磁盘文件管理工具

本项目旨在实现一个文件管理工具，主要功能是删除磁盘中的重复文件。如果判断两个文件是否完全一致，我们采用计算文件指纹的方法，通过指纹验证的方法判断两个文件是否相同。所谓的文件指纹即数字签名。

常用的计算机签名方法

数字签名，就是只有信息的发送者才能产生的别人无法伪造的一段数字串，这段数字串同时也是对信息的发送者发送信息真实性的一个有效证明。 是一种类似写在纸上的普通的物理签名 ，比如合同签名。

数字签名有两种功效：一是能确定消息确实是由发送方签名并发出来的，因为别人假冒不了发送方的签名。二是数字签名能确定消息的完整性。因为数字签名的特点是它代表了文件的特征，文件如果发生改变，数字摘要的值也将发生变化。不同的文件将得到不同的数字摘要。

常用的数字签名方法主要有：HASH算法 ，此算法主要包括MD（Message-Digest，信息摘要）和SHA（安全散列算法,Secure Hash Algorithm）两类 。Digital Signature Algorithm (DSA)， ECDSA (Elliptic Curve Digital Signature Algorithm)，椭圆曲线数字签名算法，微软产品的序列号验证算法使用的就是ECDSA。与传统的数字签名算法相比，速度快，强度高，签名短。

计算机签名MD5

1. what's MD5

MD5是由Ron Rivest在1991设计的一种信息摘要(message-digest)算法，当给定任意长度的信息，MD5会产生一个固定的128位"指纹"或者叫信息摘要。从理论的角度，所有的信息产生的MD5值都不同，也无法通过给定的MD5值产生任何信息，即不可逆。

MD5算法在RFC 1321（Request For Comments，征求修正意见书）做了详细描述。

1.1 MD5功能特点

1.输入任意长度的信息，经过处理，输出为128位的信息（数字指纹）

2.不同的输入得到的不同的结果（唯一性）。要使两个不同的信息产生相同的摘要，操作数量级在2^64次方。

3.根据128位的输出结果不可能反推出输入的信息。根据给定的摘要反推原始信息，它的操作数量级在2^128次。

1.2 MD5争议

MD5是否为加密算法：

<1>：MD5不能从密文反过来得到原文，即没有解密算法，故不能称为加密算法；

<2>：MD5处理后看不到原文，即已经将原文加密，故MD5属于加密算法。

1.3 MD5用途

1>防止信息被篡改

a) 电邮文档一致性验证：在发送一个电子文档前，先得到MD5的输出结果a。然后在对方收到电子文档后，对方也得到一个MD5的输出结果b。如果a与b一样就代表中途未被篡改。

b)文件下载安全验证：在下载网络文件/程序时，为了防止不法分子在安装程序中添加木马，一般会在网站上公布由安装文件得到的MD5输出结果。用户下载完之后，在本地计算MD5值，在于网站公布的MD5值作比较，验证文件是否完整和安全。

c)SVN update:SVN在检测文件是否在CheckOut后被修改过，也是用到了MD5.

2>防止直接看到密码明文：现在很多网站在数据库存储用户的密码的时候都是存储用户密码的MD5值。就算不法分子得到数据库种用户密码的MD5值，也无法知道用户的密码。比如在UNIX系统中用户的密码就是以MD5（或其它类似的算法）经加密后存储在文件系统中。当用户登录的时候，系统把用户输入的密码计算成MD5值，然后再去和保存在文件系统中的MD5值进行比较，进而确定输入的密码是否正确。通过这样的步骤，系统在并不知道用户密码的明码的情况下就可以确定用户登录系统的合法性。这不但可以避免用户的密码被具有系统管理员权限的用户知道，而且还在一定程度上增加了密码被破解的难度。

3>防止抵赖（数字签名）：这需要一个第三方认证机构。例如A写了一个文件，认证机构对此文件用MD5算法产生摘要信息并做好记录。若以后A说这文件不是他写的，权威机构只需对此文件重新产生摘要信息，然后跟记录在册的摘要信息进行比对，相同的话，就证明是A写的了。这就是所谓的“数字签名”。

