视频位置：P:\00\_传智播客\黑马程序员训练营\_公开课视频教程\中关村黑马程序员训练营\_公开课视频教程\_Java安全

Java安全，100个人中只有一个人会这门技术。学这门技术的目的有两个：

1. 让自己不会随随便便就被别人替换的程序员，一但在公司里立了足，老板就会发现我们掌握的技术别人没法替代，只有我会，别人都不会。别人可以随便换，但是你不会被换
2. 对找工作有帮助，在找工作时，虽然有的公司不使用这门技术，但是在简历上写上：“熟练掌握Java安全”，面试官看到这个就会对你另眼相看

Java安全学起来不难，说出去还很有档次。

# 对称加密与非对称加密

对称加密：加密与解密用的是同一个密码，加密速度快

非对称加密：加密解密用的密码不相同，可以使用公钥加密也可以使用私钥加密，解密就需要相对的另一个密钥。非对称加密比对称加密算法上要更复杂，所以加密速度慢

非对称加密应用场景：多个人可以加密，但是只有一个人可以解密。比如我们访问工商银行，数据传给工商银肯定需要加密的，假设工商银行和用户A、用户B分别约定了两个密码，用户A用密码一加密后把数据传工商银行，用户B用密码二加密后把数据传给工商银行，银行使用相应的密码进行解密，如果此时来了用户C，银行又得跟用户C商量好一个密码，如果有一万个用户呢？这时，使用非对称加密就能解决问题：工商银行把公钥放在网站上，所有人都使用这个公钥进行加密，工商银行使用私钥进行解密。 解密的密码不能对外公开，叫私钥，而外面用来加密的可以公开的叫公钥

所有加密里的密码统称为密钥，对称加密的密码称为密钥（SecretKey），非对称加密的有公钥（PublicKey）和私钥（PrivateKey）

Key：SecretKey，PublicKey，PrivateKey

网络传输走的是电信号，通过高电压低电压组合可以表示各种0101的数据，所以只有拨开网线的皮，搭上另外的线，电信号也是会传过去的，黑客就可以这样盗取别人的信息。只要大家在一个网络里面，数据传输就都会经过连在一起的线，比如在一个局域网里所有的电脑都是连着网线的，那发送的数据就会经过每一台电脑。A用户给B用户发送QQ信息，好像只有B用户收到了，其实是同一个局域网中的所有电脑都收到了，只是网卡收到后发现不是自己的数据就没有拿进来处理而已，我们可以写程序告诉网卡，不管是发给谁的数据都拿进来处理，这样只要我们会分析会解密，就能查看发给别人的信息。

非对称加密另一个例子：武侠小说里的毒药和解药，毒药可以随便卖给别人，但是解压只有自己才有。

面试技巧，写代码时import包名写上面试的公司域名

## Serializable知识点

面试经常问Serializable

在工作中，经常需要把一个对象保存起来持久化到硬盘中（或者sdcard），保存在内存中不叫持久化。持久化之后方便以后 再次使用，如果是我们自己做，我们可以把对象的各个属性持久化，需要恢复时再读取这些属性，创建一个Java对象并设置这些属性。也可以通过Gson把一个对象转成json再保存，以后读取json再转换回Java对象。因为保存对象的需求很常见，所以系统提供了Serialiable来帮助我们轻松的保存对象，至于以什么样的格式来保存我们不需要关心，你想使用ObjectOutputStream保存对象，只要确保你的对象实现了Serializable接口即可。ObjectOputStream是一个包装流，它只负责把对象转换为byte[]，把byte写到哪里由它包装的那个输出流决定。

一个对象可以通过流写到本地，也可以写到网络传给别人，这时别人在恢复时，也需要有对应的Java类，如果两个类结构不一致的话，将导致恢复失败，所以如何确定两个类是一至的呢？通过类中的：

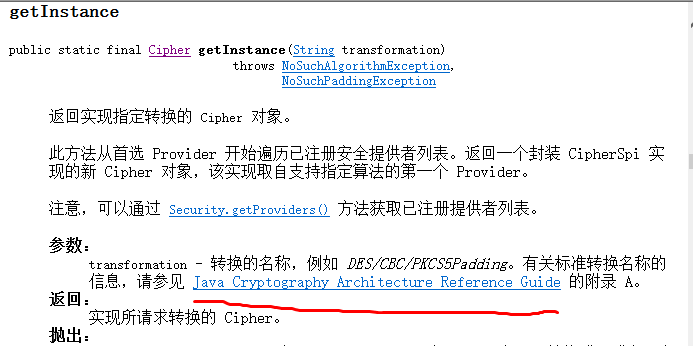
**public static final long *serialVersionUID*** = 42L;// 这个属性的public可以改成任意修饰符，只要是静态的即可。

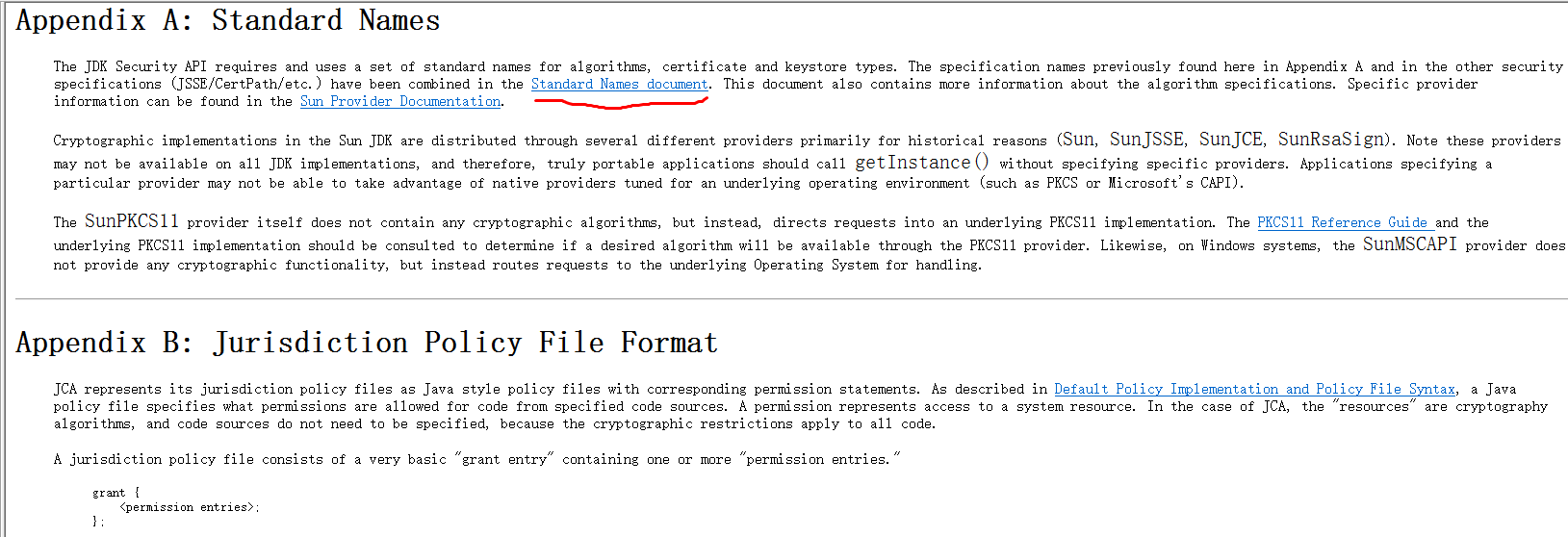
只要你写出的对象和别人读时使用的对象的serialVersionUID是同一个，就可以实现恢复，当然你要是硬把其实一 个对象的属性名改了，肯定也是会出问题的。如果我们没有定义这个属性，则系统会根据对象的属性自动生成一个。为了确保稳定，我们还是自己定义一个比较好，当我们修改属性名或者增加属性时，最好把***serialVersionUID***这个的值也增加一下，代表这个类有变动，则别人在恢复对象时发现恢复失败可以比较这个***serialVersionUID***是否一致，如果不一致，说明类结构已经被修改过了，需要使用相同结构的类。

