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：

```
/**
 * BASE64 解密  http://www.bt285.cn http://www.5a520.cn
 *
 * @param key
 * @return
 * @throws Exception
 */
```

```
public static byte[] decryptBASE64(String key) throws Exception {
    return (new BASE64Decoder()).decodeBuffer(key);
}

/**
 * BASE64 加密
 *
 * @param key
 * @return
 * @throws Exception
 */
public static String encryptBASE64(byte[] key) throws Exception {
    return (new BASE64Encoder()).encodeBuffer(key);
}
```

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：

```
/**
 * MD5 加密 http://www.bt285.cn http://www.5a520.cn
 *
 * @param data
 * @return
 * @throws Exception
 */
public static byte[] encryptMD5(byte[] data) throws Exception {

    MessageDigest md5 = MessageDigest.getInstance(KEY_MD5);
    md5.update(data);

    return md5.digest();

}
```

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， 但是SHA仍然是公认的安全加密算法，较之MD5 更为安全。

通过 java 代码实现如下：

```
/**
     * SHA加密 http://www.5a520.cn http://www.bt285.cn
     *
     * @param data
     * @return
     * @throws Exception
     */
    public static byte[] encryptSHA(byte[] data) throws Exception {

        MessageDigest sha = MessageDigest.getInstance(KEY_SHA);
        sha.update(data);

        return sha.digest();

    }
}
```

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：

```
/**
 * 初始化HMAC密钥  http://www.guihua.org  http://www.feng123.com
 *
 * @return
 * @throws Exception
 */
public static String initMacKey() throws Exception {
```

```
    KeyGenerator keyGenerator = KeyGenerator.getInstance(KEY_MAC);

    SecretKey secretKey = keyGenerator.generateKey();
    return encryptBASE64(secretKey.getEncoded());
}

/**
 * HMAC加密
 *
 * @param data
 * @param key
 * @return
 * @throws Exception
 */
public static byte[] encryptHMAC(byte[] data, String key) throws Exception {

    SecretKey secretKey = new SecretKeySpec(decryptBASE64(key), KEY_MAC);
    Mac mac = Mac.getInstance(secretKey.getAlgorithm());
    mac.init(secretKey);

    return mac.doFinal(data);

}
```

给出一个完整类，如下：

```
import java.security.MessageDigest;

 import javax.crypto.KeyGenerator;
 import javax.crypto.Mac;
 import javax.crypto.SecretKey;

 import sun.misc.BASE64Decoder;
 import sun.misc.BASE64Encoder;

/**
 * 基础加密组件  http://www.bt285.cn http://www.feng123.com
 *
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
```

```
public abstract class Coder {
    public static final String KEY_SHA = "SHA";
    public static final String KEY_MD5 = "MD5";

    /**
     * MAC算法可选以下多种算法
     *
     * <pre>
     * HmacMD5
     * HmacSHA1
     * HmacSHA256
     * HmacSHA384
     * HmacSHA512
     * </pre>
     */
    public static final String KEY_MAC = "HmacMD5";

    /**
     * BASE64 解密
     *
     * @param key
     * @return
     * @throws Exception
     */
    public static byte[] decryptBASE64(String key) throws Exception {
        return (new BASE64Decoder()).decodeBuffer(key);
    }

    /**
     * BASE64 加密
     *
     * @param key
     * @return
     * @throws Exception
     */
    public static String encryptBASE64(byte[] key) throws Exception {
        return (new BASE64Encoder()).encodeBuffer(key);
    }

    /**
     * MD5 加密
     *
     * @param data
     * @return
```

```
     * @throws Exception
     */
    public static byte[] encryptMD5(byte[] data) throws Exception {

        MessageDigest md5 = MessageDigest.getInstance(KEY_MD5);
        md5.update(data);

        return md5.digest();

    }

    /**
     * SHA加密
     *
     * @param data
     * @return
     * @throws Exception
     */
    public static byte[] encryptSHA(byte[] data) throws Exception {

        MessageDigest sha = MessageDigest.getInstance(KEY_SHA);
        sha.update(data);

        return sha.digest();

    }

    /**
     * 初始化HMAC密钥
     *
     * @return
     * @throws Exception
     */
    public static String initMacKey() throws Exception {
        KeyGenerator keyGenerator = KeyGenerator.getInstance(KEY_MAC);

        SecretKey secretKey = keyGenerator.generateKey();
        return encryptBASE64(secretKey.getEncoded());
    }

    /**
     * HMAC加密
     *
     * @param data
```

```
     * @param key
     * @return
     * @throws Exception
     */

    public static byte[] encryptHMAC(byte[] data, String key) throws Exceptio
n {

        SecretKey secretKey = new SecretKeySpec(decryptBASE64(key), KEY_MAC);

        Mac mac = Mac.getInstance(secretKey.getAlgorithm());
        mac.init(secretKey);

        return mac.doFinal(data);

    }
}
```

再给出一个测试类：

```
import static org.junit.Assert.*;

import org.junit.Test;

/**
 *
 * @author 梁栋  http://www.feng123.com
 * @version 1.0
 * @since 1.0
 */
public class CoderTest {

    @Test
    public void test() throws Exception {
        String inputStr = "简单加密";
        System.err.println("原文:\n" + inputStr);

        byte[] inputData = inputStr.getBytes();
        String code = Coder.encryptBASE64(inputData);

        System.err.println("BASE64 加密后:\n" + code);

        byte[] output = Coder.decryptBASE64(code);
```

```
        String outputStr = new String(output);

        System.err.println("BASE64 解密后:\n" + outputStr);

        // 验证 BASE64 加密解密一致性
        assertEquals(inputStr, outputStr);

        // 验证 MD5 对于同一内容加密是否一致
        assertArrayEquals(Coder.encryptMD5(inputData), Coder
                .encryptMD5(inputData));

        // 验证 SHA 对于同一内容加密是否一致
        assertArrayEquals(Coder.encryptSHA(inputData), Coder
                .encryptSHA(inputData));

        String key = Coder.initMacKey();
        System.err.println("Mac 密钥:\n" + key);

        // 验证 HMAC 对于同一内容，同一密钥加密是否一致
        assertArrayEquals(Coder.encryptHMAC(inputData, key),
Coder.encryptHMAC(
                inputData, key));

        BigInteger md5 = new BigInteger(Coder.encryptMD5(inputData));
        System.err.println("MD5:\n" + md5.toString(16));

        BigInteger sha = new BigInteger(Coder.encryptSHA(inputData));
        System.err.println("SHA:\n" + sha.toString(32));

        BigInteger mac = new BigInteger(Coder.encryptHMAC(inputData,
inputStr));
        System.err.println("HMAC:\n" + mac.toString(16));
    }
```

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

```
import java.security.Key;
 import java.security.SecureRandom;

 import javax.crypto.Cipher;
 import javax.crypto.KeyGenerator;
 import javax.crypto.SecretKey;
 import javax.crypto.SecretKeyFactory;
 import javax.crypto.spec.DESKeySpec;


/**
 * DES安全编码组件  author by http://www.bt285.cn http://www.5a520.cn
 *
 * <pre>
 * 支持 DES、DESede(TripleDES,就是 3DES)、AES、Blowfish、RC2、RC4(ARCFOUR)
 * DES                    key size must be equal to 56
 * DESede(TripleDES)      key size must be equal to 112 or 168
 * AES                    key size must be equal to 128, 192 or 256,but 192 and
 256 bits may not be available
 * Blowfish               key size must be multiple of 8, and can only range fr
om 32 to 448 (inclusive)
 * RC2                    key size must be between 40 and 1024 bits
 * RC4(ARCFOUR)           key size must be between 40 and 1024 bits
 * 具体内容 需要关注 JDK Document http://…
/docs/technotes/guides/security/SunProviders.html
 * </pre>
 *
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public abstract class DESCoder extends Coder {
    /**
     * ALGORITHM 算法 <br>
     * 可替换为以下任意一种算法，同时key值的size相应改变。
     *
     * <pre>
     * DES                    key size must be equal to 56
     * DESede(TripleDES)      key size must be equal to 112 or 168
     * AES                    key size must be equal to 128, 192 or 256,but 192
 and 256 bits may not be available
```

```
     * Blowfish             key size must be multiple of 8, and can only rang
e from 32 to 448 (inclusive)
     * RC2                  key size must be between 40 and 1024 bits
     * RC4(ARCFOUR)         key size must be between 40 and 1024 bits
     * </pre>
     *
     * 在Key toKey(byte[] key)方法中使用下述代码
     * <code>SecretKey secretKey = new SecretKeySpec(key, ALGORITHM);</code>
替换
     * <code>
     * DESKeySpec dks = new DESKeySpec(key);
     * SecretKeyFactory keyFactory = SecretKeyFactory.getInstance(ALGORITHM);

     * SecretKey secretKey = keyFactory.generateSecret(dks);
     * </code>
     */
    public static final String ALGORITHM = "DES";

    /**
     * 转换密钥<br>
     *
     * @param key
     * @return
     * @throws Exception
     */
    private static Key toKey(byte[] key) throws Exception {
        DESKeySpec dks = new DESKeySpec(key);
        SecretKeyFactory keyFactory = SecretKeyFactory.getInstance(ALGORITHM)
;

        SecretKey secretKey = keyFactory.generateSecret(dks);

        // 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
        // SecretKey secretKey = new SecretKeySpec(key, ALGORITHM);

        return secretKey;
    }

    /**
     * 解密
     *
     * @param data
     * @param key
     * @return
```

```
     * @throws Exception
     */

    public static byte[] decrypt(byte[] data, String key) throws Exception {

        Key k = toKey(decryptBASE64(key));

        Cipher cipher = Cipher.getInstance(ALGORITHM);
        cipher.init(Cipher.DECRYPT_MODE, k);

        return cipher.doFinal(data);
    }

    /**
     * 加密
     *
     * @param data
     * @param key
     * @return
     * @throws Exception
     */

    public static byte[] encrypt(byte[] data, String key) throws Exception {

        Key k = toKey(decryptBASE64(key));
        Cipher cipher = Cipher.getInstance(ALGORITHM);
        cipher.init(Cipher.ENCRYPT_MODE, k);

        return cipher.doFinal(data);
    }

    /**
     * 生成密钥
     *
     * @return
     * @throws Exception
     */
    public static String initKey() throws Exception {
        return initKey(null);
    }

    /**
     * 生成密钥
     *
```

```
     * @param seed
     * @return
     * @throws Exception
     */
    public static String initKey(String seed) throws Exception {
        SecureRandom secureRandom = null;

        if (seed != null) {
            secureRandom = new SecureRandom(decryptBASE64(seed));
        } else {
            secureRandom = new SecureRandom();
        }

        KeyGenerator kg = KeyGenerator.getInstance(ALGORITHM);
        kg.init(secureRandom);

        SecretKey secretKey = kg.generateKey();

        return encryptBASE64(secretKey.getEncoded());
    }
}
```

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：

```
import static org.junit.Assert.*;

 import org.junit.Test;

/**
 *
 * @author by http://www.bt285.cn http://www.5a520.cn
 * @version 1.0
 * @since 1.0
 */
public class DESCoderTest {

    @Test
    public void test() throws Exception {
        String inputStr = "DES";
        String key = DESCoder.initKey();
```

```
        System.err.println("原文:\t" + inputStr);

        System.err.println("密钥:\t" + key);

        byte[] inputData = inputStr.getBytes();
        inputData = DESCoder.encrypt(inputData, key);

        System.err.println("加密
后:\t" + DESCoder.encryptBASE64(inputData));

        byte[] outputData = DESCoder.decrypt(inputData, key);
        String outputStr = new String(outputData);

        System.err.println("解密后:\t" + outputStr);

        assertEquals(inputStr, outputStr);
    }
}
```

得到的输出内容如下：

原文： DES

密钥： f3wEtRrV6q0=

加密后： C6qe9oNIzRY=

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。

/**

```
 * DES          key size must be equal to 56
 * DESede(TripleDES) key size must be equal to 112 or 168
 * AES          key size must be equal to 128, 192 or 256,but 192 and 256 bits
may not be available
 * Blowfish     key size must be multiple of 8, and can only range from 32 to
```

```
448 (inclusive)
 * RC2          key size must be between 40 and 1024 bits
 * RC4(ARCFOUR) key size must be between 40 and 1024 bits
 **/
```

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：

```
import java.security.Key;
import java.util.Random;

import javax.crypto.Cipher;
import javax.crypto.SecretKey;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.PBEKeySpec;
import javax.crypto.spec.PBEParameterSpec;

/**
 * PBE安全编码组件 http://www.bt285.cn   http://www.5a520.cn
 *
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public abstract class PBECoder extends Coder {
    /**
     * 支持以下任意一种算法
     *
     * <pre>
     * PBEWithMD5AndDES
     * PBEWithMD5AndTripleDES
     * PBEWithSHA1AndDESede
     * PBEWithSHA1AndRC2_40
     * </pre>
     */
```

```
    public static final String ALGORITHM = "PBEWITHMD5andDES";

    /**
     * 盐初始化
     *
     * @return
     * @throws Exception
     */
    public static byte[] initSalt() throws Exception {
        byte[] salt = new byte[8];
        Random random = new Random();
        random.nextBytes(salt);
        return salt;
    }

    /**
     * 转换密钥<br>
     *
     * @param password
     * @return
     * @throws Exception
     */
    private static Key toKey(String password) throws Exception {
        PBEKeySpec keySpec = new PBEKeySpec(password.toCharArray());
        SecretKeyFactory keyFactory = SecretKeyFactory.getInstance(ALGORITHM)
;
        SecretKey secretKey = keyFactory.generateSecret(keySpec);

        return secretKey;
    }

    /**
     * 加密
     *
     * @param data
     *            数据
     * @param password
     *            密码
     * @param salt
     *            盐
     * @return
     * @throws Exception
     */
    public static byte[] encrypt(byte[] data, String password, byte[] salt)
```

```
            throws Exception {

        Key key = toKey(password);

        PBEParameterSpec paramSpec = new PBEParameterSpec(salt, 100);
        Cipher cipher = Cipher.getInstance(ALGORITHM);
        cipher.init(Cipher.ENCRYPT_MODE, key, paramSpec);

        return cipher.doFinal(data);

    }

    /**
     * 解密
     *
     * @param data
     *            数据
     * @param password
     *            密码
     * @param salt
     *            盐
     * @return
     * @throws Exception
     */
    public static byte[] decrypt(byte[] data, String password, byte[] salt)

            throws Exception {

        Key key = toKey(password);

        PBEParameterSpec paramSpec = new PBEParameterSpec(salt, 100);
        Cipher cipher = Cipher.getInstance(ALGORITHM);
        cipher.init(Cipher.DECRYPT_MODE, key, paramSpec);

        return cipher.doFinal(data);

    }
}
```

再给出一个测试类：

```
import static org.junit.Assert.*;
```

```
import org.junit.Test;

/**
 *
 * @author 梁栋  http://www.5a520.cn http://www.feng123.com
 * @version 1.0
 * @since 1.0
 */
public class PBECoderTest {

    @Test
    public void test() throws Exception {
        String inputStr = "abc";
        System.err.println("原文: " + inputStr);
        byte[] input = inputStr.getBytes();

        String pwd = "efg";
        System.err.println("密码: " + pwd);

        byte[] salt = PBECoder.initSalt();

        byte[] data = PBECoder.encrypt(input, pwd, salt);

        System.err.println("加密后: " + PBECoder.encryptBASE64(data));

        byte[] output = PBECoder.decrypt(data, pwd, salt);
        String outputStr = new String(output);

        System.err.println("解密后: " + outputStr);
        assertEquals(inputStr, outputStr);
    }

}
```

