代码随想录项目精讲-webserver

代码随想录知识星球在23年4月份发布了第一份项目精讲（java 论坛项目）以来，录友们就一直在期待C++项目的文档。

目前知识星球已经发布了：论坛项目精讲和前端项目精讲

那么正如录友们所期待的，webserver（C++）系列来了！

这份PDF不仅告诉大家，为什么还可以做webserver，适合哪类人群，还系统讲了webserver所需要的基础知识，并且把面试官常问问题、项目细节、优化策略以及测试方案，统统系统梳理出来，帮助 CPP录友们克服项目这一难关。

这份PDF有多全，大家看看目录就知道了：

        - Channel
        - Poller
    - 日志系统
    - 内存池
    - 线程池
    - LFU
- 写好了就完了吗?
    - 部署（没有云服务器的同学可以跳过）
    - 性能测试
- 面试问题
    - 项目介绍
        - 简单介绍一下你的项目
        - 项目中的难点?
        - 项目中遇到的困难？是如何解决的?
        - 针对项目做了哪些优化?
        - 项目中用到哪些设计模式?
        - 这个web服务器是你自己申请的域名吗
        - C++ 面向对象特性在项目中的体现
- 项目细节
    - 线程池
        - 你的线程池工作线程处理完一个任务后的状态是什么?
        - 讲一下你项目中线程池的作用？具体是怎么实现的？有参考开源的线程池实现吗?
        - 请你实现一个简单的线程池（现场手撕）
    - 日志系统
    - 缓存机制
    - 内存池
        - 讲一讲为什么要加入内存池？项目中所有的内存申请都走内存池吗?
    - 并发性问题
        - 如果同时1000个客户端进行访问请求，线程数不多，怎么能及时响应处理每一个呢?
        - 如果一个客户请求需要占用线程很久的时间，会不会影响接下来的客户请求呢，有什么好的策略呢?
    - IO多路复用
        - 说一下什么是ET，什么是LT，有什么区别?
        - LT什么时候会触发？ET呢?
        - 为什么ET模式不可以文件描述符阻塞，而LT模式可以呢?
        - 你用了epoll，说一下为什么用epoll，还有其他多路复用方式吗？区别是什么?
    - 并发模型
        - reactor、proactor模型的区别?
        - reactor模式中，各个模式的区别?
    - 跳表 skiplist
    - 测试相关问题

    - 你是如何对项目进行测试的?
- 星球资料

前言

为什么还要做 WebServer?

各位主攻 cpp 的同学肯定都听过 WebServer 这个烂大街的项目，很多同学或多或少都会想过这个问题：**既然 WebServer 都人手一个了，为什么还要做这个项目，都202x年了还没有新的cpp项目出来吗**?

> 真的没有新的 cpp 项目出来吗?

新的cpp项目当然有，只是这些新项目的前置知识太多、可供参考的文档又不多，还要结合许多开源库使用，对于校招生（或者说对没有具体主攻方向的同学）来说学习成本太高了。

下面列举几个例子（from HelloGitHub-c++分类）：

- 一款简单、高效的实时视频服务器（音视频方向）
- 轻量级中文 OCR 项目（OCR）
- 游戏《金庸群侠传》的 C++ 复刻版（游戏方向）
- 在 C++ 上实现类似 Go goroutine 的库（工具库）
- 阿里开源的轻量级 C++ 异步框架

> 为什么推荐做 WebServer?

- cpper 能够找到的项目（有详细资料的）确实不多

对于科班 or 有充足实习经历 or 课题组项目丰富的同学来说，自然有许多相关的项目能够写在简历上。

但是对于项目经历贫乏的cpp转码选手来说，在网上能够找到的项目（有较详细资料的），无非就是比较简单的xx管理系统、五子棋等简易游戏、各种工具库、烂大街的WebServer，然后你就可以发现里面最高大上的还是WebServer（还有个 RPC 框架）。

- 能够将面试所需的基础知识串联起来

WebServer 能够串联绝大部分的面试八股，语言（C/C++全覆盖，可以扩展至C++11/17）+操作系统（含有大量的 I/O 系统调用及其封装，还有 EPOLL 等多路复用机制）+计算机网络（本身就是一个网络框架，对网络异常的处理）+数据库（注册中心的数据库语句、负载均衡等）。

当你的项目中涵盖了所有面试所需的基础知识后，面试官更倾向结合你的项目去考察知识，而不是东问一个西问一个，**这样可以将面试的问题限制在一定的范围中，一定程度上降低了面试准备的难度**。

- 万物均能集成到 WebServer 中

WebServer 的本质上是一个高性能网络框架，它提供了一个单服务端（当然也可以扩展为多服务端）与多客户端的高效连接框架，但是客户端与服务端连接上以后具体应该做些什么（也就是有哪些业务），这就可以由我们自由发挥了，这就是 WebServer 的功能扩展。

目前大多数的 WebServer 都将从服务端获取 MIME 作为主要功能。

但实际上，之前说到的游戏可以放到上面（是不是就有点像对战平台了），管理系统可以放到上面、存储引擎（卡哥的skipList：https://github.com/youngyangyang04/Skiplist-CPP ）等你能想到的都可以部署在 WebServer 上，这样看来，WebServer 本身具备相当的工作量。

> 那么如果每个 cpper 简历上都有这个项目，那面试官还会问吗?

WebServer 就跟核武器一样，你可以不用，但不能没有（逃

这个问题其实是看面试官的，有的面试官就喜欢问一些每个候选人都有的东西，然后根据回答的差异（项目的新颖点、熟悉程度等）去进行筛选。有的面试官则会挑他自己所感兴趣的项目去提问。

如果面试官在你的简历上没有发现特别感兴趣的项目时，他会让你自己挑一个你觉得做得最好的项目来讲，这个时候你就可以把 WebServer 拿出来讲了（此时大概率会是地狱难度，会挖的很细，因为面试官觉得你做的东西他都能看懂）。

但是这样有个问题，要是大伙都讲这个项目的话，我们肯定得有和别人不一样的地方，这也是面试官最看重的地方。**至于怎么做到和别人不同，大概可以从性能优化和功能扩展两部分入手，这个我们后面再讲。**

WebServer所需要的基础知识

编程语言

WebServer 对编程语言的宽容度较高，掌握基本的 C/C++ 语法即可开始做，而掌握语言的新特性则能够让项目更上一层楼。

在做 WebServer 之前，最少需要掌握以下两点（最低配置要求

- 基本的 C/C++ 语法，毕竟 linux 的系统调用是用 C 语言写的，保证自己会用和能看懂即可。
- C++11 的特性（智能指针、function等），能够掌握 C++14/17 则更好，讲语言的新特性用到自己的项目中去也算是一个加分项，在项目中也可以提高自己对语言的掌握程度。

简单来说，只要你能在力扣上用 C++ 刷个一两百道题，就完全可以开始做项目了。

操作系统

- 基本的 Linux 命令，调试 WebServer 用，可以参考《鸟哥的 Linux 私房菜》这本书，讲得很全面，可以当字典使用。
- 常见的系统调用，主要就是 read/write AND socket 等函数，参考 CSAPP 来看即可。

计算机网络

- TCP 和 UDP 的连接机制及对应的函数
- 常见的服务器模式，单对单、多对单、多对多
- 抓包工具的简单使用，如 tcpdump

数据库

- 常用的 MySQL 语句，参考《MySQL 必知必会》
- 数据库的安装

参考书籍

- 游双. Linux高性能服务器编程[M]. 机械工业出版社, 2013.
- 陈硕. Linux多线程服务端编程：使用muduo C++网络库[M]. 电子工业出版社, 2013.
- 徐晓鑫. 后台开发:核心技术与应用实践[M]. 机械工业出版社, 2016.
- Bryant R E, David Richard O H, David Richard O H. Computer systems: a programmer's perspective[M]. Upper Saddle River: Prentice Hall, 2003.（《深入理解计算机系统》）

怎么找到一个靠谱的WebServer

好的，你现在终于决定去做一个 WebServer 了，但是咱们之前都没接触过相关的项目，完全白手起家不太现实，找一个现成的 WebServer 来参考是一个比较合理的做法，那么问题又来了，去哪找一个靠谱的 WebServer 来参考呢?

对于新手来说，当然是资料越详细越好，这里推荐牛客的两个 WebServer，配套有视频，非常容易上手，能够非常好地帮助我们梳理服务端开发的整个流程，缺点是整个项目的内容相对简单，需要我们对其进行性能优化和功能扩展（不然就真成了自己口中的烂大街了~）。

- linux高并发服务器开发-基础版
- linux高并发服务器开发-进阶版

此外，你还可以到 GitHub/Gitee 等开源社区上找相关项目，选高star的clone就完事了，至于如何在开源社区上搜索自己想要的资源，b站上面有许多相关的视频，我们这里就不再赘述了，下面推荐几个我在秋招过程中看见比较好的几个 WebServer（from GitHub）

- sylar-yin/sylar: C++高性能分布式服务器框架（附带博主的视频链接：[C++高级教程]从零开始开发服务器框架(sylar)）
- qinguoyi/TinyWebServer: Linux下C++轻量级WebServer服务器学习
- linyacool/WebServer: A C++ High Performance Web Server
- markparticle/WebServer: C++ Linux WebServer服务器

找到了合适的项目以后，我们可以通过以下指令去将项目克隆到本地（前提是你的电脑上已经安装了git）

```
git init  ## 将本地仓库初始化
git clone <url>  ## 将需要的项目从 github 上克隆下来，url为项目地址
```

拿到源代码先不要急着写

功能测试

此时我们已经拿到了 WebServer 的源代码，但是先不要着急去分析或者去写代码，我们首先要保证我们拿到的代码是能编译运行以及是功能正常的，否则等你耗费大半个月写好了代码，最后要么一直编译出错，要么反复 core dump，最后耗费许多精力去排查问题才发现源代码本来就是有问题的（当然这也是一种锻炼），我想是谁都会心态爆炸吧。

所以我们要从一开始就保证我们拿到的代码，不说有多高级多优雅，但是起码是能用的。首先，我们需要先打开项目的 `README.md` 文件，这个文件是项目的第一个入口点，通过这个文件我们可以了解到项目的背景、如何安装项目、如何使用项目、项目的版本迭代是怎么样的，下面我以qinguoyi/TinyWebServer: Linux下C++轻量级WebServer服务器学习 (github.com)的 `README.md` 文件为例来讲一下。

首先找到运行相关的部分：

快速运行（使用默认参数运行）

- 服务器测试环境

    - Ubuntu版本16.04
    - MySQL版本5.7.29
- 浏览器测试环境
    - Windows、Linux均可
    - Chrome
    - FireFox
    - 其他浏览器暂无测试
- 测试前确认已安装MySQL数据库（mysql的配置）

```
// 建立yourdb库
create database yourdb;

// 创建user表
USE yourdb;
CREATE TABLE user(
    username char(50) NULL,
    passwd char(50) NULL
)ENGINE=InnoDB;

// 添加数据
INSERT INTO user(username, passwd) VALUES('name', 'passwd');
```

- 修改main.cpp中的数据库初始化信息

```
//数据库登录名,密码,库名
string user = "root";
string passwd = "root";
string databasename = "yourdb";
```

- build（通过脚本编译项目）

```
sh ./build.sh
```

- 启动server（运行编译后的可执行文件）

```
./server
```

- 浏览器端（说明服务器默认端口是 9006）

```
ip:9006
```

个性化运行（使用自己的参数运行）

```
./server [-p port] [-l LOGWrite] [-m TRIGMode] [-o OPT_LINGER] [-s sql_num] [-t
thread_num] [-c close_log] [-a actor_model]
```

温馨提示:以上参数不是非必须，不用全部使用，根据个人情况搭配选用即可.

- -p，自定义端口号
  - 默认9006
- -l，选择日志写入方式，默认同步写入
  - 0，同步写入
  - 1，异步写入
- -m，listenfd和connfd的模式组合，默认使用LT + LT
  - 0，表示使用LT + LT
  - 1，表示使用LT + ET
  - 2，表示使用ET + LT
  - 3，表示使用ET + ET
- -o，优雅关闭连接，默认不使用
  - 0，不使用
  - 1，使用
- -s，数据库连接数量
  - 默认为8
- -t，线程数量
  - 默认为8
- -c，关闭日志，默认打开
  - 0，打开日志
  - 1，关闭日志
- -a，选择反应堆模型，默认Proactor
  - 0，Proactor模型
  - 1，Reactor模型

OK，现在通过项目作者的指示，我们终于把项目跑起来了，然后就可以对作者的项目进行功能测试了。我们首先看下作者提供的测试 demo：

demo 里面是通过浏览器去访问前面运行的服务器的，因此我们也跟着他来，在浏览器中输入URL：`113.54.159.5:9000`，然后你就会发现，一点反应都没有:)，此时你可能已经在怀疑项目代码是不是有问题了，但是先别急，在说别人有问题的时候，得先反思一下自己有没有问题。

根据计网的知识，一个基本的URL 是由协议 + IP地址 + 端口号 + 路径组成的，我们之前输入 URL 里面 的 IP地址我们没有见过（猜测是作者自己的 IP），并且端口号9000在 `README.md` 文件也没有出现过，可以猜测是作者自定义的端口号。

为了验证猜测，我们在终端里面 ping 一下作者的 IP 地址：

```
ping 113.54.159.5
```

毫无疑问，没有响应，说明这个 IP 地址已经失效了。但是我们是在本地环境下测试的，是否有办法在本地访问 WebServer 呢？此时就需要用到回环地址 `127.0.0.1` （不知道可以翻一下计网的书）

于是在浏览器里面URL：`127.0.0.1:9006` 或者 `localhost:9006`，然后我们终于可以看见 demo 中的页面了。

Attention：有些同学的 Linux 环境可能不是 Ubuntu 那种带有图形界面的 Linux 发行版本，而是 CentOs 那种命令行的形式，此时可以使用以下两个命令去向服务器发送 HTTP 请求：

```
curl "127.0.0.1:9006"
wget "127.0.0.1:9006"
```

只要能够获取到 HTTP 响应，就说明 WebServer 起码是能够完成响应 HTTP 这一个基本功能的，其余部分的测试我们后面再讲。

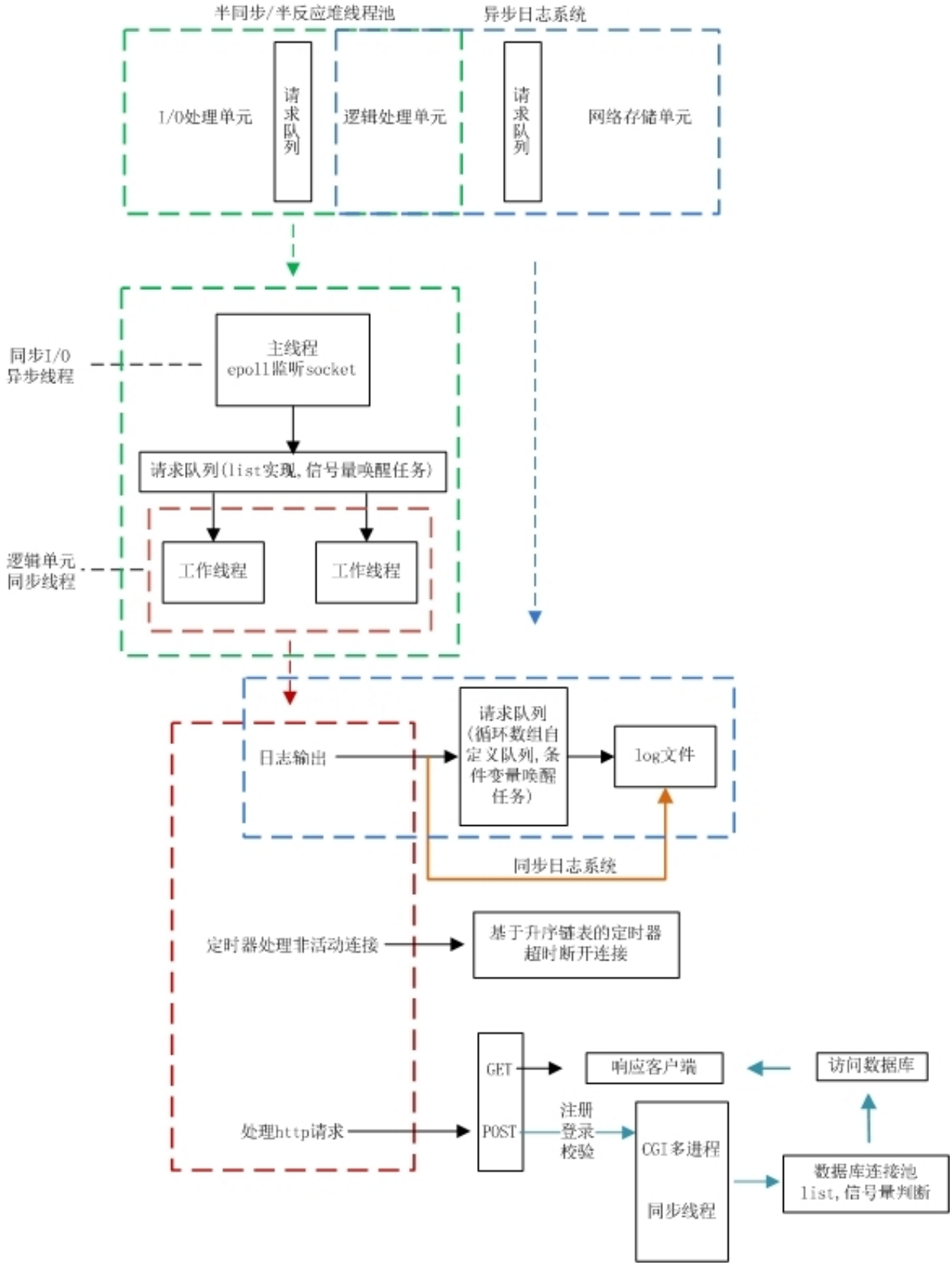
框架梳理

在进行了功能测试以后，我们确认了 clone 下来的项目起码是能够编译运行并且基本功能是正确的，但是此时我们还是先不要急着去写代码（就算想写也大概率不知道从哪个部分下手- -），因此在进行一个较大的项目时我们首先需要理清楚整个项目的框架，整个项目大概由哪些模块组成，各个模块的功能是什么，模块之间的关系具体是怎么样的。只能你脑袋里对整个项目框架有清晰的认识，你在写代码的时候才能够明确当前部分的代码应该实现怎样的功能，给下一部分的代码什么样的输入，而不会”猪脑过载“，陷入到局部细节中去。

那么问题来了，我们应该如何进行项目框架的梳理呢？首先我们需要明确一点，WebServer 已经是一个非常成熟的项目了（oppo的面试官还问过我，这个 WebServer 都已经出书了，为什么你们还在做），它的基本项目框架大致相同（reactor和proactor），只能具体功能和一些模块的实现略有差别。因此，如果你在做这个项目之前已经看过陈硕的《Linux 多线程服务端编程》这本书的话，你就会发现 GitHub 上的 高并发WebServer 只有 reactor 和模拟的 proactor 两种框架。

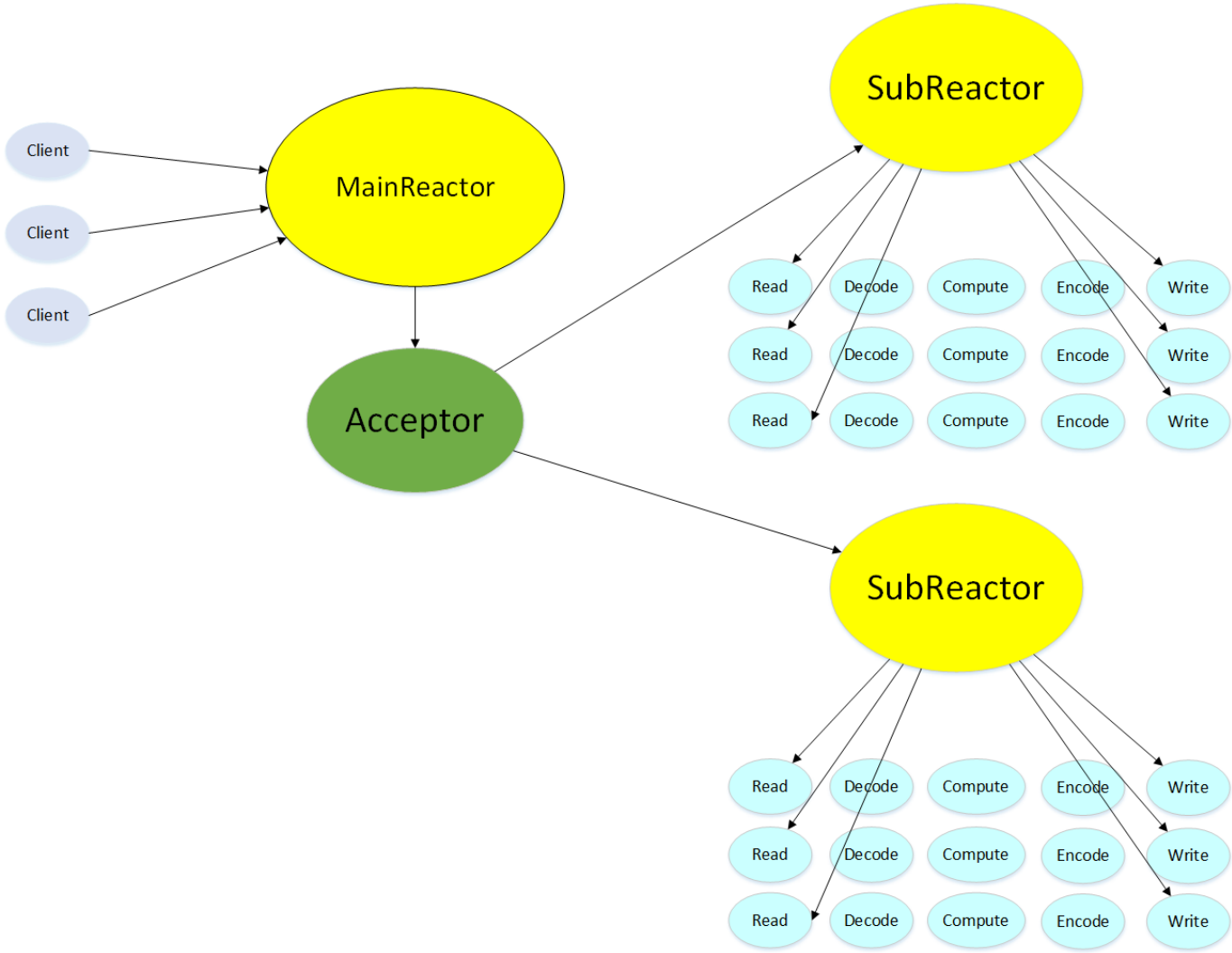
当然，如果你之前没看过相关的内容，对服务器的高并发模型也一无所知（我当时就是），也是可以继续做下去的，毕竟在实际中也不会有这么多的前置资料让你理解透彻再去进行，从一定程度上来看，我们在梳理别人代码的过程也可以看作对服务器并发模型的学习。我们首先还是先看下项目的文档，看下作者有没有提供代码的框架，有的话就简单了，对着框架图去看代码，可以迅速理清代码的模块组成，快速上手其他人的代码，然后就可以自己去写了。

