RSA加密算法研究-伟鹏

**package** com.sunshine.utils;

**import** java.io.ByteArrayOutputStream;

**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.RSAKey;

**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;

**import** sun.misc.BASE64Decoder;

**import** sun.misc.BASE64Encoder;

**public** **class** RSAEncryption {

**public** **static** **final** String ***KEY\_ALGORITHM*** = "RSA";

**private** **static** **final** String ***PUBLIC\_KEY*** = "RSAPublicKey";

**private** **static** **final** String ***PRIVATE\_KEY*** = "RSAPrivateKey";

**public** **static** **final** String ***SIGNATURE\_ALGORITHM***="MD5withRSA";

/\*\*

\* RSA最大加密明文大小

\*/

**private** **static** **final** **int** ***MAX\_ENCRYPT\_BLOCK*** = 117;

/\*\*

\* RSA最大解密密文大小

\*/

**private** **static** **final** **int** ***MAX\_DECRYPT\_BLOCK*** = 128;

//初始化RSAKey

**public** **static** Map<String, RSAKey> initKey() **throws** Exception {

KeyPairGenerator keyPairGen = KeyPairGenerator

.*getInstance*(***KEY\_ALGORITHM***);

keyPairGen.initialize(1024);

KeyPair keyPair = keyPairGen.generateKeyPair();

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

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

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

keyMap.put(***PUBLIC\_KEY***, publicKey);

keyMap.put(***PRIVATE\_KEY***, privateKey);

**return** keyMap;

}

//获得公钥字符串

**public** **static** String getPublicKeyStr(Map<String, RSAKey> keyMap) **throws** Exception {

//获得map中的公钥对象 转为key对象

Key key = (Key) keyMap.get(***PUBLIC\_KEY***);

//编码返回字符串

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

}

//获得私钥字符串

**public** **static** String getPrivateKeyStr(Map<String, RSAKey> keyMap) **throws** Exception {

//获得map中的私钥对象 转为key对象

Key key = (Key) keyMap.get(***PRIVATE\_KEY***);

//编码返回字符串

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

}

//获取公钥

**public** **static** PublicKey getPublicKey(String key) **throws** Exception {

**byte**[] keyBytes;

keyBytes = (**new** BASE64Decoder()).decodeBuffer(key);

X509EncodedKeySpec keySpec = **new** X509EncodedKeySpec(keyBytes);

KeyFactory keyFactory = KeyFactory.*getInstance*(***KEY\_ALGORITHM***);

PublicKey publicKey = keyFactory.generatePublic(keySpec);

**return** publicKey;

}

//获取私钥

**public** **static** PrivateKey getPrivateKey(String key) **throws** Exception {

**byte**[] keyBytes;

keyBytes = (**new** BASE64Decoder()).decodeBuffer(key);

PKCS8EncodedKeySpec keySpec = **new** PKCS8EncodedKeySpec(keyBytes);

KeyFactory keyFactory = KeyFactory.*getInstance*(***KEY\_ALGORITHM***);

PrivateKey privateKey = keyFactory.generatePrivate(keySpec);

**return** privateKey;

}

//解码返回byte

**public** **static** **byte**[] decryptBASE64(String key) **throws** Exception {

**return** (**new** BASE64Decoder()).decodeBuffer(key);

}

//编码返回字符串

**public** **static** String encryptBASE64(**byte**[] key) **throws** Exception {

**return** (**new** BASE64Encoder()).encodeBuffer(key);

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*签名和验证\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**public** **static** **byte**[] sign(**byte**[] data,String privateKeyStr) **throws** Exception{

PrivateKey priK = *getPrivateKey*(privateKeyStr);

Signature sig = Signature.*getInstance*(***SIGNATURE\_ALGORITHM***);

sig.initSign(priK);

sig.update(data);

**return** sig.sign();

}

**public** **static** **boolean** verify(**byte**[] data,**byte**[] sign,String publicKeyStr) **throws** Exception{

PublicKey pubK = *getPublicKey*(publicKeyStr);

Signature sig = Signature.*getInstance*(***SIGNATURE\_ALGORITHM***);

sig.initVerify(pubK);

sig.update(data);

