目录

[**一.** **题目要求与分析** 1](#_Toc97058508)

[**1.1 题目要求** 1](#_Toc97058509)

[**1.2 任务分析** 1](#_Toc97058510)

[**1.3 设计规划** 1](#_Toc97058511)

[**二.** **核心算法** 2](#_Toc97058512)

[**2.1 算法分析** 2](#_Toc97058513)

[**2.2 代码实现** 4](#_Toc97058514)

[**2.3 性能分析** 4](#_Toc97058515)

[**三.** **模块设计** 5](#_Toc97058516)

[**3.1 基本信息与设计思路** 5](#_Toc97058517)

[**3.2 参数、底层函数与工具函数** 6](#_Toc97058518)

[**3.3 核心函数** 8](#_Toc97058519)

[**3.4 公有函数接口** 11](#_Toc97058520)

[**四.** **模块分析** 12](#_Toc97058521)

[**4.1 分析环境、内容与方法** 12](#_Toc97058522)

[**4.2 性能分析：时间** 12](#_Toc97058523)

[**4.3 性能分析：内存占用** 15](#_Toc97058524)

[**4.4 压缩效率分析** 17](#_Toc97058525)

[**五.** **Demo设计** 20](#_Toc97058526)

[**5.1 Demo说明** 20](#_Toc97058527)

[**六.** **改进方向** 21](#_Toc97058528)

[**6.1 性能改进：多线程压缩** 21](#_Toc97058529)

[**6.2 效率改进：基于位的操作** 22](#_Toc97058530)

[**6.3 功能改进：更多功能** 22](#_Toc97058531)

[**附录A lzwf标准文件格式** 22](#_Toc97058532)

[**附录B 原始数据：性能测试** 23](#_Toc97058533)

[**附录C 原始数据：效率测试** 24](#_Toc97058534)

图片目录

图 1模块设置思路 6

图 2单个文件压缩性能（时间） 13

图 3单个文件解压性能（时间） 13

图 4多个文件压缩性能（时间） 14

图 5多个文件解压性能（时间） 14

图 6单个文件压缩性能（内存） 16

图 7单个文件压缩性能（内存） 16

图 8不同设置下性能对比（内存） 17

图 9压缩率总览（mapCodeLen=4） 18

图 10压缩率分览（mapCodeLen=4） 18

图 11 压缩率总览（mapCodeLen=2） 19

图 12压缩率分览（mapCodeLen=2） 19

图 13帮助文本 20

图 14首次打开程序时的初始化结果 21

图 15 lzwf文件结构 22

代码目录

代码段 1压缩核心代码（算法） 4

代码段 2解压核心代码（算法） 4

代码段 3字典刷新函数 6

代码段 4初始字典构造函数 7

代码段 5文件名获取函数 7

代码段 6压缩核心外围函数 8

代码段 7解压核心外围函数 9

代码段 8压缩核心代码（模块） 10

代码段 9解压核心代码（模块） 11

1. **题目要求与分析**

**1.1 题目要求**

本次设计的题目要求共包括三点：①在一个文本文档上实现LZW的压缩和解压操作；②对LZW算法的设计思路、时间复杂度和空间复杂度进行分析；③使用LZW算法对bmp图片进行压缩。

题目的三个要求都是围绕LZW算法展开且层层递进的。

**1.2 任务分析**

显然，无论是设计任务还是分析任务，整个题目的核心都在于LZW算法。考虑到要求①和要求③只存在文件处理方式上的区别（①要求从字符层面上对文件进行压缩和解压操作，③则要求从二进制层面上对文件进行压缩和解压操作），核心压缩操作层面基本相同，我们更倾向于设计一个核心类，再用不同的方式（主要体现在文件读写的差异上）使用这个类，从而实现两个要求。

进一步分析，由于对文本文档的操作也可以在二进制层面进行，我们实际上完全可以将要求①和要求③合并成一个要求，即利用LZW算法对各种类型的文件进行压缩和解压。

对于要求②，这实际上属于学习和使用任何一个算法过程中都需要解决的问题，因此，这部分的内容将会直接在报告的相应部分给出。

**1.3 设计规划**

根据上述的分析，加之LZW算法属于应用较为广泛的算法之一，我们在此次任务中**不倾向于**设计一个依赖于任何系统特性的、完整的、能够实现功能的程序，而是**倾向于**给出一个不依赖环境的、高度可移植的模块。

显然，为了能够正常对模块的功能进行演示，我们需要设计一个简单的Demo，因此，我们将这次的设计任务分为两个部分，即核心模块的设计和Demo的设计。

由于Demo只起到简单的演示作用，我们将对Demo进行简化处理，这意味着这个Demo没有可视化界面和较强的实用性。

另外，虽然模块是具有高度可移植性且不依赖于环境的，但Demo并不具备这样的特性，它是高度依赖于系统环境的。

1. **核心算法** 时间：2022-1

**2.1 算法分析**

LZW算法，即“串表压缩算法”，其核心在于串表的建立。相较于其他的一些压缩算法，LZW算法具有以下几个特点：

1. LZW算法得到的串表（即字典）是自解释的，这意味着我们不需要在压缩后的文件中保存这个串表；
2. 通过LZW算法得到的字典是低复用的，即对于字典中的所有编码而言，其在压缩后的文件中出现的次数都是不高的（通常不会大于256次）；
3. LZW算法得到的映射是前缀存在的，即对于一个字符串s和一个字符a，以及两个映射后的编码C1和C2，如果存在映射(s+a)→C1，那么映射s→C2一定存在且C2<C1，正是这个特性保证了LZW的字典是自解释的。

上述的几个特点的共同作用决定了LZW算法在压缩时的高效性，这种高效性在具有较多重复数据文件的压缩中尤为明显。

LZW算法的压缩过程十分简单，可以看作是下面描述的步骤的重复：

步骤开始时，我们拥有一个数据块（字符串）s，和从文件中读取的一个数据元c（字符）。

首先判断数据块(s+c)是否位于字典中：

1. 如果(s+c)位于字典中，则使s=(s+c)；
2. 如果(s+c)不位于字典中，那么s一定位于字典中，设s映射为m，则输出m，将(s+c)添加到字典中，最后使s=c。

在上述过程中，我们允许s为空。另外，在执行包括上述步骤的循环前，我们需要将所有由单个数据元构成的数据块（具有单字符的字符串）添加到字典中。

另一方面，当压缩执行到文件尾部时，我们需要固定执行步骤中的b项，这样才能使最后一个数据块完成压缩。

与之相对的，LZW算法的解压过程也可以归约成为一个步骤：

步骤开始时，我们拥有一个数据块（字符串）last，和从文件中读取的一个编码m。

首先判断编码m是否位于字典中。

1. 如果m位于字典中，且由数据块s映射而来，则输出数据块s，同时将(last+s[0])添加到字典中，最后使last=s；
2. 如果m不位于字典中，则首先使last=(last+last[0])，然后输出数据块last，并将last添加到字典中。

需要注意的是，在进行第一步之前我们已经对初始的字典进行了构造，因此第一步解压一定会执行a步骤，但由于相同的原因，这次执行我们不进行添加字典的操作。

相较于压缩过程，解压过程并不十分直观。想要更深入地理解解压过程，我们需要解决以下两个问题：（1）数据块last代表什么；（2）为什么当编码m不在字典中时输出(last+last[0])。

对于第一个问题，我们可以通过从头开始进行推演来解决。

现在假设压缩时的数据元只有字符a与b，且使a→1，b→2，则对于数据块“abab”，压缩后得到的结果为{1,2,3}。

现在对其进行解压操作，初始时，last为空，字典中含有a与b的映射。

1. 现在读取到编码1，我们能够从字典中找到，因此执行a步骤，此时输出a，last=“a”，根据特殊规则，我们这一步不进行写字典操作；
2. 现在读取到编码2，我们仍能从字典中找到，同样执行a步骤，此时输出b，添加新的映射ab→3，last=“b”；
3. 现在读取到编码3，我们同样能在字典中找到，执行a步骤，此时输出ab，添加新的映射ba→4，last=“ab”。

不难看出，last在本次循环开始时代表上一轮循环时读取到的编码所对应的数据块，本轮循环结束时代表本轮循坏读取的编码所对应的数据块。

对于第二个问题，我们可以通过相同的方法，利用一个特殊的例子来解决。

这个例子中的初始假设与上一实例相同，现在对数据块“aaa”进行压缩，压缩后得到的结果为{1,3}。

现在对其进行解压操作，初始时，last为空，字典中含有a与b的映射。

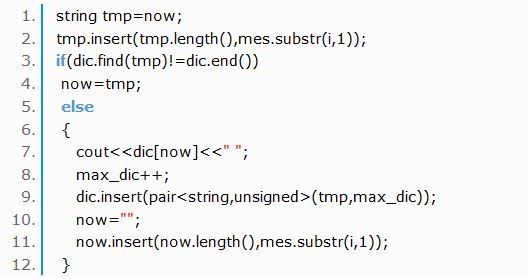
1. 现在读取到编码1，我们能够从字典中找到，因此执行a步骤，此时输出a，last=“a”，根据特殊规则，我们这一步不进行写字典操作；
2. 现在读取到编码3，我们不能在字典中找到，执行b步骤，首先我们使last=“aa”，然后输出last并将其添加到字典中。

通过例子不难发现，由于压缩过程和解压过程存在错位同步关系，即解压总比压缩“快一步”（压缩时，算法会先输出数据块s的映射m，再将s+c添加到字典中；解压时，算法会先输出n对应的数据块r，再将last+r[0]添加到字典中，这里的last与上文含义相同），因此，当且仅当压缩时出现“上一步添加的新字典项在这一步接着被使用”这种情况时，其对应的解压过程才会出现这一步读取到的编码没有出现在当前的字典中这一情况。

接下来的问题是，压缩在什么时候会出现上述情况。不难看出，当上一步添加的新字典项头数据元与尾数据元相同（即s[0]=s[s.length()-1]），同时这一步的数据块与上一步一致时会出现这种情况，例如“ababaaba”和上述例子中的“aaa”都是这种情况。

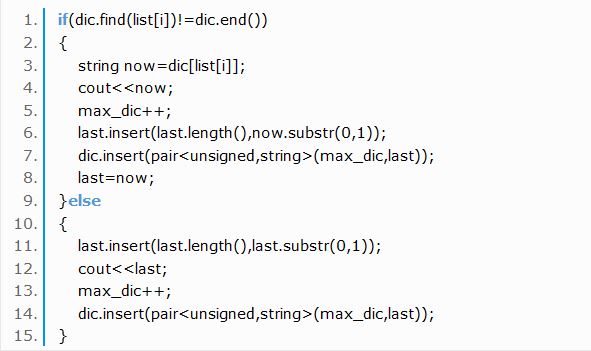
**2.2 代码实现**

根据上述分析，我们可以很容易地得到LZW算法压缩与解压的核心代码，为了使核心代码更加直观，我们将上文中所描述的步骤单独提取并进行实现，即下面的代码都是不包含外层的循环的。



