

SHA256 实现及其在区块链中的作用分析

姓名：毕予然

学号：3180102070

课程：区块链与数字货币

指导老师：杨小虎

I. 实验要求

本实验要求提交一个 SHA256 的代码实现，并用实验数据说明 SHA256 在比特币区块链中发挥的作用。

II. SHA256 代码实现

SHA256 是一种安全散列算法，也就是一个哈希函数。对于任意长度的消息，SHA256 都会产生一个 256bit 长的哈希值，称作消息摘要，相当于是个长度为 32 个字节的数组，通常用一个长度为 64 的十六进制字符串来表示。

A. 实现原理

1) 常量初始化

SHA256 算法中用到了 8 个哈希初值以及 64 个哈希常量。其中，8 个哈希初值 $h_0 \sim h_7$ 是对自然数中前 8 个质数的平方根的小数部分取前 32bit，64 个哈希常量 $k_0 \sim k_{63}$ 是对自然数中前 64 个质数的立方根的小数部分取前 32bit。

2) 信息预处理

信息预处理是在原文后填充一定内容，使整个消息满足指定的结构并达到 64 字节即 512 位。首先附加填充比特——先补第一个比特为 1，然后都补 0，直到长度满足对 512 取模后余数是 448（一定要填充）；其次附加长度值——一个 64bit 的数据（足够长），用来表示原始报文的长度信息。

3) 设置逻辑运算

除了常规与、或、右移等逻辑运算，还需要循环右移运算，可以自己提前在宏定义处定义好。

4) 循环计算摘要

这是计算的主体部分。首先将消息分解成 x 个 512-bit 大小的块；我们需要作 x 次迭代计算得到最终 hash 值，做法是将一个 256-bit 的摘要的初始值 H_0 （由 $h_0 \sim h_7$ 构成）经过第一个数据块进行运算得到 H_1 ，即完成了第一次迭代，在将 H_1 经过第二个数据块得到 H_2 ，依次处理，最后得到 H_n ， H_n 即为最终的 256-bit 消

息摘要；每次迭代中，首先需要构造 64 个字，前 16 个字 $w_0 \sim w_{15}$ 由块分解得到，其余由公式¹计算得出；其次进行 64 次加密循环完成一次迭代，64 个常量 $k_0 \sim k_{63}$ 分别为其密钥，加密方式是一些逻辑运算公式²。

```
s0 := (w[i-15] rightrotate 7) xor (w[i-15] rightrotate 18) xor(w[i-15] rightshift 3)
s1 := (w[i-2] rightrotate 17) xor (w[i-2] rightrotate 19) xor(w[i-2] rightshift 10)
w[i] := w[i-16] + s0 + w[i-7] + s1
```

公式 1

```
s0 := (a rightrotate 2) xor (a rightrotate 13) xor(a rightrotate 22)
maj := (a and b) xor (a and c) xor(b and c)
t2 := s0 + maj
s1 := (e rightrotate 6) xor (e rightrotate 11) xor(e rightrotate 25)
ch := (e and f) xor ((not e) and g)
t1 := h + s1 + ch + k[i] + w[i]
```

公式 2

B. 实现过程

我们使用伪代码形式将整个过程显示如下：

```
4 //Initialize variables
5 h0 := 0x6a09e667
6 h1 := 0xbb67ae85
7 h2 := 0x3c6ef372
8 h3 := 0xa54ff53a
9 h4 := 0x510e527f
10 h5 := 0x9b05688c
11 h6 := 0x1f83d9ab
12 h7 := 0x5be0cd19
13
14 ▽ k[0..63] :=
15     0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
16     0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
17     0xe49b69c1, 0xefbe4786, 0xfc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
18     0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
19     0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
20     0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
21     0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
22     0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2
23
24
25 //Pre-processing
26 append the bit '1' to the message
27 append k bits '0',such that the resulting message length (in bits) = 448(mod 512)
28 append length of message as 64-bit big-endian integer
29
30
31 //Process the message in successive 512-bit chunks:
32 break message into 512-bit chunks
33 for each chunk
34     break chunk into sixteen 32-bit big-endian words w[0..15]
35
36     //Extend the sixteen 32-bit words into sixty-four 32-bit words:
37     for i from 16 to 63
38         s0 := (w[i-15] rightrotate 7) xor (w[i-15] rightrotate 18) xor(w[i-15] rightshift 3)
39         s1 := (w[i-2] rightrotate 17) xor (w[i-2] rightrotate 19) xor(w[i-2] rightshift 10)
40         w[i] := w[i-16] + s0 + w[i-7] + s1
41     end for
42 end for
```

```

//Main loop:
for i from 0 to 63
    s0 := (a rightrotate 2) xor (a rightrotate 13) xor(a rightrotate 22)
    maj := (a and b) xor (a and c) xor(b and c)
    t2 := s0 + maj
    s1 := (e rightrotate 6) xor (e rightrotate 11) xor(e rightrotate 25)
    ch := (e and f) xor ((not e) and g)
    t1 := h + s1 + ch + k[i] + w[i]
    h := g
    g := f
    f := e
    e := d + t1
    d := c
    c := b
    b := a
    a := t1 + t2