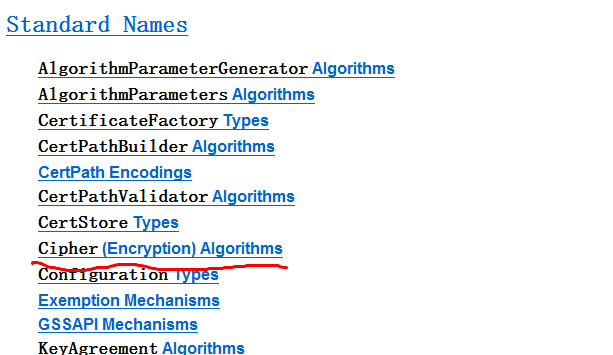
## 对称加密示例

对称加密也叫密钥加密，英文中对应的密钥类是：SecretKey

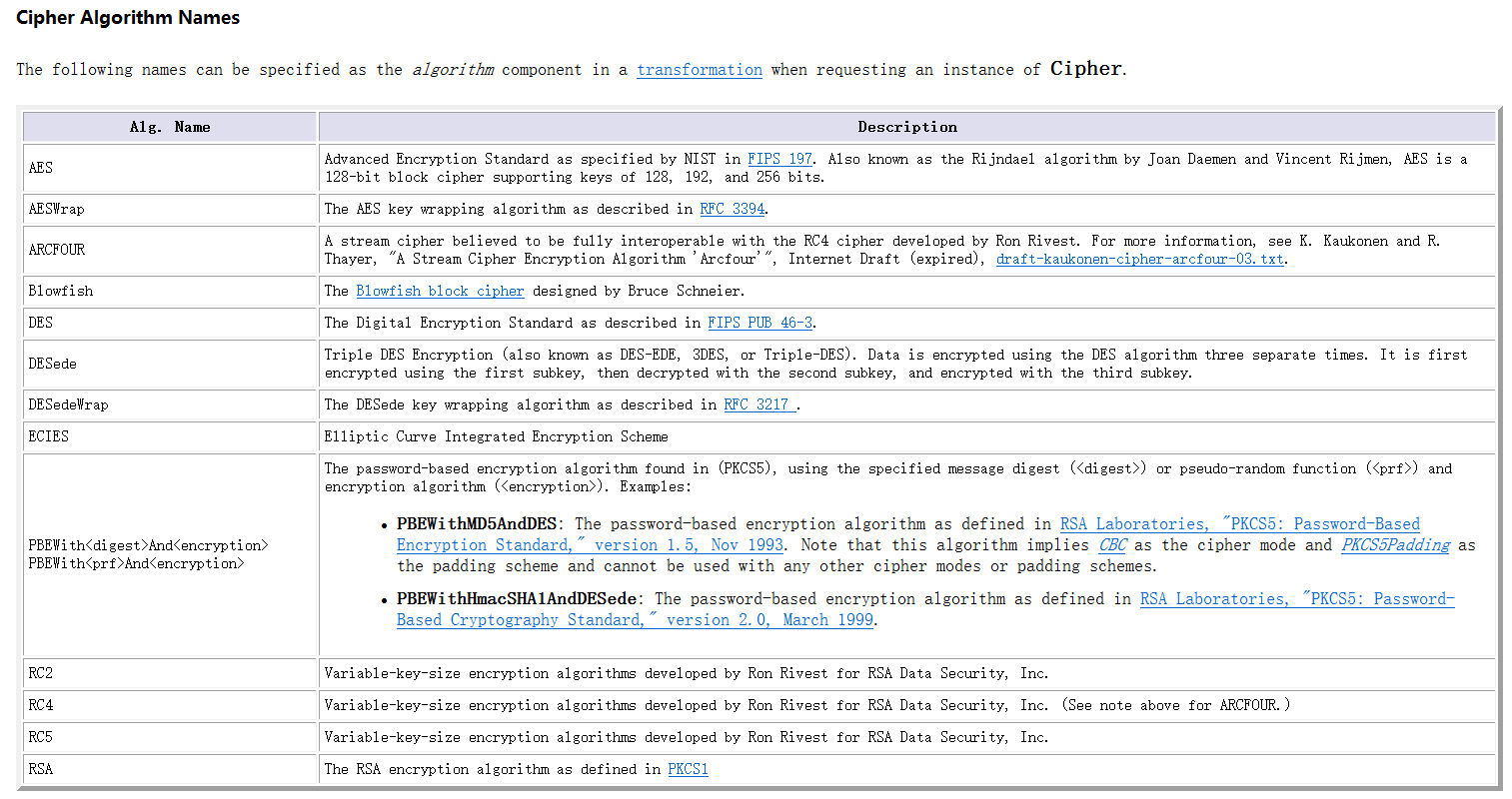
加密的单词叫Cipher，所以有对应的这样一个类用于加密，查看JDK文档可知它的创建方式为：Cipher c = Cipher.getInstance("*DES/CBC/PKCS5Padding*"); 这里并不是说有3个算法，而是算法是DES，而这个算法下面又有很多的细节，如CBC为循环检验，PKCS5Padding为填充方式，这里我们可以只填入“DES”：Cipher.getInstance(“DES”)，那具体有哪些算法呢，可查看getInstance方法有介绍，如下：



点开这个连接：  


点击上图中的Standard Names doucument：  


再点击上图中的连接：



如上图，这是Cipher类支持的所有加密方式。 加密方法为：

cipher.doFinal(byte []) 把指定的字节进行加密

还有另一种实现，可能需要加密的数据是一点一点送过来的，则可以使用：

cipher.update(byte[])，能过这个方法不断接收数据，最后接收完时调用cipher.doFinal()完成加密。张老师讲课时说有同学试验了说update之后doFinal并不能完成加密，也就是这些方法并没有真正实现。

加密实现思路：

1. 找到拥有加密功能的类：Cipheer
2. 调用这个类的加密方法：doFinal(byte[])

Cipher cipher = Cipher.*getInstance*(**"DES"**);  
cipher.doFinal(**"你好"**.getBytes(**"UTF-8"**));

运行会报错，说Cipher没有初始化。我们在写代码时，都会先用整体的思维想问题，在写的时候不可能面面具到，所以会有一些小的细节遗漏，这是正常的。所以真实开发不要想着一敲代码就是完善的，肯定是先按主体去敲，主体整出来了再根据报错的细节慢慢修正代码。

Cipher含义为“密码”，它即可用来加密，也可以加密，所以调用doFinal方法的时候，你要告诉它是加密呢，还是解密呢。如下：  
Key key = KeyGenerator.*getInstance*(**"DES"**).generateKey();  
cipher.init(Cipher.***ENCRYPT\_MODE***, key);

解密和加密是一样的，只需要再调用init方法一次，把参数设置为解密即可，如下：  
cipher.init(Cipher.***DECRYPT\_MODE***, key);  
bytes = cipher.doFinal(bytes); // 注：参数中的bytes是加密后的字节码，放进来解密  
System.***out***.println(**new** String(bytes, **"UTF-8"**));

注：AES和DES加/解密一模一样，只需要把使用“DES”的地方换成“AES”

完整示例如下：

**public** **class** Demo {

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

// createKey();

// encript();

*decript*();

}

/\*\* 创建密钥并保存起来，方便下次使用 \*/

**private** **static** **void** createKey() **throws** Exception {

Key key = KeyGenerator.*getInstance*("DES").generateKey();// 创建密钥用于DES算法的密钥

ObjectOutputStream oos = **new** ObjectOutputStream(**new** FileOutputStream("EvenDai.key"));

oos.writeObject(key);

oos.close();

