RSA 非对称加密算法的一个实现

作者：胡辉 17727536501

一 引言

RSA 非对称加密算法应用广泛，实用性强。据我所知，它在数据通信和数据存储领域是普遍使用的基础技术。

科技时代，数学就是魔法，不懂数学者是麻瓜。或用先贤的话，有“不懂几何者不得入此们”的道理。本文重点描述编程方面的数据结构和算法，数学性质和证明是算法成立的依据，在这方面的描述我采取简介形式，力求百分百正确，但不一定够详细。

二 数学性质与证明

大（整）数，二进制 1024 位或 2048 位的大数字因数分解在数学上很困难，当前的计算机硬件和算法据说最多能分解 150 位（比较接近 1024 二进制位，远不到 2048 二进制位）十进制位的数。将一个大数的高次幂对另一个大数模运算取剩余，确是可行并且容易用计算机实现的，大数的这个性质被用来设计一种非对称的加密/解密算法，即 RSA（三位发明者的名字缩写）算法。

本文主题涉及到的数学知识具体是指数论，其他分支不涉及（如果学过抽象代数群论知识，会更容易理解）。我把他们分成三部分（可对应到算法实现的三个部分）来介绍：**算数基本定理之求最大公约数，费马小定理和欧拉公式，拉宾米勒素性测试公式**。另外再单独讲解述 RSA 算法的数学证明。

基本符号和术语：

1.a | b　a 整除 b，a 是分，即 b 除以 a 没有剩余（剩余为 0）

2.模运算 整数相除，得商与剩余

3.同余式 a≡b mod m，这个是模运算的数学符号描述形式，即数 a 与 b 模 m

同余

4.gcd 最大公约数的数学简写形式，gcd (a,b)即数 a 与 b 的最大公约数

5.a 的 b 次幂记做 a^b

2.1 算数基本定理和 G.C.D 算法：

任何整数 n>=2 可唯一分解成素数的乘积形如 n=p1*p2*p3*…*pn

欧几里得带余除法（或称辗转相除法）是求 gcd 的数学算法，要求数 A 与 B 的最大公约数，写成 A=Q*B+R（Q 即商 Quotient 的缩写 R 即余 Remainder 的缩写）。然后将 B 放到公式中 A 的位置，R 放到公式中 B 的位置得到新的 Q 与 R，按此步骤一直进行下去直到得到 R 等于 0，则倒数第二步的 R 值即为数 A 与 B 的最大公约数。

欧几里得算法不仅可用于计算出最大公约数，还可以用于解形如 ax+by=gcd(a, b)的线性方程。即求数 a 与 b 的最大公约数的欧几里得算法过程的中间结果中含有以上方程的根。

2.2 费马小定理和欧拉公式：

p 是任意素数，a 是任意整数且 a 不被 p 整除，则有

a^p-1≡1 mod p

欧拉Φ函数：在 1 与 m 之间且与 m 互素的整数的个数(计数函数)，记做Φ(m)。

对于素数 p 有，Φ(p)=p-1

对任意数 m 和 n，如果 gcd(m,n)=1 有Φ(mn)=Φ(m)Φ(n)

对任意素数 p 有 Φ(p^k)=p^k-p^(k-1)

欧拉公式：如果数 a 与 m 互素，记 gcd(a,m)=1，则有

a^Φ(m)≡1 mod m

2.3 拉宾米勒素性测试公式：

设 n 是奇数，记 n-1=(2^k)*q，q 是奇数，对不被 n 整除的某个 a，如果下述两个条件都成立，则 n 是合数。

(a) a^q !≡1 mod n

(b) 对所有 i=0,1,2,…,k-1, a^(2iq) !≡−1 mod n

由于素数分解的困难性，要判断一个数是否是素数没有简单的方法，只有找出它的因数。拉宾米勒测试公式并不能证明一个数是否是素数，它能证明一个数“很可能是素数”。选 a 的 100 个不同值，其中没有一个能通过测试，则 n 是合数的概率近似等于 6*10^(-61)。（非常小，所以可以确定 n 是素数）

2.4 RSA 算法的数学证明:

算法描述：有数字（自然数）a,k,m,b,u，有以下关系，

1.m=p1p2p3...pn，p1 与 p2...pn 两两互不相等，即数 **m 没有相同的因数**。

2.**k 与Φ(m)互素**，即 gcd(k,Φ(m))=1

3.u 是方程 ku≡1 mod(Φ(m))的（同余）解

4.a<m，b 是 a 的 k 次幂模 m 剩余，即有同余方程 **a^k≡b mod m**

若数字 a 是已知的数，则对 a 求 k 次幂然后模 m 可得数 b

若将以上同余方程中的 **a 看做未知数**，其他三个数 k,b,m 已知数，需要证明方程 x^k≡b mod m 的解是 a。已知 k 与 m 可通过解方程 ku≡1 mod(Φ(m)求出 u，若证明 b^u≡a mod m 即证明了 **b^u（即 a）是方程 x^k≡b mod m 的解**。

