

记录一下在看WebServer源码, muduo源码以及编写自己的网络库时的一些心得

一、在muduo和WebServer源码中，Channel类和其他类的关系

Channel类其实就是最直接的挂在epoll队列上，被监听的文件描述符和事件的封装。并且也是epoll队列中，如果事件发生，最先执行的就是Channel类中的回调函数。

而更上层的封装，在muduo中， TcpConnection类、Acceptor类、EventLoop类、TimerQueue类中都有Channel类的成员变量。因为这些类中都有文件描述符需要被监控，比如TcpConnection类中的connfd, Acceptor类中的socketfd, EventLoop类中的eventfd(用来唤醒沉睡的IO线程), TimerQueue类中的Timerfd (如果时间到，就会触发这个文件描述符)。

所以在muduo中基于对象编程最明显的体现就是Channel类，

 | 基于对象编程就是将类对象作为成员变量

 | 面向对象编程就是继承类

至于WebServer中， Server类、HttpData类、EventLoop类中都有Channel类的成员变量。这些类中各自的文件描述符如下：

类	与Channel 关联的文件 描述符	备注
Server	Socketfd	在muduo中， Server类是没有Channel类成员的，因为WebServer中省略了Acceptor类，所以就直接放到Server中了
HttpData	connfd	HttpData类似muduo中的TcpConnection类
EventLoop	eventfd	这个和muduo中一样

二、定时器的作用

在网络库中，定时器究竟有什么用，为什么网络库中一定要有定时器？

在WebServer中，定时器作用有一个

①在服务端接收到客户端请求，并建立起http连接以后，如果客户端在建立连接以后一段时间内没有发送具体请求。就会断开

出现这种情况的原因是：在客户端和服务端建立http连接的过程中，客户端先调用connect函数与服务端建立http连接，然后再用write函数向connfd中写入请求并发送过去。而如果在connect和write之间如果睡三秒，或者网络拥塞的情况下，就可能触发定时器的超时函数，也就是在服务端把这个文件描述符给删除了

②在长连接中，如果长时间没有传递消息，也会断开

三、socket函数中IP地址设置

在设置监听IP的过程中，是在sockaddr_in结构体中设置的，一般为如下形式：

```
1 | addr.sin_addr.s_addr = htonl(INADDR_ANY);
```

这句程序表示监听本机所有的IP，所以本地IP127.0.0.1和inet地址都可以被监听到。就相当于应用层输入0.0.0.0一样效果。

四、vector的赋值运算符

不确定会不会完全拷贝，验证之后确实是的

```
1 #include <iostream>
2 #include <vector>
3 using namespace std;
4 void show(vector<int> &t)
5 {
6     for (int i=0; i<t.size(); i++)
7         cout<<t[i]<<" ";
8     cout<<endl;
9 }
10 int main()
11 {
12     int a[]={1,2,3,4};
13     vector<int> b(a, a+2);
14     show(b);
15     vector<int> c(a, a+3);
16     show(c);
17     c=b;
18     show(c);
19     return 0;
20 }
```

结果图：

```
gmq@gmq-OptiPlex-3020:~/桌面/FILE$ ./a.out
1 2
1 2 3
1 2
```

五、运算符的优先级

尤其注意里面第三级别，第五～第十四级别，第十六级别中的运算符。

尤其注意辅助运算符=的优先级特别低，所以在如下的情况下一定要加括号

```
1 int a=10;
2 int b;
3 if (b = a+11>0)//这里本意是想把a+11的值赋给b，然后和0比较，但是如果没加括号，  
4 //b的值只可能是0或者1，因为会先计算a+11，然后和0比较，得到的布尔  
5 值再赋给b
6 ...
```

一定要注意这种小错误，在这个地方卡了半天。

(C++符号优先级表)

运算符	描述	例子	可重载性
第一级别			
::	作用域解析符	Class::age = 2;	不可重载
第二级别			
()	函数调用	isdigit('1')	可重载
()	成员初始化	c_tor(int x, int y) : _x(x), _y(y*10) {};	可重载
[]	数组数据获取	array[4] = 2;	可重载
->	指针型成员调用	ptr->age = 34;	可重载
.	对象型成员调用	obj.age = 34;	不可重载
第三级别			
++	后自增运算符	for(int i = 0; i < 10; i++) cout << i;	可重载
--	后自减运算符	for(int i = 10; i > 0; i--) cout << i;	可重载
const_cast	特殊属性转换	const_cast<type_to>(type_from);	不可重载
dynamic_cast	特殊属性转换	dynamic_cast<type_to>(.....);	不可重载

	转换	(type_irrom);	里载 可重
运算符 static_cast	描述属性 转换	例子 static_cast<type_to>(type_from);	操作 重载
reinterpret_cast	特殊属性 转换	reinterpret_cast<type_to> (type_from);	不可 重载
typeid	对象类型 符	cout << typeid(var).name(); cout << typeid(type).name();	不可 重载
第三级别(具有右结合性)			
!	逻辑取反	if(!done) ...	可重 载
not	!的另一种 表达		
~	按位取反	flags = ~flags;	可重 载
compl	~的另一种 表达		
++	预自增运 算符	for(i = 0; i < 10; ++i) cout << i;	可重 载
--	预自减运 算符	for(i = 10; i > 0; --i) cout << i;	可重 载
-	负号	int i = -1;	可重 载
+	正号	int i = +1;	可重 载
第四级别(具有左结合性)			
*	指针取值	int data = *intPtr;	可重 载
&	值取指针	int *intPtr = &data;	可重 载
new	动态元素 内存分配	long *pVar = new long; MyClass *ptr = new MyClass(args);	可重 载
new []	动态数组	long *array = new long[n];	可重 载

内存分配			
运算符	描述	例子	重载
delete	动态析构 元素内存	delete pVar;	可重载
delete []	动态析构 数组内存	delete [] array;	可重载
(type)	强制类型 转换	int i = (int) floatNum;	可重载
sizeof	返回类型 内存	int size = sizeof floatNum; int size = sizeof(float);	不可重载
第四级别			
->*	类指针成 员引用	ptr->*var = 24;	可重载
.*	类对象成 员引用	obj.*var = 24;	不可重载
第五级别			
*	乘法	int i = 2 * 4;	可重载
/	除法	float f = 10.0 / 3.0;	可重载
%	取余数(模 运算)	int rem = 4 % 3;	可重载
第六级别			
+	加法	int i = 2 + 3;	可重载
-	减法	int i = 5 - 1;	可重载
第七级别			
<<	位左移	int flags = 33 << 1;	可重载
>>	位右移	int flags = 33 >> 1;	可重载

第八级别 运算符	描述	例子	可重载
<	小于	if(i < 42) ...	可重载
<=	小于等于	if(i <= 42) ...	可重载
>	大于	if(i > 42) ...	可重载
>=	大于等于	if(i >= 42) ...	可重载
第九级别			
==	恒等于	if(i == 42) ...	可重载
eq	== 的另一种表达		
!=	不等于	if(i != 42) ...	可重载
not_eq	!=的另一种表达		
第十级别			
&	位且运算	flags = flags & 42;	可重载
bitand	&的另一种表达		
第十一级别			
^	位异或运算	flags = flags ^ 42;	可重载
xor	^的另一种表达		
第十二级别			
	位或运算	flags = flags 42;	可重载

运算符	的另一种 描述	例子	可重 载性
第十三级别			
&&	逻辑且运 算	if(conditionA && conditionB) ...	可重 载
and	&&的另一 种表达		
第十四级别			
	逻辑或运 算	if(conditionA conditionB) ...	可重 载
or	的另一 种表达		
第十五级别(具有 右结合性)			
? :	条件运算 符	int i = (a > b) ? a : b;	不可 重载
第十六级别(具有 右结合性)			
=	赋值	int a = b;	可重 载
+=	加赋值运 算	a += 3;	可重 载
-=	减赋值运 算	b -= 4;	可重 载
<hr/>			
*=	乘赋值运 算	a *= 5;	可重 载
/=	除赋值运 算	a /= 2;	可重 载
%=	模赋值运 算	a %= 3;	可重 载
<hr/>			
&=	位且赋值 二元	flags &= new_flags;	可重 载

运算符	描述	例子	可重载性
and_eq	&=的另一种表达		
$\wedge=$	位异或赋值运算	flags $\wedge=$ new_flags;	可重载
xor_eq	$\wedge=$ 的另一种表达		
$ =$	位或赋值运算	flags $ =$ new_flags;	可重载
or_eq	$ =$ 的另一种表达		
$<<=$	位左移赋值运算	flags $<<=$ 2;	可重载
$>>=$	位右移赋值运算	flags $>>=$ 2;	可重载
第十七级别			
throw	异常抛出	throw EClass("Message");	不可重载
第十八级别			
,	逗号分隔符	for(i = 0, j = 0; i < 10; i++, j++) ...	可重载

六、非阻塞客户端编写

在看muduo库的时候，我们一般都是看重的muduo服务端的非阻塞是如何去编写的，但是在muduo中，客户端类TcpClient也是中客户端连接符同样是非阻塞的，但是当我自己编写非阻塞描述符时，遇到了一些问题，这里把它记录下来。

```

1 #include <iostream>
2 #include <sys/types.h>
3 #include <sys/socket.h>
4 #include <unistd.h>
5 #include <ctype.h>
6 #include <arpa/inet.h>
7 #include <stdlib.h>
8 #include <string.h>
9 #include <stdio.h>
10 #include <fcntl.h>
11 #include <errno.h>
12 #include <sys/select.h>
13 using namespace std;
14

