**Usage Example**

We often get questions about how the deflate() and inflate() functions should be used. Users wonder when they should provide more input, when they should use more output, what to do with a Z\_BUF\_ERROR, how to make sure the process terminates properly, and so on. So for those who have read zlib.h (a few times), and would like further edification, below is an annotated example in C of simple routines to compress and decompress from an input file to an output file using deflate() and inflate() respectively. The annotations are interspersed between lines of the code. So please read between the lines. We hope this helps explain some of the intricacies of *zlib*.

Without further adieu, here is the program [zpipe.c](http://www.zlib.net/zpipe.c):

**/\* zpipe.c: example of proper use of zlib's inflate() and deflate()**

**Not copyrighted -- provided to the public domain**

**Version 1.4 11 December 2005 Mark Adler \*/**

**/\* Version history:**

**1.0 30 Oct 2004 First version**

**1.1 8 Nov 2004 Add void casting for unused return values**

**Use switch statement for inflate() return values**

**1.2 9 Nov 2004 Add assertions to document zlib guarantees**

**1.3 6 Apr 2005 Remove incorrect assertion in inf()**

**1.4 11 Dec 2005 Add hack to avoid MSDOS end-of-line conversions**

**Avoid some compiler warnings for input and output buffers**

**\*/**

We now include the header files for the required definitions. From stdio.h we use fopen(), fread(), fwrite(), feof(), ferror(), and fclose() for file i/o, and fputs() for error messages. From string.h we use strcmp() for command line argument processing. From assert.h we use the assert() macro. From zlib.h we use the basic compression functions deflateInit(), deflate(), and deflateEnd(), and the basic decompression functions inflateInit(), inflate(), and inflateEnd().

**#include <stdio.h>**

**#include <string.h>**

**#include <assert.h>**

**#include "zlib.h"**

This is an ugly hack required to avoid corruption of the input and output data on Windows/MS-DOS systems. Without this, those systems would assume that the input and output files are text, and try to convert the end-of-line characters from one standard to another. That would corrupt binary data, and in particular would render the compressed data unusable. This sets the input and output to binary which suppresses the end-of-line conversions. SET\_BINARY\_MODE() will be used later on stdin and stdout, at the beginning of main().

**这段代码是为了防止在WIN/MS-DOS系统上出现输入输出的问题。如果没有这段代码，上述操作系统会将输入、输出文件视为文本，将zlib文件中可能存在的EOF字符转换为它们自己的标准。这会破坏二进制数据，尤其会使压缩数据不可用。这段代码将输入输出均设置为二进制模式，消除了EOF转换的问题，将会在main函数里用到。**

**#if defined(MSDOS) || defined(OS2) || defined(WIN32) || defined(\_\_CYGWIN\_\_)**

**# include <fcntl.h>**

**# include <io.h>**

**# define SET\_BINARY\_MODE(file) setmode(fileno(file), O\_BINARY)**

**#else**

**# define SET\_BINARY\_MODE(file)**

**#endif**

CHUNK is simply the buffer size for feeding data to and pulling data from the *zlib* routines. Larger buffer sizes would be more efficient, especially for inflate(). If the memory is available, buffers sizes on the order of 128K or 256K bytes should be used.

**CHUNK是向zlib流程输入数据及提取数据的缓冲区大小。缓冲区越大，压缩、解压效率越高，尤其是解压。如果内存足够，尽量使用128K、256K之类的大小。**

**#define CHUNK 16384**

The def() routine compresses data from an input file to an output file. The output data will be in the *zlib* format, which is different from the *gzip* or *zip* formats. The *zlib* format has a very small header of only two bytes to identify it as a *zlib* stream and to provide decoding information, and a four-byte trailer with a fast check value to verify the integrity of the uncompressed data after decoding.

**def()函数从输入文件压缩数据，写入输出文件。输出文件是zlib格式（与gzip和zip不同）的。Zlib格式有一个2字节的头部，标识这是一个zlib流，并提供解压信息；还有一个4字节的尾部，是用来在解压后校验数据完整性的。**

**/\* Compress from file source to file dest until EOF on source.**

**def() returns Z\_OK on success, Z\_MEM\_ERROR if memory could not be**

**allocated for processing, Z\_STREAM\_ERROR if an invalid compression**

**level is supplied, Z\_VERSION\_ERROR if the version of zlib.h and the**

**version of the library linked do not match, or Z\_ERRNO if there is**

**an error reading or writing the files. \*/**

**int def(FILE \*source, FILE \*dest, int level)**

**{**

Here are the local variables for def(). ret will be used for *zlib* return codes. flush will keep track of the current flushing state for deflate(), which is either no flushing, or flush to completion after the end of the input file is reached. have is the amount of data returned from deflate(). The strm structure is used to pass information to and from the *zlib* routines, and to maintain the deflate() state. in and out are the input and output buffers for deflate().