以上，数字 a 是原文，b 是密文，**(k,m)构成公钥，(u,m)构成私钥**。加密过程的计算是将原文 a 做 k 次幂后模 m 得剩余 b，将 b 当做密文。解密的过程是将密文 b 做 u 次幂后模 m 得剩余，这个剩余就是原文 a。

以下是证明过程：

由 3 可将同余方程转为等式方程 ku-1=Φ(m)*v,变形为 ku=Φ(m)v+1

由 4 即 a^k≡b mod m 两边自乘 u 次得 **a^ku≡b^u mod m**

把上一步中的等式代换进同余式，得 **a^(Φ(m)v+1) ≡b^u mod m**

左边展开并减 a 得 **a(a^Φ(m)v - 1)**

另外，m 是不同素数的乘积，a<m，所以 a 不整除 m，假设 gcd(a,m)=p1 可得 gcd(a,p2p3...pn)=1，所以 **a^Φ(p2p3...pn)**≡1 mod p2p3…pn,两边自乘Φ(p1)v 次可得 a^Φ(m)v ≡ 1 mod p2p3…pn,所以 a^Φ(m)v-1 整除 p2p3…pn。

根据假设 gcd(a,m)=p1,所以 a(a^Φ(m)v-1)整除 p1*(p2p3…pn)=m 即

a(a^Φ(m)v-1) ≡ 0 mod m 所以 a^(Φ(m)v+1) ≡a≡ b^u mod m

证明完毕 QED

三 概要设计

RSA 加密算法的核心是算数运算，包括加法，乘法，除法和取余运算。参与运算的数学对象是自然数，这刚好可用最常用的整型类型表示。C 语言提供现成的整型算数运算符，然而语言的内置整型类型最多 8 字节（64 位），所以现成的类型和运算符是不够的。

基础假定：

硬件：x86-64

操作系统：Unix 和 Windows

编程语言：C

专用硬件：随机数发生器

从抽象的数学性质到具体的软件代码，实际上换了一种思维角度，即从软件代码的角度去拼接实现数学性质。整型数据计算实在是非常基础的计算，不需要太多的基础假定。

硬件假定为 x86-64， 实际上 arm 和 x86-32 的硬件理论上也能行，或者只需要少量修改。操作系统跟这些算法的关系更加疏远，因为算法的核心是用户代码，会用到少量通用的 I/O 接口，只考虑 posix 标准，所以系统是 Unix 还是 Windows 没有关系。编程语言，C 是不二选择。随机数发生器，这个一般也是硬件集成的，但是需要操作系统提供统一的访问接口（如/dev/random）。

我的开发机器是一台 macbookpro， intel i5 处理器。

关于硬件有一点需要明确，这里只支持小端机器。

算数运算：

在集合和函数理论中，算术运算加减乘除本质上是函数，只不过人们习惯用+-*\这样的符号表示它们。C 语言中现成的+-*\运算符仅作用于内置数据类型，因此需要做的事情就是设计正确的算术运算函数，作用于更大的数据类型。

大数字类型需要使用 C 的派生类型实现，即数组和结构。

最基础的算术运算包括，加法，减法，乘法，除法和取余。

素数生成器

素数难以分解的性质是 RSA 算法成立的理论依据，我们需要一个实用的素数生成器来生成和分发素数。这些素数必须具有随机性以至于不能被人轻易猜到，而且需要有足够大以至于两个素数的乘积不被人轻易分解，“不轻易“这个词在这里实际上意思是”不可能“。

随机数来自硬件设备“随机数发生器“，从生成的随机数里面分辨出合数和素数，则是依靠”拉宾米勒素性测试“算法。

性能设计

算术运算是基础的软件组件，肯定有现成的优秀的实现。既然打算重新造盒子，那就要要达到一定的水准，达到实用的水平。

软件的性能受很多因素影响，硬件和环境只能去适应而不能预设，但是程序员能控制程序的设计，通过良好的设计保证软件的功能和性能。

算术运算函数的设计思路十分简单，就是模仿 ISA 层的算术运算指令。以加法为例，x86-64 机器上最常用的加法指令有 addb,addw,addl,addq，他们分别是字节，字，双字和四字加法指令。受此启发，我将数据类型设计成 1024 位，2048 位和 4096 位等不同规格的类型，设计处理对应数据类型的加法函数。

一个 1024 位二进制数字转成对应的十进制有 300 多位，2048 位对应几乎翻倍的大小。任何一个这样的数字都是天文数字，超过地球上的沙子数量。一次 RSA 加密（或解密）运算实际上有 3 个数字对象参与（公式 a^u mod m）。假设三个数字都是 1024 位的，使用“蒙哥马利算法“，这样一次计算需要进行大约 1024 次平方并 1024 次取余运算。

