**一.项目实现功能**

**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。**

**二.基本原理**

本项目要用到的是MD5签名判断方法，因此首先我们了解下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。因此我们便可以计算出指定文件的信息摘要MD5值，进而筛选删除相同的文件。

MD5的特点

* 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。

**三.基本过程**

* **扫描文件夹拿到文件夹下所有文件，**
* **对扫描后的文件进行MD5签名计算，**
* **删除重复MD5签名的文件，实现对磁盘文件的管理。**

**3.1  MD5算法步骤及实现**

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。**可以参考**<https://datatracker.ietf.org/doc/rfc1321/?include_text=1>

**文件---》MD5摘要信息大致可分为5个步骤：**

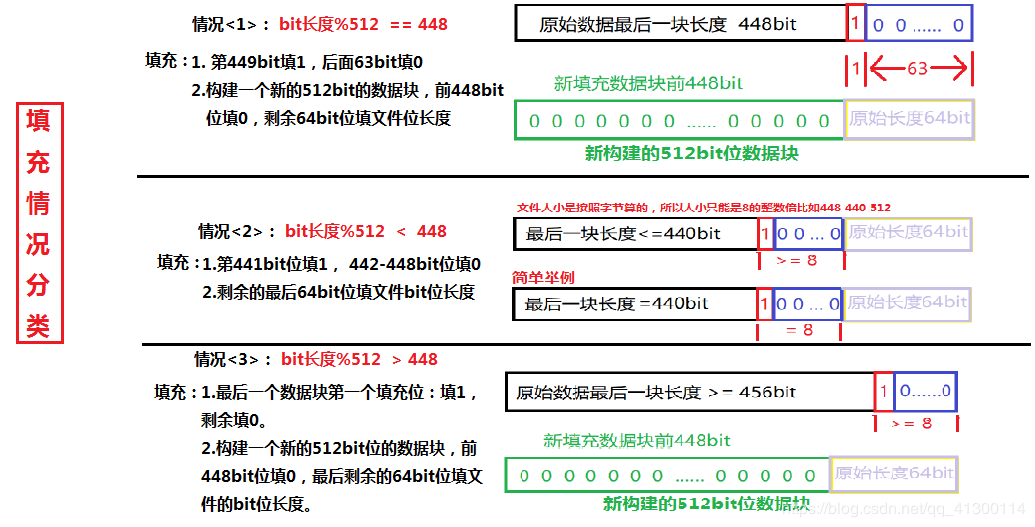
* 文件填充：填充 冗余信息填充位+文件bit长度
* 转换算法MD5摘要计算
* 摘要输出

**3.1.1 文件填充**

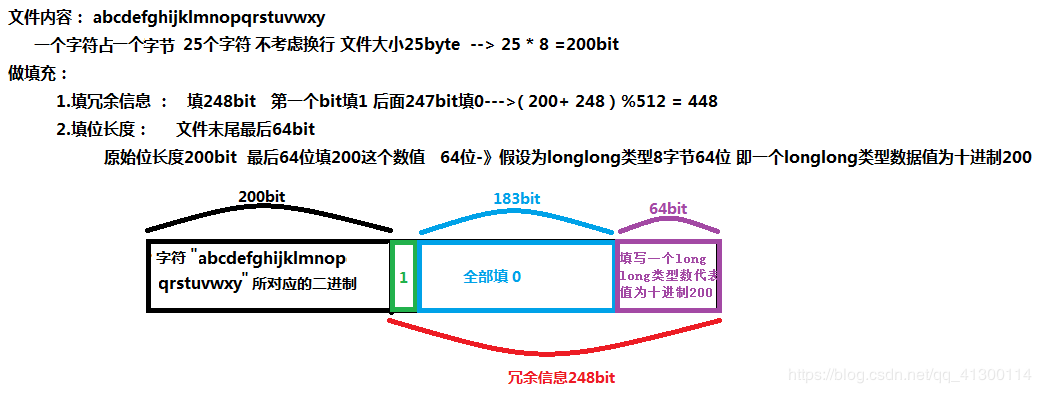
**填充规则：在信息的最尾部(不是每一块的尾部)要进行填充，使其最终的长度length(以bit为单位)满足length % 512 = 448，这一步始终要执行，在任何情况下都要进行填充操作（文件末尾）。**

**填充的内容：**

* **<1>冗余信息：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>文件位长度（原始长度）： 填充在最后一个数据块的最后64bit位。**

​

**举例：**



**3.1.2MD5摘要计算**

**3.1.2.1 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的变量固定值不可修改，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;**