**return** sig.verify(sign);

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*加密解密\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**public** **static** **byte**[] encrypt(**byte**[] plainText,String publicKeyStr)**throws** Exception{

PublicKey publicKey = *getPublicKey*(publicKeyStr);

Cipher cipher = Cipher.*getInstance*(***KEY\_ALGORITHM***);

cipher.init(Cipher.***ENCRYPT\_MODE***, publicKey);

**int** inputLen = plainText.length;

ByteArrayOutputStream out = **new** ByteArrayOutputStream();

**int** offSet = 0;

**int** i = 0;

**byte**[] cache;

**while** (inputLen - offSet > 0) {

**if** (inputLen - offSet > ***MAX\_ENCRYPT\_BLOCK***) {

cache = cipher.doFinal(plainText, offSet, ***MAX\_ENCRYPT\_BLOCK***);

} **else** {

cache = cipher.doFinal(plainText, offSet, inputLen - offSet);

}

out.write(cache, 0, cache.length);

i++;

offSet = i \* ***MAX\_ENCRYPT\_BLOCK***;

}

**byte**[] encryptText = out.toByteArray();

out.close();

**return** encryptText;

}

**public** **static** **byte**[] decrypt(**byte**[] encryptText,String privateKeyStr)**throws** Exception{

PrivateKey privateKey = *getPrivateKey*(privateKeyStr);

Cipher cipher = Cipher.*getInstance*(***KEY\_ALGORITHM***);

cipher.init(Cipher.***DECRYPT\_MODE***, privateKey);

**int** inputLen = encryptText.length;

ByteArrayOutputStream out = **new** ByteArrayOutputStream();

**int** offSet = 0;

**byte**[] cache;

**int** i = 0;

// 对数据分段解密

**while** (inputLen - offSet > 0) {

**if** (inputLen - offSet > ***MAX\_DECRYPT\_BLOCK***) {

cache = cipher.doFinal(encryptText, offSet, ***MAX\_DECRYPT\_BLOCK***);

} **else** {

cache = cipher.doFinal(encryptText, offSet, inputLen - offSet);

}

out.write(cache, 0, cache.length);

i++;

offSet = i \* ***MAX\_DECRYPT\_BLOCK***;

}

**byte**[] plainText = out.toByteArray();

out.close();

**return** plainText;

}

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

Map<String, RSAKey> keyMap;

**byte**[] cipherText;

String input = "Hello World!";

**try** {

keyMap = *initKey*();

String publicKey = *getPublicKeyStr*(keyMap);

System.***out***.println("公钥------------------");

System.***out***.println(publicKey);

String privateKey = *getPrivateKeyStr*(keyMap);

System.***out***.println("私钥------------------");

System.***out***.println(privateKey);

System.***out***.println("测试可行性-------------------");

System.***out***.println("明文======="+input);

cipherText = *encrypt*(input.getBytes(),publicKey);

//加密后的东西

System.***out***.println("Dddddd---------------");

System.***out***.println("密文=======++"+**new** String(cipherText));

System.***out***.println("Dddddd---------------");

//开始解密

**byte**[] plainText = *decrypt*(cipherText,privateKey);

System.***out***.println("解密后明文===== " + **new** String(plainText));

System.***out***.println("验证签名-----------");

String str="被签名的内容";

System.***out***.println("\n原文:"+str);

**byte**[] signature=*sign*(str.getBytes(),privateKey);

**boolean** status=*verify*(str.getBytes(), signature,publicKey);

System.***out***.println("验证情况："+status);

} **catch** (Exception e) {

e.printStackTrace();

