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：

文件内容：abcdefghijklmnopqrstuvwxy

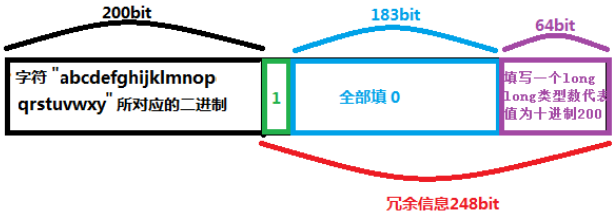
一个字符占一个字节 25个字符 不考虑换行 文件大小25byte --> 25 * 8 =200bit

做填充：

1.填冗余信息 ： 填248bit 第一个bit填1 后面247bit填0--->(200+ 248) %512 = 448

2.填位长度： 文件末尾最后64bit

原始位长度200bit 最后64位填200这个数值 64位-》假设为longlong类型8字节64位 即一个longlong类型数据值为十进制200



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)

重点： 信息摘要计算

1.按照数据块逐块处理，处理单元大小为512bit。

2.每处理一个数据块，4个整形值A,B,C,D都会更新一次（关键） ⇨ 通过四个处理函数实现数据变换

3.重复第2步处理完所有的数据块，最终的A,B,C,D值就是文件的MD5值。

第2步涉及操作

<1> F,G,H,I 四个函数

F,G,H,I 四个函数

F(B,C,D) = (B & C) | ((~B) & D)

G(B,C,D) = (B & C) | (C & (~D))

H(B,C,D) = B ^ C ^ D

I(B,C,D) = C ^ (B | (~D))

<3> 循环左移

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

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

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

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

<2> 加法

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

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

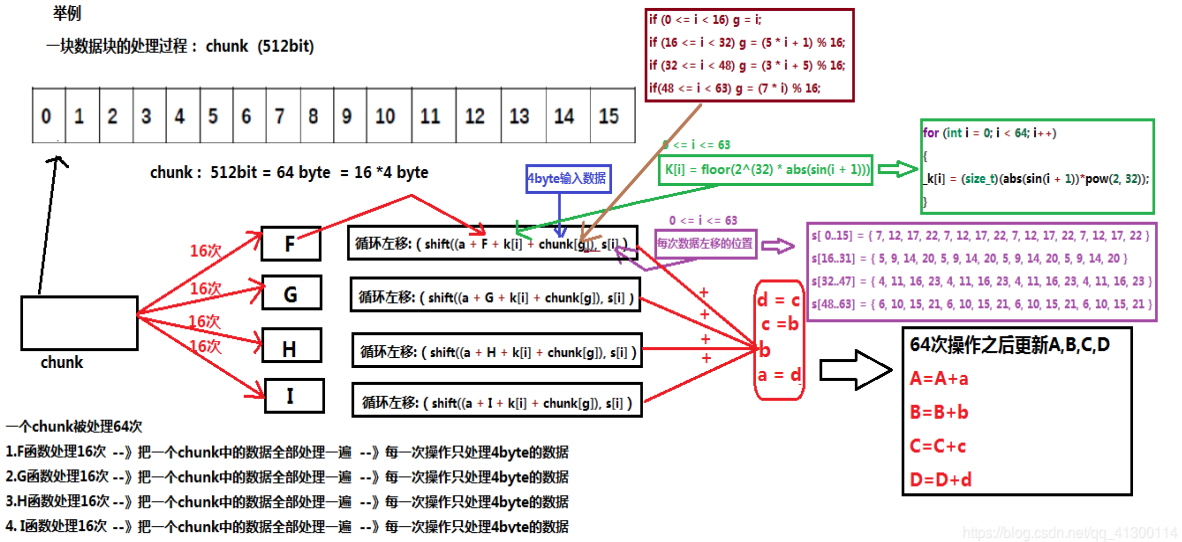
b = b + shift((a + H + k[i] + chunk[g]),s[i])