}

/\*\* 读取密钥 \*/

**private** **static** Key readKey() **throws** Exception {

ObjectInputStream ois = **new** ObjectInputStream(**new** FileInputStream("EvenDai.key"));

Key key = (Key) ois.readObject();

ois.close();

**return** key;

}

/\*\* 加密 \*/

**private** **static** **void** encript() **throws** Exception {

Cipher cipher = Cipher.*getInstance*("DES"); // 创建DES算法的加密对象

cipher.init(Cipher.***ENCRYPT\_MODE***, *readKey*()); // 初始化为加密模式

String data = "好好学习，天天向上！"; // 要加密的数据

**byte**[] encriptBytes = cipher.doFinal(data.getBytes()); // 进行加密

System.***out***.println(**new** String(encriptBytes)); // 显示加密后的数据

// 保存加密后的数据

FileOutputStream fos = **new** FileOutputStream("encript.data");

fos.write(encriptBytes);

fos.close();

}

**private** **static** **void** decript() **throws** Exception {

Cipher cipher = Cipher.*getInstance*("DES"); // 创建DES算法的加密对象

cipher.init(Cipher.***DECRYPT\_MODE***, *readKey*()); // 初始化为解密模式

// 读取加密的数据

ByteArrayOutputStream baos = **new** ByteArrayOutputStream();

FileInputStream fis = **new** FileInputStream("encript.data");

**byte**[] buf = **new** **byte**[1024];

**int** len;

**while**((len = fis.read(buf)) != -1) {

baos.write(buf, 0, len);

}

fis.close();

**byte**[] decriptBytes = cipher.doFinal(baos.toByteArray());// 进行解密

System.***out***.println(**new** String(decriptBytes)); // 显示解密后的数据

}

}

读一个文件流，也可以通过流的available()方法，availabe只适用于文件流，表示连续有多少长度的内容可读，等同于File的length()，而对于网络流，只是说连续可读的数量，因为网络数据是不稳定的，网速慢时，发出连结可读的一段数据就少一些，网络的数据是一段一段来的。

上面代码生产的Key，就相当于是银行的U盾 ，即U盘里的证书就是Key。

还有另一种加密的方式是使用密码，如银行卡取款的密码，这时不能使用DES算法了，要用另一种叫“**PBEWithMD5AndDES**”的算法，其中的PBE代表“password-based encryption（基于密码的加密）”，代码和上面基本相同，需要改变的地方只有两处如下：

1、Key的创建：

注：这里的密码要求必须是8位长度，否则会抛异常

**private** **static** Key createKey(String password) **throws** Exception {

KeySpec keySpec = **new** PBEKeySpec(password.toCharArray());

SecretKey key = SecretKeyFactory.*getInstance*("PBEWithMD5AndDES").generateSecret(keySpec);

**return** key; // 这时的key没有必要保存了，我们只要记得自己设置的password字符串密码即可。

}

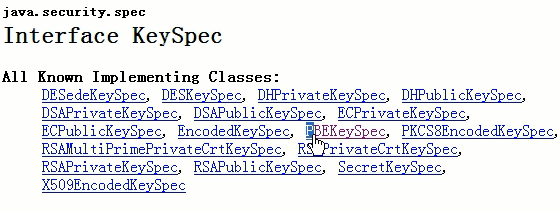
2、Cipher类的创建：

上面的PBEParameterSpec就是传说中的加盐，解密的时候也得加这个参数。这是PBE算法比较特殊的，初始化Cipher时必须要加盐。加盐主要就是让加密更加安全，在网络上拦截了数据，它知道了算法也知道了密码，则就可以解密了，这样不太安全，如果加了盐了，他解密时不知道我们加的盐是什么，所以虽然知道了密码也没法解密。

PBEParameterSpec parameterSpec = **new** PBEParameterSpec(**new** **byte**[]{1,2,3,4,5,6,7,8},1000);

Cipher cipher = Cipher.*getInstance*("PBEWithMD5AndDES"); // 创建DES算法的加密对象

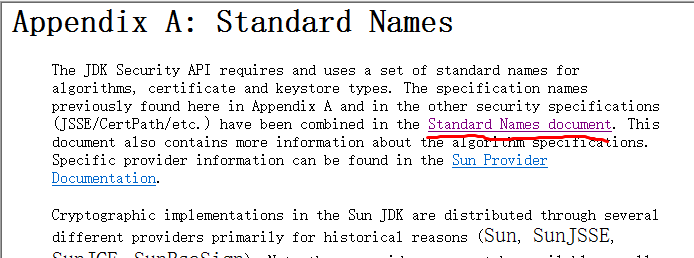
cipher.init(Cipher.***ENCRYPT\_MODE***, *createKey*("12345678"), parameterSpec); // 初始化为加密模式

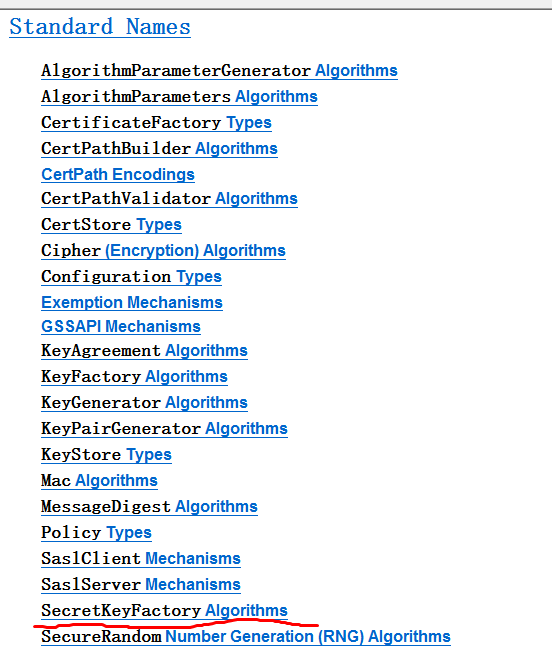
KeySpeck为含义为Key的规范，则表示你这个密钥是怎样的，它有很多子类，如下：  


PBEKeySpec为基于密码的加密。

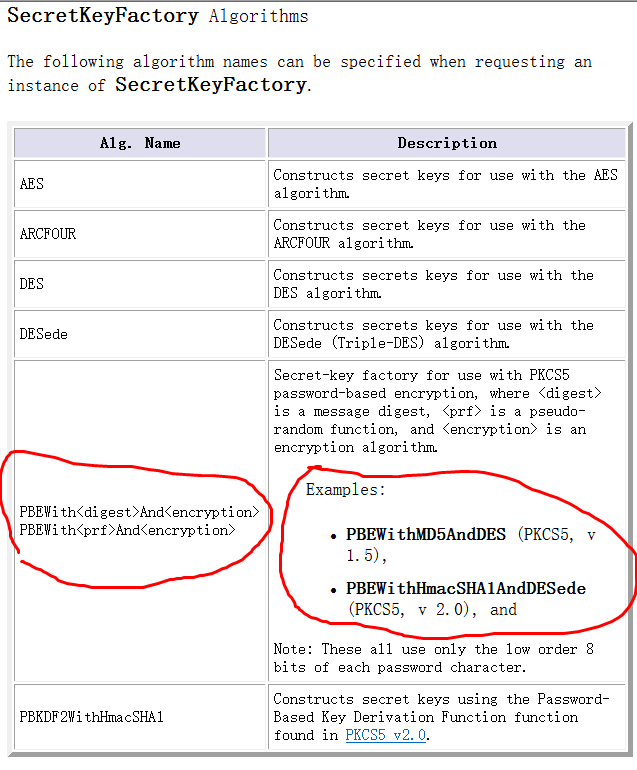
“**PBEWithMD5AndDES**”这个算法名是在哪里找的？查看SecretKeyFactory.*getInstance方法，如下：*