```

C. 主要代码

```

//使用宏定义规定循环右移逻辑运算
#define RR(w, n) ((w >> n) | (w << (32-(n))))

void sha256(char *data, unsigned char *out) {

    //信息预处理1-附加填充比特
    size_t len=strlen(data);
    int r = (int)(len * 8 % 512);
    int append = ((r < 448) ? (448 - r) : (448 + 512 - r)) / 8;
    size_t new_len = len + append + 8;
    unsigned char buf[new_len];
    memset(buf+len,0,append);
    if (len > 0) {
        memcpy(buf, data, len);
    }
    buf[len] = (unsigned char)0x80;

    //信息预处理2-附加长度值
    uint64_t bits_len = len * 8;
    for (int i = 0; i < 8; i++) {
        buf[len + append + i] = (bits_len >> ((7 - i) * 8)) & 0xff;
    }

    //计算消息摘要1-构造64个字
    uint32_t w[64];
    memset(w,0,64);
    size_t chunk_len = new_len / 64; //分成512-bit chunks
    for (int idx = 0; idx < chunk_len; idx++) {
        uint32_t val = 0;
        for (int i = 0; i < 64; i++) { //前16个字直接由消息的第i个块分解得到
            val = val | (*(buf + idx * 64 + i) << (8 * (3 - i)));
            if (i % 4 == 3) {
                w[i / 4] = val;
                val = 0;
            }
        }
        for (int i = 16; i < 64; i++) { //其余的字由迭代公式得到
            uint32_t s0 = RR(w[i - 15], 7) ^ RR(w[i - 15], 18) ^ (w[i - 15] >> 3);
            uint32_t s1 = RR(w[i - 2], 17) ^ RR(w[i - 2], 19) ^ (w[i - 2] >> 10);
            w[i] = w[i - 16] + s0 + w[i - 7] + s1;
        }
    }
}

```

```
//初始化Hash
uint32_t a = h0;
uint32_t b = h1;
uint32_t c = h2;
uint32_t d = h3;
uint32_t e = h4;
uint32_t f = h5;
uint32_t g = h6;
uint32_t h = h7;

//计算消息摘要2-进行64次循环迭代
for (int i = 0; i < 64; i++) {
    uint32_t s_1 = RR(e, 6) ^ RR(e, 11) ^ RR(e, 25);
    uint32_t ch = (e & f) ^ (~e & g);
    uint32_t temp1 = h + s_1 + ch + k[i] + w[i];
    uint32_t s_0 = RR(a, 2) ^ RR(a, 13) ^ RR(a, 22);
    uint32_t maj = (a & b) ^ (a & c) ^ (b & c);
    uint32_t temp2 = s_0 + maj;
    h = g;
    g = f;
    f = e;
    e = d + temp1;
    d = c;
    c = b;
    b = a;
    a = temp1 + temp2;
}
h0 += a;
h1 += b;
h2 += c;
h3 += d;
h4 += e;
h5 += f;
h6 += g;
h7 += h;

//小头存储
*((uint32_t *)out) = __builtin_bswap32(h0);
*((uint32_t *)out+4) = __builtin_bswap32(h1);
*((uint32_t *)out+8) = __builtin_bswap32(h2);
*((uint32_t *)out+12) = __builtin_bswap32(h3);
*((uint32_t *)out+16) = __builtin_bswap32(h4);
*((uint32_t *)out+20) = __builtin_bswap32(h5);
*((uint32_t *)out+24) = __builtin_bswap32(h6);
*((uint32_t *)out+28) = __builtin_bswap32(h7);

int main(){
    FILE *f=fopen("out.txt", "w+");

    unsigned char out[64];
    char data[MAX_IN];
    gets(data);

    sha256(data, out);
    for(int i=0; i<32; i++){
        printf("%02x", out[i]);
        fprintf(f, "%02x", out[i]);
    }

    return 0;
}
```