控制台输出：

原文： abc

密码： efg

加密后： iCZOuRtaAhE=

解密后： abc

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。

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。

通过 java 代码实现如下：

```
import java.security.Key;
import java.security.KeyFactory;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.Signature;
import java.security.interfaces.RSAPrivateKey;
import java.security.interfaces.RSAPublicKey;
import java.security.spec.PKCS8EncodedKeySpec;
import java.security.spec.X509EncodedKeySpec;

import java.util.HashMap;
import java.util.Map;

import javax.crypto.Cipher;

/**
 * RSA安全编码组件
 *
 * @author 梁栋  http://www.bt285.cn http://www.5a520.cn
 * @version 1.0
```

```
 * @since 1.0
 */
public abstract class RSACoder extends Coder {
    public static final String KEY_ALGORITHM = "RSA";
    public static final String SIGNATURE_ALGORITHM = "MD5withRSA";

    private static final String PUBLIC_KEY = "RSAPublicKey";
    private static final String PRIVATE_KEY = "RSAPrivateKey";

    /**
     * 用私钥对信息生成数字签名
     *
     * @param data
     *            加密数据
     * @param privateKey
     *            私钥
     *
     * @return
     * @throws Exception
     */

    public static String sign(byte[] data, String privateKey) throws Exception {
        // 解密由base64 编码的私钥
        byte[] keyBytes = decryptBASE64(privateKey);

        // 构造PKCS8EncodedKeySpec对象
        PKCS8EncodedKeySpec pkcs8KeySpec = new PKCS8EncodedKeySpec(keyBytes);

        // KEY_ALGORITHM 指定的加密算法
        KeyFactory keyFactory = KeyFactory.getInstance(KEY_ALGORITHM);

        // 取私钥匙对象
        PrivateKey priKey = keyFactory.generatePrivate(pkcs8KeySpec);

        // 用私钥对信息生成数字签名
        Signature signature = Signature.getInstance(SIGNATURE_ALGORITHM);
        signature.initSign(priKey);
        signature.update(data);

        return encryptBASE64(signature.sign());
    }

    /**
     * 校验数字签名
```

```
     *
     * @param data
     *            加密数据
     * @param publicKey
     *            公钥
     * @param sign
     *            数字签名
     *
     * @return 校验成功返回true 失败返回false
     * @throws Exception
     *
     */
    public static boolean verify(byte[] data, String publicKey, String sign)
            throws Exception {

        // 解密由base64 编码的公钥
        byte[] keyBytes = decryptBASE64(publicKey);

        // 构造X509EncodedKeySpec对象
        X509EncodedKeySpec keySpec = new X509EncodedKeySpec(keyBytes);

        // KEY_ALGORITHM 指定的加密算法
        KeyFactory keyFactory = KeyFactory.getInstance(KEY_ALGORITHM);

        // 取公钥匙对象
        PublicKey pubKey = keyFactory.generatePublic(keySpec);

        Signature signature = Signature.getInstance(SIGNATURE_ALGORITHM);
        signature.initVerify(pubKey);
        signature.update(data);

        // 验证签名是否正常
        return signature.verify(decryptBASE64(sign));
    }

    /**
     * 解密<br>
     * 用私钥解密  http://www.5a520.cn http://www.feng123.com
     *
     * @param data
     * @param key
     * @return
     * @throws Exception
     */
```

```
    public static byte[] decryptByPrivateKey(byte[] data, String key)
            throws Exception {
        // 对密钥解密
        byte[] keyBytes = decryptBASE64(key);

        // 取得私钥
        PKCS8EncodedKeySpec pkcs8KeySpec = new PKCS8EncodedKeySpec(keyBytes);
        KeyFactory keyFactory = KeyFactory.getInstance(KEY_ALGORITHM);
        Key privateKey = keyFactory.generatePrivate(pkcs8KeySpec);

        // 对数据解密
        Cipher cipher = Cipher.getInstance(keyFactory.getAlgorithm());
        cipher.init(Cipher.DECRYPT_MODE, privateKey);

        return cipher.doFinal(data);
    }

    /**
     * 解密<br>
     * 用私钥解密
     *
     * @param data
     * @param key
     * @return
     * @throws Exception
     */
    public static byte[] decryptByPublicKey(byte[] data, String key)
            throws Exception {
        // 对密钥解密
        byte[] keyBytes = decryptBASE64(key);

        // 取得公钥
        X509EncodedKeySpec x509KeySpec = new X509EncodedKeySpec(keyBytes);
        KeyFactory keyFactory = KeyFactory.getInstance(KEY_ALGORITHM);
        Key publicKey = keyFactory.generatePublic(x509KeySpec);

        // 对数据解密
        Cipher cipher = Cipher.getInstance(keyFactory.getAlgorithm());
        cipher.init(Cipher.DECRYPT_MODE, publicKey);

        return cipher.doFinal(data);
    }

    /**
```

```
     * 加密<br>
     * 用公钥加密
     *
     * @param data
     * @param key
     * @return
     * @throws Exception
     */
    public static byte[] encryptByPublicKey(byte[] data, String key)
            throws Exception {
        // 对公钥解密
        byte[] keyBytes = decryptBASE64(key);

        // 取得公钥
        X509EncodedKeySpec x509KeySpec = new X509EncodedKeySpec(keyBytes);
        KeyFactory keyFactory = KeyFactory.getInstance(KEY_ALGORITHM);
        Key publicKey = keyFactory.generatePublic(x509KeySpec);

        // 对数据加密
        Cipher cipher = Cipher.getInstance(keyFactory.getAlgorithm());
        cipher.init(Cipher.ENCRYPT_MODE, publicKey);

        return cipher.doFinal(data);
    }

    /**
     * 加密<br>
     * 用私钥加密
     *
     * @param data
     * @param key
     * @return
     * @throws Exception
     */
    public static byte[] encryptByPrivateKey(byte[] data, String key)
            throws Exception {
        // 对密钥解密
        byte[] keyBytes = decryptBASE64(key);

        // 取得私钥
        PKCS8EncodedKeySpec pkcs8KeySpec = new PKCS8EncodedKeySpec(keyBytes);
        KeyFactory keyFactory = KeyFactory.getInstance(KEY_ALGORITHM);
        Key privateKey = keyFactory.generatePrivate(pkcs8KeySpec);

```

```
        // 对数据加密
        Cipher cipher = Cipher.getInstance(keyFactory.getAlgorithm());
        cipher.init(Cipher.ENCRYPT_MODE, privateKey);

        return cipher.doFinal(data);
    }

    /**
     * 取得私钥
     *
     * @param keyMap
     * @return
     * @throws Exception
     */
    public static String getPrivateKey(Map<String, Object> keyMap)
            throws Exception {
        Key key = (Key) keyMap.get(PRIVATE_KEY);

        return encryptBASE64(key.getEncoded());
    }

    /**
     * 取得公钥
     *
     * @param keyMap
     * @return
     * @throws Exception
     */
    public static String getPublicKey(Map<String, Object> keyMap)
            throws Exception {
        Key key = (Key) keyMap.get(PUBLIC_KEY);

        return encryptBASE64(key.getEncoded());
    }

    /**
     * 初始化密钥
     *
     * @return
     * @throws Exception
     */
    public static Map<String, Object> initKey() throws Exception {
        KeyPairGenerator keyPairGen = KeyPairGenerator
                .getInstance(KEY_ALGORITHM);
```

```
        keyPairGen.initialize(1024);

        KeyPair keyPair = keyPairGen.generateKeyPair();

        // 公钥
        RSAPublicKey publicKey = (RSAPublicKey) keyPair.getPublic();

        // 私钥
        RSAPrivateKey privateKey = (RSAPrivateKey) keyPair.getPrivate();

        Map<String, Object> keyMap = new HashMap<String, Object>(2);

        keyMap.put(PUBLIC_KEY, publicKey);
        keyMap.put(PRIVATE_KEY, privateKey);
        return keyMap;
    }
}
```

再给出一个测试类：

```
import static org.junit.Assert.*;

 import org.junit.Before;
 import org.junit.Test;

 import java.util.Map;

/**
 *
 * @author 梁栋  http://www.bt285.cn  http://www.guihua.org
 * @version 1.0
 * @since 1.0
 */
public class RSACoderTest {
    private String publicKey;
    private String privateKey;

    @Before
     public void setUp() throws Exception {
        Map<String, Object> keyMap = RSACoder.initKey();

        publicKey = RSACoder.getPublicKey(keyMap);
        privateKey = RSACoder.getPrivateKey(keyMap);
```

```
        System.err.println("公钥: \n\r" + publicKey);
        System.err.println("私钥:  \n\r" + privateKey);
    }

    @Test
    public void test() throws Exception {
        System.err.println("公钥加密——私钥解密");
        String inputStr = "abc";
        byte[] data = inputStr.getBytes();

        byte[] encodedData = RSACoder.encryptByPublicKey(data, publicKey);

        byte[] decodedData = RSACoder.decryptByPrivateKey(encodedData,
                privateKey);

        String outputStr = new String(decodedData);
        System.err.println("加密前: " + inputStr + "\n\r" + "解密
后: " + outputStr);
        assertEquals(inputStr, outputStr);

    }

    @Test
    public void testSign() throws Exception {
        System.err.println("私钥加密——公钥解密");
        String inputStr = "sign";
        byte[] data = inputStr.getBytes();

        byte[] encodedData = RSACoder.encryptByPrivateKey(data, privateKey);


        byte[] decodedData = RSACoder
                .decryptByPublicKey(encodedData, publicKey);

        String outputStr = new String(decodedData);
        System.err.println("加密前: " + inputStr + "\n\r" + "解密
后: " + outputStr);
        assertEquals(inputStr, outputStr);

        System.err.println("私钥签名——公钥验证签名");
        // 产生签名
        String sign = RSACoder.sign(encodedData, privateKey);
        System.err.println("签名:\r" + sign);

```

```
        // 验证签名
        boolean status = RSACoder.verify(encodedData, publicKey, sign);
        System.err.println("状态:\r" + status);
        assertTrue(status);

    }

}
```

Console 代码：

```
公钥:

MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCYU/+IO+z1aBl5X6DUUOHQ7FZpmBSDbKTtx89J
EcB64jFCkunELT8qiKly7fzEqDO3g8ALlu5XvX+bBqHFy7YPJJPOekE2X3wjUnh2NxlqpH3/B/xm
1ZdSlCwDIkbijhBVDjA/bu5BObhZqQmDwIxlQInL9oVz+o6FbAZCyHBd7wIDAQAB

私钥:

MIICdgIBADANBgkqhkiG9w0BAQEFAASCAmAwggJcAgEAAoGBAJhT/4jT7PVoGXlfoNRQ4dDsVmmY
FINspO3HzOkRwHriMUKS6cQtPyqIqXLt/MSoPTeDwAuW7le9f5sGocXLtg8kk/R6QTZffCNSeHY3
GWqkff8H/GbVllKULAMiRuKOEFUOMD9u7kE5uFmpCYPAjGVAicv2hXP6joVsBkLIcF3vAgMBAAEC
gYBvZHWoZHmS2EZQqKqeuGr58eobG9hcZzWQoJ4nq/CarBAjw/VovUHE490uK3S9ht4FW7Yzg3LV
/MBO6Huifh6qf/X9NQA7SeZRRC8gnCQk6JuDIEVJOud5jU+9tyumJakDKodQ3Jf2zQtNr+5ZdEP1
uwWgv9c4kmpjhAdyMuQmYQJBANn6pcgvyYaia52dnu+yBUsGkaFfwXkzFSExIbiOMXTkhEb/ER/D
rLytukkUu5S5ecz/KBa8U4xIslZDYQbLz5ECQQCy5dutt7RsxN4+dxCWnO/1FrkW12G329Ucewm3
QU9CKu4D+7Kqdj+Ha3lXP8F0Etaaapi7+EfkRUpukn2ItZV/AkEAlk+IOiphxT1rCBOQ5CjWDY5S
Df2B5JmdEG5Y2oOnLXwG2w44OLct/k2uD4cEcuITY5Dvi/4BftMCZwm/dnhEgQJACIktJSnJwxLV
o9dchENPtlsCM9C/Sd2EWpqISSUlmfugZbJBwR5pQ5XeMUqKeXZYpP+HEBj1nS+tMH9u2/IGEwJA
fL8mZiZXan/oBKrblAbplNcKWGRVD/3y65042PAEeghahlJMiYquV5DzZajuuTOwbJ5xQuZBO1+X
nfpFpBJ2dw==

公钥加密——私钥解密
加密前: abc

解密后: abc
公钥:

MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDdOj40yEB48XqWxmPILmJAc7UecIN7F32etSHF
9rwbuEh3+iTPOGSxhoSQpOED0vOb0ZIMkBXZSgsxLaBSin2RZO9YKWRjtpCA0kDkiD11gj4tzTiM
l9qq1kwSK7ZkGAgodEn3yIILVmQDuEImHOXFtulvJ71ka07u3LuwUNdB/wIDAQAB

私钥:
```

```
MIICdwIBADANBgkqhkiG9w0BAQEFAASCAmEwggJdAgEAAoGBAN06PjTIQHjxepbGY8guYkBztR5w
g3sXfZ61IcX2vBu4SHf6JM84ZLGGhJCk4QPS85vRkgyQFdlKCzEtoFKKfZFnT1gpZGO2kIDSQOSI
PXWCPi3NOIyX2qrWTBIrtmQYCCh0SffIggtWZAO4QiYc5cW26W8nvWRrTu7cu7BQ1OH/AgMBAAEC
gYEAz2JWBizjI31bqhP4XiP9PuY5F3vqBW4T+L9cFbQiyumKJc58yzTWUAUGKIIn3enXLG7dNqGr
mbJro4JeFIJ3CiVDpXR9+FluIgI4SXm7ioGKF2NOMA9LR5Fu82W+pLfpTN2y2SaLYWEDZyp53BxY
j9gUxaxi1MQs+C1ZgDF2xmECQQDy70bQntbRfysP+ppCtd56YRnES1TyekwOwryS2tr+ivQJl7JF
gp5rPAOXpgrq36xHDwUspQ0sJ0vj007ywxr1AkEA6SAaLhrJJrYucC0jxwAhUYyaPN+a0sWymaRh
9jA/Wc0wp29SbGTh5CcMuGpXm1gOM+FKW3dGiHgS3rVUKim4owJAbnxgapUzAgiiHxxMeDaavnHW
9C2GrtjsO7qtZOTgYI/1uT8itvZW8lJTF+9OW8/qXE76fXl7ai9dFnl5kzMk2QJBALfHz/vCsArt
mkRiwY6zApE4Z6tPl1V33ymSVovvUzHnOdD1SKQdD5t+UV/crb3QVi8ED0t2B0u0ZSPfDT/D7kMC
QDpwdj9k2F5aokLHBHUNJPFDAp7a5QMaT64gv/d48ITJ68Co+v5WzLMpzJBYXK6PAtqIhxbuPEc2
I2k1Afmrwyw=

私钥加密——公钥解密
加密前: sign

解密后: sign
私钥签名——公钥验证签名
签名:
ud1RsIwmSC1pN22I4IXteg1VD2FbiehKUfNxgVSHzvQNIK+d20FCkHCqh9djP3h94iWnIUY0ifU+
mbJkhAl/i5krExOE0hknOnPMcEP+lZV1RbJI2zG2YooSp2XDleqrQk5e/QF2Mx0Zxt8Xsg7ucVpn
i3wwbYWs9wSzIf0UjlM=

状态:
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！

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。

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。

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。

通过 java 代码实现如下：

```
import java.security.Key;
import java.security.KeyFactory;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PublicKey;
import java.security.spec.PKCS8EncodedKeySpec;
import java.security.spec.X509EncodedKeySpec;
import java.util.HashMap;
import java.util.Map;

import javax.crypto.Cipher;
import javax.crypto.KeyAgreement;
import javax.crypto.SecretKey;
import javax.crypto.interfaces.DHPrivateKey;
import javax.crypto.interfaces.DHPublicKey;
import javax.crypto.spec.DHParameterSpec;

⊟⊞/**
 * DH安全编码组件
```

```
 *
 * @author 梁栋  http://www.bt285.cn http://www.5a520.cn
 * @version 1.0
 * @since 1.0
 */
public abstract class DHCoder extends Coder {
    public static final String ALGORITHM = "DH";

    /**
     * 默认密钥字节数
     *
     * <pre>
     * DH
     * Default Keysize 1024
     * Keysize must be a multiple of 64, ranging from 512 to 1024 (inclusive).
     * </pre>
     */
    private static final int KEY_SIZE = 1024;

    /**
     * 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。
     */
    public static final String SECRET_ALGORITHM = "DES";
    private static final String PUBLIC_KEY = "DHPublicKey";
    private static final String PRIVATE_KEY = "DHPrivateKey";

    /**
     * 初始化甲方密钥
     *
     * @return
     * @throws Exception
     */
    public static Map<String, Object> initKey() throws Exception {
        KeyPairGenerator keyPairGenerator = KeyPairGenerator
                .getInstance(ALGORITHM);
        keyPairGenerator.initialize(KEY_SIZE);

        KeyPair keyPair = keyPairGenerator.generateKeyPair();

        // 甲方公钥
        DHPublicKey publicKey = (DHPublicKey) keyPair.getPublic();

        // 甲方私钥
```

```
        DHPrivateKey privateKey = (DHPrivateKey) keyPair.getPrivate();

        Map<String, Object> keyMap = new HashMap<String, Object>(2);

        keyMap.put(PUBLIC_KEY, publicKey);
        keyMap.put(PRIVATE_KEY, privateKey);
        return keyMap;
    }

    /**
     * 初始化乙方密钥
     *
     * @param key
     *            甲方公钥
     * @return
     * @throws Exception
     */
    public static Map<String, Object> initKey(String key) throws Exception {
        // 解析甲方公钥
        byte[] keyBytes = decryptBASE64(key);
        X509EncodedKeySpec x509KeySpec = new X509EncodedKeySpec(keyBytes);
        KeyFactory keyFactory = KeyFactory.getInstance(ALGORITHM);
        PublicKey pubKey = keyFactory.generatePublic(x509KeySpec);

        // 由甲方公钥构建乙方密钥
        DHParameterSpec dhParamSpec = ((DHPublicKey) pubKey).getParams();

        KeyPairGenerator keyPairGenerator = KeyPairGenerator
                .getInstance(keyFactory.getAlgorithm());
        keyPairGenerator.initialize(dhParamSpec);

        KeyPair keyPair = keyPairGenerator.generateKeyPair();

        // 乙方公钥
        DHPublicKey publicKey = (DHPublicKey) keyPair.getPublic();

        // 乙方私钥
        DHPrivateKey privateKey = (DHPrivateKey) keyPair.getPrivate();

        Map<String, Object> keyMap = new HashMap<String, Object>(2);

        keyMap.put(PUBLIC_KEY, publicKey);
        keyMap.put(PRIVATE_KEY, privateKey);

```

```
        return keyMap;
    }

    /**
     * 加密<br>
     *
     * @param data
     *            待加密数据
     * @param publicKey
     *            甲方公钥
     * @param privateKey
     *            乙方私钥
     * @return
     * @throws Exception
     */
    public static byte[] encrypt(byte[] data, String publicKey,
            String privateKey) throws Exception {

        // 生成本地密钥
        SecretKey secretKey = getSecretKey(publicKey, privateKey);

        // 数据加密
        Cipher cipher = Cipher.getInstance(secretKey.getAlgorithm());
        cipher.init(Cipher.ENCRYPT_MODE, secretKey);

        return cipher.doFinal(data);
    }

    /**
     * 解密<br>
     *
     * @param data
     *            待解密数据
     * @param publicKey
     *            乙方公钥
     * @param privateKey
     *            乙方私钥
     * @return
     * @throws Exception
     */
    public static byte[] decrypt(byte[] data, String publicKey,
            String privateKey) throws Exception {

        // 生成本地密钥
```

```
        SecretKey secretKey = getSecretKey(publicKey, privateKey);
        // 数据解密
        Cipher cipher = Cipher.getInstance(secretKey.getAlgorithm());
        cipher.init(Cipher.DECRYPT_MODE, secretKey);

        return cipher.doFinal(data);
    }

    /**
     * 构建密钥
     *
     * @param publicKey
     *            公钥
     * @param privateKey
     *            私钥
     * @return
     * @throws Exception
     */
    private static SecretKey getSecretKey(String publicKey, String privateKey)
            throws Exception {
        // 初始化公钥
        byte[] pubKeyBytes = decryptBASE64(publicKey);

        KeyFactory keyFactory = KeyFactory.getInstance(ALGORITHM);
        X509EncodedKeySpec x509KeySpec = new X509EncodedKeySpec(pubKeyBytes);
        PublicKey pubKey = keyFactory.generatePublic(x509KeySpec);

        // 初始化私钥
        byte[] priKeyBytes = decryptBASE64(privateKey);

        PKCS8EncodedKeySpec pkcs8KeySpec = new PKCS8EncodedKeySpec(priKeyBytes);
        Key priKey = keyFactory.generatePrivate(pkcs8KeySpec);

        KeyAgreement keyAgree = KeyAgreement.getInstance(keyFactory
                .getAlgorithm());
        keyAgree.init(priKey);
        keyAgree.doPhase(pubKey, true);

