《密码学》实验四报告

MD5哈希函数实现

Part 1. 对原始消息进行填充

对于输入的任意消息m，设其长度为$\ell = ||m||$，则对消息的填充如下：

- 填充`0x1000000....`，直至消息的长度$\ell \equiv 448 \pmod{512}$。即使消息的原始长度满足这个要求，我们也需要对消息进行填充；
- 最后在消息的末尾添加上一个以小端序组织的消息长度的比特串（64位）。例如，如果$\ell = 12$，则最后会添加上：

```
[message] [1000000000.....] [1100 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000]
```

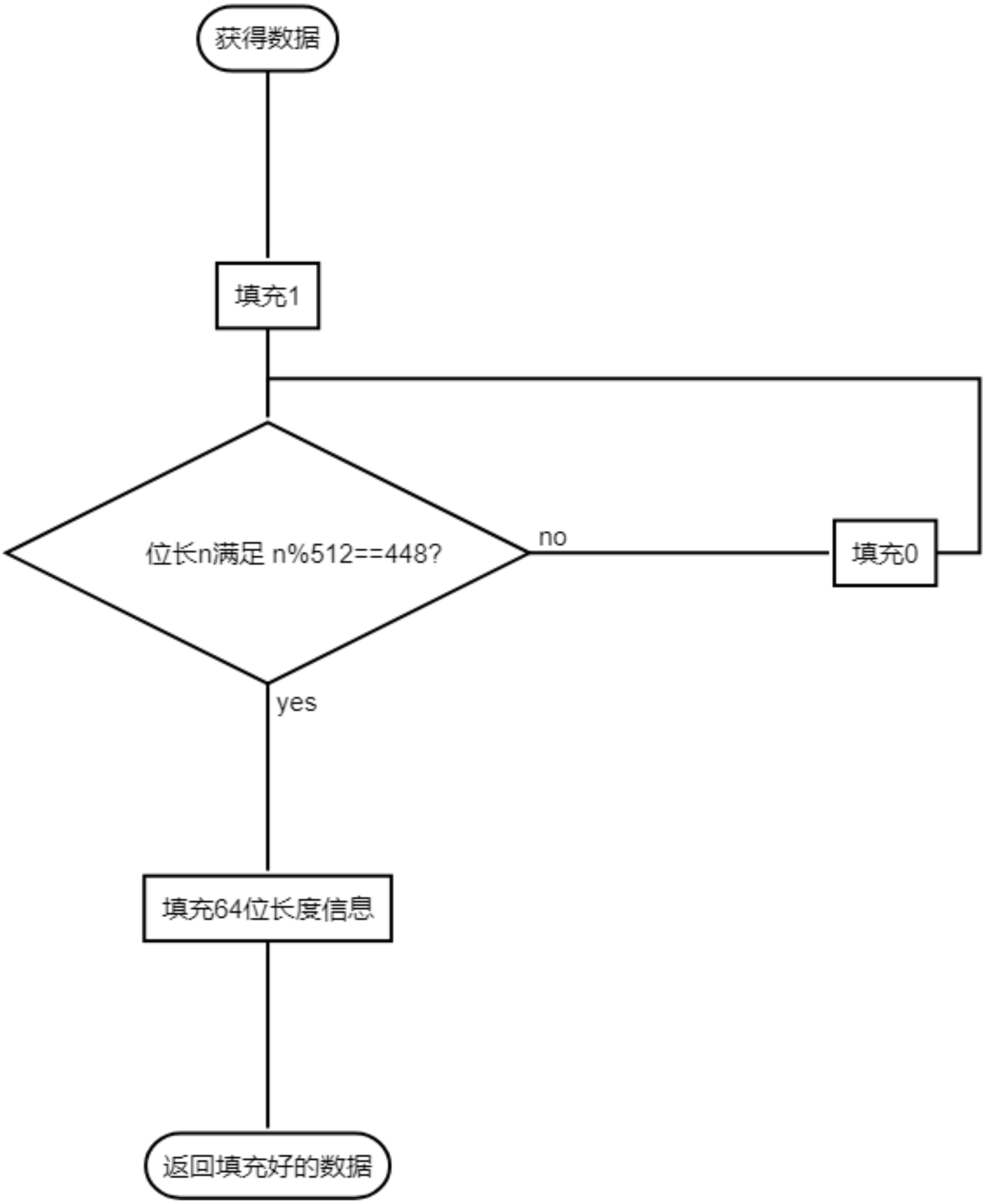
C++ 实现

```cpp

void pad_message(uint8_t* input, uint8_t* output, size_t* message_len) {
  // Calculate how many bits remaining.
  uint64_t diff = (MD5_BLOCK_LEN + MD5_MESSAGE_LEN_REMAINDER -
                   *message_len % MD5_BLOCK_LEN) %
                  MD5_BLOCK_LEN;
  // If diff = 0, we set it to 512.
  if (diff == 0) {
    diff = MD5_BLOCK_LEN;
  }

  uint8_t* buffer = new uint8_t[*message_len + diff];
  memset(buffer, 0, *message_len + diff);
  // First we do a memcpy on the output.
  memcpy(static_cast<void*>(buffer), static_cast<void*>(input), *message_len);
  // Pad 1 => 10000000 = 0x80.
  *(buffer + *message_len) = 0x80;
  // Pad zeros.
  memset(buffer + *message_len + 1, 0, diff - 1);
  // Prepare for the output.
  memset(output, 0, MD5_HEADER + *message_len + diff);
  // Get the 64-bit string of the length variable and then pad it to output.
```

```
  to_64_bits(*message_len * 8, output + *message_len + diff);
  memcpy(output, buffer, *message_len + diff);
  // Finally set the length of the message.
  *message_len = *message_len + diff + MD5_HEADER;
}
```