**3.1.2.2按512位数据逐块处理输入信息【重点】**

通过一定算法改变前面四个固定变量的整形值，最终改变后的整形值即指纹信息摘要。

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。这四个函数为：

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));  
}

/\*

//每次数据左移的位置

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 }  
\*/

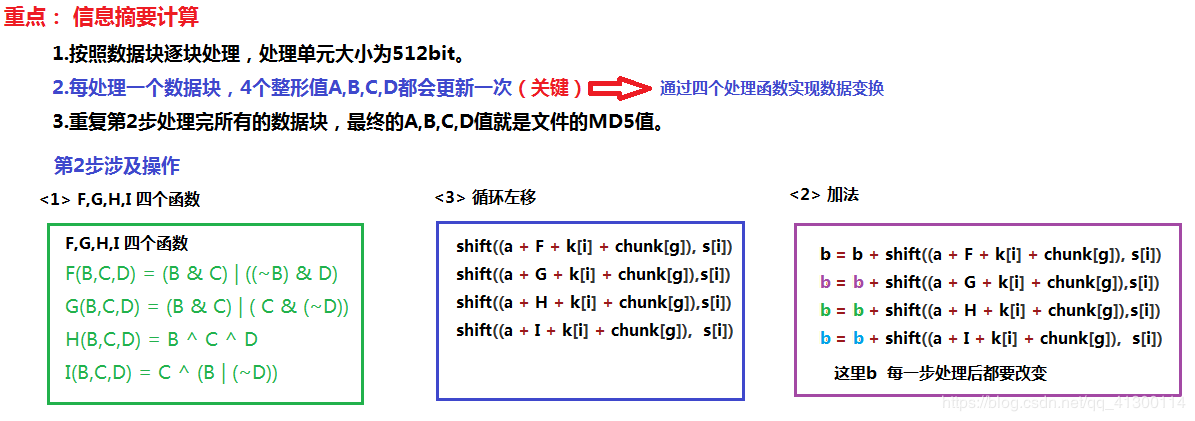
//g 和i关系相互对应

f (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;

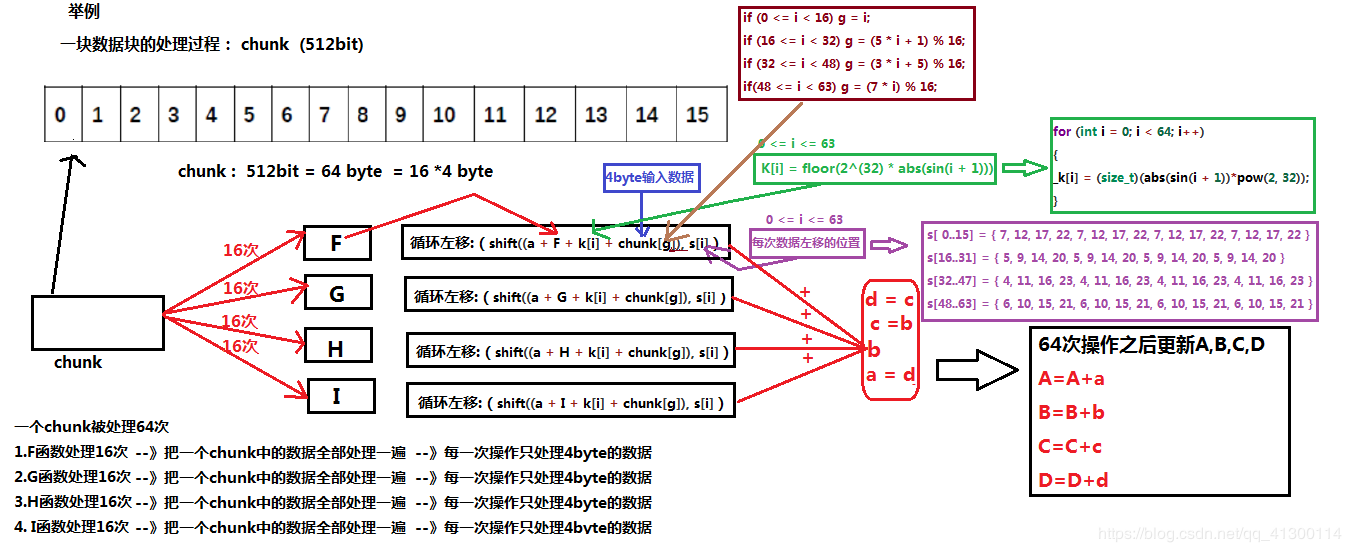
**为什么上面要进行这么多的乱序调整 比如移位？**

以上数据处理过程中的乱序处理，循环移位的改变，后面输入数据顺序的改变，都是为了增加雪崩效应。**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)

