**[Java加密技术（一）——BASE64与单向加密算法MD5&SHA&MAC](http://snowolf.iteye.com/blog/379860)**

**博客分类：**

* [Java／Security](http://snowolf.iteye.com/category/68576)

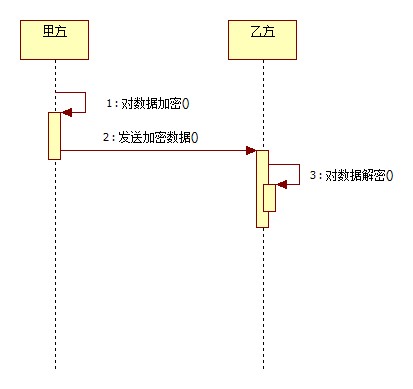
[Java](http://www.iteye.com/blogs/tag/Java)[base64](http://www.iteye.com/blogs/tag/base64)[mac](http://www.iteye.com/blogs/tag/mac)[md5](http://www.iteye.com/blogs/tag/md5)[sha](http://www.iteye.com/blogs/tag/sha)

    加密解密，曾经是我一个毕业设计的重要组件。在工作了多年以后回想当时那个加密、解密算法，实在是太单纯了。   
    言归正传，这里我们主要描述Java已经实现的一些加密解密算法，最后介绍数字证书。   
    如基本的单向加密算法：

* BASE64 严格地说，属于编码格式，而非加密算法
* MD5(Message Digest algorithm 5，信息摘要算法)
* SHA(Secure Hash Algorithm，安全散列算法)
* HMAC(Hash Message Authentication Code，散列消息鉴别码)

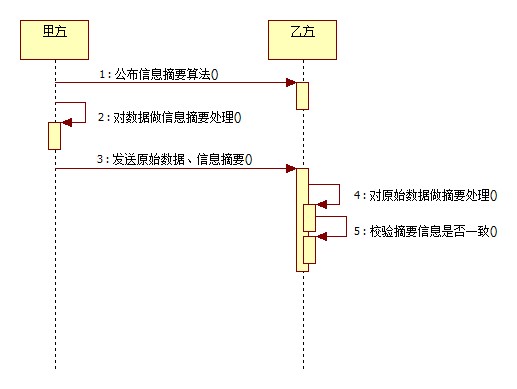
    复杂的对称加密（DES、PBE）、非对称加密算法：

* DES(Data Encryption Standard，数据加密算法)
* PBE(Password-based encryption，基于密码验证)
* RSA(算法的名字以发明者的名字命名：Ron Rivest, AdiShamir 和Leonard Adleman)
* DH(Diffie-Hellman算法，密钥一致协议)
* DSA(Digital Signature Algorithm，数字签名)
* ECC(Elliptic Curves Cryptography，椭圆曲线密码编码学)

    本篇内容简要介绍**BASE64**、**MD5**、**SHA**、**HMAC**几种方法。   
    **MD5**、**SHA**、**HMAC**这三种加密算法，可谓是非可逆加密，就是不可解密的加密方法。我们通常只把他们作为加密的基础。单纯的以上三种的加密并不可靠。   
  
**BASE64**   
按照RFC2045的定义，Base64被定义为：Base64内容传送编码被设计用来把任意序列的8位字节描述为一种不易被人直接识别的形式。（The Base64 Content-Transfer-Encoding is designed to represent arbitrary sequences of octets in a form that need not be humanly readable.）   
常见于邮件、http加密，截取http信息，你就会发现登录操作的用户名、密码字段通过BASE64加密的。   
  
   
  
通过java代码实现如下：

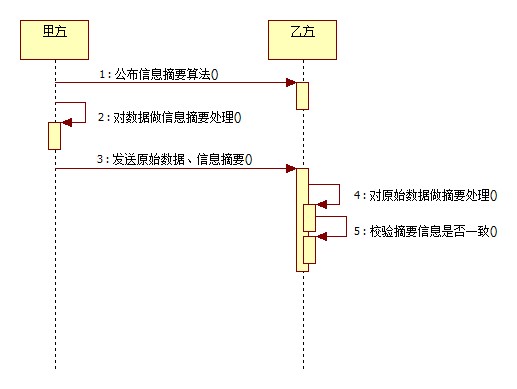
**Java代码  [收藏代码](javascript:void())**

1. /\*\*
2. \* BASE64解密
3. \*
4. \* @param key
5. \* @return
6. \* @throws Exception
7. \*/
8. **public** **static** **byte**[] decryptBASE64(String key) **throws** Exception {
9. **return** (**new** BASE64Decoder()).decodeBuffer(key);
10. }
12. /\*\*
13. \* BASE64加密
14. \*
15. \* @param key
16. \* @return
17. \* @throws Exception
18. \*/
19. **public** **static** String encryptBASE64(**byte**[] key) **throws** Exception {
20. **return** (**new** BASE64Encoder()).encodeBuffer(key);
21. }

主要就是BASE64Encoder、BASE64Decoder两个类，我们只需要知道使用对应的方法即可。另，BASE加密后产生的字节位数是8的倍数，如果不够位数以**=**符号填充。   
  
**MD5**   
MD5 -- message-digest algorithm 5 （信息-摘要算法）缩写，广泛用于加密和解密技术，常用于文件校验。校验？不管文件多大，经过MD5后都能生成唯一的MD5值。好比现在的ISO校验，都是MD5校验。怎么用？当然是把ISO经过MD5后产生MD5的值。一般下载linux-ISO的朋友都见过下载链接旁边放着MD5的串。就是用来验证文件是否一致的。   
  
   
  
通过java代码实现如下：

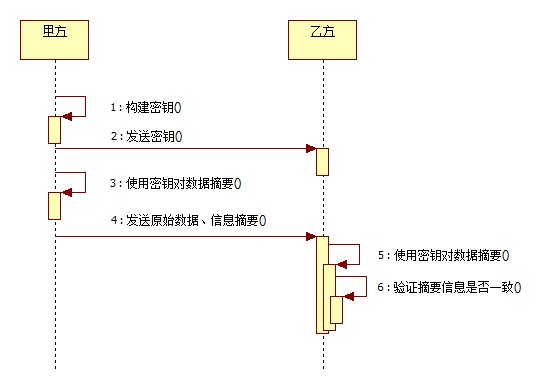
**Java代码  [收藏代码](javascript:void())**

1. /\*\*
2. \* MD5加密
3. \*
4. \* @param data
5. \* @return
6. \* @throws Exception
7. \*/
8. **public** **static** **byte**[] encryptMD5(**byte**[] data) **throws** Exception {
10. MessageDigest md5 = MessageDigest.getInstance(KEY\_MD5);
11. md5.update(data);
13. **return** md5.digest();
15. }

通常我们不直接使用上述MD5加密。通常将MD5产生的字节数组交给BASE64再加密一把，得到相应的字符串。   
  
**SHA**   
SHA(Secure Hash Algorithm，安全散列算法），数字签名等密码学应用中重要的工具，被广泛地应用于电子商务等信息安全领域。虽然，SHA与MD5通过碰撞法都被破解了，http://snowolf.iteye.com/images/smiles/icon_sad.gif 但是SHA仍然是公认的安全加密算法，较之MD5更为安全。http://snowolf.iteye.com/images/smiles/icon_biggrin.gif   
  
   
  
通过java代码实现如下：

**Java代码  [收藏代码](javascript:void())**

1. /\*\*
2. \* SHA加密
3. \*
4. \* @param data
5. \* @return
6. \* @throws Exception
7. \*/
8. **public** **static** **byte**[] encryptSHA(**byte**[] data) **throws** Exception {
10. MessageDigest sha = MessageDigest.getInstance(KEY\_SHA);
11. sha.update(data);
13. **return** sha.digest();
15. }
16. }

**HMAC**   
HMAC(Hash Message Authentication Code，散列消息鉴别码，基于密钥的Hash算法的认证协议。消息鉴别码实现鉴别的原理是，用公开函数和密钥产生一个固定长度的值作为认证标识，用这个标识鉴别消息的完整性。使用一个密钥生成一个固定大小的小数据块，即MAC，并将其加入到消息中，然后传输。接收方利用与发送方共享的密钥进行鉴别认证等。   
  
   
  
通过java代码实现如下：

**Java代码  [收藏代码](javascript:void())**

1. /\*\*
2. \* 初始化HMAC密钥
3. \*
4. \* @return
5. \* @throws Exception
6. \*/
7. **public** **static** String initMacKey() **throws** Exception {
8. KeyGenerator keyGenerator = KeyGenerator.getInstance(KEY\_MAC);
10. SecretKey secretKey = keyGenerator.generateKey();
11. **return** encryptBASE64(secretKey.getEncoded());
12. }
14. /\*\*
15. \* HMAC加密
16. \*
17. \* @param data
18. \* @param key
19. \* @return
20. \* @throws Exception
21. \*/
22. **public** **static** **byte**[] encryptHMAC(**byte**[] data, String key) **throws** Exception {
24. SecretKey secretKey = **new** SecretKeySpec(decryptBASE64(key), KEY\_MAC);
25. Mac mac = Mac.getInstance(secretKey.getAlgorithm());
26. mac.init(secretKey);
28. **return** mac.doFinal(data);
30. }

给出一个完整类，如下：

**Java代码  [收藏代码](javascript:void())**

1. **import** java.security.MessageDigest;
3. **import** javax.crypto.KeyGenerator;
4. **import** javax.crypto.Mac;
5. **import** javax.crypto.SecretKey;
7. **import** sun.misc.BASE64Decoder;
8. **import** sun.misc.BASE64Encoder;
10. /\*\*
11. \* 基础加密组件
12. \*
13. \* @author 梁栋
14. \* @version 1.0
15. \* @since 1.0
16. \*/
17. **public** **abstract** **class** Coder {
18. **public** **static** **final** String KEY\_SHA = "SHA";
19. **public** **static** **final** String KEY\_MD5 = "MD5";
21. /\*\*
22. \* MAC算法可选以下多种算法
23. \*
24. \* <pre>
25. \* HmacMD5
26. \* HmacSHA1
27. \* HmacSHA256
28. \* HmacSHA384
29. \* HmacSHA512
30. \* </pre>
31. \*/
32. **public** **static** **final** String KEY\_MAC = "HmacMD5";
34. /\*\*
35. \* BASE64解密
36. \*
37. \* @param key
38. \* @return
39. \* @throws Exception
40. \*/
41. **public** **static** **byte**[] decryptBASE64(String key) **throws** Exception {
42. **return** (**new** BASE64Decoder()).decodeBuffer(key);
43. }
45. /\*\*
46. \* BASE64加密
47. \*
48. \* @param key
49. \* @return
50. \* @throws Exception
51. \*/
52. **public** **static** String encryptBASE64(**byte**[] key) **throws** Exception {
53. **return** (**new** BASE64Encoder()).encodeBuffer(key);
54. }
56. /\*\*
57. \* MD5加密
58. \*
59. \* @param data
60. \* @return
61. \* @throws Exception
62. \*/
63. **public** **static** **byte**[] encryptMD5(**byte**[] data) **throws** Exception {
65. MessageDigest md5 = MessageDigest.getInstance(KEY\_MD5);
66. md5.update(data);
68. **return** md5.digest();
70. }
72. /\*\*
73. \* SHA加密
74. \*
75. \* @param data
76. \* @return
77. \* @throws Exception
78. \*/
79. **public** **static** **byte**[] encryptSHA(**byte**[] data) **throws** Exception {
81. MessageDigest sha = MessageDigest.getInstance(KEY\_SHA);
82. sha.update(data);
84. **return** sha.digest();
86. }
88. /\*\*
89. \* 初始化HMAC密钥
90. \*
91. \* @return
92. \* @throws Exception
93. \*/
94. **public** **static** String initMacKey() **throws** Exception {
95. KeyGenerator keyGenerator = KeyGenerator.getInstance(KEY\_MAC);
97. SecretKey secretKey = keyGenerator.generateKey();
98. **return** encryptBASE64(secretKey.getEncoded());
99. }
101. /\*\*
102. \* HMAC加密
103. \*
104. \* @param data
105. \* @param key
106. \* @return
107. \* @throws Exception
108. \*/
109. **public** **static** **byte**[] encryptHMAC(**byte**[] data, String key) **throws** Exception {
111. SecretKey secretKey = **new** SecretKeySpec(decryptBASE64(key), KEY\_MAC);
112. Mac mac = Mac.getInstance(secretKey.getAlgorithm());
113. mac.init(secretKey);
115. **return** mac.doFinal(data);
117. }
118. }

再给出一个测试类：

**Java代码  [收藏代码](javascript:void())**

1. **import** **static** org.junit.Assert.\*;
3. **import** org.junit.Test;
5. /\*\*
6. \*
7. \* @author 梁栋
8. \* @version 1.0
9. \* @since 1.0
10. \*/
11. **public** **class** CoderTest {
13. @Test
14. **public** **void** test() **throws** Exception {
15. String inputStr = "简单加密";
16. System.err.println("原文:\n" + inputStr);
18. **byte**[] inputData = inputStr.getBytes();
19. String code = Coder.encryptBASE64(inputData);
21. System.err.println("BASE64加密后:\n" + code);
23. **byte**[] output = Coder.decryptBASE64(code);
25. String outputStr = **new** String(output);
27. System.err.println("BASE64解密后:\n" + outputStr);
29. // 验证BASE64加密解密一致性
30. assertEquals(inputStr, outputStr);
32. // 验证MD5对于同一内容加密是否一致
33. assertArrayEquals(Coder.encryptMD5(inputData), Coder
34. .encryptMD5(inputData));
36. // 验证SHA对于同一内容加密是否一致
37. assertArrayEquals(Coder.encryptSHA(inputData), Coder
38. .encryptSHA(inputData));
40. String key = Coder.initMacKey();
41. System.err.println("Mac密钥:\n" + key);
43. // 验证HMAC对于同一内容，同一密钥加密是否一致
44. assertArrayEquals(Coder.encryptHMAC(inputData, key), Coder.encryptHMAC(
45. inputData, key));
47. BigInteger md5 = **new** BigInteger(Coder.encryptMD5(inputData));
48. System.err.println("MD5:\n" + md5.toString(16));
50. BigInteger sha = **new** BigInteger(Coder.encryptSHA(inputData));
51. System.err.println("SHA:\n" + sha.toString(32));
53. BigInteger mac = **new** BigInteger(Coder.encryptHMAC(inputData, inputStr));
54. System.err.println("HMAC:\n" + mac.toString(16));
55. }
56. }

控制台输出：

**Console代码  [收藏代码](javascript:void())**

1. 原文:
2. 简单加密
3. BASE64加密后:
4. 566A5Y2V5Yqg5a+G
6. BASE64解密后:
7. 简单加密
8. Mac密钥:
9. uGxdHC+6ylRDaik++leFtGwiMbuYUJ6mqHWyhSgF4trVkVBBSQvY/a22xU8XT1RUemdCWW155Bke
10. pBIpkd7QHg==
12. MD5:
13. -550b4d90349ad4629462113e7934de56
14. SHA:
15. 91k9vo7p400cjkgfhjh0ia9qthsjagfn
16. HMAC:
17. 2287d192387e95694bdbba2fa941009a

注意   
编译时，可能会看到如下提示：

**引用**

警告：sun.misc.BASE64Decoder 是 Sun 的专用 API，可能会在未来版本中删除   
  
import sun.misc.BASE64Decoder;   
               ^   
警告：sun.misc.BASE64Encoder 是 Sun 的专用 API，可能会在未来版本中删除   
  
import sun.misc.BASE64Encoder;   
               ^

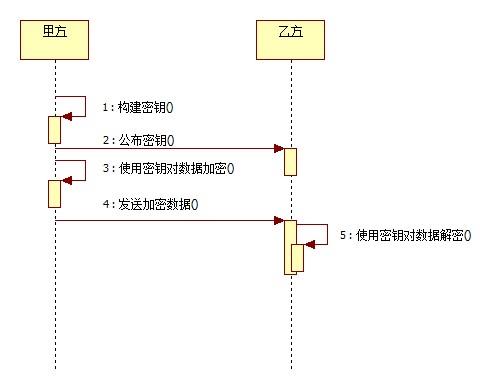
BASE64Encoder和BASE64Decoder是非官方JDK实现类。虽然可以在JDK里能找到并使用，但是在API里查不到。JRE 中 sun 和 com.sun 开头包的类都是未被文档化的，他们属于 java, javax 类库的基础，其中的实现大多数与底层平台有关，一般来说是不推荐使用的。   
  
  
    BASE64的加密解密是双向的，可以求反解。   
    MD5、SHA以及HMAC是单向加密，任何数据加密后只会产生唯一的一个加密串，通常用来校验数据在传输过程中是否被修改。其中HMAC算法有一个密钥，增强了数据传输过程中的安全性，强化了算法外的不可控因素。http://snowolf.iteye.com/images/smiles/icon_biggrin.gif   
    单向加密的用途主要是为了校验数据在传输过程中是否被修改。

[**Java加密技术（二）——对称加密算法DES&AES**](http://snowolf.iteye.com/blog/380034)

**博客分类：**

* [Java／Security](http://snowolf.iteye.com/category/68576)

[des](http://www.iteye.com/blogs/tag/des)[desede](http://www.iteye.com/blogs/tag/desede)[java](http://www.iteye.com/blogs/tag/java)[aes](http://www.iteye.com/blogs/tag/aes)

    接下来我们介绍对称加密算法，最常用的莫过于DES数据加密算法。   
**DES**   
DES-Data Encryption Standard,即数据加密算法。是IBM公司于1975年研究成功并公开发表的。DES算法的入口参数有三个:Key、Data、Mode。其中Key为8个字节共64位,是DES算法的工作密钥;Data也为8个字节64位,是要被加密或被解密的数据;Mode为DES的工作方式,有两种:加密或解密。   
　　DES算法把64位的明文输入块变为64位的密文输出块,它所使用的密钥也是64位。   
  
   
  
通过java代码实现如下：**Coder类见**[**Java加密技术（一）**](http://snowolf.iteye.com/blog/379860)

**Java代码  [收藏代码](javascript:void())**

1. **import** java.security.Key;
2. **import** java.security.SecureRandom;
4. **import** javax.crypto.Cipher;
5. **import** javax.crypto.KeyGenerator;
6. **import** javax.crypto.SecretKey;
7. **import** javax.crypto.SecretKeyFactory;
8. **import** javax.crypto.spec.DESKeySpec;

11. /\*\*
12. \* DES安全编码组件
13. \*
14. \* <pre>
15. \* 支持 DES、DESede(TripleDES,就是3DES)、AES、Blowfish、RC2、RC4(ARCFOUR)
16. \* DES                  key size must be equal to 56
17. \* DESede(TripleDES)    key size must be equal to 112 or 168
18. \* AES                  key size must be equal to 128, 192 or 256,but 192 and 256 bits may not be available
19. \* Blowfish             key size must be multiple of 8, and can only range from 32 to 448 (inclusive)
20. \* RC2                  key size must be between 40 and 1024 bits
21. \* RC4(ARCFOUR)         key size must be between 40 and 1024 bits
22. \* 具体内容 需要关注 JDK Document http://.../docs/technotes/guides/security/SunProviders.html
23. \* </pre>
24. \*
25. \* @author 梁栋
26. \* @version 1.0
27. \* @since 1.0
28. \*/
29. **public** **abstract** **class** DESCoder **extends** Coder {
30. /\*\*
31. \* ALGORITHM 算法 <br>
32. \* 可替换为以下任意一种算法，同时key值的size相应改变。
33. \*
34. \* <pre>
35. \* DES                  key size must be equal to 56
36. \* DESede(TripleDES)    key size must be equal to 112 or 168
37. \* AES                  key size must be equal to 128, 192 or 256,but 192 and 256 bits may not be available
38. \* Blowfish             key size must be multiple of 8, and can only range from 32 to 448 (inclusive)
39. \* RC2                  key size must be between 40 and 1024 bits
40. \* RC4(ARCFOUR)         key size must be between 40 and 1024 bits
41. \* </pre>
42. \*
43. \* 在Key toKey(byte[] key)方法中使用下述代码
44. \* <code>SecretKey secretKey = new SecretKeySpec(key, ALGORITHM);</code> 替换
45. \* <code>
46. \* DESKeySpec dks = new DESKeySpec(key);
47. \* SecretKeyFactory keyFactory = SecretKeyFactory.getInstance(ALGORITHM);
48. \* SecretKey secretKey = keyFactory.generateSecret(dks);
49. \* </code>
50. \*/
51. **public** **static** **final** String ALGORITHM = "DES";
53. /\*\*
54. \* 转换密钥<br>
55. \*
56. \* @param key
57. \* @return
58. \* @throws Exception
59. \*/
60. **private** **static** Key toKey(**byte**[] key) **throws** Exception {
61. DESKeySpec dks = **new** DESKeySpec(key);
62. SecretKeyFactory keyFactory = SecretKeyFactory.getInstance(ALGORITHM);
63. SecretKey secretKey = keyFactory.generateSecret(dks);
65. // 当使用其他对称加密算法时，如AES、Blowfish等算法时，用下述代码替换上述三行代码
66. // SecretKey secretKey = new SecretKeySpec(key, ALGORITHM);
68. **return** secretKey;
69. }
71. /\*\*
72. \* 解密
73. \*
74. \* @param data
75. \* @param key
76. \* @return
77. \* @throws Exception
78. \*/
79. **public** **static** **byte**[] decrypt(**byte**[] data, String key) **throws** Exception {
80. Key k = toKey(decryptBASE64(key));
82. Cipher cipher = Cipher.getInstance(ALGORITHM);
83. cipher.init(Cipher.DECRYPT\_MODE, k);
85. **return** cipher.doFinal(data);
86. }
88. /\*\*
89. \* 加密
90. \*
91. \* @param data
92. \* @param key
93. \* @return
94. \* @throws Exception
95. \*/
96. **public** **static** **byte**[] encrypt(**byte**[] data, String key) **throws** Exception {
97. Key k = toKey(decryptBASE64(key));
98. Cipher cipher = Cipher.getInstance(ALGORITHM);
99. cipher.init(Cipher.ENCRYPT\_MODE, k);
101. **return** cipher.doFinal(data);
102. }
104. /\*\*
105. \* 生成密钥
106. \*
107. \* @return
108. \* @throws Exception
109. \*/
110. **public** **static** String initKey() **throws** Exception {
111. **return** initKey(**null**);
112. }
114. /\*\*
115. \* 生成密钥
116. \*
117. \* @param seed
118. \* @return
119. \* @throws Exception
120. \*/
121. **public** **static** String initKey(String seed) **throws** Exception {
122. SecureRandom secureRandom = **null**;
124. **if** (seed != **null**) {
125. secureRandom = **new** SecureRandom(decryptBASE64(seed));
126. } **else** {
127. secureRandom = **new** SecureRandom();
128. }
130. KeyGenerator kg = KeyGenerator.getInstance(ALGORITHM);
131. kg.init(secureRandom);
133. SecretKey secretKey = kg.generateKey();
135. **return** encryptBASE64(secretKey.getEncoded());
136. }
137. }

延续上一个类的实现，我们通过MD5以及SHA对字符串加密生成密钥，这是比较常见的密钥生成方式。   
再给出一个测试类：

**Java代码  [收藏代码](javascript:void())**

1. **import** **static** org.junit.Assert.\*;

4. **import** org.junit.Test;
6. /\*\*
7. \*
8. \* @author 梁栋
9. \* @version 1.0
10. \* @since 1.0
11. \*/
12. **public** **class** DESCoderTest {
14. @Test
15. **public** **void** test() **throws** Exception {
16. String inputStr = "DES";
17. String key = DESCoder.initKey();
18. System.err.println("原文:\t" + inputStr);
20. System.err.println("密钥:\t" + key);
22. **byte**[] inputData = inputStr.getBytes();
23. inputData = DESCoder.encrypt(inputData, key);
25. System.err.println("加密后:\t" + DESCoder.encryptBASE64(inputData));
27. **byte**[] outputData = DESCoder.decrypt(inputData, key);
28. String outputStr = **new** String(outputData);
30. System.err.println("解密后:\t" + outputStr);
32. assertEquals(inputStr, outputStr);
33. }
34. }

得到的输出内容如下：

**Console代码  [收藏代码](javascript:void())**

1. 原文: DES
2. 密钥: f3wEtRrV6q0=
4. 加密后:    C6qe9oNIzRY=
6. 解密后:    DES

    由控制台得到的输出，我们能够比对加密、解密后结果一致。这是一种简单的加密解密方式，只有一个密钥。   
    其实DES有很多同胞兄弟，如DESede(TripleDES)、AES、Blowfish、RC2、RC4(ARCFOUR)。这里就不过多阐述了，大同小异，只要换掉ALGORITHM换成对应的值，同时做一个代码替换**SecretKey secretKey = new SecretKeySpec(key, ALGORITHM);**就可以了，此外就是密钥长度不同了。

**Java代码  [收藏代码](javascript:void())**

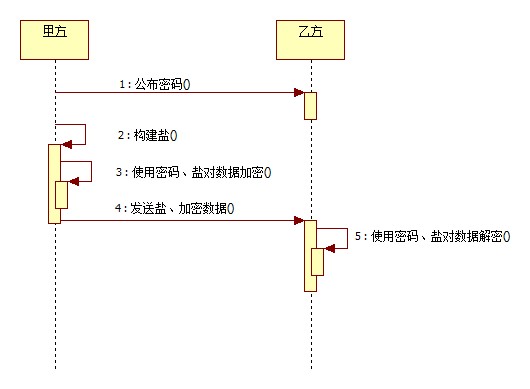
1. /\*\*
2. \* DES          key size must be equal to 56
3. \* DESede(TripleDES) key size must be equal to 112 or 168
4. \* AES          key size must be equal to 128, 192 or 256,but 192 and 256 bits may not be available
5. \* Blowfish     key size must be multiple of 8, and can only range from 32 to 448 (inclusive)
6. \* RC2          key size must be between 40 and 1024 bits
7. \* RC4(ARCFOUR) key size must be between 40 and 1024 bits
8. \*\*/

[**Java加密技术（三）——PBE算法**](http://snowolf.iteye.com/blog/380761)

**博客分类：**

* [Java／Security](http://snowolf.iteye.com/category/68576)

[Java](http://www.iteye.com/blogs/tag/Java)[pbe](http://www.iteye.com/blogs/tag/pbe)[对称加密](http://www.iteye.com/blogs/tag/%E5%AF%B9%E7%A7%B0%E5%8A%A0%E5%AF%86)

    除了DES，我们还知道有DESede(TripleDES,就是3DES)、AES、Blowfish、RC2、RC4(ARCFOUR)等多种对称加密方式，其实现方式大同小异，这里介绍对称加密的另一个算法——PBE   
  
**PBE**   
    PBE——Password-based encryption（基于密码加密）。其特点在于口令由用户自己掌管，不借助任何物理媒体；采用随机数（这里我们叫做盐）杂凑多重加密等方法保证数据的安全性。是一种简便的加密方式。   
  
   
  
通过java代码实现如下：**Coder类见**[**Java加密技术（一）**](http://snowolf.iteye.com/blog/379860)

**Java代码  [收藏代码](javascript:void())**

1. **import** java.security.Key;
2. **import** java.util.Random;
4. **import** javax.crypto.Cipher;
5. **import** javax.crypto.SecretKey;
6. **import** javax.crypto.SecretKeyFactory;
7. **import** javax.crypto.spec.PBEKeySpec;
8. **import** javax.crypto.spec.PBEParameterSpec;
10. /\*\*
11. \* PBE安全编码组件
12. \*
13. \* @author 梁栋
14. \* @version 1.0
15. \* @since 1.0
16. \*/
17. **public** **abstract** **class** PBECoder **extends** Coder {
18. /\*\*
19. \* 支持以下任意一种算法
20. \*
21. \* <pre>
22. \* PBEWithMD5AndDES
23. \* PBEWithMD5AndTripleDES
24. \* PBEWithSHA1AndDESede
25. \* PBEWithSHA1AndRC2\_40
26. \* </pre>
27. \*/
28. **public** **static** **final** String ALGORITHM = "PBEWITHMD5andDES";
30. /\*\*
31. \* 盐初始化
32. \*
33. \* @return
34. \* @throws Exception
35. \*/
36. **public** **static** **byte**[] initSalt() **throws** Exception {
37. **byte**[] salt = **new** **byte**[8];
38. Random random = **new** Random();
39. random.nextBytes(salt);
40. **return** salt;
41. }
43. /\*\*
44. \* 转换密钥<br>
45. \*
46. \* @param password
47. \* @return
48. \* @throws Exception
49. \*/
50. **private** **static** Key toKey(String password) **throws** Exception {
51. PBEKeySpec keySpec = **new** PBEKeySpec(password.toCharArray());
52. SecretKeyFactory keyFactory = SecretKeyFactory.getInstance(ALGORITHM);
53. SecretKey secretKey = keyFactory.generateSecret(keySpec);
55. **return** secretKey;
56. }
58. /\*\*
59. \* 加密
60. \*
61. \* @param data
62. \*            数据
63. \* @param password
64. \*            密码
65. \* @param salt
66. \*            盐
67. \* @return
68. \* @throws Exception
69. \*/
70. **public** **static** **byte**[] encrypt(**byte**[] data, String password, **byte**[] salt)
71. **throws** Exception {
73. Key key = toKey(password);
75. PBEParameterSpec paramSpec = **new** PBEParameterSpec(salt, 100);
76. Cipher cipher = Cipher.getInstance(ALGORITHM);
77. cipher.init(Cipher.ENCRYPT\_MODE, key, paramSpec);
79. **return** cipher.doFinal(data);
81. }
83. /\*\*
84. \* 解密
85. \*
86. \* @param data
87. \*            数据
88. \* @param password
89. \*            密码
90. \* @param salt
91. \*            盐
92. \* @return
93. \* @throws Exception
94. \*/
95. **public** **static** **byte**[] decrypt(**byte**[] data, String password, **byte**[] salt)
96. **throws** Exception {
98. Key key = toKey(password);
100. PBEParameterSpec paramSpec = **new** PBEParameterSpec(salt, 100);
101. Cipher cipher = Cipher.getInstance(ALGORITHM);
102. cipher.init(Cipher.DECRYPT\_MODE, key, paramSpec);
104. **return** cipher.doFinal(data);
106. }
107. }