D. 运行结果

```
abcd  
88d4266fd4e6338d13b845fcf289579d209c897823b9217da3e161936f031589
```

abcd====>88d4266fd4e6338d13b845fcf289579d209c897823b9217da3e161936f031589

根据 <https://www.online-convert.com> 网站上在线验证，结果正确。

Conversion Completed

Your hash has been successfully generated.

```
hex: 88d4266fd4e6338d13b845fcf289579d209c897823b9217da3e161936f031589
```

对中文字符的 hash 实现需要使用 UTF-8 编码，在这里进入虚拟机 linux 环境进行运行，结果也正确。

```
byr@ubuntu:~$ ./test  
加油打工人！  
32f93ed25cfff8b677ee3d2487b1480080a6c5c0fcebfe700ec3a06edd846cc5
```

Conversion Completed

Your hash has been successfully generated.

```
hex: 32f93ed25cfff8b677ee3d2487b1480080a6c5c0fcebfe700ec3a06edd846cc5
```

III. SHA256 作用 1——数字签名

SHA256 算法由于其安全性——逆向转化困难、正向抗碰撞，成为比特币支撑算法，可解决数字货币“虚假货币”的问题。解决方法是数字签名，SHA256 对原文生成一个签名文件，再利用非对称加密技术进行传递。

A. 实验原理

验证数字签名的过程是：发送方用 sha256 算法对原文件生成一个签名文件，即 32 个字节的 hash 码。然后用加密算法（这里采用 rsa 算法进行实验）对此算法加密。接收方对加密的签名解密，得到一个 32 个字节的 hash 码。对原文件进行 sha256 签名计算，得到 32 字节的 hash。将这两个 hash 码比较，是否相等。若相等，即 RSA_SHA256 数字签名验证通过。

我们用实验模拟 RSA_SHA256 数字签名全过程。首先用 SHA256 生成签名，再利用 RSA 进行加密；用 RSA 解密，与一开始生成的 SHA256 签名进行比较，得出的结果是验证通过。

B. 实验代码

我们的实验主要由三个难点，SHA256 生成 Hash 签名、RSA 得到密文、解密得到明文。这里只展示 main 函数主体过程，RSA 的细节暂不多述。

```
int main(){
    //SHA256生成数字签名
    unsigned char out[64];
    char data[MAX_IN];

    if(f == NULL) {
        printf("Failed to open file \"text.txt\". Does it exist?\n");
        return EXIT_FAILURE;
    }
    printf("输入原文: ");
    gets(data);

    sha256(data, out);
    printf("\nSHA256生成数字签名开始-----\n数字签名: ");
    for(int i=0;i<32;i++){
        printf("%02x",out[i]);
        //if((i+1)%4==0) printf(" ");
        fprintf(f,"%02x",out[i]);
    }
    printf("\n");

    //RSA准备工作
    int p, q, n, phi, e, d, bytes, len;
    int *encoded, *decoded;
    char *buffer;

    srand(time(NULL));
    while(1) {
        p = randPrime(SINGLE_MAX);
        //printf("生成第一个随机素数, p = %d ... ", p);

        q = randPrime(SINGLE_MAX);
        //printf("生成第二个随机素数, q = %d ... ", q);

        n = p * q;
        //printf("计算p和q的乘积n, n = pq = %d ... ", n);
        if(n < 128) {
            //printf("Modulus is less than 128, cannot encode single bytes. Trying again ... ");
        }
        else break;
    }
    if(n >> 21) bytes = 3;
    else if(n >> 14) bytes = 2;
    else bytes = 1;
```

```
printf("RSA加密开始-----\n");

phi = (p - 1) * (q - 1);
//printf("计算欧拉函数的值phi, phi = %d ... ", phi);

e = randExponent(phi, EXPONENT_MAX);
//printf("选取一个随机素数e, e = %d...\n获得公钥 (%d, %d) ... ", e, e, n);

d = inverse(e, phi);
//printf("计算模反元素d, d = %d...\n获得密钥 (%d, %d) ... ", d, d, n);
printf("RSA加密密钥 (%d, %d) \n", d, n);

//printf("打开文件 \"text.txt\" 用于读取信息\n");
len = readFile(f, &buffer, bytes);
fclose(f);
//printf("文件 \"text.txt\" 读取成功, 读取到%d字节. 以%d字节的字节流编码 ... ", len, bytes);
getchar();
printf("加密得密文为: ");

encoded = encodeMessage(len, bytes, buffer, e, n);
//printf("\n编码成功完成 ... ");

getchar();
printf("\n解码得明文为: ");
decoded = decodeMessage(len/bytes, bytes, encoded, d, n);
```