代码段 1压缩核心代码（算法）

LZW算法的压缩操作核心代码如下：



代码段 2解压核心代码（算法）

LZW算法的解压操作核心代码如下：

需要注意的是，上面给出的代码是最为基础的代码，而非设计模块时所使用的代码，模块中的相应代码会在后文对模块的介绍中给出。

**2.3 性能分析**

由于LZW算法的时间复杂度和空间复杂度的分析较为复杂，因此这次的分析我们主要给出定性描述而非定量的描述。这些分析只是基于算法本身的，对于设计结果（模块）的性能分析，我们会在后文中进行详细的定量描述。

在上一节的代码中，我们没有给出字典的定义，实际上，无论是压缩还是解压字典，我们均使用了STL中的std::map容器作为字典的存储形式。由于std::map容器是通过红黑树实现的，因此对一个大小为n的字典进行一次查询的时间复杂度为O(lg n)，这意味着对于LZW算法而言，其压缩操作和解压操作的时间复杂度在数量级上是相同的。

不过，从代码中我们可以看出，LZW算法在压缩时的字典是由字符串到整数的映射（键值为字符串），而在解压时的字典则是由整数到字符串的映射（键值为整数），由于对字符串的比较操作一般而言是慢于整数的，因此在时间复杂度的系数上，解压操作是略低于压缩操作的。

另一方面，LZW算法中的压缩过程是依据数据元（字符）数量进行循环的，而其解压过程则是依据输出的编码数量进行循环的，因此对于一个原文件-压缩文件对而言，在理论上其压缩操作是略慢于解压操作的。

LZW算法的空间复杂度主要取决于字典，由于解压和压缩时构造出的字典是完全对称的，因此两个过程的空间复杂度无论是在数量级上还是系数上都是几乎相同的。另一方面，由于即使是规模相同的文件，字典的大小也会因为文件内容的不同出现较大的波动，LZW算法的空间复杂度受到输入（目标文件）的影响是较大的。

总的来说，从理论上进行分析，LZW算法的时间复杂度和空间复杂度都是较为优秀的，并且两者均依赖于字典的设计，因此，一个优秀的字典结构对于LZW算法性能的提升是很大的。

1. **模块设计** 时间：2022-1，2022-2

**3.1 基本信息与设计思路**

这个模块分为两个部分，其中被命名为“LzwCore”的类（在lzw\_core.h中被定义，在lzw\_core.cpp中被实现）是整个模块的核心，而在文件lzw\_exception.h中，我们定义了LzwCore类在运行过程中可能出现的一些异常。

不过需要注意的是，并不是所有可能出现的异常都被包含在了lzw\_exception.h文件中，这个文件中只包含了几个具有后效性（即出现之后不会立即显现效果，但是会在后续的操作中引发严重的异常）的异常，而对于其他程序能够立刻抛出的标准异常并没有在文件中进行更为细致的定义。

对于LzwCore类而言，其字典同样是采用了std::map作为容器，而除了字典和函数外，我们还为其设计了四个参数。这四个参数中，有一个参数用于保证其跨系统特性，而其余的三个参数则保证了模块在使用时具有一定的自由度。

在设计模块的核心部分（LzwCore类）的函数时，我们大体上采用了自底而上的设计思路，首先对底层的私有操作函数（例如添加新的字典项、构造数据块）进行了设计，然后又设计了一些通用的私有函数工具（例如获取文件名、构造文件名），随后对私有的核心函数（压缩核心函数和解压核心函数）进行了实现，最后对公有函数（包括用于实现功能的接口和调整参数的接口）进行了实现。上述函数间的关系与设计顺序如图所示：

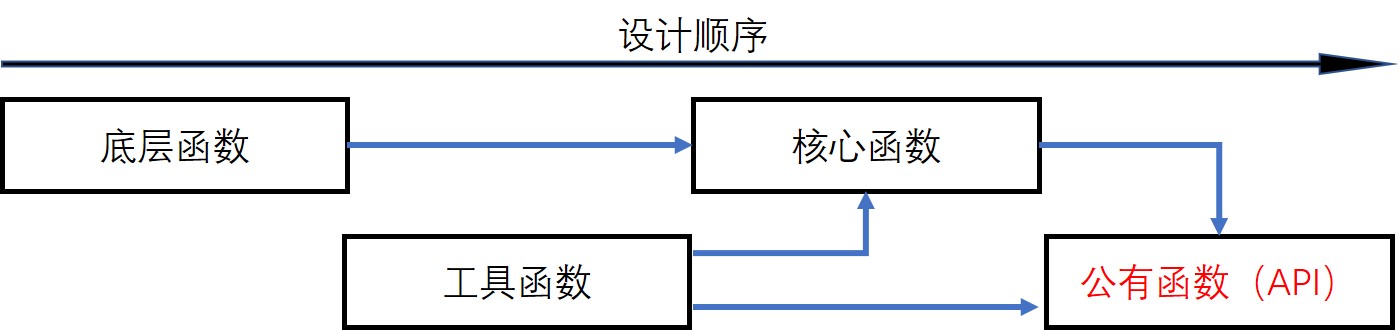


图 1模块设置思路

下面我们将大致按照这个顺序对其进行简单的说明，更为详细的信息可以通过附件中的代码文档查看。

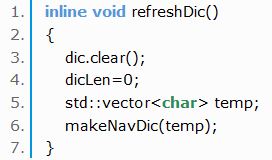
**3.2 参数、底层函数与工具函数**

LzwCore类所涉及的四个参数的基本信息如下：

|  |  |  |
| --- | --- | --- |
| 变量名 | 作用 | 备注 |
| fileTag | 定义路径分隔符 | 静态，跨系统时使用 |
| navDataLen | 压缩时读取原始数据的宽度 | 单位为字节，最大小于8 |
| mapCodeLen | 压缩时产生数据的宽度 | 单位为字节，最大为8 |
| docSizeLen | 用于记录文件大小的数据的宽度 | 单位为字节，最大为8 |

LzwCore类的底层函数实际上可以更加细致地分为几层，为了缩短篇幅，我们只选取其中较具有代表性的函数进行说明。

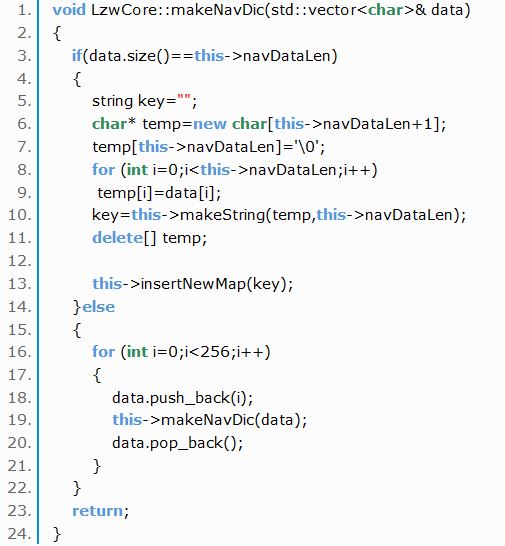
所有底层函数中最具有代表性的函数是用于构造初始字典的两个函数makeNavDic和makeNavTDic，但是，这两个函数并不能直接进行使用。实际上，在此之上这两个函数还进行了进一步包装，包装得到的函数即为字典刷新函数refresh(T)Dic：



代码段 3字典刷新函数

另一个刷新函数refreshTDic只与其在第6行有所不同，因此不再给出

makeNavDic函数的代码则如下所示：

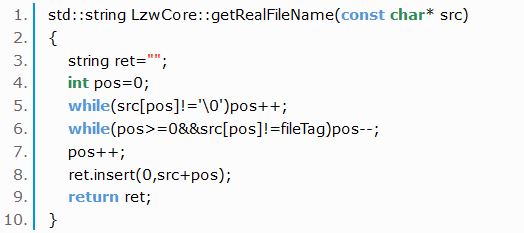


代码段 4初始字典构造函数

makeNavTDic函数与其在映射方向上相反（code→data），其余部分几乎完全相同，这里不再给出代码。

显然，这两个函数的主要作用是构造初始字典中的字典项。虽然大多数情况下，我们将一个字节的数据作为一个数据元进行压缩，但是为了支持多字节数据元压缩，我们对这两个函数进行递归式的设计。经过这样的设计，理论上我们可以在压缩时将数据元的大小设置成任意正整数，但是为了平衡模块的稳定性以及效率，我们只允许将数据元的大小调整成为一个不大于8的正整数（参见上文中对于navDataLen的描述）。

LzwCore类的工具函数主要用于完成几个辅助性质的功能，例如获取文件的文件名或路径。其中，从字符串中获取文件名（不含路径）的代码如下：



代码段 5文件名获取函数

从这个函数里，我们可以清晰地看出参数fileTag的作用，实际上，这个函数在某种程度上也是依赖于这个参数运行的。

除此之外，LzwCore还具有很多较为简单的工具函数，例如调试用函数（debugTool）、文件头生成函数（makeFileHeader）。这些函数的实现与功能都较为简单，这里不再进行详细的介绍，完整的功能说明请查看附件中的代码文档。

