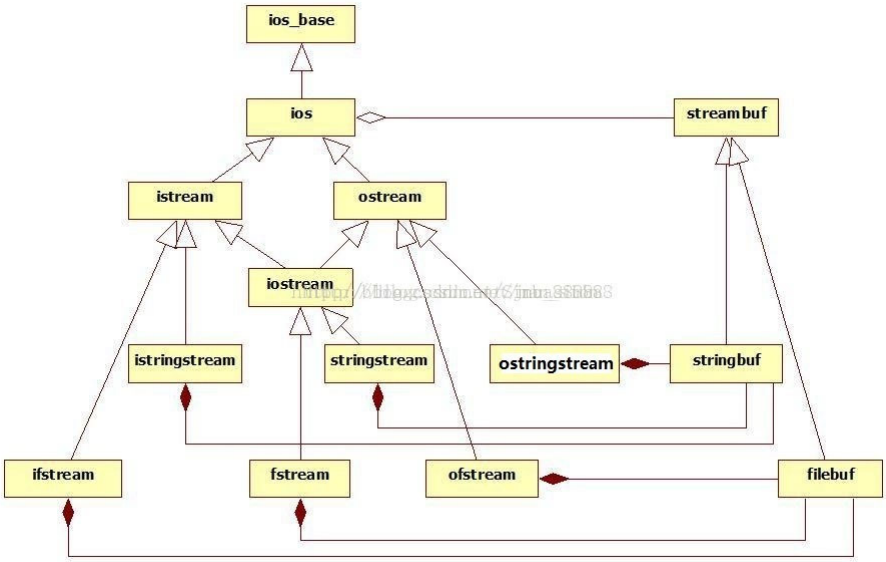
c++ 文件读写总结（streambuf） - marblemm - 博客园

marblemm 粉丝 - 20 关注 - 25

在C++ 中引入了流的概念，我们很方便的通过流来读写文本数据和二进制数据，那么流对象的数据究竟是怎么存储的呢，为了搞清这个问题，先来看一看c++ 的 io 体系：



由图可以看出，在stream 的实现中，除了虚基类IOS_BASE之外，所有的类内部都有一个streambuf, streambuf 是一个虚基类（不能被实例化，因此所内部包含streambuf(这个虚基类而非其子类)的类也是虚基类），代表流对象内部的缓冲区，就是我们流操作中输入输出的内容在内存中的缓冲区。

Streambuf有两个子类，分别是stringbuf 和 filebuf,这两个子类可以被实例化，我们常用的文件流和字符串流，内部的缓冲区就是这两个类。

我们平常使用到的流基本是标准输入输出流，文件流和字符串流。在每个流初始化的时候都会初始化相应的streambuf(其实是它的子类)用来缓冲数据。

当我们用文件或者字符串初始化流的时候，流内部会保存该文件和字符串的信息，而在内部实例化一个streambuf用来缓冲数据，些数据时，当缓冲区满的时候再将数据写到文件或者字符串，读数据时当缓冲区没有数据时从文件或字符串读数据到缓冲区。

在文件流这种情况下，streambuf 是为了避免大量的IO 操作

在字符串流的情况下，streambuf（其实是套在上面的流对象）是为了提供字符串的格式化读取和输出操作（想象字符串是你从键盘输入的数据）

所以streambuf 可以看作一块缓冲区，用来存储数据，在这种情况下，我们常常在程序中用的 char数组缓冲区是不是可以被替代呢？答案是of course

而且，有了streambuf ,缓冲区的管理和写入写出都非常方便，最好的是流对象有复制拷贝等构造函数可以方便参数传递等需要拷贝的情景。

但是streambuf 本身是个虚基类，不能实例化，所以要用streambuf 就需要自己继承streambuf 写一个新的类出来才能用，这个实现方法最后介绍，好在c++ 标准类库实现了两个子类stringbuf 和 filebuf ,所以我们可以选stringbuf 来作为我们的数据缓冲对象（不选filebuf 是因为它的实现和文件紧耦合的，只适合文件流）