**ret保存zlib函数的返回值**

**flush跟踪deflate()的flushing状态，要么是无flushing，要么在读到输入文件底后全部flush**

**have是deflate()返回的数据数量**

**strm是核心结构**

**in和out是deflate()的输入输出缓冲区**

**int ret, flush;**

**unsigned have;**

**z\_stream strm;**

**unsigned char in[CHUNK];**

**unsigned char out[CHUNK];**

The first thing we do is to initialize the *zlib* state for compression using deflateInit(). This must be done before the first use of deflate(). The zalloc, zfree, and opaque fields in the strm structure must be initialized before calling deflateInit(). Here they are set to the *zlib* constant Z\_NULL to request that *zlib* use the default memory allocation routines. An application may also choose to provide custom memory allocation routines here. deflateInit() will allocate on the order of 256K bytes for the internal state. (See [*zlib Technical Details*](http://www.zlib.net/zlib_tech.html).)

**首先我们要用deflateInit()初始化zlib状态。Strm结构中的zalloc, zfree, opaque必须在调用deflateInit()之前初始化。本例中都设为了Z\_NULL，要求zlib使用默认的内存分配策略。应用程序也可以在这里使用定制的内存分配策略。deflateInit()会为内部状态分配256K字节之类大小的内存。**

deflateInit() is called with a pointer to the structure to be initialized and the compression level, which is an integer in the range of -1 to 9. Lower compression levels result in faster execution, but less compression. Higher levels result in greater compression, but slower execution. The *zlib* constant Z\_DEFAULT\_COMPRESSION, equal to -1, provides a good compromise between compression and speed and is equivalent to level 6. Level 0 actually does no compression at all, and in fact expands the data slightly to produce the *zlib* format (it is not a byte-for-byte copy of the input). More advanced applications of *zlib* may use deflateInit2() here instead. Such an application may want to reduce how much memory will be used, at some price in compression. Or it may need to request a *gzip* header and trailer instead of a *zlib* header and trailer, or raw encoding with no header or trailer at all.

**deflateInit()有两个参数，一个是要初始化的结构的指针（strm），一个是压缩的等级（level）。level的值在-1~9之间。压缩等级越低，执行速度越快，压缩率越低。常量Z\_DEFAULT\_COMPRESSION（等于-1）表示在压缩率和速度方面寻求平衡，实际上和等级6一样。等级0实际上不做任何压缩，只对输入数据做略微改变以形成zlib格式（并不是简单的一字节一字节的拷贝）。高级程序可以在这里使用deflateInit2()以：降低内存消耗，但是同时要付出压缩率的代价；或者使用gzip头部和尾部来代替zlib；或者不要头尾部的原始编码。**

We must check the return value of deflateInit() against the *zlib* constant Z\_OK to make sure that it was able to allocate memory for the internal state, and that the provided arguments were valid. deflateInit() will also check that the version of *zlib* that the zlib.h file came from matches the version of *zlib* actually linked with the program. This is especially important for environments in which *zlib* is a shared library.

**我们必须检查deflateInit()的返回值，如果为Z\_OK，则说明内存分配成功，参数合法。deflateInit()还会检查zlib.h头文件所使用的zlib库版本和链接器使用的zlib库版本是否一致，这对于共享zlib库的环境尤为重要。**

Note that an application can initialize multiple, independent *zlib* streams, which can operate in parallel. The state information maintained in the structure allows the *zlib* routines to be reentrant.

**注意，应用程序可以初始化多个相互独立的zlib流，它们可以并行执行。z\_stream结构中保存的状态信息可以让zlib方法可重入。**

**/\* allocate deflate state \*/**

**strm.zalloc = Z\_NULL;**

**strm.zfree = Z\_NULL;**

**strm.opaque = Z\_NULL;**

**ret = deflateInit(&strm, level);**

**if (ret != Z\_OK)**

**return ret;**

With the pleasantries out of the way, now we can get down to business. The outer do-loop reads all of the input file and exits at the bottom of the loop once end-of-file is reached. This loop contains the only call of deflate(). So we must make sure that all of the input data has been processed and that all of the output data has been generated and consumed before we fall out of the loop at the bottom.