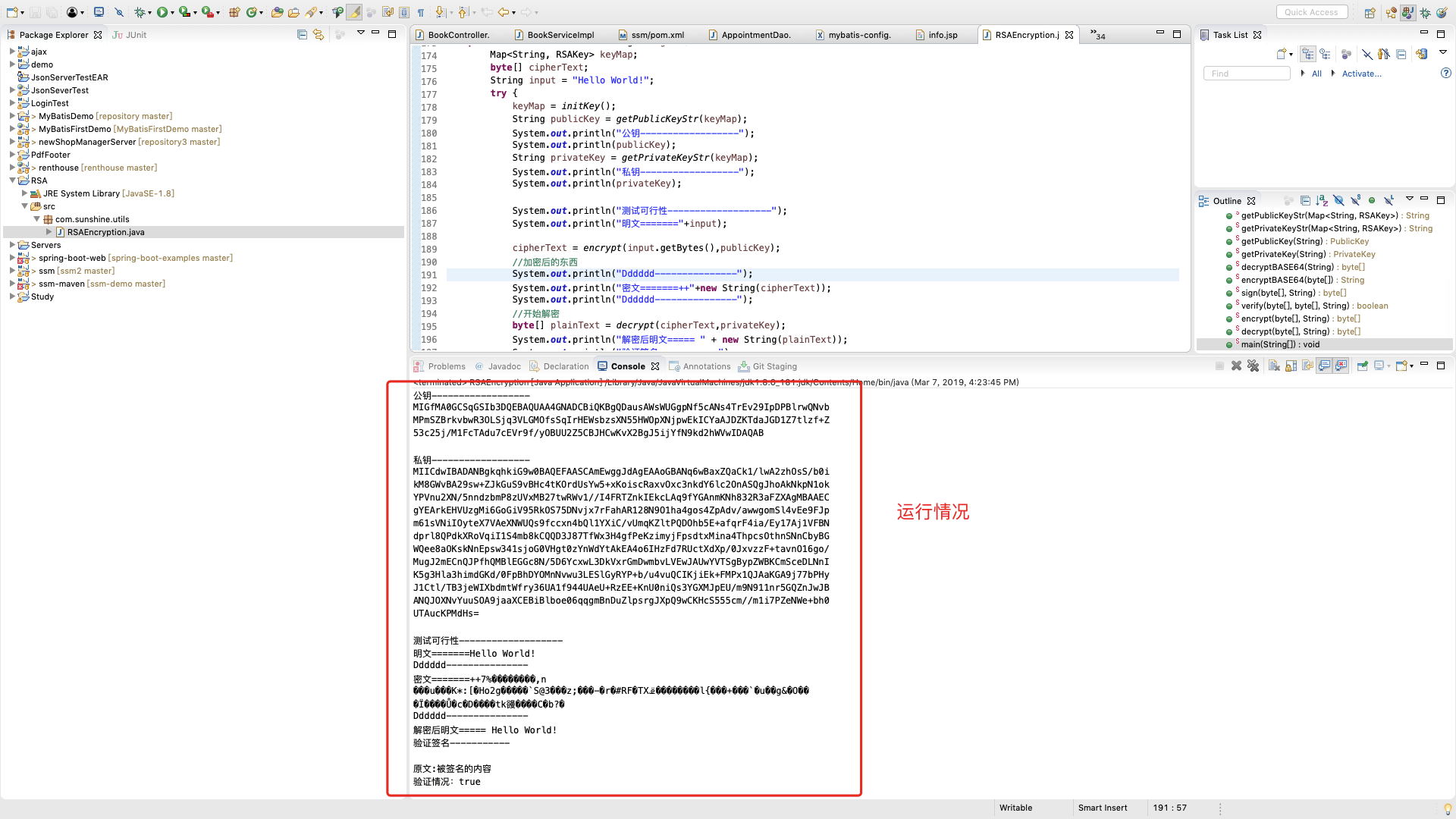
}

}

}

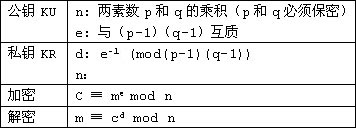
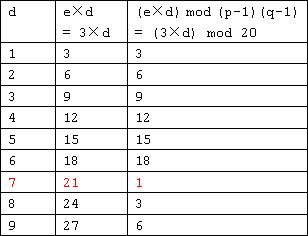
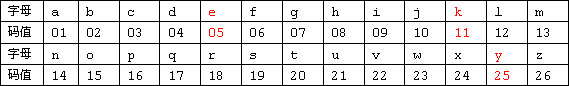
Idea可以直接使用，但是eclipse会发生错误。解决办法如下：

