SM3的C/C++实现-20307130135李钧

根据GMT0004-2012(SM3密码杂凑算法)官方说明文档，本人使用C/C++语言进行了SM3密码杂凑算法的计算步骤复现，可以对输入的字符串信息进行数字签名和验证等。首先展示文件结构如下所示：

```
E:\0Tree_Down\crypto\0last2\SM3>tree /F
卷 新加卷 的文件夹 PATH 列表
卷序列号为 2A31-0171
E:.
│  CMakelists.txt
│  SM3.cpp
│  SM3.exe
│  SM3_h.h
│  Smain.cpp
│
├─.vscode
│      settings.json
│
└─build
    │  CMakeCache.txt
    │  cmake_install.cmake
```

相关常数与函数

根据说明文档，在.h文件中定义常数T和初始向量IV，并声明基本函数函数FF等，如下图所示

```
// 常数、常量
const unsigned int IV[8] = {0x7380166f, 0x4914b2b9, 0x172442d7, 0xda8a0600,
                            0xa96f30bc, 0x163138aa, 0xe38dee4d, 0xb0fb0e4e};
const int T_0_15 = 0x79cc4519;
const int T_16_63 = 0x7a879d8a;
```

```
// 基本函数
unsigned int FF(unsigned int X, unsigned int Y, unsigned int Z, int j);
unsigned int GG(unsigned int X, unsigned int Y, unsigned int Z, int j);
unsigned int P0(unsigned int x);
unsigned int P1(unsigned int x);
```

除此之外，为了方便在实现函数功能时简化部分移位操作、统一分组长度和消息长度的二进制数长度值，在头文件中还对如下常量进行声明

```
#define ROTATE_LEFT(x, n) (((x) << (n)) | ((x) >> (32 - (n))))
const int GROUP_SIZE = 512 / 8; // 分组长度，以字节为单位
const int MESSAGE_LENGTH_SIZE = 64 / 8; // 表示消息长度的二进制数长度
```

下面是对上述基本函数的具体定义，这些定义放在SM3.cpp源文件当中

基本函数 C++ 复制代码

```cpp
unsigned int FF(unsigned int X, unsigned int Y, unsigned int Z, int j){
    if(j <= 15) return (X ^ Y ^ Z);
    else return ((X&Y) | (X&Z) | (Y&Z));
}
unsigned int GG(unsigned int X, unsigned int Y, unsigned int Z, int j){
    if(j <= 15) return (X ^ Y ^ Z);
    else return (X&Y) | ((~X)&Z);
}
unsigned int P0(unsigned int x) {
    return x ^ ROTATE_LEFT(x, 9) ^ ROTATE_LEFT(x, 17);
}
unsigned int P1(unsigned int x) {
    return x ^ ROTATE_LEFT(x, 15) ^ ROTATE_LEFT(x, 23);
}
```

算法描述

概述

对长度l(l < 2^64)比特的消息m，SM3杂凑算法经过填充、迭代压缩和输出选裁，生成杂凑值，杂凑值输出长度为256比特。

填充

假设消息m的长度为l比特，则首先将比特"1"添加到消息的末尾，再添加k个"0"，k是满足以下条件的最小非负整数：l+1+k=448 (mod 512)。然后再添加一个64位比特串，该比特串是长度l的二进制表示。填充后的消息m'的比特长度为512的倍数。

EXAMPLE For the message 011000010110001001100011, with length l =24, the bit string after padding is:

$$01100001011000100110001 1 \overbrace{00\ldots00}^{423\text{ bits}} \overbrace{\underbrace{00\ldots011000}_{\substack{\text{binary}\\ \text{expression of } l}}}^{64\text{ bits}}.$$

填充部分的代码如下所示

```
// message_len是指message有多少个字节
// 在填充0之前，消息可以分成多少组(group)
int test = (message_len + MESSAGE_LENGTH_SIZE + 1) % GROUP_SIZE;
unsigned int message_group_num;
if(test == 0) message_group_num = (message_len + MESSAGE_LENGTH_SIZE + 1) / GROUP_SIZE;
else message_group_num = (message_len + MESSAGE_LENGTH_SIZE + 1) / GROUP_SIZE + 1;
```

```
// pad添加字节之后，1的位置
unsigned int message_pad_len = message_group_num * GROUP_SIZE - MESSAGE_LENGTH_SIZE;
// padding之后的初始化，将比特"1"添加到消息末尾
unsigned int sizeof_pad = message_group_num * GROUP_SIZE;
unsigned char message_pad[sizeof_pad];
memset(message_pad, 0, sizeof(message_pad)); // 以字节为单位
memcpy(message_pad, message, message_len);
message_pad[message_len] = 0x80;
```

```
// 加上l的二进制表示
unsigned long long bit_msg_l = message_len * 8;
for (unsigned int i = 0; i < MESSAGE_LENGTH_SIZE; ++i) {
    message_pad[message_pad_len + i] = ((unsigned char*)&bit_msg_l)[MESSAGE_LENGTH_SIZE - i - 1];
}
```