首先以项目qinguoyi/TinyWebServer: Linux下C++轻量级WebServer服务器 (github.com)中的框架图为例。

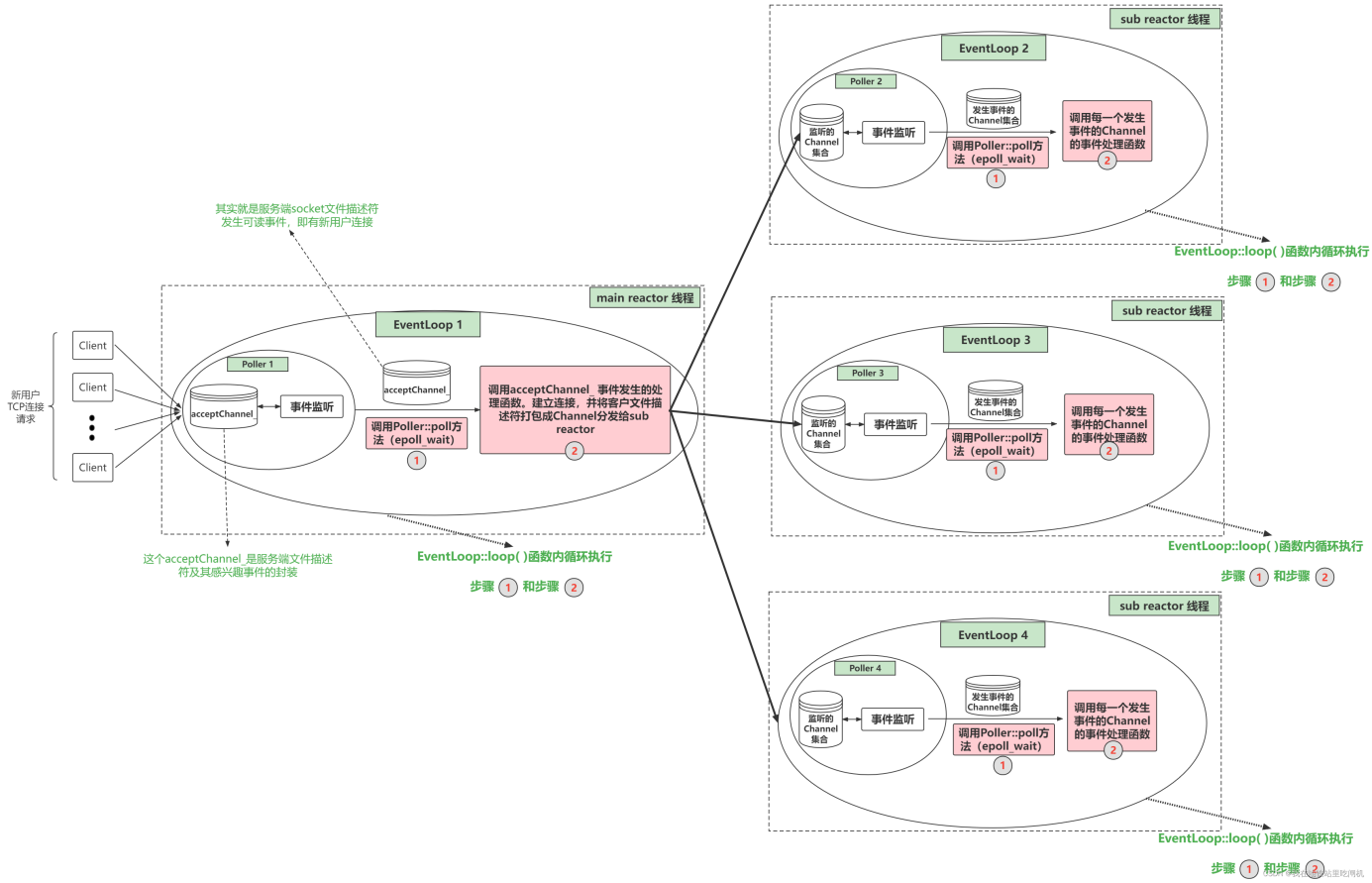


从图中我们可以看到，项目中搭建了一个半同步/反应堆线程池，在其中维护了一个请求队列，线程池中的主线程通过 epoll 来监听 socket，并且将请求队列中的任务分配给线程池中的工作线程，其中工作线程能够处理的任务分为日志输出、定时器处理非活动连接以及处理 HTTP 请求三种。

然后再看下项目[linyacool/WebServer: A C++ High Performance Web Server (github.com)](linyacool/WebServer: A C++ High Performance Web Server (github.com))中的框架图。



这个框架图就更加简洁明了，在 WebServer 中，许多 client 在 MainReactor 中得到了连接请求的响应，并与 WebServer 建立具体的连接。然后通过一个叫 Acceptor 的模块，将具体的连接分配给到一些叫做 SubReactor 的模块，在 SubReactor 中对具体的连接进行读、编码、计算、解码和写操作（即对 client 请求的响应）。这也是 muduo 网络库中所提出的 Multi-Reactor 并发框架，如下图（来源见水印）。



如果你对 muduo 中的并发模型感兴趣的话，可以详细看下前文提到的书《Linux多线程服务端编程》或者文章[长文梳理Muduo库核心代码及优秀编程细节剖析*陈硕muduo*我在地铁站里吃闸机的博客-CSDN博客]。

当然，在项目找不到相关的框架文档时，我们也可以从代码入手去分析。我以项目[linyacool/WebServer: A C++ High Performance Web Server (github.com)]为例大致讲一下如何去看代码。

```
.
└─WebServer-master
    │  build.sh
    │  CMakeLists.txt
    │  README.md
    │  并发模型.md
    │  测试及改进.md
    │  版本历史.md
    │  连接的维护.md
    │  遇到的困难.md
    │  项目目的.md
    ├─datum
    ├─old_version
    ├─WebBench
    └─WebServer
        │  Channel.cpp
        │  Channel.h
        │  CMakeLists.txt
        │  config.h
        │  Epoll.cpp
        │  Epoll.h
```

```
        │  EventLoop.cpp
        │  EventLoop.h
        │  EventLoopThread.cpp
        │  EventLoopThread.h
        │  EventLoopThreadPool.cpp
        │  EventLoopThreadPool.h
        │  HttpData.cpp
        │  HttpData.h
        │  Main.cpp
        │  Makefile
        │  Makefile.bak
        │  Server.cpp
        │  Server.h
        │  ThreadPool.cpp
        │  ThreadPool.h
        │  Timer.cpp
        │  Timer.h
        │  Util.cpp
        │  Util.h
        ├─base
        │  │  AsyncLogging.cpp
        │  │  AsyncLogging.h
        │  │  CMakeLists.txt
        │  │  Condition.h
        │  │  CountDownLatch.cpp
        │  │  CountDownLatch.h
        │  │  CurrentThread.h
        │  │  FileUtil.cpp
        │  │  FileUtil.h
        │  │  LogFile.cpp
        │  │  LogFile.h
        │  │  Logging.cpp
        │  │  Logging.h
        │  │  LogStream.cpp
        │  │  LogStream.h
        │  │  Log的设计.txt
        │  │  MutexLock.h
        │  │  noncopyable.h
        │  │  Thread.cpp
        │  │  Thread.h
        │  └─tests
        └─tests
```

我们首先看下程序的入口，即 `main()` 函数所在的位置——`Main.cpp` 文件。

```
int main(int argc, char *argv[]) {
    int threadNum = 4;
    int port = 80;
    std::string logPath = "./WebServer.log";
```

```
    // 参数解析
    int opt;
    const char *str = "t:l:p:";
    while ((opt = getopt(argc, argv, str)) != -1) {
        switch (opt) {
        case 't': {
            threadNum = atoi(optarg);
            break;
        }
        case 'l': {
            logPath = optarg;
            if (logPath.size() < 2 || optarg[0] != '/') {
            printf("logPath should start with \"/\"\n");
            abort();
            }
            break;
        }
        case 'p': {
            port = atoi(optarg);
            break;
        }
        default:
            break;
        }
    }
    Logger::setLogFileName(logPath);
// STL库在多线程上应用
#ifndef _PTHREADS
    LOG << "_PTHREADS is not defined !";
#endif
    EventLoop mainLoop;
    Server myHTTPServer(&mainLoop, threadNum, port);
    myHTTPServer.start();
    mainLoop.loop();
    return 0;
}
```

从 `main.cpp` 文件中可以看到，作者在运行服务器前先进行了参数配置、设置日志的相关参数（存储路径、日志等级等），然后实例化了一个主循环对象，初始化了 WebServer 的单例对象，最后启动 WebServer 并开启 主循环。

显然，这里最主要的就是两个函数 `myHTTPServer.start()` 和 `mainLoop.loop()`，所以着重分析这两个函数的功能即可。

首先看下 `myHTTPServer.start()`

```
void Server::start() {
    eventLoopThreadPool_->start();
    // acceptChannel_->setEvents(EPOLLIN | EPOLLET | EPOLLONESHOT);
    acceptChannel_->setEvents(EPOLLIN | EPOLLET);
    acceptChannel_->setReadHandler(bind(&Server::handNewConn, this));
    acceptChannel_->setConnHandler(bind(&Server::handThisConn, this));
    loop_->addToPoller(acceptChannel_, 0);
    started_ = true;
}
```

从函数命名就可以看出函数的主要功能主要有三部分：

- 启动线程池
- 设置了 `acceptChannel_` 的 handler，分别为读回调和连接回调
- 将 `acceptChannel_` 加入了 `poller`

可以看出是做了一下建立连接的工作。

然后再看下 `mainLoop.loop()`

```
void EventLoop::loop() {
    assert(!looping_);
    assert(isInLoopThread());
    looping_ = true;
    quit_ = false;
    // LOG_TRACE << "EventLoop " << this << " start looping";
    std::vector<SP_Channel> ret;
    while (!quit_) {
        // cout << "doing" << endl;
        ret.clear();
        ret = poller_->poll();
        eventHandling_ = true;
        for (auto& it : ret) it->handleEvents();
        eventHandling_ = false;
        doPendingFunctors();
        poller_->handleExpired();
    }
    looping_ = false;
}
```

从函数命名就可以看出函数的主要功能主要有四部分：

- 从 `poller` 中取出所有活动事件
- 调用活动事件的回调函数
- 执行额外的函数
- 执行超时的回调函数

明显这里地方才是真正进行业务处理的地方。然后再看下 `acceptChannel_` 的读回调函数 （有新连接时执行的函数）

```
void Server::handNewConn() {
    struct sockaddr_in client_addr;
    memset(&client_addr, 0, sizeof(struct sockaddr_in));
    socklen_t client_addr_len = sizeof(client_addr);
    int accept_fd = 0;
    while ((accept_fd = accept(listenFd_, (struct sockaddr *)&client_addr,
                               &client_addr_len)) > 0) {
        EventLoop *loop = eventLoopThreadPool_->getNextLoop();
        LOG << "New connection from " << inet_ntoa(client_addr.sin_addr) << ":"
            << ntohs(client_addr.sin_port);
        // 限制服务器的最大并发连接数
        if (accept_fd >= MAXFDS) {
            close(accept_fd);
            continue;
        }
        // 设为非阻塞模式
        if (setSocketNonBlocking(accept_fd) < 0) {
            LOG << "Set non block failed!";
            // perror("Set non block failed!");
            return;
        }
        setSocketNodelay(accept_fd);
        shared_ptr<HttpData> req_info(new HttpData(loop, accept_fd));
        req_info->getChannel()->setHolder(req_info);
        loop->queueInLoop(std::bind(&HttpData::newEvent, req_info));
    }
    acceptChannel_->setEvents(EPOLLIN | EPOLLET);
}
```

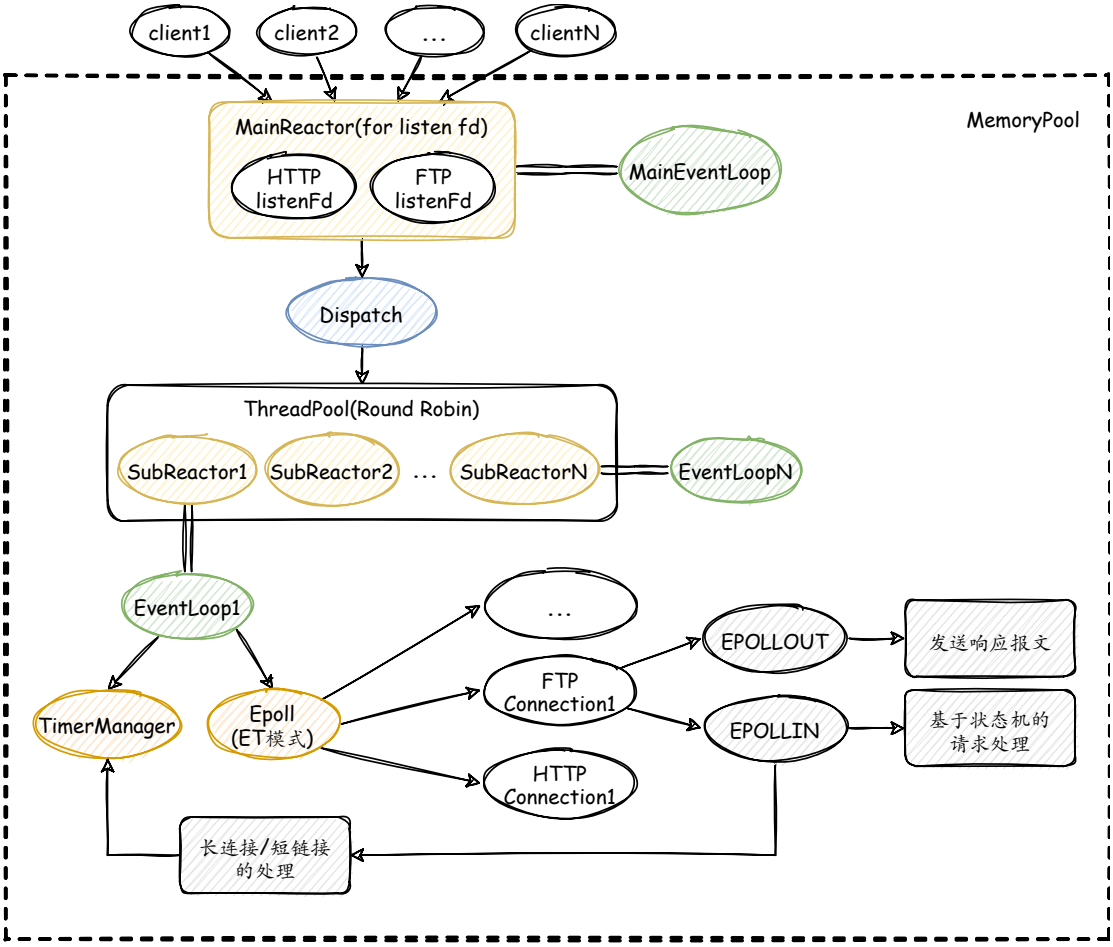
这个回调函数做了以下的工作：

- 通过 `accept(3)` 来建立 TCP 连接
- 从线程池中获取一个 `eventLoop`
- 将建立的连接放入 `eventLoop` 中

通过这样逐步分析，是不是发现，建立连接的方式与 muduo 的并发模型很像呢?

终于可以开始写了

经过一番摸索，我们终于画出自己的框架图，下面我将仿照 muduo 的框架对 WebServer 各个常见模块的原理进行讲解。

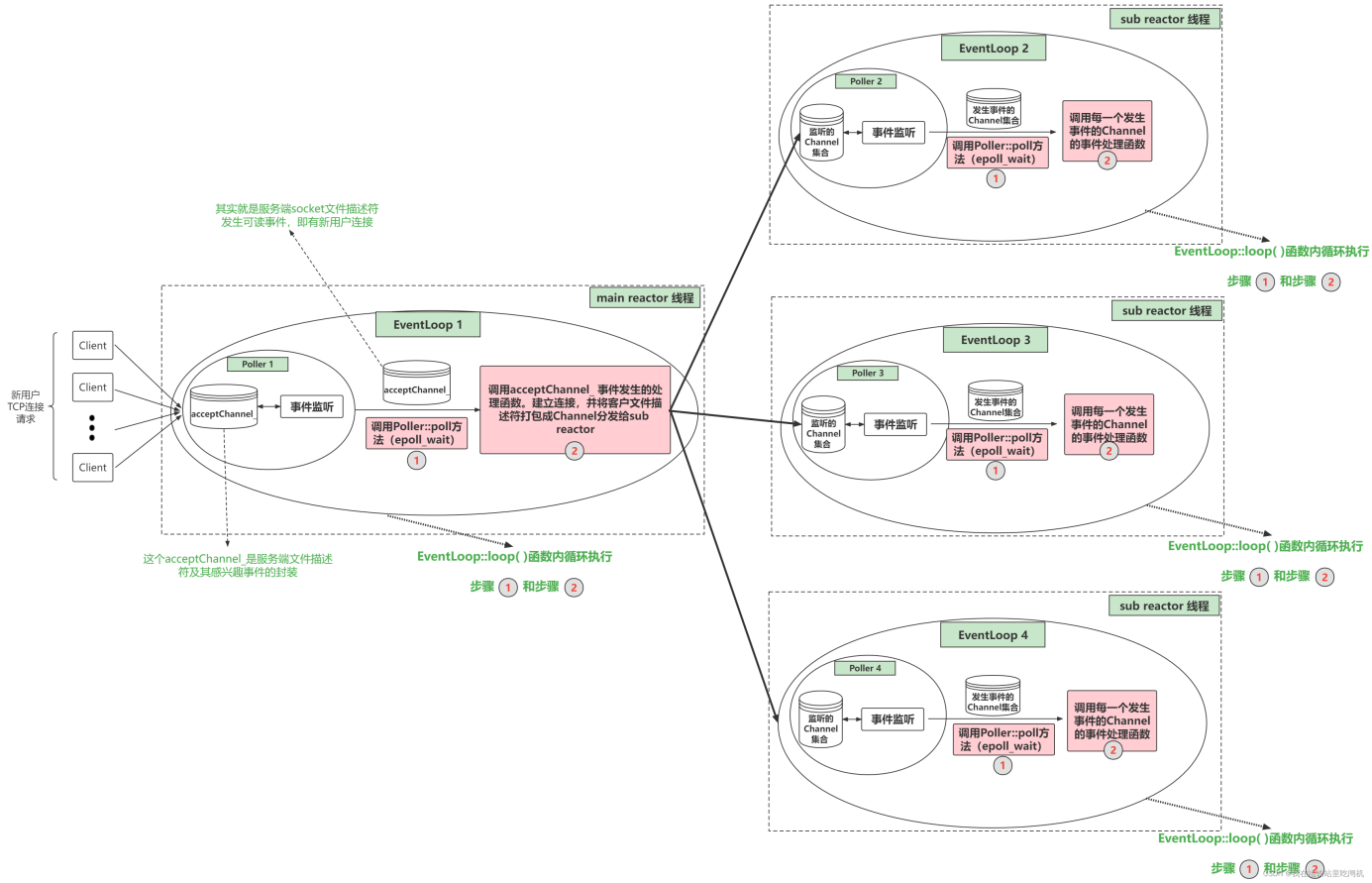


首先，许多 client 在访问 WebServer 时，并不是每个 client 都由一个服务端的线程/进程进行业务处理，这样在高并发（多个 client 同时访问服务端）的场景中，服务端的响应速度，并且服务端本身可以建立的线程/进程数也是有限的，这样的方式也很容易导致服务端的崩溃。

