key key,
                char[] password, Certificate[] chain)
public final void deleteEntry(String alias)
public final Key getKey(String alias, char[] password)
public final Certificate getCertificate(String alias)
public final Certificate[] getCertificateChain(String alias)
public final String getCertificateAlias(Certificate cert)
public final void store(OutputStream stream, char[] password)
```

7.4 随机性和种子产生器

密码学的一个基本概念就是随机数字的产生。这是因为在密码学中随机性是安全性的源泉，并且在产生密钥和提供唯一鉴别(例如在难题-响应协议中)时,非常有用(并且有时是最根本的)。

随机数字产生器的基类是 JDK1.0中介绍的java.util.Random。产生器实际上并不产生纯粹的随机种子，而是产生伪随机数。

Random使用一个48位的种子，通过使用一个、线性同余公式来修改该种子 [37]。以下是它的界面：

```
public Random()
public Random(long seed)
void setSeed(long seed)
protected int next(int bits)
boolean nextBoolean()
vid nextBytes(byte[] bytes)
double nextDouble()
float nextFloat()
double nextGaussian()
int nextInt()
int nextInt(int n)
```

China-pub.com 下载

```
long nextLong()
```

可以构造一个Random对象，并且在构造器里或者通过 setSeed方法给这个对象分配一个种子。如果没有清楚地分配种子，那么种子将会是一个基于对象创立时间的缺省值。对象初始化之后，可以调用不同的 next方法以获取下一个随机数字。产生器是确定性的，这是因为如果两个Random实例以相同的种子创建并且如果每个实例以相同的序列来调用方法，那么这两个实例将会生成并返回相同序列的数字。允许 Random的子类使用其他的算法。

对安全性有经验的读者到现在为止可能已经注意到，无论是缺省的种子产生方案还是子序列数字产生算法都不能产生安全性应用程序通常所要求的不可预测性的数字。这就是为什么需要java.security.SecureRandom —— 它提供了一个密码的伪随机数字产生器 (PRNG)。我们将在以后讨论这个类。

java.security.SecureRandom

就像JDK中其他的基于算法的类一样， SecureRandom类提供了与实现无关的算法，应用程序需要一个特殊的 PRNG算法并且为算法返回一个 SecureRandom对象。它也可以从一个特定的提供器那里申请一个特殊的算法。例如，缺省的提供器 SUN支持一个名为SHA1PRNG的组合算法。

```
public static SecureRandom getInstance(String algorithm)
public static SecureRandom getInstance(String algorithm, String provider)
public SecureRandom()
public SecureRandom(byte[] seed)
public void setSeed(byte[] seed)
public void setSeed(long seed)
protected final int next(int numBits)
public static byte[] getSeed(int numBytes)
```

使用getInstance是获取SecureRandom对象更好的方法，尽管仍然可以调用公用构造器。如果调用了这些构造器，那么将会使用缺省算法的缺省提供器。

SecureRandom实现试图使产生器自身的内部状态彻底随机化。然而，播种过程直到第一次需要随机输出时才会发生 (也就是当调用 nextBytes时)。因此，通过在 getInstance后调用setSeed，调用者可以明确给SecureRandom对象提供种子，如下例所示：

```
SecureRandom random = SecureRandom.getInstance("SHA1PRNG");
random.setSeed(seed);
```

一旦SecureRandom对象已经被播种,该对象就可以产生像最初种子一样的随机位。任何时候都可以使用setSeed方法来对 SecureRandom对象重新播种。新赋予的种子补充而不是代替已存在的种子，因此重复调用不会减少随机性。 SecureRandom自身也可以有助于种子的产生，例如，对另一个SecureRandom对象,通过generateSeed方法。

7.5 代码实例

本节提供了几个实例以便进一步阐明如何使用本章所讨论的类。

7.5.1 实例1: 计算消息摘要

第一个例子使用SHA算法来计算消息摘要。假设有一个由三个字节数组： i1,i2和i3构成的

China-pub.com
下载

消息。首先创建一个正确初始化的消息摘要对象，然后通过消息摘要对象来运行这三个字节数组,以便计算散列函数值,如下所示:

```
MessageDigest sha = MessageDigest.getInstance("SHA");
sha.update(i1);
sha.update(i2);
sha.update(i3);
byte[] hash = sha.digest();
```

对digest方法的调用告知输入消息结束。 digest方法也可以把输入的最后一部分作为参数，如下所示:

```
sha.update(i1);
sha.update(i2);
byte[]hash = sha.digest(i3);
```

消息摘要计算结束后,消息摘要类自动复位并且准备接收新的数据及计算其摘要。所有前面的状态(用来更新旧的数据)被丢弃。

在一些散列函数的实现中，可以通过复制来获得中间散列函数值。假设希望计算这张表中三条分离消息各自的散列函数值:

i1

i1和i2

i1、i2和i3

则可以进行以下计算：

```
/* compute the hash for i1 */
sha.update(i1);
byte[] i1Hash = sha.clone().digest();

/* compute the hash for i1 and i2 */
sha.update(i2);
byte[] i12Hash = sha.clone().digest();

/* compute the hash for i1, i2 and i3 */
sha.update(i3);
byte[] i123hash = sha.digest();
```

只有在SHA实现可复制时才可进行这项工作。一种确定是否可复制的方法是复制MessageDigest对象，看看是否会出现可能的异常。

7.5.2 实例2：产生公钥/私钥对

第二个例子为 DSA算法产生一个公钥 /私钥对。通过使用一个用户派生的种子 (名为userSeed)，产生密钥和1024位的模数。首先，为了产生针对 DSA算法的密钥，要获得一个KeyPairGenerator对象。然后，为了初始化 KeyPairGenerator，需要一个从SecureRandom对象获得的随机种子。

```
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA");
SecureRandom random =
      SecureRandom.getInstance("SHA1PRNG","SUN");
random.setSeed(userSeed);
```

China-pub.com 下载

```
keyGen.initialize(1024,random);
```

假设已经有一组具体的DSA参数：p，q和g，你希望用这些参数产生密钥对。然后有差别地初始化密钥对产生器，如下所示：

```
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA");
DSAParameterSpec dsaSpec = new DSAParameterSpec(p,q,g);
SecureRandom random =
      SecureRandom.getInstance("SHA1PRNG","SUN");
random.setSeed(userSeed);
keyGen.initialize(dsaSpec,random);
```

最后,可以产生这个密钥对。

```
KeyPair pair = keyGen.generateKeyPair();
```

7.5.3 实例3：产生并且验证签名

这个例子通过使用例 2中产生的密钥对来产生并验证一个签名。首先，需要创建一个Signature对象。然后，通过使用例2中产生的密钥对，可以利用私钥来初始化对象并且给一个名为data的byte数组签名。

```
Signature dsa = Signature.getInstance("SHA1withDSA");
/*Initializing the object with a private key
PrivateKey priv = pair.getPrivate();
dsa.initSign(priv);
/*Update and sign the data /
dsa.update(data);
byte[] sig=dsa.sign();
```

验证签名是直接的。

```
/*Initializing the object with the public key
PublicKey pub=pair.getPublic();
dsa.initVerify(pub);

/*Updata and verify the data
dsa.update(data);
boolean verifies=dsa.verify(sig);
System.out.println("signature verifies:"+verifies);
```

假设没有一个公钥/私钥对，而只有DSA私钥的部件：x、p、q和g。进一步假设你想要利用私钥来给一些数据作数字化签名，这些数据位于名为 someData的byte数组中。然后使用下面的代码。该例也阐明了如何创建一个密钥说明并且使用密钥工厂从密钥说明中获取Privatekey。

```
DSAPrivateKeySpec dsaPrivKeySpec =
      new DSAPrivateKeySpec(x,p,q,g);
KeyFactory keyFactory = KeyFactory.getInstance("DSA");
PrivateKey privKey =
       keyFactory.generatePrivate(dsaPrivKey-Spec);
Signature sig = Signature.getInstance("SHA1withDSA");
sig.initSign(privKey);
```

China-pub.com
下载


```
sig.update(someData);
byte[] signature=sig.sign();
```

现在假设你的私人代理Alice想要使用由你签名的数据。为了让她能够做到这一点以使她能验证你的签名，你需要给她三件东西：数据、签名和与你用来给数据签名的私钥相对应的公钥。你可以把someData字节存储在一个文件里并且把签名字节存储在另一个文件里，然后把两个文件都发给Alice。对公钥来说，像前面的签名例子那样假设你有DSA公钥的部件，而该公钥与用来给数据签名的 DSA私钥相对应。然后你可以从这些部件里创建一个DSAPublicKeySpec:

```
DSAPublicKeySpec dsaPubKeySpec =
      new DSAPublicKeySpec(y, p, q, g);
```

你仍然需要提取密钥字节以便把它们放到一个文件里。为了做到这一点，首先需要调用generatePublic方法，该方法在例2中已创建的DSA密钥工厂中。然后，提取(已经编码的)密钥字节。

```
PublicKey pubKey = keyFactory.generatePublic(dsaPubKeySpec);
byte[] encKey = pubKey.getEncoded();
```

现在可以把这些字节存储到一个文件里并且把这个文件和包含数据和签名的文件一起发送给Alice。

一旦Alice收到这些文件，她把数据文件中的数据代码拷贝到一个名为 data的byte数组中，其中签名文件中的签名字节拷贝到一个名为 signature的byte数组中，而公钥文件中的已编码公钥字节拷贝到名为encodedPubkey的byte数组中。为了验证签名， Alice运行下面的代码，这些代码使用一个密钥工厂来实例化一个DSA公钥。

```
X509EncodedKeySpec pubKeySpec =
      new X509EncodedKeySpec(encodedPubKey);
KeyFactory keyFactory = keyFactory.getInstance("DSA");
PublicKey pubKey = keyFactory.generatePublic(pubKeySpec);
Signature sig = Signature.getInstance("SHA1withDSA");
sig.initVerify(pubKey);
sig.update(data);
sig.verify(signature);
```

为了访问密钥部件， Alice也可以把PublicKey转换为一个DSAPublicKeySpec。

```
DSAPublicKeySpec dsaPubKeySpec =
      (DSAPublicKeySpec)  keyFactory.getKeySpec(pubKey,DsaPublicKeySpec.class)
BigInteger y = dsaPubKeySpec.getY()
BigInteger p = dsaPubKeySpec.getP()
BigInteger q = dsaPubKeySpec.getQ()
BigInteger g = dsaPubKeySpec.getG()
```

7.5.4 实例4：读取包含证书的文件

本章的最后一个例子为读取一个包含证书的文件。首先，证书是以 64为基编码的。这些证书都有各自的开头和结尾，由

```
-----BEGIN CERTIFICATE-----
```

China-pub.com 下载

和

```
-----END CERTIFICATE-----
```

来限定。

在此过程中，可以把不支持 mark和reset方法的FileInputStream转换为支持这些方法的ByteArrayInputStream。这样可使，每一个对 generateCertificate的调用只消耗一个证书，而输入流的读位置被定位到文件中的下一个证书中。

```
FileInputStream fis = new FileInputStream(filename);
DataInputStream dis = new DataInputStream(fis);
CertificateFactory.cf =
    CertificateFactory.getInstance("X.509");
byte[] bytes = new byte[dis.available()];
dis.readFully (bytes);
ByteArrayInputStream bais = new ByteArrayInputStream(bytes);
while (bais.available() > 0) {
   Certificate cert = cf.generateCertificate(bais);
   System.out.printin(cert.toString());
}
```

下一步，从语法上分析存储在文件中的一个 PKCS#7格式的证书答复，并且从中提取所有的证书。

```
FileInputStream fis = new FileInputStream(filename);
CertificateFactory.cf =
    CertificateFactory.getInstance("X.509");
Collection c = cf.generateCertificates(fis);
Iterator i = c.iterator();
while (i.hasNext()) {
   Certificate cert = (Certificate) i.next();
   System.out.printin(cert);
}
```

7.6 标准名

Java文件或代码、算法、证书以及密钥库类型由具体的名字来指明。这些名字并非随便选择的，而是根据一个已采用的标准。本节列举了这些名字并且揭示了它们的背景。

7.6.1 消息摘要算法

消息摘要算法名可在产生一个 MessageDigest实例时指明。

- SHA　安全散列标准(NIST FIB 180-1)中定义的安全散列算法。
- MD2　RFC1319中定义的MD2消息摘要算法。
- MD5　RFC1321中定义的MD5消息摘要算法。

7.6.2 密钥和参数算法

密钥和参数算法名可在产生一个KeyPairGenerator，KeyFactory，AlgorithmParameterGenerator及AlgorithmParameters实例时指明。

China-pub.com
下载

- RSA PKCS#1中定义的RSA加密算法。
- DSA FIPS PUB 186中定义的数字签名算法。

7.6.3 数字签名算法

DSA数字签名算法名在生成Signature实例时指明

- DSA SHA-1。签名算法，使用SHA-1摘要算法和DSA来创建和验证FIPS PUB 186中定义的DSA数字签名。
- RSA MD2。加密签名算法，使用 MD2摘要算法和RSA来创建和验证PKCS#1中定义的RSA数字签名。
- RSA MD5。此加密签名算法，使用MD5摘要算法和RSA来创建和验证PKCS#1中定义的RSA数字签名。
- RSA SHA-1。此签名算法(在OSI Interoperability Workshop中定义)，使用PKCS#1中定义的填充协定。

7.6.4 随机数产生算法

随机数产生算法名可以在产生一个SecureRandom实例时指明。

- SHA1PRNG 由SUN提供器提供的PRNG算法名。

该实现符合IEEE P1363标准，并且使用SHA-1作为PRNG的基础。该算法通过一个真正随机的种子值来计算SHA-1 散列值，该种子值与一个每次操作加 1的64位计数器相关联。在160位的SHA-1输出中，只使用了64位。

7.6.5 证书类型

证书类型可以在产生一个CertificateFactory实例时指明。

- X.509。X.509中定义的证书类型。

7.6.6 密钥库类型

密钥库类型可以在产生一个KeyStore实例时指明。

- JKS。由SUN提供器提供的密钥库实现名。
- PKCS12。PKCS#12中定义的个人身份信息的传递语法。

7.7 算法

在实现密码算法时，提供器应该符合已存在的标准规范。下面是这些规范中的一部分以及它们与JDK实现的关系。对每一个算法，特别给出了全部或部分的域。

- 名字。通过名字该算法为人所知。该名传递给 getInstance方法(在需要该算法时)，并且由getAlgorithm方法返回以确定一个已存在算法对象名。这些方法在服务类 Signature，MessageDigest，KeyPairGenerator和AlgorithmParameterGenerator中。
- 类型。算法类型有：Signature，MessageDigest，KeyPairGenerator，或ParameterGenerator。
- 描述。关于算法的一般注解，包括由算法、合适的许可等实现的任何标准。
- 密钥对算法(可选)。算法的密钥对算法。

China-pub.com 下载

- 密钥大小。(可选)。密钥算法或密钥产生算法的合法密钥大小。
- 大小(可选)。算法参数产生的算法所用的算法参数的大小。
- 参数缺省(可选)。密钥产生算法的缺省参数值。
- 签名格式(可选)。Signature算法的签名格式，也就是 verify和sign方法各自的输入和输出。

7.7.1 SHA-1消息摘要算法

名字：SHA
类型：MessageDigest
描述：消息摘要算法在NIST的 FIPS 180-1种定义。算法输出一个160位的摘要。

7.7.2 MD2消息摘要算法

名字：MD2
类型：MessageDigest
描述：消息摘要算法在RFC1319中定义。算法输出一个128位(16字节)的摘要。

7.7.3 MD5消息摘要算法

名字：MD5
类型：MessageDigest
描述：消息摘要算法在RFC1321中定义。算法输出一个128位(16字节)的摘要。

7.7.4 数字签名算法

名字：DSA SHA-1
类型：Signature
描述：NIST FIPS 186中描述的Signature算法使用带SHA-1消息摘要算法的DSA。
密钥对算法：DSA
签名格式：两个INTEGER值r和s的ASN.1序列：

```
SEQUENCE ::= {r INTEGER,s INTEGER}
```

7.7.5 基于RSA的签名算法

名字：RSA MD2，RSA MD5和RSA SHA-1
类型：Signature
描述：此签名算法使用MD2、MD5和带RSA加密的SH1-1消息摘要算法。
密钥对算法：RSA
签名格式：DER编码PKCS#1块。加密数据是已签署数据的摘要。

7.7.6 DSA密钥对产生算法

名字：DSA
类型：KeyPairGenerator

China-pub.com
下载

描述：DSA密钥对产生算法在NIST FIPS 186中描述

密钥大小：以位的形式表示模数 p的长度。在512和1024之间，是64 的整倍数。缺省大小是1024。

参数缺省：下面是密钥大小为 512、768和1024位时所用的缺省参数值， counter参数的用法请参阅FIPS文献。

512位密钥

```
SEED = b869c82b 35d70e1b 1ff91b28 e37a62ec dc34409b

counter = 123

p = fca682ce 8e12caba 26efccf7 110e526d b078b05e decbcd1e
    b4a208f3 ae1617ae 01f35b91 a47e6df6 3413c5e1 2ed0899b
    cd132acd 50d99151 bdc43ee7 37592e17

q = 962eddcc 369cba8e bb260ee6 b6a126d9 346e38c5

g = 678471b2 7a9cf44e e91a49c5 147db1a9 aaf244f0 5a434d64
    86931d2d 14271b9e 35030b71 fd73da17 9069b32e 2935630e
    1c206235 4d0da20a 6c416e50 be794ca4
```

768位密钥

```
SEED = 77d0f8c4 dad15eb8 c4f2f8d6 726cefd9 6d5bb399

counter = 263

p = e9e64259 9d355f37 c97ffd35 67120b8e 25c9cd43 e927b3a9
    670fbec5 d8901419 22d2c3b3 ad248009 3799869d 1e846aab
    49fab0ad 26d2ce6a 22219d47 0bce7d77 7d4a21fb e9c270b5
    7f607002 f3cef839 3694cf45 ee3688c1 1a8c56ab 127a3daf

q = 9cdbd84c 9f1ac2f3 8d0f80f4 2ab952e7 338bf511

g = 30470ad5 a005fb14 ce2d9dcd 87e38bc7 d1b1c5fa cbaecbe9
    5f190aa7 a31d23c4 dbbcbe06 17454440 1a5b2c02 0965d8c2
    bd2171d3 66844577 1f74ba08 4d2029d8 3c1c1585 47f3a9f1
    a2715be2 3d51ae4d 3e5a1f6a 7064f316 933a346d 3f529252
```

1024位密钥

```
SEED = 8d515589 4229d5e6 89ee01e6 018a237e 2cae64cd

counter = 92

p = fd7f5381 1d751229 52df4a9c 2eece4e7 f611b752 3cef4400
    c31e3f80 b6512669 455d4022 51fb593d 8d58fabf c5f5ba30
    f6cb9b55 6cd7813b 801d346f f26660b7 6b9950a5 a49f9fe8
    047b1022 c24fbba9 d7feb7c6 1bf83b57 e7c6a8a6 150f04fb
    83f6d3c5 1ec30235 54135a16 9132f675 f3ae2b61 d72aeff2
    2203199d d14801c7

q = 9760508f 15230bcc b292b982 a2eb840b f0581cf5

g = f7e1a085 d69b3dde cbbcab5c 36b857b9 7994afbb fa3aea82
    f9574c0b 3d078267 5159578e bad4594f e6710710 8180b449
    167123e8 4c281613 b7cf0932 8cc8a6e1 3c167a8b 547c8d28
```

China-pub.com 下载

```
e0a3ae1e 2bb3a675 916ea37f 0bfa2135 62f1fb62 7a01243b
cca4f1be a8519089 a883dfe1 5ae59f06 928b665e 807b5525
64014c3b fecf492a
```

7.7.7 RSA密钥对产生算法

名字：RSA
类型：KeyPairGenerator
描述：RSA密钥对产生算法在PKCS#1中描述
密钥大小：大于或等于512并且是8的整倍数的任意整数

7.7.8 DSA参数产生算法

名字：DSA
类型：ParameterGenerator
描述：DSA参数产生算法在NIST FIPS 186中描述
大小：以位的形式表示模数p的长度。在512和1024之间，是64 的整倍数。缺省大小是1024。

China-pub.com
下载

China-pub.com
下载

第6章　对象的安全

如果我们不能保障我们所有的权力，那么就让我们保障那些我们能够做到的。

——托马斯·杰弗逊

当使用Java语言和平台进一步开发应用程序,尤其是考虑到安全因素时，你自觉或不自觉地就会依赖于所指定对象的方向，诸如数据压缩，对象名空间分割和类的安全。这种依赖性很明显地体现在运行时间的内部状态保护上。而这种状态经常被维持并表示为 Java对象。例如，当应用Java RMI()程序包以跨越多重JVMS的方式建立分布式Java应用程序时，你有时会发现，当对象状态从一台计算机传递到另一台计算机时，为了可靠和保密，确保保护对象的可靠性和保密性是必须的而且很方便。当所指对象处于运行系统中 (例如，在内存中),存储在系统内部(比如，存储在程序包IP中)，或存储在系统外部(例如，存储在磁盘上)，这些安全性需求是有必要的。

这就意味着在系统开发过程中，必须正确解决全局的对象安全问题，此外还有代码签署，策略驱动，细微访问控制机制，本章提供了许多技术，以便在 Java 语言中获得安全编程。另外，本章还描述了三个新型界面，以签署、约束和监护 Java对象。下面我们开始讨论一些一般性的实例。

6.1　安全异常

在低级代码中出现异常较为常见，因此要么通过把它传递到到高级异常中暂时隐蔽起来待以后再次出现，要么作一些起基本的处理。

例如，假设你定义了一个类 MyPasswordchecker，以便用户登录时核对密码。要是因为用户名不存在，而密码核对失败，让用户知道名字写错了不是一个好办法，因为这样做会使攻击者去猜测用户名。相反，应该给出一个一般性的错误信息，如“登录失败”。

```
public class MyPasswordChecker {
   public void check(String name, String password)
            throws LoginFailureException {
      try {
      // Call the real password checking routine.
      ...
      } catch (NoSuchUserException e) {
         throw new LoginFailureException():
      }
   }
}
```

然而，当编写代码以掩盖它或处理安全异常 (如AccessException和SecurityException)时，必须非常小心，因为这一操作可能会潜在地掩盖严重违反安全规则的情况。有时开发者会因安全异常感到烦恼，于是用自己定义的安全策略来代替 AccessControll或SecurityManager，这种“替代代码工作”的方式非常危险，尤其是当代码被发展成系统代码，而且被完全信任时。

China-pub.com
下载

通常情况下，开发设计应尽可能避免出现安全异常而且也不要有意掩盖它。

6.2 域和方法

Java语言提供四种访问模式，即： public(公用的),protected(保护的),package private(包私有的)和private(私有的)，应该正确地应用它们。以下给出了一个不正确应用的示例：没有经验的编程者，在定义一个时间区域类时，错误的将域和变量声明为可公开访问的。

```
public TimeZone defaultZone;
```

这个设计会出现一系列问题。首先，任何人和代码 (包括不受信任的代码)都可以访问此域,而且可直接修改缺省时间区域的值，其次，由于多线程也可访问此域，故需要同步。下面的例子则是一个较好的设计：

```
private TimeZone defaultZone= null;
public synchronized void setDefault(TimeZone zone) {
   defaultZone = zone;
}
```

假设现在需要核查安全状况，以防止未经准许就修改默认时间区域的值，则可以加上两行安全代码，如下所示：

```
private TimeZone defaultZone= null;
public synchronized void setDefault(TimeZone zone) {
   AccessController.checkPermission(new
       TimeZonePermission("setDefault"));
   defaultZone = zone;
}
```

关于这个更新后的设计，重要的一点是它不会破坏向后的兼容性，也就是说运行在较早发行版本的第三方应用程序，当在新版本上运行时，仍有相同的 API。如果TimeZone如第一种设计那样直接放在公用域上，而且不改变 API，则安全核查(或同步特性)就无法执行。

综上所述，最好不要设计成可以直接访问的公用域和变量。相反，将这些域声明为私有域，并提供公用的访问方法以间接访问这些域。并且，当访问很敏感 ,需要安全核查时，对每个公用的访问方法都要小心的决定。若将域声明为常量，那么它是公用的，但应定义为静态的量,且为最终值,这一点我们将在下一部分中讨论。

即使有些方法或域为保护域，只要此类不是最终的子类，都可以访问它们。因为攻击者很容易就能提供一个子类，所以安全性得不到保证。与此类似，包私有方法和域可被同一个程序包中的任何类访问。如果此程序包没有被适当的约束，攻击者就会很容易定义一个类，且声明此类属于这个私有程序包。这样他就能访问此包私有方法和域。应当经常检查所有保护与以及包私有方法和域，以确定它们是否被定义为私有的，如果没有，应当检查它们是否通过执行安全核查的访问方法来访问。

6.3 静态域

静态域(static field)是一个类域，从某种意义上说，它的值被同一类中的所有对象共享。静态域是一种资源域，可能会引起假设独立子系统间意想不到的相互作用。它们提供的保护甚至比对象域还少。这是因为在后一种情况中，为了访问域，必须引用对象。而在前一种情

China-pub.com 下载

况中，只要用类名就可以访问域。

所以，若公用静态变量不是最终变量，这对安全性很不利，不要设计带有这种变量的类。相反，应把它们声明为静态私有变量，用恰当的公用访问方法来访问它们。当这些方法需要调用安全核查时，仍需慎重判决这一操作。

若已有一个带这些危险变量的设计，则必须复查所有非最终公用静态变量，认真估计它们所造成的潜在危害。必须清除最坏的危害，即使这样做会破坏向后的兼容性，若为保持向后的兼容性而必须保留它们，那么，就只能希望没有人采用这种方式来开发它们。

静态域的另外一个危险因素是当它们被偶然使用时，可能产生安全问题.例如:系统代码的一部分可被设计为在内部共享的静态域，且定义一静态域Foo。若Foo被定义得太随意，不可信applet和应用程序可能产生一个类型或子类型的对象，它与程序员所期望的不一致(例如,当Foo被声明为java.lang.Object时)。这种替代会产生极小的安全问题，系统很难检测和改正它。

6.4 私有对象状态和对象不变性

大部分对象都有私有的内部状态，而且此状态不能随意修改。其他对象经常会询问状态信息。许多程序员运用简单的返回状态来执行询问，如下所示：

```
public MyClass {
   private boolean status = false;

   public void setStatus(boolean s) {
      status = s;
   }
   public boolean getStatus() {
      return s;
   }
}
```

目前不会出现问题。可是，若status不是一个简单的布尔变量，而是布尔数组，就会出现严重的问题，如下所示：

```
public MyClass {
   private boolean[] status = null;

   public boolean[] getStatus() {
      return s;
   }
}
```

在本例中，一旦另一对象与状态s有关，它不经MyClass的许可就可改变s的值。这是因为，与布尔变量不同，布尔数组(或任何数组)是可变的，这样的结果并不是MyClass的设计者所期望的，因为不受控地修改内部状态可能导致不正确甚至可怕的结果。

对此问题有一个误解，当例中有布尔数组存在时，执行setStatus函数最简单的方法如下：

```
public void setStatus(boolean[] s) {
   status = s;
}
```

但由于s是可变的,即使MyClass拥有许可权，加载到MyClass上的对象仍可以改变s的值。

China-pub.com
下载

许多程序员忽视了这一可能性。

因此，决不要把可变对象返回到可能有害的代码中，其次，如果可变对象来源不可靠，决不要直接存储(通过分配引用内部变量的数组)。因为任何代码都可能是有害的。最好的方法是在返回和存储对象时进行备份。

上面的讨论最重要的一点是：区分不可变对象与可变对象，应尽可能保持对象不变所带来的好处。例如：array，Vector,Hashtable是可变量。即使数组只包含不可变对象 (如String)，数组本身依然是可变的，引用数组的任何人都可以改变包含在数组中的全部对象。

但是要想区分出对象是否为不变量并不容易，这是因为不可变性依赖于所采用的域和方法，以及在如此情况下作用对象是否也为不变量。此分析可能需要作递归处理，甚至跟踪到所有的末端,而且确定为不变量。

关于不变对象的最后一点是：因为口令一般都被认为是字符串，所以在 Java程序中，用String来表示密码是很常见的。 String是不可变的，然而对应用程序来说，在不需要它时，却无法删除它，如果要删除它 ,则必须使用JVM回收站。为安全性着想，最好用 char[]来表示密码，用完以后，删除数组的内容。

6.5 特权代码

回想一下，可以用 Privilegedaction 界面来标记一段代码的特权，这意味着此段代码可使用自己的特权而独立于调用者的特权。一条可信代码 (如系统代码)被赋予特权，它就可以加载库(包括内部代码)，读取任何文件，读系统属性等等。这个特权代码段是一个关键区域，在此区域中，有可能会出现错误和重大失误。

当定义有特权的代码时 ,越短越好。这不仅能减少出错的机会，而且也可使核查代码变得容易，以保证它只访问最少数量的被保护资源。

另外，注意tainted变量的应用，也就是说，变量被调用 (作为传递参数)后，就不受此特权代码的控制了。例如，用下列特权代码打开一个文件：

```
public FileInputStream getFileToRead(String filename) {
   FileInputStream fn =
      (FileInputStream) AccessController.doPrivileged(new
               PrivilegedAction() {
         public Object run() {
            return new FileInputStream(filename);
         }
      }
   );
}
```

当显示applet时，既使原来调用的 applet类不能访问此文件，此代码也可打开一个文件。但是，这个例子有两个缺陷：一个是方法是公用的，任何人都可以调用它，另一个是，此文件名无正常核查，代码将盲目地打开任何所需的文件。每一个缺点都可能是一个难题。合起来，它们将可能产生最坏的情况，也就是说，任何人都能调用这一方法来打开所需的文件 (假设此特权代码和任何系统代码一样，有一个特定的许可权)。

此问题并不仅仅停留在公用方法上。即使把 getFileToRead改为非公用的，其他公用方法也会出现而调用get File To Read。在这种情况下，一旦tainted变量被特权代码使用,安全性便

China-pub.com 下载

得不到保障。

最传统的方法就是把这些方法设计成为私有的,这样它们就不会被类外的用户所使用。

6.6 连续性

连续性使一个类或对象的非瞬态得以连续，例如 :为了把类和对象发送到另一台机器上，然后在本地机中断连续。RMI同其他程序包一样都运用外部连续。writeObject和readObject这两种缺省的实现方法，分别用于连续性和解除连续性。

一个有安全意识的程序员应该记住：一个连续类的 readObject方法实际上是一个公用的构造器,为每一个连续对象的非瞬态域输入值而且创建一个新的对象。如果这一输入到 readObject的值是由对手提供的，他的目标是扰乱对象的建立，那么通过一个正确的已构造对象的连续性所生成的输入内容是不可靠的。因此 ,若readObject盲目接受了输入，会出现不同的安全问题，而不管readObject是隐含的(也就是说，由JDK缺省提供)还是明确的(由正在讨论的连续性对象提供)，实际上，readObject的缺省实现不能进行有效的核查。

在好的编程中，若一个类有包含不变量的私有或包私有域。应提供明确的 readObject方法，以核查这些不变量是否合适，举例如下：

```
private void readObject(ObjectInputStream s) throws
        IOException, ClassNotFoundException {
   s.defaultReadObject();
   if (<invariants are not satisfied>)
      throw new java.io.StreamCorruptedException();
}
```

更进一步，若类中任何对象的参考域是私有或包私有域，而且这些对象在类 (或程序包)外是不可见的，则与这些域有关的对象作为非连续性过程的一部分，必须加以保护性地复制。也就是说，从数据流中中断连续的子对象 ,应该被新创建的对象视为不可靠的输入，而且初始化为相同的值，并由readObject方法来代替。

例如：假设一个对象有一个私有byte数组域b，那么它必须保留这个私有域，因此b应该是从readObject处复制的结果，如下所示：

```
private void readObject(ObjectInputStream s) throws
        IOException, ClassNotFoundException {
   s.defaultReadObject();
   b = (byte[])b.clone();
   if (<invariants are not satisfied>)
      throw new java.io.StreamCorruptedException();
}
```

注意:以调用clone的方式，有保护地复制子对象并不是一直正确的。若此 clone方法不能产生一独立性的副本，如此子对象的类不是最终值时，应该用一替换方法来生成副本。

用一个明确的readObject方法来保证非连续对象的完整性是一个传统的替代方法，也可以用readResolve方法，在此方法中调用一公用的构造器。这样可完全保证用一个公用的构造器来生成非连续的对象。

当实现一个Serializable界面时，还有几点要牢记 :第一，当域中包含有可直接处理的系统资源和与地址空间有关的信息时，使用 transient关键字。否则，若诸如文件处理等资源没有被

China-pub.com
下载

声明为临时状态，当对象被连续化时，状态的一部分会被改变。在对象中，此结果在中断连续后，不能正确访问资源而违反安全约定，从而产生安全错误。为正确起见，在不同环境中，若对象的连续性被中断,那么系统专用的引用没有意义，因此它们应声明为临时状态。

其次，如本章以前所指出的那样，应当保证通过类自定义非连续方法的方式，使非连续对象不会与一些常量发生冲突。因为中断对象连续性是一种产生对象的过程，如果不可信代码对对象的创建有一定限制，必须确保当此代码中断对象连续性时，有着相同的限制。举例说明：当一个应用程序产生一方框时，出于安全需要，方框中会有一警告标记“这是一个应用窗口”。假如此方框被连续化，然后由调用者中断连续，则必须确保方框还有相同的警告。

最后，当一个连续化的对象处于 JVM 外时，如被发送到另一台机器上，此状态有被破坏的潜在危险。尽管 Java安全系统直接阻止这种侵害，但如果发生了侵害也可以采取措施来检测它。一种方法是加密连续化所产生的字节流。另一种方法是运用 SignedObject和Sealed Object，本章后面将会讲到。但是这些方法并不能轻易得到，需要学习一些加密技术，这是一个复杂的工作。

6.7 内部类

定义内部类的方法也有某些安全含义。假设类A有一私有域，此域只能在类的内部被访问。再假设使用内部类重写了类 A，并使内部类B访问私有域。在编译过程中，编译器自动在 A的定义中插入一访问私有域的方法，以使 B能调用此方法。这种设计的副作用是任何与 A和B在同一程序包内的类可调用此访问方法，从而访问此私有域，但此域在应用内部类之前，是禁止访问的。把一私有域转变为包私有域不会自发地导致安全问题。但必须密切关注这种变换后的结果。

应用内部类还有一种副作用。假设类 B是类A的一个受保护成员。编译后， B的类文件定义类B是一个公用类，但类文件属性却记录为保护模式。类似地，若 B是A的一个私有成员，B的类文件定义它的程序包范围和属性为保护访问模式，这种副作用对它自身来说并不是一个问题。但是，任何JVM 实现必须进行额外的核查，以确保真正的保护属性。

6.8 本机方法

应当小心应用本机方法。本机方法，顾名思义，是在 Java安全系统之外。安全管理员和其他Java安全机制都不能控制本机代码的运行。因此本机代码出现错误或违反安全规则时，将会是致命的。应当检测本机方法所用的参数和它们的返回值。尤其是当一个本机方法有效地绕过Java安全核查时。必须仔细研究此方法的访问模式，若此模式是公用的，那么任何人都可以调用此方法。必须检测结果,以决定此方法是否可以不定义为私有的。

6.9 签署对象

回想一下前面所讨论的内容，当一个对象处于连续化状态而且在传输过程中时，需要保护此对象。事实上，存在着多种情况，需要保护对象的可靠性和状态。下面给出了两个例子：

- 作为鉴别或鉴别标记的对象，作为安全系统功能的一部分在内部通过任何 Java运行时间。此标志不能伪造，任何对其状态的修改都必须经过核查。
- 对象在机器间(JVM)传输，它的可靠性仍需证实。

China-pub.com 下载

• 对象的状态存储在Java运行时间之外。例如：存到磁盘上，以便重启 JVM时使用。

类java.security.SignedObject定义了签署对象的界面。一系列网状 signedObjects可用来创建表示一系列授权的逻辑签名序列。

SignedObject包含了被签署的对象和它的签名。被签署的对象必须具有连续性。简略地说,就是Serializable对象是实现Serialzable界面的对象。从这个意义上说，提供 readObject()和WriteObject( )方法的界面可理解为以“在线(on-the-wire)”格式改变对象在内存中的表现形式,这种格式可由Java平台提供的输入输出流来传输。

若签名不为空，则它包含被签署对象的一系列有效数字签名，见图 6-1。

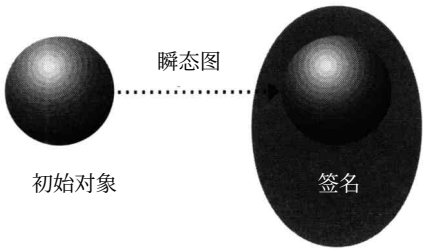


图6-1 签署对象

基本的签名算法使用 DSA 和SHA-1,还有其他符合 NIST标准的DSA。此算法用与签名同样的规则来说明，诸如SHA-1和DSA。Sun的JDK使用DSA和SHA-1内置的实现来实现签名。

被签署的对象是一个初始对象的“深入复制”(以连续化的形式)。一旦进行复制，进一步对初始对象进行控制将没有副作用。事实上， SingnedObject是不变量。

SingnedObject类的签名如下。为简略起见，异常处理没有被列出。

```
public SignedObject(Serializable object, PrivateKey signingKey,
       Signature signingEngine)
public Object getObject();
public byte[] getSignature();
public String getAlgorithm();
public boolean verify(PublicKey verificationKey,
      Signature verificationEngine);
```

此类将在以后被分为子类，以允许同一被签署的对象上有多重签名。在这种情况下，基类中所存在的方法调用在语义上是完全兼容的。特别是当有一个签名时，得到唯一的返回值；若有多个签名时，得到任意值。

基本的签名算法使用一个Signature参数调用sign和verify方法。一个典型的应用例子如下所示：

```
Signature signingEngine =
      Signature.getInstance(algorithm, provider);
SignedObject so =
      new SignedObject(myobject, privatekey, signingEngine);
```

验证和对象查找的一个典型应用如下：

```
Signature verificationEngine =
     Signature.getInstance(algorithm, provider);
if (so.verify(publickey, verificationEngine))
   try {
```

China-pub.com
下载


```
        Object myobj = so.getObject();
    } catch (ClassNotFoundException e) {};
```

很明显，为了进行成功的验证，特定的公钥必须与用来产生签名的私钥相对应。SignedObject的安全性依赖于基本的数字签名算法及其密钥管理系统没有被泄密。签名和证实引擎不必初始化，它会在验证方法内部重新初始化。

getObject( )因返回一个Object类型对象而丢失了类型信息，因此被签名的类将在合作体中使用，以便能做出正确的安排。例如前面代码可更改如下：