2. MD5算法步骤

MD5的算法输入为以bit为单位的信息(1 byte = 8 * bit)，经过处理，得到一个128bit的摘要信息。这128位的摘要信息在计算过程中分成4个32bit的子信息，存储在4个buffer(A，B，C，D)中，它们初始化为固定常量。MD5算法然后使用每一个512bit的数据块去改变A，B，C，D中值，所有的数据处理完之后，把最终的A，B，C，D值拼接在一起，组成128bit的输出。处理每一块数据有四个类似的过程，每一个过程由16个相似的操作流组成，操作流中包括非

线性函数，相加以及循环左移。MD5算法大致可分为5个步骤：

添加填充位

添加bit长度

初始化MD buffer（A,B,C,D）

按512位数据逐块处理输入信息

摘要输出

2.1 添加填充位

信息的最尾部(不是每一块的尾部)要进行填充，使其最终的长度length(以bit为单位)满足length % 512 = 448，**这一步始终要执行，即使信息的原始长度恰好符合上述要求。**

填充规则：第一个bit填充位填 '**1**' ，后续bit填充位都填 '**0**' ，最终使消息的总体长度满足上述要求。总之，至少要填充 **1** bit，至多填充 **512** bit。

2.2 添加bit长度

在2.1之后，添加一个64bit大小的length，length表示原始消息(未填充之前)的bit长度，极端情况，如果消息长度超过2^64，那么只使用前2^64bit消息。

这一步完成之后，消息的最终长度变为(length + 64) % 512 = 0，即length为512的整数倍。从这里再去看第一步，至少需要填充 **8** bit，为什么？我们假设几种情况分析一下：

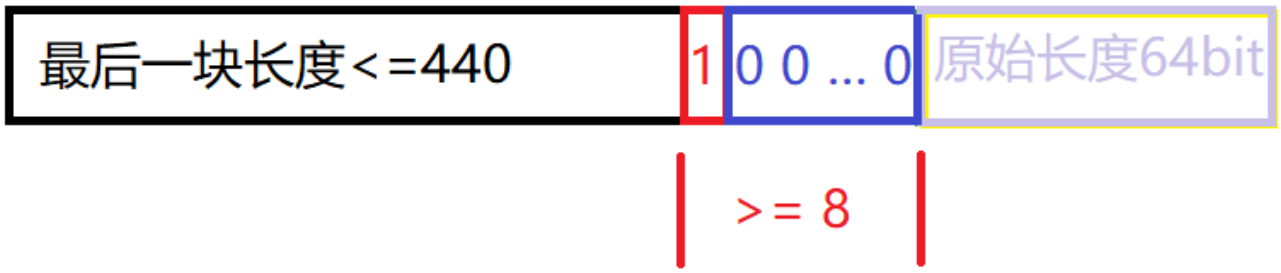
首先要明确一个字符占1byte(8bit, 中文字符的话占16bit)，所以原始信息bit长度一定是8的倍数。

假设1：消息原始长度 % 512 = 448

这时候原始长度符合要求，但是根据填充规则，仍然要至少填1bit的 '**1**'，后面还剩63bit，不够添加长度，所以需要再加一块数据(512bit)，这样后面63bit填0，新加的数据前448bit填0，最后64bit填数据原始长度，一定要记住长度值是放在最后一块数据的最后64bit。

