RSA算法

简介

RSA是1977年由[罗纳德·李维斯特](https://baike.baidu.com/item/%E7%BD%97%E7%BA%B3%E5%BE%B7%C2%B7%E6%9D%8E%E7%BB%B4%E6%96%AF%E7%89%B9/700199" \t "/home/lilin/文档\\x/_blank)（Ron Rivest）、[阿迪·萨莫尔](https://baike.baidu.com/item/%E9%98%BF%E8%BF%AA%C2%B7%E8%90%A8%E8%8E%AB%E5%B0%94" \t "/home/lilin/文档\\x/_blank)（Adi Shamir）和[伦纳德·阿德曼](https://baike.baidu.com/item/%E4%BC%A6%E7%BA%B3%E5%BE%B7%C2%B7%E9%98%BF%E5%BE%B7%E6%9B%BC/12575612" \t "/home/lilin/文档\\x/_blank)（Leonard Adleman）一起提出的。当时他们三人都在[麻省理工学院](https://baike.baidu.com/item/%E9%BA%BB%E7%9C%81%E7%90%86%E5%B7%A5%E5%AD%A6%E9%99%A2/117999" \t "/home/lilin/文档\\x/_blank)工作。RSA就是他们三人姓氏开头字母拼在一起组成的

RSA[公开密钥密码体制](https://baike.baidu.com/item/%E5%85%AC%E5%BC%80%E5%AF%86%E9%92%A5%E5%AF%86%E7%A0%81%E4%BD%93%E5%88%B6/11048976" \t "/home/lilin/文档\\x/_blank)是一种使用不同的加密密钥与解密密钥，“由已知加密密钥推导出解密密钥在计算上是不可行的”密码体制 。

在公开密钥密码体制中，加密密钥（即公开密钥）PK是公开信息，而解密密钥（即秘密密钥）SK是需要保密的。加密算法E和解密算法D也都是公开的。虽然解密密钥SK是由公开密钥PK决定的，但却不能根据PK计算出SK。

正是基于这种理论，1978年出现了著名的RSA算法，它通常是先生成一对RSA密钥，其中之一是保密密钥，由用户保存；另一个为公开密钥，可对外公开，甚至可在网络服务器中注册。为提高保密强度，RSA密钥至少为500位长，一般推荐使用1024位。这就使加密的计算量很大。为减少计算量，在传送信息时，常采用传统加密方法与[公开密钥加密](https://baike.baidu.com/item/%E5%85%AC%E5%BC%80%E5%AF%86%E9%92%A5%E5%8A%A0%E5%AF%86/8090774" \t "/home/lilin/文档\\x/_blank)方法相结合的方式，即信息采用改进的DES或IDEA对话密钥加密，然后使用RSA密钥加密对话密钥和信息摘要。对方收到信息后，用不同的密钥解密并可核对信息摘要。

RSA是被研究得最广泛的公钥算法，从提出到现在已近三十年，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。1983年[麻省理工学院](https://baike.baidu.com/item/%E9%BA%BB%E7%9C%81%E7%90%86%E5%B7%A5%E5%AD%A6%E9%99%A2/117999" \t "/home/lilin/文档\\x/_blank)在美国为RSA算法申请了专利

RSA允许你选择公钥的大小。512位的密钥被视为不安全的；768位的密钥不用担心受到除了国家安全管理（NSA）外的其他事物的危害；1024位的密钥几乎是安全的。RSA在一些主要产品内部都有嵌入，像 Windows、网景 Navigator、 Quicken和 Lotus Notes。

算法原理

RSA公开密钥密码体制的原理是：根据数论，寻求两个大素数比较简单，而将它们的乘积进行因式分解却极其困难，因此可以将乘积公开作为加密密钥

算法描述

RSA算法的具体描述如下:

（1）任意选取两个不同的大素数p和q计算乘积 n = pq, φ(n) = (p-1)(q-1)；

（2）任意选取一个大整数e，满足gcd(e, φ(n))= 1，整数e用做加密钥（注意：e的选取是很容易的，例如，所有大于p和q的素数都可用）；

（3）确定的解密钥d，满足 (de)mod φ(n) = 1，即 de = kφ(n)+1,k>=1是一个任意的整数；所以，若知道e和φ(n)，则很容易计算出d；

（4）公开整数n和e，秘密保存d；

（5）将明文m（m<n是一个整数）加密成密文c，加密算法为

C = E(m)=m^e mod n

1. 将密文c解密为明文m，解密算法为

m = D(c)=c^d mod n

然而只根据n和e（注意：不是p和q）要计算出d是不可能的。因此，任何人都可对明文进行加密，但只有授权用户（知道d）才可对密文解密

安全性

RSA的安全性依赖于大数分解，但是否等同于大数分解一直未能得到理论上的证明，也并没有从理论上证明破译。RSA的难度与大数分解难度等价。因为没有证明破解RSA就一定需要做大数分解。假设存在一种无须分解大数的算法，那它肯定可以修改成为大数分解算法，即RSA的重大缺陷是无法从理论上把握它的保密性能如何，而且密码学界多数人士倾向于因子分解不是[NPC问题](https://baike.baidu.com/item/NPC%E9%97%AE%E9%A2%98/8698778" \t "/home/lilin/文档\\x/_blank)。

目前，RSA的一些变种算法已被证明等价于大数分解。不管怎样，分解n是最显然的攻击方法。现在，人们已能分解140多个十进制位的大素数。因此，模数n必须选大些，视具体适用情况而定。

RSA算法的保密强度随其密钥的长度增加而增强。但是，密钥越长，其加解密所耗用的时间也越长。因此，要根据所保护信息的敏感程度与攻击者破解所要花费的代价值不值得以及系统所要求的反应时间来综合考虑，尤其对于商业信息领域更是如此

实现步骤

(1)选择p、q两个超级大的质数，都是1024位。

( 2 ) 令 n = p ∗ q 。 取 φ ( n ) = ( p − 1 ) ∗ ( q − 1 ) 。 计 算 与 n 互 质 的 整 数 的 个 数 。 (2)令n = p \* q。取 φ(n) =(p-1) \* (q-1)。 计算与n互质的整数的个数。 (2)令n=p∗q。取φ(n)=(p−1)∗(q−1)。计算与n互质的整数的个数。

( 3 ) 取 e ∈ { 1 < e < φ ( n ) } ， ( n , e ) 作 为 公 钥 对 (3)取 e ∈ \{1 < e < φ(n)\} ，( n , e )作为公钥对 (3)取e∈{1<e<φ(n)}，(n,e)作为公钥对

正 式 环 境 中 取 65537 , 依 据 来 源 于 H T T P 证 书 。 正式环境中取65537,依据来源于HTTP证书。 正式环境中取65537,依据来源于HTTP证书。

( 4 ) 令 e d m o d φ ( n ) = 1 ， 计 算 d ， ( n , d ) 作 为 私 钥 对 。 (4)令 ed \quad mod \quad φ(n) = 1，计算d，( n , d ) 作为私钥对。 (4)令edmodφ(n)=1，计算d，(n,d)作为私钥对。

计 算 d 可 以 利 用 扩 展 欧 几 里 的 算 法 进 行 计 算 ， 非 常 简 单 。 计算d可以利用扩展欧几里的算法进行计算，非常简单。 计算d可以利用扩展欧几里的算法进行计算，非常简单。

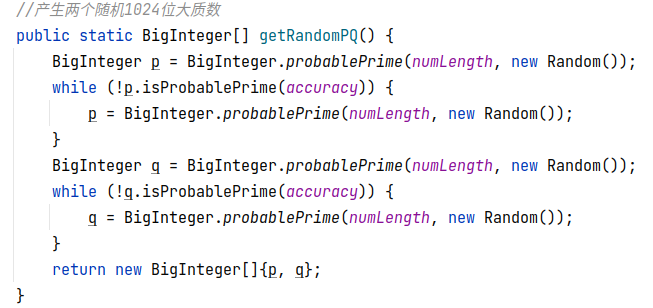
( 5 ) 销 毁 p 、 q 。 密 文 = 明 文 e m o d n ， 明 文 = 密 文 d m o d n 。