C. 实验结果

```
输入原文: abcd
SHA256生成数字签名开始-----
数字签名: 88d4266fd4e6338d13b845fcf289579d209c897823b9217da3e161936f031589
RSA加密开始-----
RSA加密密钥 (256089, 20960119)
加密得密文为: 14684493 5595606 4821849 5714449 19229469 17239021 6948417 6758338 843114 19327696 2094774 102
23205 12339385 4535606 17295836 10327658 12409682 8571111 16331063 14357058 6929511 0
解码得明文为: 88d4266fd4e6338d13b845fcf289579d209c897823b9217da3e161936f03158
RSA_SHA256数字签名完成-----
```

可以看出, 解密后重新得到的明文与直接对原文进行数字签名的结果一致, 数字签名验证成功。

在生成数字签名过程中采用 SHA256 算法的优势是不可逆向, 这里可以通过观察其不通风原文对应的结果差异很大这一点形成认识, 并且到目前并未破解成功。

 D:\浙大学习工作\区块链\hw1\sha256.exe


```
ijkl
005c19658919186b85618c5870463eecd9b8c1a9d00208a5352891ba5bbe086
```

 D:\浙大学习工作\区块链\hw1\sha256.exe

```
mno
p
f1afc31479522d6cfff1ed068f93998f05a8cd3b22f5c37d7f307084f62d1d270
```

 D:\浙大学习工作\区块链\hw1\sha256.exe

```
abcd
88d4266fd4e6338d13b845fcf289579d209c897823b9217da3e161936f031589
```

 D:\浙大学习工作\区块链\hw1\sha256.exe

```
efgh
e5e088a0b66163a0a26a5e053d2a4496dc16ab6e0e3dd1adf2d16aa84a078c9d
-----
```

IV.SHA256 作用 2——工作量证明

SHA256 算法还可以应用于解决数字货币多重支付的问题，帮助建立分布式的可信记账机制。解决记账权的方法是工作量证明，利用高强度的哈希计算进行算力竞争达到共识。SHA256 在这里的作用就是进行算力竞争“挖矿”找随机数，满足前导零要求的拥有记账权，其余节点通过收到广播并验证的方法达成共识。也就是说，区块的创建分为三步——竞争胜出的节点创建区块并广播、其他独立节点校验新区块、区块链的组装和选择，而 SHA256 在这个过程中作用就是作为竞争的手段之一。

A. 实验原理

区块头的结构中有 nBits，表示为挖矿设置的难度，即 Hash 值中须有前导零的位数；还有 nNonce，表示在挖矿中使 Hash 值达到 nBits 字段所要求前导零位数的试凑值。

我们设计 nBits=4，nNonce 为 ppt 示例中的 I am Satoshi Nakamoto 和依次递增的序号数，用实验模拟 SHA256 算力竞争的过程，观察拥有四个前导零所需要的时间和难度。