加法是最简单的，理论上，早期的 ALU 里面的算术运算模块就只有加法器，其他运算通过加法间接实现。现代的 CPU 有乘法甚至除法器，但是运算理论并没有发展，乘法器内部其实还是加法器。

而除法是最复杂和耗时的，除法跟乘法一样，通过加法间接实，可以用一个比喻和伪代码形象的说明除法的复杂性：

1. 小学生除法：

```
def r,q; //r 是余 remainder 的缩写，q 是商 quotient 的缩写
q = 0;
while(a>b) {
    a -= b;
    q++;
}
r = a;
```

2.中学生除法：

```
def r, q;
q = 0;
again:
midb = b*w;
while(a>midb) {
a -= midb;
q++;
}
q *= w;
if(a>b) goto again;
```

3.大学生除法：

```
def r;
again:
a = a1*w + a2;
r = ((a1 mod b)*w + (a2 mod b)) mod b;
goto again;
```

用简单的语言描述，计算 a/b 的商和余。小学生除法就是一次从 a 中减掉一个 b，不断循环直到余数小于 b 为止。中学生除法是一次从 a 中减掉 b 的 w 倍，不断循环直到余数小于 b。大学生除法是将 a 拆分成两个数字，然后两个数字分别跟 b 做“与“位运算，运算结果相加再取余。

三种算法的性能一次从低到高，大学生除法平均性能最高，但是只得到余，得不到商，所以中学生除法和大学生除法都是用的到好的算法，而小学生除法是

明显低效的，在计算两个大数字（一大一小）相除的时候并不实用。

四 数据结构

```c
/*basic integer size*/
#define NUMBERSIZE 128

/**
 * Big integer type identifier, mainly used 1024, 2048, 4096 bits Integer
 * Word: 128 bytes, 1024 bits
 * DWord: 256 bytes, 2048 bits
 * QWrod: 512 bytes, 4096 bits
 * Note: they woking on little edian machine only
 */
typedef struct {
    uint8_t val[NUMBERSIZE];
} N128;
typedef struct {
    uint8_t val[NUMBERSIZE * 2];
} N256;
typedef struct {
    uint8_t val[NUMBERSIZE * 4];
} N512;
```

C 的内置算术整型，在最常用的 X86-64 机器上最大为 64 bits，显然不满足我满需求的大数字运算。这里使用标准库的 uint8_t 类型数组，将数组作为命名结构的唯一成员，结构命名规则为 Nxxx，xxx 为数组的字节长度。N128 即 128 bytes 的整型（1024 位）， N256 是 256 bytes 的整型（2048 位）。

借鉴机器对内置类型长度的分类方式，我将 N128 类型简称为字（word），N256 类型简称为双字（DWord）， N512 类型简称为四字（QWord），这样在变量和函数命名的时候，使用简称 w，d，q 表示他们。

物理机器的字长通常是 32 位或 64 位，我的程序实际上模拟一个字长为 1024 位的虚拟机器。物理机器的算数运算指令通常的命名风格是“运算+长度”，如 addb,addw,addl,addq 分别是 1 字节， 双字节，四字节，八字节的加法指令。因此，我的大数字运算函数按照类似风格命名，如：

```
addw(N128 *, const N128 *)
addd(N256 *, const N256 *)
addq(N512 *, const N512 *)
cmpw(const N128 *, const N128 *)
cmpd(const N256 *, const N256 *)
cmpq(const N512 *, const N512 *)
incw(N128 *)
```

带 w 后缀的是 1024 位运算，d 后缀的是 2048 位运算，q 后缀的是 4096 位运算。

五 算法实现

5.1 大数字对象加法

两数相加实际上参与运算的有 3 个数，第 3 个数是进位。

```c
static inline void addw(N128 *a, const N128 *b)
{
    uint32_t *pa, *pb;
    uint64_t sum, of;

    pa = (uint32_t *)(a->val);
    pb = (uint32_t *)(b->val);
    of = 0;

    for(int i = 0; i < (sizeof(a->val) >> 2); i++) {
        sum = *pa;
        sum += *pb;
        sum += of;
        of = (sum >> 32);
        *pa = sum;

        pa++;
        pb++;
    }
}
```

5.2 大数字对象减法（和取加法逆）

加法就是加法逆运算，固定宽度的数字提供固定的取值范围，因此取加法逆变得很容易（按位取反，再加 1）。

```c
static inline void invw(const N128 *restrict a, N128 *restrict inv)
{
    uint64_t *pa, *pi;

    pa = (uint64_t *)(a->val);
    pi = (uint64_t *)(inv->val);
    for(int i = 0; i < (sizeof(a->val) >> 3); i++) {
        pi[i] = ~pa[i];
    }
    incw(inv);
}
```

