第17课 项目总结

这个C++ WebServer是一个高性能、事件驱动的Web服务器，它利用了C++的新特性来提高代码的可读性和效率。以下是该服务器的主要特点和优势功能的梳理：

不是只看了这个文章就完全够了，比如老师追问理论知识还是要回去看之前的课程文档的

重点掌握部分：

第五课服务器基本内容 第八课数据库注入与预编译 **第九课事件驱动 第十课多线程** 第十三课分布式 第十四课大数据 小班单独加分项

选择性掌握一些C++新特性，尽量掌握C++基本特性

复习逻辑：先理解基本服务器的运行原理- > 去复习重点掌握部分 -> 选择自己擅长的写入简历，并自己能复述出来其 原理 优势 代码中的使用（结合注释介绍那块逻辑）

项目基本架构

- **事件驱动模型**：采用 `epoll` 作为I/O多路复用技术，实现非阻塞I/O操作，提高服务器处理并发连接的能力。
  - 重点掌握事件驱动的理论原理、优势、epoll引入后服务器的运行逻辑复述）
- 线程池：通过预创建一定数量的线程并将它们放入线程池中，可以有效地管理和复用线程资源，减少线程创建和销毁的开销，提高服务器响应速度。
- 线程安全：使用互斥锁（如 `std::mutex` ）保护共享资源，确保服务器在多线程环境下的稳定运行。
- 日志系统：用宏函数方便的实现Logger，便于记录服务器运行信息
  - （这里主要是宏函数相关）
- 前端交互界面：提供静态页面服务和RESTful API，支持与前端页面的交互，实现动态Web应用。

优势功能（你写进简历的前提是你确实能说上一些，千万不要写自己讲不出来的东西进去）

- C++新特性：大量使用C++11及更高版本的新特性，如智能指针、lambda表达式、并发库等，提高了代码的安全性和简洁性。**（掌握哪些就写哪些）**
  - 线程池这里用到了很多C++新特性，可以在这里选择C++新特性在面试中介绍
- 分布式反向代理：通过与NGINX等反向代理服务器配合，实现负载均衡和分布式部署，提高服务器的可扩展性和可靠性。

- SQL防注入：采用预处理语句和参数化查询，有效防止SQL注入攻击，保证数据库操作的安全。
- MongoDB数据库：引入MongoDB作为大数据数据库，支持高性能、高可用性和易扩展性，适合处理大规模数据存储。
- SSL安全性：数字签名 数字证书 对称加密和非对称加密等概念，SSL握手的步骤

总结

这个C++ WebServer结合了现代Web服务器的多项先进技术和实践，包括事件驱动模型、线程池技术、反向代理与负载均衡、以及对新兴数据库的支持。通过这些技术的应用，服务器能够提供高性能、高可靠性和高扩展性的Web服务。同时，服务器还注重安全性，采取了预编译措施来防止常见的SQL注入攻击，通过SSL实现服务器安全性的提高

（具体介绍在下面的问题里也有

重要知识点

1. **关于事件驱动模型：**
   - 介绍一下什么是事件驱动模型，和传统的模型有什么区别
     - **事件驱动模型是一种编程范式，服务器在这种模型下通过监听事件来驱动程序运行，而不是顺序执行或者等待某个操作完成**。这些事件通常包括网络IO事件（如可读、可写），定时器事件等。
     - 传统的阻塞IO模型在执行IO操作（如读取网络数据）时，如果没有数据可读，会导致线程挂起等待，直到有数据可读或者超时。这种模型在处理大量并发连接时效率低下，因为大部分时间线程都在等待。
     - 事件驱动模型则不同，它允许程序在一个线程内监听多个IO事件，当某个事件就绪（例如某个socket可读）时，才会执行相应的处理，从而提高效率和减少资源消耗。
   - 请详细描述一下在您的Web服务器中如何利用epoll实现I/O多路复用技术，并阐述其在处理高并发连接时的具体优势？

