**POCO C++库学习和分析 -- 流 （二）**

**2. 创建自己的流**

**2.1 自定义流的结构**

          在Poco中提供了一个框架用于创建自己的流，并且创建的流都符合C++标准。想一下标准库中流的层次和结构。每一个流都必须有对应的流缓冲，并且在流初始化时提供此流缓冲的指针。Poco中提供了3种流缓冲供选择，分别是BasicBufferedStreamBuf、BasicUnbufferedStreamBuf、BasicBufferedBidirectionalStreamBuf。首先来看一个例子：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8851662)

1. #include "Poco/UnbufferedStreamBuf.h"
2. #include <ostream>
3. #include <cctype>
4. **class** UpperStreamBuf: **public** UnbufferedStreamBuf
5. {
6. **public**:
7. UpperStreamBuf(std::ostream& ostr): \_ostr(ostr)
8. {}
9. **protected**:
10. **int** writeToDevice(**char** c)
11. {
12. \_ostr.put(toupper(c));
13. **return** charToInt(c);
14. }
15. **private**:
16. std::ostream& \_ostr;
17. };
19. **class** UpperIOS: **public** **virtual** std::ios
20. {
21. **public**:
22. UpperIOS(std::ostream& ostr): \_buf(ostr)
23. {
24. poco\_ios\_init(&\_buf);
25. }
26. **protected**:
27. UpperStreamBuf \_buf;
28. };
30. **class** UpperOutputStream: **public** UpperIOS, **public** std::ostream
31. {
32. **public**:
33. UpperOutputStream(std::ostream& ostr):
34. UpperIOS(ostr),
35. std::ostream(&\_buf)
36. {}
37. };
39. **int** main(**int** argc, **char**\*\* argv)
40. {
41. UpperOutputStream upper(std::cout);
42. upper << "Hello, world!" << std::endl;
43. **return** 0;
44. }

          从上面的例子中，我们看到用户自定流缓冲类型UpperStreamBuf，并自定义流类型UpperIOS和UpperOutputStream。其中UpperIOS从std::ios虚拟继承。注意这里是虚拟继承，这是为了保证UpperOutputStream多重继承时的菱形结构(std::ostream也是从std::ios虚拟继承的)。再回忆一下，std::ios的作用，其中最重要的就是定义了缓冲区指针。那么很自然的，定义UpperIOS，并让UpperOutputStream从UpperIOS继承的原因，就是为了让UpperOutputStream类拥有自己的缓冲区。再来看从流std::ostream继承的原因，很简单，为了继承"<<"操作符。那么清楚了，如果是输入流，从std::istream继承；是输入输出流从std::iostream继承。

          再从上面的例子看：为什么说，Poco中流框架提供的是一个中介流的框架。当创建UpperOutputStream对象时，我们看它做了什么.  
          在调用UpperOutputStream upper(std::cout)时，传入参数是一个std::cout，这是一个已定义的流对象。这个流对象被赋予给UpperIOS类，用来初始化UpperIOS类中的UpperStreamBuf对象；而UpperStreamBuf又并不负责直接输出，它的输出依赖于内部的std::ostream对象，在这里就是std::cout。另一方面由于UpperOutputStream对象同时也继承自std::ostream，而std::ostream必须要求用一个流缓冲对象来初始化，所以在UpperIOS类中的UpperStreamBuf对象初始化后，又被赋给std::ostream对象用作初始化。  
  
          再来看 "upper << "Hello, world!" << std::endl "语句的执行过程：  
          1. "<<"操作符是std::ostream的成员函数。也就是说 "Hello, world!"被首先调用了std::ostream函数"<<"。

          2.  我们知道，流的输入输出是委托给其内部的缓冲区对象的。也就是说，在这里"<<"操作符，会委托到std::ostream类内部关联的流缓冲UpperIOS::UpperStreamBuf对象\_buf上。  
          2. UpperStreamBuf类型的\_buf对象并不输出，它把输出任务继续委托给内部的流对象\_ostr。在程序里，\_ost对象为std::cout。  
          3. std::cout对象在收到数据后，委托给std::cout内部关联的流缓冲对象，最终输出。至于std::cout对象关联的流对象是什么，那就不用去管了。这是系统内部实现的。  
  
