std::streambuf从示例到应用

2017-05-14 #streambuf

关于`streambuf`的资料并不多，IO Streams作者Jerry Schwarz这样说道

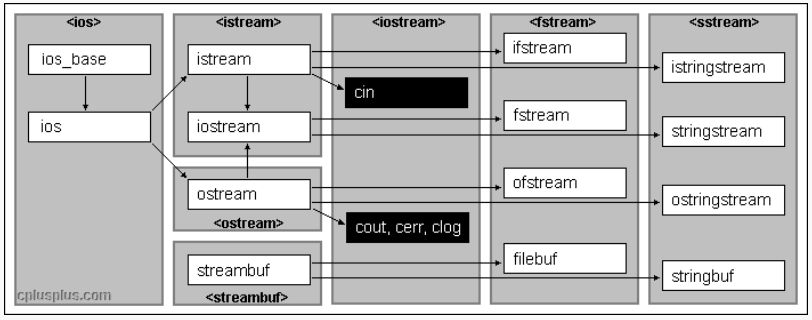
> “A major goal in my original design was that it be extensible in interesting ways. In particular, in the stream library the streambuf class was an implementation detail, but in the

> iostream library I intended it to be a usable class in its own right. I was hoping for the promulgation of many streambufs with varied functionality. I wrote a few myself, but almost no one else did. I answered many more questions of the form “how do I make my numbers look like this” than “how do I write a streambuf”. And textbook authors also tended to ignore streambufs. Apparently they did not share my view that the architecture of the input/output library was an interesting case study.”

本文试图从`streambuf`最基本的概念讲起，从简单的例子到实际项目里的用法。

1. streambuf类简介

先看下STL流相关的类关系：



流(streams)是STL里一个很重要的概念，例如`std::cin std::cout`用于终端的输入/输出。而实际

上，真正的读/写操作并不是`stream`完成的，而是由`stream`调用`stream buffer`完成。

用《The C++ Standard Library 》的原文来说：

> the actual reading and writing is not done by the streams directly, but is delegated to stream buffers.

而了解stream buf最推荐的也是本书里的The Stream Buffer Classes一节。

`streambuf`实际上是一个模板类

```
typedef basic_streambuf<char> streambuf;
typedef basic_streambuf<wchar_t> wstreambuf;
```

两个类都是虚基类，因此没法直接创建。可以派生自己的子类，以便提供其他设备/数据输入的接口。

STL标准库实现了两个子类，分别是`filebuf`和`stringbuf`。

对用户来讲，`streambuf`有两种用法，一是直接使用各个接口，二是继承并实现新的I/O channels，例如封装

C里的FILE*为C++流读写的方式，封装日志库为C++输出流的方式。

2. 何为正交

每个流(`cout/cin/clog/ifstream/ofstream`)都有自己的流缓冲区(`streambuf`)。通过`rdbuf`接口可以获取当前的streambuf，也可以设置新的streambuf。

例如我们可以修改`std::cout`和`std::stringstream`使用同一块缓冲区

```
#include <stdio.h>
#include <iostream>
#include <sstream>

int main() {
    std::stringstream ss;
    std::streambuf* cout_buf = std::cout.rdbuf();
    std::cout.rdbuf(ss.rdbuf());
    //使用了新的缓冲区后，字符串不会输出到屏幕，而是由stri
    std::cout << "std::cout hello world";

    //printf std::cout hello world
    printf("printf %s\n", ss.str().c_str());

    std::cout.rdbuf(cout_buf);
```

```
    return 0;
}
```

当然，也可以自定义缓冲区。

3. 自定义缓冲区

`streambuf`的设计思想里，将外部设备看做一个可以顺序读写的字符序列，`streambuf`则起到一个transport的作用，同时通过良好的接口设计，支持用户自定义的行为，例如尽量减少对外部设备的读写。

