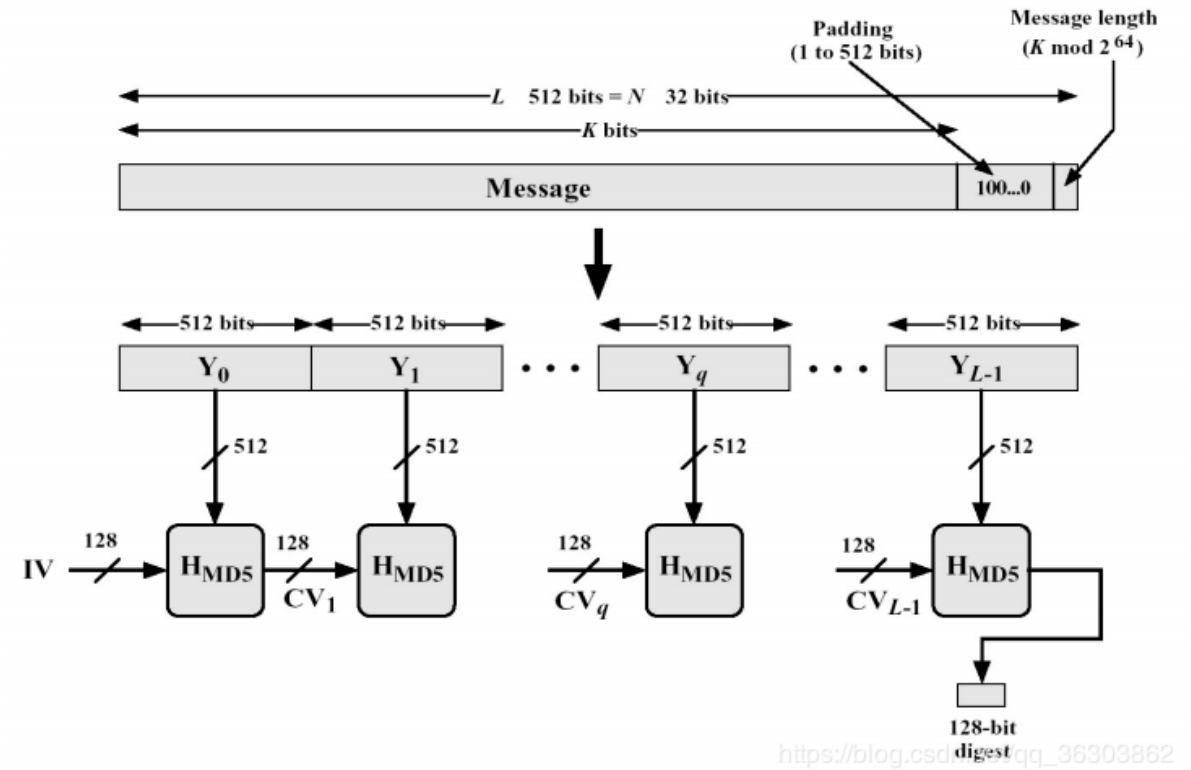
算法原理概述

MD5算法将一串信息流分成信息块，对每一个信息块进行压缩操作，最终实现任意长度的信息输入，定长字节串输出。MD5算法利用了哈希函数的方法，对任意输入的信息，找到具有相同输出的信息，其概率为 $\frac{1}{2^n}$,其中n为输出长度。MD5算法常用于消息验证。

总体结构

MD5算法步骤包括：填充、分块、缓冲区初始化、循环压缩和得出结果。

❑ MD5 的基本流程



模块分解

填充

MD5处理的信息块是定长的，为512bits且需要64bit位置存储消息长度，消息长度+填充长度 = 448 (mod 512)。填充位数为1~512，至少填充1bit，且第一位填充 '1'，其余位填充 '0'。经过填充操作，得到字节流 L。

分块

填充完成后，需将L分为512bits 长度的字节块组。本例是将512bit信息存储于16长度的`unsigned char`数组中，需要注意的是，MD5 使用的是 little-endian模式，在转换时需注意bit的存储顺序。