B. 实验代码

我们的实验难点在于 nNonce 包括其序号数的构造。此外，我们设置宏定义 NBITS 和 TURN 规定前导零位数和循环次数即参与算力竞争的次数。

这里只展示 main 函数主体过程，包括构造 nNonce、循环和计时。SHA256 的细节上述已列出这里不加赘述。

```
int cnt=0,nBits=NBITS,turn=TURN;
unsigned char out[64];
char data[MAX_IN];
char str[100];

int i=0,j;
clock_t begin = clock();
while(i!=TURN){
    memset(data,'\0',MAX_IN);
    memset(out,'\0',64);
    strcpy(data,"I am Satoshi Nakamoto");
    itoa(i,str,10);
    strcat(data,str);
    printf("%s====>",data);
    sha256(data, out);
    for(j=0;j<32;j++){
        printf("%02x",out[j]);
    }
    printf("\n");
    if(nBits%2==0){
        for(j=0;j<nBits/2;j++){
            if(out[j]!=0){
                break;
            }
        }
        if(j==nBits/2){
            printf("-----Get!-----\n");
            cnt++;
        }
    }
    else{
        for(j=0;j<nBits/2;j++){
            if(out[j]!=0){
                break;
            }
        }
        if(j==nBits/2&&out[j]/16==0){
            printf("-----Get!-----\n");
            cnt++;
        }
    }
    i++;
}
```



```

clock_t end = clock();
double elapsedTime = (double)(end - begin) / CLOCKS_PER_SEC;
printf("The 0 before is %d.\n", nBits);
printf("The number of GET is %d in %d turns.\n", cnt, turn);
if(cnt!=0)printf("The average time for one GET is %.2f s.\n", elapsedTime/cnt);
if(cnt!=0)printf("Getting 2016 valid blocks needs %.2f s.\n", 2016*(elapsedTime/cnt));
return 0;

```

C. 实验结果

1) nBits=2, TURN=1000:

```

I am Satoshi Nakamoto996=====>b469fbc5315f5da68427ae2b4ce433d844cb00e9c9769f5339ec059721517f72
I am Satoshi Nakamoto997=====>221a540d93d5761a42f25fef9268c622a2a211933a2d7b3b75264ed056d8de27
I am Satoshi Nakamoto998=====>99dd5e04ca18a8c715fc19b781110437bcb1b6a7aa19a515f4872e06bd14f4ad
I am Satoshi Nakamoto999=====>d71874f7430e9e561525d26133ebc65b4e3ed4c3f17a05d6ccd0db11e075fd1f
The 0 before is 2.
The number of GET is 5 in 1000 turns.
The average time for one GET is 0.82 s.
Getting 2016 valid blocks needs 1644.65 s.

```

在 1000 次尝试中出现了 5 个前置零为 2 的 hash 值。

2) nBits=4, TURN=1000:

```

I am Satoshi Nakamoto995=====>ee01d3f4f15bc466ac1b2f1ac005e0bb5fblab6cc42a3d75401f4f978749d7e1
I am Satoshi Nakamoto996=====>b469fbc5315f5da68427ae2b4ce433d844cb00e9c9769f5339ec059721517f72
I am Satoshi Nakamoto997=====>221a540d93d5761a42f25fef9268c622a2a211933a2d7b3b75264ed056d8de27
I am Satoshi Nakamoto998=====>99dd5e04ca18a8c715fc19b781110437bcb1b6a7aa19a515f4872e06bd14f4ad
I am Satoshi Nakamoto999=====>d71874f7430e9e561525d26133ebc65b4e3ed4c3f17a05d6ccd0db11e075fd1f
The 0 before is 4.
The number of GET is 0 in 1000 turns.

