 Java语言拥有三大特征：平台无关性、网络移动性和安全性，而Java体系结构对这三大特征提供了强大的支持和保证，本文着重介绍Java体系结构对支持信息安全的原理和使用方法。

**Java体系结构**

    首先Java的源代码Java文件由编译器编译成 Java的二进制字节码class文件，然后class文件由Java虚拟机中的类装载器进行加载，同时类装载器还会加载Java的原始 API Class文件，类加载器主要负责加载、连接和初始化这些class文件以后，就交给虚拟机中的执行引擎运行，执行引擎将class文件中的Java指令解释成具体的本地操作系统方法来执行,而安全管理器将在执行过程中根据设置的安全策略控制指令对外部资源的访问。

    Java的执行方式不是编译执行而是解释执行，不同平台上面相同的源代码编译成符合Java规范的相同的二进制字节码，然后再交给支持各自平台的虚拟机去解释执行，"先编译，后解释，再执行"三步走的方式使得Java实现了"一次编写，到处运行"，如果Java应用使用的是100％标准Java API并且没有直接调用本地方法，那就可以不加修改地运用在多种平台上，这样的平台无关性使得在异构的网络环境或者嵌入式方面的应用更方便和现实。 Java的网络移动性带来了一种全新的软件模式，在分布式处理模式的基础之上，可以将软件和数据通过网络传送到客户端去，这样确保了客户端有必备的软件来浏览和操纵通过网络传输的数据，Java体系结构支持把单一的执行文件切割成小的二进制字节码文件Class文件，而这些文件可以按照应用的需要动态连接、动态扩展。Java体系结构对安全性的支持主要是通过Java语言本身安全性、虚拟机的类加载器和安全管理器以及Java提供的安全API几个方面来实现：防止恶意程序的攻击，程序不能破坏用户计算机环境；防止入侵，程序不能获取主机或所在内网的保密信息；鉴别，验证程序提供者和使用者的身份；加密，对传输交换的数据进行加密，或者给持久化的数据进行加密；验证,对操作设置规则并且进行验证。

**Java信息安全的必要性**

    随着互联网应用越来越广泛，并且互联网其本身独特的资源共享性，因此能够按照用户需求及时准确获得信息和处理信息的应用对用户而言就相当重要，这也是Java得以迅速发展和被广泛接受的原因。但同时网络也提供了一条攻击接入计算机的潜在途径，特别是当用户下载网络软件在本地运行，这就要求Java能够对病毒/木马的问题加以防范，对信息以及本地环境进行保护。比如我们浏览一个网页的时候，网页上的Applet可能会自动下载并且运行，而这个Applet完全有可能来自不可靠的地方，又或者我们使用通过 JINI服务查找到的网络上不可靠的服务对象来获得服务，如果没有Java体系结构提供的安全机制，这就很有可能引入了一个怀有敌意的程序造成信息丢失、资料泄密、相信伪造数据和修改本地计算机安全设置等等后果，带来未知的严重后果。

**Java语言本身安全性**

    Java语言的设计者们是在C++的基础上设计出来 Java的，因此与C++相比它的语法更加简单清晰，结构、单元、运算符重载、虚拟基础类等在Java中都没有采用，并且取消了多重继承而采用实现多个接口的方式。这样能降低开发人员犯错误的几率，帮助他们写出更安全的代码。

    Java中去除了C++语言中的令人费解、容易出错的"指针"，用列表、堆、哈希表等结构来代替，避免了任何不安全的结构。Java也没有索引核查的数组访问，因为这往往会导致不定的、不可预测的程序操作，它所有的数组访问都必须先检查是否越界。Java要求所有的变量在初始化以前不能使用，对于基本数据类型变量都会自动地赋给某个初始值，避免了未初始化变量获取内存信息。所有这些都使得程序不能访问任意的内存地址，对于内存中的实体信息只能通过有权限的对象进行访问，而不会出现象C++那样把类型指针强制转换成内存的指针，然后通过内存查找的方法找到私有的变量。