5.3 大数字对象乘法

在通常的理解上，机器的乘法是加法实现的，但是至少 x86 机器是有乘法指令的。因此，乘法运算有两种实现方式，一是逐位移位相加，一是用直接乘法。据我所知，乘法指令并不需要逐位移位相加，而是使用一种连加器的技术实现，得到所有的移位结果后，一次性将他们相加。例如，两个 32 位数字相乘，并不需要做 32 移位并且 32 次加法（假设被乘数的位全是 1）， 而是做 32 次移位并 1 次加法，所以性能得到提升。

然而，“连加器”是 ISA 层以下的逻辑，我们的代码无法直接使用到它们。所以我只实现了用内置乘法实现的大数字乘法，没有做移位加法的实现。

使用“竖式乘法“实现的大数字乘法代码：

```c
static inline void vmultw(N128 *a, N128 *b)
{
    N256 midval;                                                        /*middle and final result*/
    uint32_t *pa, *pb, *pmid;                                           /*travel pointers*/
    uint64_t sum, ofadd, ofmlt;                                         /*ofadd is addition overflow bits, ofmlt is mult overflow bits*/
    size_t zcnta, zcntb;                                                /*leading zero counters*/
    union {
        uint64_t v64;
        struct {
            uint32_t l;
            uint32_t r;
        } v32;
    } va, vb;                                                           /*union va and vb used to do multipication*/

    pa = (uint32_t *)(a->val);                                          /*init vars*/
    pb = (uint32_t *)(b->val);
    pmid = (uint32_t *)(midval.val);
    zcnta = skiplzw(a, NULL);
    zcntb = skiplzw(b, NULL);
    dzero(&midval);

    for(size_t cnt1 = 0; cnt1 < (sizeof(b->val) >> 2) - zcntb; cnt1++) {    /*outter loop depends on len of b*/
        vb.v64 = *pb++;                                                 /*set vb*/
        pa = (uint32_t *)(a->val);                                      /*set pa*/
        pmid = (uint32_t *)(midval.val) + cnt1;                         /*set pmid, pmid moving with outter loop*/

        ofadd = 0;                                                      /*init ofadd and ofmlt*/
        ofmlt = 0;
        for(size_t cnt2 = 0; cnt2 < (sizeof(a->val) >> 2) -zcnta; cnt2++) { /*inner loop depends on len of a*/
            va.v64 = *pa++;                                             /*set va*/
            va.v64 *= vb.v64;
            va.v64 += ofmlt;
            ofmlt = va.v32.r;                                           /*reset ofmlt*/

            sum = *pmid;                                                /*read pmid value*/
            sum += va.v32.l;
            sum += ofadd;
            ofadd = (sum >> 32);                                        /*reset ofadd*/
            *pmid++ = sum;                                              /*write back pmid value*/
        }
        *pmid = ofmlt + ofadd;                                          /*write back last pmid*/
    }

    *a = castdtw(&midval);                                              /*write final result to a, cast from N256 to N128*/
}
```

5.4 大数字对象的除法（和取剩余）

除法实际上分为取模和常规除法，模运算是数论中很基础的一种运算，除法

也是所有数学中的常用运算，他们的最明显区别是，模运算的计算结果是“余“，常规的除法运算结果就是“商“。

在计算机上，乘法是用加法实现，除法也是，并且，除法是所有整型运算的**性能瓶颈**。

乘法运算非常直观的可以分解为逐个移位，然后相加。除法则更复杂，因为把分母拆分之后，并不能把整个数字拆分，所以这里需要一些拆分的数学技巧。

“除法“实现的核心代码：

```
if(quo) {                                                   /*with quo*/
    wzero(quo);                                             /*init quo*/
    wzero(&quotient);                                       /*init quotient*/
    extrashift = 0;                                         /*init extrashift flag*/
    while(cmpw(&remainder, b) >= 0) {
        if(zcntb > zcnta) {
            /* check the value of first non-zero byte of remainder to do extra 4 bits shift or not */
            if (remainder.val[sizeof(remainder.val) - zcnta - 1] & 0xf0) {
                extrashift = 1;
                salw(&origin, (zcntb - zcnta - 1) * 8 + 4); /*left shift origin and inverse, key operation for division efficiency*/
                salw(&inverse, (zcntb - zcnta - 1) * 8 + 4);
            } else {
                salw(&origin, (zcntb - zcnta - 1) * 8);     /*left shift origin and inverse, key operation for division efficiency*/
                salw(&inverse, (zcntb - zcnta - 1) * 8);
            }
        }
        while(cmpw(&remainder, &origin) >= 0) {
            addw(&remainder, &inverse);
            incw(&quotient);
        }
        if(zcntb > zcnta) {
            if (extrashift) {
                extrashift = 0;
                salw(&quotient, (zcntb - zcnta - 1) * 8 + 4);/*left shift quotient the same distance*/
            }
            else
                salw(&quotient, (zcntb - zcnta - 1) * 8);
        }
        addw(quo, &quotient);                               /*add middle result quotient to final result quo*/
        wzero(&quotient);                                   /*clear quotient*/

        copyw(&origin, b);                                  /*reset origin and inverse for next loop*/
        invw(&origin, &inverse);

        zcnta = cntlzw(&remainder);                         /*upgrate zcnta value, increasing at here*/
    }
```