        // 生成本地密钥
        SecretKey secretKey = keyAgree.generateSecret(SECRET_ALGORITHM);

        return secretKey;
    }
```

```
    /**
     * 取得私钥
     *
     * @param keyMap
     * @return
     * @throws Exception
     */
    public static String getPrivateKey(Map<String, Object> keyMap)
            throws Exception {
        Key key = (Key) keyMap.get(PRIVATE_KEY);

        return encryptBASE64(key.getEncoded());
    }

    /**
     * 取得公钥
     *
     * @param keyMap
     * @return
     * @throws Exception
     */
    public static String getPublicKey(Map<String, Object> keyMap)
            throws Exception {
        Key key = (Key) keyMap.get(PUBLIC_KEY);

        return encryptBASE64(key.getEncoded());
    }
}
```

再给出一个测试类：

```
import static org.junit.Assert.*;

 import java.util.Map;

 import org.junit.Test;

/**
 *
 * @author 梁栋  http://www.bt285.cn http://www.feng123.com
 * @version 1.0
 * @since 1.0
 */
```

```
public class DHCoderTest {

    @Test
    public void test() throws Exception {
        // 生成甲方密钥对儿
        Map<String, Object> aKeyMap = DHCoder.initKey();
        String aPublicKey = DHCoder.getPublicKey(aKeyMap);
        String aPrivateKey = DHCoder.getPrivateKey(aKeyMap);

        System.err.println("甲方公钥:\r" + aPublicKey);
        System.err.println("甲方私钥:\r" + aPrivateKey);

        // 由甲方公钥产生本地密钥对儿
        Map<String, Object> bKeyMap = DHCoder.initKey(aPublicKey);
        String bPublicKey = DHCoder.getPublicKey(bKeyMap);
        String bPrivateKey = DHCoder.getPrivateKey(bKeyMap);

        System.err.println("乙方公钥:\r" + bPublicKey);
        System.err.println("乙方私钥:\r" + bPrivateKey);

        String aInput = "abc ";
        System.err.println("原文: " + aInput);

        // 由甲方公钥，乙方私钥构建密文
        byte[] aCode = DHCoder.encrypt(aInput.getBytes(), aPublicKey,
                bPrivateKey);

        // 由乙方公钥，甲方私钥解密
        byte[] aDecode = DHCoder.decrypt(aCode, bPublicKey, aPrivateKey);
        String aOutput = (new String(aDecode));

        System.err.println("解密: " + aOutput);

        assertEquals(aInput, aOutput);

        System.err.println(" ===============反过来加密解密
================= ");
        String bInput = "def ";
        System.err.println("原文: " + bInput);

        // 由乙方公钥，甲方私钥构建密文
        byte[] bCode = DHCoder.encrypt(bInput.getBytes(), bPublicKey,
                aPrivateKey);

```

```
        // 由甲方公钥，乙方私钥解密
        byte[] bDecode = DHCoder.decrypt(bCode, aPublicKey, bPrivateKey);
        String bOutput = (new String(bDecode));

        System.err.println("解密: " + bOutput);

        assertEquals(bInput, bOutput);
    }

}
```

控制台输出：

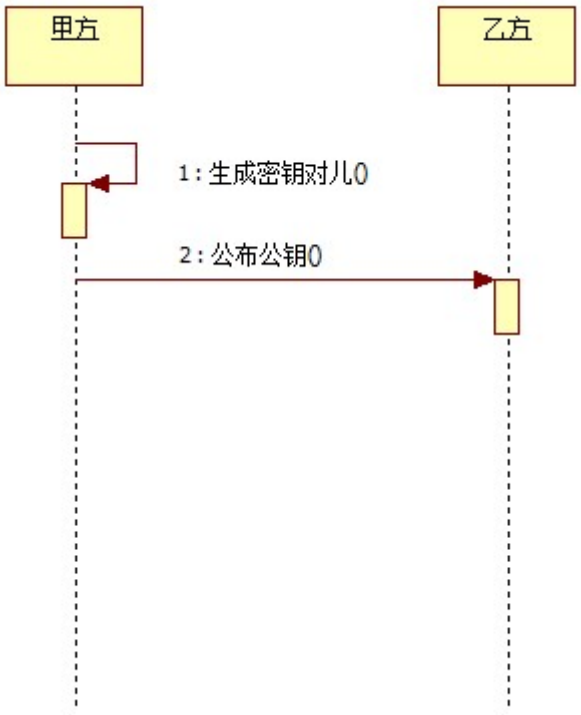
```
甲方公钥:
MIHfMIGXBgkqhkiG9w0BAwEwgYkCQQD8poLOjhLKuibvzPcRDlJtsHiwXt7LzR60ogjzrhYXrgHz
W5Gkfm32NBPF4S7QiZvNEyrNUNmRUb3EPuc3WS4XAkBnhHGyepz0TukaScUUfbGpqvJE8FpDTWSG
kx0tFCcbnjUDC3H9c9oXkGmzLik1Yw4cIGI1TQ2iCmxBblC+eUykAgIBgANDAAJAdAWBVmIzqcko
Ej6qFjLDL2+Y3FPq1iRbnOyOpDj71yKaK1K+FhTv04BOzy4DKcvAASV7/Gv0W+bgqdmffRkqrQ==

甲方私钥:
MIHRAgEAMIGXBgkqhkiG9w0BAwEwgYkCQQD8poLOjhLKuibvzPcRDlJtsHiwXt7LzR60ogjzrhYX
rgHzW5Gkfm32NBPF4S7QiZvNEyrNUNmRUb3EPuc3WS4XAkBnhHGyepz0TukaScUUfbGpqvJE8FpD
TWSGkx0tFCcbnjUDC3H9c9oXkGmzLik1Yw4cIGI1TQ2iCmxBblC+eUykAgIBgAQyAjACJRfy1LyR
eHyD+4Hfb+xR0uoIGR1oL9i9Nk6g2AAuaDPgEVWHn+QXID13yL/uDos=

乙方公钥:
MIHfMIGXBgkqhkiG9w0BAwEwgYkCQQD8poLOjhLKuibvzPcRDlJtsHiwXt7LzR60ogjzrhYXrgHz
W5Gkfm32NBPF4S7QiZvNEyrNUNmRUb3EPuc3WS4XAkBnhHGyepz0TukaScUUfbGpqvJE8FpDTWSG
kx0tFCcbnjUDC3H9c9oXkGmzLik1Yw4cIGI1TQ2iCmxBblC+eUykAgIBgANDAAJAVEYSfBA+I9nr
dWw3OBv475C+eBrWBBYqt0m6/eu4ptuDQHwV4MmUtKAC2wc2nNrdb1wmBhY1X8RnWkJ1XmdDbQ==

乙方私钥:
MIHSAgEAMIGXBgkqhkiG9w0BAwEwgYkCQQD8poLOjhLKuibvzPcRDlJtsHiwXt7LzR60ogjzrhYX
rgHzW5Gkfm32NBPF4S7QiZvNEyrNUNmRUb3EPuc3WS4XAkBnhHGyepz0TukaScUUfbGpqvJE8FpD
TWSGkx0tFCcbnjUDC3H9c9oXkGmzLik1Yw4cIGI1TQ2iCmxBblC+eUykAgIBgAQzAjEAqaZiCdXp
2iNpdBlHRaO9ir70wo2n32xNlIzIX19VLSPCDdeUWkgRv4CEj/8k+/yd

原文: abc
解密: abc
 ==============反过来加密解密=================
原文: def
解密: def
```

接下来我们介绍 DSA 数字签名，非对称加密的另一种实现。

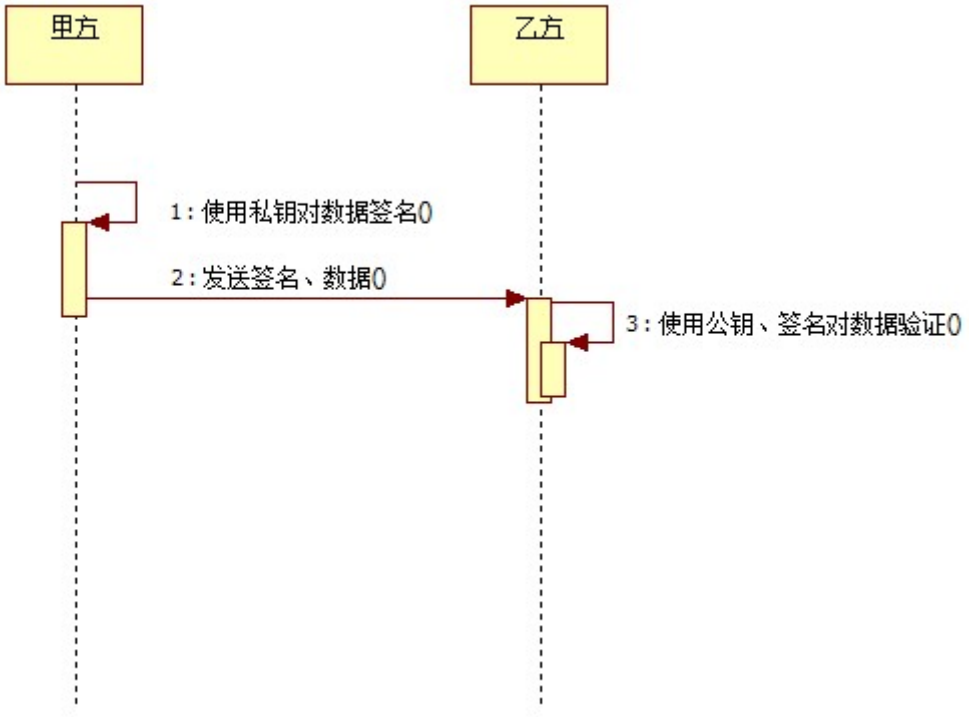
DSA

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！

1.



2.



通过 java 代码实现如下：

```
  import java.security.Key;
import java.security.KeyFactory;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.SecureRandom;
import java.security.Signature;
import java.security.interfaces.DSAPrivateKey;
import java.security.interfaces.DSAPublicKey;
import java.security.spec.PKCS8EncodedKeySpec;
import java.security.spec.X509EncodedKeySpec;
import java.util.HashMap;
import java.util.Map;

/**
 * DSA安全编码组件
 *
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
```

```
public abstract class DSACoder extends Coder {

    public static final String ALGORITHM = "DSA";

    /**
     * 默认密钥字节数
     *
     * <pre>
     * DSA
     * Default Keysize 1024
     * Keysize must be a multiple of 64, ranging from 512 to 1024 (inclusive).
     * </pre>
     */
    private static final int KEY_SIZE = 1024;

    /**
     * 默认种子
     */
    private static final String DEFAULT_SEED = "0f22507a10bbddd07d8a3082122966e3";

    private static final String PUBLIC_KEY = "DSAPublicKey";
    private static final String PRIVATE_KEY = "DSAPrivateKey";

    /**
     * 用私钥对信息生成数字签名
     *
     * @param data
     *            加密数据
     * @param privateKey
     *            私钥
     *
     * @return
     * @throws Exception
     */
    public static String sign(byte[] data, String privateKey) throws Exception {
        // 解密由base64编码的私钥
        byte[] keyBytes = decryptBASE64(privateKey);

        // 构造PKCS8EncodedKeySpec对象
        PKCS8EncodedKeySpec pkcs8KeySpec = new PKCS8EncodedKeySpec(keyBytes);

        // KEY_ALGORITHM 指定的加密算法
        KeyFactory keyFactory = KeyFactory.getInstance(ALGORITHM);
```

```
        // 取私钥匙对象
        PrivateKey priKey = keyFactory.generatePrivate(pkcs8KeySpec);

        // 用私钥对信息生成数字签名
        Signature signature = Signature.getInstance(keyFactory.getAlgorithm());
        signature.initSign(priKey);
        signature.update(data);

        return encryptBASE64(signature.sign());
    }

    /**
     * 校验数字签名
     *
     * @param data
     *            加密数据
     * @param publicKey
     *            公钥
     * @param sign
     *            数字签名
     *
     * @return 校验成功返回 true 失败返回 false
     * @throws Exception
     *
     */
    public static boolean verify(byte[] data, String publicKey, String sign)
            throws Exception {

        // 解密由 base64 编码的公钥
        byte[] keyBytes = decryptBASE64(publicKey);

        // 构造 X509EncodedKeySpec 对象
        X509EncodedKeySpec keySpec = new X509EncodedKeySpec(keyBytes);

        // ALGORITHM 指定的加密算法
        KeyFactory keyFactory = KeyFactory.getInstance(ALGORITHM);

        // 取公钥匙对象
        PublicKey pubKey = keyFactory.generatePublic(keySpec);

        Signature signature = Signature.getInstance(keyFactory.getAlgorithm());
        signature.initVerify(pubKey);
        signature.update(data);

        // 验证签名是否正常
```

```
        return signature.verify(decryptBASE64(sign));
    }

    /**
     * 生成密钥
     *
     * @param seed
     *            种子
     * @return 密钥对象
     * @throws Exception
     */
    public static Map<String, Object> initKey(String seed) throws Exception {
        KeyPairGenerator keygen = KeyPairGenerator.getInstance(ALGORITHM);
        // 初始化随机产生器
        SecureRandom secureRandom = new SecureRandom();
        secureRandom.setSeed(seed.getBytes());
        keygen.initialize(KEY_SIZE, secureRandom);

        KeyPair keys = keygen.genKeyPair();

        DSAPublicKey publicKey = (DSAPublicKey) keys.getPublic();
        DSAPrivateKey privateKey = (DSAPrivateKey) keys.getPrivate();

        Map<String, Object> map = new HashMap<String, Object>(2);
        map.put(PUBLIC_KEY, publicKey);
        map.put(PRIVATE_KEY, privateKey);

        return map;
    }

    /**
     * 默认生成密钥
     *
     * @return 密钥对象
     * @throws Exception
     */
    public static Map<String, Object> initKey() throws Exception {
        return initKey(DEFAULT_SEED);
    }

    /**
     * 取得私钥
     *
     * @param keyMap
```

```
     * @return
     * @throws Exception
     */
    public static String getPrivateKey(Map<String, Object> keyMap)
            throws Exception {
        Key key = (Key) keyMap.get(PRIVATE_KEY);

        return encryptBASE64(key.getEncoded());
    }

    /**
     * 取得公钥
     *
     * @param keyMap
     * @return
     * @throws Exception
     */
    public static String getPublicKey(Map<String, Object> keyMap)
            throws Exception {
        Key key = (Key) keyMap.get(PUBLIC_KEY);

        return encryptBASE64(key.getEncoded());
    }
}
```

再给出一个测试类：

```
  import static org.junit.Assert.*;

import java.util.Map;

import org.junit.Test;

/**
 *
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public class DSACoderTest {

    @Test
    public void test() throws Exception {
        String inputStr = "abc";
        byte[] data = inputStr.getBytes();
```

```
        // 构建密钥
        Map<String, Object> keyMap = DSACoder.initKey();

        // 获得密钥
        String publicKey = DSACoder.getPublicKey(keyMap);
        String privateKey = DSACoder.getPrivateKey(keyMap);

        System.err.println("公钥:\r" + publicKey);
        System.err.println("私钥:\r" + privateKey);

        // 产生签名
        String sign = DSACoder.sign(data, privateKey);
        System.err.println("签名:\r" + sign);

        // 验证签名
        boolean status = DSACoder.verify(data, publicKey, sign);
        System.err.println("状态:\r" + status);
        assertTrue(status);

    }

}
```

控制台输出：

```
 公钥:
MIIBtzCCASwGByqGSM44BAEwggEfAoGBAP1/U4EddRIpUt9KnC7s5Of2EbdSPO9EAMMeP4C2USZp
RV1AIlH7WT2NWPq/xfW6MPbLm1Vs14E7gB00b/JmYLdrmVClpJ+f6AR7ECLCT7up1/63xhv4O1fn
xqimFQ8E+4P208UewwI1VBNaFpEy9nXzrith1yrv8iIDGZ3RSAHHAhUAl2BQjxUjC8yykrmCouuE
C/BYHPUCgYEA9+GghdabPd7LvKtcNrhXuXmUr7v6OuqC+VdMCz0HgmdRWVeOutRZT+ZxBxCBgLRJ
FnEj6EwoFhO3zwkyjMim4TwWeotUfI0o4KOuHiuzpnWRbqN/C/ohNWLx+2J6ASQ7zKTxvqhRkImo
g9/hWuWfBpKLZl6Ae1UlZAFMO/7PSSoDgYQAAoGAIu4RUlcQLp49PI0MrbssOY+3uySVnp0TULSv
5T4VaHoKzsLHgGTrwOvsGA+V3yCNl2WDu3D84bSLF7liTWgOj+SMOEaPk4VyRTlLXZWGPsf1Mfd9
21XAbMeVyKDSHHVGbMjBScajf3bXooYQMlyoHiOt/WrCo+mv7efstMM0PGo=

私钥:
MIIBTAIBADCCASwGByqGSM44BAEwggEfAoGBAP1/U4EddRIpUt9KnC7s5Of2EbdSPO9EAMMeP4C2
USZpRV1AIlH7WT2NWPq/xfW6MPbLm1Vs14E7gB00b/JmYLdrmVClpJ+f6AR7ECLCT7up1/63xhv4
O1fnxqimFQ8E+4P208UewwI1VBNaFpEy9nXzrith1yrv8iIDGZ3RSAHHAhUAl2BQjxUjC8yykrmC
ouuEC/BYHPUCgYEA9+GghdabPd7LvKtcNrhXuXmUr7v6OuqC+VdMCz0HgmdRWVeOutRZT+ZxBxCB
gLRJFnEj6EwoFhO3zwkyjMim4TwWeotUfI0o4KOuHiuzpnWRbqN/C/ohNWLx+2J6ASQ7zKTxvqhR
kImog9/hWuWfBpKLZl6Ae1UlZAFMO/7PSSoEFwIVAIegLUtmm2oQKQJTOiLugHTSjl/q

签名:
MC0CFQCMg0J/uZmF8GuRpr3TNq48w60nDwIUJCyYNah+HtbU6NcQfy8Ac6LeLQs=
```

```
状态:
true
```

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。

尽管如此，我照旧提供相应的 Java 实现代码，以供大家参考。

通过 java 代码实现如下：

```
 import  java.math.BigInteger;
import  java.security.Key;
import  java.security.KeyFactory;
import  java.security.interfaces.ECPrivateKey;
import  java.security.interfaces.ECPublicKey;
import  java.security.spec.ECFieldF2m;
import  java.security.spec.ECParameterSpec;
import  java.security.spec.ECPoint;
import  java.security.spec.ECPrivateKeySpec;
import  java.security.spec.ECPublicKeySpec;
import  java.security.spec.EllipticCurve;
import  java.security.spec.PKCS8EncodedKeySpec;
import  java.security.spec.X509EncodedKeySpec;
import  java.util.HashMap;
import  java.util.Map;

import  javax.crypto.Cipher;
import  javax.crypto.NullCipher;

import  sun.security.ec.ECKeyFactory;
import  sun.security.ec.ECPrivateKeyImpl;
import  sun.security.ec.ECPublicKeyImpl;

