# 仿mudou库one thread oneloop式并发服务器实现


# 一、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，要明确的是要实现的是一一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。**

# 二、前置知识

## （一）HTTP服务器

1.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

搭建一个TCP服务器，接收客户端请求。
以HTTP协议格式进行解析请求数据，明确客户端目的。
明确客户端请求目的后提供对应服务。
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。

## （二）Reactor模型：

1.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。

2.分类

（1）单Reactor单线程：单I/O多路复用+业务处理。

通过IO多路复用模型进行客户端请求监控。
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。

（2）单Reactor多线程：单I/O多路复用+线程池（业务处理）

Reactor线程通过I/O多路复用模型进行客户端请求监控
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
加粗样式缺点：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）
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
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
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

## （三）目标定位-One Thread One Loop主从Reactor模型高并发服务器

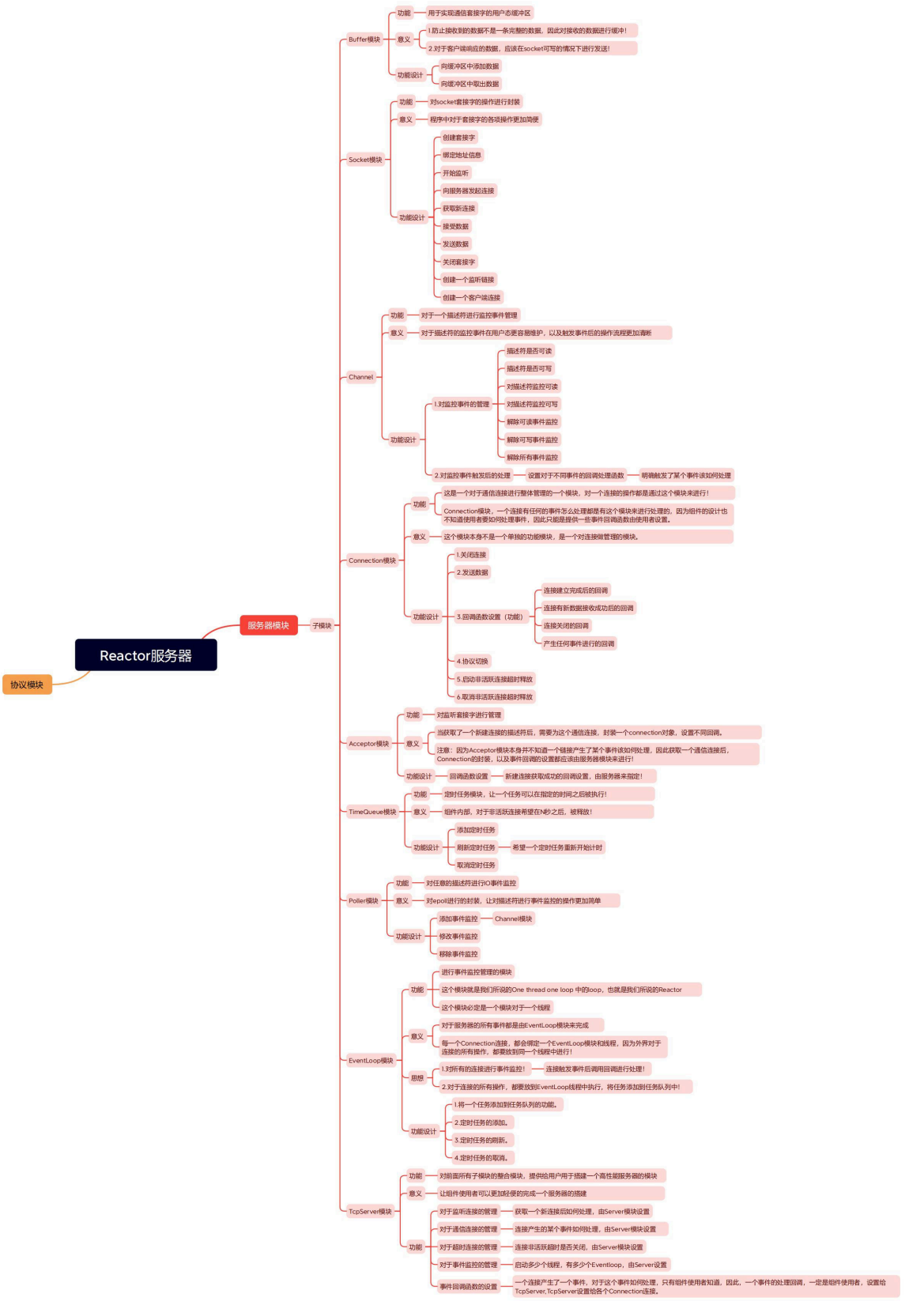
咱们要实现的是主从Reactor模型服务器，也就是主Reactor线程仅仅监控监听描述符，获取新建连接，保证获取新连接的高效性，提高服务器的并发性能。
主Reactor获取到新连接后分发给子Reactor进行通信事件监控。而子Reactor线程监控各自的描述符的读写事件进行数据读写以及业务处理。

