链滴

常见算法和加密算法

作者：someone31642
原文链接：https://ld246.com/article/1541414511532
来源网站：链滴
许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

本文主要对消息摘要算法和加密算法做了整理，包括MD5、SHA、DES、AES、RSA等，并且提供了应算法的Java实现和测试。

一 消息摘要算法

1. 简介：

- **消息摘要算法的主要特征是加密过程不需要密钥，并且经过加密的数据无法被解密**
- **只有输入相同的明文数据经过相同的消息摘要算法才能得到相同的密文。**
- **消息摘要算法主要应用在“数字签名”领域，作为对明文的摘要算法。**
- **著名的摘要算法有RSA公司的MD5算法和SHA-1算法及其大量的变体。**

2. 特点：

1. **无论输入的消息有多长，计算出来的消息摘要的长度总是固定的。**
2. **消息摘要看起来是“伪随机的”。也就是说对相同的信息求摘要结果相同。**
3. **消息轻微改变生成的摘要变化会很大**
4. **只能进行正向的信息摘要，而无法从摘要中恢复出任何的消息，甚至根本就找不到任何与原信息相的信息**

3. 应用：

消息摘要算法最常用的场景就是数字签名以及数据（密码）加密了。(一般平时做项目用的比较多的就使用MD5对用户密码进行加密)

4. 何谓数字签名：

数字签名主要用到了非对称密钥加密技术与数字摘要技术。数字签名技术是将摘要信息用发送者的私加密，与原文一起传送给接收者。接收者只有用发送者的公钥才能解密被加密的摘要信息，然后用HAH函数对收到的原文产生一个摘要信息，与解密的摘要信息对比。

如果相同，则说明收到的信息是完整的，在传输过程中没有被修改，否则说明信息被修改过.

因此数字签名能够验证信息的完整性。

数字签名是个加密的过程，数字签名验证是个解密的过程。

5. 常见消息/数字摘要算法：

MD5:

简介：

MD5的作用是让大容量信息在用数字签名软件签署私人密钥前被"压缩"成一种保密的格式（也就是把一个任意长度的字节串变换成一定长的十六进制数字串）。

特点：

1. **压缩性：** 任意长度的数据，算出的MD5值长度都是固定的。
2. **容易计算：** 从原数据计算出MD5值很容易。
3. **抗修改性：** 对原数据进行任何改动，哪怕只修改1个字节，所得到的MD5值都有很大区别。
4. **强抗碰撞：** 已知原数据和其MD5值，想找到一个具有相同MD5值的数据（即伪造数据）是非常困的。

代码实现：

利用JDK提供java.security.MessageDigest类实现MD5算法：

```
package com.snailclimb.ks.securityAlgorithm;

import java.security.MessageDigest;

public class MD5Demo {

    // test
    public static void main(String[] args) {
        System.out.println(getMD5Code("你若安好，便是晴天"));
    }

    private MD5Demo() {
    }

    // md5加密
    public static String getMD5Code(String message) {
        String md5Str = "";
        try {
            //创建MD5算法消息摘要
            MessageDigest md = MessageDigest.getInstance("MD5");
            //生成的哈希值的字节数组
            byte[] md5Bytes = md.digest(message.getBytes());
            md5Str = bytes2Hex(md5Bytes);
        }catch(Exception e) {
            e.printStackTrace();
        }
        return md5Str;
    }

    // 2进制转16进制
    public static String bytes2Hex(byte[] bytes) {
        StringBuffer result = new StringBuffer();
        int temp;
        try {
            for (int i = 0; i < bytes.length; i++) {
                temp = bytes[i];
                if(temp < 0) {
                    temp += 256;
                }
                if (temp < 16) {
```

```
                result.append("0");
            }
            result.append(Integer.toHexString(temp));
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
    return result.toString();
  }
}
```