```
String myobject = new String("Greetings.");
...
if (so.verify(publickey, verificationEngine))
   try {
      String myobj = (String) so.getObject();
   } catch (ClassNotFoundException e) {};
```

事实上，把SignedObject设为子类，以便正确安排在子类中执行，这样就能更好地保存静态类型信息。

更为重要的是，由于适用性等原因， verify方法允许定制执行签名算法的签名引擎，此算法没有作为加密的一部分而被正式安装。然而，更重要的一点是 :程序员在编写验证代码时，必须注意所用的 Signature引擎，及验证签名时所采用的 verify实现。换句话说，一个有害的Signature在试图绕过安全核查时，可能会在验证时返回真值。出于类似的原因，在SignedObject类中的verify应是最终的。

注意：签署对象不同于签署包括类文件的 JAR文件，此JAR文件最早出现在 JDK1.1x、navigator 3.x及IE3.x中。签名代码使确认静态代码变得容易，而对象在表示一复杂处理的应用时，签名对象的处理将与活跃的状态信息发生竞争。

6.10 密封对象

前一节所讨论的SignedObject类提供了对象的可靠性和完整性。另一方面， SealedObject类保护对象的秘密。㊀这两种类可同时使用，以确保完整性和保密性。㊁事实上，从设计技术前景来看，把这两种类设计到一起会是一种更好的选择。现实情况中， SealedObject类并不仅在java.security程序包中。相反，在javax.crypto程序包中(此程序包是Java密码学扩展(JCE)的一部分)也有此类。这种设计选择完全受现在美国出口加密软件管理机制的影响。

给定任何Serializable对象，可以创建一个SealObject类，已连续化格式嵌入到原始对象中，然后，应用加密算法(如DES)以保护它的秘密。加密内容可在以后使用带有正确解密密钥的相应算法来解密。

解密后，通过中断连续，在对象中获得原始内容。在安全的密码系统中，对没有正确的解密密钥的人来说，加密的内容是不可能被访问的。

SealedObject类的签名如下，异常处理略去：

㊀ 对于那些有兴趣致力于安全对象的历史研究的人来说，早期使用Modula-3和Oblique[178]的安全网络对象与SignedObject和SealedObject有关，其中存在着安全远程对象调用的高级抽象概念。然后，这一抽象概念可通过在两个终点间建立通信频道，并使用该频道来发送对象和数据来实现。换句话说，不存在明显的直接签署和密封对象的概念。

㊁ 安全系统的设计经验表明，采用盲签名来加密数据有时很危险。因此应当首先创建并签署 SignedObject，然后使用SignedObject创建SealedObject。

China-pub.com 下载

```
public SealedObject(Serializable object, Cipher c);
public final Object getObject(Cipher, c);
public final Object getObject(Key, k)
public final Object getObject(Key, k, String provider);
```

此类的典型应用用以下代码来说明。首先，创建一个 DES密钥，初始化DES密码。

```
KeyGenerator keyGen = KeyGenerator.getInstance("DES");
SecretKey desKey = keyGen.generateKey();
Cipher cipher = Cipher.getInstance("DES");
cipher.init(Cipher.ENCRYPT_MODE, desKey);
```

其次，创建并加密一个 SealedObject对象，注意：在执行 SealedObject之前,Cipher对象必须用正确的算法、密钥填充方案等完全初始化。

```
String s = new String("Greetings");
SealedObject so = new SealedObject(s, cipher);
```

最后，解密并恢复初始对象。

```
cipher.init(Cipher.DECRYPT_MODE, desKey);
try {
   String s = (String) so.getObject(cipher);
} catch (ClassNotFoundException e) {};
```

SignedObject，SealedObject可被初始化为子类,以提供更好的静态类型信息。

6.11 监护对象

除了SignedObject可提供对象的可靠性外， JDK1.2介绍了一个界面， Guard和一个类GuardedObject，它们可以用于对象级的访问控制。 java.security.GuardedObject用于保护对另一个对象的访问。一旦某个对象被一个 GuardedObject压缩，对此对象的访问就由 getObject方法来控制。此方法通过调用 java.security.Guard对象中的checkGuard方法来控制对对象的访问。若不允许访问，将出现 SecurityException。图6-2作了说明，图中实线代表方法调用，虚线代表对象引用。

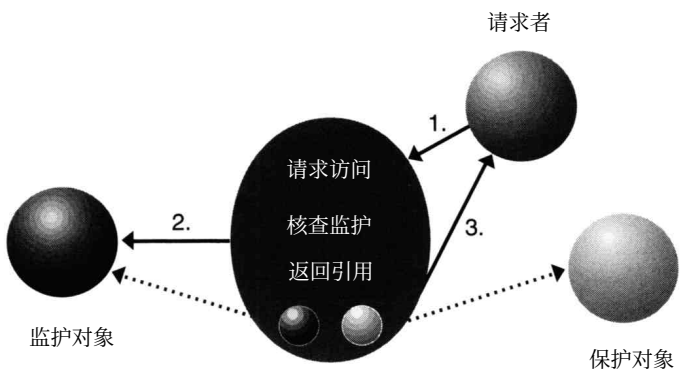


图6-2 Guard和GuardObject

当一个请求者询问一个被 GuardedObject监护的对象时，首先咨询 Guard，若得到Guard的

China-pub.com
下载

准许，则返回给请求者所需对象的引用。

使用GuardedObject类的一个重要目的是因为资源的提供者和使用者不再同一级执行状态中(如线程中)。在这种情况下，提供者安全语境内部的安全核查由于出现在使用者安全语境内部，故不准确。

例如：当一文件服务器线程响应一个请求者的请求而打开一个读操作文件，且此请求来源于不同的环境时，决定是否提供此文件应考虑请求者的信息，诸如 AccessControlContext。但是，因某些原因，使用者往往不能给提供者提供这些信息。

- 使用者的程序事先并不知道应提供什么信息 (在动态环境中尤其如此)，对每次请求都弹出对话框是很不方便的。
- 使用者把执行环境的信息对安全性的影响看得过于小心，以至于不能传递给其他参考者。
- 信息和数据太多，不能传递。
- 提供者不能解释使用者关于执行环境的信息。

为使这些情况下的访问控制对程序来说更加一致和简单，设计了 GuardedObject，使资源提供者创建一个代表资源的对象， GuardedObject包含一个内置资源对象 ,然后把GuardedObject提供给使用者。在创建 GuardedObject过程中，提供者还定义了一个 Guard类，以使当Guard内部检测(如安全检测)合格时，任何人(包括使用者)都能获得资源对象。 Guard是一个界面，任何对象都可选择成为一个 Guard。

使用GuardedObject有以下好处：

- 可正确地把保护对象嵌入到保护机制中,以便在保护机制允许的语境中监护对对象的访问。
- 可在实际的请求访问过程中，延迟访问控制，以简化服务程序。
- 可用对象存储替换常用访问控制目录 ,而且可以简便存储一系列GuardedObject。
- 类的设计者并不需要确定类的保护语义，只要任何类中对象被保护在 GuardedObject内，且联合的Guard对象实现正确的安全核查。
- 可用相同的编程模式来压缩一个对象的保护机制，使机制随对象的调用方法而变，所有这一切都在一个Guard内。

注意：因为内置基类java.security.Permission执行Guard界面，此类所有的许可权，包括定义在JDK内的许可权，可被Guard对象即时使用。

Guard界面只包含一个方法：

```
void checkGuard(Object object);
```

下面是GuardedObject 类的签名:

```
public GuardedObject(Object object,Guard guard);
public Object getObject( );
```

GuardedObject应用实例

下面的例子是使用 GuardedObject在一个恰当的Guard对象内压缩对象的保护语义。注意，这仅仅是一个实例。我们并无意用 GuardedObject去大量改变JDK中的类。

类java.io.FileInputStream用作下例，所给文件名产生一数据流，如下所示：

```
FileInputStream fis = new FileInputStream("/a/b/c");
```

此构造器的实现必须做安全核查，必须知道哪种核查是合适的，而且对于所有的构造器必须相同的(或相似的)核查。

此类可重新定义如下。首先用如下所示的新函数，使类 java.security.Permission产生一个

China-pub.com 下载

Guard对象。

```
public abstract Permission implements Guard{
   public void checkGuard( ) {
      AccessController.checkPermission(this);
   }
}
```

当数据流第一次被请求访问时，这种实现可保证在使用者内部产生一正确的访问控制核查。

现在提供方的代码如下：

```
FileInputStream fis = new FileInputStream("/a/b/c");
FilePermission p= new FilePermission("/a/b/c", "read");
GuardedObject g  = new GuardedObject(fis, p);
```

对象被传送到使用者之后，只有当使用者被准许获得文件 "a/b/c"的读访问时,以下代码才会恢复FileInputStream。

```
FileInputStream fis = (FileInputStream) g.getObject( );
```

注意:FileInputStream本身的实现并不需要完全提醒 (只要它被一个GuardedObject保护),一旦FileInputStream返回到使用者中，这种设计就不能执行进一步核查。这跟目前所用的FileInputStream是相同的。

GuardedObject的另一种潜在应用存在于Java IDL(界面定义语言)或类似的产品中迟延对象请求的实现中。此CORBA类型的API更明显的实现是把一独立线程加到 ORB实现中，以产生请求。ORB实现创建了新线程，因此任何有关产生此请求的代码丢失，这使得安全核查非常困难。使用GuardedObject，新线程可以简单地返回一个合适的监护对象。这样就使得当访问者试图得到该对象时，必须进行安全核查。

Guard和GuardedObject可被扩展实现任意的监护性语义。例如， Guard对象能在类文件中核查签名信息，因此产生一个与Java 电子商务框架[20]中的Gate模式和Permit类相似的设计。事实上，GuardedObject中的监护概念与我们熟知的编程语言的监护概念也是相似的。例如，作为一个模式[16]，监护概念在其他地方也被应用，虽然大多是专业领域。它与java.security.Permission一同使用是Java的一个神奇特点，该特点使得在Java平台上利用Guard作为访问控制非常有效。

另举一例，假设我们完全像下面所说一样重写 FileInputStream类。那么每一个没有将Guard对象作为参数的构造器会自动产生一个相应的 Guard。对于每一个访问方式 (如read(bytes))，统一的安全核查g.checkGuard()将首先被调用。

如同SignedObject一样，子类GuardedObject能更好地存储静态类型的信息，在该信息中，基类被应用在发送方和接收方之间，以保证接收方知道可以得到何种类型的对象。

China-pub.com
下载

China-pub.com
下载

第5章　结构定制安全

政府机关不会给人们带来幸福，但能给人们提供创造幸福的机会。

——Winlliam Ellery Channing

第4章讨论了配置Java安全结构的几种可能性。本章进一步研究一些具体的定制例子。

5.1　创建新的许可权类型

回想一下，JDK 1.2介绍了由抽象类确立的一种新的类型化、参数化的访问许可权java.security.Peimission。其他的许可权或者是Permission类的子类。或者是Permission子类的子类。所以一般情况下应该归入它们自己的包中。例如，表示文件系统访问的许可权位于Java的I/O包中，像java.io.FilePermission是java.io。JKD 1.2中其他新的许可权类包括：

- java.net.SocketPermission用于对网络资源的访问。
- java.lang.RuntimePermission用于对运行系统资源的访问，如属性等。
- java.awt.AWTPermission用于对窗口资源的访问。

换句话说，对于大多数的控制资源，包括对Java属性和包的访问方法和参数时由新的许可权类表示的。

应用程序可以自由地添加新种类的许可权，但必须注意的是：除非是正式发行，不管是通过给像java.lang.RuntimePermission这样的类中添加新功能还是引入另外的关键字，没有人能扩展内置于JDK中的许可权，如果要创建一个新的许可权，建议采用如下例所示的步骤。

假设ABC公司的一位软件开发人员想要定制一个许可权来“看电视”。首要问题是：他是直接用现有的Permission对象，比如任何可能的RuntimePermission对象，还是需要一个定制的对象？假设想要创建一个新的Permission类，命名为com.abc.Permission，它扩展了抽象类java.security.Permission(或它的任一个子类)。还需设计另外的新类com.abc.TVPermission，它扩展了com.abc.Permission。⊖

```
public class com.abc.Permission extends java.security.Permission
public class com.abc.TVPermission extends com.abc.Permission
```

必须确保在其他的方法中，implies()方法得到正确的实现。如果包含有更复杂的TV Permissions，例如channel-1:13或channel-*，那么也许需要实现一个TV PermissionCollection对象，它知道如何处理这些语义学。接着，需要把这些许可权类包含进应用程序包中，以便当应用程序需要它们的时候，类加载器能够找到它们。

然后，当核查一个许可权是否应该得到准许时，需要应用程序的资源管理代码来调用AccessController的checkPermission方法，使用com.abc.TVPermission 对象作为参数。

```
public void switchChannel(int channel)  {
   com.abc.TVPermission tvperm = new
            com.abc.TVPermission(channel,"watch");
   AccessController.checkPermission(tvperm);
...
}
```

⊖ com.abc.TVPermission类可以直接扩展抽象类java.security.Permission，而不必以com,abc.Permission类作为过渡。

China-pub.com
下载

最后，为了给应用程序和applet赋予许可权，需要输入合适的输入项进入安全策略。如何去设定安全策略已经在前面的章节中讨论过。一般情况下，应把此许可权的字符串表达式放进安全策略文件中，使得每一个被准许的域能自动地去设定它。下例为一个指定观看 5频道的许可权的策略文件输入项，它给任何代码赋予了观看 5频道的特权。

```
grant {
   permission com.abc.TVPermission "5" watch";
}
```

当添加一个新的许可权时，应当创建一个新的许可权类，并且不要在安全管理器中添加新方法。在JDK1.2以前，为了能够核查访问一种新类型，必须给安全管理器中添加一种新方法。在JKD1.2中，新引入的checkPermission()方法适用于所有的许可权类型。

要运用内置的访问控制算法，代码就必须通过直接调用 AccessController类的checkPermission()方法来激活一个许可权核查。不必检查是否有类加载器或安全管理器。另一方面，如果访问控制算法应该留在已安装的 SecurityManager类中，那么， SecurityManage . checkPermission()方法就应该被激活。尽管 SecurityManager.checkPermission()的缺省实现方法是反过去调用 AccessController.checkPermission()，但SecurityManager类能够定制。本章后面将会有介绍。

5.2 复合许可权

一个应用程序可能会需要很多许可权才能做它的工作，一个接一个地说明所有的许可权实在是太烦琐了。例如，一个电脑游戏可能需要以下几种许可权：套接字许可权用来和游戏服务器相连接，文件许可权用来访问一个本地存储的高分文件；属性许可权用来查找游戏者的专业级别。

在这种情况下，一个表示所有上述许可权的“速记”将会特别有用。解决这个问题的一个途径是创建一个复合许可权，如 MyGamePermission，它包含所有需要的许可权。这样以来，游戏软件就仅仅只需要一个 MyGamePermission许可权。

一个复合许可权能以几种方法中的任一种得到实现。例如： PermissionCollection类就包含了一系列不同类型的许可权，或者也可定义一个新类 CompositePermission。实际上，AllPermission就是一个隐含所有许可权的一个复合许可权。⊖

不过，这种做法有一定复杂度。当为了匹配一个所需的许可权而查找整个许可权集时，AccessControler的缺省实现可以通过只比较许可权集中与所需许可权类型相同的那些许可权优化查找过程。例如，当有一个文件访问时， AccessController查找所有给定的FilePermission并核查是否存在 AllPermission。当引入一个新类型的许可权时，缺省 AccessController实现并不能预先知道。为了解决这个问题， JDK1.2中作了修改，它通常比较每一个给定许可权和所需许可权。但是，这种方法会影响性能。比较好的解决方案就是引入新的许可权类型即CompositePermission，并要求所有的复合许可权都是这种类型。缺省 AccessController实现就可以只核查给定的 CompositePermission。这样，只有存在大量的 CompositePermission时才会导致性能下降。

⊖ NoPermission(或ZeroPermission)也可以实现，但是我们并不建议使用它。

China-pub.com 下载

5.3 定制安全策略

安全策略首先经过Policy类处理，接着由SecurityManager执行，因此定制任一个类都会定制Policy的实现。举第一个例子：假设你只想在办公时间9AM到5PM允许访问文件，即，在办公时间内安全策略能确定某人能访问某个文件，办公时间之外，不管安全策略是什么，任何人都不能访问任何文件。为了达到这个目的，可以实现一个TimeOfDaySecurityManager类，如下所示：

```
public class TimeOfDaySecurityManager extends SecurityManager {
   public void checkPermisssion(Permission perm){
      if (perm instanceof FilePermission){
         Date d = new Date();
         int i = d.getHours();
         if((i> = 9) & & (i < 17))
           super.checkPermission(perm);
         else
            throw new SecurityException("Out of office hour");
   }else super.checkPermission(perm);
   }
}
```

类TimeOfDaySecurityManager查看将要核查的许可权是否是 FilePermission。如果是， TimeOf DaySecurity计算当前时间。如果当前时间在办公时间内，它就激活了内置 SecurityManager来核查安全策略。否则，它就会提出一个安全异常。一个希望执行给定办公时间限制的应用程序应该安装这个TimeOfDaySecurityManage来替换内置的SecurityManager。

为了便于以后审计，下一个例子记录了被准许的或被拒绝的资源访问。假设你设计了一个简单的AuditingSecurityManager类，如下所示：

```
public class AuditSecurityManager extends SecurityManager {
   public void checkPermission(Permission perm) {
       Audit.enterRecord(perm);
       super.checkPermission(perm);
   }
}
```

假定还有一个在安全地方存储一个统计记录的类 Audit，它有一个变量用来在 checkPermission 之后进入审计记录记下访问控制结果。为了实现这一点，必须首先捕获可能出现的 SecurityException，并且在后面把它再提出来。

```
public class AuditSecurityManager extends SecurityManager{
   public void checkPermission(Permission perm){
      try{
         super.checkPermission(perm);
         Audit.enterRecord(perm, true);
         }catch(SecurityException e) {
            Audit.enterRecord(perm, false);
            throw e;
         }
   }
}
```

China-pub.com
下载

为了实现复杂的安全策略，必须付出更多的努力。例如：如果想实现一个多级安全策略，就首先必须为每一个对象创建比较敏感的标签。[⊖] JVM还会跟踪对象之间的交互，并且得自动地改变对象标签(当处于高水印模式时)。接着，类SecurityManager的checkPermission方法就会把它的决定建立在执行线程所包括的对象的标签上。再举一个例子，为了实现一个中国墙模型(责任划分)，JVM不但需要监视对象之间的交互，而且还必须把它记载下来。在这里需要做很多的研究和试验。

5.4 移植基于JDK1.1的安全管理器

在JDK1.1中，类java.lang.SecurityManager是抽象类，其核查方法的缺省实现总是仅仅提出异常。结果就造成了那些想要给自己的应用程序 (如浏览器)安装一个安全管理器的开发人员不得不自己定制安全管理器，并且要提供合适的提出异常的方法的具体实现。

在JDK1.2中，SecurityManager是具体类，所以它像应用程序缺省的那样去应用安全管理器，而且，它的设计也得到了大幅度的提高。因此，已经为自己的应用程序写过安全管理器的开发人员应当考虑把它移植到基于 JDK1.2的安全管理器类。在大多数情况下，他们应该只使用内置的缺省实现。

在移植的过程中需要做很多技术细节。本节致力于解决这些问题，首先回顾一下 JDK1.1的安全管理器类，接着检查JDK1.2中的改进。

5.4.1 JDK1.1安全管理器类

在JDK1.1中，本地应用程序和具有正确数字签名的 applet的确对那些像文件系统等的重要资源享有充分的访问权利，但没有签名的 applet并没有这样的特权，且只能访问有限的资源。一个安全管理器有责任决定哪个资源访问是允许的。无论是在 JDK1.1中，还是在JDK1.2中，SecurityManager类拥有很多以 check开头的方法，有时也称为核查方法，如 checkRead和checkConnect。Java库中各种方法在进行所有具有安全敏感的操作前都要调用一个核查方法。如果操作被允许，一个安全管理器常规返回；如果操作被禁止，它将提出一个安全异常。因此安全管理器可以通过提出异常的方式阻止操作的完成。只有 checkTopLevelWindow是个例外，它返回的是一个boolean值。

SecurityManager中包含的其他方法与类加载器的存在和范围相关：

```
currentClassLoader
currentLoaderClass
inClassLoader
classLoaderDepth
```

理解这四个方法很重要，如JDK1.1的安全管理器一般基于两个变量来决定访问控制：

(1) 一个带有类加载器的类(即JDK1.1中的一个applet)是否在堆栈上。

(2) 类加载器的深度，也就是用类加载器定义的一个类的一个方法在堆栈中最近出现的位置有多远。

例如，一个典型的JDK1.1安全管理器就具有如下所示的checkExit方法。

```
public void checkExit(int status) {
   if(inClassLoader()) {
```

⊖ 这通过在基类(对象类)中添加一个安全级别属性就可以简单地实现，但这是一个很有意义的变化。

China-pub.com 下载

```
        throw new SecurityException();
    }
}
```

这样的方法不允许在用类加载器定义的任何一个类还处于堆栈中时调用 Runtime.exit。

这里还有另外的一个例子：

```
public void checkCreateClassLoader() {
    if(classLoaderDepth() == 2) {
        throw new SecurityException();
    }
}
```

这个方法说明类加载器的深度不能是2。即，那些调用已经调用了checkCreateClassLoader的方法肯定不能处于用类加载器定义的类中。例如： java.lang.ClassLoader的构造器调用了checkCreateClassLoader,这就意味着调用java.lang.ClassLoader构造器的方法肯定不能有类加载器。因此， applet不能直接创建类加载器。

虽然这两个例子用于阻止applet执行操作，但它们之间有很大区别。在第一个例子中，无论applet处于堆栈中任何位置， checkExit都会提出异常。因此，甚至内置的JDK代码在被一个applet调用时也不能退出JVM。在第二个例子中， JDK代码即使被一个applet调用时也被允许创建一个类加载器(例如当堆栈的深度不是2时)。这时由于具有类加载器的类的深度的原因，而不是因为具有类加载器这个事实。

前面的章节讨论了这些类加载器深度的方法会有怎样的错误倾向。对于 JDK1.2的SecurityManager来说，这样的方法是不允许的，并且不能再被任何核查方法调用。它们应该被任何新写的安全管理器所应用，也应该从现有的安全管理器中删除掉。但是，为了向后兼容，界面应得到保留。

5.4.2 使JDK1.1安全管理器适应JDK1.2

有时候，用旧的JDK1.1安全管理器建立的应用程序必须在JDK1.2平台上运行。为了适应这一点， SecurityManager类中一些方法的实现应该在考虑到JDK1.2中引入的新特征的前提下得到改进。

前面章节中所提及的四个方法可用以下三个方法加以改进：

(1) 跳过系统类加载器。

(2) 到达堆栈中标有特权的一个方法之后停止核查。

(3) 在一定的情况下，完全放心地去处理当前堆栈。

1. 跳过系统类加载器

一个系统类加载器被定义为一个等价于这个系统类加载器的类加载器 (就像被ClassLoader.getSystemClassLoader所返回)或者它的一个祖先类。既然由系统类加载器所加载的类包含有应用程序类(由CLASSPATH所卸载)、扩展类和内置JDK类，这样的改进使得这些方法可以忽略这样的代码。

因为，当运行一个装有定制的安全管理器的应用程序，且这个安全管理器是在JDK1.2中由CLASSPATH卸载时，就会有一个与之相关联的系统类加载器。重新调用该应用程序类就没有一个JDK1.1中的类加载器。如果从用户自己定制的安全管理器中调用一个像ClassLoaderDepth的

China-pub.com
下载

方法，并且该方法并没有被改进为可以忽略掉由系统类加载器加载的类时，总是返回0。这就没有意义了。同样的，如果类加载器方法没有改进为可以跳过系统类，并且一个用户定制的安全管理器可从CLASSPATH中卸载掉，那么，当安全管理器基于像“当classLoaderDepth()=2时禁止”这样的条件上作出决断时，就会出现安全漏洞问题。

2. 到达一个具有特权的方法后停止核查

当这四个方法到达堆栈中被标为具有特权 (依据JDK1.2中特权的定义)的方法后就停止校验。这种改进非常必要，这是因为，比如： JDK代码为了内部应用需要打开文件。某些JDK1.1安全管理器都具有一个与下面所示相似的 checkRead方法：

```
public void checkRead(String file){
   if(inClassLoader()){
      throw new SecurityException(..);
   }
}
```

如果没有这样的改进，那么当运行 JDK1.2时，如果JDK自身要去读一个文件，堆栈中有一个具有非系统类加载器的类，则激活这样一个核查会出现安全异常。在这种新的安全模式下，想要执行一个其调用者可能并未得到允许的操作的所有 JDK代码在其周围具有一个doPrivileged块。修改inClassLoader来向上检查堆栈，且包括含有特权代码的结构会导致inClassLoader方法返回false。这样反过来又允许当堆栈顶部的代码是 JDK代码，并且是由系统类加载器或它的一个祖先类加载的时，读操作可以发生。

JDK1.2尽量保持JDK1.1安全管理器中堆栈的深度。例如， java.Security.SecureClassLoader的构造器可以直接调用 SecurityManager.checkCreateClassLoader，即使它的超级类(ClassLoader)的构造器也这样做。如果这个核查没有被放在 SecureClassLoader的构造器中，那么，由于类加载器的深度总是大于 2，因此JDK1.1安全管理器会允许无法信任的代码去扩展SecureClassLoader和创建类加载器。

如果你持有一个JDK1.1安全管理器，建议你在JDK1.2之下运行该安全管理器之前，首先要分析自己定制的安全管理器方法，此举失败会导致一个安全漏洞，或者会阻止 JDK的正常操作，这是源于JDK1.1安全管理器的不稳定特性。如果可能，应当使用 JDK1.2安全管理器缺省的实现方法，这样有助于使用户和管理人员具有一致行为。如果不可能，那么在扩展SecurityManager类并且覆盖现存方法时要十分当心。

例如，如果覆盖了checkRead(String file)方法使得它总能提出安全异常，那么 JDK本身也许就不能进行正常的操作。即，如果一些 JDK代码需要打开一个文件 (读属性文件或加载一个JAR文件)，那么，对于每一个读操作出现的安全异常会导致打开文件总是失败。

一般而言，只有当要放松而不是要加强安全意识时，才应当覆盖缺省的方法。为了加强安全，应当修改缺省的策略文件并 (或)安装一个定制的java.Security.Policy类或对象。当覆盖安全管理器方法时，应当在被覆盖的核查方法出现异常的地方放置一个对 super.check方法的调用。如下所示：

```
public class MySecurityManager extends SecurityManager{
   public void checkRead(Strting file){
      if (someCustomSecurityCheckFails()){
         super.checkRead(file);
```

China-pub.com 下载

```
    }
  }
}
```

这里，如果定制的核查方法失败，那么 super.checkRead就能被调用。checkRead的缺省实现方法激活checkPermission，checkPermission在缺省情况下又激活 AccessController，此后的过程激活了读取文件之前已经执行 AccessController.doPrivileged的系统代码，因此，JDK本身工作正常。其他所有的代码将会与目前有效的策略保持一致。当对文件的访问没有得到准许时就会出现一个访问控制异常。换句话说， JDK1.1安全管理器代码通过调用 super.check，给JDK1.2内置的SecurityManager和AccessController提供了一个机会去处理仅出现在 JDK1.2中的特殊情况。

然而，当覆盖某些核查方法时，不应该调用 super.check方法，这是因为，这些核查方法的缺省实现也许并不像覆盖方法中实现的策略那么严密。例如， checkAccess(ThreadGroupg)方法在缺省情况下只保护系统线程组。如果想保护不同线程组中的线程 (如applet线程组)，那么你不会在出现安全异常的地方调用 super.checkAccess()方法。因为那样做将会导致你定制的核查方法失败。相反地，应该在覆盖方法中调用 super.checkAccess()作为首条语句。

```
public class MySecurityManager extends SecurityManager{
   public void checkAccess(ThreadGroup g){
   //A call to super will throw an exception if someone
   //is trying to modify the system thread group.
   super.checkAccess(g);
   ...
   //Now perform checks to see if the current caller
   //can modify thread group g,based on to which applet
   //thread group the caller belongs.
   ...
}
}
```

3. 完全放心地处理当前堆栈

第三个改进与第二个改进相关联，并且给这四条方法提供了如下行为：当它们遇到的是一个被赋予 AllPermission的具有SecurityContext(或AccessControlContext)的doPrivileged的堆栈结构时，它们可以完全放心地去处理当前堆栈，或者从 JDK1.1的意义上来讲，似乎堆栈上没有类加载器。这是因为一个具有 AllPermission的doPrivileged结构意味着无论这个方法的调用者是什么，都可以去做任何它想做的事情。这个具有充分特权的情形等于是，当用java.security.AllPermission调用checkPermission时不会造成任何安全异常。

5.4.3 为JDK1.2而改进JDK1.1安全管理器

本节列出了JDK1.2中对java.lang.SecurityManager方法所做的改进，还包括了一些对你可能想要做的覆盖方法的建议。当改进 JDK1.1安全管理器时，应该尽可能地使用 JDK1.2内置的安全管理器。

如前所述，下列方法已经被禁用：

```
public boolean getInCheck();
```

China-pub.com
下载


```
protected ClassLoader currentClassLoader();
protected Class currentLoadedClass();
protected int classDepth(String name);
protected int classLoaderDepth();
protected boolean inClass(String name);
protected boolean inClassLoader();
```

下面是其他的一些方法：

- protected boolean inCheck。这个域已经被取消。JDK中对它的所有使用均已删除，应该把checkPermission同doPrivileged一起使用,而不是使用inCheck。
- public SecurityManager()。这个构造器经过改造之后，假定调用者具有 RuntimePermission("createSecurityManager")许可权，则可允许创建多个SecurityManager。
- protected native Class[] getClassContext()。此方法没有改变。对它的调用可用来模拟JDK1.2中已被改变的JDK1.1中的行为(currentClassLoader，currentLoadedClass，classLoaderDepth和inClassLoader)。
- public Object getSecurityContext()。此方法经过改进之后返回一个java . Security . Access Control Context对象，这个对象通过调用java.Security.AccessController.getcontex 生成。在JDK1.1中，它的缺省返回值是null。
- public void checkPermission(Permission perm)。此方法是JDK1.2中的新方法。它在给定的许可权下调用java.Security. AccessController.checkPermission。在内部，JDK总是调用SecurityManager.checkPermission ，而不去直接调用AccessController。这允许覆盖这个方法来提供附加的功能。如：审计和图形用户界面(GUI)对话框。
- public void checkPermission(permission perm,Object context)。此方法是JDK1.2中的新方法，如果context是AccessControllContext的一个实例，那么，在给定的context，及指定的许可权下， AccessControlContext.checkPermission方法被激活。如果context不是AccessControlContext的实例，则出现异常。
- public void checkCreateClassLoader()。此方法经过改进，在 Runtime Permission ("createClassLoader")许可权下调用checkPermission。如果此方法被覆盖，那么，应该在覆盖方法中出现异常的地方调用super.checkCreateClassLoader。同5.4.2节中的例2。
- public void checkAccess(Thread t)。此方法已被改进。如果线程参数是一个系统线程——属于无父级的线程组——那么，这个方法在RuntimePermission("modifyThread")许可权下调用checkPermission。需要严格策略的应用程序应该覆盖这个方法，且其首条语句应调用super.checkAccess或它的等价方法。而且，作为结果的方法还应该核查是否调用线程具有RuntimePermission("modifyThread")许可权，如果有，返回。这也是确保被赋予许可权的代码(如JDK本身)被允许操纵任何线程。

如下所示为一个方法覆盖的实现例子：

```
public class MySecurityManager extends SecurityManager{
   public void checkAccess(Thread t){
      //A call to super will throw an exception if someone
      //is trying to modify a system thread.
      super.checkAccess(t);
```

China-pub.com 下载

```
        ...
        if(someCustomSecurityCheckForOtherThreadsFails()){
           //If the check fails,instead of throwing an
           //exception,call checkPermission,which will
           //throw an exception if needed.
           checkPermission(new
                 RuntimePermission("modifyThread"));
           }
         ...
    }
}
```

- public void checkAccess(ThreadGroup g)。此方法已被改进。如果线程组参数是一个系统线程组(即无父级的线程)，那么，这个方法在RuntimePermission("modifyThreadGroup")许可权下调用checkPermission。需要严格策略的应用程序应该覆盖这个方法，且其首条语句应调用super.checkAccess或它的等价方法。而且，作为结果的方法还应该核查是否调用线程具有RuntimePermission("modifyThreadGroup")许可权，如果是，返回。这也是确保被赋予许可权的代码(如JDK本身)被允许操纵任何线程组。
- public void checkExit(int status)。此方法经过改进，在RuntimePermission("exitVM")许可权下调用checkPermission。如果此方法被覆盖，那么，应该在覆盖方法中出现异常的地方调用super.checkExit。
- public void checkExec(String cmd)。此方法经改进，可以在FilePermission许可权下调用checkPermission。如果cmd 是一个绝对路径，那么它像FilePermission的目标一样被传递。如果cmd不是一个绝对路径，那么专门的目标《 ALL FILES》被应用。因为很难确定由于环境变量等原因而必须在个人平台上执行命令的真实路径，所以才用到目标。如果这个方法被覆盖，那么，应该在覆盖方法中出现异常的地方调用 super.checkExec。
- public void checkLink(String lib)。此方法经过改进，在 Runtime Permission ("LoadLibrary."+lib)许可权下调用checkPermission。如果此方法被覆盖，那么，应该在覆盖方法中出现异常的地方调用 super.checkLink。
- public void checkRead(FileDescriptor fd)。此方法经过改进，在 Runtime Permission ("readFileDescriptor")许可权下调用checkPermission。如果此方法被覆盖，那么，应该在覆盖方法中出现异常的地方调用 super.checkRead。
- public void checkRead(String file)。此方法经过改进，在FilePermission(file,"read")许可权下调用checkPermission。如果此方法被覆盖，那么，应该在覆盖方法中出现异常的地方调用super.checkRead。
- public void checkRead(String file,Object context)。这个方法已被改进。如果 context是AccessControlContext的一个实例，那么，在 FilePermission(file,"read")许可权下，AccessControlContext.checkPermission方法将在给定的context处被激活。如果context不是AccessControlContext的实例，则将出现SecurityException。如果这个方法被覆盖，那么应该在覆盖方法中出现异常的地方调用 super.checkread。
- public void checkWrite(FileDescriptor fd)。此方法经过改进，在 Runtime Permission ("writeFileDescriptor")许可权下调用checkPermission。如果此方法被覆盖，那么，应该

China-pub.com
下载

在覆盖方法中出现异常的地方调用 super.checkWrite。

- public void checkWrite(String file)。此方法经过改进，在FilePermission(file,"write")许可权下调用checkPermission。如果此方法被覆盖，那么，应该在覆盖方法中出现异常的地方调用super.checkWrite。
- public void checkDelete(String file)。此方法经过改进，在FilePermission(file,"delete")许可权下调用checkPermission。如果此方法被覆盖，那么，应该在覆盖方法中出现异常的地方调用super.checkDelete。
- public void checkConnect(String host,int port)。此方法经过改进，在 Socket Prmission (host+":"+port,"connect")许可权(port不等于－1时)或SocketPermission(host,"resolve")许可权(port等于－1时)下调用checkPermission方法。这种行为同JDK1.1一致。在JDK1.1中，port等于－1意味着正在执行IP地址查找。如果这个方法被覆盖，那么应该在覆盖方法中出现异常的地方调用super.checkConnect。
- public void checkConnect(String host,int port,Object context)。此方法已被改进。如果context是AccessControlContext的一个实例并且如果 port不等于－1，那么，在 Socket Permission (host+":"+port,"connect")许可权下， AccessControlContext.checkPermission方法将在给定的context处被激活。如果 context不是AccessControlContext的实例，则将出现SecurityException。如果这个方法被覆盖，那么应该在覆盖方法中出现异常的地方调用super.checkConnect.
- public void checkListen(int port)。这个方法已被改进，在 Socket permission ("localhost:" +port,"listen")许可权(port不等于0时)或Socket Permission ("localhost:1024-","listen")许可权(port等于0时)下调用checkPermission。如果这个方法被覆盖，那么应该在覆盖方法中出现异常的地方调用super.checkListen.
- public void checkAccept(String host,int port)。此方法经过改进，在 Socket Permission (host+":"+port,"accept")许可权下调用checkPermissio。如果此方法被覆盖，那么，应该在覆盖方法中出现异常的地方调用 super.checkAccept。
- public void checkMulticast(InetAddress maddr)。此方法经过改进，在 Socket Permission (maddr.getHostAddress(),"accept,connect")许可权下调用checkPermission。如果此方法被覆盖，那么，应该在覆盖方法中出现异常的地方调用 super.checkMulticast。
- public void checkMulticast(InetAddress maddr,byte ttl)。此方法经过改进，在 Socket Permission (maddr.getHostAddress(),"accept,connect")许可权下调用checkPermission。如果此方法被覆盖，那么，应该在覆盖方法中出现异常的地方调用 super.checkMulticast。
- public void checkPropertiesAccess()。此方法经过改进，在 Property Permission ("*","read,write")许可权下调用 checkPermission。如果此方法被覆盖，那么，应该在覆盖方法中出现异常的地方调用 super.checkPropertiesAccess。
- public void checkPropertyAccess(String key)。此方法经过改进，在 Property Permission (key,"read")许可权下调用checkPermission。如果此方法被覆盖，那么，应该在覆盖方法中出现异常的地方调用 super.checkPropertyAccess。
- public boolean checkTopLevelWindow(Object window)。此方法经过改进，在 AWTPermission("showWindowWithoutWarningBanner")许可权下调用 checkPermission。

China-pub.com 下载

如果没有出现SecurityException，则返回true；否则，返回false。如果此方法被覆盖，那么，应该在覆盖方法中出现异常的地方调用 super.checkTopLevelWindow，并且应该返回super.checkTopLevelWindow的值。例如：

```
public class MySecurityManager extends SecurityManager{
   public void checkTopLevelWindow(Object window){
      if (someCustomSecurityCheckFails()){
         return super.checkTopLevelWindow(window);
      }else return true;
   }
}
```

- public void checkPrintJobAccess()。此方法经过改进，在 Runtime Permission ("queuePrintJob")许可权下调用checkPermission。如果此方法被覆盖，那么，应该在覆盖方法中出现异常的地方调用super.checkPrintJobAccess。
- public void checkSystemClipboardAccess()。此方法经过改进，在AWTPermission("accessClipboard")许可权下调用checkPermission。如果此方法被覆盖，那么，应该在覆盖方法中出现异常的地方调用super.checkSystemClipboardAccess。
- public void checkAwtEventQueueAccess()。此方法经过改进，在 Property Permission ("accessEventQueue")许可权下调用checkPermission。如果此方法被覆盖，那么，应该在覆盖方法中出现异常的地方调用super.checkAwtEventQueueAccess。
- public void checkPackageAccess(String pkg)。此方法经过改进，通过调用java.security.getProperty("package.access")首次获得了一列用逗号分隔的包，并且核查pkg是否以这些包中的任一个开始或等于其中任一个。如果是，那么在Runtimepermission("accessClassInPackage."+pkg)许可权下调用checkPermission。如果此方法被覆盖，那么，应该在覆盖方法中出现异常的地方调用 super.checkPackageAccess。
- public void checkPackageDefinition(String pkg)。此方法经过改进，通过调用java. security . getProperty("package.access")首次获得了一列用逗号分隔的包，并且核查pkg是否以这些包中的任一个开始或等于其中任一个。如果是，那么在 Runtime Permission ("define ClassIn Package."+pkg)许可权下调用checkPermission。如果此方法被覆盖，那么，应该在覆盖方法中出现异常的地方调用super.checkPackageDefinition。
- public void checkSetFactory()。此方法经过改进，在 RuntimePermission("setFactory")许可权下调用checkPermission。如果此方法被覆盖，那么，应该在覆盖方法中出现异常的地方调用super.checkSetFactory。
- public void checkMemberAccess(Class clazz,int which)。此方法已被改进。缺省的安全策略允许对公共成员进行访问，也可对与调用者具有相同类加载器的类进行访问。在所有其他情况下，在 RuntimePermission("accessDeclareMembers")许可权下调用checkPermission。如果此方法被覆盖，那么不能调用 super.checkMemberAccess。因为checkMemberAccess缺省的实现方式依赖于“被核查的代码是否位于深度为 4的堆栈上”这个事实。如下：