```

在 1000 次尝试中并未找到前置零为 4 的 hash 值。我们需要继续加大循环次数 TURN。

3) nBits=4, TURN=1000000:

```

I am Satoshi Nakamoto999989=====>088333443369796bd8f809c5807b864d0674804f2aa437658b710fd1fe9005db
I am Satoshi Nakamoto999990=====>106913618364dbdd03f74626ecbba6070cd656b7027984777f5cccd5a2f17d67
I am Satoshi Nakamoto999991=====>1258e710dbc6c78a0206f1256eb398764508fa9bebbcb75364f6ef41295628c0
I am Satoshi Nakamoto999992=====>2f062edeaeef7a78e8cbeb20665aa7d48168fa325af41a7aea10f5a3fa7952e4e
I am Satoshi Nakamoto999993=====>214c187ceee96734b736afa8d75ddae3dcc8d309aa878dd452d2c831a597f510
I am Satoshi Nakamoto999994=====>8228fefd95adb9580bae8e2b835503ff8ad07fa4f9b8591fe314bflfca5c38d3
I am Satoshi Nakamoto999995=====>52d85b51dbce3b9b4ff9e8dedbc2a56f2b3adc9d06affbdfb7b27bfd6d612c2
I am Satoshi Nakamoto999996=====>71281a0e82cd805b18933921bfff84cd6db1aefc3d2caf68c0e50b3e8e4e71c7
I am Satoshi Nakamoto999997=====>8dd63e38970ccfbb5204c97ba12c17283bf5dbd6a24c662c36ace71bd944a5d4
I am Satoshi Nakamoto999998=====>ea394396c85b11cd160eddfaa56e7416bdcee73dd95692974d60d607edd1649b
I am Satoshi Nakamoto999999=====>7bb2340a56eada49d16e44ee5bc51a0bec5bf5bed9c300b2c747516667eb90e0
The 0 before is 4.
The number of GET is 20 in 1000000 turns.
The average time for one GET is 203.85 s.
Getting 2016 valid blocks needs 410954.64 s.

```

可以看出，在 1000000 次挖矿中只有 20 次出现了四个前导零，平均每挖到一个需要三分多钟。因此，挖矿值的设定和调整很有必要，难度是根据实际时间与期望时间的比值进行相应调整的（或变难或变易），由于实际区块节点内容不同，设定为平均挖到一个区块 10 分钟是合理的。

也可以看出，工作量证明需要很大的算力。

V. SHA256 作用 3——Merkle 树

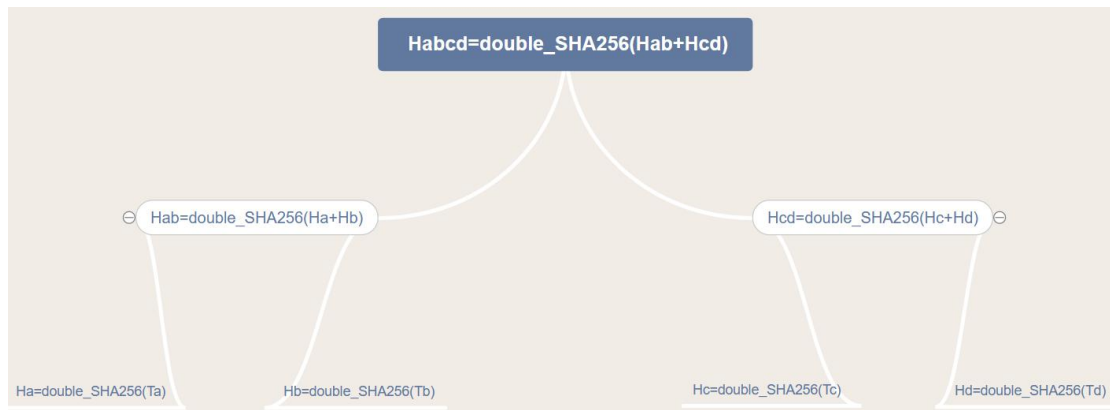
区块的微观结构里，区块标识符由区块头的哈希值及其区块高度组成，其主标识符是唯一的加密哈希值，该哈希值就是有 SHA256 对区块二次哈希计算得到的数字指纹，利用了该算法不可逆向的优点。区块哈希值实际存于 Merkle 树中，Merkle 树的根存放于区块头中。Merkle 树用于对交易数据列表进行快速寻址，归纳一个区块中的所有交易，同时生成整个交易集合的数字指纹，且提供了一种校验区块是否存在某交易的高效途径——可以快速比较大量数据（只需比较根的哈希值即可）、快速定位修改（从根向下比较）、快速验证其中数据（根据已给路径计算哈希值并比对）。