取模运算的代码：

```c
static inline void modw(N128 *a, const N128 *b, const N128 *bmask)
{
    size_t lenOfA, lenOfB, weight, cnt;                 /*bits width, value weight and loop counter*/
    N128 ha, la, aval, maskb, invb;                     /*middle results*/

    if (bmask)                                          /*init vars*/
        copyw(&maskb, bmask);
    else
        wToMask(b, &maskb);

    invw(b, &invb);                                     /*-b*/
    lenOfB = bitLenw(b);
    lenOfA = bitLenw(a);

    if (lenOfB == 0) {                                  /*arithmetic error*/
        raise(SIGFPE);
        return;
    }

    wzero(&aval);                                       /*a=0*/
    weight = 0;                                         /*init weight = 0*/
    while (lenOfA) {
        copyw(&la, a);
        sarw(&la, weight * lenOfB);
        maskw(&la, &maskb);
        copyw(&ha, a);
        sarw(&ha, (weight + 1) * lenOfB);
        maskw(&ha, &maskb);

        if (cmpw(&la, b) >= 0) {
            addw(&la, &invb);
        }
        cnt = (weight * lenOfB);
        while (cnt) {
            salw(&la, 1);
            if (cmpw(&la, b) >= 0) {
                addw(&la, &invb);
            }
            cnt--;
        }
        addw(&aval, &la);

        if (cmpw(&aval, b) >= 0) {
            addw(&aval, &invb);
        }
```

```c
        if (lenOfA > lenOfB) {                          /*if lenOfA > lenOfB, split a*/
            if (cmpw(&ha, b) >= 0) {
                addw(&ha, &invb);
            }
            cnt = (weight + 1) * lenOfB;
            while (cnt) {
                salw(&ha, 1);
                if (cmpw(&ha, b) >= 0) {
                    addw(&ha, &invb);
                }
                cnt--;
            }
            addw(&aval, &ha);

            if (cmpw(&aval, b) >= 0) {
                addw(&aval, &invb);
            }
        }

        weight += 2;
        if (lenOfA > (lenOfB << 1))
            lenOfA -= (lenOfB << 1);
        else
            lenOfA = 0;
    }
    copyw(a, &aval);
}
```

简述就是，除法运算不对被除数进行拆分，而是直接使用移位和“减法“。 取模运算对被除数进行拆分（通过移位掩码和按位于运算实现），然后分段取模后叠加。大部分情况下，后者性能更好。

5.5 求最大公约数和解线性等式方程

古老的欧几里得除法，又称辗转相除法。

题目描述：求解等式方程 ax+by=gcd(a,b)=1（用字母 g 代换 gcd(x,y)，等式方程等价于同余式方程 ax≡g mod b 在等式方程的意义下有无穷多解，我们只需要求出值在范围(0, b)之间的一个特解，特解（或称同余方程的解）有且仅有一个。

算法步骤如下：

1. 置 x=1，g=a，v=0 与 w=b
2. 如果 w=0，则置 y=（g-ax）/b 返回值(g,x,y)
3. g 除以 w 得余 t，g=qw+t (0<=t<w)
4. 置 s=x-qv
5. 置(x,y)=(v,w)

6. 置(v,w)=(s,t)
7. 转到第 2 步

首先是求最大公约数算法，代码如下：

```
/**
 * GCD func
 * input: a, b
 * output: g
 * return: a for success, -1 for fail
 */
int gcdw(const N128 *a, const N128 *b, N128 *g)
{
    N128 aval, bval, rem;                              /*middle results*/

    if((a == NULL) || (b == NULL)) {                   /*illegal para*/
        raise(SIGFPE);                                 /*raise SIGFPE, should terminal the process by default*/
        return -1;
    }
    copyw(&aval, a);                                   /*init vars*/
    copyw(&bval, b);

    while(skiplzw(&bval, NULL) != (sizeof(bval.val) / 4)) {    /*loop condition: bval not equals 0*/
        modew(&aval, &bval, &rem, NULL, NULL);               /*aval mod bval -> rem*/
        copyw(&aval, &bval);                           /*bval -> aval*/
        copyw(&bval, &rem);                            /*rem -> bval*/
    }

    if(g) {                                            /*g is not NULL*/
        copyw(g, &aval);                               /*aval -> g*/
        return 1;
    }
    raise(SIGFPE);                                     /*g is NULL, exception*/
    return -1;
}
```