     在我的Web服务器中，通过epoll实现I/O多路复用技术主要分为几个步骤：

     i. 创建epoll实例：首先通过调用 `epoll_create1(0)` 创建一个epoll的文件描述符。

     ii. 注册感兴趣的事件：使用 `epoll_ctl` 将服务器的监听套接字（server_fd）添加到epoll实例中，并注册EPOLLIN事件（表示对读事件感兴趣）。对于每个客户端连接，根据需要也会注册EPOLLOUT事件（表示对写事件感兴趣）。

iii. 等待事件发生：通过调用 `epoll_wait` 等待事件的发生。这个调用是非阻塞的，它会返回一组发生了注册事件的文件描述符。

iv. 处理事件：根据 `epoll_wait` 返回的事件类型（读或写），进行相应的操作，如接受新连接、读取数据或发送数据。

优势：epoll相较于select和poll的主要优势在于它能够更高效地处理大量并发连接。epoll不需要像select和poll那样每次调用时都重复传递监听列表，它仅通知活跃的连接，减少了不必要的检查，大大提升了性能。

- 介绍reactor和proactor的区别与模式
  - Reactor模式：在Reactor模式中，应用程序反应于I/O事件，并且同步地进行读写操作。即，当I/O事件就绪时，事件分发器将事件通知给相应的事件处理器，然后同步执行I/O操作（如读取数据、写入数据）。Reactor模式适合于I/O操作不会导致阻塞的场景，因为I/O操作是同步执行的。
  - Proactor模式：Proactor模式采用异步I/O操作。应用程序启动一个操作，然后立即返回，不需要等待I/O操作完成。当I/O操作完成后，系统会自动通知应用程序，应用程序随后处理这个通知。这种模式适合于I/O操作可能导致阻塞的场景，因为它允许应用程序在等待I/O完成的同时继续执行其他任务。
- 当接收到大量并发请求时，您的事件循环是如何设计和管理这些事件的？有没有遇到过什么挑战或者性能瓶颈？

  我的事件循环设计侧重于通过epoll高效地管理和调度事件。对于每个事件（连接请求、读事件、写事件），事件循环都会根据事件类型调用相应的处理函数。

  在处理大量并发请求时，我遇到的挑战主要是如何有效地分配和管理线程池中的线程，以及如何避免锁竞争和提高内存使用效率。为了应对这些挑战，我采取了以下措施：
  - 线程池：使用固定大小的线程池来处理事件，减少了线程创建和销毁的开销，同时也避免了过多的线程竞争导致的性能下降。
  - 细粒度锁：尽量减少锁的使用范围和持有时间，避免大锁带来的性能瓶颈。
  - 连接状态管理：通过在每个连接上维护状态（如是否完成读取、是否准备好写入），减少不必要的操作，提升效率。

  在实践中，我发现合理的设计和优化可以有效地提高服务器处理高并发请求的能力，但同时也需要不断地根据实际运行情况进行调整和优化

2. 线程池设计：

- 服务器中线程池和多线程的作用是什么

  i. 并发处理能力：
  - 多线程允许服务器应用程序同时执行多个任务，每个线程负责处理一个客户端请求或一部分任务。这样可以充分利用多核CPU的计算能力，提高系统吞吐量和服务响应速度。

ii. 资源复用：

- 线程池是对多线程概念的一种优化实现。它创建一组可重用的线程来服务请求，而不是为每个新的请求创建新线程。通过复用线程，减少了线程创建和销毁带来的开销（如内存分配、上下文切换），使得系统能够更高效地处理大量短生命周期的任务。

- 在线程池中，任务调度和分配机制是怎样的？如何避免线程饥饿或优先级反转等问题？

  在提供的线程池实现中，任务调度和分配机制如下：

  - 所有工作线程从一个共享的任务队列中取任务执行。
  - 使用互斥锁 `queue_mutex` 保证任务队列的线程安全。
  - 使用条件变量 `condition` 来等待任务的到来或线程池的停止指令。

  为了避免线程饥饿或优先级反转等问题，可以采取以下措施：

  - 公平调度：确保任务按照到达线程池的顺序被处理，这是通过使用先进先出（FIFO）队列实现的。
  - 动态任务优先级：如果需要，可以实现一个优先队列来处理不同优先级的任务，确保高优先级任务能被优先处理。
  - 避免长时间任务：对于可能执行很长时间的任务，考虑将其分解为更小的子任务，这样可以避免单个任务占用线程过久，导致其他任务饥饿。