Part 2. 基本的非线性函数

MD5中有A、B、C、D，4个32位被称为链接变量的整数参数，它们的初始值分别为：

```
A0 = 0x01234567
B0 = 0x89abcdef
C0 = 0xfedcba98
D0 = 0x76543210
```

当设置好这4个链接变量后，就开始进入算法的4轮循环运算。循环的次数是信息中512位信息分组数目。

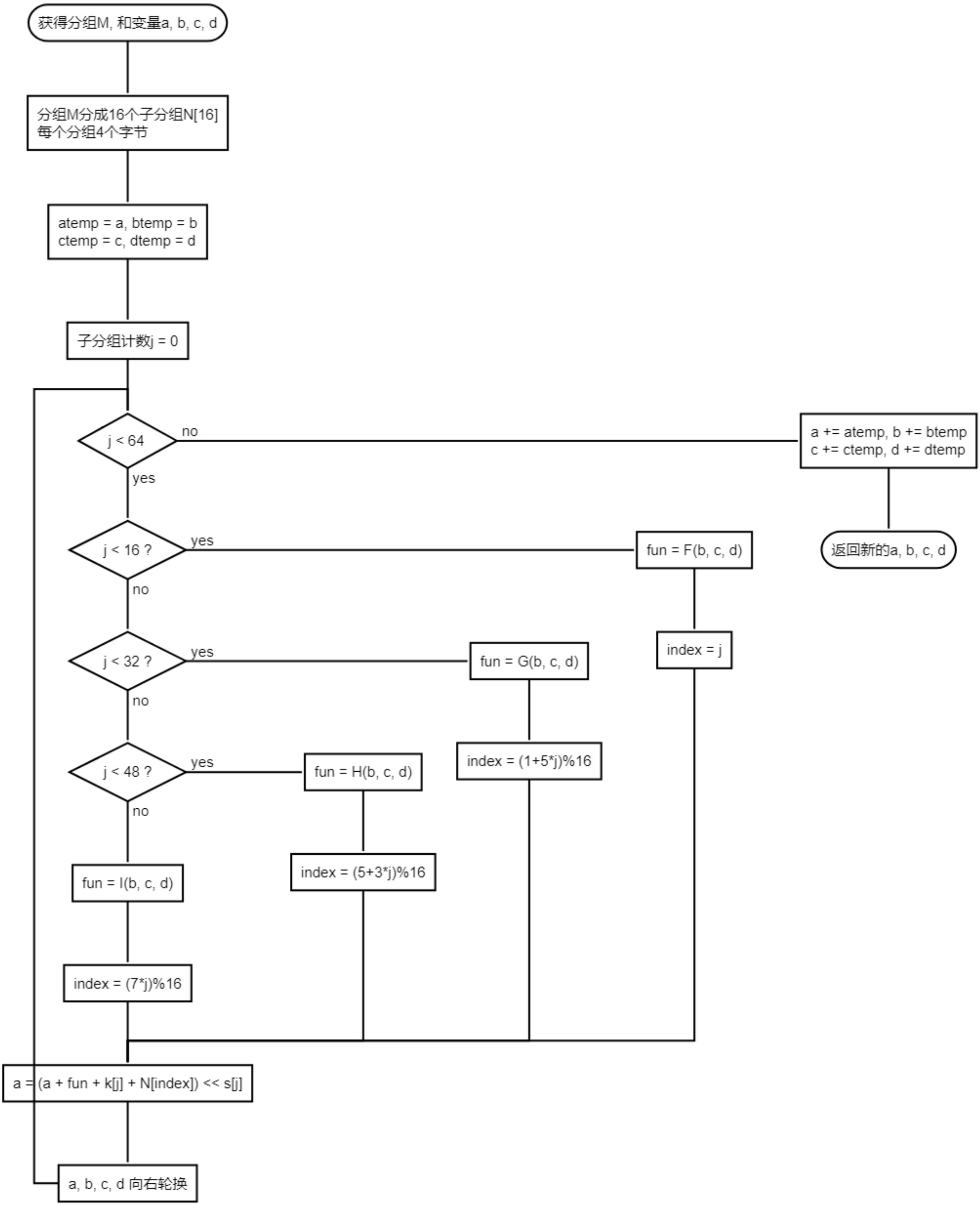
然后进入主循环，主循环有4轮，每轮循环都很相似。第一轮进行16次操作，每次操作对A、B、C、D中的3个做一次非线性函数运算，然后将所得结果加上第四个变量，文本的一个子分组（32位）和一个常数。再将所得结果向左循环移S位，并加上A、B、C、D其中之一。最后用该结果取代A、B、C、D其中之一。其中四个非线性函数为