**3.3 核心函数**

模块的核心函数设计分为两层，我们在这里将其简单地称为“内层核心”和“外层核心”。外层核心主要包括算法中的循环层及其辅助操作、文件操作以及特殊状态的处理。

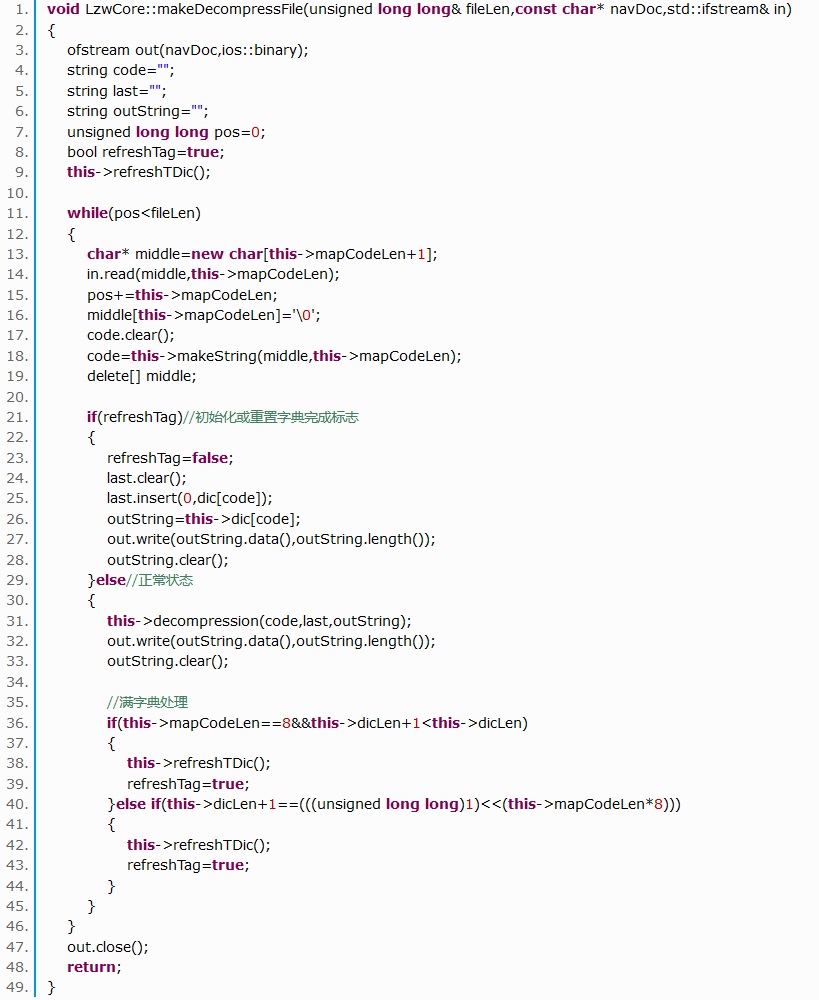
压缩时的外层函数代码如下：



代码段 6压缩核心外围函数

代码的第15行对压缩核心函数compression进行了调用。

解压时的外层函数代码如下：



代码段 7解压核心外围函数

代码在第31行对解压核心函数进行了调用。

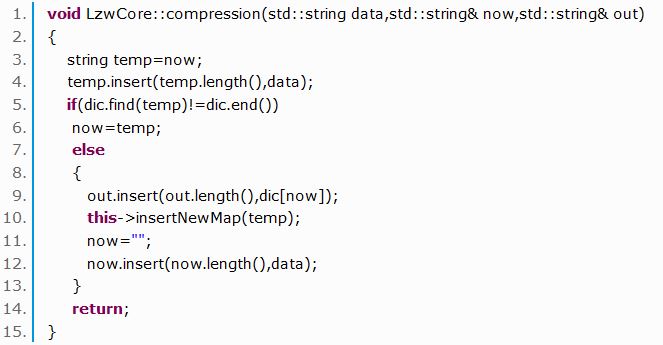
相较而言，二者的主要区别在于对操作结果的处理。由于核心函数只能进行单个文件的操作，而对于外部的调用而言，压缩过程中可能涉及到多个文件的压缩（多对一）以及在原有压缩文件上的追加，一次调用的结果可能并不是我们想要得到的最终结果，因此，我们使用临时文件的形式对操作结果进行回返，让外部调用决定如何处理本次压缩的结果。需要注意的是，C++的临时文件具有关闭即删除的特点，因此在要进行的操作完成之前，临时文件不能被关闭。

与之相对的，解压过程不存在这个问题。虽然调用一次解压同样只能解压出单个文件，但是每个文件都不需要进行进一步的处理，因此，我们可以直接将操作的结果作为最终的文件。

另外，我们能够注意到，两个函数中均含有一段用于重置字典的代码。考虑到操作过程中可能需要处理非常大的文件，而过大的文件可能会导致满字典情况的出现，因此，我们在字典长度达到极限时，对字典进行重置操作，这样可以使得模块有能力处理更大的文件。

内层核心函数相较于外层较为简洁，这是因为内层函数只进行单步压缩（解压）的操作，不需要关心其他问题。

压缩内层核心函数代码如下：



代码段 8压缩核心代码（模块）

总的来说，模块中的这一段代码无论是在思想上还是结构上，都与上文中算法部分给出的代码基本相同，唯一需要注意的是函数的三个参数所代表的含义：第一个字符串data代表这一步读取到的元数据块（字符串的长度在不计算尾部结束符时长度总是等于参数navDataLen）；第二个字符串now代表目前已被读取但还未进行编码输出的数据块，函数调用后会根据运行结果进行追加（本次执行不进行编码输出）或刷新（本次执行进行编码输出）；第三个字符串out代表压缩得到的结果，如果本次执行进行了编码输出则进行追加，否则不对其进行操作。

对于这个函数更为详细的信息请查阅代码文档。

解压内层核心函数代码如下：



代码段 9解压核心代码（模块）

与上面提到的压缩内层核心函数相似，这个函数主要也是需要注意三个参数的含义：字符串code代表当前读取到的一个编码；字符串last表示上一轮循环时输出的数据块；字符串out表示输出的结果，由于每轮都会被外层函数抓取并清空，因此可以看作是本轮的输出结果。

以上的四个函数构成了整个LzwCore类的核心，其他所有与压缩和解压相关的公有函数接口都会直接使用两个外层核心函数。

**3.4 公有函数接口**

除构造函数和析构函数外，LzwCore类的公有函数接口可以分为两类，一类是用于获取和调整参数的get/set函数对，另一类是用于进行压缩与解压的函数。

对于第一类函数我们不再赘述，唯一需要注意的是，我们并没有提供静态参数fileTag的get函数，因为我们认为这是没有必要的——这个参数与系统密切相关，任何人在使用这个模块前都应该清楚地知道这个参数应该被设置成的内容并进行相应的设置，在此之后，使用者便不应该对这个参数进行任何的修改。

对于第二类函数，我们主要实现了四个基础的功能：（1）单个文件的压缩；（2）多个文件的压缩；（3）在已有压缩文件后追加新的压缩内容；（4）压缩文件的解压。

这些公有函数主要进行了获取或设置压缩文件参数、确定压缩文件中各个子文件的位置以及压缩后临时文件的合并三项工作，这些工作是依赖于私有工具函数的。

除了这些辅助性的工作，模块最为重要的压缩和解压工作均是这些函数通过调用私有的核心函数完成的，这些函数本身并不包含核心的操作，因此，我们可以很方便的对这些接口进行扩展而不必担心对模块造成毁灭性的破坏。

不过，这些函数接口之间并不是完全独立的。为了降低冗余代码的占比，这其中存在一些相互的调用。例如，不提供路径的压缩函数会将缺省路径作为参数，然后调用提供路径的压缩函数来完成压缩工作。

另外，公有函数接口的行为决定了模块需要一种新的文件格式来规范压缩和解压行为，为此，我们定义了一种专门的文件格式来对压缩后的文件进行存储，详细的定义会在本文的附录A中给出。

1. **模块分析** 时间：2022-2

**4.1 分析环境、内容与方法**

由于我们设计的模块没有独立运行的能力，因此，对整个模块的运行与分析是基于演示Demo的，这个Demo与这次课程设计所提交的可执行文件是等价的，在下一部分中我们将对其进行详细的介绍，另一方面，这个外部的Demo对我们分析的影响几乎可以忽略不计，因此在这里我们不再对其进行说明，下面的分析完全可以看作是只基于模块得到的结果。

这次分析的环境与开发环境基本相同，具体配置如下：

硬件环境：

CPU：Intel(R) Core(TM) i5-10300H；

RAM：8GB.

软件环境：

系统：Windows 10 家庭中文版20H2 19042.1466

编译器：TDM-GCC 4.9.2 64-bit Release

硬件环境：

CPU：Intel(R) Core(TM) i5-10300H；

RAM：8GB.

软件环境：

系统：Windows 10 家庭中文版20H2 19042.1466

编译器：TDM-GCC 4.9.2 64-bit Release

硬件环境：

CPU：Intel(R) Core(TM) i5-10300H；

RAM：8GB.

软件环境：

系统：Windows 10 家庭中文版20H2 19042.1466

编译器：TDM-GCC 4.9.2 64-bit Release

硬件环境：

CPU：Intel(R) Core(TM) i5-10300H；

RAM：8GB.

软件环境：

系统：Windows 10 家庭中文版20H2 19042.1466

编译器：TDM-GCC 4.9.2 64-bit Release

这次测试主要对模块运行时的性能表现（主要为不同情况下压缩和解压消耗的时间以及内存占用情况）和压缩率进行了分析。其中，测试性能表现时仅采用了随机字符文件作为样本，而测试压缩率时则采用了四类文件作为样本。

除了部分较为简单的测试采用了手动测试的方法，大部分的测试都使用了批处理文件和简单的测试程序。

**4.2 性能分析：时间**

这次对于模块在运行时间上的分析主要分为了两类，一类是对单个文件进行压缩和解压时的时间消耗（变量是文件大小），另一类则是对多个文件进行压缩和解压时的时间消耗（变量是文件数量）。

在进行单个文件的性能测试时，我们生成了5组不同规模的随机字符构成的文件，每组由10个不同的文件组成，测试的结果取10个文件操作时间的平均数。

单个文件的性能测试得到的统计图分别如下所示：

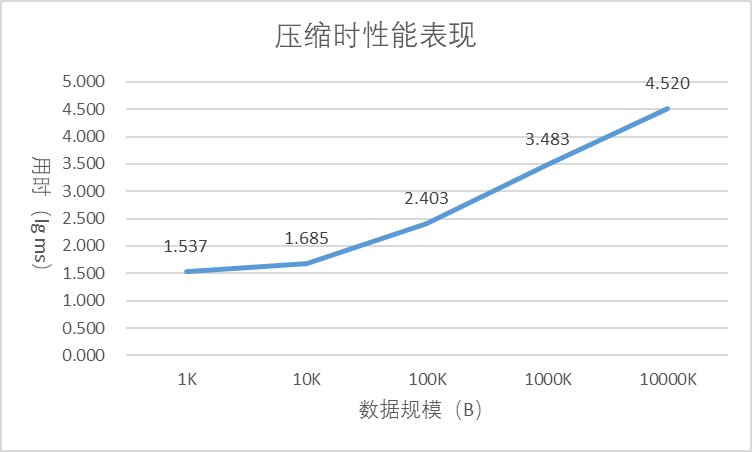


图 2单个文件压缩性能（时间）

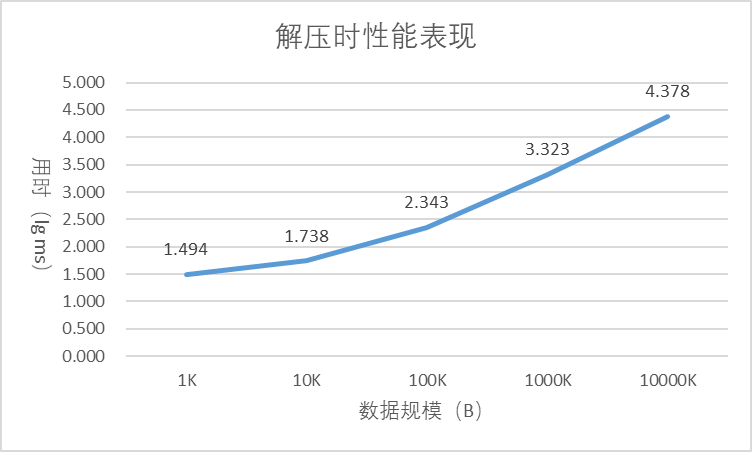


图 3单个文件解压性能（时间）

从上图中可以非常清晰的看出，对于单个文件的操作而言，无论是压缩还是解压，耗费的时间与数据规模都是呈现多项式相关性的，即对于耗费时间的对数(lg t)与数据规模的对数(lg n)二者间的函数(lg t)=f(lg n)而言，f是多项式复杂度的。更进一步的，我们可以根据折线图以及各项的差值关系大致推测，f的最高次数应该不会很大。

另外，由于上述所有的解压目标均是由压缩过程得到的，因此压缩与解压在时间上的性能对比实际上也在某种程度上印证了我们在前文中提到的，对于一个原文件-压缩文件对而言，解压总体上会略快于压缩这一推论。

在进行多个文件的性能测试时，我们将测试数据分为了不同规模的4组，每组由5个文件集构成，这些文件同样也是由随机字符构成的。需要注意的是，4个测试组中所有文件集的大小都是相同的（10,000KB），因此文件集中的文件数量越多，单个文件的大小就越小。

