**POCO C++库学习和分析 -- 流 （三）**

**5 . ZLib Stream流**

           Poco::DeflatingInputStream、Poco::DeflatingOutputStream、Poco::InflatingInputStream和Poco::InflatingOutputStream把zlib的压缩过程封装成为流方式。在讨论其实现之前，先来了解一下涉及到的压缩和校验算法。

**5.1 zlib库**

           在Poco中实现的压缩过程是通过zlib库实现的。下面对zlib的介绍主要来自于[wiki百科](http://zh.wikipedia.org/wiki/Zlib)。  
           zlib是提供资料压缩之用的函式库，由Jean-loup Gailly与Mark Adler所开发，初版0.9版在1995年5月1日发表。  
           zlib目前应用很广泛，下面是其一些应用例子:  
                 \* Linux核心：使用zlib以实作网络协定的压缩、档案系统的压缩以及开机时解压缩自身的核心。  
                 \* libpng，用于PNG图形格式的一个实现，对bitmap数据规定了DEFLATE作为流压缩方法。  
                 \* Apache：使用zlib实作http 1.1。  
                 \* OpenSSH、OpenSSL：以zlib达到最佳化加密网络传输。  
                 \* FFmpeg：以zlib读写Matroska等以DEFLATE算法压缩的多媒体串流格式。  
                 \* rsync：以zlib最佳化远端同步时的传输。  
                 \* The dpkg and RPM package managers, which use zlib to unpack files from compressed software packages.  
                 \* Subversion 、Git和 CVS 版本控制 系统，使用zlib来压缩和远端仓库的通讯流量。  
                 \* dpkg和RPM等包管理软件：以zlib解压缩RPM或者其他封包。  
                \* 因为其代码的可移植性，宽松的许可以及较小的内存占用，zlib在许多嵌入式设备中也有应用。  
  
           zlib支持两种封装格式：gzip和zlib stream。二者都是对deflate压缩算法的封装。HTTP1.1(rfc2616)里的gzip就代表gzip，而deflate则代表zlib stream，并非raw deflate。(这里raw deflate是指不加头的使用deflate算法压缩的原始数据)。下面是gzip和zlib的格式示意。

           按照gzip的RFC 1952，gzip压缩格式包含如下部分：

**[plain]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8851666)

1. +---+---+---+---+---+---+---+---+---+---+
2. |ID1|ID2|CM |FLG|     MTIME     |XFL|OS | (more-->)
3. +---+---+---+---+---+---+---+---+---+---+
4. +=======================+
5. |...compressed blocks...| (more-->)
6. +=======================+
7. 0   1   2   3   4   5   6   7
8. +---+---+---+---+---+---+---+---+
9. |     CRC32     |     ISIZE     |
10. +---+---+---+---+---+---+---+---+

           zlib的格式大致如下：

**[plain]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8851666)

1. +----+-----+
2. |CMF|FLG|    （more-->）
3. +----+-----+
4. (if FLG.FDICT set)
5. 0     1     2     3
6. +-----+-----+-----+-----+
7. |       DICTID          | （more-->）
8. +-----+-----+-----+-----+
9. +========================
10. | ...compressed data... |
11. +========================
12. +-----+-----+-----+-----+
13. |         ADLER32       |
14. +-----+-----+-----+-----+

           注:  
                    在上面的图中，像这样的框，代表一个字节

**[plain]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8851666)

1. +---+
2. |   |  <-- 垂直的边框可能会丢掉
3. +---+

                    像这样的框，代表可变数目的字节。

**[plain]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8851666)

1. +==============+
2. |              |
3. +==============+

           比较上面的两张图，可以看出在zlib库支持的两种格式之间只有头和尾不同而已。中间的"compressed data"数据段就是所谓的raw deflate。

           raw deflate数据段使用deflate算法压缩的，而defalte算法是LZ77 压缩算法和霍夫曼编码压缩的结合。有点绕，下面把涉及到的名词列一下：  
               **LZ77**: 是一种基于字典的无损数据压缩算法(还有 LZ78, LZW 等)  
                **deflate**: 也是一种数据压缩算法，实际上就是先用 LZ77 压缩，然后用霍夫曼编码压缩  
                **gzip**: 是一种文件结构，也可以算一种压缩格式，通过 defalte 算法压缩数据，然后加上文件头和adler32校验  
                **zlib**: 是一个提供了 deflate, zlib, gzip 压缩方法的函数库；也是一种压缩格式(用 deflate 压缩数据，然后加上 zlib 头和 CRC 校验)