```
someCaller[3]
java.lang.Class.someReflectionAPI[2]
```

China-pub.com
下载


```
java.lang.Class.checkMemberAccess[1]
SecurityManager.checkMemberAccess[0]
```

为了模拟此行为，需要调用 getClassContext，并且在index等于3时检查该类的类加载器，如同缺省的checkMemberAccess方法。

```
if (which ! = Member.PUBLIC){
   Class stack[] = getClassContext();
if((stack.length<4)||
  (stack[3].getClassLoader() ! = clazz.getClassLoader())) {
  if (checkMemberAccessPermission = = null)
      checkMemberAccessPermission=
          new RuntimePermission("accessDeclaredMembers");
      checkPermission(checkMemberAccessPermission);
  }
}
```

这是JDK1.2中唯一的仍旧基于调用者深度的安全管理器方法。也允许调用者引用来自于同一类加载器的类。

- public void checkSecurityAccess(String target)。此方法经过改进，当 SecurityPermission 对象作为给定目标时调用 checkPermission。如果此方法被覆盖，那么应该在覆盖方法中出现异常的地方调用 super.checkSecurityAccess。
- public ThreadGroup getThreadGroup()。此方法没有改变。

China-pub.com
下载

China-pub.com
下载

第4章　安全结构的使用

策略必须清楚、一致和可信。

—— 艾森豪威尔

为了利用JDK1.2中的安全机制，必须将Java环境升级到最新的版本。通常,新版本的操作系统和Internet浏览器都支持JDK1.2。为了将JVM升级到1.2版,你可以下载并安装用于开发JDK的运行Java应用程序或applet的Java运行环境，或者能够升级IE和Navigator内的JVM的Java插件程序。

4.1　安装JDK1.2

所有Win32(MS-Windows或Windows NT)和Sun Solaris平台上的JDK1.2可在位于http://java.sun.com/products/jdk/1.2/.的Sun Microsystem的Java站点上得到。出于演示的目的，本章所用的指令假设你正在使用一个运行Solaris的Sun Sparc工作站。

(1) 从Web站点上下载Solaris版本JDK1.2,并将它存到一个名为jdk12-solaris2-sparc.sh的文件中。

(2) 根据站点的安装指令,按如下所示打开下载的文件：

```
%chmod a+x jdk12-solaris2-sparc.sh
%./jdk12-solaris2-sparc.sh
```

显示出一个问你是否同意许可协议的对话框，点击同意键，现在你拥有在 ./jdk1.2/bin目录下打开的所有二进制文件。

(3) 增加./jdk1.2/bin目录到搜索路径：

```
%set path=($path ./jdk1.2/bin)
```

(4) 如果想确保程序是由UNIX 内核首次定位的新安装的JDK的一部分,把该目录移至搜索目录的开始:

```
%set path=(./jdk1.2/bin $path)
```

并证实它:

```
%which java
./jdk1.2/bin/java
%
```

(5) 接下来,可以运行带有appletviewer的applet来测试:

```
%appletviewer http://java.sun.com/applets/other/TumblingDuke/index.html
```

这个命令应在一个外壳窗口的一行中键入。也可以测试在第 2章开始所描述的应用程序:

```
%java Test Hello
```

如果你正在使用appletviewer或一个使用这种新的安全结构的新版本浏览器,你可以继续使用以前的工作方式。这意味着JDK1.1x中的沙盒模型继续可用。如果你是一个“权力用户”，那么可以使用内置policytool工具(或配有浏览器的等同工具)的JDK1.2来定制安全策略，这样就可以利用新安全结构的所有优点。这种定制涉及到建立一个证书仓库 ,它可由keytool工具制作,用来产生和管理密钥库。这个工具在本章随后会讲到,它适用于Solaris和MS-Windows平台。

China-pub.com
下载

应用开发人员无需对JDK1.2的安全特性做任何特殊的工作,因为当应用程序运行在JDK1.2的上层时,那些特性会自动被激活。开发人员也许想使用内置工具来把应用程序包装成 JAR文件,并对它进行数字签名。

一名编码控制着某种资源的软件库开发人员，需要把已存在的许可权类层次扩展到建立特定应用程序的许可权。开发人员也要学会使用由 AccessController类提供的特性 ,如doPrivilege界面。

4.2 策略配置

到目前为止演示的例子中,没有为了运行applet或应用程序而指定安全策略。因此,JVM会缺省使用JDK1.2以前的沙盒安全模型,所以远程的applet是不可靠的而本地的applet是可靠的。为了利用新的安全模型,必须先配置一个安全策略,接着指定当运行Java程序时使用哪一个策略。

在JDK1.2中策略API的设计并没有要求一个安全策略如何被永久地表达成 Java运行系统。它只指明了针对对象的 API,这样一个JDK实现可以有选择地把信息存储在一个数据库、目录服务、文件系统或其他位置。

缺省的JDK实现支持安全策略规范。配置一个安全策略包括首先指明位置，然后是策略文件的内容。显然，策略文件应当被很好地保护以防不可靠的修改。

4.2.1 配置系统型和用户专用型策略

Policy对象使用的策略信息的源位置由 Policy类的实现来确定。 JDK1.2的实现包括来自发现策略配置文件的静态位置的信息。策略文件可用一个简单的文本编辑器或 policytool来组成,policytool是一个图形工具，下一节讨论策略文件的内容。

单个系统型策略文件和单个用户型策略文件都有缺省的位置。缺省的系统策略文件被定位在：

```
<java.home>/lib/security/java.policy
```

其中，系统属性<java.home>指定了JDK被安装的目录。缺省的用户型策略文件被定位在：

```
<user.home>/.java.policy
```

其中，系统属性<user.home>指定了用户的主文件目录。

缺省的Policy对象在当getPermission方法被首次调用或refresh方法随时被调用时被初始化，初始化包括从语法上分析策略配置文件，然后填充 Policy对象。当Policy对象被初始化时,系统策略先被加载，随后是用户策略。如果这两个策略都不存在，则使用一个内置的策略。这个内置的策略等同于最初的沙盒策略。

策略文件的位置被指定在安全属性文件中，位于：

```
<java.home>/lib/security/java.security
```

这些位置作为policy.url.n形式的属性值的名称被指定，其中 n是一个数。如，缺省的系统和用户策略文件被定义在下列安全属性文件中：

```
policy.url.1=${java.home}/lib/security/java.policy
policy.url.2=${user.home}/.java.policy
```

${java.home}特别指明了属性的扩展,随后会讨论。

安全属性文件可以改变。例如，可以跳过缺省的用户策略文件，也可以指定用策略文件来形成一个组合的安全策略。通过定位几个指向该文件位置的 URL来做到这点。那么，所有

China-pub.com 下载

被指明的策略文件的内容将被用于填充 Policy对象。

注意url.n中的n，n必须用1开始且必须是连续的整数。当policy对象被初始化时，第一个策略文件由policy.url.1给出，第二个由policy.url.2给出，等等，直到没有更多的策略文件。例如，如果指定了policy.url.1和policy.url.3，但没有指定policy.url.2，那么，policy.url.3将不可读。

4.2.2 配置应用程序专用型策略

安全属性文件中给出的策略文件感觉上是系统范围型的，当运行任何 applet或应用程序时，相同策略文件的集合将被使用。当激活应用程序时，可以指定附加的或不同的策略文件。这借助于-Djava.security.policy命令行参数，该参数设置了java.security.policy属性，如:

```
java-Djava.security.manager-Djava.security.policy=someURL someApp
```

在这里， SomeURL为指定了别的策略文件位置的URL。在这种情况下,这个策略文件和指定在安全属性文件中的策略都被使用。

-Djava.security.manager参数确保缺省的安全管理器被安装 ,以便带有安全策略的所有应用程序都能有效运行。如果应用程序 SomeApp本身安装了安全管理器 ,这一选择就并不需要。假设使用下面的用法(注意赋值号==)：

```
java -Djava.security.manager -Djava.security.policy==someURL someApp
```

位于someURL的唯一被指定的策略文件将被使用，其余的策略文件则被忽略。

当使用appletviewer运行applet时，可以使用如下的 -Djava.security.policy参数来指定一个策略:

```
appletviewer -Djava.security.policy=someURL someApplet
```

如果属性policy.allowSystemProperty被设定为false,那么在Djava.security.policy选项中给出的策略文件的值无论是java还是appletviewer命令,都被忽略。安全属性文件中的属性值被缺省赋值为true。

4.2.3 配置一个可选Policy类的实现

一个给出的可选的策略类可以代替缺省的策略类，只要这个可选类是抽象 Policy类的一个子类，并且实现了getPermission方法(和其他所需的方法)。

无需改变JDK代码就可以改变缺省的 Policy实现。 policy.provider属性可在安全属性文件java.security中给出,如下:

```
policy.provider=PolicyClassName
```

缺省的属性为:

```
policy.provider=sun.security.provider.PolicyFile
```

通过改变属性值以指定另一个类，可以用一个新的 Policy类来代替它。例如：

```
policy.provide=com.mycom.MyPolicy
```

当这个Policy对象被初始化时,这个新的类被使用,而不是缺省的实现类Policy File。当指定一个可选的Policy类时，必须指定一个想要实现的 Policy实现类的合法名字，如com. sun . security . MyPolicyClass。

4.2.4 缺省的策略文件格式

为JDK安装的策略配置文件指定了由来自于指定代码源所允许的许可权 (系统资源访问的类

China-pub.com
下载

型)。对一个允许执行安全行为的applet或应用程序(例如读或写一个文件),它必须拥有对特定行为的许可权,在缺省的Policy实现中,这个许可权由策略配置文件中的授权输入项所赋予。⊖

缺省策略配置文件格式的语法包括一个输入项列表,它包括零个或多个输入项,该输入项以grant关键字和一个可选的keystore输入项开始。密钥库(keystore)是一个被保护的私钥数据库,它的相关数字证书,如X.509证书链,用来鉴别相应的公钥。缺省的密钥库的实现作为一个文件在JDK1.2中实现。X. 509证书将在4.4节讨论。可以使用keytool来创建和管理密钥库,在策略配置文件中指定的密钥库用于查找签名者的公钥,该签名者在文件的准许输入项中指定。如果任何准许输入项指明了签名者的别名,那么一个密钥库输入项会出现在策略配置文件中。

这时,策略文件只接受一个密钥库输入项——第一个之后的其他输入项均被忽略。这个输入项可以出现在文件的准许输入项之外的任何地方,且可以使用下面的语法:

```
keystore  "some-keystore-url","keystore-type";
```

其中"some-keystore-url"指定了密钥库的URL位置,而"keystore-type"指定了密钥库的类型。因此,如果策略文件在下面的安全属性文件中被指定:

```
policy.url.1=http://foo.bar.com/fum/some.policy
```

且这个策略文件有一个密钥库输入项"keystore",则密钥库将从下面的位置被加载:

```
http://foo.bar.com/fum/.keystore
```

密钥库的URL也会完全给出,如:

```
keystore //http://foo.bar.com/fum/.keystore"
```

密钥库类型定义了密钥库信息的存储和数据格式及用于保护密钥库中私匙和密钥库本身完整性的数学算法。JDK1.2所支持的缺省类型是一个被称为JKS的专有的密钥库类型。

被执行的代码总被认为是来自一个特定的代码源(由类型为CodeSoure的对象表示)。代码源不但包括applet起源处的URL,而且包括证书的引用,该引用包含与用于签名的私匙相对应的公匙。代码源中的证书由来自用户密钥库的别名引用。

策略文件中每个准许输入项基本上由一个CodeSoure和它的许可权组成。为了表示可能是CodeSoure的一部分的证书集合,一个策略文件简单地包括一列签名者的名字,这些名字借助于一个密钥库与实际证书相对应的别名。别名的设计是有用的,因为证书很大且包括二进制数据和不可打印的字符,而一个策略文件却便于核查和编辑。

每个grant输入项的许可权段包括大量的许可权输入项,下面为grant输入项的基本格式:

```
grant signedBy"signer-name",codeBase "URL"{
permission permission-class-name "target-name","action",
    signedBy "signer-names"
...
permission permission-class-name "target-name"," action",
   signedBy "signer-names"
};
```

准许输入项必须用grant开始。signedBy和codeBase名/值对是可选的。这些域的位置无关紧要。

⊖ 也有例外情况,代码自动被赋予许可权,可以读自身CodeSource和CodeSource子目录下的文件。代码并不需要明确的许可权就可以做到这一点。

China-pub.com 下载

signedBy域是可选的，因为若它被省略了，它指定“任何签名者”，即代码是否被签署和由谁签署都不重要。当它的值给定时，是一个别名字符串，该字符串与签名者相联系的公匙集相对应(利用密钥库)。这些密钥用于验证来自于指定代码源的类是否由它们的签名者签名，这些值可以是逗号分隔的字符串，字符串包括多个签名者的名字，例如"Adam,Eve,Charles"表示由“Adam和Eve与Charles签名”注意关系符为“与”而不是“或”。

简单地说，缺少codeBase输入项表示“任意代码”，即代码出处无关紧要。

策略文件格式的非正式的BNF语法如下：

```
PolicyFile --> PolicyEntry | PolicyEntry; PolicyFile
PolicyEntry -->    grant {PermissionEntry}; |
            grant SignerEntry {PermissionEntry} |
            grant CodebaseEntry {PermissionEntry} |
            grant SignerEntry, CodebaseEntry {Permission-Entry} |
            grant CodebaseEntry, SignerEntry {PermissionEntry} |
            keystore "url"
SignerEntry -->   signedBy (a comma-separated list of strings)
CodebaseEntry --> codeBase (a string representation of a URL)
PermissionEntry --> OnePermission | OnePermission Permission-Entry
OnePermission --> permission permission-class-name
            [ "target-name" ] [, "action-list"]
            [, SignerEntry];
```

注意codeBase的值是一个URL，因此一个正斜杠“/”(不是反斜杠“\”)被用于目录分隔符，即使代码源在MS-Windows系统上。例如，若MS-Windows系统上代码源位置为C:"somepath"app"，则策略codeBase输入项如下：

```
grant codeBase "file:/C:/somepath/api/"{
...
}
```

许可权输入项必须以Permission开始。在前面的语法中，Permission-class-name实际上为一个特定的许可权类型，如java.io.FilePermission或java.lang.Runtime Permission。

像read,write,access或其他一些行为，对许多许可权类型来说是必须的，如java.io.FilePermission(它指定的文件类型有访问许可权)。而一些行为不需要这些,如java.lang.RuntimePermission——可以获得由permission-class-name后的"targe-name"指定的许可权，也可以不获得这些许可权。

signedBy名/值对对于许可权输入项来说可选的。若选上了，它指明一个被签名的许可权。即Permission类本身由给定的别名签署，可使许可权被授权。如，假设有下面的许可权输入项。

```
grant{
   permission  Foo"foobar",signedBy"FooSoft"
}
```

若Foo.class许可权已经被“FooSoft”别名签署了，或者如果Foo.class是一个系统类，因为系统类不受策略的限制，这时，类型Foo被赋予许可权。

每个许可权签名者的域都被包括，以防当许可权类并不驻留在java运行安装程序中时的电子欺骗。例如，com.abc.TVPermission类的副本可以作为远程JAR文件的一部分被下载，且用

China-pub.com
下载

户策略也可包括指向它的输入项。因为记录并不是长期有效的，所以当 com.abc.TVPermission 类被第二次下载时，也许来自不同的 Web站点，但第二次复制是绝对可靠的。这是因为在用户策略中许可权输入项的出现也许反映了用户对第一个副本类字节代码的信任。

我们选择使用数字签名来确保可靠性，而不是存储字节代码的第一个副本，并把它和第二个副本比较。这样做是因为 Permission类的作者可以合法地更新类文件来反映一个新的设计或实现。

许可权输入项的条目可按以下顺序出现：

(permission、permission-class-name, "target-name", "action",signedBy, "signer-names")。

每个输入项以分号结束。对标示符 (permission,signedBy,codeBase等等)来讲，事件是不重要的，但对permission-class-name或任何作为值传递的字符串是重要的。

在java.io.FilePermission的说明书中，"targe-name"是一个文件路径。在 MS-Windows系统上，当在一个字符串中 (不是codeBase的URL)直接指定了文件路径时，需要使用两个反斜杠“ \\ ”来代替一个反斜杠，如下例：

```
grant{
  permission  java.io.FilePermission
"C:\\users\\cathy\\foo.bat","read";
};
```

这是因为这个字符串是由 tokenizer(java . io . StreamTokenizer)处理的，它使用“ \ ”作为一个转义字符串。举一个例子，“ \n ”表示换行。因此要求用两个反斜杠来表示一个反斜杠。在 tokenizer已经处理了上面的文件路径串后 ,即把两个反斜杠转换为一个反斜杠 ,那么结果为 "C:\users\cathy\foo.bat"。

4.2.5 策略文件举例

本节提供了几个策略文件的例子。下面为一个策略配置文件中两个输入项的例子。像 Java程序一样，以//开始的行为注解行。

```
// If the code is signed by "Duke", grant it read/write access
// to all files in /tmp:
grant signedBy "Duke" {
   permission java.io.FilePermission "/tmp/*", "read,write";
};

// Grant everyone the following permission:
grant {
   permission java.util.PropertyPermission "java.vendor";
};
```

以下是另一个策略配置文件例子：

```
grant signedBy "sysadmin", codeBase "file:/home/sysadmin/" {
   permission java.security.SecurityPermission
      "Security.insertProvider.*";
   permission java.security.SecurityPermission
      "Security.removeProvider.*";
   permission java.security.SecurityPermission
```

China-pub.com 下载

```
        "Security.setProperty.*";
};
```

这个例子指明了唯一的applet代码由本地文件系统上/home/syadmin/目录下被签署的JAR文件加载(applet的签名可以使用公钥来验证,该公钥由密钥库中名为sysadmin的别名提供引用),这个applet可以调用Security类的方法来添加或删除提供器或设置安全属性。

因为代码源包括两个组件：codeBase和signedBy，任一个(或两个)组件都可省略，下面的策略仍然有效：

```
grant signedBy "sysadmin" {
   permission java.security.SecurityPermission
      "Security.insertProvider.*";
   permission java.security.SecurityPermission
      "Security.removeProvider.*";
};
```

这个策略表明来自由sysadmin签署的JAR文件的代码可以添加或删除提供器，而无须考虑JAR文件的出处。下面的例子无签署人：

```
grant codeBase "file:/home/sysadmin/" {
   permission java.security.SecurityPermission
      "Security.insertProvider.*";
   permission java.security.SecurityPermission
      "Security.removeProvider.*";
};
```

这个例子中，代码来自本地文件系统下/home/sysadmin/目录下的任何地方，有权添加或删除提供器。这个代码不须签名。

下面的例子没有codeBase和signedBy：

```
grant {
   permission java.security.SecurityPermission
      "Security.insertProvider.*";
   permission java.security.SecurityPermission
      "Security.removeProvider.*";
};
```

在这个安全策略下，任何代码(不管它的出处，是否被签名，或由谁签名)都可以添加或删除提供器。显然，这个策略文件对许多情况都太随意了。

4.2.6 策略文件中的属性扩展

为了使策略配置和规范更为容易，JDK1.2允许策略文件和安全属性文件中的属性扩展。属性扩展类似于UNIX外壳中变量的扩展。即，当格式为｛some.property｝的字符串出现在策略文件或安全属性文件中时，它将会扩展为系统的属性值。假如有：

```
permission java .io. FilePermission"${user.home} ", "read";
```

当这个输入项被处理时，"$｛user.home｝"将会扩展为user.home系统属性。若系统属性值为"/home/cathy"，则前面的许可权行等价为：

```
permission java .io. FilePermission"/home/cathy", "read";
```

为了帮助产生与平台无关的策略文件，JDK1.2引进了特殊的符号"$｛/｝"作为"$｛file.separator｝"的快捷办法。因此可以写出下面的命令行：

China-pub.com
下载


```
permission java.io.FilePermission "${user.home} $ {/} *", "read";
```

如果使用Solaris系统，且user.home的系统属性为"/home/cathy"，前面的命令行扩展为：

```
permission java.io.FilePermission "/user/cathy/*", "read";
```

如果使用的为MS-Windows系统，且user.home系统值为"C:\home\cathy"则扩展结果为：

```
permission java.io.FilePermission "C:\user\cathy\*", "read";
```

举一个特殊的例子，如果codeBase字符串中的属性，如准许codeBase "file: $ {java.home} /lib/ext/"被扩展了，则系统认为你在UNIX系统上(因为斜杠的使用)且那个准许输入项中的所有文件分隔符被自动地扩展(或转换)为"/"。若这个输入项在MS-Windows系统上，则扩展结果为：

```
grant codeBase"file:C:/jdk1.2/lib/ext/"
```

即使java.home被设为C:\jdk1.2。因此，当指定一个codeBase字符串时，应该使用${/}。

因为属性扩展可以随时发生，所以策略文件允许出现双引号的字符串： "signer-names", "URL","target-name"和"action"等域都可以扩展。可以通过设置安全属性文件中 policy.expand Properties的值为false，来禁止属性扩展。这个属性的缺省值为 true。

嵌套式的属性并不能适当地扩展。例如， "${user.${foo}}"不会产生${user.home}，即使foo属性被设为"home"。这是因为属性的分析程序并不识别嵌套式的属性。它宁可简单地寻找第一个“ ${ ”并一直跟踪它，直到找到第一个“ } ”。它试图把这个结果 (此例为，“ ${user.$foo} ”)作为一个属性解释，但是当没有这样的属性时，解释失效。

如果在准许输入项中给出一个属性扩展 ,而属性扩展却失败了 ,那么这个输入项将被忽略。例如，假定系统属性foo没有被定义,那么有如下的代码:

```
grant codeBase "${foo}" {
   permission...;
   permission...;
};
```

则在这个准许输入项中所有的许可权被忽略了。

另一方面，如果有如下的代码：

```
grant{
   permission Foo "${foo}"
   permission Bar;
};
```

则唯一的permission Foo输入项将被忽略而Permission Bar被准许。

如果有"${foo}"密钥库，且系统属性foo未被定义，则整个密钥库的输入项都被忽略。

在tokenizer处理字符串后，字符串中属性扩展就会发生，因此 tokenizer首先处理这个字符串"${user.home}\\foo.bat",将双反斜杠转变为单反斜杠 ,结果为"${user.home}\foo.bat"。假定user.home的值为"C:\users\cathy", ${user.home}被扩展了，且最后结果为"C:\users\cathy\foo.bat"。在这个例子中，为了实现平台的独立性，应该一开始就指定字符串，且没有任何显式斜杠，即通过使用"${/}"属性来代替，就像在"${user.home}${/}foo.bat"中那样。

4.3 数字证书

在安全策略中，关键字signedBy用来指定一段代码被一个实体数字签名。这个实体可以是个人、组织、程序、计算机、商业、银行或其他。 signedBy只给出了实体的别名，而数据库调

China-pub.com 下载

用密钥库来维持一个别名和它的公钥之间的映射。实际上，公钥常常被存储在公钥证书内。

公钥证书(public-key certificate)是由一个实体数字签名的语句，用来表明另一个实体的公钥(和别的信息)有一些特定的值。根据该描述，一个公钥和与它有关的信息将被另一个公钥所证明，因此就有了一个证书链。

如图4-1所描述，在这里每个证书都包括一个用来证明后继证书公钥的公钥。顶层的证书常被称为根证书(root certificate)，没有任何公钥证明它。因此，它是一个自签名的证书，这意味着用它自己的公钥来证明它本身。

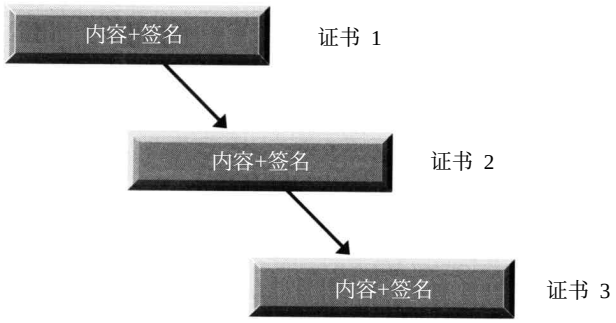


图4-1 证书链

根证书通常由证书机关发布，证书机关作为一个值得信赖的第三方 (TTP)发布证书，证书机关(CA)是一个商业实体，它可以值得信赖地为别的实体签署 (或发布)证书。通常认为CA只创建有效和可靠的证书。使用这样的证书表明着你信赖为该证书签名的实体。在某些情况下，在根或者顶层CA证书中，发布者签署他们自己的证书，可以利用许多公共的 CA，如VeriSign公司，但任何人可以运行他自己的CA，例如通过使用一个易于获得的商业服务产品。

为了促进互操作性，国际组织CCITT指定了国际通讯的标准，创建了一个标准的证书格式。这个证书被称为X.509，它的最新版本为X.509 v3。它是最广泛应用的格式，尤其在Web浏览器(如Navigator和IE)上很普及，这种格式支持SSL(安全套接字层)协议。SSL是一个安全协议，它为网络传输提供了保密性和验证，并且使用证书在浏览器和 Web服务器上协商和建立一个安全的通信通道。这是由IETF(Internet工程任务组)定义的，并且由传输层安全(TLS)协议继承。在安全电子邮件产品(如PEM和S/MIME)和电子商务协议(SET)中，X.509证书也可以用来签署JAR文件。

所有的X.509证书除签名之外还包括如下数据：

- 版本(version)：X.509标准版本适用于该证书，这个版本影响了证书中指定的信息，到目前为止，已经定义了三个版本。⊖
- 序列号(serial number)：序列号由创建证书的实体赋值，以便区别该证书和实体发布的其他证书。这个信息可以用在许多方法上，例如当一个证书被废除了，它的序列号被放置到一个证书废除列表上，证书废除列表 (Certificate Revocation List，CRL)是一个时间戳列表，可以用来鉴别被废除的证书。 CRL是由CA签署的，在公共仓库里可以自由得到。
- 签名算法标识符(signature algorithm identifier)：CA使用这个算法来签署证书。
- 发行者姓名(issuer name)：签署证书实体的X.500名字，通常为一个CA。
- 有效期(validity period)：证书有效的时间周期。每个证书都在一个有限的时间周期内有

⊖ IETF公钥基础结构工作组(PKIX) 正在为Internet公钥基础结构定义标准。

China-pub.com
下载

效，这个时间周期由一个起始日期和时间以及一个结束日期和时间来描述，可以短到几秒钟，也可以长到一个世纪。对实体来讲，只要相关的私钥没有捕获，它的有效期依赖于它的公共值。有效期的选择依赖于许多因素，如用于签署证书的私钥的强度和 (或)某人为证书所付的费用。

- 主体名字(subject name)：证书所鉴别的公钥的实体名。该名使用 X.500标准，在Internet上它是唯一的，该名是实体的特异的名字 (DN)，例如：

CN=Java Duke，OU=Java Software，O=Sun Microsystems，C=US

它们分别指主体的普通名字 (CN)、组织单元(OU)、组织(O)、国家(C)。附加的域包括localityName(城市名)，如Palo Alto，还有stateName(州或省名)，如California。

- 主体公钥信息(subject public key information)：被命名实体的公钥以及一个算术标识符，该标识符指定了这个公钥属于哪个公钥密码体制和有关的密钥参数。

证书中这些数据用两种相关的标准来编码，ASN.1/DER。抽象语法表达(ASN.1)描述数据，确定编码规则(DER)描述了存储和传输数据的单一方法。

证书通常使用可打印的编码格式来存储数据。该格式由 Internet RFC 1421标准，而不是由二进制编码定义。该证书格式，也称为 Base64编码。它便于向其他应用程序输出证书 (如借助于e-mail)。在Base64编码中，被编码的证书在开始和结尾处分别由

-----BEGIN CERTIFICATE-----

和

-----END CERTIFICATE-----

来限定。

可以使用三种版本的X.509。

(1) X.509 v1，1988出现，是一种被广泛配置的、最普通的版本。

(2) X.509 v2引入了主体和发行人独特标识符的概念来处理重用主体和 (或)发行人名字的可能性。多数证书的概述文件强烈建议名字是不可重用的，且证书不应使用独特的标识符。该版本的证书没有被广泛应用。

(3) X.509 v3是最新版本(自1996年以来)，支持扩展的概念。任何人都可以定义一个扩展并把它包含在证书内，目前使用的一些常用的扩展有：

(a) KeyUsage，它限制了特定目的密钥的使用，如只是签名。

(b) AlternativeNames，它允许其他标记也与公钥有关，如 DNS名、e-mail地址、IP地址。扩展可以被标记为“关键性的”来指定这个扩展应该被核查和执行 /使用。例如，一个证书标有“关键性的”keyUsage扩展，且设置为keyCertSign，那么如果这个证书在 SSL通讯期间被提出，它应该被拒绝。这是因为证书扩展表明这个相关的私钥只能为签名证书使用，而不能为SSL使用。

有许多方法可以得到证书，可以使用正确的工具来创建一个自签名的证书，如 keytool(本章随后会解释)。然而，一些人只接受由 CA签名的证书，CA提供的是一种中性的可信赖的介绍服务，这种服务一部分基于它自身的验证需要，这公开发布在它的认证服务应用 (CSP)中。

你还可以从CA处申请一个证书。这时，keytool有助于生成申请，该申请名为证书签名申请(CSR)。为了从CA处获得一个证书，需要一个相匹配的公钥私钥对，它们通常由一种特殊的工具生成(如keytool或浏览器)。还需要提供有关被证明实体的信息，如名字和地址，还需

China-pub.com 下载

要提供表明这些信息正确性的证据。然后在一个自签名的证书中提交所需的信息，以使 CA可以验证它的完整性。

为了访问和管理证书，JDK1.2包括丰富的Java API集。证书的API在java.security.cert包里，包括下面的类：

- CertificateFactory。定义了证书工厂的职能，证书工厂用来生成来自编码的证书 (和相关的CRL对象)。
- Certificate：管理不同证书的抽象类，这些证书具有多种不同格式。例如，不同类型的证书，像X.509证书和从加密工具Pretty Good Privacy(PGP)获取的证书，共享一些通用的证书功能(如编码和验证)和一些类型的信息(如公钥)。
- CRL:管理多种CRL的抽象类。
- X 509Certificate：X.509证书的抽象类。它提供了访问一个X.509证书所有属性的标准方法。
- X 509Extention：X.509扩展的界面，由X.509 v3证书和X.509 v2 CRL所定义。这些扩展提供了把附加属性和用户域公钥相联系的机制，如管理证书层次和管理 CRL分配。
- X 509CRL：X.509CRL的抽象类。
- X 509CRLEntry：CRL输入项的抽象类。

下一节讲述keytool，它可以生成、显示、引入和引出X.509证书。

4.4 有用的安全工具

为了帮助开发人员， JDK1.2带有以下的安全工具： keytool、policytool和jarsigner。本节将涉及这些内容。首先，我们讨论密钥数据库。

4.4.1 密钥数据库

在4.2.4节中提到过，密钥库是一个被保护的数据库，含有私钥和企业的证书。在 JDK1.2中密钥库的缺省实现是一个文件 ,由图4-2详解。密钥库的访问由口令保护。当创建密钥库时，通常由创建密钥库的人选好口令。密钥库是被保护的，只能由提供当前口令的人改变。另外，为了额外的安全性，密钥库中的私钥由它自己的口令保护。

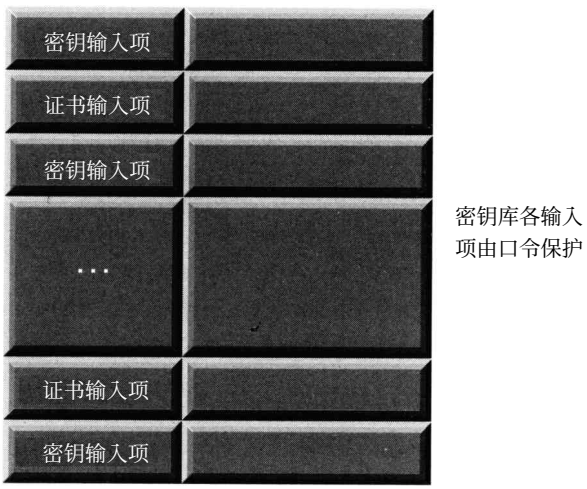


图4-2 密钥库

China-pub.com
下载

来自密钥库的信息可被其他工具使用 (如jarsigner)，以产生或证实 JAR文件的数字签名。JAR文件包括类文件、图像、声音和 (或)单个文件中的数字数据。 jarsigner通过使用它的证书 (位于JAR文件的签名数据块文件中)来验证JAR文件的数字签名。 jarsigner接着核查证书的公钥是否可信赖，即它是否包含在指定的密钥库中。⊖

密钥库包括两种输入项：密钥输入项和可信证书输入项。密钥输入项包含敏感的加密密钥信息，并且存储在被保护的格式中以防止不被允许的访问。一个存储在这种类型的输入项中的密钥，或者是一个密钥或者是伴随有相应公钥的证书链的私钥。 JDK1.2中的keytool和jarsigner不处理密钥。

可信证书输入项包括属于这个实体的一个单一公钥证书，称为可信证书 (trusted certificate)。这是因为通过将这个输入项接纳进密钥库内，密钥库的主人相信证书内的公钥确实属于由主体识别的它本身 —— 即证书的主人。证书的发行者通过签署证书来担保这一切。

访问所有的密钥库输入项(密钥和可信证书输入项)都要借助于独特的别名。别名对事件是不敏感的；例如，别名Hugo和hugo指向同一个密钥库输入项。当使用 -genkey命令来生成一个密钥对(公钥和私钥)以向密钥库中增加一个实体时，或者当使用 -import命令来增加一个证书或证书链到可信证书列表中时，也就指定了一个别名。随后的 keytool命令必须使用相同的别名用来指向这个实体。例如，若使用别名 duke来生成一个新的公/私钥对，并用下面的命令将公钥包含在一个自签名的证书内：

```
keytool -genkey -alias duke -keypass dukekeypasswd
```

这个命令指定了最初口令为 dukekeypasswd。这个口令对随后访问与别名 duke相关的私钥的命令是必须的。为了改变 duke的密钥口令，可使用下面的口令，将 dukekeypaswd的口令改为一个新口令newpass：

```
keytool -keypasswd -alias duke -keypass dukekeypasswd -newnewpass
```

为了更好的安全性，口令不应该在命令行中指定，除非出于测试目的或处于一个安全系统上。如果命令行中并没有指定一个必要的口令， 系统将提示安装一个。

前面讲过，密钥库缺省地以文件形式实现。每一个 keytool命令都可以为持久的密钥库文件指定名字和位置。在密钥库的生成期间，如果没有指定一个 -keystore选项，密钥库被缺省地存储在用户主目录的 .keystore文件中,该目录由user.home的系统属性来确定。在 Solaris系统上,user.home缺省为用户的主目录。在 MS-Windows系统上，如果用户叫 uName，则user.home的缺省属性值如下：

```
C:\Winnt\Profiles\uName          多用户Windows NT系统
C:\Windows\Profiles\uName        多用户Windows 95系统
C:\Windows                       单用户Windows 95系统
```

因此，如果用户名为cathy，则user.home缺省为：

```
C:\Winnt\Profiles\cathy          多用户Windows NT系统
C:\Windows\Profiles\cathy        多用户Windows 95系统
```

java.security包中的KeyStore类提供了访问和修改数据库信息的界面。不过，它有多种具体实现方法，并且，每一种实现对应特定类型的密钥库。同时，密钥库的实现也是基于提供

⊖ keytool和jarsigner工具代替了JDK1.1提供的javakey工具。这些新工具相对于javakey而言有更多的特点，它可以利用口令来保护密钥库和私钥并且可以验证证书，而不仅仅是生成它们。新的密钥库结构代替了 javakey创建和管理的等同数据库。可以通过 Keytool的-identitydb命令，把等同数据库中的信息输入至一个密钥库中。

China-pub.com 下载

器的。具体地讲，由KeyStore所提供的应用程序界面是根据一个服务提供器界面 (SPI)实现的。提供器(provider)是一个包或包集，它提供了可由Java安全API访问的服务子集的具体实现。因此，相应的KeystoreSpi类也属于java.security包，它定义了提供器必须实现的 SPI方法。为了提供一个密钥库的实现，客户必须实现一个提供器并且必须提供 KeystoreSpi子类的实现。第7章讲述了如何实现一个提供器。

应用程序可以从不同的提供器中通过调用 KeyStore类的getInstance方法选择不同类型密钥库的实现。密钥库类型定义了密钥库信息的存储和数据格式，以及密钥库所用的保护私钥的算法和密钥库本身的完整性。不同类型的密钥库在实现细节上并不兼容。在 JDK1.2中，密钥库的缺省实现是名为JKS的私有密钥库类型。密钥库类型对大小写不敏感。因此， jks可等同于JKS。

keytool只对基于文件的密钥库实现起作用。它将密钥库的位置信息以文件名的形式传递给命令行，并将其转换成含有密钥库信息的 FileInputStream。也可以在命令行指定一个密钥库类型。例如，如果你拥有一个提供器包，它为 pkcs12密钥库类型提供了密钥库的实现，那么可以使用下列命令：

```
keystore.type=pkcs12
```

如果你并没有清楚地指定一个密钥库类型， keytool依据安全属性文件中指定的keystore.type属性值来选择一个密钥库的实现。 KeyStore类定义了一个静态方法getDefaultType，调用这种方法，应用程序和小应用程序可以获得 keystore.type 的属性值。下一行代码为密钥库类型创建了一个缺省实例。

```
KeyStore  KeyStore=KeyStore.getInstance(KeyStore.getDefault-Type());
```

4.4.2 keytool

keytool可以创建和管理公/私钥对和自签名的证书，而且这些密钥和证书都包含在密钥库内，密钥库可用keytool管理。例如，你可以显示、引入和导出一些以文件形式存储的 X.509 v1、v2和v3证书并产生新的自签名的v1证书。⊖

keytool允许用户指定在Java的运行环境中注册的由加密服务提供器提供的密钥对的生成算法和签名算法。缺省的密钥对生成算法为数字签名算法 (DSA)。DSA密钥的大小在512和1024字节之间，但必须是64的倍数。算法中缺省的私钥大小为 1024字节。签名算法源于底层的私钥算法。如，若底层私钥类型为 DSA，则缺省的签名算法为带有DSA的SHA1，如底层的私钥类型为RSA，则缺省的签名算法为带有RSA的MD5。

keytool的缺省实现用于处理X.509的证书。假设给定一个DN字符串：