- 你是否引入了性能监控工具来跟踪服务器的运行状态？在遇到性能瓶颈时，你是如何定位和解决问题的？

  - 用ApacheBench（ab）工具对服务器性能进行基准测试，来测试服务器的能力。我是增量判断，比如开始当我引入事件驱动后发现性能反而变差了，我就判断是新加入的事件驱动逻辑有问题，后来果然发现是事件管理的代码逻辑出了问题（连接忘了关闭或者事件没有及时处理，或者就是语法错了，随便找一个理由就行）

3. C++新特性应用：

- 可否举例说明在该项目中如何使用C++新特性
  - 以线程池为例：
    1. `std::thread`：用于创建和管理线程。这使得编写多线程程序变得更简单，不再依赖于操作系统特定的线程库。
    2. Lambda表达式：用于创建匿名函数对象，使得代码更加简洁。在这里，它被用于定义线程执行的任务和入队任务。
    3. `std::mutex` 和 `std::unique_lock`：用于同步访问共享数据。`std::mutex` 提供了互斥锁的基本功能，而 `std::unique_lock` 则是一个灵活的互斥锁管理器，它提供了锁的所有权的自动管理，简化了线程同步操作。
    4. `std::condition_variable`：用于线程间的条件同步。它可以让一个或多个线程在某个条件成立之前处于等待状态，从而有效地管理线程间的工作流程。

5. `std::function` 和 `std::bind`： `std::function` 是一个通用的函数包装器，可以存储、调用任何函数、Lambda表达式、函数指针、以及其他可调用对象。 `std::bind` 用于绑定函数调用的参数，这两者结合使用，提供了强大的函数操作能力，使得任务调度更加灵活。
6. `std::future` 和 `std::packaged_task`：这些用于异步操作和结果传递。 `std::packaged_task` 包装一个可调用对象，并允许异步执行它。 `std::future` 提供了一种访问异步操作结果的机制。
7. 自动类型推导 (`auto`)：用于自动推导变量的类型，减少了代码的冗余，使得代码更加清晰。
8. 尾置返回类型 (`->`)：允许在函数声明之后指定返回类型，特别是在返回类型依赖于模板参数时非常有用。

- 对于并发库的使用，具体在哪些部分应用了C++11或更高版本的并发特性？它们在提升服务器性能方面起到了什么作用？

4. 安全性

- 能否详细讲解SQL注入是什么，你项目中如何预防的？/你的项目有没有考虑服务器的安全性
  - SQL注入（SQL Injection）是一种常见的网络安全攻击手段，它发生在应用程序没有对用户输入的数据进行有效验证和过滤的情况下。攻击者通过在提交到服务器的查询语句中插入恶意的SQL代码片段，从而篡改原意、获取未经授权的信息或执行非预期的操作。
  - 预编译SQL语句（sqlite3_prepare_v2）

i. 将SQL语句转换为预编译语句对象：

- **当使用 `sqlite3_prepare_v2` 函数时，提供的SQL语句文本不会立即执行。相反，它被编译成一个预编译语句对象**，这个对象在数据库内部表示SQL语句的结构和操作。
- 这个预编译过程类似于编译器如何处理源代码。它解析SQL语句，将其转换为数据库可以理解的形式。

ii. 固定SQL语句结构：

- 预编译的过程中，SQL语句的结构被固定下来。这意味着，**一旦语句被预编译，其基本结构（如查询的表和列）就不能被更改。**
- **这种方式使得外部输入无法改变SQL语句的基本结构**，从而有效防止了SQL注入攻击。

iii. 预编译与直接拼接的对比：

- 直接拼接SQL字符串可能会受到用户输入的影响。如果用户输入包含SQL语句的一部分，那么拼接后的SQL可能会执行意外的命令。

- **预编译语句不受用户输入的特殊字符影响，因为这些输入仅在执行时被视为数据，而不是SQL命令的一部分**

- 线程安全如何处理
  - 使用互斥锁（如 `std::mutex`）保护共享资源，确保服务器在多线程环境下的稳定运行。
  - 也就是操作数据库之前先上锁，保证同一时刻只有一个线程在访问数据库

```
bool storeImage(const std::string& imageName, const std::string& imagePath, const std::string& description) {
    // 使用std::lock_guard保证在操作数据库时线程安全，自动锁定并解锁互斥锁dbMutex
    std::lock_guard<std::mutex> guard(dbMutex);

    try {
        // 创建一个BSON文档构建器，将图片的相关信息（名称、路径和描述）添加到文档中
        bsoncxx::builder::stream::document document{};
        document << "name" << imageName    // 图片的名称
```