          转了半天，输出最终还是依赖于系统的默认输出。如果仅是如此，就没有意思了。Poco流在类BasicBufferedStreamBuf、BasicUnbufferedStreamBuf、BasicBufferedBidirectionalStreamBuf上开了一个小口，供用户自定义中间操作。这两个小口就是:

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8851662)

1. **virtual** **int** readFromDevice(char\_type\* buffer, std::streamsize length)；
2. **virtual** **int** writeToDevice(**const** char\_type\* buffer, std::streamsize length)；

          通过子类继承，用户可以自定义转换行为。任何从Poco流自定义框架出来的类都必须重新定义这两个虚函数。readFromDevice和writeToDevice会在std::ostream中的虚函数overflow和underflow中被调用。  
  
          在了解了Poco自定义流类的流转次序后，再去看Poco中的自定义流Base64Decoder，Base64Encoder等的数据流转，会发现过程都是类似的。下面就不再对这一方面进行叙述。只是讲一下Poco中已定义的3个流缓冲类BasicBufferedStreamBuf、BasicUnbufferedStreamBuf、BasicBufferedBidirectionalStreamBuf。

**2.2 BasicUnbufferedStreamBuf**

          Poco::BasicUnbufferedStreamBuf是一个使用char作为模板参数类型的偏特化模板类。其完整定义为：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8851662)

1. **template** <**typename** ch, **typename** tr>
2. **class** BasicUnbufferedStreamBuf: **public** std::basic\_streambuf<ch, tr>
3. /// This is an implementation of an unbuffered streambuf
4. /// that greatly simplifies the implementation of
5. /// custom streambufs of various kinds.
6. /// Derived classes only have to override the methods
7. /// readFromDevice() or writeToDevice().
8. {
9. // .....
10. int\_type \_pb;
11. **bool**     \_ispb;
12. }
14. **typedef** BasicUnbufferedStreamBuf<**char**, std::char\_traits<**char**> > UnbufferedStreamBuf;

          Poco::BasicUnbufferedStreamBuf可以说是最简单的创建用户自定流的方法，在其内部没有任何缓冲对象。从Poco::BasicUnbufferedStreamBuf继承的类需要重载readFromDevice和writeToDevice接口。  
  
          对应重载的要求如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8851662)

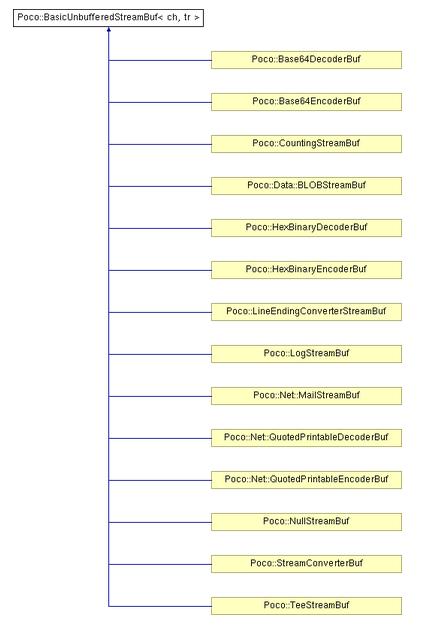
1. **int** readFromDevice()

          读取并返回单字节无符号字符。如果没有更多的合适数据时，返回char\_traits::eof()(-1)。需要注意的是，不要直接返回一个char类型值，因为char类型值是一个有符号数。调用int charToInt(char c)返回一个char转换到int的值。

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8851662)

1. **int** writeToDevice(**char** c)

          写入一个单字节字符，如果成功返回写入的字符(已整形方式)，否则返回char\_traits::eof() (-1).  
  
