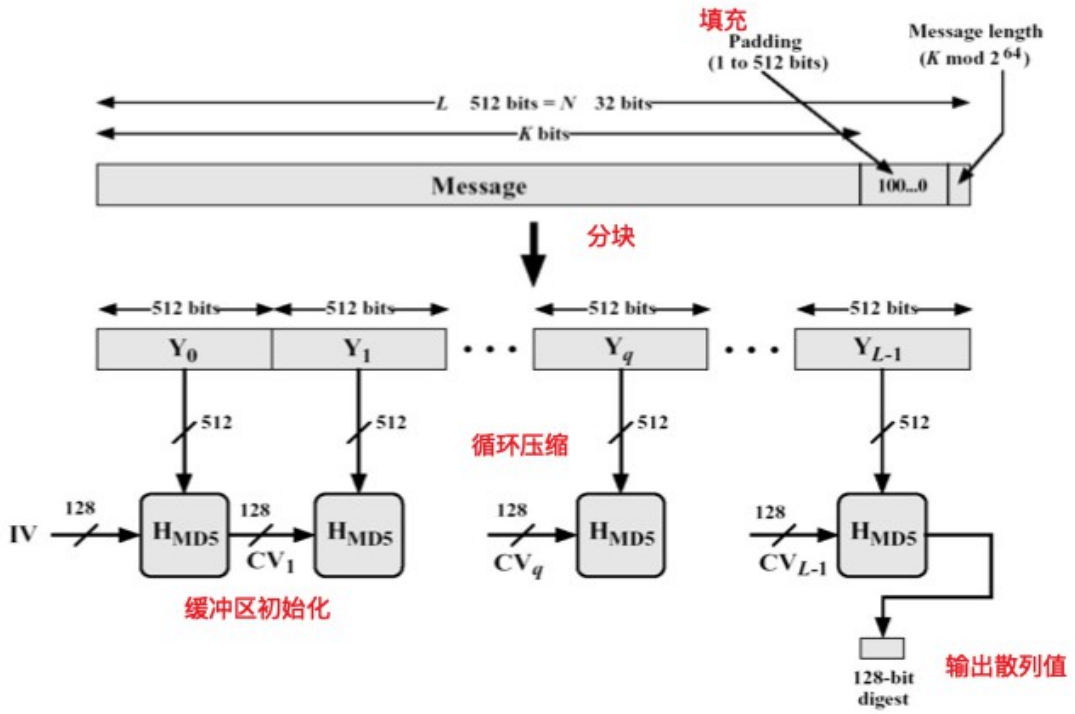
MD5

算法原理概述

MD5 使用 little-endian (小端模式)，输入任意不定长度信息，以 512-bit 进行分组，生成四个32-bit 数据，最后联合输出固定 128-bit 的信息摘要。

总体结构



模块分解

1. 填充和分组

对长度为 K bits 的原始消息数据补位，设补位后信息的长度为LEN(bit)，则LEN%512 = 448(bit)，即数据扩展至 K * 512 + 448(bit)。即K * 64+56(byte)，K为整数。补位操作始终要执行，如果补位前信息的长度对512求余的结果是448，则补512位。

具体补位操作：补一个1，然后补0至满足上述要求。总共最少要补1bit，最多补512bit。

再向上述填充好的消息尾部附加 K 值的低64位 (即 K mod 2^64)， 最后得到的数据长度是16个字(32byte)的整数倍。

把填充后的消息结果分割为 L 个 512-bit 分组：Y0，Y1，...，YL-1，

分组结果也可表示成 N 个32-bit 字：M0，M1，...，MN-1，N=L*16

2. 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。

初始化使用的是十六进制表示的数字，注意低字节在前：

word A: 01 23 45 67

word B: 89 ab cd ef

word C: fe dc ba 98

word D: 76 54 32 10

寄存器 (*A*, *B*, *C*, *D*) 置16进制初值作为初始向量 *IV*，并采用小端存储 (little-endian) 的存储结构：

- *A* = 0x67452301
- *B* = 0xEFCDAB89
- *C* = 0x98BADCFE
- *D* = 0x10325476

Word *A*	01	23	45	67
Word *B*	89	AB	CD	EF
Word *C*	FE	DC	BA	98
Word *D*	76	54	32	10

