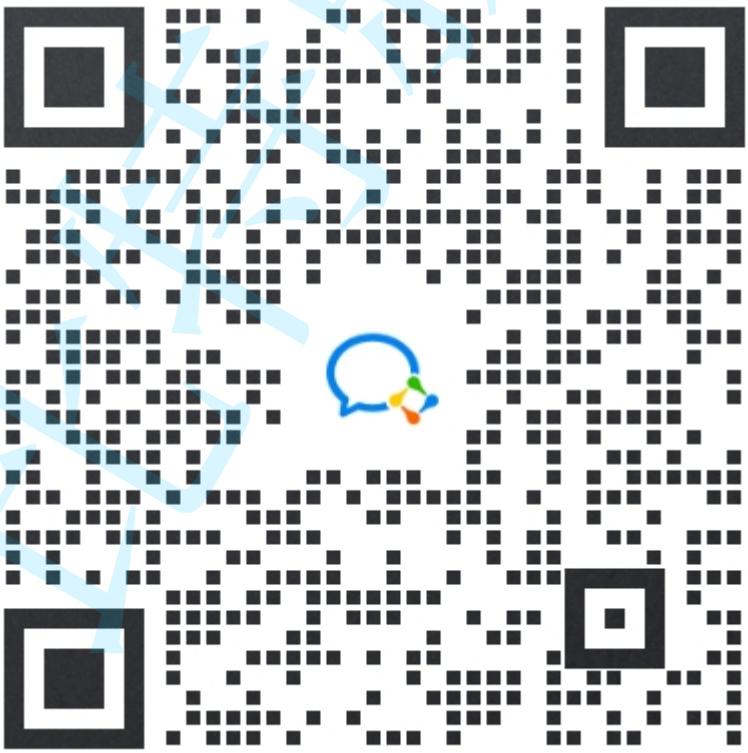
# Version1 - C++ - 仿mudou库one thread one loop式并发服务器实现

## 版权说明





## 实现目标

## 仿muduo库One Thread One Loop式主从Reactor模型实现高并发服务器：

通过咱们实现的高并发服务器组件，可以简洁快速的完成一个高性能的服务器搭建。

并且，通过组件内提供的不同应用层协议支持，也可以快速完成一个高性能应用服务器的搭建（当前为了便于项目的演示，项目中提供HTTP协议组件的支持）。

在这里，要明确的是咱们要实现的是一个高并发服务器组件，因此当前的项目中并不包含实际的业务内容。

代码仓库：https://gitee.com/qigezi/tcp-server.git

## HTTP服务器：

### 概念：

HTTP（Hyper Text Transfer Protocol），超文本传输协议是应用层协议，是一种简单的请求-响应协议（客户端根据自己的需要向服务器发送请求，服务器针对请求提供服务，完毕后通信结束）。

协议细节在课堂上已经讲过，这里不再赘述。但是需要注意的是HTTP协议是一个运行在TCP协议之上的应用层协议，这一点本质上是告诉我们，HTTP服务器其实就是个TCP服务器，只不过在应用层基于HTTP协议格式进行数据的组织和解析来明确客户端的请求并完成业务处理。

因此实现HTTP服务器简单理解，只需要以下几步即可

1. 搭建一个TCP服务器，接收客户端请求。
2. 以HTTP协议格式进行解析请求数据，明确客户端目的。
3. 明确客户端请求目的后提供对应服务。
4. 将服务结果一HTTP协议格式进行组织，发送给客户端

实现一个HTTP服务器很简单，但是实现一个高性能的服务器并不简单，这个单元中将讲解基于Reactor模式的高性能服务器实现。

当然准确来说，因为我们要实现的服务器本身并不存在业务，咱们要实现的应该算是一个高性能服务器基础库，是一个基础组件。

## Reactor模型：

### 概念

Reactor 模式，是指通过一个或多个输入同时传递给服务器进行请求处理时的事件驱动处理模式。

服务端程序处理传入多路请求，并将它们同步分派给请求对应的处理线程，Reactor 模式也叫 Dispatcher 模式。

简单理解就是使用 `I/O多路复用` 统一监听事件，收到事件后分发给处理进程或线程，是编写高性能网络服务器的必备技术之一。

分类：

单Reactor单线程：单I/O多路复用+业务处理

1. 通过IO多路复用模型进行客户端请求监控
2. 触发事件后，进行事件处理
    a. 如果是新建连接请求，则获取新建连接，并添加至多路复用模型进行事件监控。
    b. 如果是数据通信请求，则进行对应数据处理（接收数据，处理数据，发送响应）。

优点：所有操作均在同一线程中完成，思想流程较为简单，不涉及进程/线程间通信及资源争抢问题。

缺点：无法有效利用CPU多核资源，很容易达到性能瓶颈。

适用场景：适用于客户端数量较少，且处理速度较为快速的场景。（处理较慢或活跃连接较多，会导致串行处理的情况下，后处理的连接长时间无法得到响应）。

单Reactor多线程：单I/O多路复用+线程池（业务处理）

1. Reactor线程通过I/O多路复用模型进行客户端请求监控
2. 触发事件后，进行事件处理
    a. 如果是新建连接请求，则获取新建连接，并添加至多路复用模型进行事件监控。
    b. 如果是数据通信请求，则接收数据后分发给Worker线程池进行业务处理。
    c. 工作线程处理完毕后，将响应交给Reactor线程进行数据响应

优点：充分利用CPU多核资源

缺点：多线程间的数据共享访问控制较为复杂，单个Reactor 承担所有事件的监听和响应，在单线程中运行，高并发场景下容易成为性能瓶颈。

多Reactor多线程：多I/O多路复用+线程池（业务处理）

1. 在主Reactor中处理新连接请求事件，有新连接到来则分发到子Reactor中监控
2. 在子Reactor中进行客户端通信监控，有事件触发，则接收数据分发给Worker线程池
3. Worker线程池分配独立的线程进行具体的业务处理
    a. 工作线程处理完毕后，将响应交给子Reactor线程进行数据响应

优点：充分利用CPU多核资源，主从Reactor各司其职

## 目标定位：One Thread One Loop主从Reactor模型高并发服务器

咱们要实现的是主从Reactor模型服务器，也就是主Reactor线程仅仅监控监听描述符，获取新建连接，保证获取新连接的高效性，提高服务器的并发性能。

主Reactor获取到新连接后分发给子Reactor进行通信事件监控。而子Reactor线程监控各自的描述符的读写事件进行数据读写以及业务处理。

One Thread One Loop的思想就是把所有的操作都放到一个线程中进行，一个线程对应一个事件处理的循环。

当前实现中，因为并不确定组件使用者的使用意向，因此并不提供业务层工作线程池的实现，只实现主从Reactor，而Worker工作线程池，可由组件库的使用者的需要自行决定是否使用和实现。

## 功能模块划分：

基于以上的理解，我们要实现的是一个带有协议支持的Reactor模型高性能服务器，因此将整个项目的实现划分为两个大的模块：

- SERVER模块：实现Reactor模型的TCP服务器；
- 协议模块：对当前的Reactor模型服务器提供应用层协议支持。

### SERVER模块：

SERVER模块就是对所有的连接以及线程进行管理，让它们各司其职，在合适的时候做合适的事，最终完成高性能服务器组件的实现。

而具体的管理也分为三个方面：

- 监听连接管理：对监听连接进行管理。
- 通信连接管理：对通信连接进行管理。
- 超时连接管理：对超时连接进行管理。

基于以上的管理思想，将这个模块进行细致的划分又可以划分为以下多个子模块：

#### Buffer模块：

Buffer模块是一个缓冲区模块，用于实现通信中用户态的接收缓冲区和发送缓冲区功能

#### Socket模块：

Socket模块是对套接字操作封装的一个模块，主要实现的socket的各项操作。

#### Channel模块：

Channel模块是对一个描述符需要进行的IO事件管理的模块，实现对描述符可读，可写，错误...事件的管理操作，以及Poller模块对描述符进行IO事件监控就绪后，根据不同的事件，回调不同的处理函数功能。

#### Connection模块

Connection模块是对Buffer模块，Socket模块，Channel模块的一个整体封装，实现了对一个通信套接字的整体的管理，每一个进行数据通信的套接字（也就是accept获取到的新连接）都会使用Connection进行管理。

- Connection模块内部包含有三个由组件使用者传入的回调函数：连接建立完成回调，事件回调，新数据回调，关闭回调。

- Connection模块内部包含有两个组件使用者提供的接口：数据发送接口，连接关闭接口
- Connection模块内部包含有两个用户态缓冲区：用户态接收缓冲区，用户态发送缓冲区
- Connection模块内部包含有一个Socket对象：完成描述符面向系统的IO操作
- Connection模块内部包含有一个Channel对象：完成描述符IO事件就绪的处理

具体处理流程如下：

1. 实现向Channel提供可读，可写，错误等不同事件的IO事件回调函数，然后将Channel和对应的描述符添加到Poller事件监控中。
2. 当描述符在Poller模块中就绪了IO可读事件，则调用描述符对应Channel中保存的读事件处理函数，进行数据读取，将socket接收缓冲区全部读取到Connection管理的用户态接收缓冲区中。然后调用由组件使用者传入的新数据到来回调函数进行处理。
3. 组件使用者进行数据的业务处理完毕后，通过Connection向使用者提供的数据发送接口，将数据写入Connection的发送缓冲区中。
4. 启动描述符在Poll模块中的IO写事件监控，就绪后，调用Channel中保存的写事件处理函数，将发送缓冲区中的数据通过Socket进行面向系统的实际数据发送。

**Acceptor模块：**

Acceptor模块是对Socket模块，Channel模块的一个整体封装，实现了对一个监听套接字的整体的管理。

- Acceptor模块内部包含有一个Socket对象：实现监听套接字的操作
- Acceptor模块内部包含有一个Channel对象：实现监听套接字IO事件就绪的处理

具体处理流程如下：

1. 实现向Channel提供可读事件的IO事件处理回调函数，函数的功能其实也就是获取新连接
2. 为新连接构建一个Connection对象出来。

**TimerQueue模块：**

TimerQueue模块是实现固定时间定时任务的模块，可以理解就是要给定时任务管理器，向定时任务管理器中添加一个任务，任务将在固定时间后被执行，同时也可以通过刷新定时任务来延迟任务的执行。

这个模块主要是对Connection对象的生命周期管理，对非活跃连接进行超时后的释放功能。

TimerQueue模块内部包含有一个timerfd：linux系统提供的定时器。

TimerQueue模块内部包含有一个Channel对象：实现对timerfd的IO时间就绪回调处理

**Poller模块：**

Poller模块是对epoll进行封装的一个模块，主要实现epoll的IO事件添加，修改，移除，获取活跃连接功能。

**EventLoop模块：**

EventLoop模块可以理解就是我们上边所说的Reactor模块，它是对Poller模块，TimerQueue模块，Socket模块的一个整体封装，进行所有描述符的事件监控。

EventLoop模块必然是一个对象对应一个线程的模块，线程内部的目的就是运行EventLoop的启动函数。

EventLoop模块为了保证整个服务器的线程安全问题，因此要求使用者对于Connection的所有操作一定要在其对应的EventLoop线程内完成，不能在其他线程中进行（比如组件使用者使用Connection发送数据，以及关闭连接这种操作）。

EventLoop模块保证自己内部所监控的所有描述符，都要是活跃连接，非活跃连接就要及时释放避免资源浪费。

- EventLoop模块内部包含有一个eventfd：eventfd其实就是linux内核提供的一个事件fd，专门用于事件通知。
- EventLoop模块内部包含有一个Poller对象：用于进行描述符的IO事件监控。
- EventLoop模块内部包含有一个TimerQueue对象：用于进行定时任务的管理。
- EventLoop模块内部包含有一个PendingTask队列：组件使用者将对Connection进行的所有操作，都加入到任务队列中，由EventLoop模块进行管理，并在EventLoop对应的线程中进行执行。
- 每一个Connection对象都会绑定到一个EventLoop上，这样能保证对这个连接的所有操作都是在一个线程中完成的。

具体操作流程：

1. 通过Poller模块对当前模块管理内的所有描述符进行IO事件监控，有描述符事件就绪后，通过描述符对应的Channel进行事件处理。
2. 所有就绪的描述符IO事件处理完毕后，对任务队列中的所有操作顺序进行执行。
3. 由于epoll的事件监控，有可能会因为没有事件到来而持续阻塞，导致任务队列中的任务不能及时得到执行，因此创建了eventfd，添加到Poller的事件监控中，用于实现每次向任务队列添加任务的时候，通过向eventfd写入数据来唤醒epoll的阻塞。

**TcpServer模块：**

这个模块是一个整体Tcp服务器模块的封装，内部封装了Acceptor模块，EventLoopThreadPool模块。

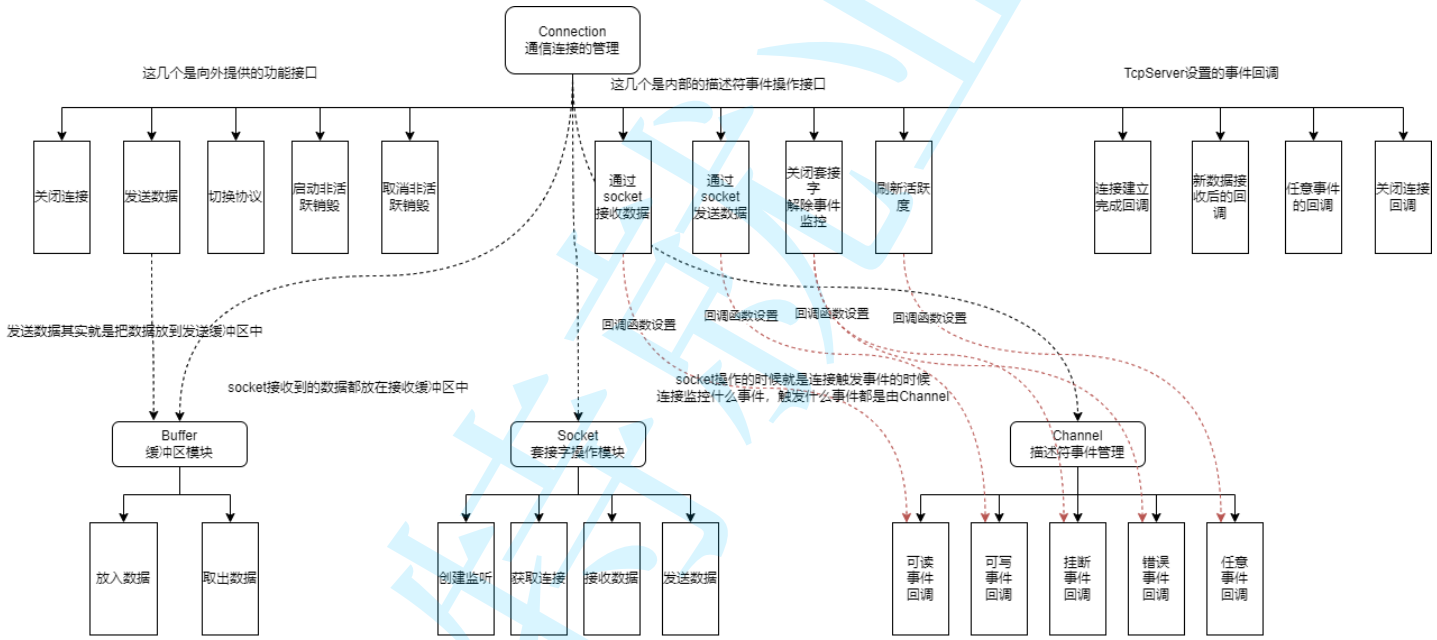
- TcpServer中包含有一个EventLoop对象：以备在超轻量使用场景中不需要EventLoop线程池，只需要在主线程中完成所有操作的情况。
- TcpServer模块内部包含有一个EventLoopThreadPool对象：其实就是EventLoop线程池，也就是子Reactor线程池
- TcpServer模块内部包含有一个Acceptor对象：一个TcpServer服务器，必然对应有一个监听套接字，能够完成获取客户端新连接，并处理的任务。

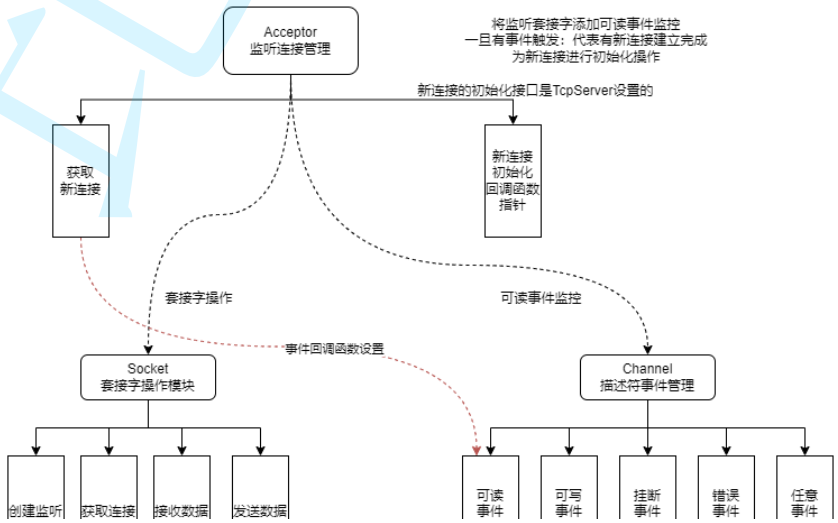
- TcpServer模块内部包含有一个std::shared_ptr<Connection>的hash表：保存了所有的新建连接对应的Connection，注意，所有的Connection使用shared_ptr进行管理，这样能够保证在hash表中删除了Connection信息后，在shared_ptr计数器为0的情况下完成对Connection资源的释放操作。

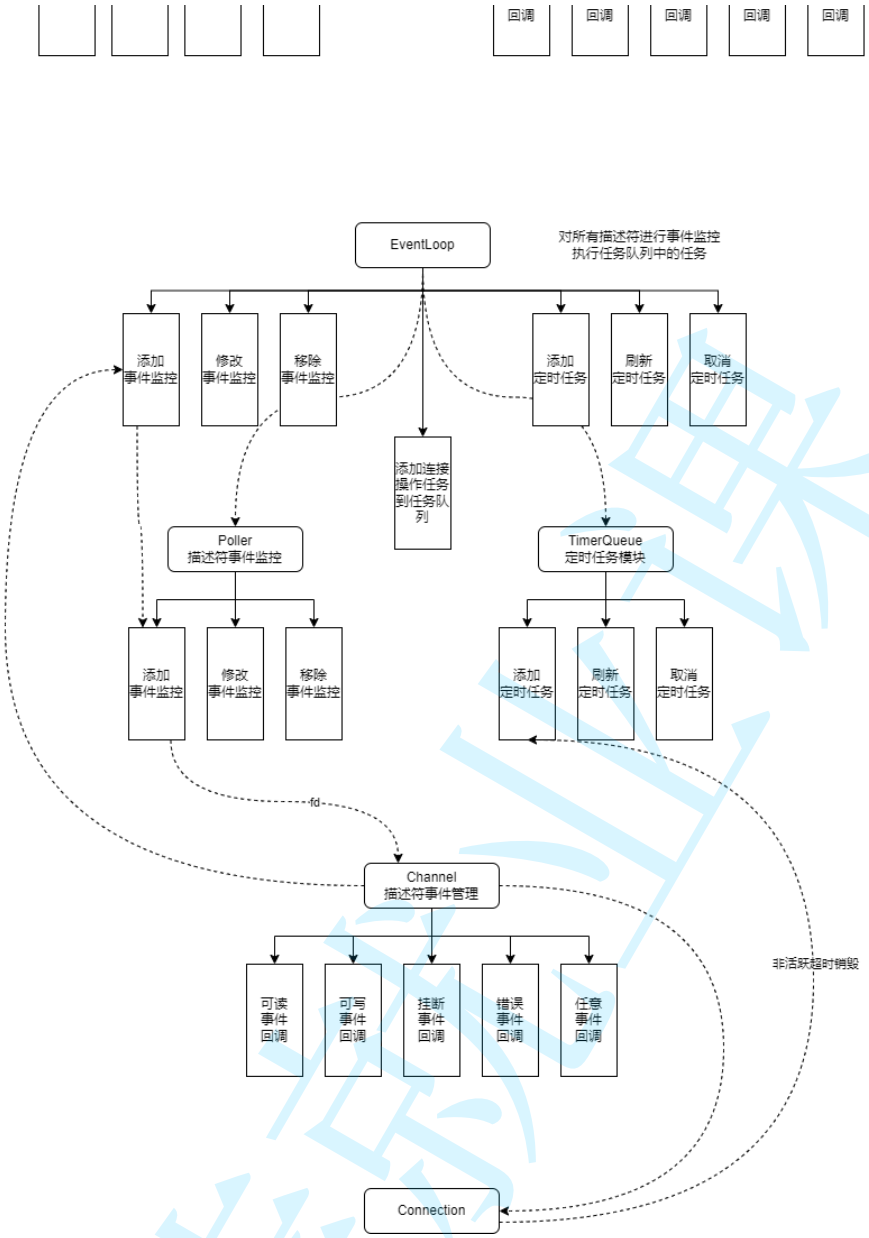
具体操作流程如下：

1. 在实例化TcpServer对象过程中，完成BaseLoop的设置，Acceptor对象的实例化，以及EventLoop线程池的实例化，以及std::shared_ptr<Connection>的hash表的实例化。
2. 为Acceptor对象设置回调函数：获取到新连接后，为新连接构建Connection对象，设置Connection的各项回调，并使用shared_ptr进行管理，并添加到hash表中进行管理，并为Connection选择一个EventLoop线程，为Connection添加一个定时销毁任务，为Connection添加事件监控，
3. 启动BaseLoop。

**模块关系图：**






## HTTP协议模块：

HTTP协议模块用于对高并发服务器模块进行协议支持，基于提供的协议支持能够更方便的完成指定协议服务器的搭建。

而HTTP协议支持模块的实现，可以细分为以下几个模块。

### Util模块：

这个模块是一个工具模块，主要提供HTTP协议模块所用到的一些工具函数，比如url编解码，文件读写....等。

### HttpRequest模块：

这个模块是HTTP请求数据模块，用于保存HTTP请求数据被解析后的各项请求元素信息。

### HttpResponse模块：

这个模块是HTTP响应数据模块，用于业务处理后设置并保存HTTP响应数据的的各项元素信息，最终会被按照HTTP协议响应格式组织成为响应信息发送给客户端。

HttpContext模块：

这个模块是一个HTTP请求接收的上下文模块，主要是为了防止在一次接收的数据中，不是一个完整的HTTP请求，则解析过程并未完成，无法进行完整的请求处理，需要在下次接收到新数据后继续根据上下文进行解析，最终得到一个HttpRequest请求信息对象，因此在请求数据的接收以及解析部分需要一个上下文来进行控制接收和处理节奏。

HttpServer模块：

这个模块是最终给组件使用者提供的HTTP服务器模块了，用于以简单的接口实现HTTP服务器的搭建。

HttpServer模块内部包含有一个TcpServer对象：TcpServer对象实现服务器的搭建

HttpServer模块内部包含有两个提供给TcpServer对象的接口：连接建立成功设置上下文接口，数据处理接口。

HttpServer模块内部包含有一个hash-map表存储请求与处理函数的映射表：组件使用者向HttpServer设置哪些请求应该使用哪些函数进行处理，等TcpServer收到对应的请求就会使用对应的函数进行处理。