当我们需要封装自定义设备的读写时，可以通过自定义缓冲区来实现。通常都从`streambuf/wstreambuf`继承而来，并实现对应的虚函数。

对于输出、输入都有三个重要的指针来管理缓冲区，同时提供了一系列的虚接口。接下来我们逐步介绍下如何实现自定义的`streambuf`。

3.1 自定义输出流缓冲区

不得不说`streambuf`的接口命名上太简化了，从这三个指针函数的命名上可见一般。

1. `pbase()`: put base，输出缓冲流的首指针。
2. `pptr()`: put pointer，当前可写位置
3. `epptr()`: end put pointer，输出缓冲流的尾指针+1，即one past the last character。

通过这三个指针，我们就可以定位当前的输出缓冲区的使用情况，当然，前提是我们统一假定输出设备可以接受一个连续的字符序列。对这三个指针位置参考资料里有更直观的图解。

当`pptr != epptr`时，数据默认行为是顺序更新到缓冲区，直到`pptr == epptr`，表示当前缓冲区已满(overflow)，此时会调用`overflow`方法。这是一个虚函数接口，在这个函数里我们更新到外部设备并且重置缓冲区可用即可。

`setp`可以指定`pbase pptr epptr`这三个指针的位置，函数原型为：

```
void setp (char* new_pbase, char* new_epptr);
```

经过`setp`设置后,`pbase pptr = new_pbase`，`epptr = new_epptr`。

先上一个例子，该例子将所有输入都大写后输出

```
//从std::streambuf继承，将输入字符全部大小输出
class UpperCaseStreamBuf : public std::streambuf {
protected:
    virtual int_type overflow(int_type c) {
        if (c != EOF) {
            //转大写
            c = std::toupper(c);
            //输出
            if (putchar(c) == EOF) {
                return EOF;
            }
        }

        return c;
    }
};//UpperCaseStreamBuf

int main() {
    UpperCaseStreamBuf upper_case_stream_buf;
    //使用我们新定义的streambuf
    std::ostream out(&upper_case_stream_buf);
    //31 HEXADECIMAL:1F
    out << "31 hexadecimal:" << std::hex << 31 << s

    return 0;
}
```

`UpperCaseStreamBuf` 接受字符并转大写，调用 `putchar` 输出。 可以看到对自定义的输出缓冲区，我们并没有定义缓冲用的内存，也就没有调用过 `setp` 。这对输出缓冲区是可行的。

这个例子实现效率并不高，因为没有实现 `xsputn` 。

`sputn` 用于一次性写入多个字符，该函数实际上是调用了虚函数 `xsputn` 。 `streambuf` 默认的实现是对每个字符逐个调用 `sputc` 。因此如果追求更高的写入效率，可以重新实现 `xsputn` 。

举个《The C++ Standard Library 》的例子：

```
#include <stdio.h>
#include <unistd.h>

#include <iostream>
#include <streambuf>

//without "real" buffer
class FdOutBuf : public std::streambuf {
protected:
    int _fd;
public:
    FdOutBuf(int fd) : _fd(fd) {
```

```
    }

protected:
    virtual int_type overflow(int_type c) {
        std::cout << "FdOutBuf::overflow" << std::e
        if (c != EOF) {
            char z = c;
            if (write(_fd, &z, 1) != 1) {
                return EOF;
            }
        }

        return c;
    }

    virtual std::streamsize xsputn(const char* s, s
        std::cout << "FdOutBuf::xsputn" << std::end
        return write(_fd, s, num);
    }
};//FdOutBuf

class FdOstream : public std::ostream {
protected:
    FdOutBuf _buf;
public:
    FdOstream(int fd) : _buf(fd), std::ostream(&_bu
    }
};//FdOstream


int main() {
    FdOstream out(1);
    //多个字符直接一次性调用xsputn输出
```

```
    out << "31 hexadecimal: " << std::hex << 31 <<

    return 0;
}
```

注意这个例子一个潜在的问题，就是`FdOstream`基类`std::ostream`以及成员变量`_buf`的初始化顺序，优先调用了基类的构造函数，然后才是`_buf`，因此`std::ostream(&_buf)`实际上传入的是一个未初始化的_buf。更好的写法可以参考这里，是南加大实现的一个同时用于输入输出的例子。