          在Poco中从UnbufferedStreamBuf继承的类见下图:



**2.3 BasicBufferedStreamBuf**

          Poco::BasicBufferedStreamBuf也同样是使用char作为模板参数类型的偏特化模板类。其完整定义为：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8851662)

1. **template** <**typename** ch, **typename** tr, **typename** ba = BufferAllocator<ch> >
2. **class** BasicBufferedStreamBuf: **public** std::basic\_streambuf<ch, tr>
3. /// This is an implementation of a buffered streambuf
4. /// that greatly simplifies the implementation of
5. /// custom streambufs of various kinds.
6. /// Derived classes only have to override the methods
7. /// readFromDevice() or writeToDevice().
8. ///
9. /// This streambuf only supports unidirectional streams.
10. /// In other words, the BasicBufferedStreamBuf can be
11. /// used for the implementation of an istream or an
12. /// ostream, but not for an iostream.
13. {
14. // ...
15. std::streamsize \_bufsize;
16. char\_type\*      \_pBuffer;
17. openmode        \_mode;
18. }
20. **typedef** BasicBufferedStreamBuf<**char**, std::char\_traits<**char**> > BufferedStreamBuf;

          在BasicBufferedStreamBuf内部存在一个\_pBuffer，用作缓冲，供输出使用。Poco::BufferedStreamBuf的实例支持读写，但不支持同时读写。从Poco::BasicBufferedStreamBuf继承的类需要重载readFromDevice和writeToDevice接口。  
  
          对应重载的要求如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8851662)

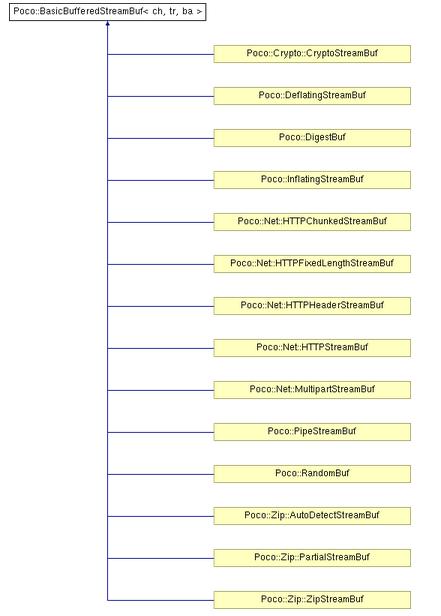
1. **int** readFromDevice(**char**\* buffer, std::streamsize length)

          读取指定长度的字符，并把它们放置在buffer中。返回读到的字节数，或者-1(当错误发生时)

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8851662)

1. **int** writeToDevice(**const** **char**\* buffer, std::streamsize length)

          写入buffer中指定数目的字节。返回读到的字节数，或者-1(当错误发生时)  
  
  
          在Poco中从BasicBufferedStreamBuf继承的类见下图:



**2.4 BasicBufferedBidirectionalStreamBuf**

          Poco::BasicBufferedBidirectionalStreamBuf一样是使用char作为模板参数类型的偏特化模板类。其完整定义为：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8851662)

1. **template** <**typename** ch, **typename** tr, **typename** ba = BufferAllocator<ch> >
2. **class** BasicBufferedBidirectionalStreamBuf: **public** std::basic\_streambuf<ch, tr>
3. /// This is an implementation of a buffered bidirectional
4. /// streambuf that greatly simplifies the implementation of
5. /// custom streambufs of various kinds.
6. /// Derived classes only have to override the methods
7. /// readFromDevice() or writeToDevice().
8. ///
9. /// In contrast to BasicBufferedStreambuf, this class supports
10. /// simultaneous read and write access, so in addition to
11. /// istream and ostream this streambuf can also be used
12. /// for implementing an iostream.
13. {
14. // ....
15. std::streamsize \_bufsize;
16. char\_type\*      \_pReadBuffer;
17. char\_type\*      \_pWriteBuffer;
18. openmode        \_mode;
19. }
21. **typedef** BasicBufferedBidirectionalStreamBuf<**char**, std::char\_traits<**char**> > BufferedBidirectionalStreamBuf;