结果：

6bab82679914f7cb480a120b532ffa80

注意MessageDigest类的几个方法：

static MessageDigest getInstance(String algorithm)//返回实现指定摘要算法的MessageDigest象

byte[] digest(byte[] input)//使用指定的字节数组对摘要执行最终更新，然后完成摘要计算。

不利用Java提供的java.security.MessageDigest类实现MD5算法：

```
package com.snailclimb.ks.securityAlgorithm;

public class MD5{
    /*
    *四个链接变量
    */
    private final int A=0x67452301;
    private final int B=0xefcdab89;
    private final int C=0x98badcfe;
    private final int D=0x10325476;
    /*
    *ABCD的临时变量
    */
    private int Atemp,Btemp,Ctemp,Dtemp;

    /*
    *常量ti
    *公式:floor(abs(sin(i+1))×(2pow32)
    */
    private final int K[]={
        0xd76aa478,0xe8c7b756,0x242070db,0xc1bdceee,
        0xf57c0faf,0x4787c62a,0xa8304613,0xfd469501,0x698098d8,
        0x8b44f7af,0xffff5bb1,0x895cd7be,0x6b901122,0xfd987193,
        0xa679438e,0x49b40821,0xf61e2562,0xc040b340,0x265e5a51,
        0xe9b6c7aa,0xd62f105d,0x02441453,0xd8a1e681,0xe7d3fbc8,
        0x21e1cde6,0xc33707d6,0xf4d50d87,0x455a14ed,0xa9e3e905,
        0xfcefa3f8,0x676f02d9,0x8d2a4c8a,0xfffa3942,0x8771f681,
        0x6d9d6122,0xfde5380c,0xa4beea44,0x4bdecfa9,0xf6bb4b60,
```

```
    0xbebfbc70,0x289b7ec6,0xeaa127fa,0xd4ef3085,0x04881d05,
    0xd9d4d039,0xe6db99e5,0x1fa27cf8,0xc4ac5665,0xf4292244,
    0x432aff97,0xab9423a7,0xfc93a039,0x655b59c3,0x8f0ccc92,
    0xffeff47d,0x85845dd1,0x6fa87e4f,0xfe2ce6e0,0xa3014314,
    0x4e0811a1,0xf7537e82,0xbd3af235,0x2ad7d2bb,0xeb86d391};
/*
*向左位移数,计算方法未知
*/
private final int s[]={7,12,17,22,7,12,17,22,7,12,17,22,7,
    12,17,22,5,9,14,20,5,9,14,20,5,9,14,20,5,9,14,20,
    4,11,16,23,4,11,16,23,4,11,16,23,4,11,16,23,6,10,
    15,21,6,10,15,21,6,10,15,21,6,10,15,21};


/*
*初始化函数
*/
private void init(){
    Atemp=A;
    Btemp=B;
    Ctemp=C;
    Dtemp=D;
}
/*
*移动一定位数
*/
private    int    shift(int a,int s){
    return(a<<s)|(a>>>(32-s));//右移的时候，高位一定要补零，而不是补充符号位
}
/*
*主循环
*/
private void MainLoop(int M[]){
    int F,g;
    int a=Atemp;
    int b=Btemp;
    int c=Ctemp;
    int d=Dtemp;
    for(int i = 0; i < 64; i ++){
        if(i<16){
            F=(b&c)|((~b)&d);
            g=i;
        }else if(i<32){
            F=(d&b)|((~d)&c);
            g=(5*i+1)%16;
        }else if(i<48){
            F=b^c^d;
            g=(3*i+5)%16;
        }else{
            F=c^(b|(~d));
            g=(7*i)%16;
        }
        int tmp=d;
        d=c;
```

```
            c=b;
            b=b+shift(a+F+K[i]+M[g],s[i]);
            a=tmp;
        }
        Atemp=a+Atemp;
        Btemp=b+Btemp;
        Ctemp=c+Ctemp;
        Dtemp=d+Dtemp;

    }
    /*
    *填充函数
    *处理后应满足bits≡448(mod512),字节就是bytes≡56（mode64)
    *填充方式为先加一个0,其它位补零
    *最后加上64位的原来长度
    */
    private int[] add(String str){
        int num=((str.length()+8)/64)+1;//以512位，64个字节为一组
        int strByte[]=new int[num*16];//64/4=16，所以有16个整数
        for(int i=0;i<num*16;i++){//全部初始化0
            strByte[i]=0;
        }
        int    i;
        for(i=0;i<str.length();i++){
            strByte[i>>2]|=str.charAt(i)<<((i%4)*8);//一个整数存储四个字节，小端序
        }
        strByte[i>>2]|=0x80<<((i%4)*8);//尾部添加1
        /*
        *添加原长度，长度指位的长度，所以要乘8，然后是小端序，所以放在倒数第二个,这里长度只
了32位
        */
        strByte[num*16-2]=str.length()*8;
            return strByte;
    }
    /*
    *调用函数
    */
    public String getMD5(String source){
        init();
        int strByte[]=add(source);
        for(int i=0;i<strByte.length/16;i++){
        int num[]=new int[16];
        for(int j=0;j<16;j++){
            num[j]=strByte[i*16+j];
        }
        MainLoop(num);
        }
        return changeHex(Atemp)+changeHex(Btemp)+changeHex(Ctemp)+changeHex(Dtemp);

    }
    /*
    *整数变成16进制字符串
    */
    private String changeHex(int a){
```

```
        String str="";
        for(int i=0;i<4;i++){
            str+=String.format("%2s", Integer.toHexString(((a>>i*8)%(1<<8))&0xff)).replace(' ', '0')


        }
        return str;
    }
    /*
    *单例
    */
    private static MD5 instance;
    public static MD5 getInstance(){
        if(instance==null){
            instance=new MD5();
        }
        return instance;
    }

    private MD5(){};

    public static void main(String[] args){
        String str=MD5.getInstance().getMD5("你若安好，便是晴天");
        System.out.println(str);
    }
}
```