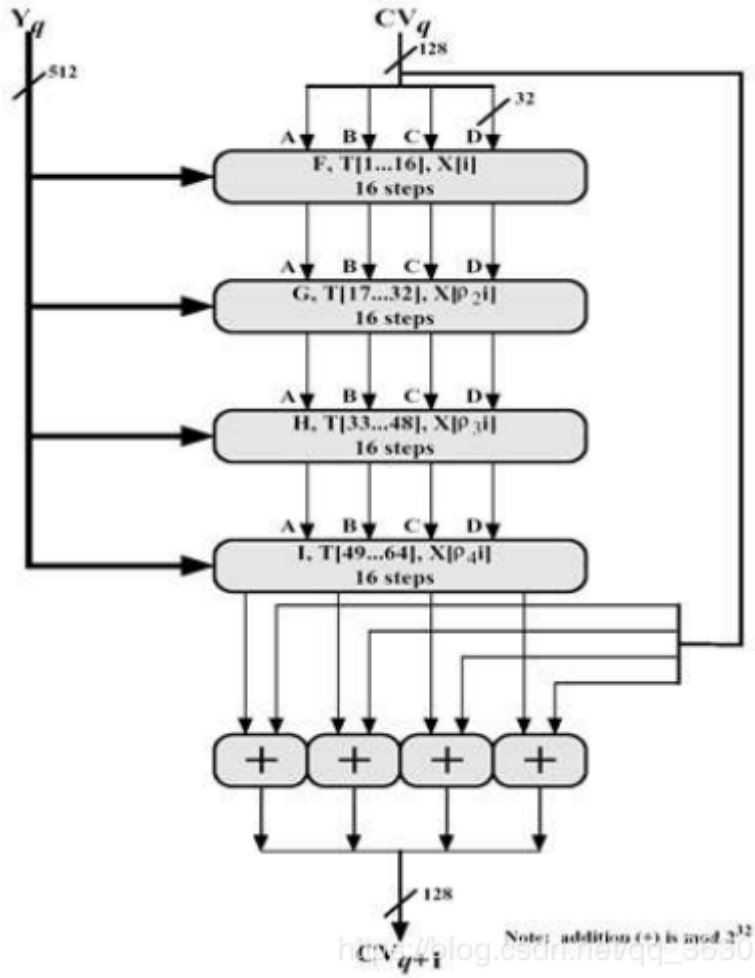
缓冲区初始化

初始化一个128-bit 的 MD 缓冲区，记为 CVq，表示成4个32-bit寄存器 (A, B, C, D)；CV0 = IV。迭代在 MD 缓冲区进行，最后一步的128-bit 输出即为算法结果。