因为算法名用于SecretKeyFactory，所以这里找对对应的地方



## 另一种基于密码的对称加密：

String password = **"12345678"**; *// 明文密码  
  
// 基于明文密码生成Java中的密码对象*DESKeySpec keySpec = **new** DESKeySpec(password.getBytes());  
Key key = SecretKeyFactory.*getInstance*(**"DES"**).generateSecret(keySpec);  
  
*// 加密*String rawData = **"Hello world!"**;  
Cipher cipher = Cipher.*getInstance*(**"DES"**);  
cipher.init(Cipher.***ENCRYPT\_MODE***, key);  
**byte**[] bytes = cipher.doFinal(rawData.getBytes());  
  
*// 解密*cipher.init(Cipher.***DECRYPT\_MODE***, key);  
bytes = cipher.doFinal(bytes);  
System.***out***.println(**new** String(bytes));*// 输出Hello world!*

## 非对称加密

一般都是公钥加密，私钥解密。代码与对称加密的方式基本相同，不同的是算法改成RSA，密钥的生成使用KeyPairGenerator来生成公钥和私钥，加密用公钥，解密就用私钥。

非对称加密一般用在JavaEE，因为一个服务器对应很多的用户去访问，如果想要工资高，那么需要懂后台和客户端，如果只做Android，就相当于只是给医生做工具是一样的，重要的东西都在医生那里，你只是个做道具的，公司的后台网络是如何搭建，这个是最重要的。

**public** **class** Demo {

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

// createKey();

*encript*();

*decript*();

}

/\*\* 创建密钥并保存起来，方便下次使用 \*/

**private** **static** **void** createKey() **throws** Exception {

KeyPairGenerator generator = KeyPairGenerator.*getInstance*("RSA");

KeyPair keyPair = generator.generateKeyPair();

PublicKey publicKey = keyPair.getPublic();

PrivateKey privateKey = keyPair.getPrivate();

*saveKey*(publicKey, "publicKey.key");

*saveKey*(privateKey, "privateKey.key");

}

**private** **static** **void** saveKey(Key key, String keyFileName) **throws** Exception {

ObjectOutputStream oos = **new** ObjectOutputStream(**new** FileOutputStream(keyFileName));

oos.writeObject(key);

oos.close();

}

/\*\* 读取密钥 \*/

**private** **static** Key readKey(String keyFileName) **throws** Exception {

ObjectInputStream ois = **new** ObjectInputStream(**new** FileInputStream(keyFileName));

Key key = (Key) ois.readObject();

ois.close();

**return** key;

}

/\*\* 加密 \*/

**private** **static** **void** encript() **throws** Exception {

Cipher cipher = Cipher.*getInstance*("RSA"); // 创建DES算法的加密对象

cipher.init(Cipher.***ENCRYPT\_MODE***, *readKey*("publicKey.key")); // 初始化为加密模式

String data = "好好学习，天天向上！"; // 要加密的数据

**byte**[] encriptBytes = cipher.doFinal(data.getBytes()); // 进行加密

System.***out***.println(**new** String(encriptBytes)); // 显示加密后的数据

// 保存加密后的数据

FileOutputStream fos = **new** FileOutputStream("encript.data");

fos.write(encriptBytes);

fos.close();

}

**private** **static** **void** decript() **throws** Exception {

Cipher cipher = Cipher.*getInstance*("RSA"); // 创建DES算法的加密对象

cipher.init(Cipher.***DECRYPT\_MODE***, *readKey*("privateKey.key"));// 初始化为加密模式

// 读取加密的数据

ByteArrayOutputStream baos = **new** ByteArrayOutputStream();

FileInputStream fis = **new** FileInputStream("encript.data");

**byte**[] buf = **new** **byte**[1024];

**int** len;

**while**((len = fis.read(buf)) != -1) {

baos.write(buf, 0, len);

}

fis.close();

**byte**[] decriptBytes = cipher.doFinal(baos.toByteArray());// 进行解密

System.***out***.println(**new** String(decriptBytes)); // 显示解密后的数据

}

}

注：cipher.update方法在RSA加密中不起作用，RSA加密只能使用cihper.doFinal(byte[] data)方法

## 简便解密类：CipherInputStream和CipherOutputStream

CipherInputStream和CipherOutputStream是用于在读或写的时候就解密或者解密，就看会传入的chipher对象的初始化模式，如果设置为解密，则CipherInputStream用于把加密的文件读进来，读进来就解密了，而CipherOutputStream用于把加密的数据写到文件，写的时候就解密了，也就是说这两个流这时都是用在解密：具体例子如下：

**private** **static** **void** decript() **throws** Exception {

Cipher cipher = Cipher.*getInstance*("RSA"); // 创建DES算法的加密对象

cipher.init(Cipher.***DECRYPT\_MODE***, *readKey*("privateKey.key"));// 初始化为加密模式

FileInputStream fis = **new** FileInputStream("encript.data");

CipherInputStream cis = **new** CipherInputStream(fis, cipher); // 从这个流中读取的内容就是解密的

// 读取加密的数据

ByteArrayOutputStream baos = **new** ByteArrayOutputStream();

**byte**[] buf = **new** **byte**[1024];

**int** len;

**while**((len = cis.read(buf)) != -1) {

baos.write(buf, 0, len);

}

cis.close();

fis.close();

System.***out***.println(**new** String(baos.toByteArray()));

FileInputStream fis2 = **new** FileInputStream("encript.data");

FileOutputStream fos2 = **new** FileOutputStream("unencript.data"); // 把解密后的数据写到这个文件里

CipherOutputStream cos = **new** CipherOutputStream(fos2, cipher); // 用这个流写出去的内容就是解密的

**while**((len = fis2.read(buf)) != -1) {

cos.write(buf, 0, len);

}

cos.close();

fos2.close();

fis2.close();

}

# 数字摘要：MD5/SHA

* 又称为数据的指纹，用于表示数据的完整性，可以知道数据有没有被修改
* **消息摘要是一种算法：无论原始数据多长，消息摘要的结果都是固定长度的；原始数据任意bit位的变化，都会导致消息摘要的结果有很大的不同，且根据结果推算出原始数据的概率极低。消息摘要可以看作原始数据的指纹，指纹不同则原始数据不同。**

比如MD5，给MD5传入一堆二进制数据，这些数据无论多少（如1K、1M、1G），通过Md5算法对这些二进制进行计算，计算的结果长度是固定不变，都是128位，即16个字节。只要传进来的数据有一点点变化，则算出来的结果就大相径庭。既然结果总是128位，能表示的结果组合是有限的，而数据是无限的，所以不同数据产生的MD5结果还是有可能相同，但是这是很难碰得到相同的情况的，可以略等于不会有相同的，比如人的指纹是不可能有一样的，如果哪一天出现了一个罪犯，他的指纹和你一样，然后你被抓了，警察认为你就是罪犯把你枪毕了也不怨枉，你太幸运，活该！

SHA产生的数字摘要是20个字节，所以相比MD5要更安全。

MD5不是加密，因为通过MD5结果是不能恢复出原始数据的。

我们在注册用户时，不需要把密码发给服务器端，服务器端也不希望保存明文密码，所以我们只需要把md5后的密码发给服务器端保存即可。用户名是直接保存的，密码则是保存数字摘要的。

## MD5产生数字摘要示例：

**public** **class** Demo {

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

**byte**[] bytes = "zxx".getBytes();

String md5 = *md5*(bytes);

System.***out***.println("md5的16进制字符串：" + md5 + ", 长度：" + md5.length()); // 长度为32

}

**private** **static** String md5(**byte**[] bytes) **throws** Exception {

MessageDigest md = MessageDigest.*getInstance*("MD5");