多个文件的性能测试得到的统计图分别如下所示：

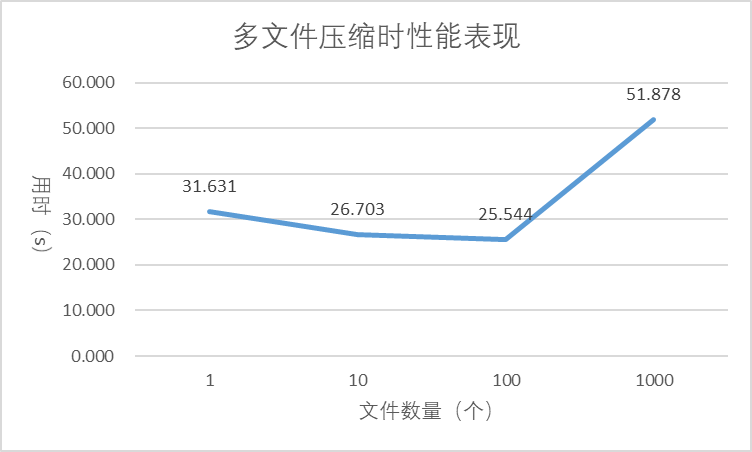


图 4多个文件压缩性能（时间）

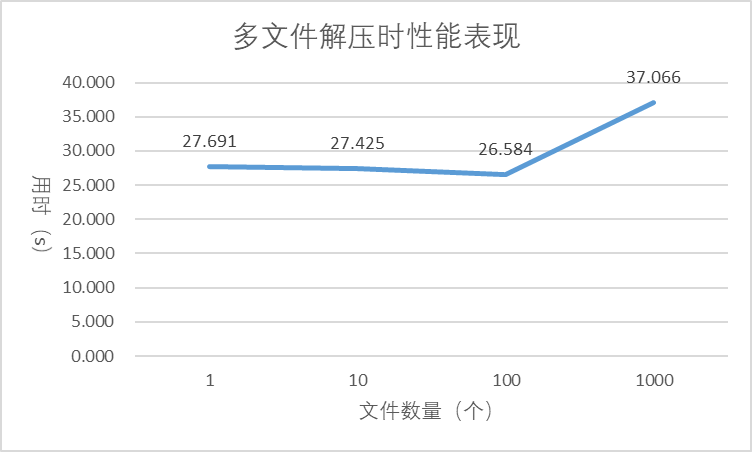


图 5多个文件解压性能（时间）

可以看到，当文件数的数量级在100及以下时，无论是压缩还是解压操作，对时间的影响主要体现在单个文件的大小上，即瓶颈在于模块中的压缩/解压内层核心函数，因此随着文件数量的不断增多，其操作用时是保持稳定甚至略有缩短的。但是，当文件数的数量级达到1,000时，对文件的操作成为了时间的决定性影响因素，此时的瓶颈在于模块中的压缩/解压外层核心函数以及公有函数接口。

因此，我们可以认为，在欲压缩的文件数量不太大（数量级在1,000以下）时，整个模块的多个文件压缩/解压性能与单个文件压缩/解压性能差距不大，总体而言是稳定的；而当欲压缩的文件数量很大时，我们不能对模块性能的稳定性做出保证。

需要注意的是，这些测试均是在默认的参数设置下（navDataLen=1，mapCodeLen=4，docSizeLen=8）得出的，实际上，当mapCodeLen设置为2时，在压缩大文件时的速度会有一定的提升，这个现象的出现是因为字典大小被限制，在进行字典的写入和搜索时耗费的时间减少导致的。当然，由于具体的性能表现涉及到重置字典耗费时间和搜索/写入字典耗费时间谁占主导地位的问题，在某些特殊大小（例如恰好能让更小的字典被重置）的文件上，两种设置的表现可能会与此相反。就目前已经进行的测试而言，更小的字典快于更大的字典这一现象，在文件大小的数量级在100MB时已经被证实会出现且表现十分明显（根据主观感受，更小字典耗费的时间约为更大字典的一半，前者消耗的时间约为4分钟，而后者则消耗了将近9分钟；这一结果未进行检验，仅为进行其他测试时对耗费时间的估算值）。

基于进行下一节测试时的直观感受与合理推测，我们认为，当文件大小的数量级在1,000K以上时，这种现象就会出现（100K、1,000K、10,000K的单组测试结果分别为0.266s、1.922s、18.656s；与之相对的，默认设置下的结果分别为0.253s、3.044s、33.139s），当然，由于时间原因，我们并未对这一猜想进行严格的、由多次实验组成的、控制了变量的测试，因此这个结果的有效性仍有待检验。

测试得到的原始数据会在本文的附录B中给出。

**4.3 性能分析：内存占用**

一个程序在运行过程中占用的内存具有很多的衡量标准，这其中包括但不限于单纯的堆栈分析、内存活动页的数量以及专用内存大小。在这次的测试中，我们采用程序专用内存的峰值大小作为衡量指标。

这一节中的所有测试数据均是通过微软公司推出的工具ProcessExplorer（版本号：16.32.0.0）进行测量的，由于这个工具的最小刷新时间是0.5s，因此对于运行时间极短的测试，我们不能保证其准确性，这主要体现在数量级不超过10K的数据组中。

另外，由于模块对于内存的占用几乎完全取决于目标文件的大小，因此，这一部分我们不进行多文件压缩和解压的测试。

单个文件解压和压缩的资源占用情况如下：

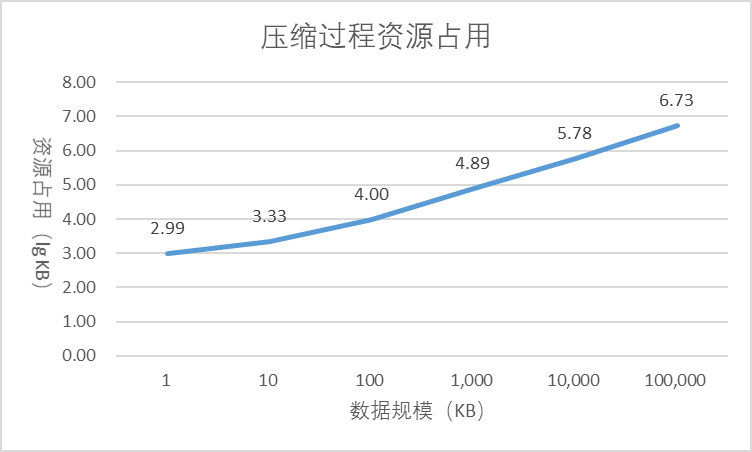


图 6单个文件压缩性能（内存）

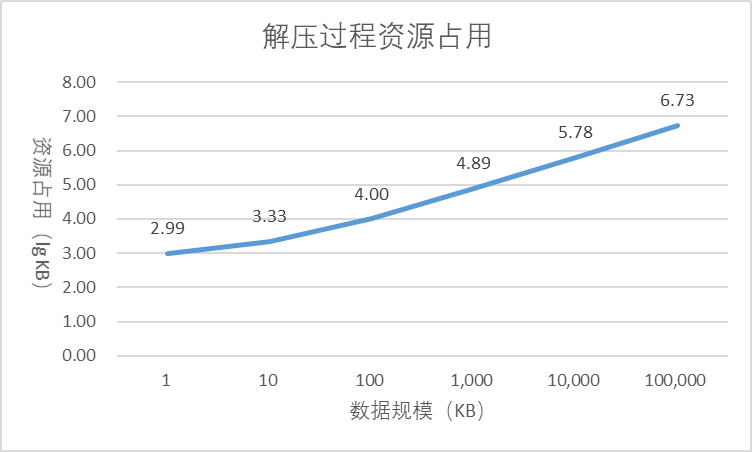


图 7单个文件压缩性能（内存）

可以看到，在达到字典项数极限前，内存占用量与数据规模基本上呈现线性正相关的关系，但是我们需要注意的是，模块占用的内存体积相较于目标文件的体积几乎大了约2个数量级，这意味着在压缩一个较大的文件时，模块占用的内存大小是十分恐怖的。例如，在测试中对约100MB（100,000KB）的文件进行测试时，程序的专用内存峰值全部达到了5GB以上。

另外，与我们在前面算法部分的分析结果相似，对于一个原文件-压缩文件对而言，由于其压缩和解压过程的字典是完全镜像的，因此，它们的内存占用量几乎一致，二者只存在几个内存页（4KB/页）的差距，这一点可以从上图中看出。

除此之外，根据上文中的得到的结论（达到字典项数极限前内存占用量与数据规模大致存在线性正相关关系），我们可以很容易地想到，如果我们将字典大小进行限制，其内存占用量会有较大幅度的下降，因此对于100,000KB规模的文件，我们在将参数mapCodeLen设置为2后，又进行了一次压缩的测试。

测试的结果如下：

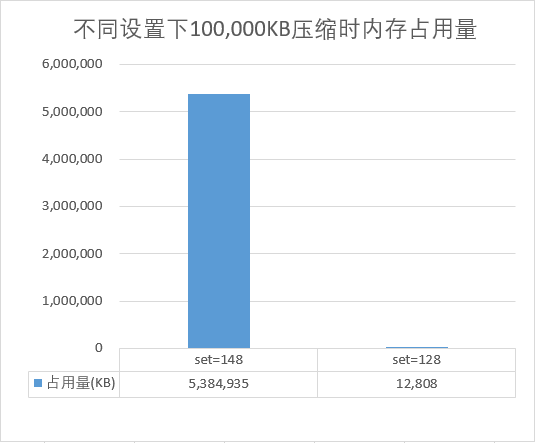


图 8不同设置下性能对比（内存）

可以看出，在限制字典大小后，其内存占用量的改善十分明显，只使用了限制前不到1%的内存。

测试过程中得到的原始数据会在本文的附录B中给出。

**4.4 压缩效率分析**

对于一个压缩模块而言，压缩率是衡量其质量的最为重要的指标之一，LZW压缩由于其独特的自解释字典，相较于其他的压缩方式省去了存储字典的空间，从理论上来说，其压缩率应该是较为优秀的。

由于字典大小对于模块的压缩率影响非常显著，我们将分别对mapCodeLen=2和mapCodeLen=4两种参数设置下的压缩率进行测试。在测试过程中，除了这个参数外，其他所有的参数我们均采用默认值（navDataLen=1,docSizeLen=8）。

在测试中共使用了四类数据，这四类数据分别为：a.多色块构成的bmp图片文件（类别名“half\_repeat”）；b.规则性文本（类别名“order”）；c.随机字符文本（类别名“random”）；d.重复字符（类别名“repeat”）。