**外层的do-while循环读入所有的输入数据，如果读到了文件结尾则结束循环。这个循环里面只调用了函数deflate()。我们必须确保所有的输入数据都被处理而且所有的输出数据都被输出了（此例中是写入输出文件），然后才可以退出循环。**

**/\* compress until end of file \*/**

**do {**

We start off by reading data from the input file. The number of bytes read is put directly into avail\_in, and a pointer to those bytes is put into next\_in. We also check to see if end-of-file on the input has been reached using feof(). If we are at the end of file, then flush is set to the *zlib* constant Z\_FINISH, which is later passed to deflate() to indicate that this is the last chunk of input data to compress. If we are not yet at the end of the input, then the *zlib* constant Z\_NO\_FLUSH will be passed to deflate to indicate that we are still in the middle of the uncompressed data.

**从输入文件读入数据，读取的字节数被赋给avail\_in，指向这些数据的指针被赋给next\_in。用feof检查是否读到了输入文件的文件尾，如果读到了文件尾，那么flush被置为Z\_FINISH，flush变量稍后会传递给deflate()，表明这是最后一段要被压缩的输入数据了。如果还没到文件尾，flush被置为Z\_NO\_FLUSH，表明我们还有未压缩的数据。**

If there is an error in reading from the input file, the process is aborted with deflateEnd() being called to free the allocated *zlib* state before returning the error. We wouldn't want a memory leak, now would we? deflateEnd() can be called at any time after the state has been initialized. Once that's done, deflateInit() (or deflateInit2()) would have to be called to start a new compression process. There is no point here in checking the deflateEnd() return code. The deallocation can't fail.

**如果在读输入文件中遇到错误，结束进程。在结束之前，要调用deflateEnd()释放zlib的状态。deflateEnd()不会出错，不必检查返回值。**

**strm.avail\_in = fread(in, 1, CHUNK, source);**

**if (ferror(source)) {**

**(void)deflateEnd(&strm);**

**return Z\_ERRNO;**

**}**

**flush = feof(source) ? Z\_FINISH : Z\_NO\_FLUSH;**

**strm.next\_in = in;**

The inner do-loop passes our chunk of input data to deflate(), and then keeps calling deflate() until it is done producing output. Once there is no more new output, deflate() is guaranteed to have consumed all of the input, i.e., avail\_in will be zero.

**内层do-while循环将我们读到的一段输入数据传递给deflate()，然后不停调用deflate()直到输出产生完毕。一旦没有新的输出，也就意味着deflate()已经处理掉了所有的输入，avail\_in的值已经变为0**

**/\* run deflate() on input until output buffer not full, finish**

**compression if all of source has been read in \*/**

**do {**

Output space is provided to deflate() by setting avail\_out to the number of available output bytes and next\_out to a pointer to that space.

**输出缓冲区的设置：设置avail\_out为可用输出缓冲字节数，next\_out为指向缓冲区的指针。**

**strm.avail\_out = CHUNK;**

**strm.next\_out = out;**

Now we call the compression engine itself, deflate(). It takes as many of the avail\_in bytes at next\_in as it can process, and writes as many as avail\_out bytes to next\_out. Those counters and pointers are then updated past the input data consumed and the output data written. It is the amount of output space available that may limit how much input is consumed. Hence the inner loop to make sure that all of the input is consumed by providing more output space each time. Since avail\_in and next\_in are updated by deflate(), we don't have to mess with those between deflate() calls until it's all used up.