**5.2**[**LZ77算法**](http://www.ringsd.com/2012/01/zlib-ya-suo-suan-fa-xi-lie-1lz77-suan-fa/)

           LZ77算法是一种基于字典的压缩算法，并且是基于动态字典的压缩算法。既然存在动态词典，那么就一定也存在静态词典。先来看定义:  
          **静态字典**：预先定义好的字典。  
           **动态字典**：采用之前的数据作为字典。  
  
           那二者之间的区别又在什么地方呢？  
           1. 静态字典字典本身就是一个数据量很大的数据块，在压缩出来的数据中必须维护这样一个数据块，很大程度上会影响到压缩率。而动态压缩则不同，由于动态压缩的字典是基于已经压缩好的数据的，因此动态压缩不需要另外维护一个字典，这样就可以一定程度上保证压缩率。  
           2. 静态字典都是已经预定义好的，因此针对不同数据我们需要维护不同的字典，但是动态字典则不同，动态字典是自适应的，因此不需要去维护不同的字典，更适合我们的需求。  
           
　　LZ77算法又称之为“滑动窗口压缩”，因为该算法在原始数据上虚拟了一个滑动的窗口，这个窗口中的所有元素序列就构成了一个字典，由于窗口是滑动的，因此辞典也是实时变化的，所以这就是一个典型的动态辞典。  
  
　　LZ77压缩过程如下图：



         大致过程如下：  
         1. 从当前压缩位置开始，考察未编码的数据，并试图在滑动窗口中找出最长的匹配字符串，如果找到，则进行步骤 2，否则进行步骤 3。  
         2. 输出三元符号组 ( off, len, c )。其中 off 为窗口中匹配字符串相对窗口边界的偏移，len 为可匹配的长度，c 为下一个字符。然后将窗口向后滑动 len + 1 个字符，继续步骤 1。  
         3. 输出三元符号组 ( 0, 0, c )。其中 c 为下一个字符。然后将窗口向后滑动 len + 1 个字符，继续步骤 1。  
         这样，我们将可以匹配的字符串都变成了指向窗口内的指针，并由此完成了对上述数据的压缩。  
  
　　使用LZ77算法的解压缩的过程十分简单，只要我们向压缩时那样维护好滑动的窗口，随着三元组的不断输入，我们在窗口中找到相应的匹配串，缀上后继字符c 输出（如果 off 和 len 都为 0 则只输出后继字符 c )即可还原出原始数据。

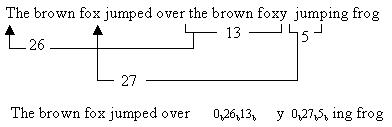
         举个例子，考虑这样一句话：

         the brown fox jumped over the brown foxy jumping frog

         这个短语的长度总共是53个八位组 = 424 bit。算法从左向右处理这个文本。初始时，每个字符被映射成9 bit的编码，二进制的1跟着该字符的8 bit ASCII码。在处理进行时，算法查找重复的序列。当碰到一个重复时，算法继续扫描直到该重复序列终止。换句话说，每次出现一个重复时，算法包括尽可能多的字符。碰到的第一个这样的序列是the brown fox。这个序列被替换成指向前一个序列的指针和序列的长度。在这种情况下，前一个序列的the brown fox出现在26个字符之前，序列的长度是13个字符。对于这个例子，假定存在两种编码选项：8 bit的指针和4 bit的长度，或者12 bit的指针和6 bit的长度。使用2 bit的首部来指示选择了哪种选项，00表示第一种选项，01表示第二种选项。因此，the brown fox的第二次出现被编码为 <00b><26d><13d >，或者00000110101101。

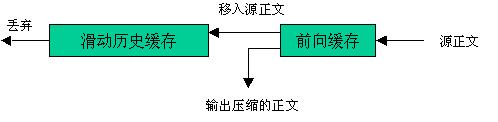
          压缩报文的剩余部分是字母y；序列<00b><27d><5d >替换了由一个空格跟着jump组成的序列，以及字符序列ing frog。

          下图演示了压缩映射的过程。压缩过的报文由35个9 bit字符和两个编码组成，总长度为35 x 9 + 2 x 14 = 343比特。和原来未压缩的长度为424比特的报文相比，压缩比为1.24。