/**
 * ECC安全编码组件
 *
```

```
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public abstract class ECCCoder extends Coder {

    public static final String ALGORITHM = "EC";
    private static final String PUBLIC_KEY = "ECCPublicKey";
    private static final String PRIVATE_KEY = "ECCPrivateKey";

    /**
     * 解密<br>
     * 用私钥解密
     *
     * @param data
     * @param key
     * @return
     * @throws Exception
     */
    public static byte[] decrypt(byte[] data, String key) throws Exception {
        // 对密钥解密
        byte[] keyBytes = decryptBASE64(key);

        // 取得私钥
        PKCS8EncodedKeySpec pkcs8KeySpec = new PKCS8EncodedKeySpec(keyBytes);
        KeyFactory keyFactory = ECKeyFactory.INSTANCE;

        ECPrivateKey priKey = (ECPrivateKey) keyFactory
                .generatePrivate(pkcs8KeySpec);

        ECPrivateKeySpec ecPrivateKeySpec = new ECPrivateKeySpec(priKey.getS(),
                priKey.getParams());

        // 对数据解密
        // TODO Chipher 不支持 EC 算法 未能实现
        Cipher cipher = new NullCipher();
        // Cipher.getInstance(ALGORITHM, keyFactory.getProvider());
        cipher.init(Cipher.DECRYPT_MODE, priKey, ecPrivateKeySpec.getParams());

        return cipher.doFinal(data);
    }

    /**
     * 加密<br>
```

```
     * 用公钥加密
     *
     * @param data
     * @param privateKey
     * @return
     * @throws Exception
     */
    public static byte[] encrypt(byte[] data, String privateKey)
            throws Exception {
        // 对公钥解密
        byte[] keyBytes = decryptBASE64(privateKey);

        // 取得公钥
        X509EncodedKeySpec x509KeySpec = new X509EncodedKeySpec(keyBytes);
        KeyFactory keyFactory = ECKeyFactory.INSTANCE;

        ECPublicKey pubKey = (ECPublicKey) keyFactory
                .generatePublic(x509KeySpec);

        ECPublicKeySpec ecPublicKeySpec = new ECPublicKeySpec(pubKey.getW(),
                pubKey.getParams());

        // 对数据加密
        // TODO Chipher 不支持 EC 算法 未能实现
        Cipher cipher = new NullCipher();
        // Cipher.getInstance(ALGORITHM, keyFactory.getProvider());
        cipher.init(Cipher.ENCRYPT_MODE, pubKey, ecPublicKeySpec.getParams());

        return cipher.doFinal(data);
    }

    /**
     * 取得私钥
     *
     * @param keyMap
     * @return
     * @throws Exception
     */
    public static String getPrivateKey(Map<String, Object> keyMap)
            throws Exception {
        Key key = (Key) keyMap.get(PRIVATE_KEY);

        return encryptBASE64(key.getEncoded());
```

```
}

/**
 * 取得公钥
 *
 * @param keyMap
 * @return
 * @throws Exception
 */
public static String getPublicKey(Map<String, Object> keyMap)
        throws Exception {
    Key key = (Key) keyMap.get(PUBLIC_KEY);

    return encryptBASE64(key.getEncoded());
}

/**
 * 初始化密钥
 *
 * @return
 * @throws Exception
 */
public static Map<String, Object> initKey() throws Exception {
    BigInteger x1 = new BigInteger(
            "2fe13c0537bbc11acaa07d793de4e6d5e5c94eee8", 16);
    BigInteger x2 = new BigInteger(
            "289070fb05d38ff58321f2e800536d538ccdaa3d9", 16);

    ECPoint g = new ECPoint(x1, x2);

    // the order of generator
    BigInteger n = new BigInteger(
            "5846006549323611672814741753598448348329118574063", 10);
    // the cofactor
    int h = 2;
    int m = 163;
    int[] ks = { 7, 6, 3 };
    ECFieldF2m ecField = new ECFieldF2m(m, ks);
    // y^2+xy=x^3+x^2+1
    BigInteger a = new BigInteger("1", 2);
    BigInteger b = new BigInteger("1", 2);

    EllipticCurve ellipticCurve = new EllipticCurve(ecField, a, b);
```

```
        ECParameterSpec ecParameterSpec = new ECParameterSpec(ellipticCurve, g,
                n, h);
        // 公钥
        ECPublicKey publicKey = new ECPublicKeyImpl(g, ecParameterSpec);

        BigInteger s = new BigInteger(
                "1234006549323611672814741753598448348329118574063", 10);
        // 私钥
        ECPrivateKey privateKey = new ECPrivateKeyImpl(s, ecParameterSpec);

        Map<String, Object> keyMap = new HashMap<String, Object>(2);

        keyMap.put(PUBLIC_KEY, publicKey);
        keyMap.put(PRIVATE_KEY, privateKey);

        return keyMap;
    }

}
```

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：

```
 import static org.junit.Assert.*;

import java.math.BigInteger;
import java.security.spec.ECFieldF2m;
import java.security.spec.ECParameterSpec;
import java.security.spec.ECPoint;
import java.security.spec.ECPrivateKeySpec;
import java.security.spec.ECPublicKeySpec;
import java.security.spec.EllipticCurve;
import java.util.Map;

import org.junit.Test;

/**
 *
 * @author 梁栋
 * @version 1.0
 * @since 1.0
```

```
 */
public class ECCCoderTest {

    @Test
    public void test() throws Exception {
        String inputStr = "abc";
        byte[] data = inputStr.getBytes();

        Map<String, Object> keyMap = ECCCoder.initKey();

        String publicKey = ECCCoder.getPublicKey(keyMap);
        String privateKey = ECCCoder.getPrivateKey(keyMap);
        System.err.println("公钥: \n" + publicKey);
        System.err.println("私钥:  \n" + privateKey);

        byte[] encodedData = ECCCoder.encrypt(data, publicKey);

        byte[] decodedData = ECCCoder.decrypt(encodedData, privateKey);

        String outputStr = new String(decodedData);
        System.err.println("加密前:" + inputStr + "\n\r" + "解密后:" + outputStr);
        assertEquals(inputStr, outputStr);
    }
}
```

控制台输出：

```
 公钥:
MEAwEAYHKoZIzj0CAQYFK4EEAAEDLAAEAv4TwFN7vBGsqgfXk95ObV5c1O7oAokHD7BdOP9YMh8u
gAU21TjM2qPZ

私钥:
MDICAQAwEAYHKoZIzj0CAQYFK4EEAAEEGzAZAgEBBBTYJsR3BN7TFw7JHcAHFkwNmfil7w==

加密前: abc

解密后: abc
```

本篇的主要内容为Java证书体系的实现。

请大家在阅读本篇内容时先阅读 Java加密技术（四），预先了解RSA加密算法。

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：

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

```
输入 keystore 密码:

再次输入新密码:

您的名字与姓氏是什么?

[Unknown]:  www.zlex.org

您的组织单位名称是什么?

[Unknown]:  zlex

您的组织名称是什么?

[Unknown]:  zlex

您所在的城市或区域名称是什么?

[Unknown]:  BJ

您所在的州或省份名称是什么?

[Unknown]:  BJ
```

```
该单位的两字母国家代码是什么

[Unknown]:  CN

CN=www.zlex.org, OU=zlex, O=zlex, L=BJ, ST=BJ, C=CN 正确吗?

[否]:  Y

输入<tomcat>的主密码

（如果和 keystore 密码相同，按回车）:

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

```
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

输入 keystore 密码：

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！

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）

Java 代码

```
 import java.io.FileInputStream;
import java.security.KeyStore;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.Signature;
import java.security.cert.Certificate;
import java.security.cert.CertificateFactory;
import java.security.cert.X509Certificate;
import java.util.Date;

import javax.crypto.Cipher;

/**
 * 证书组件
 *
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public abstract class CertificateCoder extends Coder {


    /**
     * Java 密钥库(Java Key Store，JKS)KEY_STORE
     */
    public static final String KEY_STORE = "JKS";

    public static final String X509 = "X.509";

    /**
     * 由 KeyStore 获得私钥
```

```
     *
     * @param keyStorePath
     * @param alias
     * @param password
     * @return
     * @throws Exception
     */
    private static PrivateKey getPrivateKey(String keyStorePath, String alias,
            String password) throws Exception {
        KeyStore ks = getKeyStore(keyStorePath, password);
        PrivateKey key = (PrivateKey) ks.getKey(alias, password.toCharArray());
        return key;
    }

    /**
     * 由Certificate获得公钥
     *
     * @param certificatePath
     * @return
     * @throws Exception
     */
    private static PublicKey getPublicKey(String certificatePath)
            throws Exception {
        Certificate certificate = getCertificate(certificatePath);
        PublicKey key = certificate.getPublicKey();
        return key;
    }

    /**
     * 获得Certificate
     *
     * @param certificatePath
     * @return
     * @throws Exception
     */
    private static Certificate getCertificate(String certificatePath)
            throws Exception {
        CertificateFactory certificateFactory = CertificateFactory
                .getInstance(X509);
        FileInputStream in = new FileInputStream(certificatePath);

        Certificate certificate = certificateFactory.generateCertificate(in);
        in.close();
```

```
        return certificate;
    }

    /**
     * 获得Certificate
     *
     * @param keyStorePath
     * @param alias
     * @param password
     * @return
     * @throws Exception
     */
    private static Certificate getCertificate(String keyStorePath,
            String alias, String password) throws Exception {
        KeyStore ks = getKeyStore(keyStorePath, password);
        Certificate certificate = ks.getCertificate(alias);

        return certificate;
    }

    /**
     * 获得KeyStore
     *
     * @param keyStorePath
     * @param password
     * @return
     * @throws Exception
     */
    private static KeyStore getKeyStore(String keyStorePath, String password)
            throws Exception {
        FileInputStream is = new FileInputStream(keyStorePath);
        KeyStore ks = KeyStore.getInstance(KEY_STORE);
        ks.load(is, password.toCharArray());
        is.close();
        return ks;
    }

    /**
     * 私钥加密
     *
     * @param data
     * @param keyStorePath
     * @param alias
     * @param password
```

```
     * @return
     * @throws Exception
     */
    public static byte[] encryptByPrivateKey(byte[] data, String keyStorePath,
            String alias, String password) throws Exception {
        // 取得私钥
        PrivateKey privateKey = getPrivateKey(keyStorePath, alias, password);

        // 对数据加密
        Cipher cipher = Cipher.getInstance(privateKey.getAlgorithm());
        cipher.init(Cipher.ENCRYPT_MODE, privateKey);

        return cipher.doFinal(data);

    }

    /**
     * 私钥解密
     *
     * @param data
     * @param keyStorePath
     * @param alias
     * @param password
     * @return
     * @throws Exception
     */
    public static byte[] decryptByPrivateKey(byte[] data, String keyStorePath,
            String alias, String password) throws Exception {
        // 取得私钥
        PrivateKey privateKey = getPrivateKey(keyStorePath, alias, password);

        // 对数据加密
        Cipher cipher = Cipher.getInstance(privateKey.getAlgorithm());
        cipher.init(Cipher.DECRYPT_MODE, privateKey);

        return cipher.doFinal(data);

    }

    /**
     * 公钥加密
     *
     * @param data
     * @param certificatePath
```

```
     * @return
     * @throws Exception
     */
    public static byte[] encryptByPublicKey(byte[] data, String certificatePath)
            throws Exception {

        // 取得公钥
        PublicKey publicKey = getPublicKey(certificatePath);
        // 对数据加密
        Cipher cipher = Cipher.getInstance(publicKey.getAlgorithm());
        cipher.init(Cipher.ENCRYPT_MODE, publicKey);

        return cipher.doFinal(data);

    }

    /**
     * 公钥解密
     *
     * @param data
     * @param certificatePath
     * @return
     * @throws Exception
     */
    public static byte[] decryptByPublicKey(byte[] data, String certificatePath)
            throws Exception {
        // 取得公钥
        PublicKey publicKey = getPublicKey(certificatePath);

        // 对数据加密
        Cipher cipher = Cipher.getInstance(publicKey.getAlgorithm());
        cipher.init(Cipher.DECRYPT_MODE, publicKey);

        return cipher.doFinal(data);

    }

    /**
     * 验证Certificate
     *
     * @param certificatePath
     * @return
     */
    public static boolean verifyCertificate(String certificatePath) {
```

```
        return verifyCertificate(new Date(), certificatePath);
    }

    /**
     * 验证Certificate是否过期或无效
     *
     * @param date
     * @param certificatePath
     * @return
     */
    public static boolean verifyCertificate(Date date, String certificatePath) {
        boolean status = true;
        try {
            // 取得证书
            Certificate certificate = getCertificate(certificatePath);
            // 验证证书是否过期或无效
            status = verifyCertificate(date, certificate);
        } catch (Exception e) {
            status = false;
        }
        return status;
    }

    /**
     * 验证证书是否过期或无效
     *
     * @param date
     * @param certificate
     * @return
     */
    private static boolean verifyCertificate(Date date, Certificate certificate)
{
        boolean status = true;
        try {
            X509Certificate x509Certificate = (X509Certificate) certificate;
            x509Certificate.checkValidity(date);
        } catch (Exception e) {
            status = false;
        }
        return status;
    }

    /**
     * 签名
```

```
     *
     * @param keyStorePath
     * @param alias
     * @param password
     *
     * @return
     * @throws Exception
     */
    public static String sign(byte[] sign, String keyStorePath, String alias,
            String password) throws Exception {
        // 获得证书
        X509Certificate x509Certificate = (X509Certificate) getCertificate(
                keyStorePath, alias, password);
        // 获取私钥
        KeyStore ks = getKeyStore(keyStorePath, password);
        // 取得私钥
        PrivateKey privateKey = (PrivateKey) ks.getKey(alias, password
                .toCharArray());

        // 构建签名
        Signature signature = Signature.getInstance(x509Certificate
                .getSigAlgName());
        signature.initSign(privateKey);
        signature.update(sign);
        return encryptBASE64(signature.sign());
    }

    /**
     * 验证签名
     *
     * @param data
     * @param sign
     * @param certificatePath
     * @return
     * @throws Exception
     */
    public static boolean verify(byte[] data, String sign,
            String certificatePath) throws Exception {
        // 获得证书
        X509Certificate x509Certificate = (X509Certificate)
getCertificate(certificatePath);
        // 获得公钥
        PublicKey publicKey = x509Certificate.getPublicKey();
        // 构建签名
```

```
        Signature signature = Signature.getInstance(x509Certificate
                .getSigAlgName());
        signature.initVerify(publicKey);
        signature.update(data);

        return signature.verify(decryptBASE64(sign));

    }

    /**
     * 验证Certificate
     *
     * @param keyStorePath
     * @param alias
     * @param password
     * @return
     */
    public static boolean verifyCertificate(Date date, String keyStorePath,
            String alias, String password) {
        boolean status = true;
        try {
            Certificate certificate = getCertificate(keyStorePath, alias,
                    password);
            status = verifyCertificate(date, certificate);
        } catch (Exception e) {
            status = false;
        }
        return status;
    }

    /**
     * 验证Certificate
     *
     * @param keyStorePath
     * @param alias
     * @param password
     * @return
     */
    public static boolean verifyCertificate(String keyStorePath, String alias,
            String password) {
        return verifyCertificate(new Date(), keyStorePath, alias, password);
    }
}
```

再给出一个测试类：

```
 import static org.junit.Assert.*;

import org.junit.Test;

/**
 *
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public class CertificateCoderTest {
    private String password = "123456";
    private String alias = "www.zlex.org";
    private String certificatePath = "d:/zlex.cer";
    private String keyStorePath = "d:/zlex.keystore";

    @Test
    public void test() throws Exception {
        System.err.println("公钥加密——私钥解密");
        String inputStr = "Ceritifcate";
        byte[] data = inputStr.getBytes();

        byte[] encrypt = CertificateCoder.encryptByPublicKey(data,
                certificatePath);

        byte[] decrypt = CertificateCoder.decryptByPrivateKey(encrypt,
                keyStorePath, alias, password);
        String outputStr = new String(decrypt);

        System.err.println("加密前: " + inputStr + "\n\r" + "解密后: " + outputStr);

        // 验证数据一致
        assertArrayEquals(data, decrypt);

        // 验证证书有效
        assertTrue(CertificateCoder.verifyCertificate(certificatePath));

    }

    @Test
    public void testSign() throws Exception {
        System.err.println("私钥加密——公钥解密");

        String inputStr = "sign";
```

```
        byte[] data = inputStr.getBytes();

        byte[] encodedData = CertificateCoder.encryptByPrivateKey(data,
                keyStorePath, alias, password);

        byte[] decodedData = CertificateCoder.decryptByPublicKey(encodedData,
                certificatePath);

        String outputStr = new String(decodedData);
        System.err.println("加密前: " + inputStr + "\n\r" + "解密后: " + outputStr);
        assertEquals(inputStr, outputStr);

        System.err.println("私钥签名——公钥验证签名");
        // 产生签名
        String sign = CertificateCoder.sign(encodedData, keyStorePath, alias,
                password);
        System.err.println("签名:\r" + sign);

        // 验证签名
        boolean status = CertificateCoder.verify(encodedData, sign,
                certificatePath);
        System.err.println("状态:\r" + status);
        assertTrue(status);

    }
}
```

控制台输出：

Console 代码

```
 公钥加密——私钥解密
加密前: Ceritificate

解密后: Ceritificate

私钥加密——公钥解密
加密前: sign

解密后: sign
私钥签名——公钥验证签名
签名:
pqBn5m6PJlfOjHOA6U2o2mUmBsfgyEY1NWCbiyA/I5Gc3gaVNVIdj/zkGNZRqTjhf3+J9a9z9EI7
6F2eWYd7punHx5oh6hfNgcKbVb52EfItl4QEN+djbXiPynn07+Lbg1NOjULnpEd6ZhLP1YwrEAuM
```

```
OfvXOe7/wplxLbySaKQ=

状态:
true
```

由此完成了证书验证体系！

同样，我们可以对代码做签名——代码签名！

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

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

控制台输出：

Console 代码

```
 输入密钥库的口令短语:
 正在更新:  META-INF/WWW_ZLEX.SF
 正在更新:  META-INF/WWW_ZLEX.RSA
  正在签名:  org/zlex/security/Security.class
  正在签名:  org/zlex/tool/Main$1.class
  正在签名:  org/zlex/tool/Main$2.class
  正在签名:  org/zlex/tool/Main.class

警告:
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！

验证 tools.jar，命令如下：

Shell 代码

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

控制台输出：

Console 代码

```
   402 Sat Jun 20 16:25:14 CST 2009 META-INF/MANIFEST.MF
         532 Sat Jun 20 16:25:14 CST 2009 META-INF/WWW_ZLEX.SF
         889 Sat Jun 20 16:25:14 CST 2009 META-INF/WWW_ZLEX.RSA
sm       590 Wed Dec 10 13:03:42 CST 2008 org/zlex/security/Security.class

      X.509, CN=www.zlex.org, OU=zlex, O=zlex, L=BJ, ST=BJ, C=CN
      [证书将在 09-9-18 下午 3:27 到期]

sm       705 Tue Dec 16 18:00:56 CST 2008 org/zlex/tool/Main$1.class

      X.509, CN=www.zlex.org, OU=zlex, O=zlex, L=BJ, ST=BJ, C=CN
      [证书将在 09-9-18 下午 3:27 到期]

sm       779 Tue Dec 16 18:00:56 CST 2008 org/zlex/tool/Main$2.class

      X.509, CN=www.zlex.org, OU=zlex, O=zlex, L=BJ, ST=BJ, C=CN
      [证书将在 09-9-18 下午 3:27 到期]

sm     12672 Tue Dec 16 18:00:56 CST 2008 org/zlex/tool/Main.class

      X.509, CN=www.zlex.org, OU=zlex, O=zlex, L=BJ, ST=BJ, C=CN
      [证书将在 09-9-18 下午 3:27 到期]


  s = 已验证签名
  m = 在清单中列出条目
  k = 在密钥库中至少找到了一个证书
  i = 在身份作用域内至少找到了一个证书

jar 已验证。

警告:
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。

在Java加密技术（八）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

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

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

```
<Connector
    SSLEnabled="true"
    URIEncoding="UTF-8"
    clientAuth="false"
    keystoreFile="conf/zlex.keystore"
    keystorePass="123456"
    maxThreads="150"
    port="443"
    protocol="HTTP/1.1"
    scheme="https"
    secure="true"
    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/.

