**RSA** 加密算法研究与实现

## 陈伟剑 (Chen Weijian) 2022 年 5 月 8 日

摘要**:** RSA 密码算法是目前在信息安全领域广为使用的公钥加密方法。该算法涉及欧几里得、欧拉函数、乘法逆元等基本数论知识，易于实现。基于 Java 的BigInterger 类型，设计并实现了RSA 公钥密码系统中的密钥生成、加密算法、解密算法。其中，通过将扩展欧几里得算法转换成递推形式降低了算法的空间复杂度。

关键字**:** RSA Java BigInteger

**Abstract:** RSA cryptosystem is the most widely used public key cryptosystem in the field of information security. This algorithm involves Euclid, Euler function, multiplicative inverse and other basic number theory knowledge, which is easy to implement. Based on Java language and its BigInteger class , the key generation , encryption , decryption and digital signature of RSA public key cryptosystem are designed and implementd. Especially, the extended Euclidean algorithm is transformed into a recursive form to reduce the space complexity of the algorithm.

**Keywords:** RSA Java BigInteger

# 引言

现代密码学体系加密算法一般分为对称加密算 法和非对称加密算法（或公钥加密算法）两种。所谓对称加密就是通信双方加密和解密消息时使用相同 的密钥，对称密码算法的优点是加解密运算速度较 快，适用于处理大批量消息处理，缺点是很难解决密钥分发的安全性和数字签名等问题。在非对称加密 算法中，发送方和接收方使用不同的密钥，发送方使用接收放生成的公钥对明文进行加密生成密文，接 收方通过自己生成的私钥对密文进行解密获取明文， 非对称密码算法的优点是不需要额外开辟安全信道 保证传输的安全，缺点是加密速度较慢，故非对称密码算法更适用于处理密钥分发和数字签名。一般实 践中，两种加密算法会一起使用，形成混合密码算 法，这种算法使用对称加密算法处理传输内容，使用非对称加密算法对密钥进行加密。

RSA 密码体系(又成RSA 算法) 是目前比较流行

的密码加密算法，该算法在1977 年由Riverst、Shamir、Adleman3 位专家提出并于 1978 年首次发表。RSA 算法整体上易于实现，但是当算法中相关的参数值很 大时，需要考虑溢出问题，为此可以使用 Java 中的BigInteger 类型。不过鉴于 Java 中使用 BigInteger 类型会形成大量的对象方法调用从而导致代码可读性 较低，故本文只在一处较为简单的辅助算法中给出 一般实现和 BigInteger 实现的两种代码。

1. **RSA** 算法原理

## 算法概述

整个加密过程需要用到的参数如下

表 1: RSA 算法相关符号

 *ax* + *by* = *gcd*(*a, b*)

 *bx*

′

|  |  |
| --- | --- |
| 符号 | 说明 |
| *p*、*q* | 两个随机素数 |
| *n l e d* | p 和 q 的乘积n 的欧拉数  (1*, l*) 区间内和 *l* 互质的随机整数  *e* 在模 *l* 下的乘法逆元 |

*a*

+ (*a* − *b*⌊ *b* ⌋)*y*′

*a*

= *gcd*(*b, a* − *b*⌊ *b* ⌋)

由于等式右侧相等，所以有 *ax* + *by* = *bx*′ +(*a* −

接收方生产私钥和公钥的基本过程如下: 1、随机生成两个大素数 *p, q*

2、令 *n* = *p* ∗ *q*

3、令 *l* = *ϕ*(*n*) = (*p* − 1) ∗ (*q* − 1)