```
"CN=Mark Smith,OU=JavaSoft,O=Sun, L=Palo Alto, S=CA,C=US"
```

可以使用下面的命令为这个DN产生一个密钥：

```
keytool -genkey -dname "CN=Mark Smith,OU=JavaSoft,O=Sun,
   L=Palo Alto, S=CA,C=US"alias mark
```

关键字的缩写对大小写不敏感，例如， CN、Cn和cN都是等同的。但是，关键字的每个子部件必须严格按CN、OU、O、L、S、C的顺序。当然并不是必须包含所有的子部件， DN字

⊖ 即使底层的证书包支持 X.509 v3 格式，但是由于需要处理多种扩展名和选项，命令行会变更很复杂，所以keytool只能产生X.509 v1 格式的证书。人们自然想到扩展或用户化 keytool，以便利用v3 格式的优点。

China-pub.com
下载

符串中允许使用若干子部件组成的子集如：

```
CN=Mark Smith,OU=JavaSoft,O=Sun, C=US
```

如果一个命令需要一个DN部件字符串，但命令中并未给出，用户将被提示给出DN字符串。

keytool可以创建和管理密钥库的密钥输入项，输入项包括一个私钥和相关的证书链。链中的第一个证书包括与私钥对应的公钥。当创建密钥时，链只有一个元素，即自签名的证书。当产生一个私/公钥对时，公钥被包含在自签名的证书内。当 CSR产生后，再传递给CA，CA的响应被导入，自签名的证书被证书链所替代。在链的底部是由 CA发布的证书，用来证明主体的公钥。

链中的下一个证书用于证明CA的公钥。通常，它是一个自签名的证书，也是链中的最后一个证书。在其他情况下， CA返回一个证书链。其中，链的底部证书是相同的，但链中的第二个证书却由一个不同的CA所签名，它可被用于CSR的CA公钥的证明。下一个证书证明CA的第二个公钥等等，直到自签名的根证书。因此，链中的每一个证书用于证明链中的前一个证书的签名者的公钥。

许多CA只返回发布的证书，没有证书链，特别是在没有层次结构的链中就不存在中介CA。这时，证书链只能通过存储在密钥库中的可信证书信息建立。

在PKCS#7标准中，定义了另一个附加的响应格式，包括支持的证书和发布的证书，所有的响应格式都由keytool处理。

根CA证书是自签名的。不过，由于任何人都可以产生带有 DN的自签名的证书，所以根公钥的可靠性并不来自根证书自身。在你向密钥库中增加根 CA证书之前，应该确保它的真实性。例如，假设证书在一个名为/tmp/cert的文件中。当你考虑在可信证书链中增加一个证书时，执行-printcert命令来查看它的指纹，例如：

```
keytool --printcert --file /tmp/cert
Owner: CN=11, OU=11, O=11, L=11, S=11, C=11
Issuer: CN=11, OU=11, O=11, L=11, S=11, C=11
Serial Number: 59092b34
Valid from: Thu Sep 25 18:01:13 PDT 1997 until: Wed Dec 24 17:01:13
    PST 1997
Certificate Fingerprints:
MD5: 11:81:AD:92:C8:E5:0E:A2:01:2E:D4:7A:D7:5F:07:6F
SHA1: 20:B6:17:FA:EF:E5:55:8A:D0:71:1F:E8:D6:9D:C0:37:13:0E:5E:FE
```

实际上，在向密钥库的可信证书列表中增加一个证书之前， keytool将打印出证书信息并提示你对其进行验证，当然，你也可以放弃输入操作。

keytool可以从一个文件中导入证书，如，响应下面的命令：

```
keytool --import--alias  joe --file jcertfile.cer
```

这个命令向jcertfile.cer中导入证书，并存储在别名为joe的密钥库中。证书可以以Base64或二进制代码的形式由-import和-printcert命令读出。你可以向可信证书列表中导入一个证书或导入一个从CA处收到的证书响应，作为向CA所提交的CSR的结果。由别名选定采用哪一个导入。如果别名存在于数据库中并采用私钥来证实输入项，则应该导入一个证书响应。keytool核查证书中的公钥响应是否匹配与别名密钥库所存储的公钥，如果用别名识别一个已存在的证书输入项，则新的证书将不会被导入。否则密钥库将会生成，并与输入的证书关联。

向文件中导入一个证书，使用-export命令，形式如下：

```
keytool  --export  -alias  jane  --file  janecertfile.cer
```

China-pub.com 下载

这个命令将jane的证书导入janecertfile.cert文件，缺省情况下输出一个二进制编码的证书，当然也可以输出一个Base64证书。

为了打印密钥库输入项的内容，可以使用 -list命令：

```
keytool -list -alias joe
```

如果别名未被指定，则密钥库的内容将被打印， -list命令缺省地打印证书的MD5指纹。如果采用-verbose选项，将以一个可读的格式打印证书。

keytool有下列内置的缺省值：

-alias	**"mykey"**
-keyalg	**"DSA"**
-keysize	**1024**
-validity	**90**
-keystore	用户主目录下的**.keystore**文件
-file	若读则为标准输入文件，若写则为标准输出文件

在JDK1.2的在线文档中对keytool命令有详细的解释。下面的命令输出帮助信息：

```
keytool
keytool -help
```

keytool用法举例

下面是一个生成和管理密钥库的例子，这个密钥库含有公 /私钥对和所信任实体的证书。

首先，需要产生一个密钥库来生成密钥对，可以使用下面的命令：

```
keytool -genkey -dname "cn=Mark Smith,ou=JavaSoft,o=Sun,
   c=US" --alias business  --keypass  kpi135 --keystore /working/
   mykeystore -storepass ab987c  --validity 180
```

这个命令在working目录下产生mykeystore密钥库(假设在这之前这个目录并不存在)，并给它赋予一个口令(storepass)ab987c。同时也为实体产生公/私钥对，这个实体的DN(dname)是MarkSmith，组织机构单位名称是JavaSoft，组织机构是Sun，国家代码是US。它使用缺省的DSA密钥生成算法来产生1024字节的密钥。

这个命令也产生一个自签名的证书(使用缺省的带DSA的SHA1签名算法)，包含公钥DN信息。该证书有效期为180天，与别名为business的密钥库输入项的私钥有关。私钥被赋予的口令为kpi135。

如果接受系统缺省的选项，只需给出没有指定和缺省的选项值，这样可以简单地利用下面的命令：

```
keytool --genkey
```

这时，就产生了一个带有别名为 my key的密钥库实体，同时生成一个新的密钥对和有效期为90天的证书。本节的其余例子都假设执行了 -genkey命令，故只需对提示的值做出响应，这些值等价于本节最初使用-genkey命令例子中的值。

到目前为止，你已经有一个自签名的证书。由 CA签名的证书更易被其他人信任。为了获得这样一个签名，必须先产生一个CSR，可以使用下面的命令：

```
keystool  --certreq  --file MarkJ.csr
```

这个命令生成一个CSR (为由缺省别名mykey所识别的实体)，并在MarkJ.csr文件中放置请求。接着，向CA提交这个文件， CA会将你作为请求者来证明(这属于离线操作)，并返回一个由它签名的证书来证明你的公钥(有时，它返回一个证书链)。

China-pub.com
下载

你需要用证书链来替代自签名的证书，链中的每一个证书都识别链中前一个证书的签名者的公钥，一直到根 CA。在你从CA处导入证书响应之前，需要一个到多个可信证书。可以这样定义它们：

(1) 若证书响应为一个证书链，则只需要链中的顶层证书 (即证明CA公钥的根CA证书)

(2) 若证书响应为单个证书，则发行的 CA需要一个证书，若那个证书不是自签名的，则它的签名者需要一个证书，直到一个自我认证的根 CA证书。

JDK1.2的缺省密钥库中有 5个VeriSign根CA证书，所以你需要向你的密钥库中导入一个VeriSign证书来作为可信证书。但若你从不同的 CA处请求一个签名的证书，由于你的密钥库并不包含一个证明CA公钥的证书，故需要从CA处导入一个可信证书。

来自CA的证书或是自签名的，或是由别的 CA签名(这时你需要一个证书来证明 CA的公钥)。如公司ABC是一个CA，你获得了一个名为 ABCCA.cer的文件，这个文件包括一个来自ABC的自签名的证书，它可用来证明 CA的公钥。在将它作为一个可信证书导入之前，需要确定它的有效期。若你信任这个证书，可以使用下面的命令，将它添加到你的密钥库中：

```
keytool --import --alias abc --file ABCCA.cer
```

这个命令在密钥库中生成一个可信证书输入项，具有来自 ABCCA.cer的数据，这个输入项别名为abc。

一旦你导入一个证书来证明 CA的公钥，这个CA是提交你的CSR的CA，则可以导入证书响应，用证书链来替代自签名的证书。这个证书是根据你的 CSR返回的(若CA响应为证书链)，或是利用一个证书响应和可信证书构造的 (若CA响应为单个证书)。

例如，你向VeriSign发出你的CSR，可以使用下面的命令来输入响应 (假设返回的证书为VSMarkJ.cer)：

```
keytool --import -trustcacerts --file VSMarkJ.cer
```

假设使用jarsigner来签名JAR文件，打算使用该文件的用户将会证明你的签名。这些用户会向他们的密钥库中输入你的公钥证书来作为一个可信输入项，或者是你导出证书并将它提供给你的客户。例如，你可以将你的证书复制到别名为 MJ.cer的文件中(假设输入项别名为mykey)：

```
keytool --export --alias mykey --file MJ.cer
```

使用证书和签名的JAR文件，客户可以用jarsigner来证明你的签名。

假设你的DN变化了，如你转换了部门或到了一个不同的城市，在更新 DN时，仍可使用相同的公/私钥对。例如，若你叫SuSan Miller,可以利用别名sMiller和DN串：

```
"cn=Susan Miller, ou=Finance Department,o=BlueSoft, c=us"
```

生成你最初的密钥。

如果你从金融部调到会计部，仍可以使用先前的公 /私钥，DN则需要用下列命令来更新。

(1) 复制密钥输入项：

```
keytool --keyclone --alias sMiller --dest sMillerNew
```

由于命令行中没有提供私钥口令，将采用缺省值 storepass。

(2) 更改与这个副本相关的证书链，以使证书链的第一个证书能使用新的 DN。首先用合适的名字产生一个自签名的证书：

```
keytool --selfcert --alias sMillerNew -dname "cn=SuSan Miller,
   ou=Accounting Department,o=BlueSoft c=us"
```

China-pub.com 下载

(3) 利用新证书的信息产生一个CSR：

```
keytool --certreq --alias sMillerNew
```

并导入CA证书响应：

```
keytool --import --alias sMillerNew --file VSSMillerNew.cer
```

(4) 可以删去在先前DN串中最初的密钥输入项：

```
keytool --delete --alias sMiller
```

从JDK1.2在线文档中可看到，MS-Windows中也含有keytool，它的用法与Solaris版相同。

4.4.3 策略工具

policytool可以创建新的策略文件和修改已存在的策略文件。在命令行中键入 policytool命令，可以显示Policy Tool窗口(见图4-3)。

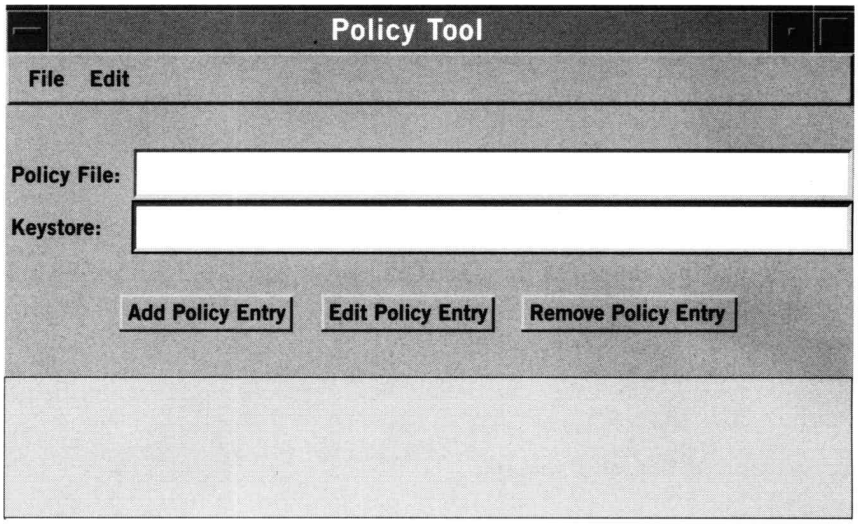


图4-3 policytool瞬态图

PolicyTool窗显示的是你所调用的用户策略文件的相关信息，包括策略文件名、密钥库URL、codeBase和策略文件中每个策略输入项的 signedBy部分，该文件在你的主目录下缺省名为.java.policy。如果policytool找不到这个文件,它就会给出提示并且显示一个空的 Policy Tool窗口(第一次运行policytool时，不存在用户策略文件，除非你已经手工创建了一个)。然后可以打开任何一个你想在上面工作的策略文件，或者通过增加策略输入项、有选择地指定一个密钥仓库以及存储该文件来创建一个新策略文件。

例如，假设你想指定一个位于 /tests/目录，名为mykeystore的密钥仓库。在Solaris系统上，可以进行以下操作：

(1) 把下面的URL输入至标有New KeyStore URL的文本框：

```
file：/tests/mykeystore
```

policytool可以从由URL指定的任意位置读取一个密钥仓库。

(2) 如果需要的话，可以通过把类型输入文本框 New KeyStoreType来指定密钥仓库类型。例如，由Sun Microsystem支持的JKS keystore类型。如果你没有清楚地指定一个密钥仓库类型，policytool选择一个密钥仓库实现，该实现基于在安全属性文件中指定的 keystore.type属性值。

China-pub.com
下载

(3) 点击OK。标有密钥仓库的文本框显示密钥仓库的 URL和类型。

为了增加一个新的策略输入项，进行以下操作：

(1) 点击Policy Tool窗口下的Add Policy Entry按钮，显示Policy Entry对话框。

(2) 使用对话框以指定一个可选 codeBase输入项，该输入项指示代码起源的 URL位置。例如为了指示来自于local/JavaSoft/TESTS/目录的代码，把下面的内容输入 CodeBase文本框：

```
file：/JavaSoft/TESTS/
```

或者把下面的内容输入文本框：

(a) 可选signedBy输入项是来自于密钥仓库的别名，用来给签名者提供参考 —— 使用了谁的私钥给代码签名。例如，为了指示别名 duke，把duke输入到signedBy文本框。

(b) 一个或更多许可权输入项指示把哪一个许可权赋予代码，该代码来自于由 codeBase和signedBy值指示的源。(如果没有指定该值的话，可以赋予任何代码。)

为了增加一个新的许可权，进行以下操作：

(1) 点击Policy Entry对话框中的Add Permission按钮以显示Permission对话框，然后输入或选择以下内容：

(a) 许可权类型(permission type)。为了指定由你或其他人定义的许可权类型，把许可权类型输入至文本框，或双击 Permission下拉菜单下的内置类型之一。完整的类型名出现在下拉菜单右边的文本框。

(b) 许可权目标名(permission target name)。如果你从Permission下拉菜单中选择了一个内置许可权类型并且该类型的许可权有具体的目标名值，然后可以从 Target Name的下拉菜单中选择一个值。在这种情况下，目标名的可能取值是无穷的。但有一些有特殊含义的内置的目标名规范，这样的目标名将会出现在下拉菜单中。例如，《ALL FILES的特殊的目标名将会出现在File Permissions菜单中。

为了指定下拉菜单中的目标名不可用，把目标名输入至TargetName下拉菜单右边的文本框中。

(c) 一个或更多行为(one or more action)。如果行为是相关的。一些许可权只有一个目标名而没有行为。针对这种情况，把文本框放到 Actions下拉菜单空白的右边(文本框将自动变黑并对内置许可权类型不可用)。为了许可权能够申请行为规范，把逗号分割的行为输入至文本框或从下拉菜单中选择。例如，为了指定对一个规定 FilePermission的文件的读写访问，首先从菜单中选择读(或者写 —— 顺序无关紧要)。读出的词将会在文本框中出现。然后选择写。写入词将被附加上，并由一个逗号和一个空格前导。

(d) signedBy别名(signedBy alias)。如果需要的话，把别名输入至 SignedBy右边的文本框中。一个许可权输入项的 signedBy值是可选的。如果选中的话，它表明了一个签名的许可权。也就是说，为了赋予许可权， Permission类自身必须由给定别名签名。

(2) 完成指定许可权信息，点击OK。

新的许可权出现在Policy Entry对话框中的行中。如要加入更多的许可权可以重复上述步骤。

一旦完成加入策略输入项，点击 Policy Entry对话框中的Done，显示出一个包含新策略输入项行的Policy Tool窗口，该行只包括 codeBase和signedBy信息。如果在Policy Entry对话框中，两种信息都没有指定，那么将会显示 codeBase<ALL>。如果策略输入项中包括任何在你的密钥仓库中还不存在的别名，那么在你关闭策略输入项时将会显示警告信息。在这种情况下，点击OK并且创建一个别名的注解，或者编辑策略输入项以修复别名 (如果别名有误的话)。

China-pub.com 下载

也可以编辑或移开一个已存在许可权。为了编辑一个已存在许可权，进行以下步骤：

(1) 在Policy Entry对话框中，点击许可权行，然后点击 Edit Permission按钮。(或者双击许可权行。)

显示Permission对话框。当加入一个新的许可权时，对话框将会正常工作，除非对话框充满了被选择许可权的信息。

(2) 为了改变信息，可以从下拉菜单中重新选择，或者替换文本框中的信息。

(3) 完成这些工作后，点击OK。

Policy Entry对话框显示由你修改后的许可权。

为了删除一个已存在许可权，选择 Policy Entry对话框中该许可权的行，然后点击 Remove Permission按钮。

如果policytool报告说警告已经被存储至 Warning Log，可以点击 Edit菜单中的 View Warning Log命令以观察该记录。例如，如果你有一个带密钥仓库 URL的策略文件，该URL指定了一个还不存在的密钥仓库，那么你将会在不同的时候得到警告，例如当你打开文件，警告将会存储到记录中。你可以继续对策略文件进行操作，而不用管警告的存在。

MS-Windows包括一个policytool版本，该版本本质上相同的方式工作。

4.4.4 jarsigner

JDK1.2引入了一个名为jarsigner的新工具。JAR可以封装类文件、图像、声音和单个文件中的其他数据，以利于更快、更容易的分配。一个名为 jar的工具使开发人员可以生成 JAR文件。jarsigner可以给JAR文件签名并且验证签名和已签名 JAR文件的完整性。把数字签名同一个JAR文件相联系有助于保证该签名可以被验证，这种验证通过重新计算基于目前 JAR内容的签名以及把该签名同存储的签名做比较来进行。如果两者不匹配，则意味着或者是 JAR文件中的内容，或者是 JAR文件中的签名被修改了。这样的话，只要私钥仍然保密，没有私钥的人就不能伪造一个已签名JAR文件。

jarsigner使用私钥和来自于一个密钥仓库的证书信息来生成 JAR文件的数字签名。这样的话，当使用jarsigner来给一个JAR文件签名，首先必须给密钥仓库指定一个 URL，以及为密钥仓库输入项指定别名，该输入项包含用来生成签名的私钥。例如 ,在当前工作目录下的密钥库mystore中，用一个与别名duke相关的私钥将JAR文件标识为MyJARFile.jar，由于没有指定输出文件,所以已标识的JAR 文件用MyJARFile.jar 替代。

```
jarsigner -keystore /working/mystore -storepass myspass
-keypass dukekeypasswd MyJARFile.jar duke
```

因为密钥库也许属于不同的类型，所以如果不明确指明密钥库的类型，那么 jarsigner将选择一个基于keystore.type属性值的密钥库。keystore.type的属性在安全性属性文件中指定。

目前，在JDK1.2中，jarsigner的缺省运行只能标识由JDK jartool创建的ZIP和JAR文件。[⊖]此操作采用带有DSA的SHA1算法和带有RSA的MD5算法来标识 JAR文件。带有DSA的SHA1算法由一个缺省的SUN提供器提供。

当用jarsigner标识一个JAR文件时,已签名的输出JAR文件名与输入的JAR文件名一致。除

⊖ JAR文件与ZIP文件基本相同，除了后者有一个 META-INF/MANIFEST.MF文件。该文件当jarsigner对一个ZIP文件签名时自动创建。

China-pub.com
下载

此以外，在目录META-INF中增加了两个附加文件：带 SF扩展名的签名文件和带 DSA扩展名的签名块文件。

一个签名文件(SF文件)看上去更像一个列表文件，这是因为对于每个包含在 JAR文件的源文件，都包含有文件名，摘要算法名和 SHA摘要值。每一项都以单独一行列出。在列表文件中，每个源文件的SHA摘要值是源文件中二进制数据的散列。在 SF文件中，给定源文件的摘要值是源文件的列表文件中三行的散列，并且 SF文件缺省地包含一个由整个列表文件的散列组成的文件头。文件头可以优化验证，这一点将在本章的后半部分中加以讨论。⊖

SF文件被标识后，签名位于 DSA文件中。DSA文件包含证书和证书链，可用来鉴定与已签名的私钥相应的公钥。 jarsigner 可使用证书或证书链来验证签名。

如果签名是有效的或者当JAR文件的签名生成后，文件再也没有被改动，那么 JAR文件的验证是成功的。JAR文件验证包含下列步骤：

(1) 验证SF文件本身的签名，此验证确保存储于每个签名块 (DSA)中的签名是由与公钥相对应的私钥产生，而且公钥的证书 (或证书链)出现在DSA文件里。同时也确保签名是相关 SF文件的有效签名，因此SF文件是不能随意改动。

(2) 验证SF文件中列出的摘要与列表文件中的相应部分是否一致。 SF文件也许包含一个文件头。如果文件头存在，那么验证首先只是简单的核查这个文件头是否与列表的文件的文件头真正匹配，如果两者一致，那么就进行下一步验证。否则 SF文件中的源文件信息必须核查它是否与列表文件的相应部分一致。存储于 SF文件中的列表文件也许和当前的列表文件不相等，这样列表文件就发生了变化 (增加的部分就是新文件)。但是SF文件没有改变。如果感兴趣的仅仅是SF文件，那么验证还是成功的。

(3) 验证每一个SF文件提及的文件， jartool读取每一个在SF文件中标识的JAR文件，读取时，它计算文件的摘要并将结果与列表文件中的相应部分作对比，摘要结果应一致，否则其他的验证将终止。如果在验证过程中，任何安全性敏感验证失败，那么验证过程终止，并由jarsigner提示出现安全异常。

一个JAR文件可由多人在不同时刻简单地运行 jarsigner来签署，而且每次为不同的人指定别名，命名如下：

```
jarsigner myBundle.jar susan
jarsigner myBundle.jar kevin
```

当JAR文件在不同时刻被签名时， JAR文件包含多个SF和DSA文件。每对SF和DSA文件对应于一个签名，例如：以前举例的文件若采用此方式签名，输出的 JAR文件包含下列命名：

SUSAN.SF

SUSAN.DSA

KEVIN.SF

KEVIN.DSA

参见JDK1.2中jarsigner选项的在线帮助。

jarsigner 与JDK1.1的兼容性

在JDK1.1中，提供一种称为javakey的工具来运行签名任务。这个工具对于复杂的签名来

⊖ 被签名的文件头也可用于封装存储在 JAR中的Jave 软件包，以使别的类不属于同一个包，除非该类被同一个签名密钥所签名。

China-pub.com 下载

说，显得过于简单。在JDK1.2中已被keytool和jarsigner所替代。这些工具提供了比javakey更多的特性，并增加一些功能，例如：通过口令来保护keystore和私钥，除了生产签名外，还可验证签名。

javakey生成并管理的身份数据库是安全策略和密钥库的组合。在JDK1.2中，一种新的密钥库结构体替代了这个身份数据库，而且采用了更加标准的存储格式以确保向后兼容性。JDK1.2具有以下特性：

- 通过keytool中的-identitydb命令把来自身份数据库的信息导入密钥库，只有值得信任的数据才可导入JDK1.2的密钥库。
- jarsigner可签署以前由javakey签署的JAR文件。
- jarsigner可验证由javakey签署的JAR文件。因此可识别和利用签名者的别名，这些别名是从JDK1.1数据库产生的，而不是来自于JDK1.2密钥库。

因为密钥库在安全策略中起着非常重要的作用，随之而来的问题是如何将JDK1.1中javakey生成的二进制数据模式与JDk1.2中的细微模式集成。更为特别是，在一个由身份数据库、密钥库与安全策略混合组成的运行环境里，当JVM处理一个JAR文件时，JVM授予JAR文件中的类何种许可权?在解决这个问题的同时，还有许多其他问题必须仔细考虑。

- 问题1：JAR文件被标识了吗？
- 问题2：在身份数据库中，存在这个被验证的签名者吗？签名者值得信赖吗？
- 问题3：已验证的签名者已经从身份数据库导入密钥库了吗？
- 问题4：安全策略是否已明确赋予已验证的签名者许可权？

表4-1显示了已在JDK1.1x中标识的JAR文件在JDK1.2中是如何被处理的。缺省许可权表示由系统缺省地赋予给所有代码的许可权。策略许可权表示明确地赋予特定签名者集的许可权。所有许可权表示那些隐含着其他许可权的许可权。选择项是/否表示是要么含有,要么不含该选项。

表4-1中 ,还有几点需要解释，第一，在策略文件中提及的签名者 (真实身份或别名)，为了对赋予的许可权施加影响，必须位于密钥库中。第二，如果在身份数据库中存在策略文件和密钥库的组合，那么该组合拥有最高优先级，于是与在JDK1.1中一样，并不是将所有的许可权赋予可信的签名者。第三，JDK1.1身份数据库中不可信身份在JDK1.2中被忽略，如果不可信身份出现在keystore中并在策略文件中提及，那么JDK1.2的策略是有效的，而且身份数据库输入项被忽略。否则JAR文件被当作未标识文件处理。最后，如果可信身份没有出现在密钥库中或者在策略文件中也没有提及，那么JAR文件以二进制模式处理，而且赋予签名者所有许可权。

表4-1 组合javakey和密钥库时的许可权

问题1	问题2	问题3	问题4	许可权
否	是/否	是/否	是/否	缺省
是	否	是/否	否	缺省
是	否	是	是	缺省+策略
是	是/不可信	是/否	否	缺省
是	是/不可信	否	是	缺省
是	是/不可信	是	是	缺省
是	是/不可信	是/否	否	所有
是	是/不可信	是	是	所有
是	是/不可信	是	是	缺省+策略

China-pub.com
下载

4.4.5 代码签名实例

下面讲解一个JAR文件签名和验证的例子。假设有一个JAR文件bundle.jar，期望用用户的私钥签名，该用户位于working目录下，其密钥库名为mystore，密钥库别名是jane。假定密钥库的口令是mypass，jane的私钥口令是j638klm，则可以通过下列命令签名JAR文件，并将签名的JAR文件命名为sbundle.jar。

```
jarsigner -keystore  /working/mystore -storepass myspass
-keypass j638klm -signedjar  sbundle,jar  bundle.jar  JANE
```

相应的SF和DSA文件名为JANE.SF和JANE.DSA。

通过下列命令可以验证被签署的文件：

```
jarsigner -verify sbundle.jar
```

如果验证成功，显示"jar verified"信息。否则显示出错信息。为了了解更多的验证结果信息，可以使用带有-verbose选项的命令：

```
jarsigner -verify -verbose sbundle.jar

      198 Fri Sep 26 16:14:06 PDT 1997 META-INF/MANIFEST.MF
      199 Fri Sep 26 16:22:10 PDT 1997 META-INF/JANE.SF
      1013 Fri Sep 26 16:22:10 PDT 1997 META-INF/JANE.DSA
smk 2752 Fri Sep 26 16:12:30 PDT 1997 AclEx.class
smk 849 Fri Sep 26 16:12:46 PDT 1997 test.class

s = signature was verified
m = entry is listed in manifest
k = at least one certificate was found in keysyore

jar verified.
```

如果命令中再加上-certs选项，输出将包含下列信息：

- JAR文件的每一个签名者的证书信息，包含证书类型。
- 签名者的DN信息(当且仅当证书符合X.509标准)。
- 如果JAR文件的公钥证书和密钥库输入项匹配，在圆括号内可表示出签名者的密钥库别名。

下面是一个例子：

```
jarsigner -keystore mystore -verify -verbose -certs myTest.jar

        198 Fri Sep 26 16:14:06 PDT 1997 META-INF/MANIFEST.MF
        199 Fri Sep 26 16:22:10 PDT 1997 META-INF/JANE.SF
       1013 Fri Sep 26 16:22:10 PDT 1997 META-INF/JANE.DSA
        208 Fri Sep 26 16:23:30 PDT 1997 META-INF/JAVATEST.SF
       1087 Fri Sep 26 16:23:30 PDT 1997 META-INF/JAVATEST.DSA
smk    2752 Fri Sep 26 16:12:30 PDT 1997 Tst.class

X.509, CN=Test Group, OU=Java Software, O=Sun Microsystems, L=CUP,
   S=CA, C=US (javatest)
X.509, CN=Jane Smith, OU=Java Software, O=Sun, L=cup, S=ca, C=us
   (JANE)

s = signature was verified
m = entry is listed in manifest
k = at least one certificate was found in keystore
```

China-pub.com 下载

```
jar verified.
```

如果签名者的证书不符合 X.509标准，则显示信息中不包括 DN信息，只含有证书类型和别名。例如，如果证书是PGP证书，别名为bob，则显示信息为：

```
PGP , (bob)
```

如果JAR文件用JDK1.1的javakey工具签名，那么签名者具有身份数据库的别名，验证输出信息含有i符号。如果JAR文件同时被身份数据库和密钥库的别名签署，则验证输出信息中含有i和“ k ”符号标识。当使用-certs选项时，身份数据库的别名信息显示在方括号中 (如下例中的duke)，而不像密钥库别名那样显示在圆括号内。例如：

```
jarsigner -keystore mystore -verify -verbose -certs writeFile.jar

         198 Fri Sep 26 16:14:06 PDT 1997 META-INF/MANIFEST.MF
         199 Fri Sep 26 16:22:10 PDT 1997 META-INF/JANE.SF
        1013 Fri Sep 26 16:22:10 PDT 1997 META-INF/JANE.DSA
         199 Fri Sep 27 12:22:30 PDT 1997 META-INF/DUKE.SF
        1013 Fri Sep 27 12:22:30 PDT 1997 META-INF/DUKE.DSA
smki    2752 Fri Sep 26 16:12:30 PDT 1997 writeFile.html

X.509, CN=Jane Smith, OU=Java Software, O=Sun, L=cup, S=ca, C=us
   (JANE)
X.509, CN=Duke, OU=Java Software, O=Sun, L=cup, S=ca, C=us [duke]

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.
```

MS-Windows中也包含jarsigner，它本质上与上述工作方式相同。

4.5 非专业安全策略管理

本章已经讨论了 JDK1.2安全结构中的技术细节，如何配置安全策略、创建密钥和证书、Java签名类。对非专业的计算机用户来说，复杂度似乎是不可克服的。 JDK1.2富含安全结构，由于必须适应更广范围的应用，如在可编程安全企业应用程序中。这自然使系统变得很复杂。

对于非专业人士，处理好这个问题，有两种方法：一种方法是安排专门人员管理系统安全。在企业中，系统管理员和信息资源部门可以代表公司雇员来负责创建和实施安全策略。在本章和后续章节中可见， Java安全体系设计已经为用户考虑并介绍了多种安全策略方法。例如，雇员被引导配置他们自己的浏览器以直接显示中心控制主页，从而获得当前安全策略，或者公司可以定制一种浏览器并发布给每一位雇员。

企业应用程序开发者也可以以某种方式并入安全策略管理，这样用户不必去处理甚至不必知道底层的安全管理特性。在公司外部， Internet服务供应商(ISP)也能提供好的安全管理方法。例如，许多ISP已经对外提供有限的安全机制如防火墙和垃圾邮件过滤器。这样， ISP在可执行内容和移动代码方面提供安全策略管理帮助就显得合情合理。

第二种方法，就是关注人机界面。由于每个人的研究领域不同，对大多数的计算机和Internet用户来说，理解安全方面的术语和方案等极其困难。而且，不同的人对不同的事情理

China-pub.com
下载

解也不一样，这就难以用统一的描述方式去适应形形色色的人们。因此， JDK1.2并不试图为大众提供统一的人机界面去处理安全策略和管理问题，而是希望软件供应商能把人机界面功能融入他们自己的系统环境中并定制这些内容，从而为不同用户提供不同的人机界面。

例如，计算机系统供应商们传统上是同时提供安全管理软件和操作系统软件，在不远将来，可将Java技术与两者捆绑或集成在一起，并且附带的安全管理软件将为管理 Java安全问题加入一些合适的组件。这样，原有安全管理软件的使用者可以继续使用熟悉的软件和熟悉的界面。

此外，应用程序开发者可以选用这种对用户不可见的嵌入式安全方案。例如，基于 Java可提供AOL美国在线型Internet访问和用户经历的应用程序，这样应用程序就具有 Java的许多安全特性，如：动态组件升级和对 e-mail信息的安全访问。因此，为实现安全管理的应用程序就必须依靠广泛的安全技术。在这种情况下，应用程序固定采用某种特殊的安全策略而不必提供任何用户化功能。

这样，除了最初的登录过程，用户不必处理更多的安全问题，甚至不必知道整个应用程序所具有的安全特性。

安全管理和用户界面仍在研究中，一是由于 Internet使安全技术首次成为主流并被人们所关注。二是由于过去的技术通常不把安全性作为设计目标。随着时间推移和更广泛的安全方案的实施，开发者对安全技术的关键环节会获得有价值的见解。

China-pub.com
下载

第3章　JDK1.2安全结构

本章主要讨论JDK1.2安全结构的内部机制，它支持策略驱动、基于许可权的、灵活的及可扩展的访问控制。本章将回顾一下 Policy和Permission类的设计方法、安全类加载的内部机制及访问控制算法。下面首先概述这种新的结构产生的动机和它的发展历史。

3.1　起源

1996年8月底，JDK1.2安全结构开始被酝酿，次年2月开发了相应的代码，3月设计出第一个许可权控制AppletViewer appletviewer原型。1997年5月获得具有完备特性的AppletViewer原型。它们和说明Java安全新结构的出版物大约同期。在1997年2月的IEEE COMPON[29]会议上，我发表了一篇论文，后来，在 IEEE Micro 5/6月双月刊[23]上对这篇论文作了一些修改和扩展。1997年4月在JavaOne 的讨论中，我可以自信地对手边的原型讲出它们的一些技术细节。在后来的12个月中，安全结构基本保持不变，而 API却在不断的精化当中。在 1997年12月的USENIX会议上发表了关于API的一个概要性文件[25]。1998年初，在一个国际社会专题讨论会上，安全结构的技术细节被发表[28]。

我们把JDK1.2的安全工程形象地称为“直布罗陀(Gibraltar)”⊖，这主要是因为Java安全开发小组人员把它看作是 Java技术的重要基石，而且它的顺利完成也颇费周折。目前，开发小组人员正在构想用其他大力神柱 (古代称为Abyla，现在称为Mount Hacho)来命名下一代安全工程。

3.2　为什么需要一个新型的安全结构

如在第2章讨论的，由于并没有多少技术人员把安全性作为设计目标，而 JDK1.0的发行版着重考虑了安全问题并提供了沙盒安全模型，所以，可以说正是在 Java技术、Internet和电子商务的推动下使安全技术成为计算机工业的主流，这是一个非常重要的成就。接下来应该改进最初的设计方案，弥补早期版本中的一些不足之处，使 Java平台上的安全解决方案更易于使用、更为强大。

3.2.1　关于applet的沙盒模型的局限性

沙盒模型严格地限制了几种行为，而这些行为 applet可以执行。尽管沙盒模型是产生安全网络计算的必要因素，但它却把所有的 applet看成是潜在可疑的。因此，一些 applet程序，例如那些公司财务组编制出的用以处理内部交易的程序，即使它们比从陌生的 Web站点随意下载的applet 程序可靠许多，但在使用上也要受到限制。

像这样对所有的 applet的全面限制是有局限性的。例如，假设总部在旧金山的经纪公司Charles Schwab的一个客户正在运行该公司网页上的 applet程序来进行股票交易，他也许想用

⊖ Gibraltar一词来源于calpe，两个大力神柱之一，见Brewer's Dictionary of Phrase & Fable。

China-pub.com 下载

applet更新Charles Schwab的公事包中的本地文件；不过，沙盒模型禁止访问客户方的文件系统。因此，这位客户需要灵活的访问控制，某些 applet可以访问沙盒以外的东西，或者说，沙盒模型应该可以定制(例如，由客户系统)成灵活的外形和边界。

另一方面，这位客户也许会在本地计算机上安装 Quicken 软件来处理所得税事务，它也许会感到极不舒服，因为 Charles Schwab的applet可以自由地控制它的整个计算机系统。在这种情况下，最好对 applet所能访问的文件系统加以限制，让它只能访问 Charles Schwab文件夹，即就是说，需要“ 细致的访问控制系统 ”。

在JDK1.2之前，Java平台理论上可以获得更为灵活和细致的访问控制。为了具体实现它，一些人 (如应用程序员)必须要做大量的编程工作，例如划分子类和定制 SercurityManager ClassLoader 等类。由Sun公司开发的支持Java开发环境的Internet浏览器HotJava便是这样的例子，它提供一系列有限的用户可定义的安全特性。但是，这样的编程工作对安全性极为敏感，需要丰富的计算机安全知识和强大的编程能力。

JDK1.2的结构旨在消除编写安全代码的必要。少数情况除外，如在军事环境中，因为它需要特别的安全特性(如多级安全特性)[44]。即使在军事环境下，在JDK1.2平台上编写安全代码也会变得更为简便和安全。

3.2.2 策略和实施分离的不彻底性

沙盒模型，由SecurityManager 类编码，以软件的形式实现了一种特定的安全策略，而这种软件以实施特定的安全策略为主要目的，这意味着要实施不同的安全策略，必须使用一种特定版本的软件 —— 显然这并不是我们所需要的，相反，我们所需要的是一种支持一系列易于配置的安全策略的底层结构。

JDK1.2安全结构清楚地分开了实施机制和安全策略声明。因此，应用程序员和用户无须编写特殊的程序就可以配置自己喜欢的安全策略。

3.2.3 安全核查的不易扩展性

最初的设计使得 JDK执行的安全核查难以编写相应的代码。例如，当检查一个文件是否可读打开时，需要调用 SecurityManager类中的checkRead()方法。显然，这样的设计方法不适合今后的JDK的新型核查处理，所以它不易扩展。并且，也不易改变规模。例如，为产生一种新型访问核查，如某人去检查钱是否从银行帐户中提出一类的事情，必须要在 SecurityManager类或它的子类中增加checkAccoutWithdraw()方法。许多不同目的的核查可能同时存在，如果为这些大量的核查而生成相应的方法，可能会造成 SecurityManager 类的过度拥挤。事实上，许多核查都是针对特殊应用的，所以并不是所有的核查都可以在 JDK内部定义。我们所需要的是一种易于扩展的访问控制结构。 JDK1.2结构引入了可定义类型的访问控制许可权和一种自动处理机制来实现可扩展性和规模可改变性。在理论上， SecurityManager 类中不需要再加入新的方法。在 JDK1.2的发展过程中，当我们引入大量新型安全核查时，并不需要新的方法，一个名叫 checkPermission()的方法，已经足以处理所有的安全核查。

3.2.4 对本地安装的applet过于信任

最初的安全模型默认所有本地安装的 Java应用程序都是可以完全信任的，应该在最高的

China-pub.com
下载

优先级下运行。这样使沙盒模型只适用于下载的 applet。然后，不应该赋予本地安装的软件对系统中所有部分的访问权力。例如，一个用户在本地安装了 Demo版的程序并试着运行它，如果赋予这样的一个 Demo版程序享有最高访问许可权，则会引起一些潜在的危害。又如，将applet存储在本地文件系统中会改进性能，但是不应该把 applet视为可靠的代码并改变安全模型，即使这些applet已存在于本地系统中。而且，由于存储 applet，本地代码和远程代码的区分很快会变得模糊不清。在现代社会的软件组织中，一个应用程序会用到许多组件，如JavaBeans，而它们常常存在于Internet的每一个角落。所以，安全核查应该扩展到所有的 Java程序，包括applet和应用程序。

在新的结构中，像applet一样，本地代码遇到了同样的安全控制，不过用户可以决定是否给某些(或全部)本地代码赋予最高访问许可权，这样的话，才能像完全被信任的代码一样有效地运用它们。这样一个选择可以通过简单地配置合适的安全策略来完成。

3.2.5 内部安全机制的脆弱性

在最初的JDK1.0和随后的JDK1.1中，一些内部安全机制在设计和实现时所用的技术是相当脆弱的。尽管它们在这些版本中运行得很好，但维护和扩展它们相当困难。因此，为了降低在程序中产生这些小安全漏洞的危险性，我们作了一些重要的结构性调整。这些工作涉及到对SecurityManager和ClassLoader类的设计和实现上的修正，和基本访问控制核查机制的修正。下面的3.8和3.9节会谈到一些历史上的细节。

3.2.6 总结

总之，由于需要支持灵活和细致的访问控制的安全策略，同时具有可扩展性和规模可变性，所以需要一种新型的经过改进的安全结构，这就是 JDK1.2。这种新型结构使用安全策略来决定赋予它的运行代码何种访问许可权。这些许可权是基于代码特征的。例如，代码源于何处，它是否为数字签名。还有，企图访问被保护的资源会引发调用安全检查，被调用安全检查将比较赋予的许可权和企图访问的许可权。如果赋予的许可权高于或包括企图访问的许可权，访问是允许的。否则，访问被拒绝。如果一个安全策略没有明确地给出，传统的实现于JDK1.0和JDK1.1中的沙盒模型策略为默认策略。随后章节将会谈到对这种模型使用的诸多告戒、精化和异常情况。

的确，我们不得不设计新的方法去处理诸多微小的安全事例，而这些安全事例对面向对象系统来说也是独一无二的，但是， JDK1.2安全结构并没有产生新的计算机安全理论，不过，它提供了一个现实的例子，在这个例子中，把众所周知的安全原则应用到工程实践中从而构造了一个实际的、广泛应用的安全系统。

本章的其余部分描述了类实现的技术细节。新安全模型中最主要部件包括：安全策略、访问许可权、保护域、访问控制核查、优先级操作和 Java类的加载和实施方案。安全策略和访问许可权定义了可允许的行为，而保护域和访问控制核查提供了实际的实施，优先级操作和类加载、实施方案在整个保护机制中提供了很有价值的帮助。

3.3 java.Security.GeneralSecurityException

首先，我们要指定一个异常类 GeneralSecurityException。也许有人不禁会问，已有

China-pub.com 下载

java.lang SecurityException 类，为何要引入这个类呢？SecurityException类和它的子类是一种运行异常，只在安全核查失败时出现，如一个人非法访问一个被保护文件。这样的运行异常不会被声明也不会被核查，但它们会中断程序运行，除非应用程序员编写了详尽的代码才能捕获它们。

尽管如此，其他一些错误情况(如语法错误)与安全机制有关，但却不是失败的安全核查。在这种情况下，执行一个Secu\rityException类是不正确的，而应当执行GeneralSecurityException。例如，引入一个无效Policy对象与安全性有关，但并不很重要，并且这种异常不是安全侵犯且应该被开发人员捕获和处理。

GeneralSecurityException是java.lang.Exception的一个子类，必须声明和捕获。这种异常在所有安全包中都将出现，除非发现了某种安全侵犯，这时会出现 SecurityException。⊖

3.4 安全策略

Java运行系统的安全行为由安全策略来指定。在抽象性术语中，安全策略是一个典型的访问控制矩阵，即什么样的系统资源，用哪种形式，在哪些环境下可以被访问。例如，图 3-1 中矩阵的一项“当运行一个从 http://java.sun.com 下载的applet时，允许它读文件 x”。更具体一些，一个安全策略是从描述运行代码的一套特征到代码所允许的访问许可权的映射。



图3-1 策略矩阵

JDK1.2运行时，系统的安全策略由一组策略组成，可由用户或系统管理员配置。 Java运行环境外的策略有许多种表示形式。如， JDK1.2的缺省实现使用 ASCII形式，且策略存储在ASCII文件中。策略文件可由HTTP或其他协议获得，策略文件的格式规范将在第 4章中说明。

因此Java运行环境内的安全机制可以引用策略，策略内容必须在 Policy对象内表达，而Policy对象是java.security.Policy类的子类的实例化。由于任何人都可以实例化这样的一个对象，因此Policy 对象的许多实例可以同时存在。然而，任何时刻都只有一个 Policy对象起作用，感觉上是安全机制在对 Policy实例提问。

Policy类是一个抽象类，所以Policy对象是从Policy子类，而非Policy类实例化来的。而安全策略由 Policy子类来表示，这个子类提供这个 Policy类中抽象方法的一个实现。下面为Policy的四个重要方法：

⊖ java security包由产生异常的两部分组成，它们是 java.lang.RuntimeException的子类。JDK 1.1 中包括该类，并保留了向下兼容性，在此我们不改变 GeneralSecurityException的子类。

China-pub.com
下载