再给出一个测试类：

**Java代码  [收藏代码](javascript:void())**

1. **import** **static** org.junit.Assert.\*;
3. **import** org.junit.Test;
5. /\*\*
6. \*
7. \* @author 梁栋
8. \* @version 1.0
9. \* @since 1.0
10. \*/
11. **public** **class** PBECoderTest {
13. @Test
14. **public** **void** test() **throws** Exception {
15. String inputStr = "abc";
16. System.err.println("原文: " + inputStr);
17. **byte**[] input = inputStr.getBytes();
19. String pwd = "efg";
20. System.err.println("密码: " + pwd);
22. **byte**[] salt = PBECoder.initSalt();
24. **byte**[] data = PBECoder.encrypt(input, pwd, salt);
26. System.err.println("加密后: " + PBECoder.encryptBASE64(data));
28. **byte**[] output = PBECoder.decrypt(data, pwd, salt);
29. String outputStr = **new** String(output);
31. System.err.println("解密后: " + outputStr);
32. assertEquals(inputStr, outputStr);
33. }
35. }

控制台输出：

**Console代码  [收藏代码](javascript:void())**

1. 原文: abc
2. 密码: efg
3. 加密后: iCZ0uRtaAhE=
5. 解密后: abc

    后续我们会介绍非对称加密算法，如RSA、DSA、DH、ECC等。

[**Java加密技术（四）——非对称加密算法RSA**](http://snowolf.iteye.com/blog/381767)

**博客分类：**

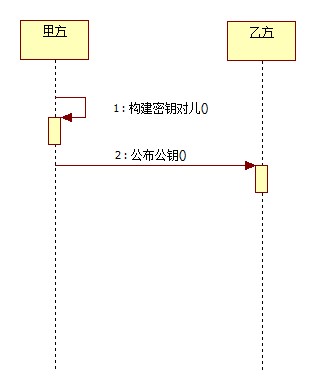
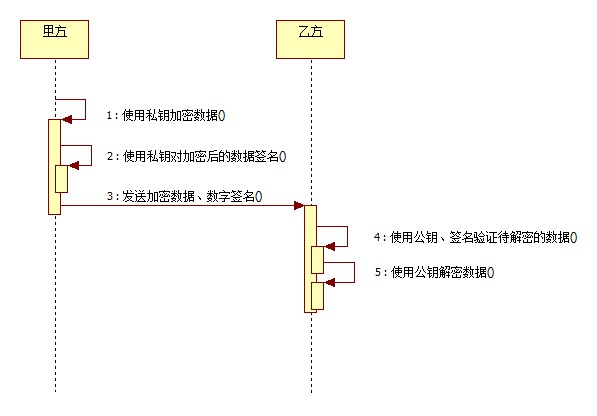
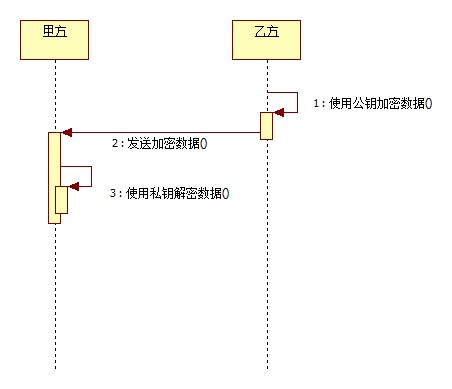
* [Java／Security](http://snowolf.iteye.com/category/68576)

[Java](http://www.iteye.com/blogs/tag/Java)[非对称加密算法](http://www.iteye.com/blogs/tag/%E9%9D%9E%E5%AF%B9%E7%A7%B0%E5%8A%A0%E5%AF%86%E7%AE%97%E6%B3%95)[rsa](http://www.iteye.com/blogs/tag/rsa)

    接下来我们介绍典型的非对称加密算法——RSA   
  
**RSA**   
    这种算法1978年就出现了，它是第一个既能用于数据加密也能用于数字签名的算法。它易于理解和操作，也很流行。算法的名字以发明者的名字命名：Ron Rivest, AdiShamir 和Leonard Adleman。   
    这种加密算法的特点主要是密钥的变化，上文我们看到DES只有一个密钥。相当于只有一把钥匙，如果这把钥匙丢了，数据也就不安全了。RSA同时有两把钥匙，公钥与私钥。同时支持数字签名。数字签名的意义在于，对传输过来的数据进行校验。确保数据在传输工程中不被修改。http://snowolf.iteye.com/images/smiles/icon_biggrin.gif   
  
**流程分析：**

1. 甲方构建密钥对儿，将公钥公布给乙方，将私钥保留。
2. 甲方使用私钥加密数据，然后用私钥对加密后的数据签名，发送给乙方签名以及加密后的数据；乙方使用公钥、签名来验证待解密数据是否有效，如果有效使用公钥对数据解密。
3. 乙方使用公钥加密数据，向甲方发送经过加密后的数据；甲方获得加密数据，通过私钥解密。

按如上步骤给出序列图，如下：

1. 
2. 
3. 

通过java代码实现如下：**Coder类见**[**Java加密技术（一）**](http://snowolf.iteye.com/blog/379860)

**Java代码  [收藏代码](javascript:void())**

1. **import** java.security.Key;
2. **import** java.security.KeyFactory;
3. **import** java.security.KeyPair;
4. **import** java.security.KeyPairGenerator;
5. **import** java.security.PrivateKey;
6. **import** java.security.PublicKey;
7. **import** java.security.Signature;
8. **import** java.security.interfaces.RSAPrivateKey;
9. **import** java.security.interfaces.RSAPublicKey;
10. **import** java.security.spec.PKCS8EncodedKeySpec;
11. **import** java.security.spec.X509EncodedKeySpec;
13. **import** java.util.HashMap;
14. **import** java.util.Map;
16. **import** javax.crypto.Cipher;
18. /\*\*
19. \* RSA安全编码组件
20. \*
21. \* @author 梁栋
22. \* @version 1.0
23. \* @since 1.0
24. \*/
25. **public** **abstract** **class** RSACoder **extends** Coder {
26. **public** **static** **final** String KEY\_ALGORITHM = "RSA";
27. **public** **static** **final** String SIGNATURE\_ALGORITHM = "MD5withRSA";
29. **private** **static** **final** String PUBLIC\_KEY = "RSAPublicKey";
30. **private** **static** **final** String PRIVATE\_KEY = "RSAPrivateKey";
32. /\*\*
33. \* 用私钥对信息生成数字签名
34. \*
35. \* @param data
36. \*            加密数据
37. \* @param privateKey
38. \*            私钥
39. \*
40. \* @return
41. \* @throws Exception
42. \*/
43. **public** **static** String sign(**byte**[] data, String privateKey) **throws** Exception {
44. // 解密由base64编码的私钥
45. **byte**[] keyBytes = decryptBASE64(privateKey);
47. // 构造PKCS8EncodedKeySpec对象
48. PKCS8EncodedKeySpec pkcs8KeySpec = **new** PKCS8EncodedKeySpec(keyBytes);
50. // KEY\_ALGORITHM 指定的加密算法
51. KeyFactory keyFactory = KeyFactory.getInstance(KEY\_ALGORITHM);
53. // 取私钥匙对象
54. PrivateKey priKey = keyFactory.generatePrivate(pkcs8KeySpec);
56. // 用私钥对信息生成数字签名
57. Signature signature = Signature.getInstance(SIGNATURE\_ALGORITHM);
58. signature.initSign(priKey);
59. signature.update(data);
61. **return** encryptBASE64(signature.sign());
62. }
64. /\*\*
65. \* 校验数字签名
66. \*
67. \* @param data
68. \*            加密数据
69. \* @param publicKey
70. \*            公钥
71. \* @param sign
72. \*            数字签名
73. \*
74. \* @return 校验成功返回true 失败返回false
75. \* @throws Exception
76. \*
77. \*/
78. **public** **static** **boolean** verify(**byte**[] data, String publicKey, String sign)
79. **throws** Exception {
81. // 解密由base64编码的公钥
82. **byte**[] keyBytes = decryptBASE64(publicKey);
84. // 构造X509EncodedKeySpec对象
85. X509EncodedKeySpec keySpec = **new** X509EncodedKeySpec(keyBytes);
87. // KEY\_ALGORITHM 指定的加密算法
88. KeyFactory keyFactory = KeyFactory.getInstance(KEY\_ALGORITHM);
90. // 取公钥匙对象
91. PublicKey pubKey = keyFactory.generatePublic(keySpec);
93. Signature signature = Signature.getInstance(SIGNATURE\_ALGORITHM);
94. signature.initVerify(pubKey);
95. signature.update(data);
97. // 验证签名是否正常
98. **return** signature.verify(decryptBASE64(sign));
99. }
101. /\*\*
102. \* 解密<br>
103. \* 用私钥解密
104. \*
105. \* @param data
106. \* @param key
107. \* @return
108. \* @throws Exception
109. \*/
110. **public** **static** **byte**[] decryptByPrivateKey(**byte**[] data, String key)
111. **throws** Exception {
112. // 对密钥解密
113. **byte**[] keyBytes = decryptBASE64(key);
115. // 取得私钥
116. PKCS8EncodedKeySpec pkcs8KeySpec = **new** PKCS8EncodedKeySpec(keyBytes);
117. KeyFactory keyFactory = KeyFactory.getInstance(KEY\_ALGORITHM);
118. Key privateKey = keyFactory.generatePrivate(pkcs8KeySpec);
120. // 对数据解密
121. Cipher cipher = Cipher.getInstance(keyFactory.getAlgorithm());
122. cipher.init(Cipher.DECRYPT\_MODE, privateKey);
124. **return** cipher.doFinal(data);
125. }
127. /\*\*
128. \* 解密<br>
129. \* 用公钥解密
130. \*
131. \* @param data
132. \* @param key
133. \* @return
134. \* @throws Exception
135. \*/
136. **public** **static** **byte**[] decryptByPublicKey(**byte**[] data, String key)
137. **throws** Exception {
138. // 对密钥解密
139. **byte**[] keyBytes = decryptBASE64(key);
141. // 取得公钥
142. X509EncodedKeySpec x509KeySpec = **new** X509EncodedKeySpec(keyBytes);
143. KeyFactory keyFactory = KeyFactory.getInstance(KEY\_ALGORITHM);
144. Key publicKey = keyFactory.generatePublic(x509KeySpec);
146. // 对数据解密
147. Cipher cipher = Cipher.getInstance(keyFactory.getAlgorithm());
148. cipher.init(Cipher.DECRYPT\_MODE, publicKey);
150. **return** cipher.doFinal(data);
151. }
153. /\*\*
154. \* 加密<br>
155. \* 用公钥加密
156. \*
157. \* @param data
158. \* @param key
159. \* @return
160. \* @throws Exception
161. \*/
162. **public** **static** **byte**[] encryptByPublicKey(**byte**[] data, String key)
163. **throws** Exception {
164. // 对公钥解密
165. **byte**[] keyBytes = decryptBASE64(key);
167. // 取得公钥
168. X509EncodedKeySpec x509KeySpec = **new** X509EncodedKeySpec(keyBytes);
169. KeyFactory keyFactory = KeyFactory.getInstance(KEY\_ALGORITHM);
170. Key publicKey = keyFactory.generatePublic(x509KeySpec);
172. // 对数据加密
173. Cipher cipher = Cipher.getInstance(keyFactory.getAlgorithm());
174. cipher.init(Cipher.ENCRYPT\_MODE, publicKey);
176. **return** cipher.doFinal(data);
177. }
179. /\*\*
180. \* 加密<br>
181. \* 用私钥加密
182. \*
183. \* @param data
184. \* @param key
185. \* @return
186. \* @throws Exception
187. \*/
188. **public** **static** **byte**[] encryptByPrivateKey(**byte**[] data, String key)
189. **throws** Exception {
190. // 对密钥解密
191. **byte**[] keyBytes = decryptBASE64(key);
193. // 取得私钥
194. PKCS8EncodedKeySpec pkcs8KeySpec = **new** PKCS8EncodedKeySpec(keyBytes);
195. KeyFactory keyFactory = KeyFactory.getInstance(KEY\_ALGORITHM);
196. Key privateKey = keyFactory.generatePrivate(pkcs8KeySpec);
198. // 对数据加密
199. Cipher cipher = Cipher.getInstance(keyFactory.getAlgorithm());
200. cipher.init(Cipher.ENCRYPT\_MODE, privateKey);
202. **return** cipher.doFinal(data);
203. }
205. /\*\*
206. \* 取得私钥
207. \*
208. \* @param keyMap
209. \* @return
210. \* @throws Exception
211. \*/
212. **public** **static** String getPrivateKey(Map<String, Object> keyMap)
213. **throws** Exception {
214. Key key = (Key) keyMap.get(PRIVATE\_KEY);
216. **return** encryptBASE64(key.getEncoded());
217. }
219. /\*\*
220. \* 取得公钥
221. \*
222. \* @param keyMap
223. \* @return
224. \* @throws Exception
225. \*/
226. **public** **static** String getPublicKey(Map<String, Object> keyMap)
227. **throws** Exception {
228. Key key = (Key) keyMap.get(PUBLIC\_KEY);
230. **return** encryptBASE64(key.getEncoded());
231. }
233. /\*\*
234. \* 初始化密钥
235. \*
236. \* @return
237. \* @throws Exception
238. \*/
239. **public** **static** Map<String, Object> initKey() **throws** Exception {
240. KeyPairGenerator keyPairGen = KeyPairGenerator
241. .getInstance(KEY\_ALGORITHM);
242. keyPairGen.initialize(1024);
244. KeyPair keyPair = keyPairGen.generateKeyPair();
246. // 公钥
247. RSAPublicKey publicKey = (RSAPublicKey) keyPair.getPublic();
249. // 私钥
250. RSAPrivateKey privateKey = (RSAPrivateKey) keyPair.getPrivate();
252. Map<String, Object> keyMap = **new** HashMap<String, Object>(2);
254. keyMap.put(PUBLIC\_KEY, publicKey);
255. keyMap.put(PRIVATE\_KEY, privateKey);
256. **return** keyMap;
257. }
258. }

再给出一个测试类：

**Java代码  [收藏代码](javascript:void())**

1. **import** **static** org.junit.Assert.\*;
3. **import** org.junit.Before;
4. **import** org.junit.Test;
6. **import** java.util.Map;
8. /\*\*
9. \*
10. \* @author 梁栋
11. \* @version 1.0
12. \* @since 1.0
13. \*/
14. **public** **class** RSACoderTest {
15. **private** String publicKey;
16. **private** String privateKey;
18. @Before
19. **public** **void** setUp() **throws** Exception {
20. Map<String, Object> keyMap = RSACoder.initKey();
22. publicKey = RSACoder.getPublicKey(keyMap);
23. privateKey = RSACoder.getPrivateKey(keyMap);
24. System.err.println("公钥: \n\r" + publicKey);
25. System.err.println("私钥： \n\r" + privateKey);
26. }
28. @Test
29. **public** **void** test() **throws** Exception {
30. System.err.println("公钥加密——私钥解密");
31. String inputStr = "abc";
32. **byte**[] data = inputStr.getBytes();
34. **byte**[] encodedData = RSACoder.encryptByPublicKey(data, publicKey);
36. **byte**[] decodedData = RSACoder.decryptByPrivateKey(encodedData,
37. privateKey);
39. String outputStr = **new** String(decodedData);
40. System.err.println("加密前: " + inputStr + "\n\r" + "解密后: " + outputStr);
41. assertEquals(inputStr, outputStr);
43. }
45. @Test
46. **public** **void** testSign() **throws** Exception {
47. System.err.println("私钥加密——公钥解密");
48. String inputStr = "sign";
49. **byte**[] data = inputStr.getBytes();
51. **byte**[] encodedData = RSACoder.encryptByPrivateKey(data, privateKey);
53. **byte**[] decodedData = RSACoder
54. .decryptByPublicKey(encodedData, publicKey);
56. String outputStr = **new** String(decodedData);
57. System.err.println("加密前: " + inputStr + "\n\r" + "解密后: " + outputStr);
58. assertEquals(inputStr, outputStr);
60. System.err.println("私钥签名——公钥验证签名");
61. // 产生签名
62. String sign = RSACoder.sign(encodedData, privateKey);
63. System.err.println("签名:\r" + sign);
65. // 验证签名
66. **boolean** status = RSACoder.verify(encodedData, publicKey, sign);
67. System.err.println("状态:\r" + status);
68. assertTrue(status);
70. }
72. }

控制台输出：

**Console代码  [收藏代码](javascript:void())**

1. 公钥:
3. MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCYU/+I0+z1aBl5X6DUUOHQ7FZpmBSDbKTtx89J
4. EcB64jFCkunELT8qiKly7fzEqD03g8ALlu5XvX+bBqHFy7YPJJP0ekE2X3wjUnh2NxlqpH3/B/xm
5. 1ZdSlCwDIkbijhBVDjA/bu5BObhZqQmDwIxlQInL9oVz+o6FbAZCyHBd7wIDAQAB
7. 私钥：
9. MIICdgIBADANBgkqhkiG9w0BAQEFAASCAmAwggJcAgEAAoGBAJhT/4jT7PVoGXlfoNRQ4dDsVmmY
10. FINspO3Hz0kRwHriMUKS6cQtPyqIqXLt/MSoPTeDwAuW7le9f5sGocXLtg8kk/R6QTZffCNSeHY3
11. GWqkff8H/GbVl1KULAMiRuKOEFUOMD9u7kE5uFmpCYPAjGVAicv2hXP6joVsBkLIcF3vAgMBAAEC
12. gYBvZHWoZHmS2EZQqKqeuGr58eobG9hcZzWQoJ4nq/CarBAjw/VovUHE490uK3S9ht4FW7Yzg3LV
13. /MB06Huifh6qf/X9NQA7SeZRRC8gnCQk6JuDIEVJOud5jU+9tyumJakDKodQ3Jf2zQtNr+5ZdEPl
14. uwWgv9c4kmpjhAdyMuQmYQJBANn6pcgvyYaia52dnu+yBUsGkaFfwXkzFSExIbi0MXTkhEb/ER/D
15. rLytukkUu5S5ecz/KBa8U4xIslZDYQbLz5ECQQCy5dutt7RsxN4+dxCWn0/1FrkWl2G329Ucewm3
16. QU9CKu4D+7Kqdj+Ha3lXP8F0Etaaapi7+EfkRUpukn2ItZV/AkEAlk+I0iphxT1rCB0Q5CjWDY5S
17. Df2B5JmdEG5Y2o0nLXwG2w44OLct/k2uD4cEcuITY5Dvi/4BftMCZwm/dnhEgQJACIktJSnJwxLV
18. o9dchENPtlsCM9C/Sd2EWpqISSUlmfugZbJBwR5pQ5XeMUqKeXZYpP+HEBj1nS+tMH9u2/IGEwJA
19. fL8mZiZXan/oBKrblAbplNcKWGRVD/3y65042PAEeghahlJMiYquV5DzZajuuT0wbJ5xQuZB01+X
20. nfpFpBJ2dw==
22. 公钥加密——私钥解密
23. 加密前: abc
25. 解密后: abc
26. 公钥:
28. MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDdOj40yEB48XqWxmPILmJAc7UecIN7F32etSHF
29. 9rwbuEh3+iTPOGSxhoSQpOED0vOb0ZIMkBXZSgsxLaBSin2RZ09YKWRjtpCA0kDkiD11gj4tzTiM
30. l9qq1kwSK7ZkGAgodEn3yIILVmQDuEImHOXFtulvJ71ka07u3LuwUNdB/wIDAQAB
32. 私钥：
34. MIICdwIBADANBgkqhkiG9w0BAQEFAASCAmEwggJdAgEAAoGBAN06PjTIQHjxepbGY8guYkBztR5w
35. g3sXfZ61IcX2vBu4SHf6JM84ZLGGhJCk4QPS85vRkgyQFdlKCzEtoFKKfZFnT1gpZGO2kIDSQOSI
36. PXWCPi3NOIyX2qrWTBIrtmQYCCh0SffIggtWZAO4QiYc5cW26W8nvWRrTu7cu7BQ10H/AgMBAAEC
37. gYEAz2JWBizjI31bqhP4XiP9PuY5F3vqBW4T+L9cFbQiyumKJc58yzTWUAUGKIIn3enXLG7dNqGr
38. mbJro4JeFIJ3CiVDpXR9+FluIgI4SXm7ioGKF2NOMA9LR5Fu82W+pLfpTN2y2SaLYWEDZyp53BxY
39. j9gUxaxi1MQs+C1ZgDF2xmECQQDy70bQntbRfysP+ppCtd56YRnES1Tyekw0wryS2tr+ivQJl7JF
40. gp5rPAOXpgrq36xHDwUspQ0sJ0vj0O7ywxr1AkEA6SAaLhrJJrYucC0jxwAhUYyaPN+aOsWymaRh
41. 9jA/Wc0wp29SbGTh5CcMuGpXm1g0M+FKW3dGiHgS3rVUKim4owJAbnxgapUzAgiiHxxMeDaavnHW
42. 9C2GrtjsO7qtZOTgYI/1uT8itvZW8lJTF+9OW8/qXE76fXl7ai9dFnl5kzMk2QJBALfHz/vCsArt
43. mkRiwY6zApE4Z6tPl1V33ymSVovvUzHnOdD1SKQdD5t+UV/crb3QVi8ED0t2B0u0ZSPfDT/D7kMC
44. QDpwdj9k2F5aokLHBHUNJPFDAp7a5QMaT64gv/d48ITJ68Co+v5WzLMpzJBYXK6PAtqIhxbuPEc2
45. I2k1Afmrwyw=
47. 私钥加密——公钥解密
48. 加密前: sign
50. 解密后: sign
51. 私钥签名——公钥验证签名
52. 签名:
53. ud1RsIwmSC1pN22I4IXteg1VD2FbiehKUfNxgVSHzvQNIK+d20FCkHCqh9djP3h94iWnIUY0ifU+
54. mbJkhAl/i5krExOE0hknOnPMcEP+lZV1RbJI2zG2YooSp2XDleqrQk5e/QF2Mx0Zxt8Xsg7ucVpn
55. i3wwbYWs9wSzIf0UjlM=
57. 状态:
58. true

    简要总结一下，使用公钥加密、私钥解密，完成了乙方到甲方的一次数据传递，通过私钥加密、公钥解密，同时通过私钥签名、公钥验证签名，完成了一次甲方到乙方的数据传递与验证，两次数据传递完成一整套的数据交互！http://snowolf.iteye.com/images/smiles/icon_biggrin.gif   
  
类似数字签名，数字信封是这样描述的：   
  
**数字信封**   
　　数字信封用加密技术来保证只有特定的收信人才能阅读信的内容。   
流程：   
    信息发送方采用对称密钥来加密信息，然后再用接收方的公钥来加密此对称密钥（这部分称为数字信封），再将它和信息一起发送给接收方；接收方先用相应的私钥打开数字信封，得到对称密钥，然后使用对称密钥再解开信息。

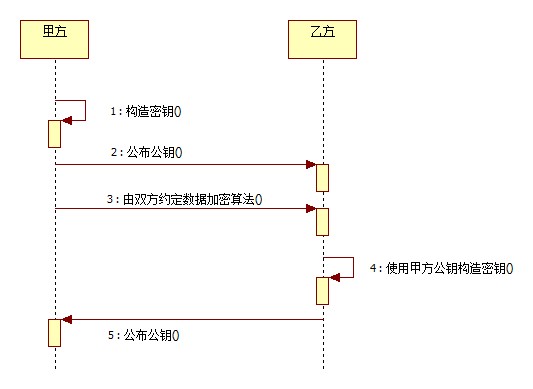
[**Java加密技术（五）——非对称加密算法的由来DH**](http://snowolf.iteye.com/blog/382422)

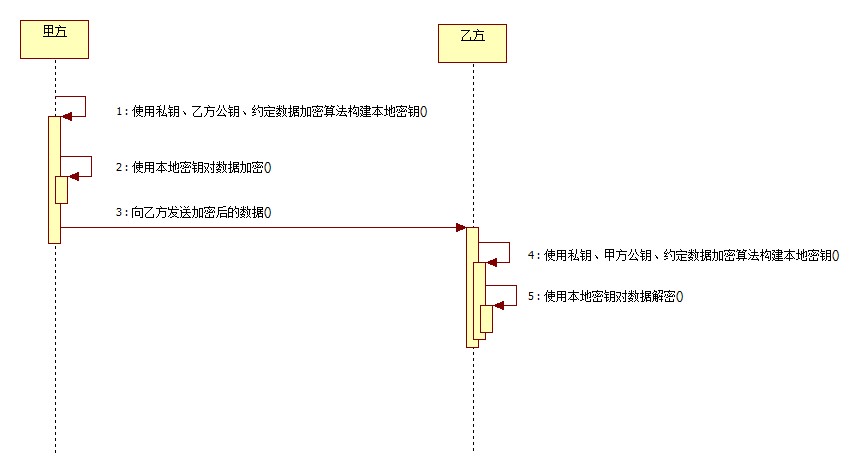
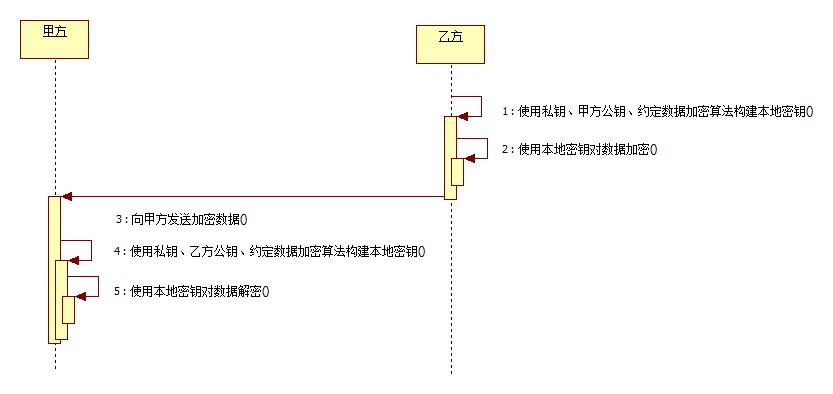
**博客分类：**

* [Java／Security](http://snowolf.iteye.com/category/68576)

[Java](http://www.iteye.com/blogs/tag/Java)[非对称加密算法](http://www.iteye.com/blogs/tag/%E9%9D%9E%E5%AF%B9%E7%A7%B0%E5%8A%A0%E5%AF%86%E7%AE%97%E6%B3%95)[dh](http://www.iteye.com/blogs/tag/dh)

    接下来我们分析DH加密算法，一种适基于密钥一致协议的加密算法。   
**DH**   
Diffie-Hellman算法(D-H算法)，密钥一致协议。是由公开密钥密码体制的奠基人Diffie和Hellman所提出的一种思想。简单的说就是允许两名用户在公开媒体上交换信息以生成"一致"的、可以共享的密钥。换句话说，就是由甲方产出一对密钥（公钥、私钥），乙方依照甲方公钥产生乙方密钥对（公钥、私钥）。以此为基线，作为数据传输保密基础，同时双方使用同一种对称加密算法构建本地密钥（SecretKey）对数据加密。这样，在互通了本地密钥（SecretKey）算法后，甲乙双方公开自己的公钥，使用对方的公钥和刚才产生的私钥加密数据，同时可以使用对方的公钥和自己的私钥对数据解密。不单单是甲乙双方两方，可以扩展为多方共享数据通讯，这样就完成了网络交互数据的安全通讯！该算法源于中国的同余定理——中国馀数定理。http://snowolf.iteye.com/images/smiles/icon_biggrin.gif    
  
流程分析：   
  
1.甲方构建密钥对儿，将公钥公布给乙方，将私钥保留；双方约定数据加密算法；乙方通过甲方公钥构建密钥对儿，将公钥公布给甲方，将私钥保留。   
2.甲方使用私钥、乙方公钥、约定数据加密算法构建本地密钥，然后通过本地密钥加密数据，发送给乙方加密后的数据；乙方使用私钥、甲方公钥、约定数据加密算法构建本地密钥，然后通过本地密钥对数据解密。   
3.乙方使用私钥、甲方公钥、约定数据加密算法构建本地密钥，然后通过本地密钥加密数据，发送给甲方加密后的数据；甲方使用私钥、乙方公钥、约定数据加密算法构建本地密钥，然后通过本地密钥对数据解密。



1. 
2. 