循环压缩

将前一步的输出（或初始CV）和当前步骤的消息块作为压缩函数的输入，不断压缩消息块，最终得到一个128bits的字节串。

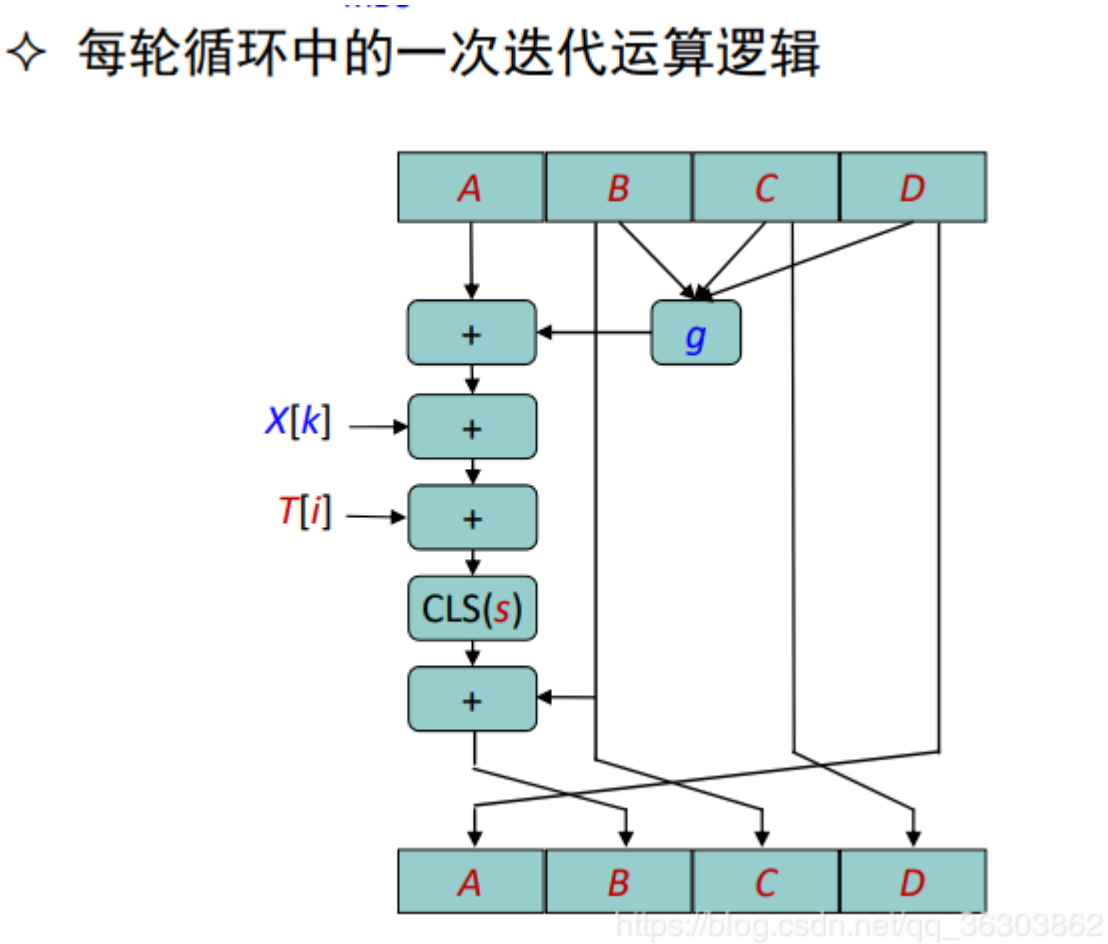
压缩函数有四轮外层循环，每个外层循环嵌套16轮循环。



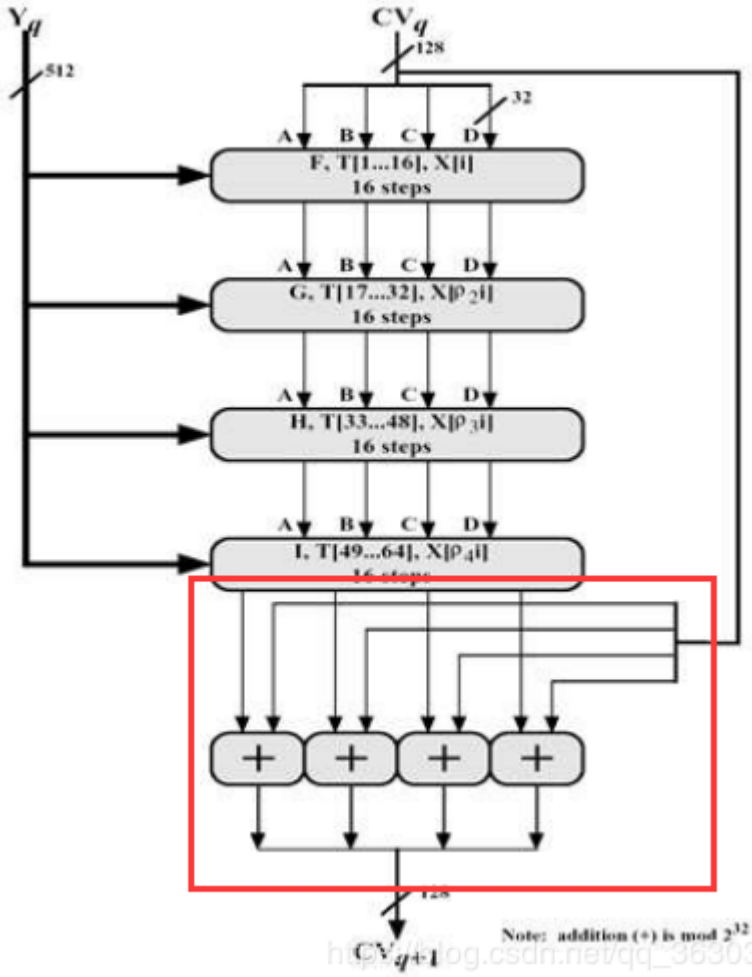
内层循环执行 a → b + ((a + g(b, c, d) + X[k] + T[i]) << s 操作，g函数如下图：

轮次	Function *g*	*g*(*b*, *c*, *d*)
1	*F*(*b*, *c*, *d*)	$(b \wedge c) \vee (\neg b \wedge d)$
2	*G*(*b*, *c*, *d*)	$(b \wedge d) \vee (c \wedge \neg d)$
3	*H*(*b*, *c*, *d*)	$b \oplus c \oplus d$
4	*I*(*b*, *c*, *d*)	$c \oplus (b \vee \neg d)$