当参数为4时的压缩率测试结果如下：

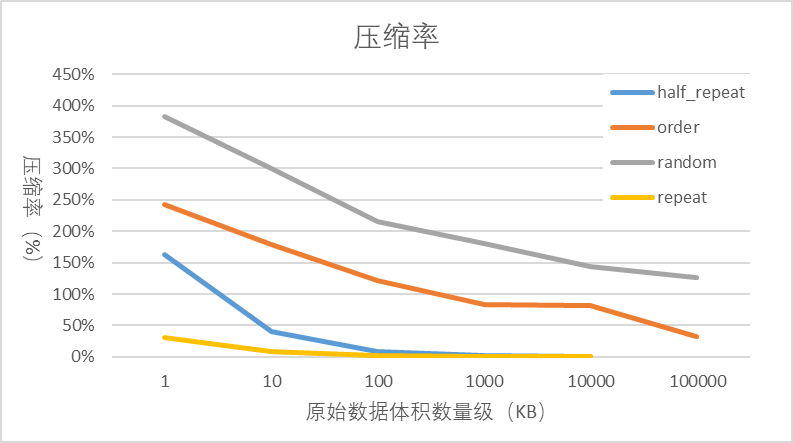


图 9压缩率总览（mapCodeLen=4）

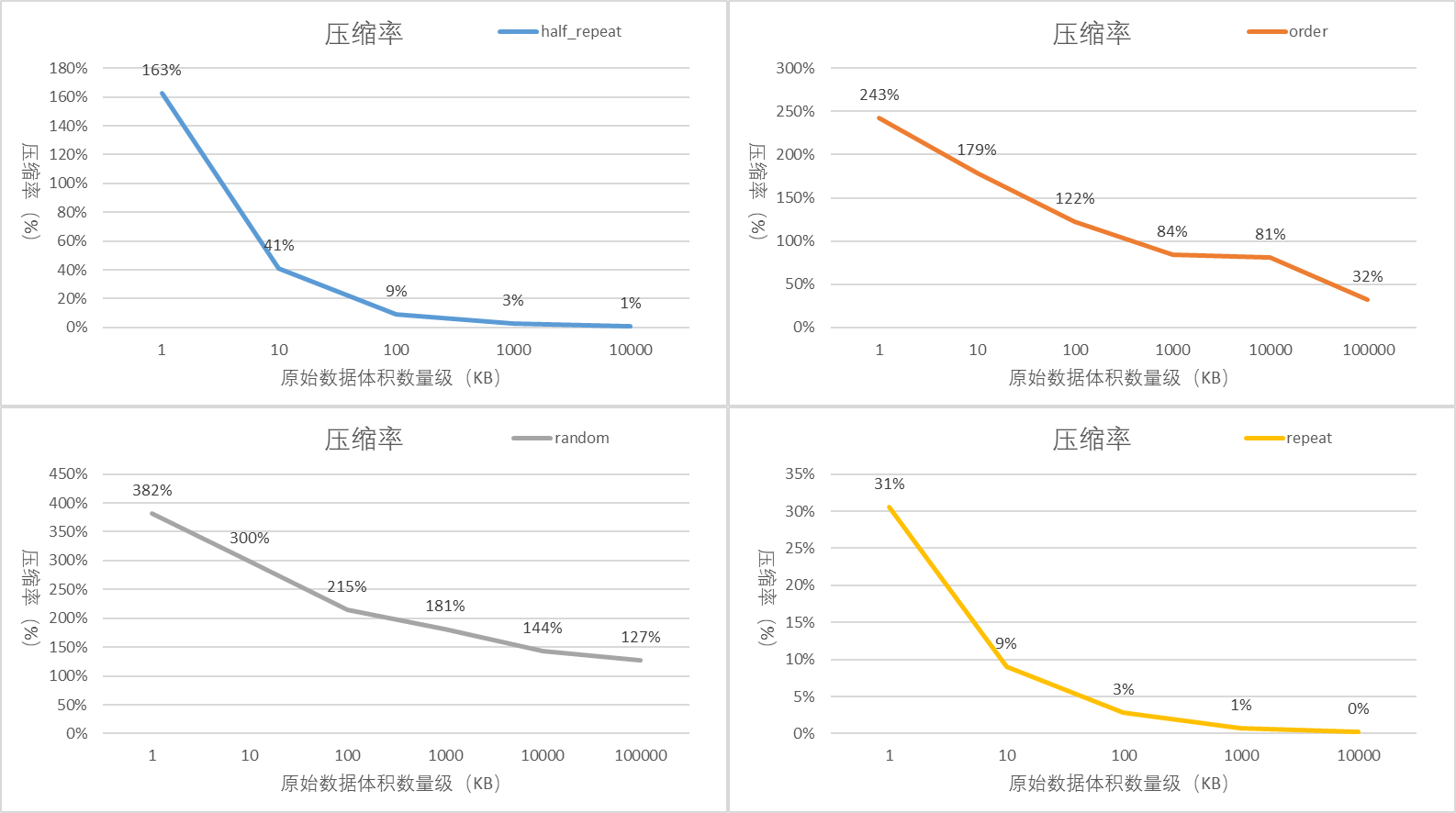


图 10压缩率分览（mapCodeLen=4）

可以看出，当mapCodeLen为4时，模块对随机字符文件的压缩性能极差，完全起不到压缩的作用，不过随着数据规模的增加，其压缩率整体呈现下降趋势，可以预测，当数据规模达到一定程度后，模块是能够对随机字符文件起到一定的压缩作用的。

除了随机字符文件外，此时模块对体积较小的规则性文本能够起到的压缩作用也是十分有限的；不过对于具有较强重复性的文件，模块的压缩效果是十分优秀的。

当参数为2时的压缩率测试结果如下：

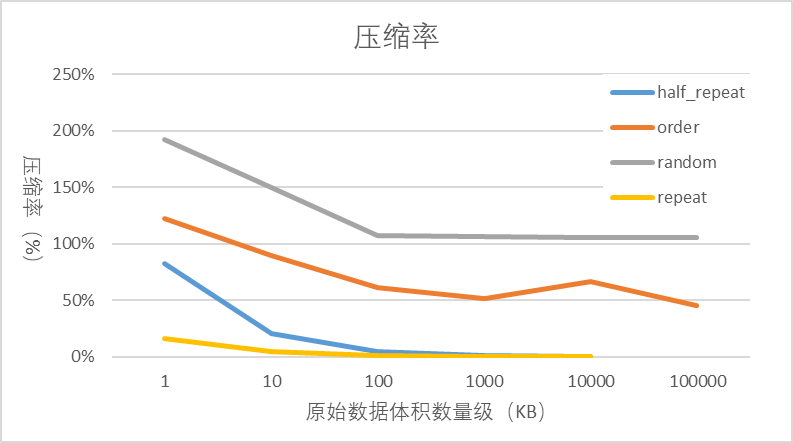


图 11 压缩率总览（mapCodeLen=2）

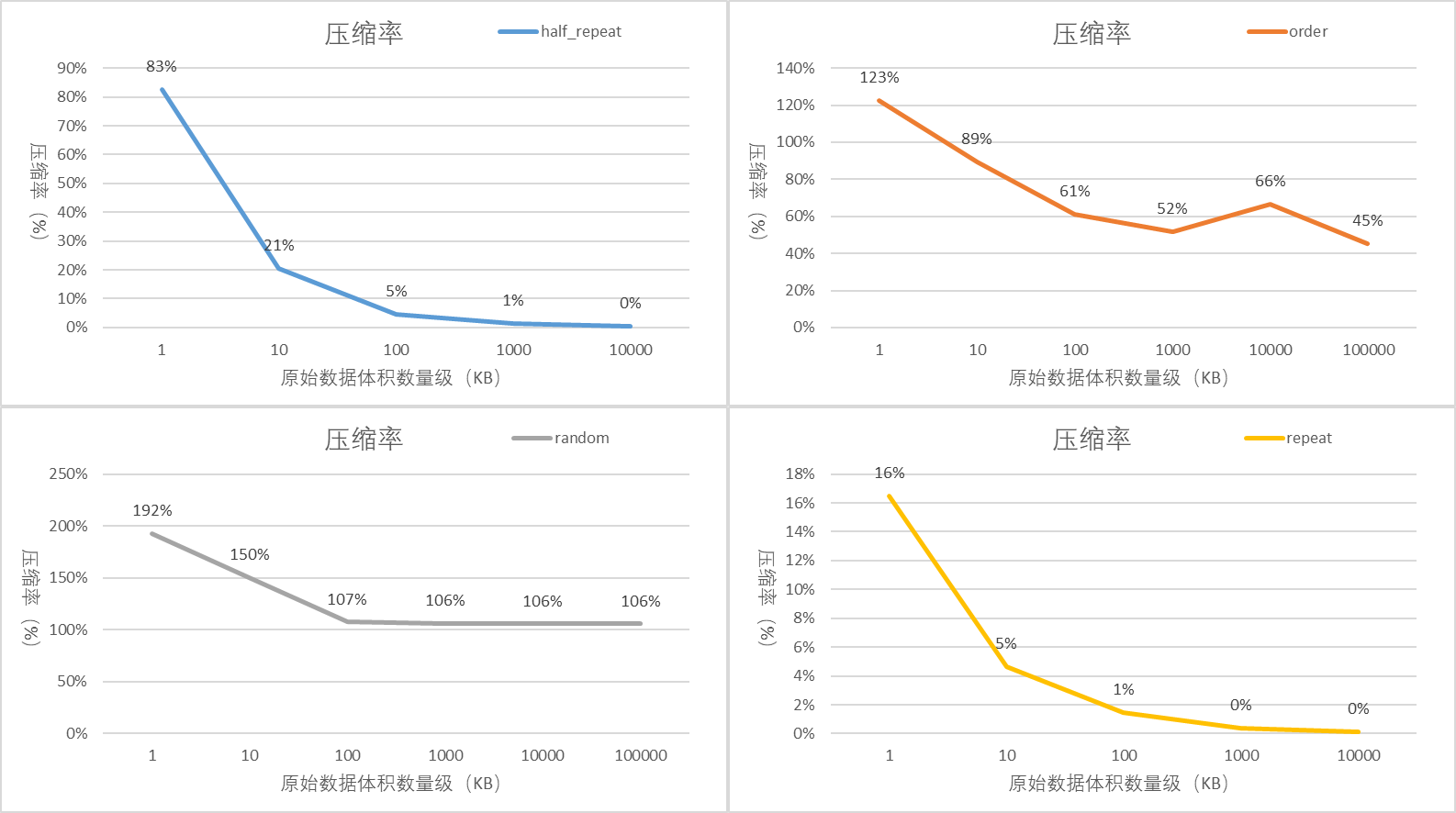


图 12压缩率分览（mapCodeLen=2）

可以看出，当mapCodeLen为2时，模块的压缩率相较于参数为4时有了十分明显的优化，很多情况下甚至只有参数为4时的一半。

但是在这种情况下，模块对于随机字符文件仍然起不到压缩的作用，而且曲线的趋势令人担忧，我们猜测，当mapCodeLen与navDataLen的比值更小时，模块才能对随机文件起到压缩作用。但由于mapCodeLen实际上最小只能为2，我们只能通过增大navDataLen 的方式降低比率。基于这个猜测，我们对navDataLen=3，mapCodeLen=4的情况进行了测试，但效果并不尽人意，模块仍然对10,000KB数量级的数据起不到压缩作用。通过分析结果来看，我们认为这是模块设计的问题而非算法的问题。

不过除了随机字符文件一项外，对于其他类型的文件，模块在此设置下都起到了非常好的压缩效果，虽然对于体积在1KB量级（包括1KB以下）的规则性文本仍旧起不到压缩作用，但这在某种程度上属于压缩算法的通病。因此总体而言，模块在此设置下的压缩效果是非常优秀的。