          在BasicBufferedBidirectionalStreamBuf内部存在两个缓冲，\_pReadBuffer和\_pWriteBuffer，作为缓冲，分别供输入或输出使用。  
          Poco::BasicBufferedBidirectionalStreamBuf的实例支持同时读写。从Poco::BasicBufferedBidirectionalStreamBuf继承的类需要重载readFromDevice和writeToDevice接口。  
  
          对应重载的要求如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8851662)

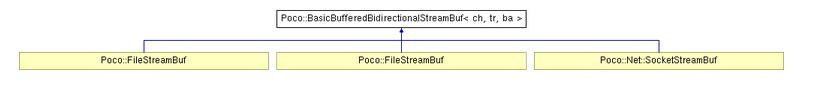
1. **int** readFromDevice(**char**\* buffer, std::streamsize length)

          读取指定长度的字符，并把它们放置在buffer中。返回读到的字节数，或者-1(当错误发生时)

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8851662)

1. **int** writeToDevice(**const** **char**\* buffer, std::streamsize length)

          写入buffer中指定数目的字节。返回读到的字节数，或者-1(当错误发生时)  
  
          在Poco中从BasicBufferedBidirectionalStreamBuf继承的类见下图:



**3. Poco::Base64Encoder和Poco::Base64Decoder**

**3.1 Base64编码的概念**

          关于Base64编码的概念的概念出自[wiki百科](http://zh.wikipedia.org/wiki/Base64)。想要得到更加详细的资料，可以去看[RFC 4648](http://datatracker.ietf.org/doc/rfc4648/?include_text=1)。

          Base64是一种基于64个可打印字符来表示二进制数据的表示方法。由于2的6次方等于64，所以每6个位元为一个单元，对应某个可打印字符。三个字节有24个位元，对应于4个Base64单元，即3个字节需要用4个可打印字符来表示。它可用来作为电子邮件的传输编码。在Base64中的可打印字符包括字母A-Z、a-z、数字0-9 ，这样共有62个字符，此外两个可打印符号在不同的系统中而不同。  
          Base64常用于在通常处理文本数据的场合，表示、传输、存储一些二进制数据。包括MIME的email，email via MIME, 在XML中存储复杂数据.

**Base64索引表：**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Value** | **Char** |  | **Value** | **Char** |  | **Value** | **Char** |  | **Value** | **Char** |
| 0 | A | 16 | Q | 32 | g | 48 | w |
| 1 | B | 17 | R | 33 | h | 49 | x |
| 2 | C | 18 | S | 34 | i | 50 | y |
| 3 | D | 19 | T | 35 | j | 51 | z |
| 4 | E | 20 | U | 36 | k | 52 | 0 |
| 5 | F | 21 | V | 37 | l | 53 | 1 |
| 6 | G | 22 | W | 38 | m | 54 | 2 |
| 7 | H | 23 | X | 39 | n | 55 | 3 |
| 8 | I | 24 | Y | 40 | o | 56 | 4 |
| 9 | J | 25 | Z | 41 | p | 57 | 5 |
| 10 | K | 26 | a | 42 | q | 58 | 6 |
| 11 | L | 27 | b | 43 | r | 59 | 7 |
| 12 | M | 28 | c | 44 | s | 60 | 8 |
| 13 | N | 29 | d | 45 | t | 61 | 9 |
| 14 | O | 30 | e | 46 | u | 62 | + |
| 15 | P | 31 | f | 47 | v | 63 | / |

    Base64是一种基于64个可打印字符来表示二进制数据的表示方法。由于2的6次方等于64，所以每6个位元为一个单元，对应某个可打印字符。三个字节有24个位元，对应于4个Base64单元，即3个字节需要用4个可打印字符来表示。它可用来作为电子邮件的传输编码。在Base64中的可打印字符包括字母A-Z、a-z、数字0-9 ，这样共有62个字符，此外两个可打印符号在不同的系统中而不同。