流对象有一个构造函数是通过streambuf 来构造:

```
stringbuf sb;
istream is(&sb);
```

有了流对象我们就可以在流上进行各种输入输出操作，输入会从缓冲区读数据，输出会将数据写到缓冲区

注意对缓冲区的读写一定要注意方法，流符号是格式话输入输出，get,put,read,write等是二进制读写。

格式化输入的内容应当格式化读取，二进制写入应当二进制读取否则会出现写入和读出数据不一致的问题

格式化写入一个int 数据时，会将该数据每位分离出来，按照字符编码写到缓冲区，例如 int x= 123, 格式化写入以后缓冲区存以后，缓冲区有三个字节分别存放1，2，3的字符编码。格式化读出是相反的过程，将读到的字符转成相应的类型的数据

二进制写入时进行直接的内存拷贝不做任何动作，例如 int x = 123 二进制写入后（二进制写时需要取地址，转成char* 并指出要写入的字节数，如f.write((char*)&x,sizeof(int))

写完后缓冲区的数据是0x0000007b,是计算机内存中对123 的内存的完全拷贝

streambuf是C++流(iostream)与**流实体**(或者叫**原始流**，文件、标准输入输出等)交互的桥梁

```
# 文件流
fstream <--> filebuf <--> file
# 字符串流
stringstream <--> stringbuf <--> string
```

streambuf内部实现

术语说明：

get 相当于 **从流中读取数据**

put 相当于 **写入数据到流中**

字符，C/C++中的**char**，也可以理解为**字节**

streambuf内部持有三个用于get的指针`gfirst,gnext,glast`和三个用于put的指针`pfirst,pnext,plast`，这些指针分别可以使用`eback(),gptr(),egptr()`和`pbase(),pptr(),epptr()`函数获得，在代码中需要使用这些函数获取指针，为了方便描述，我直接使用这些指针变量名

下面是其他几个受保护的成员函数的作用

gbump(n) : gnext+=n

setg : setg(gfirst, gnext, glast)

pbump(n) : pnext+=n

setp : setp(pfirst, pnext, plast)

小结：

get缓冲区通过setg()设置，setg的三个参数分别对应gfirst,gnext,glast

put缓冲区通过setp()设置，setp的两个参数分别对应pfirst,plast

如果继承自streambuf的子类不通过setg和setp设置缓冲区，也就是**读写缓冲区为空**，那么这个流可以说是**不带读缓冲和写缓冲的流**，这时gfirst = gnext = glast = pfirst = pnext = plast = NULL

子类需要override(覆写)几个虚函数来封装具体的流的实现

虚函数(protected)

这些函数有些需要子类实现，来屏蔽不同的流的具体实现，向上提供统一的接口

缓冲区管理

setbuf ---------- 设置缓冲区

seekoff --------- 根据相对位置移动内部指针

seekpos --------- 根据绝对位置移动内部指针

sync ------------ 同步缓冲区数据(flush)，默认什么都不做

showmanyc ------- 流中可获取的字符数，默认返回0

输入函数(get)

underflow(c) ---- 当get缓冲区不可用时调用，用于获取流中当前的字符，注意获取和读入的区别，获取并不使gnext指针前移，默认返回EOF

uflow() --------- 默认返回underflow()，并使gnext++

xsgetn(s, n) ---- 从流中读取n个字符到缓冲区s中并返回读到的字符数:默认从当前缓冲区中读取n个字符，若当前缓冲区不可用，则调用一次uflow()

pbackfail ------- 回写失败时调用

输出函数(put)

overflow(c) ----- 当put缓冲区不可用时调用，向流中写入一个字符；当c==EOF时，流写入结束；与输入函数的**uflow()**相对

xsputn(s, n) ---- 将缓冲区s的n个字符写入到流中并返回写入的字符数；与输入函数的**xsputn**相对

缓冲区不可用是指gnext(pnext) == NULL或者gnext(pnext) >= glast(plast)

public函数

缓冲区管理

pubsetbuf : setbuf()

pubseekoff : seekoff()

pubseekpos : seekpos()

pubsync : sync()

输入函数(get)

in_avail : (用于get的)缓冲区内还有多少个字符可获取，缓冲区可用时返回`glast-gnext`，否则返回`showmanyc()`

snextc : `return sbumpc() == EOF ? EOF : sgetc()`

sbumpc : 缓冲区不可用时返回uflow()；否则返回(++gnext)[-1]

sgetc : 缓冲区不可用时返回underflow()；否则返回*gnext

sgetn : xsgetn()

sputbackc : 缓冲区不可用时返回pbackfail(c)；否则返回*(--gnext)

sungetc : 类似于sputbackc，不过默认调用pbackfail(EOF)