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。

# 三、功能划分

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；




# 四、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：

（一）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
代码和思路具体在:1.4.C++项目：仿muduo库实现并发服务器之buffer模块的设计https://blog.csdn.net/weixin_54447296/article/details/133429811?spm=1001.2014.3001.5501

（二）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。
代码和思路具体在:1.5.C++项目：仿muduo库实现并发服务器之socket模块的设计https://blog.csdn.net/weixin_54447296/article/details/133435778?spm=1001.2014.3001.5501

（三）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。
代码和思路具体在:1.6.C++项目：仿muduo库实现并发服务器之channel模块的设计https://blog.csdn.net/weixin_54447296/article/details/133439943?spm=1001.2014.3001.5501

（四）Poller模块

对epoll进行的封装，让对描述符进行事件监控的操作更加简单。
代码和思路具体在:1.7.C++项目：仿muduo库实现并发服务器之Poller模块的设计https://blog.csdn.net/weixin_54447296/article/details/133497920?spm=1001.2014.3001.5501

（五）EventLoop模块

这个模块和线程是一一对应的！
监听了一个链接，如果这个连接一旦就绪，就要进行事件处理！
但是如果这个描述符，在多个线程中都触发了了事件，进行处理，就会存在线程安全问题！
因此我们需要将一个链接的事件监控， 以及连接事件处理，以及其他操作都放在同一个线程中！
如何保证一个连接的所有操作都在eventloop对应的线程中！
给eventLOOP模块中，都添加一个任务队列！
对连接的所有操作，都进行一次封装，将对连接的操作当作任务都添加到任务队列中！
代码和思路具体在:1.8.C++项目：仿muduo库实现并发服务器之EventLoop模块的设计https://blog.csdn.net/weixin_54447296/article/details/133498144?spm=1001.2014.3001.5501