*b*⌊ *a* ⌋)*y*′，整理得到 *a*(*x* − *y*′) + *b*(*y* −(*x*′

因为 *a, b* 均不为 0，所以有

*b*



*x* − *y*′ = 0 *a*



*y* − (*x*′ − ⌊ *b* ⌋*y*′) = 0

即

*a*

−⌊ *b* ⌋

*y*′)) = 0，

4、随机生成一个数 *e*，满足 1 *< e < l* 且 *e* 和 *l*

互质

5、由乘法逆元的原理，通过(*d* ∗ *e*)*modl* = 1 计

 *x* = *y*′

 *y* = *x*

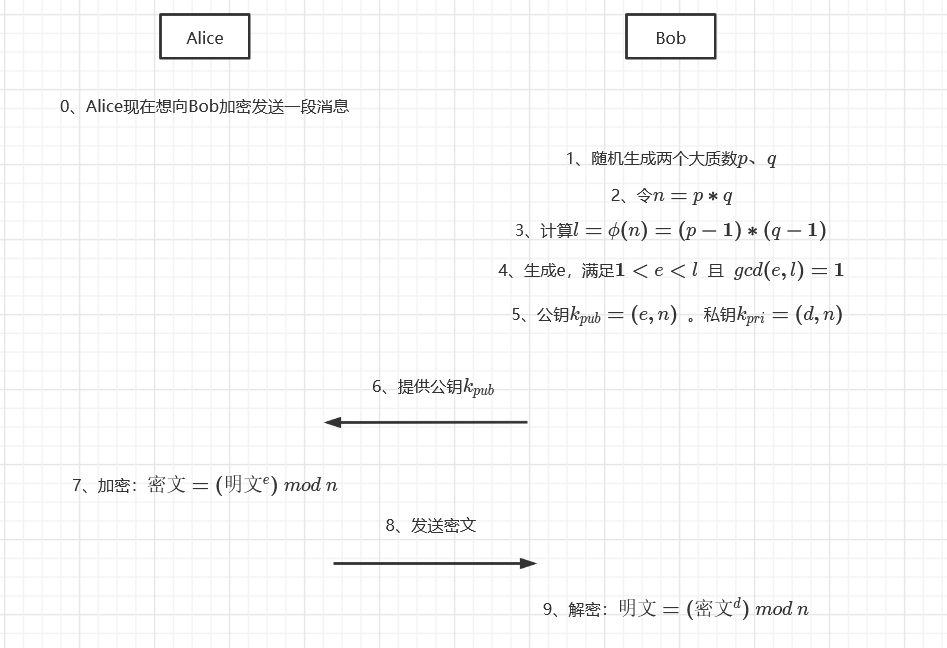
′



*a*

— ⌊ *b* ⌋*y*′

算得到 *d*

6、(*e, n*) 作为公钥，(*d, n*) 作为私钥整体流程 如图:

[[2](#_bookmark0)]

因此，要求解 *x, y*，只需要递归求解更小规模的 *x*′*, y*′ 即可。计算到最后的边界条件 *b* = 0 时，有*x* = 1*, y* = 0。

在数据足够大时，采用递归方式的空间复杂度会急剧增加。现考虑将扩展欧几里得算法转化为递推形式以降低空间复杂度。

对于先前已知的递归式

 *x* = *y*′

 *y* = *x*

′

可以转换为矩阵形式

*a*

— ⌊ *b* ⌋*y*′

(*x*) = (0 1

*y*

1 −*d*

) (*x*′) *,* (*d* = *a* )

1

## 辅助算法实现

*y*′

1

⌊ *b* ⌋

* + 1. 扩展欧几里得算法与乘法逆元

*x*′*, y*′ 同样可以展开成上述形式，由此一直递推到 *x* = 1*, y* = 0，形成下式

* + - 1. 递推形式扩展欧几里得算法推导

欧几里得算法的一个用途是求解(*a, b*) = *ax*+*by*

中的 *x, y*。在欧几里得算法中，递归部分的核心等式

(*x*) =

(0 1

) (0 1

) (0 1

) *...*

(0 1

) (1

为 *gcd*(*a, b*) = *gcd*(*b, a mod b*) = *gcd*(*b, a* − *b*

*a*

⌊ *b* ⌋) ，

*y* 1 −*d*1

1 −*d*2

1 −*d*3

1 −*dn* 0

由此式可以得出和 *x, y* 相关的递归关系。根据裴蜀

定理，可以找到四个整数 *x, y, x*′*, y*′, 使得

每次迭代时，都可以从左向右计算一个矩阵，从

而达到了迭代的目的。

假设在某次迭代之前计算的矩阵之积为

( )

*m*1 *m*2 (迭代开始之前是一个单位阵)，下一个待

*n*1 *n*2

(

return a;

}

右乘的矩阵为 0 1

1 −*dt*

(*m*1 *m*2) (0 1

)，二者之积可以写成

) = (*m*2 *m*1 − *dtm*2)

下面为采用 BigInteger 的代码形式 (由于可读性相对低，本文只在此处展示 BigInteger 相关代码)

public static BigInteger exgcd(BigInteger

*n*1 *n*2

1 −*dt*

*n*2 *n*1 − *dtn*2

a,BigInteger b,BigInteger[] array){

BigInteger m=new BigInteger("0");

所 有 矩 阵 计 算 完 成 之 后， 得 到 一 个 最

终 矩 阵

(*M*1

*M*2)

， 这 个 矩 阵 满 足

(*x*) =

BigInteger n=new BigInteger("1"); while(b.compareTo(zero)>0){

*N*1 *N*2 *y*

( ) ( )，展开可得 *x* = *M , y* = *N* 。

*M*1 *M*2 1

1 1

*N*1 *N*2 0

综上，递推的算法步骤如下:

( ) ( )

