互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

第 3 章 Java加密利器

通过对第1章和第2章内容的学习，我们已经对密码学的理论有了一定的认识。那么，该如何将如此深奥却又如此关键的技术运用到我们的应用平台中呢？先别急，“工欲善其事，必先利其器”，先看看我们手里都有哪些工具可以驾驭这枚银弹！

从本章开始，我们将进入本书的主题——Java加密与解密的艺术。

3.1 Java与密码学

离开了安全问题，密码学就没有存在的价值。因此，在Java的世界里，密码学是其安全模块的重要组成部分。

3.1.1 Java安全领域组成部分

Java安全领域总共分为4个部分：JCA（Java Cryptography Architecture，Java加密体系结构）、JCE（Java Cryptography Extension，Java加密扩展包）、JSSE（Java Secure Sockets Extension，Java安全套接字扩展包）、JAAS（Java Authentication and Authentication Service，Java鉴别与安全服务）。

- ❑ JCA提供基本的加密框架，如证书、数字签名、消息摘要和密钥对产生器。
- ❑ JCE在JCA的基础上作了扩展，提供了各种加密算法、消息摘要算法和密钥管理等功能。我们已经有所了解的DES算法、AES算法、RSA算法、DSA算法等就是通过JCE来提供的。有关JCE的实现主要在javax.crypto包（及其子包）中。
- ❑ JSSE提供了基于SSL（Secure Sockets Layer，安全套接字层）的加密功能。在网络的传输过程中，信息会经过多个主机（很有可能其中一台就被窃听），最终传送给接收者，这是不安全的。这种确保网络通信安全的服务就是由JSSE来提供的。
- ❑ JAAS提供了在Java平台上进行用户身份鉴别的功能。如何提供一个符合标准安全机制的登录模块，通过可配置的方式集成至各个系统中呢？这是由JAAS来提供的。

本书将要讨论的主要是JCA、JCE和JSSE。

JCA和JCE是Java平台提供的用于安全和加密服务的两组API。它们并不执行任何算法，

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

它们只是连接应用和实际算法实现程序的一组接口。软件开发商可以根据JCE接口（又称安全提供者接口）将各种算法实现后，打包成一个Provider（安全提供者），动态地加载到Java运行环境中。

根据美国出口限制规定，JCA可出口（JCA和Sun的一些默认实现包含在Java发行版中），但JCE对部分国家是限制出口的。因此，要实现一个完整的安全结构，就需要一个或多个第三方厂商提供的JCE产品，称为安全提供者。BouncyCastle JCE就是其中的一个安全提供者。

安全提供者是承担特定安全机制实现的第三方。有些提供者是完全免费的，而另一些提供者则需要付费。提供安全提供者的公司有Sun、Bouncy Castle等，Sun提供了如何开发安全提供者的细节。Bouncy Castle提供了可以在J2ME/J2EE/J2SE平台得到支持的API，而且Bouncy Castle的API是免费的。

JDK 1.4版本及其后续版本中包含了上述扩展包，无须进行配置。在此之前，安装JDK后需要对上述扩展包进行相应配置。

3.1.2 安全提供者体系结构

Java安全体系结构通过扩展的方式，加入了更多的算法实现及相应的安全机制。我们把这些提供者称为安全提供者（以下简称“提供者”）。以下内容是JDK 1.7所提供的安全提供者的配置信息。

```
security.provider.1=sun.security.provider.Sun
security.provider.2=sun.security.rsa.SunRsaSign
security.provider.3=sun.security.ec.SunEC
security.provider.4=com.sun.net.ssl.internal.ssl.Provider
security.provider.5=com.sun.crypto.provider.SunJCE
security.provider.6=sun.security.jgss.SunProvider
security.provider.7=com.sun.security.sasl.Provider
security.provider.8=org.jcp.xml.dsig.internal.dom.XMLDSigRI
security.provider.9=sun.security.smartcardio.SunPCSC
security.provider.10=sun.security.mscapi.SunMSCAPI
```

上述这些提供者均是Provider类（java.security.Provider）的子类。其中sun.security.provider.Sun是基本安全提供者，sun.security.rsa.SunRsaSign是实现RSA算法的提供者。

与上一版本对比，Java 7新增了EC算法安全提供者——sun.security.ec.SunEC，暗示在该版本中可能支持相应的算法实现。

Java安全体系不仅支持来自Sun官方提供的安全提供者，同时也可配置第三方安全提供者以扩展相应的算法实现等。

安全提供者实现了两个概念的抽象：引擎和算法。引擎可以理解为操作，如加密、解密等。算法则定义了操作如何执行，如一个算法可以理解为一个引擎的具体实现。当然，一个算法可以有多种实现方式，这就意味着同一个算法可能与多个引擎的具体实现相对应。

安全提供者接口的目的就是提供一个简单的机制，从而可以很方便地改变或替换算法及其

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

实现。在实际开发中，程序员只需要用引擎类实现特定的操作，而不需要关心实际进行运算的类是哪一个。

Provider类和Security类（java.security.Security）共同构成了安全提供者的概念。我们将在本章中介绍这两个类的Java API。

3.1.3 关于出口的限制

密码学作为第二次世界大战时决胜关键的一项秘密武器，自然受到各个国家的重视。尤其是军事领域，其出口受到严格的限制。例如，美国以及其他国家都限制加密软件的出口与进口（Sun公司的产品是不能出口到缅甸的）。近年来，美国放松了加密软件出口的条件。尽管如此，JCE和JSSE出口仍受到种种限制。这种出口与进口的限制主要体现在JCE和JSSE的加密算法中。

虽然，核心Java API提供了相应的加密与解密实现，但其加密强度仍不具备军事意义。我们知道密钥长度通常与加密强度成正比，而我们所使用的由Sun公司提供的Java API，因受到美国军事出口限制在其密钥长度上有所缩减，加密强度有所降低。例如，DES算法因受到军事出口限制，目前仅提供56位的密钥长度，而事实上，安全要求则至少需要128位。除此之外，在算法实现上受军事出口限制，其实现内容本身保密，且分为军事与非军事两种实现。显然，非军事加密算法实现远远不如军事加密算法的加密强度高。

3.1.4 关于本章内容

本章将挑选与本书关联较为紧密的Java API作为主要内容。

本章主要详解了java.security包与javax.crypto包，这两个包中包含了Java加密与解密的核心部分，这将是本书着重叙述的内容。

在java.security.interfaces包和javax.crypto.interfaces包中包含了密钥相关的接口，本章将在3.2节中简要介绍。

在java.security.spec包和javax.crypto.spec包中包含了密钥规范和算法参数规范的类和接口，本章将详述如何通过密钥规范获得相应的密钥。

3.1节将着重描述如何完成加密与解密的算法的实现，3.2节和3.3节将描述如何在Java开发环境中使用数字证书，如何通过SSLSocket获取数字证书，以及如何通过HTTPS协议构建加密网络环境。

3.2 java.security包详解

java.security包为安全框架提供类和接口。对该包有所了解的读者一定知道，这只是探索Java加密世界的第一步。通过该包中的Java实现，仅能完成消息摘要算法的实现（消息摘要处理的MessageDigest、DigestInputStream和DigestOutputStream类），并且其源代码是可见的。而要实现真正的加密与解密，需要参考javax.crypto包中的内容。当然，这个包中的源代码内容是不可见的。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

对于该包中与本书相关程度较少的Java API的内容（如java.security.Permission类，与JAAS相关）这里不做介绍。如果读者对这些内容感兴趣，请自行参考其他相应的Java API的内容。

3.2.1 Provider类

Provider类实现了 Java 安全性的一部分或全部，我们称之为提供者，如下所示：

```
// 提供者抽象类
public abstract class Provider
extends Properties
```

Provider类可能实现的服务包括：

- ❑ 算法（如DSA、RSA、MD5或SHA-1）。
- ❑ 密钥的生成、转换和管理设施（如用于特定于算法的密钥）。 每个提供者都有一个名称和一个版本号，并且在它每次装入运行时中进行配置。
- ❑ 方法详述。在实际开发中，很少会直接使用Provider类，我们通常使用的是以下几种方法：

```
// 返回此提供者的名称，如 SunRsaSign 表示该提供者的名称
public String getName()
// 返回此提供者的版本号，如1.5表示该提供者的版本号
public double getVersion()
/* 返回此提供者的信息串。如Sun RSA signature provider表示这是一个
用于RSA算法的数字签名提供者*/
public String getInfo()
```

Provider类覆盖了Object类的以下方法，用以输出自身提供者信息：

```
/* 返回包含此提供者的名称和版本号的字符串。如SunRsaSign version 1.5
表示提供者名称为SunRsaSign，其版本号为1.5*/
public String toString()
```

Provider类继承于Properties类，并覆盖了Properties类的几种常用方法。

我们先来看一下和线程安全相关的方法。

我们可以通过读取输入流的信息，载入提供者相关信息，如下所示：

```
// 从输入流中读取属性列表（键和元素对）
public synchronized void load(InputStream inStream)
```

我们也可以通过下述方法添加提供者相关信息：

```
// 设置键属性，使其具有指定的值
public synchronized Object put(Object key, Object value)
// 将指定 Map 中所有映射关系复制到此提供者中
public synchronized void putAll(Map<?,?> t)
```

我们可以通过如下方法获得当前提供者的相关信息：

```
// 返回此提供者中所包含的属性项的一个不可修改的 Set 视图
public synchronized Set<Map.Entry<Object,Object>> entrySet()
```

我们可以通过下述方法，清理提供者的相关信息：

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
// 移除键属性（及其相应的值）
public synchronized Object remove(Object key)
```

如果使用如下方法，将清理掉有关提供者所支持的全部算法信息：

```
// 清除此提供者，使其不再包含用来查找由该提供者实现的设施的属性
public synchronized void clear()
```

接下来，我们来了解一下非线程安全相关的方法。

Provider类是Properties类的子类，提供了相应的键值操作方法。

以下两个方法均通过给定键获得相应的值：

```
// 返回指定键所映射到的值，如果此映射不包含此键的映射，则返回 null
public Object get(Object key)
// 用指定的键在此属性列表中搜索属性
public String getProperty(String key)
```

以下两个方法可以用枚举的方式获得相应的键值信息：

```
// 返回此散列表中的键的枚举
public Enumeration<Object> keys()
// 返回此散列表中的值的枚举
public Enumeration<Object> elements()
```

通过以下方法可以获得集合方式的键值信息：

```
// 返回此提供者中所包含的属性键的一个不可修改的 Set 视图
public Set<Object> keySet()
// 返回此提供者中所包含的属性值的一个不可修改的 Collection 视图
public Collection<Object> values()
```

Provider类覆盖上述Properties类方法的目的在于确保程序有足够的权限执行相应的操作。这方面涉及安全管理器的概念，已超出本书的范畴，有兴趣的读者可以自己学习SecurityManager类，它位于java.lang包内。

自Java 5开始，Provider类中加入了内部类——Service类。Service类封装了服务的属性，并提供一个用于获得该服务的实现实例的工厂方法。以下为Provider类对Service类的调用支持：

```
// 安全服务的描述。
public static class Provider.Service
// 获取描述此算法或别名的指定类型的此提供者实现的服务
public synchronized Provider.Service getService(String type, String algorithm)
// 获取此提供者支持的所有服务的一个不可修改的 Set。
public synchronized Set<Provider.Service> getServices()
```

在这里讲述Provider类，目的在于引导大家了解“提供者”这一概念，故不对Service类做过多讲述。在Java的安全提供者体系结构中，安全提供者可能不是来自Sun本身。不同的算法可能需要由不同的提供者提供，甚至其提供者本身属于第三方。图3-1是作者使用的Java开发环境下的Provider类及其子类。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

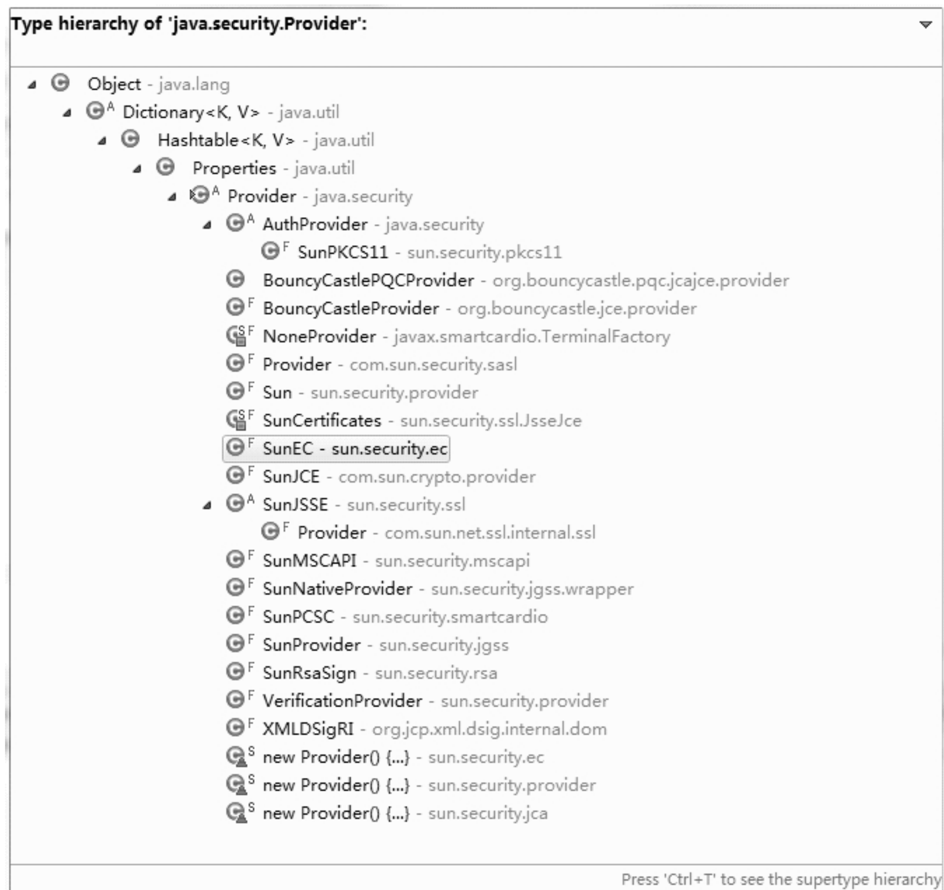

图3-1 Provier类及其子类

相信大家看到位于sun.security.provider包中的Sun类时并不感到陌生。当大家看到位于org.bouncycastle.jce.provider包中的BouncyCastleProvider类时，一定会感到疑惑。从包名结构来看，这不像是由Sun提供的JCE引擎提供者。没错，这是由第三方开源组织Bouncy Castle（http://www.bouncycastle.org/）来提供的。有关于Bouncy Castle加密组件配置及使用，请参考本书第4章相关内容。

那么，在Java 7的环境中都有哪些提供者呢？

查看%JDK_HOME%\jre\lib\security目录，用记事本打开java.security文件。我们会发现这个Properties文件中可以找到以下10种安全提供者：

```
security.provider.1=sun.security.provider.Sun
security.provider.2=sun.security.rsa.SunRsaSign
security.provider.3=sun.security.ec.SunEC
security.provider.4=com.sun.net.ssl.internal.ssl.Provider
security.provider.5=com.sun.crypto.provider.SunJCE
security.provider.6=sun.security.jgss.SunProvider
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
security.provider.7=com.sun.security.sasl.Provider
security.provider.8=org.jcp.xml.dsig.internal.dom.XMLDSigRI
security.provider.9=sun.security.smartcardio.SunPCSC
security.provider.10=sun.security.mscapi.SunMSCAPI
```

上述这些配置是按照以下方式来配置的：

```
security.provider.<n>=<className>
```

很显然，为了加入Bouncy Castle加密组件的安全提供者只需要这样做：

```
#增加BouncyCastleProvider
security.provider.11=org.bouncycastle.jce.provider.BouncyCastleProvider
```

有关如何配置Bouncy Castle加密组件，请参考本书第4章相关内容。Provider类的常用方法调用基本上是和Security类结合在一起的，有关Provider类方法的调用将在下一节中展示。

3.2.2 Security类

Security类的任务就是管理Java 程序中所用到的提供者类。Security类是一个终态类，除了它的私有构造方法外，其余均为静态方法。

```
// 管理提供者
public final class Security
extends Object
```

1. 方法详述

Security类最主要的工作就是对提供者进行管理。

我们可以通过如下方法向系统中追加一个新的提供者：

```
// 将提供者添加到下一个可用位置。在提供者数组尾部追加新的提供者
public static int addProvider(Provider provider)
// 或者，我们直接指定这个提供者位于提供者列表中的位置，请看如下方法
// 在指定的位置添加新的提供者。位置计数从1开始
public static int insertProviderAt(Provider provider, int position)
```

这里有一个优先使用的问题，也就是说位置与优先级成正比，提供者位置越靠前，优先级越高。

能够向系统添加提供者，就能将其移除系统，可以使用如下方法：

```
// 移除带有指定名称的提供者。其余提供者在提供者数组中的位置将可能上移
public static void removeProvider(String name)
```

当然，移除时需要提供提供者的缩写名称，如缩写BC指的是Bouncy Castle提供者。

与移除方法相类似，从系统中获得一个提供者可使用如下方法：

```
// 返回使用指定的名称安装的提供者（如果有）
public static Provider getProvider(String name)
```

当然，也可以使用如下方法直接获得全部提供者：

```
// 返回包含所有已安装的提供者的数组（拷贝）
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
public static Provider[] getProviders()
```

在获得全部提供者的前提下，我们也可以对提供者做一些过滤，可使用如下方法：

```
/* 返回包含满足指定的选择标准的所有已安装的提供者的数组（拷贝），如果尚未安装此类提供者，则返回 null*/
public static Provider[] getProviders(Map<String,String> filter)
/* 返回包含满足指定的选择标准的所有已安装的提供者的数组（拷贝），如果尚未安装此类提供者，则返回 null*/
public static Provider[] getProviders(String filter)
```

大家对%JDK_HOME%\jre\lib\security\java.security文件中的配置应该还有些印象吧？以下方法可以用来设置该文件的相关配置：

```
//设置安全属性值
public static void setProperty(String key, String datum)
//获取安全属性值
public static String getProperty(String key)
```

要取到java.security文件中security.provider.1对应的值（sun.security.provider.Sun），或者对它进行设置就可以使用上述方法来实现。

除此之外，我们还可以通过下述方法获得指定加密服务所对应的可用算法或类型的名称：

```
/* 返回Set视图，这些Set视图中的字符串包含了指定的 Java 加密服务的所有可用算法或类型的名称（例如，Signature、MessageDigest、Cipher、Mac、KeyStore）*/
public static Set<String> getAlgorithms(String serviceName)
```

2. 实现示例

我们可以通过代码查看当前环境中的安全提供者信息，如代码清单3-1所示。

代码清单3-1　打印当前系统所配置的全部安全提供者

```
// 遍历目前环境中的安全提供者
for (Provider p : Security.getProviders()) {
    // 打印当前提供者信息
    System.out.println(p);
    // 遍历提供者Set实体
    for (Map.Entry<Object, Object> entry : p.entrySet()) {
        // 打印提供者键值
        System.out.println("\t" + entry.getKey());
    }
}
```

在上述程序执行后，我们将在控制台中看到以下内容：

```
SUN version 1.7
    Alg.Alias.Signature.SHA1/DSA
    Alg.Alias.Signature.1.2.840.10040.4.3
    Alg.Alias.Signature.DSS
    SecureRandom.SHA1PRNG ImplementedIn
    KeyStore.JKS
    Alg.Alias.MessageDigest.SHA-1
    MessageDigest.SHA
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
KeyStore.CaseExactJKS
CertStore.com.sun.security.IndexedCollection ImplementedIn
Alg.Alias.Signature.DSA
KeyFactory.DSA ImplementedIn
KeyStore.JKS ImplementedIn
AlgorithmParameters.DSA ImplementedIn
Signature.NONEwithDSA
Alg.Alias.CertificateFactory.X509
CertStore.com.sun.security.IndexedCollection
Provider.id className
Alg.Alias.Signature.SHA-1/DSA
CertificateFactory.X.509 ImplementedIn
Signature.SHA1withDSA KeySize
KeyFactory.DSA
CertPathValidator.PKIX ImplementedIn
Configuration.JavaLoginConfig
Alg.Alias.Signature.OID.1.2.840.10040.4.3
Alg.Alias.KeyFactory.1.2.840.10040.4.1
MessageDigest.MD5 ImplementedIn
Alg.Alias.Signature.RawDSA
Provider.id name
Alg.Alias.AlgorithmParameters.1.2.840.10040.4.1
CertPathBuilder.PKIX ValidationAlgorithm
Policy.JavaPolicy
Alg.Alias.AlgorithmParameters.1.3.14.3.2.12
Alg.Alias.Signature.SHA/DSA
Alg.Alias.KeyPairGenerator.1.3.14.3.2.12
MessageDigest.SHA-384
Signature.SHA1withDSA ImplementedIn
AlgorithmParameterGenerator.DSA
Signature.NONEwithDSA SupportedKeyClasses
MessageDigest.SHA-512
......
```

在此，本书没有将控制台获得的信息全部展示出来。在Java 5以前，上述提供者的信息还不到1张A4纸，但到了Java 7后，提供者的信息变得相当庞大。如果将上述完整的信息打印到A4纸上，可以密密麻麻地打印出20多页。这同样说明，在Java 7的环境中，对于加密算法的选择有了更广泛的空间。

3.2.3 MessageDigest类

MessageDigest类实现了消息摘要算法，它继承于MessageDigestSpi类，是Java安全提供者体系结构中最为简单的一个引擎类。在本章后续的内容中，我们还将看到更多引擎类。