```
public static Policy getPolicy();
public static void setPolicy (Policy policy);
public abstract Permissions getPermissions(CodeSourcecodesource);
public abstract void refresh();
```

通过调用getPolicy方法可以得到当前安装的Policy对象。这个对象保存着策略的运行代表，这个对象或者在Java启动时实例化，或者当安全策略首次使用时实例化。这个对象随后可以借助安全机制改变，如通过调用setPolicy方法。

策略信息的源位置由Policy对象使用直到Policy实现。它可以以简单的ASCII文件形式存储，也可以以Policy的连续二进制文件或者数据库形式存储。

refresh方法使得Policy对象更新或重装当前配置，这依赖于实现。例如，如果Policy在配置文件中存储它的策略内容，调用refresh会使它重读配置策略文件。然而，JDK1.2中的缺省实现并不影响已加载的类，它们仍保留着被赋予的许可权，即使这些许可权和新安全策略冲突。同时，在策略更新后加载的新类在新策略下也不赋予许可权，这些都依赖于类加载器是如何实现的。例如，如果它们存储了旧的策略内容，那么将会使用这些内容而不是新策略内容。3.6节将进一步详述策略更新问题。

可以通过policy.provider安全属性的值(在Java安全属性文件中)将缺省的Policy实现变为Policy实现类的合格的名字。Java安全属性文件位于：

```
JAVA-HOME/lib/security/java.security
```

其中JAVA-HOME指安装JRE(Java运行环境)的目录。例如，如果你在Solaris上安装了JDK1.2，安全属性文件位于：jdk1.2/jre/lib/security/java.security。如果你在Solaris上安装了JRE1.2，这个文件位于:jre1.2/lib/security/java.security。

把策略部件设计成一个提供器结构，以便有足够的灵活性。这样的话，就可以随意地获得策略内容。期望通过多种方法实现策略部件，然后为策略部件设计提供丰富的API是不太可能的。

策略内容很敏感，getPolicy()方法是公用的、静态的，任何人都可以调用它，这样就安装了一个合适的安全核查，使得只有那些得到许可权，可以获取策略的代码才能成功的调用这些方法。

类似的，调用setPolicy()方法时，触发了安全核查。如果调用的代码没有所需的许可权，则出现SecurityException，这表明有人正尝试做安全敏感性操作，并且由于没有充分的访问许可权而遭到了拒绝。

安全核查基于JDK1.2引入的新的安全结构(这样可以根据前面的理论作一个练习)，可以由下面的代码段加以解释。这些代码段使用新的许可权模型，它的细节将在3.5、3.7、3.8和3.9节中讲解。

```
public static Policy getPolicy() {
   SecurityManager sm = System.getSecurityManager();
   if(sm!=null) sm.checkPermission(new
      SecurityPermission("getPolicy"));
   return policy;
}
public static void setPolicy(Policy policy) {
   SecurityManager sm = System.getSecurityManager();
```

China-pub.com 下载

```
        if(sm ! = null) sm.checkPermission(new
            SecurityPermission("setPolicy"));
        Policy.policy=policy;
    }
```

应注意到：因为getPermission和refresh方法是抽象的，它们必须由 Policy类的子类实现。在这个子类中执行了正确的安全核查以保护 Policy对象的内容。

下面的java.lang.classloader代码段是关于 Policy如何使用的例子，这个代码段的defineClass方法在缺省许可权下定义类时，如下所示间接执行。

```
    ...
    Class c = defineClass0(name,b,off,len);
    c.setProtectionDomain0(getDefaultDomain());
    ...
```

其中getDefaultDomain()方法由下面的代码实现：

```
    ...
    CodeSource cs = new CodeSource(null,null);
    PermissionCollection p = Policy.getPolicy().getPermissions(cs);
    return new ProtectionDomain(cs,p);
```

3.5 CodeSource

前面已经讲过，安全策略本质上是一个访问控制矩阵，它根据代码特点和赋予的许可权权来描述代码，这一节的主要内容是如何描述代码。

代码的特点通常体现在两个方面。一方面是它的起源 (它的位置由URL来表明)，另一方面是包含公钥的一组数字认证，这里的公钥与利用一组或多组数字签名算法来签署代码的私钥相对应。在类java.security.CodeSource中，这些特点得以体现。类 java.security.CodeSource可以看作HTML中代码库概念的扩展。注意不要混淆 CodeSource类和HTML中的CodeBase标签，这一点很重要。

例如，一个applet的CodeSource以JAR文件封装，其文件名 foo.jar,Web地址是: http：//java.sun.com/classes/，故其URL为: http / /java.sun.com/classes/foo.jar。

如果JAR文件是已签名文件，它将含有 JAR文件中单独进入的数字签名或整个 JAR文件的自身签名。在这种情况下，相应的 CodeSource将会包含与签名密钥相对应的证书。请注意：如果签名不能确认，那么 JAR文件将会视作未签名的，且证书将被视为空。确认失败有两种可能原因，或者因为 JAR文件的内容被改动以至于与它的签名不再相匹配，或者因为签名密钥未能识别。

下面是调用CodeSource类最重要的方法：

```
CodeSource(URL url,  java.security.cert.Certificate certs[]);
public boolean equals(Object obj);
public boolean implies(CodeSource codesource);
```

通过把URL和证书都放到构造器里，以及复制证书 (而不仅仅保留那些认证对象的引用)，有意使CodeSource成为不可变的。注意到 URL本身已经是不可变的，所以没有必要对其进行复制。通过使CodeSource对象不可改变，可以保证这个对象在完整性不受破坏的情况下传送。CodeSource对象的完整性很重要， 3.6节将讨论其原因，访问控制决议部分基于运行代码的

China-pub.com
下载

CodeSource。例如，允许一个来自于指定 CodeSource的代码段写入本地文件系统，而禁止其他地方的代码访问本地文件系统。如果后一 CodeSource对象被非法改变，变成与前面CodeSource一致，那么后一类代码就可以非法访问本地文件系统，这就对安全造成了损害。

你或许已经注意到了，由于只需要用私钥来建立签名，且只需要用公钥对签名进行验证，证书有时就显得无关紧要。那么为什么 CodeSource类的界面只处理证书而不处理公钥呢？答案为：为了简单。理论上两种界面都可以存在，一种界面处理公钥，而另一种界面处理证书。但同时保留两种界面是多余的，这样会给下面的代码和算法增加复杂性。 JDK1.2 beta3及以前版本，只使用公钥作为界面方式。从 beta4以后，只采用处理证书的界面方式。只用证书不会产生任何问题。因为给定了一对公钥和私钥，可以轻易地生成自签名的证书。实际上，JDK1.2中使用了名为keytool的生成密钥工具，这个工具在生成密钥的同时也生成了自签名的证书。通常情况下，一个自签名的证书不会给它内部的密钥传送任何信息，除非它是传送密钥的中介。

而且，利用证书而不是公钥会使传送重要信息更容易，这些信息也许在证书内，却不能用公钥本身来表达。例如，因为 CodeSource既包含证书，又支持证书链。故可以使直到根 CA的整个证书链生效。出于审查的目的，这样的有效信息是很重要的。

3.5.1 测试等同性和利用隐含

测试两个CodeSource对象的等同性很重要，因为这样的比较是安全策略决定的中心问题。如果两个对象的URL位置相同，包含的证书相同，那么就可以认为这两个 CodeSource对象相同。也就是说，两组证书或许不会以相同的次序存储 (在阵列中)，但两组证书必须是一致的。

有时可以指定第一个CodeSource对象的代码比第二个更普遍 ,所以任何来自第二个对象的代码都可认为也来自第一个对象。这时，第一个 CodeSource “ 隐含 ” 第二个类。例如， http：//java.sun.com/classes的CodeSource比更具体的http://java.sun.com/classes/foo.jar 的CodeSource普遍的多。

基于这种 “ 隐含 ” 关系，通过给一个普遍的 CodeSource对象赋予许可权，可以简化安全策略，这意味着给许多特定的 CodeSource对象赋予相同的许可权。例如，赋 http java. sun. com/classes/许可权访问本地文件系统，同时也意味着给驻留在 Web页上的所有代码赋予了相同的许可权。

显然，必须遵守严格、简洁的规则以确定是否一个 CodeSource对象隐含另一个对象。在调用this.implies(CodeSource codesource)方法时，若this所指CodeSource对象隐含另一个特定的CodeSource对象，该方法返回值为真。更具体一些，这个方法使下面的核查按顺序进行。只要有一个核查失败，该方法返回值为假。如果所有的核查都成功，该方法返回值为真。

(1) codesource不能空。

(2) 如果该对象证书不空，则它们必须在 codesource的证书中表达。

(3) 如果该对象位置(getlocation())不空，则下列的核查不利于该位置和 codesource的位置：

(a) codesource的位置必须非空。

(b) 如果该对象的位置与codesource的位置相同，那么立即返回真，否则继续下列核查。

(c) 该对象的协议(getLocation().getProtocol())必须与codesource的协议相同。

China-pub.com 下载

(d) 如果该对象的主机 (getLocation().getHost())非空，那么由该对象主机构造的SocketPermission类必须隐含由codesource的主机构造的SocketPermission类.

(e) 如果该对象的端口(getLocation().getPort())不等于－1(也就是说，如果端口是具体指定的)，那么该对象的端口必须与codesource的端口相等。

(f) 如果该对象的文件(getLocation().getFile())与codesource的文件不相等，那么必须进行下列核查。

- 如果该对象文件以“ / ”结束，那么codesource的文件必须以前缀的形式包含这个文件。
- 如果该对象文件以“ /* ”结束，那么codesource的文件必须引用某个目录下的类或 JAR 文件，目录由没有后缀“ * ”的该对象文件指定。
- 如果该对象文件以“ /- ”结束，那么codesource的文件必须引用某个目录下的类或JAR文件，目录或者由没有后缀“ * ”的该对象文件指定，或者是该对象的任何子目录。
- 在其他情况下，作为前缀的codesource文件必须包括有“ / ”后缀的该对象文件。

(g) 如果该对象的引用 (getLocation().getRet())非空，那么它必须等于 codesource的引用。

例如，考虑以下位置的CodeSource对象和空证书。

```
http:
http://*.sun.com/
http://java.sun.com/classes/
http://java.sun.com/classes/foo.jar
```

所有这些隐含着位于http://java.sun.com/classes/foo.jar的CodeSource对象和空证书。这是因为http:http://*.sun.com和http://java.sun.com/classes/都包括http://java.sun.com/classes/foo.jar这一特殊的情况。

如果两个不同的CodeSource对象相互隐含的话，那么它们指向相同的代码源。

因为CodeSource实现了界面java.io.Serializable，所以出于连续化的原因，提供了可定制的方法writeObject()和readObject()。

下面是CodeSource类的样本用法，当定义一个从本地主机或远程主机加载的类时，为了询问安全策略以弄清代码的许可权，需要计算出它的代码源。因此使用java.net.URLClassLoader类中的findClass()方法执行下面的代码段。

```
...
URLClassPath ucp = new URLClassPath(urls);
...
Resource res = ucp.getResource(path,false);
...
byte[] b = res.getBytes();
java.security.cert.Certificate[] certs = res.getCertificates();
CodeSource cs = new CodeSource(url,certs);
...
return defineClass(name,b,0,b.length,cs);
```

3.6 许可权层次

前一节引入了安全策略以及包含一半策略内容的代码源。策略的剩余内容描述了赋予不

China-pub.com
下载

同代码源的许可权，本节首先讲述Permission类层次的设计，然后是各种特定的许可权类。

许可权类代表了对系统资源的访问控制。一般来说，所有的许可权类是肯定的，这是因为它们代表了赞成而不是拒绝的访问。这种设计极大地简化了实现，提高了效率。 Permission类层次的根类， java.security.Permission是一个抽象类，它的子类适合代表特定的访问控制。例如，下面的Java代码可用来产生读/tmp目录下，名为abc文件的许可权。

```
perm=new java.io.FilePermission("/tmp/abc","read");
```

新的许可权或者是Permission类的子类，或者是它的子类的子类。如 java. security . Basc Permission(见图3-2)。子类的许可权(不同于BasicPermission)通常属于它自己的包。因此可以在包里找到描述文件系统访问许可权的 FilePermission，而java.io包掌握着对于文件系统访问的API。

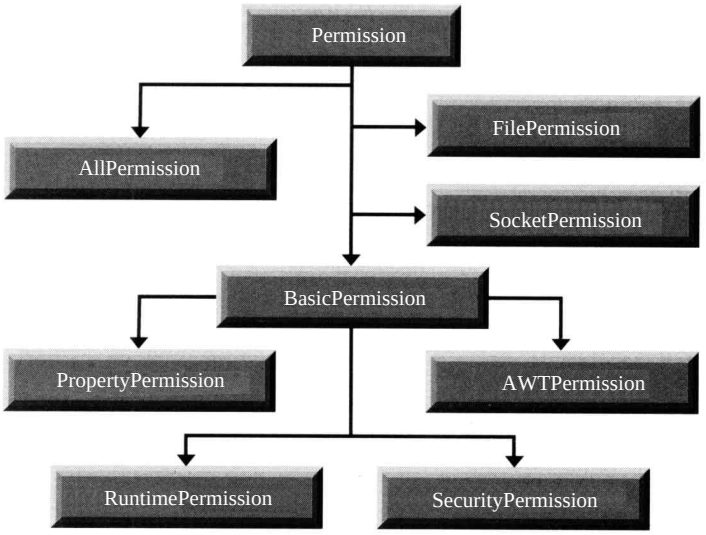


图3-2 Permission子类

3.6.1 java.security.Permission

下面是构造器和在Permission类根层次调用Permission类的方法:

```
public Permission(String name);
public abstract boolean implies(Permission permission);
public abstract boolean equals(Object obj);
public String toString();
public PermissionCollection newPermissionCollection();
```

每个Permission实例通过向构造器传递一个或多个字符串参数产生。通常情况下，有两个参数，第一个参数一般是目标的名字(如许可权目标的文件名)，第二个参数是行为(如读文件)。通常，一组行为可由逗号分隔的复合的字符串来确定。

所有许可权都有目标名，它的说明依赖于子类。对某种类型的许可权来说，目标名不重要，因而也无须解释，这是可以理解的。 Permission对象同String对象类似，这是因为一旦生成，它们都是不可变的。一旦子类已经生成，子类就不应该提供改变许可权状态的方法。

China-pub.com 下载

两个Permission对象是否相同，要看 Permission类的每个子类。对那些用来定义特定Permission子类语义的抽象方法来说，这一点同样适用。

为了比较许可权，implies是每个子类必须实现的重要的方法。“许可权p1隐含许可权p2”，意味着如果被赋予许可权 p1，自然也就被赋予许可权 p2。因此，这并不是一个真正的等同性测试，更应该说是一个子集测试。

记住对象的等同性不同于许可权的等同性，这一点很重要。对象的等同性很有用，如当在散列函数表中存储对象时，随后要确定这个实体是否已存在。这可通过调用方法实现。而许可权等同性，则意味着两个对象语义上表示相同的许可权。为了确定许可权等同性，必须调用方法来核查是否一个许可权对象隐含另一个，反之亦然。

大多数Permission对象也包括有对许可权目标可允许行为的列表。例如，对一个 java. io . FilePermission对象来说，许可权名(和目标)是一个文件(或目录)的路径名和对指定文件(或对特定目录下的文件)所赋予的行为列表。

这种行为列表对那些不需要这种行为列表的Permission对象是可选的。如在java.lang.Runtime Permission中，被命名的许可权(如exitVm)或者被准许或者没有，对不同的行为不必再细分。确实，对一些特殊情况，名字不但体现了许可权目标(如VM是exit的目标)而且体现了行为(exit)。简而言之，它们被合并成一个字符串。这种许可权类的设计通常是java.security.BasicPermission的子类。

有时，希望用可读的形式给出一个许可权的内容。 toString()方法返回一个描述许可权的字符串。可采用以下形式指定一个类名、许可权名和行为。

```
("ClassName" "name" "actions")
```

Permission类实现了两个界面：Guard和java.io.Serializable。后者是为了使Permission对象可以传输到远程机器，如借助 RMI，因此Serializable的表示是有用的。 Guard与GuardObject类有关，将在第6章讨论。应用程序(和applet)可以超出系统总是支持的类目录范围自由地引入新的 Permission类目录。如何增加一个特定应用程序许可权将在第 5章讨论。

NewPermissionCollection()方法对一个给定的 Permisssion对象返回一个空的 Permission Collection。若相应的 PermissionCollection类未定义，则返回空。下一节将讨论许可权集合。

3.6.2 许可权集合

处理多个许可权集比一次只处理一个许可权更方便。抽象类 java.security.Permission-Collection表示了一个单个类目 (例如文件许可权)便于编组的Permission对象集合(允许复制的一个集合)。每个PermissionCollection对象有一组同类的许可权集合。即 :每个实例都只有相同类型的许可权。下面为重要的方法调用 :

```
public abstract void add(Permission permission);
public abstract boolean implies(Permission permission);
public abstract Enumeration elements();
public void setReadOnly();
public boolean isReadOnly();
```

add方法增加一个Permission对象到当前的Permission对象集。这些工作留给每一个子类完成。如，文件许可权可以以任何次序加到 PermissionCollection对象中。

类似于在 Permission类中的目的， implies方法核查是否一个指定的许可权被当前

China-pub.com
下载

PermissionCollecton对象中的一个或多个许可权隐含。若是的话，可认为这个许可权由PermissionCollection对象隐含。注意在这种情况下，特定的许可权 (如读或写文件x)不应有一个单一的许可权隐含，而应由一个 PermissionCollection对象中的一个许可权集隐含。所以，当调用implies方法时，确保当前PermissionCollection的任何具体的子类保证符合正确的语义是十分重要的。

setReadOnly方法把PermissionCollection对象标识为只读。这样一来，没有一个新的Permission对象可以使用addPermission加入PermissionCollection对象。

为了对同类型的许多Permission对象编组，应该先对这种类型的Permission对象的特定类型调用newPermissionCollection方法。缺省的行为 (来自于Permission类)简单地返回空。有时，Permission的子类需要把它们的许可权存储到一个特定的 PermissionCollection对象中，这是为了当调用PermissionCollection.implies方法时可以提供正确的语义。这时候，它们忽略这种方法。若返回了一个非空值，则必须使用 PermissionCollection。若返回空值，则 new Permission Collection的调用者在它选择的任何 PermissionCollection中存储给定类型的许可权 (可以用Hashtable，Vector等等)。

java.security.Permission类表示一组Permission对象的集合，即一个各种许可权的超级集合。PermissionCollection的子类本质上是一组对象的集合，也就是说，它包含不同类型的Permission对象，这些对象组织成 PermissionCollection。例如，任一个加入该类实例的java.io.FilePermission对象都被存储在一个单一的 PermissionCollection中。通过调用FilePermission类中的newPermissionCollection方法可以返回一个PermissionCollecion。类似地，RuntimePermission对象存储在 PermissionCollection中，而PermissionCollection通过调用RuntimeCollection类中的newPermissionCollection方法返回。这样的话，该类表示一个PermissionCollection的集合。

下面两种方法值得一试：。

```
public void add(Permission permission);
public boolean implies(Permission permission);
```

当调用add方法增加一个Permission时，这个Permission存储在适当的PermisionCollection中。若这样的集合不存在，Permission对象的类就确定了。在这个类中，调用newPermissionCollection方法以产生PermissionCollection并把它加到Permission对象中。若newPermissionColection返回空，那么将会创建和使用一个缺省的使用散列函数表的PermissionCollection，每一个散列函数表的入口都有一个PermissionCollecion对象名作为密钥以及这个Permission对象作为值。

类似于对Permission Collection的行为，implies方法核查是否该对象的Permission Collection(指定许可权类型的许可权集)隐含过去的许可权对象中表达的许可权。如果适当 Permission Collection(例如一个FilePermission的FilePermissionCollection)中的许可权组合一起按指特定的许可权，那么implies方法返回真。

注意，这两个许可权集合都不是java.security.Permission的子类。

3.6.3 java.security.UnresolvedPermission

回忆一下Java运行环境的策略(指明了来自不同代码源的代码可使用的许可权)由Policy对象表示。尤其是安全策略的内部状态由与代码源CodeSource相关的Permission对象表达。因此，

China-pub.com 下载

当一个策略对象更新或初始化时，在策略允许的条件下，相应类的 Permission对象可以为了所有的许可权而创建。

许多由策略配置引用的 Permission类的类型局部地存在 (也就是说，这些类的类型可以在CLASSPATH中找到)。为这一类许可权而建立的类可以在 Policy初始化时完成实例化。例如，由于在 CLASSPATH中可以找到 FilePermission类，故总有可能实例化一个 java. io. File Permission。

然而，Java技术的动态特性使得当 Policy对象被构造时，实现一个特定 Permission类的实际代码可以未被加载(甚至不用加载)。例如，一个引用Permission类可能在随后加载的JAR文件中。这时， Permission不能在 Java运行环境下定义 Permission类。对每一个这样的类来说，一个 UnresolvedPermission对象被实例化，以当作保存许可权信息的位置保持器。因此，可以利用 UnresolvedPermission类来控制“未解决的”许可权。类似的， UnresolvedPermission Collection类存储着UnresolvedPermission许可权的集合。

一个特定类的未解决的许可权可以在访问控制决定被执行时再次解决，这个访问控制决定要考虑以前未被执行的同类的许可权，但是那些类必须已经是可用的 (它已被加载或正在被加载)。即对于每个 UnresolvedPermissionCollection，一个类的新对象被实例化，这基于 UnresolvedPermission中的信息。这个新对象然后代替了已存在的UnresolvedPermission。这时，如果这个许可权仍未被解决，它被认为是无效的，从某种意义上说这个许可权未被准许且访问请求被拒绝。当然，当相关的许可权类可用时，这个 UnresolvedPermission最终可以被解决 (经过几次尝试之后)。

在Policy初始化时，没有必要实例化所有可加载许可权类。这是因为通常只有一小部分 Policy内容需要在虚拟机上运行。这样的话，甚至在 Permission类可加载的情况下广泛使用 UnresolvedPermission，虽然这样会延迟 Permission对象实际的实例化，直到 Permission对象正确才可以使用。这种做法是合法的，甚至有时利于提高性能和效率。不过，必须密切注意 UnresolvedPermission增加的复杂性，尤其是当一个 Unresolved-Permission可以在不同的时刻分成不同的实现时。

UnresolvedPermission类中的几个方法需要解释：

```
public UnresolvedPermission(String type,String name,String actions,
      java.security.cert.Certificate certs[]);
public boolean implies(Permission p);
public boolean equals(Object obj);
```

构造器用一列证书来证实类的二进制文件的签名。其他的 Permission类并没有这一特性。这是因为对那些并不驻留在 CLASSPATH或其他系统路径的许可权来说，这些许可权更可能被第三方定义，并且作为应用程序或扩展的一部分传递。这时， Permission类的可靠性 (如它们是否服从根类java.security.Permission的意图，该类的实现是否是恶意的)就是可疑的，且在一个应用程序中难以证实。证书可以提供额外的保证，这个保证依赖于由 Permission类上的签名密钥所传达的可信度，以及应用程序如何解释证书。

当一个Permission类不是本地存在，只是在使用时被下载时，证书也是有用的。一方面，希望确保每次相同的类文件被下载。然而，除非本地存储器有一个副本或至少有前类文件的手印，否则这种证实很困难。另一方面，软件经常需要升级，因此希望同名的 Permission类文件以一种一致性的方式改变。但是通过核查类文件来检验这种一致性是很困难的。用于确认

China-pub.com
下载

类文件的数字签名一般不会改变，且相对于实际的类文件来说，更易于进行有效的管理。

到目前为止，我们熟悉的implies()方法对未解决的许可权总是返回假，这是因为从不认为一个UnresolvedPermission隐含另一个许可权。

最后，当比较两个 UnresolvedPermission对象的等同性时，需要核查第二个对象也是UnresovedPermission，且与第一个对象具有相同的类名、许可权名、行为和证书。

3.6.4 java.io.FilePermission

java. io. FilePermission类表示允许访问文件和目录的许可权， FilePermission由路径名和为路径名而设定的一套有效操作所组成，路径名的含义是指确定的文件或目录。此文件或路径被授予指定的操作。它以下列方式被确定，在这里目录名和文件名是字符串，且不包含空格：

`file`	被命名的文件
`directory`	与目录/相同
`directory/file`	被命名的文件
`directory/*`	该目录下的所有文件
`*`	当前目录下的所有文件
`directory/-`	该目录下文件系统中的所有文件
`-`	当前目录下文件系统中的所有文件
`"<<ALL FILES>>"`	文件系统中的所有文件

换句话说，以“ /* ”结尾的路径名表示一个目录和包含在此目录下的所有文件，其中“ / ”是文件分隔符， File.separatorChar。以“ /— ”结束的路径名表示一个目录和包含在此目录下的文件和子目录。包含特殊标识 "《All FILES》"的路径名与任何文件相匹配，注意，路径名包含单个“ * ”，指当前目录下的所有文件。路径名包含单个“ — ”指包含在当前目录下的所有文件和子目录。

包含0列表和逗点分隔符关键字的字符串以被准许操作的方式出现在构造器，下面给出了可能有的关键字，在处理前操作字符串被转换为小写。

- read指读许可权。
- write指写许可权。
- execute指执行许可权，它允许调用 Runtime.exec，并与在Security Manager.checkExec()方法内部所做的安全核查相对应。
- delete指删除许可权，它允许调用File. delete，并与在Security Manager.check Delete内部所做的安全核查相对应。

在java.io.FilePermission类中,有比较重要的方法被调用。如:

```
public FilePermission(String path,String actions);
public boolean implies(Permission p)
public boolean equals(Object obj)
public PermissionCollection newPermissionCOllection()
```

implies()方法核查对象 FilePermission是否含有指定许可权，更为特别的是 ,如果p是FilePermission的一个实例， p的操作是这个对象操作的真子集；且 p的路径名这个对象的路径名隐含，那么这种方法的返回值为真。例如 /tmp/*隐含/tmp/foo,因为tmp/包含tmp目录下的所有文件，包括名为foo的文件。

China-pub.com 下载

当使用equals方法核查两个 FilePermission对象的等同性时，必须核查目标文件是否为FilePermission,且在equals被调用处是否具有相同的路径名和操作，更为复杂一点的方法是返回新建的对象PermissionCollection FilePermission对象必须以某种方式被存储，即允许它们以任何顺序被插入集合，并使Permission Collection的implies()方法以有效的方式被实现。

例如，假设你有两个FilePermission：

```
"/tmp/-","read"
"/tmp/scratch/foo", "write"
```

并且你正在用FilePermission调用implies()方法:

```
"/tmp/scratch/foo", "read,write"
```

在这里，implies方法必须同时考虑tmp/-，read和/tmp/scratch/foo,write许可权。有效的许可权包含/tmp/foo,read,write，且方法implies返回真值，对于FilePermission的implses方法由对象Permission Collection来控制，而该对象由方法newPermissionCollection返回。

注意"《All FILES》"是一个特殊的字符串，表示系统中的全部文件。在 UNIX系统中包含根目录下的全部文件，在MS-DOS系统中包含所有驱动器中的文件。

下面是关于创建文件许可权的有效代码示例：

```
import java.io.FilePermission;
FilePermission p = new FilePermission("myfile","read,write");
FilePermission p = new FilePermission("/home/gong/","read");
FilePermission p = new FilePermission("/tmp/mytmp","read,delete");
FilePermission p = new FilePermission("bin/*","execute");
FilePermission p = new FilePermission("-","read");
FilePermission p = new FilePermission("/-","read,execute");
FilePermission p = new FilePermission("-","read,execute");
FilePermission p = new FilePermission("<<ALL FILES>>", "read");
```

这个类中的implies方法正确地解释了文件系统，例如FilePermission("/-", "read, execate")指FilePermission("/home/gong/public-htm1/index.htm1","read")，而FilePermission("bin/*","execute")指FilePermission("bin/exmacs 19.31","execute")。

注意：大多数字符串以与操作平台相关的格式给出，颇为不幸的是，只有当通用文件描述语言得到普遍应用时，这一点才不是必须的。例如，为表述读取在 MS-Windows系统C盘temp目录下的文件foo, 必须使用:

```
FilePermission p = new FilePermission("C:\\temp\\foo","read");
```

双斜杠线“ \\ ”是表述单个反斜线所必须的，因为串处理是由 tokenizer (java.is.Stream Tokenizer)来完成， tokenizer只允许使用" "作为结束(例如"n表示一个新行)。所以需要两个反斜线表示单个反斜线。 tokenizer在处理完FilePermission目标串，将双斜线转换为单个反斜线后，最终结果是实际路径"C:"temp"foo"。

值得注意的是，“ * ”和“ - ”这样的符号阻碍了那些带有这些符号的特定文件的使用。暂且认为这一小的限制是可以被容忍的。另外需注意的是“ /- ”和"《All FILES》"在UNIX系统中具有相同的作用，都可代表整个文件系统。它们也可代表多重物理文件，这些系统文件被组织成为一个虚拟文件系统。在UNIX系统中文件系统被划分为卷或层，“ /- ”代指当前层，而"《All FILES》"代指所有层，这两种表示方式在其他操作系统中也可能是不同的，比如 MS-Windows和MacOS。

China-pub.com
下载

最后，需要注意的一点是：read操作指定的目标名反指目录如下所示：

```
FilePermission p=new FilePermission("/home/gong/","read");
```

表明仅能列出目录下的文件，而不能读文件。为了读取文件，必须给出完整的文件名，或“*”，“—”，如下所示：

```
FilePermission p = new FilePermission("/home/gong/myfile","read");
FilePermission p = new FilePermission("/home/gong/*","read");
FilePermission p = new FilePermission("/home/gong/-","read");
```

为了举例说明如何使用许可权，这里有一段从类java.io.FileInputStream的结构中摘取的代码段。

```
public FileInputStream(String name)
      throws FileNotFoundException {
   SecurityManager security = System.getSecurityManager();
   if (security != null) {
      security.checkRead(name);
   }
(now open the file)
}
```

在类SecurityManager中对应的checkRead()方法如下所示：

```
public void checkRead(String file) {
   checkPermission(new FilePermission(file, "read"));
}
```

这个例子显示如何创建文件许可权，并用它来调用安全核查，而用 File InputStream构造器直接调用checkPermission()方法，也可以得到同样的结果，尽管如此，为了提供向后兼容，我们还是选择保持checkRead()。关于兼容性的话题，我们将在3.10节SecurityManager类中进行讨论。

3.6.5 java.net.SocketPermission

java.net.SocketPermission类表示获取网络通行界面， SecketPermission包含主机说明和一系列连接到主机的操作。

主机可被定义为:hostname：port-range，其中hostname可由下列一系列方法给定。

hostname	一个主机
IP address	一个主机
localhost	本地机器
" "	等同于localhost
hostname.domain	域内的一个主机
hostname.subdomain.domain	域内的一个主机
*.domain	域内的所有主机
*.subdomain.domain	域内的所有主机
*	所有主机

更为精确地讲，主机可用如下的BNF格式定义：

```
host = (hostname | IPaddress)[:portrange]
portrange = portnumber |-portnumber | portnumber-[portnumber]
```

China-pub.com 下载

主机可用DNS名，IP地址或localhost(本地机器)来描述。如在DNS主机说明中包含有通配符“*”，那么它一定位于最左端，例如:*.sun.com。

portrange是任选项，可用如下方式给定：

N	单个端口
N-	以N开头的所有端口
-N	以N结尾的所有端口
N1-N2	以N1开头，N2结尾的所有端口

其中N，N1和N2是非负的整数，范围从0～65535。

与主机相连接的可能性操作是：

- accept
- connect
- listen
- resolve

listen操作只有在localhost使用时才有意义。注意存在的隐含性， accept和connect操作隐含着resolve操作，即如果能听和接受外部与主机的连接，那么，可以查询得到远端主机名。

你也许会问，为什么即有listen操作又有accept操作，为什么不仅仅只有accept，它将隐含listen操作吗?明确的回答是两种操作都是必须的。 listen操作仅应用在本地主机的端口，而accept操作则在本地和远程主机的端口上都得到应用。

下列是java.net.SocketPermission类中比较有趣的方法：

```
public SocketPermission(String host,String action);
public bollean equals(Object obj);
public bollean implies(Permission p);
```

下列是用不同的方法构造的界面许可权：

```
import java.net.SocketPermission;
SocketPermission p =
    new SocketPermission("java.sun.com","accept");
p = new SocketPermission("204.160.241.90","accept");
p = new SocketPermission("*.com","connect");
p = new SocketPermission("*.sun.com:80","accept");
p = new SocketPermission("*.sun.com:-1024","accept");
p = new SocketPermission("*.sun.com:1024-","connect");
p = new SocketPermission("java.sun.com:8000-9000","connect,accept");
p = new SocketPermission("localhost:1024","accept,connect,listen");
```

假如尝试下列操作:$