通过java代码实现如下：**Coder类见**[**Java加密技术（一）**](http://snowolf.iteye.com/blog/379860)

**Java代码  [收藏代码](javascript:void())**

1. **import** java.security.Key;
2. **import** java.security.KeyFactory;
3. **import** java.security.KeyPair;
4. **import** java.security.KeyPairGenerator;
5. **import** java.security.PublicKey;
6. **import** java.security.spec.PKCS8EncodedKeySpec;
7. **import** java.security.spec.X509EncodedKeySpec;
8. **import** java.util.HashMap;
9. **import** java.util.Map;
11. **import** javax.crypto.Cipher;
12. **import** javax.crypto.KeyAgreement;
13. **import** javax.crypto.SecretKey;
14. **import** javax.crypto.interfaces.DHPrivateKey;
15. **import** javax.crypto.interfaces.DHPublicKey;
16. **import** javax.crypto.spec.DHParameterSpec;
18. /\*\*
19. \* DH安全编码组件
20. \*
21. \* @author 梁栋
22. \* @version 1.0
23. \* @since 1.0
24. \*/
25. **public** **abstract** **class** DHCoder **extends** Coder {
26. **public** **static** **final** String ALGORITHM = "DH";
28. /\*\*
29. \* 默认密钥字节数
30. \*
31. \* <pre>
32. \* DH
33. \* Default Keysize 1024
34. \* Keysize must be a multiple of 64, ranging from 512 to 1024 (inclusive).
35. \* </pre>
36. \*/
37. **private** **static** **final** **int** KEY\_SIZE = 1024;
39. /\*\*
40. \* DH加密下需要一种对称加密算法对数据加密，这里我们使用DES，也可以使用其他对称加密算法。
41. \*/
42. **public** **static** **final** String SECRET\_ALGORITHM = "DES";
43. **private** **static** **final** String PUBLIC\_KEY = "DHPublicKey";
44. **private** **static** **final** String PRIVATE\_KEY = "DHPrivateKey";
46. /\*\*
47. \* 初始化甲方密钥
48. \*
49. \* @return
50. \* @throws Exception
51. \*/
52. **public** **static** Map<String, Object> initKey() **throws** Exception {
53. KeyPairGenerator keyPairGenerator = KeyPairGenerator
54. .getInstance(ALGORITHM);
55. keyPairGenerator.initialize(KEY\_SIZE);
57. KeyPair keyPair = keyPairGenerator.generateKeyPair();
59. // 甲方公钥
60. DHPublicKey publicKey = (DHPublicKey) keyPair.getPublic();
62. // 甲方私钥
63. DHPrivateKey privateKey = (DHPrivateKey) keyPair.getPrivate();
65. Map<String, Object> keyMap = **new** HashMap<String, Object>(2);
67. keyMap.put(PUBLIC\_KEY, publicKey);
68. keyMap.put(PRIVATE\_KEY, privateKey);
69. **return** keyMap;
70. }
72. /\*\*
73. \* 初始化乙方密钥
74. \*
75. \* @param key
76. \*            甲方公钥
77. \* @return
78. \* @throws Exception
79. \*/
80. **public** **static** Map<String, Object> initKey(String key) **throws** Exception {
81. // 解析甲方公钥
82. **byte**[] keyBytes = decryptBASE64(key);
83. X509EncodedKeySpec x509KeySpec = **new** X509EncodedKeySpec(keyBytes);
84. KeyFactory keyFactory = KeyFactory.getInstance(ALGORITHM);
85. PublicKey pubKey = keyFactory.generatePublic(x509KeySpec);
87. // 由甲方公钥构建乙方密钥
88. DHParameterSpec dhParamSpec = ((DHPublicKey) pubKey).getParams();
90. KeyPairGenerator keyPairGenerator = KeyPairGenerator
91. .getInstance(keyFactory.getAlgorithm());
92. keyPairGenerator.initialize(dhParamSpec);
94. KeyPair keyPair = keyPairGenerator.generateKeyPair();
96. // 乙方公钥
97. DHPublicKey publicKey = (DHPublicKey) keyPair.getPublic();
99. // 乙方私钥
100. DHPrivateKey privateKey = (DHPrivateKey) keyPair.getPrivate();
102. Map<String, Object> keyMap = **new** HashMap<String, Object>(2);
104. keyMap.put(PUBLIC\_KEY, publicKey);
105. keyMap.put(PRIVATE\_KEY, privateKey);
107. **return** keyMap;
108. }
110. /\*\*
111. \* 加密<br>
112. \*
113. \* @param data
114. \*            待加密数据
115. \* @param publicKey
116. \*            甲方公钥
117. \* @param privateKey
118. \*            乙方私钥
119. \* @return
120. \* @throws Exception
121. \*/
122. **public** **static** **byte**[] encrypt(**byte**[] data, String publicKey,
123. String privateKey) **throws** Exception {
125. // 生成本地密钥
126. SecretKey secretKey = getSecretKey(publicKey, privateKey);
128. // 数据加密
129. Cipher cipher = Cipher.getInstance(secretKey.getAlgorithm());
130. cipher.init(Cipher.ENCRYPT\_MODE, secretKey);
132. **return** cipher.doFinal(data);
133. }
135. /\*\*
136. \* 解密<br>
137. \*
138. \* @param data
139. \*            待解密数据
140. \* @param publicKey
141. \*            乙方公钥
142. \* @param privateKey
143. \*            乙方私钥
144. \* @return
145. \* @throws Exception
146. \*/
147. **public** **static** **byte**[] decrypt(**byte**[] data, String publicKey,
148. String privateKey) **throws** Exception {
150. // 生成本地密钥
151. SecretKey secretKey = getSecretKey(publicKey, privateKey);
152. // 数据解密
153. Cipher cipher = Cipher.getInstance(secretKey.getAlgorithm());
154. cipher.init(Cipher.DECRYPT\_MODE, secretKey);
156. **return** cipher.doFinal(data);
157. }
159. /\*\*
160. \* 构建密钥
161. \*
162. \* @param publicKey
163. \*            公钥
164. \* @param privateKey
165. \*            私钥
166. \* @return
167. \* @throws Exception
168. \*/
169. **private** **static** SecretKey getSecretKey(String publicKey, String privateKey)
170. **throws** Exception {
171. // 初始化公钥
172. **byte**[] pubKeyBytes = decryptBASE64(publicKey);
174. KeyFactory keyFactory = KeyFactory.getInstance(ALGORITHM);
175. X509EncodedKeySpec x509KeySpec = **new** X509EncodedKeySpec(pubKeyBytes);
176. PublicKey pubKey = keyFactory.generatePublic(x509KeySpec);
178. // 初始化私钥
179. **byte**[] priKeyBytes = decryptBASE64(privateKey);
181. PKCS8EncodedKeySpec pkcs8KeySpec = **new** PKCS8EncodedKeySpec(priKeyBytes);
182. Key priKey = keyFactory.generatePrivate(pkcs8KeySpec);
184. KeyAgreement keyAgree = KeyAgreement.getInstance(keyFactory
185. .getAlgorithm());
186. keyAgree.init(priKey);
187. keyAgree.doPhase(pubKey, **true**);
189. // 生成本地密钥
190. SecretKey secretKey = keyAgree.generateSecret(SECRET\_ALGORITHM);
192. **return** secretKey;
193. }
195. /\*\*
196. \* 取得私钥
197. \*
198. \* @param keyMap
199. \* @return
200. \* @throws Exception
201. \*/
202. **public** **static** String getPrivateKey(Map<String, Object> keyMap)
203. **throws** Exception {
204. Key key = (Key) keyMap.get(PRIVATE\_KEY);
206. **return** encryptBASE64(key.getEncoded());
207. }
209. /\*\*
210. \* 取得公钥
211. \*
212. \* @param keyMap
213. \* @return
214. \* @throws Exception
215. \*/
216. **public** **static** String getPublicKey(Map<String, Object> keyMap)
217. **throws** Exception {
218. Key key = (Key) keyMap.get(PUBLIC\_KEY);
220. **return** encryptBASE64(key.getEncoded());
221. }
222. }

再给出一个测试类：

**Java代码  [收藏代码](javascript:void())**

1. **import** **static** org.junit.Assert.\*;
3. **import** java.util.Map;
5. **import** org.junit.Test;
7. /\*\*
8. \*
9. \* @author 梁栋
10. \* @version 1.0
11. \* @since 1.0
12. \*/
13. **public** **class** DHCoderTest {
15. @Test
16. **public** **void** test() **throws** Exception {
17. // 生成甲方密钥对儿
18. Map<String, Object> aKeyMap = DHCoder.initKey();
19. String aPublicKey = DHCoder.getPublicKey(aKeyMap);
20. String aPrivateKey = DHCoder.getPrivateKey(aKeyMap);
22. System.err.println("甲方公钥:\r" + aPublicKey);
23. System.err.println("甲方私钥:\r" + aPrivateKey);
25. // 由甲方公钥产生本地密钥对儿
26. Map<String, Object> bKeyMap = DHCoder.initKey(aPublicKey);
27. String bPublicKey = DHCoder.getPublicKey(bKeyMap);
28. String bPrivateKey = DHCoder.getPrivateKey(bKeyMap);
30. System.err.println("乙方公钥:\r" + bPublicKey);
31. System.err.println("乙方私钥:\r" + bPrivateKey);
33. String aInput = "abc ";
34. System.err.println("原文: " + aInput);
36. // 由甲方公钥，乙方私钥构建密文
37. **byte**[] aCode = DHCoder.encrypt(aInput.getBytes(), aPublicKey,
38. bPrivateKey);
40. // 由乙方公钥，甲方私钥解密
41. **byte**[] aDecode = DHCoder.decrypt(aCode, bPublicKey, aPrivateKey);
42. String aOutput = (**new** String(aDecode));
44. System.err.println("解密: " + aOutput);
46. assertEquals(aInput, aOutput);
48. System.err.println(" ===============反过来加密解密================== ");
49. String bInput = "def ";
50. System.err.println("原文: " + bInput);
52. // 由乙方公钥，甲方私钥构建密文
53. **byte**[] bCode = DHCoder.encrypt(bInput.getBytes(), bPublicKey,
54. aPrivateKey);
56. // 由甲方公钥，乙方私钥解密
57. **byte**[] bDecode = DHCoder.decrypt(bCode, aPublicKey, bPrivateKey);
58. String bOutput = (**new** String(bDecode));
60. System.err.println("解密: " + bOutput);
62. assertEquals(bInput, bOutput);
63. }
65. }

控制台输出：

**Console代码  [收藏代码](javascript:void())**

1. 甲方公钥:
2. MIHfMIGXBgkqhkiG9w0BAwEwgYkCQQD8poLOjhLKuibvzPcRDlJtsHiwXt7LzR60ogjzrhYXrgHz
3. W5Gkfm32NBPF4S7QiZvNEyrNUNmRUb3EPuc3WS4XAkBnhHGyepz0TukaScUUfbGpqvJE8FpDTWSG
4. kx0tFCcbnjUDC3H9c9oXkGmzLik1Yw4cIGI1TQ2iCmxBblC+eUykAgIBgANDAAJAdAWBVmIzqcko
5. Ej6qFjLDL2+Y3FPq1iRbnOyOpDj71yKaK1K+FhTv04B0zy4DKcvAASV7/Gv0W+bgqdmffRkqrQ==
7. 甲方私钥:
8. MIHRAgEAMIGXBgkqhkiG9w0BAwEwgYkCQQD8poLOjhLKuibvzPcRDlJtsHiwXt7LzR60ogjzrhYX
9. rgHzW5Gkfm32NBPF4S7QiZvNEyrNUNmRUb3EPuc3WS4XAkBnhHGyepz0TukaScUUfbGpqvJE8FpD
10. TWSGkx0tFCcbnjUDC3H9c9oXkGmzLik1Yw4cIGI1TQ2iCmxBblC+eUykAgIBgAQyAjACJRfy1LyR
11. eHyD+4Hfb+xR0uoIGR1oL9i9Nk6g2AAuaDPgEVWHn+QXID13yL/uDos=
13. 乙方公钥:
14. MIHfMIGXBgkqhkiG9w0BAwEwgYkCQQD8poLOjhLKuibvzPcRDlJtsHiwXt7LzR60ogjzrhYXrgHz
15. W5Gkfm32NBPF4S7QiZvNEyrNUNmRUb3EPuc3WS4XAkBnhHGyepz0TukaScUUfbGpqvJE8FpDTWSG
16. kx0tFCcbnjUDC3H9c9oXkGmzLik1Yw4cIGI1TQ2iCmxBblC+eUykAgIBgANDAAJAVEYSfBA+I9nr
17. dWw3OBv475C+eBrWBBYqt0m6/eu4ptuDQHwV4MmUtKAC2wc2nNrdb1wmBhY1X8RnWkJ1XmdDbQ==
19. 乙方私钥:
20. MIHSAgEAMIGXBgkqhkiG9w0BAwEwgYkCQQD8poLOjhLKuibvzPcRDlJtsHiwXt7LzR60ogjzrhYX
21. rgHzW5Gkfm32NBPF4S7QiZvNEyrNUNmRUb3EPuc3WS4XAkBnhHGyepz0TukaScUUfbGpqvJE8FpD
22. TWSGkx0tFCcbnjUDC3H9c9oXkGmzLik1Yw4cIGI1TQ2iCmxBblC+eUykAgIBgAQzAjEAqaZiCdXp
23. 2iNpdBlHRaO9ir70wo2n32xNlIzIX19VLSPCDdeUWkgRv4CEj/8k+/yd
25. 原文: abc
26. 解密: abc
27. ===============反过来加密解密==================
28. 原文: def
29. 解密: def

如我所言，甲乙双方在获得对方公钥后可以对发送给对方的数据加密，同时也能对接收到的数据解密，达到了数据安全通信的目的！http://snowolf.iteye.com/images/smiles/icon_biggrin.gif

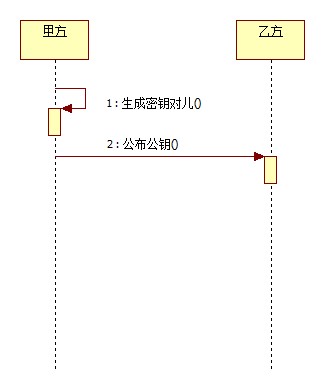
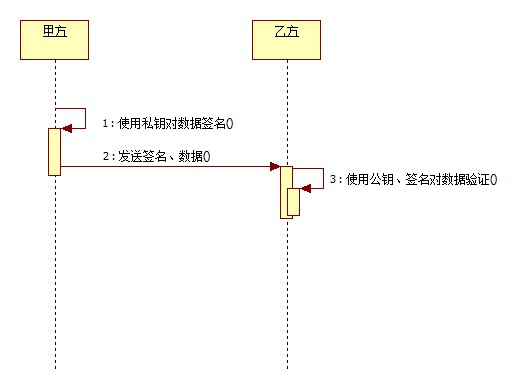
[**Java加密技术（六）——数字签名算法DSA**](http://snowolf.iteye.com/blog/382749)

**博客分类：**

* [Java／Security](http://snowolf.iteye.com/category/68576)

[Java](http://www.iteye.com/blogs/tag/Java)[Security](http://www.iteye.com/blogs/tag/Security)[算法](http://www.iteye.com/blogs/tag/%E7%AE%97%E6%B3%95)[Blog](http://www.iteye.com/blogs/tag/Blog)[junit](http://www.iteye.com/blogs/tag/junit)

    接下来我们介绍DSA数字签名，非对称加密的另一种实现。   
**DSA**   
DSA-Digital Signature Algorithm 是Schnorr和ElGamal签名算法的变种，被美国NIST作为DSS(DigitalSignature Standard)。简单的说，这是一种更高级的验证方式，用作数字签名。不单单只有公钥、私钥，还有数字签名。私钥加密生成数字签名，公钥验证数据及签名。如果数据和签名不匹配则认为验证失败！数字签名的作用就是校验数据在传输过程中不被修改。数字签名，是单向加密的升级！http://snowolf.iteye.com/images/smiles/icon_biggrin.gif

1. 
2. 

通过java代码实现如下：**Coder类见**[**Java加密技术（一）**](http://snowolf.iteye.com/blog/379860)

**Java代码  [收藏代码](javascript:void())**

1. **import** java.security.Key;
2. **import** java.security.KeyFactory;
3. **import** java.security.KeyPair;
4. **import** java.security.KeyPairGenerator;
5. **import** java.security.PrivateKey;
6. **import** java.security.PublicKey;
7. **import** java.security.SecureRandom;
8. **import** java.security.Signature;
9. **import** java.security.interfaces.DSAPrivateKey;
10. **import** java.security.interfaces.DSAPublicKey;
11. **import** java.security.spec.PKCS8EncodedKeySpec;
12. **import** java.security.spec.X509EncodedKeySpec;
13. **import** java.util.HashMap;
14. **import** java.util.Map;
16. /\*\*
17. \* DSA安全编码组件
18. \*
19. \* @author 梁栋
20. \* @version 1.0
21. \* @since 1.0
22. \*/
23. **public** **abstract** **class** DSACoder **extends** Coder {
25. **public** **static** **final** String ALGORITHM = "DSA";
27. /\*\*
28. \* 默认密钥字节数
29. \*
30. \* <pre>
31. \* DSA
32. \* Default Keysize 1024
33. \* Keysize must be a multiple of 64, ranging from 512 to 1024 (inclusive).
34. \* </pre>
35. \*/
36. **private** **static** **final** **int** KEY\_SIZE = 1024;
38. /\*\*
39. \* 默认种子
40. \*/
41. **private** **static** **final** String DEFAULT\_SEED = "0f22507a10bbddd07d8a3082122966e3";
43. **private** **static** **final** String PUBLIC\_KEY = "DSAPublicKey";
44. **private** **static** **final** String PRIVATE\_KEY = "DSAPrivateKey";
46. /\*\*
47. \* 用私钥对信息生成数字签名
48. \*
49. \* @param data
50. \*            加密数据
51. \* @param privateKey
52. \*            私钥
53. \*
54. \* @return
55. \* @throws Exception
56. \*/
57. **public** **static** String sign(**byte**[] data, String privateKey) **throws** Exception {
58. // 解密由base64编码的私钥
59. **byte**[] keyBytes = decryptBASE64(privateKey);
61. // 构造PKCS8EncodedKeySpec对象
62. PKCS8EncodedKeySpec pkcs8KeySpec = **new** PKCS8EncodedKeySpec(keyBytes);
64. // KEY\_ALGORITHM 指定的加密算法
65. KeyFactory keyFactory = KeyFactory.getInstance(ALGORITHM);
67. // 取私钥匙对象
68. PrivateKey priKey = keyFactory.generatePrivate(pkcs8KeySpec);
70. // 用私钥对信息生成数字签名
71. Signature signature = Signature.getInstance(keyFactory.getAlgorithm());
72. signature.initSign(priKey);
73. signature.update(data);
75. **return** encryptBASE64(signature.sign());
76. }
78. /\*\*
79. \* 校验数字签名
80. \*
81. \* @param data
82. \*            加密数据
83. \* @param publicKey
84. \*            公钥
85. \* @param sign
86. \*            数字签名
87. \*
88. \* @return 校验成功返回true 失败返回false
89. \* @throws Exception
90. \*
91. \*/
92. **public** **static** **boolean** verify(**byte**[] data, String publicKey, String sign)
93. **throws** Exception {
95. // 解密由base64编码的公钥
96. **byte**[] keyBytes = decryptBASE64(publicKey);
98. // 构造X509EncodedKeySpec对象
99. X509EncodedKeySpec keySpec = **new** X509EncodedKeySpec(keyBytes);
101. // ALGORITHM 指定的加密算法
102. KeyFactory keyFactory = KeyFactory.getInstance(ALGORITHM);
104. // 取公钥匙对象
105. PublicKey pubKey = keyFactory.generatePublic(keySpec);
107. Signature signature = Signature.getInstance(keyFactory.getAlgorithm());
108. signature.initVerify(pubKey);
109. signature.update(data);
111. // 验证签名是否正常
112. **return** signature.verify(decryptBASE64(sign));
113. }
115. /\*\*
116. \* 生成密钥
117. \*
118. \* @param seed
119. \*            种子
120. \* @return 密钥对象
121. \* @throws Exception
122. \*/
123. **public** **static** Map<String, Object> initKey(String seed) **throws** Exception {
124. KeyPairGenerator keygen = KeyPairGenerator.getInstance(ALGORITHM);
125. // 初始化随机产生器
126. SecureRandom secureRandom = **new** SecureRandom();
127. secureRandom.setSeed(seed.getBytes());
128. keygen.initialize(KEY\_SIZE, secureRandom);
130. KeyPair keys = keygen.genKeyPair();
132. DSAPublicKey publicKey = (DSAPublicKey) keys.getPublic();
133. DSAPrivateKey privateKey = (DSAPrivateKey) keys.getPrivate();
135. Map<String, Object> map = **new** HashMap<String, Object>(2);
136. map.put(PUBLIC\_KEY, publicKey);
137. map.put(PRIVATE\_KEY, privateKey);
139. **return** map;
140. }
142. /\*\*
143. \* 默认生成密钥
144. \*
145. \* @return 密钥对象
146. \* @throws Exception
147. \*/
148. **public** **static** Map<String, Object> initKey() **throws** Exception {
149. **return** initKey(DEFAULT\_SEED);
150. }
152. /\*\*
153. \* 取得私钥
154. \*
155. \* @param keyMap
156. \* @return
157. \* @throws Exception
158. \*/
159. **public** **static** String getPrivateKey(Map<String, Object> keyMap)
160. **throws** Exception {
161. Key key = (Key) keyMap.get(PRIVATE\_KEY);
163. **return** encryptBASE64(key.getEncoded());
164. }
166. /\*\*
167. \* 取得公钥
168. \*
169. \* @param keyMap
170. \* @return
171. \* @throws Exception
172. \*/
173. **public** **static** String getPublicKey(Map<String, Object> keyMap)
174. **throws** Exception {
175. Key key = (Key) keyMap.get(PUBLIC\_KEY);
177. **return** encryptBASE64(key.getEncoded());
178. }
179. }

再给出一个测试类：

**Java代码  [收藏代码](javascript:void())**

1. **import** **static** org.junit.Assert.\*;
3. **import** java.util.Map;
5. **import** org.junit.Test;
7. /\*\*
8. \*
9. \* @author 梁栋
10. \* @version 1.0
11. \* @since 1.0
12. \*/
13. **public** **class** DSACoderTest {
15. @Test
16. **public** **void** test() **throws** Exception {
17. String inputStr = "abc";
18. **byte**[] data = inputStr.getBytes();
20. // 构建密钥
21. Map<String, Object> keyMap = DSACoder.initKey();
23. // 获得密钥
24. String publicKey = DSACoder.getPublicKey(keyMap);
25. String privateKey = DSACoder.getPrivateKey(keyMap);
27. System.err.println("公钥:\r" + publicKey);
28. System.err.println("私钥:\r" + privateKey);
30. // 产生签名
31. String sign = DSACoder.sign(data, privateKey);
32. System.err.println("签名:\r" + sign);
34. // 验证签名
35. **boolean** status = DSACoder.verify(data, publicKey, sign);
36. System.err.println("状态:\r" + status);
37. assertTrue(status);
39. }
41. }

控制台输出：

**Console代码  [收藏代码](javascript:void())**

1. 公钥:
2. MIIBtzCCASwGByqGSM44BAEwggEfAoGBAP1/U4EddRIpUt9KnC7s5Of2EbdSPO9EAMMeP4C2USZp
3. RV1AIlH7WT2NWPq/xfW6MPbLm1Vs14E7gB00b/JmYLdrmVClpJ+f6AR7ECLCT7up1/63xhv4O1fn
4. xqimFQ8E+4P208UewwI1VBNaFpEy9nXzrith1yrv8iIDGZ3RSAHHAhUAl2BQjxUjC8yykrmCouuE
5. C/BYHPUCgYEA9+GghdabPd7LvKtcNrhXuXmUr7v6OuqC+VdMCz0HgmdRWVeOutRZT+ZxBxCBgLRJ
6. FnEj6EwoFhO3zwkyjMim4TwWeotUfI0o4KOuHiuzpnWRbqN/C/ohNWLx+2J6ASQ7zKTxvqhRkImo
7. g9/hWuWfBpKLZl6Ae1UlZAFMO/7PSSoDgYQAAoGAIu4RUlcQLp49PI0MrbssOY+3uySVnp0TULSv
8. 5T4VaHoKzsLHgGTrwOvsGA+V3yCNl2WDu3D84bSLF7liTWgOj+SMOEaPk4VyRTlLXZWGPsf1Mfd9
9. 21XAbMeVyKDSHHVGbMjBScajf3bXooYQMlyoHiOt/WrCo+mv7efstMM0PGo=
11. 私钥:
12. MIIBTAIBADCCASwGByqGSM44BAEwggEfAoGBAP1/U4EddRIpUt9KnC7s5Of2EbdSPO9EAMMeP4C2
13. USZpRV1AIlH7WT2NWPq/xfW6MPbLm1Vs14E7gB00b/JmYLdrmVClpJ+f6AR7ECLCT7up1/63xhv4
14. O1fnxqimFQ8E+4P208UewwI1VBNaFpEy9nXzrith1yrv8iIDGZ3RSAHHAhUAl2BQjxUjC8yykrmC
15. ouuEC/BYHPUCgYEA9+GghdabPd7LvKtcNrhXuXmUr7v6OuqC+VdMCz0HgmdRWVeOutRZT+ZxBxCB
16. gLRJFnEj6EwoFhO3zwkyjMim4TwWeotUfI0o4KOuHiuzpnWRbqN/C/ohNWLx+2J6ASQ7zKTxvqhR
17. kImog9/hWuWfBpKLZl6Ae1UlZAFMO/7PSSoEFwIVAIegLUtmm2oQKQJTOiLugHTSjl/q
19. 签名:
20. MC0CFQCMg0J/uZmF8GuRpr3TNq48w60nDwIUJCyYNah+HtbU6NcQfy8Ac6LeLQs=
22. 状态:
23. true

注意状态为true，就验证成功！http://snowolf.iteye.com/images/smiles/icon_biggrin.gif

[**Java加密技术（七）——非对称加密算法最高级ECC**](http://snowolf.iteye.com/blog/383412)

**博客分类：**

* [Java／Security](http://snowolf.iteye.com/category/68576)

[Java](http://www.iteye.com/blogs/tag/Java)[非对称加密算法](http://www.iteye.com/blogs/tag/%E9%9D%9E%E5%AF%B9%E7%A7%B0%E5%8A%A0%E5%AF%86%E7%AE%97%E6%B3%95)[ECC](http://www.iteye.com/blogs/tag/ECC)

**ECC**   
ECC-Elliptic Curves Cryptography，椭圆曲线密码编码学，是目前已知的公钥体制中，对每比特所提供加密强度最高的一种体制。在软件注册保护方面起到很大的作用，一般的序列号通常由该算法产生。   
    当我开始整理《Java加密技术（二）》的时候，我就已经在开始研究ECC了，但是关于Java实现ECC算法的资料实在是太少了，无论是国内还是国外的资料，无论是官方还是非官方的解释，最终只有一种答案——ECC算法在jdk1.5后加入支持，目前仅仅只能完成密钥的生成与解析。http://snowolf.iteye.com/images/smiles/icon_sad.gif 如果想要获得ECC算法实现，需要调用硬件完成加密/解密（ECC算法相当耗费资源，如果单纯使用CPU进行加密/解密，效率低下），涉及到Java Card领域，PKCS#11。http://snowolf.iteye.com/images/smiles/icon_smile.gif 其实，PKCS#11配置很简单，但缺乏硬件设备，无法尝试！http://snowolf.iteye.com/images/smiles/icon_sad.gif   
  
    尽管如此，我照旧提供相应的Java实现代码，以供大家参考。http://snowolf.iteye.com/images/smiles/icon_biggrin.gif   
  
通过java代码实现如下：**Coder类见**[**Java加密技术（一）**](http://snowolf.iteye.com/blog/379860)