```
// 实现创建和验证消息摘要的操作
public abstract class MessageDigest
extends MessageDigestSpi
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

在Java API的列表中，我们总能看到后缀名带有SPI（Service Provider Interface，服务提供者接口）的类，它们总是与引擎类形影不离。例如MessageDigestSpi类是MessageDigest 类的服务提供者接口。如果要实现自定义的消息摘要算法，则需要提供继承于MessageDigestSpi类的子类实现。MessageDigest类自然是Sun提供给我们的一个很好的用于实现自定义算法的范例。在其他引擎类的实现中（除签名算法类Signature类以外），不再继承相对应的SPI类。本书以Java 7所提供的算法为准，不计划介绍如何实现自定义算法。如果读者对SPI类有兴趣，可参照Java API中的相应内容。

1. 方法详述

我们可以通过以下方法获得MessageDigest类的实例：

```
// 返回实现指定摘要算法的 MessageDigest对象
public static MessageDigest getInstance(String algorithm)
// 返回实现指定摘要算法的 MessageDigest对象
public static MessageDigest getInstance(String algorithm, Provider provider)
// 返回实现指定摘要算法的 MessageDigest对象
public static MessageDigest getInstance(String algorithm, String provider)
```

目前Java 7支持MD2、MD5、SHA-1（SHA）、SHA -256、SHA-384、SHA-512六种消息摘要算法，算法名不区分大小写。

在本书后续的内容中，诸如上述的算法提供者参数（参数Provider provider或参数String provider）会有很多，其值可以为new com.sun.crypto.provider.SunJCE()或"SunJCE"。当使用第三方安全提供者时，需指明provider参数。如果使用BouncyCastleProvider，其值必须为new org.bouncycastle.jce.provider.BouncyCastleProvider()或"BC"。当然，使用第三方安全提供者的前提是需要配置%JDK_HOME%\jre\lib\security\java.security文件。相关内容请参见3.2.1节。

在获得MessageDigest实例化对象后，我们可以利用以下方法对此对象进行操作：

```
// 使用指定的字节更新摘要
public void update(byte input)
// 使用指定的字节数组更新摘要
public void update(byte[] input)
```

上述方法参数不同，一个是更新单字节，一个是更新字节数组。

我们也可以使用如下方法，根据偏移量更新摘要：

```
// 使用指定的字节数组，从指定的偏移量开始更新摘要
public void update(byte[] input, int offset, int len)
```

当然，我们也可以使用缓冲模式更新摘要，可使用如下方法：

```
// 使用指定的字节缓冲更新摘要
public void update(ByteBuffer input)
```

更新摘要信息，其参数可以是更新一个字节、一个字节数组，甚至是字节数组中的某一段偏移量，也可以是字节缓冲对象。

这些方法的调用顺序不受限制，在向摘要中增加所需数据时可以多次调用。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

在完成摘要更新后，我们可以通过以下方法完成摘要操作：

```
// 通过执行诸如填充之类的最终操作完成摘要计算，返回消息摘要字节数组
public byte[] digest()
```

作者通常会选择上述方法完成摘要处理，也就是在调用上述方法前使用update()方法完成摘要更新后直接使用digest()方法。如果我们只需要执行一次update()方法，不如考虑在digest()方法中直接传入待摘要信息，也就是直接使用如下方法：

```
/* 使用指定的字节数组对摘要进行最后更新，然后完成摘要计算，返回消息摘要字节数组*/
public byte[] digest(byte[] input)
/* 通过执行诸如填充之类的最终操作完成摘要计算，将消息摘要结果按照指定的偏移量存放在字节数组中，返回消息摘要的长度*/
public int digest(byte[] buf, int offset, int len)
```

完成摘要操作后，可以通过以下方法对比两个摘要信息：

```
// 比较两个摘要的相等性
public static boolean isEqual(byte[] digesta, byte[] digestb)
```

当两个摘要中的每个字节都相同，且两个摘要都有相同的长度时，则认为两个摘要相同。

```
// 重置摘要以供再次使用
public void reset()
```

执行该重置方法等同于创建一个新的MessageDigest实例化对象。

我们完成了摘要处理、获得了摘要信息后，就能获得摘要信息的长度了，如果需要这样的长度信息，可以使用如下方法：

```
/* 返回以字节为单位的摘要长度，如果提供者不支持此操作并且实现是不可复制的，则返回0*/
public final int getDigestLength()
```

除了上述方法外，常用的方法还有以下几种：

```
// 返回算法名称，如MD5
public final String getAlgorithm()
// 返回此信息摘要对象的提供者
public final Provider getProvider()
```

MessageDigest类覆盖了Object的如下方法：

```
// 返回此信息摘要对象的字符串表示形式
public String toString()
//如果实现是可复制的，则返回一个副本
Object clone()
```

2. 实现示例

我们来做一个简单的摘要处理，如代码清单3-2所示。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

代码清单3-2 SHA算法摘要处理

```
// 待做消息摘要算法的原始信息
byte[] input = "sha".getBytes();
// 初始化MessageDigest对象，将使用SHA算法
MessageDigest sha = MessageDigest.getInstance("SHA");
// 更新摘要信息
sha.update(input);
// 获取消息摘要结果
byte[] output = sha.digest();
```

关于消息摘要算法的实现

MessageDigest、DigestInputStream、DigestOutputStream、Mac均是消息认证技术的引擎类。MessageDigest提供核心的消息摘要实现；DigestInputStream、DigestOutputStream提供以MessageDigest为核心的消息摘要流实现；Mac提供基于秘密密钥的安全消息摘要实现。Mac与MessageDigest无任何依赖关系。

除了MessageDigest类提供的MD和SHA两大类算法外，还有一种摘要算法——Mac，它的算法实现是通过Mac类（javax.crypto.Mac）来实现的，具体Java API内容请参见下一节内容。

3.2.4 DigestInputStream类

通过MessageDigest类，我们实现了消息摘要算法，但是通过字节或字节数组的方式完成摘要操作，使用起来并不是很方便，因此有了消息摘要流，包含消息摘要输入流和消息摘要输出流。

DigestInputStream类继承了FilterInputStream类，可以通过读取输入流的方式完成摘要更新，因此我们称之为消息摘要输入流，在指定的读操作方法内部完成MessageDigest类的update()方法。

```
// 提供一个输入流，针对所有通过该流的数据计算出相应的消息摘要
public class DigestInputStream
extends FilterInputStream
```

1. 方法详述

可以通过以下方法实例化对象：

```
/* 使用指定的InputStream和MessageDigest创建一个DigestInputStream实例*/
public DigestInputStream(InputStream stream, MessageDigest digest)
```

当需要更新MessageDigest对象时，可以使用如下方法：

```
// 将指定的MessageDigest与此流相关联
public void setMessageDigest(MessageDigest digest)
```

与之相对应的，获得MessageDigest对象可以用以下方法：

```
// 返回与此流关联的MessageDigest
public MessageDigest getMessageDigest()
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

当通过下述方法关闭摘要功能后，DigestInputStream就变成了一般的输入流：

```
// 开启或关闭摘要功能
public void on(boolean on)
```

请注意要使用下述方法更新摘要信息时，一定要确保DigestInputStream开启摘要功能：

```
// 读取字节并更新消息摘要（如果开启了摘要功能）
public int read()
// 读入byte数组并更新消息摘要（如果开启了摘要功能）
public int read(byte[] b, int off, int len)
```

上述方法将调用MessageDigest的update()方法完成摘要更新，在此之后可以通过getMessageDigest()方法获得MessageDigest对象，并执行MessageDigest对象的digest()方法完成摘要操作。

摘要DigestInputStream类的相关源码，如代码清单3-3所示。

代码清单3-3 DigestInputStream类读操作部分源代码

```
public int read() throws IOException {
    int ch = in.read();
    if (on && ch != -1) {
        digest.update((byte)ch);
    }
    return ch;
}
public int read(byte[] b, int off, int len) throws IOException {
    int result = in.read(b, off, len);
    if (on && result != -1) {
        digest.update(b, off, result);
    }
    return result;
}
```

除上述方法外，通常还会用到以下方法：

```
// 打印此摘要输入流及其关联的消息摘要对象的字符串表示形式
public String toString()
```

2. 实现示例

我们可以通过如下方式使用该消息摘要输入流，如代码清单3-4所示。

代码清单3-4 MD5算法摘要输入流处理

```
// 待做消息摘要操作的原始信息
byte[] input = "md5".getBytes();
// 初始化MessageDigest对象，将使用MD5算法
MessageDigest md = MessageDigest.getInstance("MD5");
// 构建DigestInputStream对象。
DigestInputStream dis = new DigestInputStream(new ByteArrayInputStream(input), md);
// 流输入
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
dis.read(input, 0, input.length);
// 获得摘要信息
byte[] output = dis.getMessageDigest().digest();
// 关闭流
dis.close();
```

3.2.5　DigestOutputStream类

与DigestInputStream类相对应，DigestOutputStream类继承了FilterOutputStream类，可以通过写入输出流的方式完成摘要更新，因此我们称之为消息摘要输出流，在指定的写操作方法内部完成MessageDigest类的update()方法。

```
// 提供一个输出流，针对所有通过该流的数据计算出相应的消息摘要
public class DigestOutputStream
extends FilterOutputStream
```

1. 方法详述

可以通过以下方法实例化对象：

```
/* 使用指定的OutputStream和MessageDigest创建一个DigestOutputStream实例*/
public DigestOutputStream(OutputStream stream, MessageDigest digest)
```

以下方法与DigestInputStream类相同：

```
// 将指定的MessageDigest与此流相关联
public void setMessageDigest(MessageDigest digest)
// 返回与此流关联的MessageDigest
public MessageDigest getMessageDigest()
```

当通过下述方法关闭摘要功能后，DigestOutputStream就变成了一般的输出流：

```
// 开启或关闭摘要功能
public void on(boolean on)
```

请注意要使用下述方法更新摘要信息时，一定要确保DigestOutputStream开启摘要功能：

```
/* 使用指定的字节更新消息摘要（如果开启了摘要功能），并将字节写入输出流（不管是否开启了摘要功能）*/
public void write(int b)
/* 使用指定的子数组更新消息摘要（如果开启了摘要功能），并将子数组写入输出流（不管是否开启了摘要功能）*/
public void write(byte[] b, int off, int len)
```

上述方法将调用MessageDigest类的update()方法完成摘要更新，在此之后可以通过getMessageDigest()方法获得MessageDigest对象，并执行MessageDigest类的digest()方法完成摘要操作。

摘要DigestOutputStream类的相关源码如代码清单3-5所示。

代码清单3-5　DigestOutputStream类写操作部分源代码

```
public void write(int b) throws IOException {
    if (on) {
        digest.update((byte)b);
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
        }
        out.write(b);
    }
    public void write(byte[] b, int off, int len) throws IOException {
        if (on) {
            digest.update(b, off, len);
        }
        out.write(b, off, len);
    }
```

除上述方法外，通常还会用到以下方法：

```
// 打印此摘要输出流及其关联的消息摘要对象的字符串表示形式
public String toString()
```

2. 实现示例

我们可以通过如代码清单3-6所示的方式使用该消息摘要输出流。

代码清单3-6　MD5算法摘要输出流处理

```
// 待做消息摘要的原始信息
byte[] input = "md5".getBytes();
// 初始化MessageDigest，将使用MD5算法
MessageDigest md = MessageDigest.getInstance("MD5");
// 初始化DigestOutputStream。
DigestOutputStream dos = new DigestOutputStream(new ByteArrayOutputStream(), md);
// 流输出
dos.write(input, 0, input.length);
// 获得摘要信息
byte[] output = dos.getMessageDigest().digest();
// 清空流
dos.flush();
// 关闭流
dos.close();
```

HZ BOOKS 华章图书

3.2.6　Key接口

Key接口是所有密钥接口的顶层接口，一切与加密解密有关的操作都离不开Key接口。

```
/* 模型化密钥的概念。由于密钥必须在不同实体之间传输，因此所有的密钥都必须是可以序列化的*/
public interface Key
extends Serializable
```

所有的密钥都具有三个特征：

- 算法。这里指的是密钥的算法，如DES和DSA等，通过getAlgorithm()方法可获得算法名。
- 编码形式。这里指的是密钥的外部编码形式，密钥根据标准格式（如X.509 SubjectPublicKeyInfo或 PKCS#8）编码，使用getEncoded()方法可获得编码格式。
- 格式。这里指的是已编码密钥的格式的名称，使用getFormat()方法可获得格式。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

对应上述三个特征，Key接口提供如下三个方法：

```
// 返回此密钥的标准算法名称
public String getAlgorithm()
/* 返回基本编码格式的密钥，通常为二进制格式，如果此密钥不支持编码，则返回 null*/
public byte[] getEncoded()
// 返回此密钥的基本编码格式，如果此密钥不支持编码，则返回 null
public String getFormat()
```

SecretKey、PublicKey、PrivateKey三大接口继承于Key接口，定义了对称密钥和非对称密钥接口。

（1）SecretKey

SecretKey（javax.crypto.SecretKey）接口是对称密钥顶层接口。DES、AES等多种对称密码算法密钥均可通过该接口提供，PBE（javax.crypto.interfaces.PBE）接口提供PEB算法定义并继承了该接口。Mac算法实现过程中，通过SecretKey接口提供秘密密钥。通常使用的是SecretKey接口的实现类SecretKeySpec（javax.crypto.spec.SecretKeySpec）。

```
// 秘密（对称）密钥
public interface SecretKey
extends Key
```

（2）PublicKey和PrivateKey

PublicKey、PrivateKey接口是非对称密钥顶层接口。

```
// 公钥
public interface PublicKey
extends Key
// 私钥
public interface PrivateKey
extends Key
```

DH、RSA、DSA和EC等多种非对称密钥接口均继承了这两个接口。RSA、DSA、EC接口均在java.security.interfaces包中。DH的密钥接口位于javax.crypto.interfaces包中。关于java.security.interfaces包和javax.crypto.interfaces包本章不做细节介绍，有兴趣的读者可以自行关注相应的Java API内容。

3.2.7 AlgorithmParameters类

AlgorithmParameters类是一个引擎类，它提供密码参数的不透明表示。

不透明表示与透明表示的区别在于：

- 不透明表示：在这种表示中，不可以直接访问各参数域，只能得到与参数集相关联的算法名及该参数集的某类编码。
- 透明表示：用户可以通过相应规范类中定义的某个“get”方法来分别访问每个值。

```
// 提供密码参数不透明表示
public class AlgorithmParameters
extends Object
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

1. 方法详述

可以通过getInstance()工厂方法实例化AlgorithmParameters对象，如下所示：

```
// 通常，我们使用如下方法获得实例化对象
// 返回指定算法的AlgorithmParameters对象
public static AlgorithmParameters getInstance(String algorithm)
```

当然，我们可以同时指定算法的相应提供者来完成实例化操作，可使用如下方法：

```
// 返回指定算法的AlgorithmParameters对象
public static AlgorithmParameters getInstance(String algorithm, Provider provider)
// 返回指定算法的AlgorithmParameters对象
public static AlgorithmParameters getInstance(String algorithm, String provider)
```

完成对象实例化后，需要对其进行初始化：

```
// 使用在 paramSpec 中指定的参数初始化此AlgorithmParameters对象
public final void init(AlgorithmParameterSpec paramSpec)
//根据参数的基本解码格式导入指定的参数字节数组并对其解码
public final void init(byte[] params)
//根据指定的解码方案从参数字节数组导入参数并对其解码
public final void init(byte[] params, String format)
```

AlgorithmParameters对象只能被初始化一次，无法重用。

在上述init()方法中，我们看到AlgorithmParameterSpec参数，它是加密参数的（透明）规范接口，位于java.security.spec包中，其接口内部无任何方法，仅用于将所有参数规范分组，并为其提供类型安全。所有参数规范都必须实现此接口。

在完成初始化后，可通过以下方法获得参数对象的规范：

```
// 返回此参数对象的（透明）规范
public final <T extends AlgorithmParameterSpec> T getParameterSpec(Class<T> paramSpec)
```

我们可以通过以下方法获得算法参数：

```
// 返回基本编码格式的参数
public final byte[] getEncoded()
// 返回以指定方案编码的参数
public final byte[] getEncoded(String format)
```

此外，AlgorithmParameters类提供了以下常用方法：

```
// 返回与此参数对象关联的算法的名称
public final String getAlgorithm()
// 返回此参数对象的提供者
public final Provider getProvider()
// 返回描述参数的格式化字符串
public final String toString()
```

2. 实现示例

我们通过代码清单3-7来展示如何使用AlgorithmParameters类获得算法参数。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

代码清单3-7 构建DES算法参数

```
// 实例化AlgorithmParameters，并指定DES算法
AlgorithmParameters ap = AlgorithmParameters.getInstance("DES");
// 使用BigInteger生成参数字节数组。
ap.init(new BigInteger("19050619766489163472469").toByteArray());
// 获得参数字节数组。
byte[] b = ap.getEncoded();
/* 打印经过BigInteger处理后的字符串，将会得到与"19050619766489163472469"相同的字符串*/
System.out.println(new BigInteger(b).toString());
```

上述字符串（19050619766489163472469）是作者通过AlgorithmParameterGenerator类操作得到的，并不是随意填写的一个数值。

3.2.8 AlgorithmParameterGenerator类

AlgorithmParameterGenerator类用于生成将在某个特定算法中使用的参数集合，我们把它称为算法参数生成器，它同样是一个引擎类。

```
// 生成制定算法的参数集合
public class AlgorithmParameterGenerator
extends Object
```

1. 方法详述

AlgorithmParameterGenerator类同样需要通过getInstance()工厂方法完成实例化对象。

通过算法名称直接指定算方法参数生成器是最简便的实例化方法了，方法如下所示：

```
/* 返回生成与指定算法一起使用的参数集的AlgorithmParameterGenerator对象*/
public static AlgorithmParameterGenerator getInstance(String algorithm)
```

或者，指定算法的同时指明该算法的提供者，方法如下：

```
/* 返回生成与指定算法一起使用的参数集的AlgorithmParameterGenerator对象*/
public static AlgorithmParameterGenerator getInstance(String algorithm, Provider provider)
/* 返回生成与指定算法一起使用的参数集的AlgorithmParameterGenerator对象*/
public static AlgorithmParameterGenerator getInstance(String algorithm, String provider)
```

算法生成器共有两种初始化方式：与算法无关的方式或特定于算法的方式。

两种方式的唯一区别在于对象的初始化：

（1）与算法无关的初始化

所有参数生成器都共享“大小”概念和一个随机源。AlgorithmParameterGenerator类提供了两个init ()方法，均包含参数int size，另一个方法则要求提供SecureRandom参数。使用这一方法时，特定于算法的参数生成值（如果有）默认为某些标准值，除非它们可以从指定大小派生。

（2）特定于算法的初始化

使用特定于算法的语义初始化参数生成器对象，这些语义由特定于算法的参数生成值集合表示。有两个initialize()方法具有AlgorithmParameterSpec参数，其中一个方法还有一个SecureRandom参数，而另一个方法使用以最高优先级安装的提供者的SecureRandom实现作为

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

随机源。（如果任何安装的提供者都不提供SecureRandom的实现，则使用系统提供的随机源。）

与算法无关的初始化方法如下：

```
// 针对某个特定大小初始化此AlgorithmParameterGenerator对象
public final void init(int size)
// 针对某个特定大小和随机源初始化此AlgorithmParameterGenerator对象
public final void init(int size, SecureRandom random)
```

特定于算法的初始化方法如下：

```
/* 利用特定于算法的参数生成值集合初始化此AlgorithmParameterGenerator对象*/
public final void init(AlgorithmParameterSpec genParamSpec)
/* 利用特定于算法的参数生成值集合初始化此AlgorithmParameterGenerator对象*/
public final void init(AlgorithmParameterSpec genParamSpec, SecureRandom random)
```

在完成实例化对象操作后，我们就可以通过以下方法生成算法参数对象了：

```
// 生成AlgorithmParameters对象
public final AlgorithmParameters generateParameters()
```

此外，算法生成器还提供以下常用方法：

```
// 返回与此参数生成器关联的算法的标准名称
public final String getAlgorithm()
// 返回此算法参数生成器对象的提供者
public final Provider getProvider()
```