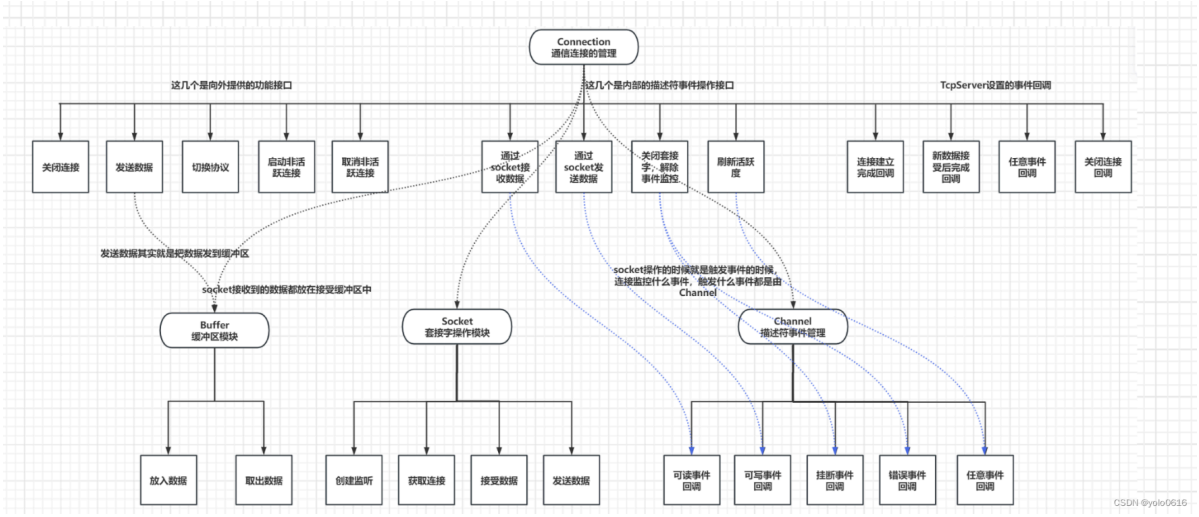
（六）Connection模块

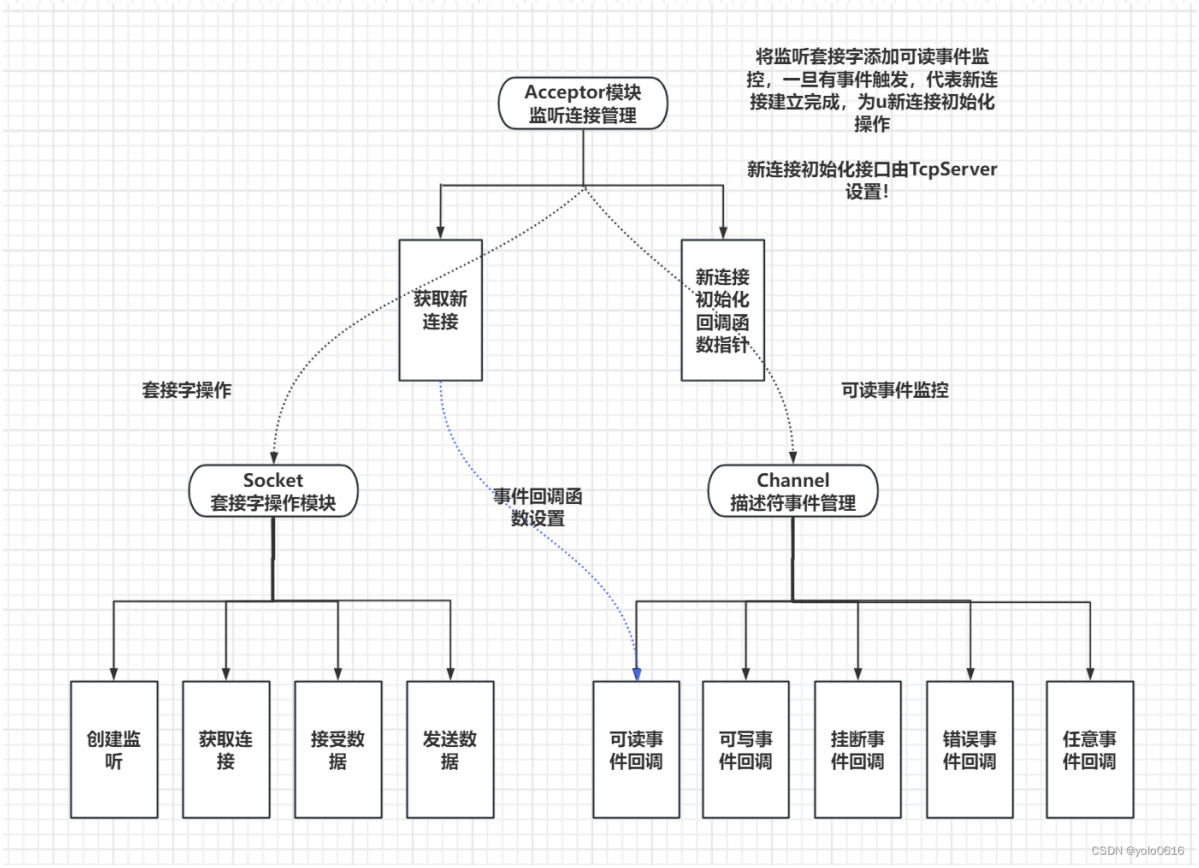
Connection模块是对Buffer模块，Socket模块，Channel模块的一个整体封装，实现了对一个通信套接字的整体的管理，每一个进行数据通信的套接字（也就是accept获取到的新连接）都会使用Connection进行管理。
• Connection模块内部包含有三个由组件使用者传入的回调函数：连接建立完成回调，事件回调，新数据回调，关闭回调。
• Connection模块内部包含有两个组件使用者提供的接口：数据发送接口，连接关闭接口
• Connection模块内部包含有两个用户态缓冲区：用户态接收缓冲区，用户态发送缓冲区
• Connection模块内部包含有一个Socket对象：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