```

```
15 int main(int argc, const char* argv[])
16 {
17     if (argc<2)//需要传入端口号
18     {
19         cout<<"input ./a.out port"=>endl;
20         exit(1);
21     }
22     int sfd;//socket文件描述符
23     int port=atoi(argv[1]);//传入的是char*类型的端口号, 转换成int型
24     /*定义sockaddr_in结构体*/
25     struct sockaddr_in addr;
26     addr.sin_family=AF_INET;
27     addr.sin_port=htons(port);
28     inet_pton(AF_INET, "127.0.0.1", &addr.sin_addr.s_addr);//将ip
    转换成int型存入sockaddr_in结构体中
29
30
31     socklen_t addrlen;
32     sfd=socket(AF_INET, SOCK_STREAM, 0);
33     /*设置非阻塞*/
34     int flag = fcntl(sfd, F_GETFL);
35     flag |= O_NONBLOCK;
36     fcntl(sfd, F_SETFL, flag);
37     /*连接客户端*/
38     cout<<"-----before connect-----\n";
39     int ret = connect(sfd, (struct sockaddr*) &addr,
40     sizeof(addr));/*问题一*/
41     if (errno==EINPROGRESS)
42     {
43         cout<<"connecting...\n";
44         fd_set write_fd;
45         FD_ZERO(&write_fd);
46         FD_SET(sfd, &write_fd); //设置关注sfd的写事件
47         struct timeval timeout;
48         timeout.tv_sec = 6;
49         timeout.tv_usec = 0;
50         ret = select(sfd + 1, NULL, &write_fd, NULL, &timeout);
51         //如果6秒后还没有可写事件产生, 就返回, 这时ret=0
52         //如果连接好, 就会触发写事件, 也会返回, 这时ret>0
53         if(ret < 0)//select函数执行出错
54         {
55             perror("select");
56             close(sfd);
57             exit(-1);
58         }
59         else if (ret == 0)//表示select阻塞超时
60         {
61             printf( "connection timeout\n" );
62             close(sfd);
63             exit(-1);
64         }
65         if (!FD_ISSET(sfd, &write_fd))
66             //虽然触发读事件, 但是sfd仍然不可写, 是其他文件描述符触发的,
```

```
67 //个人认为这种情况几乎不可能发生，这里只是为了保证所有情况都考虑到
68 {
69     cout<<"err, no events found!\n";
70     close(sfd);
71 }
72 else
73     //即便是sfd触发读事件，仍然有可能出错，因为即便连接失败，会把
74     //所以还要取sfd的错误码来查看一下是否有错误，如果没有错误，才算
75     //真正连接成功。
76     {
77         int err = 0;
78         socklen_t elen = sizeof(err);
79         ret = getsockopt(sfd, SOL_SOCKET, SO_ERROR, (char
80 * )&err, &elen);
81         //利用getsockopt函数查看套接字sfd错误码
82         if(ret < 0)
83         {
84             perror("getsockopt");
85             close(sfd);
86             exit(-1);
87         }
88         if(err != 0)
89         {
90             printf("connect failed with the error:
91 (%d)%s\n", err, strerror(err));
92             close(sfd);
93             exit(-1);
94         }
95     }
96 }

97
98
99 }
100 }
101 else if (ret<0)//如果连接出现其他错误，直接返回
102 {
103     perror("connect error is");
104     return 0;
105 }
106 cout<<"-----connect finish-----\n";
107 /*发送数据和接受数据*/
108 int len;
109 while (1)
110 {
111     char buf[1024];
112     char quit[]="quit\n";
113     bzero(buf, 1024);
114     fgets(buf, sizeof(buf), stdin); //从终端读取字符串
115     write(sfd, buf, strlen(buf));
116     bzero(buf, 1024);
```

```

117     while(1)
118     {
119         len = recv(sfd, buf, sizeof(buf), 0); /*问题二*/
120         if (len>=0)
121         {
122             cout<<buf;
123             break;
124         }
125         else
126             if(errno==EAGAIN||errno==EWOULDBLOCK||errno==EINTR)
127                 //如果出现因为非阻塞而没有读到数据的情况，就睡眠一会儿重新读取
128                 usleep(100);
129             else
130                 perror("read");
131                 cout<<"error"<<endl;
132                 exit(1);
133             }
134         }
135         if (len == 0)
136             break;
137         if (!strcmp(buf, quit))
138             break;
139     }
140     close(sfd);
141 }
```

在这里阻塞会带来两个问题,

问题一:在connect时, 由于设置了sfd是非阻塞的, 所以connect不会阻塞住, 在发送了连接请求后就立马判断是否连接成功, 但是此时就算顺利连接的情况下, 也要进行三次握手的过程, 所以很少能在用非阻塞描述符建立连接请求时, 得到0 (表示连接成功) 的返回值 (如果在网络特别通畅的情况下可能发生, 比如本地) 。所以这里的解决办法有两种:

- 方法一: 将设置非阻塞的语句放到connect后面去, 也就是在执行connect的时候, sfd还是一个阻塞套接字。但是这样再设置非阻塞, 作用就很小了。
- 方法二: 如果非阻塞文件描述符在connect以后, 如果只是因为正在进行与客户端的连接而失败, connect会返回EINPROGRESS的错误码, 这个错误码表示正在连接。所以在接受到这个错误码之后, 并不代表一定失败了。于是采取select阻塞监听sfd的写端, 如果sfd写端可以写了, 那么表示连接成功了, 如果select阻塞超时, 那么表示连接失败了。这里有两点需要注意:
 - 但是这里有一种特殊情况, 就是当连接出错时, 套接口描述符变成既可读又可写; (**注意:当一个套接口出错时,它会被select调用标记为既可读又可写;**) 所以即便触发sfd的写事件, 仍然需要加一层判断
 - 使用这种方法, 有一个很大的优点, 这也是非阻塞带来的。connect函数中如果使用的是阻塞文件描述符的话, 在执行connect函数时, 会一直阻塞在connect处等待服务器的响应, 最多等待一个超时时间。一般这个超时时间从75s到几分钟不等。并且不能自定义。但是如果使用方法二, 利用非阻塞加select的形式, 可以设置select的timeout参数, 可以自定义这个超时间。

问题二:在recv函数也就是接受对端数据时，因为非阻塞也会带来问题。可以看程序中，刚刚调用write函数，给服务端发送数据，然后客户端立马调用recv函数接受数据，但是这是往往数据还没来得及传送到客户端的端口处。由于sfd不是阻塞的文件描述符，所以recv在检查了没有数据可读的情况下，就会立马返回一个错误码。但是这并不代表recv出错，只是数据还没有到而已。（如果是阻塞的文件描述符就不会出现这种问题，因为如果没有数据可读，会阻塞在recv函数处，直到有数据可读再返回）

解决办法：当recv因为读取非阻塞文件描述符并且没有数据可读的情况而返回时，会产生EAGAIN或者EWOULDBLOCK错误码，这个错误码的意思是recv操作没有完成就返回了（也就是没有读到数据就返回了）。所以可以判断当错误码为EAGAIN或者EWOULDBLOCK时，就循环读取即可。

EAGAIN其实和EWOULDBLOCK是同一个值，源码如下所示：

```
1 #define EWOULDBLOCK EAGAIN /* Operation would block */
```

至于EINTR，则是如果被中断打断，那么同样循环读取。

注意：在muduo源码中， TcpClient类也是非阻塞的，但是在connect函数中，**如果判断connect函数的返回错误码是EINPROGRESS，就认为连接成功了**，并没有使用select去进一步的跟踪验证。个人觉得是设计不妥的地方，当然也有可能是因为后面有足够的保障措施去保证没有连接上会断开。而在recv这一步，也就是接受客户端的数据时，是将sfd挂到epoll队列上，监听sfd的读信号，这样每次使用IO读函数时，一定保证有数据可读，就不需要像我的程序中那样循环去测试了。

七、智能指针的vector数组和智能指针的普通数组

在我第一次看到这两个定义的时候，一下没有分清楚这两个定义的区别和联系。

```
1 std::vector<std::shared_ptr<Channel>> ChannelVector;//这个是存放  
shared_ptr<Channel>元素的vector  
2 std::shared_ptr<Channel> ChannelArray[MAXFDS];//这个也是存放  
shared_ptr<Channel>元素的普通数组
```

注意：这里两个都是数组，只不过一个是普通数组，一个是vector数组，并且存放的元素都是shared_ptr，很容易搞混了。

八、为什么在WebServer中，Channel中设置回调函数时，要使用右值引用作为入口参数 **(???疑问)**

源码如下：

```
1 void setReadHandler(CallBack &&readHandler) { readHandler_ =  
2     readHandler; }  
3 void setWriteHandler(CallBack &&writeHandler) {  
4     writeHandler_ = writeHandler;  
5 }  
6 void setErrorHandler(CallBack &&errorHandler) {  
7     errorHandler_ = errorHandler;  
8 }  
9 void setConnHandler(CallBack &&connHandler) { connHandler_ =  
10    connHandler; }
```

九、对于epoll类中fd2chan_的理解

在WebServer的epoll类中，有这样一个私有成员变量

```
1 std::shared_ptr<Channel> fd2chan_[MAXFDS];
```

这是一个普通数组，数组中存放了MAXFDS个shared_ptr指针，但是这些指针并没有在类构造函数中初始化，而是在epoll每加入一个channel时，会对应初始化fd2chan_数组中下标为channel->fd的智能指针。

所以其实这个数组是存放所有挂在epoll队列上的channel的，但是fd2chan_数组中Channel中的关注事件并不是与队列中的Channel对应的，因为会在getEventsRequest置零。

而在muduo中，有对应功能的私有变量如下：

```
1 typedef std::map<int, Channel*> ChannelMap;  
2 ChannelMap channels_;
```

和WebServer不同的是，muduo中使用map来存储epoll队列上的channel。

和Webserver的普通数组相比，map占用内存更小，并且很容易判断这个channel是否在队列上，但是搜索channel的时间更长。

```
1 void ConnectHandle(int fd, EventLoop* epoll_test, ChannelPtr  
&channel)  
2 {  
3     // 处理已经连接的客户端发送过来的数据  
4     /*if(!ActiveEvent[i]->GetRevent() & EPOLLIN)  
5     {  
6         continue;  
7     }*/
8     // 读数据  
9     char buf[5] = {0};  
10    int len;  
11    // 循环读数据  
12    while((len = recv(fd, buf, sizeof(buf), 0)) > 0)  
13    {  
14        // 数据打印到终端  
15        write(STDOUT_FILENO, buf, len);  
16        // 发送给客户端  
17        send(fd, buf, len, 0);  
18    }
```

```
19     if(len == 0)
20     {
21         printf("客户端断开了连接\n");
22         epoll_test->removeFromPoller(channel);
23         close(fd);
24     }
25     else if(len == -1)
26     {
27         if(errno == EAGAIN)
28         {
29             printf("缓冲区数据已经读完\n");
30         }
31         else
32         {
33             printf("recv error----\n");
34             close(fd);
35         }
36     }
37 }
38
39 void ConnectError(EventLoop* epoll_test, ChannelPtr &channel)
40 {
41     printf("连接出现错误\n");
42     epoll_test->removeFromPoller(channel);
43     close(channel->fd);
44 }
45
46 void SocketHandle(int socketfd, EventLoop* epoll_test)
47 {
48     struct sockaddr_in client_addr;
49     socklen_t client_len = sizeof(client_addr);
50     int connfd = ::accept4(socketfd, (sockaddr *)&client_addr,
51     &client_len, SOCK_NONBLOCK|SOCK_CLOEXEC);
52
53     if(connfd == -1)
54     {
55         perror("accept error");
56         abort();
57     }
58     cout<<"connect success";
59     ChannelPtr ConnectChannel(new Channel(connfd));
60     ConnectChannel->SetEvent(EPOLLIN|EPOLLET);
61     ConnectChannel->setReadHandler(std::bind(&ConnectHandle,
62     connfd, epoll_test, ConnectChannel));
63     ConnectChannel->setErrorHandler(std::bind(&ConnectError,
64     epoll_test, ConnectChannel));
65     epoll_test->addToPoller(ConnectChannel);
66     /*打印客户端信息*/
67     char ip[64] = {0};
68     printf("New Client IP: %s, Port: %d\n",
69         inet_ntop(AF_INET, &client_addr.sin_addr.s_addr, ip,
70         sizeof(ip)),
71         ntohs(client_addr.sin_port));
```