```
SocketPermission("java.sun.com:80,8080", "accept");
SocketPermission("java.sun.com,javasun.sun.com", "accept");
```

那么将遇到运行异常IllegalArgumentException，这是因为逗号分隔符在主机名或端口范围处未被接受。

核查两个SocketPermission对象是否相等相当容易，可简单地核查两者是否都是SocketPermission类型，且是否具有相同的主机名和操作，更为复杂一点的方法是核查SocketPermission对象是否隐含有另一个指定许可权。

China-pub.com
下载

首先，implies方法要确保下面两个值都为真，如果有一个为假，返回值为假。

(1) p是SocketPermission类的一个子类。

(2) p的操作是这类对象操作的真子集，并且它的端口范围包含在所属对象的 portrange内。

然后依次核查下列值，当某个确定的状态为真，那么就跳过剩余的核查，并返回一个真值。如果没有一个状态为真，那么返回值为假。

(1) 对象的IP地址是否被初始化，以及p的某一IP地址是否与对象的IP地址相等。

(2) 如果对象的域名是由通配符表达的，那么 p的正规名称以对象的法定主机名结尾，例如：*sun.com包含*.eng.sun.com。

(3) 如果对象是以主机名被初始化，而不是数字地址，那么与主机名对应的 IP地址应等于p中的某一IP地址。

(4) 对象的正规名是否与p的正规名相等。

这里，正确理解相同主机名的含义显得尤为重要。当比较相应的主机名时，有时必须以比较相应的IP地址来做为替代。当初始化一个 SocketPermission对象时，没有必要立刻做 DNS地址查询，完全可以将这一查询推迟到调用 implies方法时。也就是当主机名的真实性和其他一些信息被提供时，再做查询。当然，如果在延时过程中 DNS记录了某种变化，那么延时核查就反馈回一个异常的结果。如果用自定义的机制比较两个 SocketPermission对象，那么在以缺省值做为主机名时一定要谨慎。

为了举例说明在实际中如何使用界面许可权，下面是一段从类 java.io.Socket构造器中摘录的代码:

```
SecurityManager security = System.getSecurityManager();
if (security != null) {
   security.checkConnect(address, port);
}
```

SecurityManager类中相应的checkConnect()方法如下所示：

```
public void checkConnect(String host, int port) {
   if (port == -1) {
      checkPermission(new SocketPermission(host,"resolve"));
   } else {
      checkPermission(new SocketPermission(host+":"+port,
    "connect"));
   }
}
```

3.6.6 java.security.BasicPermission

java.security.BasicPermissren类扩展了Permission类，并提供了非常简单的命名规则，命名规则是在创建许可权类过程中常遇到的问题。 java. security.BasicPermission类可做为其他想沿用同样命名约定许可权类的基类。

BasicPermission类的名称是给定许可权的名称 (例如exitVM, setFactory和queuePrintJob)，命名规则遵循命名约定的分层属性，星号可以跟随一个“ .”,出现在名字的结尾处，或者独自出现，表示通配符匹配。例如： java.* 和*是有效的，而*java和a*b是无效的。

因此BasiePermission一般可做为命名许可权的基类。命名的许可权不仅包含名字，而且

China-pub.com 下载

也包含操作列表的许可权，要么有一个命名许可权，要么没有。 BasicPermission是抽象类，因此，不可能真正地构造它，但必须构造它的某个子类。如果希望的话，子类可以在BasicPermission层次之上运行操作，下面是BasicPermission的子类:

```
java.util.PropertyPermission
java.lang.RuntimePermission
java.awt.AWTPermisson
java.net.NetPermission
java.lang.reflect.ReflectPermission
java.io.SerializablePermission
java.security.SecurityPermission.
```

注意，即使没有使用从Permission中继承下来的操作字符串，也必须为BasicPermission提供一个构造器，因此，下面两个构造器相等：

```
public BasicPermission(String name);
public BasicPermission(String name ,String actions);
```

implies()方法核查指定许可权是否与BasicPermission的实例比较，如果是的话，那么进一步判定它的名字是否隐含在比较许可权之中，这里，名称字符串的比较应将通配符考虑进去，例如：“ a.b.* ”包含“ a.b.c ”.

当核查两个BasicPermission对象是否等同时，只要核查它们的名称字符串是否相等。

3.6.7 java.util.PropertyPermission

java.util.PropertyPermission类表示访问java属性的许可权，这个属性设置在不同的属性文件中.例如：调用use.home就是一个典型的设置用户主目录的操作。

PropertyPermission做为BasicPermission的一个子类，与FilePermission相类似,包含目标和操作，这个类的对象基本上是Java的属性名，如java. home和os. home。

命名规则遵循分层的命名约定，星号可以跟随一个“ . ”,出现在名字的结尾处或独自出现，表示与通配符匹配，例如： java.*和*是有效的，而*.java和a*b是无效的。因此目标能被定义为*(任何属性)，"a.*"(任何具有前缀a的属性)，推而广之，还有"a.b.*"等等。

这个类是BasicPermission的一个子类，在BasicPermission基础上运行操作，操作包含read和write，它们的含义被定义如下：

- 读许可权允许从java.lang.System中调用方法getProperty()，从而获得属性值。
- 写许可权允许方法setProperty()被调用，并设置属性值。

在这个类中没有什么特殊的方法，如下面列表所示：

```
public PropertyPermission(String name,String actions)
public boolean implies(Permission p)
public boolean equals(Object)obj
```

被准许的操作以包含空格和逗号分隔符的形式出现在构造器中，操作字符串在处理前被转换成小写。

下面的代码段显示这个许可权是如何被使用的。 java.System类实现两种静态方法：getProperties()和setProperties()，如下所示:

```
public static String getProperty(String key) {
   if (security != null) {
      security.checkPermission(new
```

China-pub.com
下载


```
                    PropertyPermission(key, "read"));
        }
        return props.getProperty(key);
    }

    public static String setProperty(String key, String value) {
        if (security != null)
            security.checkPermission(new
                    PropertyPermission(key, "write"));
        return (String) props.put(key, value);
    }
```

3.6.8 java.lang.RuntimePermission

java.lang.RuntimePermission是BasiePermission的一个直属子类。在RuntimePermission里目标可用任何字符串表示，并且没有与此目标相关的操作。例如： RuntimePermission("exitVM") 表示一种退出JVM的许可权。

命名的规则遵循分层的命名约定，星号可以跟随一个“ . ”，出现在名字的结尾处或独自出现，表示与通配符匹配，例如： package. *和*是有效的，而*.package和a*b是无效的。

目前，下列目标名被使用：

```
createClassLoader
getClassLoader
setContextClassLoader
createSecurityManager
setSecurityManager
exitVM
setFactory
setIO
modifyThread
modifyThreadGroup
getProtectionDomain
readFileDescriptor
writeFileDescriptor
loadLibrary.{library name}
accessClassInpackage.{package name}
defineClassInpackage.{package name}
accessDeclaredMembers.{class name}
queuePrintJob
stopThread
```

为了理解Permission类是如何被使用的，考虑一种情形 ,即一些代码试图与本地库相链接，本地库不在JVM的监控下，如果一旦链接，将执行安全敏感任务核查，因此链接本地库必须是可控操作，在java.lang.Runtime类中是下列代码段：

```
public void loadLibrary(String libname) {
    ...
    SecurityManager security = System.getSecurityManager();
    if (security != null)
```

China-pub.com 下载

```
        security.checkLink(libname);
    ...
}
```

在SecurityManager类中,方法checkLink()以下列方式实现:

```
public void checkLink(String lib) {
    checkPermission(new RuntimePermission("loadLibrary."+lib));
}
```

3.6.9 java.awt.AWTPermission

java.awt.AWTPermission类与RuntimePermission非常相似。AWTPermission包含名字但不包含操作列表，下面是这个类的一些对象：

showWindowWithoutWarningBanner
accessClipboard
accessEventQueue
listenToAllAWTEvents
readDisplayPixels

命名规则沿用分层的属性命名规则。星号可用来表示所有的 AWT许可权。

在java.awt.window类中，创建一个顶层的窗体需要执行下列代码段：

```
...
SecurityManager sm = System.getSecurityManager();
if (sm != null) {
    if (!sm.checkTopLevelWindow(this)) {
        ...
    }
    ...
}
```

这个在SecurityManager类中的核查方法可被翻译成下列代码段：

```
public boolean checkTopLevelWindow(Object window) {
    try {
        checkPermission(topLevelWindowPermission);
        return true;
    } catch (SecurityException se) {
        // Fall through to return false.
    }
    return false;
}
```

注意：与其他核查方法相同，这种核查方法返回的是布尔值而不是空值或安全异常，考虑到向后兼容的问题，我们不改变这个界面。

3.6.10 java.net.NetPermission

java.net.NetPermission是BasicPermission的一个子类，只包含对象但没有操作，命名规则沿用分层的属性命名规则，星号可以跟随一个“ · ”，出现在名字的结尾处或独自出现，表示通配符匹配，例如， foo.*和*是有效的，而*.foo和a*b是无效的，此类的一些对象如下所示：

setDefaultAuthenticator

China-pub.com
下载

specifyStreamHandler

requestPasswordAuthentication

在java.net.Authenticator类中，requestPasswordAuthentication()方法要向已在系统注册的登录者询问口令，显然，口令必须有安全保障 .因此，运行下面的核查:

```
SecurityManager sm = System.getSecurityManager();
if (sm != null) sm.checkPermission(new
   NetPermission("requestPasswordAuthentication");
```

3.6.11 java.lang.reflect ReflectPermission

java.lang.reflect.ReflectPermission类是BasicPermission的另一子类，用于反射操作。ReflectPermission是一个命名许可权(类似于RuntimePermission)没有操作列表，目前仅有命名为suppressAccessChecks的许可权，它允许在使用点通过反射对象，压缩已经过核查的标准Java语言数据，自然地，当有人试图获取类的公共型，缺省 (包)访问，保护型和私有型成员变量时，就要进行数据核查。

在java.lang.reflect.AccessibleObject类中一个比较方便的方法是set Accessible()，它为一系列对象设置获取标志，并执行单一的安全核查，这个方法执行下列核查：

```
SecurityManager sm = System.getSecurityManager();
if (sm != null) sm.checkPermission(new
            ReflectPermission("suppressAccessChecks"));
```

3.6.12 java.io .SerializablePermission

java.io.SerializablePermission类与ReflectPermission相似，只含有下列目标而没有操作:

enableSubclassImplementation

enableSubstitution

例如：在java.io.ObjectOutputStream类中，下列代码段核查是否一个子类能完全再次运行ObjectOutputStream。

```
SecurityManager sm = System.getSecurityManager();
if (sm != null) sm.checkPermission(new
   SerializablePermission("enableSubclassImplementation"));
enableSubclassImplementation = true;
```

3.6.13 java.security.SecurityPermission

SecurityPermission类控制与安全相关的对象的访问，如 Security ,Policy, Provider ,Signer与Identity，它包含下列对象，但没有操作:

getPolicy

setPolicy

getProperty.{key}

setProperty.{key}

insertProvider.{provider name}

removeProvider.{provider name}

setSystemScope

setIdentityPublicKey

China-pub.com 下载

```
setIdentityInfo
addIdentityCertificate
removeIdentityCertificate
printIdentity
clearProviderProperties.{provider name}
putProviderProperty.{provider name}
removeProviderproperty.{provider name}
getSignerPrivateKey
setSignerKeyPair
```

java.security.Policy类包含一种静态方法,用来设置系统缺省的Policy对象，因为Policy对象决定实施何种安全。所以没有合适的许可权，对象不能被改变。

```
public static Policy setPolicy() {
   SecurityManager sm = System.getSecurityManager();
   f (sm != null)
      sm.checkPermission(new SecurityPermission("setPolicy"));
   ...
}
```

注意:类java.security.Identity和java.security.IdentityScope 在JDK1.2中已被禁用。

3.6.14 java.security.AllPermission

新java.security.AllPermission类用来描述所有的许可权，介绍它的目的是为了简化系统管理员的工作，管理员也许需要完成多种任务，这些任务需要各种许可权，反复在所有的许可权中获取安全策略是很不方便的。

因此，AllPermissien并不关心真正的目标和操作，它的结构忽略所有已通过的参数。依照定义，AllPermission隐含所有的许可权，而且，任意两个 AllPermission对象总是被认为相等的。因此AllPermission类特别运行下列两种方法。

```
public boolean implies(Permission p) {
   return true;
}

public boolean equals(Object obj) {
   return (obj instanceof AllPermission);
}
```

注意：AllPermission也隐含尚未定义的新许可权，显而易见，承认这些许可权必须非常谨慎。

3.6.15 许可权隐含中的隐含

回想经常被用来相互比较的许可权，为使这种比较变得容易，每一种许可权类必须定义一个implies()方法，以表述这一特别的类与其他许可权类的关系。例如:java .io. FilePermission ("/tmp/*","read")隐含有java .io.FilePemission ("/tmp/a.txt" "read"),但是它不隐含任何java.net.NetPermission类。然而这里却有另一种更深层次的隐含，对一些读者来讲，也许不会立刻看出来。

China-pub.com
下载

假设，applet被赋予有向整个文件系统的写操作的许可权。并允许 applet替代系统数据库，包括替代JVM运行环境。这也就意味着 applet已拥有所有许可权。或者也可以假设， applet被授予运行许可权来创建类加载器，因为类加载器可以运行敏感操作，所以， applet就被授予了更多的许可权。

其他一些蕴含着潜在危险的许可权包括：

- All Permission。
- 允许设置系统属性的许可权。
- 定义程序包和加载本地代码的许可权 (因此Java安全结构并不具有阻止恶意行为的能力)。

3.7 分配许可权

以前章节所涉及的内容是基本安全策略，代码源和 Permission类层次，这一节将讨论这些许可权是如何真正被允许运行代码的。

当加载一个从特殊的 CodeSource中产生的新类时，安全机制首先询问安全策略对象有哪些许可权被允许。这一点通过调用 Policy对象中的getPermission()方法来完成：

```
public  abstract Permission getPermission(CodeSource codesource);
```

换句话说，在 Java运行中，一般来说类定义之前 ,许可权已经被赋予。但也有两种例外，第一种情况是：许可权类和对象的准许一直被延迟到安全核查发生时 ,这一点被认为是合法的，这种优化使不调用安全核查的 Java程序运行得更快。甚至对于那些调用安全核查的 Java程序，此优化也会使它启动得更快。注意 :在第一个安全核查被调用和策略类被初始化的两个时刻之间，如果策略的内容发生了变化。这种优化技术将导致 Policy对象有更多的更新内容。保持策略类的内容不断被更新当然是一件有利的事， 3.3节讨论了用refresh方法使策略类被更新。

第二种情况是: 在类被定义之后，类中已被赋予的许可权有可能发生变化或被取消。例如：由于安全事故而使程序被注销。虽然在 JDK1.2中，一旦许可权被定义 ,缺省的运行不会改变它们的内容。但只要这种运行在合适的许可权控制下，内容发生变更被视为合法。

值得强调的是 ,类中被赋予的许可权，对于类而言是静态代码，对于类中的对象来说却不是。这样做可减少复杂性，增加可管理性。对象是一个运行整体，所以在静态时是不存在的，但是安全策略在静态时必须存在 ,且与任何特定的Java运行环境无关。所以它不可能引用对象，也就是说，对在这里讨论的安全策略，不管类中的对象是何种类型，安全策略都会被执行。除此之外，类的数目相对于对象而言要少得多。即使你想支持依赖于运行环境的安全策略，正确的做法不是给对象赋予许可权，而是在运行安全策略时考虑真正的运行环境。我们将3.10节中讨论这个话题。

最后，回顾安全策略，本质上可用一系列实体 (代码源、许可权)来表述，因此，已命名代码中的代码也就被给予了已命名的许可权。自然，对于一段已给定的代码，在策略上它的代码许可权与多种实体相匹配，在这种情形下，代码也就被赋予了更多与实体相匹配的许可权。换而言之，许可权是可以被累加的(additive)。例如，如果代码A得到许可权X，代码B得到许可权Y，那么包含代码A、B的代码就得到了许可权X、Y。同样，如果URL http://java.sun.com被赋予许可权X，URLhttp://java.sun.com/people被赋予许可权Y，那么自http://java.sun.com/people/gong下载的applet将得到许可权X和Y。

关于匹配的细节，可参考3.5节CodeSource中的implies()方法，注意:这里URL的匹配是纯语法

China-pub.com 下载

的，并没有解决变化问题，例如，某个策略给出一个指定的URL ftp：//ftp.sun.com实体，这个实体只有在运行过程中，直接从FTP获得Java代码才是有用的。如果网络服务器将URL变换为不同的另一个实体，那么这个实体有可能失效。为了为本地文件系统指定URL，可以用文件URL，例如在Solaris系统中,从/home/gong/temp目录中指定文件，可以写成file:/home/gong/temp/*。在MS-Windows系统中的C盘temp目录中指定文件，可写成file:/c:/temp/*，有一点要注意，书写时用反斜杠，而不管它们在何种操作平台运行。也可以用绝对路径来描述，例如：/home/gong/bin/MyWonderfulJava。

正向许可权与负向许可权

意识到当前的Permission类层次只有正向许可权这一点非常重要，也就是说，在安全策略中如果一个许可权被表述，那么这个许可权就被赋予。负向许可权是通过不表述来实现的，而不是用明确的“负”来表达。现在的版本中没有负向许可权，并不意味着在将来这种方式不被采用。

然而，限制只针对正向许可权，对于简化运行和使运行处于良好的状态有明显的益处。这也是因为在两个正向许可权之间不存在冲突。从某种意义上说，当一个许可权被另一个否定时，获取数据不存在危险性。自然地，当检测到安全策略决定赋予代码哪一种许可权时，不必核查冲突。3.10节将讨论如何运行数据控制核查，以及检测一系列许可权中是否隐含有特定许可权的必要性。在一系列许可权中，一旦没有发现负向许可权，就可以确切地认为在这一系列许可权中隐含这个设置。JDK运行得到的益处,对于安全策略管理来说也是有益的。

从另一方面来讲，缺少负向许可权，不允许你随便指定策略。例如，除这个文件以外，允许访问所有的文件系统。然而，缺少方便性并不等于功能的减少，因为负向许可权也可由正向许可权补充表达。这就好像强大的策略处理工具中的附加语法，你可以将一个带有负向许可权含义的策略转换为只用正向策略表达。

3.8 Protection Domain

当运行许可权分配算法时，可以遵循前面的方法，将所有被赋予类的许可权压缩，这些许可权由各种Permission对象表述，并通过一个基类java.Lang.Class中的界面，与类中的许可权设置相联。然而直接与类中的许可权设置链接会产生一个呆板的API，使其不易被扩展。例如，假设你想运行一个访问控制核查，这个核查不仅基于类中被赋予类的许可权，而且也基于主要的运行代码名。为做到这一点，不得不用附加界面扩展Class类，这样就会使基类变得杂乱。

为使扩展变得容易，JDK1.2允许把许可权赋予保护域，类和对象属于保护域，但不是从被赋予的许可权中继承来的。根据保护域的经典定义[70]，域受一系列对象的限制，这些对象可通过计算机系统中已被准许的主要实体直接获得，因为，从某方面来说,在JDK1.0中,Java沙盒是一个带有固定边界的保护域，在JDK1.2中，每一个类隶属于一个域。Java运行维持类和对象到保护域及到它们许可权之前的映射关系。在类被使用之前，从类到它的域的映射仅设置一次，且在Class对象执行的整个时间范围内不变。

类java.security.ProtectionDomain的定义相当简单,如下所示:

```
public ProtectionDomain(CodeSource codesource,PermissionCollection permissions);
public boolean implies(Permission permission);
```

implies()方法核查ProtectionDomain是否含有在Permission对象中表达的许可权。

China-pub.com
下载

下面从java.lang.Class中摘取的代码段就是一个保护域如何被使用的例子：

```
public java.security.ProtectionDomain getProtectionDomain() {
   SecurityManager sm = System.getSecurityManager();
   if (sm != null) {
      getPDperm = new RuntimePermission("getProtectionDomain");
      sm.checkPermission(getPDperm);
   }
   return protectiondomain;
}
```

注意:因为ProtectionDomain对象也许包含敏感信息，所以对它的访问是一种带有运行许可权的安全核查。

许多更加细小的问题值得讨论，首先，在 JDK1.2中，当一个新类被加载到运行时，保护域按要求被创建。在JDK1.2的缺省运行中，属于相同域的类由相同的加载器加载。这一运行细节是缺省的但不是必须的。属于同一域的类被授予同样的许可权，但反向理解则是明显不对的。因为具有相同许可权的类可来自不同的代码源，因此属于不同的域。

其次大多数保护域是在 Java运行期间创建的。但有一种域是特殊的，即系统域 (system domain)。系统域是一个由所有代码组成的域，且被认为是系统内核的一部分。由于历史的原因，系统代码源由初始类加载器加载，这个加载器完全由 C语言编写。如果系统类由一指定的空类加载器加载,也是有效的，JDK1.2保持了向后的兼容性，类加载的细节将在 3.9节中讨论。从现在开始，必须记住一点:系统域中的代码被自动赋予所有许可权。所有可保护的外部资源,例如文件系统,网络功能，屏幕和键盘可以仅通过系统代码直接获得。访问请求可由少量代码调节完成。注意 :尽管系统类有一个指定的空类加载器，但它们的保护域是一被准许AllPermission的非空对象。

而且类和它的许可权通过保护域的间接性 ,对于虚拟机用户执行优化操作来说是有益的,例如，在Java运行期间希望改变一些代码的许可权，可以直接修改某内容，而不必参考与类相关的ProtectionDomain对象，为了保持参考 ProtectionDomain对象的稳定性，可以判断两类是否属于同一个域，然而执行优化操作。

最后，注意在Java运行期间,保护域会使保护单元成组或隔离更显得方便，例如，不同的域也许会禁止相互作用，这一点可通过不同的加载器加载属于不同域的类来完成。通过这种方式，任何许可权之间的相互作用必须通过系统代码或相应域的允许才能实现，这是因为,在JVM中，一个类是由其本身和类加载器来区分的，所以类加载器为类定义了不同的名字。在保护域内部,通常以拒绝从不同域加载代码的方式,实现类的隔离和代码保护。

下面讨论的是可访问的话题，也就是说,对一个对象而言，什么是可见的，调用什么方法,以及它与其他对象的关系。为定义一个新的安全机制，我们检测已存在的代码实现访问的特征，使其能被整个对象发现。如果不考虑执行特殊的安全策略，特别是在服务器程序中 ,我们注意到访问必需保持灵活性。于是我们决定维持已存在规则的访问性，使得访问性与安全性保持独立关系，也就是说，由应用程序员决定是否以及如何使对象和方法彼此被隐藏，从某种程度上说，这种Java安全机制比经典的管理系统要灵活。

从技术上讲，我们本该使域之间保持严格的隔离性，但也有必要为域间的相互沟通创建一系列界面，这就不得不修改已存在的应用程序，从而实现新的界面。为加强完全隔离，我

China-pub.com 下载

们也应重新设计一些共享的系统类及它们的静态区[33]。因此，保留访问性并从安全机制上实现隔离是最好的可行方案。

3.9 安全地加载类

动态的加载类是JVM的一个重要特征。它为Java平台提供了在运行期间安装软件组件的能力[45]。这个特征还有许多特别的特性。其中之一是惰性加载 (lazy loading)，即以命令方式加载类。另一种是动态地加载类 (dynamic class loading)。即意味着,由一个附加的时间链接核查来保持JVM的安全类型。这种加载方式只运行一次，以避免多余的运行核查。

除此之外，因为类加载的是第一类对象，程序员可定义类加载器。例如，可为远端指定某一种类被加载。也可为类加载器设置合适的安全属性。

最后，类加载器通常可为各种软件组件提供单独的名字。例如：浏览者可用单独的类加载器从不同的网页加载applet，因此保持了这些applet类的隔离度。实际上这些applet也有可能包含同名的类，这些类由JVM区分不同的类型。在2.3节中简洁地谈及了语言类型的安全性，这一点通过各种技术得到了增强，如代码核实，类加载，运行核查，这一节着重讨论查找文件的API，确定使用合适的类加载器，并为加载的类设置合适的安全属性。

3.9.1 类加载器的层次

当类加载器加载Java软件组件时，最小的组件单元是类。类的定义依赖于机器，其二进制表述叫做类文件格式(class file format)。类的单独表述叫做类文件 (class file)，尽管它并不存于真正的文件中。

一个类文件包含字节代码，引用区的符号和其他类的名字。一个 C类的示例如下所示:

```
class C {
   void f() {
      D d = new D();
      ...
   }
}
```

类文件表明C类引用D类的符号，这样,参考符在链接时就做为真正的类，为做到这一点，JVM必须加载D类的类文件，且创建该类的类型。

类加载器中加载的C类样本L称之为定义类加载器(defining class loader)。类的真正属性由其本身和定义类加载器共用决定，即由〈 C，L〉决定，换句话说，如果两个类的类型相等且它们的定义类加载器一致，那么在Java运行里两个类的类型完全相等。进一步说，在 JVM中,存在多个类加载器对象的样本，关键的问题在于 ,加载一个类时,如何决定由哪一个类加载器作为定义类加载器。

JDK1.2介绍了多种类加载器的不同属性，另一个问题是当加载一个类时，应该使用哪一个类加载器，下面将介绍类加载器的层次及在层次中 ,类间的相同点和不同点。

类加载器层次中的相互比较

回顾每一由相应的类加载器加载的类。由于类加载器本身也是一个类 ,必须由另一类加载器加载，这种就产生了一个类似“ 蛋与鸡 ”谁先出现的问题，即第一个类加载器从何而来。这个问题的答案是:初始类加载器(primordial class loader)，位于类加载过程中的第一层，这个

China-pub.com
下载

类加载器自然是由C语言编写，且不是从Java的内部来表述自己，因此只能从Java语言中间接看到或获取，初始类加载器常以平台独立的方式从本地文件系统中加载类。

一些类，例如定义在java.*程序包中的类,对于修正JVM和Java运行系统功能是必要的。常被引用作为系统类 (system class)。由于历史原因，所有系统类都由一个定义类加载器加载，它是一个空对象，这个空类加载器有时被称之为系统类加载器 (system class loader)，是初始类加载器的唯一存在形式，实际上，也可以简单地将空类加载器看做是初始类加载器。在JDK1.2开发的后期，有一个术语上的变换 ,所有位于CLASSPATH上的类现在称为系统类。这些类要么被初始类加载器加载，要么被 java.lang.ClassLoader中的一非空子类加载。一个新的名词:辅助类加载器,是指在虚拟机上将必要的类加载到辅助类的加载器上。在 JDK1.2的缺省运行中，辅助类加载器是初始类加载器。

Java运行环境中的所有类，彼此之间类加载关系可由图 3-3的加载树来反映，每一个不是加载器的类都是树状结构的分支节点，每一个类的父节点是它的定义类加载器，空类加载器做为根类，这样一个树状结构不能形成循环，也就是说每一个类加载器不能加载它的上一级类加载器。

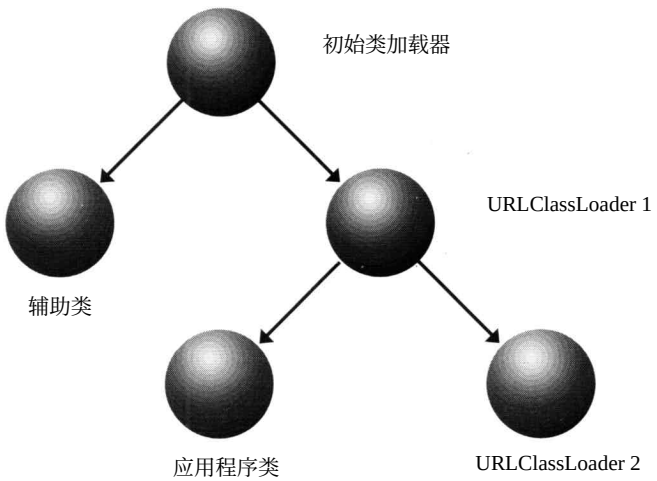


图3-3 类加载器树

类加载器是以 Java代码定义的普通对象。在类加载器的最底层是被称为 java. lang. ClassLoader的抽象类。它最初被定义在 JDK1.0中,现在已有了扩展(参见3.9.2节)，在JDK1.2中介绍的类java.security.SecureClassLoader是一个子类,且是抽象类ClassLoader的一个具体实现。JDK1.2中介绍的类 java.net.URLClassLoader，是扩展机制中的一个关键组件，并且是类SecureClassLoader的一个子类。

在JDK中创建的应用程序appletviewer,为加载applet使用了特别类sun.applet. AppletClassLoader。在JDK1.0中,AppletClassLoader是一个子类,且是类ClassLoader的实例化。在JDK1.0中，它是类URLClassLoader的子类，注意:在已存在的类和它的子类之间引入一个新类是为了解决数据向后兼容的问题。

当创建一个自定义的子类时，可以依据类加载器的特殊要求，用这一节中提到的任何类加载器生成(见图3-4)。注意：因为AppletClassLoader类是一个定义在sun.*程序包中的特有类，

China-pub.com 下载

不支持更改，所以不能由它生成子类。

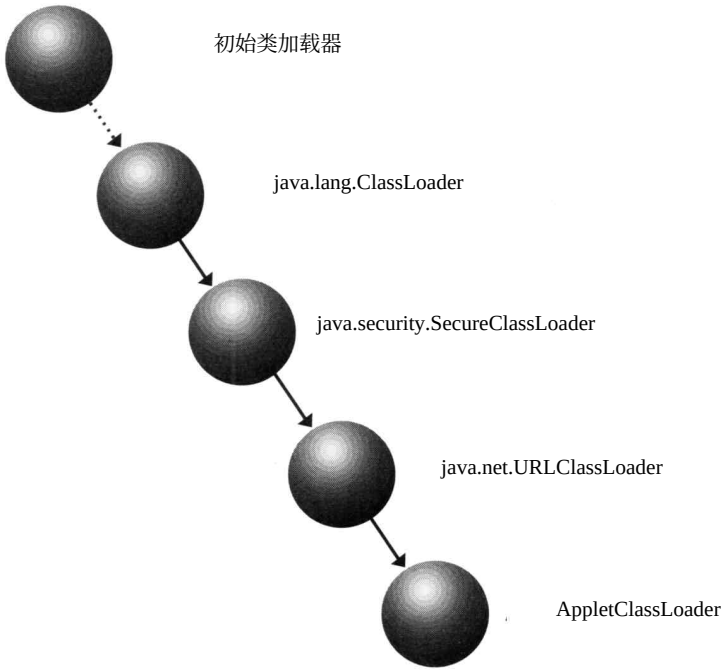


图3-4 从类加载器生成子类

3.9.2 java.lang.ClassLoader和授权

要理解抽象类java.lang.ClassLoader的功能，首先必须理解ClassLoader对象中存在的特殊关系：授权。

当一个类加载器申请加载类，要么用它自身加载，要么用另一个类加载器加载，也就是说，第一个类加载器可以授权第二个类加载器。这种授权关系与两个类加载器均无关，从这一点来说，它是虚拟的。授权关系是在 ClassLoader对象创建时形成的,并呈现出父-子关系的形式。而且初始(或辅助)类加载器是所有类加载器的授权祖先类。

ClassLoader API的第一组涉及以下构造器：

```
protected ClassLoader(ClassLoader parent)
protected ClassLoader()
```

第一个构造器创建了类加载器，并且是一个具有父授权资格的特殊类加载器。第二个构造器使用了缺省的授权，这是因为类加载器要运行诸如定义类的这样的敏感操作。在构造器中，必须严格控制由谁来创建类加载器，因为所有的类加载器都是 Class Loader的子类。子类中的结构总是调用方法super ()，设置的安全核查总是被调用。

缺省的授权父类由方法 getSystemClassLoader()决定，它是一个典型的用来驱动程序的类加载器，可用方法getParent()得到父类加载器。

```
public static Classloader getSystemClassLoader()
public ClassLoader getParent()
```

必须严格控制谁可以成功调用这些方法，这一点非常重要，因为从任何对象的内部，都

China-pub.com
下载

可以调用this.getClass.getClassLoader()方法，以得到它自己定义的类加载器。由于利用了这个类加载器，你也许会试图使用它的授权父类，并调用与其相关的方法，这种失控的使用显然不是所期望的。对于安全管理员来说,只有调用者的类加载器与当前类加载器相同或是其授权祖先类，或者调用者使用RuntimePermiesion("getClassLoader")许可权，才会成功，否则会出现安全异常。注意：允许祖先类访问的方法是合理的，因为子类在创建时必须指定它的授权父类，当然，对选用哪一授权父类要非常小心。

类似的原因，调用Class.getClassLoader()方法时，同样的安全核查被设置,因为不能引用Class 对象的操作来获取对应的ClassLoader对象，这个安全核查对JDK1.2来说是新的。

下面一组方法用来处理实际的类加载：

```
public Class loadClass(String name)
protected synchronized Class loadClass(String name,boolen resolve)
protected native final Class findLoadedClass(String name)
protected final Class findSystemClass(String name)
protected Class findClass(String name)
protected final void resolveClass(Class c)
```

前两个方法以类名做为参数，并返回一个Class对象，缺省运行将以图3-5所示的顺序来检索类。如果在某一步类被加载，那么这个方法返回一个类。

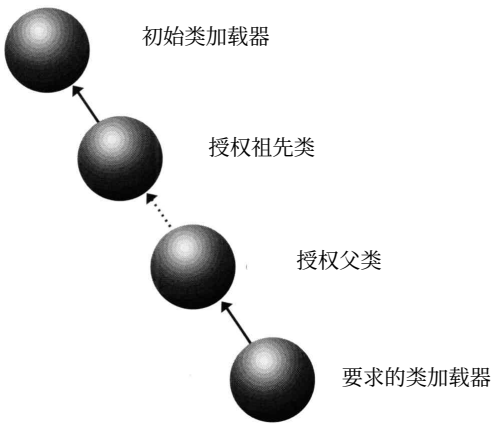


图3-5 ClassLoader检索类

(1) 调用findLoaderClass()，核查类是否已经被加载。

(2)如果当前类加载器有一个指定的授权父类，调用授权父类的方法 loadClass()以加载类，否则，调用方法findSystemClass()，核查在系统类中能否发现类。

(3) 调用方法findClass查找类。

其中，方法findLoaderClass()查询类加载器的本地缓存，核查是否有与目标类相匹配的加载类。然而，由于安全类型的原因，相同的类只能被相同的加载器加载一次。如果类的目标类不在已加载类之中，当前的类加载器将此任务授权给父类加载器。这种方法可以向上递推，以确保合适的加载器被使用，例如为定位一个系统类，授权操作一直持续到系统类加载器被选定，如果目标类是一个系统类，方法 findSystemClass()用空系统加载器加载类。

方法findClass()为查找类提供了一种自定义的机制。自定义加载器可以指定一个类如何被查找，例如，一个applet类加载器可以返回到applet的主机中定位文件系统，并跨网络加载。

China-pub.com 下载

如果用以前的步骤发现了一个类，则 resolved标签为真。loadClass()方法将调用与产生Class对象相关的resolveClass()方法。

另一个需阐明的问题是：当类名称给定并试图加载类时，你将启用哪一个类加载器？下面是在JDK1.2中实施的规则：

- 当应用程序的第一个类被加载时，使用一个新的 URLClassLoader实例。
- 当applet的第一个类被加载时,使用一个新的AppletClassLoader。
- 如果要加载的类利用已存在的类，那么相应的类加载器将加载这个已存在的类。

使用URLClassLoader与AppletClassLoader的规则有可能出现异常，因为它们严重依赖于特定的系统环境。例如，一个Web浏览器有可能从同一Web页中选择已存在的AppletClassLoader来加载applet类。

下面一组方法是将一组二进制文件转换为 Class类中的实例:

```
protected final Class defineClass(String name ,byte[ ] b,
      int off,    int len,ProtectionDomain protectionDomain)
protected final Class defineClass(String name ,byte[ ] b,
      int off,    int len)
protected final void setSigners(Class c,Object[ ] signers)
```

在3.8节中谈到,访问控制许可权被授予保护域，每个类属于且仅属于一个保护域。当定义类时,类加载器查询安全策略,并且引用类所属的Protection Domain对象,然后以Protection Domain对象做为参数,调用方法defineClass()。注意第二种方法defineClass()没有详细地提到ProtectionDomain,这是因为这种方法存在于 JDK1.2以前的版本中，在这种情形下，使用一缺省的对象 Protection Domain。当用(null, null)的Code Source 调用方法Policy.getPolicy().getPermission()时，此域包含一系列授权的许可权。

类定义的结果是类被标识并属于各自的保护域。可通过调用方法Class. getProtectionDomain()来质疑保护域上的类。因为ProtectionDomain对象是敏感的，所以必须谨慎地授权谁可利用它。因此对于安全管理员来说可通过调用getProtectionDomain()来检验调用者是否有RuntimePermission许可权，如果不这样做，将出现安全异常。

当一个类文件由一个或更多的数字签名所标识，从类文件中创建的运行类由它的签名者标定。这一点可通过调用方法 setSigrers()来实现。可调用方法 Class.getSingers()来质疑类的签名者。在这一方法中，没有安全核查，因为对于由谁来标识类来说 ,通常没有安全问题。

因为ClassLoader类中其余的方法大部分与寻找资源和封装有关，下面只给出它们的形式而不做进一步地解释。

```
protected String findLibrary(String libname)
public URL getResource(String name)
public final Enumeration getResources(String name)
public Enumeration findResources(String name)
public URL findResource(String name)
public static URL getSystemResource(String name)
public static Enumeration getSystemResources(String name)
public InputStream getResourceAsStream(String name)
public static InputStream getSyStemResourceAsStream(String name)
protected Package definePackage(...)
```

China-pub.com
下载


```
protected Package getPackage(String name)
protected Package[ ] getPackages()
```

3.9.3 java.security.SecureClassLoader

java.security.SecureClassLoader类用与代码源相关的定义类提供的额外支持扩展了ClassLoader。在JDK1.2的开发过程中，这个类最初被设计成为一个包含复杂方法调用的类，逐渐地它的功能被移植到基类 ClassLoader或者新创建的类URLClassLoader中，于是这个类目前有两种有趣的方法：

```
protected PermissionCollection getPermissions(CodeSource codesource)
protected final Class defineClass(String name,byte[ ] b,
                    int off,     int len,Codesource cs)
```

第一种方法为给定的Code Source对象返回一个许可权。这个方法的缺省运行是调用方法Policy.getPermissions(),以得到由指定代码源策略所授权的许可权，此方法由方法 defineClass调用。在为定义类创建Protection Domain时，方法define Class将Code Source对象做为参数。类加载器可以忽略这个方法，例如，在 AppletClassLoader中，自动授权一个许可权，允许 applet从applet下载处连接回到主机，虽然安全策略没有指定这个许可权。下一节将讨论 URL ClassLoader如何定义这种方法。

第二种方法从一个特定的代码源中定义类。在某种程度上，这个方法用 ProtectionDomain作为参数，复制了 ClassLoader中方法defineClass的某些功能，这样做的方便点在于有时不用为保护域的问题担心。例如，该方法的调用者也许不能确定使用哪一种保护域 ,但仍想定义类，在这种情况下，对于用来决定赋予许可权的类而言， codesource是唯一可利用的信息。

3.9.4 java.net.URLClassLoader

java.net.URLClassLoader类是SecureClassLoader类的扩展，可以用于加载关于 URL的检索路径中的JAR文件和目录的类和资源。它有两种结构：

```
public URLClassLoader(URL[] urls, ClassLoader parent)
public URLClassLoader(URL[] urls)
```

第一种方法为指定的 URL构造一个新的URLClassLoader。URL将以类和资源指定的顺序被检索，而且首先用指定的父类加载器来检索授权它的父类。

第二种方法使用缺省的授权父类加载器为指定的 URL构造一个新的URLClassLoader。