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **文本（1 Byte）** | **A** | | | | | | | |  | | | | | | | |  | | | | | | | |
| **二进制位** | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **二进制位（补0）** | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | **0** | **0** | **0** | **0** |  |  |  |  |  |  |  |  |  |  |  |  |
| **Base64编码** | **Q** | | | | | | **Q** | | | | | |  | | | | | |  | | | | | |
| **文本（2 Byte）** | **B** | | | | | | | | **C** | | | | | | | |  | | | | | | | |
| **二进制位** | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |  |  | x | x | x | x | x | x |
| **二进制位（补0）** | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | **0** | **0** | x | x | x | x | x | x |
| **Base64编码** | **Q** | | | | | | **k** | | | | | | **M** | | | | | |  | | | | | |

**例子：**  
          举例来说，一段引用自托马斯·霍布斯的利维坦的文句：

**[html]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8851662)

1. Man is distinguished, not only by his reason, but by this singular passion from other animals, which is a lust of the mind, that by a perseverance of delight in the continued and indefatigable generation of knowledge, exceeds the short vehemence of any carnal pleasure.

          经过base64编码之后变成：

**[html]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8851662)

1. TWFuIGlzIGRpc3Rpbmd1aXNoZWQsIG5vdCBvbmx5IGJ5IGhpcyByZWFzb24sIGJ1dCBieSB0aGlz
2. IHNpbmd1bGFyIHBhc3Npb24gZnJvbSBvdGhlciBhbmltYWxzLCB3aGljaCBpcyBhIGx1c3Qgb2Yg
3. dGhlIG1pbmQsIHRoYXQgYnkgYSBwZXJzZXZlcmFuY2Ugb2YgZGVsaWdodCBpbiB0aGUgY29udGlu
4. dWVkIGFuZCBpbmRlZmF0aWdhYmxlIGdlbmVyYXRpb24gb2Yga25vd2xlZGdlLCBleGNlZWRzIHRo
5. ZSBzaG9ydCB2ZWhlbWVuY2Ugb2YgYW55IGNhcm5hbCBwbGVhc3VyZS4=

         **编码“Man”**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **文本** | **M** | | | | | | | | **a** | | | | | | | | **n** | | | | | | | |
| **ASCII编码** | 77 | | | | | | | | 97 | | | | | | | | 110 | | | | | | | |
| **二进制位** | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| **索引** | 19 | | | | | | 22 | | | | | | 5 | | | | | | 46 | | | | | |
| **Base64编码** | **T** | | | | | | **W** | | | | | | **F** | | | | | | **u** | | | | | |

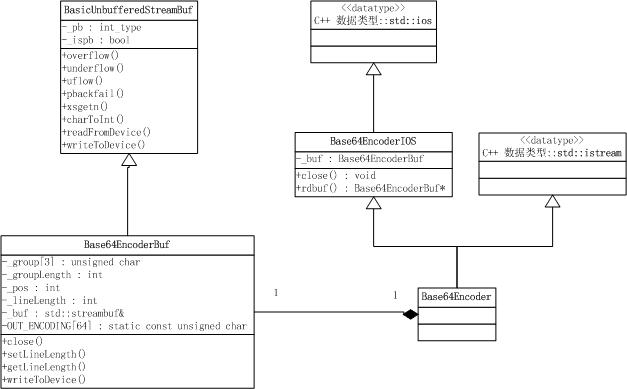
          在此例中，Base64算法将三个字符编码为4个字符

          基本上说，如果那天你看到一段以“=”或者"=="结束的字符，你就可以判断这段字符用了Base64编码。就应用而言，我只在邮件内容发送和MD5 或者SHA密码保存时见过Base64编码。

         实现上Base64算法很简单，把流变成二进制，然后按6位转换为字符保存。

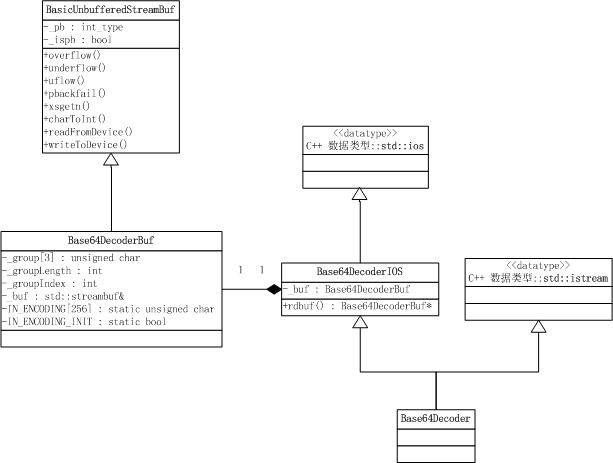
**3.2 Poco::Base64Encode**

          Poco::Base64Encode的类图如下:



**3.3 Poco::Base64Decoder**

          Poco::Base64Decoder的类图如下:



**3.4 例子**

          下面是一个例子

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8851662)

1. #include "Poco/Base64Encoder.h"
2. #include <iostream>
3. **using** Poco::Base64Encoder;
4. **int** main(**int** argc, **char**\*\* argv)
5. {
6. Base64Encoder encoder(std::cout);
7. encoder << "Hello, world!";
8. **return** 0;
9. }

**4. Poco::HexBinaryEncoder和Poco::HexBinaryDecoder**

**4.1 HexBinary编码编码概念**

          HexBinary编码又叫Base 16编码，Base16是一种基于16个可打印字符来表示二进制数据的表示方法。由于2的4次方等于16，所以每4个位元为一个单元，对应某个可打印字符。三个字节有24个位元，对应于6个Base64单元，即3个字节需要用6个可打印字符来表示。

**Base16索引表：**

The Base 16 Alphabet

Value Encoding Value Encoding Value Encoding Value Encoding

0 0 4 4 8 8 12 C

1 1 5 5 9 9 13 D

2 2 6 6 10 A 14 E

3 3 7 7 11 B 15 F

           还是以编码“Man”举例：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **文本** | **M** | | | | | | | | **a** | | | | | | | | **n** | | | | | | | |
| **ASCII编码** | 77 | | | | | | | | 97 | | | | | | | | 110 | | | | | | | |
| **二进制位** | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| **索引** | 4 | | | | 13 | | | | 6 | | | | 1 | | | | 6 | | | | 14 | | | |
| **base 16编码** | **4** | | | | **D** | | | | **6** | | | | **1** | | | | **6** | | | | **E** | | | |

         实现上Base16算法很简单，把流变成二进制，然后按4位转换为字符保存。

**4.2 Poco::HexBinaryEncoder和Poco::HexBinaryDecoder**

Poco::HexBinaryEncoder 和Poco::HexBinaryDecoder的类图同Poco::Base64Encoder和Poco::Base64Decoder是类似。

       HexBinaryDecoderBuf定义如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8851662)

1. **class** Foundation\_API HexBinaryDecoderBuf: **public** UnbufferedStreamBuf
2. /// This streambuf decodes all hexBinary-encoded data read
3. /// from the istream connected to it.
4. /// In hexBinary encoding, each binary octet is encoded as a character tuple,
5. /// consisting of two hexadecimal digits ([0-9a-fA-F]) representing the octet code.
6. /// See also: XML Schema Part 2: Datatypes (http://www.w3.org/TR/xmlschema-2/),
7. /// section 3.2.15.
8. ///
9. /// Note: For performance reasons, the characters
10. /// are read directly from the given istream's
11. /// underlying streambuf, so the state
12. /// of the istream will not reflect that of
13. /// its streambuf.
14. {
15. **public**:
16. HexBinaryDecoderBuf(std::istream& istr);
17. ~HexBinaryDecoderBuf();
19. **private**:
20. **int** readFromDevice();
21. **int** readOne();
23. std::streambuf& \_buf;
24. };

       构造文件为：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8851662)

1. HexBinaryDecoderBuf::HexBinaryDecoderBuf(std::istream& istr):
2. \_buf(\*istr.rdbuf())
3. {
4. }

       我们看到 HexBinaryDecoderBuf直接使用了istream对象内部的缓冲作为实现。同2.1节中的例子相比，可以减少函数调用的流程.

（版权所有，转载时请注明作者和出处  <http://blog.csdn.net/arau_sh/article/details/8851662>）