SHA1:

对于长度小于2^64位的消息，SHA1会产生一个160位(40个字符)的消息摘要。当接收到消息的时候这个消息摘要可以用来验证数据的完整性。在传输的过程中，数据很可能会发生变化，那么这时候就产生不同的消息摘要。

SHA1有如下特性：

- 不可以从消息摘要中复原信息；
- 两个不同的消息不会产生同样的消息摘要,(但会有1x10 ^ 48分之一的机率出现相同的消息摘要,一使用时忽略)。

代码实现：

*利用JDK提供java.security.MessageDigest类实现SHA1算法：

```
package com.snailclimb.ks.securityAlgorithm;

import java.io.UnsupportedEncodingException;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

public class SHA1Demo {

    public static void main(String[] args) {
```

```
        // TODO Auto-generated method stub
        System.out.println(getSha1("你若安好，便是晴天"));

    }

    public static String getSha1(String str) {
        if (null == str || 0 == str.length()) {
            return null;
        }
        char[] hexDigits = { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e', 'f' };
        try {
            //创建SHA1算法消息摘要对象
            MessageDigest mdTemp = MessageDigest.getInstance("SHA1");
            //使用指定的字节数组更新摘要。
            mdTemp.update(str.getBytes("UTF-8"));
            //生成的哈希值的字节数组
            byte[] md = mdTemp.digest();
            //SHA1算法生成信息摘要关键过程
            int j = md.length;
            char[] buf = new char[j * 2];
            int k = 0;
            for (int i = 0; i < j; i++) {
                byte byte0 = md[i];
                buf[k++] = hexDigits[byte0 >>> 4 & 0xf];
                buf[k++] = hexDigits[byte0 & 0xf];
            }
            return new String(buf);
        } catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
        } catch (UnsupportedEncodingException e) {
            e.printStackTrace();
        }
        return "0";

    }
}
```

结果：

8ce764110a42da9b08504b20e26b19c9e3382414

二 加密算法

1. 简介：

- **加密技术包括两个元素：加密算法和密钥。**
- **加密算法是将普通的文本（或者可以理解的信息）与一串数字（密钥）的结合，产生不可理解的密的步骤。**
- **密钥是用来对数据进行编码和解码的一种算法。**
- **在安全保密中，可通过适当的密钥加密技术和管理机制来保证网络的信息通讯安全。**

2. 分类：

密钥加密技术的密码体制分为对称密钥体制和非对称密钥体制两种。相应地，对数据加密的技术分为类，即对称加密（私人密钥加密）和非对称加密（公开密钥加密）。

对称加密以数据加密标准（DES，Data Encryption Standard）算法为典型代表，非对称加密通常RSA（Rivest Shamir Adleman）算法为代表。

