1、使用 线程池 + 非阻塞 socket + epoll(ET 和 LT 均实现) + 事件处理(Reactor 和模拟 Proactor 均实现) 的并发模型；

该项目使用**线程池**（半同步半反应堆模式）并发处理用户请求，主线程负责读写，工作线程（线程池中的线程）负责处理逻辑（HTTP 请求报文的解析等等）。

非阻塞客户端实现的基本流程：
非阻塞 socket 在 connect 后会立即返回，连接成功返回零，连接正在进行（没有立即完成）或连接失败都返回-1 并把错误代码记录到 errno 中，通过 errno 的返回代码判断 connect 函数是否处于正在连接的状态。如果 errno 错误码为 EINPROCESS，说明连接正在进行，可以将 socket 的 fd 加入到 select 的可写集合中去，设置 select 的超时时间，当 fd 变为可写时调用 getsockopt 函数判断 fd 是否只可写（errno==0)。

2、使用状态机解析 HTTP 请求报文，支持解析 GET 和 POST 请求；
3、利用 RAII 机制实现了数据库连接池，减少数据库连接建立与关闭的开销，同时实现了用户注册登录功能。

1. 为什么要使用**数据库连接池**？
- 由于服务器需要频繁地访问数据库，即需要频繁创建和断开数据库连接，该过程是一个很耗时的操作，也会对数据库造成安全隐患。
- 在程序初始化的时候，集中创建并管理多个数据库连接，可以保证较快的数据库读写速度，更加安全可靠。
2. 什么是数据库连接池？
- 池是一组资源的集合，这组资源在服务器启动之初就被完全创建好并初始化。
- 当服务器开始处理客户请求的时候，如果它需要相关的资源，可以直接从池中获取，无需动态分配。
- 当服务器处理完一个客户连接后，可以把相关的资源放回池中，无需执行系统调用释放资源。
3、**RAII 机制**（Resource Acquisition Is Initalization）
- C++的语言机制保证，当一个对象创建时会自动调用构造函数，当对象超出作用域时会自动调用析构函数。
- 所以，我们可以使用类来管理资源，在构造函数中申请分配资源，在析构函数中释放资源。
- RAII 的核心思想是将资源与对象的生命周期绑定。

4、实现同步/异步日志系统，记录服务器运行状态；使用基于跳表的定时器处理非活动链接。

日志：由服务器自动创建，并记录运行状态，错误信息，访问数据的文件。
o 同步：直接格式化输出内容，将信息写入日志文件
o 异步：（单例模式）将内容写入阻塞队列，创建一个写线程，从阻塞队列取出内容写入日志文件
定时器：如果某一用户 connect()到服务器之后，长时间不交换数据，一直占用服务器端的文件描述符，导致连接资源的浪费。这时候就应该利用定时器把这些超时的非活动连接释放掉，关闭其占用的文件描述符。
项目中使用的是 SIGALRM 信号来实现定时器，利用 alarm 函数周期性的触发 SIGALRM 信号，信号处理函数利用管道通知主循环，主循环接收到该信号后对升序链表上所有定时器进行处理，若该段时间内没有交换数据，则将该连接关闭，释放所占用的资源。

5、压力测试
listenfd 和 connfd 分别采用 ET 和 LT 模式，均可实现上万的并发连接
基本原理：Webbench 首先 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 -c 10500 -t 5 http://192.168.110.129:10000/index.html
并发连接总数：10500
访问服务器时间：5s
所有访问均成功
Proactor， ET+ LT，97459 QPS 每秒查询率
Reactor，ET+ LT，69175 QPS

2、介绍下你的项目

服务器基本框架：

主要由 I/O 单元，逻辑单元和网络存储单元组成，其中每个单元之间通过请求队列进行通信，从而协同完成任务。其中 I/O 单元用于处理客户端连接，读写网络数据；逻辑单元用于处理业务逻辑的线程；网络存储单元指本地数据库和文件等。

此项目是基于 Linux 的轻量级多线程 Web 服务器，应用层实现了一个简单的 HTTP 服务器，利用多路 IO 复用，可以同时监听多个请求，使用线程池处理请求，使用模拟 proactor 模式，主线程负责监听，监听有事件之后，从 socket 中循环读取数据，然后将读取到的数据封装成一个请求对象放入队列。睡眠在请求队列上的工作线程被唤醒进行处理，使用状态机解析 HTTP 请求报文，实现 同步/异步 日志系统，记录服务器运行状态，并对系统进行了压力测试。