    Java分配内存对于开发人员来说是透明的，开发人员使用new方法新建对象,这时候虚拟机就会从堆内存中找到合适的内存空间，开发人员不需要也不能够进行干预。而对于内存的回收，Java避免了开发人员明确干预对象的回收，比如C的free或C++的delete命令，避免了开发人员无意间对内存的破坏。Java采用虚拟机的"垃圾回收"机制来实现的内存自动管理，释放不再被使用的内存资源，内存回收器就像一台垃圾收集车，但是和我们在大街上看到的收集车，仅仅收集大家放在垃圾桶里面的垃圾不同的是，它还要到你家里去帮你找出那些东西是不要用的垃圾，然后把这些东西拿走，最后还要整理家里的空间，腾出最大的空间让你放新东西。Java的内存回收器目的就是找到不再引用的对象，释放内存空间，并且需要整理内存的碎片空间，尽量避免出现"内存不足"的情况。

    对于在网络中交换的序列化对象很容易在重建对象的时候访问到对象的私有信息，这时候Java提供了两种办法来保护信息，一种就是采用给变量加上transient关键字的方法，这样对象序列化的时候就不会读写该变量，另一种就是在实现Externalizable接口而不是Serizlizable接口，这样对象就只能通过writeExternal和 readExternal方法来保存和重建，其他方法无法进行了。

    以上这些都是Java语言本身对信息安全提供的基础。

**类加载器**

    虽然名字叫类加载器，但是实际上Java虚拟机中的类加载器不光要负责加载而且要负责连接和初始化应用程序需要用到的Java类型。加载就是把二进制形式的字节码读入虚拟机中，而连接就是给这个已经读入的类型分配类变量内存以及把类型中用到常量池中的符号转换为直接引用，最后的初始化过程就是赋给类型变量合适的初始值。

    类加载器为加载的类提供了不同的命名空间，统一源代码生成的字节码被加载到同一个命名空间中，相同命名空间不能加载类名相同的类，同一个命名空间内的类可以直接进行交互，而不同的命名空间的类是不能交互的，除非显式地提供了交互机制，通过命名空间和类成员访问权限的设置保护了被信任的类边界。

    类加载器分成了启动类加载器、标准扩展类加载器、路径类加载器和网络类加载器四种。启动类加载器从本地系统中加载原始的Java API类，用来启动Java虚拟机，而其他三种加载器是在运行时加载用户定义的类，标准扩展类加载器加载的是不同虚拟机提供商扩展的标准Java类，而在 classpath中的类由路径类加载器来加载，网络类加载器加载通过网络下载得到的类文件，每一种加载器在加载类的时候都会建立一个加载器实例。类加载器采用双亲委派链模式（这个模式很类似GOF在《设计模式》一书中提到的责任链模式）除了启动类加载器以外，每个类加载器都有自己的"双亲"。一个类可以通过有三种方法定义自己的双亲：第一种通过引用，比如A类中引用了B类（即A和B有关联关系），那么B类的加载器就会作为A类的加载器的"双亲"，早于A 类加载；第二种使用loadClass方法来自定义"双亲"，这时被load的类的"双亲"即本身这个类加载器；第三种在没有采用前两种的情况下使用的默认方式，默认把启动类加载器作为"双亲"。

    在加载过程中，当发出加载请求的时候，加载器首先询问它的"双亲"――路径类加载器――来查找并加载这个类，而这个加载器也向它的"双亲"请求加载，一层一层请求上去，直到启动加载器获得请求，来查找并加载这个类，如果这个类没有被加载并且查找不到，返回结果给它的子加载器，由子加载器加载，直到请求返回给原来的加载器，这时还没有加载成功的话，由网络类加载器试图从网络中寻找并下载，如果还不成功将抛出NoClassDefFoundError异常。