```
public Class loadMainClass()
```

这种方法用于加载应用程序中的主类， URL类路径可以从包含 Main-Class类名属性的第一个JAR文件中去检索，这个主类是应用程序为了引用主要方法而必须加载的。如果找到主类，则作为结果返回，反之返回为空。

java.net.URLClassLoader类也可以替代findClass(String name)方法和一些相关资源的加载方法，用以寻找和加载在 URL检索路径中指定名称的类和资源。在未找到类前，关于 JAR文件的URL须先被加载并一直处于打开状态。

更有趣的是，从安全观点来看，这种类可以替代下列方法：

```
protected PermissionCollection getPermissions(CodeSource cs)
```

这种返回给定的CodeSource对象的许可权的方法，通过调用 Super.getPermissions()可以得到安

China-pub.com 下载

全策略赋予的许可权。按照下列规则在代码源的 URL 基础上又增加了一些附加许可权，规则如下：

- 如果URL指定的协议是file且路径是文件，可以对文件赋予读许可权。
- 如果URL指定的协议是file且路径是个目录，可以对目录及其子目录中所有文件赋予读许可权。
- 如果URL指定的协议不是file，那么可以得到与URL的网络主机连接的许可权。

换言之，通过缺省方式，在创建 URLClassLoader时，可以由URLClassLoader加载类对指定的URL赋予访问许可权。

URLClassLoader的另一个特征是它具有两个创建新的URLClassLoader实例的静态方法：

```
public static URLClassLoader newInstance(URL[] urls, ClassLoader parent)
public static URLClassLoader newInstance(URL[] urls)
```

正如本章前面所讲的，出于安全考虑，它对能创建 URLClassLoader实例的实体作了严格限制。不过，它为应用程序和 applet规定URL位置和加载类和资源提供了一种方便的机制。这些静态方法虽然不提供其他类型的类加载器，但它允许任一程序创建 URLClassLoader类的实例。鉴于可用的公用方法和授权机制，它是可以接受的。注意，一个应用程序和 applet仍不能调用URLClassLoader及其超类中受保护的方法。

作为特例，在Web浏览器和appletviewer中，applet类加载器可为applet加载所需的类和资源。在JDK1.2中，这种类作为URLClassLoader的直接子类在私用sun.*包中有定义。

3.9.5 类的路径

前面描述的类加载器类为定位和加载类和资源提供了编程方法。为了简化在 Java使能的系统上安装软件组件的任务，可利用公用或用户专用存储区域存放一些组件，以便在 Java运行系统中自动被找寻到。

大家已熟知，在JDK1.0和JDk1.1中，内置全系统的检索路径称为CLASSPATH。它根据特定的平台加以设置。例如，在 UNIX系统中，可以通过外壳环境变量 CLASSPATH来设置CLASSPATH。实际上，在本地文件系统中，包含类的所有类或 JAR文件必须保存在这条路径中以便能被找到。这条路径也是保存所有系统类的地方。因此，本地文件系统中所有的类都被作为系统类来对待，并被赋予访问所有资源的许可权。换言之，真正属于系统代码的本地类与其他一些本地安装的应用程序的本地类是无法区分的。

这显然是不完美的，假想一个本地安装的应用程序没有被赋予访问整个系统的许可权，可能会发生什么后果。例如，在邮件中收到最新的演示版程序。又如，对于一个重要文件，你也许想用只读方式显示它以保证文件内容不会因为应用程序中软件的 bug而发生改变或丢失。

在JDK1.2的安全结构中，通过获得它们专用的细致的许可权可以保证用相同的方法对待下载的applet类和本地驻留类。这样的话，真正系统类必须要区别于其他所有类。 JDK1.2的方法是分离这种类的路径，一个为系统类服务，另一个为剩下的类服务。

在JDK1.2的测试版中路径分离的最早设计需要一个检索路径，即除已存在的 CLASSPATH之外要有一个应用程序类的路径。如同 JDK1.1一样，CLASSPATH中的所有类都像系统类一样来处理。应用程序类的路径内的所有类都是非系统类，不过，它们都作为安全策略赋予许可权的SecureClassLoader实例被加载。当激活应用程序时，应用程序类的路径既可用设置

China-pub.com
下载

java.app.class.path的Java属性加以指定，也可使用一个命令行选项来指定。命令行选项和其他使用问题在第4章讨论。

这种设计方法有向后兼容的优点。一个当前的应用程序可以快速从 CLASSPATH的设置中被移到新的应用程序类的路径中，并且不会影响其他已安装好的软件部件。在移动前，应用程序像在JDK1.0和JDK1.1中那样运行，一旦被移动，应用程序就变成了细致的访问控制的对象。然而这种移动任务不应由用户来完成。同时由于向后兼容性，可能会导致用户什么都不能干，这样他们就得不到期望中的一个更好的安全体系和一个非常强大的扩展机制。

正是由于这些考虑，在JDK1.2的最终设计中，CLASSPATH被作为应用程序类的路径。这样就不必移动使用的应用程序。当 JDK1.2安装成功后，路径中的类就被 URLClassLoader的实例加载。通过配置安全策略可以对应用程序路径内的不同的类赋予一些不同的许可权。

作为系统类的路径，已经创建出一个新的路径 Xbootclasspath。用户或开发者将会很少或不必在这个路径中安装类，除了包括 JDK中的那些类。注意，这个设计也许不能为当前的应用程序提供完整的向后兼容，即使这种应用程序的数量被认为是非常少的。这是因为直到JDK1.1版包括JDK1.1，所有CLASSPATH的类都像系统类一样被处理并且用空类加载器加载。而在JDK1.2中用URLClassLoader的实例加载它们。因此，核查空类加载器的应用程序需要升级，以便反映URLClassLoader的出现。

你也许会问为什么还保留一个独立系统类的路径。这是因为如果系统类需要所有许可权，就可以只用安全策略来赋予它们 ALLPermission，就像一种特殊的应用程序一样来处理。当然真实的情况比它稍微复杂一些。注意，系统类常常被空类加载器加载。决定一个类是否是系统类要看它是否有一个切实可行的空类类加载器。然而，在新的安全结构中还存在一些完好的未被破坏的传统代码。此外引导指令和其他问题可被专门解决。但是，由于结果被认为太不稳定，所以不能在 JDK1.2中尝试。我们希望在将来，对系统类的不同部分可赋予它们一些真正需要的细致的许可权。系统类这种细化将限制每个系统部件的许可权，并且进一步降低系统类中某个程序设计错误的影响。

3.10 java.lang.SecurityManager

JDK 1.0的最初版本中设计的java.lang.SecurityManager类是访问控制的关键。其中字节代码核查、类的装载以及运行期间的校验保证了类型安全。调用安全管理器可以决定是否接受或拒绝一个访问高度机密资源的请求。例如 java.lang.SecurityManager类在JDK 1.0中实现了沙盒安全模型。根据这个模型，应用程序 (驻留在本地文件系统中的类)被赋予全系统的访问许可权，而applets(通过网络加载的远程类)被拒绝几乎所有的基本特权。

在JDK 1.2的开发过程中，java.lang.SecurityManager也许是变化最大的。本节阐述了 JDK 1.1中已经存在的和 JDK 1.2中基本保持不变的 API，也涉及了一些在 JDK 1.2版中已经被禁用的方法。最后还介绍了一些新引入的 API和它们与以有的API的关系。

3.10.1 使用安全管理器的实例

实现安全核查的常见编程方法是首先安装 SecurityManager，然后再调用适当的 check()方法。例如,下面的代码段可以在打开一个文件时核查操作者是否具有读许可权。

```
public FileInputStream(String name) throws
```

China-pub.com 下载

```
        FileNotFoundException{
    SecurityManager security = System.getSecurityManager();
    if (security != null){
        Security.checkRead(name);
    }
    (proceed to open the file for read)
    }
```

SecurityManager因此被给予一个防止出现安全异常操作完成的机会。如果操作允许的话，SecurityManager程序(比如checkRead())简单地返回，但是如果操作不被许可权，那么SecurityManager程序会出现SecurityException。注意，SecurityException是运行期间的异常，虽然它能被捕获，但却没有被声明。

3.10.2 JDK1.2中没有改变的API

下列API都是存在于JDK1.1中而且在JDK1.2中没有被改变的，除了从protected到public的构造器已经被改变。

```
public SecurityManager()
protected native Class[] getClassContext()
public Object getSecurityContext()
```

在JDK1.2版本之前,SecurityManager是抽象类，因此供应商必须对它进行更细的分类，并还须创建出一个具体的实现，这很不方便。在JDK1.2中SecurityManager类是具体的，有一个公共的构造器。安全核查被放置在该构造器中，因为SecurityManager拥有高度机密的方法，因此不是任何人都能激活它们。传递安全核查所必须的许可权就是 Runtime Permission ("createSecurityManager")。

getSecurityContext()方法以类数组形式返回当前执行堆栈。数组的长度是执行堆栈所包含方法的数量。数组中第0个元素是当前执行方法的类，第1个元素是该方法的调用程序类，等等。这种方式对解决当前方法调用顺序非常有用，这是做出访问控制决定的基本知识。因为自查不会扰乱Java程序，所以方法必须是本地的。

getSecurityContext()方法创建一个压缩当前运行环境的对象。此方法的目的是创建一个运行环境的快速映射，当方法被调用时，你可以质疑安全核查是否可以通过。这个方法的缺省运行结果是返回一个AccessControlContext对象。特定的类AccessControlContext将在稍后的3.10.3和3.11.6节中解释。

存在一个全系统的安全管理器，java.lang.System类通过调用以下相关的方法来管理这个安全管理器：

```
public static synchronized void setSecurityManager(SecurityManager s)
public static SecurityManager getSecurityManager()
```

在set方法中，如果对于当前运行的Java应用程序还没有建立起安全管理器，那么已通过的部分被当做安全管理器。这个过程有时被称之为安装安全管理器。如果已通过的部分为空或者安全管理器还没有被建立，那么就没有操作被执行且方法返回的仅是默认值。如果安全管理器已经被安装，而且核查的调用者有RuntimePermission(“set Security Manager”)的许可权，那么安全核查被激活。如果是这样，那么已通过的部分被安装成新的安全管理器，否则

China-pub.com
下载

就会出现SecurityException。注意在JDK1.2以前的版本中，全系统的安全管理器仅能被设置一次。

方法get返回一个已创建或已安装的安全管理器，如果没有安装安全管理器，那么返回值为空。允许安全管理器为空是一个不完美的设计。它的缺点将要在稍后的章节中讨论。但这一设计特征已成为API中的一种。所以我们不打算改变它。

3.10.3 JDK1.2中禁用的方法

下列API在JDK1.2中已被禁用：

```
public boolean getInCheck()
protected boolean inClass(String name)
protected Class currentLoadedClass()
protected native ClassLoader currentClassLoader()
protected native int classDepth(String name)
```

这些曾使用的方法决定由哪一个类做出方法调用，这种做法是间断性的而且通常是特别基础的。具有代表性的是，堆栈中存在的类是否由非空的加载器决定，或者决定它有多深，也就是说，在当前的方法中，有多少方法调用了类。这样做将产生过多的代码。

这也会导致许多安全方面的漏洞。例如，在对象定位程序中，在两个已经存在的方法调用中常要加入多余的不确定的层或界面。在一个调用链中插入另一个方法将改变类的深度，这样就很难以类的深度作为定位指示的标准，特别是对于经常改动的软件代码。错误估计类的深度使得那些试图访问保护资源的代码成为值得信赖的系统代码，而实际上它们并不真正的值得信赖。

在JDK 1.2的新安全结构中,完全消除了对已经被禁用方法的需要。为了向后兼容我们只是禁用这些方法而不将它们删除，但我们强烈建议你不要使用它们。

SecurityManager包含了29个以check开头的方法。这些控制方法在执行某一个可能的敏感操作之前,从Java库中用不同的方法调用，这一规则仅有的一个例外是 :用于返回boolean的checkTopLevelWindow方法。下面列出这些核查方法：

```
public void checkCreateClassLoader()
public void checkAccess(Thread t)
public void checkAccess(ThreadGroup g)
public void checkExit(int status)
public void checkExec(String cmd)
public void checkLink(String lib)
public void checkRead(FileDescriptor fd)
public void checkRead(String file)
public void checkRead(String file, Object context)
public void checkWrite(FileDescriptor fd)
public void checkWrite(String file)
public void checkDelete(String file)
public void checkConnect(String host, int port)
public void checkConnect(String host, int port, Object context)
public void checkListen(int port)
public void checkAccept(String host, int port)
public void checkMulticast(InetAddress maddr)
public void checkMulticast(InetAddress maddr, byte ttl)
```

China-pub.com 下载

```
public void checkPropertiesAccess()
public void checkPropertyAccess(String key)
public boolean checkTopLevelWindow(Object window)
public void checkPrintJobAccess()
public void checkSystemClipboardAccess()
public void checkAwtEventQueueAccess()
public void checkPackageAccess(String pkg)
public void checkPackageDefinition(String pkg
public void checkSetFactory()
public void checkMemberAccess(Class clazz, int which)
public void checkSecurityAccess(String action)
```

这些核查方法大多数从它的名称就可以理解其含义。这种以名称命名不同安全核查方法的设计风格积累了大量的方法。这里存在一个较大的问题，因为每种现有的方法都是为一种特殊的资源类型而设计的，然而当系统增加一种新的保护资源的类型时，就需要一种适当的安全核查，而一般情况下现存的核查方法并不能预测到这一点。新的核查方法必须增加到SecurityManager 类中，这是一个非常严重的设计缺陷，因为在通常情况下扩展一个存在的SecurityManager是不可能的。例如Web浏览器就是一个固定的SecurityManager类，应用程序不能用与SecurityManager不同的新的保护资源来扩展运行系统。甚至当扩展 SecurityManager可行时，新的安全核查方法经常被包含在复杂的JVM 内部机制中，而且难以执行或正确执行。

如此苛求的条件可能导致存在的核查方法超载。例如在JDK 1.1中，System.setIn(InputStream)方法激活checkExec(),此方法检验调用者是否可以运行某个文件。另外一个超载的方法是checkConnect()方法，用一个-1端口调用这个方法意味着那个调用者试图了解主机的IP地址。超载的核查方法是非常不希望出现的，而且的确非常危险。

在JDK1.2中, 全部核查方法用下面的新方法重新执行 :

```
public void checkPermission(Permission perm)
public void checkPermission(Permission perm, Object context)
```

只有一个许可权参数的 checkPermission()通常可以在正在运行的线程范围内实现安全核查。当一个安全核查在一个指定的程序 (例如从一个线程A)内被激活，通常这个核查实际上是相对于另外一个不同的程序 (比如线程B)来运行的。在这种情况下，将采用带有合适参数 (比如线索B中的AccessControlContext)的checkPermission()。

从某种意义上说，所有存在的方法都被checkPermission所取代。为了向后兼容我们并不禁用这些核查方法。不过，我们用checkPermission方法重新执行它们并且消除核查方法超载的现象。下一个小节验证新的checkPermission方法，并且介绍它们如何与已存在的方法相关联。

新的checkPermission方法

第一种方法public void checkPermission(Permission perm)，核查由特定许可权所指定的请求访问是否允许建立在当前安全策略的基础上。如果准许，该方法返回一默认值；否则，出现安全异常。缺省的运行是调用 java.security.AccessController类中的方法checkPermission(),这一点将在3.11节中解释。

第二种方法public void checkPermission(Permission perm, Object context) ，核查由特定许可权所指定的请求访问是否允许建立在当前安全策略的基础上，如果运行过程中发出的请求通过核查。前面提到 getSecurityContext()方法创建一个压缩当前运行环境的对象，并返回AccessControlContext。如果通过的内容是AccessControlContext中的样本，那么将调用Context

China-pub.com
下载

对象中的方法checkPermission。如果请求被允许，该方法返回一默认值，否则出现安全异常。

AccessControlContext有下列API：

```
public AccessControlContext(ProtectionDomain context[])
public void checkPermission(Permission perm)
```

公用构造器用一系列给定的 ProtectionDomain对象创建一个 AccessControlContext对象，因此可以模拟这些对象的执行环境，而这些对象从不同的保护域中抽取类，并在给定的数组中按顺序相互调用。数组中的第一个元素与保护域中最新的类相对应。相同的域必须从数组中删除，并且数组不能为空。

用一个方法checkPermission()可替代许多核查方法。因为这些请求核查方法的名字不再是不变的代码，它们可以在方法checkPermission()的许可权参数中编码。这个做法有很大的益处。方法checkPermission()的运行包含了Java运行时内部状态的核查以及复杂算法的运行。这种运行方式可应用于所有的许可权类型。因此，要想添加一个新的保护域，只需简单地创建一个新Permission类，并在合适的位置设置checkPermission()，这个新Permission类可完全用Java语言描述，这样就不用修改已有的 SecurityManager类。

为实现这个新的checkPermissian方法，必须使原有的调用方法与 checkPermission()方法中合适的许可权相匹配。表3-1中列出了可用相应许可权替代的核查方法。对于每一个核查方法，其对应的许可权类别都附有目标和操作，许可权名由双引号做标记以表述真正的属性名。例如："foroperty nameg"。

表3-1 被checkPermission替代的核查方法

核查方法	RuntimePermission内容
checkCreateClassLoader	createClassLoader
checkExit	exitVM
checkSetFactory	setFactory
checkAccess(Thread)	modifyThread
checkAccess(ThreadGroup)	modifyThreadGroup
checkRead(FileDescriptor)	readFileDescriptor
checkWrite(FileDescriptor)	writeFileDescriptor
checkLink	loadLibrary.{library name}
checkPackageAccess	accessClassInPackage.{package name}
checkPackageDefinition	defineClassInPackage.{package name}
checkMemberAccess	accessDeclaredMembers
checkPrintJobAccess	queuePrintJob
java.lang.System.{setIn,setOut,SetErr}	setIO

下面是一个许可权转换的例子，以表明方法 checkLink()是如何被重复执行。

```
public void checkLink(String lib) {
checkPermission(new
RuntimePermission("loadLibrary."+lib));
}
```

表3-2列出了相匹配的保留的核查方法调用。

China-pub.com 下载

表3-2 匹配的核查方法调用

方 法	Permission类	Permission内容
checkPropertiesAccess	PropertyPermission	
checkPropertyAccess	PropertyPermission	{property name}
checkTopLevelWindow	AWTPermission	showWindowWithoutWarningBanner
checkSystemClipboardAccess	AWTPermission	accessClipboard
checkAwtEventQueueAccess	AWTPermission	accessEventQueue
checkSecurityAccess	SecurityPermission	{action}
checkExec	FilePermission	{file or directory pathname}
checkRead(String)	FilePermission	{file or directory pathname}
checkWrite(String)	FilePermission	{file or directory pathname}
checkDelete	FilePermission	{file or directory pathname}
checkConnect	SocketPermission	{host:port}
checkListen	SocketPermission	{host:port}
checkAccept	SocketPermission	{host:port}

在JDK1.2中有一些新建的安全核查许可权类，例如 Netpermission，这些许可权类与JDK1.1中的任何核查方法都不对应。

许多软件开发者也许会提出这样一个问题，当编写一段安全核查代码时，我们应调用旧的核查方法还是新的checkPermissian方法。这主要考虑向后兼容的问题，如果你喜欢 JDK1.2以前的自定义安全管理器，最好使用旧的核查方法，因为旧的安全管理器可以忽略或者自定义某些核查方法的调用。实际上，为了向后兼容，我们不修改 JDK中调用核查方法的代码。这是因为对于设置有自定义 SecurityManager的已存在的应用程序，希望 JDK中的代码调用自身的安全核查方法。如果为了调用 checkPermissian方法修改了JDK代码，那么自定义安全管理器就应该被忽略。

如果你想确保 JDK1.2中新设计风格的安全管理器被查询，那么，必须调用新的checkPermissian方法，我们建议对新的应用程序代码使用新方法，这样会更安全和简洁。我们希望在不久的将来，安全管理器中所有的check()方法都将获得支持。

3.11 java.security.AccessController

尽管SecurityManger类定义了checkPermission类,核查方法作为界面激活相应的安全核查，但是这些界面不能详细说明如何实现安全核查，尤其不能详细说明在什么情况下，一个请求可以被接受或拒绝。这是必然的，因为预先采取所有合理的方法来执行安全核查是不可能的。例如，一个应用程序执行一个多级安全策略[3]，而另一个应用程序执行支持责任分离的策略[44]。以包含所有策略规则的表达方式为多级策略提供 Policy对象，这也许不可能，或者说是非常困难的。另一种方式是忽略所有定义在 SecurityManager中的核查方法，而运行一种的安全策略，并安装与此策略相应的依赖于运行环境的安全管理器。

不能完全详尽说明如何实现安全核查有它的缺点。一是开发者可能会写出行为不一致的安全管理器。例如，两个自定义的 SecurityManger类也许会执行语义完全相反的核查方法，这样导致了不一致，特别是可能会产生危险行为；另一个问题是 SecurityManger很难更正，尤其是对安全不是特别熟悉的应用程序开发者。一些程序员倾向于固化核查方法里的安全策略的

China-pub.com
下载

代码，而没有为进一步的开发留下足够的空间，另一些程序员可能会犯细微的安全错误。

因此现急需提供一个详细说明访问控制算法,且在大多数应用程序中都可充分利用的缺省实现，开发者用户可通过不同应用程序和平台运行它 ,且行为保持一致。JDK1.2中介绍的缺省实现属于AceessController类。换句话说，通过缺省方式， SecurityManger激活了AceessController中定义的方法并必须做出安全决定。

下一节将验证Access Control的界面设计。稍候的章节将详细讨论嵌入在新类中的一般访问控制算法。

3.11.1 AceessController的界面设计

AceessController类是最后声明的，因此它不能被更细地分类。它没有公用的构造器，因此不能用具体例子说明它。它只有如下所列的静态方法：

```
public static void checkPermission(Permission perm)
public static native Object
    doPrivileged(PrivilegedAction action)
public static native Object
    doPrivileged(PrivilegedAction action,
 AccessControlContext context)
public static native Object
    doPrivileged(PrivilegedExceptionAction action)
        throws PrivilegedActionException
public static native Object
    doPrivileged(PrivilegedExceptionAction action,
        AccessControlContext context)
        throws PrivilegedActionException
public static AccessControlContext getContext()
```

目前熟悉的checkPermission()方法核查一个由许可权参数指定的访问请求在当前执行的程序中是否被允许。如果允许，此方法将返回缺省值，否则，它就会出现AccessControllerException,它是SecurityException的子类，可提供失败的详细原因。

3.11.2 基础访问控制算法

假设准许访问受控资源的决定只能在正确的程序中作出，那么这个决定必须提供下面这些问题的答案:谁在请求什么以及代表谁等问题。通常情况下访问控制程序是一个线程。少数情况下,访问控制的决定会在为获得正确的程序信息而相互关联的多个线程中执行。执行的线程可能完全发生在一个单独保护域 (就是所有类和对象包含在属于相同保护域的线程中)或者发生在多个保护域，如应用域和系统域。例如，可打印消息的应用程序将必须与指向输出流唯一通道的系统域相配合。

当前执行的程序完全反映了方法调用的当前顺序。每种方法都定义在属于同一保护域的类中。因此可为执行的程序形成一个保护域的顺序。基本访问控制的算法可归结为一句话 :只有访问请求得到准许，运行程序中的保护域才能被赋予相应的许可权。

术语caller用于表示当前执行的程序中的保护域，因为一个保护域可与多个程序联系。基本算法可按下列构造性方式来表达。

```
for each caller in the current execution context {
   if the caller does not have the requested permission
      throw an AccessControlException
```

China-pub.com 下载

```
        throw an AccessControlException
    }
    return normally
```

为了测试这个基本算法，可以假设一个小游戏程序包含一种名为 openHighScoreFile()的方法，它调入FileInputStream的构造器以打开一个高分文件，该文件保存着玩此游戏的分数列前十位选手的分数，构造器调用 checkRead()，再依次调用安全管理器中的 checkPermission()方法。安全管理器依次调用 AccessController中的checkPermission()方法，此时，执行程序看起来好像图3-6中的瞬态图。

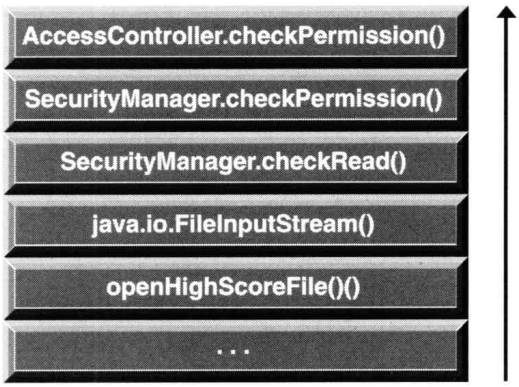


图3-6 堆栈结构瞬态图

在这个例子中,执行程序中存在着两个不同域 :系统域和为applet分配的域,只有两个域都拥有文件许可权时,算法才允许文件被打开。因为系统域缺省拥有所有的许可权，所以算法可简化为只核查applet是否被赋予了文件许可权。如果 applet没有文件许可权，文件将不能被打开，即使applet试图利用系统代码的帮助来这样做也不行。

最后一点相当重要，因为仅仅通过调用系统域并不能使应用域获得附加许可权。安全推断可导致相反的情况。

在相反的情况下，系统域激活应用域中的一种方法，例如， AWT系统代码调用 applet的paint()方法来显示applet。假设applet试图从paint()里打开高分文件。图3-7显示执行语境。

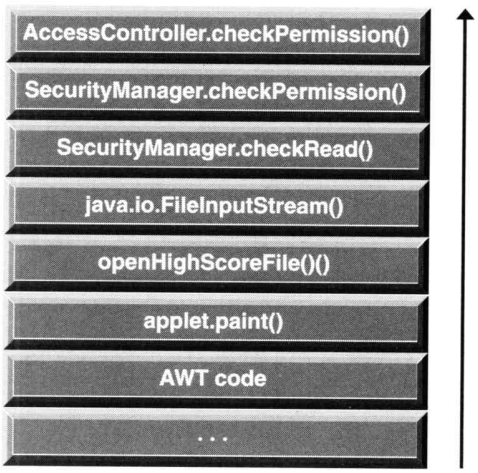


图3-7 堆栈结构执行语境

China-pub.com
下载

另外，如果 applet没有被赋予必要的文件许可权，即使出现 AWT系统代码调用 FileInputStream，文件也不能被打开。否则，因为是系统代码回调自己的文件代码， applet将获得很大的权力。在JDK1.2中,访问控制器的内置访问控制算法用于防止这种误操作。

由于调用了大权力域的结果，小权力域不能获得附加许可权，另外，当调用小权力域时大权力域会失去它的权力。最小权限原则上适用于贯穿多保护域的线程。

在JDK1.2以前,任何执行访问控制决定的代码依赖于它对调用程序状态的了解程度 (也就是，调用程序是系统代码还是 applet码)。这种布局很容易毁坏，因为只了解调用程序的状态是不够的，还必须了解那些调用程序状态的上一级程序等等。此时，把查询过程放在典型编程器上会使之成为一个沉重的负担而且容易出错。这还意味着 AWT代码书写者需注意applet会发生的情况。在AceessController中执行的算法通过自动访问核查过程减轻了这个负担。

3.11.3 继承方法

继承方法的细节需要说明，基本算法和它在后面章节讨论的扩展版本将依照调用程序的顺序来定义，每个算法由调用的方法来表示，激活方法用于鉴别它所定义的类。类与它所属的保护域相链接。保护域被赋予许可权，违背了所作的访问控制的限定。假设类 B是类A的子类，类A定义了方法x()，这时类B继承了它但却无法取代。进一步假设类 A和类B属于两个不同的保护域。当激活方法B.x()，是由类A定义并执行这个方法呢?还是类B不变地继承了这种方法呢？

任何一种选择看起来都比较合理的，但总的来说，将调用者与方法的执行联系起来显得更自然，这是因为权力大的类能以安全的方式来写出它的方法，这允许权力小的类继承它们而且完成权力小的类由于没有许可权而不能完成的任务。于是在给出的情况下，类 A被认为是真正的调用程序，因此它的保护域需做必要的访问许可权核查。注意，如果在类 B中，方法x()被取代，而不再调用它的父方法 x()，即使这种替代不改变方法的运行， B.x()的调用程序将会由B代替A。这是因为一旦子类取代了方法调用，它的父类就不再对调用程序负责。换言之，B可以任意的方式改变x()的执行，因此其保护域应得到核查，这两种情况都在图 3-8中描述。

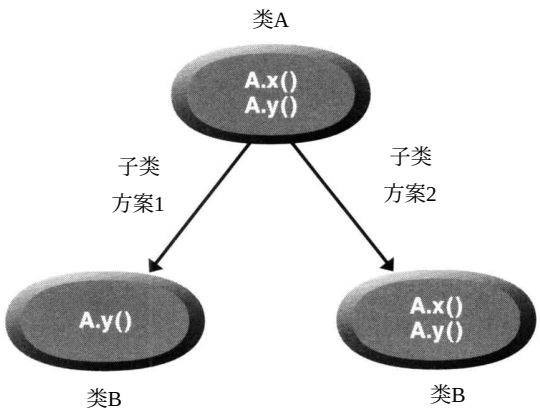


图3-8 继承方法

3.11.4 扩展带有特权操作的基本算法

基本算法简单而安全，因为所有涉及到计算的代码为了进行申请的访问必须被赋予足够

China-pub.com 下载

的许可权。然而，这种算法太受限制。例如，研究一下口令变换应用程序，当一个用户代码调用此程序，用户需要快速把新口令打两遍 (以保证口令被正确输入)，然后输入旧口令。如果旧口令与存储在口令文件中的口令相符，新口令就被存储在更新的口令文件中。注意，为了读写访问应用程序需要打开口令文件，并且假定应用程序已经被赋予足够的访问许可权。在基本算法情况下，应用程序不能打开口令文件，因为它被用户代码调用了，而用户代码没有直接访问口令文件的许可权。这样一来为了打开文件，需要给应用程序一条途径来停止基本算法，且应用程序要了解它现在所作的工作。

在另一个类似的例子里， applet也许不能直接访问特定的系统属性，但是为了完成任务，为applet服务的系统代码需要获得一些属性。

为了处理这些异常的情况， AceessController类包括一种静态方法，就是 doPrivileged()。一个调用doPrivileged()的代码告诉Java运行系统，忽略代码调用者的状态，并且代码会对使用自己的许可权负责。下面是涉及说明特权状态的一种扩展访问控制算法：

```
for each caller in the current execution context {
   if the caller does not have the requested permission
      throw an AccessControlException;
   if the caller is privileged, return normally;
}
return normally
```

在这个扩展算法中，从最近的调用者开始，调用者必须以相同的顺序被核查以确定它们是否互相调用。

口令变换应用程序带有“激活自身特权”的调用，即使用户代码没有访问许可权，它也可使用下面代码段来打开文件。

```
public void changePassword() {
   // Use own privilege to open the password file.
   AccessController.doPrivileged(new PrivilegedAction() {
      public Object run() {
         // Open file for reading and writing.
         ...
         return null;
      }
   });
   // Verify user's old and new passwords.
   ...
}
```

这里来详细看一下 doPrivileged。当执行此方法时， JVM把调用线程的堆栈结构标记为有特权的。在前面的例子里与 changePassword()方法相当的堆栈结构被标志为有特权的。在完成这种方法的执行前， JVM解除对调用线程的堆栈框架的标记，这样就表明它不再是有特权的。

通过调用 doPrivileged，调用者仅仅能够给它已经拥有的部分赋予特权，理解这一点非常重要。一个代码块获得的许可权不可能比已经被赋予的许可权集还多。被赋予特权仅仅告诉AccessController忽略它的调用者。例如， AccessController在它已经验证特权代码拥有申请许可权之后，可以停止核查。

此外特权块对使能特权的线程是特定的，也就是说，即使其他线程正在执行属于相同保

China-pub.com
下载

护域的代码，在一个线程中某种代码被赋予特权，不会对其他同时运行的线程产生影响。

其他要考虑的细微区别是doPrivileged方法能通过使用java.lang.reflect.Method.invoke()来激活。在这种情况下，被赋予特权模式的特权不是 Method.invoke(),而是激活它的非反射的代码，否则，系统特权会被错误的(或恶意的)赋予代码。

让我们再更深入的研究一下,如何正确、细致地使用doPrivileged。在口令变换应用程序的例子中，假设打开口令文件的代码实际上在另一个叫 openPasswordFile()的方法中，该方法打开口令文件并且返回对文件输入流的对象引用，实例代码会变成如下情况：

```
public void changePassword() {
   // Use own privilege to open the password file.
   AccessController.doPrivileged(new PrivilegedAction() {
      public Object run() {
         // Open file for reading and writing.
         f = openPasswordFile();
         return null;
      }
   });
   // Verify user's old and new passwords.
   ...
}
```

这个代码可以像以前一样正确的运行，从 openPasswordFile()中调用doPrivileged将会是一个错误。这是为什么呢？因为用户代码然后能直接调用它。进一步的说，因为方法中的特权，用户代码可以获得对口令文件的引用。这给我们一个教训，像 openPasswordFile()这一类的方法不知道或者不能充分控制调用它的人选的话，就不应该激活自身的特权，这是由于方法会返回一些受保护的资源。另一方面，即使任何人都能调用ChangePassword()，ChangePassword()也可安全激活自己的特权。这是因为 ChangePassword()不会把口令文件向外界显示，并且只在口令核查成功后才会在内部处理它。

你也许已经注意到特权特征的设计是不对称的。也就是说你可以选择使用自己的特权并且告诉访问控制器忽略在你的方法之前的调用者，但是不能告诉访问控制器忽略那些接着调用的调用者(在你使能自己的特权之后)。因此，如果后来调用了相应保护域没有许可权的方法，即使你有许可权，这种方法调用也不能获得许可权。这种不对称是为了保护你而设计的。如果访问控制器也忽略了那些接着调用的调用者，那么实际上已经把你的许可权赋予了那些调用者。你或许可以控制直接调用哪一个调用者，但不能控制这个调用者随后会调用谁。如果那些调用者中任何一个是恶意的或没有能力的，你的特权和许可权就会被错用或滥用。相信一系列未知用户是一个非常错误的想法。为了避免意外掉入这样的陷阱，而设计了这种算法。

3.11.5 特权操作的三种类型

上一节的代码实例通过把PrivilegedAction界面作为参数演示了doprivileged最简单的用法。这里再重复一下，这种用法模式只有在特权块中的代码不需要返回一个值时才会有用。

```
somemethod() {
   ...normal code here...
   AccessController.doPrivileged(new PrivilegedAction() {
```

China-pub.com 下载

```
        public Object run() {
            // Privileged code goes here, for example:
            System.loadLibrary("awt");
            return null; // Nothing to return.
        }
    });
    ...normal code here...
}
```

这里PrivilegedAction是一个具有单一方法的界面，该方法名为 run，可返回对象。这个实例演示了所提供的run方法的具体实现。当调用 doPrivileged时，PrivilegedAction实现的实例被传给它。doPrivileged方法在使能特权之后，从 Privileged实现中调用run方法，并且把方法的返回值作为doPrivileged的返回值(在这个事例中被忽略)。

如果特权块中的代码需要返回一个值，代码如下：

```
somemethod() {
    ...normal code here...
    String user = (String) AccessController.doPrivileged(new
        PrivilegedAction() {
        public Object run() {
            return System.getProperty("user.name");
        }
    });
    ...normal code here...
}
```

用法需要一个关于doPrivileged返回值的动态丢弃。

另一种方法是使用最后本地变量，如下所示：

```
somemethod() {
    ...normal code here...
    final String user[] = {null};
    AccessController.doPrivileged(new PrivilegedAction() {
        public Object run() {
            user[0] = System.getProperty("user.name");
            return null; // Still need this.
        }
    });
    ...normal code here...
}
```

第三种途径是写出一个具名的类，这个类可以安全处理打印信息，如下所示：

```
somemethod() {
    ...normal code here...
    GetPropertyAction gpa = new GetPropertyAction("user.name");
    AccessController.doPrivileged(gpa);
    String user = gpa.getValue();
    ...normal code here...
}
class GetPropertyAction implements PrivilegedAction {
    private String property;
    private String value;
```

China-pub.com
下载


```
    public GetPropertyAction(String prop) {
        property = prop;
    }

    public Object run() {
        value = System.getProperty(property);
        return value;
    }

    public String getValue() {
        return value;
    }
}
```

在这个例子中没有类型丢弃。由于run方法仍会返回一个值，所以可以用如下方法简化somemethod：

```
somemethod() {
    ...normal code here...
    String user = (String) AccessController.doPrivileged(new
                          GetPropertyAction("user.name"));
    ...normal code here...
}
```

最后，界面PrivilegedAction是为不会出现核查异常的特权代码 (如FileNotFoundException) 而服务的。如果代码能出现这样的异常，该异常必须在方法列出的条款中说明，然后必须使用PrivilegedExceptionAction来代替。

```
somemethod() throws FileNotFoundException {
    ...normal code here...
    try {
        FileInputStream fis = (FileInputStream)
            AccessController.doPrivileged(new
         PrivilegedExceptionAction() {
           public Object run() throws FileNotFoundException {
              return new FileInputStream("someFile");
           }
        });
    } catch (PrivilegedActionException e) {
        // e.getException() should be an instance of
        // FileNotFoundException, as only "checked" exceptions
        // will be "wrapped" in a PrivilegedActionException.
        throw (FileNotFoundException) e.getException();
    }
    ...normal code here...
}
```

doPrivileged的使用很不方便，但是这是有原因的。这个设计选择的根本理由将在3.11.9节中讨论。同时记住特权操作必须谨慎使用，因为特权操作利用了你自己的许可权，而不管你可能正在处理不可信代码。特权代码块应该尽量小，而且所有可以在块外执行的代码不应该在块内执行。

3.11.6 访问控制语境

本章前面提到过， getContext()采用当前执行语境的瞬态图，把它放在一个 Access Control

China-pub.com 下载

Context对象中并且返回该对象。在Java运行系统中，一条代码可以从任何一个子线程中开始，这些子线程随后开始它们自己的子线程等等。当一条新线程被创建时，JVM就创建一个新的执行堆栈，但要确保当前执行语境被新的子线程继承。换言之，就子线程的安全来说它可以返回到其所有祖先的语境。更详细地说，当前执行语境的瞬态图包括当前线程继承的AccessControlContext。

注意，严格地说，JVM不必强迫一个线程循环地继承它先前的语境，由于没有继承，就不会带来安全问题。然而，我们的经验表明，一个典型编程器希望安全语境被继承，并且使其感到意外的一般是不需要的东西。在这些情况下，自动继承在实际上就显得很方便。例如，在服务器应用程序中，一条主线程会创建从线程来处理单个输入申请，这时从线程手工创建考虑主线程安全语境的代码就成为一种负担。

另外需要强调的一点是当创建子线程时，继承的语境是父线程中此刻的语境。为了进一步的引用，继承的语境基本上被冻结，并且父线程随后可以继续和变换它的语境，而不会影响继承的语境。

以AccessControlContext作为参数的doPrivileged方法将调用线程堆栈结构标记为privileged，并把特权结构与给定的AccessControlContext关联起来。后来所有的访问控制校验都包含语境，而且这些语境在特权结构的ProtectionDomain被校验后才被校验。在下一小节中，当阅读完完整访问控制算法后，对这个方法如何使用的理解也许会变得比较容易一些。它的用法如下，其中acc是AccessControlContext对象。