十、出现bad_weak_ptr异常

在网络库运行过程中，出现bad_weak_ptr异常，经过查询资料发现，是因为我把shared_from_this()函数放在构造类中使用了，所以出现这样的问题。根据博客<http://blog.csdn.net/g1036583997/article/details/65626749>，总结一下shared_from_this()两种常见的使用错误。

- 不能在类的构造函数中使用shared_from_this()函数
- 如果创建的类对象没有用shared_ptr包裹，而是直接用一个裸指针指向它，也是会报错的。

十一、同一个类的不同对象，可以互相访问私有变量

举例如下：

```
1 using namespace std;
2 class test
3 {
4 public:
5     test(int a_, int b_):a(a_),b(b_){}
6     test(test &t){a=t.a+1;b=t.b+1;}//可以访问同一个类中的不同对象下的私
7     void show(){cout<<a<<" " <<b<<endl;};
8     void show(test &t){cout<<t.a<<" " <<t.b<<endl;};//可以访问同一个类
9     中的不同对象下的私有变量
10    private:
11        int a;
12        int b;
13    };
14
15 int main()
16 {
17     test t1(1,2);
18     test t2(t1);
19     t1.show(t2);
20     t2.show(t1);
21     t1.show();
22     t2.show();
23
24     return 0;
25 }
```

结果如下

```
gmq@gmq-OptiPlex-3020:~/桌面/FILE$ ./a.out
2 3
1 2
1 2
2 3
```

十二、当使用类对象作为函数的入口参数时，尽量不要使用普通的对象传值，因为这样会重新执行一次构造函数，有可能造成bug，举例如下：

```

1 | using namespace std;
2 | class test
3 | {
4 | public:
5 |     test(int a_, int b_):a(a_),b(b_){}
6 |     test(test &t){a=t.a+1;b=t.b+1;}//可以访问同一个类中的不同对象下的私
7 |     //有变量
8 |     void show(){cout<<a<<" "=>b<<endl;};
9 |     void show(test &t){cout<<t.a<<" "=>t.b<<endl;}/*情况一*/
10 |    void show(test t){cout<<t.a<<" "=>t.b<<endl;}/*情况二*/
11 | private:
12 |     int a;
13 |     int b;
14 |
15 |
16 int main()
17 {
18     test t1(1,2);
19     test t2(t1);
20     t1.show(t2);
21     t2.show(t1);
22     t1.show();
23     t2.show();
24
25     return 0;
26 }

```

情况一结果:

```

gmq@gmq-OptiPlex-3020:~/桌面/FILE$ ./a.out
2 3
1 2
1 2
2 3

```

情况二结果:

```

gmq@gmq-OptiPlex-3020:~/桌面/FILE$ ./a.out
3 4
2 3
1 2
2 3

```

针对情况二分析:

主要看第一行和第二行的对比，其实这时，下面两句程序:

```

1 | t1.show(t2);
2 | t2.show(t1);

```

经过编译，会执行如下的情况

```
1 {
2     test temp(t2);
3     cout<<temp.a<<" "<<temp.b<<endl;
4 }
5 {
6     test temp(t1);
7     cout<<temp.a<<" "<<temp.b<<endl;
8 }
```

因为作为普通对象传值时，是先调用拷贝构造函数，所以就是调用了test(test &t){a=t.a+1;b=t.b+1;}函数。

十三、智能指针重载布尔运算

首先看以下这段示例代码：

```
1 shared_ptr<int> a;
2 if (a)
3     cout<<"exist";
4 else cout<<"empty";
```

在这段代码中，按理a是一个shared_ptr的类对象，如果一个类对象进行布尔判断，是不可以的，无法从一个类的类型转换成bool类型。

但是在shared_ptr源码中，对于布尔运算进行了重载，源码如下：

```
1 explicit operator bool() const
2 { return _M_ptr == 0 ? false : true; }
```

也就是用shared_ptr对象进行布尔运算或者将shared_ptr对象强转为布尔运算时，就是拿shared_ptr中的成员变量指针和NULL比较，如果非空就是true

十四、noncopyable中构造和析构函数是受保护权限的

将构造函数和析构函数设置为受保护权限，为了让子类可以调用，但是外类调用不了。

十五、如果类成员不是对象，是指针或者引用，就不会被析构函数给析构

```
1 #include <iostream>
2 using namespace std;
3
4 class test1
5 {
6 public:
7     test1(){}
8     ~test1(){cout<<"test1析构\n";}
9 }
10
11 class test2
12 {
13 public:
14     test2(test1 &t1,test1 *m1):t(t1), m(m1){cout<<"test2构造\n";}
15     ~test2(){cout<<"test2析构\n";}
16 private:
```

```

17     test1 &t;
18     test1 *m; //这两个成员变量都不会在test2析构时被析构，但是如果把t就
19     会了
20
21
22 int main()
23 {
24     test1 t1,t2;
25     {
26         test2 t3(t1,&t2);
27     }
28     cout<<"-----\n";
29     return 0;
30 }
```

```

gmq@gmq-OptiPlex-3020:~/桌面/FILE/test$ ./a.out
test2构造
test2析构
-----
test1析构
test1析构
264万
213万 ↑
```

十六、左值引用变量妙用

先看示例程序

```

1 class MutexLock:public noncopyable
2 {
3 public:
4     MutexLock(){pthread_mutex_init(&mutex,NULL);}
5     ~MutexLock(){pthread_mutex_destroy(&mutex);}
6     void lock(){pthread_mutex_lock(&mutex);}
7     void unlock(){pthread_mutex_unlock(&mutex);}
8     pthread_mutex_t GetMutex(){return mutex;}
9 private:
10    pthread_mutex_t mutex;
11 };
12
13 class MutexGuard:public noncopyable
14 {
15 public:
16     MutexGuard(MutexLock &mutex_):mutex(mutex_) //普通对象的话，这里会报
17     错
18     {
19         mutex.lock();
20     }
21     ~MutexGuard()
22     {
23         mutex.unlock();
24     }
25 private:
26     MutexLock &mutex; //左值引用传递，重点！！！
27 };
```

重点就在于，为什么使用左值引用成员变量。

这里有两点：

- 当MutexGuard类对象析构时，不会随着MutexGuard析构函数，mutex一起被析构
- 由于MutexLock是不可拷贝的，所以如果是普通对象，在初始化列表会报错

但其实这里使用指针是一个效果，不太理解为什么不适用指针，指针和左值引用功能是差不多的，可能是因为不希望使用裸指针吧。

十七、std::function变量作为函数入口参数时，如果是引用传值，那必须是const

举个例子，看下面几种传值方式：

```
1 typedef std::function<void ()> func;
2 int callback(const func &f); //引用传值，正确
3 int callback(func f); //普通传值，正确
4 int callback(func &f); //错误，编译阶段会报错
```

至于为什么第三种方式编译会报错，网上有一个博客<https://www.jianshu.com/p/c4c84b073413>说法如下

这是因为调用callback函数的地方生成了一个临时的std::function()对象，是一个右值，否则编译会报错。

???但是不是很理解，有待解决。

十八、wakeup使用上边沿还是水平触发比较好

其实归根结底还是那个问题，就是不在epoll阻塞等待的时候，有关注的文件描述符事件产生了，那么会不会被关注到，写了一个小试验，结论是会的

```
1 #include <iostream>
2 #include <unistd.h>
3 #include <sys/eventfd.h>
4 #include <sys/epoll.h>
5 #include <pthread.h>
6 using namespace std;
7 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
8 static int t1=0;
9 static int t2=0;
10 void *thr1(void * agr)
11 {
12     int* a=(int*)agr;
13     sleep(1);
```

```

14     pthread_mutex_lock(&mutex); //上锁
15     t1++;
16     pthread_mutex_unlock(&mutex); //解锁
17     uint64_t one=1;
18     write(*a, &one, 8);
19     return NULL;
20 }
21 void *thr2(void * agr)
22 {
23     int* a=(int*)agr;
24     sleep(3);
25     pthread_mutex_lock(&mutex); //上锁
26     t2++;
27     pthread_mutex_unlock(&mutex); //解锁
28     uint64_t one=1;
29     write(*a, &one, 8);
30     return NULL;
31 }
32
33 int main()
34 {
35
36     int evefd=eventfd(0, EFD_NONBLOCK);
37     int epfd=epoll_create(300);
38     struct epoll_event ev;
39     struct epoll_event all[300];
40     ev.events=EPOLLIN|EPOLLET; //设置为上升沿触发，也就是只会触发一次
41     ev.data.fd=evefd;
42     epoll_ctl(epfd, EPOLL_CTL_ADD, evefd, &ev);
43
44     pthread_t tid1,tid2;
45     pthread_create(&tid1, NULL, thr1, &evefd);
46     pthread_create(&tid2, NULL, thr2, &evefd); //, 没写线程回收函数, 有
漏洞
47     while(1)
48     {
49         int sol=epoll_wait(epfd, all, sizeof(all)/sizeof(all[0]),
50 -1);
51         {
52             for (int i=0; i<sol; i++)
53             {
54                 int fd=all[i].data.fd;
55                 if (fd==evefd)
56                 {
57                     pthread_mutex_lock(&mutex); //上锁
58                     cout<<t1<<" "<<t2<<endl;
59                     pthread_mutex_unlock(&mutex); //解锁
60                 }
61                 sleep(4);
62                 cout<<"loop one time\n";
63             }
64         }
65     }
66     close(evefd);