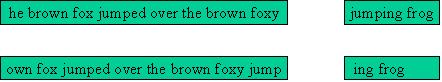


**5.2.1 压缩算法的说明**

          LZ77（及其变体）的压缩算法使用了两个缓存。滑动历史缓存包含了前面处理过的N个源字符，前向缓存包含了将要处理的下面L个字符(如下图)。算法尝试将前向缓存开始的两个或多个字符与滑动历史缓存中的字符串相匹配。如果没有发现匹配，前向缓存的第一个字符作为9 bit的字符输出并且移入滑动窗口，滑动窗口中最久的字符被移出。如果找到匹配，算法继续扫描以找出最长的匹配。然后匹配字符串作为三元组输出（指示标记、指针和长度）。对于K个字符的字符串，滑动窗口中最久的K个字符被移出，并且被编码的K个字符被移入窗口。



          下图显示了这种模式对于我们的例子的运行情况。这里假定了39个字符的滑动窗口和13个字符的前向缓存。在这个例子的上半部分，已经处理了前面的40个字符，滑动窗口中是未压缩的最近的39个字符。剩下的源字符串在前向窗口中。压缩算法确定了下一个匹配，从前向窗口将5个字符移入到滑动窗口中，并且输出了这个匹配字符串的编码。经过这些操作的缓存的状态显示在这个例子的下半部分。



          尽管LZ77是有效的，对于当前的输入情况也是合适的，但是存在一些不足。算法使用了有限的窗口在以前的文本中查找匹配，对于相对于窗口大小来说非常长的文本块，很多可能的匹配就会被丢掉。窗口大小可以增加，但这会带来两个损失：

         (1)算法的处理时间会增加，因为它必须为滑动窗口的每个位置进行一次与前向缓存的字符串匹配的工作；

         (2)<指针>字段必须更长，以允许更长的跳转。

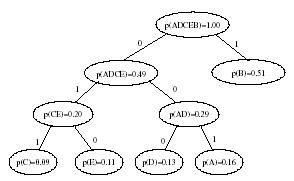
**5.2.2 编码方法**

         我们必须精心设计三元组中每个分量的表示方法，才能达到较好的压缩效果。一般来讲，编码的设计要根据待编码的数值的分布情况而定。

         对于三元组的第一个分量——**窗口内的偏移**，通常的经验是，偏移接近窗口尾部的情况要多于接近窗口头部的情况，这是因为字符串在与其接近的位置较容易找到匹配串，但对于普通的窗口大小（例如 4096 字节）来说，偏移值基本还是均匀分布的，我们完全可以用固定的位数来表示它。  
         编码 off 需要的位数 bitnum = upper\_bound( log2( MAX\_WND\_SIZE ))  
         由此，如果窗口大小为 4096，用 12 位就可以对偏移编码。如果窗口大小为 2048，用 11 位就可以了。复杂一点的程序考虑到在压缩开始时，窗口大小并没有达到 MAX\_WND\_SIZE，而是随着压缩的进行增长，因此可以根据窗口的当前大小动态计算所需要的位数，这样可以略微节省一点空间。  
         对于第二个分量——**字符串长度**，我们必须考虑到，它在大多数时候不会太大，少数情况下才会发生大字符串的匹配。显然可以使用一种变长的编码方式来表示该长度值。在前面我们已经知道，要输出变长的编码，该编码必须满足前缀编码的条件。其实 Huffman 编码也可以在此处使用，但却不是最好的选择。比较常用的是γ 编码和Golomb 编码。  
         对三元组的最后一个分量——字**符 c**，因为其分布并无规律可循，我们只能老老实实地用 8 个二进制位对其编码。