**Java代码  [收藏代码](javascript:void())**

1. **import** java.math.BigInteger;
2. **import** java.security.Key;
3. **import** java.security.KeyFactory;
4. **import** java.security.interfaces.ECPrivateKey;
5. **import** java.security.interfaces.ECPublicKey;
6. **import** java.security.spec.ECFieldF2m;
7. **import** java.security.spec.ECParameterSpec;
8. **import** java.security.spec.ECPoint;
9. **import** java.security.spec.ECPrivateKeySpec;
10. **import** java.security.spec.ECPublicKeySpec;
11. **import** java.security.spec.EllipticCurve;
12. **import** java.security.spec.PKCS8EncodedKeySpec;
13. **import** java.security.spec.X509EncodedKeySpec;
14. **import** java.util.HashMap;
15. **import** java.util.Map;
17. **import** javax.crypto.Cipher;
18. **import** javax.crypto.NullCipher;
20. **import** sun.security.ec.ECKeyFactory;
21. **import** sun.security.ec.ECPrivateKeyImpl;
22. **import** sun.security.ec.ECPublicKeyImpl;
24. /\*\*
25. \* ECC安全编码组件
26. \*
27. \* @author 梁栋
28. \* @version 1.0
29. \* @since 1.0
30. \*/
31. **public** **abstract** **class** ECCCoder **extends** Coder {
33. **public** **static** **final** String ALGORITHM = "EC";
34. **private** **static** **final** String PUBLIC\_KEY = "ECCPublicKey";
35. **private** **static** **final** String PRIVATE\_KEY = "ECCPrivateKey";
37. /\*\*
38. \* 解密<br>
39. \* 用私钥解密
40. \*
41. \* @param data
42. \* @param key
43. \* @return
44. \* @throws Exception
45. \*/
46. **public** **static** **byte**[] decrypt(**byte**[] data, String key) **throws** Exception {
47. // 对密钥解密
48. **byte**[] keyBytes = decryptBASE64(key);
50. // 取得私钥
51. PKCS8EncodedKeySpec pkcs8KeySpec = **new** PKCS8EncodedKeySpec(keyBytes);
52. KeyFactory keyFactory = ECKeyFactory.INSTANCE;
54. ECPrivateKey priKey = (ECPrivateKey) keyFactory
55. .generatePrivate(pkcs8KeySpec);
57. ECPrivateKeySpec ecPrivateKeySpec = **new** ECPrivateKeySpec(priKey.getS(),
58. priKey.getParams());
60. // 对数据解密
61. // TODO Chipher不支持EC算法 未能实现
62. Cipher cipher = **new** NullCipher();
63. // Cipher.getInstance(ALGORITHM, keyFactory.getProvider());
64. cipher.init(Cipher.DECRYPT\_MODE, priKey, ecPrivateKeySpec.getParams());
66. **return** cipher.doFinal(data);
67. }
69. /\*\*
70. \* 加密<br>
71. \* 用公钥加密
72. \*
73. \* @param data
74. \* @param privateKey
75. \* @return
76. \* @throws Exception
77. \*/
78. **public** **static** **byte**[] encrypt(**byte**[] data, String privateKey)
79. **throws** Exception {
80. // 对公钥解密
81. **byte**[] keyBytes = decryptBASE64(privateKey);
83. // 取得公钥
84. X509EncodedKeySpec x509KeySpec = **new** X509EncodedKeySpec(keyBytes);
85. KeyFactory keyFactory = ECKeyFactory.INSTANCE;
87. ECPublicKey pubKey = (ECPublicKey) keyFactory
88. .generatePublic(x509KeySpec);
90. ECPublicKeySpec ecPublicKeySpec = **new** ECPublicKeySpec(pubKey.getW(),
91. pubKey.getParams());
93. // 对数据加密
94. // TODO Chipher不支持EC算法 未能实现
95. Cipher cipher = **new** NullCipher();
96. // Cipher.getInstance(ALGORITHM, keyFactory.getProvider());
97. cipher.init(Cipher.ENCRYPT\_MODE, pubKey, ecPublicKeySpec.getParams());
99. **return** cipher.doFinal(data);
100. }
102. /\*\*
103. \* 取得私钥
104. \*
105. \* @param keyMap
106. \* @return
107. \* @throws Exception
108. \*/
109. **public** **static** String getPrivateKey(Map<String, Object> keyMap)
110. **throws** Exception {
111. Key key = (Key) keyMap.get(PRIVATE\_KEY);
113. **return** encryptBASE64(key.getEncoded());
114. }
116. /\*\*
117. \* 取得公钥
118. \*
119. \* @param keyMap
120. \* @return
121. \* @throws Exception
122. \*/
123. **public** **static** String getPublicKey(Map<String, Object> keyMap)
124. **throws** Exception {
125. Key key = (Key) keyMap.get(PUBLIC\_KEY);
127. **return** encryptBASE64(key.getEncoded());
128. }
130. /\*\*
131. \* 初始化密钥
132. \*
133. \* @return
134. \* @throws Exception
135. \*/
136. **public** **static** Map<String, Object> initKey() **throws** Exception {
137. BigInteger x1 = **new** BigInteger(
138. "2fe13c0537bbc11acaa07d793de4e6d5e5c94eee8", 16);
139. BigInteger x2 = **new** BigInteger(
140. "289070fb05d38ff58321f2e800536d538ccdaa3d9", 16);
142. ECPoint g = **new** ECPoint(x1, x2);
144. // the order of generator
145. BigInteger n = **new** BigInteger(
146. "5846006549323611672814741753598448348329118574063", 10);
147. // the cofactor
148. **int** h = 2;
149. **int** m = 163;
150. **int**[] ks = { 7, 6, 3 };
151. ECFieldF2m ecField = **new** ECFieldF2m(m, ks);
152. // y^2+xy=x^3+x^2+1
153. BigInteger a = **new** BigInteger("1", 2);
154. BigInteger b = **new** BigInteger("1", 2);
156. EllipticCurve ellipticCurve = **new** EllipticCurve(ecField, a, b);
158. ECParameterSpec ecParameterSpec = **new** ECParameterSpec(ellipticCurve, g,
159. n, h);
160. // 公钥
161. ECPublicKey publicKey = **new** ECPublicKeyImpl(g, ecParameterSpec);
163. BigInteger s = **new** BigInteger(
164. "1234006549323611672814741753598448348329118574063", 10);
165. // 私钥
166. ECPrivateKey privateKey = **new** ECPrivateKeyImpl(s, ecParameterSpec);
168. Map<String, Object> keyMap = **new** HashMap<String, Object>(2);
170. keyMap.put(PUBLIC\_KEY, publicKey);
171. keyMap.put(PRIVATE\_KEY, privateKey);
173. **return** keyMap;
174. }
176. }

    请注意上述代码中的**TODO**内容，再次提醒注意，Chipher不支持EC算法 ，以上代码仅供参考。Chipher、Signature、KeyPairGenerator、KeyAgreement、SecretKey均不支持EC算法。为了确保程序能够正常执行，我们使用了NullCipher类，验证程序。   
  
照旧提供一个测试类：

**Java代码  [收藏代码](javascript:void())**

1. **import** **static** org.junit.Assert.\*;
3. **import** java.math.BigInteger;
4. **import** java.security.spec.ECFieldF2m;
5. **import** java.security.spec.ECParameterSpec;
6. **import** java.security.spec.ECPoint;
7. **import** java.security.spec.ECPrivateKeySpec;
8. **import** java.security.spec.ECPublicKeySpec;
9. **import** java.security.spec.EllipticCurve;
10. **import** java.util.Map;
12. **import** org.junit.Test;
14. /\*\*
15. \*
16. \* @author 梁栋
17. \* @version 1.0
18. \* @since 1.0
19. \*/
20. **public** **class** ECCCoderTest {
22. @Test
23. **public** **void** test() **throws** Exception {
24. String inputStr = "abc";
25. **byte**[] data = inputStr.getBytes();
27. Map<String, Object> keyMap = ECCCoder.initKey();
29. String publicKey = ECCCoder.getPublicKey(keyMap);
30. String privateKey = ECCCoder.getPrivateKey(keyMap);
31. System.err.println("公钥: \n" + publicKey);
32. System.err.println("私钥： \n" + privateKey);
34. **byte**[] encodedData = ECCCoder.encrypt(data, publicKey);
36. **byte**[] decodedData = ECCCoder.decrypt(encodedData, privateKey);
38. String outputStr = **new** String(decodedData);
39. System.err.println("加密前: " + inputStr + "\n\r" + "解密后: " + outputStr);
40. assertEquals(inputStr, outputStr);
41. }
42. }

控制台输出：

**Console代码  [收藏代码](javascript:void())**

1. 公钥:
2. MEAwEAYHKoZIzj0CAQYFK4EEAAEDLAAEAv4TwFN7vBGsqgfXk95ObV5clO7oAokHD7BdOP9YMh8u
3. gAU21TjM2qPZ
5. 私钥：
6. MDICAQAwEAYHKoZIzj0CAQYFK4EEAAEEGzAZAgEBBBTYJsR3BN7TFw7JHcAHFkwNmfil7w==
8. 加密前: abc
10. 解密后: abc

[**Java加密技术（八）——数字证书**](http://snowolf.iteye.com/blog/391931)

**博客分类：**

* [Java／Security](http://snowolf.iteye.com/category/68576)

[Java](http://www.iteye.com/blogs/tag/Java)[keystore](http://www.iteye.com/blogs/tag/keystore)[keytool](http://www.iteye.com/blogs/tag/keytool)[数字证书](http://www.iteye.com/blogs/tag/%E6%95%B0%E5%AD%97%E8%AF%81%E4%B9%A6)

    本篇的主要内容为Java证书体系的实现。http://snowolf.iteye.com/images/smiles/icon_biggrin.gif   
  
**请大家在阅读本篇内容时先阅读**[**Java加密技术（四）**](http://snowolf.iteye.com/blog/381767)**，预先了解RSA加密算法。**http://snowolf.iteye.com/images/smiles/icon_biggrin.gif   
  
在构建Java代码实现前，我们需要完成证书的制作。   
1.生成keyStroe文件   
在命令行下执行以下命令：

**Shell代码  [收藏代码](javascript:void())**

1. keytool -genkey -validity 36000 -alias www.zlex.org -keyalg RSA -keystore d:\zlex.keystore

其中   
**-genkey**表示生成密钥   
**-validity**指定证书有效期，这里是**36000**天   
**-alias**指定别名，这里是**www.zlex.org**   
**-keyalg**指定算法，这里是**RSA**   
**-keystore**指定存储位置，这里是**d:\zlex.keystore**   
  
在这里我使用的密码为 **123456**   
  
控制台输出：

**Console代码  [收藏代码](javascript:void())**

1. 输入keystore密码：
2. 再次输入新密码:
3. 您的名字与姓氏是什么？
4. [Unknown]：  www.zlex.org
5. 您的组织单位名称是什么？
6. [Unknown]：  zlex
7. 您的组织名称是什么？
8. [Unknown]：  zlex
9. 您所在的城市或区域名称是什么？
10. [Unknown]：  BJ
11. 您所在的州或省份名称是什么？
12. [Unknown]：  BJ
13. 该单位的两字母国家代码是什么
14. [Unknown]：  CN
15. CN=www.zlex.org, OU=zlex, O=zlex, L=BJ, ST=BJ, C=CN 正确吗？
16. [否]：  Y
18. 输入<tomcat>的主密码
19. （如果和 keystore 密码相同，按回车）：
20. 再次输入新密码:

这时，在D盘下会生成一个zlex.keystore的文件。   
  
2.生成自签名证书   
光有keyStore文件是不够的，还需要证书文件，证书才是直接提供给外界使用的公钥凭证。   
导出证书：

**Shell代码  [收藏代码](javascript:void())**

1. keytool -export -keystore d:\zlex.keystore -alias www.zlex.org -file d:\zlex.cer -rfc

其中   
**-export**指定为导出操作   
**-keystore**指定**keystore文件**   
**-alias**指定导出**keystore文件中的别名**   
**-file**指向**导出路径**   
**-rfc**以文本格式输出，也就是以**BASE64编码**输出   
这里的密码是 **123456**   
  
控制台输出：

**Console代码  [收藏代码](javascript:void())**

1. 输入keystore密码：
2. 保存在文件中的认证 <d:\zlex.cer>

当然，使用方是需要导入证书的！   
可以通过自签名证书完成CAS单点登录系统的构建！http://snowolf.iteye.com/images/smiles/icon_biggrin.gif   
  
Ok，准备工作完成，开始Java实现！   
  
通过java代码实现如下：**Coder类见**[**Java加密技术（一）**](http://snowolf.iteye.com/blog/379860)

**Java代码  [收藏代码](javascript:void())**

1. **import** java.io.FileInputStream;
2. **import** java.security.KeyStore;
3. **import** java.security.PrivateKey;
4. **import** java.security.PublicKey;
5. **import** java.security.Signature;
6. **import** java.security.cert.Certificate;
7. **import** java.security.cert.CertificateFactory;
8. **import** java.security.cert.X509Certificate;
9. **import** java.util.Date;
11. **import** javax.crypto.Cipher;
13. /\*\*
14. \* 证书组件
15. \*
16. \* @author 梁栋
17. \* @version 1.0
18. \* @since 1.0
19. \*/
20. **public** **abstract** **class** CertificateCoder **extends** Coder {

23. /\*\*
24. \* Java密钥库(Java Key Store，JKS)KEY\_STORE
25. \*/
26. **public** **static** **final** String KEY\_STORE = "JKS";
28. **public** **static** **final** String X509 = "X.509";
30. /\*\*
31. \* 由KeyStore获得私钥
32. \*
33. \* @param keyStorePath
34. \* @param alias
35. \* @param password
36. \* @return
37. \* @throws Exception
38. \*/
39. **private** **static** PrivateKey getPrivateKey(String keyStorePath, String alias,
40. String password) **throws** Exception {
41. KeyStore ks = getKeyStore(keyStorePath, password);
42. PrivateKey key = (PrivateKey) ks.getKey(alias, password.toCharArray());
43. **return** key;
44. }
46. /\*\*
47. \* 由Certificate获得公钥
48. \*
49. \* @param certificatePath
50. \* @return
51. \* @throws Exception
52. \*/
53. **private** **static** PublicKey getPublicKey(String certificatePath)
54. **throws** Exception {
55. Certificate certificate = getCertificate(certificatePath);
56. PublicKey key = certificate.getPublicKey();
57. **return** key;
58. }
60. /\*\*
61. \* 获得Certificate
62. \*
63. \* @param certificatePath
64. \* @return
65. \* @throws Exception
66. \*/
67. **private** **static** Certificate getCertificate(String certificatePath)
68. **throws** Exception {
69. CertificateFactory certificateFactory = CertificateFactory
70. .getInstance(X509);
71. FileInputStream in = **new** FileInputStream(certificatePath);
73. Certificate certificate = certificateFactory.generateCertificate(in);
74. in.close();
76. **return** certificate;
77. }
79. /\*\*
80. \* 获得Certificate
81. \*
82. \* @param keyStorePath
83. \* @param alias
84. \* @param password
85. \* @return
86. \* @throws Exception
87. \*/
88. **private** **static** Certificate getCertificate(String keyStorePath,
89. String alias, String password) **throws** Exception {
90. KeyStore ks = getKeyStore(keyStorePath, password);
91. Certificate certificate = ks.getCertificate(alias);
93. **return** certificate;
94. }
96. /\*\*
97. \* 获得KeyStore
98. \*
99. \* @param keyStorePath
100. \* @param password
101. \* @return
102. \* @throws Exception
103. \*/
104. **private** **static** KeyStore getKeyStore(String keyStorePath, String password)
105. **throws** Exception {
106. FileInputStream is = **new** FileInputStream(keyStorePath);
107. KeyStore ks = KeyStore.getInstance(KEY\_STORE);
108. ks.load(is, password.toCharArray());
109. is.close();
110. **return** ks;
111. }
113. /\*\*
114. \* 私钥加密
115. \*
116. \* @param data
117. \* @param keyStorePath
118. \* @param alias
119. \* @param password
120. \* @return
121. \* @throws Exception
122. \*/
123. **public** **static** **byte**[] encryptByPrivateKey(**byte**[] data, String keyStorePath,
124. String alias, String password) **throws** Exception {
125. // 取得私钥
126. PrivateKey privateKey = getPrivateKey(keyStorePath, alias, password);
128. // 对数据加密
129. Cipher cipher = Cipher.getInstance(privateKey.getAlgorithm());
130. cipher.init(Cipher.ENCRYPT\_MODE, privateKey);
132. **return** cipher.doFinal(data);
134. }
136. /\*\*
137. \* 私钥解密
138. \*
139. \* @param data
140. \* @param keyStorePath
141. \* @param alias
142. \* @param password
143. \* @return
144. \* @throws Exception
145. \*/
146. **public** **static** **byte**[] decryptByPrivateKey(**byte**[] data, String keyStorePath,
147. String alias, String password) **throws** Exception {
148. // 取得私钥
149. PrivateKey privateKey = getPrivateKey(keyStorePath, alias, password);
151. // 对数据加密
152. Cipher cipher = Cipher.getInstance(privateKey.getAlgorithm());
153. cipher.init(Cipher.DECRYPT\_MODE, privateKey);
155. **return** cipher.doFinal(data);
157. }
159. /\*\*
160. \* 公钥加密
161. \*
162. \* @param data
163. \* @param certificatePath
164. \* @return
165. \* @throws Exception
166. \*/
167. **public** **static** **byte**[] encryptByPublicKey(**byte**[] data, String certificatePath)
168. **throws** Exception {
170. // 取得公钥
171. PublicKey publicKey = getPublicKey(certificatePath);
172. // 对数据加密
173. Cipher cipher = Cipher.getInstance(publicKey.getAlgorithm());
174. cipher.init(Cipher.ENCRYPT\_MODE, publicKey);
176. **return** cipher.doFinal(data);
178. }
180. /\*\*
181. \* 公钥解密
182. \*
183. \* @param data
184. \* @param certificatePath
185. \* @return
186. \* @throws Exception
187. \*/
188. **public** **static** **byte**[] decryptByPublicKey(**byte**[] data, String certificatePath)
189. **throws** Exception {
190. // 取得公钥
191. PublicKey publicKey = getPublicKey(certificatePath);
193. // 对数据加密
194. Cipher cipher = Cipher.getInstance(publicKey.getAlgorithm());
195. cipher.init(Cipher.DECRYPT\_MODE, publicKey);
197. **return** cipher.doFinal(data);
199. }
201. /\*\*
202. \* 验证Certificate
203. \*
204. \* @param certificatePath
205. \* @return
206. \*/
207. **public** **static** **boolean** verifyCertificate(String certificatePath) {
208. **return** verifyCertificate(**new** Date(), certificatePath);
209. }
211. /\*\*
212. \* 验证Certificate是否过期或无效
213. \*
214. \* @param date
215. \* @param certificatePath
216. \* @return
217. \*/
218. **public** **static** **boolean** verifyCertificate(Date date, String certificatePath) {
219. **boolean** status = **true**;
220. **try** {
221. // 取得证书
222. Certificate certificate = getCertificate(certificatePath);
223. // 验证证书是否过期或无效
224. status = verifyCertificate(date, certificate);
225. } **catch** (Exception e) {
226. status = **false**;
227. }
228. **return** status;
229. }
231. /\*\*
232. \* 验证证书是否过期或无效
233. \*
234. \* @param date
235. \* @param certificate
236. \* @return
237. \*/
238. **private** **static** **boolean** verifyCertificate(Date date, Certificate certificate) {
239. **boolean** status = **true**;
240. **try** {
241. X509Certificate x509Certificate = (X509Certificate) certificate;
242. x509Certificate.checkValidity(date);
243. } **catch** (Exception e) {
244. status = **false**;
245. }
246. **return** status;
247. }
249. /\*\*
250. \* 签名
251. \*
252. \* @param keyStorePath
253. \* @param alias
254. \* @param password
255. \*
256. \* @return
257. \* @throws Exception
258. \*/
259. **public** **static** String sign(**byte**[] sign, String keyStorePath, String alias,
260. String password) **throws** Exception {
261. // 获得证书
262. X509Certificate x509Certificate = (X509Certificate) getCertificate(
263. keyStorePath, alias, password);
264. // 获取私钥
265. KeyStore ks = getKeyStore(keyStorePath, password);
266. // 取得私钥
267. PrivateKey privateKey = (PrivateKey) ks.getKey(alias, password
268. .toCharArray());
270. // 构建签名
271. Signature signature = Signature.getInstance(x509Certificate
272. .getSigAlgName());
273. signature.initSign(privateKey);
274. signature.update(sign);
275. **return** encryptBASE64(signature.sign());
276. }
278. /\*\*
279. \* 验证签名
280. \*
281. \* @param data
282. \* @param sign
283. \* @param certificatePath
284. \* @return
285. \* @throws Exception
286. \*/
287. **public** **static** **boolean** verify(**byte**[] data, String sign,
288. String certificatePath) **throws** Exception {
289. // 获得证书
290. X509Certificate x509Certificate = (X509Certificate) getCertificate(certificatePath);
291. // 获得公钥
292. PublicKey publicKey = x509Certificate.getPublicKey();
293. // 构建签名
294. Signature signature = Signature.getInstance(x509Certificate
295. .getSigAlgName());
296. signature.initVerify(publicKey);
297. signature.update(data);
299. **return** signature.verify(decryptBASE64(sign));
301. }
303. /\*\*
304. \* 验证Certificate
305. \*
306. \* @param keyStorePath
307. \* @param alias
308. \* @param password
309. \* @return
310. \*/
311. **public** **static** **boolean** verifyCertificate(Date date, String keyStorePath,
312. String alias, String password) {
313. **boolean** status = **true**;
314. **try** {
315. Certificate certificate = getCertificate(keyStorePath, alias,
316. password);
317. status = verifyCertificate(date, certificate);
318. } **catch** (Exception e) {
319. status = **false**;
320. }
321. **return** status;
322. }
324. /\*\*
325. \* 验证Certificate
326. \*
327. \* @param keyStorePath
328. \* @param alias
329. \* @param password
330. \* @return
331. \*/
332. **public** **static** **boolean** verifyCertificate(String keyStorePath, String alias,
333. String password) {
334. **return** verifyCertificate(**new** Date(), keyStorePath, alias, password);
335. }
336. }

再给出一个测试类：

**Java代码  [收藏代码](javascript:void())**

1. **import** **static** org.junit.Assert.\*;
3. **import** org.junit.Test;
5. /\*\*
6. \*
7. \* @author 梁栋
8. \* @version 1.0
9. \* @since 1.0
10. \*/
11. **public** **class** CertificateCoderTest {
12. **private** String password = "123456";
13. **private** String alias = "www.zlex.org";
14. **private** String certificatePath = "d:/zlex.cer";
15. **private** String keyStorePath = "d:/zlex.keystore";
17. @Test
18. **public** **void** test() **throws** Exception {
19. System.err.println("公钥加密——私钥解密");
20. String inputStr = "Ceritifcate";
21. **byte**[] data = inputStr.getBytes();
23. **byte**[] encrypt = CertificateCoder.encryptByPublicKey(data,
24. certificatePath);
26. **byte**[] decrypt = CertificateCoder.decryptByPrivateKey(encrypt,
27. keyStorePath, alias, password);
28. String outputStr = **new** String(decrypt);
30. System.err.println("加密前: " + inputStr + "\n\r" + "解密后: " + outputStr);
32. // 验证数据一致
33. assertArrayEquals(data, decrypt);
35. // 验证证书有效
36. assertTrue(CertificateCoder.verifyCertificate(certificatePath));
38. }
40. @Test
41. **public** **void** testSign() **throws** Exception {
42. System.err.println("私钥加密——公钥解密");
44. String inputStr = "sign";
45. **byte**[] data = inputStr.getBytes();
47. **byte**[] encodedData = CertificateCoder.encryptByPrivateKey(data,
48. keyStorePath, alias, password);
50. **byte**[] decodedData = CertificateCoder.decryptByPublicKey(encodedData,
51. certificatePath);
53. String outputStr = **new** String(decodedData);
54. System.err.println("加密前: " + inputStr + "\n\r" + "解密后: " + outputStr);
55. assertEquals(inputStr, outputStr);
57. System.err.println("私钥签名——公钥验证签名");
58. // 产生签名
59. String sign = CertificateCoder.sign(encodedData, keyStorePath, alias,
60. password);
61. System.err.println("签名:\r" + sign);
63. // 验证签名
64. **boolean** status = CertificateCoder.verify(encodedData, sign,
65. certificatePath);
66. System.err.println("状态:\r" + status);
67. assertTrue(status);
69. }
70. }

控制台输出：

**Console代码  [收藏代码](javascript:void())**

1. 公钥加密——私钥解密
2. 加密前: Ceritificate
4. 解密后: Ceritificate
6. 私钥加密——公钥解密
7. 加密前: sign
9. 解密后: sign
10. 私钥签名——公钥验证签名
11. 签名:
12. pqBn5m6PJlfOjH0A6U2o2mUmBsfgyEY1NWCbiyA/I5Gc3gaVNVIdj/zkGNZRqTjhf3+J9a9z9EI7
13. 6F2eWYd7punHx5oh6hfNgcKbVb52EfItl4QEN+djbXiPynn07+Lbg1NOjULnpEd6ZhLP1YwrEAuM
14. OfvX0e7/wplxLbySaKQ=
16. 状态:
17. true

由此完成了证书验证体系！http://snowolf.iteye.com/images/smiles/icon_biggrin.gif   
  
同样，我们可以对代码做签名——代码签名！http://snowolf.iteye.com/images/smiles/icon_biggrin.gif   
通过工具JarSigner可以完成代码签名。   
这里我们对tools.jar做代码签名，命令如下：

**Shell代码  [收藏代码](javascript:void())**

1. jarsigner -storetype jks -keystore zlex.keystore -verbose tools.jar www.zlex.org

控制台输出：

**Console代码  [收藏代码](javascript:void())**

1. 输入密钥库的口令短语：
2. 正在更新： META-INF/WWW\_ZLEX.SF
3. 正在更新： META-INF/WWW\_ZLEX.RSA
4. 正在签名： org/zlex/security/Security.class
5. 正在签名： org/zlex/tool/Main$1.class
6. 正在签名： org/zlex/tool/Main$2.class
7. 正在签名： org/zlex/tool/Main.class
9. 警告：
10. 签名者证书将在六个月内过期。

此时，我们可以对签名后的jar做验证！http://snowolf.iteye.com/images/smiles/icon_biggrin.gif   
验证tools.jar，命令如下：

**Shell代码  [收藏代码](javascript:void())**

1. jarsigner -verify -verbose -certs tools.jar

控制台输出：

**Console代码  [收藏代码](javascript:void())**

1. 402 Sat Jun 20 16:25:14 CST 2009 META-INF/MANIFEST.MF
2. 532 Sat Jun 20 16:25:14 CST 2009 META-INF/WWW\_ZLEX.SF
3. 889 Sat Jun 20 16:25:14 CST 2009 META-INF/WWW\_ZLEX.RSA
4. sm       590 Wed Dec 10 13:03:42 CST 2008 org/zlex/security/Security.class
6. X.509, CN=www.zlex.org, OU=zlex, O=zlex, L=BJ, ST=BJ, C=CN
7. [证书将在 09-9-18 下午3:27 到期]
9. sm       705 Tue Dec 16 18:00:56 CST 2008 org/zlex/tool/Main$1.class
11. X.509, CN=www.zlex.org, OU=zlex, O=zlex, L=BJ, ST=BJ, C=CN
12. [证书将在 09-9-18 下午3:27 到期]
14. sm       779 Tue Dec 16 18:00:56 CST 2008 org/zlex/tool/Main$2.class
16. X.509, CN=www.zlex.org, OU=zlex, O=zlex, L=BJ, ST=BJ, C=CN
17. [证书将在 09-9-18 下午3:27 到期]
19. sm     12672 Tue Dec 16 18:00:56 CST 2008 org/zlex/tool/Main.class
21. X.509, CN=www.zlex.org, OU=zlex, O=zlex, L=BJ, ST=BJ, C=CN
22. [证书将在 09-9-18 下午3:27 到期]

25. s = 已验证签名
26. m = 在清单中列出条目
27. k = 在密钥库中至少找到了一个证书
28. i = 在身份作用域内至少找到了一个证书
30. jar 已验证。
32. 警告：
33. 此 jar 包含签名者证书将在六个月内过期的条目。

代码签名认证的用途主要是对发布的软件做验证，支持 Sun Java .jar (Java Applet) 文件(J2SE)和 J2ME MIDlet Suite 文件。

[**Java加密技术（九）——初探SSL**](http://snowolf.iteye.com/blog/397693)

**博客分类：**

* [Java／Security](http://snowolf.iteye.com/category/68576)

[Java](http://www.iteye.com/blogs/tag/Java)[Tomcat](http://www.iteye.com/blogs/tag/Tomcat)[ssl](http://www.iteye.com/blogs/tag/ssl)[keystore](http://www.iteye.com/blogs/tag/keystore)[keytool](http://www.iteye.com/blogs/tag/keytool)

    在[**Java加密技术（八）**](http://snowolf.iteye.com/blog/391931)中，我们模拟了一个基于RSA非对称加密网络的安全通信。现在我们深度了解一下现有的安全网络通信——SSL。   
    我们需要构建一个由CA机构签发的有效证书，这里我们使用上文中生成的自签名证书**zlex.cer**   
    这里，我们将证书导入到我们的密钥库。

**Shell代码  [收藏代码](javascript:void())**

1. keytool -import -alias www.zlex.org -file d:/zlex.cer -keystore d:/zlex.keystore

其中   
-import表示**导入**   
-alias指定别名，这里是**www.zlex.org**   
-file指定算法，这里是**d:/zlex.cer**   
-keystore指定存储位置，这里是**d:/zlex.keystore**   
在这里我使用的密码为**654321**   
  
控制台输出：

**Console代码  [收藏代码](javascript:void())**

1. 输入keystore密码：
2. 再次输入新密码:
3. 所有者:CN=www.zlex.org, OU=zlex, O=zlex, L=BJ, ST=BJ, C=CN
4. 签发人:CN=www.zlex.org, OU=zlex, O=zlex, L=BJ, ST=BJ, C=CN
5. 序列号:4a1e48df
6. 有效期: Thu May 28 16:18:39 CST 2009 至Wed Aug 26 16:18:39 CST 2009
7. 证书指纹:
8. MD5:19:CA:E6:36:E2:DF:AD:96:31:97:2F:A9:AD:FC:37:6A
9. SHA1:49:88:30:59:29:45:F1:69:CA:97:A9:6D:8A:CF:08:D2:C3:D5:C0:C4
10. 签名算法名称:SHA1withRSA
11. 版本: 3
12. 信任这个认证？ [否]：  y
13. 认证已添加至keystore中

OK，最复杂的准备工作已经完成。   
接下来我们将域名**www.zlex.org**定位到本机上。打开**C:\Windows\System32\drivers\etc\hosts**文件，将**www.zlex.org**绑定在本机上。在文件末尾追加**127.0.0.1       www.zlex.org**。现在通过地址栏访问[http://www.zlex.org](http://www.zlex.org/" \t "_blank)，或者通过**ping**命令，如果能够定位到本机，域名映射就搞定了。   
现在，配置tomcat。先将**zlex.keystore**拷贝到tomcat的conf目录下，然后配置**server.xml**。将如下内容加入配置文件

**Xml代码  [收藏代码](javascript:void())**

1. **<Connector**
2. SSLEnabled="true"
3. URIEncoding="UTF-8"
4. clientAuth="false"
5. keystoreFile="conf/zlex.keystore"
6. keystorePass="123456"
7. maxThreads="150"
8. port="443"
9. protocol="HTTP/1.1"
10. scheme="https"
11. secure="true"
12. sslProtocol="TLS" **/>**

注意**clientAuth="false"**测试阶段，置为**false**，正式使用时建议使用**true**。现在启动tomcat，访问<https://www.zlex.org/>。http://snowolf.iteye.com/upload/attachment/108754/66433698-0a4c-3869-9cd5-d4288a3e51e2.jpg   
显然，证书未能通过认证，这个时候你可以选择安装证书（上文中的**zlex.cer**文件就是证书），作为**受信任的根证书颁发机构**导入，再次重启浏览器（IE，其他浏览器对于域名www.zlex.org不支持本地方式访问），访问<https://www.zlex.org/>，你会看到地址栏中会有个小锁http://snowolf.iteye.com/upload/attachment/108757/1ca0c1b4-7d04-3d68-8b8b-9466380fb7b4.jpg，就说明安装成功。所有的浏览器联网操作已经在RSA加密解密系统的保护之下了。但似乎我们感受不到。   
这个时候很多人开始怀疑，如果我们要手工做一个这样的https的访问是不是需要把浏览器的这些个功能都实现呢？不需要！   
  