    这个过程保证了启动类加载器可以抢在标准扩展类加载器之前加载类，而标准扩展类加载器又可以抢在路径类加载器之前加载类，最后才由网络类加载器加载。比如应用被试图加载一个带有恶意代码的 java.lang.String类，因为它本来是Java API的一部分，它们加载到的命名空间可以得到被信任类的特殊访问权限，但是由于启动类加载器是最早被加载的，所以java.lang.String只会被启动类从Java原始的API中加载，而带有恶意代码的java.lang.String类不会被加载进来，这样有效的保护了被信任的类边界。

    类加载器中还包括了一个类型检查的功能模块，它负责保证程序的健壮性，它在类型的生命周期中要进行四次检查。第一次检查是在加载的时候，主要检查二进制字节码的结构，首先格式要满足Java语言定义的规范，然后要保证将要加载的类字节码是一组合法的Java指令。第二次检查是在连接的时候，主要是类型数据的语义检查，保证字节码在编译时候遵守了规范，比如对final类不会派生出子类，也不会重载final的方法；每个类只有一个超类；没有把基本数据类型强制转换成其他数据类型。第三次检查也是在连接的时候，关注于指令的结构，保证指令的操作数类型和值正确，操作数堆栈不会出现上溢出或者下溢出。最后一次检查在动态连接的时候，主要检查类型中的符号引用被解析时是否正确。以上的问题都会产生恶意的行为所以必须在运行前进行检查，而一部分检查工作会在虚拟机运行字节码的时候检查，比如数组越界，对象类型的转换等等，一旦检查发现了问题就会抛出异常，使得程序不被执行。

    类加载器避免了出现某些怀有敌意的人编写自己的 Java类，而这些类方法中含有跳转到方法之外的指令，导致虚拟机的崩溃和保密信息被获取的可能，保证了程序的健壮性，也不会出现替代原有Java API类的恶意代码被运行的情况，并且类加载器防止了恶意代码去干涉善意的代码，守护了被信任的API类库边界，确保了代码可以进行的操作。

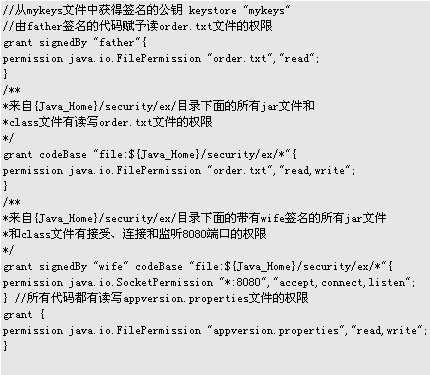
**安全管理器**

    安全管理器为Java虚拟机的环境建立了一个起到保护作用的"沙箱"，这个"沙箱"为程序提供了一个限制了应用的操作运行环境，保护了虚拟机外部的资源不会被虚拟机内部的恶意代码破坏。安全管理器的安全策略可以灵活的建立细粒度的访问控制策略，将不同的资源访问权限授予不同的代码单元，这也是Java体系结构安全模型的最大特点和优势之一。

    首先在赋予权限前我们要明确代码单元的来源，签名担保可以使用户确认文件的来源，并且这些文件在本地虚拟机加载前没有被修改，只有保证了源代码来源的可靠性，才能分配相应的操作权限。对class文件或者 jar文件的签名可以用jdk中的jarsigner这个工具。它首先对根据文件内容进行单向散列计算，产生一个散列值，然后把这个散列加到文件后面作为文件的一部分传递给用户，用户收到文件后重新生成一次散列值，然后跟文件后部的散列值进行比较，如果这两个散列值相同说明文件内容没有被改动，虽然不同的文件内容可能生成相同的散列值，但是一般Java采用的是124位散列，这个长度想要从一个不同的输入产生一个已知散列值的计算是不可行的。仅仅通过散列值是没有办法保证代码的来源的，因为怀有恶意的人完全可以把整个文件和散列值都替换掉，为了防止这种事情发生，必须在发送前用私钥对散列值进行加密，为什么不对整个文件进行加密呢，因为jarsigner默认采用的是DES算法，加密本身也是一个费时的过程，我们的目的只是为了保证文件的来源而不是文件本身的保密性，所以只需要对散列值进行加密，用户收到文件以后用文件提供者的公钥进行解密来验证散列值没有被替换。