对称加密的加密密钥和解密密钥相同。非对称加密的加密密钥和解密密钥不同，加密密钥可以公开而密密钥需要保密

3. 应用：

常被用在电子商务或者其他需要保证网络传输安全的范围。

4. 对称加密：

加密密钥和解密密钥相同的加密算法。

对称加密算法使用起来简单快捷，密钥较短，且破译困难，除了数据加密标准（DES），

另一个对称密钥加密系统是国际数据加密算法（IDEA），它比DES的加密性好，而且对计算机功能要也没有那么高。IDEA加密标准由PGP（Pretty Good Privacy）系统使用。

DES:

DES全称为Data Encryption Standard，即数据加密标准，是一种使用密钥加密的块算法，现在已经时。

代码实现：

DES算法实现：

```
package com.snailclimb.ks.securityAlgorithm;

import java.io.UnsupportedEncodingException;
import java.security.SecureRandom;
import javax.crypto.spec.DESKeySpec;
import javax.crypto.SecretKeyFactory;
import javax.crypto.SecretKey;
import javax.crypto.Cipher;

/**
 * DES加密介绍 DES是一种对称加密算法，所谓对称加密算法即：加密和解密使用相同密钥的算法。
ES加密算法出自IBM的研究，
 * 后来被美国政府正式采用，之后开始广泛流传，但是近些年使用越来越少，因为DES使用56位密钥
以现代计算能力，
 * 24小时内即可被破解。虽然如此，在某些简单应用中，我们还是可以使用DES加密算法，本文简单
解DES的JAVA实现 。
 * 注意：DES加密和解密过程中，密钥长度都必须是8的倍数
 */
public class DesDemo {
```

```
    public DesDemo() {
    }

    // 测试
    public static void main(String args[]) {
        // 待加密内容
        String str = "cryptology";
        // 密码，长度要是8的倍数
        String password = "95880288";

        byte[] result;
        try {
            result = DesDemo.encrypt(str.getBytes(), password);
            System.out.println("加密后：" + result);
            byte[] decryResult = DesDemo.decrypt(result, password);
            System.out.println("解密后：" + new String(decryResult));
        } catch (UnsupportedEncodingException e2) {
            // TODO Auto-generated catch block
            e2.printStackTrace();
        } catch (Exception e1) {
            e1.printStackTrace();
        }
    }

    // 直接将如上内容解密

    /**
     * 加密
     *
     * @param datasource
     *          byte[]
     * @param password
     *          String
     * @return byte[]
     */
    public static byte[] encrypt(byte[] datasource, String password) {
        try {
            SecureRandom random = new SecureRandom();
            DESKeySpec desKey = new DESKeySpec(password.getBytes());
            // 创建一个密匙工厂，然后用它把DESKeySpec转换成
            SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("DES");
            SecretKey securekey = keyFactory.generateSecret(desKey);
            // 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,ENCRYPT_MODE用于将 Cipher 初始化为加密模式的常量
            cipher.init(Cipher.ENCRYPT_MODE, securekey, random);
            // 现在，获取数据并加密
            // 正式执行加密操作
            return cipher.doFinal(datasource); // 按单部分操作加密或解密数据，或者结束一个多部分
作
        } catch (Throwable e) {
            e.printStackTrace();
        }
        return null;
```

```
    }

    /**
     * 解密
     *
     * @param src
     *          byte[]
     * @param password
     *          String
     * @return byte[]
     * @throws Exception
     */
    public static byte[] decrypt(byte[] src, String password) throws Exception {
        // DES算法要求有一个可信任的随机数源
        SecureRandom random = new SecureRandom();
        // 创建一个DESKeySpec对象
        DESKeySpec desKey = new DESKeySpec(password.getBytes());
        // 创建一个密匙工厂
        SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("DES");// 返回实现指定转换

                                                                    // Cipher
                                                                    // 对象
        // 将DESKeySpec对象转换成SecretKey对象
        SecretKey securekey = keyFactory.generateSecret(desKey);
        // 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, securekey, random);
        // 真正开始解密操作
        return cipher.doFinal(src);
    }
}
```

结果：

```
加密后：[B@50cbc42f
解密后：cryptology
```

IDEA:

- **这种算法是在DES算法的基础上发展出来的，类似于三重DES。**
- **发展IDEA也是因为感到DES具有密钥太短等缺点。**
- **DEA的密钥为128位，这么长的密钥在今后若干年内应该是安全的。**
- **在实际项目中用到的很少了解即可。**