```

```
67     return 0;  
68 }
```

整个逻辑应该是在线程1唤醒epoll_wait阻塞时，并在主线程执行sleep(4);过程中，线程2又唤醒了一次，结果是在主线程中还是接受到了线程2的事件。这个和我的博客中http://blog.csdn.net/qq_34489443/article/details/100574709所说的条件变量不在阻塞时，产生的条件变量却也在条件变量阻塞以后接收到。

???很奇怪，可是我在
windows中使用条件变量
时，却是会发生丢失的，以
后有时间一定要多研究一下
这个

十九、CountDownLatch为什么不可以重新设置计数器的值

???CountDownLatch为什
么不可以重新设置计数器的
值

二十、端口号80必须在root用户下使用

端口号小于1024都必须在root用户下调用，而且和socket函数没有关系，没有说socket函数必须要root用户下才可以执行的。

二十一、muduo类型网络库线程分析

把Server类所在线程称为基本线程，基本线程主要的任务是创造Server类，创造EventLoop线程池，监听监听文件描述符，也就是scoket函数得到的描述符。当有新连接来时，创建新的HttpConnect对象(自己的库中)或者TcpConnection对象(muduo库中)，并将这些对象分配给EventLoop线程池中带有EventLoop的线程去监听管理。只有当EventLoop线程池中的线程数为0时，才会由基本线程来监听新连接的文件描述符

而EventLoop线程池中每一个线程被称为IO线程，这些线程的地位是等同的，都被基本线程给管理，基本线程可以结束他们的生命周期。每个IO线程都执行同样的任务：

- poll阻塞监听挂在本EventLoop上的文件描述符
- 处理活跃的文件描述符事件
- 处理定时器超时事件
- 处理任务队列中的事件

所以如果想要让IO线程执行这些任务以外的其他任务，就需要使用runInLoop和queueInLoop两个函数。比如对epoll队列的操作(删除，增加，改变)，又比如对文件描述符IO的操作。当然还有一种方法是放在文件描述符的事件处理函数中去执行，但是比如增减文件描述符就不可行了，因为当监听到一个新的连接描述符时，是在基本线程中执行的，但是我想要增加到EventLoop线程池中的一个线程中去，显然不可以在EventLoop线程中所属的文件描述符的事件处理函数中去执行。

所以其实runInLoop和queueInLoop两个函数的作用是：想要eventloop执行上面四个任务以外的其他任务，并且发布任务的线程很有可能不是eventloop所属的线程，所以就需要使用runInLoop函数，而queueInLoop一般都是在runInLoop分配的函数中使用的，为了让一些任务再放回任务队列中，等任务队列中所有任务都执行结束以后再去执行，一般这种任务都是扫尾工作，需要等到当前的所有任务都结束再去完成，比如一些对象的销毁，完成某项任务以后的处理函数等。

二十二、std::string中find和substr函数的细节研究

find就在一个字符串中找到有没有和另一个字符串相同的子串，声明如下：

```
1 | size_t find (const string& str, size_t pos = 0) const;
```

str:需要查找的字符串

pos:开始查找的位置

返回值大于等于0，表示存在要找的子串，并且这个子串的第一个字符位置为返回值。

substr就是把字符串中指定位置的子串给提取出来，声明如下：

```
1 | string substr (size_t pos = 0, size_t len = npos) const;
```

pos:起始位置

len:子串的长度，npos就是从pos位置到字符串结尾的长度

所以子串的范围是[pos,len)，左闭右开的。

二十三、C++中switch的限制

switch中比较对象只可以是整型数据或者可以转换成整型数据的其他对象，所以大部分自定义类对象都不可以适用switch，包括string等。所以要慎用。

二十四、消息长度

在http请求头中，有一个很重要的参数就是Content-Length，这个参数在HTTP1.0版本中是可有可无的，因为HTTP1.0不支持长连接，所以我之前写代理服务器时，一直使用的是1.0版本，所以没有这个困扰。但是在我写现在这个服务器时，因为需要长连接，所以就使用的是HTTP1.1版本，Content-Length参数是一定需要的，因为这有这个参数，

才可以判断当前的响应数据包有多大，是否接收完毕。如果没有这个参数，客户端就会一直延时等待一段时间，然后再结束接收，显示响应内容。所以这个参数很重要。虽然网上说一定要是keep-alive的连接下，才一定需要Content-Length参数。可以参考这篇文章：<https://www.cnblogs.com/lovelacelee/p/5385683.html>

但是具体是怎样还没有弄清楚，有待进一步弄懂

二十五、当服务器主动关闭后，立马重启服务器，很有可能显示端口被占用

因为当服务器断开连接时，服务器很有可能正在TIME_WAIT状态，这时候可以立马用netstat去查看，可以看到该端口正处于time_wait状态。

```
gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/MyWeb4$ ./test1
epoll_add success
accept error: Address already in use
-3socket_bind_listen error
epoll_add success
listen error: Bad file descriptor
^C
gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/MyWeb4$ netstat -an | grep 8080:9287722/0
tcp        0      0 127.0.0.1:8080 0.0.0.0:136166      TIME_WAIT
```

二十五、当服务器主动关闭后，立马重启服务器，很有可能显示端口被占用

因为当服务器断开连接时，服务器很有可能正在TIME_WAIT状态，这时候可以立马用netstat去查看，可以看到该端口正处于time_wait状态。

https://blog.csdn.net/qq_39287722/article/details/78720249

二十六、文件描述符和FILE结构体

使用open函数可以得到打开文件的文件描述符，而使用fopen，得到的是一个FILE结构体，那么这两个有什么区别呢，我的理解是，FILE结构体是对文件描述符的进一步封装。所以FILE中包含了文件描述符，并且还增加了一个缓冲区，是当对被fopen打开的文件进行读写操作时，会先存放在缓冲区中，然后再写到文件中，或者先从文件写到缓冲区，再被读到程序中。而这个缓冲区的大小是需要通过setbuffer函数来设置的。

所以fopen,fclose,fflush,setbuffer是一套函数，完全不适用于open打开的文件的。并且注意与之配套的fwrite和fread应该是线程安全的，不过为了线程安全，最好的做法是使用fwrite_unlock并且在之前认为加锁。也就是不用自带的线程安全函数。

这里着重介绍一下fwrite和fread函数，这两个函数是为了配套FILE结构体，而出现的IO函数，首先说明这两个函数与read和write的区别：

- fwrite和fread是对read和write的封装，也就是fwrite和fread的底层也是用到了read和write，并且read和write是属于系统调用，非常耗费时间，而fwrite和fread则属于C语言封装，是为了避免重复使用read和write系统调用
- 从使用上，fwrite会先把数据写到缓冲区中，只有等缓冲区满了，或者调用flush函数时，才会调用write函数将缓冲区的内容写到内核缓冲区中，然后再定期将该内容写到对应文件中。fread也是一样，fread首先会调用read函数，从内核缓冲区中读取超过所需的数据存储到缓冲区中，然后每次都是从自己的缓冲区中去读取数据。

写了一个小例子来测试这个fwrite函数：

```
1 int main()
2 {
3     FILE* fp = ::fopen("file.txt", "ae");
```

```

4   char buffer[1024];
5   ::setbuffer(fp, buffer, 1024);
6   char temp[] = "hello world\n";
7   cout<<temp<<sizeof temp<<endl;
8   size_t n = fwrite(temp, 1, sizeof temp, fp);
9   cout<<"写入字符个数"<<n<<endl;
10  cout<<"-----sleep-----\n";
11  sleep(3);
12  cout<<"-----sleep finished-----\n";
13  fflush(fp);
14  return 0;
15 }

```

当睡眠时，如果你去打开文件查看，此时hello world是没有输入进去的，只有等到fflush以后，才会输入进去。

注意：平时常见的stdin,stdout,stderr宏都是FILE结构体，而不是文件描述符，他们各自对应的文件描述符分别为0,1,2。所以对这三个宏的操作也可以使用如上的函数，只是这些宏不需要fopen, fclose这两个函数了，可以直接读写。测试源码如下：

```

1 int main()
2 {
3     char buffer[1024];
4     ::setbuffer(stdout, buffer, 1024);
5     char temp[] = "hello world";
6     size_t n = fwrite(temp, 1, sizeof temp, stdout);
7     //如果这两句程序打开，就不会出现和上面例子一样的情况，在睡眠结束之后再输出，
8     //因为stdout输出时和普通文件不一样，除了内存满了，或者fflush，还有一种情况也会输出，就是遇到\n
9     //cout<<"写入字符个数"<<n<<endl;
10    //cout<<"-----sleep-----\n";
11    sleep(3);
12    cout<<"-----sleep finished-----\n";
13    fflush(stdout);
14    return 0;
15 }

```

有关这三个宏的使用，可以参考这篇文章：<https://www.cnblogs.com/Suzkfly/p/10367376.html>

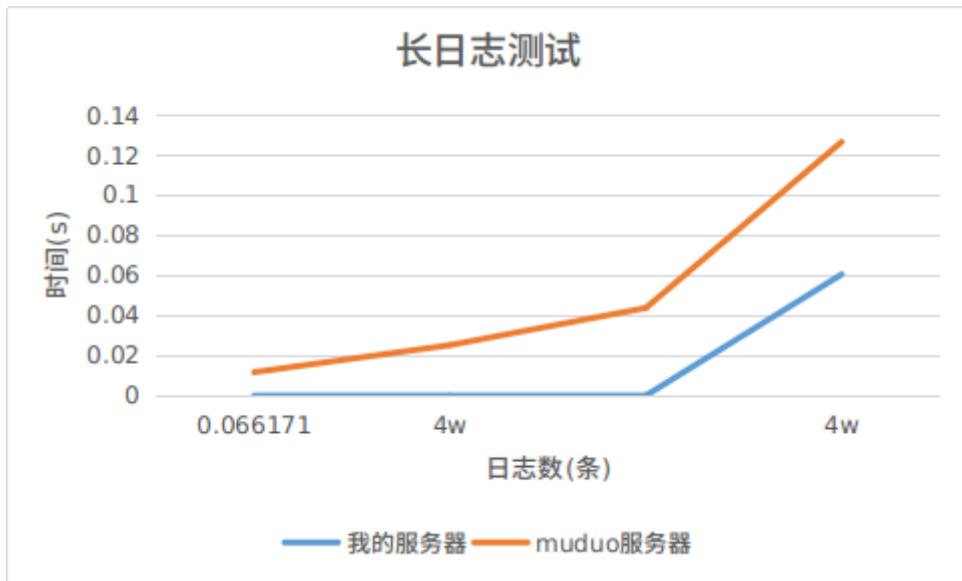
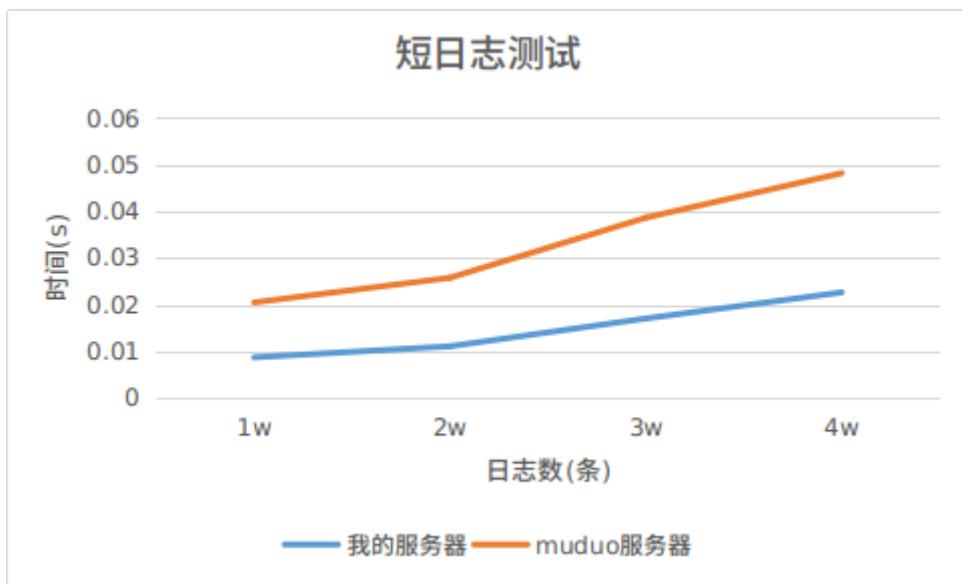
二十七、前向声明只适用于指针和引用