接着上篇内容，给出如下代码实现：

**Java代码  [收藏代码](javascript:void())**

1. **import** java.io.FileInputStream;
2. **import** java.security.KeyStore;
3. **import** java.security.PrivateKey;
4. **import** java.security.PublicKey;
5. **import** java.security.Signature;
6. **import** java.security.cert.Certificate;
7. **import** java.security.cert.CertificateFactory;
8. **import** java.security.cert.X509Certificate;
9. **import** java.util.Date;
11. **import** javax.crypto.Cipher;
12. **import** javax.net.ssl.HttpsURLConnection;
13. **import** javax.net.ssl.KeyManagerFactory;
14. **import** javax.net.ssl.SSLContext;
15. **import** javax.net.ssl.SSLSocketFactory;
16. **import** javax.net.ssl.TrustManagerFactory;
18. /\*\*
19. \* 证书组件
20. \*
21. \* @author 梁栋
22. \* @version 1.0
23. \* @since 1.0
24. \*/
25. **public** **abstract** **class** CertificateCoder **extends** Coder {
27. /\*\*
28. \* Java密钥库(Java Key Store，JKS)KEY\_STORE
29. \*/
30. **public** **static** **final** String KEY\_STORE = "JKS";
32. **public** **static** **final** String X509 = "X.509";
33. **public** **static** **final** String SunX509 = "SunX509";
34. **public** **static** **final** String SSL = "SSL";
36. /\*\*
37. \* 由KeyStore获得私钥
38. \*
39. \* @param keyStorePath
40. \* @param alias
41. \* @param password
42. \* @return
43. \* @throws Exception
44. \*/
45. **private** **static** PrivateKey getPrivateKey(String keyStorePath, String alias,
46. String password) **throws** Exception {
47. KeyStore ks = getKeyStore(keyStorePath, password);
48. PrivateKey key = (PrivateKey) ks.getKey(alias, password.toCharArray());
49. **return** key;
50. }
52. /\*\*
53. \* 由Certificate获得公钥
54. \*
55. \* @param certificatePath
56. \* @return
57. \* @throws Exception
58. \*/
59. **private** **static** PublicKey getPublicKey(String certificatePath)
60. **throws** Exception {
61. Certificate certificate = getCertificate(certificatePath);
62. PublicKey key = certificate.getPublicKey();
63. **return** key;
64. }
66. /\*\*
67. \* 获得Certificate
68. \*
69. \* @param certificatePath
70. \* @return
71. \* @throws Exception
72. \*/
73. **private** **static** Certificate getCertificate(String certificatePath)
74. **throws** Exception {
75. CertificateFactory certificateFactory = CertificateFactory
76. .getInstance(X509);
77. FileInputStream in = **new** FileInputStream(certificatePath);
79. Certificate certificate = certificateFactory.generateCertificate(in);
80. in.close();
82. **return** certificate;
83. }
85. /\*\*
86. \* 获得Certificate
87. \*
88. \* @param keyStorePath
89. \* @param alias
90. \* @param password
91. \* @return
92. \* @throws Exception
93. \*/
94. **private** **static** Certificate getCertificate(String keyStorePath,
95. String alias, String password) **throws** Exception {
96. KeyStore ks = getKeyStore(keyStorePath, password);
97. Certificate certificate = ks.getCertificate(alias);
99. **return** certificate;
100. }
102. /\*\*
103. \* 获得KeyStore
104. \*
105. \* @param keyStorePath
106. \* @param password
107. \* @return
108. \* @throws Exception
109. \*/
110. **private** **static** KeyStore getKeyStore(String keyStorePath, String password)
111. **throws** Exception {
112. FileInputStream is = **new** FileInputStream(keyStorePath);
113. KeyStore ks = KeyStore.getInstance(KEY\_STORE);
114. ks.load(is, password.toCharArray());
115. is.close();
116. **return** ks;
117. }
119. /\*\*
120. \* 私钥加密
121. \*
122. \* @param data
123. \* @param keyStorePath
124. \* @param alias
125. \* @param password
126. \* @return
127. \* @throws Exception
128. \*/
129. **public** **static** **byte**[] encryptByPrivateKey(**byte**[] data, String keyStorePath,
130. String alias, String password) **throws** Exception {
131. // 取得私钥
132. PrivateKey privateKey = getPrivateKey(keyStorePath, alias, password);
134. // 对数据加密
135. Cipher cipher = Cipher.getInstance(privateKey.getAlgorithm());
136. cipher.init(Cipher.ENCRYPT\_MODE, privateKey);
138. **return** cipher.doFinal(data);
140. }
142. /\*\*
143. \* 私钥解密
144. \*
145. \* @param data
146. \* @param keyStorePath
147. \* @param alias
148. \* @param password
149. \* @return
150. \* @throws Exception
151. \*/
152. **public** **static** **byte**[] decryptByPrivateKey(**byte**[] data, String keyStorePath,
153. String alias, String password) **throws** Exception {
154. // 取得私钥
155. PrivateKey privateKey = getPrivateKey(keyStorePath, alias, password);
157. // 对数据加密
158. Cipher cipher = Cipher.getInstance(privateKey.getAlgorithm());
159. cipher.init(Cipher.DECRYPT\_MODE, privateKey);
161. **return** cipher.doFinal(data);
163. }
165. /\*\*
166. \* 公钥加密
167. \*
168. \* @param data
169. \* @param certificatePath
170. \* @return
171. \* @throws Exception
172. \*/
173. **public** **static** **byte**[] encryptByPublicKey(**byte**[] data, String certificatePath)
174. **throws** Exception {
176. // 取得公钥
177. PublicKey publicKey = getPublicKey(certificatePath);
178. // 对数据加密
179. Cipher cipher = Cipher.getInstance(publicKey.getAlgorithm());
180. cipher.init(Cipher.ENCRYPT\_MODE, publicKey);
182. **return** cipher.doFinal(data);
184. }
186. /\*\*
187. \* 公钥解密
188. \*
189. \* @param data
190. \* @param certificatePath
191. \* @return
192. \* @throws Exception
193. \*/
194. **public** **static** **byte**[] decryptByPublicKey(**byte**[] data, String certificatePath)
195. **throws** Exception {
196. // 取得公钥
197. PublicKey publicKey = getPublicKey(certificatePath);
199. // 对数据加密
200. Cipher cipher = Cipher.getInstance(publicKey.getAlgorithm());
201. cipher.init(Cipher.DECRYPT\_MODE, publicKey);
203. **return** cipher.doFinal(data);
205. }
207. /\*\*
208. \* 验证Certificate
209. \*
210. \* @param certificatePath
211. \* @return
212. \*/
213. **public** **static** **boolean** verifyCertificate(String certificatePath) {
214. **return** verifyCertificate(**new** Date(), certificatePath);
215. }
217. /\*\*
218. \* 验证Certificate是否过期或无效
219. \*
220. \* @param date
221. \* @param certificatePath
222. \* @return
223. \*/
224. **public** **static** **boolean** verifyCertificate(Date date, String certificatePath) {
225. **boolean** status = **true**;
226. **try** {
227. // 取得证书
228. Certificate certificate = getCertificate(certificatePath);
229. // 验证证书是否过期或无效
230. status = verifyCertificate(date, certificate);
231. } **catch** (Exception e) {
232. status = **false**;
233. }
234. **return** status;
235. }
237. /\*\*
238. \* 验证证书是否过期或无效
239. \*
240. \* @param date
241. \* @param certificate
242. \* @return
243. \*/
244. **private** **static** **boolean** verifyCertificate(Date date, Certificate certificate) {
245. **boolean** status = **true**;
246. **try** {
247. X509Certificate x509Certificate = (X509Certificate) certificate;
248. x509Certificate.checkValidity(date);
249. } **catch** (Exception e) {
250. status = **false**;
251. }
252. **return** status;
253. }
255. /\*\*
256. \* 签名
257. \*
258. \* @param keyStorePath
259. \* @param alias
260. \* @param password
261. \*
262. \* @return
263. \* @throws Exception
264. \*/
265. **public** **static** String sign(**byte**[] sign, String keyStorePath, String alias,
266. String password) **throws** Exception {
267. // 获得证书
268. X509Certificate x509Certificate = (X509Certificate) getCertificate(
269. keyStorePath, alias, password);
270. // 获取私钥
271. KeyStore ks = getKeyStore(keyStorePath, password);
272. // 取得私钥
273. PrivateKey privateKey = (PrivateKey) ks.getKey(alias, password
274. .toCharArray());
276. // 构建签名
277. Signature signature = Signature.getInstance(x509Certificate
278. .getSigAlgName());
279. signature.initSign(privateKey);
280. signature.update(sign);
281. **return** encryptBASE64(signature.sign());
282. }
284. /\*\*
285. \* 验证签名
286. \*
287. \* @param data
288. \* @param sign
289. \* @param certificatePath
290. \* @return
291. \* @throws Exception
292. \*/
293. **public** **static** **boolean** verify(**byte**[] data, String sign,
294. String certificatePath) **throws** Exception {
295. // 获得证书
296. X509Certificate x509Certificate = (X509Certificate) getCertificate(certificatePath);
297. // 获得公钥
298. PublicKey publicKey = x509Certificate.getPublicKey();
299. // 构建签名
300. Signature signature = Signature.getInstance(x509Certificate
301. .getSigAlgName());
302. signature.initVerify(publicKey);
303. signature.update(data);
305. **return** signature.verify(decryptBASE64(sign));
307. }
309. /\*\*
310. \* 验证Certificate
311. \*
312. \* @param keyStorePath
313. \* @param alias
314. \* @param password
315. \* @return
316. \*/
317. **public** **static** **boolean** verifyCertificate(Date date, String keyStorePath,
318. String alias, String password) {
319. **boolean** status = **true**;
320. **try** {
321. Certificate certificate = getCertificate(keyStorePath, alias,
322. password);
323. status = verifyCertificate(date, certificate);
324. } **catch** (Exception e) {
325. status = **false**;
326. }
327. **return** status;
328. }
330. /\*\*
331. \* 验证Certificate
332. \*
333. \* @param keyStorePath
334. \* @param alias
335. \* @param password
336. \* @return
337. \*/
338. **public** **static** **boolean** verifyCertificate(String keyStorePath, String alias,
339. String password) {
340. **return** verifyCertificate(**new** Date(), keyStorePath, alias, password);
341. }
343. /\*\*
344. \* 获得SSLSocektFactory
345. \*
346. \* @param password
347. \*            密码
348. \* @param keyStorePath
349. \*            密钥库路径
350. \*
351. \* @param trustKeyStorePath
352. \*            信任库路径
353. \* @return
354. \* @throws Exception
355. \*/
356. **private** **static** SSLSocketFactory getSSLSocketFactory(String password,
357. String keyStorePath, String trustKeyStorePath) **throws** Exception {
358. // 初始化密钥库
359. KeyManagerFactory keyManagerFactory = KeyManagerFactory
360. .getInstance(SunX509);
361. KeyStore keyStore = getKeyStore(keyStorePath, password);
362. keyManagerFactory.init(keyStore, password.toCharArray());
364. // 初始化信任库
365. TrustManagerFactory trustManagerFactory = TrustManagerFactory
366. .getInstance(SunX509);
367. KeyStore trustkeyStore = getKeyStore(trustKeyStorePath, password);
368. trustManagerFactory.init(trustkeyStore);
370. // 初始化SSL上下文
371. SSLContext ctx = SSLContext.getInstance(SSL);
372. ctx.init(keyManagerFactory.getKeyManagers(), trustManagerFactory
373. .getTrustManagers(), **null**);
374. SSLSocketFactory sf = ctx.getSocketFactory();
376. **return** sf;
377. }
379. /\*\*
380. \* 为HttpsURLConnection配置SSLSocketFactory
381. \*
382. \* @param conn
383. \*            HttpsURLConnection
384. \* @param password
385. \*            密码
386. \* @param keyStorePath
387. \*            密钥库路径
388. \*
389. \* @param trustKeyStorePath
390. \*            信任库路径
391. \* @throws Exception
392. \*/
393. **public** **static** **void** configSSLSocketFactory(HttpsURLConnection conn,
394. String password, String keyStorePath, String trustKeyStorePath)
395. **throws** Exception {
396. conn.setSSLSocketFactory(getSSLSocketFactory(password, keyStorePath,
397. trustKeyStorePath));
398. }
399. }

增加了**configSSLSocketFactory**方法供外界调用，该方法为HttpsURLConnection配置了SSLSocketFactory。当HttpsURLConnection配置了SSLSocketFactory后，我们就可以通过HttpsURLConnection的getInputStream、getOutputStream，像往常使用HttpURLConnection做操作了。尤其要说明一点，未配置SSLSocketFactory前，HttpsURLConnection的getContentLength()获得值永远都是**-1**。   
  
给出相应测试类：

**Java代码  [收藏代码](javascript:void())**

1. **import** **static** org.junit.Assert.\*;
3. **import** java.io.DataInputStream;
4. **import** java.io.InputStream;
5. **import** java.net.URL;
7. **import** javax.net.ssl.HttpsURLConnection;
9. **import** org.junit.Test;
11. /\*\*
12. \*
13. \* @author 梁栋
14. \* @version 1.0
15. \* @since 1.0
16. \*/
17. **public** **class** CertificateCoderTest {
18. **private** String password = "123456";
19. **private** String alias = "www.zlex.org";
20. **private** String certificatePath = "d:/zlex.cer";
21. **private** String keyStorePath = "d:/zlex.keystore";
22. **private** String clientKeyStorePath = "d:/zlex-client.keystore";
23. **private** String clientPassword = "654321";
25. @Test
26. **public** **void** test() **throws** Exception {
27. System.err.println("公钥加密——私钥解密");
28. String inputStr = "Ceritifcate";
29. **byte**[] data = inputStr.getBytes();
31. **byte**[] encrypt = CertificateCoder.encryptByPublicKey(data,
32. certificatePath);
34. **byte**[] decrypt = CertificateCoder.decryptByPrivateKey(encrypt,
35. keyStorePath, alias, password);
36. String outputStr = **new** String(decrypt);
38. System.err.println("加密前: " + inputStr + "\n\r" + "解密后: " + outputStr);
40. // 验证数据一致
41. assertArrayEquals(data, decrypt);
43. // 验证证书有效
44. assertTrue(CertificateCoder.verifyCertificate(certificatePath));
46. }
48. @Test
49. **public** **void** testSign() **throws** Exception {
50. System.err.println("私钥加密——公钥解密");
52. String inputStr = "sign";
53. **byte**[] data = inputStr.getBytes();
55. **byte**[] encodedData = CertificateCoder.encryptByPrivateKey(data,
56. keyStorePath, alias, password);
58. **byte**[] decodedData = CertificateCoder.decryptByPublicKey(encodedData,
59. certificatePath);
61. String outputStr = **new** String(decodedData);
62. System.err.println("加密前: " + inputStr + "\n\r" + "解密后: " + outputStr);
63. assertEquals(inputStr, outputStr);
65. System.err.println("私钥签名——公钥验证签名");
66. // 产生签名
67. String sign = CertificateCoder.sign(encodedData, keyStorePath, alias,
68. password);
69. System.err.println("签名:\r" + sign);
71. // 验证签名
72. **boolean** status = CertificateCoder.verify(encodedData, sign,
73. certificatePath);
74. System.err.println("状态:\r" + status);
75. assertTrue(status);
77. }
79. @Test
80. **public** **void** testHttps() **throws** Exception {
81. URL url = **new** URL("https://www.zlex.org/examples/");
82. HttpsURLConnection conn = (HttpsURLConnection) url.openConnection();
84. conn.setDoInput(**true**);
85. conn.setDoOutput(**true**);
87. CertificateCoder.configSSLSocketFactory(conn, clientPassword,
88. clientKeyStorePath, clientKeyStorePath);
90. InputStream is = conn.getInputStream();
92. **int** length = conn.getContentLength();
94. DataInputStream dis = **new** DataInputStream(is);
95. **byte**[] data = **new** **byte**[length];
96. dis.readFully(data);
98. dis.close();
99. System.err.println(**new** String(data));
100. conn.disconnect();
101. }
102. }

注意**testHttps**方法，几乎和我们往常做HTTP访问没有差别，我们来看控制台输出：

**Console代码  [收藏代码](javascript:void())**

1. <!--
2. Licensed to the Apache Software Foundation (ASF) under one or more
3. contributor license agreements.  See the NOTICE file distributed with
4. this work for additional information regarding copyright ownership.
5. The ASF licenses this file to You under the Apache License, Version 2.0
6. (the "License"); you may not use this file except in compliance with
7. the License.  You may obtain a copy of the License at
9. http://www.apache.org/licenses/LICENSE-2.0
11. Unless required by applicable law or agreed to in writing, software
12. distributed under the License is distributed on an "AS IS" BASIS,
13. WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14. See the License for the specific language governing permissions and
15. limitations under the License.
16. -->
17. <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
18. <HTML><HEAD><TITLE>Apache Tomcat Examples</TITLE>
19. <META http-equiv=Content-Type content="text/html">
20. </HEAD>
21. <BODY>
22. <P>
23. <H3>Apache Tomcat Examples</H3>
24. <P></P>
25. <ul>
26. <li><a href="servlets">Servlets examples</a></li>
27. <li><a href="jsp">JSP Examples</a></li>
28. </ul>
29. </BODY></HTML>

通过浏览器直接访问<https://www.zlex.org/examples/>你也会获得上述内容。也就是说应用甲方作为服务器构建tomcat服务，乙方可以通过上述方式访问甲方受保护的SSL应用，并且不需要考虑具体的加密解密问题。甲乙双方可以经过相应配置，通过双方的tomcat配置有效的SSL服务，简化上述代码实现，完全通过证书配置完成SSL双向认证！http://snowolf.iteye.com/images/smiles/icon_biggrin.gif

[**Java加密技术（十）——单向认证**](http://snowolf.iteye.com/blog/398198)

**博客分类：**

* [Java／Security](http://snowolf.iteye.com/category/68576)

[Java](http://www.iteye.com/blogs/tag/Java)[tomcat](http://www.iteye.com/blogs/tag/tomcat)[keytool](http://www.iteye.com/blogs/tag/keytool)[keystore](http://www.iteye.com/blogs/tag/keystore)

    在[**Java 加密技术（九）**](http://snowolf.iteye.com/blog/397693)中，我们使用自签名证书完成了认证。接下来，我们使用第三方CA签名机构完成证书签名。   
    这里我们使用[thawte](https://www.thawte.com/)提供的测试用21天免费ca证书。   
    1.要在该网站上注明你的域名，这里使用**www.zlex.org**作为测试用域名（请勿使用该域名作为你的域名地址，该域名受法律保护！请使用其他非注册域名！）。   
    2.如果域名有效，你会收到邮件要求你访问<https://www.thawte.com/cgi/server/try.exe>获得ca证书。   
    3.复述密钥库的创建。 

**Shell代码  [收藏代码](javascript:void())**

1. keytool -genkey -validity 36000 -alias www.zlex.org -keyalg RSA -keystore d:\zlex.keystore

在这里我使用的密码为 **123456**   
  
控制台输出：

**Console代码  [收藏代码](javascript:void())**

1. 输入keystore密码：
2. 再次输入新密码:
3. 您的名字与姓氏是什么？
4. [Unknown]：  www.zlex.org
5. 您的组织单位名称是什么？
6. [Unknown]：  zlex
7. 您的组织名称是什么？
8. [Unknown]：  zlex
9. 您所在的城市或区域名称是什么？
10. [Unknown]：  BJ
11. 您所在的州或省份名称是什么？
12. [Unknown]：  BJ
13. 该单位的两字母国家代码是什么
14. [Unknown]：  CN
15. CN=www.zlex.org, OU=zlex, O=zlex, L=BJ, ST=BJ, C=CN 正确吗？
16. [否]：  Y
18. 输入<tomcat>的主密码
19. （如果和 keystore 密码相同，按回车）：
20. 再次输入新密码:

    4.通过如下命令，从zlex.keystore中导出CA证书申请。 

**Shell代码  [收藏代码](javascript:void())**

1. keytool -certreq -alias www.zlex.org -file d:\zlex.csr -keystore d:\zlex.keystore -v

你会获得zlex.csr文件，可以用记事本打开，内容如下格式：

**Text代码  [收藏代码](javascript:void())**

1. -----BEGIN NEW CERTIFICATE REQUEST-----
2. MIIBnDCCAQUCAQAwXDELMAkGA1UEBhMCQ04xCzAJBgNVBAgTAkJKMQswCQYDVQQHEwJCSjENMAsG
3. A1UEChMEemxleDENMAsGA1UECxMEemxleDEVMBMGA1UEAxMMd3d3LnpsZXgub3JnMIGfMA0GCSqG
4. SIb3DQEBAQUAA4GNADCBiQKBgQCR6DXU9Mp+mCKO7cv9JPsj0n1Ec/GpM09qvhpgX3FNad/ZWSDc
5. vU77YXZSoF9hQp3w1LC+eeKgd2MlVpXTvbVwBNVd2HiQPp37ic6BUUjSaX8LHtCl7l0BIEye9qQ2
6. j8G0kak7e8ZA0s7nb3Ymq/K8BV7v0MQIdhIc1bifK9ZDewIDAQABoAAwDQYJKoZIhvcNAQEFBQAD
7. gYEAMA1r2fbZPtNx37U9TRwadCH2TZZecwKJS/hskNm6ryPKIAp9APWwAyj8WJHRBz5SpZM4zmYO
8. oMCI8BcnY2A4JP+R7/SwXTdH/xcg7NVghd9A2SCgqMpF7KMfc5dE3iygdiPu+UhY200Dvpjx8gmJ
9. 1UbH3+nqMUyCrZgURFslOUY=
10. -----END NEW CERTIFICATE REQUEST-----

    5.将上述文件内容拷贝到<https://www.thawte.com/cgi/server/try.exe>中，点击next，获得回应内容，这里是p7b格式。   
内容如下：

**Text代码  [收藏代码](javascript:void())**

1. -----BEGIN PKCS7-----
2. MIIF3AYJKoZIhvcNAQcCoIIFzTCCBckCAQExADALBgkqhkiG9w0BBwGgggWxMIID
3. EDCCAnmgAwIBAgIQA/mx/pKoaB+KGX2hveFU9zANBgkqhkiG9w0BAQUFADCBhzEL
4. MAkGA1UEBhMCWkExIjAgBgNVBAgTGUZPUiBURVNUSU5HIFBVUlBPU0VTIE9OTFkx
5. HTAbBgNVBAoTFFRoYXd0ZSBDZXJ0aWZpY2F0aW9uMRcwFQYDVQQLEw5URVNUIFRF
6. U1QgVEVTVDEcMBoGA1UEAxMTVGhhd3RlIFRlc3QgQ0EgUm9vdDAeFw0wOTA1Mjgw
7. MDIxMzlaFw0wOTA2MTgwMDIxMzlaMFwxCzAJBgNVBAYTAkNOMQswCQYDVQQIEwJC
8. SjELMAkGA1UEBxMCQkoxDTALBgNVBAoTBHpsZXgxDTALBgNVBAsTBHpsZXgxFTAT
9. BgNVBAMTDHd3dy56bGV4Lm9yZzCBnzANBgkqhkiG9w0BAQEFAAOBjQAwgYkCgYEA
10. keg11PTKfpgiju3L/ST7I9J9RHPxqTNPar4aYF9xTWnf2Vkg3L1O+2F2UqBfYUKd
11. 8NSwvnnioHdjJVaV0721cATVXdh4kD6d+4nOgVFI0ml/Cx7Qpe5dASBMnvakNo/B
12. tJGpO3vGQNLO5292JqvyvAVe79DECHYSHNW4nyvWQ3sCAwEAAaOBpjCBozAMBgNV
13. HRMBAf8EAjAAMB0GA1UdJQQWMBQGCCsGAQUFBwMBBggrBgEFBQcDAjBABgNVHR8E
14. OTA3MDWgM6Axhi9odHRwOi8vY3JsLnRoYXd0ZS5jb20vVGhhd3RlUHJlbWl1bVNl
15. cnZlckNBLmNybDAyBggrBgEFBQcBAQQmMCQwIgYIKwYBBQUHMAGGFmh0dHA6Ly9v
16. Y3NwLnRoYXd0ZS5jb20wDQYJKoZIhvcNAQEFBQADgYEATPuxZbtJJSPmXvfrr1yz
17. xqM06IwTZ6UU0lZRG7I0WufMjNMKdpn8hklUhE17mxAhGSpewLVVeLR7uzBLFkuC
18. X7wMXxhoYdJZtNai72izU6Rd1oknao7diahvRxPK4IuQ7y2oZ511/4T4vgY6iRAj
19. q4q76HhPJrVRL/sduaiu+gYwggKZMIICAqADAgECAgEAMA0GCSqGSIb3DQEBBAUA
20. MIGHMQswCQYDVQQGEwJaQTEiMCAGA1UECBMZRk9SIFRFU1RJTkcgUFVSUE9TRVMg
21. T05MWTEdMBsGA1UEChMUVGhhd3RlIENlcnRpZmljYXRpb24xFzAVBgNVBAsTDlRF
22. U1QgVEVTVCBURVNUMRwwGgYDVQQDExNUaGF3dGUgVGVzdCBDQSBSb290MB4XDTk2
23. MDgwMTAwMDAwMFoXDTIwMTIzMTIxNTk1OVowgYcxCzAJBgNVBAYTAlpBMSIwIAYD
24. VQQIExlGT1IgVEVTVElORyBQVVJQT1NFUyBPTkxZMR0wGwYDVQQKExRUaGF3dGUg
25. Q2VydGlmaWNhdGlvbjEXMBUGA1UECxMOVEVTVCBURVNUIFRFU1QxHDAaBgNVBAMT
26. E1RoYXd0ZSBUZXN0IENBIFJvb3QwgZ8wDQYJKoZIhvcNAQEBBQADgY0AMIGJAoGB
27. ALV9kG+Os6x/DOhm+tKUQfzVMWGhE95sFmEtkMMTX2Zi4n6i6BvzoReJ5njzt1LF
28. cqu4EUk9Ji20egKKfmqRzmQFLP7+1niSdfJEUE7cKY40QoI99270PTrLjJeaMcCl
29. +AYl+kD+RL5BtuKKU3PurYcsCsre6aTvjMcqpTJOGeSPAgMBAAGjEzARMA8GA1Ud
30. EwEB/wQFMAMBAf8wDQYJKoZIhvcNAQEEBQADgYEAgozj7BkD9O8si2V0v+EZ/t7E
31. fz/LC8y6mD7IBUziHy5/53ymGAGLtyhXHvX+UIE6UWbHro3IqVkrmY5uC93Z2Wew
32. A/6edK3KFUcUikrLeewM7gmqsiASEKx2mKRKlu12jXyNS5tXrPWRDvUKtFC1uL9a
33. 12rFAQS2BkIk7aU+ghYxAA==
34. -----END PKCS7-----

将其存储为zlex.p7b   
    6.将由CA签发的证书导入密钥库。 

**Shell代码  [收藏代码](javascript:void())**

1. keytool -import -trustcacerts -alias www.zlex.org -file d:\zlex.p7b -keystore d:\zlex.keystore -v

在这里我使用的密码为 **123456**   
  
    控制台输出：

**Console代码  [收藏代码](javascript:void())**

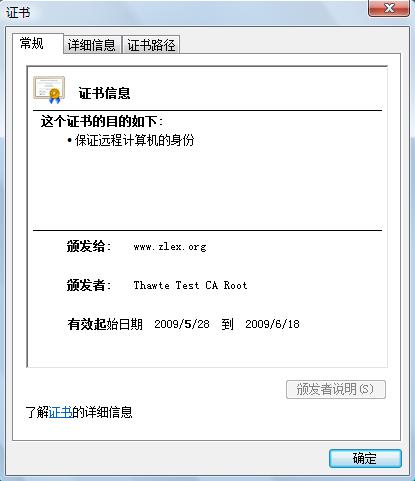
1. 输入keystore密码：
3. 回复中的最高级认证：
5. 所有者:CN=Thawte Test CA Root, OU=TEST TEST TEST, O=Thawte Certification, ST=FOR
6. TESTING PURPOSES ONLY, C=ZA
7. 签发人:CN=Thawte Test CA Root, OU=TEST TEST TEST, O=Thawte Certification, ST=FOR
8. TESTING PURPOSES ONLY, C=ZA
9. 序列号:0
10. 有效期: Thu Aug 01 08:00:00 CST 1996 至Fri Jan 01 05:59:59 CST 2021
11. 证书指纹:
12. MD5:5E:E0:0E:1D:17:B7:CA:A5:7D:36:D6:02:DF:4D:26:A4
13. SHA1:39:C6:9D:27:AF:DC:EB:47:D6:33:36:6A:B2:05:F1:47:A9:B4:DA:EA
14. 签名算法名称:MD5withRSA
15. 版本: 3
17. 扩展:
19. #1: ObjectId: 2.5.29.19 Criticality=true
20. BasicConstraints:[
21. CA:true
22. PathLen:2147483647
23. ]

26. ... 是不可信的。 还是要安装回复？ [否]：  Y
27. 认证回复已安装在 keystore中
28. [正在存储 d:\zlex.keystore]

    7.域名定位   
    将域名www.zlex.org定位到本机上。打开C:\Windows\System32\drivers\etc\hosts文件，将www.zlex.org绑定在本机上。在文件末尾追加127.0.0.1       www.zlex.org。现在通过地址栏访问http://www.zlex.org，或者通过ping命令，如果能够定位到本机，域名映射就搞定了。   
  