追问：那应该有几个锁？整个数据库用一个锁吗？

不是，理论上应该一种数据一个锁，比如用户信息一个锁，文件管理另一个锁，否则效率太低，锁的粒度越细效率越高，但管理也会越复杂。

5. 分布式反向代理与安全防护：

- 为什么要用分布式，分布式服务器优势/解决了哪些问题（不用全背下来，重点是理解。面试能答上来几个即可）

i. 可扩展性问题

随着用户数量的增加和业务逻辑的复杂化，单台服务器往往难以承载日益增长的访问量和处理需求。分布式服务器通过将负载分散到多个服务器上，实现了系统的横向扩展，即通过增加更多的服务器来提升系统的处理能力和存储容量，从而有效应对大规模并发请求和数据增长。

i. 高可用性和容错性

单点故障是单台服务器系统的一个重大弱点，一旦服务器出现故障，整个服务将不可用。分布式服务器通过在多个节点间复制数据和任务，实现了服务的高可用性。当某个节点发生故障时，其他节点可以接管任务，保证服务的持续可用。这种设计也提高了系统的容错性，即使部分系统组件失败，整个系统仍能正常运行。

ii. 资源利用率

在单服务器架构中，为了应对高峰期的访问压力，服务器往往需要配置高性能的硬件资源，这导致在非高峰期时资源利用率低下。分布式服务器可以根据实际的负载情况动态调整资源分配，通过负载均衡技术优化资源的使用效率，提高整体的资源利用率。

iii. 系统维护和升级的灵活性

在分布式服务器架构中，系统的每个组件都可以独立部署和升级，这大大提高了系统维护和升级的灵活性。开发团队可以针对特定的服务进行优化和升级，而不需要停机或影响到整个系统的运行。

iv. 地理分布式部署

分布式服务器可以根据用户的地理位置分布在不同的数据中心，这样用户可以就近访问服务器，减少网络延迟，提高访问速度和用户体验。

- 在实现与NGINX等反向代理服务器配合时，如何设计通信接口以支持负载均衡？对于分布式部署场景，您做了哪些优化以确保服务的高可用性？

  i. 负载均衡：

  - 使用 `upstream` 模块定义了一个名为 `myapp` 的后端服务器池，其中包含了三个本地服务器，并分别设置了权重。这意味着Nginx会根据这些权重将请求分发到不同的后端服务器上，实现负载均衡。（这里指负载均衡策略，根据你自己的情况换成其他的策略就可以）

  ii. 故障转移与恢复：

  - 当Nginx检测到某个后端服务器不可用时，它会自动停止向该服务器发送新的请求，并将请求转发至其他健康的服务节点，从而保证了服务的连续性和可用性。

  iii. 分布式部署优化：

  - 配置代理缓存(`proxy_cache`)，对特定路径如 `/login` 和 `/register` 的GET响应进行缓存，这有助于减轻后端服务器的压力，尤其是对于那些不常变化且计算密集型的响应内容，提高了整体性能和响应速度。
  - 合理设置缓存有效时间(`proxy_cache_valid`)

6. MongoDB数据库集成：

- 描述一下在这个项目中MongoDB是如何被用于存储大规模数据的，MongoDB有哪些特性？为什么选择MongoDB作为项目的数据库

  项目中，MongoDB作为NoSQL数据库，被用于存储用户信息、登录信息以及图片信息。MongoDB的文档模型、高性能、高可用性和易扩展性特性，使其成为处理大规模数据的理想选择。

  高性能

  MongoDB的文档数据模型允许它存储复杂的嵌套数据结构，这通常比关系型数据库的表结构更加自然和高效。MongoDB还支持索引，包括对文档内部的字段进行索引，这大大提高了查询速度，尤其是在处理大规模数据时。

  高可用性

  MongoDB通过副本集实现高可用性。副本集是MongoDB的一组服务器，其中包含一个主节点和多个副本节点，这些副本节点可以在主节点发生故障时自动接管，保证数据库服务不中断。这种机制保证了数据的高可用性和灾难恢复。