因此，muduo 使用非阻塞的poll/epoll（IO multiplexing 多路复用）轮训监听(Reactor)有无 SOCKET 读写IO事件， 将IO事件的处理回调函数分发到线程池中，实现异步返回结果。在多线程编程模型中采用了 "one loop per thread + thread pool" 的形式。一个线程中有且仅有一个EventLoop（也就是说每一个核的线程负责循环监听一组文件描述符的集合），这个线程称之为 IO 线程。如果长时间没有事件发生，IO线程将处于空闲状态，这时可以利用IO线程来执行一些额外的任务（利用定时器任务队列来处理超时连接），这就要求非阻塞的 poll/epoll 能够在无IO事件但有任务到来时能够被唤醒。

并发框架

并发框架这部分还是建议大家花时间去看一下《Linux多线程服务端编程》的第八章—muduo 网络库的设计与实现，里面对 Reactor 模型的原理和实现都讲得非常清晰。在具体的实现中，最核心的部分就是 `EventLoop` 、`Channel` 以及 `Poller` 三个类，其中 `EventLoop` 可以看作是对业务线程的封装，而 `Channel` 可以看作是对每个已经建立连接的封装（即 `accept(3)` 返回的文件描述符），三者的关系见下图（来源见水印）。



EventLoop

```
class EventLoop {
public:
    typedef std::function<void()> Function;
    // 初始化poller, event_fd, 给 event_fd 注册到 epoll 中并注册其事件处理回调
    EventLoop();
    ~EventLoop();

    // 开始事件循环 调用该函数的线程必须是该 EventLoop 所在线程，也就是 Loop 函数不能跨线程调用
    void Loop();
    // 停止 Loop
    void StopLoop();
    // 如果当前线程就是创建此EventLoop的线程 就调用callback(关闭连接 EpollDel) 否则就放入等待执行函数区
    void RunInLoop(Function&& func);
    // 把此函数放入等待执行函数区 如果当前是跨线程 或者正在调用等待的函数则唤醒
    void QueueInLoop(Function&& func);
    // 把fd和绑定的事件注册到epoll内核事件表
    void PollerAdd(std::shared_ptr<Channel> channel, int timeout = 0);
    // 在epoll内核事件表修改fd所绑定的事件
    void PollerMod(std::shared_ptr<Channel> channel, int timeout = 0);
    // 从epoll内核事件表中删除fd及其绑定的事件
    void PollerDel(std::shared_ptr<Channel> channel);
    // 只关闭连接(此时还可以把缓冲区数据写完再关闭)
    void ShutDown(std::shared_ptr<Channel> channel);
```

```
    bool is_in_loop_thread();

private:
    // 创建eventfd 类似管道的 进程间通信方式
    static int CreateEventfd();
    void HandleRead();                  // eventfd的读回调函数(因为event_fd写了数据，所以触发可读事件，从event_fd读数据)
    void HandleUpdate();                // eventfd的更新事件回调函数(更新监听事件)
    void WakeUp();                      // 异步唤醒SubLoop的epoll_wait(向event_fd中写入数据)
    void PerformPendingFunctions();     // 执行正在等待的函数(SubLoop注册EpollAdd连接套接字以及绑定事件的函数)

private:
    std::shared_ptr<Poller> poller_;            // io多路复用 分发器
    int event_fd_;                              // 用于异步唤醒 SubLoop 的 Loop 函数中的Poll(epoll_wait因为还没有注册fd会一直阻塞)
    std::shared_ptr<Channel> wakeup_channel_;   // 用于异步唤醒的 channel
    pid_t thread_id_;                           // 线程id

    mutable locker::MutexLock mutex_;
    std::vector<Function> pending_functions_;   // 正在等待处理的函数

    bool is_stop_;                              // 是否停止事件循环
    bool is_looping_;                           // 是否正在事件循环
    bool is_event_handling_;                    // 是否正在处理事件
    bool is_calling_pending_functions_;         // 是否正在调用等待处理的函数
};
```

从 `EventLoop` 的类定义中可以看出，除了一些状态量以外，每个 `EventLoop` 持有一个 `Poller` 的智能指针（对 epoll / poll 的封装），一个用于 `EventLoop` 之间通信的 `Channel`，自己的线程 id，互斥锁以及装有等待处理函数的 `vector` 。很明显，`EventLoop` 并不直接管理各个连接的 `Channel`（文件描述符的封装），而是通过 `Poller` 来进行的。`EventLoop` 中最核心的函数就是 `EventLoop::Loop()`。

```
void EventLoop::Loop() {
    // 开始事件循环 调用该函数的线程必须是该EventLoop所在线程
    assert(!is_looping_);
    assert(is_in_loop_thread());
    is_looping_ = true;
    is_stop_ = false;

    while (!is_stop_) {
        // 1、epoll_wait阻塞 等待就绪事件
        auto ready_channels = poller_->Poll();
        is_event_handling_ = true;
        // 2、处理每个就绪事件(不同channel绑定了不同的callback)
        for (auto& channel : ready_channels) {
            channel->HandleEvents();
        }
```

```
        is_event_handling_ = false;
        // 3、执行正在等待的函数(fd注册到epoll内核事件表)
        PerformPendingFunctions();
        // 4、处理超时事件 到期了就从定时器小根堆中删除(定时器析构会EpollDel掉fd)
        poller_->HandleExpire();
    }

    is_looping_ = false;
}
```

每个 `EventLoop` 对象都唯一绑定了一个线程，这个线程其实就在一直执行这个函数里面的 `while` 循环，这个 `while` 循环的大致逻辑比较简单。就是调用 `Poller::poll()` 函数获取事件监听器上的监听结果。接下来在 `Loop` 里面就会调用监听结果中每一个 `Channel` 的处理函数 `HandlerEvent()`。每一个 `Channel` 的处理函数会根据 `Channel` 中封装的实际发生的事件，执行 `Channel` 中封装的各事件处理函数。（比如一个 `Channel` 发生了可读事件，可写事件，则这个 `Channel` 的 `HandlerEvent()` 就会调用提前注册在这个 `Channel` 的可读事件和可写事件处理函数，又比如另一个 `Channel` 只发生了可读事件，那么 `HandlerEvent()` 就只会调用提前注册在这个 `Channel` 中的可读事件处理函数。

从中可以看到，每个 `EventLoop` 实际上就做了四件事

1. epoll_wait阻塞 等待就绪事件(没有注册其他fd时，可以通过event_fd来异步唤醒)
2. 处理每个就绪事件
3. 执行正在等待的函数(fd注册到epoll内核事件表)
4. 处理超时事件，到期了就从定时器小根堆中删除

Channel

在 TCP 网络编程中，想要通过 IO 多路复用（epoll / poll）监听某个文件描述符，就需要把这个 fd 和该 fd 感兴趣的事件通过 `epoll_ctl` **注册**到 IO 多路复用模块上。当 IO 多路复用模块监听到该 fd 发生了某个事件。事件监听器返回发生事件的 fd 集合（有哪些 fd 发生了事件）以及每个 fd 的事件集合（每个 fd 具体发生了什么事件）。

`Channel` 类则封装了一个 fd 和这个 fd 感兴趣事件以及 IO 多路复用模块监听到的每个 fd 的事件集合。同时Channel类还提供了**设置**该 fd 的感兴趣事件，以及将该fd及其感兴趣事件**注册**到事件监听器或从事件监听器上移除，以及保存了该 fd 的每种事件对应的处理函数。

每个 `Channel` 对象只属于一个 `EventLoop`，即只属于一个 IO 线程。只负责一个文件描述符（fd）的 IO 时间分发，但不拥有这个 fd。`Channel` 把不同的 IO 事件分发为不同的回调，回调用 C++11 的特性 `function` 表示。声明周期由拥有它的类负责。

```
class Channel {
public:
    typedef std::function<void()> EventCallBack;

    Channel();
    explicit Channel(int fd);
    ~Channel();

    // IO事件回调函数的调用接口
```

```
    // EventLoop中调用Loop开始事件循环 会调用Poll得到就绪事件
    // 然后依次调用此函数处理就绪事件
    void HandleEvents();
    void HandleRead();        // 处理读事件的回调
    void HandleWrite();       // 处理写事件的回调
    void HandleUpdate();      // 处理更新事件的回调
    void HandleError();       // 处理错误事件的回调

    int get_fd();
    void set_fd(int fd);

    // 返回weak_ptr所指向的shared_ptr对象
    std::shared_ptr<http::HttpConnection> holder();
    void set_holder(std::shared_ptr<http::HttpConnection> holder);

    // 设置回调函数
    void set_read_handler(EventCallBack&& read_handler);
    void set_write_handler(EventCallBack&& write_handler);
    void set_update_handler(EventCallBack&& update_handler);
    void set_error_handler(EventCallBack&& error_handler);

    void set_revents(int revents);
    int& events();
    void set_events(int events);
    int last_events();
    bool update_last_events();

private:
    int fd_;             // Channel的fd
    int events_;         // Channel正在监听的事件（或者说感兴趣的时间）
    int revents_;        // 返回的就绪事件
    int last_events_;    // 上一此事件（主要用于记录如果本次事件和上次事件一样 就没必要调用
epoll_mod）

    // weak_ptr是一个观测者（不会增加或减少引用计数）,同时也没有重载->,和*等运算符 所以不能直接使用
    // 可以通过lock函数得到它的shared_ptr（对象没销毁就返回，销毁了就返回空shared_ptr）
    // expired函数判断当前对象是否销毁了
    std::weak_ptr<http::HttpConnection> holder_;

    EventCallBack read_handler_;
    EventCallBack write_handler_;
    EventCallBack update_handler_;
    EventCallBack error_handler_;
};
```

从 `Channel` 的类定义中可以看出，每个 `Channel` 持有一个文件描述符，正在监听的事件，已经发生的事件（由 `Poller` 返回），以及各个事件（读、写、更新、错误）回调函数的 `Function` 对象。

总的来说，`Channel` 就是对 fd 事件的封装，包括注册它的事件以及回调。`EventLoop` 通过调用 `Channel::handleEvent()` 来执行 `Channel` 的读写事件。`Channel::handleEvent()` 的实现也非常简单，就是比较已经发生的事件（由 `Poller` 返回），来调用对应的回调函数（读、写、错误）。

```
// IO事件的回调函数 EventLoop中调用Loop开始事件循环 会调用Poll得到就绪事件
// 然后依次调用此函数处理就绪事件
void Channel::HandleEvents() {
    events_ = 0;
    // 触发挂起事件 并且没触发可读事件
    if ((revents_ & EPOLLHUP) && !(revents_ & EPOLLIN)) {
        events_ = 0;
        return;
    }
    // 触发错误事件
    if (revents_ & EPOLLERR) {
        HandleError();
        events_ = 0;
        return;
    }
    // 触发可读事件 | 高优先级可读 | 对端（客户端）关闭连接
    if (revents_ & (EPOLLIN | EPOLLPRI | EPOLLRDHUP)) {
        HandleRead();
    }
    // 触发可写事件
    if (revents_ & EPOLLOUT) {
        HandleWrite();
    }

    //处理更新监听事件(EpollMod)
    HandleUpdate();
}
```

`Channel` 这个类是不是很像文件描述符的保姆呢 :)

Poller

`Poller` 类的作用就是**负责监听文件描述符事件是否触发**以及**返回发生事件的文件描述符以及具体事件**。所以一个 `Poller` 对象对应一个 IO 多路复用模块。在 muduo 中，一个 `EventLoop` 对应一个 `Poller`。

```
class Epoll {
public:
    Epoll();
    ~Epoll();
    void epoll_add(const sp_Channel& request);
    void epoll_mod(const sp_Channel& request);
    void epoll_del(const sp_Channel& request);
    void poll(std::vector<sp_Channel>& req);
```

```
private:
    int epollFd_;
    std::vector<epoll_event> events_;           // epoll_wait()返回的活动事件都放在这个数组
里
    std::unordered_map<int, sp_Channel> channelMap_;
};
```

`Poller` 的主要成员变量就三个：

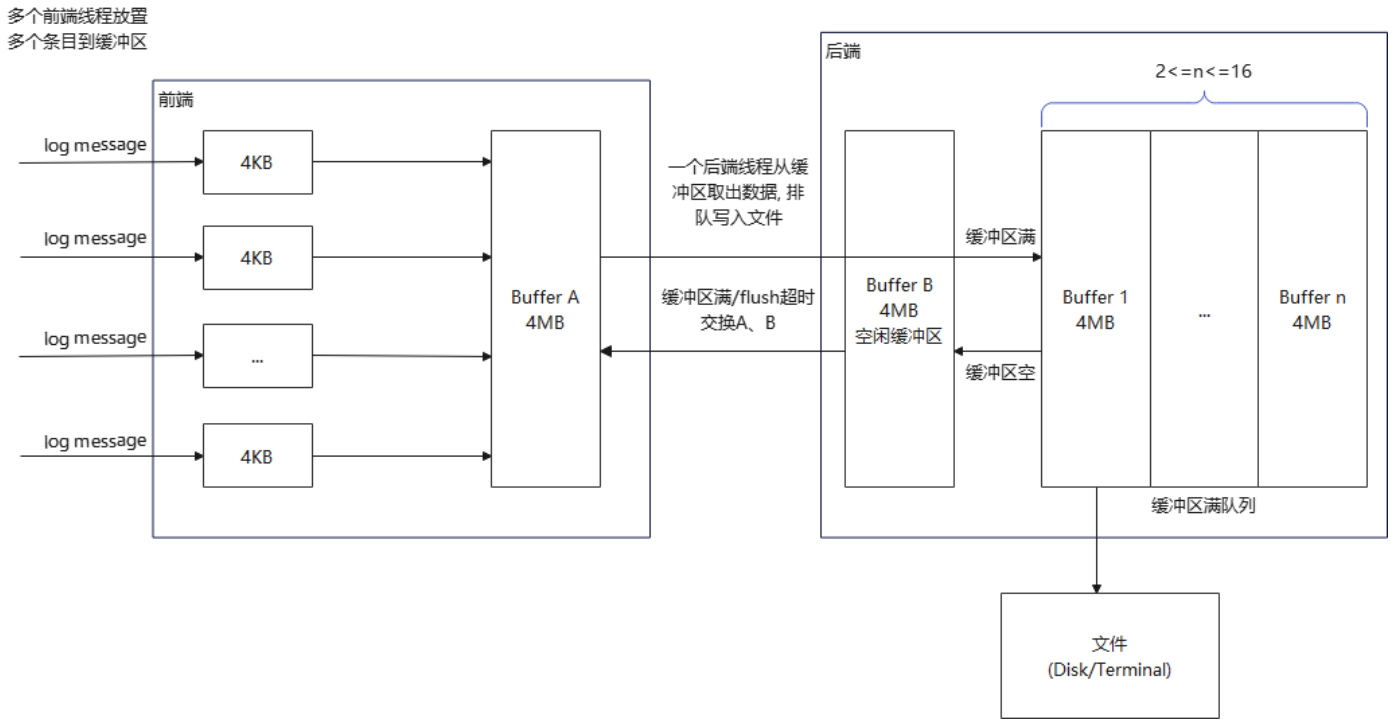
1. `epollFd_`：就是用 `epoll_create` 方法返回的 epoll 句柄，这个是常识。
2. `events_`：存放`epoll_wait()` 返回的活动事件（是一个结构体）
3. `channelMap_`：这个变量是 `std::unordered_map<int, std::shared_ptr<Channel>>` 类型，负责记录文件描述符fd -> Channel 的映射，也帮忙保管所有注册在你这个 `Poller` 上的 `Channel`。

其他函数无非就是对 `Epoll_ctl(4)` 和 `Epoll_wait(4)` 的封装。

```
void Epoll::poll(std::vector<sp_Channel>& req) {
    int event_count =
        Epoll_wait(epollFd_, &*events_.begin(), events_.size(), EPOLLWAIT_TIME);
  for(int i = 0; i < event_count; ++i) {
    int fd = events_[i].data.fd;
    sp_Channel temp = channelMap_[fd];
    temp->setRevents(events_[i].events);
    req.emplace_back(std::move(temp));
  }
    // LOG << "Epoll finished";
}
```

`Epoll::poll(1)` 这个函数可以说是 `Poller` 的核心了，当外部调用 `poll` 方法的时候，该方法底层其实是通过 `epoll_wait` 获取这个事件监听器上发生事件的 fd 及其对应发生的事件，我们知道每个 fd 都是由一个Channel封装的，通过哈希表`channelMap_`可以根据 fd 找到封装这个 fd 的 `Channel`。将 IO 多路复用模块监听到该 fd 发生的事件写进这个 Channel 中的 revents 成员变量中。然后把这个 Channel 装进 `req` 中。这样，当外界调用完 `poll` 之后就能拿到 IO 多路复用模块的**监听结果**（`std::vector<sp_Channel>& req`）。

日志系统



服务器的日志系统是一个多生产者，单消费者的任务场景：多生产者负责把日志写入缓冲区，单消费者负责把缓冲区中数据写入文件。如果只用一个缓冲区，不光要同步各个生产者,还要同步生产者和消费者。而且最重要的是需要保证生产者与消费者的并发，也就是前端不断写日志到缓冲区的同时，后端可以把缓冲区写入文件。

LOG 的实现参照了 muduo，但是比 muduo 要简化一点，大致的实现如上图所示，看图还是比较好懂的。

- 首先是 `Logger` 类，`Logger` 类里面有 Impl 类，其实具体实现是 Impl 类，我也不懂muduo为何要再封装一层，那么我们来说说 Impl 干了什么，在初始化的时候 Impl 会把时间信息存到 `LogStream` 的缓冲区里，在我们实际用 Log 的时候，实际写入的缓冲区也是 `LogStream`，在析构的时候 Impl 会把当前文件和行数等信息写入到 `LogStream`，再把 `LogStream` 里的内容写到 `AsyncLogging` 的缓冲区中，当然这时候我们要先开启一个后端线程用于把缓冲区的信息写到文件里。
- `LogStream` 类，里面其实就一个 Buffer 缓冲区，是用来暂时存放我们写入的信息的。还有就是重载运算符，因为我们采用的是 C++ 的流式风格。
- `AsyncLogging` 类，最核心的部分，在多线程程序中写 Log 无非就是前端往后端写，后端往硬盘写，首先将 LogStream 的内容写到了 AsyncLogging 缓冲区里，也就是前端往后端写，这个过程通过 append 函数实现，后端实现通过 threadfunc 函数，两个线程的同步和等待通过互斥锁和条件变量来实现，具体实现使用了双缓冲技术。
- 双缓冲技术的基本思路：准备两块 buffer，A 和 B,前端往 A 写数据，后端从 B 里面往硬盘写数据，当 A 写满后，交换 A 和 B，如此反复。使用两个 buffer 的好处是在新建日志消息的时候不必等待磁盘文件操作，也避免每条新日志消息都触发后端日志线程。换句话说，前端不是将一条条日志消息分别送给后端，而是将多条日志消息拼接成一个大的 buffer 传送给后端，相当于批处理，减少了线程唤醒的开销。不过实际的实现的话和这个还是有点区别，具体看代码吧。

我们首先看下 `logStream` 类的实现，`logStream` 类的主要作用是将各个类型的数据转换为 char 的形式放入字符数组中（也就是前端日志写入 Buffer A 的这个过程），方便后端线程写入硬盘。

```
class LogStream : noncopyable {
    typedef LogStream self;
public:
    typedef FixedBuffer<kSmallBuffer> Buffer;
```

```
    self& operator<< (bool v) {
        buffer_.append(v ? "1" : "0", 1);
        return *this;
    }

    self& operator<< (short);
    self& operator<< (unsigned short);
    self& operator<< (int);
    self& operator<< (unsigned int);
    self& operator<< (long);
    self& operator<< (unsigned long);
    self& operator<< (long long);
    self& operator<< (unsigned long long);
    self& operator<< (const void*);
    self& operator<< (float v) {
        *this << static_cast<double>(v);
        return *this;
    }
    self& operator<< (double);
    self& operator<< (long double);
    self& operator<< (char v) {
        buffer_.append(&v, 1);
        return *this;
    }
    self& operator<< (const char* str) {
        if(str)
            buffer_.append(str, strlen(str));
        else
            buffer_.append("(null)", 6);
        return *this;
    }
    self& operator<< (const unsigned char* str) {
        // reinterpret_cast用在任意指针(或引用)类型之间的转换
        return operator<<(reinterpret_cast<const char*>(str));
    }
    self& operator<< (const std::string& v) {
        buffer_.append(v.c_str(), v.size());
        return *this;
    }

    void append(const char* data, int len) { buffer_.append(data, len); }
    const Buffer& buffer() const { return buffer_; }
    void resetBuffer() { buffer_.reset(); }
private:
    template <typename T>
        void formatInteger(T);
    Buffer buffer_;
    static const int kMaxNumericSize = 32;
```

```
};
```

`logStream` 类实际上只持有一个缓冲区 `Buffer` （这个缓冲区的实现比较简单，大家看代码就能懂了），然后重载对流式运算符 `<<` 进行了重载。这里需要注意的是，由于日志输入的类型繁多，为了后端写入方便（也为了缓冲区的格式统一），需要将输入的数据转换为 `char` 字符类型再进行写入。这样就很有意思了， `int` 类型的转成字符数组很容易实现（leetcode 简单题的水平），但如果是浮点数或者指针呢？这里的实现很有意思，大家可以思考一下。