上面的例子介绍了`streambuf`的基本用法，但都没有真正的buffer，接下来再看一个`overflow`使用的例子，定义了自己的接收buffer，并在buffer满后调用`overflow`输出。

```
#include <stdio.h>
#include <unistd.h>
#include <iostream>
#include <locale>
#include <streambuf>

class FdOutRealBuf : public std::streambuf {
protected:
    static const int s_buffer_size = 10;
```

```
    //接收用的缓存，大小为10
    char _buffer[s_buffer_size];

public:
    FdOutRealBuf() {
        //设置pbase epptr分别指向_buffer的头尾
        setp(_buffer + 0, _buffer + (s_buffer_size
    }
    virtual ~FdOutRealBuf() {
        sync();
    }

protected:
    int flush_buffer() {
        int num = pptr() - pbase();
        //写到标准输出
        if (write(1, _buffer, num) != num) {
            return EOF;
        }
        //移动回buffer首部
        pbump(-num);
        return num;
    }

    //buffer满时（即pptr == epptr)调用
    virtual int_type overflow(int_type c) {
        if (c != EOF) {
            //c记录到buffer
            *pptr() = c;
            //往前移动一个元素
            pbump(1);
        }
```

```
        if (flush_buffer() == EOF) {
            return EOF;
        }

        return c;
    }

    virtual int sync() {
        if (flush_buffer() == EOF) {
            return -1;
        }

        return 0;
    }
};//FdOutRealBuf

int main() {
    std::streambuf* last_buf = std::cout.rdbuf();
    FdOutRealBuf fd_out_read_buf;
    //std::cout使用新定义的fd_out_read_buf输出
    std::cout.rdbuf(&fd_out_read_buf);

    std::cout << "hello world" << "123456789abc" <<
    std::cout << 123 << std::endl;

    std::cout.rdbuf(last_buf);

    return 0;
}
```

相关的一些实例还可参考这个输出增加时间戳的例子

3.2. 自定义输入流缓冲区

与输出流对应的，对于输入流，同样存在三个指针位置：

1. eback(): end back, 是指缓冲区的首部
2. gptr(): get pointer, 缓冲区当前读位置
3. egptr(): end get pointer, 缓冲区尾部

前面介绍输出流缓冲区时，有的例子没有引入真正的buffer缓冲。但是如果要实现一个实际意义的输入流缓冲区，这点几乎是不可能的。

原因在于输入流缓冲区有一个`putback`

```
#include <stdio.h>
#include <string>
#include <iostream>

int main() {
    char c;
    int n;
    std::string str;

    std::cin >> c;
    if ((c >= '0') && (c <= '9')) {
        std::cin.putback(c);
        std::cin >> n;
        printf("n:%d\n", n);
```

```
    } else {
        std::cin.putback(c);
        std::cin >> str;
        printf("str:%s\n", str.c_str());
    }

    return 0;
}
```

该例子读取一个字符，判断是否在[‘0’, ‘9’]内，putback，再根据判断的结果分别读取为整数或者字符串。

当缓冲区被填充满时，会调用`underflow`接口。

可以参考这里的**Example 1: FILE buffers to integrate with C code**例子。

当然也可以同时实现输入输出流，例如南加州大学的例子。

4. glog日志的应用

glog记录的输入形式采用了C++流的方式，代码里也采用了继承`std::streambuf std::ostream`的实现方式。

摘抄下相关代码：

```
class LogStreamBuf : public std::streambuf {
 public:
  // REQUIREMENTS: "len" must be >= 2 to account f
  LogStreamBuf(char *buf, int len) {
    se*tp(buf, buf + len - 2);
  }
  // This effectively ignores overflow.
  virtual int_type overflow(int_type ch) {
    return ch;
  }
  ...

LogStream(char *buf, int len, int ctr)
        : std::ostream(NULL),
          streambuf_(buf, len),
          ctr_(ctr),
          self_(this) {
      rdbuf(&streambuf_);
}
```

`LogStream`更详细的信息会在接下来的glog源码解析里介绍。

5. 参考资料

1. The C++ Standard Library

2. A beginner’s guide to writing a custom stream buffer

3. Standard C++ IOStreams and Locales

github博客绑定域名 ❮

❯ 三座城市的印象：苏州、杭州和南京

HelloCodeYing

Hello,Code,Life,Reading,Ying.

查看熊掌号