3、你的项目的技术难点是什么？

1、如何提高服务器的并发能力

2、由于涉及到 I/O 操作，当单条日志比较大的时候，同步模式会阻塞整个处理流程

3、多线程并发的情况下，保证线程的同步

4、你是如何克服这个技术难点的？

5、你做这个项目的收获是什么？

6、为什么使用这个技术 / 组件？

7、如何解决项目中的 BUG

（1）运行代码，发现错误，找到报错的位置。

（2）如果注释后运行正常了，那么就是注释掉的部分有误了

（3）若不是，则从 main 函数里边调用的开始下手，查看定义，跳转到定义的功能，这很大程度上能让我们快速的找到 bug。

9、项目的异常处理有哪些

（1）登录异常

* 用户不存在：根据账号在数据库查，如果查不到就是
* 用户名或密码错误：数据库查询的用户密码和 http 请求用户输入的密码比对，如果不一致
* 登录成功

try-catch 语句

程序先执行 try 中的代码

如果 try 中的代码出现异常，就会结束 try 中的代码，看和 catch 中的异常类型是否匹配.

如果找到匹配的异常类型，就会执行 catch 中的代码

如果没有找到匹配的异常类型，就会将异常向上传递到上层调用者.

无论是否找到匹配的异常类型，finally 中的代码都会被执行到(在该方法结束之前执行).

如果上层调用者也没有处理的了异常，就继续向上传递

一直到 main 方法也没有合适的代码处理异常，就会交给 JVM 来进行处理，此时程序就会异常终止

线程池相关

1、为什么使用线程池

每个请求对应一个线程方法的不足之一是：为每个请求创建一个新线程的开销很大；为每个请求创建新线程的服务器在创建和销毁线程上花费的时间和消耗的系统资源要比花在处理实际的用户请求的时间和资源更多。

线程池是为了避免创建和销毁线程所产生的开销，避免活动的线程消耗的系统资源；

提高响应速度，任务到达时，无需等待线程即可立即执行；

提高线程的可管理性：线程的不合理分布导致资源调度失衡，降低系统的稳定性。使用线程池可以进行统一的分配、

调优和监控。

2、怎么创建线程池（线程池运行逻辑）
该项目使用线程池（半同步半反应堆模式）并发处理用户请求，主线程负责读写，工作线程（线程池中的线程）负责处理逻辑（HTTP 请求报文的解析等）。
具体的：主线程为异步线程，负责监听文件描述符，接收 socket 新连接，若当前监听的 socket 发生了读写事件，然后将任务插入到请求队列。工作线程从请求队列中取出任务，完成读写数据的处理。

线程池是空间换时间,浪费服务器的硬件资源,换取运行效率.
当服务器进入正式运行阶段,开始处理客户请求的时候,如果它需要相关的资源,可以直接从池中获取,无需动态分配
当服务器处理完一个客户连接后,可以把相关的资源放回池中,无需执行系统调用释放资源.

半同步/半反应堆工作流程（以 Proactor 模式为例）
主线程充当异步线程，负责监听所有 socket 上的事件
若有新请求到来，主线程接收之以得到新的连接 socket，然后往 epoll 内核事件表中注册该 socket 上的读写事件
如果连接 socket 上有读写事件发生，主线程从 socket 上接收数据，并将数据封装成请求对象插入到请求队列中
所有工作线程睡眠在请求队列上，当有任务到来时，通过竞争（如互斥锁）获得任务的接管权