然后是 `AsynLogging` 类，这个类的作用则是将从前端获得的 Buffer A 放入 后端的 Buffer B中，并且将 Buffer B 的内容最终写入到磁盘中（也就是整个后端所作的内容）。

```
class AsyncLogging : noncopyable {
public:
    AsyncLogging(const std::string basename, int flushInterval = 2);
    ~AsyncLogging() {
        if(running_) stop();
    }
    void append(const char* logline, int len);
    void start();
    void stop();

private:
    typedef FixedBuffer<kLargeBuffer> Buffer;
    typedef std::vector<std::shared_ptr<Buffer>> BufferVector;
    typedef std::shared_ptr<Buffer> BufferPtr;

    void threadFunc();              //后端日志线程函数，用于把缓冲区日志写入文件

    const int flushInterval_;       //超时时间，每隔一段时间写日志
    bool running_;                  //FIXME:std::atomic<bool> running_;
    const std::string basename_;    //日志名字
    Thread thread_;                 //后端线程，用于将日志写入文件
    MutexLock mutex_;
    Condition cond_;
    BufferPtr currentBuffer_;
    BufferPtr nextBuffer_;
    BufferVector buffers_;
    CountDownLatch latch_;          //倒计时，用于指示日志记录器何时开始工作
};
```

这里有几个比较关键的内容：

1. `thread_` ： 后端通过一个单独的线程来实现将日志写入文件的，这就是那个线程对象。
2. `currentBuffer_` 和 `nextBuffer_` ： 双缓冲区，减少前端等待的开销
3. `buffers_` ： 缓冲区队列，实际写入文件的内容也是从这里拿的。

线程函数的实现也比较有意思，大家可以看一下。

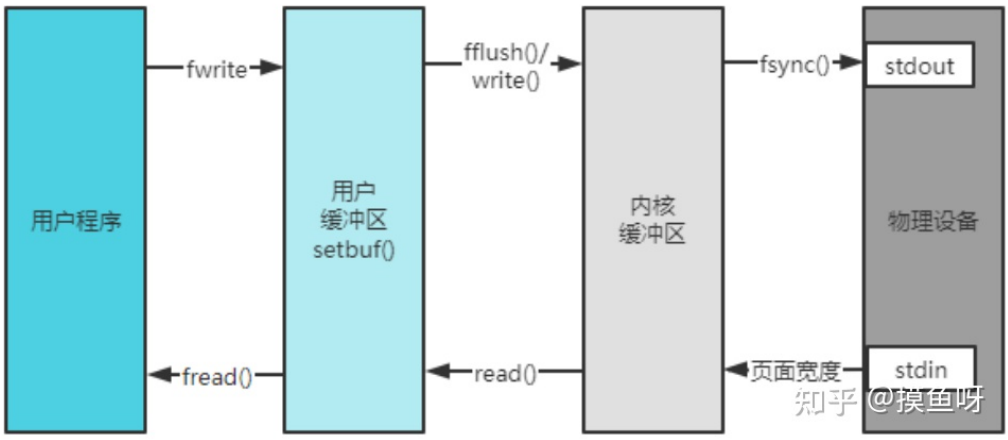
```
void AsyncLogging::threadFunc() {
```

```
    assert(running_ == true);
    latch_.countDown();
    LogFile output(basename_);  //LogFile用于将日志写入文件
    BufferPtr newBuffer1(new Buffer);
    BufferPtr newBuffer2(new Buffer);
    newBuffer1->bzero();
    newBuffer2->bzero();
    BufferVector buffersToWrite;    //该vector属于后端线程，用于和前端的buffers进行交换
    buffersToWrite.reserve(16);

    while(running_) {
        assert(newBuffer1 && newBuffer1->length() == 0);
        assert(newBuffer2 && newBuffer2->length() == 0);
        assert(buffersToWrite.empty());

        //将前端buffers_中的数据交换到buffersToWrite中
        {
            MutexLockGuard lock(mutex_);
            //每隔3s，或者currentBuffer满了，就将currentBuffer放入buffers_中
            if(buffers_.empty())
                cond_.waitForSeconds(flushInterval_);
            buffers_.push_back(std::move(currentBuffer_));
            // buffers_.push_back(currentBuffer_);
            // currentBuffer_.reset();
            currentBuffer_ = std::move(newBuffer1);

            buffersToWrite.swap(buffers_);
            if(!nextBuffer_)
                nextBuffer_ = std::move(newBuffer2);
        }

        assert(!buffersToWrite.empty());

        //如果队列中buffer数目大于25，就删除多余数据
        //避免日志堆积：前端日志记录过快，后端来不及写入文件
        if (buffersToWrite.size() > 25) {
            //TODO：删除数据时加错误提示
            //只留原始的两个buffer，其余的删除
            buffersToWrite.erase(buffersToWrite.begin() + 2, buffersToWrite.end());
        }

        for (const auto& buffer : buffersToWrite) {
            output.append(buffer->data(), buffer->length());
        }

        //重新调整buffersToWrite的大小，仅保留两个原始buffer
        if (buffersToWrite.size() > 2) {
            buffersToWrite.resize(2);
        }
```

```
        if (!newBuffer1) {
            assert(!buffersToWrite.empty());
            newBuffer1 = buffersToWrite.back();
            buffersToWrite.pop_back();
            newBuffer1->reset();
        }

        if (!newBuffer2) {
            assert(!buffersToWrite.empty());
            newBuffer2 = buffersToWrite.back();
            buffersToWrite.pop_back();
            newBuffer2->reset();
        }

        buffersToWrite.clear();
        output.flush();
    }
    output.flush();
}
```

线程函数做的事情也比较简单：

1. 每隔3s，或者 Buffer B 满了，就将 Buffer B 放入 buffers_ （缓冲区队列）中，这里涉及到多个线程的同步，需要加互斥锁，还需要判断缓冲区队列是否已经满了，里面可以用 `std::move` 和 `std::swap` 进一步提高效率（想想为什么）。
2. 最后再将 buffers_ （缓冲区队列）里的每个 Buffer 放到文件输出区 output里，再 flush 一下就可以了。（至于为什么要 flush，可以看下面这个图，来源见水印）



> 一些问题

1. **什么时候切换写到另一个日志文件？**前一个 Buffer 已经写满了，则交换两个 Buffer（写满的 Buffer 置空）。
2. **日志串写入过多，日志线程来不及消费，怎么办？**直接丢掉多余的日志 Buffer，腾出内存，防止引起程序故障。
3. **什么时候唤醒日志线程从 Buffer 中取数据？**其一是超时，其二是前端写满了一个或者多个 Buffer。

之后进一步将 `logStream` 类和 `AsynLogging` 类封装为 `Logger` 类就可以完成我们了日志库了。

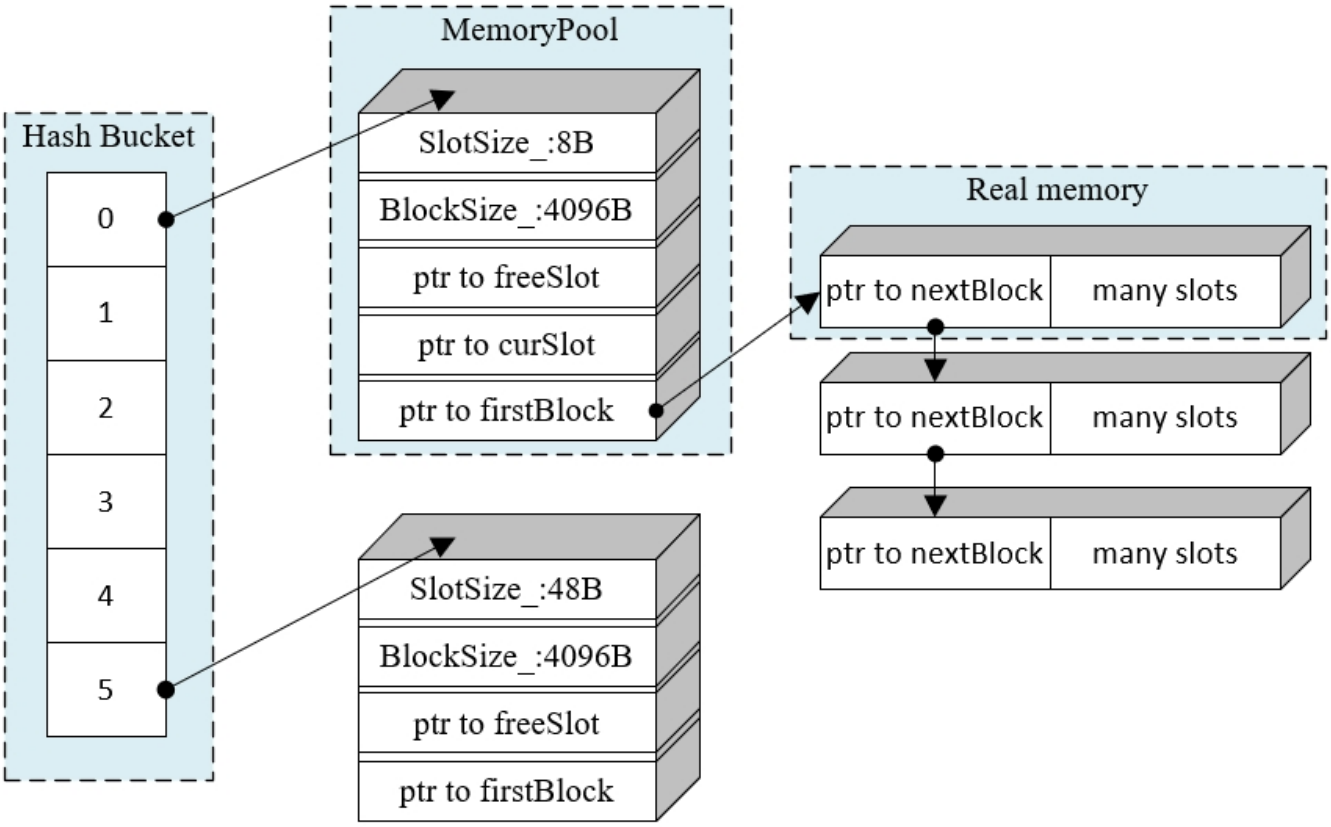
```
class Logger {
public:
    Logger(const char* filename, int line);
    ~Logger();
    LogStream& stream() { return impl_.stream_; }

    static void setLogFileName(std::string fileName) { logFileName_ = fileName; }
    static std::string getLogFileName() { return logFileName_; }
private:
    //Logger的内部实现类Impl：负责把日志头信息写入logstream
    //包含：时间戳，LogStream数据流，源文件行号，源文件名字
    //TODO:加入日志级别
    class Impl {
     public:
         Impl(const char* fileName, int line);
         void formatTime();

         LogStream stream_;
         int line_;
         std::string basename_;
    };
    Impl impl_;
    static std::string logFileName_;
};


// C/C++提供了三个宏来定位程序运行时的错误
// __FUNCTION__:返回当前所在的函数名
// __FILE__:返回当前的文件名
// __LINE__:当前执行行所在行的行号
#define LOG Logger(__FILE__, __LINE__).stream()
```

这里仿照 muduo 用了一个比较有意思的方法，通过宏定义 `#define LOG Logger(__FILE__, __LINE__).stream()` 使得我们在使用日志写入语句 `LOG << info;` 之类的内容时，先构建一个临时的 `Logger` 对象，在构造时生成详细的日志信息，由于临时对象会在语句结束后析构，我们可以在析构的时候再将日志真正地写入文件，来保证实现的简洁性。

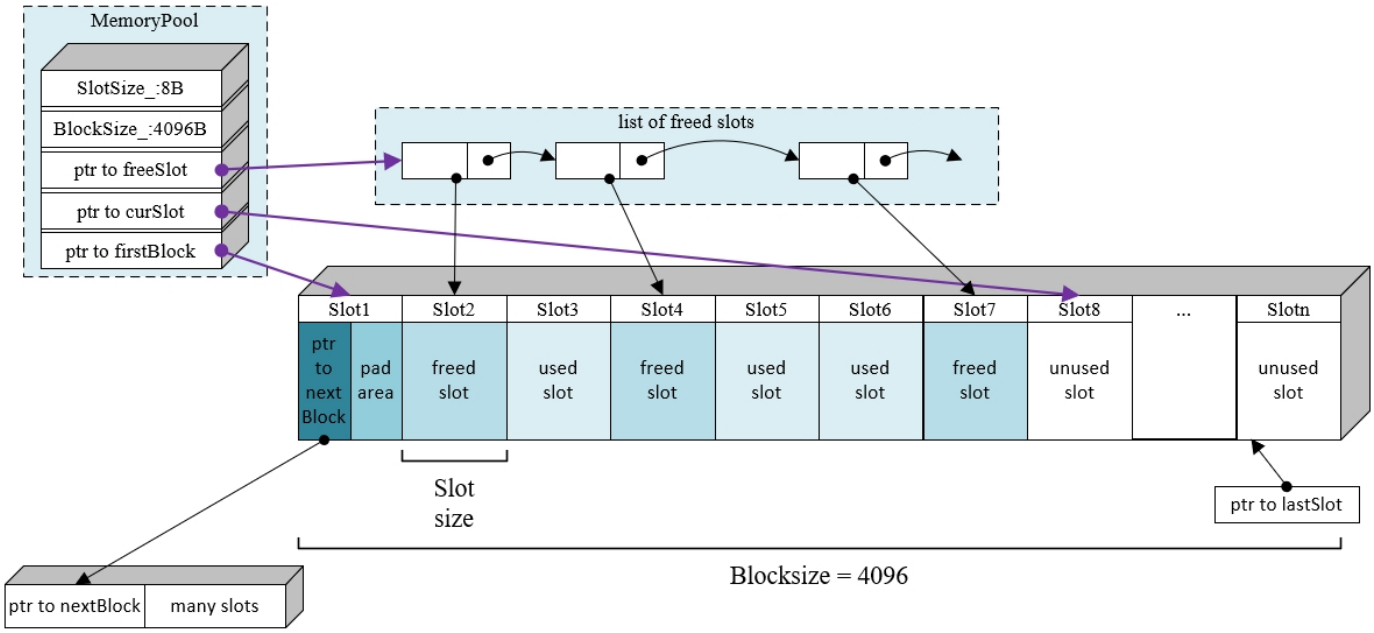
```
Logger::Logger(const char* fileName, int line)
    : impl_(fileName, line) { }

Logger::~Logger() {
    impl_.stream_ << " -- " << impl_.basename_ << ':' << impl_.line_ << '\n';
    const LogStream::Buffer& buf(stream().buffer());
    output(buf.data(), buf.length());
}
```

内存池

内存池中哈希桶的思想借鉴了STL allocator，具体每个小内存池的实现参照了GitHub上的项目 [An easy to use and efficient memory pool allocator written in C++.]



主要框架如上图所示，主要就是维护一个哈希桶 `MemoryPools`，里面每项对应一个内存池 `MemoryPool`，哈希桶中每个内存池的块大小 `BlockSize` 是相同的（4096字节，当然也可以设置为不同的），但是每个内存池里每个块分割的大小（槽大小） `SlotSize` 是不同的，依次为8,16,32,...,512字节（需要的内存超过512字节就用 `new/malloc` ），这样设置的好处是可以保证内存碎片在可控范围内。



内存池的内部结构如上图所示，主要的对象有：指向第一个可用内存块的指针`Slot currentBlock`（也是图中的`ptr to firstBlock`，图片已经上传懒得改了），被释放对象的slot链表`Slot freeSlot`，未使用的slot链表`Slot* currentSlot`，下面讲下具体的作用：

- `Slot currentBlock`：内存池实际上是一个一个的`Block`以链表的形式连接起来，每一个`Block`是一块大的内存，当内存池的内存不足的时候，就会向操作系统申请新的`block`加入链表。
- `Slot freeSlot`：链表里面的每一项都是对象被释放后归还给内存池的空间，内存池刚创建时`freeSlot`是空的。用户创建对象，将对象释放时，把内存归还给内存池，此时内存池不会将内存归还给系统（`delete/free`），而是把指向这个对象的内存的指针加到`freeSlot`链表的前面（前插），之后用户每次申请内存时，`memoryPool`就先在这个`freeSlot`链表里面找。
- `Slot curretSlot`：用户在创建对象的时候，先检查`freeSlot`是否为空，不为空的时候直接取出一项作为分配出的空间。否则就在当前`Block`将`currentSlot`所指的内存分配出去，如果`Block`里面的内存已经使用完，就向操作系统申请一个新的`Block`。

有了上面的两个图，再看代码就很好理解了。

```
#define BlockSize 4096

struct Slot {
    Slot* next;
};

class MemoryPool {
public:
    MemoryPool();
    ~MemoryPool();

    void init(int size);

    // 分配或收回一个元素的内存空间
    Slot* allocate();
    void deAllocate(Slot* p);
private:
    int slotSize_;          // 每个槽所占字节数

    Slot* currentBolck_;    // 内存块链表的头指针
    Slot* currentSlot_;     // 元素链表的头指针
    Slot* lastSlot_;        // 可存放元素的最后指针
    Slot* freeSlot_;        // 元素构造后释放掉的内存链表头指针

    MutexLock mutex_freeSlot_;
    MutexLock mutex_other_;

    size_t padPointer(char* p, size_t align);   // 计算对齐所需空间
    Slot* allocateBlock();      // 申请内存块放进内存池
    Slot* nofree_solve();
};

void init_MemoryPool();
```

```
void* use_Memory(size_t size);
void free_Memory(size_t size, void* p);
MemoryPool& get_MemoryPool(int id);

template<typename T, typename... Args>
T* newElement(Args&&... args) {
    T* p;
    if((p = reinterpret_cast<T *>(use_Memory(sizeof(T)))) != nullptr)
        // new(p) T1(value);
        // placement new:在指针p所指向的内存空间创建一个T1类型的对象，类似与realloc
        // 把已有的空间当成一个缓冲区来使用，减少了分配空间所耗费的时间
        // 因为直接用new操作符分配内存的话，在堆中查找足够大的剩余空间速度是比较慢的
        new(p) T(std::forward<Args>(args)...); // 完美转发

    return p;
}

// 调用p的析构函数，然后将其总内存池中释放
template<typename T>
void deleteElement(T* p) {
    // printf("deleteElement...\n");
    if(p)
        p->~T();
    free_Memory(sizeof(T), reinterpret_cast<void *>(p));
    // printf("deleteElement success\n");
}
```

这里接口定义得不太好，而且应该用 namespace 限定一下的，大家有兴趣的话可以优化一下。。

```
MemoryPool::MemoryPool() {}

MemoryPool::~MemoryPool() {
    Slot* cur = currentBolck_;
    while(cur) {
        Slot* next = cur->next;
        // free(reinterpret_cast<void *>(cur));
        // 转化为 void 指针，是因为 void 类型不需要调用析构函数,只释放空间
        operator delete(reinterpret_cast<void*>(cur));
        cur = next;
    }
}

void MemoryPool::init(int size) {
    assert(size > 0);
    slotSize_ = size;
    currentBolck_ = NULL;
    currentSlot_ = NULL;
    lastSlot_ = NULL;
    freeSlot_ = NULL;
```

```
}

// 计算对齐所需补的空间
inline size_t MemoryPool::padPointer(char* p, size_t align) {
    size_t result = reinterpret_cast<size_t>(p);
    return ((align - result) % align);
}

Slot* MemoryPool::allocateBlock() {
    char* newBlock = reinterpret_cast<char *>(operator new(BlockSize));

    char* body = newBlock + sizeof(Slot*);
    // 计算为了对齐需要空出多少位置
    // size_t bodyPadding = padPointer(body, sizeof(slotSize_));
    size_t bodyPadding = padPointer(body, static_cast<size_t>(slotSize_));

    // 注意：多个线程（eventLoop 共用一个MemoryPool）
    Slot* useSlot;
    {
        MutexLockGuard lock(mutex_other_);
        // newBlock接到Block链表的头部
        reinterpret_cast<Slot *>(newBlock)->next = currentBolck_;
        currentBolck_ = reinterpret_cast<Slot *>(newBlock);
        // 为该Block开始的地方加上bodyPadding个char* 空间
        currentSlot_ = reinterpret_cast<Slot *>(body + bodyPadding);
        lastSlot_ = reinterpret_cast<Slot *>(newBlock + BlockSize - slotSize_ + 1);
        useSlot = currentSlot_;

        // slot指针一次移动8个字节
        currentSlot_ += (slotSize_ >> 3);
    }

    return useSlot;
}

Slot* MemoryPool::nofree_solve() {
    if(currentSlot_ >= lastSlot_)
        return allocateBlock();
    Slot* useSlot;
    {
        MutexLockGuard lock(mutex_other_);
        useSlot = currentSlot_;
        currentSlot_ += (slotSize_ >> 3);
    }

    return useSlot;
}
```

```
Slot* MemoryPool::allocate() {
    if(freeSlot_) {
        {
            MutexLockGuard lock(mutex_freeSlot_);
            if(freeSlot_) {
                Slot* result = freeSlot_;
                freeSlot_ = freeSlot_->next;
                return result;
            }
        }
    }

    return nofree_solve();
}

inline void MemoryPool::deAllocate(Slot* p) {
    if(p) {
        // 将slot加入释放队列
        MutexLockGuard lock(mutex_freeSlot_);
        p->next = freeSlot_;
        freeSlot_ = p;
    }
}

MemoryPool& get_MemoryPool(int id) {
    static MemoryPool memorypool_[64];
    return memorypool_[id];
}

// 数组中分别存放Slot大小为8，16，...，512字节的BLock链表
void init_MemoryPool() {
    for(int i = 0; i < 64; ++i) {
        get_MemoryPool(i).init((i + 1) << 3);
    }
}

// 超过512字节就直接new
void* use_Memory(size_t size) {
    if(!size)
        return nullptr;
    if(size > 512)
        return operator new(size);

    // 相当于(size / 8)向上取整
    return reinterpret_cast<void *>(get_MemoryPool(((size + 7) >> 3) - 1).allocate());
}

void free_Memory(size_t size, void* p) {
    if(!p)  return;
```

```
    if(size > 512) {
        operator delete (p);
        return;
    }
    get_MemoryPool(((size + 7) >> 3) - 1).deAllocate(reinterpret_cast<Slot *>(p));
}
```