## 代码实现

### 前置知识技术点功能用例：

C++11中的bind：

```
bind (Fn&& fn, Args&&... args);
```

官方文档对于bind接口的概述解释：Bind function arguments

我们可以将bind接口看作是一个通用的函数适配器，它接受一个函数对象，以及函数的各项参数，然后返回一个新的函数对象，但是这个函数对象的参数已经被绑定为设置的参数。运行的时候相当于总是调用传入固定参数的原函数。

但是如果进行绑定的时候，给与的参数为 `std::placeholders::_1, _2...` 则相当于为新适配生成的函数对象的调用预留一个参数进行传递。

```
#include <iostream>
#include <functional>
#include <unistd.h>
class Test {
    public:
        Test() { std::cout << "构造" << std::endl; }
```

```
        ~Test() { std::cout << "析构" << std::endl; }
};

void del(const Test *t, int num) {
    std::cout << num << std::endl;
    delete t;
}
int main()
{
    Test *t = new Test;
    /*bind作用也可以简单理解为给一个函数绑定好参数，然后返回一个参数已经设定好或者预留好的函数，
    可以在合适的时候进行调用*/
    /*
    比如，del函数，要求有两个参数，一个Test*， 一个int，
    而这里，想要基于del函数，适配生成一个新的函数，这个函数固定第1个参数传递t变量，
    第二个参数预留出来，在调用的时候进行设置
    */
    std::function<void(int)> cb = std::bind(del, t, std::placeholders::_1);
    cb(10);
    while(1) sleep(1);
    return 0;
}
```

```
[dev@localhost example]$ g++ -std=c++11 bind.cpp -o bind
[dev@localhost example]$ ./bind
构造
10
析构

^C
```

基于bind的作用，当我们在设计一些线程池，或者任务池的时候，就可以将将任务池中的任务设置为函数类型，函数的参数由添加任务者直接使用bind进行适配绑定设置，而任务池中的任务被处理，只需要取出一个个的函数进行执行即可。

这样做有个好处就是，这种任务池在设计的时候，不用考虑都有哪些任务处理方式了，处理函数该如何设计，有多少个什么样的参数，这些都不用考虑了，降低了代码之间的耦合度。

```
#include <iostream>
#include <string>
#include <vector>
#include <functional>
```

```cpp

void print(const std::string &str) {
    std::cout << str << std::endl;
}

int main()
{
    using Functor = std::function<void()>;
    std::vector<Functor> task_pool;

    task_pool.push_back(std::bind(print, "比特"));
    task_pool.push_back(std::bind(print, "你好"));
    task_pool.push_back(std::bind(print, "我是"));
    task_pool.push_back(std::bind(print, "好学生"));

    for (auto &functor : task_pool) {
        functor();
    }
    return 0;
}
```

```
[dev@localhost example]$ g++ -std=c++11 bind.cpp -o bind
[dev@localhost example]$ ./bind
比特
你好
我是
好学生
[dev@localhost example]$
```

**简单的秒级定时任务实现：**

在当前的高并发服务器中，我们不得不考虑一个问题，那就是连接的超时关闭问题。我们需要避免一个连接长时间不通信，但是也不关闭，空耗资源的情况。

这时候我们就需要一个定时任务，定时的将超时过期的连接进行释放。

**Linux提供给我们的定时器：**

```cpp
#include <sys/timerfd.h>

int timerfd_create(int clockid, int flags);
    clockid:  CLOCK_REALTIME-系统实时时间，如果修改了系统时间就会出问题;
              CLOCK_MONOTONIC-从开机到现在的时间是一种相对时间;
    flags: 0-默认阻塞属性
```

```
int timerfd_settime(int fd, int flags, struct itimerspec *new, struct
itimerspec *old);
    fd: timerfd_create返回的文件描述符
    flags: 0-相对时间， 1-绝对时间；默认设置为0即可.
    new:  用于设置定时器的新超时时间
    old:  用于接收原来的超时时间
    struct timespec {
       time_t tv_sec;                /* Seconds */
       long   tv_nsec;               /* Nanoseconds */
    };
    struct itimerspec {
       struct timespec it_interval;  /* 第一次之后的超时间隔时间 */
       struct timespec it_value;     /* 第一次超时时间 */
    };
定时器会在每次超时时，自动给fd中写入8字节的数据，表示在上一次读取数据到当前读取数据期间超
时了多少次。
```

示例：

```
#include <iostream>
#include <cstdio>
#include <string>
#include <ctime>
#include <cstdlib>
#include <unistd.h>
#include <sys/timerfd.h>
#include <sys/select.h>

int main()
{
    /*创建一个定时器 */
    int timerfd = timerfd_create(CLOCK_MONOTONIC, 0);

    struct itimerspec itm;
    itm.it_value.tv_sec = 3;//设置第一次超时的时间
    itm.it_value.tv_nsec = 0;
    itm.it_interval.tv_sec = 3;//第一次超时后，每隔多长时间超时
    itm.it_interval.tv_nsec = 0;
    timerfd_settime(timerfd, 0, &itm, NULL);//启动定时器
    /*这个定时器描述符将每隔三秒都会触发一次可读事件*/
    time_t start = time(NULL);
    while(1) {
        uint64_t tmp;
```

```
25          /*需要注意的是定时器超时后,则描述符触发可读事件，必须读取8字节的数据，保存的是自上次启动定时器或read后的超时次数*/
26          int ret = read(timerfd, &tmp, sizeof(tmp));
27          if (ret < 0) {
28              return -1;
29          }
30          std::cout << tmp << " " << time(NULL) - start << std::endl;
31      }
32      close(timerfd);
33
34      return 0;
35  }
```

```
1  [dev@localhost example]$ g++ -std=c++11 timerfd.cpp -o timerfd
2  [dev@localhost example]$ ./timerfd
3  1 3
4  1 6
5  1 9
6  1 12
7  ^C
8  [dev@localhost example]$
```

上边例子，是一个定时器的使用示例，是每隔3s钟触发一次定时器超时，否则就会阻塞在read读取数据这里。

基于这个例子，则我们可以实现每隔3s，检测一下哪些连接超时了，然后将超时的连接释放掉。

**时间轮思想：**

上述的例子，存在一个很大的问题，每次超时都要将所有的连接遍历一遍，如果有上万个连接，效率无疑是较为低下的。

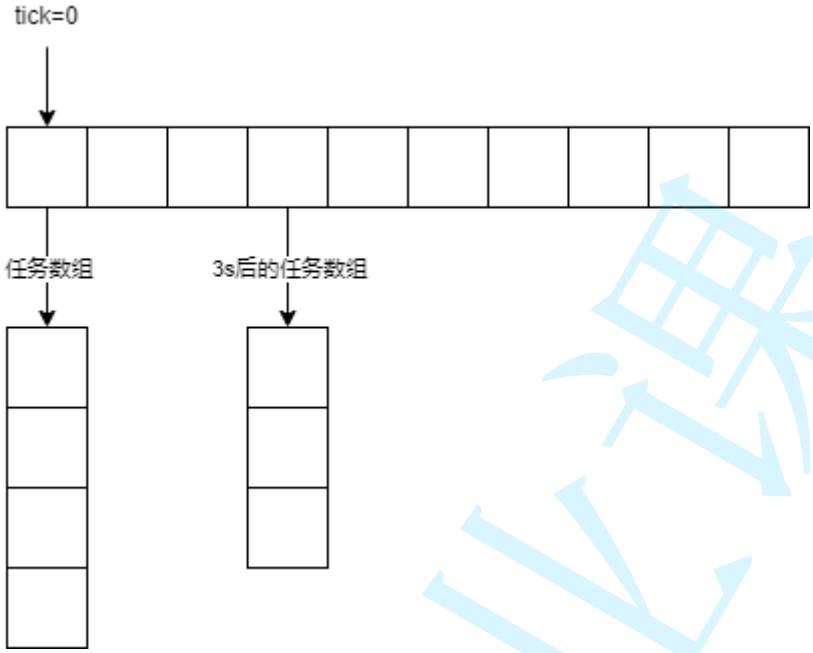
这时候大家就会想到，我们可以针对所有的连接，根据每个连接最近一次通信的系统时间建立一个小根堆，这样只需要每次针对堆顶部分的连接逐个释放，直到没有超时的连接为止，这样也可以大大提高处理的效率。

上述方法可以实现定时任务，但是这里给大家介绍另一种方案：时间轮

时间轮的思想来源于钟表，如果我们定了一个3点钟的闹铃，则当时针走到3的时候，就代表时间到了。

同样的道理，如果我们定义了一个数组，并且有一个指针，指向数组起始位置，这个指针每秒钟向后走动一步，走到哪里，则代表哪里的任务该被执行了，那么如果我们想要定一个3s后的任务，则只需要将任务添加到tick+3位置，则每秒中走一步，三秒钟后tick走到对应位置，这时候执行对应位置的任务即可。

但是，同一时间可能会有大批量的定时任务，因此我们可以给数组对应位置下拉一个数组，这样就可以在同一个时刻上添加多个定时任务了。




当然，上述操作也有一些缺陷，比如我们如果要定义一个60s后的任务，则需要将数组的元素个数设置为60才可以，如果设置一小时后的定时任务，则需要定义3600个元素的数组，这样无疑是比较麻烦的。

因此，可以采用多层级的时间轮，有秒针轮，分针轮，时针轮， 60<time<3600则time/60就是分针轮对应存储的位置，当tick/3600等于对应位置的时候，将其位置的任务向分针，秒针轮进行移动。

因为当前我们的应用中，倒是不用设计的这么麻烦，因为我们的定时任务通常设置的30s以内，所以简单的单层时间轮就够用了。

但是，我们也得考虑一个问题，当前的设计是时间到了，则主动去执行定时任务，释放连接，那能不能在时间到了后，自动执行定时任务呢，这时候我们就想到一个操作-类的析构函数。

一个类的析构函数，在对象被释放时会自动被执行，那么我们如果将一个定时任务作为一个类的析构函数内的操作，则这个定时任务在对象被释放的时候就会执行。

但是仅仅为了这个目的，而设计一个额外的任务类，好像有些不划算，但是，这里我们又要考虑另一个问题，那就是假如有一个连接建立成功了，我们给这个连接设置了一个30s后的定时销毁任务，但是在第10s的时候，这个连接进行了一次通信，那么我们应该时在第30s的时候关闭，还是第40s的时候关闭呢？无疑应该是第40s的时候。也就是说，这时候，我们需要让这个第30s的任务失效，但是我们该如何实现这个操作呢？

这里，我们就用到了智能指针shared_ptr，shared_ptr有个计数器，当计数为0的时候，才会真正释放一个对象，那么如果连接在第10s进行了一次通信，则我们继续向定时任务中，添加一个30s后（也就是第40s）的任务类对象的shared_ptr，则这时候两个任务shared_ptr计数为2，则第30s的定时任务被释放的时候，计数-1，变为1，并不为0，则并不会执行实际的析构函数，那么就相当于这个第30s的任务失效了，只有在第40s的时候，这个任务才会被真正释放。

上述过程就是时间轮定时任务的思想了，当然这里为了更加简便的实现，进行了一些小小的调整实现。

```cpp
#include <iostream>
#include <list>
#include <vector>
#include <unordered_set>
#include <functional>
#include <memory>
#include <unordered_map>
#include <cassert>
#include <unistd.h>
/*定时任务类*/
using OnTimerCallback = std::function<void()>;
using ReleaseCallback = std::function<void()>;
class Timer {
    private:
        int _timeout;       /*当前定时器任务的延迟时间*/
        bool _canceled;     /*false-任务正常执行;  true-任务被取消*/
        uint64_t _timer_id; /*定时器ID*/
        OnTimerCallback _timer_callback;
        /*实际释任务类对象实际析构时要执行的操作，主要用于移除TimerQueue中保存的weak_ptr<Timer>信息*/
        ReleaseCallback _release_callback;
    public:
        Timer(uint64_t timer_id, int timeout): _timer_id(timer_id), _timeout(timeout), _canceled(false){}
        ~Timer(){
            /*先从timerqueue保存的weak_ptr<timer>数组中，将weak_ptr移除，避免下边的定时任务回调中有对定时器任务的操作*/
            /*比如在_timer_callback中调用 canceled，取消的时候是可以找到定时器任务的，因为判断定时器任务是否存在的依据是timerqueue中是否有信息*/
            /*而能走到析构，证明shared_ptr计数为0了， 这时候timequeue中的weak_ptr获取到的shared_ptr就是空了，需要谨慎操作*/
            if (_release_callback) _release_callback();
            if (_timer_callback && !_canceled) _timer_callback(); /*定时器被取消，则析构的时候不再执行任务*/
        }
        uint64_t id();
        int delay_time() { return _timeout; }/*获取定时器任务的初始延迟时间*/
        void canceled() { _canceled = true; }/*取消定时器任务*/
        void set_on_time_callback(const OnTimerCallback &cb) {_timer_callback = cb; }
        void set_release_callback(const ReleaseCallback &cb) {_release_callback = cb; }
};
```

```
36
37 #define MAX_TIMEOUT 60
38 class TimerQueue {
39     private:
40         using WeakTimer = std::weak_ptr<Timer>;
41         using PtrTimer = std::shared_ptr<Timer>;
42         /*时间轮每个节点都是一个vector，这样同一个时刻可以添加多个定时器任务*/
43         using Bucket = std::vector<PtrTimer>;
44         /*为了提高通过下标随机访问的效率，这里使用了数组实现单层时间轮*/
45         using BucketList = std::vector<Bucket>;
46         /*滴答秒针，每次执行一次定时器时间到，就会向后走一步，走到哪里就清空BucketList的
   哪个数组*/
47         int _tick;
48         /*时间轮的容量，这个容量其实就是最大的定时时长*/
49         /*如果想要设计更大时间的定时器，可以再设计一个具有60个节点的分针定时器，它是60s滴
   答一次，定时任务就是到期了将实际任务加入到秒针轮中*/
50         int _capacity;
51         /*实现单层时间轮的数组，每秒钟tick向后走一步，走到哪里，就将哪里的vector进行
   clear，则其内的定时器PtrTimer将全部被释放*/
52         /*若PtrTimer的shared_ptr计数器位0，将将会真正被释放，执行Timer的析构函数，完成
   定时任务的执行*/
53         BucketList _conns;
54         /*保存所有定时任务对象的weak_ptr，这样才能在不影响shared_ptr计数器的同时，获取
   shared_ptr*/
55         std::unordered_map<uint64_t, WeakTimer> _timers;
56     public:
57         TimerQueue():
58             _tick(0),
59             _capacity(MAX_TIMEOUT),
60             _conns(_capacity){ }
61         /*判断定时任务是否存在， 注意：这个接口是一个非线程安全接口，只能内部使用，不能外
   部其他线程调用，目前只在enable_inactive_release_in_loop中使用了*/
62         bool has_timer(uint64_t id) {
63             auto it = _timers.find(id);
64             if (it != _timers.end()) { return true; }
65             return false;
66         }
67         /*添加一个定时任务*/
68         void timer_add(const OnTimerCallback &cb, int delay, uint64_t id) {
69             /*60s以上和0s以下的定时任务不支持....*/
70             if (delay > _capacity || delay <= 0)  return;
71             /*new一个定时任务对象出来*/
72             PtrTimer timer(new Timer(id, delay));
73             /*设置定时任务对象要执行的定时任务--会在对象被析构时执行*/
74             timer->set_on_time_callback(cb);
75             /*因为当前类成员_timers中保存了一份定时任务对象的weak_ptr，因此希望在析构的
   同时进行移除*/
```

```
76              timer->set_release_callback(std::bind(&TimerQueue::remove_weaktimer_from_timerqueue, this, id));
77              /*根据定时任务对象的shared_ptr获取其weak_ptr保存起来，以便于二次刷新任务，
78              也就是任务需要延迟执行的时候，重新继续添加一个定时任务进去，可以使用相同的shared_ptr计数*/
79              _timers[id] = WeakTimer(timer);
80              /*tick是当前的指针位置*/
81              _conns[(_tick + delay) % _capacity].push_back(timer);
82          }
83          /*根据id刷新定时任务*/
84          void timer_refresh(uint64_t id) {
85              auto it = _timers.find(id);
86              /*不可能存在这种情况呀，添加的定时任务找不着，其实也不能因为一个定时任务找不到而让程序退出，return其实就行，这里主要是调试用断言*/
87              assert (it != _timers.end());
88              int delay = it->second.lock()->delay_time();
89              _conns[(_tick + delay) % _capacity].push_back(PtrTimer(it->second));
90          }
91          void timer_cancel(uint64_t id) {
92              auto it = _timers.find(id);
93              assert (it != _timers.end());
94              PtrTimer pt = it->second.lock();
95              if (pt) pt->canceled();
96          }
97          /*设置给Timer，最终定时任务执行完毕后从timequeue移除timer信息的回调函数*/
98          void remove_weaktimer_from_timerqueue(uint64_t id) {
99              auto it = _timers.find(id);
100             if (it != _timers.end()) {
101                 _timers.erase(it);
102             }
103         }
104         void run_ontime_task() {
105             /*每滴答一次被执行_tick++就会向后走一步，走到哪里，释放哪里的定时器，也就是执行哪里的定时任务*/
106             _tick = (_tick + 1) % _capacity;
107             _conns[_tick].clear();
108         }
109 };
```

```
1
2 class TimerTest{
3     private:
4         int _data;
```

```
    public:
        TimerTest(int data):_data(data) { std::cout << "test 构造!\n"; }
        ~TimerTest() {std::cout << "test 析构!\n"; }
};

void del(TimerTest *t) {
    delete t;
}
int main()
{
    /*创建一个固定5s的定时任务队列， 每个添加的任务将在被添加5s后执行*/
    TimerQueue tq;

    /*new 了一个对象*/
    TimerTest *t = new TimerTest(10);
    /*随便设置了一个定时器ID*/
    int id = 3;
    /*std::bind(del, t) 构建适配了一个释放函数，作为定时任务执行函数*/
    /*这里其实就是添加了一个5秒后执行的定时任务，任务是销毁t指针指向的空间*/
    tq.timer_add(std::bind(del, t), 5, id);
    /*按理说这个任务5s后就会被执行，析构，但是因为循环内总是在刷新任务，也就是二次添加任务，
    因此，它的计数总是>0，不会被释放，之后最后一个任务对象shared_ptr被释放才会真正析构*/
    for (int i = 0; i < 5; i++) {
        sleep(1);
        tq.timer_refresh(id);
        std::cout << "刷新了一下3号定时任务!\n";
        tq.run_ontime_task();//每秒调用一次，模拟定时器
    }
    std::cout << "刷新定时任务停止，5s后释放任务将被执行\n";
    while (1){
        std::cout << "--------------------\n";
        sleep(1);
        tq.run_ontime_task();//每秒调用一次，模拟定时器
        if (tq.has_timer(id) == false) {
            std::cout << "定时任务已经被执行完毕! \n";
            break;
        }
    }

    return 0;
}
```

```
[dev@localhost example]$ g++ -std=c++11 timer_test.cpp -o timer_test
[dev@localhost example]$ ./timer_test
```

```
test 构造!
刷新了一下3号定时任务!
刷新了一下3号定时任务!
刷新了一下3号定时任务!
刷新了一下3号定时任务!
刷新了一下3号定时任务!
刷新定时任务停止，5s后释放任务将被执行
--------------------
--------------------
--------------------
--------------------
test 析构!
[dev@localhost example]$
```

**正则库的简单使用：**

正则表达式(regular expression)描述了一种字符串匹配的模式（pattern），可以用来检查一个串是否含有某种子串、将匹配的子串替换或者从某个串中取出符合某个条件的子串等。

正则表达式的使用，可以使得HTTP请求的解析更加简单（这里指的时程序员的工作变得的简单，这并不代表处理效率会变高，实际上效率上是低于直接的字符串处理的），使我们实现的HTTP组件库使用起来更加灵活。

```

#include <iostream>
#include <string>
#include <regex>

void req_line() {
    std::cout << "-----------------first line start-----------------\n";
    //std::string str = "GET /bitejiuyeke HTTP/1.1\r\n";
    //std::string str = "GET /bitejiuyeke HTTP/1.1\n";
    std::string str = "GET /bitejiuyeke?a=b&c=d HTTP/1.1\r\n";
    std::regex re("(GET|HEAD|POST|PUT|DELETE) (([^?]+)(?:\\?(.*?))?) (HTTP/1\\.[01])(?:\r\n|\n)");
    std::smatch matches;
    std::regex_match(str, matches, re);
    /*正则匹配获取完毕之后matches中的存储情况*/
    /*  matches[0]    整体首行    GET /bitejiuyeke?a=b&c=d HTTP/1.1
        matches[1]    请求方法    GET
        matches[2]    整体URL     /bitejiuyeke?a=b&c=d
        matches[3]    ?之前       /bitejiuyeke
        matches[4]    查询字符串  a=b&c=d
        matches[5]    协议版本    HTTP/1.1    */
    int i = 0;
    for (const auto &it:matches) {
```

```
        std::cout << i++ << ": ";
        std::cout << it << std::endl;
    }
    if (matches[4].length() > 0) {
        std::cout << "have param!\n";
    }else {
        std::cout << "have not param!\n";
    }
    std::cout << "------------------first line start-----------------\n";
    return;
}
void method_match(const std::string str) {
    std::cout << "------------------method start-----------------\n";
    std::regex re("(GET|HEAD|POST|PUT|DELETE) .*");
    /*  () 表示捕捉符合括号内格式的数据
     *  GET|HEAD|POST...    |表示或，也就是匹配这几个字符串中的任意一个
     *  .* 中.表示匹配除换行外的任意单字符，*表示匹配前边的字符任意次；  合起来在这里就是表示空格后匹配任意字符
     *  最终合并起来表示匹配以GET或者POST或者PUT...几个字符串开始，然后后边有个空格的字符串，并在匹配成功后捕捉前边的请求方法字符串
     */
    std::smatch matches;
    std::regex_match(str, matches, re);
    std::cout << matches[0] << std::endl;
    std::cout << matches[1] << std::endl;
    std::cout << "------------------method over-----------------\n";
}
void path_match(const std::string str) {
    //std::regex re("(([^?]+)(?:\\?(.*?))?)");
    std::cout << "------------------path start-----------------\n";
    std::regex re("([^?]+).*");
    /*
     * 最外层的() 表示捕捉提取括号内指定格式的内容
     * ([^?]+)  [^xyz] 负值匹配集合，指匹配非^之后的字符，  比如[^abc] 则plain就匹配到plin字符
     *          +匹配前面的子表达式一次或多次
     * 合并合并起来就是匹配非?字符一次或多次
     */
    std::smatch matches;
    std::regex_match(str, matches, re);
    std::cout << matches[0] << std::endl;
    std::cout << matches[1] << std::endl;
    std::cout << "------------------path over-----------------\n";
}
void query_match(const std::string str) {
    std::cout << "------------------query start-----------------\n";
    std::regex re("(?:\\?(.*?))? .*");
```

```
    /*
     * (?:\\?(.*?))?      最后的?表示匹配前边的表达式0次或1次，因为有的请求可能没有查询字符串
     * (?:\\?(.*?))       (?:pattern)表示匹配pattern但是不获取匹配结果
     * \\?(.*?) \\?表示原始的?字符，这里表示以?字符作为起始
     *          .表示\n之外任意单字符，
                *表示匹配前边的字符0次或多次，
                ?跟在*或+之后表示懒惰模式，  也就是说以?开始的字符串就只匹配这一次就行，后边还有以?开始的同格式字符串也不不会匹配
                () 表示捕捉获取符合内部格式的数据
     *  合并起来表示的就是，匹配以?开始的字符串，但是字符串整体不要，
     *  只捕捉获取?之后的字符串,且只匹配一次，就算后边还有以?开始的同格式字符串也不不会匹配
     */
    std::smatch matches;
    std::regex_match(str, matches, re);
    std::cout << matches[0] << std::endl;
    std::cout << matches[1] << std::endl;

    std::cout << "------------------query over------------------\n";
}
void version_mathch(const std::string str) {
    std::cout << "------------------version start------------------\n";
    std::regex re("(HTTP/1\\.[01])(?:\r\n|\n)");
    /*
     *  (HTTP/1\\.[01])  外层的括号表示捕捉字符串
     *  HTTP/1  表示以HTTP/1开始的字符串
     *  \\.     表示匹配 . 原始字符
     *  [01]    表示匹配0字符或者1字符
     *  (?:\r\n|\n)  表示匹配一个\r\n或者\n字符，但是并不捕捉这个内容
     *  合并起来就是匹配以HTTP/1.开始，后边跟了一个0或1的字符，且最终以\n或者\r\n作为结尾的字符串
     */
    std::smatch matches;
    std::regex_match(str, matches, re);
    std::cout << matches[0] << std::endl;
    std::cout << matches[1] << std::endl;

    std::cout << "------------------version over------------------\n";

}
int main()
{
    req_line();
    method_match("GET /s");
    path_match("/search?name=bitejiuyeke ");
    query_match("?name=xiaoming&age=19 HTTP/1.1");
```

```
    version_mathch("HTTP/1.1\r\n");
    return 0;
}