在使用前向声明时，需要注意如下问题：

- 1.前向声明的类不能定义对象。
- 2.可以用于定义指向这个类型的指针和引用。
- 3.可以用于申明使用该类型作为形参或返回类型的函数。

二十八、日志测试结果

我的服务器	muduo服务器	输出日志个数	每条日志长度
0.008693	0.011784	1w	20-40字节
0.011663	0.016167	1w	100-150字节
0.011066	0.014693	2w	20-40字节
0.025211	0.026666	2w	100-150字节
0.017035	0.02161	3w	20-40字节
0.043772	0.042018	3w	100-150字节
0.02262	0.02556	4w	20-40字节
0.060553	0.066171	4w	100-150字节



1W我的服务器:

```
gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/MyWeb4/base/test_log$ ./run  
pid = 29190  
-----pid:29191-----  
0.008693  
gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/MyWeb4/base/test_log$ ./run 1  
pid = 29198  
-----pid:29199-----  
0.011663
```

muduo服务器:

```
gmq@gmq-OptiPlex-3020:~/桌面/FILE/jmuduo/muduo/base/tests$ ./run 的类不能定  
pid = 29194  
0.011784  
gmq@gmq-OptiPlex-3020:~/桌面/FILE/jmuduo/muduo/base/tests$ ./run 1 可以用于定义指向这  
pid = 29196  
0.016167
```

2W我的服务器:

```
gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/MyWeb4/base/test_log$ ./run  
pid = 29658  
-----pid:29659-----./run 1 定义指向这  
0.011066  
gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/MyWeb4/base/test_log$ ./run 1  
pid = 29660  
-----pid:29661-----  
0.025211
```

muduo服务器:

```
gmq@gmq-OptiPlex-3020:~/桌面/FILE/jmuduo/muduo/base/tests$ ./run  
pid = 29638  
0.014693  
gmq@gmq-OptiPlex-3020:~/桌面/FILE/jmuduo/muduo/base/tests$ ./run 1  
pid = 29649  
0.026666
```

3W我的服务器:

```
gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/MyWeb4/base/test_log$ ./run  
pid = 29877  
-----pid:29878-----  
0.017035  
gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/MyWeb4/base/test_log$ ./run 1  
pid = 29881  
-----pid:29882-----  
0.043772
```

muduo服务器:

```
gmq@gmq-OptiPlex-3020:~/桌面/FILE/jmuduo/muduo/base/tests$ ./run  
pid = 29867  
0.021610  
gmq@gmq-OptiPlex-3020:~/桌面/FILE/jmuduo/muduo/base/tests$ ./run 1 muduo 服务器:  
pid = 29875  
0.042018
```

4W我的服务器:

```

ogmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/MyWeb4/base/test_log$ ./run
pid = 30073
master@mpc64:/base/test_log$ ./run
0.022620
ogmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/MyWeb4/base/test_log$ ./run 1
pid = 30075
master@mpc64:/base/test_log$ ./run 1
0.060553

```

muduo服务器:

```

gmq@gmq-OptiPlex-3020:~/桌面/FILE/jmuduo/muduo/base/tests$ ./run
pid = 30063
0.025560
gmq@gmq-OptiPlex-3020:~/桌面/FILE/jmuduo/muduo/base/tests$ ./run 1
ogmq@gmq-OptiPlex-3020
pid = 30067
0.066171

```

二十九、muduo日志再解读

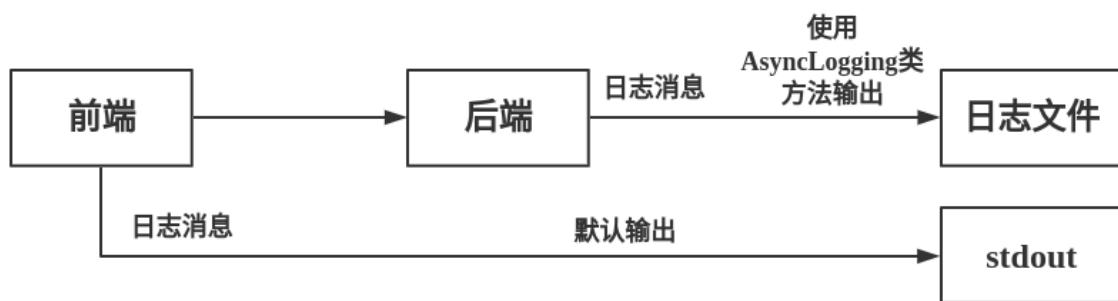
在把muduo日志库重新写一遍以后，我对日志库有了更加深刻的理解，这里我就把自己的理解再次记录下来：

1.整体架构

在muduo中有关日志库的文件主要有四个，分别是

- LogFile头文件和源文件，包含了File类和LogFile类
- LogStream头文件和源文件，包含了FixedBuffer类和LogStream类
- Logging头文件和源文件，包含了Logger类
- AsyncLogging头文件和源文件，包含了AsyncLogging类

在muduo中，分为前端和后端，所谓前端就是产生日志消息的地方，前端可以是任意线程，任意代码位置，只要你使用了LOG_XXX宏来产生代码，那么该处就是前端。后端则是负责将日志消息从其他线程收集起来，并集中起来输出。后端在唯一线程上发生，就是由AsyncLogging类创建的一个线程，这个线程就是不断地收集其他线程产生的日志消息，并输出到日志文件或者终端上。



所以根据这个过程来说明上述类的作用：

- File类主要是操作日志文件的类，包括打开日志文件，写入日志文件，刷新流等。而LogFile类则是对File类的进一步封装（LogFile是LogFile成员变量），并且添加了对日志文件的管理，包括①创建统一命名格式的日志。②如果隔一段时间，会自动刷新用户层缓冲区。③如果日志创建时间和当前时间不在一天时，会回滚日志。**相当于LogFile文件实现了从后端写到日志文件的功能。**
- FixedBuffer类是日志库中的缓冲区，是双缓冲的基础构件，以及在LogStream流中也使用到了FixedBuffer类作为缓冲区。LogStream则是日志流，主要重载了<<符号，也就是把<<后面的内容变成字符串输出到LogStream类中的FixedBuffer缓冲区中。

- Logger类对LogStream进一步封装（**LogStream是Logger类的成员变量**），主要实现了通过宏以及<<可以编写日志内容，**也是日志库最后的使用接口**，并且自动加入了日志产生的时间，日志产生的文件以及行数，形成一条完整的日志，在Logger析构函数中，将这一条日志消息调用output函数，输出出去。默认的输出是stdout，而可以使用AsyncLogging类中的函数输出到日志文件，也可以自定义输出函数。所有Logger类在使用时都是创建一个临时变量，创建完以后立刻消失，所以输出函数被放在了析构函数中。**LogStream文件和Logging文件相当于实现了前端产生日志消息的功能，并提供了后端的接口或者直接输出到stdout上。**
- AsyncLogging类则是实现异步日志的核心类，AsyncLogging类中单独开启了一个线程去循环条件等待双缓冲是否满了，如果满了，就把缓冲块替换下来，然后将日志消息输出到日志文件当中。**所以AsyncLogging文件实现了后端的功能，连接了Logger类和LogFile类**（AsyncLogging类的append函数作为Logger的输出函数，另外LogFile类作为AsyncLogging类线程回调函数的局部变量）

所以其实只适用LogStream和Logging就可以实现日志打印，而加上LogFile和AsyncLogging就可以实现异步日志。

2. 缓冲区分

在日志库中，有好多个缓冲，这里详细区别一下，在日志库中总共有三个地方用到了缓冲

- 在File类中使用到了缓冲，**这个缓冲是一个普通的char数组**，这个缓冲是因为对FILE结构体进行操作时，可以给fwrite设置缓冲，也就是在向磁盘上的日志文件写内容时，是先写到File类的缓冲中去，只有等到File类的缓冲写满，或者调用::fflush函数时，才会将缓冲中的内容调用系统函数write写入到内核中去，这样是因为write这种IO操作比较消耗时间，所以尽可能少的调用write。
- 在LogStream类中也有缓冲，**这个缓冲是由一个FixedBuffer类对象构成的**，但是尺寸比较小，**这个缓冲只会存储一条日志消息**。也就是在前端调用LOG宏生成一条日志消息以后就会暂时保存到FixedBuffer缓冲中，等到这个临时变量析构时，会将日志消息从这个缓冲中取出并输出给stdout或者AsyncLogging类中的双缓冲。
- 在AsyncLogging类中使用到了双缓冲，**这个双缓冲是由两个FixedBuffer对象以及一个FixedBuffer数组构成的**。这个缓冲主要用于**存储后端收集到的所有日志消息**，**存储到一定限度就会输出给File类的缓冲，然后写进日志文件中**。这个双缓冲的运作原理陈硕讲的很多，这里我讲一下我的理解：就是由一个主缓冲和一个备用缓冲构成，收集到的日志消息优先放置在主缓冲中，如果主缓冲放满了，那么就把主缓冲放进缓冲数组中，然后将备用缓冲拿来作为主缓冲，如果备用缓冲也放满了，就将备用缓冲也放到缓冲数组中，并新建一个缓冲继续存放。