    8.配置server.xml

**Xml代码  [收藏代码](javascript:void())**

1. **<Connector**
2. keystoreFile="conf/zlex.keystore"
3. keystorePass="123456"
4. truststoreFile="conf/zlex.keystore"
5. truststorePass="123456"
6. SSLEnabled="true"
7. URIEncoding="UTF-8"
8. clientAuth="false"
9. maxThreads="150"
10. port="443"
11. protocol="HTTP/1.1"
12. scheme="https"
13. secure="true"
14. sslProtocol="TLS" **/>**

将文件**zlex.keystore**拷贝到tomcat的**conf**目录下，重新启动tomcat。访问<https://www.zlex.org/>，我们发现联网有些迟钝。大约5秒钟后，网页正常显示，同时有如下图所示：   
http://snowolf.iteye.com/upload/attachment/109146/ba0a327e-afd9-35ed-8836-8ec92b72e8cf.jpg   
浏览器验证了该CA机构的有效性。   
  
打开证书，如下图所示：   
   
  
调整测试类：

**Java代码  [收藏代码](javascript:void())**

1. **import** **static** org.junit.Assert.\*;
3. **import** java.io.DataInputStream;
4. **import** java.io.InputStream;
5. **import** java.net.URL;
7. **import** javax.net.ssl.HttpsURLConnection;
9. **import** org.junit.Test;
11. /\*\*
12. \*
13. \* @author 梁栋
14. \* @version 1.0
15. \* @since 1.0
16. \*/
17. **public** **class** CertificateCoderTest {
18. **private** String password = "123456";
19. **private** String alias = "www.zlex.org";
20. **private** String certificatePath = "d:/zlex.cer";
21. **private** String keyStorePath = "d:/zlex.keystore";
23. @Test
24. **public** **void** test() **throws** Exception {
25. System.err.println("公钥加密——私钥解密");
26. String inputStr = "Ceritifcate";
27. **byte**[] data = inputStr.getBytes();
29. **byte**[] encrypt = CertificateCoder.encryptByPublicKey(data,
30. certificatePath);
32. **byte**[] decrypt = CertificateCoder.decryptByPrivateKey(encrypt,
33. keyStorePath, alias, password);
34. String outputStr = **new** String(decrypt);
36. System.err.println("加密前: " + inputStr + "\n\r" + "解密后: " + outputStr);
38. // 验证数据一致
39. assertArrayEquals(data, decrypt);
41. // 验证证书有效
42. assertTrue(CertificateCoder.verifyCertificate(certificatePath));
44. }
46. @Test
47. **public** **void** testSign() **throws** Exception {
48. System.err.println("私钥加密——公钥解密");
50. String inputStr = "sign";
51. **byte**[] data = inputStr.getBytes();
53. **byte**[] encodedData = CertificateCoder.encryptByPrivateKey(data,
54. keyStorePath, alias, password);
56. **byte**[] decodedData = CertificateCoder.decryptByPublicKey(encodedData,
57. certificatePath);
59. String outputStr = **new** String(decodedData);
60. System.err.println("加密前: " + inputStr + "\n\r" + "解密后: " + outputStr);
61. assertEquals(inputStr, outputStr);
63. System.err.println("私钥签名——公钥验证签名");
64. // 产生签名
65. String sign = CertificateCoder.sign(encodedData, keyStorePath, alias,
66. password);
67. System.err.println("签名:\r" + sign);
69. // 验证签名
70. **boolean** status = CertificateCoder.verify(encodedData, sign,
71. certificatePath);
72. System.err.println("状态:\r" + status);
73. assertTrue(status);
75. }
77. @Test
78. **public** **void** testHttps() **throws** Exception {
79. URL url = **new** URL("https://www.zlex.org/examples/");
80. HttpsURLConnection conn = (HttpsURLConnection) url.openConnection();
82. conn.setDoInput(**true**);
83. conn.setDoOutput(**true**);
85. CertificateCoder.configSSLSocketFactory(conn, password, keyStorePath,
86. keyStorePath);
88. InputStream is = conn.getInputStream();
90. **int** length = conn.getContentLength();
92. DataInputStream dis = **new** DataInputStream(is);
93. **byte**[] data = **new** **byte**[length];
94. dis.readFully(data);
96. dis.close();
97. conn.disconnect();
98. System.err.println(**new** String(data));
99. }
100. }

再次执行，验证通过！http://snowolf.iteye.com/images/smiles/icon_biggrin.gif   
由此，我们了基于SSL协议的认证过程。测试类的testHttps方法模拟了一次浏览器的HTTPS访问。http://snowolf.iteye.com/images/smiles/icon_biggrin.gif

[**Java加密技术（十一）——双向认证**](http://snowolf.iteye.com/blog/510985)

**博客分类：**

* [Java／Security](http://snowolf.iteye.com/category/68576)

[openssl](http://www.iteye.com/blogs/tag/openssl)[pem](http://www.iteye.com/blogs/tag/pem)[ca](http://www.iteye.com/blogs/tag/ca)[PKCS#7](http://www.iteye.com/blogs/tag/PKCS%237)[PKCS#10](http://www.iteye.com/blogs/tag/PKCS%2310)

对于双向认证，做一个简单的描述。   
服务器端下发证书，客户端接受证书。证书带有公钥信息，用于验证服务器端、对数据加密/解密，起到OSI五类服务的认证（鉴别）服务和保密性服务。   
  
这只是单向认证，为什么？因为客户端可以验证服务器端，但服务器端不能验证客户端！   
如果客户端也有这样一个证书，服务器端也就能够验证客户端，这就是双向认证了！   
  
换言之，当你用银行的“U盾”之类的U盘与银行账户交互时，在你验证银行服务器的同时，服务器也在验证你！这种双重验证，正是网银系统的安全关键！   
  
单向认证见**[Java加密技术（十)](http://snowolf.iteye.com/blog/398198" \t "_blank)**   
双向认证需要一个CA机构签发这样的客户端、服务器端证书，首先需要CA机构构建一个根证书。keytool可以构建证书但不能构建我们需要的根证书，openssl则可以！   
  
根证书签发客户端证书，根私钥签发服务器端证书！   
  
我们直接使用linux下的openssl来完成CA，需要修改openssl.cnf文件，在ubuntu下的**/etc/ssl/**目录下，找到[ CA\_default ]修改dir变量。   
原文

**引用**

[ CA\_default ]   
  
#dir = ./demoCA # Where everything is kept

我们把c盘的ca目录作为CA认证的根目录，文件修改后如下所示：

**引用**

[ CA\_default ]   
  
dir = $ENV::HOME/ca # Where everything is kept

我们需要在用户目录下构建一个ca目录，以及子目录，如下所下：   
ca   
|\_\_certs   
|\_\_newcerts   
|\_\_private   
|\_\_crl   
  
执行如下操作：

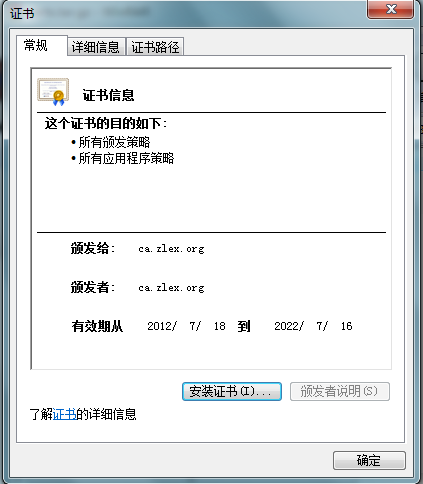
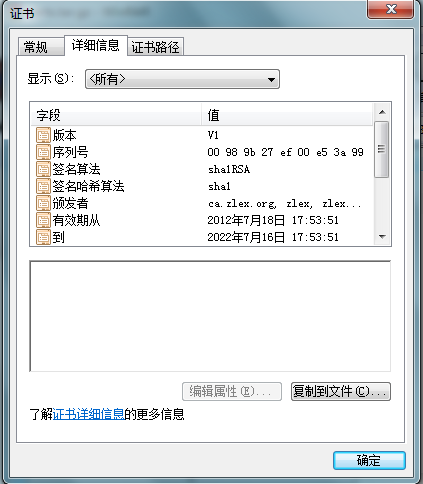
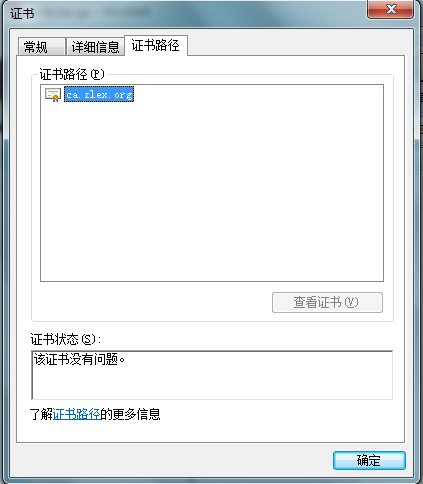
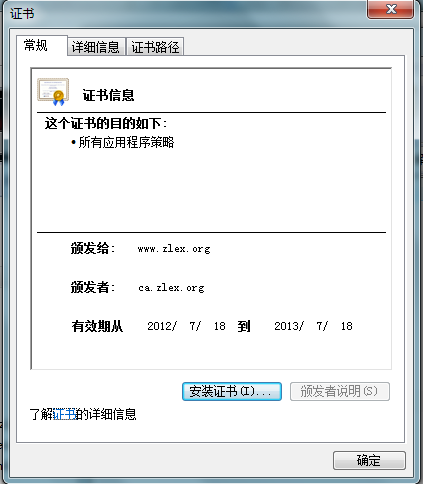
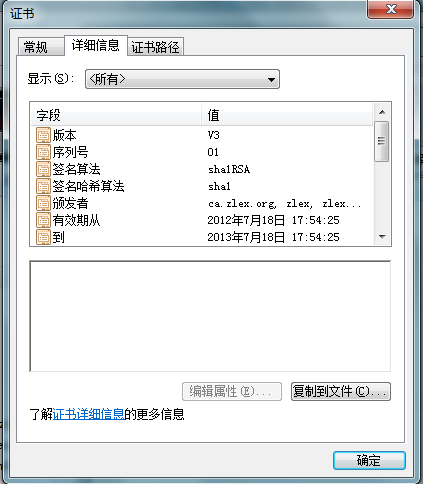
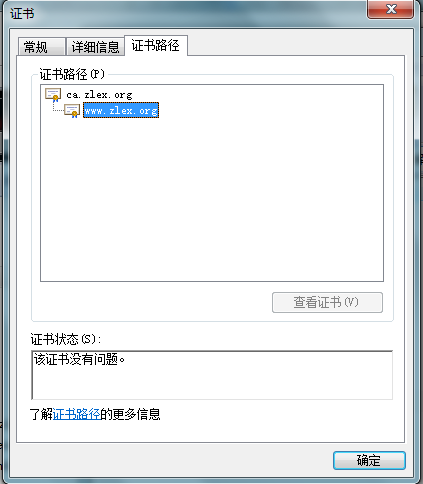
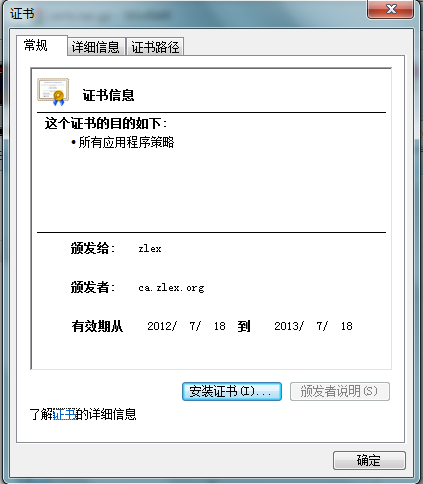
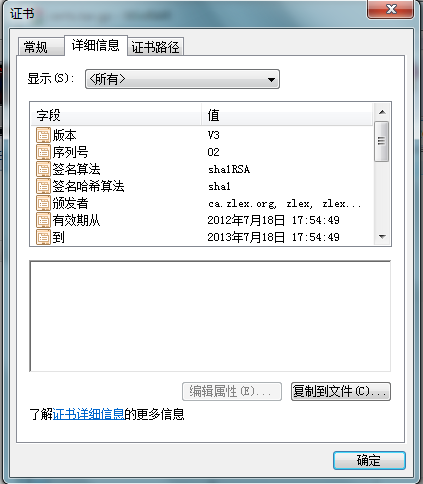
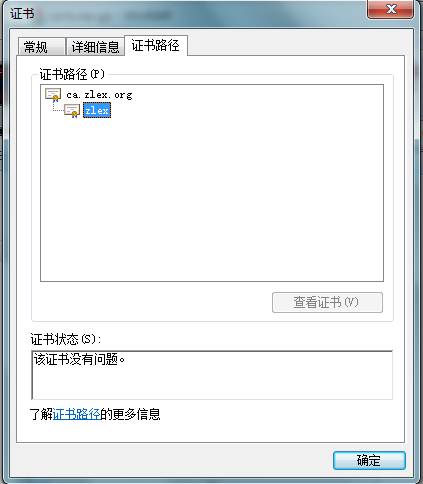
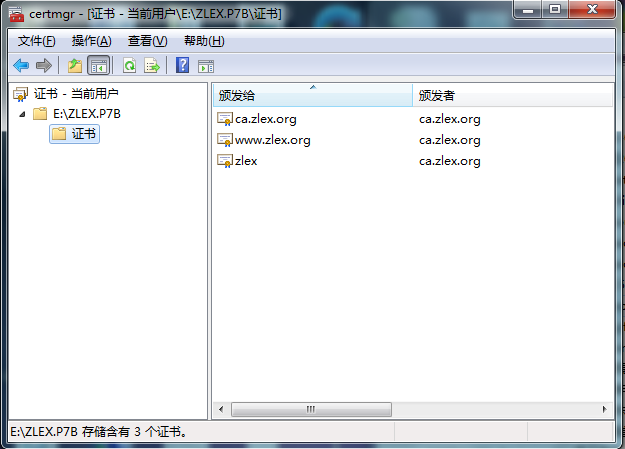
**Shell代码  [收藏代码](javascript:void())**

1. #!/bin/bash
3. ca\_path=ca
4. certs\_path=$ca\_path/certs
5. newcerts\_path=$ca\_path/newcerts
6. private\_path=$ca\_path/private
7. crl\_path=$ca\_path/crl
9. echo 移除CA根目录
10. rm -rf ca
12. echo 构建CA根目录
13. mkdir ca
15. echo 构建子目录
16. mkdir certs
17. mkdir newcerts
18. mkdir private
19. mkdir crl
21. #构建文件
22. touch $ca\_path/index.txt
23. echo 01 > $ca\_path/serial
24. echo
26. #构建随机数
27. openssl rand -out $private\_path/.rand 1000
28. echo
30. echo 生成根证书私钥
31. openssl genrsa -des3 -out $private\_path/ca.pem 2048
32. echo
34. echo 查看私钥信息
35. openssl rsa -noout -text -in $private\_path/ca.pem
36. echo
38. echo 生成根证书请求
39. openssl req -new -key $private\_path/ca.pem -out $certs\_path/ca.csr -subj "/C=CN/ST=BJ/L=BJ/O=zlex/OU=zlex/CN=ca.zlex.org"
40. echo
42. echo 查看证书请求
43. openssl req -in $certs\_path/ca.csr -text -noout
44. echo
46. echo 签发根证书
47. openssl ca -create\_serial -out $certs\_path/ca.crt -days 3650 -batch -keyfile $private\_path/ca.pem -selfsign -extensions v3\_ca -infiles $certs\_path/ca.csr
48. #openssl x509 -req -sha1 -extensions v3\_ca -signkey $private\_path/ca.pem -in $certs\_path/ca.csr -out $certs\_path/ca.crt -days 3650
49. echo
51. echo 查看证书详情
52. openssl x509 -in $certs\_path/ca.crt -text -noout
53. echo
55. echo 证书转换——根证书
56. openssl pkcs12 -export -clcerts -in $certs\_path/ca.crt -inkey $private\_path/ca.pem -out $certs\_path/ca.p12
57. echo
59. echo 生成服务器端私钥
60. openssl genrsa -des3 -out $private\_path/server.pem 1024
61. echo
63. echo 查看私钥信息
64. openssl rsa -noout -text -in $private\_path/server.pem
65. echo
67. echo 生成服务器端证书请求
68. openssl req -new -key $private\_path/server.pem -out $certs\_path/server.csr -subj "/C=CN/ST=BJ/L=BJ/O=zlex/OU=zlex/CN=www.zlex.org"
69. echo
71. echo 查看证书请求
72. openssl req -in $certs\_path/server.csr -text -noout
73. echo
75. echo 签发服务器端证书
76. openssl ca -in $certs\_path/server.csr -out $certs\_path/server.crt -cert $certs\_path/ca.crt -keyfile $private\_path/ca.pem -days 365 -notext
77. #openssl x509 -req -days 365 -sha1 -extensions v3\_req -CA $certs\_path/ca.crt -CAkey $private\_path/ca.pem -CAserial $ca\_path/serial -CAcreateserial -in $certs\_path/server.csr -out $certs\_path/server.crt
78. echo
80. echo 查看证书详情
81. openssl x509 -in $certs\_path/server.crt -text -noout
82. echo
84. echo 证书转换——服务器端
85. openssl pkcs12 -export -clcerts -in $certs\_path/server.crt -inkey $private\_path/server.pem -out $certs\_path/server.p12
86. echo
88. echo 生成客户端私钥
89. openssl genrsa -des3 -out $private\_path/client.pem 1024
90. echo
92. echo 生成客户端私钥
93. openssl genrsa -des3 -out $private\_path/client.pem 1024
94. echo
96. echo 查看私钥信息
97. openssl rsa -noout -text -in $private\_path/client.pem
98. echo
100. echo 生成客户端证书请求
101. openssl req -new -key $private\_path/client.pem -out $certs\_path/client.csr -subj "/C=CN/ST=BJ/L=BJ/O=zlex/OU=zlex/CN=zlex"
102. echo
104. echo 查看证书请求
105. openssl req -in $certs\_path/client.csr -text -noout
106. echo
108. echo 签发客户端证书
109. openssl ca -in $certs\_path/client.csr -out $certs\_path/client.crt -cert $certs\_path/ca.crt -keyfile $private\_path/ca.pem -days 365 -notext
110. #openssl x509 -req -days 365 -sha1 -extensions dir\_sect -CA $certs\_path/ca.crt -CAkey $private\_path/ca.pem -CAserial $ca\_path/serial -in $certs\_path/client.csr -out $certs\_path/client.crt
111. echo
113. echo 查看证书详情
114. openssl x509 -in $certs\_path/client.crt -text -noout
115. echo
117. echo 证书转换——客户端
118. openssl pkcs12 -export -clcerts -in $certs\_path/client.crt -inkey $private\_path/client.pem -out $certs\_path/client.p12
119. echo
121. echo 生成证书链PKCS#7
122. openssl crl2pkcs7 -nocrl -certfile $certs\_path/server.crt -certfile $certs\_path/ca.crt -certfile $certs\_path/client.crt -out
123. form PEM -out $certs\_path/zlex.p7b
124. echo
126. echo 查看证书链
127. openssl pkcs7 -in $certs\_path/zlex.p7b -print\_certs -noout

这个脚本就是最重要的结晶了！http://snowolf.iteye.com/images/smiles/icon_smile.gif   
  
执行结果，如下：

**引用**

生成根证书私钥   
Generating RSA private key, 2048 bit long modulus   
..................................+++   
.............................................................+++   
e is 65537 (0x10001)   
Enter pass phrase for ca/private/ca.pem:   
Verifying - Enter pass phrase for ca/private/ca.pem:   
  
查看私钥信息   
Enter pass phrase for ca/private/ca.pem:   
Private-Key: (2048 bit)   
modulus:   
    00:d4:18:ab:5f:ad:b7:d0:09:d4:68:63:b5:db:8a:   
    d1:a1:db:7e:f3:bb:bb:c2:be:a7:35:17:9e:bb:20:   
    d3:1f:ed:63:e7:7d:29:6d:d2:7c:60:06:47:53:a6:   
    23:b0:bd:94:65:3f:57:1e:00:51:f3:a1:9a:1b:83:   
    14:a5:53:72:86:21:a2:57:22:2f:6a:a9:46:50:8c:   
    f0:51:cf:e6:83:5b:23:dc:f9:ea:6c:2e:51:20:61:   
    d1:84:9f:28:e8:01:89:b5:cb:55:68:4a:11:b1:06:   
    56:31:21:16:c8:ac:2b:68:31:e1:de:12:d3:21:12:   
    83:36:4c:ca:a8:b5:7e:b9:a7:63:4e:8e:e0:79:0f:   
    0e:91:36:28:7c:dd:9a:e2:e0:98:8b:91:7f:09:7d:   
    20:bb:37:f2:ab:aa:f0:ef:ae:68:7e:db:ca:db:33:   
    84:48:5a:e3:ff:0b:08:0e:96:6d:01:c8:12:35:ec:   
    9f:31:55:7f:53:7e:bd:fb:c4:16:b8:1f:17:29:42:   
    0f:0e:04:57:14:18:fd:e5:d6:3f:40:04:cd:85:dd:   
    d3:eb:2f:9a:bf:3c:8a:60:01:88:2f:43:0a:8b:bb:   
    50:13:f8:cc:68:f9:10:eb:f9:7e:63:de:62:55:32:   
    a8:fe:ce:51:67:79:c9:a6:3b:a3:c9:d7:81:7c:48:   
    f3:d1   
publicExponent: 65537 (0x10001)   
privateExponent:   
    00:b0:8a:e4:43:1c:df:6e:bc:6f:e0:80:76:c4:8a:   
    75:5a:0b:d1:4d:61:cb:b5:1b:6b:24:c7:47:69:ad:   
    b5:ee:d2:73:a1:21:4e:95:ca:69:9a:a8:3f:40:c2:   
    7e:dc:c3:c0:bc:d2:0f:5a:ba:9b:7c:76:dc:46:e0:   
    42:14:27:34:a1:af:67:68:ad:dc:d8:24:94:91:c1:   
    ee:db:ba:78:be:87:e3:7f:31:4b:4e:c6:f2:e2:48:   
    69:d4:c1:82:94:33:8b:84:15:ff:3e:72:c0:ed:20:   
    40:28:5e:c9:8f:39:b8:5b:df:81:89:8f:13:cc:68:   
    93:6d:64:58:20:3c:0a:82:ce:ec:2f:9b:b2:9d:ca:   
    e7:19:22:98:29:6e:7c:4d:85:45:17:50:8f:5d:b1:   
    45:be:42:af:1a:7f:84:26:b4:5d:a6:22:8a:07:e8:   
    b3:b4:5a:59:45:20:b5:ef:1c:81:25:9e:73:74:04:   
    d6:57:30:2c:a7:25:50:7c:d7:87:73:b3:d0:c2:8b:   
    c9:02:8e:15:9e:40:41:a5:7a:a9:d8:85:fb:5b:9a:   
    59:83:bc:80:fa:74:e6:88:14:70:33:61:d7:f5:51:   
    47:8f:60:51:cb:c4:97:66:65:94:f0:ed:58:ca:80:   
    c1:89:e0:55:68:4c:69:21:0f:08:27:e0:87:11:df:   
    b7:bd   
prime1:   
    00:f7:ff:b0:40:de:62:b6:a2:e5:d0:f5:fa:28:3d:   
    d3:30:30:89:8f:d1:ae:df:e9:09:ee:a0:b0:a5:a5:   
    a4:e5:93:97:7e:e6:0b:09:70:4c:62:99:5e:7d:45:   
    2f:fd:21:5a:31:d9:26:7f:39:5f:6e:eb:36:02:4e:   
    18:99:1b:38:13:99:f5:f3:a3:6b:93:83:67:fb:58:   
    67:d4:07:eb:e3:2f:31:b3:97:8f:f6:86:1f:15:08:   
    1a:4b:b5:a8:06:97:72:9c:74:ab:53:1f:ac:ee:fb:   
    59:03:39:a6:5c:a8:77:43:c0:2c:14:60:0e:71:3d:   
    70:b6:59:09:40:86:04:54:bf   
prime2:   
    00:da:f0:73:2c:bd:52:a5:0d:9a:40:c4:34:fc:c9:   
    cf:0f:67:8a:02:01:ca:e7:b8:4e:57:da:0c:0d:b2:   
    f9:f3:f2:e4:4c:82:61:aa:04:2c:88:39:18:bd:86:   
    d6:dc:d0:e9:6c:c6:6f:d9:87:59:57:9b:1a:6b:c9:   
    56:c1:4d:33:ce:3e:15:b9:42:4e:e0:f8:14:91:c3:   
    fe:63:b2:13:29:99:a7:a6:13:cc:f8:9c:38:29:28:   
    dd:ed:d1:a3:7c:05:2c:26:a0:84:c6:09:9e:42:ef:   
    7b:5e:50:c7:57:e3:bc:02:93:0b:74:a1:b5:0b:6e:   
    23:18:8b:82:6f:ac:3c:0b:6f   
exponent1:   
    7c:a1:23:4b:46:37:27:7f:6f:ac:f6:a0:93:ae:96:   
    3e:46:76:2b:2f:7e:09:8a:8c:72:3e:90:e7:7d:fa:   
    03:61:8b:a5:bb:27:da:c3:73:af:ad:51:9d:f4:b2:   
    2c:2c:a1:ae:21:69:c6:4f:e7:d4:cf:21:a2:40:ea:   
    fd:ae:7f:1c:e2:a7:86:9c:1e:c8:d0:25:e6:5b:44:   
    3a:7b:0c:a1:6c:2b:37:0c:b8:cd:74:13:94:b7:30:   
    b7:d1:7f:b2:68:53:b1:aa:b4:1a:9e:f5:82:58:10:   
    20:9d:cd:2c:0d:81:7a:2b:ce:3b:23:16:be:f3:d8:   
    7b:da:fc:da:4f:3f:47:f3   
exponent2:   
    66:c9:5c:49:34:d9:08:04:4a:d6:fd:46:a3:27:5b:   
    be:af:ad:6b:23:cc:4e:dd:88:6a:56:44:32:6a:44:   
    4e:f3:49:9b:61:da:d8:26:fd:81:36:cd:16:ad:a7:   
    52:24:02:72:be:f6:e3:f9:57:48:79:d8:fd:a1:98:   
    c9:47:a5:7a:be:4b:14:9e:bc:c9:81:ae:a6:80:8d:   
    7d:e0:ac:7e:6b:54:f9:f3:71:d7:86:00:17:d2:c7:   
    de:4e:fd:a1:cc:0b:de:56:9d:ff:1b:a4:e1:67:ed:   
    53:6a:39:2c:5a:0e:7a:66:ee:89:e3:21:4c:2c:78:   
    ed:9d:11:af:bb:fc:b4:a1   
coefficient:   
    00:b1:23:a8:cc:b1:5e:2e:38:09:0c:b5:df:2c:c6:   
    15:e8:08:48:45:b9:9d:ec:6f:27:45:5b:a7:bc:b6:   
    b1:ec:a5:39:b4:40:8e:bc:40:1f:b9:4d:14:2e:18:   
    fb:87:1e:20:91:34:58:e3:ac:c3:4a:dc:a8:2a:97:   
    ce:aa:8d:62:0e:91:af:1f:53:d6:37:55:1d:14:9c:   
    01:98:34:77:28:d7:cf:f7:a0:2d:73:40:48:5e:ed:   
    ae:9b:15:42:06:e6:a3:5a:2b:b0:bc:ee:7a:bb:52:   
    e6:28:19:c2:e5:de:6f:4d:fa:fb:69:81:7b:13:2b:   
    01:87:bf:bf:66:8f:24:a1:8f   
  
生成根证书请求   
Enter pass phrase for ca/private/ca.pem:   
  