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  ，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

```
 import java.io.FileInputStream;
import java.security.KeyStore;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.Signature;
import java.security.cert.Certificate;
import java.security.cert.CertificateFactory;
```

```
import java.security.cert.X509Certificate;
import java.util.Date;

import javax.crypto.Cipher;
import javax.net.ssl.HttpsURLConnection;
import javax.net.ssl.KeyManagerFactory;
import javax.net.ssl.SSLContext;
import javax.net.ssl.SSLSocketFactory;
import javax.net.ssl.TrustManagerFactory;

/**
 * 证书组件
 *
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public abstract class CertificateCoder extends Coder {

    /**
     * Java密钥库(Java Key Store，JKS)KEY_STORE
     */
    public static final String KEY_STORE = "JKS";

    public static final String X509 = "X.509";
    public static final String SunX509 = "SunX509";
    public static final String SSL = "SSL";

    /**
     * 由KeyStore获得私钥
     *
     * @param keyStorePath
     * @param alias
     * @param password
     * @return
     * @throws Exception
     */
    private static PrivateKey getPrivateKey(String keyStorePath, String alias,
            String password) throws Exception {
        KeyStore ks = getKeyStore(keyStorePath, password);
        PrivateKey key = (PrivateKey) ks.getKey(alias, password.toCharArray());
        return key;
    }
```

```
/**
 * 由Certificate获得公钥
 *
 * @param certificatePath
 * @return
 * @throws Exception
 */
private static PublicKey getPublicKey(String certificatePath)
        throws Exception {
    Certificate certificate = getCertificate(certificatePath);
    PublicKey key = certificate.getPublicKey();
    return key;
}

/**
 * 获得Certificate
 *
 * @param certificatePath
 * @return
 * @throws Exception
 */
private static Certificate getCertificate(String certificatePath)
        throws Exception {
    CertificateFactory certificateFactory = CertificateFactory
            .getInstance(X509);
    FileInputStream in = new FileInputStream(certificatePath);

    Certificate certificate = certificateFactory.generateCertificate(in);
    in.close();

    return certificate;
}

/**
 * 获得Certificate
 *
 * @param keyStorePath
 * @param alias
 * @param password
 * @return
 * @throws Exception
 */
private static Certificate getCertificate(String keyStorePath,
        String alias, String password) throws Exception {
```

```
        KeyStore ks = getKeyStore(keyStorePath, password);
        Certificate certificate = ks.getCertificate(alias);

        return certificate;
    }

    /**
     * 获得KeyStore
     *
     * @param keyStorePath
     * @param password
     * @return
     * @throws Exception
     */
    private static KeyStore getKeyStore(String keyStorePath, String password)
            throws Exception {
        FileInputStream is = new FileInputStream(keyStorePath);
        KeyStore ks = KeyStore.getInstance(KEY_STORE);
        ks.load(is, password.toCharArray());
        is.close();
        return ks;
    }

    /**
     * 私钥加密
     *
     * @param data
     * @param keyStorePath
     * @param alias
     * @param password
     * @return
     * @throws Exception
     */
    public static byte[] encryptByPrivateKey(byte[] data, String keyStorePath,
            String alias, String password) throws Exception {
        // 取得私钥
        PrivateKey privateKey = getPrivateKey(keyStorePath, alias, password);

        // 对数据加密
        Cipher cipher = Cipher.getInstance(privateKey.getAlgorithm());
        cipher.init(Cipher.ENCRYPT_MODE, privateKey);

        return cipher.doFinal(data);
```

```
    }

    /**
     * 私钥解密
     *
     * @param data
     * @param keyStorePath
     * @param alias
     * @param password
     * @return
     * @throws Exception
     */
    public static byte[] decryptByPrivateKey(byte[] data, String keyStorePath,
            String alias, String password) throws Exception {
        // 取得私钥
        PrivateKey privateKey = getPrivateKey(keyStorePath, alias, password);

        // 对数据加密
        Cipher cipher = Cipher.getInstance(privateKey.getAlgorithm());
        cipher.init(Cipher.DECRYPT_MODE, privateKey);

        return cipher.doFinal(data);

    }

    /**
     * 公钥加密
     *
     * @param data
     * @param certificatePath
     * @return
     * @throws Exception
     */
    public static byte[] encryptByPublicKey(byte[] data, String certificatePath)
            throws Exception {

        // 取得公钥
        PublicKey publicKey = getPublicKey(certificatePath);
        // 对数据加密
        Cipher cipher = Cipher.getInstance(publicKey.getAlgorithm());
        cipher.init(Cipher.ENCRYPT_MODE, publicKey);

        return cipher.doFinal(data);
```

```
    }

    /**
     * 公钥解密
     *
     * @param data
     * @param certificatePath
     * @return
     * @throws Exception
     */
    public static byte[] decryptByPublicKey(byte[] data, String certificatePath)
            throws Exception {
        // 取得公钥
        PublicKey publicKey = getPublicKey(certificatePath);

        // 对数据加密
        Cipher cipher = Cipher.getInstance(publicKey.getAlgorithm());
        cipher.init(Cipher.DECRYPT_MODE, publicKey);

        return cipher.doFinal(data);

    }

    /**
     * 验证Certificate
     *
     * @param certificatePath
     * @return
     */
    public static boolean verifyCertificate(String certificatePath) {
        return verifyCertificate(new Date(), certificatePath);
    }

    /**
     * 验证Certificate是否过期或无效
     *
     * @param date
     * @param certificatePath
     * @return
     */
    public static boolean verifyCertificate(Date date, String certificatePath) {
        boolean status = true;
        try {
            // 取得证书
```

```
            Certificate certificate = getCertificate(certificatePath);
            // 验证证书是否过期或无效
            status = verifyCertificate(date, certificate);
        } catch (Exception e) {
            status = false;
        }
        return status;
    }

    /**
     * 验证证书是否过期或无效
     *
     * @param date
     * @param certificate
     * @return
     */
    private static boolean verifyCertificate(Date date, Certificate certificate)
{
        boolean status = true;
        try {
            X509Certificate x509Certificate = (X509Certificate) certificate;
            x509Certificate.checkValidity(date);
        } catch (Exception e) {
            status = false;
        }
        return status;
    }

    /**
     * 签名
     *
     * @param keyStorePath
     * @param alias
     * @param password
     *
     * @return
     * @throws Exception
     */
    public static String sign(byte[] sign, String keyStorePath, String alias,
            String password) throws Exception {
        // 获得证书
        X509Certificate x509Certificate = (X509Certificate) getCertificate(
                keyStorePath, alias, password);
        // 获取私钥
```

```
        KeyStore ks = getKeyStore(keyStorePath, password);
        // 取得私钥
        PrivateKey privateKey = (PrivateKey) ks.getKey(alias, password
                .toCharArray());

        // 构建签名
        Signature signature = Signature.getInstance(x509Certificate
                .getSigAlgName());
        signature.initSign(privateKey);
        signature.update(sign);
        return encryptBASE64(signature.sign());
    }

    /**
     * 验证签名
     *
     * @param data
     * @param sign
     * @param certificatePath
     * @return
     * @throws Exception
     */
    public static boolean verify(byte[] data, String sign,
            String certificatePath) throws Exception {
        // 获得证书
        X509Certificate x509Certificate = (X509Certificate)
getCertificate(certificatePath);
        // 获得公钥
        PublicKey publicKey = x509Certificate.getPublicKey();
        // 构建签名
        Signature signature = Signature.getInstance(x509Certificate
                .getSigAlgName());
        signature.initVerify(publicKey);
        signature.update(data);

        return signature.verify(decryptBASE64(sign));

    }

    /**
     * 验证Certificate
     *
     * @param keyStorePath
     * @param alias
```

```
     * @param password
     * @return
     */
    public static boolean verifyCertificate(Date date, String keyStorePath,
            String alias, String password) {
        boolean status = true;
        try {
            Certificate certificate = getCertificate(keyStorePath, alias,
                    password);
            status = verifyCertificate(date, certificate);
        } catch (Exception e) {
            status = false;
        }
        return status;
    }

    /**
     * 验证Certificate
     *
     * @param keyStorePath
     * @param alias
     * @param password
     * @return
     */
    public static boolean verifyCertificate(String keyStorePath, String alias,
            String password) {
        return verifyCertificate(new Date(), keyStorePath, alias, password);
    }

    /**
     * 获得SSLSocektFactory
     *
     * @param password
     *            密码
     * @param keyStorePath
     *            密钥库路径
     *
     * @param trustKeyStorePath
     *            信任库路径
     * @return
     * @throws Exception
     */
    private static SSLSocketFactory getSSLSocketFactory(String password,
            String keyStorePath, String trustKeyStorePath) throws Exception {
```

```
        // 初始化密钥库
        KeyManagerFactory keyManagerFactory = KeyManagerFactory
                .getInstance(SunX509);
        KeyStore keyStore = getKeyStore(keyStorePath, password);
        keyManagerFactory.init(keyStore, password.toCharArray());

        // 初始化信任库
        TrustManagerFactory trustManagerFactory = TrustManagerFactory
                .getInstance(SunX509);
        KeyStore trustkeyStore = getKeyStore(trustKeyStorePath, password);
        trustManagerFactory.init(trustkeyStore);

        // 初始化 SSL 上下文
        SSLContext ctx = SSLContext.getInstance(SSL);
        ctx.init(keyManagerFactory.getKeyManagers(), trustManagerFactory
                .getTrustManagers(), null);
        SSLSocketFactory sf = ctx.getSocketFactory();

        return sf;
    }

    /**
     * 为 HttpsURLConnection 配置 SSLSocketFactory
     *
     * @param conn
     *            HttpsURLConnection
     * @param password
     *            密码
     * @param keyStorePath
     *            密钥库路径
     *
     * @param trustKeyStorePath
     *            信任库路径
     * @throws Exception
     */
    public static void configSSLSocketFactory(HttpsURLConnection conn,
            String password, String keyStorePath, String trustKeyStorePath)
            throws Exception {
        conn.setSSLSocketFactory(getSSLSocketFactory(password, keyStorePath,
                trustKeyStorePath));
    }
}
```

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

```
 import static org.junit.Assert.*;

import java.io.DataInputStream;
import java.io.InputStream;
import java.net.URL;

import javax.net.ssl.HttpsURLConnection;

import org.junit.Test;

/**
 *
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public class CertificateCoderTest {
    private String password = "123456";
    private String alias = "www.zlex.org";
    private String certificatePath = "d:/zlex.cer";
    private String keyStorePath = "d:/zlex.keystore";
    private String clientKeyStorePath = "d:/zlex-client.keystore";
    private String clientPassword = "654321";

    @Test
    public void test() throws Exception {
        System.err.println("公钥加密——私钥解密");
        String inputStr = "Ceritifcate";
        byte[] data = inputStr.getBytes();

        byte[] encrypt = CertificateCoder.encryptByPublicKey(data,
                certificatePath);

        byte[] decrypt = CertificateCoder.decryptByPrivateKey(encrypt,
                keyStorePath, alias, password);
```

```
        String outputStr = new String(decrypt);

        System.err.println("加密前:" + inputStr + "\n\r" + "解密后:" + outputStr);

        // 验证数据一致
        assertArrayEquals(data, decrypt);

        // 验证证书有效
        assertTrue(CertificateCoder.verifyCertificate(certificatePath));

    }

    @Test
    public void testSign() throws Exception {
        System.err.println("私钥加密——公钥解密");

        String inputStr = "sign";
        byte[] data = inputStr.getBytes();

        byte[] encodedData = CertificateCoder.encryptByPrivateKey(data,
                keyStorePath, alias, password);

        byte[] decodedData = CertificateCoder.decryptByPublicKey(encodedData,
                certificatePath);

        String outputStr = new String(decodedData);
        System.err.println("加密前:" + inputStr + "\n\r" + "解密后:" + outputStr);
        assertEquals(inputStr, outputStr);

        System.err.println("私钥签名——公钥验证签名");
        // 产生签名
        String sign = CertificateCoder.sign(encodedData, keyStorePath, alias,
                password);
        System.err.println("签名:\r" + sign);

        // 验证签名
        boolean status = CertificateCoder.verify(encodedData, sign,
                certificatePath);
        System.err.println("状态:\r" + status);
        assertTrue(status);

    }

    @Test
    public void testHttps() throws Exception {
```

```
        URL url = new URL("https://www.zlex.org/examples/");
        HttpsURLConnection conn = (HttpsURLConnection) url.openConnection();

        conn.setDoInput(true);
        conn.setDoOutput(true);

        CertificateCoder.configSSLSocketFactory(conn, clientPassword,
                clientKeyStorePath, clientKeyStorePath);

        InputStream is = conn.getInputStream();

        int length = conn.getContentLength();

        DataInputStream dis = new DataInputStream(is);
        byte[] data = new byte[length];
        dis.readFully(data);

        dis.close();
        System.err.println(new String(data));
        conn.disconnect();
    }
}
```

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

```
<!--
 Licensed to the Apache Software Foundation (ASF) under one or more
 contributor license agreements.  See the NOTICE file distributed with
 this work for additional information regarding copyright ownership.
 The ASF licenses this file to You under the Apache License, Version 2.0
 (the "License"); you may not use this file except in compliance with
 the License.  You may obtain a copy of the License at

     http://www.apache.org/licenses/LICENSE-2.0

 Unless required by applicable law or agreed to in writing, software
 distributed under the License is distributed on an "AS IS" BASIS,
 WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 See the License for the specific language governing permissions and
 limitations under the License.
-->
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
```

```
<HTML><HEAD><TITLE>Apache Tomcat Examples</TITLE>
<META http-equiv=Content-Type content="text/html">
</HEAD>
<BODY>
<P>
<H3>Apache Tomcat Examples</H3>
<P></P>
<ul>
<li><a href="servlets">Servlets examples</a></li>
<li><a href="jsp">JSP Examples</a></li>
</ul>
</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！

在Java 加密技术（九）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

```
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

```
 输入keystore密码:
再次输入新密码:
您的名字与姓氏是什么?
```

```
  [Unknown]:   www.zlex.org
您的组织单位名称是什么？
  [Unknown]:   zlex
您的组织名称是什么？
  [Unknown]:   zlex
您所在的城市或区域名称是什么？
  [Unknown]:   BJ
您所在的州或省份名称是什么？
  [Unknown]:   BJ
该单位的两字母国家代码是什么
  [Unknown]:   CN
CN=www.zlex.org, OU=zlex, O=zlex, L=BJ, ST=BJ, C=CN 正确吗？
  [否]:   Y

输入<tomcat>的主密码
        （如果和 keystore 密码相同，按回车）：
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

```
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

```
 -----BEGIN NEW CERTIFICATE REQUEST-----
MIIBnDCCAQUCAQAwXDELMAkGA1UEBhMCQ04xCzAJBgNVBAgTAkJKMQswCQYDVQQHEwJCSjENMAsG
A1UEChMEemxleDENMAsGA1UECxMEemxleDEVMBMGA1UEAxMMd3d3LnpsZXgub3JnMIGfMA0GCSqG
SIb3DQEBAQUAA4GNADCBiQKBgQCR6DXU9Mp+mCKO7cv9JPsj0n1Ec/GpM09qvhpgX3FNad/ZWSDc
vU77YXZSoF9hQp3w1LC+eeKgd2MlVpXTvbVwBNVd2HiQPp37ic6BUUjSaX8LHtCl7l0BIEye9qQ2
j8G0kak7e8ZA0s7nb3Ymq/K8BV7v0MQIdhIc1bifK9ZDewIDAQABoAAwDQYJKoZIhvcNAQEFBQAD
gYEAMA1r2fbZPtNx37U9TRwadCH2TZZecwKJS/hskNm6ryPKIAp9APWwAyj8WJHRBz5SpZM4zmYO
oMCI8BcnY2A4JP+R7/SwXTdH/xcg7NVghd9A2SCgqMpF7KMfc5dE3iygdiPu+UhY200Dvpjx8gmJ
1UbH3+nqMUyCrZgURFsl0UY=
-----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