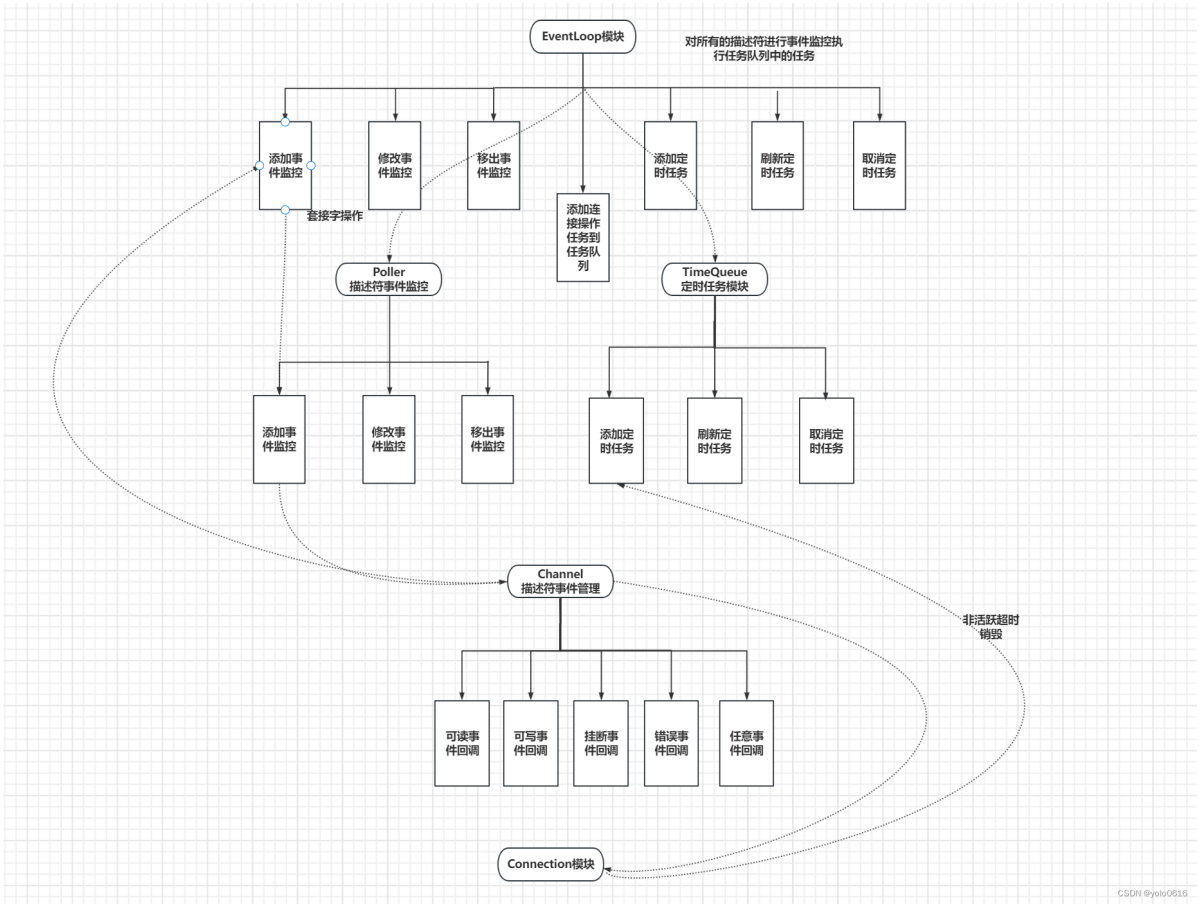
代码和思路具体在:1.9.C++项目：仿muduo库实现并发服务器之Connection模块的设计https://blog.csdn.net/weixin_54447296/article/details/133621187?spm=1001.2014.3001.5501

（七）Acceptor模块

这是一个对于通信连接进行整体管理的一个模块，对一个连接的操作都是通过这个模块来进行！
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。
代码和思路具体在:1.10.C++项目：仿muduo库实现并发服务器之Acceptor模块的设计https://blog.csdn.net/weixin_54447296/article/details/133682170?spm=1001.2014.3001.5501






（八）LoopThread模块

目标：将eventloop模块和线程整合起来！
eventloop 和 线程是一一对应的！
eventloop 模块实例化的对象，在构造的时候就会初始化！ _thread_id;
而后面当运行一个操作的时候判断是否运行在eventloop所对应的线程中，就是将线程ID与EventLoop模块中的thread_id 进行一个比较，相同就表示在同一个线程，不同就表示当前运行线程并不是eventloop线程！
eventloop 模块在实例化对象的时候，必须在线程内部！
eventloop 实例化对象会设置自己的 thread_id;
如果我们先创建了多个 eventloop 对象，然后创建了多个线程，将各自的线程id，重新给 eventloop 进行设置！
存在问题：在构造 eventloop对象，到设置新的 thread_id 期间将是不可控的！
因此，必须先创建线程，然后在线程的入口函数中，去实例化 eventloop 对象！
构造一个新的模块：LoopThread