所以每一条异步日志消息都经过这三种缓冲最后才到日志文件中的，日志消息的传递过程如下：

产生日志消息->LogStream类的缓冲->AsyncLogging类双缓冲->File类fwrite缓冲->日志文件

3. 日志库中的线程局部变量

在我测试我写的日志库时，发现我写的库输出速度只有muduo库的一半不到，后来我发现是因为muduo库中Logging文件中Logger::Impl::formatTime函数的问题，其中将时间组织成规则字符串的函数，无论是snprintf还是strftime非常的耗时，所以muduo中使用了一个小技巧，就是将上一次的时间字符串保存起来，如果这次的时间和上次一样，那么就直接使用上一次的字符串就好了，就大大缩短了时间，而用来保存上一次时间字符串的变量是由thread关键词修饰的，也就是这个字符串记录的是当前线程上一次调用日志库的时间。

三十、 getopt_long 函数的理解

在webbench中，看到了 getopt_long 函数，从来没有碰见过，在查阅了一些文件以后，写下自己的理解。

getopt_long函数主要是为了实现解析入口参数（命令行参数）的功能，平时可以看到，在终端使用命令时，后面可以跟很多参数，比如ls，后面可以跟-a,-l等参数，拿这些参数是如何识别并执行具体程序的呢，主要依靠**getopt_long**函数。

1.长选项和短选项

在讲解getopt_long函数之前，首先讲解一下长选项和短选项。

- 长选项就是命令参数前加--，短选项就是命令参数前加-，以ls中部分命令行参数为例，所有一个杠的都是短选项，所有两个杠的都是长选项

```
gnq@gnq-OptiPlex-3020:~$ ls --help
用法: ls [选项]... [文件]...
List information about the FILEs (the current directory by default).
Sort entries alphabetically if none of -cftuvSUX nor --sort is specified.

必选参数对长短选项同时适用。
-a, --all          不隐藏任何以. 开始的项目
-A, --almost-all   列出除. 及.. 以外的任何项目
--author          中的双缓冲。
-b, --escape        与-l 同时使用时列出每个文件的作者
--block-size=SIZE   以八进制溢出序列表示不可打印的字符
scale sizes by SIZE before printing them; e.g.,
  '--block-size=M' prints sizes in units of
    1,048,576 bytes; see SIZE format below
-c                do not list implied entries ending with ~ 志文件中。这个双缓
                  with -lt: sort by, and show, ctime (time of last
                  modification of file status information); 构成，收集到的
                  with -l: show ctime and sort by name;
                  otherwise: sort by ctime, newest first. 然后将备用缓冲拿
                  list entries by columns
colorize the output; WHEN can be 'always' (default
  if omitted), 'auto', or 'never'; more info below
-d, --directory    list directories themselves, not their contents
-D, --dired         generate output designed for Emacs' dired mode
-f                do not sort, enable -u, disable -ls --color
-F, --classify     append indicator (one of */=>@|) to entries Stream类的缓冲
```

- 长选项一般都是参数名比较完整的选项，而短选项一般是参数名缩写的选项，但是大部分情况下长选项和短选项是互相对应的。也就是你想输入一个参数，既可以输入长选项也可以输入短选项，实现的功能是一样的。

2.getopt_long函数

```
1 #include <unistd.h>
2 extern char *optarg;
3 extern int optind, opterr, getopt;
4 #include <getopt.h>
5 int getopt_long(int argc,
6                 char * const argv[],
7                 const char *optstring,
8                 const struct option *longopts,
9                 int *longindex);
```

参数解析:

①argc和argv: 就是main函数中的入口参数，也就是命令行中的入口参数数组

②optstring: 是表示短选项的字符串

形式如ab::c:d，分别表示程序支持的命令行短选项有-a、-b、-c、-d，冒号含义如下：

- 只有一个字符，不带冒号——只表示选项，如-a
 - 一个字符，后接一个冒号——表示选项后面必须带一个参数，如-c 100或者-c100

- 一个字符，后接两个冒号——表示选项后面带一个可选参数，即参数可有可无，如果带参数，则选项与参数直接不能有空格，如-b50

注意：如果选项后面加参数，那么参数会存储到全局变量optarg中去

③longopts: 表示长选项结构体，结构体源码如下：

```

1 struct option
2 {
3     const char *name;
4     int      has_arg;
5     int      *flag;
6     int      val;
7 };
8
9 //以webbench为例
10 static const struct option long_options[]=
11 {
12     {"force", no_argument, &force, 1}, //当使用长选项--force时，会把1赋值给
13     //force变量
14     {"reload", no_argument, &force_reload, 1},
15     {"time", required_argument, NULL, 't'}, //当使用--time时，会让
16     // getopt_long函数返回字符t，相当于把短选项-t和长选项--time绑定起来
17     {"help", no_argument, NULL, '?'},
18     {"http09", no_argument, NULL, '9'},
19     {"http10", no_argument, NULL, '1'},
20     {"http11", no_argument, NULL, '2'},
21     {"get", no_argument, &method, METHOD_GET},
22     {"head", no_argument, &method, METHOD_HEAD},
23     {"options", no_argument, &method, METHOD_OPTIONS},
24     {"trace", no_argument, &method, METHOD_TRACE},
25     {"version", no_argument, NULL, 'V'},
26     {"proxy", required_argument, NULL, 'p'},
27     {"clients", required_argument, NULL, 'c'},
28     {NULL, 0, NULL, 0} //必须存在，否则会报错
29 };

```

- name:表示长选项的名称，也就是在--后面应该填写的字符。
- has_arg:表示选项后面是否携带参数。该参数有三个不同值，如下：
 - no_argument(或者是0)时 ——参数后面不跟参数值，eg: --version, --help
 - required_argument(或者是1)时 ——参数输入格式为：--参数值 或者 --参数=值。eg: --dir=/home
 - optional_argument(或者是2)时 ——参数输入格式只能为：--参数=值
- flag:只可以是空或者非空
 - 如果flag为NULL，那么当选中某个长选项的时候，getopt_long将返回val值。eg，以help为例，如果使用--help，getopt_long函数的返回值就是字符?
 - 如果flag不为空，那么当选中某个长选项的时候，getopt_long将返回0，并且将flag指针参数指向val值。eg，以force为例，使用--force时，getopt_long函数返回0，并且force变变量的值为1。
- val: 就是和flag配合使用

注意：longopts的最后一个元素必须是全0填充，否则会报段错误

④longindex: longindex可以填NULL,如果不填NULL,它指向的变量将记录当前找到参数符合longopts里的第几个元素的描述,即是longopts的下标值。

⑤返回值

- 如果短选项找到，那么将返回短选项对应的字符。
 - 如果长选项找到，如果flag为NULL，返回val。如果flag不为空，返回0
 - 如果遇到一个选项没有在短字符、长字符里面。或者在长字符里面存在二义性的，返回"?"
 - 如果解析完所有字符没有找到（一般是输入命令参数格式错误，eg: 连斜杠都没有加的选项），返回"-1"
 - 如果选项需要参数，忘了添加参数。返回值取决于optstring，如果其第一个字符是":："，则返回":："，否则返回"?"。

⑥全局变量optarg, optind, opterr, optopt

- (1) `optarg`: 表示当前选项对应的参数值。
 - (2) `optind`: 表示的是下一个将被处理到的参数在`argv`中的下标值。
 - (3) `opterr`: 如果`opterr = 0`, 在`getopt`、`getopt_long`、`getopt_long_only`遇到错误将不会输出错误信息到标准输出流。`opterr`在非0时, 向屏幕输出错误。
 - (4) `optopt`: 表示没有被未标识的选项。

可以参考这篇博客: https://blog.csdn.net/qq_33850438/article/details/80172275

三十一、压测结果

muduo

1000个客户端，60秒时间

muduo长连接:

```
gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/WebBench$ ./webbench -c 1000 -t 60 -2 -k http://127.0.0.1:8000/
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.

Request: GET / HTTP/1.1
User-Agent: WebBench 1.5
Host: 127.0.0.1
Connection: Keep-Alive
boost::bind(&TcpConnection::sh...
void TcpConnection::shutdown...
Running info: 1000 clients, running 60 sec.

Speed=8322071 pages/min, 29959456 bytes/sec.
Requests: 8322071 susced, 0 failed.
```

muduo短连接http1.1版本：

```
gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/WebBench$ ./webbench -c 1000 -t 60 -2 http://127.0.0.1:8000/
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.

Request:
GET / HTTP/1.1
User-Agent: WebBench 1.5
Host: 127.0.0.1
Connection: close

Runing info: 1000 clients, running 60 sec.

Speed=2801698 pages/min, 8872021 bytes/sec.
Requests: 2801690 susceed, 8 failed.
```

muduo短连接http1.1版本:

```
gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/0.0.1:8000/hello
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.

Request:
GET /hello HTTP/1.1
User-Agent: WebBench 1.5
Host: 127.0.0.1
Connection: close
```

muduo短连接http1.0版本:

```
gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/WebBench$ ./webbench -c 1000 -t 60 http://127.0.0.1:8000/
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.

Request:
GET / HTTP/1.0
User-Agent: WebBench 1.5
Host: 127.0.0.1

Runing info: 1000 clients, running 60 sec.

Speed=699254 pages/min, 1019101 bytes/sec.
Requests: 657519 susceed, 41735 failed.
```

500个客户端, 60秒时间

muduo长连接:

```
gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/WebBench$ ./webbench -c 500 -t 60 -k http://127.0.0.1:8000/
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.
```

muduo短连接http1.1版本:

```
gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/0.0.1:8000/hello
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.

Request:
```

muduo短连接http1.1版本:

```
gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/WebBench$ ./webbench -c 500 -t 60 -2 http://127.0.0.1:8000/
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.

Request:
GET / HTTP/1.1
User-Agent: WebBench 1.5
Host: 127.0.0.1
Connection: close

Runing info: 500 clients, running 60 sec.

Speed=2472467 pages/min, 7829472 bytes/sec.
Requests: 2472465 susceed, 2 failed.
```

Runing info: 500 clients, running 60 sec.

Speed=2702689 pages/min, 8558512 bytes/sec.