```
 -----BEGIN PKCS7-----
MIIF3AYJKoZIhvcNAQcCoIIFzTCCBckCAQExADALBgkqhkiG9w0BBwGgggWxMIID
EDCCAnmgAwIBAgIQA/mx/pKoaB+KGX2hveFU9zANBgkqhkiG9w0BAQUFADCBhzEL
MAkGA1UEBhMCWkExIjAgBgNVBAgTGUZPUiBURVNUSU5HIFBVUlBPU0VTIE9OTFkx
HTAbBgNVBAoTFFRoYXd0ZSBDZXJ0aWZpY2F0aW9uMRcwFQYDVQQLEw5URVNUIFRF
U1QgVEVTVDEcMBoGA1UEAxMTVGhhd3RlIFRlc3QgQ0EgUm9vdDAeFw0wOTA1Mjgw
MDIxMzlaFw0wOTA2MTgwMDIxMzlaMFwxCzAJBgNVBAYTAkNOMQswCQYDVQQIEwJC
SjELMAkGA1UEBxMCQkoxDTALBgNVBAoTBHpsZXgxDTALBgNVBAsTBHpsZXgxFTAT
BgNVBAMTDHd3dy56bGV4Lm9yZzCBnzANBgkqhkiG9w0BAQEFAAOBjQAwgYkCgYEA
keg11PTKfpgiju3L/ST7I9J9RHPxqTNPar4aYF9xTWnf2Vkg3L1O+2F2UqBfYUKd
8NSwvnnioHdjJVaV0721cATVXdh4kD6d+4n0gVFI0ml/Cx7Qpe5dASBMnvakNo/B
tJGpO3vGQNLO5292JqvyvAVe79DECHYSHNW4nyvWQ3sCAwEAAaOBpjCBozAMBgNV
HRMBAf8EAjAAMB0GA1UdJQQWMBQGCCsGAQUFBwMBBggrBgEFBQcDAjBABgNVHR8E
OTA3MDWgM6Axhi9odHRwOi8vY3JsLnRoYXd0ZS5jb20vVGhhd3RlUHJlbWl1bVNl
cnZlckNBLmNybDAyBggrBgEFBQcBAQQmMCQwIgYIKwYBBQUHMAGGFmh0dHA6Ly9v
Y3NwLnRoYXd0ZS5jb20wDQYJKoZIhvcNAQEFBQADgYEATPuxZbtJJSPmXvfrr1yz
xqM06IwTZ6UUOlZRG7I0WufMjNMKdpn8hklUhE17mxAhGSpewLVVeLR7uzBLFkuC
X7wMXxhoYdJZtNai72izU6Rd1oknao7diahvRxPK4IuQ7y2oZ511/4T4vgY6iRAj
q4q76HhPJrVRL/sduaiu+gYwggKZMIICAqADAgECAgEAMA0GCSqGSIb3DQEBBAUA
MIGHMQswCQYDVQQGEwJaQTEiMCAGA1UECBMZRk9SIFRFU1RJTkcgUFVSUE9TRVMg
T05MWTEdMBsGA1UEChMUVGhhd3RlIENlcnRpZmljYXRpb24xFzAVBgNVBAsTDlRF
U1QgVEVTVCBURVNUMRwwGgYDVQQDExNUaGF3dGUgVGVzdCBDQSBSb290MB4XDTk2
MDgwMTAwMDAwMFoXDTIwMTIzMTIxNTk1OVowgYcxCzAJBgNVBAYTAlpBMSIwIAYD
VQQIExlGT1IgVEVTVElORyBQVVJQT1NFUyBPTkxZMR0wGwYDVQQKExRUaGF3dGUg
Q2VydGlmaWNhdGlvbjEXMBUGA1UECxMOVEVTVCBURVNUIFRFU1QxHDAaBgNVBAMT
E1RoYXd0ZSBUZXN0IENBIFJvb3QwgZ8wDQYJKoZIhvcNAQEBBQADgY0AMIGJAoGB
ALV9kG+Os6x/DOhm+tKUQfzVMWGhE95sFmEtkMMTX2Zi4n6i6BvzoReJ5njzt1LF
cqu4EUk9Ji20egKKfmqRzmQFLP7+1niSdfJEUE7cKY40QoI99270PTrLjJeaMcCl
+AYl+kD+RL5BtuKKU3PurYcsCsre6aTvjMcqpTJOGeSPAgMBAAGjEzarmA8GA1Ud
EwEB/wQFMAMBAf8wDQYJKoZIhvcNAQEEBQADgYEAgozj7BkD9O8si2V0v+EZ/t7E
fz/LC8y6mD7IBUziHy5/53ymGAGLtyhXHvX+UIE6UWbHro3IqVkrmY5uC93Z2Wew
A/6edK3KFUcUikrLeewM7gmqsiASEKx2mKRKlu12jXyNS5tXrPWRDvUKtFC1uL9a
12rFAQS2BkIk7aU+ghYxAA==
-----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

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

```
 输入 keystore 密码：

回复中的最高级认证：

所有者:CN=Thawte Test CA Root, OU=TEST TEST TEST, O=Thawte Certification, ST=FOR
 TESTING PURPOSES ONLY, C=ZA
签发人:CN=Thawte Test CA Root, OU=TEST TEST TEST, O=Thawte Certification, ST=FOR
 TESTING PURPOSES ONLY, C=ZA
序列号:0
有效期: Thu Aug 01 08:00:00 CST 1996 至 Fri Jan 01 05:59:59 CST 2021
证书指纹:
         MD5:5E:E0:0E:1D:17:B7:CA:A5:7D:36:D6:02:DF:4D:26:A4
         SHA1:39:C6:9D:27:AF:DC:EB:47:D6:33:36:6A:B2:05:F1:47:A9:B4:DA:EA
         签名算法名称:MD5withRSA
         版本: 3

扩展:

#1: ObjectId: 2.5.29.19 Criticality=true
BasicConstraints:[
  CA:true
  PathLen:2147483647
]


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

7.域名定位

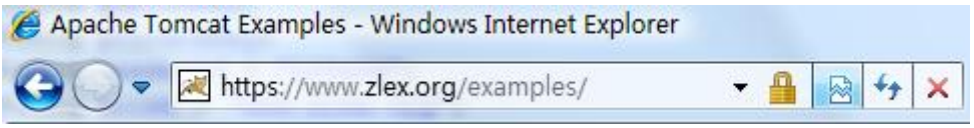
将域名www.zlex.org定位到本机上。打开C：\Windows\System32\drivers\etc\hosts文件，将www.zlex.org绑定在本机上。在文件末尾追加 127.0.0.1 www.zlex.org.现在通过地址栏访问http://www.zlex.org，或者通过ping命令，如果能够定位到本机，域名映射就搞定了。

8.配置 server.xml

Xml 代码

```
<Connector
    keystoreFile="conf/zlex.keystore"
    keystorePass="123456"
    truststoreFile="conf/zlex.keystore"
    truststorePass="123456"
    SSLEnabled="true"
    URIEncoding="UTF-8"
    clientAuth="false"
    maxThreads="150"
    port="443"
    protocol="HTTP/1.1"
    scheme="https"
    secure="true"
    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：



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

```
 import static org.junit.Assert.*;

import java.io.DataInputStream;
import java.io.InputStream;
import java.net.URL;

import javax.net.ssl.HttpsURLConnection;

import org.junit.Test;

/**
 *
 * @author 梁栋
 * @version 1.0
 * @since 1.0
```

```
 */
public class CertificateCoderTest {
    private String password = "123456";
    private String alias = "www.zlex.org";
    private String certificatePath = "d:/zlex.cer";
    private String keyStorePath = "d:/zlex.keystore";

    @Test
    public void test() throws Exception {
        System.err.println("公钥加密——私钥解密");
        String inputStr = "Ceritifcate";
        byte[] data = inputStr.getBytes();

        byte[] encrypt = CertificateCoder.encryptByPublicKey(data,
                certificatePath);

        byte[] decrypt = CertificateCoder.decryptByPrivateKey(encrypt,
                keyStorePath, alias, password);
        String outputStr = new String(decrypt);

        System.err.println("加密前: " + inputStr + "\n\r" + "解密后: " + outputStr);

        // 验证数据一致
        assertArrayEquals(data, decrypt);

        // 验证证书有效
        assertTrue(CertificateCoder.verifyCertificate(certificatePath));

    }

    @Test
    public void testSign() throws Exception {
        System.err.println("私钥加密——公钥解密");

        String inputStr = "sign";
        byte[] data = inputStr.getBytes();

        byte[] encodedData = CertificateCoder.encryptByPrivateKey(data,
                keyStorePath, alias, password);

        byte[] decodedData = CertificateCoder.decryptByPublicKey(encodedData,
                certificatePath);

        String outputStr = new String(decodedData);
        System.err.println("加密前: " + inputStr + "\n\r" + "解密后: " + outputStr);
```

```
        assertEquals(inputStr, outputStr);

        System.err.println("私钥签名——公钥验证签名");
        // 产生签名
        String sign = CertificateCoder.sign(encodedData, keyStorePath, alias,
                password);
        System.err.println("签名:\r" + sign);

        // 验证签名
        boolean status = CertificateCoder.verify(encodedData, sign,
                certificatePath);
        System.err.println("状态:\r" + status);
        assertTrue(status);

    }

    @Test
    public void testHttps() throws Exception {
        URL url = new URL("https://www.zlex.org/examples/");
        HttpsURLConnection conn = (HttpsURLConnection) url.openConnection();

        conn.setDoInput(true);
        conn.setDoOutput(true);

        CertificateCoder.configSSLSocketFactory(conn, password, keyStorePath,
                keyStorePath);

        InputStream is = conn.getInputStream();

        int length = conn.getContentLength();

        DataInputStream dis = new DataInputStream(is);
        byte[] data = new byte[length];
        dis.readFully(data);

        dis.close();
        conn.disconnect();
        System.err.println(new String(data));
    }
}
```

再次执行，验证通过！

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。

本文主要谈一下密码学中的加密和数字签名，以及其在java中如何进行使用。对密码学有兴趣的伙伴，推荐看Bruce Schneier的著作：Applied Crypotography.在jdk1.5的发行版本中安全性方面有了很大的改进，也提供了对RSA算法的直接支持，现在我们从实例入手解决问题（本文仅是作为简单介绍）：

一、密码学上常用的概念

1）消息摘要：

这是一种与消息认证码结合使用以确保消息完整性的技术。主要使用单向散列函数算法，可用于检验消息的完整性，和通过散列密码直接以文本形式保存等，目前广泛使用的算法有 MD4、MD5、SHA-1，jdk1.5 对上面都提供了支持，在 java 中进行消息摘要很简单，java.security.MessageDigest 提供了一个简易的操作方法：

Java代码

```
/**
*MessageDigestExample.java
*Copyright 2005-2-16
*/
import java.security.MessageDigest;
/**
*单一的消息摘要算法，不使用密码.可以用来对明文消息（如：密码）隐藏保存
*/
public class MessageDigestExample{
  public static void main(String[] args) throws Exception{
    if(args.length!=1){
      System.err.println("Usage:java MessageDigestExample text");
      System.exit(1);
    }
    byte[] plainText=args[0].getBytes("UTF8");
    //使用 getInstance("算法")来获得消息摘要,这里使用 SHA-1 的 160 位算法
    MessageDigest messageDigest=MessageDigest.getInstance("SHA-1");
    System.out.println("
"+messageDigest.getProvider().getInfo());
    //开始使用算法
    messageDigest.update(plainText);
    System.out.println("
Digest:");
    //输出算法运算结果
    System.out.println(new String(messageDigest.digest(),"UTF8"));
  }
}
```

还可以通过消息认证码来进行加密实现，javax.crypto.Mac 提供了一个解决方案，有兴趣者可以参考相关 API 文档，本文只是简单介绍什么是摘要算法。

2）私钥加密：

消息摘要只能检查消息的完整性，但是单向的，对明文消息并不能加密，要加密明文的消息的话，就要使用其他的算法，要确保机密性，我们需要使用私钥密码术来交换私有消息。

这种最好理解，使用对称算法。比如：A 用一个密钥对一个文件加密，而 B 读取这个文件的话，则需要和 A 一样的密钥，双方共享一个私钥（而在 web 环境下，私钥在传递时容易被侦听）：

使用私钥加密的话，首先需要一个密钥，可用 javax.crypto.KeyGenerator 产生一个密钥（java.security.Key），然后传递给一个加密工具（javax.crypto.Cipher），该工具再使用相应的算法来进行加密，主要对称算法有：DES（实际密钥只用到 56 位），AES（支持三种密钥长度：128、192、256 位），通常首先 128 位，其他的还有 DESede 等，jdk1.5 种也提供了对对称算法的支持，以下例子使用 AES 算法来加密：

Java代码

```
/**
*PrivateExmaple.java
*Copyright 2005-2-16
*/
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import java.security.Key;
/**
*私鈅加密，保证消息机密性
*/
public class PrivateExample{
  public static void main(String[] args) throws Exception{
    if(args.length!=1){
      System.err.println("Usage:java PrivateExample <text>");
      System.exit(1);
    }
    byte[] plainText=args[0].getBytes("UTF8");
    //通过 KeyGenerator 形成一个 key
    System.out.println("
Start generate AES key");
    KeyGenerator keyGen=KeyGenerator.getInstance("AES");
    keyGen.init(128);
    Key key=keyGen.generateKey();
    System.out.println("Finish generating DES key");
```

```
    //获得一个私钥加密类 Cipher，ECB 是加密方式，PKCS5Padding 是填充方法
    Cipher cipher=Cipher.getInstance("AES/ECB/PKCS5Padding");
    System.out.println("
"+cipher.getProvider().getInfo());
    //使用私钥加密
    System.out.println("
Start encryption:");
    cipher.init(Cipher.ENCRYPT_MODE,key);
    byte[] cipherText=cipher.doFinal(plainText);
    System.out.println("Finish encryption:");
    System.out.println(new String(cipherText,"UTF8"));
    System.out.println("
Start decryption:");
    cipher.init(Cipher.DECRYPT_MODE,key);
    byte[] newPlainText=cipher.doFinal(cipherText);
    System.out.println("Finish decryption:");
    System.out.println(new String(newPlainText,"UTF8"));
  }
}
```

3）公钥加密：

上面提到，私钥加密需要一个共享的密钥，那么如何传递密钥呢？web 环境下，直接传递的话很容易被侦听到，幸好有了公钥加密的出现。公钥加密也叫不对称加密，不对称算法使用一对密钥对，一个公钥，一个私钥，使用公钥加密的数据，只有私钥能解开（可用于加密）；同时，使用私钥加密的数据，只有公钥能解开（签名）。但是速度很慢（比私钥加密慢 100 到 1000 倍），公钥的主要算法有 RSA，还包括 Blowfish，Diffie-Helman 等，jdk1.5 种提供了对 RSA 的支持，是一个改进的地方：

Java代码

```
/**
*PublicExample.java
*Copyright 2005-2-16
*/
import java.security.Key;
import javax.crypto.Cipher;
import java.security.KeyPairGenerator;
import java.security.KeyPair;
/**
*一个简单的公钥加密例子,Cipher 类使用 KeyPairGenerator 生成的公钥和私钥
*/
public class PublicExample{
```

```
  public static void main(String[] args) throws Exception{
    if(args.length!=1){
      System.err.println("Usage:java PublicExample <text>");
      System.exit(1);
    }
    byte[] plainText=args[0].getBytes("UTF8");
    //构成一个 RSA 密钥
    System.out.println("
Start generating RSA key");
    KeyPairGenerator keyGen=KeyPairGenerator.getInstance("RSA");
    keyGen.initialize(1024);
    KeyPair key=keyGen.generateKeyPair();
    System.out.println("Finish generating RSA key");
    //获得一个 RSA 的 Cipher 类，使用公鈅加密
    Cipher cipher=Cipher.getInstance("RSA/ECB/PKCS1Padding");
    System.out.println("
"+cipher.getProvider().getInfo());
    System.out.println("
Start encryption");
    cipher.init(Cipher.ENCRYPT_MODE,key.getPublic());
    byte[] cipherText=cipher.doFinal(plainText);
    System.out.println("Finish encryption:");
    System.out.println(new String(cipherText,"UTF8"));
    //使用私鈅解密
    System.out.println("
Start decryption");
    cipher.init(Cipher.DECRYPT_MODE,key.getPrivate());
    byte[] newPlainText=cipher.doFinal(cipherText);
    System.out.println("Finish decryption:");
    System.out.println(new String(newPlainText,"UTF8"));
  }
}
```

4）数字签名：

数字签名，它是确定交换消息的通信方身份的第一个级别。上面A通过使用公钥加密数据后发给B，B利用私钥解密就得到了需要的数据，问题来了，由于都是使用公钥加密，那么如何检验是A发过来的消息呢？上面也提到了一点，私钥是唯一的，那么A就可以利用A自己的私钥进行加密，然后B再利用A的公钥来解密，就可以了；数字签名的原理就基于此，而通常为了证明发送数据的真实性，通过利用消息摘要获得简短的消息内容，然后再利用私钥进行加密散列数据和消息一起发送。java中为数字签名提供了良好的支持，
java.security.Signature类提供了消息签名：

Java代码

```
/**
*DigitalSignature2Example.java
*Copyright 2005-2-16
*/
import java.security.Signature;
import java.security.KeyPairGenerator;
import java.security.KeyPair;
import java.security.SignatureException;
/**
*数字签名，使用RSA私钥对对消息摘要签名，然后使用公鈅验证 测试
*/
public class DigitalSignature2Example{
  public static void main(String[] args) throws Exception{
    if(args.length!=1){
      System.err.println("Usage:java DigitalSignature2Example <text>");
      System.exit(1);
    }
    byte[] plainText=args[0].getBytes("UTF8");
    //形成RSA公钥对
    System.out.println("
Start generating RSA key");
    KeyPairGenerator keyGen=KeyPairGenerator.getInstance("RSA");
    keyGen.initialize(1024);
    KeyPair key=keyGen.generateKeyPair();
    System.out.println("Finish generating RSA key");
    //使用私鈅签名
    Signature sig=Signature.getInstance("SHA1WithRSA");
    sig.initSign(key.getPrivate());
    sig.update(plainText);
    byte[] signature=sig.sign();
    System.out.println(sig.getProvider().getInfo());
    System.out.println("
Signature:");
    System.out.println(new String(signature,"UTF8"));
    //使用公鈅验证
    System.out.println("
Start signature verification");
    sig.initVerify(key.getPublic());
    sig.update(plainText);
    try{
      if(sig.verify(signature)){
        System.out.println("Signature verified");
      }else System.out.println("Signature failed");
      }catch(SignatureException e){
```

```
        System.out.println("Signature failed");
      }
    }
```

5)数字证书。

还有个问题，就是公钥问题，A 用私钥加密了，那么 B 接受到消息后，用 A 提供的公钥解密；那么现在有个讨厌的 C，他把消息拦截了，然后用自己的私钥加密，同时把自己的公钥发给 B，并告诉 B，那是 A 的公钥，结果……，这时候就需要一个中间机构出来说话了（相信权威，我是正确的），就出现了 Certificate Authority（也即 CA），有名的 CA 机构有 Verisign 等，目前数字认证的工业标准是：CCITT 的 X.509：