<https://blog.csdn.net/erlian1992/article/details/79518416>



<https://www.cnblogs.com/jiftle/p/7903762.html>

# [用实例给新手讲解RSA加密算法](https://www.cnblogs.com/jiftle/p/7903762.html)

RSA加密算法是最常用的非对称加密算法，CFCA在证书服务中离不了它。但是有不少新来的同事对它不太了解，恰好看到一本书中作者用实例对它进行了简化而生动的描述，使得高深的数学理论能够被容易地理解。我们经过整理和改写特别推荐给大家阅读，希望能够对时间紧张但是又想了解它的同事有所帮助。  
　 　RSA是第一个比较完善的公开密钥算法，它既能用于加密，也能用于数字签名。RSA以它的三个发明者Ron Rivest, Adi Shamir, Leonard Adleman的名字首字母命名，这个算法经受住了多年深入的密码分析，虽然密码分析者既不能证明也不能否定RSA的安全性，但这恰恰说明该算法有一定的可信性，目前它已经成为最流行的公开密钥算法。  
　　RSA的安全基于大数分解的难度。其公钥和私钥是一对大素数（100到200位十进制数或更大）的函数。从一个公钥和密文恢复出明文的难度，等价于分解两个大素数之积（这是公认的数学难题）。   
　　RSA的公钥、私钥的组成，以及加密、解密的公式可见于下表：  
  
  
　　可能各位同事好久没有接触数学了，看了这些公式不免一头雾水。别急，在没有正式讲解RSA加密算法以前，让我们先复习一下数学上的几个基本概念，它们在后面的介绍中要用到：  
  
**一、 什么是“素数”？**  
　　素数是这样的整数，它除了能表示为它自己和1的乘积以外，不能表示为任何其它两个整数的乘积。例如，15＝3＊5，所以15不是素数；又如，12＝6＊2＝4＊3，所以12也不是素数。另一方面，13除了等于13＊1以外，不能表示为其它任何两个整数的乘积，所以13是一个素数。素数也称为“质数”。  
  
**二、什么是“互质数”（或“互素数”）？**  
　　小学数学教材对互质数是这样定义的：“公约数只有1的两个数，叫做互质数。”这里所说的“两个数”是指自然数。  
　　判别方法主要有以下几种（不限于此）：  
（1）两个质数一定是互质数。例如，2与7、13与19。  
（2）一个质数如果不能整除另一个合数，这两个数为互质数。例如，3与10、5与 26。  
（3）1不是质数也不是合数，它和任何一个自然数在一起都是互质数。如1和9908。  
（4）相邻的两个自然数是互质数。如 15与 16。  
（5）相邻的两个奇数是互质数。如 49与 51。  
（6）大数是质数的两个数是互质数。如97与88。  
（7）小数是质数，大数不是小数的倍数的两个数是互质数。如 7和 16。  
（8）两个数都是合数（二数差又较大），小数所有的质因数，都不是大数的约数，这两个数是互质数。如357与715，357=3×7×17，而3、7和17都不是715的约数，这两个数为互质数。等等。  
  
**三、什么是模指数运算？**  
　　指数运算谁都懂，不必说了，先说说模运算。模运算是整数运算，有一个整数m，以n为模做模运算，即m mod n。怎样做呢？让m去被n整除，只取所得的余数作为结果，就叫做模运算。例如，10 mod 3=1；26 mod 6=2；28 mod 2 =0等等。   
　　模指数运算就是先做指数运算，取其结果再做模运算。如http://img.hexun.com/2009-06-24/118958534.gif  
　　好，现在开始正式讲解RSA加密算法。  
算法描述：  
（1）选择一对不同的、足够大的素数p，q。  
（2）计算n=pq。  
（3）计算f(n)=(p-1)(q-1)，同时对p, q严加保密，不让任何人知道。  
（4）找一个与f(n)互质的数e，且1<e<f(n)。  
（5）计算d，使得de≡1 mod f(n)。这个公式也可以表达为d ≡e-1 mod f(n)  
这里要解释一下，≡是数论中表示同余的符号。公式中，≡符号的左边必须和符号右边同余，也就是两边模运算结果相同。显而易见，不管f(n)取什么值，符号右边1 mod f(n)的结果都等于1；符号的左边d与e的乘积做模运算后的结果也必须等于1。这就需要计算出d的值，让这个同余等式能够成立。  
（6）公钥KU=(e,n)，私钥KR=(d,n)。  
（7）加密时，先将明文变换成0至n-1的一个整数M。若明文较长，可先分割成适当的组，然后再进行交换。设密文为C，则加密过程为：http://img.hexun.com/2009-06-24/118958535.gif。  
（8）解密过程为：http://img.hexun.com/2009-06-24/118958536.gif。   
  