2. 实现示例

在3.2.7节中，曾提到用于AlgorithmParameter初始化的字符串（19050619766489163472469）是通过AlgorithmParameterGenerator操作来生成的。代码清单3-8就是获得该字符串的代码。

代码清单3-8　通过算法参数生成器获得DES算法相应的算法参数

```
// 实例化AlgorithmParameterGenerator对象，并指定DES算法
AlgorithmParameterGenerator apg = AlgorithmParameterGenerator.getInstance("DES");
// 初始化
apg.init(56);
// 生成AlgorithmParameters对象
AlgorithmParameters ap = apg.generateParameters();
// 获得参数字节数组
byte[] b = ap.getEncoded();
/* 打印经过BigInteger处理后的字符串，该字符串就是"19050619766489163472469"*/
System.out.println(new BigInteger(b).toString());
```

当使用Java提供的加密组件时，很少会用到AlgorithmParameters和AlgorithmParameter-Generator两个类，当对算法参数要求极为严格的情况下才会考虑使用这种方式。

3.2.9　KeyPair类

KeyPair类是对非对称密钥的扩展，它是密钥对的载体，我们把它称为密钥对。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
// 建立一个包括公钥和私钥的数据对象
public final class KeyPair
extends Object
implements Serializable
```

KeyPair类包含两个信息：公钥和私钥。

我们可以像使用一般数据对象一样使用它。

构造一个KeyPair对象很简单，以下是它的构造方法：

```
// 根据给定的公钥和私钥构造KeyPair对象
public KeyPair(PublicKey publicKey, PrivateKey privateKey)
```

KeyPair只能通过构造方法初始化内部的公钥和私钥，此外不提供设置公钥和私钥的方法，仅提供获取公钥和私钥的方法：

```
// 返回对此KeyPair对象的公钥组件的引用
public PublicKey getPublic()
// 返回对此KeyPair对象的私钥组件的引用
public PrivateKey getPrivate()
```

KeyPair通常由KeyPairGenerator的generateKeyPair()方法来提供。

关于KeyPair对象的生成实现，请关注3.2.10节的内容。

3.2.10　KeyPairGenerator类

公钥和私钥的生成是由KeyPairGenerator类来实现的，因此我们把它称为密钥对生成器，它同样是一个引擎类。

```
// 生成用于非对称加密算法中含有公钥/私钥的密钥对，并提供相关信息
public abstract class KeyPairGenerator
extends KeyPairGeneratorSpi
```

在Java 7中，提供了DH、ECDH、ECDSA、DSA和RSA等多种非对称加密算法或签名算法实现。

关于Java 7中提供的算法信息，请查看本书附录。

1. 方法详述

KeyPairGenerator本身是一个抽象类，可通过getInstance()工厂方法实例化对象。

通常我们指定算法名称，直接获得密钥对实例化对象，方法如下所示：

```
// 返回生成指定算法的公钥/私钥密钥对的KeyPairGenerator对象
public static KeyPairGenerator getInstance(String algorithm)
```

或者，指定算法的同时指明该算法的提供者，方法如下所示：

```
// 返回生成指定算法的公钥/私钥密钥对的KeyPairGenerator对象
public static KeyPairGenerator getInstance(String algorithm, Provider provider)
// 返回生成指定算法的公钥/私钥密钥对的KeyPairGenerator对象
public static KeyPairGenerator getInstance(String algorithm, String provider)
```

密钥对生成器共有两种初始化方式：与算法无关的方式和特定于算法的方式，这一点类似

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

于算法参数生成器。

两种方式的唯一区别在于对象的初始化：

- 与算法无关的初始化。所有的密钥对生成器遵循密钥大小和随机源的概念。KeyPairGenerator类提供了两个initialize()方法，这两个方法都含有密钥大小参数，另一个方法还要求提供SecureRandom参数。
- 特定于算法的初始化。对于特定于算法的参数集合已存在的情况（例如，DSA中所谓的公用参数），KeyPairGenerator类提供了两个initialize()方法，都具有AlgorithmParameterSpec参数。其中一个方法还有一个SecureRandom参数，而另一个方法使用以最高优先级安装的提供者的 SecureRandom实现作为随机源。（如果任何安装的提供者都不提供SecureRandom的实现，则使用系统提供的随机源。）

与算法无关的初始化方法如下：

```
/* 初始化给定密钥大小的密钥对生成器，使用默认的参数集合，并使用以最高优先级安装的提供者的SecureRandom 实现作为随机源*/
public void initialize(int keysize)
// 使用给定的随机源（和默认的参数集合）初始化确定密钥大小的密钥对生成器
public void initialize(int keysize, SecureRandom random)
```

特定于算法的初始化方法如下：

```
/* 初始化KeyPairGenerator对象，使用指定参数集合，并使用以最高优先级安装的提供者的SecureRandom 的实现作为随机源*/
public void initialize(AlgorithmParameterSpec params)
// 使用给定参数集合和随机源初始化KeyPairGenerator对象
public void initialize(AlgorithmParameterSpec params, SecureRandom random)
```

通常，较为常用的是前两个方法。注意参数int keysize，这个参数用来控制密钥长度，单位为位。

我们通过对第2章的学习已经知道，密钥长度决定密码安全强度。在使用该类实现具体某一种算法时，需要切合实际设置相应的密钥长度。如构建DH算法密钥对，其密钥长度要求为64位的倍数，密钥长度范围为512～1024位，即DH算法密钥长度可为512、576、640位等，按64位的整数倍递增至1024位。

如果在实例化操作后未执行初始化操作，密钥长度将设置为默认长度。如DH算法密钥长度默认值为1024位。关于Java 7中密码算法与密钥长度说明，请见本书附录。

在完成上述操作后，可以通过如下方法生成密钥对：

```
// 生成一个KeyPair对象
public KeyPair generateKeyPair()
// 生成KeyPair对象
Public final KeyPair genKeyPair()
```

注意这两个方法，genKeyPair()方法内部实现时调用了generateKeyPair()方法，给出相应的代码片段：

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
public KeyPair generateKeyPair() {
    return null;
}
public final KeyPair genKeyPair() {
    return generateKeyPair();
}
```

generateKeyPair()方法由具体的密钥对生成器提供者实现，通常使用genKeyPair()方法获得密钥对。

与其他引擎类一样，密钥对生成器提供如下两种方法：

```
// 返回此密钥对生成器算法的标准名称
public String getAlgorithm()
// 返回此密钥对生成器对象的提供者
public final Provider getProvider()
```

2. 实现示例

经过上述方法相结合，我们可以很方便地生成一个密钥对，如生成一个DSA算法密钥对：

```
// 实例化KeyPairGenerator对象
KeyPairGenerator kpg = KeyPairGenerator.getInstance("DSA");
// 初始化KeyPairGenerator对象
kpg.initialize(1024);
// 生成KeyPair对象
KeyPair keys = kpg.genKeyPair();
```

KeyPairGenerator类提供了非对称密钥对的生成实现，如果要生成私钥，则可使用KeyGenerator（javax.crypto.KeyGenerator）类。我们将在下一节中讲述它。

3.2.11　KeyFactory类

KeyFactory类也是用来生成密钥（公钥和私钥）的引擎类，我们将它称为密钥工厂，它用来生成公钥/私钥，或者说它的作用是通过密钥规范还原密钥。与KeyFactory类相对应的类是SecretKeyFactory类，这个类用于生成秘密密钥，我们将在3.3.4节中详述。KeySpec接口定义了密钥规范，有关KeySpec接口，请参见3.4.1节。

```
// 按照指定的编码格式或密钥参数，提供一个用于输入和输出密钥的基础设施
public class KeyFactory
extends Object
```

1. 方法详述

KeyFactory类同样通过getInstance()工厂方法来获得实例化对象。

我们可以通过指定算法名称的方式获得实例化对象，如下所示：

```
// 返回转换指定算法的公钥/私钥关键字的KeyFactory对象
public static KeyFactory getInstance(String algorithm)
```

另外一种方式就是指定算法名称的时候指定该算法的提供者，如下所示：

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
// 返回转换指定算法的公钥/私钥关键字的KeyFactory对象
public static KeyFactory getInstance(String algorithm, Provider provider)
// 返回转换指定算法的公钥/私钥关键字的KeyFactory对象
public static KeyFactory getInstance(String algorithm, String provider)
```

KeyFactory类最常用的操作就是通过密钥规范获得相应的密钥，其方法如下：

```
// 根据提供的密钥规范（密钥材料）生成PublicKey对象
public final PublicKey generatePublic(KeySpec keySpec)
// 根据提供的密钥规范（密钥材料）生成PrivateKey对象
public final PrivateKey generatePrivate(KeySpec keySpec)
```

可以通过以下方法获得密钥规范：

```
// 返回给定密钥对象的规范（密钥材料）
public final <T extends KeySpec> T getKeySpec(Key key, Class<T> keySpec)
```

可以通过以下方法转换来自不同提供者装载的密钥类型：

```
// 将提供者可能未知或不受信任的密钥对象转换成此密钥工厂对应的密钥对象
public final Key translateKey(Key key)
```

密钥工厂同样提供以下两个常用方法：

```
// 获取与此 KeyFactory关联的算法的名称
public final String getAlgorithm()
// 返回此KeyFactory对象的提供者
public final Provider getProvider()
```

2. 实现示例

通过代码清单3-9来说明密钥工厂如何将密钥字节数组转换为密钥对象。

代码清单3-9　构建密钥对与还原密钥

```
// 实例化KeyPairGenerator对象，并指定RSA算法
KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("RSA");
// 初始化KeyPairGenerator对象
keyPairGen.initialize(1024);
// 生成KeyPair对象
KeyPair keyPair = keyPairGen.generateKeyPair();
// 获得私钥密钥字节数组。实际使用过程中该密钥以此种形式保存传递给另一方
byte[] keyBytes = keyPair.getPrivate().getEncoded();
// 由私钥密钥字节数组获得密钥规范
PKCS8EncodedKeySpec pkcs8KeySpec = new PKCS8EncodedKeySpec(keyBytes);
// 实例化密钥工厂，并指定RSA算法
KeyFactory keyFactory = KeyFactory.getInstance("RSA");
// 生成私钥
Key privateKey = keyFactory.generatePrivate(pkcs8KeySpec);
```

有关密钥规范请参见3.4节的相关内容。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

3.2.12　SecureRandom类

SecureRandom类继承于Random类（java.util.Random），它起到强加密随机数生成器（Random Number Generator，RNG）的作用，我们称之为安全随机数生成器，它同样是一个引擎类。

```
// 提供安全随机数
publicclass SecureRandom
extends Random
```

1. 方法详述

安全随机数生成器可通过以下构造方法进行实例化对象：

```
// 构造一个实现默认随机数算法的SecureRandom对象
public SecureRandom()
// 在给定种子的前提下，构造一个实现默认随机数算法的SecureRandom对象
public SecureRandom(byte[] seed)
```

当然，我们可以通过给定的其他参数，使用getInstance()工厂方法来完成实例化对象。

我们可以通过指定算法名称的方式获得安全随机数对象，方法如下：

```
// 返回实现指定随机数生成器算法的SecureRandom对象
public static SecureRandom getInstance(String algorithm)
```

或者，指明该算法名的同时指定该算法的提供者：

```
// 返回实现指定随机数生成器算法的SecureRandom对象
public static SecureRandom getInstance(String algorithm, Provider provider)
// 返回实现指定随机数生成器算法的SecureRandom对象
public static SecureRandom getInstance(String algorithm, String provider)
```

SHA1PRNG算法是SecureRandom的默认算法。

在获得实例化对象后，可以多次使用如下方法生成种子：

```
/* 返回给定的种子字节数量，该数量可使用此类来将自身设置为种子的种子生成算法来计算*/
public byte[] generateSeed(int numBytes)
```

SecureRandom类覆盖了Random类的以下几种方法：

```
// 生成用户指定的随机字节数。其结果将填充在参数byte[] bytes中
public synchronized void nextBytes(byte[] bytes)
// 使用给定 long seed 中包含的8个字节，重新设置此随机对象的种子
public void setSeed(long seed)
```

使用如下方法将重置随机数对象中的种子：

```
//重新设置此随机对象的种子
public synchronized void setSeed(byte[] seed)
```

使用如下方法将获得新的随机数：

```
/* 返回给定的种子字节数量，该数量可使用此类来将自身设置为种子的种子生成算法来计算*/
public static byte[] getSeed(int numBytes)
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

此外，安全随机数生成器作为引擎类，提供如下常用方法：

```
// 返回此安全随机数生成器对象实现的算法的名称
public String getAlgorithm()
// 返回此安全随机数生成器对象的提供者
public final Provider getProvider()
```

2. 实现示例

安全随机数生成器常用来配合生成秘密密钥，如代码清单3-10所示。

代码清单3-10 构建安全随机数对象及秘密密钥对象

```
// 实例化SecureRandom对象
SecureRandom secureRandom = new SecureRandom();
// 实例化KeyGenerator对象
KeyGenerator kg = KeyGenerator.getInstance("DES");
// 初始化KeyGenerator对象
kg.init(secureRandom);
// 生成SecretKey对象
SecretKey secretKey = kg.generateKey();
```

SecureRandom类的另一个用途为：在数字签名Signature类中辅助完成初始化操作。

3.2.13 Signature类

Signature 类用来生成和验证数字签名，它同样是一个引擎类。

```
// 提供一个引擎，用以创建和验证数字签名
public abstract class Signature
extends SignatureSpi
```

1. 方法详述

使用Signature对象签名数据或验证签名包括以下三个阶段：

1）初始化。

- 初始化验证签名的公钥
- 初始化签署签名的私钥

2）更新。

根据初始化类型，可更新要签名或验证的字节。

3）签署或验证所有更新字节的签名。

按照以上描述，我们先通过getInstance()工厂方法完成实例化对象。

一种简单的方式就是指定算法名称，直接获得Signature对象，方法如下：

```
// 返回实现指定签名算法的 Signature对象
public static Signature getInstance(String algorithm)
```

另一种方式就是指定算法名称的同时指定该算法的提供者，方法如下：

```
// 返回实现指定签名算法的 Signature对象
```

HZ BOOKS 华章图书

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
public static Signature getInstance(String algorithm, Provider provider)
// 返回实现指定签名算法的 Signature对象
public static Signature getInstance(String algorithm, String provider)
```

在以下几步操作中，我们会看到消息摘要处理的一些类似的方法，大家可以参见3.2.3节和3.3.1节的内容。

目前，Signature支持NONEwithDSA和SHA1withDSA两种基于DSA算法的签名算法，同时还支持MD2withRSA、MD5withRSA、SHA1withRSA、SHA256withRSA、SHA384withRSA和SHA512withRSA六种基于RSA算法的签名算法。

关于Java 7版本中提供的算法信息，请参见本书附录。

完成实例化对象后，我们要初始化Signature对象：

```
// 初始化这个用于签名的Signature对象
public final void initSign(PrivateKey privateKey)
// 初始化这个用于签名的Signature对象
public final void initSign(PrivateKey privateKey, SecureRandom random)
```

上述两种方法是用于签名操作的初始化，如果用于验证操作则需要通过以下方法：

```
// 初始化此用于验证的Signature对象
public final void initVerify(PublicKey publicKey)
// 使用来自给定证书的公钥初始化用于验证的Signature对象
public final void initVerify(Certificate certificate)
```

注意上述方法的参数有所不同，前一种方法用于一般数字签名的验证操作，后一种方法则用于数字证书的验证操作。

完成初始化操作后，我们就可以通过以下方法更新Signature对象中的数据了。

Signature类的update()方法定义与MessageDigest类的update()方法类似，可以通过更新一个字节，也可以更新一个字节数组，方法如下：

```
// 更新要由字节签名或验证的数据
public final void update(byte b)
// 使用指定的字节数组更新要签名或验证的数据
public final void update(byte[] data)
```

另一种方式就是根据偏离量做更新处理，方法如下：

```
// 从指定的偏移量开始，使用指定的 byte 数组更新要签名或验证的数据
public final void update(byte[] data, int off, int len)
```

或者使用缓冲方式，方法如下：

```
// 使用指定的 ByteBuffer 更新要签名或验证的数据
public final void update(ByteBuffer data)
```

在完成更新操作后，我们就可以做签名操作了：

```
// 返回所有已更新数据的签名字节
public final byte[] sign()
// 完成签名操作，并得到存储在缓冲区中的签名字节长度
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


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

终于到了验证操作这一步，可通过以下方法完成：

```
// 验证传入的签名，并返回验证结果
public final boolean verify(byte[] signature)
// 从指定的偏移量开始，验证指定的字节数组中传入的签名，并返回验证结果
public final boolean verify(byte[] signature, int offset, int length)
```

此外，Signature提供了以下两种方法来设置和获取算法参数：

```
// 使用指定的参数集初始化此签名引擎
public final void setParameter(AlgorithmParameterSpec params)
// 返回与此签名对象一起使用的参数
public final AlgorithmParameters getParameters()
```

Signature类作为引擎类，同样提供如下两个常用方法：

```
// 返回此签名对象的算法名称
public final String getAlgorithm()
// 返回此签名对象的提供者
public final Provider getProvider()
```

Signature类覆盖了SignatureSpi类，方法如下：

```
// 如果此实现可以复制，则返回副本
public Object clone()
/* 返回此签名对象的字符串表示形式，以提供包括对象状态和所用算法名称在内的信息*/
public String toString()
```

2. 实现示例

代码清单3-11展示了如何使用私钥完成数字签名的操作。

代码清单3-11　数字签名处理

```
// 待做数字签名的原始信息
byte[] data = "Data Signature".getBytes();
// 实例化KeyPairGenerator对象，并指定DSA算法
KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("DSA");
// 初始化KeyPairGenerator对象
keyPairGen.initialize(1024);
// 生成KeyPair对象
KeyPair keyPair = keyPairGen.generateKeyPair();
// 实例化Signature对象
Signature signature = Signature.getInstance(keyPairGen.getAlgorithm());
// 初始化用于签名操作的Signature对象
signature.initSign(keyPair.getPrivate());
// 更新
signature.update(data);
// 获得签名，即字节数组sign
byte[] sign = signature.sign();
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

私钥完成签名，公钥则用于完成验证，方法如下：

```
// 初始化用于验证操作的Signature对象
signature.initVerify(keyPair.getPublic());
// 更新
signature.update(data);
// 获得验证结果
boolean status = signature.verify(sign);
```

在上述示例代码中，如果变量status值为true，则认为验证成功。上述代码稍加改动就可用于数字证书的签名和认证操作。

3.2.14 SignedObject类

SignedObject类是一个用来创建实际运行时对象的类，在检测不到这些对象的情况下，其完整性不会遭受损害。更明确地说，SignedObject包含另外一个Serializable对象，即要签名的对象及其签名，我们可以称之为签名对象。

签名对象是对原始对象的“深层复制”（以序列化形式），一旦生成了副本，对原始对象的进一步操作就不再影响该副本。

```
// 实现对象与数字签名的封装
public final class SignedObject
extends Object
implements Serializable
```

1. 方法详述

签名对象通过以下构造方法完成实例化对象：

```
// 通过任何可序列化对象构造 SignedObject对象
public SignedObject(Serializable object, PrivateKey signingKey, Signature signingEngine)
```

在完成上述实例化操作后，可通过以下方法获得封装后的对象和签名：

```
// 获取已封装的对象
public Object getObject()
// 在已签名对象上按字节数组的形式获取签名
public byte[] getSignature()
```

接着，可以通过公钥和Signature进行验证操作：

```
/* 使用指派的验证引擎，通过给定的验证密钥验证SignedObject中的签名是否为内部存储对象的有效签名*/
public boolean verify(PublicKey verificationKey, Signature verificationEngine)
```

此外，SignedObject还提供了以下方法：

```
// 获取签名算法的名称
String getAlgorithm()
```

2. 实现示例

我们对3.2.13节中的签名验证代码稍作改动，如代码清单3-12所示。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

代码清单3-12 数字签名对象处理

```
// 待做数字签名的原始信息
byte[] data = "Data Signature".getBytes();
// 实例化KeyPairGenerator对象，并指定DSA算法
KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("DSA");
// 初始化KeyPairGenerator对象
keyPairGen.initialize(1024);
// 生成KeyPair对象
KeyPair keyPair = keyPairGen.generateKeyPair();
// 实例化Signature对象
Signature signature = Signature.getInstance(keyPairGen.getAlgorithm());
```

这里与3.2.13节略有不同：

```
// 实例化SignedObject对象
SignedObject s = new SignedObject(data, keyPair.getPrivate(), signature);
```

我们通过另一种方式得到了签名：

```
// 获得签名值
byte[] sign = s.getSignature();
```

验证方式也有所不同：

```
// 验证签名
boolean status = s.verify(keyPair.getPublic(), signature);
```

在上述示例代码中，如果变量status值为true，则认为验证成功。

3.2.15 Timestamp类

Timestamp类用于封装有关签署时间戳的信息，且它是不可更改的。它包括时间戳的日期和时间，以及有关生成和签署时间戳的Timestamping Authority（TSA）的信息。为了区别于一般时间戳（java.sql.Timestamp），我们称之为数字时间戳。