  易扩展性

MongoDB支持水平扩展，通过分片技术可以将数据分布在多个服务器上。随着数据量的增加，可以通过增加更多的服务器来分散负载和存储需求，从而轻松扩展数据库的规模。这种灵活的扩展能力使得MongoDB非常适合处理大规模数据集。

7. **前端交互支持（这部分不想记就说前端做的比较简单，或者和别人合作，他负责的，自己重心在后端）**
   - 这个Web服务器是如何提供静态页面服务和RESTful API的？在设计API时遵循了哪些原则以保证良好的前后端分离及可维护性？
     - 服务器能够响应客户端请求并返回静态HTML页面。这是通过设置特定的路由和处理函数来实现的。例如，对于登录和注册页面，服务器配置了特定的路由来返回相应的HTML文件。这些HTML文件位于服务器的文件系统中，当请求到来时，服务器读取相应的文件内容并作为HTTP响应的正文返回给客户端。
     - 在设计API时，服务器遵循了以下原则以保证良好的前后端分离及可维护性：
       - 清晰的资源定位：每个API都有一个清晰和直观的URI，代表了服务器上的一个资源或资源集合，如 `/login`、`/register`、`/upload` 等。
       - 使用标准的HTTP方法：API设计遵循HTTP方法的语义，使用GET方法获取资源，POST方法创建或更新资源。
       - 无状态：每次API请求都是独立的，服务器不保存客户端的状态，这简化了服务器的设计，也使得每个请求可以独立处理。
       - 统一的响应结构：无论是成功还是错误的响应，服务器都返回统一格式的JSON对象，包含了必要的信息，如错误代码和描述信息。这使得客户端可以更容易地处理响应。
   - 什么是restful API
     - REST（Representational State Transfer，表现层状态转移）是一组架构约束条件和原则，RESTful API就是遵循这些原则和约束条件的API。它使用HTTP协议的标准方法（如GET、POST、PUT、DELETE等）来实现资源的创建、读取、更新和删除操作（CRUD操作）。

       RESTful API的核心原则包括：

       1. 资源（Resources）：在RESTful架构中，一切都被视为资源，每个资源都有其唯一的资源标识符（URI），通过这个URI就可以访问到该资源。
       2. 无状态（Stateless）：每次请求之间是相互独立的，服务器不存储之前请求的状态信息。这意味着每个请求都必须包含所有必要的信息，服务器通过这些信息来处理请求。
       3. 统一接口（Uniform Interface）：RESTful API具有统一的接口，使用标准的HTTP方法对资源进行操作，使得从服务器到客户端的交互变得简单、统一。

4. 可缓存（Cacheable）：RESTful API的响应结果可以被标记为可缓存或不可缓存。如果是可缓存的，客户端就可以重用之前请求的响应结果，提高应用性能和效率。
5. 层次化系统（Layered System）：REST允许使用层次化的系统架构，每层可以独立实现特定的功能。客户端通常只与最外层的资源进行交互，不需要了解后端的细节。

常见问题

8. 介绍一下你这个项目：

a. 自己组织语言，下面的信息根据你自己的掌握程度进行替换，也可以说是独立开发了一个网站系统/软件/和朋友开发了一个网站，他前端你后端，也可以直接说做了一个C++web服务器，避免和其他同学重复，但最终落脚是服务器。把小班加分的方向加进去用来体现差异性

b. 核心点：这是你**自学**了一些C++学习资料和网络编程的书后，后端部分独立开发的一个C++分布式 web高性能服务器，服务器的运行环境是ubuntu，采用事件驱动模型和线程池来提高性能，同时用了很多C++新特性提高代码的安全性和简洁性。并且通过Nginx实现了分布式，在数据库方面前期用的SQLite，预防了SQL注入，后期换成了Mongo DB来作为大数据数据库，支持更灵活更多的数据文件类型，实现文件的上传和下载，并用SSL提高了服务器的安全性

9. 你这个服务器性能如何

a. 用ApacheBench进行压力测试，一分钟内可以完成100个连接发送的一共100W次请求的处理，

- 50% 的请求在 `5 ms` 内完成。
- 90% 的请求在 `8 ms` 内完成。
- 99% 的请求在 `21 ms` 内完成。
- 最长请求耗时 `79 ms` 。