- Little-Endian 将低位字节排放在内存的低地址端，高位字节排放在内存的高地址端。相反 Big-Endian 将高位字节排放在内存的低地址端，低位字节排放在内存的高地址端。存储结构与 CPU 体系结构和语言编译器有关。PowerPC 系列采用 Big Endian 方式存储数据，而 Intel x86系列则采用 Little Endian 方式存储。

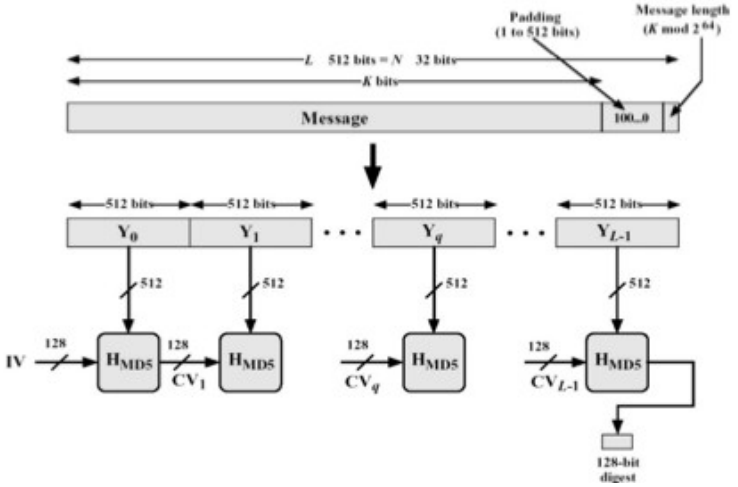
3. 总控流程

以512-bit 消息分组为单位，每一分组 Y_q (q = 0, 1, ..., L-1) 经过4个循环的压缩算法，表示为：