解线性等式方程式基于 GCD 算法，解方程实际上是它的“副产物“，核心代码：

```
qzero(&x512);                                          /*init vars*/
x512.val[0] = 1;                                       /*give x 1*/
qzero(&v);                                             /*give v 0*/
a512 = castdtq(a);
b512 = castdtq(b);

while(skiplzq(&b512, NULL) != (sizeof(b512.val) / 4)) {    /*loop condition: b not equals 0*/
    modeq(&a512, &b512, &r512, &q512, NULL);               /*a mod b, remainder to rem, quotient to quo*/

    invq(&v, &inv);                                    /*v = -v*/
    vmultq(&q512, &inv);                               /*q=q*(-v)*/
    addq(&x512, &q512);                                /*x=x-q*v*/
    copyq(&s, &x512);                                  /* s = x - q*v */

    copyq(&x512, &v);                                  /*v->x*/
    copyq(&v, &s);                                     /*s->v*/

    copyq(&a512, &b512);                               /*b->a*/
    copyq(&b512, &r512);                               /*r->b*/
}
```

注：这里涉及到了“减法“， 所以需要处理”符号溢出“，将我们的无符号数据升到更大的宽度以容纳可能的”符号溢出“。

另外，这里还用到“除法“，所以要使用带”商“的版本。

5.6 幂模求剩余算法（或称逐次平方求剩余算法）

逐次平方计算 a^k mod m 的数学描述：

1. 将数字 k 进行二进制展开，形如

k=(2^0)*u[0]+(2^1)u[1]+(2^2)*u[2]+…+(2^r)*u[r] 其中 u 是二进制位的值，2 的幂即是权的值

2. 逐次平方制作模 m 的幂次表，利用性质 a^2 mod m 等于 a mod m 的平方，a^4 mod m 等于 a^2 mod m 的平方。所以次数再高也只需要对两个小于 m 的数做乘法然后模 m 求剩余。用代码实现的时候当然不需要存储一张这样的“幂次表”，只需要把它们作为中间结果存在同一个对象中。
3. 乘积（累乘）

   a^k=a^((2^0)*u[0]+(2^1)u[1]+(2^2)*u[2]+…+(2^r)*u[r])

   =a^((2^0)*u[0]))*a^((2^1)*u[1]))*…

代码实现逻辑描述：

1. 置 b=1
2. 当 k>=1 时，循环

   2.1 如果 k 是奇数，则置 b=a*b(mod m)

   2.2 置 a=a^2 mod m

   2.3 置 k=k/2
3. 结束循环，b 的值就是最终计算结果

逐次平方法在数学上学名好像叫做“蒙哥马利法“， 实现代码：

```c
int powerModew(const N128 *restrict a, const N128 *restrict b, const N128 *restrict k, N128 *restrict rem)
{
    N256 a256, b256, r256, b256mask;                                    /*middle results*/
    uint32_t wk, *pwk;                                                  /*used for binary explore k*/
    int lenOfk, i, j;                                                   /*len of k and loop counters*/

    if((a == NULL) || (b == NULL) || (k == NULL) || (rem == NULL)) {    /*illegal para*/
        raise(SIGFPE);                                                  /*raise SIGFPE, should terminal the process by default*/
        return -1;
    }

    a256 = castwtd(a);                                                  /*init vars, cast a and b so that mult overflow won't happen*/
    b256 = castwtd(b);
    dToMask(&b256, &b256mask);
    lenOfk = (sizeof(k->val) >> 2) - skiplzw(k, NULL);
    pwk = (uint32_t *)(k->val);

    moded(&a256, &b256, &r256, NULL, &b256mask);
    if(skiplzd(&r256, NULL) == (sizeof(r256.val) >> 2)) {              /*check if a % b = 0*/
        wzero(rem);                                                     /*set rem=0*/
        return 1;
    }

    dzero(&r256);
    r256.val[0] = 1;                                                    /*init r=1*/
    for(i = 0; i < lenOfk; i++) {                                       /*binary expansion k*/
        wk = pwk[i];
        if (i == (lenOfk - 1)) {
            for(; wk >= 1;) {                                           /*the highest part of k, only expand the significant bits*/
                if(1 == (wk & 0x01)) {
                    vmultd(&r256, &a256);
                    modd(&r256, &b256, &b256mask);                      /*(r*a) mod b*/
                }
                vmultd(&a256, &a256);
                modd(&a256, &b256, &b256mask);                          /*(a*a) mod b*/
                wk >>= 1;
            }
        } else {
            for(j = 0; j < 32; j++) {                                   /*the reset parts of k, expand all 32 bits*/
                if(1 == (wk & 0x01)) {
                    vmultd(&r256, &a256);
                    modd(&r256, &b256, &b256mask);
                }
                vmultd(&a256, &a256);
                modd(&a256, &b256, &b256mask);
                wk >>= 1;
            }
        }
    }

    *rem = castdtw(&r256);                                              /*set rem*/
    return 1;
}
```