1、给矩阵赋初值 *x m* = 1 0

BigInteger d=a.divide(b); BigInteger t;

t=m;

m=array[0].add(d.negate().multiply(t)); array[0]=t;

t=n;

*y n*

2、每次迭代时，计算 *d* =

3、更新矩阵

0 1

*a*

⌊ *b* ⌋

n=array[1].add(d.negate().multiply(t)); array[1]=t;

t=a.mod(b); a=b; b=t;

}

(*x m*) (*x m*) (0 1 )

(*m x* − *dm*)

return a;

*y n* ← *y n*

=

1 −*d*

}

*n y* − *dn*

4、进行一般欧几里得算法的迭代

1. ← ( *b* )

*b*

*a mod b*

* + - 1. 乘法逆元计算

当使用扩展欧几里得算法找到一组 *x, y* 满足

5、*b* = 0 时，计算结束，此时已经求出需要的

*x, y*，并且最大公约数存如变量 *a* 中。递推代码如下

//java无法传入指针，这里array[0]代表x，array[1]代表y public static long exgcd(long a,long

b,long[] array){ long m=0;

long n=1; while(b>0){

long d=a/b; long t;

t=m; m=array[0]-d\*t; array[0]=t; t=n; n=array[1]-d\*t; array[1]=t; t=a%b; a=b; b=t;

}

*ax* + *by* = 1 时，就可以得出 *ax* ≡ 1 (*mod b*)，即*x* 是 *a* 的乘法逆元, 那么对 ∀*x* + *kb, k* ∈ N 也都是 *a* 的乘法逆元，这表明，一定有 *a* 的一个乘法逆元在区间 [0*, b*) 内。

利用前面完成的扩展欧几里得算法，找到 *n* 在模 *p* 下的乘法逆元。代码如下

public static long inv(long a,long p){ long[] array={1,0};

//array[0]代表x，array[1]代表y if(exgcd(a,p,array)==1){

array[0]%=p;

return array[0]>=0 ? array[0]:array[0]+p;

}

else return -1;

//-1代表不存在对应的乘法逆元

}

* + 1. 快速模幂运算

快速模幂算法基于快速幂多了一些取模运算，整体思想相同。快速幂的核心思想是每一步都将指数减半，相应的底数做平方运算，这样可以达到快速降低指数的目的。

例如：已知 *a, b* 求 *ab*，假设 *b* = 11。

这里可以将表达式的指数写成二进制的形式

*a*11 = *a*23+21+20 = *a*(1011)2

指数减半等价于对指数的二进制形式右移一位， 假如当前末位为 0，则直接进行移位运算。若当前末尾不为 0，则提出一个当前底数后，再将指数右移一位。如 *a*(1011)2 = *a* ∗ *a*(1010)2 = *a* ∗ (*a*2)(101)2

相关代码如下

public static long qmod(long a,long p,long mod){

long ans=1; a%=mod; while(p>0){

if((p&1)==1){ ans=(ans\*a)%mod;

}

a=(a\*a)%mod; p>>=1;

}

return ans;

}

* + 1. 大素数判定
       1. 费马素性测试

该测试基于费马小定理。费马小定理对所有素数都成立，因此，对于一个待测数 *p*，可以随机挑选[2*, p* − 2] 中的任一整数 *t*，若存在 *t* 不满足 *at*−1 ≡ 1 (*mod p*)，说明 *p* 不是素数。测试次数越大，则素数判定的正确性越高。

但是某些合数也满足费马小定理，这样的合数

称为卡迈克尔数。给定卡迈克尔数 *C*，对所有满足

*gcd*(*a, C*) = 1 的整数 *a*，都满足 *aC*−1 ≡ 1 (*mod C*)。*n* = 561 = 3 ∗ 11 ∗ 17 就是一个卡迈克尔数。

* + - 1. **Miller-Rabin** 算法测试

费马素性测试的测试范围不够大，实践中更多会采用 Miller-Rabin 算法进行测试 。

[[4](#_bookmark1)]

假设 *n* 是奇素数，则 *n* − 1 必为偶数，令 *n* − 1 =

2*q* ∗ *m*。

随机选取整数 *a*(1 *< a < n*)，由费马小定理知

(*a*2*q* ∗*m* = *an*−1) ≡ 1 (*mod n*) 。

由二次探测定理知：*a*2*q−*1∗*m* ≡ 1 (*mod n*) 或

*a*2*q−*1∗*m* ≡ *n* − 1 (*mod n*)。

若 *a*2 ∗*m* ≡ 1 (*mod n*) 成立，则再由二次探测定理可知：*a*2 ∗*m* ≡ 1 (*mod n*) 或 *a*2 ∗*m* ≡ *n* − 1 (*mod n*)。