    明确了代码来源，并且保证没有被修改以后，就可以给它分配操作权限。安全策略是一个java.sercurity.Policy的实现类，Policy中主要包含了代码来源和相应的权限的对应关系信息，安全管理器的check方法将根据Policy对象判断给导入的代码赋予什么样的操作权限。代码来源是由java.security.CodeSource 表示的，这个对象中包含了一组Certificate对象，说明了为这个代码文件担保的签名。而权限是用 java.sercurity.Permission的子类实例来表示的，每一个Permission有三个属性：类型、名称和可进行的操作，类型说明了权限控制的资源类型，比如FilePermission说明了是对文件操作的权限，名称说明了这个权限控制的对象，比如FilePermission的名称为"/mydoc/order.txt",说明了对文件order.txt的操作权限，可进行的操作属性说明了这个权限可以进行的操作，比如 "read"说明这个FilePermission可以对order.txt进行读操作。

    开发人员可以通过编写代码继承SecurityManager类建立自己的安全管理器来设置安全策略，更方便和常用的方法是通过设置策略文件来实现。这是一个策略文件的例子：



    如果一个应用程序没有指定启动安全管理器进行干预，它的所有操作不会受到管理器的限制，指定启动安全管理器有两种方式，一种是在运行应用程序的时候指定，一种是在代码中指定。

    java -Djava.serciruty.manager myapplication

    这里指定了启动默认安全管理器，这时候将根据Java默认的全局策略文件来设置安全策略，这个文件是

    /JAVA\_HOME/lib/security/java.policy。

    java -Djava.serciruty.manager －Djava.serciruty.policy＝ myapplication

    这里启动安全管理器，除了默认的全局策略文件外，也会按照Djava.serciruty.policy参数指定的策略文件来设置安全策略，文件指定可以用URL也可以直接用文件名称。

    如果是在代码中指定，程序通过 setSecurityManager方法设置一个java.lang.SecurityManager的实例时候，这时候安全管理器就会开始控制应用程度的操作了。这时当类加载器加载类到虚拟机的时候，安全管理器根据代码来源和签名找到相应的权限，然后建立这个类的保护域 ProtectionDomain，保护域定义了这个类所有的权限。当应用程序用到这个类并要进行任何可能的不安全操作时，都会向安全管理器请求操作许可，安全管理器中有一系列的check方法来检查操作是否可行，比如checkRead方法会检查是否能够读取某个文件，checkWrite方法检查是否能够写入某个文件，管理器根据操作请求类型调用相关的check方法，check方法根据类的保护域判断操作是否可行，如果操作没有权限的话将抛出一个异常，操作收到这个异常后将不能执行，如果操作可行的话，check方法就顺利返回，操作继续执行。

    以上签名验证、定义安全策略以及安全管理器检查操作权限一系列过程，有效的保护了被Java应用使用的外部资源的安全性。要注意的是而由启动类加载器加载的核心API类是不受这个安全管理器的权限控制，这也是为什么要用类加载器防止核心API被恶意代码替换的原因。

**Java体系结构提供的安全API**

    Java体系结构提供了三类主要的安全API： Java 认证和授权服务（Java Authentication and Authorization Service，JAAS）、Java 安全套接字扩展（Java Secure Socket Extension，JSSE）和 Java 加密扩展（Java Cryptography Extension，JCE）。前面已经提到了安全管理器中用到的JAAS，这一节我们提到的是API主要指Java提供的对信息进行加密的Java加密扩展包JCE和Java安全套接字扩展包JSSE, 这些API提供了加密算法可以按照我们的需要，实现对任意数据的加密和解密。