每次循环结束还需交换缓冲区内容：



外层循环结束后输出需与CV对应部分进行模2^{32}加法操作：



重复执行循环压缩函数，直到消息处理完毕，输出即为压缩后的消息。

数据结构

使用数组向量。

源代码

转换和填充函数：根据小端规则将字符流赋值给`unsigned char`数组，再进行填充和加入长度值，此处先将消息存储于大数组中，之后再转入分块向量中，虽然效率降低了，但易于理解和操作。

```
void MD5::read(string str)
{
    unsigned int big_temp[1000] = {0};
    unsigned int sub_index = 0;
    unsigned int temp;
    big_temp[sub_index] = 0;
    int i;
    long long int   count = str.length() * 8;
    for (i = 0; i < str.length(); i++)//transfer message
    {
        temp = str[i];
        big_temp[sub_index] += temp << ((i % 4) * 8);
        if (i % 4 == 3)
            sub_index++;
    }

    temp = 0x80;//padding
    big_temp[sub_index] += temp << ((i % 4) * 8);
    if (i % 4 == 3)
        sub_index++;
    i++;
    temp = 0x00;
    while ((i * 8) % 512 != 448)
    {
        big_temp[sub_index] += temp << ((i % 4) * 8);
        if (i % 4 == 3)
            sub_index++;
        i++;
    }
    big_temp[sub_index++] = (count << 32) >> 32;
    big_temp[sub_index] = count >> 32;
    unsigned int * M;
    for (int i = 0; i <= sub_index; i++)//add message block to MessageFlow
    {
        if (i % 16 == 0)
        {
            M = new unsigned int[16];
        }
        else if (i % 16 == 15)
        {
            MessageFlow.push_back(M);
        }
        M[i % 16] = big_temp[i];
    }

}
```

压缩函数：实现方法与上面描述一致

```
void MD5::HMD5(int q)
{
    unsigned int a, b, c, d;
    a = A;
    b = B;
    c = C;
    d = D;
    unsigned int temp;
    unsigned int X, sub_index;
    for (int i = 0; i < 4; i++)
    {
        for (int j = 0; j < 16; j++)
        {
            sub_index = i * 16 + j;
            X = MessageFlow[q][index[sub_index]];
            temp = a + g2(i, b, c, d) + X + T[sub_index];

            temp = (temp << s[sub_index]) | (temp >> (32 - s[sub_index]));
            a = b + temp;
            temp = d;
            d = c;
            c = b;
            b = a;
            a = temp;
        }
    }
    A = A + a;
    B = B + b;
    C = C + c;
    D = D + d;
}
```

编译运行结果

此处使用的测试例子来自于 RFC 1321，结果如下：

```cpp
int main()
{
    int t;
    test("");
    cout << "d41d8cd98f00b204e9800998ecf8427e" << endl << endl;

    test("ab");
    cout << "187ef4436122d1cc2f40dc2b92f0eba0" << endl << endl;

    test("abc");
    cout << "900150983cd24fb0d6963f7d28e17f72" << endl << endl;

    test("message digest");
    cout << "f96b697d7cb7938d525a2f31aaf161d0" << endl << endl;

    test("abcdefghijklmnopqrstuvwxyz");
    cout << "c3fcd3d76192e4007dfb496cca67e13b" << endl << endl;

    test("ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789");
    cout << "d174ab98d277d9f5a5611c2c9f419d9f" << endl << endl;

    test("12345678901234567890123456789012345678901234567890123456789012345678901234567890");
    cout << "57edf4a22be3c955ac49da2e2107b67a" << endl << endl;
    cin >> t;
}
```

G:\githubCode\MD5\Debug\MD5.exe

```
d41d8cd98f00b204e9800998ecf8427e
d41d8cd98f00b204e9800998ecf8427e

187ef4436122d1cc2f40dc2b92f0eba0
187ef4436122d1cc2f40dc2b92f0eba0

900150983cd24fb0d6963f7d28e17f72
900150983cd24fb0d6963f7d28e17f72

f96b697d7cb7938d525a2f31aaf161d0
f96b697d7cb7938d525a2f31aaf161d0

c3fcd3d76192e4007dfb496cca67e13b
c3fcd3d76192e4007dfb496cca67e13b

d174ab98d277d9f5a5611c2c9f419d9f
d174ab98d277d9f5a5611c2c9f419d9f

57edf4a22be3c955ac49da2e2107b67a
57edf4a22be3c955ac49da2e2107b67a
```