$$
\begin{gathered}
F(B,C,D) = (B \wedge C) \vee (\overline{B} \wedge D) \\
G(B,C,D) = (B \wedge D) \vee (C \wedge \overline{D}) \\
H(B,C,D) = B \oplus C \oplus D \\
I(B,C,D) = C \oplus (B \vee \overline{D})
\end{gathered}
$$

C++ 实现

```cpp
// Basic logic functions used at each round.
#define F(b, c, d) (((b) & (c)) | ((~b) & (d)))
#define G(b, c, d) (((b) & (d)) | ((c) & (~d)))
#define H(b, c, d) ((b) ^ (c) ^ (d))
#define I(b, c, d) ((b) ^ ((b) | (~d)))
```

Part 3. MD5主循环



我们可以定义此处的`a = b + ((a + fun(b,c,d) + x[index] + constant) << s[j])`包装成宏定义的函数，例如

```
#define II(a, b, c, d, x, s, constant)        \
  {                                           \
    (a) += I((b), (c), (d)) + (x) + constant; \
    (a) = ROT_LEFT((a), (s));                 \
    (a) += (b);                               \
  }
```

最后可以定义主循环一轮为

```
void digest_one_block(unsigned char* input, uint32_t* state) {
  uint32_t* pointer = reinterpret_cast<uint32_t*>(input);

  uint32_t a = state[0];
  uint32_t b = state[1];
  uint32_t c = state[2];
  uint32_t d = state[3];

  /* Round 1 */
  FF(a, b, c, d, pointer[0], shift_table[0][0], 0xd76aa478);
  FF(d, a, b, c, pointer[1], shift_table[0][1], 0xe8c7b756);
  FF(c, d, a, b, pointer[2], shift_table[0][2], 0x242070db);
  FF(b, c, d, a, pointer[3], shift_table[0][3], 0xc1bdceee);

  FF(a, b, c, d, pointer[4], shift_table[0][0], 0xf57c0faf);
  FF(d, a, b, c, pointer[5], shift_table[0][1], 0x4787c62a);
  FF(c, d, a, b, pointer[6], shift_table[0][2], 0xa8304613);
  FF(b, c, d, a, pointer[7], shift_table[0][3], 0xfd469501);

  FF(a, b, c, d, pointer[8], shift_table[0][0], 0x698098d8);
  FF(d, a, b, c, pointer[9], shift_table[0][1], 0x8b44f7af);
  FF(c, d, a, b, pointer[10], shift_table[0][2], 0xffff5bb1);
  FF(b, c, d, a, pointer[11], shift_table[0][3], 0x895cd7be);

  FF(a, b, c, d, pointer[12], shift_table[0][0], 0x6b901122);
  FF(d, a, b, c, pointer[13], shift_table[0][1], 0xfd987193);
  FF(c, d, a, b, pointer[14], shift_table[0][2], 0xa679438e);
  FF(b, c, d, a, pointer[15], shift_table[0][3], 0x49b40821);

  /* Round 2 */
  GG(a, b, c, d, pointer[1], shift_table[1][0], 0xf61e2562);
  GG(d, a, b, c, pointer[6], shift_table[1][1], 0xc040b340);
  GG(c, d, a, b, pointer[11], shift_table[1][2], 0x265e5a51);
```

```
    GG(b, c, d, a, pointer[0], shift_table[1][3], 0xe9b6c7aa);

    GG(a, b, c, d, pointer[5], shift_table[1][0], 0xd62f105d);
    GG(d, a, b, c, pointer[10], shift_table[1][1], 0x2441453);
    GG(c, d, a, b, pointer[15], shift_table[1][2], 0xd8a1e681);
    GG(b, c, d, a, pointer[4], shift_table[1][3], 0xe7d3fbc8);

    GG(a, b, c, d, pointer[9], shift_table[1][0], 0x21e1cde6);