Requests: 2702688 susceed, 1 failed.

muduo短连接http1.0版本:

```
gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/WebBench$ ./webbench -c 500 -t 60 http://127.0.0.1:8000/
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.

Request:
GET / HTTP/1.0
User-Agent: WebBench 1.5
Host: 127.0.0.1

Running info: 500 clients, running 60 sec.

Speed=2702689 pages/min, 8558512 bytes/sec.
Requests: 2702688 succeeded, 1 failed.
```

100个客户端，60秒时间

muduo长连接:

```
gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/WebBench$ ./webbench -c 100 -t 60 -2 -k http://127.0.0.1:8000/
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software

Request:
GET / HTTP/1.1
User-Agent: WebBench 1.5
Host: 127.0.0.1
Connection: Keep-Alive

Runing info: 100 clients, running 60 sec.

Speed=4320948 pages/min, 3600787 bytes/sec.
Requests: 4320946 susceed, 2 failed.

MyServer短连接http1.0版本:

gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master
:8080/
Webbench - Simple Web Benchmark 1.5
```

muduo短连接http1.1版本:

```
gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/WebBench$ ./webbench -c 100 -t 60 -2 http://127.0.0.1:8000/
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.

Request: GET / HTTP/1.1
User-Agent: WebBench 1.5
Host: 127.0.0.1
Connection: close

Running info: 100 clients, running 60 sec.
Speed=2764112 pages/min, 8753012 bytes/sec.
Requests: 2764110 sucseed, 2 failed.
```

muduo短连接http1.0版本：

```
gma@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/WebBench$ ./webbench -c 100 -t 60 http://127.0.0.1:  
8000/  
Webbench - Simple Web Benchmark 1.5  
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.  
  
Request:  
GET / HTTP/1.0  
User-Agent: WebBench 1.5  
Host: 127.0.0.1  
  
Runing info: 100 clients, running 60 sec.  
Speed=728017 pages/min, 1128426 bytes/sec.  
Requests: 728017 susceed, 0 failed.  
  
Runing info: 100 clients, running 60 sec.  
Speed=2726546 pages/min, 8634062 bytes/sec.  
Requests: 2726546 susceed, 0 failed.  
  
MyServer  
1000个客户端, 60秒时间
```

MyServer

1000个客户端，60秒时间

MyServer长连接:

```
gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/WebBench$ ./webbench -c 1000 -t 60 -2 -k http://127.0.0.1:8080/
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.

Request:
GET / HTTP/1.1
User-Agent: WebBench 1.5
Host: 127.0.0.1
Connection: Keep-Alive
Content-Type: application/x-www-form-urlencoded

Running info: 1000 clients, running 60 sec.
Speed=9847154 pages/min, 16411918 bytes/sec.
Requests: 9847154 succeed, 0 failed.
```

MyServer短连接http1.1版本:

```
gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/WebBench$ ./webbench -c 1000 -t 60 -2 http://127.0.0.1:8080/
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.

Request:
GET / HTTP/1.1
User-Agent: WebBench 1.5
Host: 127.0.0.1
Connection: close

Running info: 1000 clients, running 60 sec.
Speed=4083690 pages/min, 3403073 bytes/sec.
Requests: 4083689 succeed, 1 failed.
```

muduo短连接http1.0版本:

```
gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/WebBench$ ./webbench -c 1000 -t 60 -2 http://127.0.0.1:8080/hello
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.

Request:
GET /hello HTTP/1.0
User-Agent: WebBench 1.5
```

MyServer短连接http1.0版本:

```
gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/WebBench$ ./webbench -c 1000 -t 60 http://127.0.0.1:8080/
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.

Request:
GET / HTTP/1.0
User-Agent: WebBench 1.5
Host: 127.0.0.1

Running info: 1000 clients, running 60 sec.
Speed=4230030 pages/min, 3525020 bytes/sec.
Requests: 4230025 succeed, 5 failed.
```

muduo长连接:

```
gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/WebBench$ ./webbench -c 1000 -t 60 -2 http://127.0.0.1:8080/hello
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.

Request:
GET /hello HTTP/1.1
User-Agent: WebBench 1.5
Host: 127.0.0.1
Connection: Keep-Alive

Running info: 100 clients, running 60 sec.
```

500个客户端， 60秒时间

MyServer长连接:

```
gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/WebBench$ ./webbench -c 500 -t 60 -2 -k http://127.0.0.1:8080/
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.

Request:
GET / HTTP/1.1
User-Agent: WebBench 1.5
Host: 127.0.0.1
Connection: Keep-Alive

Running info: 500 clients, running 60 sec.
Speed=10840119 pages/min, 18066860 bytes/sec.
Requests: 10840119 succeed, 0 failed.
```

MyServer短连接http1.1版本:

```
gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/WebBench$ ./webbench -c 500 -t 60 -2 http://127.0.0.1:8080/
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.

Request:
GET / HTTP/1.1
User-Agent: WebBench 1.5
Host: 127.0.0.1
Connection: close

Runing info: 500 clients, running 60 sec.
Speed=4200713 pages/min, 3500591 bytes/sec.
Requests: 4200711 susceed, 2 failed.
```

```
Request:
HTTP/1.1
User-Agent: WebBench 1.5
Host: 127.0.0.1
Connection: close

Runing info: 500 clients, running 60 sec.
Speed=4142451 pages/min, 3452042 bytes/sec.
Requests: 4142449 susceed, 2 failed.
```

MyServer短连接http1.0版本:

```
gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/WebBench$ ./webbench -c 500 -t 60 http://127.0.0.1:8080/
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.

Request:
GET / HTTP/1.0
User-Agent: WebBench 1.5
Host: 127.0.0.1

Runing info: 500 clients, running 60 sec.
Speed=4372952 pages/min, 3644124 bytes/sec.
Requests: 4372951 susceed, 1 failed.
```

```
Runing info: 1000 clients, running 60 sec.
Speed=11520379 pages/min, 14592478 bytes/sec.
Requests: 11520379 susceed, 0 failed.
```

WebServer短连接http1.1版本:

```
gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/WebBench$ ./webbench -c 100 -t 60 -2 -k http://127.0.0.1:8080/
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.
```

100个客户端，60秒时间

MyServer长连接:

```
gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/WebBench$ ./webbench -c 100 -t 60 -2 -k http://127.0.0.1:8080/
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.

Load average: 4.88 4.10 3.77
Uptime: 02:04:45

Request:
GET / HTTP/1.1
User-Agent: WebBench 1.5
Host: 127.0.0.1
Connection: Keep-Alive

Runing info: 100 clients, running 60 sec.
Speed=11259845 pages/min, 18766404 bytes/sec.
Requests: 11259845 susceed, 0 failed.
```

MyServer短连接http1.1版本:

```
gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/WebBench$ ./webbench -c 100 -t 60 -2 http://127.0.0.1:8080/
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.

Request:
GET / HTTP/1.1
User-Agent: WebBench 1.5
Host: 127.0.0.1
Connection: close

Runing info: 100 clients, running 60 sec.
Speed=4320948 pages/min, 3600787 bytes/sec.
Requests: 4320946 susceed, 2 failed.
```

MyServer短连接http1.0版本:

```
gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/WebBench$ ./webbench -c 100 -t 60 http://127.0.0.1:8080/
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.

Request: GET / HTTP/1.0
User-Agent: WebBench 1.5
Host: 127.0.0.1

Runing info: 100 clients, running 60 sec.
Speed=4484463 pages/min, 3737051 bytes/sec.
Requests: 4484463 susceed, 0 failed.
```

WebServer

1000个客户端，60秒时间

WebServer长连接:

```
gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/WebBench$ ./webbench -c 1000 -t 60 -2 -k http://127.0.0.1:8001/hello
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.

Request: GET /hello HTTP/1.1
User-Agent: WebBench 1.5
Host: 127.0.0.1
Connection: Keep-Alive

Runing info: 1000 clients, running 60 sec.
Speed=11253784 pages/min, 14442353 bytes/sec.
Requests: 11253784 susceed, 0 failed.

Runing info: 1000 clients, running 60 sec.
Speed=10383445 pages/min, 13325411 bytes/sec.
Requests: 10383445 susceed, 0 failed.
```

WebServer短连接http1.1版本:

```
gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/WebBench$ ./webbench -c 1000 -t 60 -2 http://127.0.0.1:8001/
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.

Request: GET / HTTP/1.1
User-Agent: WebBench 1.5
Host: 127.0.0.1
Connection: close

Runing info: 1000 clients, running 60 sec.
Speed=4273746 pages/min, 16952510 bytes/sec.
Requests: 4273744 susceed, 2 failed.
```

WebServer短连接http1.1版本:

WebServer短连接http1.0版本:

WebServer短连接http1.0版本:

```
gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/WebBench$ ./webbench -c 1000 -t 60 http://127.0.0.1:8001/
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.

Request: GET / HTTP/1.0
User-Agent: WebBench 1.5
Host: 127.0.0.1

Runing info: 1000 clients, running 60 sec.
Speed=4354700 pages/min, 17273620 bytes/sec.
Requests: 4354693 susceed, 7 failed.
```

500个客户端，60秒时间

WebServer长连接:

WebServer短连接http1.1版本：

```
gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/WebBench$ ./webbench -c 500 -t 60 -g 2 http://127.0.0.1:8001/  
Webbench - Simple Web Benchmark 1.5  
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software!  
  
Request: 100个客户端, 60秒时间  
GET / HTTP/1.1  
User-Agent: WebBench 1.5  
Host: 127.0.0.1  
Connection: close  
  
Runing info: 500 clients, running 60 sec.  
Speed=4610660 pages/min, 18288936 bytes/sec.  
Requests: 4610657 susceed, 3 failed.  
  
WebServer长连接:  
  
WebServer短连接http1.1版本:
```

WebServer短连接http1.0版本:

```
gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/WebBench$ ./webbench -c 500 -t 60 http://127.0.0.1:  
8001/  
Webbench - Simple Web Benchmark 1.5  
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.  
  
Request:  
GET / HTTP/1.0  
User-Agent: WebBench 1.5  
Host: 127.0.0.1  
  
Runing info: 500 clients, running 60 sec.  
  
Speed=4475067 pages/min, 17751080 bytes/sec.  
Requests: 4475062 susceed. 5 failed.
```

100个客户端，60秒时间

WebServer长连接.

```
gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/WebBench$ ./webbench -c 100 -t 60 -2 -k http://127.0.0.1:8001/hello
Webbenchng Simple Web Benchmark 1.50 sec.
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.
Speed=11253784 pages/min, 14442353 bytes/sec.
Requests: 11253784 succeed, 0 failed.
Request:
GET /hello HTTP/1.1
User-Agent: WebBench 1.5612121711419 [/home/gmq/.config/Typora/typora-user-images/image-2
Host: 127.0.0.1
Connection: Keep-Alive .png)

gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/WebBench$ ./webbench -c 1000 -t 60 -2 -k http://127.0.0.1:8001/hello
Running info: 100 clients, running 60 sec.
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.
Speed=13290742 pages/min, 17056452 bytes/sec.
Requests: 13290742 succeed, 0 failed.
```

WebServer短连接http1.1版本：

```

gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/WebBench$ ./webbench -c 100 -t 60 -2 http://127.0.0.1:8001/
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.

Request:
GET / HTTP/1.0
User-Agent: WebBench 1.5
Host: 127.0.0.1
Connection: close

Runing info: 100 clients, running 60 sec.

Speed=4925398 pages/min, 19537394 bytes/sec.
Requests: 4925398 susceed, 0 failed.

gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/WebBench$ ./webbench -c 100 -t 60 -2 http://127.0.0.1:8001/
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.

Request:
GET / HTTP/1.0
User-Agent: WebBench 1.5
Host: 127.0.0.1
Connection: close

Runing info: 100 clients, running 60 sec.

Speed=4925398 pages/min, 19537394 bytes/sec.
Requests: 4925398 susceed, 0 failed.