对于测试数据的具体说明和测试结果的原始数据会在本文的附录C中给出。

1. **Demo设计**

**5.1 Demo说明**

正如我们在第一部分中所提到的，这个Demo仅起到演示的作用，因此我们使其成为了一个基于Windows命令行的程序。

这个命令行程序与常见的命令行工具类似，均是通过参数决定操作的。相关的配置完成后，在不使用任何参数、使用参数“-help”或使用了错误的参数时，演示程序会显示如下所示的帮助文本：

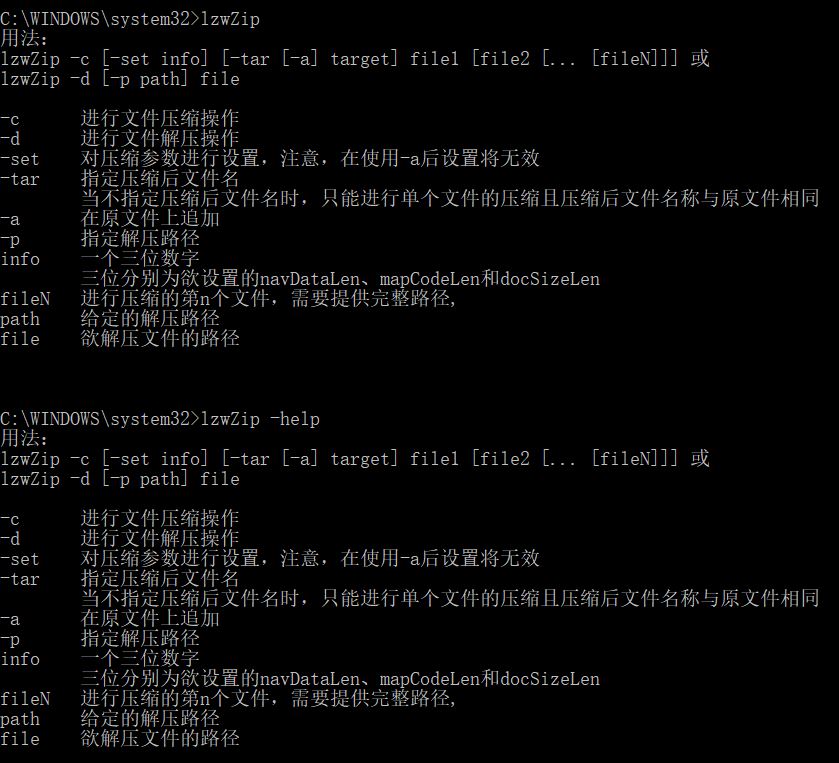


图 13帮助文本

由于程序并未处理任何来自LzwCore类的异常，因此来自于LzwCore类的异常会被直接显示到界面中，而当程序压缩成功时，程序会直接退出。

这个演示程序支持进行目前LzwCore类中提供的所有压缩与解压操作。

另外，为了方便使用，演示程序支持对后缀正确的文件（即.lzwf文件）直接进行打开，即设置为打开方式。

为了在不改变系统环境变量的前提下使自己能够直接被运行，演示程序会将自身复制到“C:\windows\system32”目录中并进行重新命名。由于在Windows10下默认非管理员用户不能对C盘进行文件读写操作，因此需要使用管理员身份打开演示程序。而在首次以管理员身份打开演示程序时，会显示以下的提示：

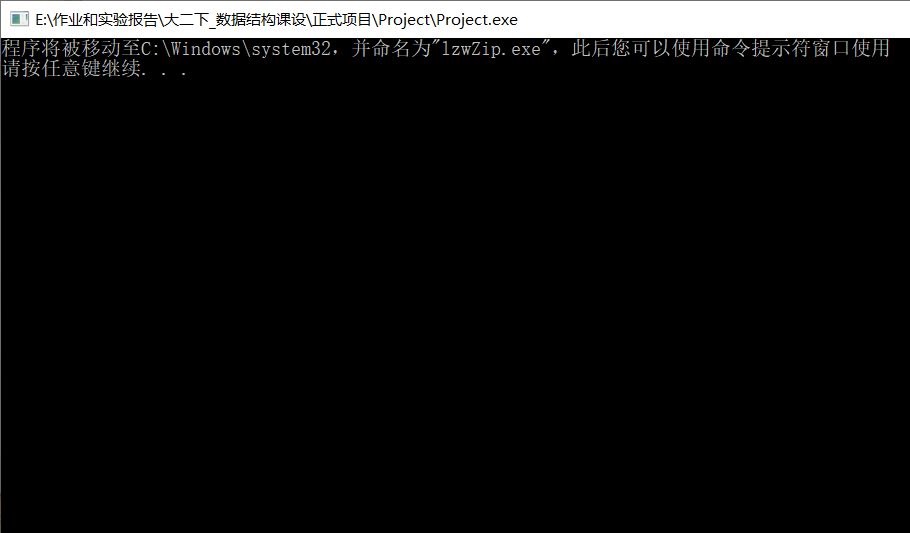


图 14首次打开程序时的初始化结果

此后，演示程序自动完成配置，之后则可以使用命令提示符窗口进行操作，当前程序则会被弃用。

需要注意的是，由于LzwCore类会进行临时文件操作，而由于相同的原因（在Windows10下默认非管理员用户不能对C盘进行文件读写操作），我们仍然需要以管理员身份运行此演示程序。

1. **改进方向**

**6.1 性能改进：多线程压缩**

当进行对多个文件的压缩时，我们显然可以通过使用多线程来提高压缩的效率。

事实上，压缩过程中使用临时文件就是为了对多线程压缩进行支持。理论上，如果我们想要在目前LzwCore类的基础上进行多线程化改造，那么只需要定义一个LzwCore类的友元函数来调用LzwCore类的核心压缩函数，再将这个友元函数进行实现即可完成多线程化的改造。

不过，考虑到稳定性等因素，LzwCore类并未对多线程解压进行支持，因此如果想要进行多线程解压，可以需要对LzwCore类进行大量的修改。

**6.2 效率改进：基于位的操作**

从上文中的第四部分对压缩率的测试可以看出，当我们对体积不太大的文件进行压缩时，在一定范围内，字典的大小越小，压缩率越高。事实上，LzwCore类目前支持的最小字典编码大小为2字节（16位），而通常采用的方案中字典编码的大小为12位。

因此，如果我们将基于字节的映射操作改为基于位的映射操作，便可以使用标准的8位对12位的映射方案，理论上，这可以显著提升小文件压缩时的压缩效率。

**6.3 功能改进：更多功能**

事实上，模块有很多较为重要的功能没有进行实现，例如：（1）对压缩文件中的某个子文件进行删除；（2）对压缩文件中的某一个子文件进行单独的解压；（3）查看压缩文件中每个文件的文件名。

这些功能仅仅是因为时间原因而没有完成，借助LzwCore类现有的成员函数，这些功能实际上都可以比较容易的进行实现。

**附录A lzwf标准文件格式**

lzwf文件的结构如下所示：

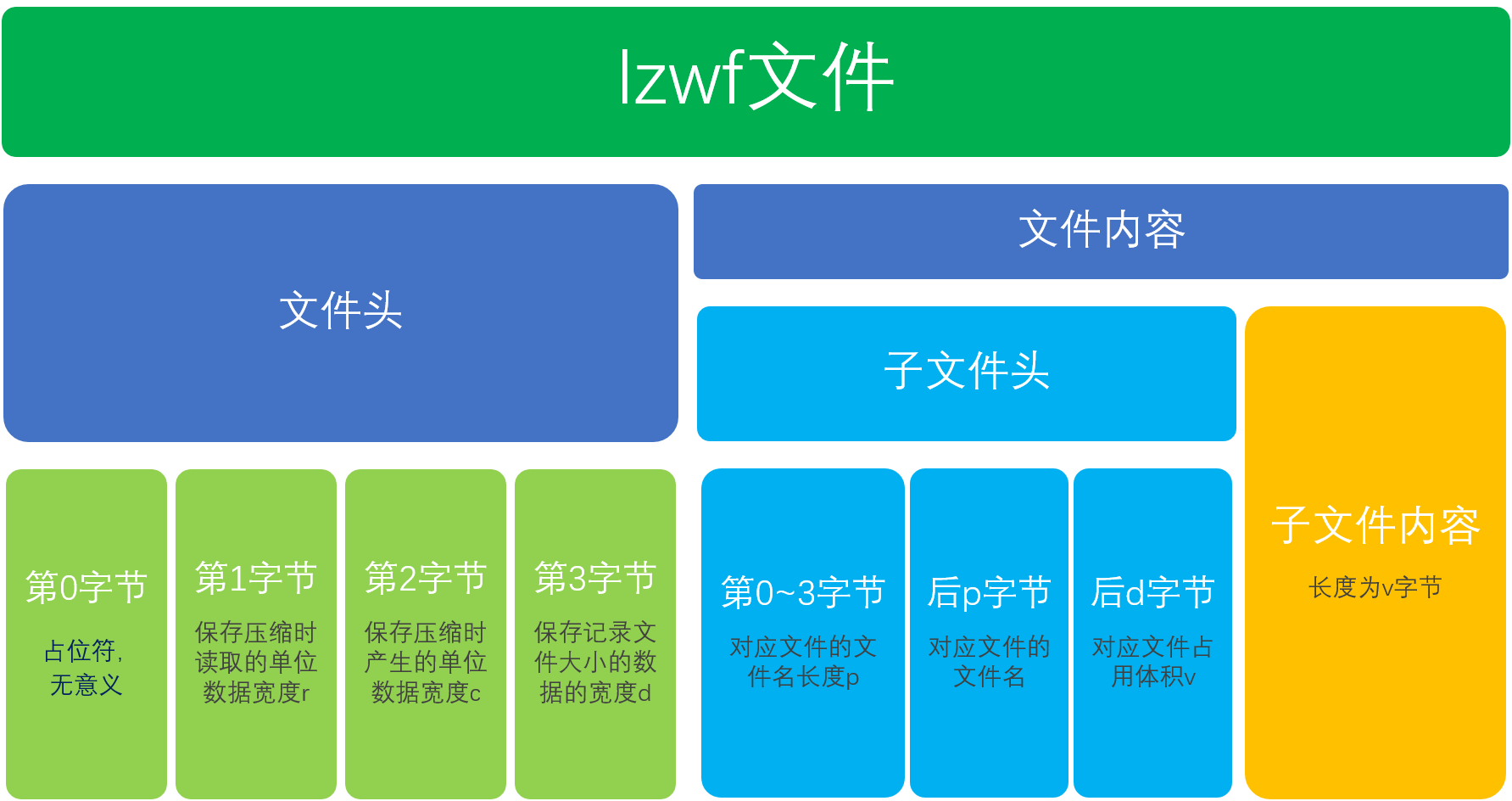
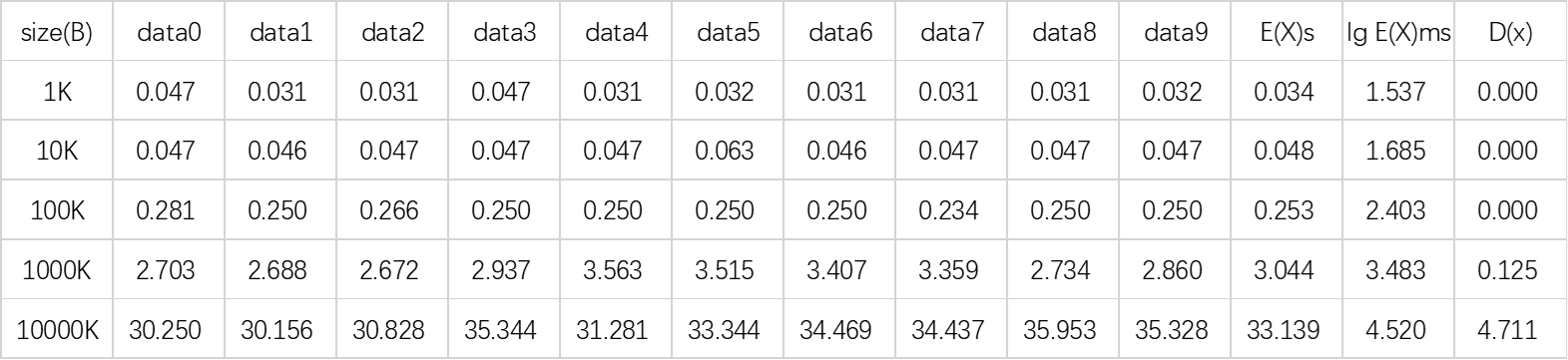