```

```
[dev@localhost example]$ ./regex
------------------first line start-----------------
0: GET /bitejiuyeke?a=b&c=d HTTP/1.1

1: GET
2: /bitejiuyeke?a=b&c=d
3: /bitejiuyeke
4: a=b&c=d
5: HTTP/1.1
have param!
------------------first line start-----------------
------------------method start-----------------
GET /s
GET
------------------method over-----------------
------------------path start-----------------
/search?name=bitejiuyeke
/search
------------------path over-----------------
------------------query start-----------------
?name=xiaoming&age=19 HTTP/1.1
name=xiaoming&age=19
------------------query over-----------------
------------------version start-----------------
HTTP/1.1

HTTP/1.1
------------------version over-----------------
[dev@localhost example]$
```

**通用类型any类型的实现：**

每一个Connection对连接进行管理，最终都不可避免需要涉及到应用层协议的处理，因此在Connection中需要设置协议处理的上下文来控制处理节奏。但是应用层协议千千万，为了降低耦合度，这个协议接收解析上下文就不能有明显的协议倾向，它可以是任意协议的上下文信息，因此就需要一个通用的类型来保存各种不同的数据结构。

在C语言中，通用类型可以使用void*来管理，但是在C++中，boost库和C++17给我们提供了一个通用类型any来灵活使用，如果考虑增加代码的移植性，尽量减少第三方库的依赖，则可以使用C++17特性

中的any，或者自己来实现。而这个any通用类型类的实现其实并不复杂，以下是简单的部分实现。

```cpp
#include <iostream>
#include <string>
#include <cassert>
#include <typeinfo>
#include <typeindex>
#include <unistd.h>

/*Any类主要是实现一个通用类型出来，在c++17和boost库中都有现成的可以使用，但是这里实现一下了解其思想，这样也就避免了第三方库的使用了*/

/*首先Any类肯定不能是一个模板类，否则编译的时候 Any<int> a， Any<float>b,需要传类型作为模板参数，也就是说在使用的时候就要确定其类型*/
/*这是行不通的，因为保存在Content中的协议上下文，我们在定义any对象的时候是不知道他们的协议类型的，因此无法传递类型作为模板参数*/
/*因此考虑Any内部设计一个模板容器holder类，可以保存各种类型数据*/
/*而因为在Any类中无法定义这个holder对象或指针，因为any也不知道这个类要保存什么类型的数据，因此无法传递类型参数*/
/*所以，定义一个基类placehoder，让holder继承于placehold，而Any类保存父类指针即可*/
/*当需要保存数据时，则new一个带有模板参数的子类holder对象出来保存数据，然后让Any类中的父类指针，指向这个子类对象就搞定了*/
class Any {
    public:
        Any():_content(NULL) {}
        /*为了能够接收所有类型的对象，因此将构造函数定义为一个模板函数*/
        template<typename T>
        Any(const T &val) : _content(new holder<T>(val)){}
        Any(const Any &other): _content(other._content?other._content->clone() : NULL) {}
        ~Any() { if (_content) delete _content; }
        const std::type_info &type() { return _content ? _content->type() : typeid(void); }
        Any& swap(Any &other) {
            std::swap(_content, other._content);
            return *this;
        }
        template<typename T>
        T* get() {
            assert(typeid(T) == _content->type());
            return &((holder<T>*)_content)->val;
        }
        template<typename T>
        Any& operator=(const T &val) {
            /*为val构建一个临时对象出来，然后进行交换，这样临时对象销毁的时候，顺带原先保存的placeholder也会被销毁*/
```

```
37              Any(val).swap(*this);
38              return *this;
39          }
40          Any& operator=(Any other) {
41              /*这里要注意形参只是一个临时对象，进行交换后就会释放，所以交换后，原先保存的placeholder指针也会被销毁*/
42              other.swap(*this);
43              return *this;
44          }
45
46      private:
47          /*因为模板类编译时就会确定类型，因此*/
48          class placeholder {
49              public:
50                  virtual ~placeholder() {}
51                  virtual const std::type_info &type()  = 0;
52                  virtual placeholder *clone() = 0;
53          };
54          /*当前的Any类中无法保存所有类型的对象，或者说不能整成模板类，因此声明一个holder模板类出来使用holder类来管理传入的对象*/
55          /*而Any类只需要管理holder对象即可*/
56          template <typename T>
57          class holder : public placeholder {
58              public:
59                  holder(const T &v):val(v) {}
60                  ~holder() {}
61                  const std::type_info &type() { return typeid(T); }
62                  placeholder *clone() { return new holder(val); }
63              public:
64                  T val;
65          };
66          placeholder *_content;
67  };
68
69  class Test {
70      public:
71          std::string _data;
72      public:
73          Test(const std::string &data):_data(data) { std::cout << "构造" << _data << std::endl; }
74          Test(const Test &other) { _data = other._data;  std::cout << "拷贝" << _data << std::endl; }
75          ~Test() { std::cout << "析构" << _data << std::endl; }
76  };
77  int main()
78  {
79      {
```

```
80          int a = 10;
81          float b = 20;
82          std::string c = "Hello World";
83          Any any_a(a);
84          Any any_b(b);
85          Any any_c(c);
86          int *aa = any_a.get<int>();
87          float *bb = any_b.get<float>();
88          std::string *cc = any_c.get<std::string>();
89
90          std::cout << *aa << std::endl;
91          std::cout << *bb << std::endl;
92          std::cout << *cc << std::endl;
93      }
94      {
95          std::cout << "---------------通过构造和析构看看有没有内存泄漏-------------
    \n";
96          Test d("Leihou");
97
98          Any any_d = d;
99          Any any_e(d);
100         Any any_f(any_d);
101         Any any_g = any_d;
102     }
103     {
104         std::cout << "---------------any之间相互的赋值-就算any保存的类型不同也可以--
    -----------\n";
105         Any any_f;
106
107         any_f = 33;
108         int *ff = any_f.get<int>();
109         std::cout << *ff << std::endl;
110
111         std::string c = "Hello World";
112         any_f = c;
113         std::string *gg = any_f.get<std::string>();
114         std::cout << *gg << std::endl;
115
116         any_f = Any(Test("test"));
117         Test *hh = any_f.get<Test>();
118         std::cout << hh->_data << std::endl;
119     }
120
121     while(1) sleep(1);
122     return 0;
123 }
```

```
[dev@localhost example]$ g++ -std=c++11 any.cpp -o any
[dev@localhost example]$ ./any
10
20
Hello World
---------------通过构造和析构看看有没有内存泄漏-------------
构造Leihou
拷贝Leihou
拷贝Leihou
拷贝Leihou
拷贝Leihou
析构Leihou
析构Leihou
析构Leihou
析构Leihou
析构Leihou
---------------any之间相互的赋值-就算any保存的类型不同也可以-------------
33
Hello World
构造test
拷贝test
析构test
test
析构test
^C
[dev@localhost example]$
```

下面是C++17中any的使用用例：需要注意的是，C++17的特性需要高版本的g++编译器支持，建议g++ 7.3及以上版本。

```
[dev@localhost example]$ sudo yum install centos-release-scl-rh centos-release-scl
[dev@localhost example]$ sudo yum install devtoolset-7-gcc  devtoolset-7-gcc-c++
[dev@localhost example]$ source /opt/rh/devtoolset-7/enable
[dev@localhost example]$ echo "source /opt/rh/devtoolset-7/enable" >> ~/.bashrc
[dev@localhost example]$ g++ -v
Using built-in specs.
COLLECT_GCC=g++
COLLECT_LTO_WRAPPER=/opt/rh/devtoolset-7/root/usr/libexec/gcc/x86_64-redhat-linux/7/lto-wrapper
Target: x86_64-redhat-linux
Configured with: ../configure --enable-bootstrap --enable-languages=c,c++,fortran,lto --prefix=/opt/rh/devtoolset-7/root/usr --
```

```
mandir=/opt/rh/devtoolset-7/root/usr/share/man --infodir=/opt/rh/devtoolset-7/root/usr/share/info --with-bugurl=http://bugzilla.redhat.com/bugzilla --enable-shared --enable-threads=posix --enable-checking=release --enable-multilib --with-system-zlib --enable-__cxa_atexit --disable-libunwind-exceptions --enable-gnu-unique-object --enable-linker-build-id --with-gcc-major-version-only --enable-plugin --with-linker-hash-style=gnu --enable-initfini-array --with-default-libstdcxx-abi=gcc4-compatible --with-isl=/builddir/build/BUILD/gcc-7.3.1-20180303/obj-x86_64-redhat-linux/isl-install --enable-libmpx --enable-gnu-indirect-function --with-tune=generic --with-arch_32=i686 --build=x86_64-redhat-linux
Thread model: posix
gcc version 7.3.1 20180303 (Red Hat 7.3.1-5) (GCC)
```

```cpp
#include <iostream>
#include <string>
#include <any>
int main()
{
    {
        std::any a = 10;
        std::any b = 88.88;
        std::any c = std::string("bitejiuyeke");
        /* T* any_cast<class T>() 成员函数用于返回any对象值的地址 */
        int *aa = std::any_cast<int>(&a);
        std::cout << *aa << std::endl;
        std::cout << *std::any_cast<double>(&b) << std::endl;
        std::cout << *std::any_cast<std::string>(&c) << std::endl;
    }
    {
        Test d("Leihou");

        std::any any_d = d;
        std::any any_e(d);
        std::any any_f(any_d);
        std::any any_g = any_d;
    }
    {
        std::cout << "---------------any之间相互的赋值-就算any保存的类型不同也可以----------\n";
        std::any any_f;

        any_f = 33;
        std::cout << *std::any_cast<int>(&any_f) << std::endl;

        std::string c = "Hello World";
```

```
        any_f = c;
        std::cout << *std::any_cast<std::string>(&any_f) << std::endl;

        any_f = std::any(Test("test"));
        std::cout << std::any_cast<Test>(&any_f)->_data << std::endl;
    }

    return 0;
}
```

```
[dev@localhost example]$ g++ -std=c++17 any.cpp -o any
[dev@localhost example]$ ./any
10
88.88
bitejiuyeke
构造Leihou
拷贝Leihou
拷贝Leihou
拷贝Leihou
拷贝Leihou
析构Leihou
析构Leihou
析构Leihou
析构Leihou
析构Leihou
---------------any之间相互的赋值-就算any保存的类型不同也可以-------------
33
Hello World
构造test
拷贝test
析构test
test
析构test
[dev@localhost example]$
```

## SERVER服务器模块实现：

### 缓冲区Buffer类实现：

Buffer类用于实现用户态缓冲区，提供数据缓冲，取出等功能。

```
class Buffer {
    private:
        std::vector<char> _buffer; //使用vector进行内存空间管理
```

```
4         uint64_t _reader_idx; //读偏移
5         uint64_t _writer_idx; //写偏移
6     public:
7         Buffer():_reader_idx(0), _writer_idx(0), _buffer(BUFFER_DEFAULT_SIZE){}
8         char *Begin() { return &*_buffer.begin(); }
9         //获取当前写入起始地址，_buffer的空间起始地址，加上写偏移量
10        char *WritePosition() { return Begin() + _writer_idx; }
11        //获取当前读取起始地址
12        char *ReadPosition() { return Begin() + _reader_idx; }
13        //获取缓冲区末尾空闲空间大小--写偏移之后的空闲空间，总体空间大小减去写偏移
14        uint64_t TailIdleSize() { return _buffer.size() - _writer_idx; }
15        //获取缓冲区起始空闲空间大小--读偏移之前的空闲空间
16        uint64_t HeadIdleSize() { return _reader_idx; }
17        //获取可读数据大小 = 写偏移 - 读偏移
18        uint64_t ReadAbleSize() { return _writer_idx - _reader_idx; }
19        //将读偏移向后移动
20        void MoveReadOffset(uint64_t len) {
21            if (len == 0) return;
22            //向后移动的大小，必须小于可读数据大小
23            assert(len <= ReadAbleSize());
24            _reader_idx += len;
25        }
26        //将写偏移向后移动
27        void MoveWriteOffset(uint64_t len) {
28            //向后移动的大小，必须小于当前后边的空闲空间大小
29            assert(len <= TailIdleSize());
30            _writer_idx += len;
31        }
32        //确保可写空间足够（整体空闲空间够了就移动数据，否则就扩容）
33        void EnsureWriteSpace(uint64_t len) {
34            //如果末尾空闲空间大小足够，直接返回
35            if (TailIdleSize() >= len) { return; }
36            //末尾空闲空间不够，则判断加上起始位置的空闲空间大小是否足够，够了就将数据移动到起始位置
37            if (len <= TailIdleSize() + HeadIdleSize()) {
38                //将数据移动到起始位置
39                uint64_t rsz = ReadAbleSize();//把当前数据大小先保存起来
40                std::copy(ReadPosition(), ReadPosition() + rsz, Begin());//把可读数据拷贝到起始位置
41                _reader_idx = 0;    //将读偏移归0
42                _writer_idx = rsz;  //将写位置置为可读数据大小， 因为当前的可读数据大小就是写偏移量
43            }else {
44                //总体空间不够，则需要扩容，不移动数据，直接给写偏移之后扩容足够空间即可
45                DBG_LOG("RESIZE %ld", _writer_idx + len);
46                _buffer.resize(_writer_idx + len);
47            }
```

```
48          }
49          //写入数据
50          void Write(const void *data, uint64_t len) {
51              //1. 保证有足够空间，2. 拷贝数据进去
52              if (len == 0) return;
53              EnsureWriteSpace(len);
54              const char *d = (const char *)data;
55              std::copy(d, d + len, WritePosition());
56          }
57          void WriteAndPush(const void *data, uint64_t len) {
58              Write(data, len);
59              MoveWriteOffset(len);
60          }
61          void WriteString(const std::string &data) {
62              return Write(data.c_str(), data.size());
63          }
64          void WriteStringAndPush(const std::string &data) {
65              WriteString(data);
66              MoveWriteOffset(data.size());
67          }
68          void WriteBuffer(Buffer &data) {
69              return Write(data.ReadPosition(), data.ReadAbleSize());
70          }
71          void WriteBufferAndPush(Buffer &data) {
72              WriteBuffer(data);
73              MoveWriteOffset(data.ReadAbleSize());
74          }
75          //读取数据
76          void Read(void *buf, uint64_t len) {
77              //要求要获取的数据大小必须小于可读数据大小
78              assert(len <= ReadAbleSize());
79              std::copy(ReadPosition(), ReadPosition() + len, (char*)buf);
80          }
81          void ReadAndPop(void *buf, uint64_t len) {
82              Read(buf, len);
83              MoveReadOffset(len);
84          }
85          std::string ReadAsString(uint64_t len) {
86              //要求要获取的数据大小必须小于可读数据大小
87              assert(len <= ReadAbleSize());
88              std::string str;
89              str.resize(len);
90              Read(&str[0], len);
91              return str;
92          }
93          std::string ReadAsStringAndPop(uint64_t len) {
94              assert(len <= ReadAbleSize());
```

```
            std::string str = ReadAsString(len);
            MoveReadOffset(len);
            return str;
        }
        char *FindCRLF() {
            char *res = (char*)memchr(ReadPosition(), '\n', ReadAbleSize());
            return res;
        }
        /*通常获取一行数据，这种情况针对是*/
        std::string GetLine() {
            char *pos = FindCRLF();
            if (pos == NULL) {
                return "";
            }
            // +1是为了把换行字符也取出来。
            return ReadAsString(pos - ReadPosition() + 1);
        }
        std::string GetLineAndPop() {
            std::string str = GetLine();
            MoveReadOffset(str.size());
            return str;
        }
        //清空缓冲区
        void Clear() {
            //只需要将偏移量归0即可
            _reader_idx = 0;
            _writer_idx = 0;
        }
};
```

日志宏的实现：

```
#define INF 0
#define DBG 1
#define ERR 2
#define DEFAULT_LOG_LEVEL DBG
#define LOG(level, format, ...) {\
    if (level >= DEFAULT_LOG_LEVEL) {\
        time_t t = time(NULL);\
        struct tm *m = localtime(&t);\
        char ts[32] = {0};\
        strftime(ts, 31, "%H:%M:%S", m);\
        fprintf(stdout, "[%p %s %s:%d] " format "\n", (void*)pthread_self(), ts, __FILE__, __LINE__, ##__VA_ARGS__);\
    }\
```

```
}
#define INF_LOG(format, ...) LOG(INF, format, ##__VA_ARGS__);
#define DBG_LOG(format, ...) LOG(DBG, format, ##__VA_ARGS__);
#define ERR_LOG(format, ...) LOG(ERR, format, ##__VA_ARGS__);
```

套接字Socket类实现：

```
#define MAX_LISTEN 1024
class NetWork {
    public:
        NetWork() {
            DBG_LOG("SIGPIPE INIT");
            signal(SIGPIPE, SIG_IGN);
        }
};
/*避免服务器因为给断开连接的客户端进行send触发异常导致程序崩溃，因此忽略SIGPIPE信号*/
/*定义静态全局是为了保证构造函数中的信号忽略处理能够在程序启动阶段就被直接执行*/
static NetWork nw;

class Socket {
    private:
        int _sockfd;
    public:
        Socket():_sockfd(-1) {}
        Socket(int fd): _sockfd(fd) {}
        ~Socket() { Close(); }
        int Fd() { return _sockfd; }
        //创建套接字
        bool Create() {
            // int socket(int domain, int type, int protocol)
            _sockfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
            if (_sockfd < 0) {
                ERR_LOG("CREATE SOCKET FAILED!!");
                return false;
            }
            return true;
        }
        //绑定地址信息
        bool Bind(const std::string &ip, uint16_t port) {
            struct sockaddr_in addr;
            addr.sin_family = AF_INET;
            addr.sin_port = htons(port);
            addr.sin_addr.s_addr = inet_addr(ip.c_str());
            socklen_t len = sizeof(struct sockaddr_in);
            // int bind(int sockfd, struct sockaddr*addr, socklen_t len);
```

```
39              int ret = bind(_sockfd, (struct sockaddr*)&addr, len);
40              if (ret < 0) {
41                  ERR_LOG("BIND ADDRESS FAILED!");
42                  return false;
43              }
44              return true;
45          }
46          //开始监听
47          bool Listen(int backlog = MAX_LISTEN) {
48              // int listen(int backlog)
49              int ret = listen(_sockfd, backlog);
50              if (ret < 0) {
51                  ERR_LOG("SOCKET LISTEN FAILED!");
52                  return false;
53              }
54              return true;
55          }
56          //向服务器发起连接
57          bool Connect(const std::string &ip, uint16_t port) {
58              struct sockaddr_in addr;
59              addr.sin_family = AF_INET;
60              addr.sin_port = htons(port);
61              addr.sin_addr.s_addr = inet_addr(ip.c_str());
62              socklen_t len = sizeof(struct sockaddr_in);
63              // int connect(int sockfd, struct sockaddr*addr, socklen_t len);
64              int ret = connect(_sockfd, (struct sockaddr*)&addr, len);
65              if (ret < 0) {
66                  ERR_LOG("CONNECT SERVER FAILED!");
67                  return false;
68              }
69              return true;
70          }
71          //获取新连接
72          int Accept() {
73              // int accept(int sockfd, struct sockaddr *addr, socklen_t *len);
74              int newfd = accept(_sockfd, NULL, NULL);
75              if (newfd < 0) {
76                  ERR_LOG("SOCKET ACCEPT FAILED!");
77                  return -1;
78              }
79              return newfd;
80          }
81          //接收数据
82          ssize_t Recv(void *buf, size_t len, int flag = 0) {
83              // ssize_t recv(int sockfd, void *buf, size_t len, int flag);
84              ssize_t ret = recv(_sockfd, buf, len, flag);
85              if (ret <= 0) {
```

```
86                  //EAGAIN 当前socket的接收缓冲区中没有数据了，在非阻塞的情况下才会有这个错误
87                  //EINTR  表示当前socket的阻塞等待，被信号打断了，
88                  if (errno == EAGAIN || errno == EINTR) {
89                      return 0;//表示这次接收没有接收到数据
90                  }
91                  ERR_LOG("SOCKET RECV FAILED!!");
92                  return -1;
93              }
94              return ret; //实际接收的数据长度
95          }
96          ssize_t NonBlockRecv(void *buf, size_t len) {
97              return Recv(buf, len, MSG_DONTWAIT); // MSG_DONTWAIT 表示当前接收为非阻塞。
98          }
99          //发送数据
100         ssize_t Send(const void *buf, size_t len, int flag = 0) {
101             // ssize_t send(int sockfd, void *data, size_t len, int flag);
102             ssize_t ret = send(_sockfd, buf, len, flag);
103             if (ret < 0) {
104                 if (errno == EAGAIN || errno == EINTR) {
105                     return 0;
106                 }
107                 ERR_LOG("SOCKET SEND FAILED!!");
108                 return -1;
109             }
110             return ret;//实际发送的数据长度
111         }
112         ssize_t NonBlockSend(void *buf, size_t len) {
113             if (len == 0) return 0;
114             return Send(buf, len, MSG_DONTWAIT); // MSG_DONTWAIT 表示当前发送为非阻塞。
115         }
116         //关闭套接字
117         void Close() {
118             if (_sockfd != -1) {
119                 close(_sockfd);
120                 _sockfd = -1;
121             }
122         }
123         //创建一个服务端连接
124         bool CreateServer(uint16_t port, const std::string &ip = "0.0.0.0", bool block_flag = false) {
125             //1. 创建套接字，2. 绑定地址，3. 开始监听，4. 设置非阻塞， 5. 启动地址重用
126             if (Create() == false) return false;
127             if (block_flag) NonBlock();
128             if (Bind(ip, port) == false) return false;
```

```
            if (Listen() == false) return false;
            ReuseAddress();
            return true;
        }
        //创建一个客户端连接
        bool CreateClient(uint16_t port, const std::string &ip) {
            //1. 创建套接字，2.指向连接服务器
            if (Create() == false) return false;
            if (Connect(ip, port) == false) return false;
            return true;
        }
        //设置套接字选项---开启地址端口重用
        void ReuseAddress() {
            // int setsockopt(int fd, int leve, int optname, void *val, int vallen)
            int val = 1;
            setsockopt(_sockfd, SOL_SOCKET, SO_REUSEADDR, (void*)&val, sizeof(int));
            val = 1;
            setsockopt(_sockfd, SOL_SOCKET, SO_REUSEPORT, (void*)&val, sizeof(int));
        }
        //设置套接字阻塞属性-- 设置为非阻塞
        void NonBlock() {
            //int fcntl(int fd, int cmd, ... /* arg */ );
            int flag = fcntl(_sockfd, F_GETFL, 0);
            fcntl(_sockfd, F_SETFL, flag | O_NONBLOCK);
        }
};
```