数字证书：它将一个身份标识连同公钥一起进行封装，并由称为认证中心或 CA 的第三方进行数字签名。

密钥库：java 平台为你提供了密钥库，用作密钥和证书的资源库。从物理上讲，密钥库是缺省名称为 .keystore 的文件（有一个选项使它成为加密文件）。密钥和证书可以拥有名称（称为别名），每个别名都由唯一的密码保护。密钥库本身也受密码保护；您可以选择让每个别名密码与主密钥库密码匹配。

使用工具 keytool，我们来做一件自我认证的事情吧（相信我的认证）：

1、创建密钥库 keytool -genkey -v -alias feiUserKey -keyalg RSA 默认在自己的 home 目录下(windows 系统是 c:documents and settings<你的用户名> 目录下的。keystore 文件），创建我们用 RSA 算法生成别名为 feiUserKey 的自签名的证书，如果使用了 -keystore mm 就在当前目录下创建一个密钥库 mm 文件来保存密钥和证书。

2、查看证书：keytool -list 列举了密钥库的所有的证书

也可以在 dos 下输入 keytool -help 查看帮助。

二、JAR 的签名

我们已经学会了怎样创建自己的证书了，现在可以开始了解怎样对JAR文件签名，JAR 文件在Java中相当于 ZIP 文件，允许将多个 Java 类文件打包到一个具有 .jar 扩展名的文件中，然后可以对这个jar文件进行数字签名，以证实其来源和真实性。该 JAR 文件的接收方可以根据发送方的签名决定是否信任该代码，并可以确信该内容在接收之前没有被篡改过。同时在部署中，可以通过在策略文件中放置访问控制语句根据签名者的身份分配对机器资源的访问权。这样，有些Applet的安全检验访问就得以进行。

使用 jarsigner 工具可以对 jar 文件进行签名：

现在假设我们有个 Test.jar 文件（可以使用 jar 命令行工具生成）：

jarsigner Test.jar feiUserKey （这里我们上面创建了该别名的证书），详细信息可以输入 jarsigner 查看帮助

验证其真实性：jarsigner -verify Test.jar（注意，验证的是 jar 是否被修改了，但不检验减少的，如果增加了新的内容，也提示，但减少的不会提示。）

使用 Applet 中：<applet code="Test.class" archive="Test.jar" width="150" height="100"></applet>然后浏览器就会提示你：准许这个会话-拒绝-始终准许-查看证书等。

三、安全套接字层（SSL Secure Sockets Layer）和传输层安全性（TLS Transport Layer Security）

安全套接字层和传输层安全性是用于在客户机和服务器之间构建安全的通信通道的协议。它也用来为客户机认证服务器，以及（不太常用的）为服务器认证客户机。该协议在浏览器应用程序中比较常见，浏览器窗口底部的锁表明 SSL/TLS 有效：

1）当使用 SSL/TLS（通常使用 https:// URL）向站点进行请求时，从服务器向客户机发送一个证书。客户机使用已安装的公共 CA 证书通过这个证书验证服务器的身份，然后检查 IP 名称（机器名）与客户机连接的机器是否匹配。

2）客户机生成一些可以用来生成对话的私钥（称为会话密钥）的随机信息，然后用服务器的公钥对它加密并将它发送到服务器。服务器用自己的私钥解密消息，然后用该随机信息派生出和客户机一样的私有会话密钥。通常在这个阶段使用 RSA 公钥算法。

3）客户机和服务器使用私有会话密钥和私钥算法（通常是 RC4）进行通信。使用另一个密钥的消息认证码来确保消息的完整性。

java 中 javax.net.ssl.SSLServerSocketFactory 类提供了一个很好的 SSLServerSocker 的工厂类，熟悉 Socket 编程的读者可以去练习。当编写完服务器端之后，在浏览器上输入 https://主机名：端口 就会通过 SSL/TLS 进行通话了。注意：运行服务端的时候要带系统环境变量运行：javax.net.ssl.keyStore=密钥库（创建证书时，名字应该为主机名，比如 localhost）和 javax.net.ssl.keyStorePassword=你的密码

Java 代码

```
package security;

import java.security.*;
import java.security.PublicKey;
import java.io.*;
import java.util.*;
import java.security.*;
import java.security.cert.*;
import sun.security.x509.*;
```

```
import java.security.cert.Certificate;
import java.security.cert.CertificateFactory;

/**
 * 此例是对“数字证书”文件的操作
 *java 平台（在机器上安装 jdk）为你提供了密钥库(证书库)，cmd 下提供了 keytool 命令就可以创建证书库
 *
 * 在运行此例前：
 * 在 c 盘目录下创建一个证书，指定证书库为 BocsoftKeyLib，创建别名为 TestCertification 的一条证书，它指定用
 * RSA 算法生成，且指定密钥长度为 1024，证书有效期为 1 年
 * 导出证书文件为 TC.cer 已存于本地磁盘 C:/
 * 密码是 keystore
 */

public class DigitalCertificate {

    public static void main(String[] args) {
        try {
            //前提：将证书库中的一条证书导出到证书文件(我写的例子里证书文件叫 TC.cer)
            //从证书文件 TC.cer 里读取证书信息
            /*CertificateFactory cf = CertificateFactory.getInstance("X.509");
            FileInputStream in = new FileInputStream("C:/TC.cer");
            //将文件以文件流的形式读入证书类 Certificate 中
            Certificate c = cf.generateCertificate(in);
            System.err.println("转换成 String 后的证书信息：
"+c.toString());*/


            //或者不用上面代码的方法，直接从证书库中读取证书信息，和上面的结果一摸一样
            String pass="keystore";
            FileInputStream in2=new FileInputStream("C:/BocsoftKeyLib");
            KeyStore ks=KeyStore.getInstance("JKS");
            ks.load(in2,pass.toCharArray());
            String alias = "TestCertification"; //alias 为条目的别名
            Certificate c=ks.getCertificate(alias);
            System.err.println("转换成 String 后的证书信息：
"+c.toString());


            //获取获取 X509Certificate 类型的对象，这是证书类获取 Certificate
```

```
的子类，实现了更多方法
            X509Certificate t=(X509Certificate)c;
            //从信息中提取需要信息
            System.out.println("版本号:"+t.getVersion());
            System.out.println("序列号:"+t.getSerialNumber().toString(16));
            System.out.println("主体名："+t.getSubjectDN());
            System.out.println("签发者："+t.getIssuerDN());
            System.out.println("有效期："+t.getNotBefore());
            System.out.println("签名算法："+t.getSigAlgName());
            byte [] sig=t.getSignature();//签名值
            PublicKey pk = t.getPublicKey();
            byte [] pkenc=pk.getEncoded();
            System.out.println("公钥：");
            for(int i=0;i<pkenc.length;i++){
                System.out.print(pkenc[i]+",");
            }
            System.err.println();


            //证书的日期有效性检查，颁发的证书都有一个有效性的日期区间
            Date TimeNow=new Date();
            t.checkValidity(TimeNow);
            System.out.println("证书的日期有效性检查:有效的证书日期！");


            //验证证书签名的有效性,通过数字证书认证中心(CA)机构颁布给客户的
CA 证书，比如：caroot.crt 文件
            //我手里没有 CA 颁给我的证书，所以下面代码执行不了
            /*FileInputStream in3=new FileInputStream("caroot.crt");
            //获取 CA 证书
            Certificate cac = cf.generateCertificate(in3);
            //获取 CA 的公钥
            PublicKey pbk=cac.getPublicKey();
            //c 为本地证书，也就是待检验的证书，用 CA 的公钥校验数字证书 c 的
有效性
            c.verify(pbk);*/


        } catch(CertificateExpiredException e){//证书的日期有效性检查:过期
            System.out.println("证书的日期有效性检查:过期");
        } catch(CertificateNotYetValidException e){ //证书的日期有效性检查:
尚未生效
            System.out.println("证书的日期有效性检查:尚未生效");
        } catch (CertificateException ce) {
```

```
            ce.printStackTrace();
        } catch (FileNotFoundException fe) {
            fe.printStackTrace();
        } /*catch (IOException ioe){

        } catch (KeyStoreException kse){

        }*/ catch (Exception e){
            e.printStackTrace();
        }

    }
}
```

Java 中 3DES 加密解密调用示例

在 java 中调用 sun 公司提供的 3DES 加密解密算法时，需要使用到$JAVA_HOME/jre/lib/目录下如下的 4 个 jar 包：

jce.jar
security/US_export_policy.jar
security/local_policy.jar
ext/sunjce_provider.jar

Java运行时会自动加载这些包，因此对于带main函数的应用程序不需要设置到CLASSPATH环境变量中。对于WEB应用，不需要把这些包加到WEB-INF/lib目录下。

以下是 java 中调用 sun 公司提供的 3DES 加密解密算法的样本代码：

```
/*字符串 DESede(3DES) 加密*/
import java.security.*;
import javax.crypto.*;
import javax.crypto.spec.SecretKeySpec;
public class ThreeDes {
private static final String Algorithm = "DESede"; // 定义 加密算法,可用 DES,DESede,Blowfish

//keybyte为加密密钥，长度为 24 字节
//src为被加密的数据缓冲区（源）
public static byte[] encryptMode(byte[] keybyte, byte[] src) {
try {
//生成密钥
SecretKey deskey = new SecretKeySpec(keybyte, Algorithm);
//加密
Cipher c1 = Cipher.getInstance(Algorithm);
c1.init(Cipher.ENCRYPT_MODE, deskey);
return c1.doFinal(src);
}
catch (java.security.NoSuchAlgorithmException e1) {
e1.printStackTrace();
}
catch (javax.crypto.NoSuchPaddingException e2) {
e2.printStackTrace();
}
catch (java.lang.Exception e3) {
e3.printStackTrace();
}
return null;
}

//keybyte为加密密钥，长度为 24 字节
//src为加密后的缓冲区
```

```
public static byte[] decryptMode(byte[] keybyte, byte[] src) {
try {
//生成密钥
SecretKey deskey = new SecretKeySpec(keybyte, Algorithm);
//解密
Cipher c1 = Cipher.getInstance(Algorithm);
c1.init(Cipher.DECRYPT_MODE, deskey);
return c1.doFinal(src);
}
catch (java.security.NoSuchAlgorithmException e1) {
e1.printStackTrace();
}
catch (javax.crypto.NoSuchPaddingException e2) {
e2.printStackTrace();
}
catch (java.lang.Exception e3) {
e3.printStackTrace();
}
return null;
}

//转换成十六进制字符串
public static String byte2hex(byte[] b) {
String hs="";
String stmp="";
for (int n=0;n<b.length;n++) {
stmp=(java.lang.Integer.toHexString(b[n] & 0XFF));
if (stmp.length()==1) hs=hs+"0"+stmp;
else hs=hs+stmp;
if (n<b.length-1) hs=hs+":";
}
return hs.toUpperCase();
}

public static void main(String[] args){

//添加新安全算法,如果用JCE就要把它添加进去
Security.addProvider(new com.sun.crypto.provider.SunJCE());
final byte[] keyBytes = {0x11, 0x22, 0x4F, 0x58,
(byte)0x88, 0x10, 0x40, 0x38, 0x28, 0x25, 0x79, 0x51,
(byte)0xCB, (byte)0xDD, 0x55, 0x66, 0x77, 0x29, 0x74,
(byte)0x98, 0x30, 0x40, 0x36, (byte)0xE2
}; //24 字节的密钥
```

```
String szSrc = "This is a 3DES test. 测试";
System.out.println("加密前的字符串:" + szSrc);

byte[] encoded = encryptMode(keyBytes, szSrc.getBytes());
System.out.println("加密后的字符串:" + new String(encoded));

byte[] srcBytes = decryptMode(keyBytes, encoded);
System.out.println("解密后的字符串:" + (new String(srcBytes)));
}
}
```

Java 中常用的加密算法应用

1. MD5 加密，常用于加密用户名密码，当用户验证时。

```
protected byte[] encrypt(byte[] obj) ...{
    try ...{
    MessageDigest md5 = MessageDigest.getInstance("MD5");
    md5.update(obj);
    return md5.digest();
    } catch (NoSuchAlgorithmException e) ...{
    e.printStackTrace();
    }
    }
```

2. SHA 加密，与 MD5 相似的用法，只是两者的算法不同。

```
protected byte[] encrypt(byte[] obj) ...{
    try ...{
    MessageDigest sha = MessageDigest.getInstance("SHA");
    sha.update(obj);
    return sha.digest();
    } catch (NoSuchAlgorithmException e) ...{
    e.printStackTrace();
    }
    }
```

3. RSA 加密，RAS 加密允许解密。常用于文本内容的加密。

```
import java.security.KeyPair;
    import java.security.KeyPairGenerator;
    import java.security.interfaces.RSAPrivateKey;
    import java.security.interfaces.RSAPublicKey;
    import javax.crypto.Cipher;         /** *//**
    * RSAEncrypt
    *
    * @author maqujun
    * @see
```

```
*/
public class RSAEncrypt ...{
/** *//**
* Main method for RSAEncrypt.
* @param args
*/
public static void main(String[] args) ...{
try ...{
RSAEncrypt encrypt = new RSAEncrypt();
String encryptText = "encryptText";
KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("RSA");
keyPairGen.initialize(1024);
KeyPair keyPair = keyPairGen.generateKeyPair();
// Generate keys
RSAPrivateKey privateKey = (RSAPrivateKey) keyPair.getPrivate();
RSAPublicKey publicKey = (RSAPublicKey) keyPair.getPublic();
byte[] e = encrypt.encrypt(publicKey, encryptText.getBytes());
byte[] de = encrypt.decrypt(privateKey,e);
System.out.println(encrypt.bytesToString(e));
System.out.println(encrypt.bytesToString(de));
} catch (Exception e) ...{
e.printStackTrace();
}
}
/** *//**
* Change byte array to String.
* @return byte[]
*/
protected String bytesToString(byte[] encrytpByte) ...{
String result = "";
for (Byte bytes : encrytpByte) ...{
result += (char) bytes.intValue();
}
return result;
}
/** *//**
* Encrypt String.
* @return byte[]
*/
protected byte[] encrypt(RSAPublicKey publicKey, byte[] obj) ...{
if (publicKey != null) ...{
try ...{
Cipher cipher = Cipher.getInstance("RSA");
cipher.init(Cipher.ENCRYPT_MODE, publicKey);
```

```
return cipher.doFinal(obj);
} catch (Exception e) ...{
e.printStackTrace();
}
}
return null;
}
/** *//**
* Basic decrypt method
* @return byte[]
*/
protected byte[] decrypt(RSAPrivateKey privateKey, byte[] obj) ...{
if (privateKey != null) ...{
try ...{
Cipher cipher = Cipher.getInstance("RSA");
cipher.init(Cipher.DECRYPT_MODE, privateKey);
return cipher.doFinal(obj);
} catch (Exception e) ...{
e.printStackTrace();
}
}
return null;
}
}
```

用 Java 的加密机制来保护你的数据

Java streams 是一个强大的编程工具。java.io包提供了很多标准的流类型，并能很容易的建立自己的流类型。流的一个有用的特点是和链表一样的简单处理过程。将 FileReader和BufferedReader链接起来。我们在用客户机/服务器应用程序的时候也会用到类似的概念。

关键字

对于验证来说，关键字很重要，运行KeyGen来产生一个关键字。我们采用同步方法，所以客户机和服务器必须用相同的关键字。

安全socket

我们从一个简单的类开始，它提供我们在普通 socket 对象之上的加密。构造器创建了变量并初始化了密码：

```
outCipher = Cipher.getInstance(algorithm);
outCipher.init(Cipher.ENCRYPT_MODE, key);
inCipher = Cipher.getInstance(algorithm);
inCipher.init(Cipher.DECRYPT_MODE, key);
```

因为 socket 是双向的通信，所以我们采用两个密码。加密输出的数据并解密输入的数据。我们使用 getInputStream()和 getOutputStream(),这两种方法来加密合解密通用的输入和输出的经过包装的数据流。

在 JCE 的 javax.crypto 包中包含 CipherInputStream 和 CipherOutputStream 这两种流类型。他们接收输入输出的流对象和密码对象。

Socket 服务器

开始写我们的 socket 服务器类吧。SecretSocketServer 在一个端口打开 ServerSocket，当接收到连接时，使用 SocketHandler 产生一个线程来操作连接。

Socket 句柄

通过 KeyGen 来定位关键字，并建立一个 SecretSocket 对象。

Key key = KeyGen.getSecretKey();

this.ss = new SecretSocket(s, key);

所有的 socket 处理都是通过 SecretSocket 而不是 Socket 对象。然后我们使用下面的代码：

in = ss.getInputStream();

记住，在 SecretSocket 中，getInputStream 是和 CipherInputStream 以及 InputStream 相结合的。因为 SocketHandler 是一个可执行的界面，我们为它生成一个 run()方法。这个方法只是在等待 socket 的数据。

Java 程序的加密和反加密

首先我们来看看Java程序的反加密，也就是通常所说的Crack过程，只有明白了这个过程，我们才能有效的对我们的程序进行加密。

通常我们得到的Java程序的Crack包有两种，一种属于KeyGen（注册码生成器）、一种属于替换修改；

我们先看第一种，当我们找到一个应用程序的 KeyGen 的时候我们总是很佩服那个做出 KeyGen 的人，觉得他很厉害，但是你仔细分析一下，为什么他能做出 KeyGen 呢？只有他对这个 Java 程序的加密算法了解的非常清楚；这种人有哪些呢？一个是那个公司里面的人，那不可能，除非内讧，还又呢，就是猜想，反推，这个可能吗？呵呵，更不可能，那这个算法从哪里来呢？呵呵，往往泄漏秘密的就是秘密本身……回过头来想想，Java 应用程序怎么知道你输入的注册码是否正确呢？呵呵，那你就该从应用程序入手……

得到的它的加密算法，自然 KeyGen 就不在话下了……（但是这也有列外，如果它是用的公钥秘钥对加密的，就没有办法喽，只能用第二种方法。）

这种办法只适合对付只要一个注册号，别的什么都不要的情况，经典代表 Borland JBuilder & Optimizeit Suite

再看第二种，为什么要用替换修改？我们是修改了那部分呢？不用想，肯定是 License 验证的部分，为什么我们不像上面的方法那样找加密算法呢？原因有两种：（1）使用上面的办法搞不定；（2）Java 程序不仅要 Key，还有其他的 License 配置；遇到这种情况，我们只要找到用于 License 验证的类，进行修改替换就行了。

这种办法使用于任何情况，经典代表 BEA WebLogic

经过上面的分析，我们的问题就集中了，关键就是怎么找到用于 License 验证的部分或加密算法的部分，我们需要 3 个工具：一个是 Sun 公司提供的标准 JVM：），一个是你的耐心和细心：），一个是 Jad（经典 Java 反编译工具）。

第一步是定位，这也是最关键的一步，我们这里以 Together For JBuilder Edition 为例，启动 Together，先看看长什么样子？喔，上来就问我要 License；Ok，没关系，退出；找到 Together 的启动 Bat 文件，找到它的启动命令：java ……，OK，在 Java 启动的时候给一个参数：“ -Xrunhprof：cpu=times”，保存，在启动，还是要 License，退出，这个时候，我们可以发现，在这个目录下多了一个“java.hprof.txt”文件，打开一看，就是我要的 JVM 的 Dump 文件，好多内容啊，没关系，慢慢看来。

我们可以看见这个文件里面有好多熟悉的东西啊：java.*/com.sun.*/javax.*等等，但这个

不是我们关心的，我们要的是 com.togethersoft.*或者是一些没有包名的 zd.d 等等。（这里插一句，几乎所有的 Java 应用程序都会混淆的，其实混淆的原理也很简单，我们后面再说。）先找找有没有 License 有关的，Serach 一下，嘿嘿，果然，474 行：com.togethersoft.together.impl.ide.license.LicenseSetup.execute （ [DashoPro-V2-050200] ： Unknown line），Ok 上那堆 classpath 中的 Jar 包里面找一下吧（推荐用 WinRAR），找到了之后用 Jad 反编译，一看，这个没有混淆，但是用了一个 zae 的类，这个看名字就知道混淆过了，先不理它，再看看下面一句 IdeLicenseAccess.setLicense（zae1），Ok 接着找到 IdeLicenseAccess，哈哈，就这点名堂，所有的 License 验证都是走的这个类，面向对象的思想不错，呵呵：）

定位定完了，接下来的事情就是按猜想的方法修改这两个类，屏蔽掉LicenseSetup里面execute方法的实际内容，修改IdeLicenseAccess，让多有的验证都返回true，然后编译，替换；不要高兴太早，这还没有完呢，要有责任心！！启动Together，果然，这下不要License了，有启动画面，进去了，但是一片灰色，怎么回事，一看控制台，一堆错，没关系，就怕不出错，查找根源，还有一个IdeLicenseUtil类出了问题，再反编译，修改，替换；这下搞定了。再启动，测试一下，OK MB7-222 70-210 1Y0-327 .