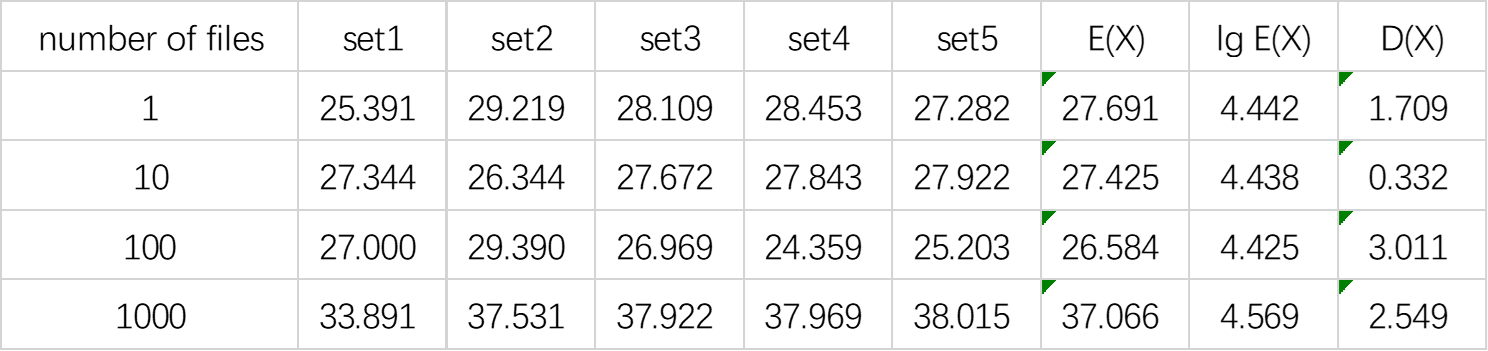
图 15 lzwf文件结构

文件头中的1-3字节分别对应LzwCore中的三个参数，即navDataLen、mapCodeLen和docSizeLen。

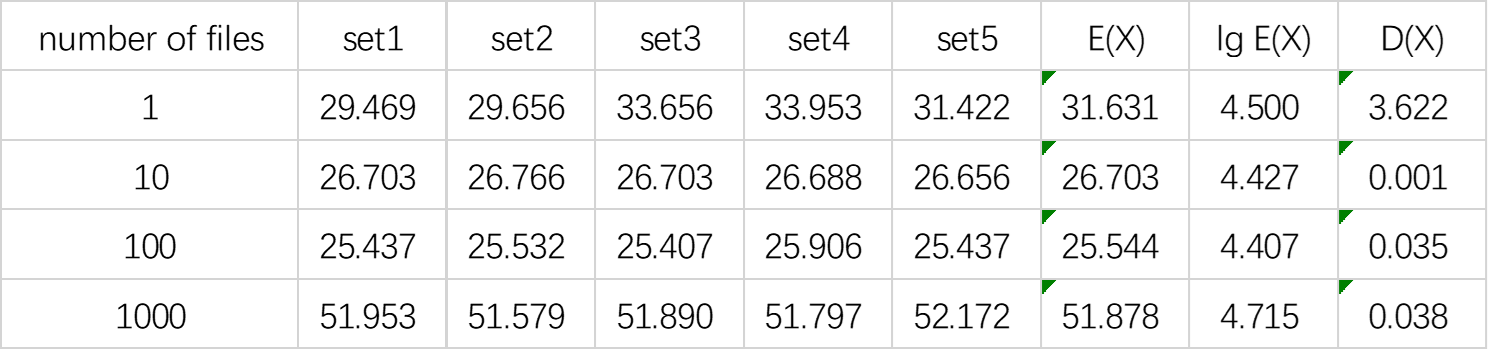
**附录B 原始数据：性能测试**



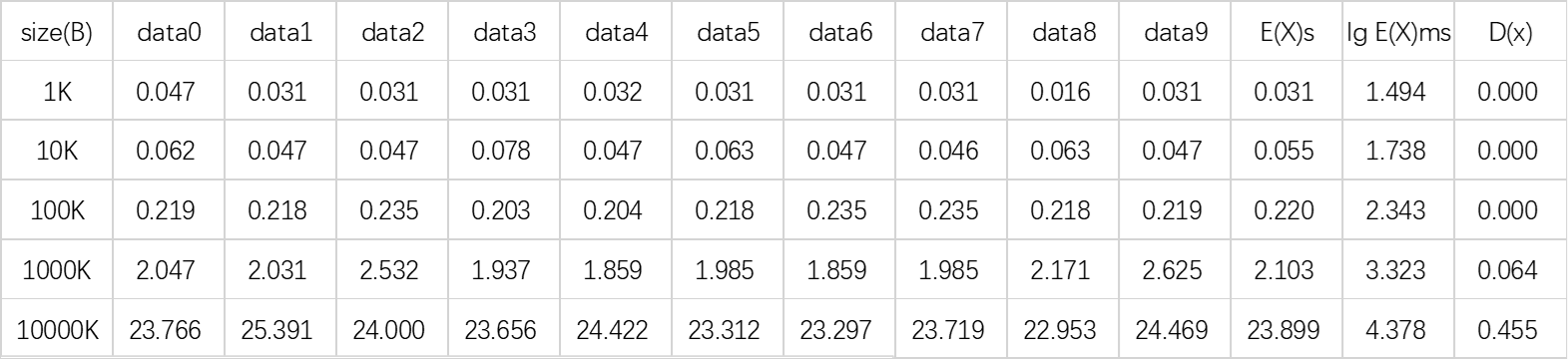
数据 1单文件压缩时间消耗



数据 4 多文件解压时间消耗

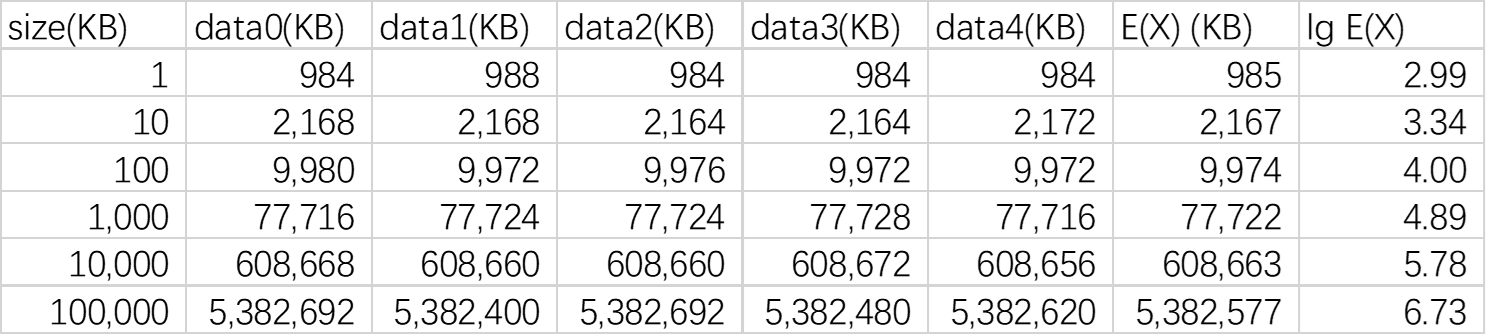


数据 3 多文件压缩时间消耗