事件管理Channel类实现：

```

class Poller;
class EventLoop;
class Channel {
    private:
        int _fd;
        EventLoop *_loop;
        uint32_t _events;  // 当前需要监控的事件
        uint32_t _revents; // 当前连接触发的事件
        using EventCallback = std::function<void()>;
        EventCallback _read_callback;   //可读事件被触发的回调函数
        EventCallback _write_callback;  //可写事件被触发的回调函数
        EventCallback _error_callback;  //错误事件被触发的回调函数
```

```
14          EventCallback _close_callback;  //连接断开事件被触发的回调函数
15          EventCallback _event_callback;  //任意事件被触发的回调函数
16      public:
17          Channel(EventLoop *loop, int fd):_fd(fd), _events(0), _revents(0),
    _loop(loop) {}
18          int Fd() { return _fd; }
19          uint32_t Events() { return _events; }//获取想要监控的事件
20          void SetREvents(uint32_t events) { _revents = events; }//设置实际就绪的事
    件
21          void SetReadCallback(const EventCallback &cb) { _read_callback = cb; }
22          void SetWriteCallback(const EventCallback &cb) { _write_callback = cb;
    }
23          void SetErrorCallback(const EventCallback &cb) { _error_callback = cb;
    }
24          void SetCloseCallback(const EventCallback &cb) { _close_callback = cb;
    }
25          void SetEventCallback(const EventCallback &cb) { _event_callback = cb;
    }
26          //当前是否监控了可读
27          bool ReadAble() { return (_events & EPOLLIN); }
28          //当前是否监控了可写
29          bool WriteAble() { return (_events & EPOLLOUT); }
30          //启动读事件监控
31          void EnableRead() { _events |= EPOLLIN; Update(); }
32          //启动写事件监控
33          void EnableWrite() { _events |= EPOLLOUT; Update(); }
34          //关闭读事件监控
35          void DisableRead() { _events &= ~EPOLLIN; Update(); }
36          //关闭写事件监控
37          void DisableWrite() { _events &= ~EPOLLOUT; Update(); }
38          //关闭所有事件监控
39          void DisableAll() { _events = 0; Update(); }
40          //移除监控
41          void Remove();
42          void Update();
43          //事件处理，一旦连接触发了事件，就调用这个函数，自己触发了什么事件如何处理自己决
    定
44          void HandleEvent() {
45              if ((_revents & EPOLLIN) || (_revents & EPOLLRDHUP) || (_revents &
    EPOLLPRI)) {
46                  /*不管任何事件，都调用的回调函数*/
47                  if (_read_callback) _read_callback();
48              }
49              /*有可能会释放连接的操作事件，一次只处理一个*/
50              if (_revents & EPOLLOUT) {
51                  if (_write_callback) _write_callback();
52              }else if (_revents & EPOLLERR) {
```

```
                if (_error_callback) _error_callback();
            }else if (_revents & EPOLLHUP) {
                if (_close_callback) _close_callback();
            }
            if (_event_callback) _event_callback();
        }
};
void Channel::Remove() { return _loop->RemoveEvent(this); }
void Channel::Update() { return _loop->UpdateEvent(this); }
```

描述符事件监控Poller类实现：

```
#define MAX_EPOLLEVENTS 1024
class Poller {
    private:
        int _epfd;
        struct epoll_event _evs[MAX_EPOLLEVENTS];
        std::unordered_map<int, Channel *> _channels;
    private:
        //对epoll的直接操作
        void Update(Channel *channel, int op) {
            // int epoll_ctl(int epfd, int op,  int fd,  struct epoll_event *ev);
            int fd = channel->Fd();
            struct epoll_event ev;
            ev.data.fd = fd;
            ev.events = channel->Events();
            int ret = epoll_ctl(_epfd, op, fd, &ev);
            if (ret < 0) {
                ERR_LOG("EPOLLCTL FAILED!");
            }
            return;
        }
        //判断一个Channel是否已经添加了事件监控
        bool HasChannel(Channel *channel) {
            auto it = _channels.find(channel->Fd());
            if (it == _channels.end()) {
                return false;
            }
            return true;
        }
    public:
        Poller() {
            _epfd = epoll_create(MAX_EPOLLEVENTS);
            if (_epfd < 0) {
```

```
33                  ERR_LOG("EPOLL CREATE FAILED!!");
34                  abort();//退出程序
35              }
36          }
37          //添加或修改监控事件
38          void UpdateEvent(Channel *channel) {
39              bool ret = HasChannel(channel);
40              if (ret == false) {
41                  //不存在则添加
42                  _channels.insert(std::make_pair(channel->Fd(), channel));
43                  return Update(channel, EPOLL_CTL_ADD);
44              }
45              return Update(channel, EPOLL_CTL_MOD);
46          }
47          //移除监控
48          void RemoveEvent(Channel *channel) {
49              auto it = _channels.find(channel->Fd());
50              if (it != _channels.end()) {
51                  _channels.erase(it);
52              }
53              Update(channel, EPOLL_CTL_DEL);
54          }
55          //开始监控，返回活跃连接
56          void Poll(std::vector<Channel*> *active) {
57              // int epoll_wait(int epfd, struct epoll_event *evs, int
    maxevents, int timeout)
58              int nfds = epoll_wait(_epfd, _evs, MAX_EPOLLEVENTS, -1);
59              if (nfds < 0) {
60                  if (errno == EINTR) {
61                      return ;
62                  }
63                  ERR_LOG("EPOLL WAIT ERROR:%s\n", strerror(errno));
64                  abort();//退出程序
65              }
66              for (int i = 0; i < nfds; i++) {
67                  auto it = _channels.find(_evs[i].data.fd);
68                  assert(it != _channels.end());
69                  it->second->SetREvents(_evs[i].events);//设置实际就绪的事件
70                  active->push_back(it->second);
71              }
72              return;
73          }
74  };
75
```

定时任务管理TimerWheel类实现：

```cpp


using TaskFunc = std::function<void()>;
using ReleaseFunc = std::function<void()>;
class TimerTask{
    private:
        uint64_t _id;       // 定时器任务对象ID
        uint32_t _timeout;  //定时任务的超时时间
        bool _canceled;     // false-表示没有被取消， true-表示被取消
        TaskFunc _task_cb;  //定时器对象要执行的定时任务
        ReleaseFunc _release; //用于删除TimerWheel中保存的定时器对象信息
    public:
        TimerTask(uint64_t id, uint32_t delay, const TaskFunc &cb):
            _id(id), _timeout(delay), _task_cb(cb), _canceled(false) {}
        ~TimerTask() {
            if (_canceled == false) _task_cb();
            _release();
        }
        void Cancel() { _canceled = true; }
        void SetRelease(const ReleaseFunc &cb) { _release = cb; }
        uint32_t DelayTime() { return _timeout; }
};

class TimerWheel {
    private:
        using WeakTask = std::weak_ptr<TimerTask>;
        using PtrTask = std::shared_ptr<TimerTask>;
        int _tick;      //当前的秒针，走到哪里释放哪里，释放哪里，就相当于执行哪里的任务
        int _capacity;  //表盘最大数量---其实就是最大延迟时间
        std::vector<std::vector<PtrTask>> _wheel;
        std::unordered_map<uint64_t, WeakTask> _timers;

        EventLoop *_loop;
        int _timerfd;//定时器描述符--可读事件回调就是读取计数器，执行定时任务
        std::unique_ptr<Channel> _timer_channel;
    private:
        void RemoveTimer(uint64_t id) {
            auto it = _timers.find(id);
            if (it != _timers.end()) {
                _timers.erase(it);
            }
        }
        static int CreateTimerfd() {
            int timerfd = timerfd_create(CLOCK_MONOTONIC, 0);
            if (timerfd < 0) {
```

```
46                 ERR_LOG("TIMERFD CREATE FAILED!");
47                 abort();
48             }
49             //int timerfd_settime(int fd, int flags, struct itimerspec *new, struct itimerspec *old);
50             struct itimerspec itime;
51             itime.it_value.tv_sec = 1;
52             itime.it_value.tv_nsec = 0;//第一次超时时间为1s后
53             itime.it_interval.tv_sec = 1;
54             itime.it_interval.tv_nsec = 0; //第一次超时后，每次超时的间隔时
55             timerfd_settime(timerfd, 0, &itime, NULL);
56             return timerfd;
57         }
58         int ReadTimefd() {
59             uint64_t times;
60             //有可能因为其他描述符的事件处理花费事件比较长，然后在处理定时器描述符事件的时候，有可能就已经超时了很多次
61             //read读取到的数据times就是从上一次read之后超时的次数
62             int ret = read(_timerfd, &times, 8);
63             if (ret < 0) {
64                 ERR_LOG("READ TIMEFD FAILED!");
65                 abort();
66             }
67             return times;
68         }
69         //这个函数应该每秒钟被执行一次，相当于秒针向后走了一步
70         void RunTimerTask() {
71             _tick = (_tick + 1) % _capacity;
72             _wheel[_tick].clear();//清空指定位置的数组，就会把数组中保存的所有管理定时器对象的shared_ptr释放掉
73         }
74         void OnTime() {
75             //根据实际超时的次数，执行对应的超时任务
76             int times = ReadTimefd();
77             for (int i = 0; i < times; i++) {
78                 RunTimerTask();
79             }
80         }
81         void TimerAddInLoop(uint64_t id, uint32_t delay, const TaskFunc &cb) {
82             PtrTask pt(new TimerTask(id, delay, cb));
83             pt->SetRelease(std::bind(&TimerWheel::RemoveTimer, this, id));
84             int pos = (_tick + delay) % _capacity;
85             _wheel[pos].push_back(pt);
86             _timers[id] = WeakTask(pt);
87         }
88         void TimerRefreshInLoop(uint64_t id) {
89             //通过保存的定时器对象的weak_ptr构造一个shared_ptr出来，添加到轮子中
```

```
90              auto it = _timers.find(id);
91              if (it == _timers.end()) {
92                  return;//没找着定时任务，没法刷新，没法延迟
93              }
94              PtrTask pt = it->second.lock();//lock获取weak_ptr管理的对象对应的
    shared_ptr
95              int delay = pt->DelayTime();
96              int pos = (_tick + delay) % _capacity;
97              _wheel[pos].push_back(pt);
98          }
99          void TimerCancelInLoop(uint64_t id) {
100             auto it = _timers.find(id);
101             if (it == _timers.end()) {
102                 return;//没找着定时任务，没法刷新，没法延迟
103             }
104             PtrTask pt = it->second.lock();
105             if (pt) pt->Cancel();
106         }
107     public:
108         TimerWheel(EventLoop *loop):_capacity(60), _tick(0),
    _wheel(_capacity), _loop(loop),
109             _timerfd(CreateTimerfd()), _timer_channel(new Channel(_loop,
    _timerfd)) {
110             _timer_channel->SetReadCallback(std::bind(&TimerWheel::OnTime,
    this));
111             _timer_channel->EnableRead();//启动读事件监控
112         }
113         /*定时器中有个_timers成员，定时器信息的操作有可能在多线程中进行，因此需要考虑线
    程安全问题*/
114         /*如果不想加锁，那就把对定期的所有操作，都放到一个线程中进行*/
115         void TimerAdd(uint64_t id, uint32_t delay, const TaskFunc &cb);
116         //刷新/延迟定时任务
117         void TimerRefresh(uint64_t id);
118         void TimerCancel(uint64_t id);
119         /*这个接口存在线程安全问题--这个接口实际上不能被外界使用者调用，只能在模块内，在
    对应的EventLoop线程内执行*/
120         bool HasTimer(uint64_t id) {
121             auto it = _timers.find(id);
122             if (it == _timers.end()) {
123                 return false;
124             }
125             return true;
126         }
127 };
128
129 void TimerWheel::TimerAdd(uint64_t id, uint32_t delay, const TaskFunc &cb) {
```

```
    _loop->RunInLoop(std::bind(&TimerWheel::TimerAddInLoop, this, id, delay, cb));
}
//刷新/延迟定时任务
void TimerWheel::TimerRefresh(uint64_t id) {
    _loop->RunInLoop(std::bind(&TimerWheel::TimerRefreshInLoop, this, id));
}
void TimerWheel::TimerCancel(uint64_t id) {
    _loop->RunInLoop(std::bind(&TimerWheel::TimerCancelInLoop, this, id));
}
```

Reactor-EventLoop线程池类实现：

```

class EventLoop {
    private:
        using Functor = std::function<void()>;
        std::thread::id _thread_id;//线程ID
        int _event_fd;//eventfd唤醒IO事件监控有可能导致的阻塞
        std::unique_ptr<Channel> _event_channel;
        Poller _poller;//进行所有描述符的事件监控
        std::vector<Functor> _tasks;//任务池
        std::mutex _mutex;//实现任务池操作的线程安全
        TimerWheel _timer_wheel;//定时器模块
    public:
        //执行任务池中的所有任务
        void RunAllTask() {
            std::vector<Functor> functor;
            {
                std::unique_lock<std::mutex> _lock(_mutex);
                _tasks.swap(functor);
            }
            for (auto &f : functor) {
                f();
            }
            return ;
        }
        static int CreateEventFd() {
            int efd = eventfd(0, EFD_CLOEXEC | EFD_NONBLOCK);
            if (efd < 0) {
                ERR_LOG("CREATE EVENTFD FAILED!!");
                abort();//让程序异常退出
            }
            return efd;
        }
```

```
33          void ReadEventfd() {
34              uint64_t res = 0;
35              int ret = read(_event_fd, &res, sizeof(res));
36              if (ret < 0) {
37                  //EINTR -- 被信号打断;    EAGAIN -- 表示无数据可读
38                  if (errno == EINTR || errno == EAGAIN) {
39                      return;
40                  }
41                  ERR_LOG("READ EVENTFD FAILED!");
42                  abort();
43              }
44              return ;
45          }
46          void WeakUpEventFd() {
47              uint64_t val = 1;
48              int ret = write(_event_fd, &val, sizeof(val));
49              if (ret < 0) {
50                  if (errno == EINTR) {
51                      return;
52                  }
53                  ERR_LOG("READ EVENTFD FAILED!");
54                  abort();
55              }
56              return ;
57          }
58      public:
59          EventLoop():_thread_id(std::this_thread::get_id()),
60                      _event_fd(CreateEventFd()),
61                      _event_channel(new Channel(this, _event_fd)),
62                      _timer_wheel(this) {
63              //给eventfd添加可读事件回调函数，读取eventfd事件通知次数
64              _event_channel->SetReadCallback(std::bind(&EventLoop::ReadEventfd,
    this));
65              //启动eventfd的读事件监控
66              _event_channel->EnableRead();
67          }
68          //三步走--事件监控-》就绪事件处理-》执行任务
69          void Start() {
70              while(1) {
71                  //1. 事件监控,
72                  std::vector<Channel *> actives;
73                  _poller.Poll(&actives);
74                  //2. 事件处理。
75                  for (auto &channel : actives) {
76                      channel->HandleEvent();
77                  }
78                  //3. 执行任务
```

```
79                  RunAllTask();
80              }
81          }
82          //用于判断当前线程是否是EventLoop对应的线程;
83          bool IsInLoop() {
84              return (_thread_id == std::this_thread::get_id());
85          }
86          void AssertInLoop() {
87              assert(_thread_id == std::this_thread::get_id());
88          }
89          //判断将要执行的任务是否处于当前线程中，如果是则执行，不是则压入队列。
90          void RunInLoop(const Functor &cb) {
91              if (IsInLoop()) {
92                  return cb();
93              }
94              return QueueInLoop(cb);
95          }
96          //将操作压入任务池
97          void QueueInLoop(const Functor &cb) {
98              {
99                  std::unique_lock<std::mutex> _lock(_mutex);
100                 _tasks.push_back(cb);
101             }
102             //唤醒有可能因为没有事件就绪，而导致的epoll阻塞;
103             //其实就是给eventfd写入一个数据，eventfd就会触发可读事件
104             WeakUpEventFd();
105         }
106         //添加/修改描述符的事件监控
107         void UpdateEvent(Channel *channel) { return
    _poller.UpdateEvent(channel); }
108         //移除描述符的监控
109         void RemoveEvent(Channel *channel) { return
    _poller.RemoveEvent(channel); }
110         void TimerAdd(uint64_t id, uint32_t delay, const TaskFunc &cb) { return
     _timer_wheel.TimerAdd(id, delay, cb); }
111         void TimerRefresh(uint64_t id) { return _timer_wheel.TimerRefresh(id);
    }
112         void TimerCancel(uint64_t id) { return _timer_wheel.TimerCancel(id); }
113         bool HasTimer(uint64_t id) { return _timer_wheel.HasTimer(id); }
114 };
115 class LoopThread {
116     private:
117         /*用于实现_loop获取的同步关系，避免线程创建了，但是_loop还没有实例化之前去获取
    _loop*/
118         std::mutex _mutex;          // 互斥锁
119         std::condition_variable _cond;    // 条件变量
120         EventLoop *_loop;       // EventLoop指针变量，这个对象需要在线程内实例化
```

```
121         std::thread _thread;    // EventLoop对应的线程
122     private:
123         /*实例化 EventLoop 对象，唤醒_cond上有可能阻塞的线程，并且开始运行EventLoop模块的功能*/
124         void ThreadEntry() {
125             EventLoop loop;
126             {
127                 std::unique_lock<std::mutex> lock(_mutex);//加锁
128                 _loop = &loop;
129                 _cond.notify_all();
130             }
131             loop.Start();
132         }
133     public:
134         /*创建线程，设定线程入口函数*/
135         LoopThread():_loop(NULL), _thread(std::thread(&LoopThread::ThreadEntry, this)) {}
136         /*返回当前线程关联的EventLoop对象指针*/
137         EventLoop *GetLoop() {
138             EventLoop *loop = NULL;
139             {
140                 std::unique_lock<std::mutex> lock(_mutex);//加锁
141                 _cond.wait(lock, [&](){ return _loop != NULL; });//loop为NULL就一直阻塞
142                 loop = _loop;
143             }
144             return loop;
145         }
146 };
147 
148 class LoopThreadPool {
149     private:
150         int _thread_count;
151         int _next_idx;
152         EventLoop *_baseloop;
153         std::vector<LoopThread*> _threads;
154         std::vector<EventLoop *> _loops;
155     public:
156         LoopThreadPool(EventLoop *baseloop):_thread_count(0), _next_idx(0), _baseloop(baseloop) {}
157         void SetThreadCount(int count) { _thread_count = count; }
158         void Create() {
159             if (_thread_count > 0) {
160                 _threads.resize(_thread_count);
161                 _loops.resize(_thread_count);
162                 for (int i = 0; i < _thread_count; i++) {
163                     _threads[i] = new LoopThread();
```

```
                    _loops[i] = _threads[i]->GetLoop();
                }
            }
            return ;
        }
        EventLoop *NextLoop() {
            if (_thread_count == 0) {
                return _baseloop;
            }
            _next_idx = (_next_idx + 1) % _thread_count;
            return _loops[_next_idx];
        }
};
```