**现在调用压缩引擎deflate()。它尽可能多地使用next\_in指向的长度为avail\_in的输入缓冲区中的数据，向next\_out指向的输出缓冲区中写入长度为next\_out的数据。上述计数器和指针随之更新，跳过已经消耗掉的输入数据和已经写出的输出数据。输出缓冲区的大小可能会限制能消耗多少输入数据。因此内层循环通过每次增加输出缓冲区来确保所有的输入数据都被处理了。因为avail\_in和next\_in都是由deflate()更新的，在输入缓冲区内的数据被消耗完之前我们不必管它们。（但是因为每次deflate()都输出avail\_out的数据，将输出缓冲区填满，所以next\_out和avail\_outout必须由应用程序自己更新）**

The parameters to deflate() are a pointer to the strm structure containing the input and output information and the internal compression engine state, and a parameter indicating whether and how to flush data to the output. Normally deflate will consume several K bytes of input data before producing any output (except for the header), in order to accumulate statistics on the data for optimum compression. It will then put out a burst of compressed data, and proceed to consume more input before the next burst. Eventually, deflate() must be told to terminate the stream, complete the compression with provided input data, and write out the trailer check value. deflate() will continue to compress normally as long as the flush parameter is Z\_NO\_FLUSH. Once the Z\_FINISH parameter is provided, deflate() will begin to complete the compressed output stream. However depending on how much output space is provided, deflate() may have to be called several times until it has provided the complete compressed stream, even after it has consumed all of the input. The flush parameter must continue to be Z\_FINISH for those subsequent calls.

**deflate()有两个参数:z\_stream结构的参数保存了输入输出信息和内部压缩引擎的状态；int类型的flush参数指明了是否以及如何将数据flush到输出缓冲区。通常，为了加速压缩，deflate会处理K字节的输入数据，然后才产生输出（头部除外）。Deflate会突然输出一段压缩后的数据，然后再获取更多输入，然后再突然输出。最后，必须告知deflate()停止读取新的数据，将已读取的输入数据进行压缩，输出，并加上最后的尾部的校验。只要flush参数是Z\_NO\_FLUSH ，deflate()会一直进行压缩。一旦flush参数变为Z\_FINISH，deflate()会开始结束压缩过程。但是，取决于输出缓冲区的大小，即使已经读取完所有输入，仍可能要多次调用deflate()才能使其完成全部压缩输出。在这些连续调用过程中，flush参数必须保持为Z\_FINISH。**

There are other values of the flush parameter that are used in more advanced applications. You can force deflate() to produce a burst of output that encodes all of the input data provided so far, even if it wouldn't have otherwise, for example to control data latency on a link with compressed data. You can also ask that deflate() do that as well as erase any history up to that point so that what follows can be decompressed independently, for example for random access applications. Both requests will degrade compression by an amount depending on how often such requests are made.

**在高级程序中，flush参数还可以有其他值。**

deflate() has a return value that can indicate errors, yet we do not check it here. Why not? Well, it turns out that deflate() can do no wrong here. Let's go through deflate()'s return values and dispense with them one by one. The possible values are Z\_OK, Z\_STREAM\_END, Z\_STREAM\_ERROR, or Z\_BUF\_ERROR. Z\_OK is, well, ok. Z\_STREAM\_END is also ok and will be returned for the last call of deflate(). This is already guaranteed by calling deflate() with Z\_FINISH until it has no more output. Z\_STREAM\_ERROR is only possible if the stream is not initialized properly, but we did initialize it properly. There is no harm in checking for Z\_STREAM\_ERROR here, for example to check for the possibility that some other part of the application inadvertently clobbered the memory containing the *zlib* state. Z\_BUF\_ERROR will be explained further below, but suffice it to say that this is simply an indication that deflate() could not consume more input or produce more output. deflate() can be called again with more output space or more available input, which it will be in this code.

**deflate()有返回值来告知错误，但是这个例子中我们并不需要检查返回值。为什么？让我们逐一来看deflate()可能的返回值。Z\_OK，没有错误。Z\_STREAM\_END，说明读到输入文件尾部了，但并没有关系，我们的代码连续调用deflate()直到不再产生输出。(产生 输入已读完，而仍需要调用deflate()进行压缩 的原因是输出缓冲区的大小的限制)。Z\_STREAM\_ERROR只会在流未被正确初始化的情况下出现，但是我们确实正确初始化了。当然，检查一下Z\_STREAM\_ERROR也没有坏处，因为有可能程序的其他部分不经意地改变了zlib的内存。Z\_BUF\_ERROR表明deflate()不能再读取更多输入或者产生任何输出了，这种情况下可以通过给予更多输入或者分配更多输出缓冲来再次调用deflate()，下面还会提到Z\_BUF\_ERROR。**

**ret = deflate(&strm, flush); /\* no bad return value \*/**

**assert(ret != Z\_STREAM\_ERROR); /\* state not clobbered \*/**

Now we compute how much output deflate() provided on the last call, which is the difference between how much space was provided before the call, and how much output space is still available after the call. Then that data, if any, is written to the output file. We can then reuse the output buffer for the next call of deflate(). Again if there is a file i/o error, we call deflateEnd() before returning to avoid a memory leak.