就这样，一个 Java 应用程序搞定了。看看其实也很简单。

再来说说混淆，大家可能都知道没有经过混淆的 Java 的 Class 反编译回来连方法和变量的名字都不会变，这是什么原因呢？这就要追述到 Class 文件的结构了，简单来说，Class 文件种包含又一个常数池（constant pool）这个里面就存放了变量和方法的名称等一下和 Class 相关的东西，我们通常所说的混淆就是用一种工具把这个常数池里面的东东弄的胡涂一点，这样就能骗过反编译器和你，呵呵：）这就是为什么有时候反编译回来的东西编译不过去的原因。

再回过头来说说 Java 程序的加密；从上面的两种方法来看，Java 程序似乎是没有什么完美的办法进行加密的，其实不然，我们必须遵循一些原则，才能有效的保护你的产品。

原则一，尽量使用公钥和秘钥对进行加密；

原则二，不要在加密验证的部分使用面向对象思想：）把验证的方法写在程序的各个角落，并标注为 private final void，让编译器替你处理成内联方法；

原则三，尽可能的大幅度混淆：）找个好点的混淆器

JAVA 加密解密---自定义类加载器应用

最近在研究JAVA CLASS LOADING技术，已实现了一个自定义的加载器。对目前自定义加载器的应用，还在探讨中。下面是自定义的CLASSLOADER在JAVA加密解密方面的一些研究。

JAVA安全

JAVA是解释执行的语言，对于不同的操作平台都有相应的JVM对字节码文件进行解释执行。而这个字节码文件，也就是我们平时所看到的每一个.class文件。

这是我们大家都知道的常识，也就是由.java文件，经过编译器编译，变成JVM所能解释的.class文件。

而这个过程，在现在公开的网络技术中，利用一个反编译器，任何人都可以很容易的获取它的源文件。这对于很多人来说是不希望看到的。

对于加密解密技术，我懂的不多，有些可以利用某种技术“模糊”JAVA类文件。这样能够使反编译的难度增加。但估计反编译器的技术水平也在不断提升，导致这种方法层层受阻。另外还有很多其他的技术也可以实现对JAVA文件的加密解密。我现在所想要研究的，就是其中的一种。

JAVA的灵活性使反编译变得容易，同时，也让我们的加密解密的方法变得灵活。

利用自定义的CLASSLOADER

参照：http://www.blogjava.net/realsmy/archive/2007/04/18/111582.html

JAVA中的每一个类都是通过类加载器加载到内存中的。对于类加载器的工作流程如下表示：
1.searchfile()
找到我所要加载的类文件。（抛除JAR包的概念，现在只是要加载一个.class文件）
2.loadDataClass()
读取这个类文件的字节码。
3.difineClass()
加载类文件。（加载的过程其实很复杂，我们现在先不研究它。）

从这个过程中我们能很清楚的发现，自定义的类加载能够很轻松的控制每个类文件的加载过程。这样在第二步（loadDataClass）和第三步（difineClass）之间，我们将会有自己的空间灵活的控制这个过程。

我们加密解密的技术就应用到这里。

加密解密

JAVA加密解密的技术有很多。JAVA自己提供了良好的类库对各种算法进行支持。对于采用哪种算法，网络上说法不一，自己去GOOGLE一下吧。

下面用DES对称加密算法（设定一个密钥，然后对所有的数据进行加密）来简单举个例子。

首先，生成一个密钥KEY。
我把它保存到key.txt中。这个文件就象是一把钥匙。谁拥有它，谁就能解开我们的类文件。
代码参考如下：

```
package com.neusoft.jiami;

import java.io.File;
import java.io.FileOutputStream;
import java.security.SecureRandom;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
```

```
class Key {

  private String keyName;

   public Key(String keyName) {
    this.keyName = keyName;
  }

   public void createKey(String keyName) throws Exception {

    // 创建一个可信任的随机数源，DES 算法需要
    SecureRandom sr = new SecureRandom();
    // 用 DES 算法创建一个 KeyGenerator 对象
    KeyGenerator kg = KeyGenerator.getInstance("DES");
    // 初始化此密钥生成器,使其具有确定的密钥长度
    kg.init(sr);
    // 生成密匙
    SecretKey key = kg.generateKey();
    // 获取密钥数据
    byte rawKeyData[] = key.getEncoded();
    // 将获取到密钥数据保存到文件中，待解密时使用
    FileOutputStream fo = new FileOutputStream(new File(keyName));
    fo.write(rawKeyData);
  }

   public static void main(String args[]) {
     try {
      new Key("key.txt");
     } catch (Exception e) {
      e.printStackTrace();
    }

  }
}
```

第二步，对我们所要进行加密的类文件进行加密。

比如我有一个 DigestPass 类，已经被正常编译好生成 DigestPass.class 文件。此时，这个类文件是任何人都可以用的。因为系统的类加载器可以自动的加载它。那么下一步，我们要做的就是把这个类文件加密。使系统的类加载器无法读取到正确的字节码文件。参考代码如下：

```
package com.neusoft.jiami;

import java.io.File;
import java.io.FileInputStream;
```

```
import java.io.FileOutputStream;
import java.security.SecureRandom;

import javax.crypto.Cipher;
import javax.crypto.SecretKey;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.DESKeySpec;

public class JiaMi {

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

        // DES 算法要求有一个可信任的随机数源
        SecureRandom sr = new SecureRandom();
        // 获得密匙数据
        FileInputStream fi = new FileInputStream(new File("key.txt"));
        byte rawKeyData[] = new byte[fi.available()];
        fi.read(rawKeyData);
        fi.close();
        // 从原始密匙数据创建 DESKeySpec 对象
        DESKeySpec dks = new DESKeySpec(rawKeyData);
        // 创建一个密匙工厂，然后用它把 DESKeySpec 转换成一个 SecretKey 对象
        SecretKey key = SecretKeyFactory.getInstance("DES").generateSecret(dks);
        // Cipher 对象实际完成加密操作
        Cipher cipher = Cipher.getInstance("DES");
        // 用密匙初始化 Cipher 对象
        cipher.init(Cipher.ENCRYPT_MODE, key, sr);
        // 现在，获取要加密的文件数据
        FileInputStream fi2 = new FileInputStream(new File("DigestPass.class"));
        byte data[] = new byte[fi2.available()];
        fi2.read(data);
        fi2.close();
        // 正式执行加密操作
        byte encryptedData[] = cipher.doFinal(data);
        // 用加密后的数据覆盖原文件
        FileOutputStream fo = new FileOutputStream(new File("DigestPass.class"));
        fo.write(encryptedData);
        fo.close();
    }
}
```

第三步，用自定义的 CLASSLOADER 进行加载。参考代码如下：

```
package com.neusoft.jiami;
```

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.security.SecureRandom;

import javax.crypto.Cipher;
import javax.crypto.SecretKey;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.DESKeySpec;
import com.neusoft.classloader.MyClassLoader;

public class JieMi {

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

        // DES 算法要求有一个可信任的随机数源
        SecureRandom sr = new SecureRandom();
        // 获得密匙数据
        FileInputStream fi = new FileInputStream(new File("key.txt"));
        byte rawKeyData[] = new byte[fi.available()];// = new byte[5];
        fi.read(rawKeyData);
        fi.close();
        // 从原始密匙数据创建一个 DESKeySpec 对象
        DESKeySpec dks = new DESKeySpec(rawKeyData);
        // 创建一个密匙工厂，然后用它把 DESKeySpec 对象转换成一个 SecretKey 对象
        SecretKey key = SecretKeyFactory.getInstance("DES").generateSecret(dks);
        // Cipher 对象实际完成解密操作
        Cipher cipher = Cipher.getInstance("DES");
        // 用密匙初始化 Cipher 对象
        cipher.init(Cipher.DECRYPT_MODE, key, sr);
        // 现在，获取数据并解密
        FileInputStream fi2 = new FileInputStream(new File("DigestPass.class"));
        byte encryptedData[] = new byte[fi2.available()];
        fi2.read(encryptedData);
        fi2.close();
        // 正式执行解密操作
        byte decryptedData[] = cipher.doFinal(encryptedData);
        // 这时把数据还原成原有的类文件
        // FileOutputStream fo = new FileOutputStream(new
        // File("DigestPass.class"));
        // fo.write(decryptedData);
        // 用解密后的数据加载类并应用
        MyClassloader mcl = new MyClassloader("E:/");
```

```
        Class cl = mcl.loadClass(decryptedData, "com.neusoft.jiami.DigestPass");
        DigestPass dp = cl.newInstance();
    }
}
```

这样，我们就完成了类的加密解密。这个过程留给我们修改的空间还很多。我也在理解过程中。

总结

自定义的类加载器能够灵活的控制类的加载过程。从而可以实现一些我们所要的功能。

但是，即使是这样的加密技术，对于某些高手来说，依然是脆弱的。我们所需要做的就是，理解这其中的过程，掌握这样的技术，最终能够应用到我们自己的实际项目中来。

利用 DES 加密算法保护 Java 源代码

Java语言是一种非常适用于网络编程的语言，它的基本结构与C++极为相似，但抛弃了C/C++中指针等内容，同时它吸收了Smalltalk、C++面向对象的编程思想。它具有简单性、鲁棒性、可移植性、动态性等特点。这些特点使得Java成为跨平台应用开发的一种规范，在世界范围内广泛流传。

加密Java源码的原因

Java源代码经过编译以后在JVM中执行。由于JVM界面是完全透明的，Java类文件能够很容易通过反编译器重新转换成源代码。因此，所有的算法、类文件等都可以以源代码的形式被公开，使得软件不能受到保护，为了保护产权，一般可以有以下几种方法：

（1）"模糊"类文件，加大反编译器反编译源代码文件的难度。然而，可以修改反编译器，使之能够处理这些模糊类文件。所以仅仅依赖"模糊类文件"来保证代码的安全是不够的。

（2）流行的加密工具对源文件进行加密，比如PGP（Pretty Good Privacy）或GPG（GNU Privacy Guard）。这时，最终用户在运行应用之前必须先进行解密。但解密之后，最终用户就有了一份不加密的类文件，这和事先不进行加密没有什么差别。

（3）加密类文件，在运行中JVM用定制的类装载器（Class Loader）解密类文件。Java运行时装入字节码的机制隐含地意味着可以对字节码进行修改。JVM每次装入类文件时都需要一个称为ClassLoader的对象，这个对象负责把新的类装入正在运行的JVM。JVM给ClassLoader一个包含了待装入类（例如java.lang.Object）名字的字符串，然后由ClassLoader负责找到类文件，装入原始数据，并把它转换成一个Class对象。

用户下载的是加密过的类文件，在加密类文件装入之时进行解密，因此可以看成是一种即时解密器。由于解密后的字节码文件永远不会保存到文件系统，所以窃密者很难得到解密后的代码。

由于把原始字节码转换成Class对象的过程完全由系统负责，所以创建定制ClassLoader对象其实并不困难，只需先获得原始数据，接着就可以进行包含解密在内的任何转换。

Java密码体系和Java密码扩展

Java密码体系(JCA)和Java密码扩展(JCE)的设计目的是为Java提供与实现无关的加密函数API。它们都用factory方法来创建类的例程，然后把实际的加密函数委托给提供者指定的底层引擎,引擎中为类提供了服务提供者接口在Java中实现数据的加密/解密，是使用其内置的JCE(Java加密扩展)来实现的。Java开发工具集 1.1 为实现包括数字签名和信息摘要在内的加密功能，推出了一种基于供应商的新型灵活应用编程接口。Java密码体系结构支持供应商的互操作,同时支持硬件和软件实现。

Java密码学结构设计遵循两个原则:

(1)算法的独立性和可靠性。

(2)实现的独立性和相互作用性。

算法的独立性是通过定义密码服务类来获得。用户只需了解密码算法的概念,而不用去关心如何实现这些概念。实现的独立性和相互作用性通过密码服务提供器来实现。密码服务提供器是实现一个或多个密码服务的一个或多个程序包。软件开发商根据一定接口,将各种算法实现后,打包成一个提供器,用户可以安装不同的提供器。安装和配置提供器,可将包含提供器的ZIP和JAR文件放在CLASSPATH下,再编辑Java安全属性文件来设置定义一个提供器。Java运行环境Sun版本时，提供一个缺省的提供器Sun。

下面介绍DES算法及如何利用DES算法加密和解密类文件的步骤。

DES算法简介

DES（Data Encryption Standard）是发明最早的最广泛使用的分组对称加密算法。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算法工作流程如下：若Mode为加密模式，则利用Key 对数据Data进行加密， 生成Data的密码形式（64 位）作为DES的输出结果；如Mode为解密模式，则利用Key对密码形式的数据Data进行解密，还原为Data的明码形式（64 位）作为DES的输出结果。在通信网络的两端，双方约定一致的Key，在通信的源点用Key对核心数据进行DES加密，然后以密码形式在公共通信网（如电话网）中传输到通信网络的终点，数据到达目的地后，用同样的Key对密码数据进行解密，便再现了明码形式的核心数据。这样，便保证了核心数据在公共通信网中传输的安全性和可靠性。

也可以通过定期在通信网络的源端和目的端同时改用新的Key，便能更进一步提高数据的保密性。

利用DES算法加密的步骤

（1）生成一个安全密钥。在加密或解密任何数据之前需要有一个密钥。密钥是随同被加密的应用程序一起发布的一段数据，密钥代码如下所示。

【生成一个密钥代码】

```
// 生成一个可信任的随机数源
Secure Random sr = new SecureRandom();
// 为我们选择的DES算法生成一个KeyGenerator对象
KeyGenerator kg = KeyGenerator.getInstance ("DES" );
Kg.init (sr);
// 生成密钥
Secret Key key = kg.generateKey();
// 将密钥数据保存为文件供以后使用，其中key Filename为保存的文件名
Util.writeFile (key Filename, key.getEncoded () );
```

（2）加密数据。得到密钥之后，接下来就可以用它加密数据。如下所示。

【用密钥加密原始数据】

```
// 产生一个可信任的随机数源
SecureRandom sr = new SecureRandom();
//从密钥文件key Filename中得到密钥数据
Byte rawKeyData [] = Util.readFile (key Filename);
// 从原始密钥数据创建DESKeySpec对象
DESKeySpec dks = new DESKeySpec (rawKeyData);
// 创建一个密钥工厂，然后用它把DESKeySpec转换成Secret Key对象
SecretKeyFactory key Factory = SecretKeyFactory.getInstance("DES" );
Secret Key key = keyFactory.generateSecret( dks );
// Cipher对象实际完成加密操作
Cipher cipher = Cipher.getInstance( "DES" );
// 用密钥初始化Cipher对象
cipher.init( Cipher.ENCRYPT_MODE, key, sr );
// 通过读类文件获取需要加密的数据
Byte data [] = Util.readFile (filename);
// 执行加密操作
Byte encryptedClassData [] = cipher.doFinal(data );
// 保存加密后的文件，覆盖原有的类文件。
Util.writeFile( filename, encryptedClassData );
```

（3）解密数据。运行经过加密的程序时，ClassLoader分析并解密类文件。操作步骤如下所示。

【用密钥解密数据】

```
// 生成一个可信任的随机数源
SecureRandom sr = new SecureRandom();
// 从密钥文件中获取原始密钥数据
Byte rawKeyData[] = Util.readFile( keyFilename );
// 创建一个DESKeySpec对象
DESKeySpec dks = new DESKeySpec (rawKeyData);
// 创建一个密钥工厂，然后用它把DESKeySpec对象转换成Secret Key对象
SecretKeyFactory key Factory = SecretKeyFactory.getInstance( "DES" );
SecretKey key = keyFactory.generateSecret( dks );
// Cipher对象实际完成解密操作
Cipher cipher = Cipher.getInstance( "DES" );
// 用密钥初始化Cipher对象
Cipher.init( Cipher.DECRYPT_MODE, key, sr );
// 获得经过加密的数据
Byte encrypted Data [] = Util.readFile (Filename);
//执行解密操作
Byte decryptedData [] = cipher.doFinal( encryptedData );
// 然后将解密后的数据转化成原来的类文件。
```

将上述代码与自定义的类装载器结合就可以做到边解密边运行，从而起到保护源代码的作用。

结束语

加密/解密是数据传输中保证数据安全性和完整性的常用方法，Java语言因其平台无关性，在Internet上的应用非常之广泛。使用DES算法加密Java源码在一定程度上能保护软件的产权。