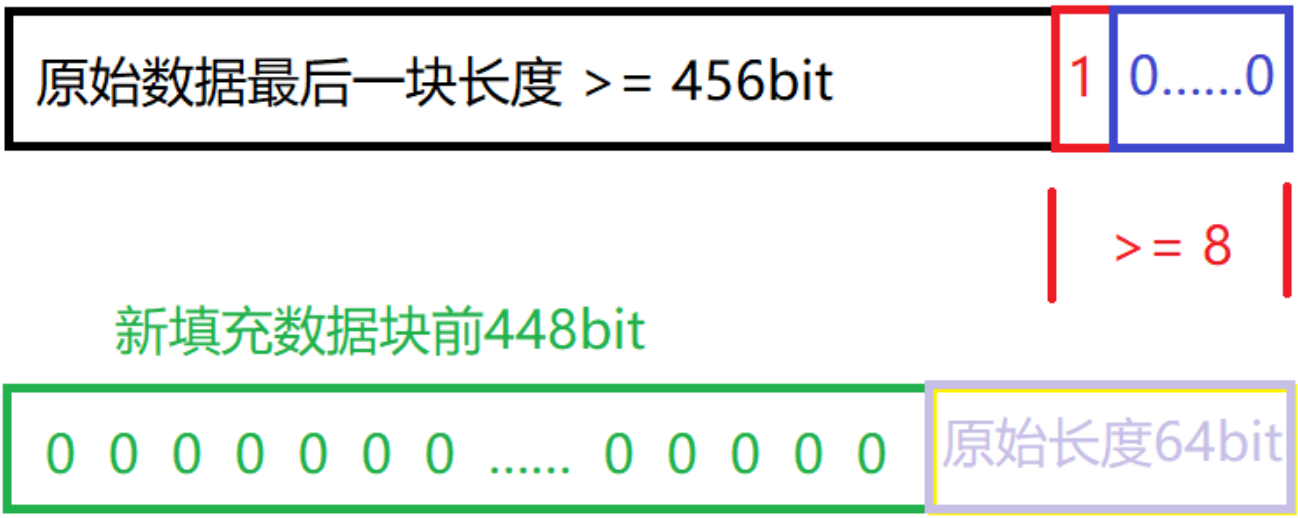
假设2：消息原始长度 % 512 < 448

此时最后一块数据长度不大于440bit，最后64bit填数据长度，需要填充的bit数不小于8。



假设3：消息原始长度 % 512 > 448

此时最后一块数据长度不小于456，最多504，剩余bit不够添加64位长度，和假设1相同，需要增加一块数据，最后64位添加长度，其余填充0。



结论：最少填充8bit，最少填充内容1000 0000，16进制即为0x80。

2.3 初始化MD buffer

用4-word buffer(A, B, C, D)计算摘要，这里A,B,C,D各为一个32bit的变量，这些变量初始化为下面的十六进制值，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
*/
// 初始化A,B,C,D
_atemp = 0x67452301;
_btemp = 0xefcdab89;
_ctemp = 0x98badcfe;
_dtemp = 0x10325476;
```

2.4 按512位数据逐块处理输入信息

512bit数据为一个处理单位，暂且称为一个数据块chunk，每个chunk经过4个函数(F, G, H, I)处理,这四个函数输入为3个32位(4字节)的值，产生一个32位的输出。四个函数如下所示：

```
/*
F(x,y,z) = (x & y) | ((~x) & z)
G(x,y,z) = (x & z) | ( y & (~z))
H(x,y,z) = x ^ y ^ z
I(x,y,z) = y ^ (x | (~z))
*/
```

处理过程中要用一个含有64个元素的表K[1.....64]，表中的元素值由sin函数构建，K[i]等于2^(32) * abs(sin(i))的整数部分，即：