**byte**[] md5Bytes = md.digest(bytes); // 产生的md5数字摘要长度是16个字节

System.***out***.println("md5字节数组长度为：" + md5Bytes.length); // 长度为16

// 1个字节是8个比特位，如果把每个字节用10进制表示，则每个字节范围是-128 ~ 127，假设每个字节都是最大值，则会有16个127，

// 把16个127连接起来显示会很长，所以不使用10进制，使用16进制，4个位正好可以表示0 ~ 15，所以 1个字节会使用两个16进制数来表示，也就是1个字节最大为ff，比127短一位, md5长度是16个字节，每个字节用两个16进制数表示，则转换为16进制显示需要32个16进制数

String hexMd5 = *byteToHex*(md5Bytes);

**return** hexMd5;

}

/\*\* 把字节转换为16进制字符串 \*/

**private** **static** String byteToHex(**byte**[] bytes) {

StringBuilder sb = **new** StringBuilder();

**for** (**byte** b: bytes) {

// 我们希望1个byte用两个16进制数表示，1个byte8位，通过位运算取高4位和低4位分别转换为16进制

// byte hight4Bit = (byte) (b >>> 4); 这样取高4位是不对的，原因如下：

// 位运算会提升为10进制再位移，如果这个byte是负数，则提升为10进制时为32位，除了自己的8位，高位多出来的24位全是1，

// 无符号右移4位，则高4位变成了0，但是高24位中还有20位还是1,而我们希望32个二进位上只保留后8位中的前面4位

// 所以不能用无符号右移操作

**int** hight4Bit = b >> 4 & 0xF; // 把高4位右移，低4位就被移出去了，&F就是只保留最低的4个二进制位，其它全部变成0。因为一个f正好是4个位，且4个位都是1，则高28位全是0，所以能保留下低4位。

**int** low4Bit = b & 0xF; // 注：&0F跟&F没有任何区别，推荐使用&F。因为&0F很容易让人认为是把4位与0相与，把4位与1相与，其实是错的，是低4位与1相与，剩余的位(28位)与0相与

// 现在的hight4Bit和low4Bit都是正数，且值最大是15，也就是16进制的F

// 我们不知道ASCCI码表中字符对应的十进制值是多少，但是我们可以直接使用这些字符进行运算，实际上是用这些字符对应的十进制值进行运算

// 如果4个位的值 >= 10，则我们减去10再加上a，结果就是那4个位的值对应的16进制的字母所对应的数值。字符a的对应的ASCII数值为97，

如4个位的值为10，则10 – 10 + 97 = 97（a）

如4个位的值为11，则11 – 10 + 97 = 98（b）

如4个位的值为12，则12 – 10 + 97 = 98（c）

如4个位的值为13，则13 – 10 + 97 = 98（d）

如4个位的值为14，则14 – 10 + 97 = 98（e）

如4个位的值为15，则15 – 10 + 97 = 98（f）

// 如果4个位的值 < 10，则加上字符0对应的十进制，结果就是这4个位的值对应的ASCII中的数值。字符0的对应的ASCII数值为48

如4个位的值为0，则0 + 48 = 48（0）

如4个位的值为1，则1 + 48 = 49（1）

如4个位的值为2，则2 + 48 = 50（2）

如4个位的值为3，则3 + 48 = 51（3）

如4个位的值为4，则4 + 48 = 52（4）

// 为什么 > 9就去减10加a，因为我们希望大于9的值用a ~ f表示

// 为什么 <= 9 就加'0'，因为我们希望小于或等于9的值用0 ~ 9表示

**char** hight4BitHex = (**char**) (hight4Bit >= 10 ? hight4Bit - 10 + 'a' : hight4Bit + '0');

**char** low4BitHex = (**char**) (low4Bit >= 10 ? low4Bit - 10 + 'a' : low4Bit + '0');

sb.append(hight4BitHex).append(low4BitHex);

}

**return** sb.toString();

}

}

注：MessageDigest有update(byte[])方法，跟加密时用到的类的使用相同，就是可以先不停调用update方法不停地添加字节，添加够了再调用digest()无参方法生成数字摘要。

md5生成的结果也称为I 哈希值

## 使用了Integer的方法将十进制转换为16进制字符，代码更简单：

/\*\* 把字节转换为16进制字符串 \*/

**private** **static** String byteToHex(**byte**[] bytes) {

StringBuilder sb = **new** StringBuilder();

**for** (**int** b: bytes) { // 这里把byte提升成了int，如果是负数，int中多余的位会补-1，所以需要把除了低8位，高24位都要变成0

**int** hight4Bit = b & 0xFF; // 保留下低8位，把高24位变成0

String hexString = Integer.*toHexString*(hight4Bit);

//如果一个byte只用到了低4位，则只需要一个16进制数值即可表示，但我们希望用两个，所以补0

**if** (hexString.length() == 1) {

sb.append("0");

}

sb.append(hexString);

}

**return** sb.toString();

}

## 反向操作，把16进制转换回二进制

/\*\* 把16进制的字符串转换回原始字节码 \*/

**private** **static** **byte**[] hex2byte(String str) {

**int** size = str.length() / 2; // 每两个字符转换为一个字节

**byte**[] bytes = **new** **byte**[size];

**for** (**int** i = 0; i < size; ++i) {

int startIndex = 2 \* i;

String \_2bitHex = str.substring(startIndex, startIndex + 2); // 获取两位16进制字符

bytes[i] = (**byte**) (Integer.parseInt(\_2bitHex, 16) & 0xFF);

}

**return** bytes;

}

## MD5生成文件的数字摘要

我们看到MD5是接收字节的，所以不管什么数据，都是字节，都可以进行MD5，只要把文件读取成字节即可。但是有另外两个更便捷的类DigestInputStream、DigestOutputStream，它们的使用跟加密时类型的流的使用也差不多，这两个类是用流读数据或写数据，我们的MD5需要完整的字节码，所以这两个流需要获取到完整的字节吗，它们要包装基本的IO流，等到读或者写完成的时候，再调用digest方法获取md5的哈希值。不论是读还是写，都可以获取哈希值。读，用于获取已存在的文件的哈希值，写，用于在生成一个文件的时候就获取哈希值。示例如下：

**public** **class** Demo {

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

String md5 = *md5*("privateKey.key");

System.***out***.println("md5的16进制字符串：" + md5 + ", 长度：" + md5.length());

}

**private** **static** String md5(String filePath) **throws** Exception {

MessageDigest md = MessageDigest.*getInstance*("MD5");

FileInputStream fis = **new** FileInputStream(filePath);

DigestInputStream dis = **new** DigestInputStream(fis, md);

**byte**[] buf = **new** **byte**[1024];

**while** (dis.read(buf) != -1);

dis.close();

fis.close();

**byte**[] md5Bytes = md.digest();

String hexMd5 = *byteToHex*(md5Bytes);

**return** hexMd5;

}

**private** **static** String byteToHex(**byte**[] bytes) {

StringBuilder sb = **new** StringBuilder();

**for** (**int** b: bytes) {

**int** hight4Bit = b & 0xFF;

String hexString = Integer.*toHexString*(hight4Bit);

**if** (hexString.length() == 1) {

sb.append("0");

}

sb.append(hexString);

}

**return** sb.toString();

}

}

## 基于MAC（消息验证码）的数字摘要

MAC，Message Authentication Code ，密钥被用作消息摘要生成过程的一部分。

防止内容和摘要同时被篡改，在一定程度上起到了验证发送者身份。如用普通的MD5，发送数据时对数据生成了md5摘要，然后和数据一起发给别人，如果中间被拦截，然后修改了数据，你是可以通过md5摘要查看数据是否被修改了，但是如果拦截的人，把数据修改后，又把修改后的数据也生成md5，替换原来的md5，则你就没法验证数据是否被修改了。所以使用基于消息验证码的数字摘要就能解决这个问题，因为这样的摘要需要使用密钥参与，别人拦截数据时它没有密钥，没密钥生成的数字摘要跟使用密钥后生成的数字摘要是不一样的，所以我们能知道数据是否被修改。