线程池的实现还需要依靠 锁机制 以及 信号量 机制来实现线程同步，保证操作的原子性。

3、线程的同步机制有哪些？
（1）同步 I/O

同步 I/O 指内核向应用程序通知的是就绪事件，比如只通知有客户端连接，要求用户代码自行执行 I/O 操作

a）阻塞 IO: 调用者调用了某个函数，等待这个函数返回，期间什么也不做，不停的去检查这个函数有没有返回，必须等这个函数返回才能进行下一步动作

b）非阻塞 IO: 非阻塞等待，每隔一段时间就去检测 IO 事件是否就绪。没有就绪就可以做其他事。非阻塞 I/O 执行系统调用总是立即返回，不管事件是否已经发生，若事件没有发生，则返回-1，此时可以根据 errno 区分这两种情况，对于 accept，recv 和 send，事件未发生时，errno 通常被设置成 eagain

c）信号驱动 IO: linux 用套接口进行信号驱动 IO，安装一个信号处理函数，进程继续运行并不阻塞，当 IO 时间就绪，进程收到 SIGIO 信号。然后处理 IO 事件。

d)IO 复用: linux 用 select/poll 函数实现 IO 复用模型，这两个函数也会使进程阻塞，但是和阻塞 IO 所不同的是这两个函数可以同时阻塞多个 IO 操作。而且可以同时对多个读操作、写操作的 IO 函数进行检测。直到有数据可读或可写时，才真正调用 IO 操作函数
多路 IO 复用是一种同步 IO 模型，，它可以实现一个线程可以同时监视多个文件描述符，一旦有某个文件描述符准备就绪，就会通知应用程序，对该文件描述符进行操作。在监视的各个文件描述符没有准备就绪的时候，应用程序线程就会阻塞，交出 cpu，让 cpu 去执行其他的任务，以此提高 cpu 的利用率。

（2）异步 I/O
异步 I/O 是指内核向应用程序通知的是完成事件，比如读取客户端的数据后才通知应用程序，由内核完成 I/O 操作
linux 中，可以调用 aio_read 函数告诉内核描述字缓冲区指针和缓冲区的大小、文件偏移及通知的方式，然后立即返回，当内核将数据拷贝到缓冲区后，再通知应用程序。

4、线程池中的工作线程是一直等待吗？

在 run 函数中，我们为了能够处理高并发的问题，将线程池中的工作线程都设置为阻塞等待在请求队列是否不为空的条件上，因此项目中线程池中的工作线程是处于一直阻塞等待的模式下的。

5、你的线程池工作线程处理完一个任务后的状态是什么？

（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。

6、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？

本项目是通过对子线程循环调用来解决高并发的问题的。

首先在创建线程的同时就调用了 pthread_detach 将线程进行分离，不用单独对工作线程进行回收，资源自动回收。我们通过子线程的 run 调用函数进行 while 循环，让每一个线程池中的线程永远都不会停止，访问请求被封装到请求队列(list)中，如果没有任务线程就一直阻塞等待，有任务线程就抢占式进行处理，直到请求队列为空，表示任务全部处理完成。

7、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?

会，因为线程池内线程的数量时有限的，如果客户请求占用线程时间过久的话会影响到处理请求的效率，当请求处理过慢时会造成后续接受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:

我们可以为线程处理请求对象设置处理超时时间，超过时间先发送信号告知线程处理超时，然后设定一个时间间隔再次检测，若此时这个请求还占用线程则直接将其断开连接。

9、介绍一下几种典型的锁?

线程池的实现还需要依靠锁机制以及信号量机制来实现线程同步，保证操作的原子性

（1）读写锁

（2）互斥锁

（3）条件变量

互斥锁一个明显的缺点是他只有两种状态：锁定和非锁定。而条件变量通过允许线程阻塞和等待另一个线程发送信号的方法弥补了互斥锁的不足，他常和互斥锁一起使用，以免出现竞态条件。当条件不满足时，线程往往解开相应的互斥锁并阻塞线程然后等待条件发生变化。一旦其他的某个线程改变了条件变量，他将通知相应的条件变量唤醒一个或多个正被此条件变量阻塞的线程。总的来说互斥锁是线程间互斥的机制，条件变量则是同步机制。

（4）自旋锁

10、如何销毁线程

1、通过判断标志位，主动退出

2、通过 Thread 类中成员方法 interrupt()，主动退出

3、通过 Thread 类中成员方法 stop()，强行退出

11、detach 和 join 有什么区别

（1）当调用 join()，主线程等待子线程执行完之后，主线程才可以继续执行，此时主线程会释放掉执行完后的子线程资源。主线程等待子线程执行完，可能会造成性能损失。

（2）当调用 detach()，主线程与子线程分离，他们成为了两个独立的线程遵循 cpu 的时间片调度分配策略。子线程执行完成后会自己释放掉资源。分离后的线程，主线程将对它没有控制权。