注：这里大量使用“取模运算“，本来取模运算就是性能瓶颈，而这个逐次平方运算又是 RSA 加解密的**主要运算**。所以，提升取模运算的性能至关重要。

5.7 拉宾米勒测试

拉宾米勒测试算法数学语言描述：

设 n 是奇数，记 n-1=(2^k)*q，q 是奇数，对不被 n 整除的某个 a，如果下述两个条件都成立，则 n 是合数。

(a) a^q !≡1 mod n

(b) 对所有 i=0,1,2,...,k-1, a^((2^i)q) !≡−1 mod n

拉宾米勒测试是核心的数学理论，但是在计算机算法上，这里算不上不是核心了（核心算法是取模和逐次平方），拉宾米勒算法的关键代码：

```
for(i = 0; i < t; i++) {
    cont:
    read(fdUrandom, evidence.val, lenOfn - 1);          /*make sure evidence < n*/
    evidence.val[0] |= 0x02;                            /*make sure evidence > 1*/

    /*sorting order of evidences to avoid duplicating*/
    while(1) {
        cmp = cmpw(&evidence, &(candidate->stack[candidate->rsp]));
        if (cmp > 0) {
            pushEvdw(candidate, &evidence);
            while(exchange->rsp) {
                pushEvdw(candidate, popEvd(exchange));
            }
            break;
        } else if (cmp < 0) {
            pushEvdw(exchange, popEvd(candidate));
            continue;
        } else {
            while(exchange->rsp) {
                pushEvdw(candidate, popEvd(exchange));
            }
            goto cont;
        }
    }

    powerModew(&evidence, &nval, &qval, &rem);          /*a^q mod n remaind rem*/
    if(cmpw(&rem, &one128) == 0)                        /*a^q mod n remaind 1*/
        continue;
    if(cmpw(&rem, &nsub1) == 0)                         /*a^q mod n remaind -1*/
        continue;

    rem256 = castwtd(&rem);
    for(j = 1; j < k; j++) {                            /*loop count k-1*/
        vmultd(&rem256, &rem256);                       /*(a^q)^2*/
        modd(&rem256, &n256, &n256mask);                /*(a^q)^2 mod n*/
        if(cmpd(&rem256, &nsub1256) == 0)               /*compare (a^q)^2 mod n remaind and -1*/
            goto ppoint;
    }
    ppoint:
    if (j == k) {
        free(candidate);
        free(exchange);
        return -1;                                      /*passed one evidence's power mode calculation*/
    }
}
```

5.8（随机数）素数生成器

随机数的生成需要有“随机“的输入，目前可靠的随机数通常使用硬件产生。在 Unix 系统上，此设备是/dev/random 和/dev/urandom。据说是根据硬件的热和电磁在物理上的随机特性，产生随机输入而生成随机数的。

使用的方法也很直接，用 read 系统调用直接读取设备就行，其中/dev/urand om 的生产的性能更高，但是据说随机性强度不如/dev/random。

对随机数进行拉宾米勒测试，就能得到素数，以下是“素数生成器“的代码，使用的是/dev/urandom 设备：

```c
int primeGenw(N128 *prime, int t, size_t *realLen)
{
    N128 pTest;
    int fd, cnt;

    cnt = 0;
    if (-1 == (fd = open("/dev/urandom", O_RDONLY))) {
        return -1;
    }

    if(*realLen > sizeof(pTest.val))
        *realLen = sizeof(pTest.val);
    if(*realLen < 9)
        *realLen = 9;

    wzero(&pTest);
    for(;;) {
        if (*realLen != read(fd, pTest.val, *realLen)) {
            printf("\n(pid:%u)random device error\n",getpid());
            close(fd);
            return -1;
        }

        if (pTest.val[0] & 0x01) {
            cnt++;
            if (MRtestw(&pTest, t, fd) == 1) {
                copyw(prime, &pTest);
                close(fd);
                printf("\n(pid:%u)total tested %d numbers\n",getpid(), cnt);
                return cnt;
            } else {
                write(STDOUT_FILENO, ".", 1);
            }
        }
    }
}
```