代码和思路具体在:1.11.C++项目：仿muduo库实现并发服务器之LoopThread模块的设计https://blog.csdn.net/weixin_54447296/article/details/133682565?spm=1001.2014.3001.5501

（九）LoopThreadPool模块

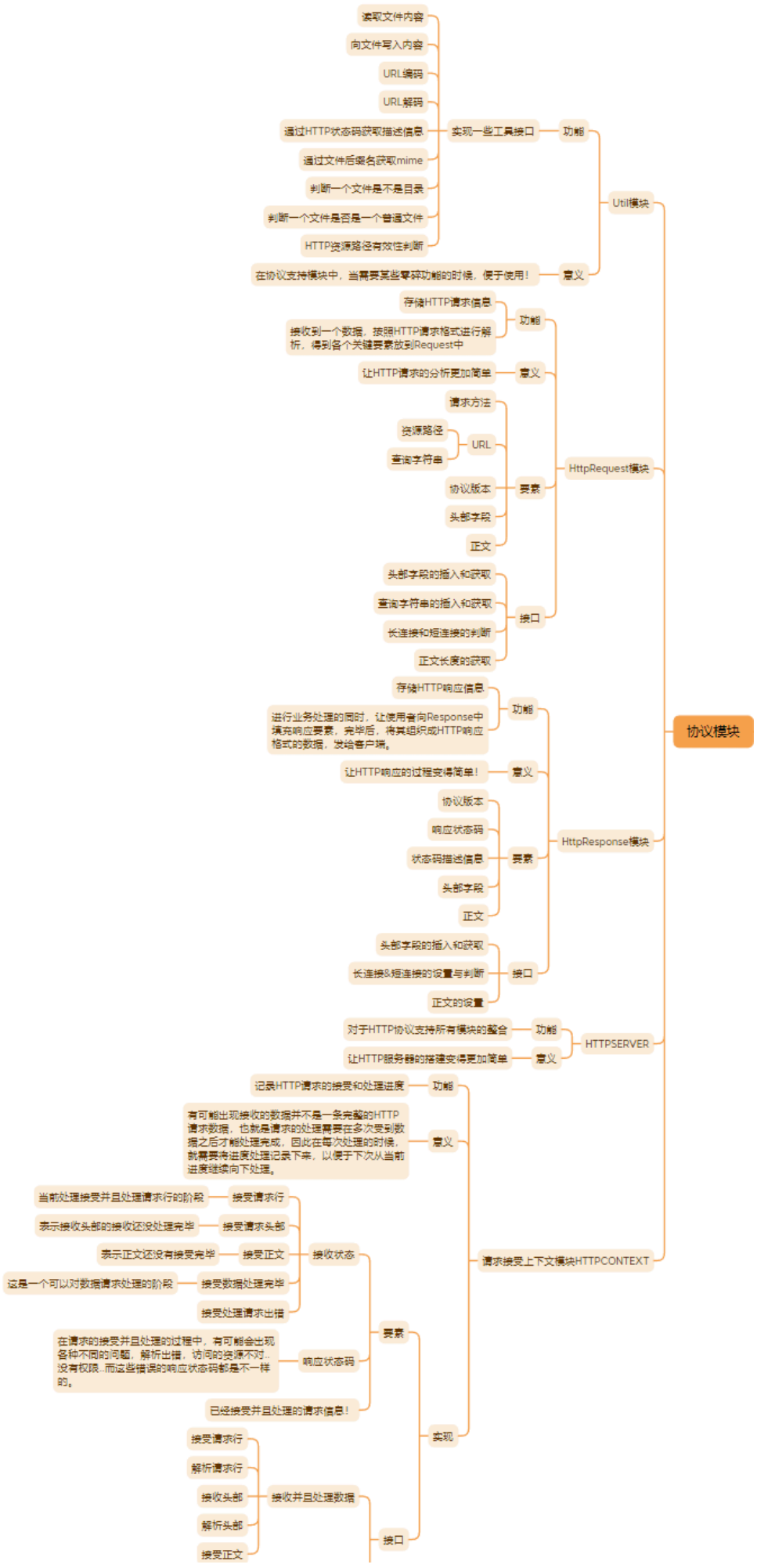
线程数量可配置（0或多个）
对所有的线程进行管理，其实也就是管理0个或多个LoopThread对象！
提供线程分配的功能！
代码和思路具体在:1.12.C++项目：仿muduo库实现并发服务器之LoopThreadPool模块的设计https://blog.csdn.net/weixin_54447296/article/details/133829550?spm=1001.2014.3001.5501