也就是说，SHA256 算法在生成区块，更确切地说生成其中的 Merkle 树中的作用就是两次哈希计算得到各节点值。

A. 实验原理

Merkle 树是哈希二叉树，叶节点是数据块的哈希值，非叶节点的哈希值是根据下面子节点的值哈希计算得到的。生成一棵完整的 Merkle 树需要递归地对一对节点进行哈希，并将新生成的哈希节点插入到 Merkle 树中，直到只剩一个哈希节点，该节点就是 Merkle 树的根。

我们设计实验通过二次 SHA256 构造 Merkle 树。Merkle 树是自底向上构建的。从 a~d 四个构成 Merkle 树树叶的交易开始，先将其数据哈希化，然后将哈希值存储至相应的叶子节点 $H_a \sim H_d$ 。通过串联相邻叶子节点的哈希值然后对其哈希，这对叶子节点随后被归纳为父节点。如为了创建父节点 H_{ab} ，子节点 a 和子节点 b 的两个 32 字节的哈希值将被串联成 64 字节的字符串。随后将字符串进行两次哈希来产生父节点的哈希值，继续类似的操作直到只剩下顶部的一个节点，即 Merkle 根。产生的 32 字节哈希值存储在区块头，同时归纳了四个交易的所有数据。



B. 实验代码

这里只展示两次 SHA256 哈希的过程。SHA256 的细节上述已列出这里不加赘述。Merkle 树的实现代码较多，这里不作展示。

```
FILE *f=fopen("out.txt","w+");
unsigned char out[64];
char data[MAX_IN];
gets(data);

//1st
sha256(data, out);
printf("1st:\n");
for(int i=0;i<32;i++){
    printf("%02x",out[i]);
    fprintf(f,"%02x",out[i]);
}

memset(out,0,64);
fseek(f,0,SEEK_SET);
fgets(data,65,f);

getchar();
//2nd
sha256(data, out);
printf("\n2nd:\n");
for(int i=0;i<32;i++){
    printf("%02x",out[i]);
}
```

C. 实验结果

首先验证两次 SHA256 的结果是否正确。

```
abcd
1st:
88d4266fd4e6338d13b845fcf289579d209c897823b9217da3e161936f031589
2nd:
2889adaad581fae88a6a1ae7f3335dde60bfeaf2963a1a3dc1c1c5aa2deaec85
```

与在线验证网站上的结果相同。

Conversion Completed

Your hash has been successfully generated.

hex: 2889adaad581fae88a6a1ae7f3335dde60bfeaf2963a1a3dc1c1c5aa2deaec85

其次，对交易 a、b、c、d（内容即为 a、b、c、d）进行构建 Merkle 树，结果如下图：

```
[0]: a3712fb404bdd9a9155501160b342ae8e4e92b108e5bba4078a14a7ab619b295
[1]: 10b20385f7e8b7dba224562d84da6775012ba1f2eaa7a2cefd279ad685be7a9d
    7012c94a527f15cf2eec2b35dab53acbe9400919a7c8751f7b09ef3f0d8ec6f5
[2]: da3811154d59c4267077ddd8bb768fa9b06399c486e1fc00485116b57c9872f5
    dba1de6de88c058e5e0922171e0bd97e79e20e9fc6c1d2737a91146765527305
    c8623e984cf78e0f3a959827b540a5cdbe4da246f5302ec8de4270300cf724c1
    d5fb46b830284b74f779bd39b16f254497a2ad46a098c14470a317f0a9d01ac8
```

若交易数为奇数，通过复制最后一个交易也能成功，下面展示对交易 a、b、c 的结果：

```
[0]: 2d0c972ba719aea3b82f19b0e3b536bebd7a806fbf96c507754150516a126569
[1]: 10b20385f7e8b7dba224562d84da6775012ba1f2eaa7a2cefd279ad685be7a9d
    a123b75068f2a7472451a6b6aedaf997f0d70d71c5395735e96a0dfde678087d
[2]: da3811154d59c4267077ddd8bb768fa9b06399c486e1fc00485116b57c9872f5
    dba1de6de88c058e5e0922171e0bd97e79e20e9fc6c1d2737a91146765527305
    c8623e984cf78e0f3a959827b540a5cdbe4da246f5302ec8de4270300cf724c1
    c8623e984cf78e0f3a959827b540a5cdbe4da246f5302ec8de4270300cf724c1
```