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

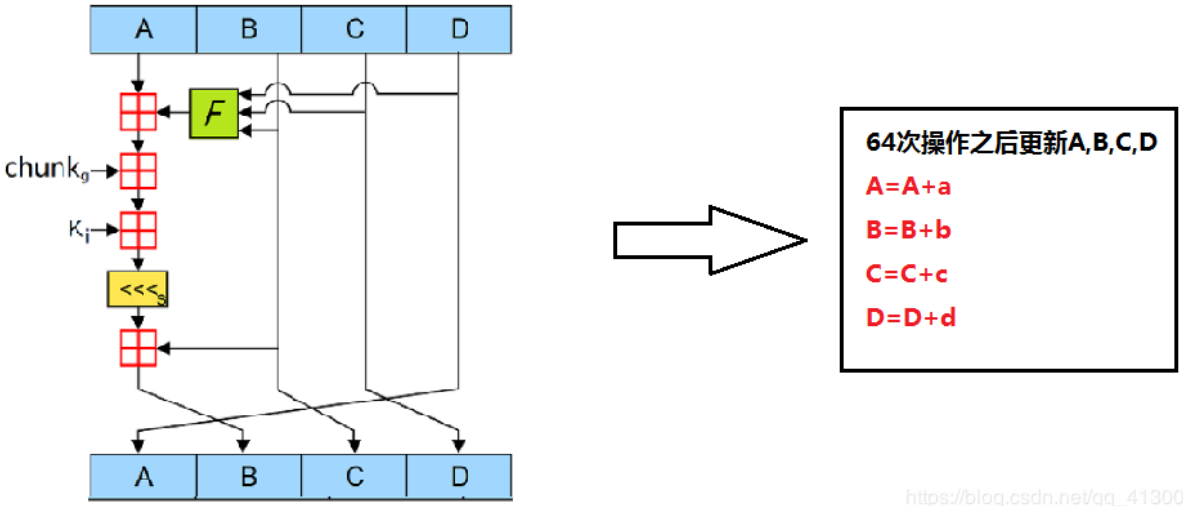
这里b 每一步处理后都要改变

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

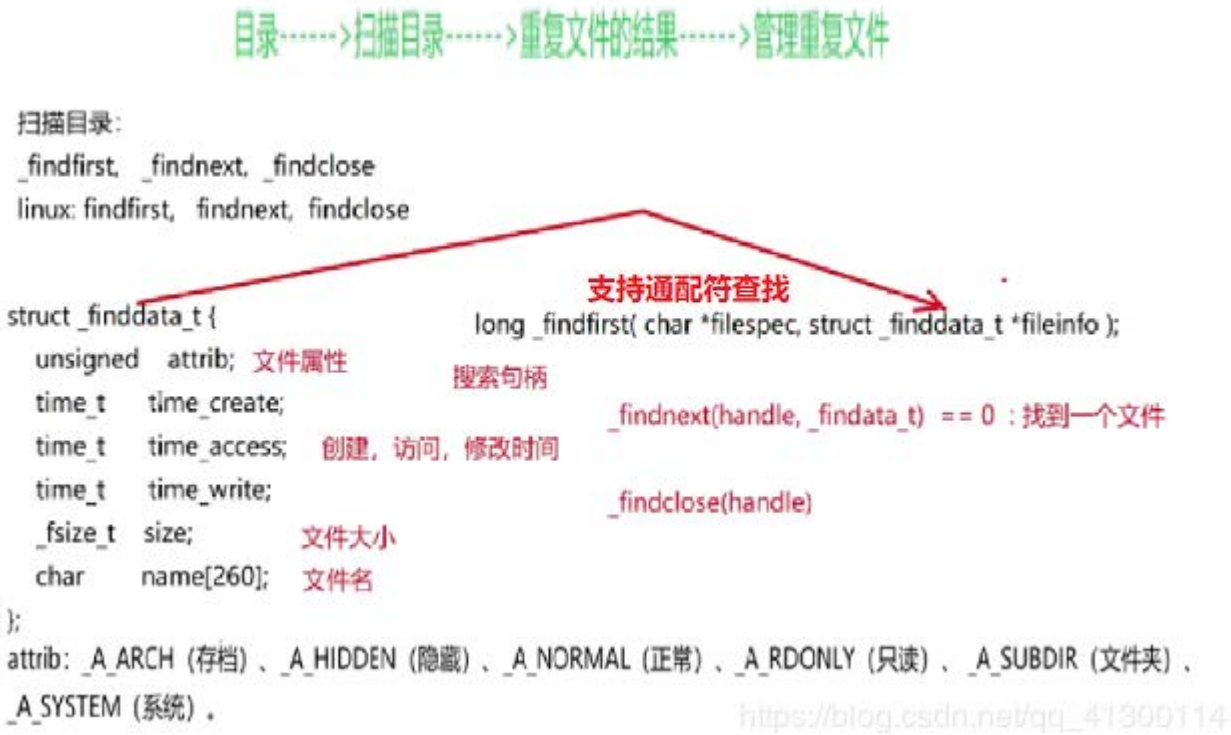
这一步拼接 4 个 buffer(A，B，C，D)中的摘要信息，以 A 中的低位字节开始，D 的高位字节结束。最终的输出是 128bit 摘要信息的 16 进制表示，故最后输出一个 32 长度的摘要信息。