（十）TcpServer模块

目的:TcpServer模块： 对所有模块的整合，通过 tcpserver 模块实例化的对象，可以非常简单的完成一个服务器的搭建。
对前面所有子模块的整合模块，提供给用户用于搭建一个高性能服务器的模块！

1.13.C++项目：仿muduo库实现并发服务器之TcpServer模块的设计https://blog.csdn.net/weixin_54447296/article/details/133829919?spm=1001.2014.3001.5501

## 五、HTTP模块：


协议模块
Util模块
功能
实现一些工具接口
读取文件内容
向文件写入内容
URL编码
URL解码
通过HTTP状态码获取描述信息
通过文件后缀名获取mime
判断一个文件是不是目录
判断一个文件是否是一个普通文件
HTTP资源路径有效性判断
意义
在协议支持模块中，当需要某些零碎功能的时候，便于使用！
HttpRequest模块
功能
存储HTTP请求信息
接收到一个数据，按照HTTP请求格式进行解析，得到各个关键要素放到Request中
意义
让HTTP请求的分析更加简单
要素
请求方法
URL
资源路径
查询字符串
协议版本
头部字段
正文
接口
头部字段的插入和获取
查询字符串的插入和获取
长连接和短连接的判断
正文长度的获取
HttpResponse模块
功能
存储HTTP响应信息
进行业务处理的同时，让使用者向Response中填充响应要素，完毕后，将其组织成HTTP响应格式的数据，发给客户端。
意义
让HTTP响应的过程变得简单！
要素
协议版本
响应状态码
状态码描述信息
头部字段
正文
接口
头部字段的插入和获取
长连接&短连接的设置与判断
正文的设置
HTTPSERVER
功能
对于HTTP协议支持所有模块的整合
意义
让HTTP服务器的搭建变得更加简单
请求接受上下文模块HTTPCONTEXT
功能
记录HTTP请求的接受和处理进度
意义
有可能出现接收的数据并不是一条完整的HTTP请求数据，也就是请求的处理需要在多次受到数据之后才能处理完成，因此在每次处理的时候，就需要将进度处理记录下来，以便于下次从当前进度继续向下处理。
要素
接收状态
接受请求行
当前处理接受并且处理请求行的阶段
接受请求头部
表示接收头部的接收还没处理完毕
接受正文
表示正文还没有接受完毕
接受数据处理完毕
这是一个可以对数据请求处理的阶段
接受处理请求出错
响应状态码
在请求的接受并且处理的过程中，有可能会出现各种不同的问题，解析出错，访问的资源不对，没有权限..而这些错误的响应状态码都是不一样的。
已经接受并且处理的请求信息！
接口
接收并且处理数据
接受请求行
解析请求行
接收头部
解析头部
接受正文
实现

## （一）Util模块

目的:实现一些工具接口
读取文件内容
向文件写入内容
URL编码
URL解码
通过HTTP状态码获取描述信息
通过文件后缀名获取mime
判断一个文件是不是目录
判断一个文件是否是一个普通文件
HTTP资源路径有效性判断

代码和思路具体在:1.14.C++项目：仿muduo库实现并发服务器之Util模块的设计https://blog.csdn.net/weixin_54447296/article/details/133845525?spm=1001.2014.3001.5501

## （二）HttpRequest模块

目的:存储HTTP请求信息
接收到一个数据，按照HTTP请求格式进行解析，得到各个关键要素放到Request中

代码和思路具体在:1.15.C++项目：仿muduo库实现并发服务器之Util模块的设计https://blog.csdn.net/weixin_54447296/article/details/1.15.C++%E9%A1%B9%E7%9B%AE%EF%BC%9A%E4%BB%BFmuduo%E5%BA%93%E5%AE%9E%E7%8E%B0%E5%B9%B6%E5%8F%91%E6%9C%8D%E5%8A%A1%E5%99%A8%E4%B9%8BHttpRequest%E5%92%8CHttpResponse%E6%A8%A1%E5%9D%97%E7%9A%84%E8%AE%BE%E8%AE%A1