步骤： KeyGenerator.getInstance🡪KeyGeneraotr.generateKey 🡪 Mac.getInstance 🡪 Mac.init(secretyKey) 🡪Mac.update🡪Mac.doFinal

简单理解：

就是把md5当成签名，我们把一些数据传给服务器，可以给这些数据生成md5，但是这样并没有什么用，因为别人修改了数据之后可以重新生成md5，解决办法就是我们可以在生成md5中加入一些其它的字符（密码），这样别人修改了数据虽然可以再生成md5，但是它没有密码，所以我们在验证的时候能知道数据被修改了。这种使用很少，一般这种需求使用数字签名的方式来完成。Rsa使用私钥签名，使用公钥验证签名

示例代码如下：

**private** **static** String macDigest(**byte**[] bytes) **throws** Exception{

String password = "123456f";

PBEKeySpec keySpec = **new** PBEKeySpec(password.toCharArray());

SecretKeyFactory keyFactory = SecretKeyFactory.*getInstance*("PBEWithMD5AndDES");

SecretKey key = keyFactory.generateSecret(keySpec);

Mac mac = Mac.*getInstance*("HmacMD5");

mac.init(key);

**byte**[] md5Bytes = mac.doFinal(bytes);

String hexMd5 = *byteToHex*(md5Bytes);

**return** hexMd5;

}

## SHA 1生产数字摘要

使用SHA1算法产生的数字摘要长度是20个字节。

与MD5 的使用一模一样，只需要把MessageDigest.*getInstance*("MD5")中的参数改成："SHA-1"

如果是要生成基于MAC的数字摘要，则把Mac mac = Mac.*getInstance*("HmacMD5");中的参数改成："HmacSHA1"

# 数字签名

数字签名的基础是公钥和私钥的非对称加密。

对于加密来说，加密一般使用公钥加密，私钥解密，如果使用私钥加密，则这个加密就没有意义了，因为很多人都有公钥，都可以解密。

对于数字签名来说，使用私钥加密是有用的，作用就是别人可以用公钥来认证你的身份，因为公钥能解密，说明你就是拥有私钥的那个人，如果你的私钥被别人偷了，那也没办法。就像银行的公章，被偷来盖章转走了钱了，银行的人说不是他们干的，那也没办法，确实是你们的公章，谁让你们被别人偷走呢。所以私钥加密公钥解密可以用来认证一个人的身份。

预防数据被修改那使用私钥加密就行了，为什么还需要签名呢？因为非对称加密是比较耗时的，SHA签名的原理是这样的：使用SHA生成数字摘要(这个速度是比较快的)，防止数据被人修改，但是任何人都可以使用SHA生成数字摘要，怎么知道这个数字摘要就是你生成的呢？我们使用私钥对这个签名进行加密，虽然说加密能被所有拥有公钥的人打开，但是无所谓了，我们的目的是让有公钥的人能验证我的身份即可，他能打开，说明这个数字摘要就是拥有私钥的人生成的，这样身份得以确定。

SHA数字签名示例代码：

**public** **class** Demo {

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

String jieTiao = "我是小明，向小红借300万，下个月1号前还";

**byte**[] sign = *sign*(jieTiao);

**boolean** result = *verify*(jieTiao, sign);

System.***out***.println(result ? "验证通过" : "验证失败");

}

/\*\* 读取密钥 \*/

**private** **static** Key readKey(String keyFileName) **throws** Exception {

ObjectInputStream ois = **new** ObjectInputStream(**new** FileInputStream(keyFileName));

Key key = (Key) ois.readObject();

ois.close();

**return** key;

}

**public** **static** **byte**[] sign(String msg) **throws** Exception {

// 参数"SHA1withRSA"前面的SHA1，代表数字摘要是使用SHA1算法进行计算的，后面的RSA代表使用RSA对这个签名进行加密

// 所以参数也可以使用"MD5withRSA"（代表数字摘要是使用md5进行计算的的），当然查看文档还有更多参数可用

Signature signature = Signature.*getInstance*("SHA1withRSA");

signature.initSign((PrivateKey) *readKey*("privateKey.key"));

signature.update(msg.getBytes());

**byte**[] sign = signature.sign();

**return** sign;

}

**private** **static** **boolean** verify(String jieTiao, **byte**[] sign) **throws** Exception {

Signature signature = Signature.*getInstance*("SHA1withRSA");

signature.initVerify((PublicKey) *readKey*("publicKey.key"));

signature.update(jieTiao.getBytes());

**boolean** result = signature.verify(sign);

**return** result;

}

}

现在还有一个问题：就是安全的问题，张三生产了一对密钥，把公钥给了李四，然后写了一张借条使用私钥对进行了签名，签名给了李四，李四一验证通过，然后借了300万给张三。问题来了，到还款期的时候，张三说李四的公钥不是它给的，这怎么确定是不是张三给的呀，没法确定，因为张三也可以说它没有对应的私钥。所以为了安全，生产密钥的人需要把公钥拿到一个权威的机构进行认证，认证过的公钥就是安全的，认证通过会给你发一个证书，证书是一个文件形式，里面装着你的公钥，并且有权威机构的签名，所以可以把这个证书拿到这个权威机构的网站进行验证，验证通过则是安全的证书，则证书里的公钥也是安全的公钥。

# 数字证书

数字证书原理：你把你的公钥拿到权威机构进行认证，权威机构会要求你提供身份证啊等各种信息，验证通过了，它就会把你的公钥进行签名，然后把你的公钥、签名、认证机构信息等封装成一个文件，这个文件就是证书。所以这个证书里包含了我们的公钥，还有机构信息和机构对这个公钥的签名。我们怎么确认这个证书是可信任的呢？我们可以根据证书上的机构信息找到权威机构网站，权威机构网站上提供了权威机构的公钥，使用这个公钥可以验证我们的证书里面的签名，如果能验证通过，则这个证书就是可信任的。

在国外有一些权威机构，全世界的人都信任它们，在任何的电脑在出厂时就已经信任了一些全球知名机构的证书，也就是把这些机构的带有公钥的证书装在了电脑上，查看IE上的证书，如下：

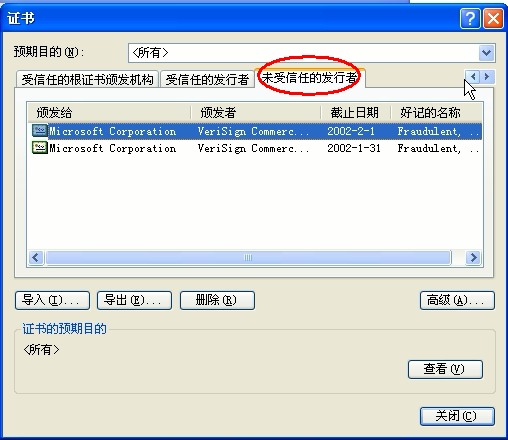
 

“受信任的根证书颁发机构”，意思是这些机构都是直接颁发证书的机构。

假设中国的公安部拿着公钥去一个机构进行了认证得到了证书，那我们可以找公安部给我们的公钥进行认证，公安部认证后也给我们一个证书，这个证书是使用公安部的私钥进行签名的。电脑会不会信任我们的证书呢？会，因为电脑看到我们的证书，它能知道是公安部签名的，而且电脑能知道公安部是权威机构签名的，所以可以信任。 像这样类似公安部颁发的证书机构就称为“中间证书颁发机构”，如下图：