**5.3**[**霍夫曼编码**](http://jpkc.zust.edu.cn/2007/dmt/course/MMT03_02_2.htm)

         霍夫曼（Huffman）编码属于码词长度可变的编码类，是霍夫曼在1952年提出的一种编码方法，即从下到上的编码方法。同其他码词长度可变的编码一样，可区别的不同码词的生成是基于不同符号出现的不同概率。生成霍夫曼编码算法基于一种称为“编码树”（coding tree）的技术。算法步骤如下：  
         （1）初始化，根据符号概率的大小按由大到小顺序对符号进行排序。  
         （2）把概率最小的两个符号组成一个新符号（节点），即新符号的概率等于这两个符号概率之和。  
         （3）重复第2步，直到形成一个符号为止（树），其概率最后等于1。  
         （4）从编码树的根开始回溯到原始的符号，并将每一下分枝赋值为1，上分枝赋值为0。  
         以下这个简单例子说明了这一过程。  
         1)．字母A,B,C,D,E已被编码，相应的出现概率如下：  
              p(A)=0.16, p(B)=0.51, p(C)=0.09, p(D)=0.13, p(E)=0.11  
         2)．C和E概率最小，被排在第一棵二叉树中作为树叶。它们的根节点CE的组合概率为0.20。从CE到C的一边被标记为1，从CE到E的一边被标记为0。这种标记是强制性的。所以，不同的哈夫曼编码可能由相同的数据产生。  
         3)．各节点相应的概率如下：  
             p(A)=0.16, p(B)=0.51, p(CE)=0.20, p(D)=0.13  
             D和A两个节点的概率最小。这两个节点作为叶子组合成一棵新的二叉树。根节点AD的组合概率为0.29。由AD到A的一边标记为1，由AD到D的一边标记为0。  
             如果不同的二叉树的根节点有相同的概率，那么具有从根到节点最短的最大路径的二叉树应先生成。这样能保持编码的长度基本稳定。  
         4)．剩下节点的概率如下：  
            p(AD)=0.29, p(B)=0.51, p(CE)=0.20  
            AD和CE两节点的概率最小。它们生成一棵二叉树。其根节点ADCE的组合概率为0.49。由ADCE到AD一边标记为0，由ADCE到CE的一边标记为1。  
         5)．剩下两个节点相应的概率如下：  
            p(ADCE)=0.49, p(B)=0.51  
            它们生成最后一棵根节点为ADCEB的二叉树。由ADCEB到B的一边记为1，由ADCEB到ADCE的一边记为0。  
         6)．下图为霍夫曼编码。编码结果被存放在一个表中：  
             w(A)=001, w(B)=1, w(C)=011, w(D)=000, w(E)=010



**5.4 CRC循环冗余校验**

         循环冗余校验（英语：Cyclic redundancy check，通称“CRC”）是一种根据网络数据数据包或电脑文件等数据产生简短固定位数校验码的一种散列函數，主要用来检测或校验数据传输或者保存后可能出现的错误。生成的数字在传输或者存储之前计算出来并且附加到数据后面，然后接收方进行检验确定数据是否发生变化。一般来说，循环冗余校验的值都是32位的整数。由于本函数易于用二进制的电脑硬件使用、容易进行数学分析并且尤其善于检测传输通道干扰引起的错误，因此获得广泛应用。具体算法可以看<[循环冗余校验（CRC）算法入门引导](http://blog.csdn.net/liyuanbhu/article/details/7882789)>

**5.5 ADLER32校验**

         Adler-32通过求解两个16位的数值A、B实现，并将结果连结成一个32位整数。其算法描述如下：  
                  A = 1 + D1 + D2 + ... + Dn (mod 65521)  
                  B = (1 + D1) + (1 + D1 + D2) + ... + (1 + D1 + D2 + ... + Dn) (mod 65521) = n×D1 + (n-1)×D2 + (n-2)×D3 + ... + Dn + n (mod 65521)  
                  Adler-32(D) = B × 65536 + A

         举个例子：  
         ASCII code           A                                      B  
         （10进制）    
            W :  87           1 +  87 =  88              0 +  88 =   88  
            i   :  105         88 + 105 = 193         88 + 193 =  281  
            k  :  107         193 + 107 = 300       281 + 300 =  581  
            i   :  105         300 + 105 = 405       581 + 405 =  986  
            p  :  112         405 + 112 = 517       986 + 517 = 1503  
            e  :  101         517 + 101 = 618       1503 + 618 = 2121  
            d  :  100         618 + 100 = 718       2121 + 718 = 2839  
            i   :  105         718 + 105 = 823       2839 + 823 = 3662  
            a  :   97          823 +  97 = 920        3662 + 920 = 4582  
  
         经过上面的一轮运算：  
          A = 920    
          B = 4582  
          Adler-32(D)  = B × 65536 + A = 4582× 65536 +920 = 300286872  
  
          300286872 转化成十六进制 11E60398(此数即是输出的结果)。

**5.6 Zlib流类**

          在Poco库中，存在两个流类用来把原始数据转换成为压缩数据，分别是Poco::InflatingInputStream和Poco::InflatingOutputStream。这两个类通过另外一个输入、输出流构建，并可以指定数据的压缩格式。