六 代码实现（C 语言）

https://github.com/adam-hh/lean01.git

七 演示

在 Makefile 中定义了两个 Demo 小程序，make 出来即可进行演示。

1.构建

make pgen

make caltest

2.素数生成器

```
adamhs-MacBook-Pro:lean01 adamh$ ./pgen 512
generating prime number...
......................................................................................................
......................................................................................................
......................................................................................................
......................
(pid:91612)total tested 160 numbers
generated a 512 bits prime number
54513463430754647448470820264693835564429203749678837908628074126278275873788091675108322762955195309912175547851428053002253549705
45671737112078729377031
.......................
(pid:91613)total tested 280 numbers
generated a 512 bits prime number
90751539628285304150282287156523443705614179511771531023982692509845005170428733834656078773063206024954846742199120535931729468977
90218066591189959307863
wait finished: No child processes
adamhs-MacBook-Pro:lean01 adamh$ 
```

3. 2048 数据位加密解密

我从一个 json 文件中读取明文（”a”），素数和公钥，执行 caltest 程序，caltest 程序计算出私钥，并演示将一段明文加密，然后解密打印出来

testdata.json 内容（可以任意修改 “a” 字段值，演示不同明文的加解密结果）

```
{} testdata.json > ...
{
    "a" : "RSA RSA RSA!",
    "b" : "1374419086053712692567468369329850322866944142739185120321438040400543720357409383194792535817019862276756877804489268262626674519876130796
    "c" : "5866572343006918675018826205300876233291545576054495876686092361984196222257682813130949395101407801633640335353289356265824975208480071157
    "p" : "2244242233",
    "RSA1024" : {
        "mode1024" : "",
        "fai1024" : "",
        "publicKey1024" : "65537",
        "privateKey1024" : "123",
        "p1024" : "16165111324360650043426666192590643219889177992355738447490628230025618913039752021394893236742399441765181639366772012828774889408
        "q1024" : "66925421668499308446541808940759023366406365034195119546792271228555168024863337985862498498228028098507727297929514428433233861265
    },
    "RSA2048" : {
        "mode2048" : "",
        "fai2048" : "",
        "publicKey2048": "65537",
        "p2048" : "61057349364537628643099989813526132899821090361993759100516569069279332083019577182927427884587660710969140362553151118445957569843
        "q2048" : "14521466510813568029373321401583217894087958772798497521484352849626577841754062630641629972314057930936445453773740707203087317024
    }
}
```

演示结果

```
adamhs-MacBook-Pro:lean01 adamh$ ./caltest
rsa2048.mode :
8866422540361772633768585409316870730430990292836365921956343090373817070668851417736242286296904498488569556341175278753650435598
95264662850379140253365166793356591796016758567230823035161062656846239835457488741971302958214443811250861229085937603202718705698
01688663241321537712864328494831613843441673910081749605773633094073441407896560889314470744269311319126780883365883489033705108906
71024915981758062957748566534139794433828583574915263116302480467563910693021641321414331081178091336057137670978569090563218465110
92206499592177511464142651208754162615887986032429511109706605186312097652057239267530764429
rsa2048.fai :
8866422540361772633768585409316870730430990292836365921956343090373817070668851417736242286296904498488569556341175278753650435598
95264662850379140253365166793356591796016758567230823035161062656846239835457488741971302958214443811250861229085937603202718705698
01688663241321537712864328494831613843441673889454548158506302200390121024960729705244402935271437887590771126811372438977684759972
33748838699357859621799563628557889665521182678622542948552823103139669751609429427026377708625285391648144590874670336818634821257
12578699023994984353631102210346311407689871809759285622319415285623345453756215576848886460
rsa2048.publicKey :
65537
rsa2048.p :
61057349364537628643099989813526132899821090361993759100516569069279332083019577182927427884587660710969140362553151118445957569843
16931389704927445694268287605142012864461600797210729479791175487444590810248506372838451781666786611765247546694201731792286468518
9165479551912619297881781215546891248251466699
rsa2048.q :
14521466510813568029373321401583217894087958772798497521484352849626577841754062630641629972314057930936445453773740707203087317024
64599581197256912794997412180898010607941793292831732996429875531359439672719733006569951099839003164094580360820563905332853334290
33504745935474570602806970982754132442430411271
rsa2048.privateKey :
13973980868730144732623665821266453724417332174721254003202573626813420762667478924787488694556673734911400200611074651788248704537
91876172277973757414544590198034396396860660378731513055681197128073680688167606103663908757730701590344540421978861551543111242674
44064302647658729618935496727392400314157026559106703509913660915632841907118412150776957106953609135922075697221945861455368812819
54123499014688913621825345876670803398922262170796530907969576114749372856010708402150776742182134867484530654121251657979457086359
84027730628787466521028665558854800349879132183697814382607339983294986575397835584376339293
plain text a
RSA RSA RSA!
cypher text a
qjM&B0?q_aIv???{?1@h????-??v???Q??vʋ?_U?g?y?M
                                          ???L?Yc
decrypted text a
RSA RSA RSA!
adamhs-MacBook-Pro:lean01 adamh$
```