对于千万级别就有可能引起服务器崩溃，因为服务器运行在本机docker上，整体还是比较轻量级

具体图片里的数据含义可以参照最底下的压力测试部分的结果分析

10. 服务器部署在哪里，为什么使用docker

a. 部署在电脑本机的docker里，为了方便环境的迁移，我在网上查了一下docker官网的教程，觉得比虚拟机方便一些，因为我的本意是想学习服务器的完整开发，不想花太多时间配置环境，并且docker在window和苹果系统都能运行，选择C++是因为C++更底层一些

相较于传统的虚拟机，Docker有以下优势：

a. 轻量级：每个容器共享主机的操作系统内核，因此相比运行完整的操作系统，容器启动更快，占用资源更少。

b. 高效性：通过层叠的文件系统和高效的资源隔离机制，Docker能够实现高密度的容器部署，显著提高硬件利用率。

c. 便携性：由于容器包含了应用程序的所有依赖，它们可以在任何支持Docker的平台上轻松迁移和运行，确保了一致性和标准化。

d. 隔离性：容器之间在文件系统、网络配置以及进程空间上相互隔离，提供了安全可靠的多租户环境。

11. 为什么想到做这个项目

a. 之前项目基础不多，考研主要是学习了很多理论知识，但我自己写代码经验太少了，只写过一点简单的算法题，想在读研前提高自己的技术开发能力/将考研学的知识实践一下/为未来从事互联网工作打基础，然后问了学长/查了资料/听别人说/老师建议 可以做一个后端相关的项目，觉得自己独立开发一个服务器可能可以搞懂网络通信的流程，前后端的逻辑，多线程，所以决定做了这么一个项目（可以加一些别的）

b. **核心是表达（主动学习的意愿，自学的能力，顺便提两句我们服务器项目的优势）**

12. 你怎么学习的这个项目

a. 语法方面是看了网课/教程书籍，服务器开发是查询了一些Unix/Linux/C++服务器网络编程的书和文章，大致理解了服务器的运行原理，但还是想自己动手做，于是照着网上的博客/书籍/自学 写了一个简单的服务器，然后一步步添加功能。过程中学习了一些C++新特性，多线程开发，事件驱动模型和分布式的知识

b. **核心是表达自己不是跟着一个教程抄下来的，而是搜集了很多方方面面的东西**

13. 开发过程中遇到了哪些困难，怎么解决？

a. 一种思路：我发现XX，想做成/优化XX，不知道怎么做，查了博客/书籍/资料后，进行了优化

b. 比如，发现单个服务器崩溃后整个瘫痪，想有没有办法同时运行多个服务器，于是查了资料，发现可以用分布式来做，我选择了Nginx作为反向代理，（介绍代理的逻辑），然后实现了这个功能（过程中语法问题问大模型/查资料/博客等等）

c. 另一种思路：代码能力很差，经常网上的代码不能看懂，就一个个语法去查，至少知道一些网络编程常用的函数或者C++系统库函数的用法/看别人写的博客（举个C++新特性的例子）

d. **核心是一定要举出具体的例子，越具体的例子越说明你掌握的扎实，**千万不要泛泛而谈

14. 你这个服务器出现bug怎么办

a. 我学习了简单的GDB调试，但主要开发还是靠日志系统来定位问题，更多时候是新特性语法不熟悉，或者项目运行逻辑有问题（不会GDB调试就说没怎么用也可以，主要靠日志系统）

15. 这个Web服务器项目背后的业务需求是什么？你是如何根据这些需求来设计服务器的架构的？

a. 需求是想做一个高性能的安全的服务器，能够比较快的响应，所以最开始实现简单的单线程阻塞I/O后，又学习了事件驱动，改成了事件驱动模型（这里需要你理解事件驱动模型和其他模型比较起来优势是什么，可以现场讲出来），然后又加入了线程池

b. 类似的话术可以换成SQL放注入提高安全性，分布式提高稳定性，mongodb体现大数据，未来可以应对高负载和灵活的数据类型

16. 你为什么选择XX（比如nginx mongodb 事件驱动 SQLite)

a. 这个XX一定是你简历或者你介绍的时候说了，所以你说的内容一定是你已经掌握了它的基本理论知识，这个时候就说它的优点和项目里怎么实现即可

17. 在项目的架构设计中，有没有考虑到未来的扩展性和灵活性？具体是如何做到的？

a. 考虑了，最开始我都写在同一个文件里，然后后来加了一些功能后发现很繁杂，于是重新整理了，将不同的部分分到不同文件里，降低了代码耦合度，每个文件负责一个模块。日志，response，request，线程池的库函数，这样很便于长远开放