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8851666)

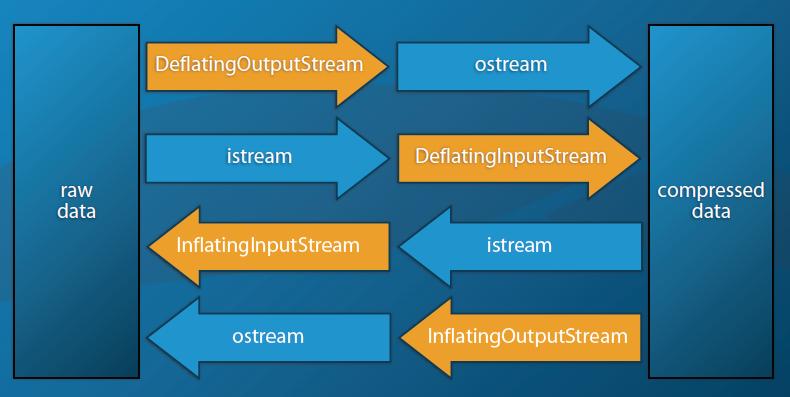
1. Poco::DeflatingStreamBuf::STREAM\_ZLIB (deflate/zlib 格式)
2. Poco::DeflatingStreamBuf::STREAM\_GZIP (gzip 格式)

          而Poco::InflatingInputStream类和Poco::InflatingOutputStream类，用来把压缩数据还原成为原始数据。同样的它们也是通过另外一个输入、输出流构建，并可以指定数据的压缩格式。

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8851666)

1. Poco::InflatingStreamBuf::STREAM\_ZLIB (deflate/zlib 格式)
2. Poco::InflatingStreamBuf::STREAM\_GZIP (gzip 格式)

          原始数据和压缩数据之间的转换过程可见下图：



          下面来看一个例子：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8851666)

1. #include "Poco/DeflatingStream.h"
2. #include "Poco/InflatingStream.h"
3. #include <fstream>
4. **using** Poco::DeflatingOutputStream;
5. **using** Poco::DeflatingStreamBuf;
6. **using** Poco::InflatingInputStream;
7. **using** Poco::InflatingStreamBuf;
9. **int** main(**int** argc, **char**\*\* argv)
10. {
11. std::ofstream ostr("test.gz", std::ios::binary);
12. DeflatingOutputStream deflater(ostr, DeflatingStreamBuf::STREAM\_GZIP);
13. deflater << "Hello, world! ads \n" << "12306";
14. // ensure buffers get flushed before connected stream is closed
15. deflater.close();
16. ostr.close();
18. std::ifstream istr("test.gz", std::ifstream::in);
19. InflatingInputStream infalter(istr ,InflatingStreamBuf::STREAM\_GZIP);
20. string str;
21. **while** (infalter.good())          // loop while extraction from file is possible
22. {
23. **char** c = infalter.get();    // get character from file
24. str.append(1, c);
25. }
27. **return** 0;
28. }

          在上面那个例子里，使用析取操作符">>"，会使空格和换行符丢失，所以只能逐一获取字符。

**6. Poco::CountingInputStream 和 Poco::CountingOutputStream**

          Poco::CountingInputStream和Poco::CountingOutputStream类可以用来计算文件中的字符数目和行数，同时它们也可以提供当前所在行和列数。下面是一个例子：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8851666)

1. #include "Poco/CountingStream.h"
2. #include <assert.h>
4. **using** Poco::CountingInputStream;
5. **using** Poco::CountingOutputStream;
7. **int** \_tmain(**int** argc, \_TCHAR\* argv[])
8. {
9. **char** c;
10. std::istringstream istr1("foo");
11. CountingInputStream ci1(istr1);
12. **while** (ci1.good()) ci1.get(c);
13. assert (ci1.lines() == 1);
14. assert (ci1.chars() == 3);
15. assert (ci1.pos() == 3);
17. std::ostringstream ostr2;
18. CountingOutputStream co2(ostr2);
19. co2 << "foo\nbar";
20. assert (ostr2.str() == "foo\nbar");
21. assert (co2.lines() == 2);
22. assert (co2.chars() == 7);
23. assert (co2.pos() == 3);
25. **return** 0;
26. }