    GG(d, a, b, c, pointer[14], shift_table[1][1], 0xc33707d6);
    GG(c, d, a, b, pointer[3], shift_table[1][2], 0xf4d50d87);
    GG(b, c, d, a, pointer[8], shift_table[1][3], 0x455a14ed);

    GG(a, b, c, d, pointer[13], shift_table[1][0], 0xa9e3e905);
    GG(d, a, b, c, pointer[2], shift_table[1][1], 0xfcefa3f8);
    GG(c, d, a, b, pointer[7], shift_table[1][2], 0x676f02d9);
    GG(b, c, d, a, pointer[12], shift_table[1][3], 0x8d2a4c8a);

    /* Round 3 */
    HH(a, b, c, d, pointer[5], shift_table[2][0], 0xfffa3942);
    HH(d, a, b, c, pointer[8], shift_table[2][1], 0x8771f681);
    HH(c, d, a, b, pointer[11], shift_table[2][2], 0x6d9d6122);
    HH(b, c, d, a, pointer[14], shift_table[2][3], 0xfde5380c);

    HH(a, b, c, d, pointer[1], shift_table[2][0], 0xa4beea44);
    HH(d, a, b, c, pointer[4], shift_table[2][1], 0x4bdecfa9);
    HH(c, d, a, b, pointer[7], shift_table[2][2], 0xf6bb4b60);
    HH(b, c, d, a, pointer[10], shift_table[2][3], 0xbebfbc70);

    HH(a, b, c, d, pointer[13], shift_table[2][0], 0x289b7ec6);
    HH(d, a, b, c, pointer[0], shift_table[2][1], 0xeaa127fa);
    HH(c, d, a, b, pointer[3], shift_table[2][2], 0xd4ef3085);
    HH(b, c, d, a, pointer[6], shift_table[2][3], 0x4881d05);

    HH(a, b, c, d, pointer[9], shift_table[2][0], 0xd9d4d039);
    HH(d, a, b, c, pointer[12], shift_table[2][1], 0xe6db99e5);
    HH(c, d, a, b, pointer[15], shift_table[2][2], 0x1fa27cf8);
    HH(b, c, d, a, pointer[2], shift_table[2][3], 0xc4ac5665);

    /* Round 4 */
    II(a, b, c, d, pointer[0], shift_table[3][0], 0xf4292244);
    II(d, a, b, c, pointer[7], shift_table[3][1], 0x432aff97);
    II(c, d, a, b, pointer[14], shift_table[3][2], 0xab9423a7);
```

```
    II(b, c, d, a, pointer[5], shift_table[3][3], 0xfc93a039);

    II(a, b, c, d, pointer[12], shift_table[3][0], 0x655b59c3);
    II(d, a, b, c, pointer[3], shift_table[3][1], 0x8f0ccc92);
    II(c, d, a, b, pointer[10], shift_table[3][2], 0xffeff47d);
    II(b, c, d, a, pointer[1], shift_table[3][3], 0x85845dd1);

    II(a, b, c, d, pointer[8], shift_table[3][0], 0x6fa87e4f);
    II(d, a, b, c, pointer[15], shift_table[3][1], 0xfe2ce6e0);
    II(c, d, a, b, pointer[6], shift_table[3][2], 0xa3014314);
    II(b, c, d, a, pointer[13], shift_table[3][3], 0x4e0811a1);

    II(a, b, c, d, pointer[4], shift_table[3][0], 0xf7537e82);
    II(d, a, b, c, pointer[11], shift_table[3][1], 0xbd3af235);
    II(c, d, a, b, pointer[2], shift_table[3][2], 0x2ad7d2bb);
    II(b, c, d, a, pointer[9], shift_table[3][3], 0xeb86d391);

    state[0] += a;
    state[1] += b;
    state[2] += c;
    state[3] += d;
}
```