可以看出，构造 Merkle 树成功。

为了证明区块中存在某个特定的交易，一个节点只需要计算 $\log N$ 个 32 字节的哈希值，形成一条从特定交易到树根的认证路径或者 Merkle 路径即可。这里只强调 SHA256 对生成 Merkle 树的作用，对于 Merkle 的作用不做多的证明。

VI. SHA256 作用 4——比特币地址

一个比特币钱包包含一系列的密钥对，每个密钥对都有一个公钥和私钥。私钥是由一个随机产生的数字串经过哈希的十六进制字符串，通过椭圆加密算法可以用私钥产生公钥（单向），然后通过公钥产生比特币地址。SHA256 起作用的地方就是在从公钥到比特币地址的转化过程中。最后，为了提高可读性、避免歧义并有效地防止在地址转录和输入中产生错误，我们看到的比特币地址是再经过 Base58Check 编码的。

也就是说，SHA256 算法生成比特币地址的作用是与 RIPEMD160 一起将公钥转化为地址。

A. 实验原理

比特币地址是由公钥经过一系列单向的哈希算法得到，使用的算法是 SHA256 和 RIPEMD160，即以公钥 K 为输入，计算其 SHA256 哈希，然后再计算 RIPEMD160 哈希，所得到的 160 位（20 字节）的数字就是比特币地址—— $\text{Addr} = \text{RIPEMD160}(\text{SHA256}(K))$ 。

我们设计实验通过 SHA256 和 RIPEMD160 两次哈希得到地址。

B. 实验代码

这里只列出 main 函数的主体过程，由于侧重于对 SHA256 作用的研究，RIPEMD160 的代码细节参考网络上的 <https://blog.csdn.net/ak47000gb/article/details/80387112>，在此不多加展示。


```
FILE *f=fopen("out.txt","w+");
unsigned char out[64];
char data[MAX_IN];
gets(data);

//SHA256
sha256(data, out);
printf("SHA256:\n");
for(int i=0;i<32;i++){
    printf("%02x",out[i]);
    fprintf(f,"%02x",out[i]);
}

memset(out,0,64);
fseek(f,0,SEEK_SET);
fgets(data,65,f);

getchar();
//RIPEMD160
ripemd160(data, out);
printf("RIPEMD160:\n");
for(int i=0;i<20;i++){
    printf("%02x",out[i]);
}
```

C. 实验结果

```
abcd
SHA256:
88d4266fd4e6338d13b845fcf289579d209c897823b9217da3e161936f031589
RIPEMD160:
87df20236d6cd47470b6487aad32b5bb4865f845
```

其结果与在线验证网站上的结果一致：

Conversion Completed

Your hash has been successfully generated.

hex: 87df20236d6cd47470b6487aad32b5bb4865f845

可以看出，SHA256 也在比特币地址生成过程中起重要作用。

VII. 总结与心得

本次实验完成了 SHA256 的算法实现，分为四步：

1. 常量初始化；
2. 信息预处理；
3. 设置逻辑运算；
4. 循环计算摘要。

这里的主要难点是对整个算法流程的理解，花费较长时间的是再循环计算部分的逻辑运算。

在分析 SHA256 在区块链中的作用方面，列举了四个作用：

1. 在解决虚拟货币问题的数字签名中与 RSA 算法一起生成签名并传递；
2. 在解决多重支付问题的分布式可信记账机制中参与工作量证明，是算力竞争的方法；
3. 在解决多重支付问题的账本不可篡改机制中参与区块结构 Merkle 树的节点构建；
4. 在私钥到公钥到比特币地址的产生中与 RIPEMD160 算法一起完成公钥到比特币地址的转换。

这里的主要难点是对于区块链工作方式的理解和该算法在复杂的区块链工作中的作用分析，以及思考实验如何设计才能体现该作用。在此过程中更深入地了解了区块链的工作原理，对 SHA256 的意义也有了进一步的认识，在代码编写、实验设计的能力也得到了锻炼。