**实例描述：**  
　　在这篇科普小文章里，不可能对RSA算法的正确性作严格的数学证明，但我们可以通过一个简单的例子来理解RSA的工作原理。为了便于计算。在以下实例中只选取小数值的素数p,q,以及e，假设用户A需要将明文“key”通过RSA加密后传递给用户B，过程如下：  
**（1）设计公私密钥(e,n)和(d,n)。**  
令p=3，q=11，得出n=p×q=3×11=33；f(n)=(p-1)(q-1)=2×10=20；取e=3，（3与20互质）则e×d≡1 mod f(n)，即3×d≡1 mod 20。  
d怎样取值呢？可以用试算的办法来寻找。试算结果见下表：  
  
　　通过试算我们找到，当d=7时，e×d≡1 mod f(n)同余等式成立。因此，可令d=7。从而我们可以设计出一对公私密钥，加密密钥（公钥）为：KU =(e,n)=(3,33)，解密密钥（私钥）为：KR =(d,n)=(7,33)。  
**（2）英文数字化。**　　将明文信息数字化，并将每块两个数字分组。假定明文英文字母编码表为按字母顺序排列数值，即：  
  
　　则得到分组后的key的明文信息为：11，05，25。  
**（3）明文加密**  
　　用户加密密钥(3,33) 将数字化明文分组信息加密成密文。由C≡Me(mod n)得：  
http://img.hexun.com/2009-06-24/118958539.jpg  
　　因此，得到相应的密文信息为：11，31，16。  
（**4）密文解密。**  
　　用户B收到密文，若将其解密，只需要计算http://img.hexun.com/2009-06-24/118958536.gif，即：  
http://img.hexun.com/2009-06-24/118958539.jpg  
　　用户B得到明文信息为：11，05，25。根据上面的编码表将其转换为英文，我们又得到了恢复后的原文“key”。   
　 　**你看，它的原理就可以这么简单地解释！**  
　 　当然，实际运用要比这复杂得多，由于RSA算法的公钥私钥的长度（模长度）要到1024位甚至2048位才能保证安全，因此，p、q、e的选取、公钥私钥的生成，加密解密模指数运算都有一定的计算程序，需要仰仗计算机高速完成。  
  
**最后简单谈谈RSA的安全性**  
  
　 　**首先，我们来探讨为什么RSA密码难于破解？**   
　 　在RSA密码应用中，公钥KU是被公开的，即e和n的数值可以被第三方窃听者得到。破解RSA密码的问题就是从已知的e和n的数值（n等于pq），想法求出d的数值，这样就可以得到私钥来破解密文。从上文中的公式：d ≡e-1 (mod((p-1)(q-1)))或de≡1 (mod((p-1)(q-1))) 我们可以看出。密码破解的实质问题是：从Pq的数值，去求出(p-1)和(q-1)。换句话说，只要求出p和q的值，我们就能求出d的值而得到私钥。  
　 　当p和q是一个大素数的时候，从它们的积pq去分解因子p和q，这是一个公认的数学难题。比如当pq大到1024位时，迄今为止还没有人能够利用任何计算工具去完成分解因子的任务。因此，RSA从提出到现在已近二十年，经历了各种攻击的考验，逐渐为人们接受，普遍认为是目前最优秀的公钥方案之一。  
　　然而，虽然RSA的安全性依赖于大数的因子分解，但并没有从理论上证明破译RSA的难度与大数分解难度等价。即RSA的重大缺陷是无法从理论上把握它的保密性能如何。  
　　此外，RSA的缺点还有：A)产生密钥很麻烦，受到素数产生技术的限制，因而难以做到一次一密。B)分组长度太大，为保证安全性，n 至少也要 600 bits 以上，使运算代价很高，尤其是速度较慢，较对称密码算法慢几个数量级；且随着大数分解技术的发展，这个长度还在增加，不利于数据格式的标准化。因此，使用RSA只能加密少量数据，大量的数据加密还要靠对称密码算法。