实现难点

如何储存大数，动态语言一般不存在这个问题。不过在Java中就需要借助BigInteger,其他静态语言可能需要借助第三方库才能存储。

如何计算一个超大整数的超大整数次幂，然后取超大整数模，这个文字有点绕，大概就是这个样子: 超 大 数 超 大 数 m o d 超 大 数 超大数^{超大数} \quad mod \quad 超大数 超大数超大数mod超大数,这个问题呢就需要使用到蒙哥马利方法了。

如何产生两个随机1024位大质数？这个问题在Java直接利用如下方法即可



源代码

详细可见项目工程文件夹RSA

package com.company;  
  
import javax.security.sasl.SaslServer;  
import java.math.BigInteger;  
import java.util.Random;  
import java.util.stream.StreamSupport;  
  
public class RSA {  
 private final static int numLength = 1024;*//素数长度* private final static int accuracy = 100;*//素数的准确率为1-(2^(-accuracy))  
  
 //获取最大公约数* private BigInteger getGCD(BigInteger a, BigInteger b) {  
 if (b.byteValue() == 0) return a;  
 return getGCD(b, a.mod(b));  
 }  
  
 *//扩展欧几里得方法,计算 ax + by = 1中的x与y的整数解（a与b互质）* private static BigInteger[] extGCD(BigInteger a, BigInteger b) {  
 if (b.signum() == 0) {  
 return new BigInteger[]{a, new BigInteger("1"), new BigInteger("0")};  
 } else {  
 BigInteger[] bigIntegers = extGCD(b, a.mod(b));  
 BigInteger y = bigIntegers[1].subtract(a.divide(b).multiply(bigIntegers[2]));  
 return new BigInteger[]{bigIntegers[0], bigIntegers[2], y};  
 }  
 }  
  
 *//超大整数超大次幂然后对超大的整数取模，利用蒙哥马利乘模算法,  
 //(base ^ exp) mod n  
 //依据(a \* b) mod n=(a % n)\*(b % n) mod n* private static BigInteger expMode(BigInteger base, BigInteger exp, BigInteger mod) {  
 BigInteger res = BigInteger.ONE;  
 *//拷贝一份防止修改原引用* BigInteger tempBase = new BigInteger(base.toString());  
 *//从左到右实现简答  
 /\*  
 D=1  
 WHILE E>=0  
 　　 IF E%2=0  
 　　 C=C\*C % N  
 　　 E=E/2  
 　　ELSE  
 　　 D=D\*C % N  
 　　 E=E-1  
 　　RETURN D  
 \*/* for (int i = 0; i < exp.bitLength(); i++) {  
 if (exp.testBit(i)) {*//判断对应二进制位是否为1* res = (res.multiply(tempBase)).mod(mod);  
 }  
 tempBase = tempBase.multiply(tempBase).mod(mod);  
 }  
 return res;  
 }  
  
 *//产生公钥与私钥* public static SecretKey generateKey(BigInteger p, BigInteger q) {  
 *//令n = p \* q。取 φ(n) = (p-1) \* (q-1)。* BigInteger n = p.multiply(q);  
 *//计算与n互质的整数个数 欧拉函数* BigInteger fy = p.subtract(BigInteger.ONE).multiply(q.subtract(BigInteger.ONE));  
 *//取 e ∈ [1 < e < φ(n) ] ，( n , e )作为公钥对，这里取65537* BigInteger e = new BigInteger("65537");  
 *//计算ed与fy的模反元素d。令 ed mod φ(n) = 1，计算d，然后将( n , d ) 作为私钥对* BigInteger[] bigIntegers = extGCD(e, fy);  
 *//计算出的x不能是负数，如果是负数，则进行x=x+fy。使x为正数，但是x<fy。* BigInteger x = bigIntegers[1];  
 if (x.signum() == -1) {  
 x = x.add(fy);  
 }  
 *//返回计算出的密钥* return new SecretKey(n, e, x);  
 }  
  
 public static SecretKey generateKey() {  
 BigInteger[] pq = getRandomPQ();  
 return generateKey(pq[0], pq[1]);  
 }  
  