```
somemethod() {
   AccessController.doPrivileged(new PrivilegedAction() {
      public Object run() {
         // Code goes here. Any permission checks from this
         // point forward require both the current context
         // and the snapshot's context to have the desired
         // permission.
      }
   }, acc);
   ...normal code here...
}
```

3.11.7 完整的访问控制算法

假设当前线程内含有m个调用按1, 2,…m的顺序排列。调用m激活checkPermission方法，基于下面的算法可以决定哪一个访问被准许或被拒绝。

```
i = m;
while (i <> 0) {
   if (caller i's domain does not have the permission)
      throw AccessControlException;
   if (caller i is marked as privileged) {
      if (a context was specified in the call to doPrivileged)
         context.checkPermission(permission);
      return;
   }
   i = i - 1;
};
return;
```

China-pub.com
下载

这个访问控制算法比3.10.4节的扩展算法要复杂一些。它们两者之间有一点不同。当特权结构被核查，且访问控制语境在调用 doPrivileged (PrivilegedAction,AccessControl Context)中被指定，仅当在规定的访问控制语境中需要的许可权被允许，安全核查才可以通过。

因此可以用doPrivileged(PrivilegedAction，AccessControlContext)使能特权结构。但只对在规定的访问控制语境中的许可权有效。换句话说，可用这种特性进一步限制特权扩展。如果没有一个语境被指定，特权结构可以执行所有调用者赋予的许可权。在指定语境时，可执行许可权就限制为在指定语境内的许可权。

在理论上，抽象地说，访问控制算法在计算线程任何一点时，在考虑特权状态和继承访问控制语境情况下，有效许可权是执行线程的所有保护域许可权的交点。实现这种算法有多种策略。下面讨论最明显的两种： eager evaluation和lazy evaluation。

对于eager evaluation实现来说，只要线程进入新的保护域或从已存在的保护域退出时，有效许可权集被动态更新。其优点是：对一个许可权是否被允许的核查可以被简化而且多数情况下速度会更快。其缺点是：许可权核查远远少于交叉域调用，大部分的许可权更新或许是徒劳的。

在lazy evaluation实现中，JDK1.2采用这种方法。只有当发生许可权核查请求时才检查线程状态并判决是拒绝或准许特殊的访问请求。这种方法一个替在问题是许可权核查时的性能下降，不过在eager evaluation中也存在这种问题。在JDK1.2实现中，这种算法性能是可以接受的。因此，我们认为lazy evaluation是最经济的方法。⊖

注意到：访问控制基于与当前执行语境关联的保护域，故当优化静态的， JIT，编译器或JVM特定实现时，语境必须保持完整。例如，内在方法必须小心使用，以防保护域信息丢失并确保AccessController类能正确实现。

3.11.8 SecurityManager与AccessController

回顾前面几节的内容，当调用安全核查时，可以调用方法 checkPermission()，可以调用SecurityManager类中定义的其他核查方法。这取决于是否采用了JDK1.2以前的安全管理器类。现在还有另一种选择，那就是调用 (SecurityManager)或AccessController中定义的方法checkPermission()。上述方法的不同主要集中在两方面：

首先，有时并不存在已安装的SecurityManager，所以其上的核查方法不能被激活。与此相反，AccessController中的静态方法总是可以被调用。回想以前调用SecurityManager时，应用的下列代码：

```
SecurityManager sm = System.getSecurityManager();
if (sm ! = null)
sm.checkPermission(permission);
```

但是，你总是可以调用：

```
AccessController.checkPermission(permission)
```

因此，如果想保证无论是否安装了系统型 SecurityManager，总可以调用安全核查，那么应调用AccessController，注意：一些已有的应用程序将测试是否已安装了 SecurityManager中的实例，基于得出的表明安全性状态的测试结果，这些应用程序运行不同的操作。考虑到程序的向后兼容性问题，调用SecurityManager显得更为合适。

第二个不同点在于调用 SecurityManager并不能保证某个特定的访问控制算法可用，它也

⊖ 关于保护域的实现细节和性能、优化技术的讨论请参阅文献 [28]。

China-pub.com 下载

许已被扩展或者是安装在一个自定义的安全管理器中。与此相反，调用 AccessController可以保证以前所有指定的访问控制算法都可用。于是，如果不想让安全核查从属于一个自定义的安全性管理器。那么应直接调用 AccessController，否则调用SecurityManager。

另一个需要引起注意的问题是：由于 SecurityManager为安全核查定义了一个一般性的界面，所以它不提供已在 AccessController中定义的特权机制。实际上，如果在程序代码中使用了特权机制，而后来又通过调用 SecurityManager来运行一个安全核查，那么已安装的安全管理器是否是由JDK1.2提供的，而且与AccessController一致或等效，特权状态才会考虑进去。

你也许会问为什么我们提供了这么多种选择？采用一种方式不是就足够了吗？实际上，这些选择都是依据经验得出的，是为了保持一致性和一般性的平衡所必须的。从长远来看，我们希望不再经常使用自定义安全管理器，即使它们被定义，也是建立在已存在的AccessController功能的基础上。尤其是它们所提供的应该是原有功能的扩展，而不是增加不兼容的属性。而且，在一些特殊的运行环境里，需执行大量不同类型的安全策略。自定义的安全管理器也许不可能实现那些由 AccessController实现的算法。

3.11.9 特权操作发展史

为了进一步讨论AccessController，本节将提供更多有关特权方法是如何发展的背景知识。主要目的是为了帮助程序员书写安全代码，并在程序员犯错误时，能保证系统的安全性。

将所期望的安全结果与 UNIX系统的setuid功能相比较可得出有益的启示。与 MS-DOS和MS-Windows 操作系统相比较， UNIX系统的安全性在某种程度上说,考虑得更为清晰。它将调用程序的用户所做的操作限定为用户特权。在有些时候，这些限制过于严格。 setuid机制抑制了这些限制。但是，整个受 setuid控制的程序还是“ 受限 ”，因为程序中的任何错误都会潜在地导致保持安全漏洞。在 JDK1.2中，为了避免这种可能性，创建了应用程序界面，使程序员的操作受到了限制，仅有少量的方法可调用，并且危险性操作被限制在一定的范围之内。通过这种方式，可降低由这些敏感操作而造成的意想不到的安全危害。

我们考虑了多种设计方法，其中一种是用方法修改器来扩展语言，称之为“ 特权 ”。当输入一个方法也就是赋予了相应的特权。相反，如果方法禁用，那么特权也就被取消。这种方式绝对是一个最清晰的设计方法。 但是，它为java语言增加了许多附加项，要求编译器不断被更新。这种改变不是轻易就能做到的，而且一个方法修改器不能采用相应的参数。所以我们决心放弃这种方法。同时也拒绝采用其他一些方法，例如：改变已存在非安全代码的语法，要求在JVM中得到支持。这些方法很难在所有平台都能运行。

直到JDK1.2测试版3，我们才提出了一种设计方法，在 AccessController中提供了下列两种方法：

```
public static native void beginPrivileged()
public static native void endPrivileged()
```

如下所示，将一段代码声明为特权：

```
somemethod() {
   (normal code here)
   try {
      AccessController.beginPrivileged();
      // Privileged code goes here, for example:
      System.loadLibrary("awt");
```

China-pub.com
下载


```
        } finally {
            AccessController.endPrivileged();
        }
        (more normal code here)
    }
```

这个设计的优点是：在块结构内部使用相当简单。另一个优点是，将原来限定在同一方法结构中调用的方法endPrivilieged()和beginPrivilieged()，优化为只有在特权不存在时，方可被调用。也就是特权在两者之间尽可能地做周期性转换。有时偶尔会出现程序员忘记调用endPrivilieged(),为此,我们创建了许多措施和核查，以防止从不同结构中调用 begin和end方法时出现的不匹配现象。例如，当程序员本该转换特权状态时，却忘了这样做，上述措施和核查方法就可以防止出现这种情况。最后，要说明的一点是，要求以平台独立的方式达到结构匹配而且运行精确非常困难。所以我们放弃了 doPrivilieged界面风格。

我们所采用的设计方法相当合理，只是在编程上略显复杂，这一点今后将会得到提高，例如采用合适的语言结构。

3.12 小结

为了对JDK1.2安全结构运行的整个过程做一个小结，这一节将带你浏览一下 Java应用程序的处理方式。当通过 Web浏览器和appletviewer浏览applet时，或是通过激活java程序以命令行方式运行Java应用程序时，下列一些步骤有可能发生：

(1) 类文件通过最初的代码验证，被得到和接受。

(2) 确定类代码资源，如果代码被明显的标识，这一步骤包括签名验证。

(3) 引用安全策略，基于类的代码资源，一系列许可权授予此类。这一步骤中，如果测试对象不存在，那么将被创建。

(4) 创建保护域已标识代码资源并保持许可权集，加载类，定义与其相关的保护域，如果相应的域已被创建，那么将复用 ProtectionDomain，而不再创建新的许可权集。

(5) 调用的也许是类中的对象或是运行方法，运行类型的安全核查将继续进行。

(6) 当调用安全核查而且此类的一个或几个方法在调用链中时，访问控制器核查保护域，特别是它的许可权集，以核查所请求的访问是否被赋予足够的许可权。如果满足条件，则继续执行。否则出现安全异常。 (对于所有处于调用链中的类，都以这种方式核查。)

(7) 当出现安全异常时，也就是出现运行异常时，放弃运行 JVM。

这一操作流程还有一些变动，例如 :在一个惰性方法中， Policy对象、保护域以及许可权的创建一直延迟到第一个安全核查发生时。因为只有在必须使用时，才会调用它们。这种延迟策略有利于减少运行程序的启动时间和运行时间。

新的安全结构所采用的基本思想植根于40多年的计算机安全研究基础之上。例如访问控制列表的所有思想[42]。在为文件系统和其他系统资源指定访问许可权方面，我们遵循了UNIX系统的一些规则。但是,显而易见，我们的设计思想主要受保护域以及用如Multics中的互疑程序[69、75]及Hydra中的“ 权力放大 ”[36、81]等处理方式这样一些概念的启发。

在UNIX和MS-DOS系统中没有体现出的一个新颖特性是：最小特权原则的应用。在一个调用序列中，这一原则自动贯穿于所有保护域中的一系列许可权中。通过这个方式，系统和应用软件中的一些程序错误不会成为系统的安全漏洞。

China-pub.com 下载

尽管JVM一般在Solaris这样的主机操作系统上运行,它也可以直接在硬件上运行,例如在网络计算机JavaStation上运行JavaOS[66]。为了保证平台的独立性，JDk1.2的安全结构不依赖于由底层操作系统提供的安全特点。

此外，这种结构不会忽视基本的操作系统保护机制。例如，通过配置一个细致的访问控制策略，用户可以给某些软件赋予特定的许可权。这是积极的，然而只有底层操作系统自身才可以给用户赋予这些许可权。

JDK1.2的安全机制还有另一个重要特点，就是它基于语言并且在一个单一地址空间内。这个特点是JDK1.2与其他更传统的操作系统的重要区别，该特点同近来基于软件保护和安全核心扩展(例如文献[7、10、16])方面的工作联系紧密。不同的研究组近来一直在使用不同的编程技术来达到相同的目的。在一个典型的操作系统中，一个交叉域调用非常昂贵。而在JDK1.2中，交叉域调用只不过是又一种方法调用，并且非常便宜。

新的设计有以下值得重视的特点:

(1) 安全策略的内容不但与机制的实现，而且与界面相分离。这就为发展提供了最大的空间。它也允许策略的配置与运行环境完全分离,这样就会减少系统管理的复杂性。

(2) 访问控制算法与它正在核查的许可权的语义相分离。这就允许访问控制器代码 (也许是一个具体应用程序)和JDK1.2发布后引入的许可权类可以重新使用。

(3) 许可权类层次的引入使面向对象的功能更强大 (尤其是封装)。这意味着访问控制许可权不但可以动态的而且可以静态的表达，并且每一个 Permission类可以定义它自己的语义，例如如何与它自身类型的许可权相关联，如何与不同类型的许可权相关联，以及如何解释通配符符号和其他的一些特性。

(4) 安全类加载机制和扩展机制一起把安全性的范围延伸到 Java应用程序，这样就产生了一种针对所有Java代码的、统一的安全结构和策略，而不管这些代码的起源或状态。

除了这些技术上的优点之外，我们也非常重视好的界面设计。我们在 API设计这个课题上做了大量的工作，例如不同类中合理的工作分配。我们也试图保留最小的一组类和 API，使尽可能多的类为私有，并为这些类和方法创建合适的名字，并且使这些名字尽可能短而且精确。直到我们很好地掌握了 API，才会开始使这些代码标准化。我们将可以对建议和意见做出反应并且在工程当中对 API进行广泛的修改，所有这一切都不会有太大的困难 ,而且不会使代码质量和工程的交付受到影响。另外，我们还用更强大的结构替换了一些脆弱的特点，例如我们反对在SecurityManager类中使用的一些方法。

好的界面设计的重要目标是易于应用。在这方面， JDK1.2设计优于其他一些设计。一个典型的例子就是一个商业浏览器，该浏览器实现的特点类似于 JDK1.2的特权方法调用。然而，该浏览器要求一个应用程序在使用赋予的特权之前先使能它。在这种情况下，为了打开一个该应用程序已获读许可权的文件，应用程序不能只打开一个 FileInputStream，而应该首先调用一个具体的方法——基本声明,“我将会行使读这个文件的权利”——然后打开文件。如果没有实现声明，下面的打开操作将会失败。这样的界面设计有很多缺陷。

(1) 如果JDK1.2已经采用了这样的一个设计，所有现存应用程序和 applet将不得不重写以清楚的进行这种声明。这将会严重破坏向后兼容性。

(2) 在一个多应用程序环境里，顶层应用程序为获取一个文件而提出申请，这是一种普通的情况。该申请可以由底层的应用程序通过多种方式加以满足。例如通过从文件系统中获取

China-pub.com
下载

文件，从数据库中恢复文件或从 Web服务器里下载文件。在这种情况下 ,顶层应用程序想预先知道是否赋予底层应用程序读文件、访问数据库或者连接网络的权利就会很困难，还没有一种简单的方法可以做到这一点。

然而，如果应用程序使用 JDK1.2特点，它不需要确定使用哪一个权利。只要应用程序已经被赋予正确的许可权，申请就可以用三种方式之一加以满足。换句话说，在复杂环境下，要求一个行使权力意图的声明可能会出问题，尽管在单层应用程序情况下有可能正常工作。

(3) 如果要在一个受保护的资源里布置新的或是修改过的安全核查，那么任何已存在应用程序或applet代码在像以前那样运行之前必须由一个新的声明重写以使能这个新的权利。这意味着应用程序或applet必须重新编译且重新部署以继续访问同一资源。这简直就是一个恶梦。

与之对比，在 JDK1.2中，只需修改安全策略以给访问应用程序的许可权赋予更多权利。应用程序或applet将会像以前那样运行。

我们在JDK1.0中确实遇到了两个虽然不便但却不能改变的特点。首先，系统类传统上一直和一个用本地代码写成的原始类加载器一起加载。结果，所有的系统类 (在JDK1.2中称为引导类)和一个空系统类加载器从 Java运行环境中加载。这种特别的实现特点成为一种事实上的 API。一些程序员开始测试类加载器是否存在以区别系统和非系统类 ,尤其把它当作做出安全决定过程的一部分。

为了向后兼容性， JDK1.2提供了由空类加载器加载的系统类 (或至少那些引导JVM所必需的类)。

系统类和空类加载器之间的联系与对待基于类加载器类型的类时的区别相联系，尽管把这些系统类细分为不同的包或 JAR文件然后分别给它们一组许可权，做到这一点非常困难。这种细分可以有效减少需要完全信任的代码数量，也可以减少代码中需要相信的数量。在 JDK1.2中，驻留在本地文件系统中的应用程序类现在必须作为扩展机制的一部分和非空类加载器一起加载。进一步来说，在新的安全结构下，一个和非空类加载器一起加载的类不会声明它的状况，这是由于该类可能已被赋予 AllPermission。经验告诉我们，如果所有系统类和一个特殊但非空的类加载器一起加载将会有助于设计的发展。

JDK1.0中没有改变的第二个特点是运行系统不会总是拥有一个安装的安全管理器，在这种情况下，对system.getSecurityManager()的调用会产生一个空的安全管理器 ,此外,为了向后兼容性,我们不会在JDK1.2中改变这个特点。然而，这种异常已经带来了一些不必要的复杂性。例如，无论你在哪里激活了一个安全核查，都必须对一个空的安全管理器进行测试，这就使代码变得杂乱。再者，程序员很快开始测试空的安全管理器以确定总体的状态，而不是试图在所谓的“ 宇宙大爆炸 ”之前或之后对其进行辨别。在安全管理器为空时，这会导致对虚拟机行为无保证假设，这部分是因为在一个空的安全管理器里没有激活安全核查。这些假设不应该在一个一般的水平上做出，不过很多程序员正在进行这项工作，这种假设的出现给向后兼容性带来压力。

JDK1.2中AccessController类的引入使在没有安全管理器的情况下激活安全核查成为可能 ，但由于担心破坏向后兼容性，必须小心部署这种核查。如果总是安装了安全管理器，会容易一些(也就是在引导过程之后)，尽管其行为超时后可能改变。

从这两个特点，我们得到的教训是不能轻易地发展空事物的界面设计 —— 并且肯定不能激活对空事物的方法调用。

China-pub.com
下载

China-pub.com
下载

第2章 Java语言的基本安全特点

自从Java技术出现以来[30、46]，人们对Java的安全性产生了极大的兴趣。部分因为Java语言从诞生之日起，安全性就被称为其重要设计目标之一。同时安全性也是把Java技术同其他技术区分开来的一个重要方面。

一项新技术的早期版本很少包括合理的安全特点。因此，Java作为Internet安全编程最佳平台地位的确立不但吸引了安全性专业人员，而且吸引了普通计算机公司的注意。长期从事安全研究的开发人员、学者以及学生已经总结出大量设计细节，并且用Sun Microsystems公司为此目的发行的JDK软件包编写出大量源代码。就连大众传媒也注意到了这个热潮。无论是《华尔街周刊》还是《纽约时报》都在显著位置加以报道。⊖

从Java技术提供者的角度来看，Java的安全性包括以下两个方面[23]：

- Java平台(主要通过JDK实现)是一个安全、现成的平台，在平台上可以以一种安全的方式实现基于Java的功能。
- Java所实现的安全工具和服务使得一些领域(如企业)对安全性广泛的需求得以满足。

Java技术的应用同时也提出了一系列有趣的问题，我们将在以后的章节讨论。本章主要侧重于Java语言和平台所提供的基本安全特点。

2.1 Java语言和平台

Java语言最初主要应用于嵌入式电子产品，如手持式设备和机顶盒。它是一般意义上的面向对象的编程语言并且非常简单，这样很多的程序员可以迅速掌握它。由于Java被特别设计成与平台无关，因此开发人员一旦用Java开发程序，就可以在Internet上的任何地方安全运行。Java同C和C++是有联系的，但又有许多区别，它包括了很多C和C++忽略的方面并且借鉴了其他语言的一些特点。

Java语言是很严格的，它没有任何不安全的结构，例如没有索引核查的数组访问，因为这类结构往往会导致不定的、不可预测的程序操作⊜。Java提供了一种主要由废物收集器实现的自动存储管理机制。Java语言还进一步避免了那些由内存重新分配命令(如C的“free”或C++的“delete”)所带来的问题。

Java程序一般被编译成二进制代码指令组，并且其二进制格式由JVM规范[30]定义。为了对编程提供更全面的支持，Java也定义了大量的包。Java程序通常以二进制文件的形式存储，这些文件表示编译过的类和界面。二进制类文件被加载入JVM，然后连接、初始化、执行，以下是一个简单的Java程序的例子。

```
class Test {
   public static void main(String[] args) {
      for (int i = 0; i < args.length; i++)
         System.out.print(i == 0 ? args[i] : " " + args[i]);
      System.out.println();
   }
}
```

⊖ 参见[49]

⊜ 近来的研究表明，CERT发布的警告大约有50%的是由缓冲器溢出造成的。

China-pub.com
下载

在SUN工作站上运行Solaris操作系统，存储在文 件Test.java的Test类可以由以下命令编译并执行：

```
javac Test.java
java Test Hello
```

程序将会打印出Hello。

Java平台以网络为中心，自它诞生之日起就包含这样一种思想：即同一软件应可以在不同类型的计算机、用户附件以及其他装置 (例如智能卡)上运行。采用Java技术，你可以在不同的系统上，如运行 Solaris操作系统的Sun工作站，或者是运行 MS-Windows的PC机，或者是Macintosh计算机、网络计算机，甚至是蜂窝电话或 Internet可视电话上实现同一功能。

最初的HotJava浏览器通过把Java程序嵌入HTML网页来演示其强大功能，我们称这些程序为应用小程序 (applet)，它们被透明的下载并在浏览器上运行。 Java平台已被并入所有主要的Web浏览器中，不久将嵌入新一代电话及电视机顶盒中。Java程序没有浏览器也可直接运行，并且可以在服务器和大型主机上编写并运行。

Java平台包括Java语言、JVM和应用程序界面 (API库)。JVM是一部抽象的计算机器，它不承担任何特殊的实现技术，也不主宰平台，它对 Java编程语言也一无所知。 JVM仅知道特定的文件格式 —— 类文件格式(class file format)。一个类文件包括JVM指令(或字节代码)、一张符号表和附加信息。字节代码可由本地平台解释或编译。 JVM可以以微码形式实现或直接安装在硅片中。JDK1.2中包括的JVM功能最新版本超过了基于Win32和Solaris平台的JVM。

2.2 基本安全结构

在早期的Java版本中，基本安全结构主要围绕着这一问题而展开，即如何使用户能动态地调入和运行Java applet，而用户系统不用冒多余的风险不太严格的说，任何不驻留本地系统，只有下载才能运行的代码即 applet。驻留本地系统的代码一般称为 Java应用程序(Application)。由于applet往往是在你不注意的情况下动态下载，并且你也不知道这个 applet的作者是谁，因此不能盲目地验证该程序不会造成损害，这样一来下载 applet的操作应严格局限于沙盒(sandbox)中 —— Web浏览器特别为applet分配的区域。applet可以在沙盒中运行而不会超出界限之外。例如，applet不能读取或修改存储在用户系统上的文件。在某种情况下，如果用户偶然引入了一个敌意的 applet，那么这个applet不能破坏用户的系统。因此这种沙盒模式提供了受限的环境，在此环境中可以运行从开放网络中获得的不能完全信任的代码 (即applet)。

在沙盒模式中，所有独立的应用程序 (与applet对应)都可以完全信任，它们可以充分接触关键的系统资源(例如文件系统)，要保证安全就要对系统进行物理控制，例如阻止终端用户安装可疑的软件。注意applet和独立应用程序的区别，或者说“ 内部 ”和“ 外部 ”之间的区别并不总是绝对的。对于一个基于网络的文件系统来说，一个看起来驻留本地系统的类文件可能位于几千英里之外，而一个从局域网 (LAN)上下载的 applet可能来源于用户正在运行这个 applet 的同一主机。

沙盒模式通过JDX1.0.x投入应用，一般由在JDK下的应用程序采用，而JDK包含有Java使能的Web浏览器。

通过一些新的机制，早期的基本安全结构得到了加强。首先， Java语言被设计成类型安全的，并且易于应用，这样相对于其他可能使用的编程语言 (如C和C++)，程序员不太可能犯

China-pub.com 下载

小的错误。Java语言的一些特点(如自动内存管理、废物收集、串组范围核查)是语言如何帮助程序员写出更安全代码的范例。

其次，字节代码验证器保证只有合法的代码才可以执行。编译器把 Java程序转换成与机器无关的字节代码。在新下载的 applet运行前，字节代码验证器被激活，以便核查 applet是否符合Java语言的规范，以及有没有违反Java语言规则或名字空间限制。这是因为基于安全的原因，JVM对类文件中的代码作了严格的格式和结构限制。验证器也核查是否违反内存管理规定、堆栈下溢或溢出、以及非法数据类型的丢弃。这些问题可能使一个怀有敌意的 applet对安全机制造成部分的破坏，或者 applet用自身的代码在运行其间替换系统的一部分。字节代码验证器和JVM一起用来保证运行其间语言类型的安全。例如，在数组中存储引用时， Java用运行时间类型核查来保证彻底的类型安全。

再者，运行行为实际上包括很多环节，如加载并连接执行程序时所需的类、可选机器代码的产生以及程序的动态优化，另外还有程序的实际执行。在运行期间，类加载器定义了本地名称空间以便保证不可信的 applet不能影响其他Java程序的运行。

最后，访问关键系统资源的行为由 JVM协调，并且预先要经过安全管理类的核查以便把不可信代码的影响限制到最低限度。 3.8节中将详细讨论类加载器和安全管理类。

2.3 字节代码验证和类型安全

这一节来看看类型安全的一般问题，尤其是字节代码的验证，这是由于这个问题已成为一些公开发现的潜在的安全漏洞的核心。 2.4节将会讨论一些具体的错误以及如何修复它们。

尽管值得信赖的编译器可以保证 Java源代码不会违反安全规则，但一些人仍可利用编译器产生违反安全规则的代码。 Java的Web浏览器可以从任何地方引入代码段，而不管这些代码段是否来自于可以信赖的编译器。因此在运行任何代码段前，运行系统应对它进行一系列的测试。

测试范围从验证代码段格式是否正确到使代码段通过一个简单的定理校准以使代码满足某种规则。一般来说，核查代码是为了满足以下几点：

- 代码不能假造指针。
- 代码不能违反访问许可权。例如，外部对象不能访问私有域。
- 代码严格按照规则访问对象 (例如测试保证InputStream 对象只能用做InputStream类，而不能用于其他)。
- 代码调用有正确类型的参数的方法，并且没有堆栈溢出。
- 不能做非法的数据转换，如把整数类型转换为指针类型。

从严格意义上讲，对于保证类型安全 ,静态字节码验证器并不总是必须的，这是因为 JVM在理论上可以在运行期间进行彻底的类型核查。然而，运行期间的核查往往会显著降低程序执行速度，这是由于这类核查不得不重复进行，因此把一些核查提前到类加载时间去做，这样这些核查可以只进行一次，这似乎是一种有吸引力的策略。任何下载的代码都满足这些性质，由于运行系统不必核查代码，这样运行速度可以更快。由于验证器独立于 Java语言或编译器，因此它也可以核查有非Java源语言派生的字节代码。

前面所提到几点显然并不全面，也不意味着是完全准确的。由于篇幅所限不能对许多问题作更细致的描述，例如 Java语言设计的细节、 JVM内部工作机制、流分析的背景以及定理

China-pub.com
下载

证明的艺术，而这些对于透彻理解类型安全如何实现来说是必要的背景知识。

就目前的讨论来说，理解以下的要点已经足够了。 Java安全结构的最重要的目的是保证正确的Java语言规范和JVM规范。为了理解Java安全结构，我们假设你正在编写一个日历应用程序。你通常会建立一个界面希望用户输入 1到12之间的数字来代表一年中的每一个月，你也需要初始化界面促使用户输入当前日期。由于其他界面假设月份整数在 1到12之间，因此在内部初始化过程中核查和验证用户输入是否是有效数字，这是谨慎的做法。如果不作核查，结果将导致对无效数字不加以拒绝，那样你的日历应用程序在输入超出范围的月份数字时就不能正常工作，并且可能运行异常。

同样的原则也使用于Java平台。 JVM希望它所运行的字节代码有特定的性质，而字节代码验证器的工作就是保证这一点的实现。这些工作完全由字节代码验证器来做非常困难或者说不可能，因此JVM也可以做附加的特性核查。保证字节安全没有什么神秘的，实际上就是堆积如山的细节和大量的工作。

你或许会问类型安全会给计算机的安全带来什么。我们可以从几个方面来回答着个问题。首先，类型安全有利于程序的正确执行。如果由于程序不能正确执行，实现安全功能的程序不能如同预想的那样完成它的功能，也就不能提供正确的安全保证。例如，安全决断可以体现在下列形式的等同性测试上：

```
if(the name is James Gosling){
   open the door to Hacker Lounge
}else{
   throw the person out
}
```

在这个程序中，安全提问是用 Java语言书写并执行的。这样的话，把一个像‘ James Gosling ’这样的串同一个不同的串“ Scott McNealy ”作比较，得到肯定回答是不可能的。能做到这一点显得很重要。否则，串比较时的小小错误就会导致安全上的漏洞。

另一方面，并非所有的类型安全问题都会不可避免地导致对安全的破坏，阐明这一点很重要。例如，虚拟机功能犯了一个小错误 —— 把串acegikmoqsuwy 等同于串bdfhjlnprtvxz，安全性对此作出的反应不会马上就很清楚。不过，必须密切注意类型安全问题，不给任何问题发生的机会。

Yellin早期的论文就包括了验证器的细节和其他类型的核查机制 [82]。然而，为了充分消化吸收这些知识，必须充分理解字节代码指令。 Liang Bracha最近提出了一种新的机制，这种机制可以在JDK1.2中实现，它解决了涉及动态类加载的类型安全核查问题 [45]。随着JDK开发组、实验室和各大学工作的展开，字节代码验证这一课题正在不断发展之中。我们期待语言类型安全这一课题将来会变得更正式、更精确。

2.4 签名应用小程序

JDK1.1介绍了签名应用小程序(signed applet)的概念。回忆前面讲述的沙盒模式中，所有的远程代码(即所有的applet)都被自动认为是不可信的，并且限制在沙盒模式中运行。尽管这种限制有利于一个安全的计算环境，但对某些 applet却太苛刻了。例如，一家公司在 LAN上设置了一个用来存放雇员薪金的 applet，一个雇员下载运行这个 applet来改变他的计划安排，他当然希望applet可以自动更新存放在他个人目录下的帐目。

China-pub.com 下载

为了实现这些功能，JDK1.1增加了支持数字签名的功能，这样在一个applet的类文件开发结束后就可以和类文件的签名一起标记并存储在JAR(Java Archive)格式中。在每一次安装过程中，可以详细指明哪一个签名者(或公钥)是可信的。这样当一个数字签名正确的applet下载时，applet的签名者被验证并验证为可信，那么这个applet就可以像本地代码一样自由访问系统了(见图2-1)。

无论是早期的沙盒模式还是可信applet模式，在JDK1.2(实现了基于安全策略和许可权的细致的访问控制)中都已扩展成为一种新的安全结构。

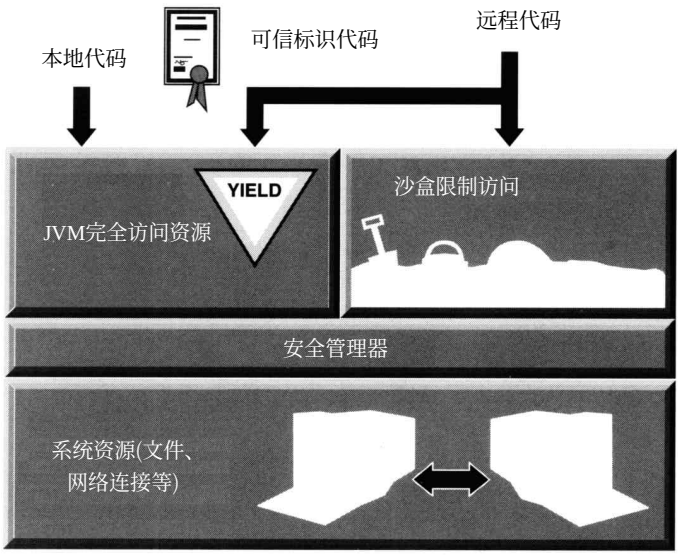


图2-1 JDK 1.1安全模型

2.5 关于安全错误及其修复的简要历史

人们常说，忘记过去意味着背叛。为了对JDK1.2的新的安全结构有全面的理解，必需了解一系列与Java安全有关的错误——这些错误由谁带来以及如何修复。JavaSoft把下列回顾放到了它的公共站点中。所有已知的错误一般在发现不久就被修复，JDK的下一版本将包括这些修复。从技术角度来说，错误和修复是模糊不清的，因此这里不作详细解释。McGraw和Felten[49]提供了额外的背景，例如媒体报道、股票市场的运作，这些可能会与错误报告有关。

有必要强调一下，对于这里讨论的所有错误来说，问题在于错误是在安全模式实现过程中产生的，而非模式本身。这样的话，关键的“诊断程序”并非是必须的，而且修复会变得很快。根据Sun的“通过开放获得安全”这一策略，源代码应是公众可以利用的，这一策略吸引了许多有能力的人在Java安全性研究上投入了时间和精力。这项政策使得安全错误可以被发现而且很快就可修复，应该说是很成功的。

- 1996年2月。普林斯顿大学的Drew Dean、Ed Felten和DanWallach(以下称为普林斯顿研究组)描述了这样一种攻击，这种攻击利用JDK1.0中的applet安全管理器，使用DNS作为主机名到IP地址解析这一特性。这种攻击叫做DNS欺骗攻击(DNS spoofing attack)。Steven Gibbons也独立提出了这种攻击的过程。

China-pub.com
下载

这种攻击利用了DNS允许一个主机可同多个IP地址匹配这一事实。这样的话，心怀不轨的家伙可以通过控制服务器错误地赋予攻击主机它的实际 IP地址和一个虚假地址，而这个虚假地址属于目标机器。 applet安全管理器允许一个 applet同任何地址相连接，这个地址同计算机名紧密相连，这种攻击充分利用了这个弱点。这个来自于攻击主机的 applet可开放一个到虚假地址的连接并且连接到目标机器上，即使目标机器并没有意识到。

applet安全管理器的修复主要是严格确定 applet可以连接到哪一个计算机上，尤其注意applet来源的实际IP地址，这样就可以允许applet只能连接到相同的数字地址上。

- 1996年3月。牛津大学的DavidHopwood在JDK1.0类加载器中发现了一个错误，它利用类加载器加载非法字节代码，非法代码根据绝对路径加载一个类，这意味着如果上载了一个FTP，攻击者可以用已知路径名在目标系统上安装一个敌意的类文件，然后攻击applet将可以加载恶意的类文件。
- 1996年3月。普林斯顿研究组在 JDK1.0.1中的字节代码验证器中发现了一个执行错误。攻击过程很复杂，它可以利用这个错误删除一个文件或者造成其他破坏。
- 1996年4月。来自于Sprint 的一个软件工程师在JDK1.0.1中发现了一个安全问题，并把它报告给了JavaSoft。对一个特殊的有防火墙保护的网络配置来说，由防火墙内部的客户下载的外部的applet可以同防火墙内部的单一的特定主机相连，这是由位于 URL的名字解析代码错误造成的。要想攻击成功，目标网络和攻击者网络应有相同的域名，且攻击者的域必须是正式的(InterNIC)、注册的网络。换句话说，目标网络必须使用 InterNIC还没有注册的内部名，并且攻击者必须已经控制了 InterNIC注册的名字。
- 1996年5月。普林斯顿研究组在 JDK1.0.1中找到了另一种利用创建类加载器来通过系统限制的办法(3.8节将会论述类加载器是一个敏感的、功能强大的对象，因此你必须严格控制创建类加载器实例的人选)。这种攻击建立在由Tom Cargill所做的早期工作上。
- 1996年6月。David Hopwood在JDK1.0.2中发现了一个基于非法类型丢弃的错误，这个错误可以操纵对象的分配和对象的合并。

1997年3月。JavaSoft的工程师们在实现字节代码验证器时遇到了一个错误。为了研究这个错误，有人不得不编制了特殊格式的字节代码。攻击在理论上非常复杂，并且似乎难以实现。

- 1997年3月。据报道 ，某个applet可以调用一个名为getlocalHost()的方法来确定当前运行该applet 的IP地址。尽管连媒体都对这条消息表现出极大的兴趣，最后证明这实际上是虚惊一场。自1996年5月JDK1.0.2发布以来，名为getlocalHost()的applet可以回送主机地址("localhost/127.0.0.1")，对本地计算机来说这只是普通的操作，不会泄漏任何私人信息。
- 1997年4月。普林斯顿研究组(现在增加了一个新成员 Dirk Balfanz)在JDK1.1.1中发现了一个错误，该错误利用了数字签名处理代码可以操纵有类文件签名权的签名者的身份鉴别这一特性，这种攻击利用数字签名代码仿冒，使代码像是由 Java运行系统所记录的签名列表上的签名者所签的。
- 1997年5月。华盛顿大学的Brian Bershad、Sean McDirmid、Emin Gun Sirer(以下简称华盛顿研究组)在JDK1.1.1验证器中发现了一个错误。验证器没有核查到用来进行方法调用的参数数量少于为此方法的本地变量分配的空间的数量。结果，参数数量过多可以导致堆栈溢出，甚至有可能导致JVM的崩溃。虽然还没有发现基于这种错误的攻击，但由

China-pub.com 下载

于这种错误与类加载有关，因而必须立即修复。

- 1997年6月。华盛顿研究组报道了JDK1.1.2字节代码验证器中的另一执行错误。这个错误可以使一个非安全类型的applet运行，搜索并定位存储在浏览器地址空间的串。
- 1999年3月。德国Marburg大学的研究生Karsten Sohr在字节代码验证器的执行过程中发现了另一错误，这个错误允许一个不可信的applet越权运行,对JDK1.1.x和JDK1.2造成影响。

现在来做一个回顾，通过更细致、更周全的设计规范和质量保证措施，这些错误完全可能尽早发现。

总之，有必要注意以下几点。首先，关于沙盒安全模式实际上存在着许多混淆。例如，我们不允许applet进行某些操作。但实际上，受限操作的名单是无穷尽的，并且不可能用精确的语言来定义。这样的话，在某些错误是安全错误这个问题上偶尔也会有争论。

其次，并非所有的错误都是安全错误，尽管从潜在意义上讲，它们都是安全错误。一方面，应该严肃地对待这些错误。另一方面，应该对这个问题有正确的理解。

再次，严格意义上说，并非所有的错误是Java安全错误，这是由于这些错误同计算环境的其他方面是相互作用的。例如操作系统、Web浏览器、Web spoofing以及其他安装的软件。事实上许多媒体所关注的错误同Java技术没有任何关系，但仅仅因为这些错误同Internet有关，在没有发现这些问题的真正原因之前，一些观察者一般把这些警告归于潜在的Java安全问题。一些人把ActiveX同Java混淆，而另一些人错误地认为Java Script同Java是一回事。

最后，目前的Java安全技术还没有做到操纵和控制由applet和应用程序所带来的资源消耗。例如，自动区分要用很长时间解压的MPEG解压器和蓄意浪费资源的一个敌意的applet是很困难的。注意，消费资源的攻击与使用Java并无太大关系，有时也不需要复杂的编程。例如，一个恶意的网络服务器可以提供一个无限大的网页，这样会把用户缓冲空间的内存占满，而一个邮寄者可以发一个巨大的电子邮件把用户的邮箱塞满。不过，Sun承认在某些情况下，希望控制拒绝服务攻击造成的影响，并且正在这一领域进行积极的研究。