代码实现：

IDEA算法实现

```
package com.snailclimb.ks.securityAlgorithm;

import java.security.Key;
```

```java
import java.security.Security;

import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;

import org.apache.commons.codec.binary.Base64;
import org.bouncycastle.jce.provider.BouncyCastleProvider;

public class IDEADemo {
    public static void main(String args[]) {
        bcIDEA();
    }
    public static void bcIDEA() {
        String src = "www.xttblog.com security idea";
        try {
            Security.addProvider(new BouncyCastleProvider());

            //生成key
            KeyGenerator keyGenerator = KeyGenerator.getInstance("IDEA");
            keyGenerator.init(128);
            SecretKey secretKey = keyGenerator.generateKey();
            byte[] keyBytes = secretKey.getEncoded();

            //转换密钥
            Key key = new SecretKeySpec(keyBytes, "IDEA");

            //加密
            Cipher cipher = Cipher.getInstance("IDEA/ECB/ISO10126Padding");
            cipher.init(Cipher.ENCRYPT_MODE, key);
            byte[] result = cipher.doFinal(src.getBytes());
            System.out.println("bc idea encrypt : " + Base64.encodeBase64String(result));

            //解密
            cipher.init(Cipher.DECRYPT_MODE, key);
            result = cipher.doFinal(result);
            System.out.println("bc idea decrypt : " + new String(result));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

5. 非对称加密：

- 与对称加密算法不同，非对称加密算法需要两个密钥：公开密钥（publickey）和私有密钥 （privat key）。
- 公开密钥与私有密钥是一对，如果用公开密钥对数据进行加密，只有用对应的私有密钥才能解密；
- 如果用私有密钥对数据进行加密，那么只有用对应的公开密钥才能解密。
- 因为加密和解密使用的是两个不同的密钥，所以这种算法叫作非对称加密算法。

RAS:

RSA是目前最有影响力和最常用的公钥加密算法。它能够抵抗到目前为止已知的绝大多数密码攻击，被ISO推荐为公钥数据加密标准。

代码实现：

RAS算法实现：

```
package com.snailclimb.ks.securityAlgorithm;

import org.apache.commons.codec.binary.Base64;

import java.security.*;
import java.security.spec.PKCS8EncodedKeySpec;
import java.security.spec.X509EncodedKeySpec;
import java.util.HashMap;
import java.util.Map;

import javax.crypto.Cipher;

/**
 * Created by humf.需要依赖 commons-codec 包
 */
public class RSADemo {

    public static void main(String[] args) throws Exception {
        Map<String, Key> keyMap = initKey();
        String publicKey = getPublicKey(keyMap);
        String privateKey = getPrivateKey(keyMap);

        System.out.println(keyMap);
        System.out.println("------------------------------------");
        System.out.println(publicKey);
        System.out.println("------------------------------------");
        System.out.println(privateKey);
        System.out.println("------------------------------------");
        byte[] encryptByPrivateKey = encryptByPrivateKey("123456".getBytes(), privateKey);
        byte[] encryptByPublicKey = encryptByPublicKey("123456", publicKey);
        System.out.println(encryptByPrivateKey);
        System.out.println("------------------------------------");
        System.out.println(encryptByPublicKey);
        System.out.println("------------------------------------");
        String sign = sign(encryptByPrivateKey, privateKey);
        System.out.println(sign);
        System.out.println("------------------------------------");
        boolean verify = verify(encryptByPrivateKey, publicKey, sign);
        System.out.println(verify);
        System.out.println("------------------------------------");
        byte[] decryptByPublicKey = decryptByPublicKey(encryptByPrivateKey, publicKey);
        byte[] decryptByPrivateKey = decryptByPrivateKey(encryptByPublicKey, privateKey);
        System.out.println(decryptByPublicKey);
        System.out.println("------------------------------------");
```

```
    System.out.println(decryptByPrivateKey);

}

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";

public static byte[] decryptBASE64(String key) {
    return Base64.decodeBase64(key);
}

public static String encryptBASE64(byte[] bytes) {
    return Base64.encodeBase64String(bytes);
}

/**
 * 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));
}

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[] decryptByPrivateKey(String data, String key) throws Exception {
    return decryptByPrivateKey(decryptBASE64(data), key);
}

/**
 * 解密<br>
 * 用公钥解密
 *
 * @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(String 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.getBytes());
}