```
// 构建数字时间戳
public final class Timestamp
extends Object
implements Serializable
```

1. 方法详述

构建一个数字时间戳需要提供时间和签名证书路径（CertPath）两个参数，方法如下：

```
// 构造一个Timestamp对象
public Timestamp(Date timestamp, CertPath signerCertPath)
```

获得数字时间戳后的主要目的在于校验给定对象是否与此数字时间戳一致，方法如下：

```
// 比较指定的对象和Timestamp对象是否相同
public boolean equals(Object obj)
```

当然，我们可以通过数字时间戳获得相应的签名证书路径和生成数字时间戳的日期和时间，

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

方法如下：

```
// 返回 Timestamping Authority 的证书路径
public CertPath getSignerCertPath()
// 返回生成数字时间戳时的日期和时间
public Date getTimestamp()
```

此外，数字时间戳覆盖了以下两种方法：

```
// 返回此数字时间戳的散列码值
public int hashCode()
// 返回描述此数字时间戳的字符串
public String toString()
```

2. 实现示例

我们可以通过代码清单3-13所示来构造一个数字时间戳。

代码清单3-13 获得数字时间戳

```
// 构建CertificateFactory对象，并指定证书类型为X.509
CertificateFactory cf = CertificateFactory.getInstance("X509");
// 生成CertPath对象
CertPath cp = cf.generateCertPath(new FileInputStream("D:\\x.cer"));
// 实例化数字时间戳
Timestamp t = new Timestamp(new Date(), cp);
```

关于CertificateFactory类的介绍请参见3.5.1节。

3.2.16 CodeSigner类

CodeSigner类封装了代码签名者的信息，且它是不可变的，我们称之为代码签名。它和数字时间戳（java.security.Timestamp）紧密相连。

```
// 用于构建代码签名
public final class CodeSigner
extends Object
implements Serializable
```

1. 方法详述

CodeSigner类是一个终态类，可以通过其构造方法完成实力化对象：

```
// 构造CodeSigner对象
public CodeSigner(CertPath signerCertPath, Timestamp timestamp)
```

完成实例化对象后，就可以通过以下方法获得其属性：

```
// 返回签名者的CertPath对象
public CertPath getSignerCertPath()
// 返回签名时间戳
public Timestamp getTimestamp()
```

注意，这里的Timestamp是java.security.Timestamp，是用做数字时间戳的Timestamp！

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

获得CodeSigner对象后的最重要的操作就是执行比对，CodeSigner覆盖了equals()方法：

```
// 测试指定对象与此CodeSigner对象是否相等
public boolean equals(Object obj)
```

如果，传入参数不是CodeSigner类的实现，则直接返回false。如果传入参数是CodeSigner类的实现，则比较其Timestamp和CertPath两个属性，此方法的实现如代码清单3-14所示。

代码清单3-14　CodeSigner类equals()方法源代码片段

```
private CertPath signerCertPath;
private Timestamp timestamp;
// ......
public boolean equals(Object obj) {
    if (obj == null || (!(obj instanceof CodeSigner))) {
            return false;
    }
    CodeSigner that = (CodeSigner)obj;
    if (this == that) {
            return true;
    }
    Timestamp thatTimestamp = that.getTimestamp();
    if (timestamp == null) {
            if (thatTimestamp != null) {
                    return false;
            }
    } else {
            if (thatTimestamp == null ||
                    (! timestamp.equals(thatTimestamp))) {
                    return false;
            }
    }
            return signerCertPath.equals(that.getSignerCertPath());
}
```

此外，CodeSigner还覆盖了以下两种方法：

```
// 返回此代码签名者的散列码值
public int hashCode()
// 返回描述此代码签名者的字符串
public String toString()
```

2. 实现示例

我们接着3.2.15节的代码实现，来继续构建一个代码签名，如代码清单3-15所示。

代码清单3-15　验证代码签名

```
// 构建CertificateFactory对象，并指定证书类型为X.509。
CertificateFactory cf = CertificateFactory.getInstance("X509");
// 生成CertPath对象
CertPath cp = cf.generateCertPath(new FileInputStream("D:\\x.cer"));
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
// 实例化Timestamp对象
Timestamp t = new Timestamp(new Date(), cp);
// 实例化CodeSigner对象
CodeSigner cs = new CodeSigner(cp, t);
// 获得比对结果
boolean status = cs.equals(new CodeSigner(cp, t));
```

得到CodeSigner对象后，可以使用它的equals()方法来进行对比。

3.2.17 KeyStore类

KeyStore类被称为密钥库，用于管理密钥和证书的存储。KeyStore 类是个引擎类，它提供一些相当完善的接口来访问和修改密钥仓库中的信息。

```
// 用于管理密钥和证书存储
public class KeyStore
extends Object
```

1. 方法详述

KeyStore类同样需要通过getInstance()工厂方法获得实例化对象。

我们可以通过指定密钥库类型获得实例化对象，方法如下所示：

```
// 返回指定类型的KeyStore对象
public static KeyStore getInstance(String type)
```

另一种方式就是指定密钥库类型时指定该密钥库类型的提供者，方法如下所示：

```
// 返回指定类型的KeyStore对象
public static KeyStore getInstance(String type, Provider provider)
// 返回指定类型的KeyStore对象
public static KeyStore getInstance(String type, String provider)
```

完成实例化操作后，我们将进行以下操作：

```
/* 返回Java安全属性文件中指定的默认密钥库类型；如果不存在此类属性，则返回字符串"jks"（"Java
keystore" 的首字母缩写）*/
public final static String getDefaultType()
```

密钥库类型不区分大小。例如，“JKS”被认为与“jks”相同。除了JKS这种类型以外，还有PKCS12和JCEKS两种类型。JCEKS本身受美国出口限制，所以通常我们可以使用的只有JKS和PKCS12两种类型。但PKCS12这种类型的密钥库管理支持不是很完备，只能够读取该类型的证书，却不能对其进行修改。因此，JKS是最好的选择。

与获取默认密钥库类型方法相对的是如下取得当前密钥库类型的方法：

```
// 返回此密钥库的类型
public final String getType()
```

通过以下两种方法来加载和存储密钥库：

```
// 从给定输入流中加载此密钥库
public final void load(InputStream stream, char[] password)
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
// 将此密钥库存储到给定输出流，并用给定密码保护其完整性
public final void store(OutputStream stream, char[] password)
```

通过以下方法获得密钥库中的条目数：

```
// 获取此密钥库中的条目数
public final int size()
```

我们依旧可以使用如下方法获得提供者：

```
// 返回此密钥库的提供者
public final Provider getProvider()
```

在密钥库中，密钥和证书都是通过别名进行组织的。

可通过以下方法获得密钥库的别名列表，以及确认给定的别名是否在当前密钥库中：

```
// 列出此密钥库的所有别名
public final Enumeration<String> aliases()
// 检查给定别名是否存在于此密钥库中
public final boolean containsAlias(String alias)
```

我们可以通过别名和密码来获得密钥：

```
// 返回与给定别名关联的密钥，并用给定密码来恢复它
public final Key getKey(String alias, char[] password)
```

这里要注意一下，虽然返回的类型是Key接口，但真正获得的是PrivateKey接口的实例。通过该方法将获得该别名在密钥库中对应的私钥。通过别名，我们还可以获得证书或证书链：

```
// 返回与给定别名关联的证书
public final Certificate getCertificate(String alias)
// 返回与给定别名关联的证书链
public final Certificate[] getCertificateChain(String alias)
```

反之，通过证书获得其在密钥库中的别名：

```
// 返回证书与给定证书匹配的第一个密钥库条目的别名
public final String getCertificateAlias(Certificate cert)
```

同样，能通过别名获得其对应条码的创建日期：

```
// 返回给定别名标识的条目的创建日期
public final Date getCreationDate(String alias)
```

通过别名来删除密钥库中与别名相对应的条目：

```
// 删除此密钥库中给定别名标识的条目。
public final void deleteEntry(String alias)
```

我们一直在说“条目”这个词，是因为在密钥库中别名可能对应密钥，有可能对应证书。我们通过以下方法设置别名与密钥和证书的对应关系。

```
// 将给定密钥（受保护的）分配给指定的别名
public final void setKeyEntry(String alias, byte[] key, Certificate[] chain)
// 将给定的密钥分配给给定的别名，并用给定密码保护它
public final void setKeyEntry(String alias, Key key, char[] password,
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
Certificate[] chain)
```

我们也可以使用如下方法将别名与证书绑定：

```
// 将给定可信证书分配给给定别名
public final void setCertificateEntry(String alias, Certificate cert)
```

既然别名可能对应密钥或证书，那么需要用方法来判别：

```
/* 如果给定别名标识的条目是通过调用 setCertificateEntry 或者以 TrustedCertificateEntry
为参数的 setEntry 创建的，则返回 true*/
public final boolean isCertificateEntry(String alias)
/* 如果给定别名标识的条目是通过调用 setKeyEntry 或者以 PrivateKeyEntry 或
SecretKeyEntry 为参数的 setEntry 创建的，则返回 true*/
public final boolean isKeyEntry(String alias)
```

在上述方法的注释中，我们注意到一些Java 5之前所没有的内容。在Java 5以后，KeyStore类中加入了新的内部接口及内部类。

KeyStore.Entry接口：

```
// 用于密钥库项类型的标记接口
public static interface KeyStore.Entry
```

KeyStore.Entry接口是一个空接口，内部没有定义代码，用于类型区分。

KeyStore管理不同类型的条目。每种类型的条目都实现KeyStore.Entry接口。提供了三种基本的KeyStore.Entry实现：

```
// 保存私钥和相应证书链的密钥库项
public static final class KeyStore.PrivateKeyEntry
// 保存秘密密钥的密钥库项
public static final class KeyStore.SecretKeyEntry
// 保存可信的证书的密钥库项
public static final class KeyStore.TrustedCertificateEntry
```

（1）KeyStore.PrivateKeyEntry

我们把KeyStore.PrivateKeyEntry称为私钥项。可通过以下方法获得实例化对象：

```
// 构造带私钥和相应证书链的私钥项
public KeyStore.PrivateKeyEntry(PrivateKey privateKey, Certificate[] chain)
```

获得私钥项对象后，就可以获得其相应的属性：

```
/* 从此私钥项内部的证书链数组首位中获取证书。如果证书的类型是 X.509，返回证书的运行时类型是
X509Certificate*/
public Certificate getCertificate()
// 从此私钥项获取证书链
public Certificate[] getCertificateChain()
// 从此私钥项获取PrivateKey对象
public PrivateKey getPrivateKey()
```

此外，私钥项覆盖了如下方法：

```
// 返回此私钥项的字符串表示形式
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
public String toString()
```

（2）KeyStore.SecretKeyEntry

我们把KeyStore. SecretKeyEntry称为秘密密钥项。可通过以下方法获得实例化对象：

```
// 用秘密密钥构造秘密密钥项
public KeyStore.SecretKeyEntry(SecretKey secretKey)
```

秘密密钥项的主要作用就是保护秘密密钥，可通过如下方法获得秘密密钥：

```
// 从此项中获取SecretKey对象
public SecretKey getSecretKey()
```

此外，秘密密钥项覆盖了如下方法：

```
// 返回此秘密密钥项的字符串表示形式
public String toString()
```

（3）KeyStore.TrustedCertificateEntry

我们把KeyStore.TrustedCertificateEntry称为信任证书项，可通过以下方法获得实例化对象：

```
// 用可信证书构造信任证书项
public KeyStore.TrustedCertificateEntry(Certificate trustedCert)
```

与秘密密钥项相似，信任证书项主要保护受信任的证书，可通过如下方法获得其证书：

```
// 从此信任证书项获取可信证书
public Certificate getTrustedCertificate()
```

此外，秘密密钥项覆盖了如下方法：

```
// 返回此信任证书项的字符串表示形式
public String toString()
```

在理解了上述各种KeyStore.Entry的具体实现后，我们来了解一下密钥库中与KeyStore.Entry相关的方法：

```
// 确定指定别名的密钥库项是否是指定密钥库项的实例或子类。
public final boolean entryInstanceOf(String alias,Class<? extends KeyStore.Entry> entryClass)
```

除了上述内部接口与内部实现外，还有其他内部接口及相应的内部实现。由于内容与本书所涉及的方面关联较少，故不在这里详述。有兴趣的读者朋友可以参考相关API文档。

2. 实现示例

我们通过代码清单3-16来演示如何获得密钥库对象。

代码清单3-16 加载密钥库

```
// 加载密钥库文件
FileInputStream is = new FileInputStream("D:\\.keystore");
// 实例化KeyStore对象
KeyStore ks = KeyStore.getInstance(KeyStore.getDefaultType());
// 加载密钥库，使用密码"password"
```

HZ BOOKS 华章图书

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
ks.load(is, "password".toCharArray());
// 关闭文件流
is.close();
```

得到密钥库对象后，可以获得其别名对应的私钥：

```
// 获得别名为"alias"所对应的私钥
PrivateKey key = (PrivateKey) ks.getKey("alias", "password".toCharArray());
```

或者使用以下私钥项的方式获得私钥：

```
// 获得私钥项
KeyStore.PrivateKeyEntry pkEntry = (KeyStore.PrivateKeyEntry)ks.getEntry("alias",
"password".toCharArray());
// 获得私钥
PrivateKey privateKey = pkEntry.getPrivateKey();
```

有关证书类Certificate请关注3.5.1节。

3.3 javax.crypto包详解

javax.crypto包为加密操作提供类和接口。在上一节中，我们了解了如何实现消息摘要算法、获得对称密钥等API内容。但是，如果缺乏Cipher类，我们将无法完成加密与解密的实现。

在本节内容中，读者将可以完成完整的算法实现，以及安全摘要算法的实现。

3.3.1 Mac类

Mac属于消息摘要的一种，但它不同于一般消息摘要（如MessageDigest提供的消息摘要实现），仅通过输入数据无法获得消息摘要，必须有一个由发送方和接收方共享的秘密密钥才能生成最终的消息摘要——安全消息摘要。安全消息摘要又称消息认证（鉴别）码（Message Authentication Code，Mac）。

在Java 7版本中支持HmacMD5、HmacSHA1、HmacSHA256、HmacSHA384和HmacSHA512算法。

关于Java 7版本中提供的算法信息，请参见本书附录。

```
// 实现创建和验证安全消息摘要的操作
public class Mac
extends Object
implements Cloneable
```

1. 方法详述

Mac与MessageDigest绝大多数方法相同，我们可以通过以下方法获得它的实例：

```
// 返回实现指定摘要算法的Mac对象
public final static Mac getInstance(String algorithm)
```

或者，指定算法名称的同时指定该算法的提供者

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
// 返回实现指定摘要算法的Mac对象
public final static Mac getInstance(String algorithm, Provider provider)
// 返回实现指定摘要算法的Mac对象
public final static Mac getInstance(String algorithm, String provider)
```

目前，Mac类支持HmacMD5、HmacSHA1、HmacSHA256、HmacSHA384、HmacSHA512 5种消息摘要算法。

在获得Mac实例化对象后，需要通过给定的密钥对Mac对象初始化，方法如下：

```
// 用给定的密钥初始化此 Mac
public final void init(Key key)
// 用给定的密钥和算法参数初始化此 Mac
public final void init(Key key, AlgorithmParameterSpec params)
```

这里的密钥Key指的是秘密密钥，请使用该密钥作为init()方法的参数。

Mac类更新摘要方法与MessageDigest类相同，方法如下：

```
// 使用指定的字节更新摘要。
public final void update(byte input)
// 使用指定的字节数组更新摘要。
public final void update(byte[] input)
```

上述方法传入参数不同，前一个传入的是字节，后一个传入的是字节数组。

我们也可以通过输入偏移量的方式做更新操作，方法如下：

```
// 使用指定的字节数组，从指定的偏移量开始更新摘要
public final void update(byte[] input, int offset, int len)
```

当然，我们还可以使用缓冲方式，方法如下：

```
// 使用指定的字节缓冲更新摘要
public final void update(ByteBuffer input)
```

与MessageDigest类相同，更新摘要信息，其参数可以是更新一个字节、一个字节数组甚至是字节数组中的某一段偏移量，也可以是字节缓冲对象。

这些方法的调用顺序不受限制，在向摘要中增加所需数据时可以多次调用。

在完成摘要更新后，我们可以通过以下方法完成摘要操作：

```
// 完成摘要操作
public final byte[] doFinal()
/* 使用指定的字节数组对摘要进行最后更新，然后完成摘要计算，返回消息摘要字节数组*/
public final byte[] doFinal(byte[] input)
// 完成摘要操作，按指定的偏移量将摘要信息保存在字节数组中
public final void doFinal(byte[] output, int outOffset)
```

与MessageDigest类相同，Mac类也有重置方法：

```
// 重置摘要以供再次使用
public final void reset()
```

执行该重置方法等同于创建一个新的Mac实例化对象。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

除了上述方法外，还常用到以下几个方法：

```
/* 返回以字节为单位的摘要长度，如果提供者不支持此操作并且实现是不可复制的，则返回0*/
public final int getMacLength()
// 返回算法名称，如HmacMD5
public final String getAlgorithm()
// 返回此信息摘要对象的提供者
public final Provider getProvider()
```

2. 实现示例

安全消息摘要算法实现也较为简单，如代码清单3-17所示。

代码清单3-17 HmacMD5算法摘要处理

```
// 待做安全消息摘要的原始信息
byte[] input = "MAC".getBytes();
// 初始化KeyGenerator对象，使用HmacMD5算法
KeyGenerator keyGenerator = KeyGenerator.getInstance("HmacMD5");
// 构建SecretKey对象
SecretKey secretKey = keyGenerator.generateKey();
// 构建Mac对象
Mac mac = Mac.getInstance(secretKey.getAlgorithm());
// 初始化Mac对象
mac.init(secretKey);
// 获得经过安全消息摘要后的信息
byte[] output = mac.doFinal(input);
```

3.3.2 KeyGenerator类

KeyGenerator类与KeyPairGenerator类相似，KeyGenerator类用来生成秘密密钥，我们称之为秘密密钥生成器。

```
// 生成用于对称加密算法的秘密密钥，并提供相关信息
public class KeyGenerator
extends Object
```

Java 7版本中提供了Blowfish、AES、DES和DESede等多种对称加密算法实现，以及HmacMD5、HmacSHA1和HmacSHA256等多种安全消息摘要算法实现。

关于Java 7版本中提供的算法信息，请参见附录。

1. 方法详述

与KeyPairGenerator类相似，KeyGenerator类通过如下方法获得实例化对象：

```
// 返回生成指定算法的秘密密钥的KeyGenerator对象
public static final KeyGenerator getInstance(String algorithm)
```

另一种方式就是指定算法名称的同时指定该算法的提供者，方法如下所示：

```
// 返回生成指定算法的秘密密钥的KeyGenerator对象
public static final KeyGenerator getInstance(String algorithm, Provider provider)
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
// 返回生成指定算法的秘密密钥的KeyGenerator对象
public static final KeyGenerator getInstance(String algorithm, String provider)
```

KeyGenerator对象可重复使用，也就是说，在生成密钥后，可以重复使用同一个KeyGenerator对象来生成更多的密钥。

生成密钥的方式有两种：与算法无关的方式和特定于算法的方式。这一点与KeyPairGenerator类生成密钥方式相类似。两者之间的唯一不同是对象的初始化：

- 与算法无关的初始化。所有密钥生成器都具有密钥大小和随机源的概念。KeyGenerator类中有一个init()方法，它带有这两个通用共享类型的参数。还有一个只带有keysize参数的init()方法，它使用具有最高优先级的已安装提供者的SecureRandom实现作为随机源（如果已安装的提供者都不提供SecureRandom实现，则使用系统提供的随机源）。KeyGenerator类还提供一个只带随机源参数的init()方法。 因为调用上述与算法无关的init()方法时未指定其他参数，所以由提供者决定如何处理要与每个密钥关联的特定于算法的参数（如果有）。
- 特定于算法的初始化。在已经存在特定于算法的参数集的情况下，有两个带AlgorithmParameterSpec参数的 init()方法。其中一个方法还有一个SecureRandom参数，而另一个方法将具有高优先级的已安装提供者的SecureRandom实现作为随机源（如果安装的提供者都不提供SecureRandom实现，则使用系统提供的随机源）。

如果客户端没有显式地初始化KeyGenerator（通过调用init()方法），那么每个提供者都必须提供（并记录）默认初始化。

与算法无关的初始化方法如下：

```
// 初始化此KeyGenerator
public final void init(SecureRandom random)
// 初始化此KeyGenerator，使其具有确定的密钥大小
public final void init(int keysize)
// 使用用户提供的随机源初始化此KeyGenerator，使其具有确定的密钥大小
public final void init(int keysize, SecureRandom random)
```

特定于算法的初始化方法如下：

```
// 用指定参数集初始化此KeyGenerator
public final void init(AlgorithmParameterSpec params)
// 用指定参数集和用户提供的随机源初始化此KeyGenerator
public final void init(AlgorithmParameterSpec params, SecureRandom random)
```

完成初始化操作后，我们就可以通过以下方法获得秘密密钥：