输出函数(put)

sputc : (用于put操作的)缓冲区不可用时，返回overflow(c)；否则*pnext++ = c，返回pnext

sputn : xsputn()

iostream与streambuf的调用关系

下面就iostream常用的几个函数说明他们的调用关系

read(char *s, int n) -> buf.sgetn(s, n)

getline() -> buf.sgetc(), buf.snextc(); 首先调用一次sgetc()来判断当前字符是否为EOF，然后不断地调用snextc()读取下一个字符，直到读到\n

peek() -> buf.sgetc()

sync() -> buf.pubsync()

总结

在istream对象中，除了read这种一次读入多个字符的函数外，一般的读取流的函数(`operator>>()`)、get、getline都是调用snextc()一次读入一个字符

istream的readsome(buf, size)函数本质还是调用了read，大致相当于`read(buf, min(in_avail(), size))`

snextc函数，当缓冲区不可用时会触发uflow()，uflow()会调用underflow()触发一次读取原始流的操作，如果读到了流的末尾，可以返回EOF；缓冲区可用时直接从缓冲区中读取一个字符`return *gnext++`

underflow函数的作用是：当读取缓冲区不足时，从原始流中读取一段数据并调用setg重新设置gfirst gnext glast三个指针，将读到的数据缓存起来，并返回当下的字符`return *gnext`；原始流中没有数据时(或者说读到了流的末尾时)返回EOF

只要原始流还可访问(读取或写入)，**xsgetn与xsputn就需要尽可能的从原始流中读取(写入)n个字符**。因为有些流比如tcp socket一次可能接收不完所需要的字符数，这就需要循环接收直到收到n个字符为止。

`[gfirst, glast)`永远是已经从流实体里读到的数据如果他们不为空的话

有两种情况会使一个istream对象的bool转型为false:读到EOF(文件结束标志)或遇到一个无效的值(输入流进入fail状态)。istream对象的bool转型为false的情况下，此后的所有读入动作都是无操作。直到调用istream对象的成员函数clear()来清除该对象的内部状态。

缺省情况下，输入操作符丢弃空白符、空格符、制表符、换行符以及回车。如果希望读入上述字符，或读入原始的输入数据，一种方法是使用istream的get()成员函数来读取一个字符，另一种方法是使用istream的getline()成员函数来读取多个字符。istream的read(char* addr, streamsize size)函数从输入流中提取size个连续的字节，并将其放在地址从addr开始的内存中。istream成员函数gcount()返回由最后的get()、getline()、read()调用实际提取的字符数。read()一般多用在读取二进制文件，读取块数据。

输入流有三个函数来测试流状态：即bad(),fail()和eof()。ignore()用来抛掉指定个数的缓冲区中的字节。如果bad()为真，代表是遇到了系统级的故障。如果fail()为真，则表示输入了非法的字符。

下面是缓冲区使用的情景：

考虑一个生产者，消费者的问题，线程A 生成的数据，线程B读取，可以解决的方案如下：

1. 设立全局变量，缓冲数据，A,B都可以访问（在这种情况下，A 生产的时候要考虑缓冲区是否够用，B读取的时候要判断当前是否有有效数据可读，而且很难设计一个合理分配内存的缓冲区（想象A 生产的数据有时很大，有时很小））

2.网络通信（TCP,UDP）

3. streambuf 登场，有了streambuf配合stream， A就像正常操作流一样往流对象里塞数据，而B 就像正常操作流一样从流里面读数据，不用关心其他问题，只要这两个流的sterambuf 是同一个对象。

上一段代码：

```
#include <iostream>
#include <streambuf>
#include <sstream>
#include <fstream>
#include <string>
#include <cstring>
#include <memory>
#include <thread>
using namespace std;
stringbuf buf;
istream in(&buf);
ostream out(&buf);
bool flag = false;
void threadb()
{
  char data;
  while (true)
  {
    if (flag)
    {
      in >> data;
      cout << "thread B recv:" << data << endl;
      flag = false;
    }
  }
}
int main()
{
  thread consumer(threadb);
  char data;
  while (true)
  {
    cin >> data;
    out << data;
    flag = true;
  }
  return 0;
}
```