**7. Poco::InputLineEndingConverter 和 Poco::OutputLineEndingConverter**

          Poco::InputLineEndingConverter和Poco::OutputLineEndingConverter用来转换不同操作系统上的文本文件的行结尾(Unix (LF)、DOS/Windows (CRLF)、Macintosh (CR))。下面是一个例子：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8851666)

1. #include "Poco/LineEndingConverter.h"
2. #include "Poco/StreamCopier.h"
3. #include <sstream>
5. **using** Poco::LineEnding;
6. **using** Poco::InputLineEndingConverter;
7. **using** Poco::OutputLineEndingConverter;
8. **using** Poco::StreamCopier;
10. **int** \_tmain(**int** argc, \_TCHAR\* argv[])
11. {
12. std::istringstream input("line1\r\nline2\r\nline3\r\n");
13. std::ostringstream output;
14. InputLineEndingConverter conv(input, LineEnding::NEWLINE\_LF);
15. StreamCopier::copyStream(conv, output);
16. std::string result = output.str();
17. assert (result == "line1\nline2\nline3\n");
19. **return** 0;
20. }

**8. Poco::TeeInputStream 和 Poco::TeeOutputStream**

          Poco::TeeInputStream和Poco::TeeOutputStream流会拷贝所有的输入和输出字符串，并把它们输出到一个或多个流上。这两个类主要被用于调试情况下。

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8851666)

1. #include "Poco/TeeStream.h"
2. #include <iostream>
3. #include <fstream>
4. **using** Poco::TeeOutputStream;
5. **int** main(**int** argc, **char**\*\* argv)
6. {
7. TeeOutputStream tee(std::cout);
8. std::ofstream fstr("output.txt");
9. tee.addStream(fstr);
10. tee << "Hello, world!" << std::endl;
11. **return** 0;
12. }

**9. Poco::NullOutputStream**

          Poco::NullOutputStream类会抛弃所有对它的写操作，对所有的读操作会返回文件结束符。下面是一个例子：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8851666)

1. #include <assert.h>
2. #include "Poco/NullStream.h"
4. **using** Poco::NullInputStream;
5. **using** Poco::NullOutputStream;
7. **int** main(**int** argc, **char**\*\* argv)
8. {
9. NullInputStream istr;
10. assert (istr.good());
11. assert (!istr.eof());
12. **int** c = istr.get();
13. assert (c == -1);
14. assert (istr.eof());
15. **return** 0;
16. }

**10. Poco::BinaryWriter和Poco::BinaryReader**

**10. 1 Poco::BinaryWriter**

            Poco::BinaryWriter被用作向一个输出流中写入基础数据类型的二进制形式。Poco::BinaryReader被用作从一个保存着Poco::BinaryWriter类存储的二进制形式的内容的输入流中读取基础数据类型的数据。  
            Poco::BinaryReader和Poco::BinaryWriter类都支持大小字节序的数据读写，并且能够在内部转换所需的字节序。这两个类主要用于不同架构的操作系统之间的数据转换。  
  
            Poco::BinaryWriter支持所有c++内建数据类型、c风格的字符串和std::string的插入操作符"<<"。在Poco::BinaryWriter类中无符号整型(包括32和64位)的存储是以一种特殊的7位编码方式进行的。  
  
            **成员函数:**  
            1.void write7BitEncoded(UInt32 value)  
                void write7BitEncoded(UInt64 value)  
              向底层输出流以简洁7位编码方式写入一个无符号整形  
            2. void writeRaw(const std::string& rawData)  
              向底层输出流写入裸数据  
            3. void writeBOM()  
              向流中写入一个字节序标志BOM(byte order mark)(一个16位值0xFEFF的本机序)。如果需要的话，BinaryReader会使用BOM标志自动进行字节序转换  
  
         **BinaryWriter类和字节序：**  
            BinaryWriter类通过一个输出流和一个字节序标志进行构造。字节序标志有如下选择：  
              NATIVE\_BYTE\_ORDER           (本机序)  
              BIG\_ENDIAN\_BYTE\_ORDER      (大头序)  
              NETWORK\_BYTE\_ORDER          (网络序)  
              LITTLE\_ENDIAN\_BYTE\_ORDER    (小头序)  
  