```
// 生成一个SecretKey对象
public final SecretKey generateKey()
```

与其他引擎类一样，KeyGenerator类提供如下两种方法：

```
// 返回此秘密密钥生成器对象的算法名称
public final String getAlgorithm()
// 返回此秘密密钥生成器对象的提供者
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
public final Provider getProvider()
```

2. 实现示例

具体的实现示例如下：

```
// 实例化KeyGenerator对象，并指定HmacMD5算法
KeyGenerator keyGenerator = KeyGenerator.getInstance("HmacMD5");
// 生成SecretKey对象
SecretKey secretKey = keyGenerator.generateKey();
```

3.3.3　KeyAgreement类

KeyAgreement类提供密钥协定协议的功能，它同样是一个引擎类。我们称它为密钥协定，将在DH算法实现中使用到它。

```
// 此类提供密钥协定（或密钥交换）协议的功能
public class KeyAgreement
extends Object
```

1. 方法详述

与我们所熟悉的其他引擎类一样，KeyAgreement类需要通过getInstance()工厂方法来获得实例化对象：

```
// 返回实现指定密钥协定算法的KeyAgreement对象
public static KeyAgreement getInstance(String algorithm)
// 返回实现指定密钥协定算法的KeyAgreement对象
public static KeyAgreement getInstance(String algorithm, Provider provider)
// 返回实现指定密钥协定算法的KeyAgreement对象
public static KeyAgreement getInstance(String algorithm, String provider)
```

算法生成器共有两种初始化方式：与算法无关的方式或特定于算法的方式。

获得实例化对象后，需要执行以下初始化方法：

```
/* 用给定密钥初始化此KeyAgreement，给定密钥需要包含此KeyAgreement所需的所有算法参数*/
public void init(Key key)
```

当然，我们也可以指定密钥的同时给出对应的算法参数，方法如下所示：

```
// 用给定密钥和算法参数集初始化此KeyAgreement
public void init(Key key, AlgorithmParameterSpec params)
```

或者，基于上述方式，再加入安全随机数参数，方法如下所示：

```
// 用给定密钥、算法参数集和随机源初始化此KeyAgreement
public void init(Key key, AlgorithmParameterSpec params, SecureRandom random)
```

除上述方式外，我们也可以仅使用密钥和安全随机数两个参数完成初始化操作，方法如下所示：

```
// 用给定密钥和随机源初始化此KeyAgreement
public void init(Key key, SecureRandom random)
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

然后，我们需要调用如下方法执行计划：

```
/* 用给定密钥执行此KeyAgreement的下一个阶段，给定密钥是从此密钥协定所涉及的其他某个参与者那里接收的*/
public Key doPhase(Key key, boolean lastPhase)
```

最后，我们可以获得共享秘密密钥：

```
// 生成共享秘密密钥并在新的缓冲区中返回它
public byte[] generateSecret()
// 生成共享秘密密钥，并将其放入缓冲区 sharedSecret，从 offset（包括）开始
public int generateSecret(byte[] sharedSecret, int offset)
// 创建共享秘密密钥并将其作为指定算法的SecretKey对象
public SecretKey generateSecret(String algorithm)
```

此外，KeyAgreement类还提供了以下常用方法：

```
// 返回此密钥协定对象的提供者
public Provider getProvider()
// 返回此密钥协定对象的算法名称
public String getAlgorithm()
```

2. 实现示例

KeyPairGenerator的实现是离不开DH算法的，如代码清单3-18所示。

代码清单3-18　DH算法密钥对生成

```
// 实例化KeyPairGenerator对象，并指定DH算法
KeyPairGenerator kpg = KeyPairGenerator.getInstance("DH");
// 生成KeyPair对象kp1
KeyPair kp1 = kpg.genKeyPair();
// 生成KeyPair对象kp2
KeyPair kp2 = kpg.genKeyPair();
// 实例化KeyAgreement对象
KeyAgreement keyAgree = KeyAgreement.getInstance(kpg.getAlgorithm());
// 初始化KeyAgreement对象
keyAgree.init(kp2.getPrivate());
// 执行计划
keyAgree.doPhase(kp1.getPublic(), true);
// 生成SecretKey对象
SecretKey secretKey = keyAgree.generateSecret("DES");
```

3.3.4　SecretKeyFactory类

SecretKeyFactory类同样属于引擎类，与KeyFactory类相对应，它用于产生秘密密钥，我们称之为秘密密钥工厂。

```
// 此类表示秘密密钥的工厂
public class SecretKeyFactory
extends Object
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

1. 方法详述

既然SecretKeyFactory类也是一个引擎类，那同样需要通过getInstance()工厂方法来实例化对象。

我们可以通过制定算法名称的方式获得秘密密钥实例化对象，方法如下：

```
// 返回转换指定算法的秘密密钥的SecretKeyFactory对象
public final static SecretKeyFactory getInstance(String algorithm)
```

或者，指定算法名称的同时制定该算法的提供者，方法如下：

```
// 返回转换指定算法的秘密密钥的SecretKeyFactory对象
public final static SecretKeyFactory getInstance(String algorithm, Provider provider)
// 返回转换指定算法的秘密密钥的SecretKeyFactory对象
public final static SecretKeyFactory getInstance(String algorithm, String provider)
```

算法生成器共有两种初始化方式：与算法无关的方式或特定于算法的方式。

得到SecretKeyFactory实例化对象后，我们就可以通过以下方法来生成秘密密钥：

```
// 根据提供的密钥规范（密钥材料）生成SecretKey对象
public final SecretKey generateSecret(KeySpec keySpec)
```

此外，还可以使用如下方法转换秘密密钥：

```
/* 将一个密钥对象（其提供者未知或可能不受信任）转换为此SecretKeyFactory的相应密钥对象*/
public final SecretKey translateKey(SecretKey key)
```

此外，可以使用如下方法获得秘密密钥的密钥规范：

```
// 以输入参数的格式返回给定密钥对象的规范（密钥材料）
public final KeySpec getKeySpec(SecretKey key, Class keySpec)
```

与其他引擎类一样，SecretKeyFactory类提供以下两种方法：

```
// 返回此秘密密钥工厂对象的提供者
public final Provider getProvider()
// 返回此秘密密钥工厂对象的算法名称
public final String getAlgorithm()
```

2. 实现示例

我们通过以下代码获得秘密密钥的密钥编码字节数组：

```
// 实例化KeyGenerator对象，并指定DES算法
KeyGenerator keyGenerator = KeyGenerator.getInstance("DES");
// 生成SecretKey对象
SecretKey secretKey = keyGenerator.generateKey();
// 获得秘密密钥的密钥编码字节数组
byte[] key = secretKey.getEncoded();
```

得到上述密钥编码字节数组后，我们就可以还原其秘密密钥对象：

```
// 由获得的密钥编码字节数组构建DESKeySpec对象
DESKeySpec dks = new DESKeySpec(key);
// 实例化SecretKeyFactory对象
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("DES");
// 生成SecretKey对象
SecretKey secretKey = keyFactory.generateSecret(dks);
```

至此，我们就完成了Java平台中有关密钥构建Java API的学习。在3.3.5节中，我们将详述如何使用Cipher类实现加密与解密算法。

3.3.5 Cipher类

Cipher类为加密和解密提供密码功能。它构成了Java Cryptographic Extension（JCE）框架的核心。在本章的上述内容中，只完成了密钥的处理，并未完成加密与解密的操作。这些核心操作需要通过Cipher类来实现。

```
// 此类为加密和解密提供密码功能
public class Cipher
extends Object
```

Cipher类是一个引擎类，它需要通过getInstance()工厂方法来实例化对象。

我们可以通过指定转换模式的方式获得实例化对象，方法如下所示：

```
// 返回实现指定转换的 Cipher对象
public static Cipher getInstance(String transformation)
```

也可以在制定转换模式的同时制定该转换模式的提供者，方法如下所示：

```
// 返回实现指定转换的 Cipher对象
public static Cipher getInstance(String transformation, Provider provider)
// 返回实现指定转换的 Cipher对象
public static Cipher getInstance(String transformation, String provider)
```

注意这里的参数String transformation，通过如下代码示例：

```
Cipher c = Cipher.getInstance("DES");
```

上述实例化操作是一种最为简单的实现，并没有考虑DES分组算法的工作模式和填充模式，可通过以下方式对其设定：

```
Cipher c = Cipher.getInstance("DES/CBC/PKCS5Padding");
```

参数String transformation的格式是“算法/工作模式/填充模式”，不同的算法支持不同的工作模式以及填充模式。具体内容请参见附录。

在对Cipher对象进行初始化前，我们先来认识如下常量：

```
// 用于将Cipher初始化为解密模式的常量
public final static int DECRYPT_MODE
// 用于将Cipher初始化为加密模式的常量
public final static int ENCRYPT_MODE
```

通过这两个常量来完成用于加密或是解密操作的初始化，可以使用如下这个最为简单也是最为常用的方法：

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
// 用密钥初始化此 Cipher对象
public final void init(int opmode, Key key)
```

或者使用算法参数规范或算法参数来完成初始化：

```
// 用密钥和一组算法参数初始化此Cipher对象
public final void init(int opmode, Key key, AlgorithmParameters params)
// 用密钥和一组算法参数初始化此Cipher对象
public final void init(int opmode, Key key, AlgorithmParameterSpec params)
```

以下三个方法加入了SecureRandom参数：

```
// 用一个密钥、一组算法参数和一个随机源初始化此Cipher对象
public final void init(int opmode, Key key, AlgorithmParameterSpec params,
SecureRandom random)
// 用一个密钥、一组算法参数和一个随机源初始化此Cipher对象
public final void init(int opmode, Key key, AlgorithmParameters params,
SecureRandom random)
// 用密钥和随机源初始化此Cipher对象
public final void init(int opmode, Key key, SecureRandom random)
```

通过以下方法可借助于证书，获取其公钥来完成加密和解密操作：

```
// 用取自给定证书的公钥初始化此Cipher对象
public final void init(int opmode, Certificate certificate)
// 用取自给定证书的公钥和随机源初始化此Cipher对象
public final void init(int opmode, Certificate certificate, SecureRandom random)
```

如果需要多次更新待加密（解密）的数据可使用如下方法。

最为常用的是通过输入给定的字节数组完成更新：

```
/* 继续多部分加密或解密操作（具体取决于此Cipher对象的初始化方式），以处理其他数据部分*/
public final byte[] update(byte[] input)
```

或者通过偏移量的方式完成更新，方法如下所示：

```
/* 继续多部分加密或解密操作（具体取决于此Cipher对象的初始化方式），以处理其他数据部分*/
public final byte[] update(byte[] input, int inputOffset, int inputLen)
```

另外一种方式就是将更新结果输出至参数中，方法如下所示：

```
/* 继续多部分加密或解密操作（具体取决于此Cipher对象的初始化方式），以处理其他数据部分*/
public final int update(byte[] input, int inputOffset, int inputLen, byte[] output)
/* 继续多部分加密或解密操作（具体取决于此Cipher对象的初始化方式），以处理其他数据部分*/
public final int update(byte[] input, int inputOffset, int inputLen, byte[]
output, int outputOffset)
```

当然，我们也可以使用如下缓冲方式：

```
/* 继续多部分加密或解密操作（具体取决于此Cipher对象的初始化方式），以处理其他数据部分*/
public final int update(ByteBuffer input, ByteBuffer output)
```

完成上述数据更新后，直接执行如下方法：

```
// 结束多部分加密或解密操作（具体取决于此Cipher对象的初始化方式）
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
public final byte[] doFinal()
```

如果，加密（解密）操作不需要多次更新数据可以直接执行如下方法：

```
// 按单部分操作加密或解密数据，或者结束一个多部分操作
public final byte[] doFinal(byte[] input)
```

或按以下偏移量的方式完成操作：

```
// 按单部分操作加密或解密数据，或者结束一个多部分操作
public final byte[] doFinal(byte[] input, int inputOffset, int inputLen)
```

以下方式将操作后的结果存于给定的参数中，与上述方法大同小异：

```
// 结束多部分加密或解密操作（具体取决于此 Cipher 的初始化方式）
public final int doFinal(byte[] output, int outputOffset)
```

与上述方法不同的是，以下方法可用于多部分操作，并将操作结果存于给定参数中：

```
// 按单部分操作加密或解密数据，或者结束一个多部分操作
public final int doFinal(byte[] input, int inputOffset, int inputLen, byte[] output)
// 按单部分操作加密或解密数据，或者结束一个多部分操作
public final int doFinal(byte[] input, int inputOffset, int inputLen, byte[]
output, int outputOffset)
```

以下方法提供了一种基于缓冲的处理方式：

```
// 按单部分操作加密或解密数据，或者结束一个多部分操作
public final int doFinal(ByteBuffer input, ByteBuffer output)
```

除了完成数据的加密与解密，Cipher类还提供了对密钥的包装与解包。

我们先来了解一下与密钥包装有关的常量：

```
// 用于将Cipher对象初始化为密钥包装模式的常量
public final static int WRAP_MODE
```

这一常量需要在进行Cipher对象初始化时使用，给出如下示例代码：

```
cipher.init(Cipher.WRAP_MODE, secretKey); //初始化
```

在此之后我们就可以执行包装操作，可使用如下方法：

```
// 包装密钥
public final byte[] wrap(Key key)
```

解包操作需要如下常量执行初始化：

```
// 用于将Cipher初始化为密钥解包模式的常量
public final static int UNWRAP_MODE
```

这个常量同样需要在初始化中执行，给出如下示例代码：

```
cipher.init(Cipher.UNWRAP _MODE, secretKey); //初始化
```

在此之后才能执行解包操作。

我们先来看一下解包方法：

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
// 解包一个以前包装的密钥
public final Key unwrap(byte[] wrappedKey, String wrappedKeyAlgorithm, int wrappedKeyType)
```

上述方法中的参数int wrappedKeyType需要使用如下常量：

```
// 用于表示要解包的密钥为“私钥”的常量
public final static int PRIVATE_KEY
// 用于表示要解包的密钥为“公钥”的常量
public final static int PUBLIC_KEY
// 用于表示要解包的密钥为“秘密密钥”的常量
public final static int SECRET_KEY
```

在执行包装操作时使用的是私钥就使用私钥常量，依此对应。

如果读者对第2章中有关分组加密工作模式的内容还有印象，应该记得文中曾提到过初始化向量。我们可以通过如下方法获得：

```
// 返回新缓冲区中的初始化向量（IV）
public final byte[] getIV()
```

通常，我们有必要通过如下方法来获悉当前转换模式所支持的密钥长度，方法如下所示：

```
// 根据所安装的 JCE 仲裁策略文件，返回指定转换的最大密钥长度
public final static int getMaxAllowedKeyLength(String transformation)
```

分组加密中，每一组都有固定的长度，也称为块，以下方法可以获得相应的块大小：

```
// 返回块的大小（以字节为单位）
public final int getBlockSize()
```

以下方法获得输出缓冲区字节长度：

```
/* 根据给定的输入长度 inputLen（以字节为单位），返回保存下一个 update 或 doFinal 操作结果所需的输出缓冲区长度（以字节为单位）*/
public final int getOutputSize(int inputLen)
```

我们也可以通过如下方法获得该Cipher对象的算法参数相关信息：

```
/* 根据仲裁策略文件，返回包含最大 Cipher 参数值的 AlgorithmParameterSpec 对象*/
public final static AlgorithmParameterSpec getMaxAllowedParameterSpec(String transformation)
// 返回此 Cipher 使用的参数
public final AlgorithmParameters getParameters()
```

此外，Cipher类还提供以下方法：

```
// 返回此 Cipher 使用的豁免（exemption）机制对象
public final ExemptionMechanism getExemptionMechanism()
```

Cipher类作为一个引擎类，同样提供如下方法：

```
// 返回此 Cipher 对象的提供者
public final Provider getProvider()
// 返回此 Cipher 对象的算法名称
public final String getAlgorithm()
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

NullCipher和Cipher是什么关系？

我们会看到在API文档中有一个NullCipher类，它是Cipher的子类，用来验证程序的有效性，并不提供具体的加密和解密实现。在验证程序的时候将会用到它。

可通过如下代码完成密钥的包装和解包操作：

```
// 实例化KeyGenerator对象，并指定DES算法
KeyGenerator keyGenerator = KeyGenerator.getInstance("DES");
// 生成SecretKey对象
SecretKey secretKey = keyGenerator.generateKey();
// 实例化Cipher对象
Cipher cipher = Cipher.getInstance("DES");
```

接下来执行包装操作：

```
// 初始化Cipher对象，用于包装
cipher.init(Cipher.WRAP_MODE, secretKey);
// 包装秘密密钥
byte[] k = cipher.wrap(Key key);
```

得到字节数组k后，可以将其传递给需要解包的一方。

省去上述实例化操作用以下代码示例：

```
// 初始化Cipher对象，用于解包
cipher.init(Cipher.UNWRAP_MODE, secretKey);
// 解包秘密密钥
Key key = cipher.unwrap(k, "DES", Cipher.SECRET_KEY);
```

如果要做加密操作可按参考如下代码：

```
// 初始化Cipher对象，用于加密操作
cipher.init(Cipher.ENCRYPT_MODE, secretKey);
// 加密
byte[] input = cipher.doFinal("DES DATA".getBytes());
```

解密操作与之相对应：

```
// 初始化Cipher对象，用于解密操作
cipher.init(Cipher.DECRYPT_MODE, secretKey);
// 解密
byte[] output = cipher.doFinal(input);
```

3.3.6 CipherInputStream类

CipherInputStream和CipherOutputStream同属Cipher类的扩展，统称为密钥流。按流的输入和输出方式分为密钥输入流和密钥输出流。

```
// 提供密钥输入流
public class CipherInputStream
extends FilterInputStream
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

1. 方法详述

可通过如下构造方法构造实例化对象：

```
// 根据 InputStream 和 Cipher 构造 CipherInputStream对象
public CipherInputStream(InputStream is, Cipher c)
```

CipherInputStream类覆盖了FilterInputStream类的以下方法。

以下是输入流的读操作，与一般FilterInputStream的子类别无二致：

```
// 从该输入流读取下一数据字节
public int read()
// 从该输入流将 b.length 个数据字节读入到字节数组中
public int read(byte[] b)
// 从该输入流将 len 个字节数据读入到字节数组中
public int read(byte[] b, int off, int len)
```

通常，我们可以通过如下方法获知是否还有可读入内容：

```
// 返回不发生阻塞地从此输入流读取的字节数
public int available()
```

或者，直接跳过某些内容，方法如下：

```
// 跳过不发生阻塞地从该输入流读取的字节中的 n 个字节的输入
public long skip(long n)
```

我们可以通过如下方法验证该输入流是否支持标记和重置操作：

```
// 测试该输入流是否支持 mark 和 reset 方法以及哪一种方法确实不受支持
public boolean markSupported()
```

完成操作后，一定要执行关闭流操作，方法如下：

```
// 关闭该输入流并释放任何与该流关联的系统资源
public void close()
```

2. 实现示例

我们通过如下代码来展示如何使用密钥输入流解密文件中的数据。

首先，构建Cipher实例化对象：

```
// 实例化KeyGenerator对象，指定DES算法
KeyGenerator kg = KeyGenerator.getInstance("DES");
// 生成SecretKey对象
SecretKey secretKey = kg.generateKey();
// 实例化Cipher对象
Cipher cipher = Cipher.getInstance("DES");
```

接着从文件中读入数据，然后进行解密操作：

```
// 初始化Cipher对象，用于解密操作
cipher.init(Cipher.DECRYPT_MODE, secretKey);
// 实例化CipherInputStream对象
CipherInputStream cis = new CipherInputStream(new FileInputStream(new
```

HZ BOOKS 华章图书

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
File("secret")), cipher);
// 使用DataInputStream对象包装CipherInputStream对象
DataInputStream dis = new DataInputStream(cis);
// 读出解密后的数据
String output = dis.readUTF();
// 关闭流
dis.close();
```

在这里，我们就能获得解密后的数据了。这个加密数据如何写进文件呢？这将在后面展示。

3.3.7 CipherOutputStream类

CipherOutputStream类与CipherInputStream类相似，称为密钥输出流。

```
// 提供密钥输出流
public class CipherOutputStream
extends FilterOutputStream
```

1. 方法详述

可通过如下构造方法构造实例化对象：

```
// 通过 OutputStream 和 Cipher 构造 CipherOutputStream对象
public CipherOutputStream(OutputStream os, Cipher c)
```

CipherOutputStream类覆盖了FilterOutputStream类的以下方法。

写操作很简单，与一般FilterOutputStream类的子类差别不大，方法如下所示：

```
// 从指定的字节数组中将 b.length 个字节写入此输出流
public void write(byte[] b)
// 将指定的字节数组中从 off 偏移量开始的 len 个字节写入此输出流
public void write(byte[] b, int off, int len)
// 将指定的字节写入此输出流
public void write(int b)
```