**我们现在计算上一次调用deflate()时产生了多少输出，即是调用前分配的输出缓冲区大小减去调用后还剩下的输出缓冲区大小。然后将输出缓冲区的数据写入输出文件。然后下次调用deflate()时又可以重新使用这片输出缓冲区了。如果有文件I/O错误，我们先调用deflateEnd()然后再返回，以免内存泄露。**

**have = CHUNK - strm.avail\_out;**

**if (fwrite(out, 1, have, dest) != have || ferror(dest)) {**

**(void)deflateEnd(&strm);**

**return Z\_ERRNO;**

**}**

The inner do-loop is repeated until the last deflate() call fails to fill the provided output buffer. Then we know that deflate() has done as much as it can with the provided input, and that all of that input has been consumed. We can then fall out of this loop and reuse the input buffer.

**内层do-while循环直到deflate()不能填满给定的输出缓冲区为止（前面提到了，每次deflate()尽可能多地消耗输入数据，但是一次“突然”burst产生的输出数据大小都和输出缓冲区大小一样，除非最后一次产生的输出填不满输出缓冲区）。此时我们知道deflate()已经消耗完了输入缓冲区内的数据，我们可以退出内层循环，然后重新利用这片输入缓冲区。**

The way we tell that deflate() has no more output is by seeing that it did not fill the output buffer, leaving avail\_out greater than zero. However suppose that deflate() has no more output, but just so happened to exactly fill the output buffer! avail\_out is zero, and we can't tell that deflate() has done all it can. As far as we know, deflate() has more output for us. So we call it again. But now deflate() produces no output at all, and avail\_out remains unchanged as CHUNK. That deflate() call wasn't able to do anything, either consume input or produce output, and so it returns Z\_BUF\_ERROR. (See, I told you I'd cover this later.) However this is not a problem at all. Now we finally have the desired indication that deflate() is really done, and so we drop out of the inner loop to provide more input to deflate().

**我们通过看deflate()是否填满了输出缓冲区来判断其是否还有更多的输出没做，如果avail\_out大于0，说明输出已经做完了。但是假设这样一种巧合，最后一次的deflate()产生的输出刚好填满了输出缓冲区，avail\_out为0，但是deflate()确实已经处理完了所有输入，所有输出也已经做了。这种情况其实没关系，我们再调用一次deflate()，此时会返回Z\_BUF\_ERROR。**

With flush set to Z\_FINISH, this final set of deflate() calls will complete the output stream. Once that is done, subsequent calls of deflate() would return Z\_STREAM\_ERROR if the flush parameter is not Z\_FINISH, and do no more processing until the state is reinitialized.

**如果flush参数设成Z\_FINISH，最后的几次deflate()调用会完成输出流工作。一旦这个做完了，之后的deflate()调用如果flush参数不为Z\_FINISH，则会返回Z\_STREAM\_ERROR，并且不进行任何操作，直到重新初始化z\_stream状态。**

Some applications of *zlib* have two loops that call deflate() instead of the single inner loop we have here. The first loop would call without flushing and feed all of the data to deflate(). The second loop would call deflate() with no more data and the Z\_FINISH parameter to complete the process. As you can see from this example, that can be avoided by simply keeping track of the current flush state.

**有些程序用两层循环来代替我们这里的内层循环。第一层循环flush设为Z\_NO\_FLUSH，将所有输入都读进去，第二层循环将flush设置为Z\_FINISH，不再输入，让deflate()完成输出。我们的代码里避免了这样做，因为保持追踪flush的状态更方便。（代码中对应flush = feof(source) ? Z\_FINISH : Z\_NO\_FLUSH;）**

**} while (strm.avail\_out == 0);**

**assert(strm.avail\_in == 0); /\* all input will be used \*/**

Now we check to see if we have already processed all of the input file. That information was saved in the flush variable, so we see if that was set to Z\_FINISH. If so, then we're done and we fall out of the outer loop. We're guaranteed to get Z\_STREAM\_END from the last deflate() call, since we ran it until the last chunk of input was consumed and all of the output was generated.