通信连接管理Connection类实现：

```

class Connection;
//DISCONECTED -- 连接关闭状态;    CONNECTING -- 连接建立成功-待处理状态
//CONNECTED -- 连接建立完成，各种设置已完成，可以通信的状态;   DISCONNECTING -- 待关闭状态
typedef enum { DISCONNECTED, CONNECTING, CONNECTED, DISCONNECTING}ConnStatu;
using PtrConnection = std::shared_ptr<Connection>;
class Connection : public std::enable_shared_from_this<Connection> {
    private:
        uint64_t _conn_id;  // 连接的唯一ID，便于连接的管理和查找
        //uint64_t _timer_id;   //定时器ID，必须是唯一的，这块为了简化操作使用conn_id作为定时器ID
        int _sockfd;            // 连接关联的文件描述符
        bool _enable_inactive_release;  // 连接是否启动非活跃销毁的判断标志，默认为false
        EventLoop *_loop;   // 连接所关联的一个EventLoop
        ConnStatu _statu;   // 连接状态
        Socket _socket;     // 套接字操作管理
        Channel _channel;   // 连接的事件管理
        Buffer _in_buffer;  // 输入缓冲区---存放从socket中读取到的数据
        Buffer _out_buffer; // 输出缓冲区---存放要发送给对端的数据
        Any _context;       // 请求的接收处理上下文

        /*这四个回调函数，是让服务器模块来设置的（其实服务器模块的处理回调也是组件使用者设置的）*/
        /*换句话说，这几个回调都是组件使用者使用的*/
        using ConnectedCallback = std::function<void(const PtrConnection&)>;
```

```
25          using MessageCallback = std::function<void(const PtrConnection&,
    Buffer *)>;
26          using ClosedCallback = std::function<void(const PtrConnection&)>;
27          using AnyEventCallback = std::function<void(const PtrConnection&)>;
28          ConnectedCallback _connected_callback;
29          MessageCallback _message_callback;
30          ClosedCallback _closed_callback;
31          AnyEventCallback _event_callback;
32          /*组件内的连接关闭回调--组件内设置的，因为服务器组件内会把所有的连接管理起来，一
    旦某个连接要关闭*/
33          /*就应该从管理的地方移除掉自己的信息*/
34          ClosedCallback _server_closed_callback;
35      private:
36          /*五个channel的事件回调函数*/
37          //描述符可读事件触发后调用的函数，接收socket数据放到接收缓冲区中，然后调用
    _message_callback
38          void HandleRead() {
39              //1. 接收socket的数据，放到缓冲区
40              char buf[65536];
41              ssize_t ret = _socket.NonBlockRecv(buf, 65535);
42              if (ret < 0) {
43                  //出错了,不能直接关闭连接
44                  return ShutdownInLoop();
45              }
46              //这里的等于0表示的是没有读取到数据，而并不是连接断开了，连接断开返回的是-1
47              //将数据放入输入缓冲区,写入之后顺便将写偏移向后移动
48              _in_buffer.WriteAndPush(buf, ret);
49              //2. 调用message_callback进行业务处理
50              if (_in_buffer.ReadAbleSize() > 0) {
51                  //shared_from_this--从当前对象自身获取自身的shared_ptr管理对象
52                  return _message_callback(shared_from_this(), &_in_buffer);
53              }
54          }
55          //描述符可写事件触发后调用的函数，将发送缓冲区中的数据进行发送
56          void HandleWrite() {
57              //_out_buffer中保存的数据就是要发送的数据
58              ssize_t ret = _socket.NonBlockSend(_out_buffer.ReadPosition(),
    _out_buffer.ReadAbleSize());
59              if (ret < 0) {
60                  //发送错误就该关闭连接了，
61                  if (_in_buffer.ReadAbleSize() > 0) {
62                      _message_callback(shared_from_this(), &_in_buffer);
63                  }
64                  return Release();//这时候就是实际的关闭释放操作了。
65              }
66              _out_buffer.MoveReadOffset(ret);//千万不要忘了，将读偏移向后移动
67              if (_out_buffer.ReadAbleSize() == 0) {
```

```
68                  _channel.DisableWrite();// 没有数据待发送了，关闭写事件监控
69                  //如果当前是连接待关闭状态，则有数据，发送完数据释放连接，没有数据则直接释放
70                  if (_statu == DISCONNECTING) {
71                      return Release();
72                  }
73              }
74              return;
75          }
76          //描述符触发挂断事件
77          void HandleClose() {
78              /*一旦连接挂断了，套接字就什么都干不了了，因此有数据待处理就处理一下，完毕关闭连接*/
79              if (_in_buffer.ReadAbleSize() > 0) {
80                  _message_callback(shared_from_this(), &_in_buffer);
81              }
82              return Release();
83          }
84          //描述符触发出错事件
85          void HandleError() {
86              return HandleClose();
87          }
88          //描述符触发任意事件：1. 刷新连接的活跃度--延迟定时销毁任务；  2. 调用组件使用者的任意事件回调
89          void HandleEvent() {
90              if (_enable_inactive_release == true)  {  _loop->TimerRefresh(_conn_id); }
91              if (_event_callback)  {  _event_callback(shared_from_this()); }
92          }
93          //连接获取之后，所处的状态下要进行各种设置（启动读监控,调用回调函数）
94          void EstablishedInLoop() {
95              // 1. 修改连接状态；  2. 启动读事件监控；  3. 调用回调函数
96              assert(_statu == CONNECTING);//当前的状态必须一定是上层的半连接状态
97              _statu = CONNECTED;//当前函数执行完毕，则连接进入已完成连接状态
98              // 一旦启动读事件监控就有可能会立即触发读事件，如果这时候启动了非活跃连接销毁
99              _channel.EnableRead();
100             if (_connected_callback) _connected_callback(shared_from_this());
101         }
102         //这个接口才是实际的释放接口
103         void ReleaseInLoop() {
104             //1. 修改连接状态，将其置为DISCONNECTED
105             _statu = DISCONNECTED;
106             //2. 移除连接的事件监控
107             _channel.Remove();
108             //3. 关闭描述符
109             _socket.Close();
110             //4. 如果当前定时器队列中还有定时销毁任务，则取消任务
```

```
111             if (_loop->HasTimer(_conn_id)) CancelInactiveReleaseInLoop();
112             //5. 调用关闭回调函数，避免先移除服务器管理的连接信息导致Connection被释放，再去处理会出错，因此先调用用户的回调函数
113             if (_closed_callback) _closed_callback(shared_from_this());
114             //移除服务器内部管理的连接信息
115             if (_server_closed_callback) _server_closed_callback(shared_from_this());
116         }
117         //这个接口并不是实际的发送接口，而只是把数据放到了发送缓冲区，启动了可写事件监控
118         void SendInLoop(Buffer &buf) {
119             if (_statu == DISCONNECTED) return ;
120             _out_buffer.WriteBufferAndPush(buf);
121             if (_channel.WriteAble() == false) {
122                 _channel.EnableWrite();
123             }
124         }
125         //这个关闭操作并非实际的连接释放操作，需要判断还有没有数据待处理，待发送
126         void ShutdownInLoop() {
127             _statu = DISCONNECTING;// 设置连接为半关闭状态
128             if (_in_buffer.ReadAbleSize() > 0) {
129                 if (_message_callback) _message_callback(shared_from_this(), &_in_buffer);
130             }
131             //要么就是写入数据的时候出错关闭，要么就是没有待发送数据，直接关闭
132             if (_out_buffer.ReadAbleSize() > 0) {
133                 if (_channel.WriteAble() == false) {
134                     _channel.EnableWrite();
135                 }
136             }
137             if (_out_buffer.ReadAbleSize() == 0) {
138                 Release();
139             }
140         }
141         //启动非活跃连接超时释放规则
142         void EnableInactiveReleaseInLoop(int sec) {
143             //1. 将判断标志 _enable_inactive_release 置为true
144             _enable_inactive_release = true;
145             //2. 如果当前定时销毁任务已经存在，那就刷新延迟一下即可
146             if (_loop->HasTimer(_conn_id)) {
147                 return _loop->TimerRefresh(_conn_id);
148             }
149             //3. 如果不存在定时销毁任务，则新增
150             _loop->TimerAdd(_conn_id, sec, std::bind(&Connection::Release, this));
151         }
152         void CancelInactiveReleaseInLoop() {
153             _enable_inactive_release = false;
```

```
154             if (_loop->HasTimer(_conn_id)) {
155                 _loop->TimerCancel(_conn_id);
156             }
157         }
158         void UpgradeInLoop(const Any &context,
159                     const ConnectedCallback &conn,
160                     const MessageCallback &msg,
161                     const ClosedCallback &closed,
162                     const AnyEventCallback &event) {
163             _context = context;
164             _connected_callback = conn;
165             _message_callback = msg;
166             _closed_callback = closed;
167             _event_callback = event;
168         }
169     public:
170         Connection(EventLoop *loop, uint64_t conn_id, int
    sockfd):_conn_id(conn_id), _sockfd(sockfd),
171             _enable_inactive_release(false), _loop(loop), _statu(CONNECTING),
    _socket(_sockfd),
172             _channel(loop, _sockfd) {
173             _channel.SetCloseCallback(std::bind(&Connection::HandleClose,
    this));
174             _channel.SetEventCallback(std::bind(&Connection::HandleEvent,
    this));
175             _channel.SetReadCallback(std::bind(&Connection::HandleRead, this));
176             _channel.SetWriteCallback(std::bind(&Connection::HandleWrite,
    this));
177             _channel.SetErrorCallback(std::bind(&Connection::HandleError,
    this));
178         }
179         ~Connection() { DBG_LOG("RELEASE CONNECTION:%p", this); }
180         //获取管理的文件描述符
181         int Fd() { return _sockfd; }
182         //获取连接ID
183         int Id() { return _conn_id; }
184         //是否处于CONNECTED状态
185         bool Connected() { return (_statu == CONNECTED); }
186         //设置上下文--连接建立完成时进行调用
187         void SetContext(const Any &context) { _context = context; }
188         //获取上下文，返回的是指针
189         Any *GetContext() { return &_context; }
190         void SetConnectedCallback(const ConnectedCallback&cb) {
    _connected_callback = cb; }
191         void SetMessageCallback(const MessageCallback&cb) { _message_callback
    = cb; }
```

```
192         void SetClosedCallback(const ClosedCallback&cb) { _closed_callback =
    cb; }
193         void SetAnyEventCallback(const AnyEventCallback&cb) { _event_callback
    = cb; }
194         void SetSrvClosedCallback(const ClosedCallback&cb) {
    _server_closed_callback = cb; }
195         //连接建立就绪后，进行channel回调设置，启动读监控，调用_connected_callback
196         void Established() {
197             _loop->RunInLoop(std::bind(&Connection::EstablishedInLoop, this));
198         }
199         //发送数据，将数据放到发送缓冲区，启动写事件监控
200         void Send(const char *data, size_t len) {
201             //外界传入的data，可能是个临时的空间，我们现在只是把发送操作压入了任务池，有
    可能并没有被立即执行
202             //因此有可能执行的时候，data指向的空间有可能已经被释放了。
203             Buffer buf;
204             buf.WriteAndPush(data, len);
205             _loop->RunInLoop(std::bind(&Connection::SendInLoop, this,
    std::move(buf)));
206         }
207         //提供给组件使用者的关闭接口--并不实际关闭，需要判断有没有数据待处理
208         void Shutdown() {
209             _loop->RunInLoop(std::bind(&Connection::ShutdownInLoop, this));
210         }
211         void Release() {
212             _loop->QueueInLoop(std::bind(&Connection::ReleaseInLoop, this));
213         }
214         //启动非活跃销毁，并定义多长时间无通信就是非活跃，添加定时任务
215         void EnableInactiveRelease(int sec) {
216             _loop-
    >RunInLoop(std::bind(&Connection::EnableInactiveReleaseInLoop, this, sec));
217         }
218         //取消非活跃销毁
219         void CancelInactiveRelease() {
220             _loop-
    >RunInLoop(std::bind(&Connection::CancelInactiveReleaseInLoop, this));
221         }
222         //切换协议---重置上下文以及阶段性回调处理函数 -- 而是这个接口必须在EventLoop线
    程中立即执行
223         //防备新的事件触发后，处理的时候，切换任务还没有被执行--会导致数据使用原协议处理
    了。
224         void Upgrade(const Any &context, const ConnectedCallback &conn, const
    MessageCallback &msg,
225                      const ClosedCallback &closed, const AnyEventCallback
    &event) {
226             _loop->AssertInLoop();
```

```
            _loop->RunInLoop(std::bind(&Connection::UpgradeInLoop, this,
context, conn, msg, closed, event));
        }
};
```

监听描述符管理Acceptor类实现：

```

class Acceptor {
    private:
        Socket _socket;//用于创建监听套接字
        EventLoop *_loop; //用于对监听套接字进行事件监控
        Channel _channel; //用于对监听套接字进行事件管理

        using AcceptCallback = std::function<void(int)>;
        AcceptCallback _accept_callback;
    private:
        /*监听套接字的读事件回调处理函数---获取新连接，调用_accept_callback函数进行新连
接处理*/
        void HandleRead() {
            int newfd = _socket.Accept();
            if (newfd < 0) {
                return ;
            }
            if (_accept_callback) _accept_callback(newfd);
        }
        int CreateServer(int port) {
            bool ret = _socket.CreateServer(port);
            assert(ret == true);
            return _socket.Fd();
        }
    public:
        /*不能将启动读事件监控，放到构造函数中，必须在设置回调函数后，再去启动*/
        /*否则有可能造成启动监控后，立即有事件，处理的时候，回调函数还没设置：新连接得不
到处理，且资源泄漏*/
        Acceptor(EventLoop *loop, int port): _socket(CreateServer(port)),
_loop(loop),
            _channel(loop, _socket.Fd()) {
            _channel.SetReadCallback(std::bind(&Acceptor::HandleRead, this));
        }
        void SetAcceptCallback(const AcceptCallback &cb) { _accept_callback =
cb; }
        void Listen() { _channel.EnableRead(); }
};
```

服务器TcpServer类实现：

```
class TcpServer {
    private:
        uint64_t _next_id;      //这是一个自动增长的连接ID，
        int _port;
        int _timeout;           //这是非活跃连接的统计时间---多长时间无通信就是非活跃连接
        bool _enable_inactive_release;//是否启动了非活跃连接超时销毁的判断标志
        EventLoop _baseloop;    //这是主线程的EventLoop对象，负责监听事件的处理
        Acceptor _acceptor;    //这是监听套接字的管理对象
        LoopThreadPool _pool;   //这是从属EventLoop线程池
        //保存管理所有连接对应的shared_ptr对象
        std::unordered_map<uint64_t, PtrConnection> _conns;

        using ConnectedCallback = std::function<void(const PtrConnection&)>;
        using MessageCallback = std::function<void(const PtrConnection&, Buffer *)>;
        using ClosedCallback = std::function<void(const PtrConnection&)>;
        using AnyEventCallback = std::function<void(const PtrConnection&)>;
        using Functor = std::function<void()>;
        ConnectedCallback _connected_callback;
        MessageCallback _message_callback;
        ClosedCallback _closed_callback;
        AnyEventCallback _event_callback;
    private:
        void RunAfterInLoop(const Functor &task, int delay) {
            _next_id++;
            _baseloop.TimerAdd(_next_id, delay, task);
        }
        //为新连接构造一个Connection进行管理
        void NewConnection(int fd) {
            _next_id++;
            PtrConnection conn(new Connection(_pool.NextLoop(), _next_id, fd));
            conn->SetMessageCallback(_message_callback);
            conn->SetClosedCallback(_closed_callback);
            conn->SetConnectedCallback(_connected_callback);
            conn->SetAnyEventCallback(_event_callback);
            conn->SetSrvClosedCallback(std::bind(&TcpServer::RemoveConnection, this, std::placeholders::_1));
            //启动非活跃超时销毁
            if (_enable_inactive_release) conn->EnableInactiveRelease(_timeout);
            conn->Established();//就绪初始化
            _conns.insert(std::make_pair(_next_id, conn));
```

```
41          }
42          void RemoveConnectionInLoop(const PtrConnection &conn) {
43              int id = conn->Id();
44              auto it = _conns.find(id);
45              if (it != _conns.end()) {
46                  _conns.erase(it);
47              }
48          }
49          //从管理Connection的_conns中移除连接信息
50          void RemoveConnection(const PtrConnection &conn) {
51              _baseloop.RunInLoop(std::bind(&TcpServer::RemoveConnectionInLoop,
    this, conn));
52          }
53      public:
54          TcpServer(int port):
55              _port(port),
56              _next_id(0),
57              _enable_inactive_release(false),
58              _acceptor(&_baseloop, port),
59              _pool(&_baseloop) {
60              _acceptor.SetAcceptCallback(std::bind(&TcpServer::NewConnection,
    this, std::placeholders::_1));
61              _acceptor.Listen();//将监听套接字挂到baseloop上
62          }
63          void SetThreadCount(int count) { return _pool.SetThreadCount(count); }
64          void SetConnectedCallback(const ConnectedCallback&cb) {
    _connected_callback = cb; }
65          void SetMessageCallback(const MessageCallback&cb) { _message_callback
    = cb; }
66          void SetClosedCallback(const ClosedCallback&cb) { _closed_callback =
    cb; }
67          void SetAnyEventCallback(const AnyEventCallback&cb) { _event_callback
    = cb; }
68          void EnableInactiveRelease(int timeout) { _timeout = timeout;
    _enable_inactive_release = true; }
69          //用于添加一个定时任务
70          void RunAfter(const Functor &task, int delay) {
71              _baseloop.RunInLoop(std::bind(&TcpServer::RunAfterInLoop, this,
    task, delay));
72          }
73          void Start() { _pool.Create();  _baseloop.Start(); }
74  };
```

基于TcpServer实现回显服务器：

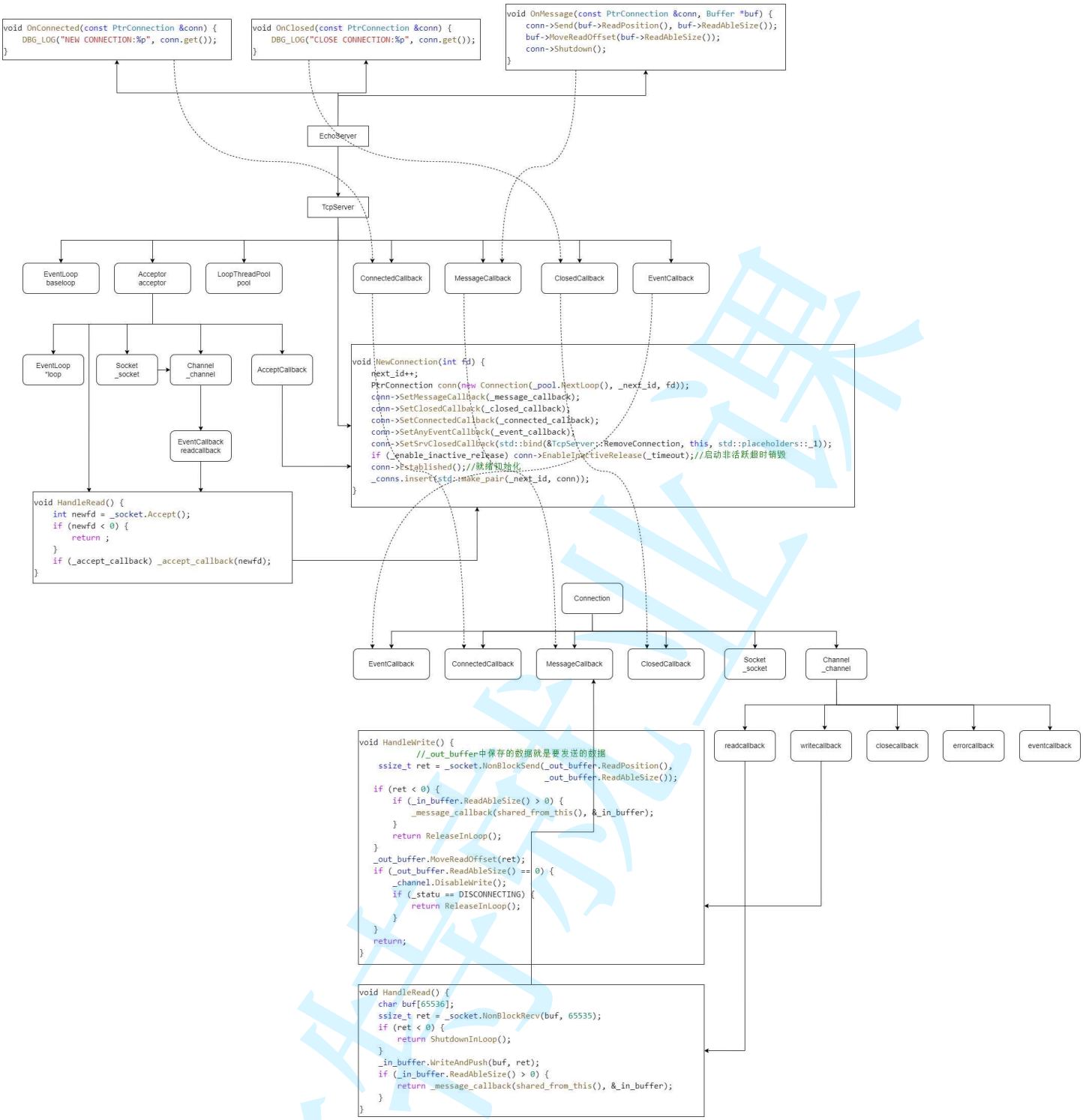
```cpp
#include "../server.hpp"

class EchoServer {
    private:
        TcpServer _server;
    private:
        void OnConnected(const PtrConnection &conn) {
            DBG_LOG("NEW CONNECTION:%p", conn.get());
        }
        void OnClosed(const PtrConnection &conn) {
            DBG_LOG("CLOSE CONNECTION:%p", conn.get());
        }
        void OnMessage(const PtrConnection &conn, Buffer *buf) {
            conn->Send(buf->ReadPosition(), buf->ReadAbleSize());
            buf->MoveReadOffset(buf->ReadAbleSize());
            conn->Shutdown();
        }
    public:
        EchoServer(int port):_server(port) {
            _server.SetThreadCount(2);
            _server.EnableInactiveRelease(10);
            _server.SetClosedCallback(std::bind(&EchoServer::OnClosed, this, std::placeholders::_1));
            _server.SetConnectedCallback(std::bind(&EchoServer::OnConnected, this, std::placeholders::_1));
            _server.SetMessageCallback(std::bind(&EchoServer::OnMessage, this, std::placeholders::_1, std::placeholders::_2));
        }
        void Start() { _server.Start(); }
};
```

```cpp
#include "echo.hpp"

int main()
{
    EchoServer server(8500);
    server.Start();
    return 0;
}
```