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;
	}
```

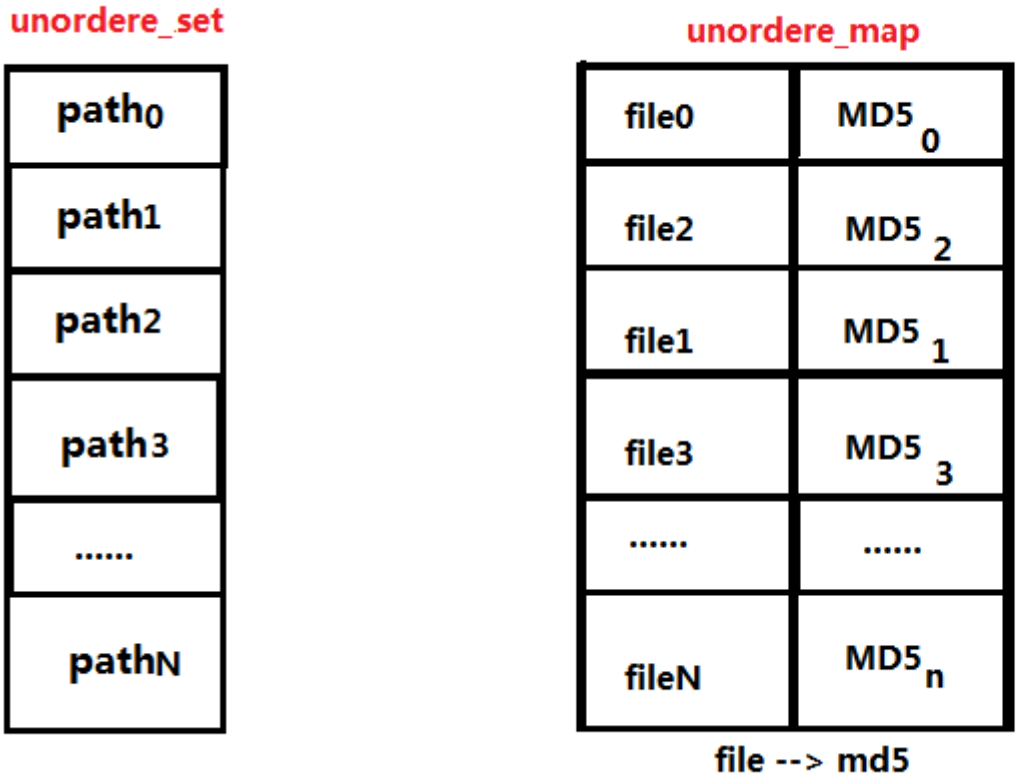
```
        //找到了

        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**

循环移位

eg: number: 1byte -->1000 1010

number循环左移一位：0001 0101

实际上是通过：（number<<1）|(number>>7) 得到的

原：1000 1010

要得到循环左移一位值处理过程：

（1000 1010）<<1 = 0001 0100

（1000 1010）>>7 = 0000 0001

结果：（（1000 1010）<<1)|((1000 1010）>>7)=0001 0101

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**

将一个int类型数字转化为16进制字符串

0x23456789 -->"89674523"

过程：

1.获取每一个字节的数据：（char*指针，位操作）

2.每4个bit位转化成一个16进制字符：数字与16进制字符进行映射，除16模16

3.拼接字符

参考文档：

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