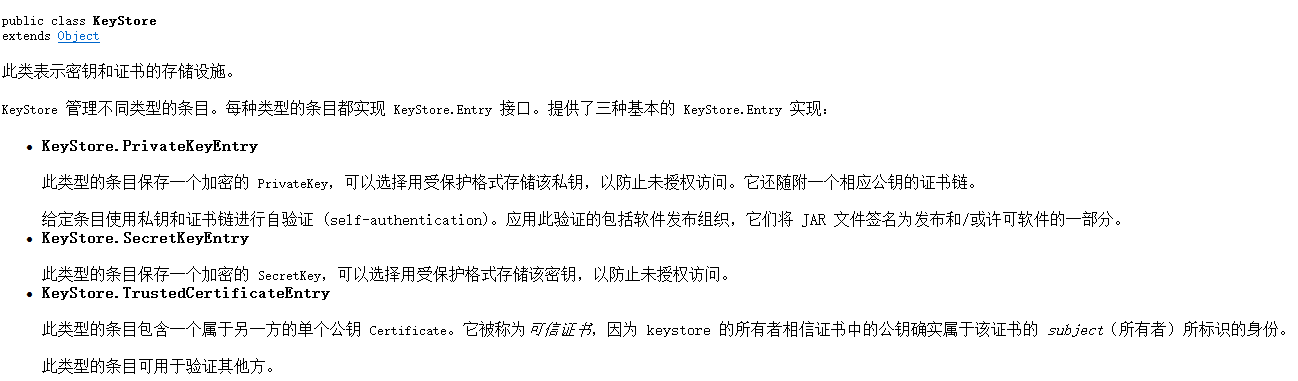
下面是公司自己发行的证书，电脑是不认的：



## 创建数字证书

通过KeyStore创建数字证书，它产生的证书保存在.keystore文件中，里面装有私钥（PrivateKeyEntry、密钥（SecretKeyEntry）、证书（TrustedCertificateEntry）。.keystore是一个密钥仓库，里面可以存储多个密钥。

KeyStore中可以存储3种东西，如下：



KeyStore.PrivateKeyEntry 保存自己的私钥，且它还随附一个相应的公钥的证书链。

KeyStore.SecretKeyEntry 用于保存对称加密的密钥

KeyStore.TrustedCertificateEntry 用于保存别人的证书，当我们去权威机构验证别人的证书且通过时，可以保存起来，这样下次再验证时，一比对证书和我们这保存的一样就验证通过了，就不用连网去访问权限机构了。

使用JDK提供的keytool工具可以产生密钥对和证书，使用：

在命令行输入keytool可查看它的使用帮助，如下：



查看genkeypair命令的用法，在命令行输入：keytool -genkeypair -help，如下：



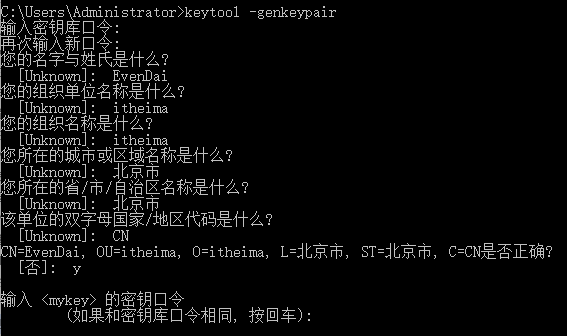
可以看到有两个密码：

storepass 为密钥库的密码

keypass 为密钥库中某个密钥的密码。因为一个密钥库中可以有多个密钥

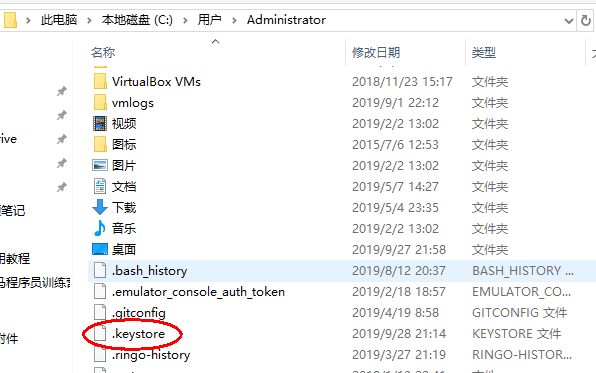
这就类似于我们家里有一个柜子，柜子有密码，柜子里面可以放多个写日记的笔记本，这些笔记本本身又有密码锁。

生产一个证书：



一般key的密码和仓库的密码设置成一样，如果设置不一样可能你的代码过不去，因为不一样的话代码写起来会比较复杂，容易出问题。

这样在电脑的用户目录下就会生产一个.keystore的文件，如下：



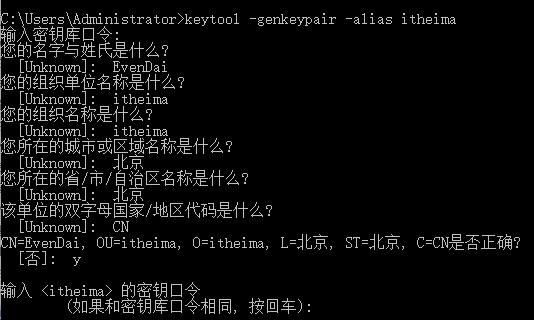
这样密钥仓库就有了。我们在输入keytool -genkeypair时它会判断是否有keystore，如果没有就创建，如果有了就不创建了，这里创建密钥对时我们并没有指定密钥的别名，它会使用默认名字为mykey做为别名。再我们在命令行再次输入keytool -genkeypair时，提示如下：



这时keytool工具检测到已经存在keystore了，则创建的keypair可以直接保存在之前创建的仓库中，所以提示输入之前创建的仓库的密码，输入密码后提示如下：