```
/*
K[i] = floor(2^(32) * abs(sin(i + 1))) // 因为此处i从0开始，所以需要sin(i + 1)
*/

for (int i = 0; i < 64; i++)
{
    _k[i] = (size_t)(abs(sin(i + 1)) * pow(2, 32));
}
```

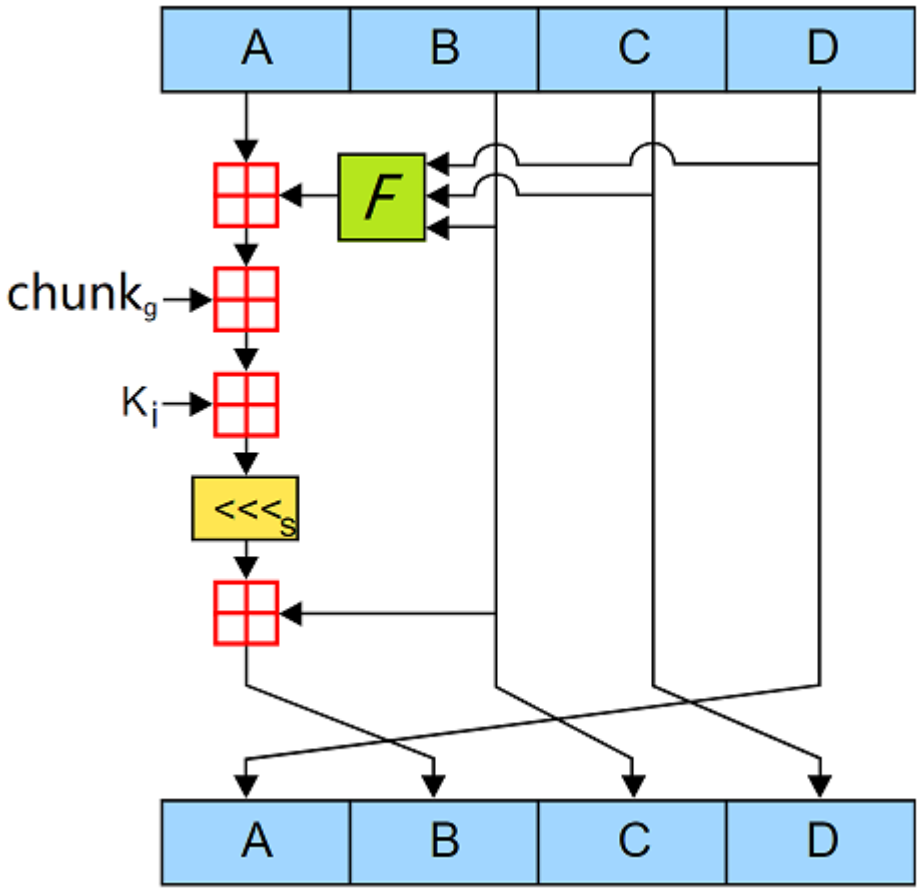
512bit = 64byte * 8 = 16 * 4byte

在处理一个chunk(512bit)的数据时,会把这个chunk再细分成**16组4字节**数据，一个chunk经过4轮进行处理，每轮都会把chunk的所有数据处理一遍，每轮有16个相似的子操作，所以一个chunk的数据要进行64个子操作。

计算之前先保存MD buffer的当前值：

a = A, b = B, c = C, d = D

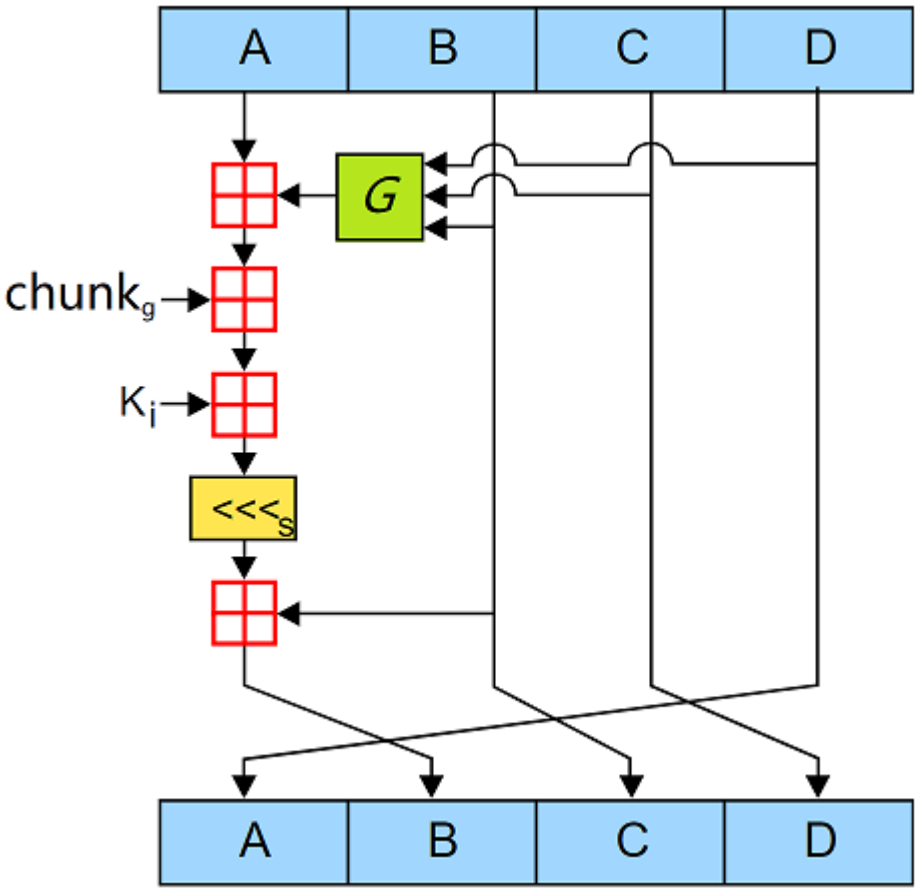
第一轮：F函数处理（0 <= i <= 15）



i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
g	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