EchoServer模块关系图：




## HTTP协议支持模块实现：

## Util实用工具类实现：

```
1
2 std::unordered_map<int, std::string> _statu_msg = {
3     {100,  "Continue"},
4     {101,  "Switching Protocol"},
5     {102,  "Processing"},
6     {103,  "Early Hints"},
7     {200,  "OK"},
8     {201,  "Created"},
```

```
    {202,  "Accepted"},
    {203,  "Non-Authoritative Information"},
    {204,  "No Content"},
    {205,  "Reset Content"},
    {206,  "Partial Content"},
    {207,  "Multi-Status"},
    {208,  "Already Reported"},
    {226,  "IM Used"},
    {300,  "Multiple Choice"},
    {301,  "Moved Permanently"},
    {302,  "Found"},
    {303,  "See Other"},
    {304,  "Not Modified"},
    {305,  "Use Proxy"},
    {306,  "unused"},
    {307,  "Temporary Redirect"},
    {308,  "Permanent Redirect"},
    {400,  "Bad Request"},
    {401,  "Unauthorized"},
    {402,  "Payment Required"},
    {403,  "Forbidden"},
    {404,  "Not Found"},
    {405,  "Method Not Allowed"},
    {406,  "Not Acceptable"},
    {407,  "Proxy Authentication Required"},
    {408,  "Request Timeout"},
    {409,  "Conflict"},
    {410,  "Gone"},
    {411,  "Length Required"},
    {412,  "Precondition Failed"},
    {413,  "Payload Too Large"},
    {414,  "URI Too Long"},
    {415,  "Unsupported Media Type"},
    {416,  "Range Not Satisfiable"},
    {417,  "Expectation Failed"},
    {418,  "I'm a teapot"},
    {421,  "Misdirected Request"},
    {422,  "Unprocessable Entity"},
    {423,  "Locked"},
    {424,  "Failed Dependency"},
    {425,  "Too Early"},
    {426,  "Upgrade Required"},
    {428,  "Precondition Required"},
    {429,  "Too Many Requests"},
    {431,  "Request Header Fields Too Large"},
    {451,  "Unavailable For Legal Reasons"},
    {501,  "Not Implemented"},
```

```
    {502,  "Bad Gateway"},
    {503,  "Service Unavailable"},
    {504,  "Gateway Timeout"},
    {505,  "HTTP Version Not Supported"},
    {506,  "Variant Also Negotiates"},
    {507,  "Insufficient Storage"},
    {508,  "Loop Detected"},
    {510,  "Not Extended"},
    {511,  "Network Authentication Required"}
};

std::unordered_map<std::string, std::string> _mime_msg = {
    {".aac",        "audio/aac"},
    {".abw",        "application/x-abiword"},
    {".arc",        "application/x-freearc"},
    {".avi",        "video/x-msvideo"},
    {".azw",        "application/vnd.amazon.ebook"},
    {".bin",        "application/octet-stream"},
    {".bmp",        "image/bmp"},
    {".bz",         "application/x-bzip"},
    {".bz2",        "application/x-bzip2"},
    {".csh",        "application/x-csh"},
    {".css",        "text/css"},
    {".csv",        "text/csv"},
    {".doc",        "application/msword"},
    {".docx",       "application/vnd.openxmlformats-
officedocument.wordprocessingml.document"},
    {".eot",        "application/vnd.ms-fontobject"},
    {".epub",       "application/epub+zip"},
    {".gif",        "image/gif"},
    {".htm",        "text/html"},
    {".html",       "text/html"},
    {".ico",        "image/vnd.microsoft.icon"},
    {".ics",        "text/calendar"},
    {".jar",        "application/java-archive"},
    {".jpeg",       "image/jpeg"},
    {".jpg",        "image/jpeg"},
    {".js",         "text/javascript"},
    {".json",       "application/json"},
    {".jsonld",     "application/ld+json"},
    {".mid",        "audio/midi"},
    {".midi",       "audio/x-midi"},
    {".mjs",        "text/javascript"},
    {".mp3",        "audio/mpeg"},
    {".mpeg",       "video/mpeg"},
    {".mpkg",       "application/vnd.apple.installer+xml"},
    {".odp",        "application/vnd.oasis.opendocument.presentation"},
```

```
102     {".ods",        "application/vnd.oasis.opendocument.spreadsheet"},
103     {".odt",        "application/vnd.oasis.opendocument.text"},
104     {".oga",        "audio/ogg"},
105     {".ogv",        "video/ogg"},
106     {".ogx",        "application/ogg"},
107     {".otf",        "font/otf"},
108     {".png",        "image/png"},
109     {".pdf",        "application/pdf"},
110     {".ppt",        "application/vnd.ms-powerpoint"},
111     {".pptx",       "application/vnd.openxmlformats-
    officedocument.presentationml.presentation"},
112     {".rar",        "application/x-rar-compressed"},
113     {".rtf",        "application/rtf"},
114     {".sh",         "application/x-sh"},
115     {".svg",        "image/svg+xml"},
116     {".swf",        "application/x-shockwave-flash"},
117     {".tar",        "application/x-tar"},
118     {".tif",        "image/tiff"},
119     {".tiff",       "image/tiff"},
120     {".ttf",        "font/ttf"},
121     {".txt",        "text/plain"},
122     {".vsd",        "application/vnd.visio"},
123     {".wav",        "audio/wav"},
124     {".weba",       "audio/webm"},
125     {".webm",       "video/webm"},
126     {".webp",       "image/webp"},
127     {".woff",       "font/woff"},
128     {".woff2",      "font/woff2"},
129     {".xhtml",      "application/xhtml+xml"},
130     {".xls",        "application/vnd.ms-excel"},
131     {".xlsx",       "application/vnd.openxmlformats-
    officedocument.spreadsheetml.sheet"},
132     {".xml",        "application/xml"},
133     {".xul",        "application/vnd.mozilla.xul+xml"},
134     {".zip",        "application/zip"},
135     {".3gp",        "video/3gpp"},
136     {".3g2",        "video/3gpp2"},
137     {".7z",         "application/x-7z-compressed"}
138 };
139
140 class Util {
141     public:
142         //字符串分割函数,将src字符串按照sep字符进行分割，得到的各个字串放到arry中，最终
    返回字串的数量
143         static size_t Split(const std::string &src, const std::string &sep,
    std::vector<std::string> *arry) {
144             size_t offset = 0;
```

```
145             // 有10个字符，offset是查找的起始位置，范围应该是0~9，offset==10就代表已经越界了
146             while(offset < src.size()) {
147                 size_t pos = src.find(sep, offset);//在src字符串偏移量offset处，开始向后查找sep字符/字串，返回查找到的位置
148                 if (pos == std::string::npos) {//没有找到特定的字符
149                     //将剩余的部分当作一个字串，放入arry中
150                     if(pos == src.size()) break;
151                     arry->push_back(src.substr(offset));
152                     return arry->size();
153                 }
154                 if (pos == offset) {
155                     offset = pos + sep.size();
156                     continue;//当前字串是一个空的，没有内容
157                 }
158                 arry->push_back(src.substr(offset, pos - offset));
159                 offset = pos + sep.size();
160             }
161             return arry->size();
162         }
163         //读取文件的所有内容，将读取的内容放到一个Buffer中
164         static bool ReadFile(const std::string &filename, std::string *buf) {
165             std::ifstream ifs(filename, std::ios::binary);
166             if (ifs.is_open() == false) {
167                 printf("OPEN %s FILE FAILED!!", filename.c_str());
168                 return false;
169             }
170             size_t fsize = 0;
171             ifs.seekg(0, ifs.end);//跳转读写位置到末尾
172             fsize = ifs.tellg();  //获取当前读写位置相对于起始位置的偏移量，从末尾偏移刚好就是文件大小
173             ifs.seekg(0, ifs.beg);//跳转到起始位置
174             buf->resize(fsize); //开辟文件大小的空间
175             ifs.read(&(*buf)[0], fsize);
176             if (ifs.good() == false) {
177                 printf("READ %s FILE FAILED!!", filename.c_str());
178                 ifs.close();
179                 return false;
180             }
181             ifs.close();
182             return true;
183         }
184         //向文件写入数据
185         static bool WriteFile(const std::string &filename, const std::string &buf) {
186             std::ofstream ofs(filename, std::ios::binary | std::ios::trunc);
187             if (ofs.is_open() == false) {
```

```
188                 printf("OPEN %s FILE FAILED!!", filename.c_str());
189                 return false;
190             }
191             ofs.write(buf.c_str(), buf.size());
192             if (ofs.good() == false) {
193                 ERR_LOG("WRITE %s FILE FAILED!", filename.c_str());
194                 ofs.close();
195                 return false;
196             }
197             ofs.close();
198             return true;
199         }
200         //URL编码，避免URL中资源路径与查询字符串中的特殊字符与HTTP请求中特殊字符产生歧义
201         //编码格式：将特殊字符的ascii值，转换为两个16进制字符，前缀%   C++ -> C%2B%2B
202         //  不编码的特殊字符： RFC3986文档规定 . - _ ~ 字母，数字属于绝对不编码字符
203         //RFC3986文档规定，编码格式 %HH
204         //W3C标准中规定，查询字符串中的空格，需要编码为+， 解码则是+转空格
205         static std::string UrlEncode(const std::string url, bool
    convert_space_to_plus) {
206             std::string res;
207             for (auto &c : url) {
208                 if (c == '.' || c == '-' || c == '_' || c == '~' ||
    isalnum(c)) {
209                     res += c;
210                     continue;
211                 }
212                 if (c == ' ' && convert_space_to_plus == true) {
213                     res += '+';
214                     continue;
215                 }
216                 //剩下的字符都是需要编码成为 %HH 格式
217                 char tmp[4] = {0};
218                 //snprintf 与 printf比较类似，都是格式化字符串，只不过一个是打印，一个
    是放到一块空间中
219                 snprintf(tmp, 4, "%%%02X", c);
220                 res += tmp;
221             }
222             return res;
223         }
224         static char HEXTOI(char c) {
225             if (c >= '0' && c <= '9') {
226                 return c - '0';
227             }else if (c >= 'a' && c <= 'z') {
228                 return c - 'a' + 10;
229             }else if (c >= 'A' && c <= 'Z') {
230                 return c - 'A' + 10;
231             }
```

```
232             return -1;
233         }
234         static std::string UrlDecode(const std::string url, bool
    convert_plus_to_space) {
235             //遇到了%，则将紧随其后的2个字符，转换为数字，第一个数字左移4位，然后加上第
    二个数字  + -> 2b  %2b->2 << 4 + 11
236             std::string res;
237             for (int i = 0; i < url.size(); i++) {
238                 if (url[i] == '+' && convert_plus_to_space == true) {
239                     res += ' ';
240                     continue;
241                 }
242                 if (url[i] == '%' && (i + 2) < url.size()) {
243                     char v1 = HEXTOI(url[i + 1]);
244                     char v2 = HEXTOI(url[i + 2]);
245                     char v = v1 * 16 + v2;
246                     res += v;
247                     i += 2;
248                     continue;
249                 }
250                 res += url[i];
251             }
252             return res;
253         }
254         //响应状态码的描述信息获取
255         static std::string StatuDesc(int statu) {
256 
257             auto it = _statu_msg.find(statu);
258             if (it != _statu_msg.end()) {
259                 return it->second;
260             }
261             return "Unknow";
262         }
263         //根据文件后缀名获取文件mime
264         static std::string ExtMime(const std::string &filename) {
265 
266             // a.b.txt  先获取文件扩展名
267             size_t pos = filename.find_last_of('.');
268             if (pos == std::string::npos) {
269                 return "application/octet-stream";
270             }
271             //根据扩展名，获取mime
272             std::string ext = filename.substr(pos);
273             auto it = _mime_msg.find(ext);
274             if (it == _mime_msg.end()) {
275                 return "application/octet-stream";
276             }
```

```
277             return it->second;
278         }
279         //判断一个文件是否是一个目录
280         static bool IsDirectory(const std::string &filename) {
281             struct stat st;
282             int ret = stat(filename.c_str(), &st);
283             if (ret < 0) {
284                 return false;
285             }
286             return S_ISDIR(st.st_mode);
287         }
288         //判断一个文件是否是一个普通文件
289         static bool IsRegular(const std::string &filename) {
290             struct stat st;
291             int ret = stat(filename.c_str(), &st);
292             if (ret < 0) {
293                 return false;
294             }
295             return S_ISREG(st.st_mode);
296         }
297         //http请求的资源路径有效性判断
298         // /index.html  --- 前边的/叫做相对根目录  映射的是某个服务器上的子目录
299         // 想表达的意思就是，客户端只能请求相对根目录中的资源，其他地方的资源都不予理会
300         // /../login, 这个路径中的..会让路径的查找跑到相对根目录之外，这是不合理的，不
    安全的
301         static bool ValidPath(const std::string &path) {
302             //思想：按照/进行路径分割，根据有多少子目录，计算目录深度，有多少层，深度不能
    小于0
303             std::vector<std::string> subdir;
304             Split(path, "/", &subdir);
305             int level = 0;
306             for (auto &dir : subdir) {
307                 if (dir == "..") {
308                     level--; //任意一层走出相对根目录，就认为有问题
309                     if (level < 0) return false;
310                     continue;
311                 }
312                 level++;
313             }
314             return true;
315         }
316 };
```

HttpRequest请求类实现：

```cpp

class HttpRequest {
    public:
        std::string _method;      //请求方法
        std::string _path;        //资源路径
        std::string _version;     //协议版本
        std::string _body;        //请求正文
        std::smatch _matches;     //资源路径的正则提取数据
        std::unordered_map<std::string, std::string> _headers;   //头部字段
        std::unordered_map<std::string, std::string> _params;    //查询字符串
    public:
        HttpRequest():_version("HTTP/1.1") {}
        void ReSet() {
            _method.clear();
            _path.clear();
            _version = "HTTP/1.1";
            _body.clear();
            std::smatch match;
            _matches.swap(match);
            _headers.clear();
            _params.clear();
        }
        //插入头部字段
        void SetHeader(const std::string &key, const std::string &val) {
            _headers.insert(std::make_pair(key, val));
        }
        //判断是否存在指定头部字段
        bool HasHeader(const std::string &key) const {
            auto it = _headers.find(key);
            if (it == _headers.end()) {
                return false;
            }
            return true;
        }
        //获取指定头部字段的值
        std::string GetHeader(const std::string &key) const {
            auto it = _headers.find(key);
            if (it == _headers.end()) {
                return "";
            }
            return it->second;
        }
        //插入查询字符串
        void SetParam(const std::string &key, const std::string &val) {
            _params.insert(std::make_pair(key, val));
        }
        //判断是否有某个指定的查询字符串
```

```
48          bool HasParam(const std::string &key) const {
49              auto it = _params.find(key);
50              if (it == _params.end()) {
51                  return false;
52              }
53              return true;
54          }
55          //获取指定的查询字符串
56          std::string GetParam(const std::string &key) const {
57              auto it = _params.find(key);
58              if (it == _params.end()) {
59                  return "";
60              }
61              return it->second;
62          }
63          //获取正文长度
64          size_t ContentLength() const {
65              // Content-Length: 1234\r\n
66              bool ret = HasHeader("Content-Length");
67              if (ret == false) {
68                  return 0;
69              }
70              std::string clen = GetHeader("Content-Length");
71              return std::stol(clen);
72          }
73          //判断是否是短链接
74          bool Close() const {
75              // 没有Connection字段，或者有Connection但是值是close，则都是短链接，否则就是长连接
76              if (HasHeader("Connection") == true && GetHeader("Connection") == "keep-alive") {
77                  return false;
78              }
79              return true;
80          }
81  };
82
```

HttpResponse响应类实现：

```

1
2   class HttpResponse {
3       public:
4           int _statu;
5           bool _redirect_flag;
```

```
6           std::string _body;
7           std::string _redirect_url;
8           std::unordered_map<std::string, std::string> _headers;
9       public:
10          HttpResponse():_redirect_flag(false), _statu(200) {}
11          HttpResponse(int statu):_redirect_flag(false), _statu(statu) {}
12          void ReSet() {
13              _statu = 200;
14              _redirect_flag = false;
15              _body.clear();
16              _redirect_url.clear();
17              _headers.clear();
18          }
19          //插入头部字段
20          void SetHeader(const std::string &key, const std::string &val) {
21              _headers.insert(std::make_pair(key, val));
22          }
23          //判断是否存在指定头部字段
24          bool HasHeader(const std::string &key) {
25              auto it = _headers.find(key);
26              if (it == _headers.end()) {
27                  return false;
28              }
29              return true;
30          }
31          //获取指定头部字段的值
32          std::string GetHeader(const std::string &key) {
33              auto it = _headers.find(key);
34              if (it == _headers.end()) {
35                  return "";
36              }
37              return it->second;
38          }
39          void SetContent(const std::string &body,  const std::string &type =
    "text/html") {
40              _body = body;
41              SetHeader("Content-Type", type);
42          }
43          void SetRedirect(const std::string &url, int statu = 302) {
44              _statu = statu;
45              _redirect_flag = true;
46              _redirect_url = url;
47          }
48          //判断是否是短链接
49          bool Close() {
50              // 没有Connection字段，或者有Connection但是值是close，则都是短链接，否则
    就是长连接
```

```
            if (HasHeader("Connection") == true && GetHeader("Connection") == "keep-alive") {
                return false;
            }
            return true;
        }
};

```

HttpContext上下文类实现：

```

typedef enum {
    RECV_HTTP_ERROR,
    RECV_HTTP_LINE,
    RECV_HTTP_HEAD,
    RECV_HTTP_BODY,
    RECV_HTTP_OVER
}HttpRecvStatu;

#define MAX_LINE 8192
class HttpContext {
    private:
        int _resp_statu; //响应状态码
        HttpRecvStatu _recv_statu; //当前接收及解析的阶段状态
        HttpRequest _request;  //已经解析得到的请求信息
    private:
        bool ParseHttpLine(const std::string &line) {
            std::smatch matches;
            std::regex e("(GET|HEAD|POST|PUT|DELETE) ([^?]*)(?:\\?(.*))? (HTTP/1\\.[01])(?:\n|\r\n)?", std::regex::icase);
            bool ret = std::regex_match(line, matches, e);
            if (ret == false) {
                _recv_statu = RECV_HTTP_ERROR;
                _resp_statu = 400;//BAD REQUEST
                return false;
            }
            //0 : GET /bitejiuyeke/login?user=xiaoming&pass=123123 HTTP/1.1
            //1 : GET
            //2 : /bitejiuyeke/login
            //3 : user=xiaoming&pass=123123
            //4 : HTTP/1.1
            //请求方法的获取
            _request._method = matches[1];

```

```
33              std::transform(_request._method.begin(), _request._method.end(),
    _request._method.begin(), ::toupper);
34              //资源路径的获取，需要进行URL解码操作，但是不需要+转空格
35              _request._path = Util::UrlDecode(matches[2], false);
36              //协议版本的获取
37              _request._version = matches[4];
38              //查询字符串的获取与处理
39              std::vector<std::string> query_string_arry;
40              std::string query_string = matches[3];
41              //查询字符串的格式 key=val&key=val....., 先以 & 符号进行分割，得到各个字
    串
42              Util::Split(query_string, "&", &query_string_arry);
43              //针对各个字串，以 = 符号进行分割，得到key 和val， 得到之后也需要进行URL解
    码
44              for (auto &str : query_string_arry) {
45                  size_t pos = str.find("=");
46                  if (pos == std::string::npos) {
47                      _recv_statu = RECV_HTTP_ERROR;
48                      _resp_statu = 400;//BAD REQUEST
49                      return false;
50                  }
51                  std::string key = Util::UrlDecode(str.substr(0, pos), true);
52                  std::string val = Util::UrlDecode(str.substr(pos + 1), true);
53                  _request.SetParam(key, val);
54              }
55              return true;
56          }
57          bool RecvHttpLine(Buffer *buf) {
58              if (_recv_statu != RECV_HTTP_LINE) return false;
59              //1. 获取一行数据，带有末尾的换行
60              std::string line = buf->GetLineAndPop();
61              //2. 需要考虑的一些要素：缓冲区中的数据不足一行， 获取的一行数据超大
62              if (line.size() == 0) {
63                  //缓冲区中的数据不足一行，则需要判断缓冲区的可读数据长度，如果很长了都不
    足一行，这是有问题的
64                  if (buf->ReadAbleSize() > MAX_LINE) {
65                      _recv_statu = RECV_HTTP_ERROR;
66                      _resp_statu = 414;//URI TOO LONG
67                      return false;
68                  }
69                  //缓冲区中数据不足一行，但是也不多，就等等新数据的到来
70                  return true;
71              }
72              if (line.size() > MAX_LINE) {
73                  _recv_statu = RECV_HTTP_ERROR;
74                  _resp_statu = 414;//URI TOO LONG
75                  return false;
```

```
76              }
77              bool ret = ParseHttpLine(line);
78              if (ret == false) {
79                  return false;
80              }
81              //首行处理完毕，进入头部获取阶段
82              _recv_statu = RECV_HTTP_HEAD;
83              return true;
84          }
85          bool RecvHttpHead(Buffer *buf) {
86              if (_recv_statu != RECV_HTTP_HEAD) return false;
87              //一行一行取出数据，直到遇到空行为止， 头部的格式 key: val\r\nkey: val\r\n....
88              while(1){
89                  std::string line = buf->GetLineAndPop();
90                  //2. 需要考虑的一些要素：缓冲区中的数据不足一行， 获取的一行数据超大
91                  if (line.size() == 0) {
92                      //缓冲区中的数据不足一行，则需要判断缓冲区的可读数据长度，如果很长了都不足一行，这是有问题的
93                      if (buf->ReadAbleSize() > MAX_LINE) {
94                          _recv_statu = RECV_HTTP_ERROR;
95                          _resp_statu = 414;//URI TOO LONG
96                          return false;
97                      }
98                      //缓冲区中数据不足一行，但是也不多，就等等新数据的到来
99                      return true;
100                 }
101                 if (line.size() > MAX_LINE) {
102                     _recv_statu = RECV_HTTP_ERROR;
103                     _resp_statu = 414;//URI TOO LONG
104                     return false;
105                 }
106                 if (line == "\n" || line == "\r\n") {
107                     break;
108                 }
109                 bool ret = ParseHttpHead(line);
110                 if (ret == false) {
111                     return false;
112                 }
113             }
114             //头部处理完毕，进入正文获取阶段
115             _recv_statu = RECV_HTTP_BODY;
116             return true;
117         }
118         bool ParseHttpHead(std::string &line) {
119             //key: val\r\nkey: val\r\n....
120             if (line.back() == '\n') line.pop_back();//末尾是换行则去掉换行字符
```

```
121             if (line.back() == '\r') line.pop_back();//末尾是回车则去掉回车字符
122             size_t pos = line.find(": ");
123             if (pos == std::string::npos) {
124                 _recv_statu = RECV_HTTP_ERROR;
125                 _resp_statu = 400;//
126                 return false;
127             }
128             std::string key = line.substr(0, pos);
129             std::string val = line.substr(pos + 2);
130             _request.SetHeader(key, val);
131             return true;
132         }
133         bool RecvHttpBody(Buffer *buf) {
134             if (_recv_statu != RECV_HTTP_BODY) return false;
135             //1. 获取正文长度
136             size_t content_length = _request.ContentLength();
137             if (content_length == 0) {
138                 //没有正文，则请求接收解析完毕
139                 _recv_statu = RECV_HTTP_OVER;
140                 return true;
141             }
142             //2. 当前已经接收了多少正文,其实就是往  _request._body 中放了多少数据了
143             size_t real_len = content_length - _request._body.size();//实际还需要
    接收的正文长度
144             //3. 接收正文放到body中，但是也要考虑当前缓冲区中的数据，是否是全部的正文
145             //  3.1 缓冲区中数据，包含了当前请求的所有正文，则取出所需的数据
146             if (buf->ReadAbleSize() >= real_len) {
147                 _request._body.append(buf->ReadPosition(), real_len);
148                 buf->MoveReadOffset(real_len);
149                 _recv_statu = RECV_HTTP_OVER;
150                 return true;
151             }
152             //  3.2 缓冲区中数据，无法满足当前正文的需要，数据不足，取出数据，然后等待新
    数据到来
153             _request._body.append(buf->ReadPosition(), buf->ReadAbleSize());
154             buf->MoveReadOffset(buf->ReadAbleSize());
155             return true;
156         }
157     public:
158         HttpContext():_resp_statu(200), _recv_statu(RECV_HTTP_LINE) {}
159         void ReSet() {
160             _resp_statu = 200;
161             _recv_statu = RECV_HTTP_LINE;
162             _request.ReSet();
163         }
164         int RespStatu() { return _resp_statu; }
165         HttpRecvStatu RecvStatu() { return _recv_statu; }
```

```
166         HttpRequest &Request() { return _request; }
167         //接收并解析HTTP请求
168         void RecvHttpRequest(Buffer *buf) {
169             //不同的状态，做不同的事情，但是这里不要break， 因为处理完请求行后，应该立即处理头部，而不是退出等新数据
170             switch(_recv_statu) {
171                 case RECV_HTTP_LINE: RecvHttpLine(buf);
172                 case RECV_HTTP_HEAD: RecvHttpHead(buf);
173                 case RECV_HTTP_BODY: RecvHttpBody(buf);
174             }
175             return;
176         }
177 };
```