迭代压缩

在开始迭代压缩之前，需要对必要的状态寄存器、消息字等进行声明、初始化，如下所示

```
unsigned int state[8];
memcpy(state, IV, sizeof(IV));
unsigned int W[68], W_[64];
unsigned int A, B, C, D, E, F, G, H;
for(unsigned int i = 0; i < message_group_num; i++){
    memset(W, 0, sizeof(W)); // 初始化每个块内容，对应每个512bit
    memset(W_, 0, sizeof(W_));
```

在循环内部，即针对每个message_group，扩展生成132个消息字W0~W67,W'0~W'63

```
// 一个W[]四个字节
for(int j = 0; j < 16; j++){
    W[j] =
        (message_pad[i*64 + j*4]   <<24) | (message_pad[i*64 + j*4+1]<<16) |
        (message_pad[i*64 + j*4+2]<<8)   | message_pad[i*64 + j*4+3];
}
// a)将消息分组划分成16个字W0...W15
// b)for j = 16 to 67
for (int j = 16; j < 68; j++) {
    W[j] = P1(W[j - 16] ^ W[j - 9] ^ ROTATE_LEFT(W[j - 3], 15)) ^ ROTATE_LEFT(W[j - 13], 7) ^ W[j - 6];
}
for(int j = 0; j < 64; j++){
    W_[j] = W[j] ^ W[j+4];
}
```

根据文档，实现CF压缩函数如下所示，在该循环之后更新state[]的值迭代到下一轮group循环

```
cout << "\nA_______B_______C_______D_______E_______F_______G_______H" << endl;
for (unsigned int j = 0; j < GROUP_SIZE; ++j) {
    int tmpp = 0;
    if(j <= 15) tmpp = T_0_15;
    else tmpp = T_16_63;
    unsigned int SS1 = ROTATE_LEFT(ROTATE_LEFT(A, 12) + E + ROTATE_LEFT(tmpp, j%32), 7);
    unsigned int SS2 = SS1 ^ ROTATE_LEFT(A, 12);
    unsigned int TT1, TT2;
    TT1 = FF(A, B, C, j) + D + SS2 + W_[j];
    TT2 = GG(E, F, G, j) + H + SS1 + W[j];

    D = C;
    C = ROTATE_LEFT(B, 9);
    B = A;
    A = TT1;
    H = G;
    G = ROTATE_LEFT(F, 19);
    F = E;
    E = P0(TT2);

    //test A~H···
}
// 迭代到下一轮
state[0] ^= A;
state[1] ^= B;
state[2] ^= C;
state[3] ^= D;
state[4] ^= E;
state[5] ^= F;
state[6] ^= G;
state[7] ^= H;
```

最终更新出哈希值

```
    for (unsigned int i = 0; i < 8; i++) { //8字节
        hash[i] = state[i];
    }

    return;
```

样例测试

在官方文档末尾的测试样例是两个字符串"abc"和"abcd"*16

"abcd"*16字符串的最终结果如下所示

The hash value is:

```
debe9ff9 2275b8a1 38604889 c18e5a4d 6fdb70e5 387e5765 293dcba3 9c0c5732
```

```
int main(){
    char *message;
    string s = "abcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcd";
    // string s = "abc";
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
final hash value is:
debe9ff9 2275b8a1 38604889 c18e5a4d 6fdb70e5 387e5765 293dcba3 9c0c5732
PS E:\0Tree_Down\crypto\0last2\SM3>
```

"abc"字符串的部分中间结果与文档相符，最终哈希值也与文档相符

The hash value is:

```
66c7f0f4 62eeedd9 d1f2d46b dc10e4e2 4167c487 5cf2f7a2 297da02b 8f4ba8e0
```

The input message is "abc", and its ASCII-coded version is:

```
616263
```

The message after padding process is:

```
61626380 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000018
```

The message after message expansion:

$W_0 W_1 \dots W_{67}$:

```
61626380 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000018
9092e200 00000000 000c0606 719c70ed 00000000 8001801f 939f7da9 00000000
2c6fa1f9 adaaef14 00000000 0001801e 9a965f89 49710048 23ce86a1 b2d12f1b
e1dae338 f8061807 055d68be 86cfd481 1f447d83 d9023dbf 185898e0 e0061807
050df55c cde0104c a5b9c955 a7df0184 6e46cd08 e3babdf8 70caa422 0353af50
a92dbca1 5f33cfd2 e16f6e89 f70fe941 ca5462dc 85a90152 76af6296 c922bdb2
68378cf5 97585344 09008723 86faee74 2ab908b0 4a64bc50 864e6e08 f07e6590
325c8f78 accb8011 e11db9dd b99c0545
```

```
6        string s = "abc";
```