```
F = F(b, c, d)

d = c

c =b

b = b + shift((a + F + k[i] + chunk[g]), s[i])

a = d
```

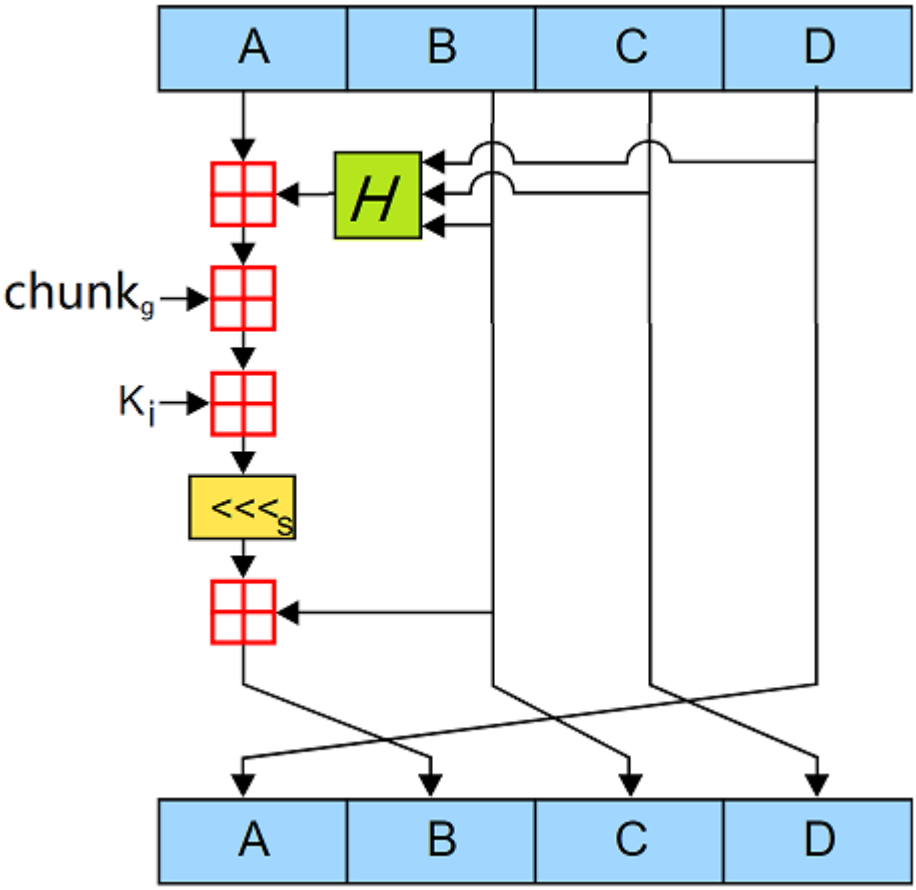
第二轮：G函数处理 (16 <= i <= 31)



i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
g	1	6	11	0	5	10	15	4	9	14	3	8	13	2	7	12

```
G = G(b, c, d)

d = c

c =b

b = b + shift((a + G + k[i] + chunk[g]), s[i])