/**
 * 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, Key> 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, Key> keyMap) throws Exception {
        Key key = keyMap.get(PUBLIC_KEY);
        return encryptBASE64(key.getEncoded());
    }

    /**
     * 初始化密钥
     *
     * @return
     * @throws Exception
     */
    public static Map<String, Key> initKey() throws Exception {
        KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance(KEY_ALGORITHM);
        keyPairGen.initialize(1024);
        KeyPair keyPair = keyPairGen.generateKeyPair();
        Map<String, Key> keyMap = new HashMap(2);
        keyMap.put(PUBLIC_KEY, keyPair.getPublic());// 公钥
        keyMap.put(PRIVATE_KEY, keyPair.getPrivate());// 私钥
        return keyMap;
    }

}
```

结果：

```
{RSAPublicKey=Sun RSA public key, 1024 bits
  modulus: 115328826086047873902606456571034976538836553998745367981848911677980625718316266744996508543182072804199607670206012530717395551613881355894878484384543940361488396771374960526883133641800172270192453762457318027635661050309809260289965219855862692230362893996010057188170525719351126759886050814842261 69
  public exponent: 65537, RSAPrivateKey=sun.security.rsa.RSAPrivateCrtKeyImpl@93479}
-----------------------------------
```

MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCkO9PBTOFJQTkzznALN62PU7ixd9YFjXr
2dPOGj3wwhymbOU8HLoCztjwpLXHgbpBUJlGmbURV955M1BkZ1kr5dkZYR5x1gO4xOnu8rEi
y4AAMcpFttfiarIZrtzL9pKEvEOxABltVN4yzFDr3IjBqY46aHna7YjwhXI0xHieQIDAQAB

MIICdQIBADANBgkqhkiG9w0BAQEFAASCAI8wggJbAgEAAoGBAKQ708FM4UlBOTPOcAs3rY9
uLF31gWNeu3Z084aPfDCHKZs5TwcugLO2PCktceBukFQmUaZtRFX3nkzUGRnWSvl2RlhHnHW
7jE6e7ysSKnLgAAxykW21+Jqshmu3Mv2koS8Q7EAGW1U3jLMUOvciMGpjjpoedrtiPCFcjTEeJ5
gMBAAECgYAK4sxOa8IjEOexv2U92Rrv/SSo3sCY7Z/QVDft2V9xrewoO9+V9HF/7iYDDWffKYInA
imvVl7JM/iSLxza0ZFv29VMpyDcr4TigYmWwBlk7ZbxSTkqLdNwxxldMmEoTn1py53MUm+1V1
3rzNvJjuZaZFAevU7vUnwQwD+JGQYQJBAM9HBaC+dF3PJ2mkXekHpDS1ZPaSFdrdzd/GvHFi/
JAMM+Uz6PmpkosNXRtOpSYWwlOMRamLZtrHhfQoqSk3S8CQQDK1qL1jGvVdqw5OjqxktR
MmOsWUVZdWiBN+6ojxBgA0yVn0n7vkdAAgEZBj89WG0VHPEu3hd4AgXFZHDfXeDXAkBvSn
nE9t/Et7ihfI2UHgGJO8UxNMfNMB5Skebyb7eMYEDs67ZHdpjMOFypcMyTatzj5wjwQ3zyMvbl
X+ONbZAkAX4ysRy9WvL+icXLUo0Gfhkk+WrnSyUldaUGH0y9Rb2kecn0OxN/lgGlxSvB+ac910
RHCOTl+Uo6nbmq0g3PFAkAyqA4eT7G9GXfncakgW1Kdkn72w/ODpozgfhTLNX0SGw1ITML3
4THTtH5h3zLi3AF9zJO2O+K6ajRbV0szHHI

[B@387c703b

[B@224aed64
---
la4Hc4n/UbeBu0z9iLRuwKVv014SiOJMXkO5qdJvKBsw0MlnsrM+89a3p73yMrb1dAnCU/2kgO
PtFpvmG8pzxTe1u/5nX/25iIyUXALlwVRptJyjzFE83g2IX0XEv/Dxqr1RCRcrMHOLQM0oBoxZCa
hmyw1Ub4wsSs6Ndxb9M=
---
true

[B@c39f790

[B@71e7a66b