完成操作后，一定要对输出流做清空和关闭操作，方法如下：

```
// 强制写出已由封装的密码对象处理的任何缓存输出字节来刷新此输出流
public void flush()
// 关闭此输出流并释放任何与此流关联的系统资源
public void close()
```

2. 实现示例

接3.3.6节的内容，我们通过代码清单3-19所示代码完成文件数据的加密。

代码清单3-19　密钥输出流加密操作

```
// 初始化Cipher对象，用于加密操作
cipher.init(Cipher.ENCRYPT_MODE, secretKey);
// 待加密的原始数据
String input = "1234567890";
// 实例化CipherOutputStream对象
```

HZ BOOKS 华章图书

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
CipherOutputStream cos = new CipherOutputStream(new FileOutputStream(new
File("secret")), cipher);
// 使用DataOuputStream对象包装CipherOutputStream对象
DataOutputStream dos = new DataOutputStream(cos);
// 向输出流写待加密的数据
dos.writeUTF(input);
// 清空流
dos.flush();
// 关闭流
dos.close();
```

3.3.8 SealedObject类

SealedObject类使程序员能够用加密算法创建对象并保护其机密性。

```
// 此类使程序员能够用加密算法创建对象并保护其机密性
public class SealedObject
extends Object
implements Serializable
```

1. 方法详述

通过以下方法可以构建一个实例化对象：

```
// 从序列化对象构造一个 SealedObject
public SealedObject(Serializable object, Cipher c)
```

在给定任何Serializable对象的情况下，程序员可以序列化格式（即“深层复制”）封装原始对象的SealedObject，并使用类似于DES的加密算法密封（加密）其序列化的内容，以保护其机密性。加密的内容以后可以解密（使用相应的算法和正确的解密密钥）和反序列化，并生成原始对象。

注意，该Cipher对象在应用于SealedObject之前必须使用正确的算法、密钥、填充方案等进行完全初始化。

已密封的原始对象可以用以下两种方式恢复：

- 使用采用Cipher对象的getObject()方法。此方法需要一个完全初始化的Cipher对象，用相同的用来密封对象的算法、密钥、填充方案等进行初始化。这样做的好处是解封密封对象的一方不需要知道解密密钥。例如，一方用所需的解密密钥初始化Cipher对象之后，它就会将Cipher对象移交给以后要解封密封对象的另一方。
- 使用采用Key对象的getObject()方法。在此方法中，getObject()方法创建一个用于适当解密算法的Cipher对象，并用给定的解密密钥和存储在密封对象中的算法参数（如果有）对其进行初始化。这样做的好处是解封此对象的一方不需要跟踪用来密封该对象的参数（如IV、初始化向量）。

使用采用Cipher对象的getObject()方法：

```
// 获取原始（封装的）对象
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
public final Object getObject(Cipher c)
```

使用采用Key对象的getObject()方法：

```
// 获取原始（封装的）对象
public final Object getObject(Key key)
// 获取原始（封装的）对象
public final Object getObject(Key key, String provider)
```

此外，提供如下常用方法：

```
// 返回用于密封此对象的算法
public final String getAlgorithm()
```

2. 实现示例

我们通过代码清单3-20展示如何对对象加密。

代码清单3-20　对象加密处理

```
// 待加密的字符串对象
String input = "SealedObject";
// 实例化KeyGenerator对象，使用DES算法
KeyGenerator kg = KeyGenerator.getInstance("DES");
// 创建秘密密钥
SecretKey key = kg.generateKey();
// 实例化用于加密的Cipher对象cipher1
Cipher cipher1 = Cipher.getInstance(key.getAlgorithm());
// 初始化
cipher1.init(Cipher.ENCRYPT_MODE, key);
// 构建SealedObject对象
SealedObject sealedObject = new SealedObject(input, cipher1);
// 实例化用于解密的Cipher对象cipher2
Cipher cipher2 = Cipher.getInstance(key.getAlgorithm());
// 初始化
cipher2.init(Cipher.DECRYPT_MODE, key);
// 获得解密后的字符串对象
String output = (String) sealedObject.getObject(cipher2);
```

3.4　java.security.spec包和javax.crypto.spec包详解

java.security.spec包和javax.crypto.spec包都提供了密钥规范和算法参数规范的类和接口。获得密钥规范后，我们将有机会还原密钥对象。本节将详述KeySpec接口及其实现。

3.4.1　KeySpec和AlgorithmParameterSpec接口

KeySpec和AlgorithmParameterSpec是java.security.spec包中两个较为重要的接口。这两个接口本身都是空接口，仅用于将所有参数规范分组，并为其提供类型安全。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

1. KeySpec

此接口不包含任何方法或常量。它仅用于将所有密钥规范分组，并为其提供类型安全。所有密钥规范都必须实现此接口。

```
// 组成加密密钥的密钥内容的（透明）规范
public interface KeySpec
```

KeySpec的抽象实现类（EncodedKeySpec）构建了用于构建公钥规范和私钥规范的两个实现（X509EncodedKeySpec用于构建公钥规范，PKCS8EncodedKeySpec用于构建私钥规范）。

SecretKeySpec接口是KeySpec的实现类，用于构建秘密密钥规范。

2. AlgorithmParameterSpec

此接口不包含任何方法或常量。它仅用于将所有参数规范分组，并为其提供类型安全。所有参数规范都必须实现此接口。

```
// 加密参数的（透明）规范
public interface AlgorithmParameterSpec
```

AlgorithmParameterSpec接口有很多的子接口和实现类，用于特定于算法的初始化。使用起来也很方便，只需要使用指定参数填充构造方法即可获得一个实例化对象。我们以DSAParameterSpec为例：

```
// 此类指定用于DSA算法的参数的集合
public class DSAParameterSpec
extends Object
implements AlgorithmParameterSpec, DSAParams
```

通过以下方法获得实例化对象：

```
// 创建一个具有指定参数值的新的DSAParameterSpec
public DSAParameterSpec(BigInteger p, BigInteger q, BigInteger g)
```

在上述代码中，参数p为素数、q为子素数、g为基数。通过给定的数学参数构造实例化对象。

获得对象后可通过如下方法获得其属性值：

```
// 返回基数g
publi c BigInteger getG()
// 返回素数p
public BigInteger getP()
// 返回子素数q
public BigInteger getQ()
```

此外，还有很多AlgorithmParameterSpec接口的实现，与DSAParameterSpec类的使用方式相似，有兴趣的读者可以查看相应的API文档。

3.4.2 EncodedKeySpec类

EncodedKeySpec类用编码格式来表示公钥和私钥，我们称之为编码密钥规范。编码密钥规范的实现子类很多，在这里不一一详述。本书将详细介绍最为常用的用于表示公钥和私钥的两

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

个实现类。

X509EncodedKeySpec和PKCS8EncodedKeySpec两个类均为EncodedKeySpec的子类，X509EncodedKeySpec类用于转换公钥编码密钥，PKCS8EncodedKeySpec类用于转换私钥编码密钥。

```
// 此类用编码格式表示公钥或私钥
public abstract class EncodedKeySpec
extends Object
implements KeySpec
```

以下是该类的方法，我们将通过其两个重要的子类来详述如何使用：

```
// 根据给定的编码密钥创建一个新的编码密钥规范
public EncodedKeySpec(byte[] encodedKey)
// 返回编码密钥
public byte[] getEncoded()
// 返回与此密钥规范相关联的编码格式的名称
public abstract  String getFormat()
```

1. X509EncodedKeySpec

X509EncodedKeySpec类继承EncodedKeySpec类，以编码格式来表示公钥。

X509EncodedKeySpec类使用X.509标准作为密钥规范管理的编码格式，该类的命名由此得来。

```
// 用编码格式表示公用
public class X509EncodedKeySpec
extends EncodedKeySpec
```

（1）方法详述

可通过如下方法实例化对象：

```
// 根据给定的编码密钥创建一个新的 X509EncodedKeySpec
public X509EncodedKeySpec(byte[] encodedKey)
```

以下两种方法均依照X.509标准：

```
// 返回按照X.509标准进行编码的密钥的字节
public byte[] getEncoded()
//返回与此密钥规范相关联的编码格式的名称。将得到字符串"X.509"
public String getFormat()
```

（2）实现示例

首先，我们通过如下代码获得密钥对：

```
// 实例化KeyPairGenerator对象，并指定DSA算法
KeyPairGenerator keygen = KeyPairGenerator.getInstance("DSA");
// 初始化KeyPairGenerator对象
keygen.initialize(1024);
// 生成KeyPair对象
KeyPair keys = keygen.genKeyPair();
```

HZ BOOKS 华章图书

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

其次，我们获得公钥的密钥字节数组：

```
// 获得公钥密钥字节数组
byte[] publicKeyBytes = keys.getPublic().getEncoded();
```

最后，我们将获得的公钥密钥字节数组再转换为公钥对象：

```
// 实例化X509EncodedKeySpec对象
X509EncodedKeySpec keySpec = new X509EncodedKeySpec(publicKeyBytes);
// 实例化KeyFactory对象，并指定DSA算法
KeyFactory keyFactory = KeyFactory.getInstance("DSA");
// 获得PublicKey对象
PublicKey publicKey = keyFactory.generatePublic(keySpec);
```

私钥对象的转换与此大致相同。

2. PKCS8EncodedKeySpec

PKCS8EncodedKeySpec类继承EncodedKeySpec类，以编码格式来表示私钥。

PKCS8EncodedKeySpec类使用PKCS#8标准作为密钥规范管理的编码格式，该类的命名由此得来。

```
// 用编码格式来表示私钥
public class PKCS8EncodedKeySpec
extends EncodedKeySpec
```

（1）方法详述

可通过如下方法实例化对象：

```
// 根据给定的编码密钥创建一个新的 PKCS8EncodedKeySpec
public PKCS8EncodedKeySpec(byte[] encodedKey)
```

以下两种方法均依照PKCS #8标准：

```
// 返回按照 PKCS#8标准编码的密钥字节
public byte[] getEncoded()
// 返回与此密钥规范相关联的编码格式的名称。将得到字符串"PKCS#8"
public String getFormat()
```

（2）实现示例

下面根据前面介绍的实现示例来展示如何获得私钥对象。

通过获得的密钥对，我们来获得私钥密钥编码字节数组：

```
// 获得私钥密钥字节数组
byte[] privateKeyBytes = keys.getPrivate().getEncoded();
```

最后，我们将获得私钥对象：

```
// 实例化PKCS8EncodedKeySpec对象
PKCS8EncodedKeySpec keySpec = new PKCS8EncodedKeySpec(privateKeyBytes);
// 实例化KeyFactory对象，并指定DSA算法
KeyFactory keyFactory = KeyFactory.getInstance("DSA");
// 获得PrivateKey对象
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
PrivateKey privateKey = keyFactory.generatePrivate (keySpec);
```

X509EncodedKeySpec和PKCS8EncodedKeySpec两个类在加密解密环节中经常会用到。密钥很可能会以二进制方式存储于文件中，由程序来读取。这时候，就需要通过这两个类将文件中的字节数组读出转换为密钥对象。

3.4.3 SecretKeySpec类

SecretKeySpec类是KeySpec接口的实现类，用于构建秘密密钥规范。可根据一个字节数组构造一个SecretKey，而无须通过一个（基于provider的）SecretKeyFactory。

```
// 此类以与provider无关的方式指定一个密钥
public class SecretKeySpec
extends Object
implements KeySpec, SecretKey
```

此类仅对能表示为一个字节数组并且没有任何与之相关联的密钥参数的原始密钥有用，如DES或Triple DES密钥。

1. 方法详述

可以通过下述方法构建一个实例化对象：

```
/* 根据给定的字节数组构造一个密钥，使用key中的始于且包含offset的前len个字节*/
public SecretKeySpec(byte[] key, int offset, int len, String algorithm)
// 根据给定的字节数组构造一个密钥
public SecretKeySpec(byte[] key, String algorithm)
```

SecretKeySpec还覆盖了以下方法：

```
// 测试给定对象与此对象的相等性
public boolean equals(Object obj)
// 算此对象的散列码值
public int hashCode()
```

此外，SecretKeySpec还提供了如下方法：

```
// 返回与此密钥相关联的算法的名称
public String getAlgorithm()
// 返回此密钥的密钥内容
public byte[] getEncoded()
// 返回此密钥编码格式的名称
public String getFormat()
```

2. 实现示例

我们先获得RC2算法的密钥字节数组：

```
// 实例化KeyGenerator对象，并指定RC2算法
KeyGenerator kg = KeyGenerator.getInstance("RC2");
// 生成SecretKey对象
SecretKey secretKey = kg.generateKey();
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
// 获得密钥编码字节数组
byte[] key = secretKey.getEncoded();
```

在得到密钥编码字节数组后，我们将通过如下方法还原秘密密钥对象：

```
// 实例化SecretKey对象。
SecretKey secretKey = new SecretKeySpec(key, "RC2");
```

3.4.4 DESKeySpec类

DESKeySpec类与SecretKeySpec类都是提供秘密密钥规范的实现类。不同之处在于，DESKeySpec类指定了DES算法，SecretKeySpec类则是兼容所有对称加密算法。

DESKeySpec类有很多的同胞，例如，DESedeKeySpec类提供了三重DES加密算法的密钥规范；PBEKeySpec类提供了PBE算法的密钥规范。我们以DESKeySpec类为代表，对其同胞类做相应说明。

```
// 此类指定一个DES密钥
public class DESKeySpec
extends Object
implements KeySpec
```

1. 方法详述

可以通过如下构造方法获得实例化对象：

```
/* 创建一个DESKeySpec对象，使用key中的前8个字节作为DES密钥的密钥内容*/
public DESKeySpec(byte[] key)
/* 创建一个DESKeySpec对象，使用key中始于且包含offset的前8个字节作为DES-EDE密钥的密钥内容。*/
public DESKeySpec(byte[] key, int offset)
```

在上述方法中的8个字节的定义就来源于下面这个常量定义：

```
// 定义以字节为单位的DES密钥长度的常量（8）
public static int DES_KEY_LEN
```

此外，DESKeySpec类还提供了如下方法：

```
// 返回DES密钥内容
public byte[] getKey()
/* 确定给定的始于且包含offset的DES密钥内容是否是奇偶校验的（parity-adjusted）*/
public static boolean isParityAdjusted(byte[] key, int offset)
// 确定给定的DES密钥内容是否是全弱或者半弱的
public static boolean isWeak(byte[] key, int offset)
```

2. 实现示例

我们修改3.4.3节的内容来演示如何使用DESKeySpec类还原密钥对象。

先获得DES算法的密钥字节数组：

```
// 实例化KeyGenerator对象，并指定DES算法
KeyGenerator kg = KeyGenerator.getInstance("DES");
// 生成SecretKey对象
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
SecretKey secretKey = kg.generateKey();
// 获得密钥编码字节数组
byte[] key = secretKey.getEncoded();
```

在得到密钥编码字节数组后，我们将通过如下方法还原秘密密钥对象：

```
// 指定DES算法，还原SecretKey对象
SecretKey secretKey = new SecretKeySpec(key, "DES");
```

如果使用DESKeySpec该如何做呢？在得到了密钥编码之后，可通过如下代码来实现：

```
// 实例化DESKeySpec对象，获得DES秘密密钥规范
DESKeySpec dks = new DESKeySpec(key);
// 实例化SecretKeyFactory对象，并指定DES算法
SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("DES");
// 获得SecretKey对象
SecretKey secretKey = keyFactory.generateSecret(dks);
```

如果是三重DES算法，该怎么做呢？除了将原来指明为“DES”算法的位置替换为“DESede”外，还需要把DESKeySpec类换为DESedeKeySpec类。示例代码如下：

```
// 实例化，并指定DESede算法
KeyGenerator kg = KeyGenerator.getInstance("DESede");
// 生成SecretKey对象
SecretKey secretKey = kg.generateKey();
// 获得密钥编码字节数组
byte[] key = secretKey.getEncoded();
```

得到密钥编码之后，可通过如下代码来实现：

```
// 实例化DESedeKeySpec对象，获得DESede秘密密钥规范
DESedeKeySpec dks = new DESedeKeySpec(key);
// 实例化SecretKeyFactory对象，并指定DESede算法
SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("DESede");
// 获得SecretKey对象
SecretKey secretKey = keyFactory.generateSecret(dks);
```

其他对称秘密密钥规范的使用与上述代码实现大致相同，有兴趣的读者可以参考相应的API内容。

3.5 java.security.cert包详解

java.security.cert包提供证书解析和管理、证书撤销列表（CRL）和证书路径的类和接口。作为Java加密与解密实现的扩展。本节着重详述Certificate类及其子类和CertificateFactory类的使用。作为补充，本节还将详述有关证书撤销的CRL类及其子类和用于证书路径的CertPath类的使用。对证书有兴趣的读者可以参考相应的Java API内容。

3.5.1 Certificate类

Certificate类是一个用于管理证书的抽象类。证书有多种类型，如X.509证书、PGP证书和

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

SDSI证书，并且它们都以不同的方式存储并存储不同的信息，但却都可以通过继承Certificate类来实现它们。

```
// 管理各种身份证书的抽象类
public abstract class Certificate
extends Object
implements Serializable
```

Certificate类提供3种基本操作，要求子类实现如下各项：

```
// 返回此证书的编码形式
public abstract byte[] getEncoded()
// 验证是否已使用与指定公钥相应的私钥签署了此证书
public abstract void verify(PublicKey key)
// 验证是否已使用与指定公钥相应的私钥签署了此证书
public abstract void verify(PublicKey key, String sigProvider)
// 从此证书中获取公钥
public abstract PublicKey getPublicKey()
```

Certificate类还要求子类必须实现以下方法：

```
// 返回此证书的字符串表示形式
public abstract String toString()
```

通过如下方法得到此证书的类型：

```
// 返回此证书的类型。如X.509、PGP和SDSI
public final String getType()
```

此外，Certificate类覆盖了以下方法：

```
// 根据此证书的编码形式返回该证书的散列码值
public int hashCode()
// 比较此证书与指定对象的相等性
public boolean equals(Object other)
```

Certificate类有一个抽象子类——X509Certificate类，我们将在后面几节中介绍它。

3.5.2 CertificateFactory类

CertificateFactory类是一个引擎类，我们称之为证书工厂，可以通过它将证书导入程序中。

```
/* 此类定义了用于从相关的编码中生成证书、证书路径(CertPath)和证书撤销列表(CRL)对象的
CertificateFactory功能*/
public class CertificateFactory
extends Object
```

1. 方法详述

可通过以下getInstance()工厂方法获得实例化对象：

```
// 返回实现指定证书类型的CertificateFactory对象
public final static CertificateFactory getInstance(String type)
// 返回指定证书类型的CertificateFactory对象
```

HZ BOOKS 华章图书

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
public final static CertificateFactory getInstance(String type, Provider provider)
// 返回指定证书类型的CertificateFactory对象
public final static CertificateFactory getInstance(String type, String provider)
```

可以通过CertificateFactory类生成证书：

```
// 生成一个证书对象，并使用从输入流inStream中读取的数据对它进行初始化
public final Certificate generateCertificate(InputStream inStream)
// 返回从给定输入流inStream中读取的证书的集合视图（可能为空）
public final Collection<? extends Certificate> generateCertificates(InputStream inStream)
```

也可以通过CertificateFactory类生成证书路径。

以下方法可以通过输入流获得证书路径：

```
/* 生成一个CertPath对象，并使用从InputStream inStream中读取的数据对它进行初始化*/
public final CertPath generateCertPath(InputStream inStream)
/* 生成一个CertPath对象，并使用从InputStream inStream中读取的数据对它进行初始化*/
public final CertPath generateCertPath(InputStream inStream, String encoding)
```

我们也可以通过如下方法生成一个正式路径：

```
/* 生成一个CertPath对象，并使用一个Certificate的List对它进行初始化*/
public final CertPath generateCertPath(List<? extends Certificate>certificates)
```

还可以通过CertificateFactory类生成证书撤销列表：

```
/* 生成一个证书撤销列表(CRL)对象，并使用从输入流inStream中读取的数据对它进行初始化*/
public final CRL generateCRL(InputStream inStream)
/* 返回从给定输入流inStream中读取的CRL的集合视图（可能为空）*/
public final Collection<? extends CRL> generateCRLs(InputStream inStream)
```

此外，CertificateFactory类还提供了以下方法：

```
/* 返回此CertificateFactory支持的CertPath编码的迭代器，默认编码方式优先*/
public final Iterator<String> getCertPathEncodings()
// 返回此CertificateFactory的提供者
public final Provider getProvider()
// 返回与此CertificateFactory相关联的证书类型的名称
public final String getType()
```

2. 实现示例

如果我们已知待载入证书的类型，就可通过代码清单3-21获得证书对象。

代码清单3-21 加载证书

```
// 实例化，并指明证书类型为“X.509”
CertificateFactory certificateFactory = CertificateFactory.getInstance("X.509");
// 获得证书输入流
FileInputStream in = new FileInputStream("D:\\x.keystore");
// 获得证书
Certificate certificate = certificateFactory.generateCertificate(in);
// 关闭流
in.close();
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

3.5.3 X509Certificate类

X509Certificate类是Certificate类的子类，它同样是一个抽象类。