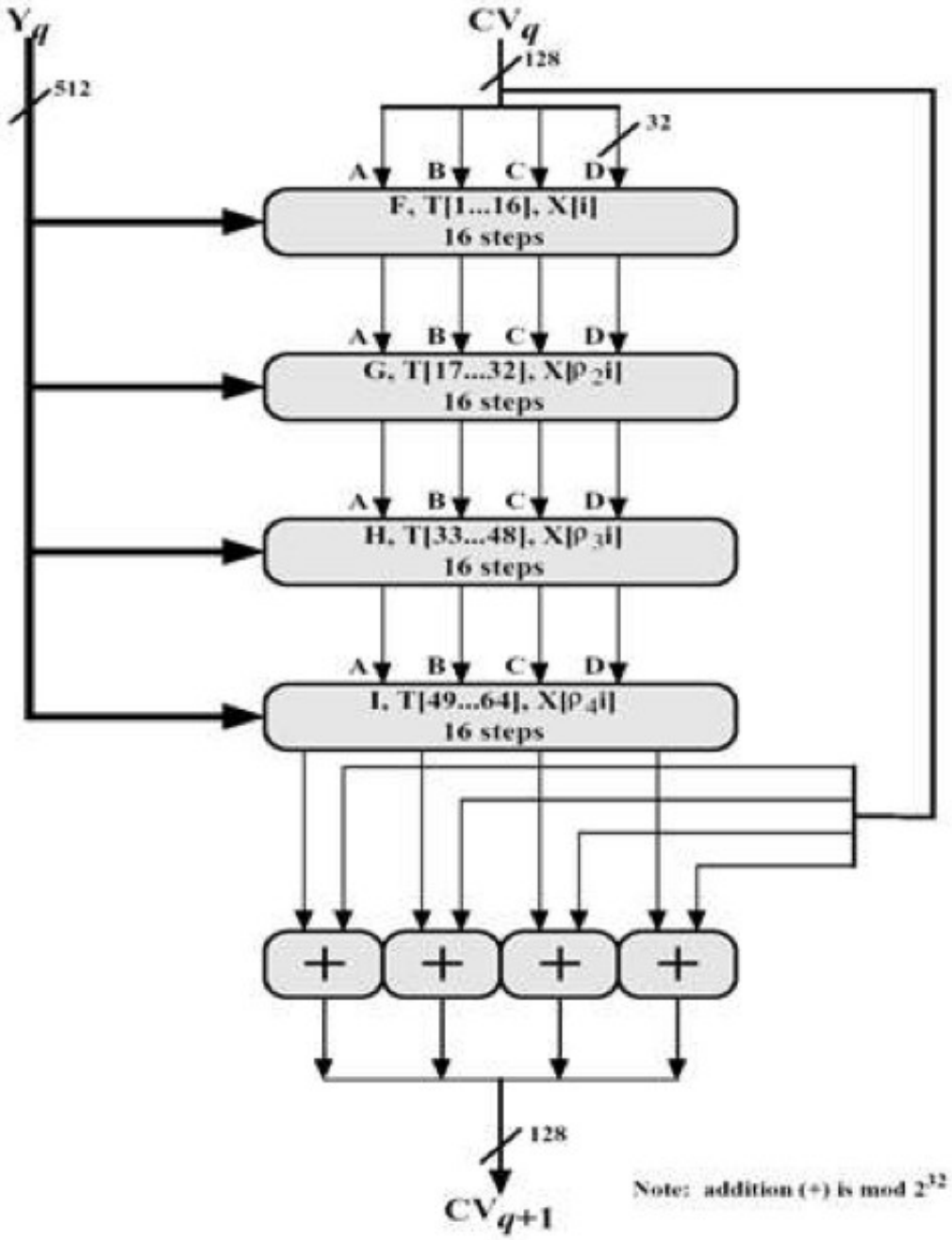
$$CV_0 = IV$$

$$CV_i = H_{MD5}(CV_{i-1}, Y_i)$$

输出结果：$MD = CV_L$.



Hmd5从CV输入128位，从消息分组输入512位，完成4轮循环后，输出128位，用于下一轮输入的 CV 值。



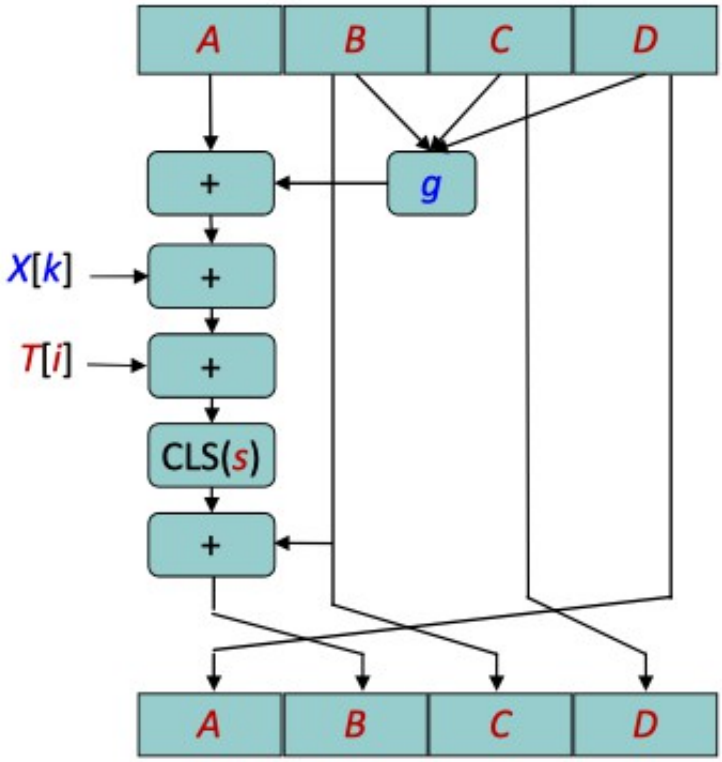
MD5 第 *q* 分组的4轮循环逻辑 (压缩函数)