*q−*1

*q−*2 *q−*2

如此反复，直到某一步*am* ≡ 1 (*mod n*) 或 *am* ≡

*n* − 1 (*mod n*) 。如果 *n* 是素数，则 *am* ≡ 1 (*mod n*)

，或存在 0 ≤ *r* ≤ *q* − 1 使 *a*2 ∗*m* ≡ 1 (*mod n*)。

*r*

算法思路为:

1、给定奇数 n，为了判定 n 是不是素数，首先测试 *a*2*q* ∗*m* ≡ 1(*modn*)，是否成立 (这一步需要使用快速模幂算法)，若不成立，则 n 一定是合数；若成立， 则继续运行算法。

2、考察下面的 Miller 序列：

*a*20*m*(*modn*)*, a*21*m*(*modn*)*, a*22*m*(*modn*)*, ...a*2*q−*1*m*(*modn*)*,*

若 *am* ≡ 1(*modn*)，或存在 0 ≤ *r* ≤ *q* − 1 使

*a*2 *m* ≡ 1(*modn*) 成立，则通过测试。

*r*

相关代码如下

public static boolean millerRabin(long num){ if(num==2) return true;

else if(num<2 || (num&1)==0) return false;

int judgeTime=100;

//随机挑100个数进行判定，可以定义判断的次数long m=num-1;

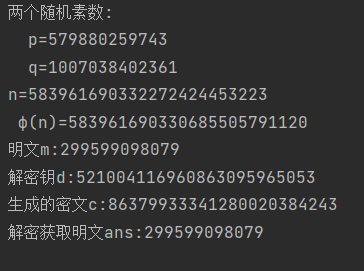
long t=0;

//计算 num-1=m\*(2^t) 的m和t while((m&1)==0) {

m >>= 1; t++;

}

**3.2** 测试 **2**

for(int i=1;i<=judgeTime;i++){

//[2,num-1]中的随机数a long

a=(long)(Math.random()\*(num-2)%num+2);

//计算 a^m % num

long x=qmod(a,m,num),y=x;

for(int j=0;j<t;j++){ y=qmod(x,2,num); //x^2 mod num

//不满足二次探测定理，也就是y==1的同时

//x并不等于1或者n-1，那么n就一定不是素数if(y==1 && x!=1 && x!=num-1){

return false;

}

x=y;

}

//子循环结束后，y仍不为1，则num不是素数if(x!=1) return false;

}

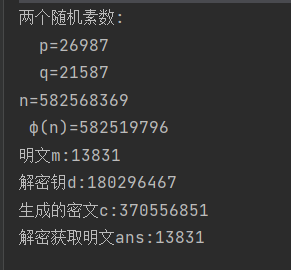
return true;

}

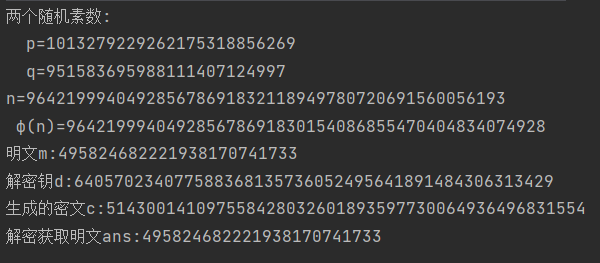
# 实验结果

测试会随机生成一串长数字作为传输消息，同时在控制台打印相关参数的信息，此处给出三次测试结果

* 1. 测试 **1**



**3.3** 测试 **3**



# 总结

RSA 是目前使用广泛的一种公钥密码系统，它的安全性基于大整数因子分解难题 。目前主流的电子计算机的算力尚不能快速破解经RSA 加密的密钥， 同时，RSA 生成的密钥长度从以前的 1024 位发展到了现在的 2048 位和 3082 位，因此目前 RSA 加密的安全性还是有所保证的。RSA 密码系统可用于消息加密和数字签名，密码强度主要与 RSA 模数 N 的长度有关，目前长度是 2048 位，即 256 字节，非常安全。经过本次 RSA 算法的粗略设计与实现，一定程度上能够提高对数论与编程的认知，这对今后了解 更前沿的密码算法大有裨益。

[[5](#_bookmark2)]

# 参考文献

1. 费马小定理https://zhuanlan.zhihu.com/p/87611586
2. RSA 算法分析https://zhuanlan.zhihu.com/p/36347853
3. 扩 展 欧 几 里 得 算 法

https://zhuanlan.zhihu.com/p/58241990

1. Miller-Rabin 素 性 测 试

https://blog.csdn.net/ECNU\_LZJ/article/details/72675595

1. 王生玉, 汪金苗, 董清风, 等. 基于属性加密技术研究综述 [J]. 信息网络安全,2019,19(9)：76-80