LZ77:

这是市面上最为通用的一种数据压缩算法，利用当前出现字符串在之前出现过，而达到压缩的效果，实际而言，压缩的效果相比Huffman压缩效率更高，但与此同时的不足是，对数据的压缩异常的慢（因为涉及到大量的字符串匹配动作），但是解压的效率又异常的高，对数据流的一次遍历就解压完毕了。

这个开源的LZ77压缩算法，对字符串匹配采用了一个更为优化的策略，即散列表筛选机制，利用前三个字符通过计算散列函数可以筛选大量无意义比较，实际上可以节约很大的时间。但是这部分代码内部未出现对散列冲突问题的解决，所带来的潜在影响是，可能会遗落部分可压缩数据，所以压缩并不是最完全的。

通常在解决冲突完冲突的时候，还会再添加Huffman编码压缩对LZ77压缩后的数据进一步压缩，这是目前通用的高效压缩的方式。但该LZ77并不具备Huffman编码压缩，因为Huffman编码压缩中，对Huffman树的构造耗费的时间也是挺长的，可能这是一个考量的范围，而且压缩后还要添加Huffman表，可能会适得其反。

预处理以工程为背景平台进行新一步的修订，1.c是部分重要的预编译分析。0.c是重要预编译后的一个完整.c文件（略过函数实现），压缩和解压的实现细节解析被额外列举在compressor.c和decompressor.c中。

这俩个宏定义在本环境下是无效的，直接删除

#define FASTLZ\_EXPECT\_CONDITIONAL(c) (c)

#define FASTLZ\_UNEXPECT\_CONDITIONAL(c) (c)

.c的注释是使用的Notepad写的，tab是俩字节对齐的

编码风格大致上被修改为K&R风格

而关键字解析如下

#define FASTLZ\_EXPECT\_CONDITIONAL(c) (\_\_builtin\_expect((c), 1))

#define FASTLZ\_UNEXPECT\_CONDITIONAL(c) (\_\_builtin\_expect((c), 0))

c表达式为真的概率和为假的概率的比例，意味着如果使用该宏，编译器会优化这部分，将原来的分支跳转（jmp）进行新一部分修订

列举：

int x

if((x > 0)){

//为真时

}else{

//为假时

}

假如x>0的概率大于90%，那么上述条件跳转是具备很高的性能的

那么这种时候，为真情况的代码段会紧接判断之后，而为假的代码段需要（jmp）到else 之后

假如x>0的概率小于10%，那么上述条件跳转的性能就很低下，

那么这种时候，大部分为假的情况下都需要jmp，就很影响处理器性能

如果我们添加一个预定断言

那么最终，编译器会按照你认定的概率事件进行代码优化，最终的效果是：

大部分情况下都不需要jmp