其中定义的各个常数表为

```
static const uint32_t initial_buffer[4] = {
    0x67452301,
    0xefcdab89,
    0x98badcfe,
    0x10325476,
};

static const uint32_t shift_table[4][4] = {
    7, 12, 17, 22, 5, 9, 14, 20, 4, 11, 16, 23, 6, 10, 15, 21,
};
```

数据测试

```
$ ./md5
0xd4 0x1d 0x8c 0xd9 0x8f 0x00 0xb2 0x04 0xe9 0x80 0x09 0x98 0xec 0xf8 0x42 0x7e
$ ./md5 a
```

```
0x0c 0xc1 0x75 0xb9 0xc0 0xf1 0xb6 0xa8 0x31 0xc3 0x99 0xe2 0x69 0x77 0x26 0x61
$ ./md5 abc
0x90 0x01 0x50 0x98 0x3c 0xd2 0x4f 0xb0 0xd6 0x96 0x3f 0x7d 0x28 0xe1 0x7f 0x72
$ ./md5 message\ digest
0xf9 0x6b 0x69 0x7d 0x7c 0xb7 0x93 0x8d 0x52 0x5a 0x2f 0x31 0xaa 0xf1 0x61 0xd0
$ ./md5 abcdefghijklmnopqrstuvwxyz
0xc3 0xfc 0xd3 0xd7 0x61 0x92 0xe4 0x00 0x7d 0xfb 0x49 0x6c 0xca 0x67 0xe1 0x3b
$ ./md5 ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789
0xd1 0x74 0xab 0x98 0xd2 0x77 0xd9 0xf5 0xa5 0x61 0x1c 0x2c 0x9f 0x41 0x9d 0x9f
$ ./md5 12345678901234567890123456789012345678901234567890123456789012345678901234567890
0x57 0xed 0xf4 0xa2 0x2b 0xe3 0xc9 0x55 0xac 0x49 0xda 0x2e 0x21 0x07 0xb6 0x7a
```

结果完全正确。

雪崩效应测试

此处我们测试五组数据，它们分别为

```
uint8_t test_case_1[] = {0x10, 0x10, 0x10, 0x10};
uint8_t test_case_2[] = {0x11, 0x10, 0x10, 0x10};
uint8_t test_case_3[] = {0x10, 0x11, 0x10, 0x10};
uint8_t test_case_4[] = {0x10, 0x10, 0x11, 0x10};
uint8_t test_case_5[] = {0x10, 0x10, 0x10, 0x11};
```

它们的输出分别为

```
0xd6 0x3e 0xc6 0x6a 0xc0 0x6a 0x81 0xf5 0x94 0x6a 0x31 0xad 0x1a 0xae 0x47 0x1a
0xcd 0x9a 0x31 0x8f 0x5a 0x09 0x42 0x1f 0x5a 0x0c 0x0c 0x90 0x7c 0xa0 0x8a 0x2d
0x2b 0x3b 0x95 0x72 0x80 0xc9 0x80 0x94 0x11 0xb4 0xc8 0x5f 0xe8 0xf3 0x01 0x09
0x76 0xce 0x4d 0xa0 0xbb 0x06 0x70 0xeb 0xda 0xcb 0x90 0xd6 0x5f 0x04 0x66 0xec
0x37 0x8c 0x2b 0x5b 0xdf 0x58 0xfb 0xee 0xf5 0xd0 0x82 0xe6 0x44 0x9a 0x96 0xda
```

可见雪崩效应是存在的。

使用方法

安装好CMake，版本大于等于3.20：

```
$ sudo apt-get install cmake  # Ubuntu 用户
```

```
$ brew install cmake          # macOS 用户
```

可以参阅CMake安装。

Linux（or WSL）或macOS系统下，在当前目录下执行以下命令：

```
$ mkdir -p build
$ cd build
$ cmake ..
$ make
$ ./md5
```