最后使用的话就非常简单了，包含了头文件以后，调用接口 `newElement()` 和 `deleteElement()` 申请内存即可，这里放一个简单的测试 demo。

```
#include "../MemoryPool.h"
#include <memory>

using namespace std;

class Person {
public:
    int id_;
    std::string name_;
public:
    Person(int id, std::string name) : id_(id), name_(name) {
        printf("构造函数调用\n");
    }
    ~Person() {
        printf("析构函数调用\n");
    }
};

void test01() {
    printf("creating a person\n");
    shared_ptr<Person> p1(newElement<Person>(11, "Lawson"), deleteElement<Person>);
    printf("sizeof(name_) = %d\n", sizeof(p1->name_));
    printf("sizeof(Person) = %d\n", sizeof(Person));
}

int main() {
    test01();
    return 0;
}
```

> 线程安全

内存池本身是线程安全的：在分配和删除 `Block` 和获取 `Slot` 时加锁 `MutexLockGuard lock(mutex_);`

> 缺陷

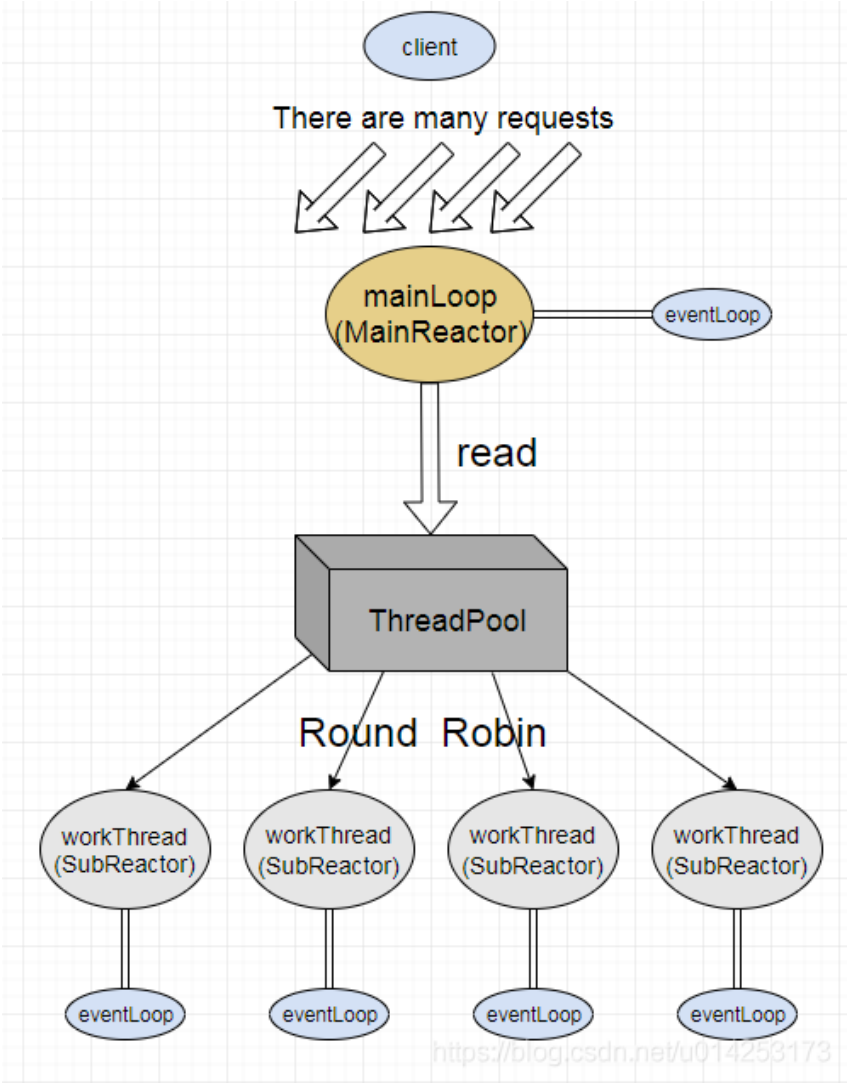
- 内存池工作期间的内存只会增长，不释放给操作系统。直到内存池销毁的时候，才把所有的 `block` 释放掉。这样在经过短时间大量请求后，会导致程序占用大量内存，然而这内存大概率是不会再用的（除非再有短时间

的大量请求）。

> 一些可以优化的地方

- `MemoryPool` 中使用了额外空间（一个链表）来维护 `freed slots`，实际上这部分是可以优化的：将 `struct Slot` 改为 `union Slot`，里面存放 `Slot next` 和 `T value`，这样可以保证每个 `Slot` 在没被分配/被释放时，在内存中就包含了指向下个可用 `Slot` 的指针，对于已经分配了的 `Slot`，就存放用户申请的值。这样可以避免建立链表的开销（这个做法忘记在哪见过了，好像是STL）
- 针对前面所说的缺陷，我觉得可以在服务器空闲（怎么判断呢？）时，将空闲内存释放（释放多少呢？释放哪部分呢？还可以结合页面置换算法？），归还给操作系统

线程池



借鉴 muduo 中 one loop per thread 的思想，每个线程与一个 EventLoop 对应，设计了 `EventLoopThread` 类封装了 thread 和 EventLoop，然后在 `EventLoopThreadPool` 类中根据需要的线程数来创建 `EventLoopThread`，最后根据 Round Robin 来选择下一个 EventLoop，实现负载均衡。

由此我们先对 `EventLoop` 进行封装，让每个 `Thread` 中运行 `EventLoop` 的 Loop。

```
class EventLoopThread {
public:
```

```
    EventLoopThread();
    ~EventLoopThread();
    void start();
    sp_EventLoop getLoop();

private:
    void threadFunc();  //线程运行函数
    sp_EventLoop loop_;
    std::unique_ptr<Thread, decltype(deleteElement<Thread>)*> thread_;
};

EventLoopThread::EventLoopThread()
    : loop_(newElement<EventLoop>(), deleteElement<EventLoop>),
      thread_(newElement<Thread>(std::bind(&EventLoopThread::threadFunc, this),
"EventLoopThread"),
              deleteElement<Thread>)
{
}

EventLoopThread::~EventLoopThread() {}

sp_EventLoop EventLoopThread::getLoop() {
    return loop_;
}

void EventLoopThread::start() {
    thread_->start();
}

void EventLoopThread::threadFunc() {
    loop_->loop();
}
```

线程池的实现方面，采用了比较朴素的 Round Robin，后期也可以在负载均衡算法上进行优化，代码比较简单。

```
// 线程池的本质是生产者消费者模型
class EventLoopThreadPool {
public:
    EventLoopThreadPool(int numThreads);
    ~EventLoopThreadPool() { LOG << "~EventLoopThreadPool()"; }

    void start();
    sp_EventLoop getNextLoop();

private:
    int numThreads_;
    int index_;
    std::vector<std::shared_ptr<EventLoopThread>> threads_;
};
```

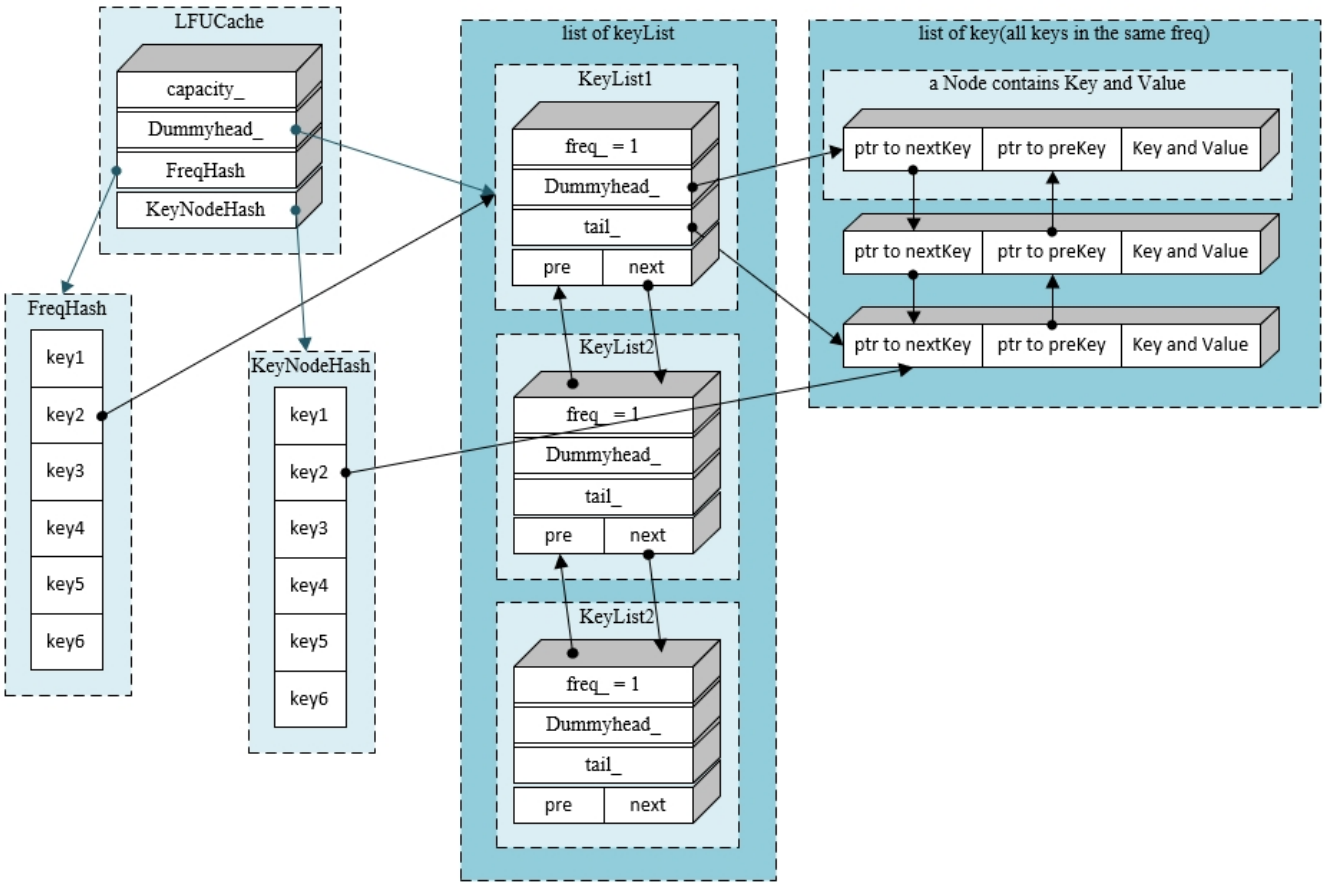
```
EventLoopThreadPool::EventLoopThreadPool(int numThreads)
    : numThreads_(numThreads),
      index_(0) {
    threads_.reserve(numThreads);
    for(int i = 0; i < numThreads_; ++i) {
        std::shared_ptr<EventLoopThread> t(newElement<EventLoopThread>(),
deleteElement<EventLoopThread>);
        threads_.emplace_back(t);
    }
}

void EventLoopThreadPool::start() {
    for(auto& thread : threads_) {
        thread->start();
    }
}

sp_EventLoop EventLoopThreadPool::getNextLoop() {
    // 类似于与环形数组的遍历
    index_ = (index_ + 1) % numThreads_;
    return threads_[index_]->getLoop();
}
```

LFU

> 为什么要使用LFU而不是LRU?

- LRU：最近最少使用，发生淘汰的时候，淘汰访问时间最旧的页面。
- LFU：最近不经常使用，发生淘汰的时候，淘汰频次低的页面。
- 对于时间相关度较低（页面访问完全是随机，与时间无关） `WebServer` 来说，经常被访问的页面在下一次有更大的可能被访问，此时使用LFU更好，并且LFU能够避免周期性或者偶发性的操作导致缓存命中率下降的问题；而对于时间相关度较高（某些页面在特定时间段访问量较大，而在整体来看频率较低）的 `WebServer` 来说，在特定的时间段内，最近访问的页面在下一次有更大的可能被访问，此时使用LRU更好。
- 目前设计的 `WebServer` 时间相关度较低，所以选择LFU。

LFU的原理比较简单，实现方法也比较多样（leetcode上面就有设计LFU和LRU的题），我目前采用的是双重链表+hash表的经典做法。



LFU的主要框架如上图所示，主要对象有四个：一个大链表 `FreqList` ，一个小链表 `KeyList` ，一个 `key->FreqList` 的哈希表 `FreqHash` 以及一个 `key->KeyNode` 的哈希表 `KeyNodeHash` ，下面简单说下作用：

- `FreqList` ：链表的每个节点都是一个小链表附带一个值表示频度，频度用来记录每个key的访问次数
- `KeyList` ：同一频度下的key value节点 `KeyNode` ，只要找到 `key` 对应的 `KeyNode` ，就可以找到对应的 `value`
- `FreqHash` ：key到大链表节点的映射，这个主要是为了更新频度，因为更新频度需要先将 `KeyNode` 节点从当前频度的小链表中删除，然后加入下一频度的小链表（通过遍历大链表即可找到）
- `KeyNodeHash` ：key到小链表节点的映射，这个Hash主要是为了根据key来找到对应的 `KeyNode` ，从而获得对应的value
- LRU中主要就两个操作： `bool LFUCache::get(string& key, string& val)` 和 `void LFUCache::set(string& key, string& val)` 。 `get(2)` 就是先在 `FreqHash` 中找有没有 `key` 对应的节点，如果没有，就是缓存未命中，由用户调用 `set(2)` 向缓存中添加节点，如果缓存已满的话就在频度最低的小链表中删除最后一个节点；如果有，就是缓存命中，此时需要更新当前节点的频度（先将 `KeyNode` 节点从当前频度的小链表中删除，然后加入下一频度的小链表），加入下一频度的小链表的头部，这部分与LRU类似。
- 而对于 `WebServer` 来说，key就是对应的文件名，value就是对应的文件内容（通过 `mmap()` 映射），这样来建立页面缓存还是比较简单的。

```
// LFUCache.h
#define LFU_CAPACITY 10

using std::string;
```

```cpp
// 链表的节点
template<typename T>
class Node {
public:
    void setPre(Node* p) { pre_ = p; }
    void setNext(Node* p) { next_ = p; }
    Node* getPre() { return pre_; }
    Node* getNext() { return next_; }
    T& getValue() { return value_; }

private:
    T value_;
    Node* pre_;
    Node* next_;
};

// 文件名->文件内容的映射
struct Key {
    string key_, value_;
};

typedef Node<Key>* key_node;

// 链表：由多个Node组成
class KeyList {
public:
    void init(int freq);
    void destory();
    int getFreq();
    void add(key_node& node);
    void del(key_node& node);
    bool isEmpty();
    key_node getLast();

private:
    int freq_;
    key_node Dummyhead_;
    key_node tail_;
};

typedef Node<KeyList>* freq_node;

/*
典型的双重链表+hash表实现
首先是一个大链表，链表的每个节点都是一个小链表附带一个值表示频度
小链表存的是同一频度下的key value节点。
hash表存key到大链表节点的映射(key,freq_node)和
key到小链表节点的映射(key,key_node).
*/
```

```
// LFU由多个链表组成
class LFUCache {
private:
    freq_node Dummyhead_;    // 大链表的头节点，里面每个节点都是小链表的头节点
    size_t capacity_;
    MutexLock mutex_;

    std::unordered_map<string, key_node> kmap_;  // key到keynode的映射
    std::unordered_map<string, freq_node> fmap_; // key到freqnode的映射

    void addFreq(key_node& nowk, freq_node& nowf);
    void del(freq_node& node);
    void init();

public:
    LFUCache(int capicity);
    ~LFUCache();

    bool get(string& key, string& value); // 通过key返回value并进行LFU操作
    void set(string& key, string& value); // 更新LFU缓存
    size_t getCapacity() const { return capacity_; }
};

LFUCache& getCache();
```

这里没有用 namespace 限定，大家可以注意优化一下。。

```
void KeyList::init(int freq) {
    freq_ = freq;
    // Dummyhead_ = tail_= new Node<Key>;
    Dummyhead_ = newElement<Node<Key>>();
    tail_ = Dummyhead_;
    Dummyhead_->setNext(nullptr);
}

// 删除整个链表
void KeyList::destory() {
    while(Dummyhead_ != nullptr) {
        key_node pre = Dummyhead_;
        Dummyhead_ = Dummyhead_->getNext();
        // delete pre;
        deleteElement(pre);
    }
}

int KeyList::getFreq() { return freq_; }

// 将节点添加到链表头部
```

```
void KeyList::add(key_node& node) {
    if(Dummyhead_->getNext() == nullptr) {
        // printf("set tail\n");

        tail_ = node;
        // printf("head_ is %p, tail_ is %p\n", Dummyhead_, tail_);
    }
    else {
        Dummyhead_->getNext()->setPre(node);
        // printf("this node is not the first\n");
    }
    node->setNext(Dummyhead_->getNext());
    node->setPre(Dummyhead_);
    Dummyhead_->setNext(node);

    assert(!isEmpty());
}

// 删除小链表中的节点
void KeyList::del(key_node& node) {
    node->getPre()->setNext(node->getNext());

    if(node->getNext() == nullptr) {
        tail_ = node->getPre();
    }
    else {
        node->getNext()->setPre(node->getPre());
    }
}

bool KeyList::isEmpty() {
    return Dummyhead_ == tail_;
}

key_node KeyList::getLast() {
    return tail_;
}

LFUCache::LFUCache(int capacity) : capacity_(capacity) {
    init();
}

LFUCache::~LFUCache() {
    while(Dummyhead_) {
        freq_node pre = Dummyhead_;
        Dummyhead_ = Dummyhead_->getNext();
        pre->getValue().destory();
        // delete pre;
        deleteElement(pre);
```

```
    }
}

void LFUCache::init() {
    // FIXME:缓存的容量动态变化

    // Dummyhead_ = new Node<KeyList>();
    Dummyhead_ = newElement<Node<KeyList>>();
    Dummyhead_->getValue().init(0);
    Dummyhead_->setNext(nullptr);
}


// 更新节点频度:
// 如果不存在下一个频度的链表，则增加一个
// 然后将当前节点放到下一个频度的链表的头位置
void LFUCache::addFreq(key_node& nowk, freq_node& nowf) {
    // printf("enter addFreq\n");
    freq_node nxt;
    // FIXME:频数有可能有溢出
    if(nowf->getNext() == nullptr ||
        nowf->getNext()->getValue().getFreq() != nowf->getValue().getFreq() + 1) {
            // 新建一个下一频度的大链表,加到nowf后面
            // nxt = new Node<KeyList>();
            nxt = newElement<Node<KeyList>>();
            nxt->getValue().init(nowf->getValue().getFreq() + 1);
            if(nowf->getNext() != nullptr)
                nowf->getNext()->setPre(nxt);
            nxt->setNext(nowf->getNext());
            nowf->setNext(nxt);
            nxt->setPre(nowf);
            // printf("the prefreq is %d, and the cur freq is %d\n", nowf-
>getValue().getFreq(), nxt->getValue().getFreq());
    }
    else {
        nxt = nowf->getNext();
    }
    fmap_[nowk->getValue().key_] = nxt;
    // 将其从上一频度的小链表删除
    // 然后加到下一频度的小链表中
    if(nowf != Dummyhead_) {
        nowf->getValue().del(nowk);
    }
    nxt->getValue().add(nowk);

    //printf("the freq now is %d\n", nxt->getValue().getFreq());
    assert(!nxt->getValue().isEmpty());

    // 如果该频度的小链表已经空了
```

```
    if(nowf != Dummyhead_ && nowf->getValue().isEmpty())
        del(nowf);
}

bool LFUCache::get(string& key, string& val) {
    if(!capacity_)  return false;
    MutexLockGuard lock(mutex_);
    if(fmap_.find(key) != fmap_.end()) {
        // 缓存命中
        key_node nowk = kmap_[key];
        freq_node nowf = fmap_[key];
        val += nowk->getValue().value_;
        addFreq(nowk, nowf);
        return true;
    }
    // 未命中
    return false;
}

void LFUCache::set(string& key, string& val) {
    if(!capacity_)  return;
    // printf("kmapsize = %d capacity = %d\n", kmap_.size(), capacity_);
    MutexLockGuard lock(mutex_);

    // 缓存满了
    // 从频度最小的小链表中的节点中删除最后一个节点（小链表中的删除符合LRU）
    if(kmap_.size() == capacity_) {

        // printf("need to delete a Node\n");
        freq_node head = Dummyhead_->getNext();
        key_node last = head->getValue().getLast();
        head->getValue().del(last);
        kmap_.erase(last->getValue().key_);

        fmap_.erase(last->getValue().key_);

        // delete last;
        deleteElement(last);
        // 如果频度最小的链表已经没有节点，就删除
        if(head->getValue().isEmpty()) {
            del(head);
        }
    }
    // key_node nowk = new Node<Key>();
    // 使用内存池
    key_node nowk = newElement<Node<Key>>();

    nowk->getValue().key_ = key;
    nowk->getValue().value_ = val;
```

```
    addFreq(nowk, Dummyhead_);
    kmap_[key] = nowk;
    fmap_[key] = Dummyhead_->getNext();
}

void LFUCache::del(freq_node& node) {
    node->getPre()->setNext(node->getNext());
    if(node->getNext() != nullptr) {
        node->getNext()->setPre(node->getPre());
    }

    node->getValue().destory();
    // delete node;
    deleteElement(node);

}

LFUCache& getCache() {
    static LFUCache cache(LFU_CAPACITY);
    return cache;
}
```