18. 项目总的代码量

a. 前后端加起来有一二三四千行吧（不要编到几万行，也不要就说几百行，实际上我们项目大概一千多行，但那是因为我代码风格比较精简，同时前端文件没有算进去，你个人的项目，老师看你跨考或者经验不多的情况下初次写一两千行已经很厉害了，太多老师也不太会信）

19. 未来可以优化的地方（不用全说）

a. 进一步提高性能，线程池引入线程优先级

b. 数据库安全进一步优化，引入更多防止SQL注入的东西

c. 分布式管理策略进一步优化，实现负载均衡

d. 数据库/http连接采用类似线程池的池管理

e. 进一步优化文件处理，提高对于大文件的上传和下载功能

20. 遇到困难怎么解决

a. 一些语法或者bug报错内容问大模型/查询博客，stackflow（国外的计算机论坛，类似国内的CSDN）

b. 概念或者编程不会，在网上找相关理论知识，比如事件驱动模型的理论知识就说查询了一些博客/看了Linux网络编程的书

c. 求助学长学姐/小伙伴

d. 自己写日志debug

21. 做完这个项目后，那你现在对哪个方向感兴趣

a. 说自己掌握的好的地方，然后背那段的原理，比如对高性能感兴趣，说一下自己对事件驱动的理解，未来进一步优化；或者安全感兴趣，就说日志系统，SQL注入，预编译等等

一些我们确实没做的工作，老师问了就说目前还没做就行，肯定不可能面面俱到，但你说的时候就说还没来得及，等我未来继续开发再把这部分加进去就行，不要不懂装懂说自己啥都做了，因为我们做的东西已经很多了，不差这点功能。

1. 你是如何保证代码的质量的？是否引入了代码审查、单元测试或静态代码分析等实践？
2. 是否用大数据做了备份数据库

面试tips

1. 自信，对项目做了和没做的东西清楚的回答，不要犹豫让人觉得你不知道自己做了啥，这个前提是你已经想好了哪些功能是你想说的并且提前准备了，哪些没掌握就别写，问了就是掌握的不太好，学了一些没学懂所以没实现
2. 不要不懂装懂，不会的没做的部分就说不会，未来可以学，不用担心老师觉得你会的太少，你本来就不可能会老师问的所有东西，而且我们项目的东西也足够多了，只要在面试里尽可能多展示自己会的就足够了。
3. 这个项目是你自学的，不是报班有人带你一步步做的，体现自己学习的能力
4. 没听懂老师的问题就表达自己的理解，问一下对方问的是不是xx，不要答非所问
5. 引导老师到自己想回答的地方，比如老师问你简历其他东西，为什么报计算机之类的，回答完加一个“然后我最近做了一个C++服务器的项目， 想提高自己的动手能力”，或者问到遇到什么困难那里时也顺便秀一下自己某个地方怎么实现的和相关理论知识
    a. 老师哪怕没问也没关系，有可能面试的位置老师急着吃饭，但他能看到你简历上这个项目，一定会综合考虑的
6. 抽的题不会就不要浪费太多时间，直接说不会问一下能不能换一个，或者老师给点提示，自己再去说
7. 不要一板一眼太简短的回答问题，尽量在回答里展现尽可能多的自己对项目的理解和相关知识，
    a. 介绍项目相关往往都是宏观的理论知识+我们项目中具体怎么实践
8. 理解大于记忆：脑子里要真的理解比如事件驱动为什么效率高，不要让老师觉得你是为了面试死记硬背

这段时间要能看着左边的目录去自己口述一遍回答问题，千万不要看两眼就觉得自己会了。

一定是能讲下来才可以

代码部分，不用力求每行代码都完全理解，主要是能讲清楚重点功能的代码逻辑，相当于口述注释或者介绍函数逻辑。