**我们通过检查flush是否为Z\_FINISH（因为flush = feof(source) ? Z\_FINISH : Z\_NO\_FLUSH;）来判断是否还有输入文件未被读取。最后一次调用deflate()的返回值必然是Z\_STREAM\_END，因为所有的输入都已经读完，所有的输出也都产生。**

**/\* done when last data in file processed \*/**

**} while (flush != Z\_FINISH);**

**assert(ret == Z\_STREAM\_END); /\* stream will be complete \*/**

The process is complete, but we still need to deallocate the state to avoid a memory leak (or rather more like a memory hemorrhage if you didn't do this). Then finally we can return with a happy return value.

**整个def()就要结束了，调用deflateEnd防止内存泄露。**

**/\* clean up and return \*/**

**(void)deflateEnd(&strm);**

**return Z\_OK;**

**}**

Now we do the same thing for decompression in the inf() routine. inf() decompresses what is hopefully a valid *zlib* stream from the input file and writes the uncompressed data to the output file. Much of the discussion above for def() applies to inf() as well, so the discussion here will focus on the differences between the two.

**下面的inf()是解压。输入数据应该是从输入文件读到的合法的zlib流，将解压后的数据写入输出文件。大部分和def()类似。下面主要讲不一样的地方。**

**/\* Decompress from file source to file dest until stream ends or EOF.**

**inf() returns Z\_OK on success, Z\_MEM\_ERROR if memory could not be**

**allocated for processing, Z\_DATA\_ERROR if the deflate data is**

**invalid or incomplete, Z\_VERSION\_ERROR if the version of zlib.h and**

**the version of the library linked do not match, or Z\_ERRNO if there**

**is an error reading or writing the files. \*/**

**int inf(FILE \*source, FILE \*dest)**

**{**

The local variables have the same functionality as they do for def(). The only difference is that there is no flush variable, since inflate() can tell from the *zlib* stream itself when the stream is complete.

**int ret;**

**unsigned have;**

**z\_stream strm;**

**unsigned char in[CHUNK];**

**unsigned char out[CHUNK];**

The initialization of the state is the same, except that there is no compression level, of course, and two more elements of the structure are initialized. avail\_in and next\_in must be initialized before calling inflateInit(). This is because the application has the option to provide the start of the zlib stream in order for inflateInit() to have access to information about the compression method to aid in memory allocation. In the current implementation of *zlib* (up through versions 1.2.x), the method-dependent memory allocations are deferred to the first call of inflate() anyway. However those fields must be initialized since later versions of *zlib* that provide more compression methods may take advantage of this interface. In any case, no decompression is performed by inflateInit(), so the avail\_out and next\_out fields do not need to be initialized before calling.

**状态的初始化是一样的，除了没有level这个参数(当然了，这个得从输入的zlib数据里获取)。Z\_stream除了要初始化zalloc, zfree ， opaque外，还要初始化next\_in, avail\_in。**

Here avail\_in is set to zero and next\_in is set to Z\_NULL to indicate that no input data is being provided.

这里avail\_in设为0，next\_in设为Z\_NULL，表示没有提供输入数据。

**/\* allocate inflate state \*/**

**strm.zalloc = Z\_NULL;**

**strm.zfree = Z\_NULL;**

**strm.opaque = Z\_NULL;**

**strm.avail\_in = 0;**

**strm.next\_in = Z\_NULL;**

**ret = inflateInit(&strm);**

**if (ret != Z\_OK)**

**return ret;**

The outer do-loop decompresses input until inflate() indicates that it has reached the end of the compressed data and has produced all of the uncompressed output. This is in contrast to def() which processes all of the input file. If end-of-file is reached before the compressed data self-terminates, then the compressed data is incomplete and an error is returned.

**外层循环以inflate()是否返回Z\_STREAM\_END作为循环终止条件。因为如果inflate()返回Z\_STREAM\_END，说明 输入已读完，而且所有输出都产生了（这里和deflate()不同，和flush参数是否为Z\_FINISH无关。deflate()返回Z\_STREAM\_END说明 输入已读完，而且如果设置了Z\_FINISH的话，所有输出都会产生）。这个和def()相反，def()是判断输入是否已经读完，inf()是判断输出是否已经全部做了。如果在读到压缩数据尾部之前就读到了文件尾，那么说明压缩数据不完整，返回错误。**

**/\* decompress until deflate stream ends or end of file \*/**

**do {**

We read input data and set the strm structure accordingly. If we've reached the end of the input file, then we leave the outer loop and report an error, since the compressed data is incomplete. Note that we may read more data than is eventually consumed by inflate(), if the input file continues past the *zlib* stream. For applications where *zlib* streams are embedded in other data, this routine would need to be modified to return the unused data, or at least indicate how much of the input data was not used, so the application would know where to pick up after the *zlib* stream.