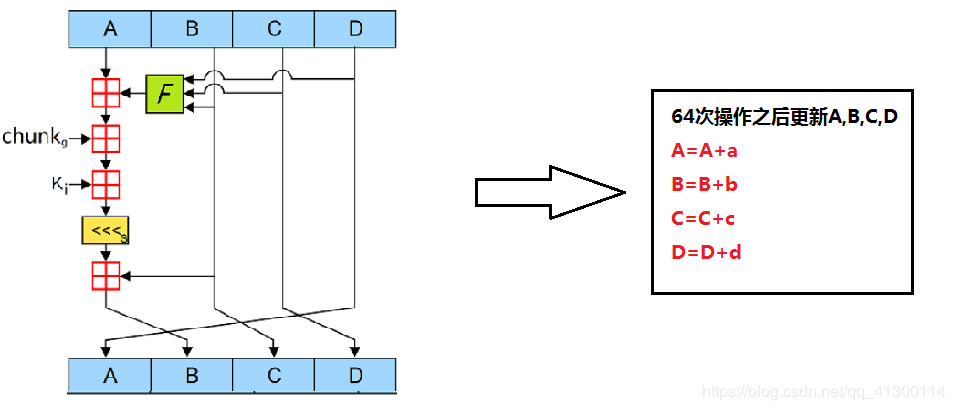
​

到此为止，一个chunk数据就处理完了，接着处理下一个chunk数据。

**图示一个数据块处理过程：**

​

**或者：**

​

**3.1.3 摘要值输出**

**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

**3.2获取当前文件夹下所有文件**

//path:目录

//搜索当前目录都有哪些文件

void searchDir(const std::string& path, std::unordered\_set<std::string>& subFiles)

{

std::string matchFile = path + "\\" + "\*.\*";

\_finddata\_t fileAttr;

long handle = \_findfirst(matchFile.c\_str(), &fileAttr);

if (handle == -1)//-1 没找到

{

perror("search failed! ");

std::cout << matchFile << std::endl;

return;

}

//找到了

do

{

if (fileAttr.attrib & \_A\_SUBDIR)//目录

{

if (strcmp(fileAttr.name, ".") != 0 && strcmp(fileAttr.name, "..") != 0)

{

//当前为目录，继续搜索

searchDir(path + "\\" + fileAttr.name, subFiles);

}

}

else//文件

{

subFiles.insert(path + "\\" + fileAttr.name);

}

} while (\_findnext(handle, &fileAttr) == 0);

\_findclose(handle);

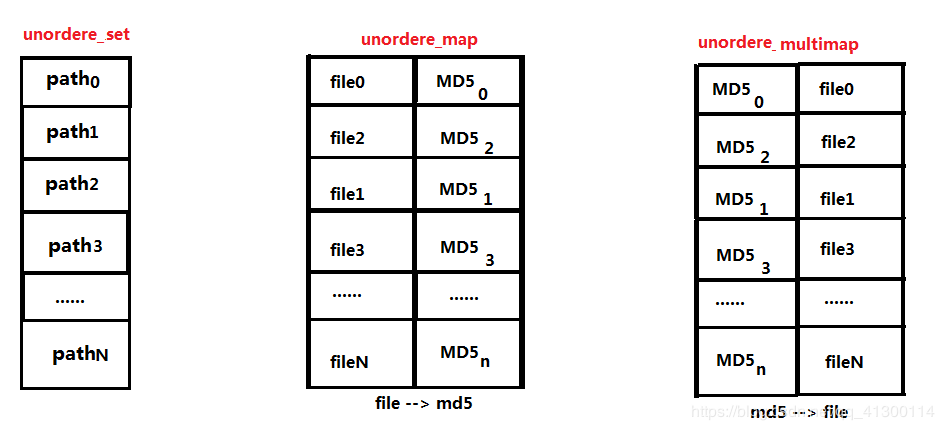
}

​

存放文件路径选用容器我们创建set/unordered\_set都可以用来存磁盘中所有放文件的路径（不选取 multiset是因为文件路径包含文件夹名称，所以path不可能重复）,在这里我们优先选用unordered\_set，不考虑顺序无序提高效率。

在保存文件到MD5值的映射关系选择unordered\_map，因为名称不会重复，所以不必要选择multimap，哈希表结构的unordered\_map可以提高效率。

在保存MD5值到文件的映射关系选择unordered\_multimap因为MD5值可能重复，所以要使用multimap，又使用unordered\_multimap提高效率。

​

**3.3管理重复文件（主要删除操作）**

下面所有删除包括按照**指定文件名删除、按照MD5值删除、删除所有文件副本、模糊匹配删除**，都要保证一个文件不存在副本。注意：在这里按照MD5值删除可以进行**代码复用**，具体遵循逻辑：查找到要删除的MD5值得所有文件为位置，再按照文件名进行删除即可。