```

WebServer短连接http1.0版本:

```

gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/WebBench$ ./webbench -c 100 -t 60 http://127.0.0.1:8001/
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.

Request:
GET / HTTP/1.0
User-Agent: WebBench 1.5
Host: 127.0.0.1
Connection: close

Runing info: 100 clients, running 60 sec.

Speed=4787334 pages/min, 18989752 bytes/sec.
Requests: 4787333 susceed, 1 failed.

gmq@gmq-OptiPlex-3020:~/桌面/FILE/WebServer-master/WebBench$ ./webbench -c 100 -t 60 http://127.0.0.1:8001/
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.

Request:
GET / HTTP/1.0
User-Agent: WebBench 1.5
Host: 127.0.0.1
Connection: close

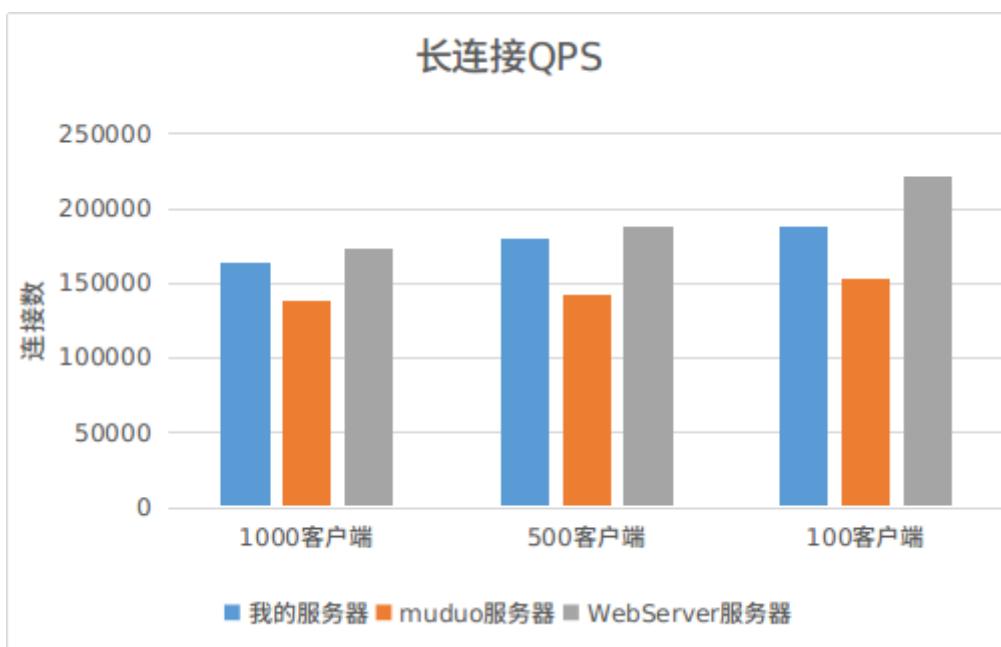
Runing info: 100 clients, running 60 sec.

Speed=4787334 pages/min, 18989752 bytes/sec.
Requests: 4787333 susceed, 1 failed.

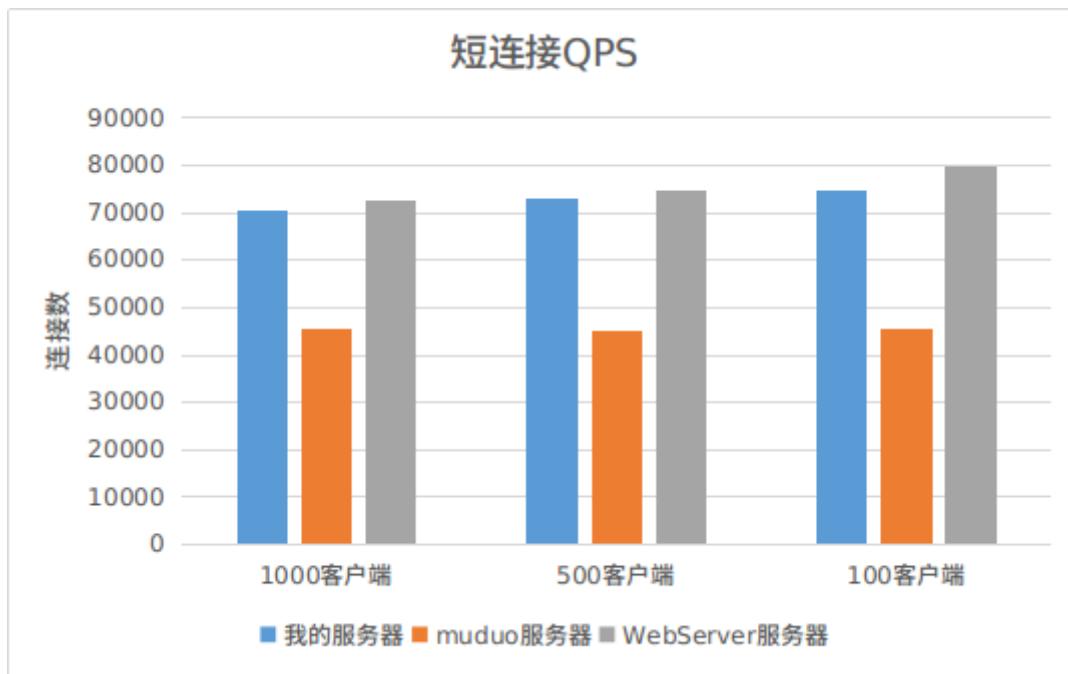
```

QPS长连接数据

	1000客户端	500客户端	100客户端
我的服务器	164119.2333	180668.65	187664.0833
muduo服务器	138701.1833	142409.3333	152630.2833
WebServer服务器	173057.4167	187563.0667	221512.3667



	1000客户端	500客户端	100客户端
我的服务器	70500.5	72882.51667	74741.05
muduo服务器	45346.73333	45044.8	45442.43333
WebServer服务器	72578.21667	74584.36667	79788.88333

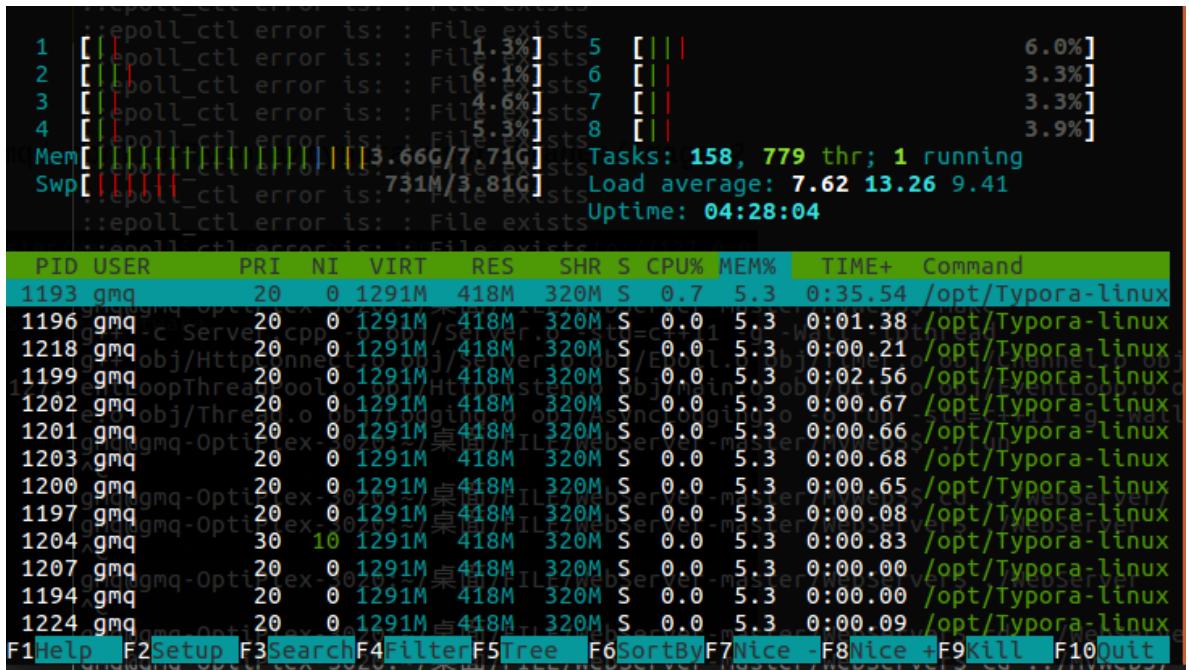


测试总结：

- WebServer的测试效果是最好的，主要是因为WebServer的封装最简单，并且我发现WebServer存在内存泄露的现象，但这个内存泄露的我并没有去深究，但肯定发生了的，所以速度也最快
- 与muduo相比较，长连接的差距主要在于我封装简单，省略了muduo中许多的回调函数调用，短连接除了这个原因以外，还主要是因为我都是使用的上升沿触发，而muduo都是水平触发，在短连接时其优势就会发挥出来。
- 短连接比长连接的连接数要少，是因为在长连接是，只需要连接一次，以后就再也不用连接了，省去了建立连接，关闭连接的过程，所以自然比短连接要快

WebServer存在内存泄露

运行前



运行结束后