HttpServer类实现：

```
1
2  class HttpServer {
3      private:
4          using Handler = std::function<void(const HttpRequest &, HttpResponse *)>;
5          using Handlers = std::vector<std::pair<std::regex, Handler>>;
6          Handlers _get_route;
7          Handlers _post_route;
8          Handlers _put_route;
9          Handlers _delete_route;
10         std::string _basedir; //静态资源根目录
11         TcpServer _server;
12     private:
13         void ErrorHandler(const HttpRequest &req, HttpResponse *rsp) {
14             //1. 组织一个错误展示页面
15             std::string body;
16             body += "<html>";
17             body += "<head>";
18             body += "<meta http-equiv='Content-Type' content='text/html;charset=utf-8'>";
19             body += "</head>";
20             body += "<body>";
21             body += "<h1>";
22             body += std::to_string(rsp->_statu);
23             body += " ";
24             body += Util::StatuDesc(rsp->_statu);
25             body += "</h1>";
26             body += "</body>";
27             body += "</html>";
```

```
28              //2. 将页面数据，当作响应正文，放入rsp中
29              rsp->SetContent(body, "text/html");
30          }
31          //将HttpResponse中的要素按照http协议格式进行组织，发送
32          void WriteReponse(const PtrConnection &conn, const HttpRequest &req,
    HttpResponse &rsp) {
33              //1. 先完善头部字段
34              if (req.Close() == true) {
35                  rsp.SetHeader("Connection", "close");
36              }else {
37                  rsp.SetHeader("Connection", "keep-alive");
38              }
39              if (rsp._body.empty() == false && rsp.HasHeader("Content-Length")
    == false) {
40                  rsp.SetHeader("Content-Length",
    std::to_string(rsp._body.size()));
41              }
42              if (rsp._body.empty() == false && rsp.HasHeader("Content-Type") ==
    false) {
43                  rsp.SetHeader("Content-Type", "application/octet-stream");
44              }
45              if (rsp._redirect_flag == true) {
46                  rsp.SetHeader("Location", rsp._redirect_url);
47              }
48              //2. 将rsp中的要素，按照http协议格式进行组织
49              std::stringstream rsp_str;
50              rsp_str << req._version << " " << std::to_string(rsp._statu) << " "
     << Util::StatuDesc(rsp._statu) << "\r\n";
51              for (auto &head : rsp._headers) {
52                  rsp_str << head.first << ": " << head.second << "\r\n";
53              }
54              rsp_str << "\r\n";
55              rsp_str << rsp._body;
56              //3. 发送数据
57              conn->Send(rsp_str.str().c_str(), rsp_str.str().size());
58          }
59          bool IsFileHandler(const HttpRequest &req) {
60              // 1. 必须设置了静态资源根目录
61              if (_basedir.empty()) {
62                  return false;
63              }
64              // 2. 请求方法，必须是GET / HEAD请求方法
65              if (req._method != "GET" && req._method != "HEAD") {
66                  return false;
67              }
68              // 3. 请求的资源路径必须是一个合法路径
69              if (Util::ValidPath(req._path) == false) {
```

```
70                  return false;
71              }
72              // 4. 请求的资源必须存在,且是一个普通文件
73              //    有一种请求比较特殊 -- 目录: /, /image/, 这种情况给后边默认追加一个 index.html
74              // index.html    /image/a.png
75              // 不要忘了前缀的相对根目录,也就是将请求路径转换为实际存在的路径  /image/a.png  ->  ./wwwroot/image/a.png
76              std::string req_path = _basedir + req._path;//为了避免直接修改请求的资源路径，因此定义一个临时对象
77              if (req._path.back() == '/')  {
78                  req_path += "index.html";
79              }
80              if (Util::IsRegular(req_path) == false) {
81                  return false;
82              }
83              return true;
84          }
85          //静态资源的请求处理 --- 将静态资源文件的数据读取出来，放到rsp的_body中，并设置 mime
86          void FileHandler(const HttpRequest &req, HttpResponse *rsp) {
87              std::string req_path = _basedir + req._path;
88              if (req._path.back() == '/')  {
89                  req_path += "index.html";
90              }
91              bool ret = Util::ReadFile(req_path, &rsp->_body);
92              if (ret == false) {
93                  return;
94              }
95              std::string mime = Util::ExtMime(req_path);
96              rsp->SetHeader("Content-Type", mime);
97              return;
98          }
99          //功能性请求的分类处理
100         void Dispatcher(HttpRequest &req, HttpResponse *rsp, Handlers &handlers) {
101             //在对应请求方法的路由表中，查找是否含有对应资源请求的处理函数，有则调用，没有则发挥404
102             //思想：路由表存储的时键值对 -- 正则表达式 & 处理函数
103             //使用正则表达式，对请求的资源路径进行正则匹配，匹配成功就使用对应函数进行处理
104             //  /numbers/(\d+)       /numbers/12345
105             for (auto &handler : handlers) {
106                 const std::regex &re = handler.first;
107                 const Handler &functor = handler.second;
108                 bool ret = std::regex_match(req._path, req._matches, re);
109                 if (ret == false) {
```

```
110                     continue;
111                 }
112                 return functor(req, rsp);//传入请求信息，和空的rsp，执行处理函数
113             }
114             rsp->_statu = 404;
115         }
116         void Route(HttpRequest &req, HttpResponse *rsp) {
117             //1. 对请求进行分辨，是一个静态资源请求，还是一个功能性请求
118             //    静态资源请求，则进行静态资源的处理
119             //    功能性请求，则需要通过几个请求路由表来确定是否有处理函数
120             //    既不是静态资源请求，也没有设置对应的功能性请求处理函数，就返回405
121             if (IsFileHandler(req) == true) {
122                 //是一个静态资源请求，则进行静态资源请求的处理
123                 return FileHandler(req, rsp);
124             }
125             if (req._method == "GET" || req._method == "HEAD") {
126                 return Dispatcher(req, rsp, _get_route);
127             }else if (req._method == "POST") {
128                 return Dispatcher(req, rsp, _post_route);
129             }else if (req._method == "PUT") {
130                 return Dispatcher(req, rsp, _put_route);
131             }else if (req._method == "DELETE") {
132                 return Dispatcher(req, rsp, _delete_route);
133             }
134             rsp->_statu = 405;// Method Not Allowed
135             return ;
136         }
137         //设置上下文
138         void OnConnected(const PtrConnection &conn) {
139             conn->SetContext(HttpContext());
140             DBG_LOG("NEW CONNECTION %p", conn.get());
141         }
142         //缓冲区数据解析+处理
143         void OnMessage(const PtrConnection &conn, Buffer *buffer) {
144             while(buffer->ReadAbleSize() > 0){
145                 //1. 获取上下文
146                 HttpContext *context = conn->GetContext()->get<HttpContext>();
147                 //2. 通过上下文对缓冲区数据进行解析，得到HttpRequest对象
148                 //  1. 如果缓冲区的数据解析出错，就直接回复出错响应
149                 //  2. 如果解析正常，且请求已经获取完毕，才开始去进行处理
150                 context->RecvHttpRequest(buffer);
151                 HttpRequest &req = context->Request();
152                 HttpResponse rsp(context->RespStatu());
153                 if (context->RespStatu() >= 400) {
154                     //进行错误响应，关闭连接
155                     ErrorHandler(req, &rsp);//填充一个错误显示页面数据到rsp中
156                     WriteReponse(conn, req, rsp);//组织响应发送给客户端
```

```
157                     context->ReSet();
158                     buffer->MoveReadOffset(buffer->ReadAbleSize());//出错了就把缓
    冲区数据清空
159                     conn->Shutdown();//关闭连接
160                     return;
161                 }
162                 if (context->RecvStatu() != RECV_HTTP_OVER) {
163                     //当前请求还没有接收完整,则退出，等新数据到来再重新继续处理
164                     return;
165                 }
166                 //3. 请求路由 + 业务处理
167                 Route(req, &rsp);
168                 //4. 对HttpResponse进行组织发送
169                 WriteReponse(conn, req, rsp);
170                 //5. 重置上下文
171                 context->ReSet();
172                 //6. 根据长短连接判断是否关闭连接或者继续处理
173                 if (rsp.Close() == true) conn->Shutdown();//短链接则直接关闭
174             }
175             return;
176         }
177     public:
178         HttpServer(int port, int timeout = DEFALT_TIMEOUT):_server(port) {
179             _server.EnableInactiveRelease(timeout);
180             _server.SetConnectedCallback(std::bind(&HttpServer::OnConnected,
    this, std::placeholders::_1));
181             _server.SetMessageCallback(std::bind(&HttpServer::OnMessage, this,
    std::placeholders::_1, std::placeholders::_2));
182         }
183         void SetBaseDir(const std::string &path) {
184             assert(Util::IsDirectory(path) == true);
185             _basedir = path;
186         }
187         /*设置/添加，请求（请求的正则表达）与处理函数的映射关系*/
188         void Get(const std::string &pattern, const Handler &handler) {
189             _get_route.push_back(std::make_pair(std::regex(pattern), handler));
190         }
191         void Post(const std::string &pattern, const Handler &handler) {
192             _post_route.push_back(std::make_pair(std::regex(pattern),
    handler));
193         }
194         void Put(const std::string &pattern, const Handler &handler) {
195             _put_route.push_back(std::make_pair(std::regex(pattern), handler));
196         }
197         void Delete(const std::string &pattern, const Handler &handler) {
198             _delete_route.push_back(std::make_pair(std::regex(pattern),
    handler));
```

```
199         }
200         void SetThreadCount(int count) {
201             _server.SetThreadCount(count);
202         }
203         void Listen() {
204             _server.Start();
205         }
206 };
```

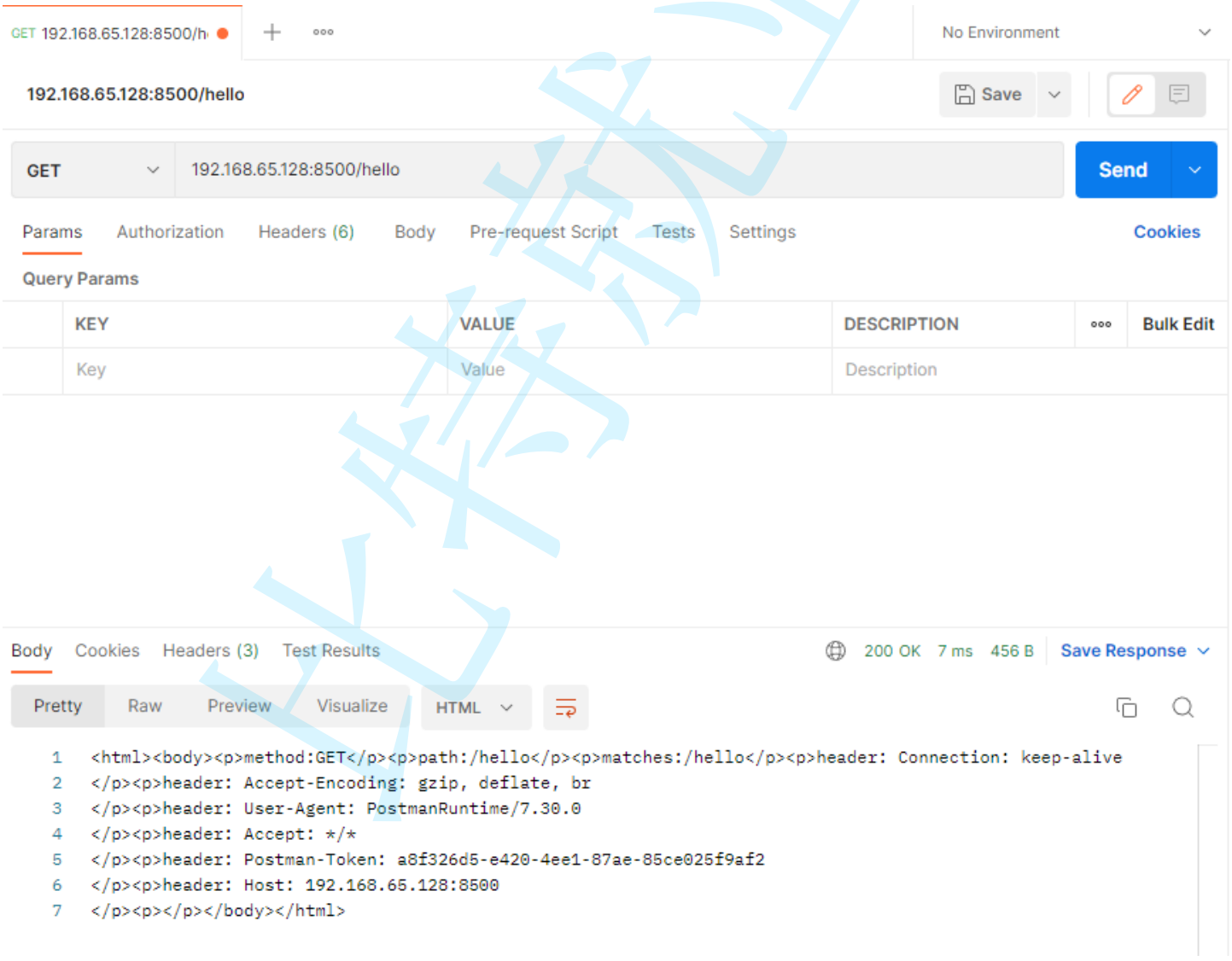
基于HttpServer搭建HTTP服务器：

```
1  #include "http.hpp"
2
3  #define WWWROOT "./wwwroot/"
4
5  std::string RequestStr(const HttpRequest &req) {
6      std::stringstream ss;
7      ss << req._method << " " << req._path << " " << req._version << "\r\n";
8      for (auto &it : req._params) {
9          ss << it.first << ": " << it.second << "\r\n";
10     }
11     for (auto &it : req._headers) {
12         ss << it.first << ": " << it.second << "\r\n";
13     }
14     ss << "\r\n";
15     ss << req._body;
16     return ss.str();
17 }
18 void Hello(const HttpRequest &req, HttpResponse *rsp)
19 {
20     rsp->SetContent(RequestStr(req), "text/plain");
21 }
22 void Login(const HttpRequest &req, HttpResponse *rsp)
23 {
24     rsp->SetContent(RequestStr(req), "text/plain");
25 }
26 void PutFile(const HttpRequest &req, HttpResponse *rsp)
27 {
28     std::string pathname = WWWROOT + req._path;
29     Util::WriteFile(pathname, req._body);
30 }
31 void DelFile(const HttpRequest &req, HttpResponse *rsp)
32 {
33     rsp->SetContent(RequestStr(req), "text/plain");
34 }
```

```cpp
int main()
{
    HttpServer server(8085);
    server.SetThreadCount(3);
    server.SetBaseDir(WWWROOT);//设置静态资源根目录，告诉服务器有静态资源请求到来，需要到哪里去找资源文件
    server.Get("/hello", Hello);
    server.Post("/login", Login);
    server.Put("/1234.txt", PutFile);
    server.Delete("/1234.txt", DelFile);
    server.Listen();
    return 0;
}
```

## 功能测试：

### 使用Postman进行基本功能测试：




### 长连接连续请求测试：

一个连接中每隔3s向服务器发送一个请求，查看是否会收到响应。

预期结果：可以正常进行长连接的通信。

```
/*  注意当前的目录结构为:
├── source
│   ├── echo
│   │   ├── echo.hpp
│   │   ├── main
│   │   ├── main.cc
│   │   └── Makefile
│   ├── http
│   │   ├── http.hpp
│   │   ├── main
│   │   ├── main.cc
│   │   ├── Makefile
│   │   ├── mime
│   │   ├── statu
│   │   └── wwwroot
│   │       └── index.html
│   └── server.hpp
└── test
    ├── client1
    ├── client1.cpp
    ├── Makefile
*/
/*长连接测试1：创建一个客户端持续给服务器发送数据，直到超过超时时间看看是否正常*/
#include "../source/server.hpp"

int main()
{
    Socket cli_sock;
    cli_sock.CreateClient(8085, "127.0.0.1");
    std::string req = "GET /hello HTTP/1.1\r\nConnection: keep-alive\r\nContent-Length: 0\r\n\r\n";
    while(1) {
        assert(cli_sock.Send(req.c_str(), req.size()) != -1);
        char buf[1024] = {0};
        assert(cli_sock.Recv(buf, 1023));
        DBG_LOG("[%s]", buf);
        sleep(3);
    }
    cli_sock.Close();
    return 0;
}

```

```
zwc@139-159-150-152:~/workspace/http-v1/test$ ./client1
[0x7facd1768100 11:34:38 ../source/server.hpp:1153] SIGPIPE INIT
[0x7facd1768100 11:34:38 client1.cpp:13] [HTTP/1.1 200 OK
Content-Length: 66
Connection: keep-alive
Content-Type: text/plain

GET /hello HTTP/1.1
Content-Length: 0
Connection: keep-alive

]
[0x7facd1768100 11:34:41 client1.cpp:13] [HTTP/1.1 200 OK
Content-Length: 66
Connection: keep-alive
Content-Type: text/plain

GET /hello HTTP/1.1
Content-Length: 0
Connection: keep-alive

]
[0x7facd1768100 11:34:44 client1.cpp:13] [HTTP/1.1 200 OK
Content-Length: 66
Connection: keep-alive
Content-Type: text/plain

GET /hello HTTP/1.1
Content-Length: 0
Connection: keep-alive

]
[0x7facd1768100 11:34:47 client1.cpp:13] [HTTP/1.1 200 OK
Content-Length: 66
Connection: keep-alive
Content-Type: text/plain

GET /hello HTTP/1.1
Content-Length: 0
Connection: keep-alive

]
[0x7facd1768100 11:34:50 client1.cpp:13] [HTTP/1.1 200 OK
Content-Length: 66
Connection: keep-alive
```

```
Content-Type: text/plain

GET /hello HTTP/1.1
Content-Length: 0
Connection: keep-alive

]
[0x7facd1768100 11:34:53 client1.cpp:13] [HTTP/1.1 200 OK
Content-Length: 66
Connection: keep-alive
Content-Type: text/plain

GET /hello HTTP/1.1
Content-Length: 0
Connection: keep-alive

]
[0x7facd1768100 11:34:56 client1.cpp:13] [HTTP/1.1 200 OK
Content-Length: 66
Connection: keep-alive
Content-Type: text/plain

GET /hello HTTP/1.1
Content-Length: 0
Connection: keep-alive

]
[0x7facd1768100 11:34:59 client1.cpp:13] [HTTP/1.1 200 OK
Content-Length: 66
Connection: keep-alive
Content-Type: text/plain

GET /hello HTTP/1.1
Content-Length: 0
Connection: keep-alive

]
```

### 超时连接释放测试1：

创建一个客户端，连接上服务器后，不进行消息发送，等待看超时后，连接是否会自动释放（当前默认设置超时时间为10s）。

预期结果：10s后连接被释放。

```
/*超时连接测试1：创建一个客户端，给服务器发送一次数据后或者不发送数据，不动了，查看服务器
```

```
是否会正常的超时关闭连接*/

#include "../source/server.hpp"

int main()
{
    Socket cli_sock;
    cli_sock.CreateClient(8085, "127.0.0.1");
    std::string req = "GET /hello HTTP/1.1\r\nConnection: keep-alive\r\nContent-Length: 0\r\n\r\n";
    while(1) {
        /*
        assert(cli_sock.Send(req.c_str(), req.size()) != -1);
        char buf[1024] = {0};
        assert(cli_sock.Recv(buf, 1023));
        DBG_LOG("[%s]", buf);
        */
        sleep(15);
    }
    cli_sock.Close();
    return 0;
}
```

测试通过，11:41:52的时候连接建立，并接收请求，11:42:01的时候连接被关闭释放。也就是第10s，新建的连接被自动释放了。

```
zwc@139-159-150-152:~/workspace/http-v1/source/http$ ./main
[0x7ffafecec700 11:41:52 http.hpp:780] NEW CONNECTION 0x560b9523b260
[0x7ffaffcef740 11:42:01 ../server.hpp:978] RELEASE CONNECTION:0x560b9523b260
```

### 超时连接释放测试2：

连接服务器，告诉服务器要发送1024字节正文数据给服务器，但是实际上发送数据不足1024字节，然后看服务器处理情况。

预期结果：服务器第一次接收请求不完整，会将后边的请求当作第一次请求的正文进行处理。最终对剩下的数据处理的时候处理出错，关闭连接。

```
/*给服务器发送一个数据，告诉服务器要发送1024字节的数据，但是实际发送的数据不足1024，查看服务器处理结果*/
/*
    1. 如果数据只发送一次，服务器将得不到完整请求，就不会进行业务处理，客户端也就得不到响应，
```

```
        最终超时关闭连接
    2. 连着给服务器发送了多次 小的请求，
        服务器会将后边的请求当作前边请求的正文进行处理，而后便处理的时候有可能就会因为处理错误而关闭连接
*/

#include "../source/server.hpp"

int main()
{
    Socket cli_sock;
    cli_sock.CreateClient(8085, "127.0.0.1");
    std::string req = "GET /hello HTTP/1.1\r\nConnection: keep-alive\r\nContent-Length: 100\r\n\r\nbitejiuyeke";
    while(1) {
        assert(cli_sock.Send(req.c_str(), req.size()) != -1);
        assert(cli_sock.Send(req.c_str(), req.size()) != -1);
        assert(cli_sock.Send(req.c_str(), req.size()) != -1);
        char buf[1024] = {0};
        assert(cli_sock.Recv(buf, 1023));
        DBG_LOG("[%s]", buf);
        sleep(3);
    }
    cli_sock.Close();
    return 0;
}
```

```
zwc@139-159-150-152:~/workspace/http-v1/test$ ./client3
[0x7f14812b5100 11:45:08 ../source/server.hpp:1153] SIGPIPE INIT
[0x7f14812b5100 11:45:08 client3.cpp:20] [HTTP/1.1 200 OK
Content-Length: 168
Connection: keep-alive
Content-Type: text/plain

GET /hello HTTP/1.1
Content-Length: 100
Connection: keep-alive

bitejiuyekeGET /hello HTTP/1.1
Connection: keep-alive
Content-Length: 100

bitejiuyekeGET /hello
#从这里能看到，上边的数据是第一次请求的处理，下边的是剩下的数据的处理，处理出错了，然后连接被关闭
```

```
HTTP/1.1 400 Bad Request
Content-Length: 129
Connection: close
Content-Type: text/html

<html><head><meta http-equiv='Content-Type' content='text/html;charset=utf-8'></head><body><h1>400 Bad Request</h1></body></html>]
[0x7f14812b5100 11:45:11 ../source/server.hpp:264] SOCKET SEND FAILED!!
client3: client3.cpp:16: int main(): Assertion `cli_sock.Send(req.c_str(), req.size()) != -1' failed.
Aborted (core dumped)
```

```
zwc@139-159-150-152:~/workspace/http-v1/source/http$ ./main
[0x7ffaffcee700 11:45:08 http.hpp:780] NEW CONNECTION 0x560b9523b260
[0x7ffaffcee700 11:45:08 ../server.hpp:978] RELEASE CONNECTION:0x560b9523b260
```