## （三）HttpResponse模块

目的:存储HTTP响应信息
意义:进行业务处理的同时，让使用者向Response中填充响应要素，完毕后，将其组织成HTTP响应格式的数据，发给客户端。

代码和思路具体在:1.15.C++项目：仿muduo库实现并发服务器之Util模块的设计https://blog.csdn.net/weixin_54447296/article/details/1.15.C++%E9%A1%B9%E7%9B%AE%EF%BC%9A%E4%BB%BFmuduo%E5%BA%93%E5%AE%9E%E7%8E%B0%E5%B9%B6%E5%8F%91%E6%9C%8D%E5%8A%A1%E5%99%A8%E4%B9%8BHttpRequest%E5%92%8CHttpResponse%E6%A8%A1%E5%9D%97%E7%9A%84%E8%AE%BE%E8%AE%A1

## （四）HttpContext模块

目的:要实现渐变的搭建HTTP服务器，所需要提供的要素和功能!
要素:

GET请求的路由映射表
POST请求的路由映射表
PUT请求的路由映射表
DELETE请求的路由映射表 —— 路由映射表记录对应请求方法的请求的处理函数映射关系
高性能TCP服务器—— 进行连接的IO操作
静态资源相对根目录 —— 实现静态资源的处理
代码和思路具体在:1.16.C++项目：仿muduo库实现并发服务器之HttpContext以及HttpServer模块的设计https://blog.csdn.net/weixin_54447296/article/details/133875869?spm=1001.2014.3001.5501

### （五）HttpServer模块

目的:记录HTTP请求的接受和处理进度。
意义:有可能出现接收的数据并不是一条完整的HTTP请求数据，也就是请求的处理需要在多次受到数据之后才能处理完成，因此在每次处理的时候，就需要将进度处理记录下来，以便于下次从当前进度继续向下处理。

代码和思路具体在:1.16.C++项目：仿muduo库实现并发服务器之HttpContext以及HttpServer模块的设计https://blog.csdn.net/weixin_54447296/article/details/133875869?spm=1001.2014.3001.5501

## 六、测试

### （一）使用Postman进行基本功能测试

post，get，delete，put请求，基本请求全部成功。

### （二）长连接连续请求测试

*/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;
}
```

客户端每三秒发送一次数据，刷新活跃度。长连接测试正常。

```
[root@VM-24-7-centos http]# ./main
[0x7f60f6a75740 15:39:06 ../source/server.hpp:3744] SIGPIPE INIT
[0x7f60f51e0700 15:39:11 http.hpp:1575] NEW CONNECTION 0x153ce70
[0x7f60f51e0700 15:41:27 ../source/server.hpp:2576] SOCKET RECV FAILED!!
[0x7f60f6a75740 15:41:27 ../source/server.hpp:3521] RELEASE CONNECTION:0x153ce70
^Z
```

```
]
[(nil) 15:41:08 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

]
[(nil) 15:41:11 client1.cpp:13] [HTTP/1.1 200 OK
Content-Length: 66
```



## （三）不完整请求测试

*/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，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;
}
```


## （四）业务处理超时测试

/*业务处理超时，查看服务器的处理情况

；　当服务器达到了一个性能瓶颈，在一次业务处理中花费了太长的时间（超过了服务器设置的非活跃超时时间）

；　1. 在一次业务处理中耗费太长时间，导致其他的连接也被连累超时，其他的连接有可能会被拖累超时释放

；　假设现在 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

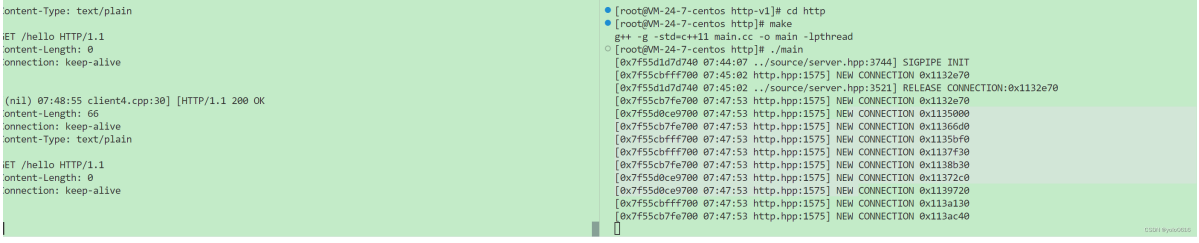
;　　1. 如果接下来的2345描述符都是通信连接描述符，如果都就绪了，则并不影响，因为接下来就会进行处理并刷新活跃度
;　　2. 如果接下来的2号描述符是定时器事件描述符，定时器触发超时，执行定时任务，就会将345描述符给释放掉
;　　这时候一旦345描述符对应的连接被释放，接下来在处理345事件的时候就会导致程序崩溃（内存访问错误）
;　　因此这时候，在本次事件处理中，并不能直接对连接进行释放，而应该将释放操作压入到任务池中，
;　　等到事件处理完了执行任务池中的任务的时候，再去释放
;*/

```
#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;

}
```