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

```
PS E:\0Tree_Down\crypto\0last2\SM3> ."E:/0Tree_Down/crypto/0last2/SM3/SM3.exe"


the W result is:
61626380 0 0 0 0 0 0 0
0 0 0 0 0 0 0 18
9092e200 0 c0606 719c70ed 0 8001801f 939f7da9 0
2c6fa1f9 adaaef14 0 1801e 9a965f89 49710048 23ce86a1 b2d12f1b
e1dae338 f8061807 55d68be 86cfd481 1f447d83 d9023dbf 185898e0 e0061807
50df55c cde0104c a5b9c955 a7df0184 6e46cd08 e3babdf8 70caa422 353af50
a92dbca1 5f33cfd2 e16f6e89 f70fe941 ca5462dc 85a90152 76af6296 c922bdb2
68378cf5 97585344 9008723 86faee74 2ab908b0 4a64bc50 864e6e08 f07e6590
325c8f78 accb8011 e11db9dd b99c0545
```

```
6        string s = "abc";
```

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

```
4a00ed2f 13826486 c5eba62b e100077a 741f675 cd09a51c 61fe1cab 5e049a4a
f4412e82 4a00ed2f 4c90c27 c5eba62b 7429807c 741f675 28e6684d 61fe1cab
549db4b7 f4412e82 1da5e94 4c90c27 f6bc15ed 7429807c b3a83a0f 28e6684d
22a79585 549db4b7 825d05e8 1da5e94 9d4db19a f6bc15ed 3e3a14c b3a83a0f
30245b78 22a79585 3b696ea9 825d05e8 f6804c82 9d4db19a af6fb5e0 3e3a14c
6598314f 30245b78 4f2b0a45 3b696ea9 f522adb2 f6804c82 8cd4ea6d af6fb5e0
c3d629a9 6598314f 48b6f060 4f2b0a45 14fb0764 f522adb2 6417b402 8cd4ea6d
ddb0a26a c3d629a9 30629ecb 48b6f060 589f7d5c 14fb0764 6d97a915 6417b402
71034d71 ddb0a26a ac535387 30629ecb 14d5c7f6 589f7d5c 3b20a7d8 6d97a915
5e636b4b 71034d71 6144d5bb ac535387 9ccd95e 14d5c7f6 eae2c4fb 3b20a7d8
2bfa5f60 5e636b4b 69ae2e2 6144d5bb 4ac3cf08 9ccd95e 3fb0a6ae eae2c4fb
1547e69b 2bfa5f60 c6d696bc 69ae2e2 e808f43b 4ac3cf08 caf04e66 3fb0a6ae

final hash value is:
66c7f0f4 62eeedd9 d1f2d46b dc10e4e2 4167c487 5cf2f7a2 297da02b 8f4ba8e0
```

重构实现三个接口

重新阅读Elearning上的文件之后，参考官方SM3函数的实现，实现了SM3_Init和SM3_Update以及SM3_Final，可以对输入流input的内容计算哈希值，下图是对SM3文档中一样例的测试，可以看出成功初始化、更新、计算输出。具体代码查看压缩包。

The hash value is:

```
debe9ff9 2275b8a1 38604889 c18e5a4d 6fdb70e5 387e5765 293dcba3 9c0c5732
```

```c
    SM3_CTX ctx1;//, ctx2;
    unsigned char md1[SM3_DIGEST_LENGTH];//, md2[SM3_DIGEST_LENGTH];

    int t1 = SM3_Init(&ctx1);
    printf("\nSM3_Init is: %d\n", t1);

    t1 = SM3_Update(&ctx1, input2, sizeof(input2));
    printf("\nSM3_Update is: %d\n", t1);

    t1 = SM3_Final(md1, &ctx1);
    printf("\nSM3_Final is: %d\n", t1);

    printf("\nhash value is:\n");
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
SM3_Init is: 1

SM3_Update is: 1

SM3_Final is: 1

hash value is:
debe9ff9 2275b8a1 38604889 c18e5a4d 6fdb70e5 387e5765 293dcba3 9c0c5732
```

除了测试固定上述固定字符串，我还可以对我们输入的内容input计算哈希值，如下所示我们输入流的内容是"abc"，最终计算出的哈希值与预期一致。此法可推广到计算文件的哈希值上。

```c
    t1 = SM3_Update(&ctx1, message, sizeof(message));//this is important
    printf("\nSM3_Update is: %d\n", t1);

    t1 = SM3_Final(md1, &ctx1);
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS E:\0Tree_Down\crypto\0last2\SM3> ."E:/0Tree_Down/crypto/0last2/SM3/SM3.exe"
input your message here: abc

SM3_Init is: 1

SM3_Update is: 1

SM3_Final is: 1

hash value is:
66f4f01e eb86f5c8 5b702d0a 93a94bbc b2224e77 0c7c9ef1 1f6b0c70 796450ab
```