每轮循环分别固定不同的生成函数F, G, H, I，结合指定的T表元素T[]和消息分组的不同部分X[] 做16次迭代运算，生成下一轮循环的输入。4轮循环总共有64次迭代运算。

4. 压缩函数

4轮循环中使用的生成函数 (轮函数) g 是一个32位非线性逻辑函数

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	$I(b, c, d)$	$c \oplus (b \vee \neg d)$



- g : 轮函数 (F, G, H, I 中的一个)。
- <<<s : 将32位输入循环左移 (CLS) s 位。
- X[k] : 当前处理消息分组的第 k 个 (k = 0..15) 32位字。
- T[i] : T 表的第 i 个元素，32位字；T表总共有64个元素，也称为加法常数。
- +: 模 2^32 加法。

各轮循环中第 i 次迭代 (i = 1..16) 使用的 $X[k]$ 的确定：

设 $j = i - 1$：

- 第1轮迭代：$k = j$.
  - 顺序使用 X[0, 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13,14,15]
- 第2轮迭代：$k = (1 + 5j) \bmod 16$.
  - 顺序使用 X[1, 6,11, 0, 5,10,15, 4, 9,14, 3, 8,13, 2, 7,12]
- 第3轮迭代：$k = (5 + 3j) \bmod 16$.
  - 顺序使用 X[5, 8,11,14, 1, 4, 7,10,13, 0, 3, 6, 9,12,15, 2]
- 第4轮迭代：$k = 7j \bmod 16$.
  - 顺序使用 X[0, 7,14, 5,12, 3,10, 1, 8,15, 6,13, 4,11, 2, 9]

T 表的生成：

T[i] = int(2^32 *abs(sin(i))

各次迭代运算采用的 T 值：

T[1.. 4] = { 0xd76aa478, 0xe8c7b756, 0x242070db, 0xc1bdceee }

T[5.. 8] = { 0xf57c0faf, 0x4787c62a, 0xa8304613, 0xfd469501 }

T[9..12] = { 0x698098d8, 0x8b44f7af, 0xffff5bb1, 0x895cd7be }

T[13..16] = { 0x6b901122, 0xfd987193, 0xa679438e, 0x49b40821 }

T[17..20] = { 0xf61e2562, 0xc040b340, 0x265e5a51, 0xe9b6c7aa }

T[21..24] = { 0xd62f105d, 0x02441453, 0xd8a1e681, 0xe7d3fbc8 }

T[25..28] = { 0x21e1cde6, 0xc33707d6, 0xf4d50d87, 0x455a14ed }

T[29..32] = { 0xa9e3e905, 0xfcefa3f8, 0x676f02d9, 0x8d2a4c8a }

各次迭代运算采用的 *T* 值：

T[33..36] = { 0xfffa3942, 0x8771f681, 0x6d9d6122, 0xfde5380c }
T[37..40] = { 0xa4beea44, 0x4bdecfa9, 0xf6bb4b60, 0xbebfbc70 }
T[41..44] = { 0x289b7ec6, 0xeaa127fa, 0xd4ef3085, 0x04881d05 }
T[45..48] = { 0xd9d4d039, 0xe6db99e5, 0x1fa27cf8, 0xc4ac5665 }
T[49..52] = { 0xf4292244, 0x432aff97, 0xab9423a7, 0xfc93a039 }
T[53..56] = { 0x655b59c3, 0x8f0ccc92, 0xffeff47d, 0x85845dd1 }
T[57..60] = { 0x6fa87e4f, 0xfe2ce6e0, 0xa3014314, 0x4e0811a1 }
T[61..64] = { 0xf7537e82, 0xbd3af235, 0x2ad7d2bb, 0xeb86d391 }

各次迭代运算采用的左循环移位的 *s* 值：

s[1..16] = { 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22 }
s[17..32] = { 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20 }
s[33..48] = { 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23 }
s[49..64] = { 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21 }

数据结构

编译运行结果