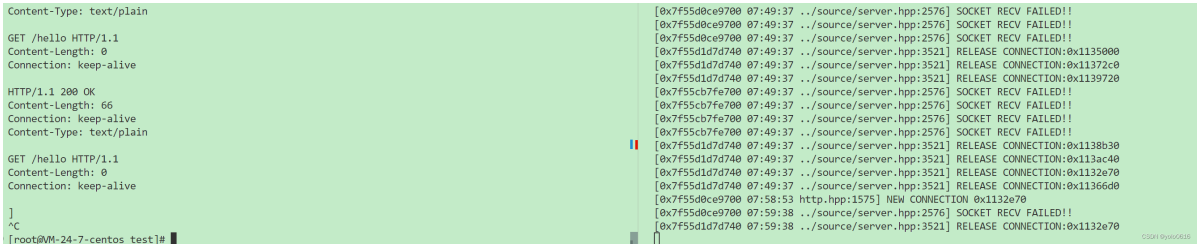


## （五）一次发送多条数据测试

*/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;
}
```




## （六）大文件传输测试

*/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;
}
```

```
[root@VM-24-7-centos test]# dd if=/dev/zero of=./hello.txt bs=1G count=1
1+0 records in
1+0 records out
1073741824 bytes (1.1 GB) copied, 7.15568 s, 150 MB/s
[root@VM-24-7-centos test]# ls -sh
total 1.1G
152K client       4.0K client2.cpp  4.0K client4.cpp  1.1G hello.txt  4.0K tcp_cli.cc
152K client1      152K client3      152K client5      4.0K makefile   1.3M tcp_ser
4.0K client1.cpp  4.0K client3.cpp  4.0K client5.cpp  1.3M server     8.0K tcp_srv.cc
152K client2      152K client4      4.0K client6.cpp  4.0K server.cc
[root@VM-24-7-centos test]# echo "mutao" >> hello.txt
[root@VM-24-7-centos test]# ls -sg
```



```
[root@VM-24-7-centos http]# ./main
[0x7f6b9ef79740 08:07:40 ../source/server.hpp:3744] SIGPIPE INIT
[0x7f6b9d6e4700 08:07:50 http.hpp:1575] NEW CONNECTION 0x169be70
[0x7f6b9d6e4700 08:07:58 ../source/server.hpp:2403] RESIZE 43854
[0x7f6b9d6e4700 08:07:58 ../source/server.hpp:2403] RESIZE 109389
```

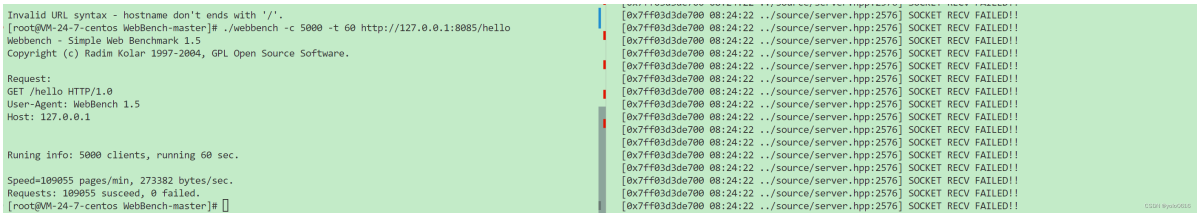


## （七）抗压力测试

性能压力测试：
并发量：可以同时处理多少客户端的请求而不会出现连接失败
QPS:每秒处理的包的数量
借助：webbench工具
原理：创建大量的进程，在进程中，创建客户端连接服务器，发送请求，收到响应后关闭连接，开始下一个连接的建立！

测试环境：服务器是1核2G带宽1M的云服务器
客户端：

模拟5000个客户端同时向服务器发送请求，没有出现连接失败。




测试结论：

服务器并发量：可以同时处理5000-6000个客户端的请求而不会出现连接失败。
QPS：(Query Per Second)每秒查询率1817左右。