在特殊的情景下可以实现自己的streambuf类，自己实现的类必须继承streambuf 类，自定义的streambuf 必须实现overflow,underflow,uflow 等方法，其中overflow在输出缓冲区不够用时调用，underflow和uflow在输入缓冲区无数据时调用,区别是uflow 会让读取位置前进一位，而underflow不会。sreambuf 内部维护着六个指针 eback,gptr,egptr,pbase,pptr,epptr.分别指向读取缓冲区的头，当前读取位置，尾，写缓冲区的头，当前写位置，尾（实际上这几个指针指向同一段缓冲区）

自定义实现方式要注意要在该返回EOF的时候，返回EOF,underflow和uflow都有可能返回EOF,一旦返回了EOF则标志着流结束，之后对流的操作无效。

如下代码实现了一个自定义的streambuf:

```
#include <iostream>
#include <streambuf>
#include <sstream>
#include <fstream>
#include <string>
#include <cstring>
#include <memory>
using namespace std;
class mybuf : public streambuf
```

```
{
public:
  enum{ SIZE = 10};
  mybuf()
  {
    memset(buffer, 'j', 10);
    //buffer[3] = ' ';
    setbuf(buffer, SIZE);
  }
  void log()
  {
    cout <<hex<<gptr() << endl;
  }
protected:
  int_type overflow( int_type c)
  {
    cout << "overflow" << endl;
    return c;
  }
  streambuf* setbuf(char* s, streamsize n)
  {
    setp(s, s + n);
    setg(s, s, s + n);
    return this;
  }
  int_type underflow() override
  {
    cout << "here"<<endl;
    memset(buffer, 'w', 10);
    setg(buffer, buffer, buffer+10);
    return ' ';
  }
  int_type uflow() override
  {
    cout << "uflow" << endl;
    memset(buffer, 'x', 10);
    setg(buffer, buffer, buffer + 10);
    return EOF;
  }
private:
  char buffer[SIZE];
};
int main()
{
  mybuf buf;
  char test[2000];
  memset(test, 'a', 2000);
  //buf.pubsetbuf(test, 1000);
  string hh;
  string xx;
  istream in(&buf);
  ostream tt(&buf);
  in>>hh;
  cout << hh << endl;
  //tt.write(test, 9);
  in >> xx;
  in.read(test, 11);
  cout<< xx << endl;
  cout << "end" << endl;
  return 0;
}
```


rdbuf函数有两种调用方法

basic_streambuf<Elem, Traits> *rdbuf( ) const;

basic_streambuf<Elem, Traits> *rdbuf(basic_streambuf<E, T> *_Sb);

1）无参数。返回调用者的流缓冲指针。

2）参数为流缓冲指针。它使调用者与参数(流缓冲指针)关联，返回自己当前关联的流缓冲区指针。

假如我们用C语言写一个文件复制程序，比如一个mp3文件，我们首先考虑的是C语言的文件输入输出功能，其思路是建一个指定大小缓冲区，我们从源文件中循环读取缓冲区大小的数据，然后写进目的文件。而在C++中，我们抛弃了这种用字符缓冲区的按字节复制的方法，因为这种方法看起来很繁琐，而且效率一点也不高。

下面可以对比这两种方法（程序可以直接执行）：

C:


```
int main()
{
    char buf[256];
    FILE *pf1, *pf2;
```

```
    if((pf1 = fopen("1.mp3", "rb")) == NULL)
    {
        printf("源文件打开失败\n");
        return 0;
    }
    if((pf2 = fopen("2.mp3","wb")) == NULL)
    {
        printf("目标文件打开失败\n");
        return 0;
    }
    while(fread(buf,1,256,pf1), !feof(pf1))
    {
         fwrite(buf,1,256,pf2);
    }
    fclose(pf1);
    fclose(pf2);
    return 0;
}
```

在C++中：

```
using namespace std;
int main()
{
    fstream fin("1.mp3",ios::in|ios::binary);
    if(!fin.is_open())
    {
        cout << "源文件打开失败" << endl;
        return 0;
    }
    fstream fout("2.mp3",ios::out|ios::binary);
    if(! fin.is_open())
    {
        cout << "目标文件打开失败!" << endl;
        return 0;
    }
    fout<<fin.rdbuf();
    fin.close();
    fout.close();
    return 0;
}
```

看起来是不是清晰多了呢，这就是C++中的流缓冲的威力了，程序通过把源文件的流重定向到关联到目的文件的流对象，通过 fout<<fin.rdbuf();一句代码就完成了在C语言中的循环读写缓冲区的功能，而且C++中使用的是底层的流缓冲，效率更高