           **BinaryWriter流状态**  
            Poco::BinaryWriter提供了一些函数用于判断底层输出流的状态  
            1. void flush()  
            刷新底层流  
            2. bool good()  
            如果流状态正常返回真  
            3. bool fail()  
            如果底层流状态为fail返回真  
            4. bool bad()  
            如果底层流状态为bad返回真

**10.2 Poco::BinaryReader**

            Poco::BinaryReader为所有c++内置数据和std::string提供析取操作符。  
  
            **成员函数：**  
            1. void read7BitEncoded(UInt32& value)  
                void read7BitEncoded(UInt64& value)  
               读取以7位压缩格式存储的整形值。  
            2. void readRaw(int length, std::string& value)  
               读取指定长度的裸数据  
            3. void readBOM()  
               读取字节序标志BOM(byte order mark)，以便在内部自动激活字节序转换  
  
            **Poco::BinaryReader流状态**  
            Poco::BinaryReader提供了一些函数用于判断底层输出流的状态  
            1. bool good()  
            如果流状态正常返回真  
            2. bool fail()  
            如果底层流状态为fail返回真  
            3. bool bad()  
            如果底层流状态为bad返回真  
            4. bool eof()  
            如果底层状态为eof返回真

            下面是两个例子：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8851666)

1. #include <fstream>
2. **using** Poco::BinaryWriter;
3. **int** main(**int** argc, **char**\*\* argv)
4. {
5. std::ofstream ostr("binary.dat", std::ios::binary);
6. BinaryWriter writer(ostr);
7. writer.writeBOM();
8. writer << "Hello, world!" << 42;
9. writer.write7BitEncoded((Poco::UInt32)123);
10. writer << **true**;
11. **return** 0;
12. }

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8851666)

1. #include "Poco/BinaryReader.h"
2. #include <fstream>
3. **using** Poco::BinaryReader;
4. **int** main(**int** argc, **char**\*\* argv)
5. {
6. std::ifstream istr("binary.dat", std::ios::binary);
7. BinaryReader reader(istr);
8. reader.readBOM();
9. std::string hello;
10. Poco::UInt32 i;
11. **bool** b;
12. reader >> hello >> i;
13. reader.read7BitEncoded(i);
14. reader >> b;
15. **return** 0;
16. }

**11. FileStream, FileInputStream, FileOutputStream**

          FileStream、FileInputStream、FileOutputStream流用于读取和写入文件。在windows平台上，传给文件流的路径必须是UTF-8编码。Poco中文件流类总是以二进制方式打开文件，并且支持文件的seek操作。下面是一个例子：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8851666)

1. #include <assert.h>
2. #include "Poco/FileStream.h"
3. #include "Poco/File.h"
4. #include "Poco/TemporaryFile.h"
5. #include "Poco/Exception.h"
7. **int** main(**int** argc, **char**\*\* argv)
8. {
9. Poco::FileOutputStream ostr("test.txt");
10. ostr << "0123456789";
11. ostr.close();
13. Poco::FileStream str1("test.txt", std::ios::ate);
14. **int** c = str1.get();
15. assert (str1.eof());
17. str1.clear();
18. str1.seekg(0);
19. c = str1.get();
20. assert (c == '0');
22. str1.close();
24. Poco::FileStream str2("test.txt", std::ios::ate);
25. str2 << "abcdef";
26. str2.seekg(0);
27. std::string s;
28. str2 >> s;
29. assert (s == "0123456789abcdef");
30. str2.close();
32. **return** 0;
33. }

**参考文章：**

      1. [Zlib压缩算法系列1：LZ77算法](http://www.ringsd.com/2012/01/zlib-ya-suo-suan-fa-xi-lie-1lz77-suan-fa/)

      2. [词典编码](http://jpkc.zust.edu.cn/2007/dmt/course/MMT03_05_2.htm)

      3.[LZ77算法](http://blog.csdn.net/zhenanlin/article/details/8731209)

      4. [笨笨数据压缩教程](http://blog.csdn.net/erikliu/article/category/762206)

      5. [GZIP源代码分析](http://blog.csdn.net/dummytea/article/details/8275484)

      6. [霍夫曼编码](http://jpkc.zust.edu.cn/2007/dmt/course/MMT03_02_2.htm)

      7. [循环冗余校验（CRC）算法入门引导](http://blog.csdn.net/liyuanbhu/article/details/7882789)

（版权所有，转载时请注明作者和出处  <http://blog.csdn.net/arau_sh/article/details/8851666>）