**如果在读到压缩文件尾之前读到文件尾，说明压缩数据不完整，结束外部循环，报错。注意读到的数据可能比inflate()最终消耗的数据要多。在这样的程序中，要注意返回未用到的数据，至少也要指明还有多少输入数据没有被inflate()使用，使得程序可以知道从哪里获取zlib数据之后非zlib数据。**

**strm.avail\_in = fread(in, 1, CHUNK, source);**

**if (ferror(source)) {**

**(void)inflateEnd(&strm);**

**return Z\_ERRNO;**

**}**

**if (strm.avail\_in == 0)**

**break;**

**strm.next\_in = in;**

The inner do-loop has the same function it did in def(), which is to keep calling inflate() until has generated all of the output it can with the provided input.

**内层循环和def()类似，不断调用inflate()直到给定输入被处理完，产生的所有输出都做了。**

**/\* run inflate() on input until output buffer not full \*/**

**do {**

Just like in def(), the same output space is provided for each call of inflate().

**和def()中一样，每次调用inflate()都提供输出缓冲区**

**strm.avail\_out = CHUNK;**

**strm.next\_out = out;**

Now we run the decompression engine itself. There is no need to adjust the flush parameter, since the *zlib* format is self-terminating. The main difference here is that there are return values that we need to pay attention to. Z\_DATA\_ERROR indicates that inflate() detected an error in the *zlib* compressed data format, which means that either the data is not a *zlib* stream to begin with, or that the data was corrupted somewhere along the way since it was compressed. The other error to be processed is Z\_MEM\_ERROR, which can occur since memory allocation is deferred until inflate() needs it, unlike deflate(), whose memory is allocated at the start by deflateInit().

**下面可以跑解压引擎了。不需要设置flush参数，因为zlib格式是自终结的（这里不明白）。主要的不同在于，需要注意inflate()的返回值。Z\_DATA\_ERROR说明读取的数据有错，有么不是zlib格式的，要么某个地方数据有错误。Z\_MEM\_ERROR说明内存不足。deflate()里的内存在deflateInit()中就被分配好了，而inflate()的内存是有“延迟”的，inflate()需要才分配。**

Advanced applications may use deflateSetDictionary() to prime deflate() with a set of likely data to improve the first 32K or so of compression. This is noted in the *zlib* header, so inflate() requests that that dictionary be provided before it can start to decompress. Without the dictionary, correct decompression is not possible. For this routine, we have no idea what the dictionary is, so the Z\_NEED\_DICT indication is converted to a Z\_DATA\_ERROR.

**高级程序中deflate()可能使用了字典（用deflateSetDictionary()设置**

**）进行压缩，那么inflate()就需要使用同样的字典进行解压。如果没有，返回Z\_NEED\_DICT。**

inflate() can also return Z\_STREAM\_ERROR, which should not be possible here, but could be checked for as noted above for def(). Z\_BUF\_ERROR does not need to be checked for here, for the same reasons noted for def(). Z\_STREAM\_END will be checked for later.

**和def()中一样，Z\_STREAM\_ERROR不可能出现。Z\_BUF\_ERROR也不必特别处理，和上面的def()一样。**

**ret = inflate(&strm, Z\_NO\_FLUSH);**

**assert(ret != Z\_STREAM\_ERROR); /\* state not clobbered \*/**

**switch (ret) {**

**case Z\_NEED\_DICT:**

**ret = Z\_DATA\_ERROR; /\* and fall through \*/**

**case Z\_DATA\_ERROR:**

**case Z\_MEM\_ERROR:**

**(void)inflateEnd(&strm);**

**return ret;**

**}**

The output of inflate() is handled identically to that of deflate().

**have = CHUNK - strm.avail\_out;**

**if (fwrite(out, 1, have, dest) != have || ferror(dest)) {**

**(void)inflateEnd(&strm);**

**return Z\_ERRNO;**

**}**

The inner do-loop ends when inflate() has no more output as indicated by not filling the output buffer, just as for deflate(). In this case, we cannot assert that strm.avail\_in will be zero, since the deflate stream may end before the file does.