具体的实现代码对着图也比较好理解，我这里就不再赘述了，值得注意的是，这里使用的是单例模式（想想为什么）。这里我提供一个简单的测试 demo。

```
#include "../LFUCache.h"
#include <iostream>
#include <vector>
#include <random>

using namespace std;


void test1() {
    init_MemoryPool();
    getCache();
    LFUCache& Cache = getCache();
    cout << "the capacity of LFU is : " << Cache.getCapacity() << endl;

    default_random_engine e;
    e.seed(time(NULL));
    uniform_int_distribution<unsigned> u(0, 25);

    vector<pair<string, string>> seed(26);
    for(int i = 0; i < 26; ++i) {
        seed[i] = make_pair('a' + i, 'A' + i);
    }
```

```
    for(int i = 0; i < 100; ++i) {
        string val;
        auto temp = seed[u(e)];
        cout << "----------round " << i << "----------" << endl;
        cout << "i want key: " << temp.first << endl;
        if(!Cache.get(temp.first, val)) {
            cout << "cache miss ";
            Cache.set(temp.first, temp.second);
            cout << "insert val..." << endl;
        }
        else {
            cout << "cache hit and value is :" << val << endl;
        }
    }

}

int main() {
    test1();
    return 0;
}
```

> 线程安全

LFU本身是线程安全的：这部分的线程安全暂时没有好的想法，简单的在 `get(2)` 和 `set(2)` 加了互斥锁 `MutexLockGuard lock(mutex_);`

> 缺陷

- 由于LFU的淘汰策略是淘汰访问次数最小的数据块，但是新插入的数据块的访问次数为1，这样就会产生缓存污染，使得新数据块被淘汰。换句话说就是，最近加入的数据因为起始的频率很低，容易被淘汰，而早期的热点数据会一直占据缓存。
- 对热点数据的访问会导致 `freq` 一直递增，我目前使用 `int` 表示 `freq_` ，实际上会有溢出的风险。

> 一些可以优化的地方

针对上文的缺陷，目前有三个方向改进思路：

- LFU-Aging：在LFU算法之上，引入访问次数平均值概念，当平均值大于最大平均值限制时，将所有节点的访问次数减去最大平均值限制的一半或一个固定值。相当于热点数据“老化”了，这样可以避免频度溢出，也能缓解缓存污染
- window-LFU：Window是用来描述算法保存的历史请求数量的窗口大小的。Window-LFU并不记录所有数据的访问历史，而只是记录过去一段时间内的访问历史。即当请求次数达到或超过window的大小后，每次有一条新的请求到来，都会清理掉最旧的一条请求，以保证算法记录的请求次数不超过window的大小。
- redis-LFU：redis的实现结合了LRU和LFU，通过定时器，每n秒淘汰一次过期缓存（LRU），过期缓存不会马上被删除，而是加入一个过期队列，然后真正需要淘汰时，从过期队列中删除使用频率最少的键（LFU）。redis不太了解，讲的是自己的理解。

写好了就完了吗?

经过一番 debug 和功能测试后，我们总算把整个项目做起来了，经过我们在编码时候的单元测试，也基本能够确保项目的基本功能没有问题。现在我们还需要对项目进行压力测试，来测试一下我们项目的性能能到什么程度（也就是面试官经常问的，你项目的性能瓶颈在什么地方，有什么优化的思路）。下面我们来讲一下如何利用压测工具 webbench 来对我们的 WebServer 进行压测。

部署（没有云服务器的同学可以跳过）

在我们编码的过程中，我们都是在本地环境中进行测试的（也是就访问的是回环ip：127.0.0.1），然而在实际环境中我们的服务器不可能放在本地使用，所以我还是建议大家能够租一个云服务器（阿里云、华为云、腾讯云等），在云服务器上面部署自己的项目，这样的好处有几个：

1. 可以在远端直接访问项目，不用每次都开虚拟机或者 WSL 了，只需要在 vscode 上连到服务器就可以进行编码，十分方便。
2. 压力测试工具本身会占用一定的资源，导致本地进行压测出来的结果不一定会准确（**这个问题面试的时候遇到过**）。
3. 让面试官知道你的项目不是只能在本地运行的玩具（虽然就是- -），而是一个已经部署上线的项目（这是加分项）。
4. 云服务器上不只有运行自己项目的作用，还可以充当云盘、生成下载链接分享的角色（重点是不限速嗷），有兴趣的可以自己了解一下。

反正就是租个云服务器是怎么都不亏的。

由于云服务器是自带 ip 地址的，因此我们可以通过其他电脑的浏览器来访问我们部署在云服务器的 WebServer（当然你也可以申请一个域名）。

性能测试

现在我们已经将 WebServer 运行在本地或者云服务器上了，可以通过 webbench 对项目进行压测，下面我先讲一下 webbench 的原理（面试的时候有面试官问过）。

webbench 的原理讲起来还是比较简单的：首先在主进程中 fork 出多个子进程，每个子进程都循环做 web 访问测试。子进程把访问的结果通过pipe告诉父进程，父进程做最终的统计结果。webbench 最多可以模拟3万个并发连接去测试网站的负载能力。

由于之前使用的 webbench 存在 `connect()` 失败时 `sockfd` 泄漏的bug，以及读取响应报文时读完了依然 read 导致阻塞的 bug （因为是BIO，读完了再读就会阻塞了）。我使用 C++11 标准重新实现了一遍 webbench ，下面是 webbench 重构后的代码，大家可以参考一下：

```
// webbench.h
#ifndef WEB_BENCH_H_
#define WEB_BENCH_H_

void ParseArg(int argc, char* argv[]);
void BuildRequest(const char* url);
void WebBench();

#endif  // WEB_BENCH_H_
```

可以看到，webbench 中定义了三个接口：

- `ParseArg(2)`：解析命令行参数
- `BuildRequest(1)`： 构造 URL 请求
- `WebBench()`：执行实际的压测

```
// webbench.cpp
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include <signal.h>
#include <time.h>
#include <unistd.h>
#include <fcntl.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <getopt.h>
#include <rpc/types.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>

#include <iostream>
using std::cout;
using std::endl;

//http请求方法
#define METHOD_GET 0
#define METHOD_HEAD 1
#define METHOD_OPTIONS 2
#define METHOD_TRACE 3
#define WEBBENCH_VERSION "v1.5"
//请求大小
#define REQUEST_SIZE 2048

static int force = 0;              //默认需要等待服务器响应
```

```
static int force_reload = 0;       //默认不重新发送请求
static int clients_num = 1;        //默认客户端数量
static int request_time = 30;      //默认模拟请求时间
static int http_version = 2;       //默认http协议版本 0:http/0.9, 1:http/1.0, 2:http/1.1
static char* proxy_host = NULL;    //默认无代理服务器
static int port = 80;              //默认访问80端口
static int is_keep_alive = 0;      //默认不支持keep alive

static int request_method = METHOD_GET;  //默认请求方法
static int pipeline[2];                  //用于父子进程通信的管道
static char host[MAXHOSTNAMELEN];        //存放目标服务器的网络地址
static char request_buf[REQUEST_SIZE];   //存放请求报文的字节流

static int success_count = 0;     //请求成功的次数
static int failed_count = 0;      //失败的次数
static int total_bytes = 0;      //服务器响应总字节数

volatile bool is_expired = false;     //子进程访问服务器 是否超时

static void Usage() {
  fprintf(stderr,
    "Usage: webbench [option]... URL\n"
    "  -f|--force               不等待服务器响应\n"
    "  -r|--reload              重新请求加载(无缓存)\n"
    "  -9|--http09              使用http0.9协议来构造请求\n"
    "  -1|--http10              使用http1.0协议来构造请求\n"
    "  -2|--http11              使用http1.1协议来构造请求\n"
    "  -k|--keep_alive          客户端是否支持keep alive\n"
    "  -t|--time <sec>          运行时间，单位：秒，默认为30秒\n"
    "  -c|--clients_num <n>     创建多少个客户端，默认为1个\n"
    "  -p|--proxy <server:port> 使用代理服务器发送请求\n"
    "  --get                    使用GET请求方法\n"
    "  --head                   使用HEAD请求方法\n"
    "  --options                使用OPTIONS请求方法\n"
    "  --trace                  使用TRACE请求方法\n"
    "  -?|-h|--help             显示帮助信息\n"
    "  -V|--version             显示版本信息\n"
  );
}

//构造长选项与短选项的对应
static const struct option OPTIONS[] = {
  {"force",      no_argument,      &force,              1},
  {"reload",     no_argument,      &force_reload,       1},
  {"http09",     no_argument,      NULL,               '9'},
  {"http10",     no_argument,      NULL,               '1'},
  {"http11",     no_argument,      NULL,               '2'},
  {"keep_alive", no_argument,      &is_keep_alive,      1},
  {"time",       required_argument,NULL,               't'},
```

```
  {"clients",   required_argument,NULL,               'c'},
  {"proxy",     required_argument,NULL,               'p'},
  {"get",       no_argument,      &request_method,     METHOD_GET},
  {"head",      no_argument,      &request_method,    METHOD_HEAD},
  {"options",   no_argument,      &request_method, METHOD_OPTIONS},
  {"trace",     no_argument,      &request_method,   METHOD_TRACE},
  {"help",      no_argument,      NULL,               '?'},
  {"version",   no_argument,      NULL,               'V'},
  {NULL,        0,                NULL,                0}
};

//打印消息
static void PrintMessage(const char* url) {
  printf("==================================开始压测
==================================\n");
  switch (request_method) {
    case METHOD_GET:
      printf("GET");
      break;
    case METHOD_OPTIONS:
      printf("OPTIONS");
      break;
    case METHOD_HEAD:
      printf("HEAD");
      break;
    case METHOD_TRACE:
      printf("TRACE");
      break;
    default:
      printf("GET");
      break;
  }
  printf(" %s", url);
  switch (http_version) {
    case 0:
      printf(" (Using HTTP/0.9)");
      break;
    case 1:
      printf(" (Using HTTP/1.0)");
      break;
    case 2:
      printf(" (Using HTTP/1.1)");
      break;
  }
  printf("\n客户端数量 %d, 每个进程运行 %d秒", clients_num, request_time);

  if (force) {
    printf(", 选择提前关闭连接");
  }
```

```
  if (proxy_host != NULL) {
    printf(", 经由代理服务器 %s:%d", proxy_host, port);
  }
  if (force_reload) {
    printf(", 选择无缓存");
  }
  printf("\n");
}
```

首先定义了一些记录状态的变量，定义了打印使用说明和结果的函数。然后是两个核心函数 `ConnectServer(2)`、`Worker(3)`，其中 `ConnectServer(2)` 的作用是与 URL 所在的服务端建立 TCP 连接，`Worker(3)` 则是在子进程中创建客户端与服务器发送信息。

```
static int ConnectServer(const char* server_host, int server_port) {
  // 协议的类型:IPv4协议 网络数据类型:字节流 网络协议:TCP协议
  int client_sockfd = socket(AF_INET, SOCK_STREAM, 0);
  if (client_sockfd < 0) {
    return client_sockfd;
  }
  struct sockaddr_in server_addr;
  struct hostent* host_name;
  memset(&server_addr, 0, sizeof(server_addr));
  server_addr.sin_family = AF_INET;

  //设置服务器ip
  unsigned long ip = inet_addr(server_host);
  if (ip != INADDR_NONE) {
    memcpy(&server_addr.sin_addr, &ip, sizeof(ip));
  } else {
    host_name = gethostbyname(server_host);
    if (host_name == NULL) {
      return -1;
    }
    memcpy(&server_addr.sin_addr, host_name->h_addr, host_name->h_length);
  }
  //设置服务器端口
  server_addr.sin_port = htons(server_port);

  //tcp三次握手建立连接
  if (connect(client_sockfd, (struct sockaddr*)&server_addr, sizeof(server_addr)) < 0)
{
        //修复套接字泄漏
        close(client_sockfd);
    return -1;
  }

  return client_sockfd;
}
```

```
//闹钟信号处理函数
static void AlarmHandler(int signal) {
  is_expired = true;
}

//子进程创建客户端去请求服务器
static void Worker(const char* server_host, const int server_port, const char* request)
{
  int client_sockfd = -1;
  int request_size = strlen(request);
  const int response_size = 1500;
  char response_buf[response_size];

  //设置闹钟信号处理函数
  struct sigaction signal_action;
  signal_action.sa_handler = AlarmHandler;
  signal_action.sa_flags = 0;
  if (sigaction(SIGALRM, &signal_action, NULL)) {
    exit(1);
  }
  alarm(request_time);

    if (is_keep_alive) {
        //一直到成功建立连接才退出while
        while (client_sockfd == -1) {
            client_sockfd = ConnectServer(host, port);
        }
        //cout << "1. 建立连接（TCP三次握手）成功!" << endl;
keep_alive:
        while(true) {
            if (is_expired) {
                //到收到闹钟信号会使is_expired为true 此时子进程工作应该结束了(每个子进程都有一个
is_expired)
                if (failed_count > 0) {
                    failed_count--;
                }
                return;
            }

            if (client_sockfd < 0) {
                failed_count++;
                continue;
            }

            //向服务器发送请求报文
            if (request_size != write(client_sockfd, request, request_size)) {
                cout << "2. 发送请求报文失败: " << strerror(errno) << endl;
                failed_count++;
```

```
                close(client_sockfd);
                //发送请求失败 要重新创建套接字
                client_sockfd = -1;
                while (client_sockfd == -1) {
                    client_sockfd = ConnectServer(host, port);
                }
                continue;
            }
            //cout << "2. 发送请求报文成功!" << endl;

            //没有设置force时 默认等到服务器的回复
            if (force == 0)  {
                //keep-alive一个进程只创建一个套接字 收发数据都用这个套接字
                //读服务器返回的响应报文到response_buf中
                while (true) {
                    if (is_expired) {
                        break;
                    }
                    int read_bytes = read(client_sockfd, response_buf, response_size);
                    if (read_bytes < 0) {
                        //cout << "3. 接收响应报文失败: " << strerror(errno) << endl;
                        failed_count++;
                        close(client_sockfd);
                        //读取响应失败 不用重新创套接字 重新发一次请求即可
                        goto keep_alive;
                    } else {
                        //cout << "3. 接收响应报文成功! " << endl;
                        total_bytes += read_bytes;
                        break;
                    }
                }
            }
            success_count++;
            //cout << "当前成功: " << success_count << ", 失败: " << failed_count << ", 总字节: " << total_bytes << endl;
        }
    } else {
not_keep_alive:
        while(true) {
            if (is_expired) {
                //到收到闹钟信号会使is_expired为true 此时子进程工作应该结束了(每个子进程都有一个is_expired)
                if (failed_count > 0) {
                    failed_count--;
                }
                return;
            }

            //与服务器建立连接
```

```
            client_sockfd = ConnectServer(host, port);
            if (client_sockfd < 0) {
                //cout << "1. 建立连接失败: " << strerror(errno) << endl;
                failed_count++;
                continue;
            }
            //cout << "1. 建立连接（TCP三次握手）成功!" << endl;

            //向服务器发送请求报文
            if (request_size != write(client_sockfd, request, request_size)) {
                cout << "2. 发送请求报文失败: " << strerror(errno) << endl;
                failed_count++;
                close(client_sockfd);
                continue;
            }
            //cout << "2. 发送请求报文成功!" << endl;

            //http0.9特殊处理
            if (http_version == 0) {
                if (shutdown(client_sockfd, 1)) {
                    failed_count++;
                    close(client_sockfd);
                    continue;
                }
            }

            //没有设置force时 默认等到服务器的回复
            if (force == 0)  {
                //not-keey-alive 每次发送完请求，读取了响应后就关闭套接字
                //下次创建新套接字 发送请求 读取响应
                while (true) {
                    if (is_expired) {
                        break;
                    }
                    int read_bytes = read(client_sockfd, response_buf, response_size);
                    if (read_bytes < 0) {
                        //cout << "3. 接收响应报文失败: " << strerror(errno) << endl;
                        failed_count++;
                        close(client_sockfd);
                        //这里读失败想退出来继续创建套接字 去请求服务器,因为有两层循环 所以用goto
                        goto not_keep_alive;
                    } else {
                        //cout << "3. 接收响应报文成功! " << endl;
                        total_bytes += read_bytes;
                        break;
                    }
                }
            }
```

```
            //一次发送 接收完成后 关闭套接字
            if (close(client_sockfd)) {
                cout << "4. 关闭套接字失败: " << strerror(errno) << endl;
                failed_count++;
                continue;
            }
            success_count++;
            //cout << "当前成功: " << success_count << ", 失败: " << failed_count << ", 总
字节: " << total_bytes << endl;
        }
    }
}
```

`ConnectServer(2)` 的实现就比较简单了，就是和 CSAPP 一样的几个系统接口的调用。`Worker(3)` 则针对长连接和短连接进行不同的操作，短连接需要在每次收发完数据后即与服务器断开连接，然后再执行 `ConnectServer(2)` ，而长连接则不用，最后在收到闹钟信号 `is_expired` 后，停止连接，并将当前子进程的结果发给父进程统计。