//按照文件名删除,保留的是指定的文件

void FileManager::deleteByName(const std::string& name)

{

if (\_filestoMd5.count(name) == 0)

{

std::cout << name << " not exist! " << std::endl;

return;

}

//拿到文件对应的md5

std::string curMD5 = \_filestoMd5[name];

std::cout << name << "--->" << \_md5toFiles.count(curMD5) << std::endl;

auto pairIt = \_md5toFiles.equal\_range(curMD5);

auto curIt = pairIt.first;

int count = 0;

while (curIt != pairIt.second)

{

if (curIt->second != name)

{

\_filestoMd5.erase(curIt->second);

\_files.erase(curIt->second);

deleteFile(curIt->second.c\_str());

++count;

}

++curIt;

}

//更新

curIt = pairIt.first;

while (curIt != pairIt.second)

{

if (curIt->second != name)

{

//key --->MD5

\_md5toFiles.erase(curIt);

pairIt = \_md5toFiles.equal\_range(curMD5);

curIt = pairIt.first;

}

++curIt;

}

std::cout << "delete files :" << count << std::endl;

}

//按照MD5值进行删除 //两个版本

void FileManager::deleteByMD5(const std::string& md5)

{

//md5 --> files

if (\_md5toFiles.count(md5) == 0)//不存在

{

std::cout<<md5 << " not exist! " << std::endl;

return;

}

//删除是需要保留一份，保留第一个文件

auto paitIt = \_md5toFiles.equal\_range(md5);

std::cout << md5 << "--->" <<\_md5toFiles.count(md5) << std::endl;

auto curIt = paitIt.first;

++curIt;

int count = 0;

while (curIt != paitIt.second)

{

\_files.erase(curIt->second);

\_filestoMd5.erase(curIt->second);

//\_md5toFiles.erase(curIt);//迭代器失效 容易出错

//文件从此磁盘中删除

deleteFile(curIt->second.c\_str());

++curIt;

++count;

}

//更新md5 ---> files

curIt = paitIt.first;

++curIt;

//erase(first,last) -- >删除区域值【first,last)

\_md5toFiles.erase(curIt, paitIt.second);

std::cout << "delete files :" << count<< std::endl;

}

void FileManager::deleteByMD5V2(const std::string& md5)

{

//md5 --> files

if (\_md5toFiles.count(md5) == 0)//不存在

{

std::cout << md5 << " not exist! " << std::endl;

return;

}

auto it = \_md5toFiles.find(md5);

deleteByName(it->second);

}

//删除所有文件的副本

//每个重复文件只保留一个

void FileManager::deleteAllCopy()

{

//先拿MD5集合

std::unordered\_set<std::string> md5set;

//找出所有的md5值

for (const auto& p : \_md5toFiles)

{

md5set.insert(p.first);

}

for (const auto& md5 : md5set)

{

deleteByMD5(md5);

}

}

//模糊删除： 删除所有模糊匹配matchName所有文件的副本

void FileManager::deleteByMatchName(const std::string& matchName)

{

//遍历所有的文件

std::unordered\_set<std::string> allFiles;

for (const auto& f : \_files)

{

if (f.find(matchName)!= std::string::npos)

{

allFiles.insert(f);

}

}

//按照文件名删除

for (const auto& f:allFiles)

{

if (\_filestoMd5.count(f) != 0)

{

deleteByName(f);

}

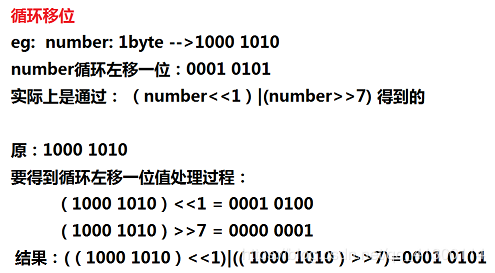
}

}

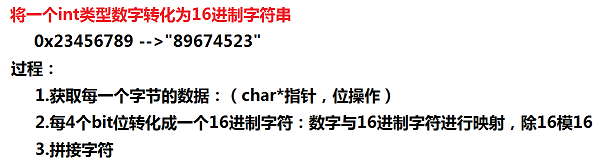
**四、一些注意点**

1**.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)。就这个项目而言:为了增加雪崩效应，我们采用了多次循环移位等措施，都是为了结果产生巨大的差异。

2.**进行MD5计算时的循环移位操作**



**3.**在进行摘要输出时，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**，**因此会涉及以下操作：**将int类型数字转化成16进制的字符串**



**参考文档：**

* <https://datatracker.ietf.org/doc/rfc1321/?include_text=1>
* <https://www.rfc-editor.org/rfc/rfc1321.txt>