查看证书请求   
Certificate Request:   
    Data:   
        Version: 0 (0x0)   
        Subject: C=CN, ST=BJ, L=BJ, O=zlex, OU=zlex, CN=ca.zlex.org   
        Subject Public Key Info:   
            Public Key Algorithm: rsaEncryption   
            RSA Public Key: (2048 bit)   
                Modulus (2048 bit):   
                    00:d4:18:ab:5f:ad:b7:d0:09:d4:68:63:b5:db:8a:   
                    d1:a1:db:7e:f3:bb:bb:c2:be:a7:35:17:9e:bb:20:   
                    d3:1f:ed:63:e7:7d:29:6d:d2:7c:60:06:47:53:a6:   
                    23:b0:bd:94:65:3f:57:1e:00:51:f3:a1:9a:1b:83:   
                    14:a5:53:72:86:21:a2:57:22:2f:6a:a9:46:50:8c:   
                    f0:51:cf:e6:83:5b:23:dc:f9:ea:6c:2e:51:20:61:   
                    d1:84:9f:28:e8:01:89:b5:cb:55:68:4a:11:b1:06:   
                    56:31:21:16:c8:ac:2b:68:31:e1:de:12:d3:21:12:   
                    83:36:4c:ca:a8:b5:7e:b9:a7:63:4e:8e:e0:79:0f:   
                    0e:91:36:28:7c:dd:9a:e2:e0:98:8b:91:7f:09:7d:   
                    20:bb:37:f2:ab:aa:f0:ef:ae:68:7e:db:ca:db:33:   
                    84:48:5a:e3:ff:0b:08:0e:96:6d:01:c8:12:35:ec:   
                    9f:31:55:7f:53:7e:bd:fb:c4:16:b8:1f:17:29:42:   
                    0f:0e:04:57:14:18:fd:e5:d6:3f:40:04:cd:85:dd:   
                    d3:eb:2f:9a:bf:3c:8a:60:01:88:2f:43:0a:8b:bb:   
                    50:13:f8:cc:68:f9:10:eb:f9:7e:63:de:62:55:32:   
                    a8:fe:ce:51:67:79:c9:a6:3b:a3:c9:d7:81:7c:48:   
                    f3:d1   
                Exponent: 65537 (0x10001)   
        Attributes:   
            a0:00   
    Signature Algorithm: sha1WithRSAEncryption   
        af:91:f8:56:6f:db:de:cb:df:2c:87:93:99:ac:4b:51:12:a2:   
        c1:2b:09:d2:58:7c:e1:07:5c:53:9f:f3:e1:b6:3a:e9:08:e7:   
        65:89:3b:0a:01:83:24:a3:b5:74:65:50:a5:77:bc:30:1b:7d:   
        80:8b:4c:92:ec:81:91:6e:b7:8f:05:e7:1d:b2:89:84:18:8c:   
        5f:66:be:19:15:ba:ba:c3:f7:0d:c3:7d:7a:11:47:17:e5:cf:   
        87:69:2e:15:91:d7:db:9d:8e:c9:0f:81:71:fa:00:93:33:2c:   
        99:e1:be:76:06:f1:8a:e6:8b:1d:9b:07:70:f0:f2:44:91:ed:   
        a2:ed:28:91:5f:6a:8a:f3:cf:ab:0d:b3:05:30:72:19:86:ae:   
        c6:2d:a4:22:9f:21:cf:55:0c:b7:79:44:01:6e:36:43:a5:dc:   
        a0:ea:46:2a:b0:9d:b3:53:4a:57:fc:72:1b:4c:52:cc:a3:39:   
        d6:49:d6:f4:8c:e2:bf:5a:a6:6e:69:7c:f2:bc:7b:02:b7:f5:   
        91:7f:94:2b:8c:58:0f:aa:a3:72:93:46:fe:08:29:08:51:eb:   
        c6:a0:4e:7a:e1:bd:c6:0b:11:9d:63:96:af:22:ee:7b:79:84:   
        cd:e7:f0:23:17:e7:9f:a2:73:c5:15:e1:f5:a1:af:8d:58:f5:   
        e0:eb:57:fd   
  
签发根证书   
Using configuration from /etc/pki/tls/openssl.cnf   
Enter pass phrase for ca/private/ca.pem:   
Check that the request matches the signature   
Signature ok   
Certificate Details:   
        Serial Number: 1 (0x1)   
        Validity   
            Not Before: Jul 24 08:15:59 2012 GMT   
            Not After : Jul 22 08:15:59 2022 GMT   
        Subject:   
            countryName               = CN   
            stateOrProvinceName       = BJ   
            organizationName          = zlex   
            organizationalUnitName    = zlex   
            commonName                = ca.zlex.org   
        X509v3 extensions:   
            X509v3 Subject Key Identifier:   
                7E:C9:9A:37:37:66:AC:79:41:63:F0:61:48:CD:24:39:2F:C2:0E:E9   
            X509v3 Authority Key Identifier:   
                keyid:7E:C9:9A:37:37:66:AC:79:41:63:F0:61:48:CD:24:39:2F:C2:0E:E9   
                DirName:/C=CN/ST=BJ/O=zlex/OU=zlex/CN=ca.zlex.org   
                serial:01   
  
            X509v3 Basic Constraints:   
                CA:TRUE   
Certificate is to be certified until Jul 22 08:15:59 2022 GMT (3650 days)   
  
Write out database with 1 new entries   
Data Base Updated   
  
查看证书详情   
Certificate:   
    Data:   
        Version: 3 (0x2)   
        Serial Number: 1 (0x1)   
        Signature Algorithm: sha1WithRSAEncryption   
        Issuer: C=CN, ST=BJ, O=zlex, OU=zlex, CN=ca.zlex.org   
        Validity   
            Not Before: Jul 24 08:15:59 2012 GMT   
            Not After : Jul 22 08:15:59 2022 GMT   
        Subject: C=CN, ST=BJ, O=zlex, OU=zlex, CN=ca.zlex.org   
        Subject Public Key Info:   
            Public Key Algorithm: rsaEncryption   
            RSA Public Key: (2048 bit)   
                Modulus (2048 bit):   
                    00:d4:18:ab:5f:ad:b7:d0:09:d4:68:63:b5:db:8a:   
                    d1:a1:db:7e:f3:bb:bb:c2:be:a7:35:17:9e:bb:20:   
                    d3:1f:ed:63:e7:7d:29:6d:d2:7c:60:06:47:53:a6:   
                    23:b0:bd:94:65:3f:57:1e:00:51:f3:a1:9a:1b:83:   
                    14:a5:53:72:86:21:a2:57:22:2f:6a:a9:46:50:8c:   
                    f0:51:cf:e6:83:5b:23:dc:f9:ea:6c:2e:51:20:61:   
                    d1:84:9f:28:e8:01:89:b5:cb:55:68:4a:11:b1:06:   
                    56:31:21:16:c8:ac:2b:68:31:e1:de:12:d3:21:12:   
                    83:36:4c:ca:a8:b5:7e:b9:a7:63:4e:8e:e0:79:0f:   
                    0e:91:36:28:7c:dd:9a:e2:e0:98:8b:91:7f:09:7d:   
                    20:bb:37:f2:ab:aa:f0:ef:ae:68:7e:db:ca:db:33:   
                    84:48:5a:e3:ff:0b:08:0e:96:6d:01:c8:12:35:ec:   
                    9f:31:55:7f:53:7e:bd:fb:c4:16:b8:1f:17:29:42:   
                    0f:0e:04:57:14:18:fd:e5:d6:3f:40:04:cd:85:dd:   
                    d3:eb:2f:9a:bf:3c:8a:60:01:88:2f:43:0a:8b:bb:   
                    50:13:f8:cc:68:f9:10:eb:f9:7e:63:de:62:55:32:   
                    a8:fe:ce:51:67:79:c9:a6:3b:a3:c9:d7:81:7c:48:   
                    f3:d1   
                Exponent: 65537 (0x10001)   
        X509v3 extensions:   
            X509v3 Subject Key Identifier:   
                7E:C9:9A:37:37:66:AC:79:41:63:F0:61:48:CD:24:39:2F:C2:0E:E9   
            X509v3 Authority Key Identifier:   
                keyid:7E:C9:9A:37:37:66:AC:79:41:63:F0:61:48:CD:24:39:2F:C2:0E:E9   
                DirName:/C=CN/ST=BJ/O=zlex/OU=zlex/CN=ca.zlex.org   
                serial:01   
  
            X509v3 Basic Constraints:   
                CA:TRUE   
    Signature Algorithm: sha1WithRSAEncryption   
        8a:99:b8:17:fc:64:7b:88:9c:1b:91:23:60:f4:5c:51:16:9a:   
        9f:42:b4:d3:a5:bb:79:ca:78:e3:fc:a7:af:66:da:ec:5a:8c:   
        81:c1:aa:04:32:a9:59:e0:d6:6a:f2:37:38:97:70:a5:27:5d:   
        14:73:2e:2d:73:78:1d:37:2c:04:f7:c3:99:9d:be:0c:dd:2a:   
        27:2c:0f:6e:95:96:01:c7:4c:99:f7:49:69:f9:ba:cb:62:b8:   
        c6:43:6c:5b:b5:cd:25:42:a7:fb:81:27:bc:d8:e4:95:26:7d:   
        da:50:f8:b8:be:0a:3d:54:35:d0:9d:22:e7:f0:f0:4c:7d:b4:   
        57:2e:98:91:1a:1d:49:e5:8e:48:f6:2b:54:7e:04:fc:1c:e3:   
        52:f7:04:f6:9b:bb:84:25:31:f7:31:6e:7f:fa:4c:e4:15:a2:   
        86:0a:1a:56:8c:ad:07:49:fb:bc:28:27:a3:95:ba:eb:b3:28:   
        db:11:78:ef:84:fc:3c:16:df:58:39:2e:14:8d:89:fe:7a:d2:   
        24:eb:a7:66:11:8c:88:55:40:e1:c3:3b:95:b2:bc:af:36:0e:   
        92:a8:cd:62:d5:57:9c:11:1b:f6:a1:36:5f:25:6c:16:c5:e2:   
        68:19:e7:12:3d:4b:07:24:81:e6:71:f9:59:c5:f9:1c:62:6d:   
        b3:24:b9:8a   
  
证书转换——根证书   
Enter pass phrase for ca/private/ca.pem:   
Enter Export Password:   
Verifying - Enter Export Password:   
  
生成服务器端私钥   
Generating RSA private key, 1024 bit long modulus   
......................................................++++++   
................++++++   
e is 65537 (0x10001)   
Enter pass phrase for ca/private/server.pem:   
Verifying - Enter pass phrase for ca/private/server.pem:   
  
查看私钥信息   
Enter pass phrase for ca/private/server.pem:   
Private-Key: (1024 bit)   
modulus:   
    00:d8:f9:bd:0a:a8:d3:97:98:b2:22:af:29:a9:31:   
    76:50:52:77:c8:3b:7c:91:75:db:b3:63:88:cc:00:   
    be:1a:6c:e6:80:23:90:37:5f:1a:d3:80:f2:7f:b5:   
    77:01:ec:85:3e:4e:c0:af:0d:77:c0:a5:8b:bc:c3:   
    fe:70:91:66:17:a4:ec:23:08:5b:e3:df:a3:40:2f:   
    e6:83:bd:3f:d0:62:9c:c0:36:ad:e7:cb:13:e8:34:   
    d7:6a:66:57:f5:bb:94:2f:7c:d5:27:7b:ee:e6:4f:   
    fc:ff:c1:a4:01:96:d6:a0:b8:46:1d:93:02:a6:c5:   
    00:bd:d9:e9:4e:2d:87:d5:95   
publicExponent: 65537 (0x10001)   
privateExponent:   
    4d:da:15:fd:6c:24:37:c1:bf:30:f8:be:af:09:a3:   
    55:20:b1:ff:f3:70:37:d5:1d:16:99:c1:2c:c9:9b:   
    6c:69:e4:ae:d7:93:d8:7a:54:6a:cd:5a:b5:7e:0c:   
    0c:71:ac:41:76:0a:67:05:23:11:c9:94:81:0f:a6:   
    0d:07:ee:a4:26:0e:20:ff:36:6c:f7:2d:fa:8e:39:   
    85:f8:b8:1a:e0:be:26:f8:24:3c:d4:d0:a0:89:9c:   
    48:15:d9:28:de:51:dd:14:3f:ca:c9:63:ed:5d:e4:   
    50:b0:06:5e:1b:f8:99:b4:49:f6:d6:cb:60:8a:7b:   
    fa:f8:6e:86:44:55:e5:45   
prime1:   
    00:ef:cc:38:ab:e6:98:71:09:32:5c:69:b3:e0:59:   
    9d:d7:7a:f9:e3:b9:cd:a8:84:74:1a:91:2a:db:2c:   
    96:40:5a:28:0b:99:6c:da:fa:ca:83:54:e0:59:06:   
    84:df:55:9a:04:9c:1c:6b:54:52:d5:31:d7:f9:0e:   
    9a:13:b0:ed:03   
prime2:   
    00:e7:a2:c3:03:55:d7:54:7c:3a:38:40:f1:ac:9a:   
    e8:dd:3a:5c:24:a6:78:34:c4:ce:24:c8:31:de:5a:   
    0e:df:09:df:7c:ad:36:14:e0:be:6d:2c:58:89:c6:   
    7e:ec:51:82:68:81:91:ed:b5:04:ff:c0:61:8e:aa:   
    5b:ee:6b:f3:87   
exponent1:   
    2a:22:0c:d7:0f:56:3b:8e:2d:1e:15:a8:78:43:e6:   
    ba:e4:ad:a1:78:95:0d:05:f0:cc:76:33:3c:7d:52:   
    0d:0e:8a:38:b7:85:6b:d8:62:da:be:80:08:c4:5f:   
    76:4a:39:1c:94:3d:5e:12:5b:d7:7f:c1:7d:ce:35:   
    fe:3d:b8:f7   
exponent2:   
    00:94:0b:ec:36:52:84:19:04:79:35:81:14:b5:ec:   
    20:8f:5d:00:8d:90:34:5e:0d:b7:6f:bc:e0:5a:ac:   
    16:bb:29:15:45:1b:73:e8:6e:28:67:a0:a3:4a:13:   
    ab:05:a1:a7:06:e2:61:81:9b:64:01:8e:55:0c:19:   
    08:3e:df:92:3b   
coefficient:   
    00:8e:4e:ee:04:55:cc:4f:0f:c0:02:a4:9d:08:a8:   
    4b:ec:72:7c:86:27:a9:0a:5e:1c:94:65:9e:c6:8a:   
    6a:5c:9b:76:5d:c0:ae:f8:36:61:15:3b:67:fb:15:   
    b3:cf:f4:2c:9b:56:66:13:89:89:69:01:d9:6e:b0:   
    f7:02:d4:06:c9   
  
生成服务器端证书请求   
Enter pass phrase for ca/private/server.pem:   
  
查看证书请求   
Certificate Request:   
    Data:   
        Version: 0 (0x0)   
        Subject: C=CN, ST=BJ, L=BJ, O=zlex, OU=zlex, CN=www.zlex.org   
        Subject Public Key Info:   
            Public Key Algorithm: rsaEncryption   
            RSA Public Key: (1024 bit)   
                Modulus (1024 bit):   
                    00:d8:f9:bd:0a:a8:d3:97:98:b2:22:af:29:a9:31:   
                    76:50:52:77:c8:3b:7c:91:75:db:b3:63:88:cc:00:   
                    be:1a:6c:e6:80:23:90:37:5f:1a:d3:80:f2:7f:b5:   
                    77:01:ec:85:3e:4e:c0:af:0d:77:c0:a5:8b:bc:c3:   
                    fe:70:91:66:17:a4:ec:23:08:5b:e3:df:a3:40:2f:   
                    e6:83:bd:3f:d0:62:9c:c0:36:ad:e7:cb:13:e8:34:   
                    d7:6a:66:57:f5:bb:94:2f:7c:d5:27:7b:ee:e6:4f:   
                    fc:ff:c1:a4:01:96:d6:a0:b8:46:1d:93:02:a6:c5:   
                    00:bd:d9:e9:4e:2d:87:d5:95   
                Exponent: 65537 (0x10001)   
        Attributes:   
            a0:00   
    Signature Algorithm: sha1WithRSAEncryption   
        2b:e9:b9:0b:e0:94:56:95:dd:59:1e:19:16:e0:f9:73:db:50:   
        63:d3:d4:4d:5c:9b:98:9f:a7:6d:9b:4d:ae:67:52:18:e1:42:   
        b0:66:7c:75:6a:db:98:bc:e6:47:08:aa:55:ca:ce:35:5c:5a:   
        60:8b:7b:c8:f0:10:8a:bd:5f:d7:c8:b8:48:03:18:7e:68:6e:   
        69:35:9c:c8:b0:c8:65:43:43:25:35:d7:d2:70:45:55:ab:78:   
        51:4d:22:c3:68:b2:97:b5:3c:86:e8:2b:43:de:5d:e4:b0:b5:   
        0e:eb:84:9d:42:81:ee:e0:0a:48:40:6a:93:a4:bd:3a:45:6f:   
        20:24   
  
签发服务器端证书   
Using configuration from /etc/pki/tls/openssl.cnf   
Enter pass phrase for ca/private/ca.pem:   
Check that the request matches the signature   
Signature ok   
Certificate Details:   
        Serial Number: 2 (0x2)   
        Validity   
            Not Before: Jul 24 08:16:15 2012 GMT   
            Not After : Jul 24 08:16:15 2013 GMT   
        Subject:   
            countryName               = CN   
            stateOrProvinceName       = BJ   
            organizationName          = zlex   
            organizationalUnitName    = zlex   
            commonName                = www.zlex.org   
        X509v3 extensions:   
            X509v3 Basic Constraints:   
                CA:FALSE   
            Netscape Comment:   
                OpenSSL Generated Certificate   
            X509v3 Subject Key Identifier:   
                CF:79:10:96:42:84:0C:51:DE:6E:DB:3C:5B:08:F1:E1:EB:0C:26:B9   
            X509v3 Authority Key Identifier:   
                keyid:7E:C9:9A:37:37:66:AC:79:41:63:F0:61:48:CD:24:39:2F:C2:0E:E9   
  
Certificate is to be certified until Jul 24 08:16:15 2013 GMT (365 days)   
Sign the certificate? [y/n]:y   
  
  
1 out of 1 certificate requests certified, commit? [y/n]y   
Write out database with 1 new entries   
Data Base Updated   
  
查看证书详情   
Certificate:   
    Data:   
        Version: 3 (0x2)   
        Serial Number: 2 (0x2)   
        Signature Algorithm: sha1WithRSAEncryption   
        Issuer: C=CN, ST=BJ, O=zlex, OU=zlex, CN=ca.zlex.org   
        Validity   
            Not Before: Jul 24 08:16:15 2012 GMT   
            Not After : Jul 24 08:16:15 2013 GMT   
        Subject: C=CN, ST=BJ, O=zlex, OU=zlex, CN=www.zlex.org   
        Subject Public Key Info:   
            Public Key Algorithm: rsaEncryption   
            RSA Public Key: (1024 bit)   
                Modulus (1024 bit):   
                    00:d8:f9:bd:0a:a8:d3:97:98:b2:22:af:29:a9:31:   
                    76:50:52:77:c8:3b:7c:91:75:db:b3:63:88:cc:00:   
                    be:1a:6c:e6:80:23:90:37:5f:1a:d3:80:f2:7f:b5:   
                    77:01:ec:85:3e:4e:c0:af:0d:77:c0:a5:8b:bc:c3:   
                    fe:70:91:66:17:a4:ec:23:08:5b:e3:df:a3:40:2f:   
                    e6:83:bd:3f:d0:62:9c:c0:36:ad:e7:cb:13:e8:34:   
                    d7:6a:66:57:f5:bb:94:2f:7c:d5:27:7b:ee:e6:4f:   
                    fc:ff:c1:a4:01:96:d6:a0:b8:46:1d:93:02:a6:c5:   
                    00:bd:d9:e9:4e:2d:87:d5:95   
                Exponent: 65537 (0x10001)   
        X509v3 extensions:   
            X509v3 Basic Constraints:   
                CA:FALSE   
            Netscape Comment:   
                OpenSSL Generated Certificate   
            X509v3 Subject Key Identifier:   
                CF:79:10:96:42:84:0C:51:DE:6E:DB:3C:5B:08:F1:E1:EB:0C:26:B9   
            X509v3 Authority Key Identifier:   
                keyid:7E:C9:9A:37:37:66:AC:79:41:63:F0:61:48:CD:24:39:2F:C2:0E:E9   
  
    Signature Algorithm: sha1WithRSAEncryption   
        3d:85:0a:f5:a6:8e:f5:13:1b:fc:74:b6:50:8f:fe:0c:e6:32:   
        0e:0c:5a:0a:75:2d:e8:15:39:2f:93:46:29:c6:cc:27:5a:36:   
        a0:93:f8:bc:38:d5:2d:55:b9:19:de:81:6f:b6:5f:1f:07:65:   
        81:c5:12:4e:ea:3e:09:d0:d5:b8:66:c1:cd:4d:5d:51:19:a1:   
        7f:7b:cb:dc:bf:b0:be:3e:f8:8b:74:d3:31:a9:95:a3:ef:25:   
        a3:1e:98:65:0f:d4:40:51:ef:42:02:72:f0:59:26:8a:e7:d6:   
        ca:34:ad:fb:3d:a8:e7:05:93:a6:78:bd:b5:90:51:83:06:2b:   
        95:db:01:0c:89:9f:74:a4:32:89:c5:15:c6:ec:e2:61:10:29:   
        70:da:c5:ea:d6:9c:be:c3:4c:a1:42:6a:26:2f:23:7c:90:51:   
        8f:51:ee:49:c9:6b:9c:0c:15:a2:d3:dc:90:19:db:4d:d1:ad:   
        ca:06:d1:e1:60:20:18:b1:6d:0b:17:f7:06:e6:e8:d1:b0:0c:   
        6d:55:16:f1:63:54:da:c2:3f:6c:e5:99:68:7a:a0:fa:29:5c:   
        dc:cf:34:90:fb:91:7b:e0:5d:bb:a0:9d:91:f3:17:bd:0b:5a:   
        69:d7:0c:24:75:ca:b2:08:da:bf:67:35:ce:01:d0:4e:45:81:   
        97:bd:fb:87   
  
证书转换——服务器端   
Enter pass phrase for ca/private/server.pem:   
Enter Export Password:   
Verifying - Enter Export Password:   
  
生成客户端私钥   
Generating RSA private key, 1024 bit long modulus   
..++++++   
...........++++++   
e is 65537 (0x10001)   
Enter pass phrase for ca/private/client.pem:   
Verifying - Enter pass phrase for ca/private/client.pem:   
  
查看私钥信息   
Enter pass phrase for ca/private/client.pem:   
Private-Key: (1024 bit)   
modulus:   
    00:b4:e9:7d:3d:6b:8b:07:94:7d:47:51:56:3e:0e:   
    92:2f:87:8c:60:0f:b8:cb:eb:90:6d:13:76:51:75:   
    e4:3e:b7:6e:1f:f0:63:5b:f7:ba:51:c0:04:1e:f1:   
    d0:ef:58:4a:35:47:4a:1a:11:72:fc:e9:10:82:ec:   
    3e:0d:ef:7d:17:a0:5e:93:b4:01:8f:a5:27:3c:3e:   
    a9:26:f0:00:ba:ca:24:98:92:51:3e:4b:d0:81:a7:   
    fc:14:e2:98:f5:27:f2:51:4c:a8:ae:b4:5f:e7:cc:   
    70:7e:23:57:92:6a:cf:d4:1d:6f:b3:52:8a:4a:1a:   
    1b:65:f0:4d:1c:0b:1f:50:eb   
publicExponent: 65537 (0x10001)   
privateExponent:   
    3a:35:b2:8d:73:af:fd:55:62:e5:f2:9e:dc:42:d5:   
    f8:a3:15:a0:c7:0e:3f:d6:e0:d6:a7:df:77:20:86:   
    bb:43:4c:14:cc:c5:3b:8f:3f:0d:14:ca:7e:a6:72:   
    02:c1:16:c7:83:d3:ad:05:96:49:18:38:ae:d7:92:   
    b3:eb:2e:05:43:d6:3d:04:3c:0b:fc:15:79:c5:85:   
    10:ed:21:6e:30:73:0b:a6:4f:9a:fe:db:4a:98:bc:   
    ec:03:7b:7f:e6:16:2f:a5:f3:5e:0d:cf:ce:eb:4a:   
    3e:c5:b9:7f:fc:4c:60:9e:0e:d4:aa:91:5a:46:f7:   
    b3:77:fc:0b:1b:62:70:b9   
prime1:   
    00:ef:6d:7f:92:6a:af:21:59:ed:fe:49:a8:7c:4a:   
    1d:4d:7c:f9:38:bf:e7:dc:42:41:e1:33:f9:e1:c7:   
    74:45:2e:1c:e4:40:8d:5f:1a:ac:11:9e:a4:6c:1d:   
    00:6d:4e:aa:4d:58:e9:92:84:ac:d9:29:67:e0:79:   
    a8:a3:15:e3:2d   
prime2:   
    00:c1:6f:21:c5:62:48:78:3a:0f:25:98:00:46:d6:   
    c2:2d:0f:96:fb:20:4b:f4:03:81:71:3f:6f:30:c0:   
    f3:a6:e6:f4:00:a4:fa:0b:97:e6:2a:21:8c:cb:c1:   
    28:eb:5f:f6:01:62:85:9a:37:98:e7:53:a4:8b:3f:   
    bd:77:eb:f3:77   
exponent1:   
    00:e3:71:e0:9b:85:af:22:7e:9c:a0:50:f6:b6:43:   
    6d:bc:bb:b8:c0:d9:44:f8:2f:15:08:4b:68:d8:bb:   
    b1:cf:3a:34:05:fc:f0:8f:64:f6:0a:b2:ea:bd:2d:   
    7b:c7:5a:d0:5b:33:d8:86:f0:74:86:c3:57:c3:9d:   
    ae:be:66:3f:6d   
exponent2:   
    00:82:4a:c9:04:9b:5f:15:1c:86:77:5c:1b:53:9b:   
    f4:cf:45:60:fd:66:93:c2:99:59:e7:5e:43:17:23:   
    e0:fa:db:36:1f:f9:00:34:2e:ec:ea:14:0f:32:6f:   
    b9:90:51:e2:f2:ab:da:32:36:a0:d7:b0:8f:74:fc:   
    4a:33:2c:cb:a1   
coefficient:   
    51:c1:7e:d7:0d:98:86:cb:ca:41:ea:aa:54:6c:00:   
    49:c3:18:12:c4:5b:75:fe:0d:0c:e2:2f:0f:93:8e:   
    8e:01:c5:9d:ff:40:2b:20:08:24:7f:a5:f2:da:67:   
    96:5e:e6:7e:1e:52:32:2f:88:ef:df:20:6a:75:ec:   
    28:cd:fa:a0   
  
生成客户端证书请求   
Enter pass phrase for ca/private/client.pem:   
  
查看证书请求   
Certificate Request:   
    Data:   
        Version: 0 (0x0)   
        Subject: C=CN, ST=BJ, L=BJ, O=zlex, OU=zlex, CN=zlex   
        Subject Public Key Info:   
            Public Key Algorithm: rsaEncryption   
            RSA Public Key: (1024 bit)   
                Modulus (1024 bit):   
                    00:b4:e9:7d:3d:6b:8b:07:94:7d:47:51:56:3e:0e:   
                    92:2f:87:8c:60:0f:b8:cb:eb:90:6d:13:76:51:75:   
                    e4:3e:b7:6e:1f:f0:63:5b:f7:ba:51:c0:04:1e:f1:   
                    d0:ef:58:4a:35:47:4a:1a:11:72:fc:e9:10:82:ec:   
                    3e:0d:ef:7d:17:a0:5e:93:b4:01:8f:a5:27:3c:3e:   
                    a9:26:f0:00:ba:ca:24:98:92:51:3e:4b:d0:81:a7:   
                    fc:14:e2:98:f5:27:f2:51:4c:a8:ae:b4:5f:e7:cc:   
                    70:7e:23:57:92:6a:cf:d4:1d:6f:b3:52:8a:4a:1a:   
                    1b:65:f0:4d:1c:0b:1f:50:eb   
                Exponent: 65537 (0x10001)   
        Attributes:   
            a0:00   
    Signature Algorithm: sha1WithRSAEncryption   
        91:5b:b2:2e:b3:54:14:92:7a:44:c0:59:11:0f:fe:08:50:33:   
        09:0f:73:d3:9d:15:43:07:66:4a:9e:7c:de:12:4d:bc:b6:3a:   
        7a:6b:36:40:3a:4b:ea:db:f7:2e:a1:de:ce:4f:a6:98:14:3b:   
        c0:f6:3d:fe:db:82:fa:c7:f1:1e:9a:6c:2b:ff:e6:a4:91:b1:   
        ab:20:44:91:a8:d9:1b:13:8f:9e:24:68:16:f3:c1:66:7b:3b:   
        29:b5:61:3d:be:88:00:d8:0a:1c:63:f0:25:6c:33:7d:86:80:   
        54:d5:75:db:6f:7e:9c:52:4c:70:0d:5a:88:ae:b5:1a:12:41:   
        e4:47   
  
签发客户端证书   
Using configuration from /etc/pki/tls/openssl.cnf   
Enter pass phrase for ca/private/ca.pem:   
Check that the request matches the signature   
Signature ok   
Certificate Details:   
        Serial Number: 3 (0x3)   
        Validity   
            Not Before: Jul 24 08:16:35 2012 GMT   
            Not After : Jul 24 08:16:35 2013 GMT   
        Subject:   
            countryName               = CN   
            stateOrProvinceName       = BJ   
            organizationName          = zlex   
            organizationalUnitName    = zlex   
            commonName                = zlex   
        X509v3 extensions:   
            X509v3 Basic Constraints:   
                CA:FALSE   
            Netscape Comment:   
                OpenSSL Generated Certificate   
            X509v3 Subject Key Identifier:   
                FD:85:1C:BA:E0:C4:81:F5:F4:92:F1:FC:8A:59:77:33:60:6F:47:F7   
            X509v3 Authority Key Identifier:   
                keyid:7E:C9:9A:37:37:66:AC:79:41:63:F0:61:48:CD:24:39:2F:C2:0E:E9   
  
Certificate is to be certified until Jul 24 08:16:35 2013 GMT (365 days)   
Sign the certificate? [y/n]:y   
  
  
1 out of 1 certificate requests certified, commit? [y/n]y   
Write out database with 1 new entries   
Data Base Updated   
  