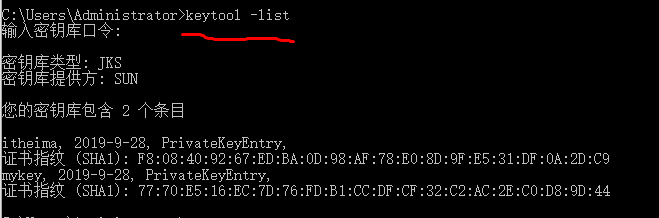


提示错误，这是因为没指定要生成的keypair的名字的话默认使用mykey，而之前已经创建过了。所以这次需要创建另外的别名，如下：



这样，在.keystore中就存储了两个密钥（其实是两个证书），分别是：mykey和itheima。

要查看仓库里有哪些证书，输入命令：



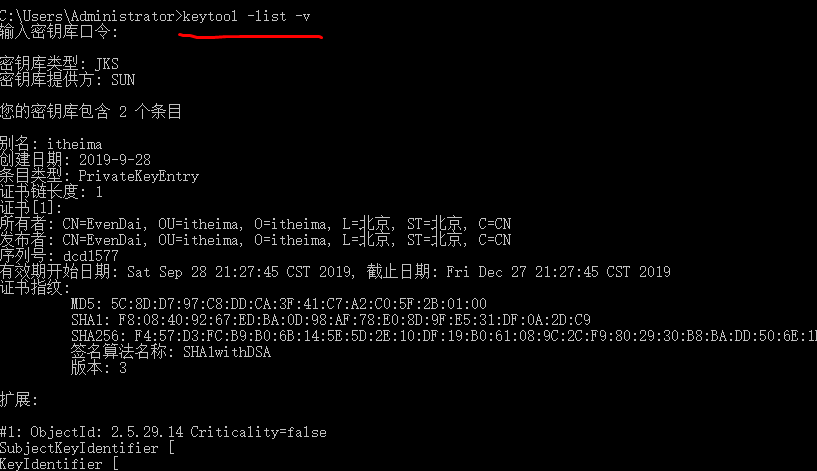
如果觉得某个证书的名字写错了，可以修改的，如下：



具体使用如下：



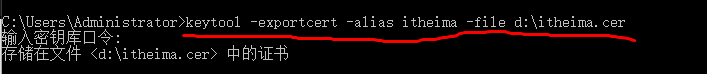
查看更详细的证书信息：



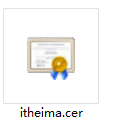
删除一个证书也可以找对应的命令去删除。

## 导出证书

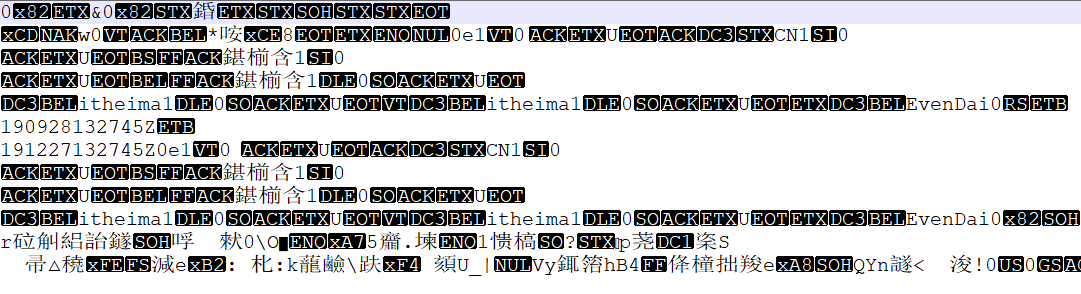
当我们需要把我们的证书给人看时，不是直接发.keystore，而是导出这里面的某个证书，然后发给别人。也可以导出一个证书签名请求，可以拿去给别人签名，签好了再导回来。



这样itheima证书就导出到D盘上了，如下：

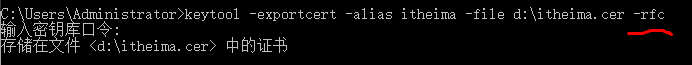


用记事本打开这个文件如下：



这是以二进制的方式直接保存的，我们也可以以字符串的方式保存，也就是把这些二进制通过一定的编码转换为可打印的字符串保存到文件中，这样我们的证书就可以打印出来发给别人，别人扫描这个字符串，通过字符串再恢复这个证书，因为二进制直接打印的话有些内容是打印机打印不出来的。

在字符串方式保存证书：



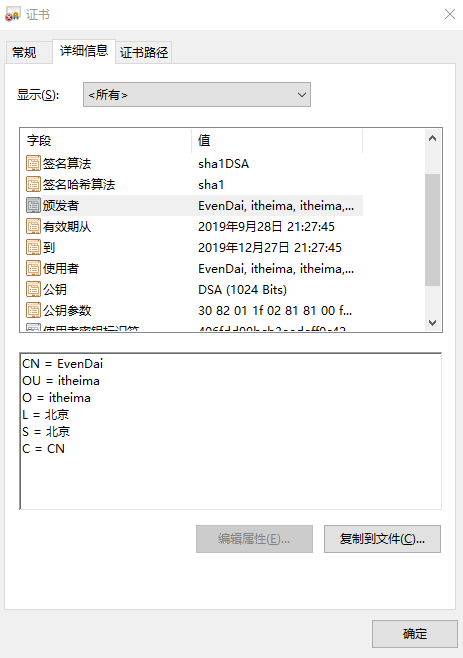
用记事本打开后内容如下：



这两种方式保存的证书IE浏览器都可以识别，双击打开，效果如下：



这里可以看到这个证书是不受信任的，因为没有权威机构的签名。点击详细信息，如下：



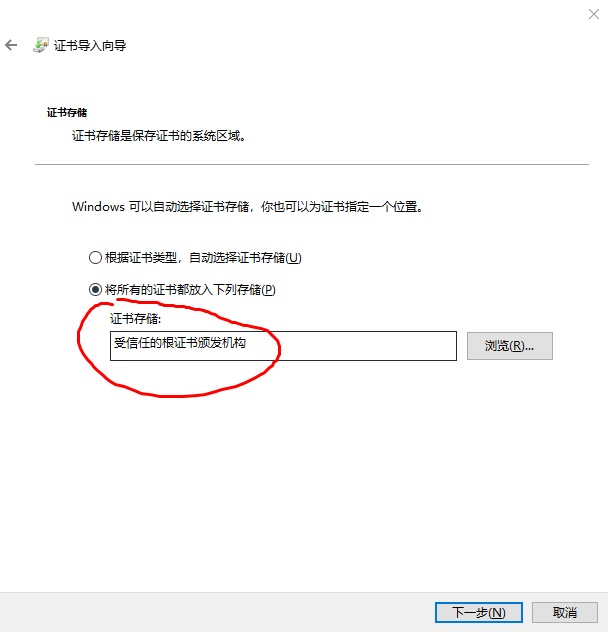
这里可以看到证书的详细信息，如证书的签名算法使用的是sha1DSA，公钥消息摘要使用的是DSA算法生成的，1024位长度。

## 安装证书

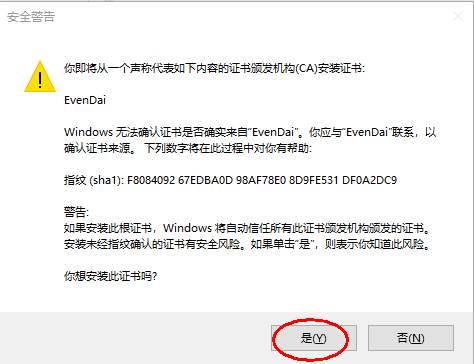
如果这个证书我们确认它是安全，如，你知道这是你的好朋友给你的证书，则我们可以信任它，把它装到我们电脑上，操作如下：

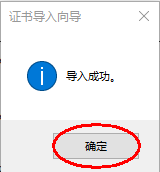












当我们再次双击这个证书时，将提示是受信任的，如下：



以后别人的电脑跟我们的电脑通讯时出示这个证书，我们就会验证通过，如果第三个人又生产了证书，找我们之前安装的证书那个人去签名他的证书，则第三个人的证书我们的电脑也会验证通过。

## 删除证书

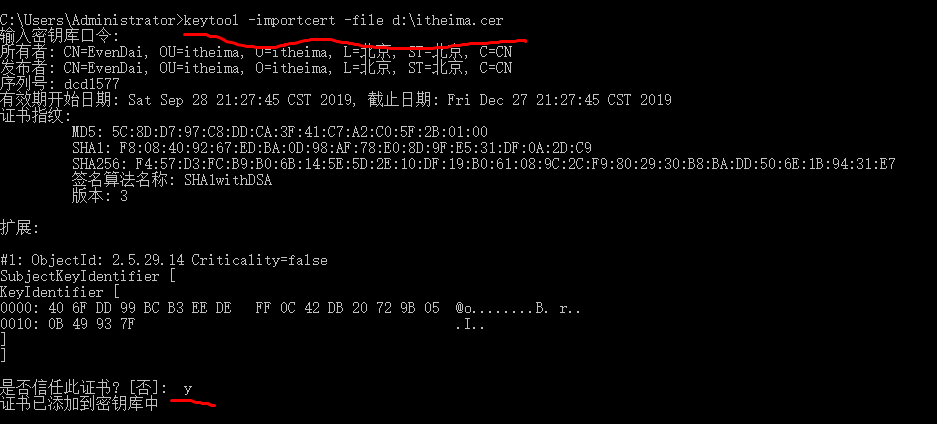


此时再查看仓库确实只有一个证书了：



## 导入证书

导入之前我们导出的证书：



在导入时，因为我们没有指定这个证书的别名，所以会使用默认别名为mykey，查看如下：



导入别人证书的作用：以后你的电脑就会信任这个证书，以及这个证书签发的其他证书。