**当inflate()没有更多输出的时候，内层循环结束。同样通过看输出缓冲区是否大于0来判断。**

**} while (strm.avail\_out == 0);**

The outer do-loop ends when inflate() reports that it has reached the end of the input *zlib* stream, has completed the decompression and integrity check, and has provided all of the output. This is indicated by the inflate() return value Z\_STREAM\_END. The inner loop is guaranteed to leave ret equal to Z\_STREAM\_END if the last chunk of the input file read contained the end of the *zlib* stream. So if the return value is not Z\_STREAM\_END, the loop continues to read more input.

**如果inflate()返回Z\_STREAM\_END（说明它读到了输入zlib流的尾部，完成了解压和完整性校验，所有的输出已经做了），外层循环结束。内层循环汇中的inflate()，在读到输入文件里有zlib流的尾部时，返回Z\_STREAM\_END，外层循环因此终结。如果不为Z\_STREAM\_END，继续进行外层循环，读输入文件。**

**/\* done when inflate() says it's done \*/**

**} while (ret != Z\_STREAM\_END);**

At this point, decompression successfully completed, or we broke out of the loop due to no more data being available from the input file. If the last inflate() return value is not Z\_STREAM\_END, then the *zlib* stream was incomplete and a data error is returned. Otherwise, we return with a happy return value. Of course, inflateEnd() is called first to avoid a memory leak.

**解压完成，如果外层循环是因为Z\_STREAM\_END结束的，happy ending，否则就是碰到了Z\_DATA\_ERROR或者Z\_MEM\_ERROR。**

**/\* clean up and return \*/**

**(void)inflateEnd(&strm);**

**return ret == Z\_STREAM\_END ? Z\_OK : Z\_DATA\_ERROR;**

**}**

That ends the routines that directly use *zlib*. The following routines make this a command-line program by running data through the above routines from stdin to stdout, and handling any errors reported by def() or inf().

zerr() is used to interpret the possible error codes from def() and inf(), as detailed in their comments above, and print out an error message. Note that these are only a subset of the possible return values from deflate() and inflate().

**/\* report a zlib or i/o error \*/**

**void zerr(int ret)**

**{**

**fputs("zpipe: ", stderr);**

**switch (ret) {**

**case Z\_ERRNO:**

**if (ferror(stdin))**

**fputs("error reading stdin\n", stderr);**

**if (ferror(stdout))**

**fputs("error writing stdout\n", stderr);**

**break;**

**case Z\_STREAM\_ERROR:**

**fputs("invalid compression level\n", stderr);**

**break;**

**case Z\_DATA\_ERROR:**

**fputs("invalid or incomplete deflate data\n", stderr);**

**break;**

**case Z\_MEM\_ERROR:**

**fputs("out of memory\n", stderr);**

**break;**

**case Z\_VERSION\_ERROR:**

**fputs("zlib version mismatch!\n", stderr);**

**}**

**}**

Here is the main() routine used to test def() and inf(). The zpipe command is simply a compression pipe from stdin to stdout, if no arguments are given, or it is a decompression pipe if zpipe -d is used. If any other arguments are provided, no compression or decompression is performed. Instead a usage message is displayed. Examples are zpipe < foo.txt > foo.txt.z to compress, and zpipe -d < foo.txt.z > foo.txt to decompress.

**/\* compress or decompress from stdin to stdout \*/**

**int main(int argc, char \*\*argv)**

**{**

**int ret;**

**/\* avoid end-of-line conversions \*/**

**SET\_BINARY\_MODE(stdin);**

**SET\_BINARY\_MODE(stdout);**

**/\* do compression if no arguments \*/**

**if (argc == 1) {**

**ret = def(stdin, stdout, Z\_DEFAULT\_COMPRESSION);**

**if (ret != Z\_OK)**

**zerr(ret);**

**return ret;**

**}**

**/\* do decompression if -d specified \*/**

**else if (argc == 2 && strcmp(argv[1], "-d") == 0) {**

**ret = inf(stdin, stdout);**

**if (ret != Z\_OK)**

**zerr(ret);**

**return ret;**

**}**

**/\* otherwise, report usage \*/**

**else {**

**fputs("zpipe usage: zpipe [-d] < source > dest\n", stderr);**

**return 1;**

**}**

**}**