查看证书详情   
Certificate:   
    Data:   
        Version: 3 (0x2)   
        Serial Number: 3 (0x3)   
        Signature Algorithm: sha1WithRSAEncryption   
        Issuer: C=CN, ST=BJ, O=zlex, OU=zlex, CN=ca.zlex.org   
        Validity   
            Not Before: Jul 24 08:16:35 2012 GMT   
            Not After : Jul 24 08:16:35 2013 GMT   
        Subject: C=CN, ST=BJ, O=zlex, OU=zlex, CN=zlex   
        Subject Public Key Info:   
            Public Key Algorithm: rsaEncryption   
            RSA Public Key: (1024 bit)   
                Modulus (1024 bit):   
                    00:b4:e9:7d:3d:6b:8b:07:94:7d:47:51:56:3e:0e:   
                    92:2f:87:8c:60:0f:b8:cb:eb:90:6d:13:76:51:75:   
                    e4:3e:b7:6e:1f:f0:63:5b:f7:ba:51:c0:04:1e:f1:   
                    d0:ef:58:4a:35:47:4a:1a:11:72:fc:e9:10:82:ec:   
                    3e:0d:ef:7d:17:a0:5e:93:b4:01:8f:a5:27:3c:3e:   
                    a9:26:f0:00:ba:ca:24:98:92:51:3e:4b:d0:81:a7:   
                    fc:14:e2:98:f5:27:f2:51:4c:a8:ae:b4:5f:e7:cc:   
                    70:7e:23:57:92:6a:cf:d4:1d:6f:b3:52:8a:4a:1a:   
                    1b:65:f0:4d:1c:0b:1f:50:eb   
                Exponent: 65537 (0x10001)   
        X509v3 extensions:   
            X509v3 Basic Constraints:   
                CA:FALSE   
            Netscape Comment:   
                OpenSSL Generated Certificate   
            X509v3 Subject Key Identifier:   
                FD:85:1C:BA:E0:C4:81:F5:F4:92:F1:FC:8A:59:77:33:60:6F:47:F7   
            X509v3 Authority Key Identifier:   
                keyid:7E:C9:9A:37:37:66:AC:79:41:63:F0:61:48:CD:24:39:2F:C2:0E:E9   
  
    Signature Algorithm: sha1WithRSAEncryption   
        b2:31:c0:15:a1:8f:2c:6d:61:0c:4f:6e:c1:fe:7a:88:e0:60:   
        ce:6d:43:b4:29:d8:4d:83:4d:ea:ce:f0:8e:c1:c7:3b:bd:30:   
        cb:92:71:11:7d:19:04:11:58:25:5d:1b:ed:6f:22:13:91:ea:   
        13:7f:0e:99:00:ec:fb:b3:a5:e2:b9:ea:ea:bb:35:09:3b:ca:   
        f5:49:ac:a1:d3:d5:ae:ff:ce:11:a9:2f:53:74:88:24:9f:f8:   
        b2:bc:02:4d:1a:bb:c1:53:3e:6e:31:52:4d:ac:f8:14:bd:b1:   
        0d:31:1d:aa:94:43:38:5e:fb:c2:26:3e:43:ba:25:3b:23:27:   
        a8:7d:5d:3d:f9:97:28:71:51:1d:a4:56:44:b4:f6:51:4a:2b:   
        8b:47:d3:10:49:04:cd:c3:58:62:75:bc:c7:6a:4c:d5:9a:a8:   
        e9:9c:23:ec:f8:26:e5:de:43:4e:f2:8d:c2:75:40:70:3f:03:   
        0f:74:78:7a:bc:ca:6f:90:a0:3e:3a:d2:92:16:d5:ca:af:93:   
        28:1f:24:3a:7e:2c:b9:db:87:10:68:e0:c9:6c:0b:5d:9f:15:   
        be:bc:13:22:af:7b:8f:e9:14:51:04:65:7a:69:18:c2:ca:4f:   
        cb:e5:4c:62:41:88:b1:ee:ac:43:14:34:6d:58:af:52:b1:25:   
        76:f3:0e:8f   
  
证书转换——客户端   
Enter pass phrase for ca/private/client.pem:   
Enter Export Password:   
Verifying - Enter Export Password:   
  
生成证书链   
  
查看证书链   
subject=/C=CN/ST=BJ/O=zlex/OU=zlex/CN=www.zlex.org   
issuer=/C=CN/ST=BJ/O=zlex/OU=zlex/CN=ca.zlex.org   
  
subject=/C=CN/ST=BJ/O=zlex/OU=zlex/CN=ca.zlex.org   
issuer=/C=CN/ST=BJ/O=zlex/OU=zlex/CN=ca.zlex.org   
  
subject=/C=CN/ST=BJ/O=zlex/OU=zlex/CN=zlex   
issuer=/C=CN/ST=BJ/O=zlex/OU=zlex/CN=ca.zlex.org

来看一下这3套证书，如下两幅图所示：   
  
CA证书   
   
   
   
  
服务器证书   
   
   
   
  
客户证书   
   
   
   
  
证书链   
   
  
"ca.zlex.org"证书充当了CA根证书，"www.zlex.org"充当服务器端证书，"zlex"充当客户端证书   
  
使用keytool将其导入本地密钥库   
导入CA证书

**Shell代码  [收藏代码](javascript:void())**

1. keytool -import -v -trustcacerts -alias ca.zlex.org -file ca.crt -storepass 123456 -keystore ca.keystore

控制台输出

**引用**

所有者:CN=ca.zlex.org, OU=zlex, O=zlex, L=BJ, ST=BJ, C=CN   
签发人:CN=ca.zlex.org, OU=zlex, O=zlex, L=BJ, ST=BJ, C=CN   
序列号:989b27ef00e53a99   
有效期: Wed Jul 18 17:53:51 CST 2012 至Sat Jul 16 17:53:51 CST 2022   
证书指纹:   
         MD5:BA:14:1F:89:3A:1E:63:7B:20:AC:5A:50:FE:65:7E:16   
         SHA1:E0:A4:0E:6F:09:7E:01:27:C0:FC:62:26:1A:0C:C6:7B:BF:6A:18:B3   
         签名算法名称:SHA1withRSA   
         版本: 1   
信任这个认证？ [否]：  y   
认证已添加至keystore中   
[正在存储 ca.keystore]

导入服务器端证书

**Shell代码  [收藏代码](javascript:void())**

1. keytool -import -v -trustcacerts -alias www.zlex.org -file server.crt -storepass 123456 -keystore server.keystore

控制台输出

**引用**

所有者:CN=www.zlex.org, OU=zlex, O=zlex, ST=BJ, C=CN   
签发人:CN=ca.zlex.org, OU=zlex, O=zlex, L=BJ, ST=BJ, C=CN   
序列号:1   
有效期: Wed Jul 18 17:54:25 CST 2012 至Thu Jul 18 17:54:25 CST 2013   
证书指纹:   
         MD5:7E:5E:66:56:AF:E7:F5:72:0F:FC:95:85:97:07:4E:2A   
         SHA1:B1:E7:E8:AC:AB:C9:72:69:D8:E2:25:D5:16:A9:AF:C1:B7:4A:74:5D   
         签名算法名称:SHA1withRSA   
         版本: 3   
  
扩展:   
  
#1: ObjectId: 2.5.29.14 Criticality=false   
SubjectKeyIdentifier [   
KeyIdentifier [   
0000: A8 49 2F E2 2D 15 9F 42   BD 76 2B 20 D3 EB A5 EE  .I/.-..B.v+ ....   
0010: 31 CA E7 63                                        1..c   
]   
]   
  
#2: ObjectId: 2.5.29.19 Criticality=false   
BasicConstraints:[   
  CA:false   
  PathLen: undefined   
]   
  
#3: ObjectId: 2.5.29.35 Criticality=false   
AuthorityKeyIdentifier [   
[CN=ca.zlex.org, OU=zlex, O=zlex, L=BJ, ST=BJ, C=CN]   
SerialNumber: [    989b27ef 00e53a99]   
]   
  
#4: ObjectId: 2.16.840.1.113730.1.13 Criticality=false   
  
信任这个认证？ [否]：  y   
认证已添加至keystore中   
[正在存储 server.keystore]

导入客户端证书

**Shell代码  [收藏代码](javascript:void())**

1. keytool -import -v -trustcacerts -alias client -file client.crt -storepass 123456 -keystore client.keystore

以下是输出内容：

**引用**

所有者:CN=zlex, OU=zlex, O=zlex, ST=BJ, C=CN   
签发人:CN=ca.zlex.org, OU=zlex, O=zlex, L=BJ, ST=BJ, C=CN   
序列号:2   
有效期: Wed Jul 18 17:54:49 CST 2012 至Thu Jul 18 17:54:49 CST 2013   
证书指纹:   
         MD5:81:16:ED:92:9E:17:DB:E3:BE:DE:CD:8D:F8:E0:EE:C4   
         SHA1:C0:E0:42:81:79:70:4C:F8:44:D4:76:2D:C5:62:7C:67:B2:41:B3:AC   
         签名算法名称:SHA1withRSA   
         版本: 3   
  
扩展:   
  
#1: ObjectId: 2.5.29.14 Criticality=false   
SubjectKeyIdentifier [   
KeyIdentifier [   
0000: 0C 2C 25 86 C6 8D 04 88   F5 63 19 DC 09 B1 3C 5D  .,%......c....<]   
0010: 59 C9 72 1B                                        Y.r.   
]   
]   
  
#2: ObjectId: 2.5.29.19 Criticality=false   
BasicConstraints:[   
  CA:false   
  PathLen: undefined   
]   
  
#3: ObjectId: 2.5.29.35 Criticality=false   
AuthorityKeyIdentifier [   
[CN=ca.zlex.org, OU=zlex, O=zlex, L=BJ, ST=BJ, C=CN]   
SerialNumber: [    989b27ef 00e53a99]   
]   
  
#4: ObjectId: 2.16.840.1.113730.1.13 Criticality=false   
  
信任这个认证？ [否]：  y   
认证已添加至keystore中   
[正在存储 client.keystore]

PS 吊销证书：

**Shell代码  [收藏代码](javascript:void())**

1. echo 吊销客户端证书
2. openssl ca -revoke $certs\_path/client.crt -cert $certs\_path/ca.crt -keyfile $private\_path/ca.pem

**引用**

Using configuration from /etc/pki/tls/openssl.cnf   
Enter pass phrase for private/ca.pem:   
Revoking Certificate 02.   
Data Base Updated

生成证书吊销列表文件（CRL）   
执行命令如下：

**Shell代码  [收藏代码](javascript:void())**

1. openssl ca -gencrl -out ca.crl -config "$HOME/testca/conf/testca.conf"

-crldays和-crlhours参数,说明下一个吊销列表将在多少天后（或多少小时候）发布。   
  
可以用以下命令检查testca.crl的内容：

**Shell代码  [收藏代码](javascript:void())**

1. openssl crl -in testca.crl -text -noout

引用   
<http://blog.csdn.net/gothicane/articles/2865818.aspx>   
<http://www.5dlinux.com/article/7/2009/linux_35291.html>   
<http://www.tc.umn.edu/~brams006/selfsign_ubuntu.html>   
<http://www.tc.umn.edu/~brams006/selfsign.html>   
<http://zhouzhk.iteye.com/blog/136943>   
<http://bbs.cfan.com.cn/thread-743287-1-1.html>   
<http://www.iteye.com/problems/4072>   
<http://blog.csdn.net/jasonhwang/archive/2008/04/26/2329589.aspx>   
<http://blog.csdn.net/jasonhwang/archive/2008/04/29/2344768.aspx>

[**Java加密技术（十二）——\*.PFX(\*.p12)&个人信息交换文件**](http://snowolf.iteye.com/blog/735294)

**博客分类：**

* [Java／Security](http://snowolf.iteye.com/category/68576)

[pfx](http://www.iteye.com/blogs/tag/pfx)[keystore](http://www.iteye.com/blogs/tag/keystore)[p12](http://www.iteye.com/blogs/tag/p12)[keytool](http://www.iteye.com/blogs/tag/keytool)

今天来点实际工作中的硬通货！   
与计费系统打交道，少不了用到加密/解密实现。为了安全起见，通过非对称加密交换对称加密密钥更是不可或缺。那么需要通过什么载体传递非对称算法公钥/私钥信息？数字证书是公钥的载体，而密钥库可以包含公钥、私钥信息。   
**JKS**和**PKCS#12**都是比较常用的两种密钥库格式/标准。对于前者，搞Java开发，尤其是接触过HTTPS平台的朋友，并不陌生。**JKS**文件（通常为\*.jks或\*.keystore，扩展名无关）可以通过Java原生工具——KeyTool生成；而后者**PKCS#12**文件（通常为\*.p12或\*.pfx，意味个人信息交换文件），则是通过更为常用的OpenSSL工具产生。   
当然，这两者之间是可以通过导入/导出的方式进行转换的！当然，这种转换需要通过KeyTool工具进行！   
回归正题，计费同事遇到一个难题：合作方交给他们一个\*.pfx文件，需要他们从中提取密钥，然后进行加密交互。其实，通过Java直接操作密钥库文件（或个人信息交换文件）对于一般Java开发人员来说，这都是个冷门。不接触数字安全，根本不知所云。况且，Java原生的密钥库文件格式为JKS，如何操作\*.pfx文件？密钥库操作需要获知密钥库别名，\*.pfx别名是什么？！接下来就解决这些问题！http://snowolf.iteye.com/images/smiles/icon_biggrin.gif   
  
方案：

1. 通过keytool密钥库导入命令importkeystore，将密钥库格式由PKCS#12转换为JKS。
2. 检索新生成的密钥库文件，提取别名信息。
3. 由密钥库文件导出数字证书（这里将用到别名）。
4. 通过代码提取公钥/私钥、签名算法等

先看格式转换：

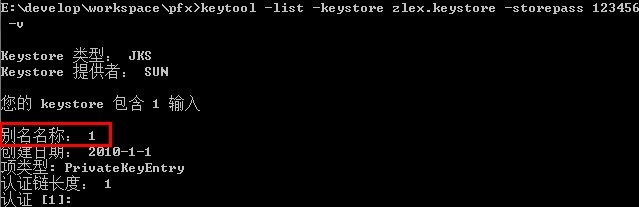
**Cmd代码  [收藏代码](javascript:void())**

1. echo 格式转换
2. keytool -importkeystore -v  -srckeystore zlex.pfx -srcstoretype pkcs12 -srcstorepass 123456 -destkeystore zlex.keystore -deststoretype jks -deststorepass 123456

**-importkeystore**导入密钥库，通过格式设定，我们可以将PKCS#12文件转换为JKS格式。   
**-v**显示详情   
**-srckeystore**源密钥库，这里是zlex.pfx   
**-srcstoretype**源密钥库格式，这里为pkcs12   
**-srcstorepass**源密钥库密码，这里为123456   
**-destkeystore**目标密钥库，这里为zlex.keystore   
**-deststoretype**目标密钥库格式，这里为jks，默认值也如此   
**-deststorepass**目标密钥库密码，这里为123456   
通过这个操作，我们能够获得所需的密钥库文件zlex.keystore。   
   
这时，我们已经获得了密钥库文件，只要确定对应的别名信息，就可以提取公钥/私钥，以及数字证书，进行加密交互了！http://snowolf.iteye.com/images/smiles/icon_biggrin.gif

**Cmd代码  [收藏代码](javascript:void())**

1. echo 查看证书
2. keytool -list -keystore zlex.keystore -storepass 123456 -v

**-list**列举密钥库   
**-keystore**密钥库，这里是zlex.keystore   
**-storepass**密钥库密码，这里是123456   
**-v**显示详情   
   
这里需要细致观察一下别名信息！！！就是红框中的数字1！！！   
  
  
现在，我们把证书导出！

**Cmd代码  [收藏代码](javascript:void())**

1. echo 导出证书
2. keytool -exportcert -alias 1 -keystore zlex.keystore -file zlex.crt -storepass 123456

**-exportcert**导出证书   
**-alias**别名，这里是1   
**-keystore**密钥库，这里是zlex.keystore   
**-file**证书文件，这里是zlex.crt   
**-storepass**密钥库密码，这里是123456   
http://dl.iteye.com/upload/attachment/291908/92e3cdbd-203d-30de-a685-fe0b1f7b494a.jpg   
现在证书也导出了，我们可以提取公钥/私钥，进行加密/解密，签名/验证操作了！当然，即便没有证书，我们也能够通过密钥库（JKS格式）文件获得证书，以及公钥/私钥、签名算法等。   
补充代码， 其实就是对[Java加密技术（八）](http://snowolf.iteye.com/blog/391931)的修改！http://snowolf.iteye.com/images/smiles/icon_biggrin.gif

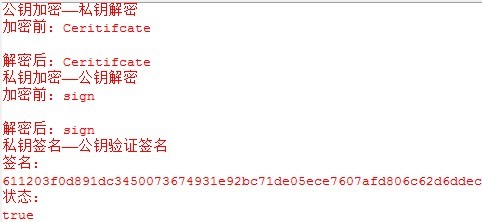
**Java代码  [收藏代码](javascript:void())**

1. /\*\*
2. \* 2010-8-11
3. \*/
5. **import** java.io.FileInputStream;
6. **import** java.security.KeyStore;
7. **import** java.security.PrivateKey;
8. **import** java.security.PublicKey;
9. **import** java.security.Signature;
10. **import** java.security.cert.Certificate;
11. **import** java.security.cert.CertificateFactory;
12. **import** java.security.cert.X509Certificate;
13. **import** java.util.Date;
15. **import** javax.crypto.Cipher;
17. /\*\*
18. \* 证书操作类
19. \*
20. \* @author <a href="mailto:zlex.dongliang@gmail.com">梁栋</a>
21. \* @since 1.0
22. \*/
23. **public** **class** CertificateCoder {
24. /\*\*
25. \* Java密钥库(Java Key Store，JKS)KEY\_STORE
26. \*/
27. **public** **static** **final** String KEY\_STORE = "JKS";
29. **public** **static** **final** String X509 = "X.509";
31. /\*\*
32. \* 由 KeyStore获得私钥
33. \*
34. \* @param keyStorePath
35. \* @param keyStorePassword
36. \* @param alias
37. \* @param aliasPassword
38. \* @return
39. \* @throws Exception
40. \*/
41. **private** **static** PrivateKey getPrivateKey(String keyStorePath,
42. String keyStorePassword, String alias, String aliasPassword)
43. **throws** Exception {
44. KeyStore ks = getKeyStore(keyStorePath, keyStorePassword);
45. PrivateKey key = (PrivateKey) ks.getKey(alias,
46. aliasPassword.toCharArray());
47. **return** key;
48. }
50. /\*\*
51. \* 由 Certificate获得公钥
52. \*
53. \* @param certificatePath
54. \* @return
55. \* @throws Exception
56. \*/
57. **private** **static** PublicKey getPublicKey(String certificatePath)
58. **throws** Exception {
59. Certificate certificate = getCertificate(certificatePath);
60. PublicKey key = certificate.getPublicKey();
61. **return** key;
62. }
64. /\*\*
65. \* 获得Certificate
66. \*
67. \* @param certificatePath
68. \* @return
69. \* @throws Exception
70. \*/
71. **private** **static** Certificate getCertificate(String certificatePath)
72. **throws** Exception {
73. CertificateFactory certificateFactory = CertificateFactory
74. .getInstance(X509);
75. FileInputStream in = **new** FileInputStream(certificatePath);
77. Certificate certificate = certificateFactory.generateCertificate(in);
78. in.close();
80. **return** certificate;
81. }
83. /\*\*
84. \* 获得Certificate
85. \*
86. \* @param keyStorePath
87. \* @param keyStorePassword
88. \* @param alias
89. \* @return
90. \* @throws Exception
91. \*/
92. **private** **static** Certificate getCertificate(String keyStorePath,
93. String keyStorePassword, String alias) **throws** Exception {
94. KeyStore ks = getKeyStore(keyStorePath, keyStorePassword);
95. Certificate certificate = ks.getCertificate(alias);
97. **return** certificate;
98. }
100. /\*\*
101. \* 获得KeyStore
102. \*
103. \* @param keyStorePath
104. \* @param password
105. \* @return
106. \* @throws Exception
107. \*/
108. **private** **static** KeyStore getKeyStore(String keyStorePath, String password)
109. **throws** Exception {
110. FileInputStream is = **new** FileInputStream(keyStorePath);
111. KeyStore ks = KeyStore.getInstance(KEY\_STORE);
112. ks.load(is, password.toCharArray());
113. is.close();
114. **return** ks;
115. }
117. /\*\*
118. \* 私钥加密
119. \*
120. \* @param data
121. \* @param keyStorePath
122. \* @param keyStorePassword
123. \* @param alias
124. \* @param aliasPassword
125. \* @return
126. \* @throws Exception
127. \*/
128. **public** **static** **byte**[] encryptByPrivateKey(**byte**[] data, String keyStorePath,
129. String keyStorePassword, String alias, String aliasPassword)
130. **throws** Exception {
131. // 取得私钥
132. PrivateKey privateKey = getPrivateKey(keyStorePath, keyStorePassword,
133. alias, aliasPassword);
135. // 对数据加密
136. Cipher cipher = Cipher.getInstance(privateKey.getAlgorithm());
137. cipher.init(Cipher.ENCRYPT\_MODE, privateKey);
139. **return** cipher.doFinal(data);
141. }
143. /\*\*
144. \* 私钥解密
145. \*
146. \* @param data
147. \* @param keyStorePath
148. \* @param alias
149. \* @param keyStorePassword
150. \* @param aliasPassword
151. \* @return
152. \* @throws Exception
153. \*/
154. **public** **static** **byte**[] decryptByPrivateKey(**byte**[] data, String keyStorePath,
155. String alias, String keyStorePassword, String aliasPassword)
156. **throws** Exception {
157. // 取得私钥
158. PrivateKey privateKey = getPrivateKey(keyStorePath, keyStorePassword,
159. alias, aliasPassword);
161. // 对数据加密
162. Cipher cipher = Cipher.getInstance(privateKey.getAlgorithm());
163. cipher.init(Cipher.DECRYPT\_MODE, privateKey);
165. **return** cipher.doFinal(data);
167. }
169. /\*\*
170. \* 公钥加密
171. \*
172. \* @param data
173. \* @param certificatePath
174. \* @return
175. \* @throws Exception
176. \*/
177. **public** **static** **byte**[] encryptByPublicKey(**byte**[] data, String certificatePath)
178. **throws** Exception {
180. // 取得公钥
181. PublicKey publicKey = getPublicKey(certificatePath);
182. // 对数据加密
183. Cipher cipher = Cipher.getInstance(publicKey.getAlgorithm());
184. cipher.init(Cipher.ENCRYPT\_MODE, publicKey);
186. **return** cipher.doFinal(data);
188. }
190. /\*\*
191. \* 公钥解密
192. \*
193. \* @param data
194. \* @param certificatePath
195. \* @return
196. \* @throws Exception
197. \*/
198. **public** **static** **byte**[] decryptByPublicKey(**byte**[] data, String certificatePath)
199. **throws** Exception {
200. // 取得公钥
201. PublicKey publicKey = getPublicKey(certificatePath);
203. // 对数据加密
204. Cipher cipher = Cipher.getInstance(publicKey.getAlgorithm());
205. cipher.init(Cipher.DECRYPT\_MODE, publicKey);
207. **return** cipher.doFinal(data);
209. }
211. /\*\*
212. \* 验证Certificate
213. \*
214. \* @param certificatePath
215. \* @return
216. \*/
217. **public** **static** **boolean** verifyCertificate(String certificatePath) {
218. **return** verifyCertificate(**new** Date(), certificatePath);
219. }
221. /\*\*
222. \* 验证Certificate是否过期或无效
223. \*
224. \* @param date
225. \* @param certificatePath
226. \* @return
227. \*/
228. **public** **static** **boolean** verifyCertificate(Date date, String certificatePath) {
229. **boolean** status = **true**;
230. **try** {
231. // 取得证书
232. Certificate certificate = getCertificate(certificatePath);
233. // 验证证书是否过期或无效
234. status = verifyCertificate(date, certificate);
235. } **catch** (Exception e) {
236. status = **false**;
237. }
238. **return** status;
239. }
241. /\*\*
242. \* 验证证书是否过期或无效
243. \*
244. \* @param date
245. \* @param certificate
246. \* @return
247. \*/
248. **private** **static** **boolean** verifyCertificate(Date date, Certificate certificate) {
249. **boolean** status = **true**;
250. **try** {
251. X509Certificate x509Certificate = (X509Certificate) certificate;
252. x509Certificate.checkValidity(date);
253. } **catch** (Exception e) {
254. status = **false**;
255. }
256. **return** status;
257. }
259. /\*\*
260. \* 签名
261. \*
262. \* @param keyStorePath
263. \* @param alias
264. \* @param keyStorePassword
265. \* @param aliasPassword
266. \* @return
267. \* @throws Exception
268. \*/
269. **public** **static** **byte**[] sign(**byte**[] sign, String keyStorePath, String alias,
270. String keyStorePassword, String aliasPassword) **throws** Exception {
271. // 获得证书
272. X509Certificate x509Certificate = (X509Certificate) getCertificate(
273. keyStorePath, keyStorePassword, alias);
275. // 取得私钥
276. PrivateKey privateKey = getPrivateKey(keyStorePath, keyStorePassword,
277. alias, aliasPassword);
279. // 构建签名
280. Signature signature = Signature.getInstance(x509Certificate
281. .getSigAlgName());
282. signature.initSign(privateKey);
283. signature.update(sign);
284. **return** signature.sign();
285. }
287. /\*\*
288. \* 验证签名
289. \*
290. \* @param data
291. \* @param sign
292. \* @param certificatePath
293. \* @return
294. \* @throws Exception
295. \*/
296. **public** **static** **boolean** verify(**byte**[] data, **byte**[] sign,
297. String certificatePath) **throws** Exception {
298. // 获得证书
299. X509Certificate x509Certificate = (X509Certificate) getCertificate(certificatePath);
300. // 获得公钥
301. PublicKey publicKey = x509Certificate.getPublicKey();
302. // 构建签名
303. Signature signature = Signature.getInstance(x509Certificate
304. .getSigAlgName());
305. signature.initVerify(publicKey);
306. signature.update(data);
308. **return** signature.verify(sign);
310. }
312. /\*\*
313. \* 验证Certificate
314. \*
315. \* @param keyStorePath
316. \* @param keyStorePassword
317. \* @param alias
318. \* @return
319. \*/
320. **public** **static** **boolean** verifyCertificate(Date date, String keyStorePath,
321. String keyStorePassword, String alias) {
322. **boolean** status = **true**;
323. **try** {
324. Certificate certificate = getCertificate(keyStorePath,
325. keyStorePassword, alias);
326. status = verifyCertificate(date, certificate);
327. } **catch** (Exception e) {
328. status = **false**;
329. }
330. **return** status;
331. }
333. /\*\*
334. \* 验证Certificate
335. \*
336. \* @param keyStorePath
337. \* @param keyStorePassword
338. \* @param alias
339. \* @return
340. \*/
341. **public** **static** **boolean** verifyCertificate(String keyStorePath,
342. String keyStorePassword, String alias) {
343. **return** verifyCertificate(**new** Date(), keyStorePath, keyStorePassword,
344. alias);
345. }
346. }

相信上述代码已经帮朋友们解决了相当多的问题！http://snowolf.iteye.com/images/smiles/icon_biggrin.gif   
给出测试类：

**Java代码  [收藏代码](javascript:void())**

1. **import** **static** org.junit.Assert.\*;
3. **import** java.util.Date;
5. **import** org.apache.commons.codec.binary.Hex;
6. **import** org.junit.Test;
8. /\*\*
9. \* 证书操作验证类
10. \*
11. \* @author <a href="mailto:zlex.dongliang@gmail.com">梁栋</a>
12. \* @version 1.0
13. \* @since 1.0
14. \*/
15. **public** **class** CertificateCoderTest {
16. **private** String certificatePath = "zlex.crt";
17. **private** String keyStorePath = "zlex.keystore";
18. **private** String keyStorePassword = "123456";
19. **private** String aliasPassword = "123456";
20. **private** String alias = "1";
22. @Test
23. **public** **void** test() **throws** Exception {
24. System.err.println("公钥加密——私钥解密");
25. String inputStr = "Ceritifcate";
26. **byte**[] data = inputStr.getBytes();
28. **byte**[] encrypt = CertificateCoder.encryptByPublicKey(data,
29. certificatePath);
31. **byte**[] decrypt = CertificateCoder.decryptByPrivateKey(encrypt,
32. keyStorePath, alias, keyStorePassword, aliasPassword);
33. String outputStr = **new** String(decrypt);
35. System.err.println("加密前: " + inputStr + "\n\r" + "解密后: " + outputStr);
37. // 验证数据一致
38. assertArrayEquals(data, decrypt);
40. // 验证证书有效
41. assertTrue(CertificateCoder.verifyCertificate(certificatePath));
43. }
45. @Test
46. **public** **void** testSign() **throws** Exception {
47. System.err.println("私钥加密——公钥解密");
49. String inputStr = "sign";
50. **byte**[] data = inputStr.getBytes();
52. **byte**[] encodedData = CertificateCoder.encryptByPrivateKey(data,
53. keyStorePath, keyStorePassword, alias, aliasPassword);
55. **byte**[] decodedData = CertificateCoder.decryptByPublicKey(encodedData,
56. certificatePath);
58. String outputStr = **new** String(decodedData);
59. System.err.println("加密前: " + inputStr + "\n\r" + "解密后: " + outputStr);
60. assertEquals(inputStr, outputStr);
62. System.err.println("私钥签名——公钥验证签名");
63. // 产生签名
64. **byte**[] sign = CertificateCoder.sign(encodedData, keyStorePath, alias,
65. keyStorePassword, aliasPassword);
66. System.err.println("签名:\r" + Hex.encodeHexString(sign));
68. // 验证签名
69. **boolean** status = CertificateCoder.verify(encodedData, sign,
70. certificatePath);
71. System.err.println("状态:\r" + status);
72. assertTrue(status);
73. }
75. @Test
76. **public** **void** testVerify() **throws** Exception {
77. System.err.println("密钥库证书有效期验证");
78. **boolean** status = CertificateCoder.verifyCertificate(**new** Date(),
79. keyStorePath, keyStorePassword, alias);
80. System.err.println("证书状态:\r" + status);
81. assertTrue(status);
82. }
83. }

第一个测试方法，用于提取公钥/私钥进行加密/解密操作。   
第二个测试方法，用于提取签名算法进行签名/验证操作。   
第三个测试方法，用于测试密钥库该别名对应的证书，当前日期下，是否有效。   
   
  
  
OK，任务完成，密钥成功提取，剩下的都是代码基本功了！http://snowolf.iteye.com/images/smiles/icon_biggrin.gif