```
// X.509证书的抽象类。此类提供了一种访问X.509证书所有属性的标准方式
public abstract class X509Certificate
extends Certificate
implements X509Extension
```

1. 方法详述

得到X.509类型的证书对象后，我们最先要做的事情就是校验证书是否有效。

证书的有效期是一个区间范围，也就是起止时间，可用以下两个方法校验：

```
// 获取证书有效期的notAfter日期
public abstract Date getNotAfter()
// 获取证书有效期的 notBefore日期
public abstract Date getNotBefore()
```

除了上述两种方法，还可以校验给定日期是否处于证书的有效期内：

```
// 检查给定的日期是否处于证书的有效期内
public abstract void checkValidity(Date date)
```

下述这个方法就更为简单了，它将校验当前时间是否处于证书的有效期内：

```
// 检查证书目前是否有效
public abstract void checkValidity()
```

除了上述校验功能外，还可通过以下方法获得证书的相应属性。

我们可以通过如下方法获得一些简单的证书基本信息：

```
// 获取证书的version（版本号）值，如1、2或3
public abstract int getVersion()
// 获取证书的serialNumber值
public abstract BigInteger getSerialNumber()
/* 从关键BasicConstraints扩展(OID = 2.5.29.19)中获取证书的限制路径长度*/
public abstract int getBasicConstraints()
```

以下方法可以获得证书中KeyUsage的相关信息：

```
/* 获取一个表示KeyUsage扩展(OID = 2.5.29.15)的各个位的boolean数组*/
public abstract boolean[] getKeyUsage()
/* 获取一个不可修改的String列表，表示已扩展的密钥使用扩展(OID = 2.5.29.37)中
ExtKeyUsageSyntax字段的对象标识符（OBJECT IDENTIFIER）*/
public List<String> getExtendedKeyUsage()
```

我们可以通过如下方法获得证书的发布者的相关信息：

```
/* 从IssuerAltName扩展(OID = 2.5.29.18)中获取一个发布方替换名称的不可变集合*/
public Collection<List<?>> getIssuerAlternativeNames()
// 获取证书的issuerUniqueID值
public abstract boolean[] getIssuerUniqueID()
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
// 以X500Principal的形式返回证书的发布方（发布方标识名）值
public X500Principal getIssuerX500Principal()
```

以下方法可以获得证书主体的一些相关信息：

```
/* 从SubjectAltName扩展(OID = 2.5.29.17)中获取一个主体替换名称的不可变集合*/
public Collection<List<?>> getSubjectAlternativeNames()
// 获取证书的subjectUniqueID值
public abstract boolean[] getSubjectUniqueID()
// 以X500Principal的形式返回证书的主体（主体标识名）值
public X500Principal getSubjectX500Principal()
```

我们也可以获得证书的DER编码的二进制信息，方法如下：

```
// 从此证书中获取以DER编码的证书信息，即tbsCertificate
public abstract byte[] getTBSCertificate()
```

从程序设计的角度来讲，我们常用到以下这些方法：

```
// 获取证书签名算法的签名算法名
public abstract String getSigAlgName()
// 获取证书的签名算法OID字符串
public abstract String getSigAlgOID()
// 从此证书的签名算法中获取DER编码形式的签名算法参数
public abstract byte[] getSigAlgParams()
```

在某些时候，我们更关注证书的签名值，方法如下：

```
// 获取证书的signature值（原始签名位）
public abstract byte[] getSignature()
```

2. 实现示例

我们通过以下代码清单3-22来展示如何通过密钥库获得证书。

代码清单3-22　获得证书签名

```
// 加载密钥库文件
FileInputStream is = new FileInputStream("D:\\x.keystore");
// 实例化KeyStore对象
KeyStore ks = KeyStore.getInstance("JKS");
// 加载密钥库
ks.load(is, "password".toCharArray());
// 关闭文件输入流
is.close();
// 获得X.509类型证书
X509Certificate x509Certificate = (X509Certificate) ks.getCertificate("alias");
// 通过证书标明的签名算法构建Signature对象
Signature signature = Signature.getInstance(x509Certificate.getSigAlgName());
```

3.5.4　CRL类

证书可能会由于各种原因失效，如由于申请证书的请求有问题或者用户使用该证书做了非

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

法操作，这时证书将立即被置为无效。将证书置为无效的结果就是产生CRL（证书撤销列表）。CA负责发布CRL，CRL中列出了该CA已经撤销的证书。验证证书时，首先需要查询此列表，然后再考虑接受证书的合法性。

CRL类作为证书抽象列表的抽象类，可通过扩展该抽象类定义专门的CRL类型。

```
// 此类是具有不同格式但很常用的证书撤销列表(CRL)的抽象
public abstract class CRL
extends Object
```

1. 方法详述

CRL类提供了获取CRL类型的方法：

```
// 返回此CRL的类型
public String getType()
```

此外，要求其子类必须实现以下方法：

```
/* 检查给定的证书是否在此CRL中。如果给定的证书在此CRL中，则返回true，否则返回false*/
public abstract boolean isRevoked(Certificate cert)
// 返回此CRL的字符串表示形式
public abstract String toString()
```

其中，isRevoked()方法是我们最为常用的方法。

2. 实现示例

CRL类的实例可通过代码清单3-23方式获得。

代码清单3-23 获得证书撤销列表

```
// 实例化，并指明证书类型为"X.509"
CertificateFactory certificateFactory = CertificateFactory.getInstance("X.509");
// 获得证书输入流
FileInputStream in = new FileInputStream("D:\\x.keystore");
// 获得证书撤销列表
CRL crl = certificateFactory.generateCRL(in);
// 关闭流
in.close();
```

3.5.5 X509CRLEntry类

X509CRLEntry类可用于撤销证书。

```
// 用于CRL（证书撤销列表）中已撤销证书的抽象类
public abstract class X509CRLEntry
extends Object
implements X509Extension
```

X509CRLEntry类实现了以下方法：

```
// 比较此CRL项与给定对象的相等性
public boolean equals(Object other)
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
// 根据此CRL项的编码形式返回该CRL项的散列码值
public int hashCode()
```

此外，要求子类实现如下方法。

获得证书撤销列表实体的DER编码二进制信息的方法如下：

```
// 返回此CRL Entry的ASN.1 DER编码形式，即内部序列值
public abstract byte[] getEncoded()
```

我们可能最为关心的是下面几个方法：

```
// 获取此X509CRLEntry的撤销日期revocationDate
public abstract Date getRevocationDate()
// 获取此X509CRLEntry的序列号userCertificate
public abstract BigInteger getSerialNumber()
// 如果此CRL项有扩展，则返回true
public abstract boolean hasExtensions()
```

子类还必须覆盖Object的下述方法：

```
// 返回此CRL项的字符串表示形式
public abstract String toString()
```

在Java 5以后，X509CRLEntry加入了如下方法，但自身并无实现，返回值为null：

```
// 获取此项所描述的X509Certificate的发布方
public X500Principal getCertificateIssuer()
```

3.5.6 X509CRL类

X509CRL类作为CRL类的子类，已标明了类型为X.509的CRL。

```
/* X.509证书撤销列表(CRL)的抽象类。CRL是标识已撤销证书的时间戳列表。它由证书颁发机构(CA)签署
并且可在公共存储库中随意使用*/
public abstract class X509CRL
extends CRL
implements X509Extension
```

1. 方法详述

X509CRL类覆盖了以下两个方法：

```
// 比较此CRL与给定对象的相等性
public boolean equals(Object other)
// 根据此CRL的编码形式返回该CRL的散列码值
public int hashCode()
```

在获得X509CRL实例化对象后，我们可以通过以下方法获得相应属性。

我们可以通过如下方法获得版本信息：

```
// 获取CRL的version（版本号）值
public abstract int getVersion()
```

可以通过如下方法获得DER编码的二进制信息：

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
// 返回此CRL的ASN.1 DER编码形式
public abstract byte[] getEncoded()
// 从此CRL中获取以DER编码的CRL信息，即tbsCertList。
public abstract byte[] getTBSCertList()
```

以下是和时间信息有关的方法：

```
// 获取CRL的thisUpdate日期
public abstract Date getThisUpdate()
// 获取CRL的nextUpdate日期
public abstract Date getNextUpdate()
```

以下是和签名相关的方法：

```
// 获取CRL签名算法的签名算法名
public abstract String getSigAlgName()
// 获取CRL的签名算法OID字符串
public abstract String getSigAlgOID()
// 获取此CRL的签名算法中DER编码形式的签名算法参数
public abstract byte[] getSigAlgParams()
```

通过上述方法，我们能够构建一个数字签名对象。

我们可以通过如下方法获得数字签名值：

```
// 获取CRL的signature值（原始签名位）
public abstract byte[] getSignature()
```

可以通过以下方法获得X509CRLEntry的实例：

```
// 获取具有给定证书serialNumber的CRL项（如果有）
public abstract X509CRLEntry getRevokedCertificate(BigInteger serialNumber)
// 获取此CRL中的所有项
public abstract Set<? extends X509CRLEntry> getRevokedCertificates()
```

在Java 5中，X509CRL类实现了以下方法，用以获得X509CRLEntry的实例：

```
// 获取给定证书的CRL项（如果有）
public X509CRLEntry getRevokedCertificate(X509Certificate certificate)
```

同时，我们可以通过以下方法校验CRL：

```
// 验证是否已使用与给定公钥相应的私钥签署了此CRL
public abstract void verify(PublicKey key)
// 验证是否已使用与给定公钥相应的私钥签署了此CRL
public abstract void verify(PublicKey key, String sigProvider)
```

此外，X509CRL类还提供了以下方法：

```
// 以X500Principal的形式返回CRL的发布方（发布方标识名）值
public X500Principal getIssuerX500Principal()
```

2. 实现示例

我和可以通过代码清单3-24获得证书撤销列表实体。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

代码清单3-24 获得证书撤销列表实体

```
// 实例化，并指明证书类型为"X.509"
CertificateFactory certificateFactory = CertificateFactory.getInstance("X.509");
// 获得证书输入流
FileInputStream in = new FileInputStream("D:\\x.keystore");
// 获得证书
X509Certificate certificate = (X509Certificate) certificateFactory.generateCertificate(in);
// 获得证书撤销列表
X509CRL x509CRL = (X509CRL) certificateFactory.generateCRL(in);
// 获得证书撤销列表实体
X509CRLEntry x509CRLEntry = x509CRL.getRevokedCertificate(certificate);
// 关闭流
in.close();
```

3.5.7 CertPath类

CertPath类是一个抽象类，定义了常用于所有CertPath的方法。其子类可处理不同类型的证书（X.509、PGP等）。

所有CertPath对象都包含类型、Certificate列表及其支持的一种或多种编码。由于CertPath类是不可变的，所以构造CertPath后无法以任何外部可见的方式更改它。此规定适用于此类的所有公共字段和方法，以及由子类添加或重写的所有公共字段和方法。

```
// 不可变的证书序列（证书路径）
public abstract class CertPath
extends Object
implements Serializable
```

1. 方法详述

CertPath类要求子类实现如下方法。

我们可以通过以下方法获得该证书路径的证书列表：

```
// 返回此证书路径中的证书列表
public abstract List<? extends Certificate> getCertificates()
```

以下方法可以获得证书路径二进制信息和编码信息：

```
// 返回此证书路径的编码形式，使用默认的编码
public abstract byte[] getEncoded()
// 返回此证书路径的编码形式，使用指定的编码
public abstract byte[] getEncoded(String encoding)
// 返回此证书路径支持的编码的迭代，默认编码方式优先
public abstract Iterator<String> getEncodings()
```

CertPath类实现了以下方法获得证书类型：

```
// 返回此证书路径中的Certificate类型，如X.509
public String getType()
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

此外，CertPath类覆盖了以下方法：

```
// 比较此证书路径与指定对象的相等性
public boolean equals(Object other)
// 返回此证书路径的散列码
public int hashCode()
// 返回此证书路径的字符串表示形式
public String toString()
```

2. 实现示例

我们可以通过代码清单3-25获得证书链。

代码清单3-25 获得证书链

```
// 实例化CertificateFactory对象，并指明证书类型为"X.509"。
CertificateFactory certificateFactory = CertificateFactory.getInstance("X.509");
// 获得证书输入流
FileInputStream in = new FileInputStream("D:\\x.keystore");
// 获得CertPath对象
CertPath certPath = certificateFactory.generateCertPath(in);
// 关闭流
in.close();
```

CertPath类作为证书链，它的操作离不开CertPathBuilder类和CertPathValidator类。有兴趣的读者请参考相应的Java API内容。

3.6 javax.net.ssl包详解

javax.net.ssl包提供用于安全套接字包的类。如果读者对构建密钥库、信任库管理及构建安全网络有兴趣，不妨仔细阅读本小节的内容。读者通过阅读这一节的内容，可以了解构建密钥库、信任库，并由此组建基于HTTPS的加密网络通信实现的知识；也可以了解通过基于SSLSocket模式获取数字证书的知识。

3.6.1 KeyManagerFactory类

KeyManagerFactory类是一个引擎类，它用来管理密钥，称为密钥管理工厂。

```
/* 此类充当基于密钥内容源的密钥管理器的工厂。每个密钥管理器管理特定类型的、由安全套接字所使用的密
钥内容。密钥内容是基于KeyStore和/或提供者特定的源*/
public class KeyManagerFactory
extends Object
```

1. 方法详述

KeyManagerFactory类是引擎类，自然少不了通过getInstance()工厂方法完成对象实例化。

我们可以通过指定算法名称获得实例化对象，方法如下所示：

```
// 返回充当密钥管理器工厂的KeyManagerFactory对象
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
public final static KeyManagerFactory getInstance(String algorithm)
```

或者，指定算法名称的同时指定该算法的提供者：

```
// 返回充当密钥管理器工厂的KeyManagerFactory对象
public final static KeyManagerFactory getInstance(String algorithm, Provider provider)
// 返回充当密钥管理器工厂的KeyManagerFactory对象
public final static KeyManagerFactory getInstance(String algorithm, String provider)
```

当我们不知道该选用何种算法实例化对象时，可使用默认算法，方法如下：

```
// 获取默认的KeyManagerFactory算法名称
public final static String getDefaultAlgorithm()
```

得到实例化对象后，可通过如下方法完成初始化：

```
// 使用密钥内容源初始化此KeyManagerFactory对象
public final void init(KeyStore ks, char[] password)
// 使用特定于提供者的密钥内容源初始化此KeyManagerFactory对象
public final void init(ManagerFactoryParameters spec)
```

我们可以通过如下方法获得密钥管理器数组：

```
// 为每类密钥内容返回一个密钥管理器
public final KeyManager[] getKeyManagers()
```

KeyManagerFactory类是引擎类，同样提供了如下方法：

```
// 返回此KeyManagerFactory对象的提供者
public final Provider getProvider()
// 返回此KeyManagerFactory对象的算法名称
public final String getAlgorithm()
```

2. 实现示例

我们可通过代码清单3-26构建密钥管理工厂。

代码清单3-26　构建密钥管理工厂

```
// 实例化KeyManagerFactory对象
KeyManagerFactory keyManagerFactory = KeyManagerFactory.getInstance("SunX509");
// 加载密钥库文件
FileInputStream is = new FileInputStream("D:\\x.keystore");
// 实例化KeyStore对象
KeyStore ks = KeyStore.getInstance("JKS");
// 加载密钥库
ks.load(is, "password".toCharArray());
// 关闭流
is.close();
// 初始化KeyManagerFactory对象
keyManagerFactory.init(keyStore, "password".toCharArray());
```

获得密钥管理工厂后有什么用呢？我们将在后面介绍。

华章图书

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

3.6.2 TrustManagerFactory类

TrustManagerFactory类是用于管理信任材料的管理器工厂。

```
/* 此类充当基于信任材料源的信任管理器的工厂。每个信任管理器管理特定类型的由安全套接字使用的信任材料。信任材料是基于KeyStore和/或提供者特定的源*/
public class TrustManagerFactory
extends Object
```

1. 方法详述

TrustManagerFactory类的操作与KeyManagerFactory类相似，同样需要getInstance()工厂方法获得实例化对象。

我们可以通过指定算法名称的方式获得实例化对象，方法如下：

```
// 返回充当信任管理器工厂的TrustManagerFactory对象
public final static TrustManagerFactory getInstance(String algorithm)
```

或者，指定算法名称的同时指定该算法的提供者，方法如下：

```
// 返回充当信任管理器工厂的TrustManagerFactory对象
public final static TrustManagerFactory getInstance(String algorithm, Provider provider)
// 返回充当信任管理器工厂的 TrustManagerFactory 对象
public final static TrustManagerFactory getInstance(String algorithm, String provider)
```

当我们不知道该选用何种算法实例化对象时，可使用默认算法，方法如下：

```
// 获取默认的TrustManagerFactory算法名称
public final static String getDefaultAlgorithm()
```

获得实例化对象后，需要通过以下方法完成初始化，这一点和KeyManagerFactory类如出一辙：

```
// 用证书授权源和相关的信任材料初始化此工厂
public final void init(KeyStore ks)
// 使用特定于提供者的信任材料源初始化此工厂
public final void init(ManagerFactoryParameters spec)
```

我们可以通过如下方法获得信任管理器数组：

```
// 为每种信任材料返回一个信任管理器
public final TrustManager[] getTrustManagers()
```

TrustManagerFactory类是引擎类，同样提供了方法如下所示：

```
// 返回此TrustManagerFactory对象的算法名称
public final String getAlgorithm()
// 返回此TrustManagerFactory对象的提供者
public final Provider getProvider()
```

2. 实现示例

我们可通过代码清单3-27构建信任管理工厂。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

代码清单3-27 构建信任管理工厂

```
// 实例化TrustManagerFactory对象
TrustManagerFactory trustManagerFactory = TrustManagerFactory.getInstance("SunX509");
// 加载密钥库文件
FileInputStream is = new FileInputStream("D:\\x.keystore");
// 实例化KeyStore对象
KeyStore ks = KeyStore.getInstance("JKS");
// 加载密钥库
ks.load(is, "password".toCharArray());
// 关闭流
is.close();
// 初始化TrustManagerFactory对象
trustManagerFactory.init(trustkeyStore);
```

我们将在后续章节中对密钥管理工厂和信任材料管理工厂的用途进行介绍。

除了通过KeyManagerFactory和TrustManagerFactory两个工厂类来设定密钥库和信任库外，还可以通过System.setProperty(String key, Object value)进行密钥库、信任库文件路径及密码的设定，具体操作如下：

```
// 密钥库
System.setProperty("javax.net.ssl.keyStore", "D:\\server.keystore");
System.setProperty("javax.net.ssl.keyStorePassword", "123456");

// 信任库
System.setProperty("javax.net.ssl.trustStore", "D:\\server.keystore");
System.setProperty("javax.net.ssl.trustStorePassword", "123456");
```

上述操作，分别等同于代码清单3-26和代码清单3-27的实现，需要在系统初始化时调用。

注意 如果系统中有多个密钥库、信任库的配置，建议使用KeyManagerFactory和TrustManagerFactory两个工厂类进行对应设定，以避免配置被覆盖。

3.6.3 SSLContext类

SSLContext类用于表示安全套接字上下文，它同样是一个引擎类。

```
/* 此类的实例表示安全套接字协议的实现，它充当用于安全套接字工厂或 SSLEngine 的工厂。用可选的一组密钥和信任管理器及安全随机字节源初始化此类*/
public class SSLContext
extends Object
```

1. 方法详述

SSLContext类需要通过getInstance()工厂方法获得实例化对象。一种最为常用的方法如下，该方法只需指定协议名称：

```
// 返回实现指定安全套接字协议的SSLContext对象
public static SSLContext getInstance(String protocol)
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

另一种方法如下，该方法需要在指定协议名称的同时指定该协议的提供者：

```
// 返回实现指定安全套接字协议的SSLContext对象
public static SSLContext getInstance(String protocol, Provider provider)
// 返回实现指定安全套接字协议的SSLContext对象
// 注意，这里使用了协议提供者的名称，而非提供者实例对象
public static SSLContext getInstance(String protocol, String provider)
```

获得实例化对象后，需要通过以下方法完成初始化：

```
// 初始化此上下文
publi c final void init(KeyManager[] km, TrustManager[] tm, SecureRandom random)
```

在完成初始化操作后，我们就可以获得该上下文中的属性，在本书中最为常用的是以下几种方法：

```
// 返回此上下文的ServerSocketFactory对象
public final SSLServerSocketFactory getServerSocketFactory()
// 返回此上下文的SocketFactory对象
public final SSLSocketFactory getSocketFactory()
```

有关SSLSessionContext类相关内容，读者可以参考相关Java API文档，以下方法提供了获得服务器端/客户端SSLSessionContext对象实现：

```
/* 返回服务器会话上下文，它表示可提供服务器端SSL套接字握手阶段使用的SSL会话集*/
public final SSLSessionContext getServerSessionContext()
/* 返回客户端会话上下文，它表示可提供客户端SSL套接字握手阶段使用的SSL会话集*/
public final SSLSessionContext getClientSessionContext()
```