a = d
```

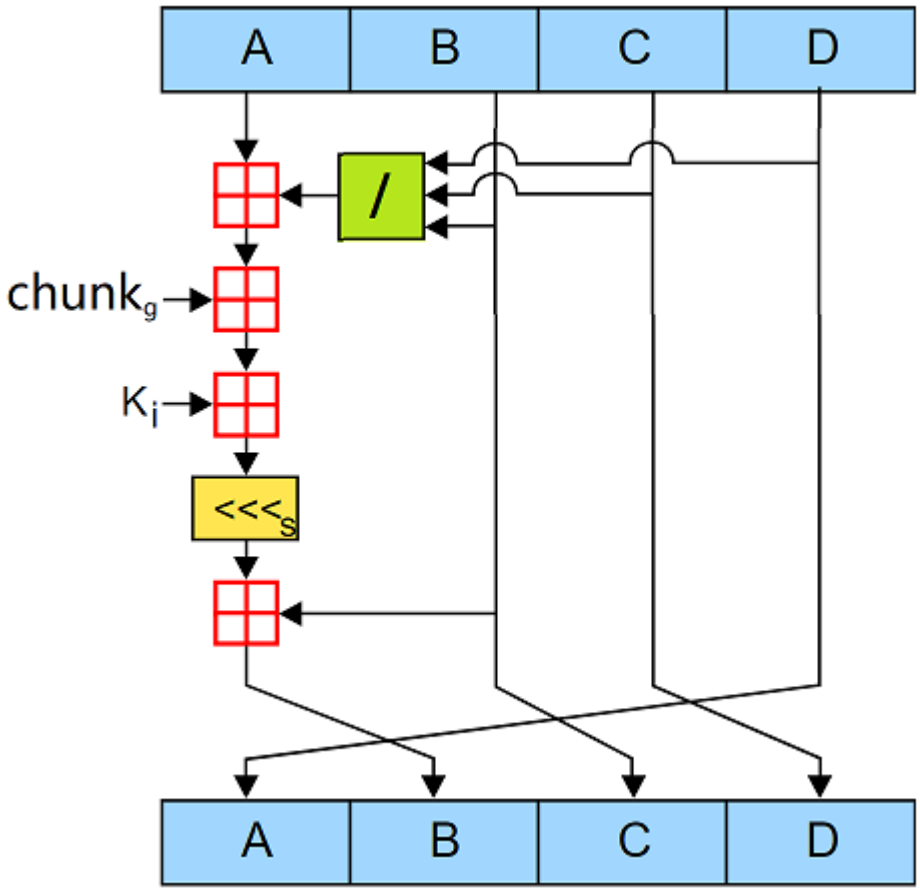
第三轮：H函数处理(32 <= i <= 47)



i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
g	5	8	11	14	1	4	7	10	13	0	3	6	9	12	15	2

```
H = H(b, c, d)

d = c

c =b

b = b + shift((a + H + k[i] + chunk[g]), s[i])

a = d
```

第四轮：I函数处理(48 <= i <= 63)



i	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
g	0	7	14	5	12	3	10	1	8	15	6	13	4	11	2	9

```
I = I(b, c, d)

d = c

c =b

b = b + shift((a + I + k[i] + chunk[g]), s[i])

a = d
```

图中的<<<s表示循环左移操作，每次左移的位数，在算法中也是和i一一对应的，也就是我们这的shift表示的含义。

```
/*
s[ 0..15] = { 7, 12, 17, 22,  7, 12, 17, 22,  7, 12, 17, 22,  7, 12, 17, 22 }
s[16..31] = { 5,  9, 14, 20,  5,  9, 14, 20,  5,  9, 14, 20,  5,  9, 14, 20 }
s[32..47] = { 4, 11, 16, 23,  4, 11, 16, 23,  4, 11, 16, 23,  4, 11, 16, 23 }
s[48..63] = { 6, 10, 15, 21,  6, 10, 15, 21,  6, 10, 15, 21,  6, 10, 15, 21 }
*/

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

从上图中可以看出g和i也存在一种确定的对应关系，关系如下：

```
if (0 <= i < 16)        g = i;
if (16 <= i < 32)       g = (5 * i + 1) % 16;
if (32 <= i < 48)       g = (3 * i + 5) % 16;
if(48 <= i < 63)        g = (7 * i) % 16;
```

一个chunk数据处理完之后，更新MD buffer的值A, B, C, D

```
A = a + A;
B = b + B;
C = c + C;
D = d + D;
```

到此为止，一个chunk数据就处理完了，接着处理下一个chunk数据。

雪崩效应（avalanche effect）

以上数据处理过程中的乱序处理，循环移位的改变，后3轮操作中输入数据顺序的改变，都是为了增加雪崩效应。

在密码学中，**雪崩效应**（**avalanche effect**）指加密算法的一种理想属性。雪崩效应是指当输入发生最微小的改变时，也会导致输出的不可区分性改变(一个小的因素导致意想不到的结果)。

2.5 摘要输出

这一步拼接4个buffer(A，B，C，D)中的摘要信息，以A中的低位字节开始，D的高位字节结束。最终的输出是128bit摘要信息的16进制表示，故最后输出一个32长度的摘要信息。

```
比如一个数，它的16进制表示为： 0x23456789，他所对应的8个摘要信息为从低位字节的89开始，高位字节的23结束，
即： 89674523
```

同学们可以用系统自带工具验证代码的正确性

```
用windows自带工具验证读取文件时MD5代码的正确性：

CertUtil -hashfile "文件路径" MD5

如果是linux可以用md5sum工具验证
```