HASH 函数实现之 SHA-1

一、说明

1.填充

需要将需要计算摘要的消息进行填充，填充成 512 的倍数，填充方法是在消息末位补一个字节 0x80，并在最后附加上 64 位的长度信息使得填充后长度是 512 的倍数，有可能填充后消息比之前多了一个块。

2.常数

①初始化向量 Hi 初始化为下述值：

H_0 = 67452301

H_1 = EFCDAB89

H_2 = 98BADCFE

H_3 = 10325476

H_4 = C3D2E1F0

②在进行迭代时会用到的 $K_t(0 \leqslant t \leqslant 79)$ 为下述值：

K_t = 5A827999 (0 <= t <= 19)

K_t = 6ED9EBA1 (20 <= t <= 39)

K_t = 8F1BBCDC (40 <= t <= 59)

K_t = CA62C1D6 (60 <= t <= 79)

3.数据结构、函数

```
uint32_t W[80];//每次处理 16 个 32 位的字要用到的结构，扩展为 80 个字
uint32_t A, B, C, D, E; //迭代时对应的 ABCDE
//四个非线性函数
inline uint32_t f1(uint32_t B, uint32_t C, uint32_t D);
inline uint32_t f2(uint32_t B, uint32_t C, uint32_t D);
inline uint32_t f3(uint32_t B, uint32_t C, uint32_t D);
inline uint32_t f4(uint32_t B, uint32_t C, uint32_t D);
inline uint32_t cirleft(uint32_t word, int bit);//word循环左移bit位
long long msgsize(char*plainaddr);//获得消息长度

//0-79 的轮函数，循环时调用
void subround(uint32_t & A, uint32_t & B, uint32_t & C, uint32_t & D, uint32_t & E,
uint32_t & W, uint32_t K, int mode);
```

二、具体实现

1.每次读取 512 位消息也就是 16 个字（如果有的话），扩展成 80 个字。实现时先把消息读到 W 的前 16 个元素，后面的用来存放扩展的。扩展时按照原始的算法，没有按照 NIST 的进行左移一位。在每一个块的处理中，初始化 A = H0，B = H1，C = H2，D = H3， E = H4。每个块处理四种迭代调用四次 subround。一直做直到做完倒数第二次（如果只填充了一个块）或者倒数第二次（在两个块里填了内容）。处理块的时候需要做一些处理，因为读取的顺序和需要的顺序不太

一样，加入文本中是“abcd”，希望读取到 W[0]中是 0x61626364，但是直接读的是 0x64636261，所以对于块 W[i](0≤i<16)，W[i] = (W[i] >> 24) + (W[i] >> 8 & 0xff00) + (W[i] << 8 & 0xff0000) + (W[i] << 24)进行顺序处理。同时输入迭代函数的 W[i]需要循环左移 1 位，当 i>15 时，要通过之前的 W[i]来生成，即 W[i] = ((W[i - 3] ^ W[i - 8] ^ W[i - 14] ^ W[i - 16]) << 1)。

2.在 80 次迭代中，0-19 次使用 f1，20-39 次使用 f2，40-59 次使用 f3，60-79 次使用 f4。0=80 轮每一轮的轮函数如下（以 0-19 轮为例），其中 cirleft(A,5)是将 A 循环左移 5 位：

```
void subround(uint32_t & A, uint32_t & B, uint32_t & C, uint32_t & D, uint32_t & E,
uint32_t W, uint32_t K, int mode)
{
    uint32_t t;//临时备份
    switch (mode)
    {
    case 1:
        t = A;
        A = f1(B, C, D) + cirleft(A, 5) + W + K;
        E = D;
        D = C;
        C = cirleft(B, 30);
        B = t;
        break;
```

f1 的实现如下所示，完成(B∧C) ∨(¬B∧D):

```
inline uint32_t f1(uint32_t B, uint32_t C, uint32_t D)
{
    return(B&C) | ((~B)&D);
}
```

3.迭代完 80 轮以后，进行加法运算：

```
        H0 = H0 + A;
        H1 = H1 + B;
        H2 = H2 + C;
        H3 = H3 + D;
        H4 = H4 + E;
```

4.最后一个消息块，要填充，如果只需要填充完这个块，还能放下长度，就此为止：

```
bytes = fread(W, sizeof(char), 64, fp);//读出一个消息块 512bits,读到 W 里面
unsigned char*p = (unsigned char*)&W[0];
    p = p + bytes;
    *p = 0x80;
    if (flag == 0)
        memcpy(&W[7], &msglen, 8);//复制长度
```

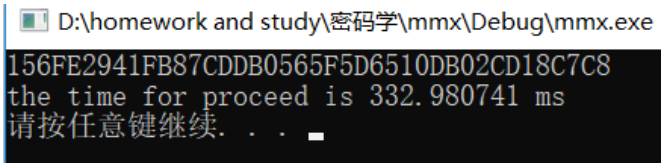
5.如果最后一个块有点大，放不下数据长度，就填到下一块，再多做一次迭代：

```
memset(W, 0, 64);
memcpy(&W[7], &msglen, 8);//复制长度
```

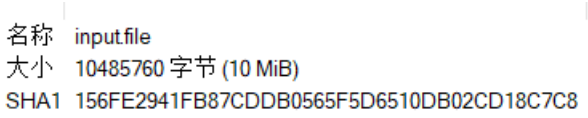
6.加速时，代码全部手动内联，发现编译器好像忽略了 inline，手动内联的耗时是自动内联的 1/2.

三、运行结果

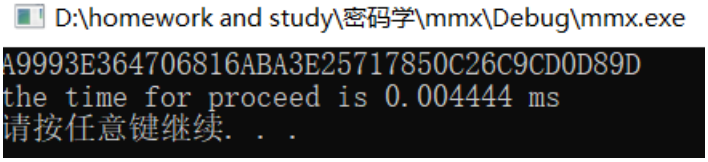
1.拿之前用于测试的 10MB.txt 作为输入 input.file 进行摘要运算，得到输出文件 output.file 和运行时间(不包括 I/O 操作),结果显示运行时间没有 RC4(55ms)那么快，但是比 SM4 的 ECB 模式（400ms）要快，比 AES 的 ECB 模式(100ms)慢。

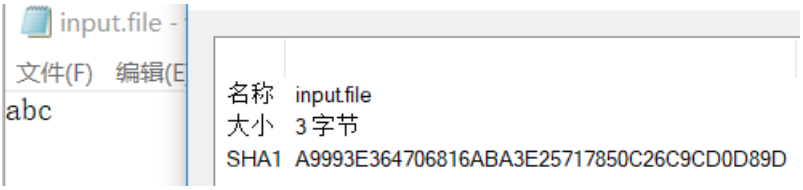


这个值与 7zip 自带的 SHA-1 计算出来是一样的：



2.对于短消息，比如内容只有“abc”，填充以后也只有一个块，能计算正确：





3.对于长一点的消息正好在 448 位的“abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq”，也能正确计算得到结果：