### 超时连接释放测试3：

接收请求的数据，但是业务处理的时间过长，超过了设置的超时销毁时间(服务器性能达到瓶颈)，观察服务端的处理。

预期结果：在一次业务处理中耗费太长时间，导致其他连接被连累超时，导致其他的连接有可能会超时释放。

假设有 12345 描述符就绪了， 在处理1的时候花费了30s处理完，超时了，导致2345描述符因为长时间没有刷新活跃度，则存在两种可能处理结果：

1. 如果接下来的2345描述符都是通信连接描述符，恰好本次也都就绪了事件，则并不影响，因为等1处理完了，接下来就会进行处理并刷新活跃度。

2. 如果接下来的2号描述符是定时器事件描述符，定时器触发超时，执行定时任务，就会将345描述符给释放掉，这时候一旦345描述符对应的连接被释放，接下来在处理345事件的时候就会导致程序崩溃（内存访问错误），因此，在任意的事件处理中，都不应该直接对连接进行释放，而应该将释放操作压入到任务池中，等所有连接事件处理完了，然后执行任务池中的任务的时候再去进行释放。

```
/* 业务处理超时，查看服务器的处理情况
*/

#include "../source/server.hpp"

int main()
{
    signal(SIGCHLD, SIG_IGN);
    for (int i = 0; i < 10; i++) {
```

```
        pid_t pid = fork();
        if (pid < 0) {
            DBG_LOG("FORK ERROR");
            return -1;
        }else if (pid == 0) {
            Socket cli_sock;
            cli_sock.CreateClient(8085, "127.0.0.1");
            std::string req = "GET /hello HTTP/1.1\r\nConnection: keep-alive\r\nContent-Length: 0\r\n\r\n";
            while(1) {
                assert(cli_sock.Send(req.c_str(), req.size()) != -1);
                char buf[1024] = {0};
                assert(cli_sock.Recv(buf, 1023));
                DBG_LOG("[%s]", buf);
            }
            cli_sock.Close();
            exit(0);
        }
    }
    while(1) sleep(1);

    return 0;
}
```

```
/*业务处理中sleep 15s，超过服务器设置的超时时间*/
void Hello(const HttpRequest &req, HttpResponse *rsp)
{
    rsp->SetContent(RequestStr(req), "text/plain");
    sleep(15);
}

int main()
{
    HttpServer server(8085);
    server.SetThreadCount(3);
    server.SetBaseDir(WWWROOT);
    server.Get("/hello", Hello);
    server.Listen();
    return 0;
}
```

```
zwc@139-159-150-152:~/workspace/http-v1/source/http$ ./main
[0x7fc5d5977740 11:01:53 ../server.hpp:1153] SIGPIPE INIT
```

```
[0x7fc5d5175700 11:02:00 http.hpp:780] NEW CONNECTION 0x55e4a1e41260
[0x7fc5d4974700 11:02:00 http.hpp:780] NEW CONNECTION 0x55e4a1e41e90
[0x7fc5d5976700 11:02:00 http.hpp:780] NEW CONNECTION 0x55e4a1e42a60
#注意上边的时间，这里是前3个连接被正常开始处理，因为业务处理耗时较长，因此剩下的连接15s后才被加入监控，
#因为压入任务池的established接口15s后才被执行
[0x7fc5d5175700 11:02:15 http.hpp:780] NEW CONNECTION 0x55e4a1e43660
[0x7fc5d5175700 11:02:15 http.hpp:780] NEW CONNECTION 0x55e4a1e45a60
[0x7fc5d5175700 11:02:15 http.hpp:780] NEW CONNECTION 0x55e4a1e47ef0
[0x7fc5d4974700 11:02:15 http.hpp:780] NEW CONNECTION 0x55e4a1e44260
[0x7fc5d4974700 11:02:15 http.hpp:780] NEW CONNECTION 0x55e4a1e46670
[0x7fc5d5976700 11:02:15 http.hpp:780] NEW CONNECTION 0x55e4a1e44e60
[0x7fc5d5976700 11:02:15 http.hpp:780] NEW CONNECTION 0x55e4a1e472b0
#因为处理时间过长，导致定时器超时了15次以上，则15s以内的定时任务都会被处理，将销毁任务压入任务池
#先进行事件处理，剩下的这7个连接事件处理完（30s或45s后，一个连接处理需要15s，三个线程分别被分配了2~3个连接）
#事件处理完毕后，开始处理任务池任务，开始了连接的销毁
[0x7fc5d5977740 11:02:45 ../server.hpp:978] RELEASE CONNECTION:0x55e4a1e44260
[0x7fc5d5977740 11:02:45 ../server.hpp:978] RELEASE CONNECTION:0x55e4a1e46670
[0x7fc5d5977740 11:02:45 ../server.hpp:978] RELEASE CONNECTION:0x55e4a1e41e90
[0x7fc5d5977740 11:02:45 ../server.hpp:978] RELEASE CONNECTION:0x55e4a1e44e60
[0x7fc5d5977740 11:02:45 ../server.hpp:978] RELEASE CONNECTION:0x55e4a1e472b0
[0x7fc5d5977740 11:02:45 ../server.hpp:978] RELEASE CONNECTION:0x55e4a1e42a60
[0x7fc5d5977740 11:03:00 ../server.hpp:978] RELEASE CONNECTION:0x55e4a1e43660
[0x7fc5d5977740 11:03:00 ../server.hpp:978] RELEASE CONNECTION:0x55e4a1e45a60
[0x7fc5d5977740 11:03:00 ../server.hpp:978] RELEASE CONNECTION:0x55e4a1e47ef0
[0x7fc5d5977740 11:03:00 ../server.hpp:978] RELEASE CONNECTION:0x55e4a1e41260

```

```
zwc@139-159-150-152:~/workspace/http-v1/test$ ./client4
[0x7fd57ebd8100 11:10:17 ../source/server.hpp:1153] SIGPIPE INIT
#前边分别分配到3个线程的连接分别在15s后得到了正常响应
[0x7fd57ebd8100 11:10:32 client4.cpp:30] [HTTP/1.1 200 OK
Content-Length: 66
Connection: keep-alive
Content-Type: text/plain

GET /hello HTTP/1.1
Content-Length: 0
Connection: keep-alive

[0x7fd57ebd8100 11:10:32 client4.cpp:30] [HTTP/1.1 200 OK
Content-Length: 66
Connection: keep-alive
```

```
Content-Type: text/plain

GET /hello HTTP/1.1
Content-Length: 0
Connection: keep-alive

]
]
[0x7fd57ebd8100 11:10:32 client4.cpp:30] [HTTP/1.1 200 OK
Content-Length: 66
Connection: keep-alive
Content-Type: text/plain

GET /hello HTTP/1.1
Content-Length: 0
Connection: keep-alive

]
#这里注意，剩下的7个连接并没有得到正常的回复，因为事件处理完毕后，有可能先执行了定时销毁任任务
#连接被销毁了，则后续的操作也就无法进行了。因此剩下的连接并没有得到正常的响应，而是连接被关闭
[0x7fd57ebd8100 11:11:02 ../source/server.hpp:248] SOCKET RECV FAILED!!
[0x7fd57ebd8100 11:11:02 ../source/server.hpp:248] SOCKET RECV FAILED!!
[0x7fd57ebd8100 11:11:02 ../source/server.hpp:248] SOCKET RECV FAILED!!
[0x7fd57ebd8100 11:11:02 client4.cpp:30] []
[0x7fd57ebd8100 11:11:02 ../source/server.hpp:264] SOCKET SEND FAILED!!
client4: client4.cpp:27: int main(): Assertion `cli_sock.Send(req.c_str(), req.size()) != -1' failed.
[0x7fd57ebd8100 11:11:02 client4.cpp:30] []
[0x7fd57ebd8100 11:11:02 ../source/server.hpp:248] SOCKET RECV FAILED!!
[0x7fd57ebd8100 11:11:02 client4.cpp:30] []
[0x7fd57ebd8100 11:11:02 ../source/server.hpp:248] SOCKET RECV FAILED!!
[0x7fd57ebd8100 11:11:02 client4.cpp:30] []
[0x7fd57ebd8100 11:11:02 ../source/server.hpp:264] SOCKET SEND FAILED!!
client4: client4.cpp:27: int main(): Assertion `cli_sock.Send(req.c_str(), req.size()) != -1' failed.
[0x7fd57ebd8100 11:11:02 ../source/server.hpp:248] SOCKET RECV FAILED!!
[0x7fd57ebd8100 11:11:02 ../source/server.hpp:248] SOCKET RECV FAILED!!
[0x7fd57ebd8100 11:11:02 client4.cpp:30] []
[0x7fd57ebd8100 11:11:02 ../source/server.hpp:264] SOCKET SEND FAILED!!
client4: client4.cpp:27: int main(): Assertion `cli_sock.Send(req.c_str(), req.size()) != -1' failed.
[0x7fd57ebd8100 11:11:02 client4.cpp:30] []
[0x7fd57ebd8100 11:11:02 ../source/server.hpp:248] SOCKET RECV FAILED!!
[0x7fd57ebd8100 11:11:02 ../source/server.hpp:248] SOCKET RECV FAILED!!
[0x7fd57ebd8100 11:11:02 client4.cpp:30] []
```

```
[0x7fd57ebd8100 11:11:02 ../source/server.hpp:248] SOCKET RECV FAILED!!
[0x7fd57ebd8100 11:11:02 client4.cpp:30] []
[0x7fd57ebd8100 11:11:02 ../source/server.hpp:264] SOCKET SEND FAILED!!
client4: client4.cpp:27: int main(): Assertion `cli_sock.Send(req.c_str(), req.size()) != -1' failed.
[0x7fd57ebd8100 11:11:02 client4.cpp:30] []
[0x7fd57ebd8100 11:11:02 client4.cpp:30] []
[0x7fd57ebd8100 11:11:02 ../source/server.hpp:264] SOCKET SEND FAILED!!
client4: client4.cpp:27: int main(): Assertion `cli_sock.Send(req.c_str(), req.size()) != -1' failed.
[0x7fd57ebd8100 11:11:02 ../source/server.hpp:264] SOCKET SEND FAILED!!
client4: client4.cpp:27: int main(): Assertion `cli_sock.Send(req.c_str(), req.size()) != -1' failed.
[0x7fd57ebd8100 11:11:17 ../source/server.hpp:248] SOCKET RECV FAILED!!
[0x7fd57ebd8100 11:11:17 ../source/server.hpp:248] SOCKET RECV FAILED!!
[0x7fd57ebd8100 11:11:17 ../source/server.hpp:248] SOCKET RECV FAILED!!
[0x7fd57ebd8100 11:11:17 ../source/server.hpp:248] SOCKET RECV FAILED!!
[0x7fd57ebd8100 11:11:17 client4.cpp:30] []
[0x7fd57ebd8100 11:11:17 ../source/server.hpp:264] SOCKET SEND FAILED!!
client4: client4.cpp:27: int main(): Assertion `cli_sock.Send(req.c_str(), req.size()) != -1' failed.
[0x7fd57ebd8100 11:11:17 client4.cpp:30] []
[0x7fd57ebd8100 11:11:17 ../source/server.hpp:248] SOCKET RECV FAILED!!
[0x7fd57ebd8100 11:11:17 client4.cpp:30] []
[0x7fd57ebd8100 11:11:17 ../source/server.hpp:248] SOCKET RECV FAILED!!
[0x7fd57ebd8100 11:11:17 client4.cpp:30] []
[0x7fd57ebd8100 11:11:17 ../source/server.hpp:264] SOCKET SEND FAILED!!
[0x7fd57ebd8100 11:11:17 client4.cpp:30] []
[0x7fd57ebd8100 11:11:17 client4.cpp:30] []
[0x7fd57ebd8100 11:11:17 ../source/server.hpp:264] SOCKET SEND FAILED!!
client4: client4.cpp:27: int main(): Assertion `cli_sock.Send(req.c_str(), req.size()) != -1' failed.
[0x7fd57ebd8100 11:11:17 ../source/server.hpp:248] SOCKET RECV FAILED!!
[0x7fd57ebd8100 11:11:17 client4.cpp:30] []
client4: client4.cpp:27: int main(): Assertion `cli_sock.Send(req.c_str(), req.size()) != -1' failed.
[0x7fd57ebd8100 11:11:17 ../source/server.hpp:264] SOCKET SEND FAILED!!
client4: client4.cpp:27: int main(): Assertion `cli_sock.Send(req.c_str(), req.size()) != -1' failed.

```

### 数据中多条请求处理测试：

给服务器发送的一条数据中包含有多个HTTP请求，观察服务器的处理。

预期结果：每一条请求都有其对应的响应

```cpp
/*一次性给服务器发送多条数据，然后查看服务器的处理结果*/
/*每一条请求都应该得到正常处理*/

#include "../source/server.hpp"

int main()
{
    Socket cli_sock;
    cli_sock.CreateClient(8085, "127.0.0.1");
    std::string req = "GET /hello HTTP/1.1\r\nConnection: keep-alive\r\nContent-Length: 0\r\n\r\n";
    req += "GET /hello HTTP/1.1\r\nConnection: keep-alive\r\nContent-Length: 0\r\n\r\n";
    req += "GET /hello HTTP/1.1\r\nConnection: keep-alive\r\nContent-Length: 0\r\n\r\n";
    while(1) {
        assert(cli_sock.Send(req.c_str(), req.size()) != -1);
        char buf[1024] = {0};
        assert(cli_sock.Recv(buf, 1023));
        DBG_LOG("[%s]", buf);
        sleep(3);
    }
    cli_sock.Close();
    return 0;
}
```

```
zwc@139-159-150-152:~/workspace/http-v1/test$ ./client5
[0x7ff75fae2100 11:50:50 ../source/server.hpp:1153] SIGPIPE INIT
[0x7ff75fae2100 11:50:50 client5.cpp:17] [HTTP/1.1 200 OK
Content-Length: 66
Connection: keep-alive
Content-Type: text/plain

GET /hello HTTP/1.1
Content-Length: 0
Connection: keep-alive

HTTP/1.1 200 OK
Content-Length: 66
Connection: keep-alive
Content-Type: text/plain

```

```
GET /hello HTTP/1.1
Content-Length: 0
Connection: keep-alive

HTTP/1.1 200 OK
Content-Length: 66
Connection: keep-alive
Content-Type: text/plain

GET /hello HTTP/1.1
Content-Length: 0
Connection: keep-alive

]

```

可以从打印结果看到，多个响应也被放到缓冲区中同时进行发送了。因此这边一次是接收了所有的响应，一次是在连接超时关闭后接收为空。

### PUT大文件上传测试：

使用put请求上传一个大文件进行保存，大文件数据的接收会被分在多次请求中接收，然后计算源文件和上传后保存的文件的MD5值，判断请求的接收处理是否存在问题。（这里主要观察的是上下文的处理过程是否正常。）

```

void PutFile(const HttpRequest &req, HttpResponse *rsp)
{
    std::string pathname = WWWROOT + req._path;
    Util::WriteFile(pathname, req._body);
}
int main()
{
    HttpServer server(8085);
    server.SetThreadCount(3);
    server.SetBaseDir(WWWROOT);
    server.Put("/1234.txt", PutFile);
    server.Listen();
    return 0;
}
```

```
/*大文件传输测试，给服务器上传一个大文件，服务器将文件保存下来，观察处理结果*/
/*
```

```
    上传的文件，和服务器保存的文件一致
*/
#include "../source/http/http.hpp"

int main()
{
    Socket cli_sock;
    cli_sock.CreateClient(8085, "127.0.0.1");
    std::string req = "PUT /1234.txt HTTP/1.1\r\nConnection: keep-alive\r\n";
    std::string body;
    Util::ReadFile("./hello.txt", &body);
    req += "Content-Length: " + std::to_string(body.size()) + "\r\n\r\n";
    assert(cli_sock.Send(req.c_str(), req.size()) != -1);
    assert(cli_sock.Send(body.c_str(), body.size()) != -1);
    char buf[1024] = {0};
    assert(cli_sock.Recv(buf, 1023));
    DBG_LOG("[%s]", buf);
    sleep(3);
    cli_sock.Close();
    return 0;
}
```

测试结果：上传文件成功，根据计算的MD5值判断两个文件完全一致。

```
# 鉴于服务器的内存较小，创建一个300M文件进行测试
zwc@139-159-150-152:~/workspace/http-v1/test$ dd if=/dev/zero of=./hello.txt bs=1M count=300
300+0 records in
300+0 records out
314572800 bytes (315 MB, 300 MiB) copied, 1.10349 s, 285 MB/szwc@139-159-150-152:~/workspace/http-v1/test$ ls -sh ./
total 303M
164K client1      4.0K client2.cpp  164K client4      4.0K client5.cpp  300M hello.txt  4.0K server.cc
4.0K client1.cpp  164K client3      4.0K client4.cpp  300K client6      4.0K Makefile   4.0K tcp_cli.cc
164K client2      4.0K client3.cpp  164K client5      4.0K client6.cpp  1.4M server     4.0K tcp_srv.cc
```

```
zwc@139-159-150-152:~/workspace/http-v1/source/http$ ./main
[0x7f2579859740 11:52:17 ../server.hpp:1153] SIGPIPE INIT
[0x7f2579057700 11:55:05 http.hpp:780] NEW CONNECTION 0x560119615260
[0x7f2579057700 11:55:07 ../server.hpp:89] RESIZE 32818
```

```
[0x7f2579057700 11:55:07 ../server.hpp:89] RESIZE 98276
[0x7f2579057700 11:55:13 ../server.hpp:248] SOCKET RECV FAILED!!  #这是客户都安连接主动关闭
[0x7f2579859740 11:55:13 ../server.hpp:978] RELEASE CONNECTION:0x560119615260
```

```
zwc@139-159-150-152:~/workspace/http-v1/test$ ./client6
[0x7f1f16d14100 11:55:05 ../source/http/../server.hpp:1153] SIGPIPE INIT
[0x7f1f16d14100 11:55:10 client6.cpp:19] [HTTP/1.1 200 OK
Connection: keep-alive

]
```

```
# 创建出来的源文件
zwc@139-159-150-152:~/workspace/http-v1/test$ ls
client1      client2      client3      client4      client5      client6      hello.txt  server     tcp_cli.cc
client1.cpp  client2.cpp  client3.cpp  client4.cpp  client5.cpp  client6.cpp  Makefile   server.cc  tcp_srv.cc
zwc@139-159-150-152:~/workspace/http-v1/test$ md5sum hello.txt
0d97a9cd8bbd7ce75a2a76bb06258915  hello.txt

# 这是服务器接收数据后存储的文件
zwc@139-159-150-152:~/workspace/http-v1/source/http/wwwroot$ ls
1234.txt  index.html
zwc@139-159-150-152:~/workspace/http-v1/source/http/wwwroot$ md5sum 1234.txt
0d97a9cd8bbd7ce75a2a76bb06258915  1234.txt

#两个文件的MD5值是完全一致的，则文件数据完全一致。
```

## 性能测试：

采用webbench进行服务器性能测试。

Webbench是知名的网站压力测试工具，它是由Lionbridge公司（http://www.lionbridge.com）开发。

webbench的标准测试可以向我们展示服务器的两项内容：`每秒钟相应请求数` 和 `每秒钟传输数据量`

webbench测试原理是，创建指定数量的进程，在每个进程中不断创建套接字向服务器发送请求，并通过管道最终将每个进程的结果返回给主进程进行数据统计。

性能测试的两个重点衡量标准：吞吐量 & QPS

### 测试环境：

任何不说明测试环境的测试都是白给....

服务器环境：4核4G虚拟机ubuntu-22.04LTS，服务器程序采用1主3从reactor模式

webbench客户端环境：同一个虚拟机....

（测试的意义不大，因为同主机会造成互相的cpu争抢，但是这里目前没办法，毕竟服务器的带宽和资源太低了....这里主要目的是告诉大家如何进行性能压力测试）

```
[dev@localhost workspace]$ ps -ef |grep main
dev         3155    3040  0 12:25 pts/5    00:00:00 ./main
dev         3173    2859  0 12:25 pts/4    00:00:00 grep --color=auto main
[dev@localhost workspace]$ cat /proc/3155/limits
Limit                     Soft Limit           Hard Limit           Units
Max cpu time              unlimited            unlimited            seconds
Max file size             unlimited            unlimited            bytes
Max data size             unlimited            unlimited            bytes
Max stack size            8388608              unlimited            bytes
Max core file size        0                    unlimited            bytes
Max resident set          unlimited            unlimited            bytes
Max processes             7170                 7170                 processes
Max open files            65535                65535                files
Max locked memory         65536                65536                bytes
Max address space         unlimited            unlimited            bytes
Max file locks            unlimited            unlimited            locks
Max pending signals       7170                 7170                 signals
Max msgqueue size         819200               819200               bytes
Max nice priority         0                    0
Max realtime priority     0                    0
Max realtime timeout      unlimited            unlimited            us
[dev@localhost workspace]$
```

将程序运行起来，根据进程ID，在/proc目录下查看程序中的各项限制信息，能够看到当前用户的进程的最大数量为7170，有些不够用，因为后边需要模拟上万并发量，需要创建上万个进程。

修改配置文件：/etc/security/limits.conf， 在末尾添加内容，nofile是修改可打开文件数，nproc是修改进程数

```
61 # End of file
62 root soft nofile 65535
63 root hard nofile 65535
64 * soft nofile 65535


65 * hard nofile 65535
```

```
 66 * soft nproc 65535
 67 * hard nproc 65535
```

修改配置文件：/etc/security/limits.d/20-nproc.conf

```
# Default limit for number of user's processes to prevent


# accidental fork bombs.
# See rhbz #432903 for reasoning.

*          soft    nproc     65535
root       soft    nproc     unlimited
```

重启机器：已经改变

```
[dev@localhost workspace]$ ps -ef |grep main
root         713       1  0 12:30 ?        00:00:00 /usr/sbin/alsactl -s -n 19 -c -E ALSA_CONFIG_PATH=/etc/alsa/alsactl.conf --initfile=/lib/alsa/init/00main rdaemon
dev         2814    2802  3 12:31 ?        00:00:01 /home/dev/.vscode-server/bin/e2816fe719a4026ffa1ee0189dc89bdfdbafb164/node /home/dev/.vscode-server/bin/e2816fe719a4026ffa1ee0189dc89bdfdbafb164/out/server-main.js --start-server --host=127.0.0.1 --accept-server-license-terms --enable-remote-auto-shutdown --port=0 --telemetry-level all --connection-token-file /home/dev/.vscode-server/.e2816fe719a4026ffa1ee0189dc89bdfdbafb164.token
dev         3269    3107  0 12:32 pts/9    00:00:00 ./main
dev         3299    3187  0 12:32 pts/10   00:00:00 grep --color=auto main
[dev@localhost workspace]$ cat /proc/3269/limits
Limit                     Soft Limit           Hard Limit           Units
Max cpu time              unlimited            unlimited            seconds
Max file size             unlimited            unlimited            bytes
Max data size             unlimited            unlimited            bytes
Max stack size            8388608              unlimited            bytes
Max core file size        0                    unlimited            bytes
Max resident set          unlimited            unlimited            bytes
Max processes             65535                65535                processes
Max open files            65535                65535                files
Max locked memory         65536                65536                bytes
Max address space         unlimited            unlimited            bytes
Max file locks            unlimited            unlimited            locks
Max pending signals       7170                 7170                 signals
Max msgqueue size         819200               819200               bytes
```

```
Max nice priority         0                   0
Max realtime priority     0                   0
Max realtime timeout      unlimited           unlimited           us
[dev@localhost workspace]$
```

### 测试1：500个客户端连接的情况下测试结果

```
[dev@localhost webbench]$ ./webbench -c 500 -t 60 http://127.0.0.1:8500/
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=391061 pages/min, 2620396 bytes/sec.
Requests: 391061 susceed, 0 failed.
```

### 测试2：5000个客户端连接的情况下测试结果

```
[dev@localhost webbench]$ ./webbench -c 5000 -t 60 http://127.0.0.1:8500/
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: 5000 clients, running 60 sec.

Speed=384577 pages/min, 2589020 bytes/sec.
Requests: 384577 susceed, 0 failed.
```

### 测试3：10000个客户端并发连接的情况下测试结果（启用3个worker线程池）

```
[dev@localhost webbench]$ ./webbench -c 10000 -t 60 http://127.0.0.1:8500/
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: 10000 clients, running 60 sec.

Speed=357157 pages/min, 2423604 bytes/sec.
Requests: 357157 susceed, 0 failed.
```

以上测试中，使用浏览器访问服务器，均能流畅获取请求的页面。但是根据测试结果能够看出，虽然并发量一直在提高，但是总的请求服务器的数量并没有增加，反而有所降低，侧面反馈了处理所耗时间更多了，基本上可以根据35w/min左右的请求量计算出10000并发量时服务器的极限了，但是这个测试其实意义不大，因为测试客户端和服务器都在同一台机器上，传输的速度更快，但同时抢占cpu也影响了处理，最好的方式就是在两台不同的机器上进行测试，这里只是通过这个方法告诉大家该如何对服务器进行性能测试。

目前受限于设备环境配置，尚未进行更多并发量的测试。