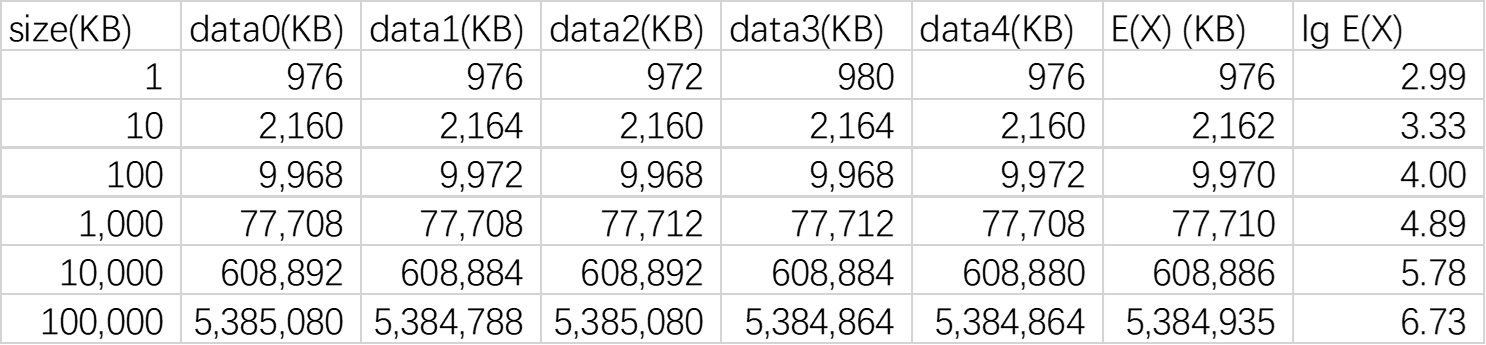


数据 2单文件解压时间消耗

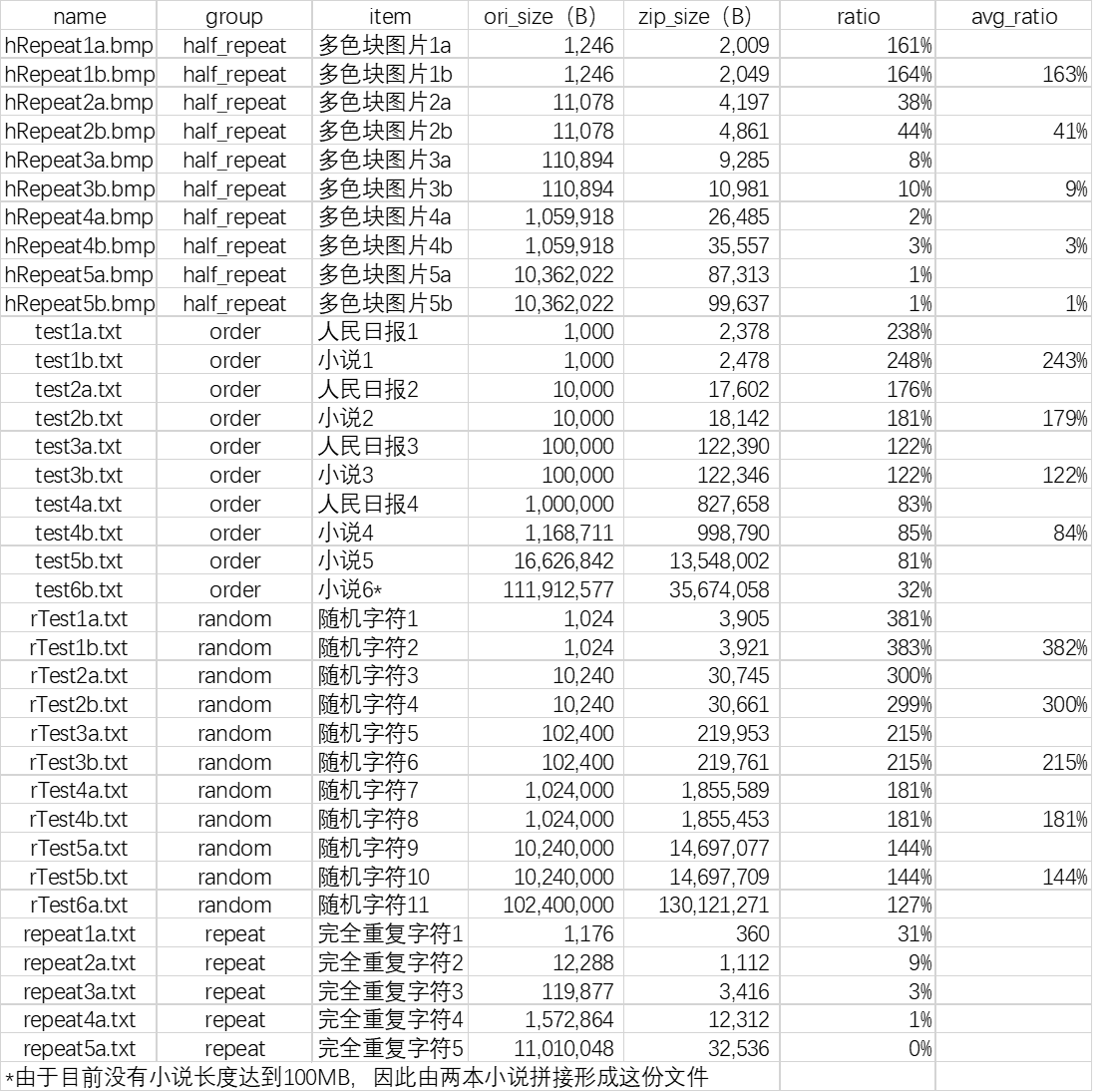
数据1-4的单位除对数列使用毫秒为取对数前单位外，其他列均采用秒作为单位。

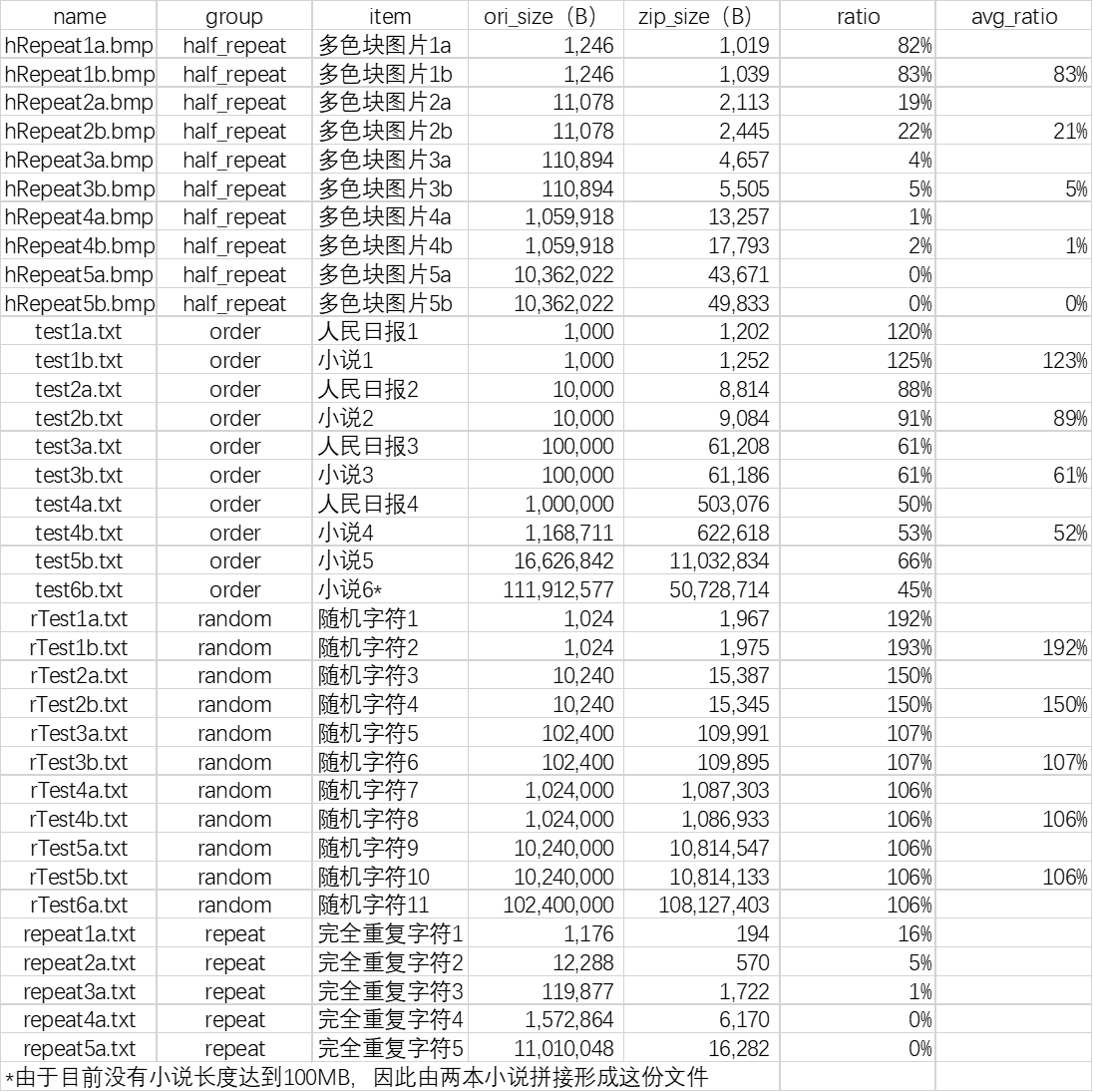


数据 6 单文件解压内存消耗（默认设置）

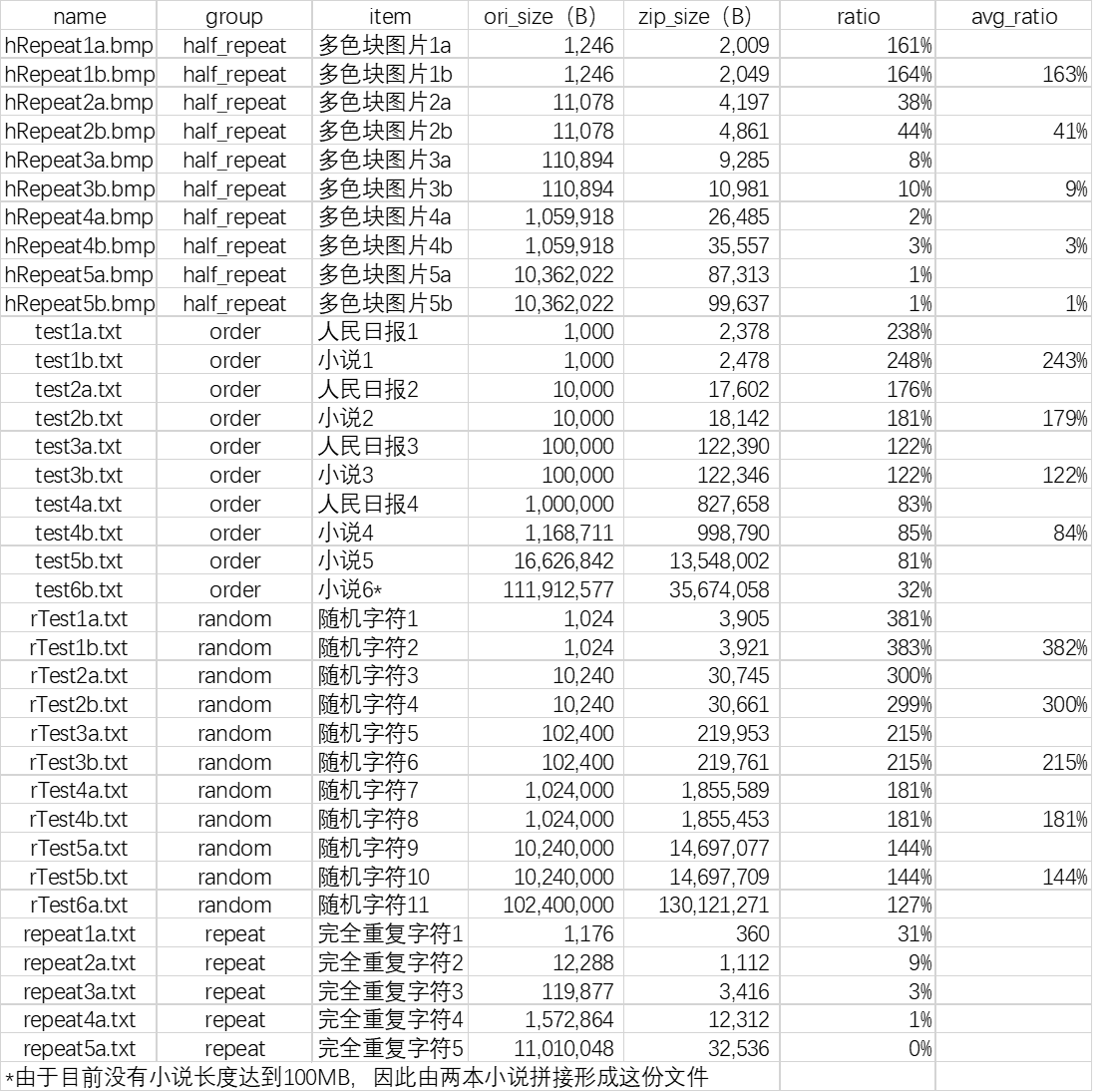


数据 5单文件压缩内存消耗（默认设置）

**附录C 原始数据：效率测试**



数据 8 mapCodeLen=2时的压缩率



数据 7 默认参数下压缩率