webbench 的三个接口的实现如下：

```
// webbench.cpp
//命令行解析
void ParseArg(int argc, char* argv[]) {
  int opt = 0;
  int options_index = 0;

  //没有输入选项
  if (argc == 1) {
    Usage();
    exit(1);
  }

  //一个一个解析输入选项
  while ((opt = getopt_long(argc, argv, "fr912kt:c:p:Vh?",
                OPTIONS, &options_index)) != EOF) {
    switch (opt) {
      case 'f':
        force = 1;
        break;
      case 'r':
        force_reload = 1;
        break;
      case '9':
        http_version = 0;
        break;
      case '1':
        http_version = 1;
        break;
      case '2':
```

```
      http_version = 2;
      break;
    case 'k':
      is_keep_alive = 1;
      break;
    case 't':
      request_time = atoi(optarg);
      break;
    case 'c':
      clients_num = atoi(optarg);
      break;
    case 'p': {
      //使用代理服务器 设置其代理网络号和端口号
      char* proxy = strrchr(optarg, ':');
      proxy_host = optarg;
      if (proxy == NULL) {
        break;
      }
      if (proxy == optarg) {
        fprintf(stderr,"选项参数错误, 代理服务器%s: 缺少主机名\n", optarg);
        exit(1);
      }
      if (proxy == optarg + strlen(optarg) - 1) {
        fprintf(stderr,"选项参数错误, 代理服务器%s: 缺少端口号\n", optarg);
        exit(1);
      }
      *proxy = '\0';
      port = atoi(proxy + 1);
      break;
    }
    case '?':
    case 'h':
      Usage();
      exit(0);
    case 'V':
      printf("WebBench %s\n", WEBBENCH_VERSION);
      exit(0);
    default:
      Usage();
      exit(1);
  }
}

//没有输入URL
if (optind == argc) {
  fprintf(stderr,"WebBench: 缺少URL参数!\n");
  Usage();
  exit(1);
}
```

```
  //如果没有设置客户端数量和连接时间，则设置默认
  if (clients_num == 0) {
    clients_num = 1;
  }
  if (request_time == 0) {
    request_time = 30;
  }
}

//构造请求
void BuildRequest(const char* url) {
  bzero(host, MAXHOSTNAMELEN);
  bzero(request_buf, REQUEST_SIZE);

  //无缓存和代理都要在http1.0以上才能使用
  if (force_reload && proxy_host != NULL && http_version < 1) {
    http_version = 1;
  }
  if (request_method == METHOD_HEAD && http_version < 1) {
    http_version = 1;
  }
  //OPTIONS和TRACE要1.1
  if (request_method == METHOD_OPTIONS && http_version < 2) {
    http_version = 2;
  }
  if (request_method == METHOD_TRACE && http_version < 2) {
    http_version = 2;
  }

  //http请求行 的请求方法
  switch (request_method) {
    case METHOD_GET:
      strcpy(request_buf, "GET");
      break;
    case METHOD_HEAD:
      strcpy(request_buf, "HEAD");
      break;
    case METHOD_OPTIONS:
      strcpy(request_buf, "OPTIONS");
      break;
    case METHOD_TRACE:
      strcpy(request_buf, "TRACE");
      break;
    default:
      strcpy(request_buf, "GET");
      break;
  }
  strcat(request_buf, " ");
```

```
  //判断url是否有效
  if (NULL == strstr(url, "://")) {
    fprintf(stderr, "\n%s: 无效的URL\n", url);
    exit(2);
  }
  if (strlen(url) > 1500) {
    fprintf(stderr,"URL长度过长\n");
    exit(2);
  }
  if (proxy_host == NULL) {
    //忽略大小写比较前7位
    if (0 != strncasecmp("http://", url, 7)) {
      fprintf(stderr,"\n无法解析URL(只支持HTTP协议, 暂不支持HTTPS协议)\n");
      exit(2);
    }
  }
  //定位url中主机名开始的位置
  int i = strstr(url, "://") - url + 3;
  //在主机名开始的位置是否有/ 没有则无效
  if (strchr(url + i, '/') == NULL) {
    fprintf(stderr,"\n无效的URL,主机名没有以'/'结尾\n");
    exit(2);
  }

  //填写请求url 无代理时
  if (proxy_host == NULL) {
    //有端口号 填写端口号
    if (index(url + i, ':') != NULL
      && index(url + i, ':') < index(url + i, '/')) {
      //设置域名或IP
      strncpy(host, url + i, strchr(url + i, ':') - url - i);
      char port_str[10];
      bzero(port_str, 10);
      strncpy(port_str, index(url + i, ':') + 1, strchr(url + i, '/') - index(url + i,
':') -1);
      //设置端口
      port = atoi(port_str);
      //避免写了: 却没写端口号
      if (port == 0) {
        port = 80;
      }
    } else {
      strncpy(host, url + i, strcspn(url + i, "/"));
    }
    strcat(request_buf + strlen(request_buf), url + i + strcspn(url + i, "/"));
  } else {
    //有代理服务器 直接填就行
    strcat(request_buf, url);
```

```
  }

  //填写http协议版本
  if (http_version == 0) {
    strcat(request_buf, " HTTP/0.9");
  } else if (http_version == 1) {
    strcat(request_buf, " HTTP/1.0");
  } else if (http_version == 2) {
    strcat(request_buf, " HTTP/1.1");
  }
  strcat(request_buf, "\r\n");

  //请求头部
  if (http_version > 0) {
    strcat(request_buf, "User-Agent: WebBench " WEBBENCH_VERSION "\r\n");
  }
  //域名或IP字段
  if (proxy_host == NULL && http_version > 0) {
    strcat(request_buf, "Host: ");
    strcat(request_buf, host);
    strcat(request_buf, "\r\n");
  }
  //强制重新加载, 无缓存字段
  if (force_reload && proxy_host != NULL) {
    strcat(request_buf, "Pragma: no-cache\r\n");
  }
  //keep alive
  if (http_version > 1) {
    strcat(request_buf, "Connection: ");
    if (is_keep_alive) {
      strcat(request_buf, "keep-alive\r\n");
    } else {
      strcat(request_buf, "close\r\n");
    }
  }
  //添加空行
  if (http_version > 0) {
    strcat(request_buf, "\r\n");
  }

  printf("Requset: %s\n", request_buf);
  PrintMessage(url);
}

//父进程的作用: 创建子进程 读子进程测试到的数据 然后处理
void WebBench() {
  pid_t pid = 0;
  FILE* pipe_fd = NULL;
  int req_succ, req_fail, resp_bytes;
```

```
  //尝试建立连接一次
  int sockfd = ConnectServer(proxy_host == NULL ? host : proxy_host, port);
  if (sockfd < 0) {
    fprintf(stderr,"\n连接服务器失败, 中断压力测试\n");
    exit(1);
  }
  close(sockfd);
  //创建父子进程通信的管道
  if (pipe(pipeline)) {
    perror("通信管道建立失败");
    exit(1);
  }

  int proc_count;
  //让子进程取测试 建立子进程数量由clients_num决定
  for (int proc_count = 0; proc_count < clients_num; proc_count++) {
    //创建子进程
    pid = fork();
    //<0是失败, =0是子进程, 这两种情况都结束循环 =0时子进程可能继续fork
    if (pid <= 0) {
      sleep(1);
      break;
    }
  }

  //fork失败
  if (pid < 0) {
    fprintf(stderr, "第%d个子进程创建失败\n", proc_count);
    perror("创建子进程失败");
    exit(1);
  }

  //子进程处理
  if (pid == 0) {
    //子进程 创建客户端连接服务端 并发出请求报文
    Worker(proxy_host == NULL ? host : proxy_host, port, request_buf);
    pipe_fd = fdopen(pipeline[1], "w");
    if (pipe_fd == NULL) {
      perror("管道写端打开失败");
      exit(1);
    }
    //进程运行完后, 向管道写端写入该子进程在这段时间内请求成功的次数 失败的次数 读取到的服务器响应总字
节数
    fprintf(pipe_fd, "%d %d %d\n", success_count, failed_count, total_bytes);
    //关闭管道写端
    fclose(pipe_fd);

    exit(1);
```

```
  } else {
    // 父进程处理
    pipe_fd = fdopen(pipeline[0], "r");
    if (pipe_fd == NULL) {
      perror("管道读端打开失败");
      exit(1);
    }
    //因为我们要求数据要及时 所以没有缓冲区是最合适的
    setvbuf(pipe_fd, NULL, _IONBF, 0);
    success_count = 0;
    failed_count = 0;
    total_bytes = 0;

    while (true) {
      //从管道读端读取数据
      pid = fscanf(pipe_fd, "%d %d %d", &req_succ, &req_fail, &resp_bytes);
      if (pid < 2) {
        fprintf(stderr, "某个子进程异常\n");
        break;
      }
      success_count += req_succ;
      failed_count += req_fail;
      total_bytes += resp_bytes;
      //我们创建了clients_num个子进程 所以要读clients_num多次数据
      if (--clients_num == 0) {
        break;
      }
    }
    fclose(pipe_fd);

    printf("\n速度: %d 请求/秒, %d 字节/秒.\n请求: %d 成功, %d 失败\n",
        (int)((success_count + failed_count) / (float)request_time),
        (int)(total_bytes / (float)request_time),
        success_count, failed_count);
  }
}
```

再写一下 Makefile：

```
# Makefile
# $^  代表所有依赖文件
# $@  代表所有目标文件
# $<  代表第一个依赖文件
# %   代表通配符

TARGET      := web_bench
STATIC_LIB  := libweb_bench.a
SHARED_LIB  := libweb_bench.so
```

```
CXX       := g++
CXXFLAGS  := -std=gnu++11 -Wfatal-errors -Wno-unused-parameter
LDFLAGS   := -Wl,-rpath=./lib
INC_DIR   := -I ./include
LIB_DIR   := -L ./
LIBS      := -lpthread

DEBUG     := 0

ifeq ($(DEBUG), 1)
  CXXFLAGS += -g -DDEBUG
else
  CXXFLAGS += -O3 -DNDEBUG
endif

all: $(STATIC_LIB) $(TARGET)

#源文件
SOURCES := web_bench.cpp
#所有.c源文件结尾变为.o 放入变量OBJS
OBJS := $(patsubst %.cpp, %.o, $(SOURCES))
#main源文件
MAIN := main.cpp
MAIN_OBJ := main.o

#编译
%.o: %.cpp
  @echo -e '\e[1;32mBuilding CXX object: $@\e[0m'
  $(CXX) -c $^ -o $@ $(CXXFLAGS) $(INC_DIR)

#链接生成静态库(这只是将源码归档到一个库文件中，什么flag都不加) @执行shell
$(STATIC_LIB): $(OBJS)
  @echo -e '\e[1;36mLinking CXX static library: $@\e[0m'
  ar -rcs -o $@ $^
  @echo -e '\e[1;35mBuilt target: $@\e[0m'

#链接生成动态库
$(SHARED_LIB): $(OBJS)
  @echo -e '\e[1;36mLinking CXX shared library: $@\e[0m'
  $(CXX) -fPIC -shared -o $@ $^
  @echo -e '\e[1;35mBuilt target: $@\e[0m'

#链接生成可执行文件
$(TARGET): $(MAIN_OBJ) $(STATIC_LIB)
  @echo -e '\e[1;36mLinking CXX executable: $@\e[0m'
  $(CXX) -o $@ $^ $(LDFLAGS)
  @echo -e '\e[1;35mBuilt target: $@\e[0m'

#声明这是伪目标文件
```

```
.PHONY: all clean

install: $(STATIC_LIB) $(TARGET)
  @if (test ! -d lib); then mkdir -p lib; fi
  @mv $(STATIC_LIB) lib
  @if (test ! -d bin); then mkdir -p bin; fi
  @mv $(TARGET) bin

clean:
  rm -f $(OBJS) $(MAIN_OBJ) $(TARGET) $(STATIC_LIB) $(SHARED_LIB) ./bin/$(TARGET)
./lib/$(STATIC_LIB) ./lib/$(SHARED_LIB)
```

以及编译的脚本文件：

```
# build.sh
#!/bin/bash

#编译
make clean && make -j8 && make install
rm -f *.o
```

再来个运行的脚本：

```
#!/bin/bash

webbench=./bin/web_bench
#服务器地址
server_ip=127.0.0.1
server_port=8888
url=http://${server_ip}:${server_port}/echo

#子进程数量
process_num=1000
#请求时间(单位s)
request_time=60
#keep-alive
is_keep_alive=1
#force
is_force=0

#命令行选项
options="-c $process_num -t $request_time $url"

if [ $is_force -eq 1 ]
then
    options="-f $options"
fi
```

```
if [ $is_keep_alive -eq 1 ]
then
    options="-k $options"
fi

#删除之前开的进程
#kill -9 `ps -ef | grep web_bench | awk '{print $2}'`

#运行
$webbench $options
```

最后编译再运行就可以看到我们 WebServer 的压测结果啦~

```
# 使用脚本构建
./build.sh
# 使用脚本运行
sh ./run_bench.sh
```

可以看到压测这个过程是比较简单的，但重点是要知道压测工具的原理，这才是面试官更看重的东西。

面试问题

项目介绍

简单介绍一下你的项目

这个项目是我在学习计算机网络和Linux socket编程过程中独立开发的轻量级Web服务器，服务器的网络模型是主从reactor加线程池的模式，IO处理使用了非阻塞IO和IO多路复用技术，具备处理多个客户端的http请求和ftp请求，以及对外提供轻量级储存的能力。这个项目从4月份开始做，到7月份完成了项目的整体功能。

项目中的工作可以分为两部分，一部分是服务器网络框架、日志系统、存储引擎等一些基本系统的搭建，另一部分是为了提高服务器性能所做的一些优化，比如缓存机制、内存池等一些额外系统的搭建。最后还对系统中的部分功能进行了功能和压力测试。对于存储引擎的压力测试，在本地测试下，存储引擎读操作的QPS可以达到36万，写操作的QPS可以达到30万。对于网络框架的测试，使用webbench创建1000个进程对服务器进行60s并发请求，测试结果表明，对于短连接的QPS为1.8万，对于长连接的QPS为5.2万。

在开发阶段对于一些功能的不同实现方法，我先思考一下各个方法的应用场景和效率，再进行选择，通过这个项目，我学习了两种Linux下的高性能网络模式，熟悉了Linux环境下的编程，并在后期优化的过程中了解了一些服务端性能调优的方法。

项目中的难点?

项目中我主要的工作可以分为两部分：

1. 一部分是服务器网络框架、日志系统、存储引擎等一些基本系统的搭建，这部分的难点主要就是技术的理解和选型，以及将一些开源的框架调整后应用到我的项目中去。
2. 另一部分是为了提高服务器性能所做的一些优化，比如缓存机制、内存池等一些额外系统的搭建。这部分的难

点主要是找出服务器的性能瓶颈，然后结合自己的想法去突破这个瓶颈，提高服务器的性能。

项目中遇到的困难？是如何解决的？

1. 一方面是对不同的技术理解不够深刻，难以选出最合适的技术框架。这部分的话我主要是反复阅读作者在GitHub提供的一些技术文档，同时也去搜索一些技术对比的文章去看，如果没有任何相关的资料我会尝试去联系作者。
2. 另一方面是编程期间遇到的困难，在代码编写的过程中由于工程能力不足，程序总会出现一些bug。这部分的话我首先是通过日志去定位bug，然后推断bug出现的原因并尝试修复，如果是自己目前水平无法修复的bug，我会先到网上去查找有没有同类型问题的解决方法，然后向同学或者直接到StackOverflow等一些国外知名论坛上求助。

针对项目做了哪些优化？

1. 程序本身
    - 减少程序等待 IO 的事件：非阻塞 IO + IO 多路复用
    - 设计高性能网络框架，同步 IO（主从 reactor + 线程池）和异步 IO（proactor）
    - 【减少系统调用】避免频繁申请/释放内存：线程池、内存池和缓存机制
    - 【减少系统调用】对于文件发送，使用零拷贝函数 `sendFile()` 来发送，避免拷贝数据到用户态
    - 【减少系统调用】尽量减少锁的使用，如果需要，尽量减小临界区（日志系统和线程池）
2. 系统参数调优
    - 最大文件描述符数（用户级和系统级）
    - tcp 连接的参数（半连接/连接队列的长度、tcp syncookies）

项目中用到哪些设计模式？

单例模式：在线程池、内存池中都有使用到。

单例模式可以分为**懒汉式**和**饿汉式**，两者之间的区别在于**创建实例的时间不同**：

- **懒汉式**：指系统运行中，实例并不存在，只有当需要使用该实例时，才会去创建并使用实例。**（这种方式要考虑线程安全）**
- **饿汉式**：指系统一运行，就初始化创建实例，当需要时，直接调用即可。**（本身就线程安全，没有多线程的问题）**

```
// 懒汉式
class Singleton
{
public:
    //返回引用，避免用户对指针使用delete
    static Singleton& getInstance(){
        static Singleton instance;
        return instance;
```

```
    }

public:
    Singleton(const Singleton&) = delete;
    Singleton(Singleton&&) = delete;
    Singleton& operator=(const Singleton&) = delete;
    Singleton& operator=(Singleton&&) = delete;

private:
    Singleton(){
        std::cout<<"constructor called!"<<std::endl;
    }
    ~Singleton(){
        std::cout<<"destructor called!"<<std::endl;
    }
};

// 饿汉式
class Singleton
{
public:
  static Singleton& getInstance() {
        return instance;
    }

public:
    Singleton(const Singleton&) = delete;
    Singleton(Singleton&&) = delete;
    Singleton& operator=(const Singleton&) = delete;
    Singleton& operator=(Singleton&&) = delete;

private:
    static Singleton instance;
    Singleton(){
        std::cout<<"constructor called!"<<std::endl;
    }
    ~Singleton(){
        std::cout<<"destructor called!"<<std::endl;
    }
};

// 默认初始化
Singleton Singleton::instance;
```

由于在main函数之前初始化，所以没有线程安全的问题。但是潜在问题在于no-local static对象（函数外的static对象）在不同编译单元中的初始化顺序是未定义的。也即，static Singleton instance;和static Singleton& getInstance()二者的初始化顺序不确定，如果在初始化完成之前调用 getInstance() 方法会返回一个未定义的实例。

这个web服务器是你自己申请的域名吗

我没有申请域名，但是阿里云服务器上自带一个公网ip，可以直接通过公网ip来访问服务器。但是本地测试的话，我一般使用本地回环ip，127.0.0.1来进行访问。能 ping 通 127.0.0.1 说明本机的网卡和IP协议安装都没有问题

C++ 面向对象特性在项目中的体现

c++面向对象特性有封装、继承、多态。

首先是封装，我在项目中将各个模块使用类进行封装，比如连接用 httpconnection/ftpconnection 类来封装，日志就用 log 类来封装，将类的属性私有化，比如请求的解析状态，并且对外的接口设置为公有，比如连接的重置，不对外暴露自身的私有方法，比如读写的回调函数等。还有一个就是，项目中每个模块都使用了各自的命名空间进行封装，避免了命名冲突或者名字污染。

然后是继承，项目中的继承用得比较少，主要是对工具类的继承，项目中多个地方使用到 noncopyable 和 enable_shared_from_this，保证了代码的复用性。实际上对共有功能可以设计一个基类来继承，比如我项目中的 connection目前有 httpconnection 和 tcpconnection 两种，可以通过继承 connection 基类来减少重复代码，因为我当时做的时候只考虑到 http连接，ftp是后面加上去的，所以就没用这样设计，后面可以优化一下。

最后是多态，我项目中的多态主要用了静态多态，动态多态没有涉及。静态多态在日志系统中对流输入运算符进行了重载，以及在日志系统和内存池中都有各种函数模板的泛型编程。实际上刚刚说的 httpconnection 和 ftpconnection 从 connection 派生出来后是可以使用动态多态的。

> 相似的问题

1. **【中兴一面】**简单介绍一下项目？项目中你的主要工作？为什么要做这么一个项目？
2. **【智行者一面】**介绍项目？跟别人不同的地方？在这个项目中有什么收获？
3. **【经纬恒润一面】**介绍项目（要求着重讲自己与别人不一样的地方，以及自己的收获，balabala）
4. **【经纬恒润二面】**服务器的功能？（HTTP，FTP，存储）为什么要做这么一个项目？（balabala）
5. **【地平线一面】**为什么要做web服务器这个项目（热爱编程，巩固所学知识balabla），有参考开源项目吗，做了哪些优化？
6. **【地平线二面】**讲一下WebServer中你和别人不一样的地方？（主要讲了下自己为了优化服务端性能所做的一些工作）
7. **【荣耀二面】**介绍项目？讲一下自己的项目中和别人不一样的部分？项目有部署上线吗？（自己租了个阿里云服务器，在上面部署过）
8. **【华为一面】**简要介绍项目？为什么要做这么一个项目？和别人不一样的地方？（增加了对服务端性能的一些优化手段）对服务端的性能优化体现在哪？用哪些指标进行评价？是如何进行测量的？你从这个项目中学到了什么，或者说有什么收获？
9. **【华为二面】**为什么要做WebServer这个项目？（学习用的）计算机相关的知识你是怎么进行学习的？
10. **【oppo一面】**介绍项目整体框架（介绍主从reactor+线程池模式）这个框架是你设计的吗？（不是，参考muduo的，然后介绍了另外几种网络框架），有参考过其他人的实现吗（有）你做的工作和其他人有什么不一样？（为了提高服务器效率加入了内存池和缓存机制，同时可以对外提供存储服务）
11. **【oppo二面】**为什么做WebServer这么一个项目，和你专业也不搭，跨度有点大（balabala）介绍一下WebServer的功能，网络框架是怎么样的，监听多少个端口（HTTP，FTP，存储引擎），怎么监听（epoll），介绍一下epoll。用的什么语言？项目中如何体现你的代码规范？（从模块解耦、命名空间、命名

规范等去讲）

12. **【远景智能三面】**很多人的简历上都有这个项目，你做的这个项目有什么和其他人不一样的地方吗？（balabala）刚才提到做了一些性能方面的优化？那对于服务端的性能优化你是怎么理解的？
13. **【蔚来二面】**很多人的简历上都会有这个项目，为什么你还要选择？（项目是在学习计网的过程中逐步搭建的，这个项目综合性比较强，从中既能学习Linux环境下的一些系统调用，也能熟悉网络编程和一些网络框架，其中也根据自己的理解加入了一些性能调优的手段）讲一下你刚才讲的那些性能调优的手段？（从程序本身的优化和系统参数的优化两方面去讲）
14. **【蔚来二面】**为了减少系统调用，你具体做了哪些工作？（分类讲：线程池、内存池、缓存机制；零拷贝；读写锁）
15. **【远景智能三面】**为什么想到去做这么一个项目？你从中学到了什么？这个项目做下来是怎么一个流程？
16. **【经纬恒润二面】**说一下你这个WebServer的整体运行流程？（对着ppt讲）你这个框架的优点是什么？（I/O任务和计算任务解耦，避免计算密集型连接占用subReactor导致无法响应其他连接，可扩展性好）还了解哪些其他网络框架？（Proactor）