有关SSLEngine类的相关内容，读者可以参考相关的Java API文档，以下方法提供了创建SSLEngine对象实现：

```
// 使用此上下文创建新的SSLEngine
public final SSLEngine createSSLEngine()
// 使用此上下文创建新的SSLEngine，并绑定主机和端口
public final SSLEngine createSSLEngine(String peerHost, int peerPort)
```

以下方法用于设置/获得默认SSL上下文：

```
// 设置默认的SSL上下文
public synchronized static void setDefault(SSLContext context)
// 返回默认的SSL上下文
public synchronized static SSLContext getDefault()
```

以下方法用于设置/获得默认SSL参数：

```
// 返回表示此SSL上下文默认设置的SSLParameters的副本
public final SSLParameters getDefaultSSLParameters()
// 返回表示此SSL上下文受支持设置的SSLParameters的副本
public final SSLParameters getSupportedSSLParameters()
```

SSLContext类作为引擎类，提供了以下方法：

```
// 返回此SSLContext对象的协议名称
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
public final String getProtocol()
// 返回此SSLContext对象的提供者
public final Provider getProvider()
```

2. 实现示例

我们通过代码清单3-28来展示如何使用KeyMangagerFactory、TrustmanagerFactory、SSLContext和SSLSocektFactory类。

代码清单3-28 构建SSLSocketFactory

```
/**
* 获得KeyStore
*
* @param keyStorePath
* @param password
* @return
* @throws Exception
*/
private static KeyStore getKeyStore(String keyStorePath, String password) throws Exception {
    // 获得密钥库文件输入流
    FileInputStream is = new FileInputStream(keyStorePath);
    // 实例化密钥库
    KeyStore ks = KeyStore.getInstance("JKS");
    // 加载密钥库
    ks.load(is, password.toCharArray());
    // 关闭流
    is.close();
    return ks;
}
/**
* 获得SSLSocektFactory
* @param password
*            密码
* @param keyStorePath
*            密钥库路径
* @param trustKeyStorePath
*            信任库路径
* @return
* @throws Exception
*/
private static SSLSocketFactory getSSLSocketFactory(String password, String
keyStorePath, String trustKeyStorePath) throws Exception {
    // 初始化密钥库
    KeyManagerFactory keyManagerFactory = KeyManagerFactory.getInstance
    ("SunX509");
    KeyStore keyStore = getKeyStore(keyStorePath, password);
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
        keyManagerFactory.init(keyStore, password.toCharArray());
        // 初始化信任库
        TrustManagerFactory trustManagerFactory = TrustManagerFactory.getInstance
        ("SunX509");
        KeyStore trustkeyStore = getKeyStore(trustKeyStorePath, password);
        trustManagerFactory.init(trustkeyStore);
        // 初始化SSL上下文
        SSLContext ctx = SSLContext.getInstance("SSL");
        ctx.init(keyManagerFactory.getKeyManagers(),trustManagerFactory.
        getTrustManagers(), null);
        SSLSocketFactory sf = ctx.getSocketFactory();
        return sf;
    }
```

至此，我们就完成了构建HTTPS协议的准备工作。接下来我们将构建一个基于HTTPS协议的网络连接。

3.6.4 HttpsURLConnection类

HttpsURLConnection类继承于HttpURLConnection类。从字面上看，两个类仅差一个字母。但在含义上，HttpsURLConnection类比HttpURLConnection类更具安全性。

```
/* HttpsURLConnection扩展了HttpURLConnection，支持各种特定于https功能*/
public abstract class HttpsURLConnection
extends HttpURLConnection
```

1. 方法详述

HttpsURLConnection类的方法有很多，但对于本书将要阐述的内容来讲，主要用到了如下几种方法。

我们可以通过如下方法设置（默认）SSLSocketFactory对象：

```
/* 设置当此实例为安全https URL连接创建套接字时使用的SSLSocketFactory*/
public void setSSLSocketFactory(SSLSocketFactory sf)
// 设置此类的新实例所继承的默认SSLSocketFactory
public static void setDefaultSSLSocketFactory(SSLSocketFactory sf)
```

相应地，我们可以通过如下方法获得（默认）SSLSocketFactory对象：

```
// 获取为安全https URL连接创建套接字时使用的SSL套接字工厂
public SSLSocketFactory getSSLSocketFactory()
// 获取此类的新实例所继承的默认静态SSLSocketFactory
public static SSLSocketFactory getDefaultSSLSocketFactory()
```

我们可以通过如下方法获得握手期间相关的证书链：

```
// 返回握手期间发送给服务器的证书
public abstract Certificate[] getLocalCertificates()
// 返回服务器的证书链，它是作为定义会话的一部分而建立的
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
public abstract Certificate[] getServerCertificates()
```

关于Principal类，请读者参考相应的Java API文档，以下是HttpsURLConnection类提供的相应方法：

```
// 返回握手期间发送到服务器的主体
public Principal getLocalPrincipal()
// 返回服务器的主体，它是作为定义会话的一部分而建立的
public Principal getPeerPrincipal()
```

以下方法用于获得密码套件：

```
// 返回在此连接上使用的密码套件
public abstract String getCipherSuite()
```

关于HostnameVerifier类，请读者参考相应的Java API文档，以下方法用于获得（默认）HostnameVerifier对象：

```
// 获取此类的新实例所继承的默认HostnameVerifier
public static HostnameVerifier getDefaultHostnameVerifier()
// 获取此实例适当的HostnameVerifier
public HostnameVerifier getHostnameVerifier()
```

以下方法用于获得（默认）HostnameVerifier对象：

```
// 设置此类的新实例所继承的默认HostnameVerifier
public static void setDefaultHostnameVerifier(HostnameVerifier v)
// 设置此实例的HostnameVerifier
public void setHostnameVerifier(HostnameVerifier v)
```

2. 实现示例

我们接3.6.3节内容，获得SSLSocketFactory对象后，可完成HttpsURLConnection对象的设置，如代码清单3-29所示。

代码清单3-29　构建HTTPS

```
// 构建URL对象
URL url = new URL( "https://www.sun.com/" );
// 获得HttpsURLConnection实例化对象
HttpsURLConnection conn = (HttpsURLConnection) url.openConnection();
// 打开输入模式
conn.setDoInput(true);
// 打开输出模式
conn.setDoOutput(true);
// 在这里调用前面介绍的getSSLSocketFactory()方法
// 设置SSLSocketFactory
conn.setSSLSocketFactory(getSSLSocketFactory(password,keyStorePath,trustKeyStorePath));
// 获得输入流
InputStream is = conn.getInputStream();
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
// 若正常打开HTTPS，将获得一个有效值（即contentLength的值不为-1）
int length = conn.getContentLength();
// ……
// 关闭流
is.close();
```

至此，我们就可以通过带有证书的HTTPS连接进行消息传递了。

3.6.5 SSLSession接口

SSLSession接口用于保持SSL协议网络交互会话状态。在SSL的会话中，可以获得加密套件（CipherSuite）、数字证书等。

```
//SSLSession用来描述两个实体之间的会话关系
public abstract interface SSLSession
```

SSLSession接口的方法很多，但对于本书将要阐述的内容来讲，我们主要用到了其中的几个方法。

加密套件中明确给出了加密参数，具体包括：协议、密钥交换算法、加密算法、工作模式和消息摘要算法。如TLS_RSA_WITH_AES_256_CBC_SHA就是一个完成加密套件信息，它表示：使用TLS协议，密钥交换算法为RSA，对称加密算法为AES（密钥长度256位），使用CBC工作模式，并使用SHA消息摘要算法。

我们可以通过以下方法获得上述信息：

```
//获得加密套件
public String getCipherSuite()
```

在SSLSession中，我们可以获得当前会话正在使用的数字证书。

```
//获得数字证书
public Certificate[] getPeerCertificates()
```

或者，我们可以直接获得基于X、509协议的数字证书链

```
//获得X、509协议X、509协议链
public X509Certificate[]getPeerCertificateChain()
```

我们可以作为客户端Socket通过上述方法获得服务器端的数字证书，并保存在本地，而后可以建立基于HTTPS的网络通信。

3.6.6 SSLSocketFactory类

我们在前面多次提到SSLSocketFactory类，通过它可创建SSLSocket，并可获得相应的加密套件。

```
// SSLSocketFactory类用于创建SSLSocket
public abstract class SSLSocketFactory
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
extends SocketFactory
```

SSLSocketFactory类比较简单，通过如下方法可完成自身初始化：

```
// 获得默认的SocketFactory实例
public static?SocketFactorygetDefault()?
```

此后，就可以通过父类方法构建Sokcet实例，如代码清单3-30所示。

代码清单3-30　获取SSLSocket实例

```
// 获得SSLSocketFactory实例
SSLSocketFactory factory = (SSLSocketFactory) SSLSocketFactory.getDefault();

// 构建SSLSocket实例
SSLSocket socket = (SSLSocket) factory.createSocket(hostname, port);
```

此外，可通过以下方法可以获得加密套件：

```
// 获得默认加密套件
public abstract String[] getDefaultCipherSuites()

// 获得当前SSL链接可支持的加密套件
public abstract String[] getSupportedCipherSuites()
```

3.6.7 SSLSocket类

SSLSocket类是基于SSL协议的Socket。用于设置加密套件、处理握手结束事件，并管理SSLSession。

```
// SSLSocket是Socket的子类，用于实现SSL协议
public abstract class SSLSocket
extends Socket
```

1. 方法详述

获得SSLSocket实例后，为了与服务器端SSLSocket确定加密套件，常用到如下方法：

```
// 获得可支持的加密套件
public abstract String[]getSupportedCipherSuites()

// 为当前SSL连接设置可用的加密套件
public abstract void setEnabledCipherSuites(String[] suites)

// 获得当前SSL连接可用的加密套件
public abstract String[] getEnabledCipherSuites()
```

目前，Java 环境中支持的协议有SSLv2Hello、SSLv3、TLSv1、TLSv1.1和TLSv1.2等。通常默认协议是SSLv3和TLSv1。后续章节中我们会提到安全协议，其实TLS协议就是SSL协议的第3个版本。这两个名称实际上是等价的。通过如下方法可以进行协议变更适配：

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
// 设置当前SSL连接可用的协议，如SSL、TLS等
public abstract void setEnabledProtocols(String[] protocols)

// 获得当前SSL连接可用的协议
public abstract String[] getEnabledProtocols()

// 获得可支持的协议
public abstract String[] getSupportedProtocols()
```

完成了加密套件和协议配置后，就可开始握手协议、建立加密套接字进行加密通信了。具体方法如下：

```
// 在当前连接建立SSL握手
public abstract void startHandshake()
```

相应地结束SSL连接，直接调用其父类的close()方法即可。

我们前面小节中提到过SSLSession接口，这个接口可以通过SSLSocket获得，方法如下：

```
// 获得当前连接的SSL会话实例
public abstract SSLSession getSession()
```

2. 实现示例

有时候，我们需要与远程服务建立基于SSLSocket的连接。远程服务仅通过SSLSocket传递数字证书。这时候，就不能通过HttpsURLConnection类来获得数字证书了。我们可以通过SSLSocket实例获得SSLSession实例，最终获得数字证书，如代码清单3-31所示。

代码清单3-31 获取数字证书

```
/**
 * 获得数字证书
 *
 * @param hostname 远程服务IP/域名
 * @param port 端口号
 * @return Certificate[] 证书链
 * @throws Exception
 */
public Certificate[] getCertificates(String hostname, int port)throws Exception {
    // 获得SSLSocketFactory实例
    SSLSocketFactory factory = (SSLSocketFactory) SSLSocketFactory.getDefault();

    // 构建SSLSocket实例
    SSLSocket socket = (SSLSocket) factory.createSocket(hostname, port);

    // SSL握手
    socket.startHandshake();

    // 获得SSLSession实例
    SSLSession session = socket.getSession();
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
        // 关闭Socket
        socket.close();

        // 获得数字证书
        return session.getPeerCertificates();
    }
```

如有必要，我们可以通过如下代码，输出当前网络下的debug日志：

```
System.setProperty("javax.net.debug", "all");
```

我们将在控制台中获得当前网络连接操作过程中使用到的数字证书信息和经过加密后的网络传输数据等。

3.6.8 SSLServerSocketFactory类

SSLServerSocketFactory类与SSLSocketFactory类相关操作几乎一致，只是所构建的Socket是SSLServerSocket类。

```
// SSLSocketFactory类用于创建SSLServerSocket
public abstract class SSLServerSocketFactory
extends ServerSocketFactory
```

参考前面的SSLSocketFactory类实现，SSLServerSocketFactory类通过如下方法完成自身初始化：

```
// 获得默认的ServerSocketFactory实例
public static ServerSocketFactory getDefault()
```

此后，可以通过父类方法构建Sokcet实例，如代码清单3-32所示。

代码清单3-32 获取SSLServerSocket实例

```
// 获得SSLSocketFactory实例
SSLSocketFactory factory = (SSLSocketFactory) SSLSocketFactory.getDefault();

// 构建SSLServerSocket实例
SSLServerSocket serverSocket = (SSLServerSocket) factory.createSocket(hostname, port);
```

与SSLSocketFactory类一样，SSLServerSocketFactory类提供相应的加密套餐操作方法：

```
// 获得默认加密套件
public abstract String[]getDefaultCipherSuites()

// 获得当前SSL链接可支持的加密套件
public abstract String[]getSupportedCipherSuites()
```

3.6.9 SSLServerSocket类

SSLServerSocket类是专用于服务器端的SSLSocket，是ServerSocket的子类。

```
// SSLServerSocket是ServerSocket的子类，用于实现服务器端SSL协议
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
public abstract class SSLServerSocket
extends ServerSocket
```

1. 方法详述

参考前面的SSLSocket类方法详述，SSLServerSocket类同样支持如下方法：

```
// 获得可支持的加密套件
public abstract String[]getSupportedCipherSuites()

// 为当前SSL连接设置可用的加密套件
public abstract void setEnabledCipherSuites(String[] suites)

// 获得当前SSL连接可用的加密套件
public abstract String[]getEnabledCipherSuites()

// 设置当前SSL连接可用的协议，如SSL、TLS等
public abstract void setEnabledProtocols(String[] protocols)

// 获得当前SSL连接可用的协议
public abstract String[]getEnabledProtocols()

// 获得可支持的协议
public abstract String[]getSupportedProtocols()
```

2. 实现示例

如果对网络套接字很了解，获得SSLSocket实例后，就可以像一般Socket一样进行相关操作了，如代码清单3-33所示。

代码清单3-33 构建SSLServerSocket实例

```
// 获得SSLServerSocketFactory实例
SSLServerSocketFactory socketFactory = (SSLServerSocketFactory) SSLServerSocket
Factory.getDefault();

// 构建SSLServerSocket实例
SSLServerSocket serverSocket = (SSLServerSocket)
socketFactory.createServerSocket(port);

// 服务器端套接字进入阻塞状态，等待来自客户端的连接请求
SSLSocket socket = (SSLSocket) serverSocket.accept();

// 通过Socket获得相应的流对象进行操作
// ……

// 关闭Socket
socket.close();
```

经过上述对SSLSession、SSLSocketFactory、SSLServerSocketFactory、SSLSocket和

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

SSLServerSocket了解，已经可以构建一套基于SSLSocket的加密网络套接字通信实现。

当然，如果直接进行上述事项可能会遇到以下异常：

```
javax.net.ssl.SSLHandshakeException: no cipher suites in common

javax.net.ssl.SSLHandshakeException: Received fatal alert: handshake_failure
```

这些异常说明在握手阶段，双方没有能够协商出加密方法等信息。

因为在默认情况下，JVM没有与SSL相关的配置，需要我们手工配置。这一过程可以参考本章3.6.3节，进行KeyManagerFactory和TrustManagerFactory的配置获得SSLSocketFactory，如代码清单3-34所示。

代码清单3-34　初始化SSLSocketFactory和SSLServerSocketFactory

```
// 初始化
KeyStore ks = KeyStore.getInstance("JKS");

ks.load(new FileInputStream(keyStoreFile), password);

// 初始化密钥库
KeyManagerFactory kmf = KeyManagerFactory.getInstance("SunX509");
kmf.init(ks, password);

// 初始化信任库
TrustManagerFactory tmf = TrustManagerFactory.getInstance("SunX509");
tmf.init(ks, password);

// 初始化SSLContext
SSLContext sslContext = SSLContext.getInstance("SSL");
sslContext.init(kmf.getKeyManagers(),tmf.getTrustManagers(), null);

// 上述代码请参考本节对应小节内容

// 获得SSLSocketFactory实例
SSLSocketFactory sslSocketFactory = sslContext.getSocketFactory();

// 获得SSLServerSocketFactory实例
SSLServerSocketFactory sslServerSocketFactory = sslContext.getServerSocketFactory();
```

上述代码中用到的“keyStoreFile”既是密钥库文件，也是信任库文件。

当然，我们也可以通过System.setProperty(String key, Object value)进行密钥库、信任库文件路径及密码的设定，如代码清单3-35所示。

代码清单3-35　设定密钥库和信任库

```
// 密钥库
System.setProperty("javax.net.ssl.keyStore", "D:\\keystore.jks");
System.setProperty("javax.net.ssl.keyStorePassword", "123456");
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
// 信任库
System.setProperty("javax.net.ssl.trustStore", "D:\\truststore.jks");
System.setProperty("javax.net.ssl.trustStorePassword", "123456");
```

或者在命令行执行java命令，如代码清单3-36所示。

代码清单3-36 命令行配置密钥库和信任库

```
java -cp
    -Djavax.net.ssl.keyStore=<Your KeyStroe File>
    -Djavax.net.ssl.keyStorePassword=<Your KeyStroe Password>
    -Djavax.net.ssl.trustStore=<Your TrustStroe File>
    -Djavax.net.ssl.trustStorePassword=<Your TrustStroe Password>
<Your Class File>
```

3.7 小结

通过对本章的学习，我们了解了有关Java安全领域的内容，这部分内容共分为三个部分：JCE（Java Cryptography Extension，Java加密扩展包）、JSSE（Java Secure Sockets Extension，Java安全套接字扩展包）、JAAS（Java Authentication and Authentication Service，Java鉴别与安全服务）。JCE和JSSE是本书中主要阐述的重点内容。

在安全提供者体系结构中，我们认识了Provider类和Security类，它们共同构成了安全提供者的概念。在Java 7版本中，共配置了10种安全提供者。它们均是Provider类的子类。Security类则用于管理这些提供者。

受军事出口的限制，密码算法的实现强度有所限制。如DES算法受出口限制，其密钥长度为56位，而实际上出于安全考虑则要求密钥长度为128位。这种安全强度已不能满足当前应用环境的需要。

在java.security包和javax.crypto包中，我们详述了如何构建对称密钥和非对称密钥，以及如何使用加密组件实现加密算法。

如Key接口，作为顶层密钥接口定义了密钥的基本操作。SecretKey、PublicKey和PrivateKey接口均是Key接口的子接口，分别定义了用于对称加密算法的秘密密钥和用于非对称加密算法的公钥/私钥。

我们可以通过相应的生成器和工厂类完成密钥的生成，如KeyGenerator和SecretKeyFactory用于秘密密钥的生成；KeyPairGenertor和KeyFactory用于公钥/私钥的生成。

通过Cipher及其扩展类CipherInputStream和CipherOutputStream，可以完成加密与解密算法的实现。通过Sinature类则可以完成签名与验证操作。

除此之外，我们还了解了消息摘要算法的两种实现方式：Mac类，用于实现安全消息摘要算法；MessageDigest类及其扩展类DigestInputStream和DigestOutputStream，用于实现一般消息摘要算法。

对于密钥管理，我们提到了KeyStore类。它将在密钥管理和证书管理的操作中，多次使

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

用到。

在java.security.spec包和javax.crypto.spec包中主要详述如何通过密钥规范还原密钥对象。我们将会多次使用到这两个包中的实现。密钥通常以二进制的方式存储在文件中，通过这两个包中的实现我们将可以将二进制数据还原为秘密密钥、公钥和私钥对象。

在java.security.cert包中，我们主要详述了如何通过CertificateFactory类构建证书（Certificate）对象和证书撤销列表（CRL）对象。在获得证书对象后，我们可以获得相应的算法、公钥等信息。这将有助于我们完成相应的加密解密操作以及数字签名与验证操作。

在了解了上述与加密与解密算法相关的实现，以及与数字证书操作有关的实现内容后，我们来关注javax.net.ssl包中的内容。在这一节中，我们最终达到了通过证书构建基于HTTPS协议的加密网络通信以及基于安全网络套接字的加密通信网络。

Java是一门不断发展的语言，所涉及的领域越来越多。网络应用、电子商务等领域都促进着Java加密与解密技术的发展，我们将在今后的开发过程中更加频繁地使用到Java所提供的安全实现。