    Java加密扩展包JCE提供的是基于密钥的加密，它通过javax.crypto.Cipher类来实现数据加密合解密，加密解密的对象可以使程序中的数组对象，也可以是通过Java流接口读出或者写入的数据。使用Cipher类加密可以选择多种加密算法、加密模式和填补机制。加密算法JCE中提供了DES、多重DES、 PBEWithMD5AndDES、RSA和Blowfish等等。DES是很多机构组织采用的数据加密标准；而多重DES使用多个DES密码进行多长 DES加密，加大了攻击的难度但是也增加了加密解密过程说花的时间；PBEWithMD5AndDES前面提到过，主要是计算散列，然后对散列进行DES 加密，来实现签名认证；RSA算法是1978年公布的一种分组加密算法，也是现在应用得最广泛的公钥密钥算法；Blowfish是由Bruce Schneier公布的一种加密算法，没有申请专利，并且公布了实现的代码，它适合不需要经常更换密钥的情况。

    选择加密模式是为了对加密数据做进一步调整，从而增加解密难度，模式还可以将分组明文作为流明文进行处理，减少每次处理的数据量，JCE中提供了电子密码本模式ECB、密码封装链接模式CBC、密码反馈模式CFB和输出反馈模式OFB。电子密码本模式ECB是最简单的一种模式，只是对明文每8个字节进行分组，并且一次完成整个明文分组的加密，它适合对二进制数据流进行加密；密码封装链接模式CBC，把一个8字节的分组作为输入数据对另一组加密结果进行修正，这样可以把明文中包含的数据类型进行隐藏，这种模式可以对文本数据进行加密；密码反馈模式CFB，类似于CBC模式只是实现稍有不同，不同的是CBC模式需要明文分组来进行修正，而CFB需要的数据量小一些，并且长度可以进行调整;输出反馈模式OFB，适合那种加密数据在传输过程中有可能产生变化的情况，产生变化出错只会对这一位产生影响而不会影响整个分组。JCE可以选择两种填补机制PCKS5Padding和NoPadding，前者将对明文进行分组时候，如果字节数不够8的倍数就进行填充，保证数据的长度为一个完整的分组，而后者不对数据进行填充，当明文进行分组时候，字节数不够8的倍数会抛出一个异常。

    Java安全套接字扩展包JSSE提供的是基于套接字之间传输的数据进行加密，它与JCE最大的不同就是数据的加密过程和传输过程是不分离的，如果说JAAS让我们可以识别应用程序提供者并限制他们只能访问授权使用的那部分系统，那么JSSE保证了我们应用程序传输的数据安全性。JSSE实现了SSL（安全套接字层）的加密，SSL作为HTTPS协议的基础，提供了在TCP套接字上对数据进行加密的方法，也是基于WEB应用最常用的一种加密方式。使用JSSE API首先我们需要建立SSL环境，SSL服务器端建立密钥库存放服务器私钥和验证身份的证书，而SSL客户端建立信任库验证信任的证书，密钥库和信任库都是通过JDK中keytool这个工具来进行管理；其次，我们需要从一个 JSSE 套接字工厂而不是直接从 java.net.Socket 类获得套接字，客户端代码从 SSLSocketFactory 获取套接字，而服务器端代码从 SSLServerSocketFactory 获取套接字；通过从这些工厂获取套接字，我们就可以利用 JSSE 提供程序提供的框架，而不是像 java.net 包允许我们所作的那样，简单地创建标准的、不安全的套接字。使用 JSSE，我们可以定义运行任意应用程序协议--包括 HTTP、TCP/IP、FTP，甚至 Telnet--的客户机与服务器之间的安全套接字连接。

**总结：**

    信息的安全性是计算机领域必须重视和解决的问题， Java体系结构对信息安全的提供灵活而健壮框架，只要我们使用得当就能够很好的保证信息安全性，降低我们的代价和风险，同时我们也要加强一些其他相关的安全工作，比如保护好我们的私钥等等，这样才能保证Java安全框架发挥最大的作用。Java安全框架还有一些不足的地方，比如应用程序不断分配内存或者新建线程造成拒绝服务、将安全模型与系统用户进行映射等等，随着信息技术的不断发展，信息安全也会面临越来越大的挑战，这些都需要Java安全框架更加完善和进一步发展。