项目细节

线程池

你的线程池工作线程处理完一个任务后的状态是什么？

这里要分**两种**情况考虑

（1）当处理完任务后如果请求队列为空时，则这个线程重新回到阻塞等待的状态

（2）当处理完任务后如果请求队列不为空时，那么这个线程将处于与其他线程竞争资源的状态，谁获得锁谁就获得了处理事件的资格。

讲一下你项目中线程池的作用？具体是怎么实现的？有参考开源的线程池实现吗？

分 I/O 线程池和计算线程池去讲。计算线程池参考了。

请你实现一个简单的线程池（现场手撕）

C++实现的简易线程池，**包含线程数量，启动标志位，线程列表以及条件变量**。

- **构造函数**主要是声明未启动和线程数量的。
- **start函数**为启动线程池，将num个线程绑定threadfunc自定义函数并执行，加入线程列表
- **stop**是暂时停止线程，并由条件变量通知所有线程。
- **析构函数**是停止，阻塞所有线程并将其从线程列表剔除后删除，清空线程列表。

```
#pragma once

#include <vector>
#include <queue>
#include <memory>
#include <thread>
```

```cpp
#include <mutex>
#include <condition_variable>
#include <future>
#include <functional>
#include <stdexcept>

class ThreadPool {
public:
    ThreadPool(size_t threads);
    ~ThreadPool();
    template<typename F, typename... Args>
    auto enqueue(F&& f, Args&&... args)
        -> std::future<typename std::result_of<F(Args...)>::type>; // result_of 的作用与
decltype 相同，不国针对的是函数返回值
private:
    std::vector<std::thread> _workers;
    std::queue<std::function<void()>> _tasks;

    std::mutex _queueMutex;
    std::condition_variable _condition;
    bool _stop;
};


inline ThreadPool::ThreadPool(size_t threads) : _stop(false) {
    // 设置线程任务
    for(size_t i = 0; i < threads; ++i) {
        // 每个线程需要做的事情很简单：
        // 1. 从任务队列中获取任务（需要保护临界区）
        // 2. 执行任务
        _workers.emplace_back([this] {
            while(true) {
                std::function<void()> task;
                {
                    std::unique_lock<std::mutex> lock(this->_queueMutex);
                    // 等待唤醒，条件是停止或者任务队列中有任务
                    this->_condition.wait(lock,
                                          [this]{ return this->_stop || !this-
>_tasks.empty(); });
                    if(this->_stop && this->_tasks.empty())
                        return;
                    task = std::move(this->_tasks.front());
                    this->_tasks.pop();
                }

                task();
            }
        });
    }
```

```
}

template<typename F, typename... Args>
auto ThreadPool::enqueue(F&& f, Args&&... args)
    -> std::future<typename std::result_of<F(Args...)>::type>
{
    using return_type = typename std::result_of<F(Args...)>::type;

    // 将需要执行的任务函数打包（bind），转换为参数列表为空的函数对象
    auto task = std::make_shared<std::packaged_task<return_type()>> (
        std::bind(std::forward<F>(f), std::forward<Args>(args)...)
    );

    std::future<return_type> res = task->get_future();
    {
        std::unique_lock<std::mutex> lock(_queueMutex);

        if(_stop)
            throw std::runtime_error("enqueue on stopped ThreadPool");

        // 最妙的地方，利用 lambda函数 包装线程函数，使其符合 function<void()> 的形式
        // 并且返回值可以通过 future 获取
        _tasks.emplace([task]() {
            (*task)();
        });
    }

    _condition.notify_all();
    return res;
}


inline ThreadPool::~ThreadPool() {
    {
        std::unique_lock<std::mutex> lock(_queueMutex);
        _stop = true;
    }
    _condition.notify_all();  // 唤醒所有线程，清空任务
    for(std::thread& worker : _workers) {
        worker.join();
    }
}
```

> 一些问题

1. **【远景智能三面】**项目的中线程池是怎么搭建的，大概讲一下原理？（项目中的线程池有两种，分别去讲了一下）

日志系统

> 一些问题

1. **【蔚来一面】**异步日志系统是怎么实现异步的？业务线程的写入是怎么同步的？（加锁），回去了解一下无锁队列
2. **【蔚来二面】**如果服务器在运行的过程中，实际存储的文件被其他用户修改了，会发生什么？（这部分没考虑到。。。）有什么优化的想法吗？（想法一（被否决）：定时和实际文件对比；想法二（被否决）：进程修改后通知缓存；想法三：类似于内存页面的换入换出，每次从cache读入时，先判断实际文件去对比，可以对比最后修改时间或者通过摘要算法判断）
3. **【经纬恒润一面】**为什么要做日志系统？讲一下日志系统的双缓冲？异步和同步的区别？
4. **【经纬恒润二面】**日志系统的异步体现在什么地方？与同步的区别？为什么设计成双缓冲而不用更多的缓冲区？（前端缓冲不足时会自动扩展，但是双缓冲足够应付使用场景，因为日志只记录必要的信息，并不会太多）
5. **【地平线二面】**讲一下你的日志系统的怎么做的（balabala）？异步指的是什么？异步和同步的区别？如果服务器崩溃的话，你的日志系统会发生什么？（未写入文件的所有日志都会丢失，但是会有时间戳）这个问题应该怎么解决？（想了好久也没什么好的想法，最后扯了点。。。如果是进程crash的话，可以使用一个日志进程来写入日志；如果是主机宕机的话可以考虑分布式日志，将日志分发到不同的主机上写入）

缓存机制

> 一些问题

1. **【中兴一面】**讲一下你项目中缓存池的作用？为什么要选择 LFU 而不是 LRU？LFU有什么缺点？（最近加入的数据因为起始的频率很低，容易被淘汰，而早期的热点数据会一直占据缓存）可以进行优化吗？（LFU-aging，或者使用LRU+LFU的形式）
2. **【蔚来二面】**缓存机制为什么选用LFU？（主要考虑热点页面）热点页面指的什么？怎么知道是热点页面？有考虑过其他的缓存算法吗？（LRU，ARC）为什么不选用？（balabala）LFU什么缺陷吗，可以怎么进行优化？（LFU-aging，window-LFU，ARC）
3. **【经纬恒润一面】**为什么用LFU，不用LRU？（保证热点页面的响应）LFU的实现可以讲一下吗？（双链表+双哈希）
4. **【oppo一面】**讲一讲为什么要加入缓存机制，常用的缓存算法有哪些？LFU的缺点是什么？怎么优化？（LFU-aging）

内存池

讲一讲为什么要加入内存池？项目中所有的内存申请都走内存池吗？

【oppo一面】连接对象和缓存对象都走内存池，**这里挖坑了**

- 相比于 `new` 的性能提升在哪？（避免了系统调用的开销），`new` 的系统调用开销有多少？（纳秒级），可以忽略吗？（不能，在高并发场景下，越往后的连接延迟越明显）
- `new` 的主要开销在哪？（系统调用和构造函数）
- `new` 一定会陷入内核态吗？（不一定，因为底层是 `malloc`，`malloc` 根据分配内存的大小不同有两种分配方式，小于128k使用`brk()`，大于 `128k` 使用 `mmap()` ）

- 那你内存池对于小内存的申请相比 `new` 还有优势吗？（到这里才明白面试官挖的坑，赶紧sorrymaker，考虑不周），但是对于缓存的对象是有必要走内存池的，下去再好好理一理。（学到了）

并发性问题

如果同时1000个客户端进行访问请求，线程数不多，怎么能及时响应处理每一个呢？

首先这种问法就相当于问服务器如何处理高并发的问题。

首先我项目中使用了I/O多路复用技术，每个线程中管理一定数量的连接，只有线程池中的连接有请求，epoll就会返回请求的连接列表，管理该连接的线程获取活动列表，然后依次处理各个连接的请求。如果该线程没有任务，就会等待主reactor分配任务。这样就能达到服务器高并发的要求，同一时刻，每个线程都在处理自己所管理连接的请求。

如果一个客户请求需要占用线程很久的时间，会不会影响接下来的客户请求呢，有什么好的策略呢？

> 影响分析

会影响这个大请求的所在线程的所有请求，因为每个eventLoop都是依次处理它通过epoll获得的活动事件，也就是活动连接。如果该eventloop处理的连接占用时间过长的话，该线程后续的请求只能在请求队列中等待被处理，从而影响接下来的客户请求。

> 应对策略

1. **主 reactor 的角度**：可以记录一下每个从reactor的阻塞连接数，主reactor根据每个reactor的当前负载来分发请求，达到负载均衡的效果。
2. **从 reactor 的角度**：
    - **超时时间**：为每个连接分配一个时间片，类似于操作系统的进程调度，当当前连接的时间片用完以后，将其重新加入请求队列，响应其他连接的请求，进一步来说，还可以为每个连接设置一个优先级，这样可以优先响应重要的连接，有点像 HTTP/2 的优先级。
    - **关闭时间**：为了避免部分连接长时间占用服务器资源，可以给每个连接设置一个最大响应时间，当一个连接的最大响应时间用完后，服务器可以主动将这个连接断开，让其重新连接。

> 相似的问题

1. **【荣耀二面】**如果一个连接请求耗时非常长，会发生什么情况？（反问是IO耗时，还是计算耗时？如果是IO耗时的话，则会阻塞同一subreactor中的所有线程，如何是计算耗时的话，因为本身也设计了计算线程池，对服务器本身并没有太多影响）
2. **【荣耀二面】**如果一个连接请求的资源非常大，在发送响应报文时会造成响应其他连接的请求吗？有什么优化的方法？（断点续传，设置响应报文的大小上限，当响应报文超出上限时，可以记录已经发送的位置，之后可以选择继续由该线程进行发送，也可以转交给其他线程进行发送）

IO多路复用

说一下什么是ET，什么是LT，有什么区别?

1. LT：水平触发模式，只要内核缓冲区有数据就一直通知，只要socket处于可读状态或可写状态，就会一直返回sockfd；是默认的工作模式，支持阻塞IO和非阻塞IO
2. ET：边沿触发模式，只有状态发生变化才通知并且这个状态只会通知一次，只有当socket由不可写到可写或由不可读到可读，才会返回其sockfd；只支持非阻塞IO

LT什么时候会触发？ET呢?

> LT模式

1. **对于读操作**
    - 只要内核读缓冲区不为空，LT模式返回读就绪。
2. **对于写操作**
    - 只要内核写缓冲区还不满，LT模式会返回写就绪。

> ET模式

1. **对于读操作**
    - 当缓冲区由不可读变为可读的时候，即缓冲区由空变为不空的时候。
    - 当有新数据到达时，即缓冲区中的待读数据变多的时候。
    - 当缓冲区有数据可读，且应用进程对相应的描述符进行EPOLL_CTL_MOD 修改EPOLLIN事件时。
2. **对于写操作**
    - 当缓冲区由不可写变为可写时。
    - 当有旧数据被发送走，即缓冲区中的内容变少的时候。
    - 当缓冲区有空间可写，且应用进程对相应的描述符进行EPOLL_CTL_MOD 修改EPOLLOUT事件时。

为什么ET模式不可以文件描述符阻塞，而LT模式可以呢?

- 因为ET模式是当fd有可读事件时，epoll_wait()只会通知一次，如果没有一次把数据读完，那么要到下一次fd有可读事件epoll才会通知。而且在ET模式下，在触发可读事件后，需要循环读取信息，直到把数据读完。如果把这个fd设置成阻塞，数据读完以后read()就阻塞在那了。无法进行后续请求的处理。
- LT模式不需要每次读完数据，只要有数据可读，epoll_wait()就会一直通知。所以 LT模式下去读的话，内核缓冲区肯定是有数据可以读的，不会造成没有数据读而阻塞的情况。

你用了epoll，说一下为什么用epoll，还有其他多路复用方式吗？区别是什么?

> 文件描述符集合的存储位置

对于 select 和 poll 来说，所有文件描述符都是在用户态被加入其文件描述符集合的，每次调用都需要将整个集合拷贝到内核态；epoll 则将整个文件描述符集合维护在内核态，每次添加文件描述符的时候都需要执行一个系统调用。系统调用的开销是很大的，而且在有很多短期活跃连接的情况下，由于这些大量的系统调用开销，epoll 可能会慢于 select 和 poll。

> 文件描述符集合的表示方法

select 使用**线性表**描述文件描述符集合，文件描述符有上限；poll使用**链表**来描述；epoll底层通过**红黑树**来描述，并且维护一个就绪列表，将事件表中已经就绪的事件添加到这里，在使用epoll_wait调用时，仅观察这个list中有没有数据即可。

> 遍历方式

select 和 poll 的最大开销来自内核判断是否有文件描述符就绪这一过程：每次执行 select 或 poll 调用时，它们会采用遍历的方式，遍历整个文件描述符集合去判断各个文件描述符是否有活动；epoll 则不需要去以这种方式检查，当有活动产生时，会自动触发 epoll 回调函数通知epoll文件描述符，然后内核将这些就绪的文件描述符放到就绪列表中等待epoll_wait调用后被处理。

> 触发模式

select和poll都只能工作在相对低效的LT模式下，而epoll同时支持LT和ET模式。

> 适用场景

当监测的fd数量较小，且各个fd都很活跃的情况下，建议使用select和poll；当监听的fd数量较多，且单位时间仅部分fd活跃的情况下，使用epoll会明显提升性能。

> 相似的问题

1. **【远景智能三面】**io多路复用是同步还是异步？同步io和异步io有什么区别？为什么在项目中不用异步io？（linux本身提供的asio目前只支持文件fd，不支持网络fd，但是可以用一个线程去模拟异步io的操作）

并发模型

reactor、proactor模型的区别?

1. **Reactor 是非阻塞同步网络模式，感知的是就绪可读写事件**。在每次感知到有事件发生（比如可读就绪事件）后，就需要应用进程主动调用 read 方法来完成数据的读取，也就是要应用进程主动将 socket 接收缓存中的数据读到应用进程内存中，这个过程是同步的，读取完数据后应用进程才能处理数据。
2. **Proactor 是异步网络模式， 感知的是已完成的读写事件**。在发起异步读写请求时，需要传入数据缓冲区的地址（用来存放结果数据）等信息，这样系统内核才可以自动帮我们把数据的读写工作完成，这里的读写工作全程由操作系统来做，并不需要像 Reactor 那样还需要应用进程主动发起 read/write 来读写数据，操作系统完成读写工作后，就会通知应用进程直接处理数据。

> Proactor这么好用，那你为什么不用?

在 Linux 下的异步 I/O 是不完善的， `aio` 系列函数是由 POSIX 定义的异步操作接口，不是真正的操作系统级别支持的，而是在**用户空间模拟出来的异步**，并且仅仅支持基于本地文件的 aio 异步操作，网络编程中的 socket 是不支持的，也有考虑过使用模拟的proactor模式来开发，但是这样需要浪费一个线程专门负责 IO 的处理。

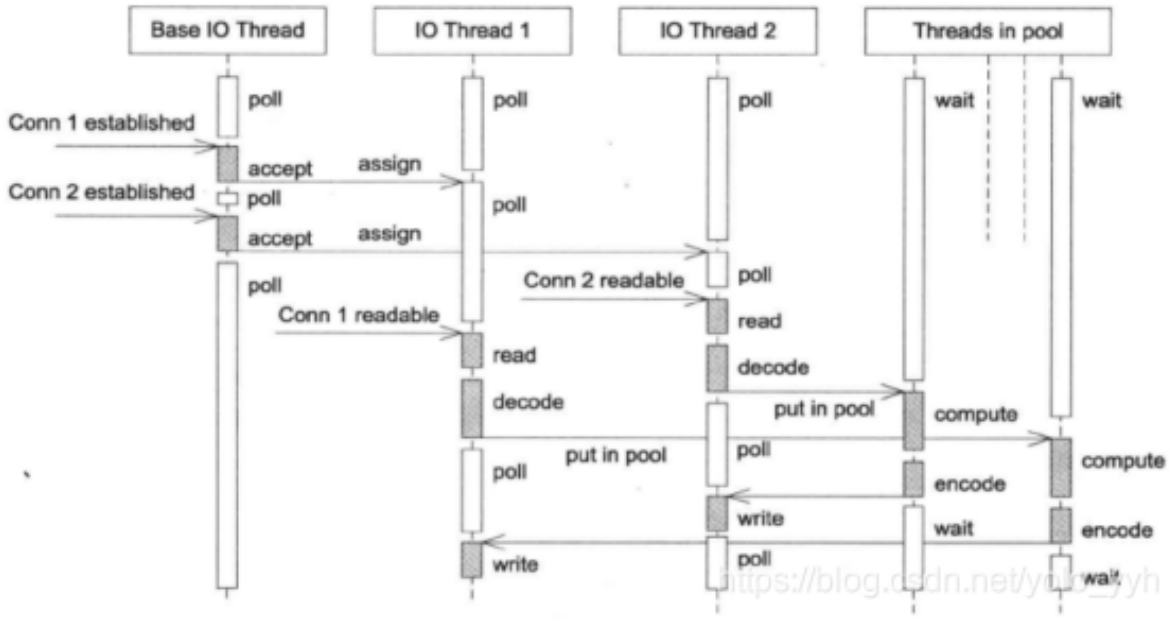
而 Windows 里实现了一套完整的支持 socket 的异步编程接口，这套接口就是 `IOCP`，是由操作系统级别实现的异步 I/O，真正意义上异步 I/O，因此在 Windows 里实现高性能网络程序可以使用效率更高的 Proactor 方案。

reactor模式中，各个模式的区别?

Reactor模型是一个针对同步I/O的网络模型，主要是使用一个reactor负责监听和分配事件，将I/O事件分派给对应的Handler。新的事件包含连接建立就绪、读就绪、写就绪等。reactor模型中又可以细分为单reactor单线程、单reactor多线程、以及主从reactor模式。

1. **单reactor单线程**模型就是使用 I/O 多路复用技术，当其获取到活动的事件列表时，就在reactor中进行读取请求、业务处理、返回响应，这样的好处是整个模型都使用一个线程，不存在资源的争夺问题。但是如果一个事件的业务处理太过耗时，会导致后续所有的事件都得不到处理。
2. **单reactor多线程**就是用于解决这个问题，这个模型中reactor中只负责数据的接收和发送，reactor将业务处理分给线程池中的线程进行处理，完成后将数据返回给reactor进行发送，避免了在reactor进行业务处理，但是 IO 操作都在reactor中进行，容易存在性能问题。而且因为是多线程，线程池中每个线程完成业务后都需要将结果传递给reactor进行发送，还会涉及到共享数据的互斥和保护机制。
3. **主从reactor**就是将reactor分为主reactor和从reactor，主reactor中只负责连接的建立和分配，读取请求、业务处理、返回响应等耗时的操作均在从reactor中处理，能够有效地应对高并发的场合。

subreactor负责读写数据，由线程池进行业务处理



> 类似的问题

1. **【中兴一面】**讲一下项目中的网络框架？为什么要采用这个框架？有了解其他的方案吗？为什么不采用？
2. **【经纬恒润一面】**为什么选择主从reactor+线程池的模式（将IO操作和业务处理解耦）？epoll配合什么IO使用？简单介绍下epoll？了解proactor吗，讲一下区别？
3. **【荣耀二面】**介绍一下项目中的网络框架？为什么选择这个网络框架？

跳表 skiplist

由于我将卡哥的跳表项目加入了 WebServer，所以面试的时候有面试官会问到这部分的内容。具体的实现比较简单，大家直接看卡哥的项目[youngyangyang04/Skiplist-CPP: A tiny KV storage based on skiplist written in C++ language| 使用C++开发，基于跳表实现的轻量级键值数据库🔥🔥🚀 (github.com)](youngyangyang04/Skiplist-CPP: A tiny KV storage based on skiplist written in C++ language| 使用C++开发，基于跳表实现的轻量级键值数据库🔥🔥🚀 (github.com))就好啦~

> 一些问题

1. **【中兴一面】**讲一下跳表的原理？使用场景？如何保证查询效率？（通过概率近似保证当前层的索引数是上一层的一半）
2. **【远景智能三面】**存储引擎的定时写入是怎么实现的？每次都将内存中的所有数据写入文件，没有改变的数据是否会重复写入？（会，可以通过在文件写入实际操作来避免）那实际操作在系统中会出现相互抵消的情况，如何优化？（开一个后台线程，定时合并实际操作，并将实际操作应用到文件中）
3. **【联想二面】**介绍一下你web服务器的存储引擎，为什么要做这么一个存储引擎？（学习数据库的时候接触到了，比较感兴趣所以模仿做了一下）讲一下跳表的原理，查找是怎么做的（从顶层节点开始找，方式类似于二分查找），插入节点的过程是怎么样的（先将节点插入最底层，然后为了保证保证查找效率，上一层的节点数需要近似为下一层的1/2，因此通过随机数的方式模拟这个概率，如果是偶数则在当前层记录节点，否则继续进入下一层），删除节点呢（从最底层往上层去删除即可）B+树了解吗，说一下B+树的底层实现，说一下B+树和跳表的区别？

测试相关问题

我在秋招的时候主攻的是软开和测开，在面试测开岗位时有的面试官会问到项目测试的相关内容，这部分只要这个项目是自己从头到尾做过的，都是能够应对的。但是面试官后续会根据这些内容去扩展，这就需要自己灵活变通了~

你是如何对项目进行测试的?

对项目的测试分为两部分：基本功能模块的测试和扩展功能模块的测试。

【基本功能测试】

1. 日志系统
    - 写入日志类型、等级
    - 多个线程同时写入的响应
    - 短期大量日志进行写入
2. 网络框架
    - 正确接收 client 连接和分发
    - 正确感知 client 连接的读写事件
    - 正确对 client 连接进行读写
    - 高并发、多请求（webbench）
3. 存储引擎
    - 正确写入和读取
    - 定时落盘（写入文件和定时写入）
    - 多个线程同时读取和写入
    - 单个线程读取和写入大量数据

【扩展功能测试】

1. 线程池
    - 能否将任务放入任务队列，任务是否能够正确执行
    - 多个线程同时往任务队列放入任务
2. 内存池
    - 能否正确分配内存（空对象、大内存、小内存），调用对象的构造函数和析构函数
    - 多个线程同时申请内存

3. 缓存机制
    - 能否成功命中缓存
    - 多个线程读取缓存

> 一些问题

1. **【华为一面】**在开发过程中有进行过测试吗？（做了单元测试、集成测试、功能测试、压力测试）压力测试是怎么做的？（分网络框架、日志系统、存储引擎三部分去讲）
2. **【联想一面】**在做这个项目的过程中做了哪些测试，发现了什么问题？有针对遇到的问题去做优化吗？（主要讲了下日志系统的并发性问题）

星球资料

- HttpServer V2.0总结
- webserver秋招面经
- webserver秋招面经
- 百度，美团offer，C++方向，webserver项目，准备的心路历程