 *//加密* public static BigInteger encrypt(BigInteger text, SecretKey.PublicKey publicKey) {  
 return expMode(text, publicKey.e, publicKey.n);  
 }  
  
 *//解密* public static BigInteger decrypt(BigInteger cipher, SecretKey.PrivateKey privateKey) {  
 return expMode(cipher, privateKey.d, privateKey.n);  
 }  
  
 *//加密* public static String encrypt(String text, SecretKey.PublicKey publicKey) {  
 return encrypt(new BigInteger(text.getBytes()), publicKey).toString();  
 }  
  
 *//解密* public static String decrypt(String chipper, SecretKey.PrivateKey privateKey) {  
 BigInteger bigInteger = expMode(new BigInteger(chipper), privateKey.d, privateKey.n);  
 byte[] bytes = new byte[bigInteger.bitLength() / 8 + 1];  
 for (int i = 0; i < bytes.length; i++) {  
 for (int j = 0; j < 8; j++) {  
 if (bigInteger.testBit(j + i \* 8)) {  
 bytes[bytes.length - 1 - i] |= 1 << j;  
 }  
 }  
 }  
 return new String(bytes);  
 }  
  
 *//产生两个随机1024位大质数* public static BigInteger[] getRandomPQ() {  
 BigInteger p = BigInteger.probablePrime(numLength, new Random());  
 while (!p.isProbablePrime(accuracy)) {  
 p = BigInteger.*probablePrime*(*numLength*, new Random());  
 }  
 BigInteger q = BigInteger.*probablePrime*(*numLength*, new Random());  
 while (!q.isProbablePrime(*accuracy*)) {  
 q = BigInteger.*probablePrime*(*numLength*, new Random());  
 }  
 return new BigInteger[]{p, q};  
 }  
  
 *//密匙对* static class SecretKey {  
 BigInteger n, e, d;  
  
 public SecretKey(BigInteger n, BigInteger e, BigInteger d) {  
 this.n = n;  
 this.e = e;  
 this.d = d;  
 }  
  
 public PublicKey getPublicKey() {  
 return new PublicKey(n, e);  
 }  
  
 public PrivateKey getPrivateKey() {  
 return new PrivateKey(n, d);  
 }  
  
 *//密钥* static class PrivateKey {  
 public BigInteger n, d;  
  
 public PrivateKey(BigInteger n, BigInteger d) {  
 this.n = n;  
 this.d = d;  
 }  
 }  
  
 *//公钥* static class PublicKey {  
 public BigInteger n, e;  
  
 public PublicKey(BigInteger n, BigInteger e) {  
 this.n = n;  
 this.e = e;  
 }  
 }  
 }  
  
  
 public static void main(String[] args) {  
 SecretKey secretKey = RSA.generateKey();  
 *//明文内容不要超过1024位,超过后需要分段加密* String text = "Hello world";  
 String chipper = RSA.encrypt(text, secretKey.getPublicKey());  
  
 System.out.println("公钥:"+secretKey.getPublicKey());  
 System.out.println("私钥:"+secretKey.getPrivateKey());  
 System.out.println("明文:" + text);  
 System.out.println("加密后:\n" +  
 *//密文长度可能会随着随机密钥的改变而改变，最长不超过2048位* "密文二进制长度:" + new BigInteger(chipper).bitLength()  
 + "\n"  
 + chipper);  
 String origin = RSA.decrypt(chipper, secretKey.getPrivateKey());  
 System.out.println("解密后:\n" + origin);  
 }  
}

运行结果