当你确定程序没有使用共享变量或引用之类的话，可以使用 detch 函数，分离线程。

14、socket 通信的基本流程（客户端和服务端的通信协议）

简单描述一下 Socket 的通信流程:
（1）服务端这边首先创建一个 Socket（Socket()），然后绑定 IP 地址和端口号（Bind()），之后注册监听（Listen()），这样服务端就可以监听指定的 Socket 地址了；
（2）客户端这边也创建一个 Socket（Socket()）并打开，然后根据服务器 IP 地址和端口号向服务器 Socket 发送连接请求（Connect()）；
（3） 服务器 Socket 监听到客户端 Socket 发来的连接请求之后，被动打开，并调用 Accept()函数接收请求，这样客户端和服务器之间的连接就建立好了；
（4）成功建立连接之后，客户端和服务器就可以进行数据交互（Receive()、Send()）；
（5）在数据交互完之后，各自关闭连接（Close()），交互结束

15、listen 函数第二个参数 backlog 参数作用
int listen(int sockfd, int backlog);
backlog 是 accept 阻塞队列的长度，即等待 accept 的 socket 的最大数量。

16、listen 底层用的是什么队列
a.半连接队列（Incomplete connection queue），又称 SYN 队列。……
b.全连接队列（Completed connection queue），又称 Accept 队列。……

17、send 函数在发送的数据长度大于发送缓冲区大小，或者大于发送缓冲区剩余大小时，socket 会怎么反应
不管是 windows 还是 linux，阻塞还是非阻塞，send 都会分帧发送，分帧到缓冲区能够接收的大小

18、多线程中线程越多越好吗
不是
（1）假设现有 8 个 CPU、8 个线程，每个线程占用一个 CPU，同一时间段内，若 8 个线程都运行往前跑，相比较 5/6/7 个线程，8 个线程的效率高。
（2）但若此时有 9 个线程，只有 8 个 CPU，9 个线程同时运行，则此时牵扯到线程切换，而线程切换是需要消耗时间的。
（3）所以随着线程数越多，效率越来越高，但到一个峰值，再增加线程数量时，就会出现问题。线程太多要来回的切换，最终可能线程切换所用时间比执行时间业务所用时间还大。
（4） 随着线程数越多,由于线程执行的时序的问题，程序可能会崩溃或产生二义性。

并发模型相关
8、epoll 如何判断数据已经读取完成
epoll ET(Edge Trigger)模式，才需要关注数据是否读取完毕了。使用 select 或者 epoll 的 LT 模式，不用关注，select/epoll 检测到有数据可读去读就 OK 了。
两种做法：
1、针对 TCP，调用 recv 方法，根据 recv 的返回值。如果返回值小于我们设定的 recv buff 的大小，那么就认为接收完毕。
2、TCP、UDP 都适用，将 socket 设为 NOBLOCK 状态（使用 fcntl 函数），然后 select 该 socket 可读的时候，使用 read/recv 函数读取数据。当返回值为 -1，并且 errno 是 EAGAIN 或 EWOULDBLOCK 的时候，表示数据读取完毕。

10、epoll_wait 函数
#include <sys/epoll.h>
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout)
该函数用于等待所监控文件描述符上有事件的产生，返回就绪的文件描述符个数
events：用来存内核得到事件的集合，

maxevents：告之内核这个 events 有多大，这个 maxevents 的值不能大于创建 epoll_create()时的 size，

timeout：是超时时间
-1：阻塞
0：立即返回，非阻塞
>0：指定毫秒
返回值：成功返回有多少文件描述符就绪，时间到时返回 0，出错返回-1

HTTP 报文解析相关

1、用了状态机啊，为什么要用状态机？

在逻辑处理模块中，响应 HTTP 请求采用主从状态机来完成
传统的控制流程都是按照顺序执行的，状态机能处理任意顺序的事件，并能提供有意义的响应—即使这些事件发生的顺序和预计的不同。
项目中使用主从状态机的模式进行解析，从状态机（parse_line）负责读取报文的一行，主状态机负责对该行数据进行解析，主状态机内部调用从状态机，从状态机驱动主状态机。每解析一部分都会将整个请求的 m_check_state 状态改变，状态机也就是根据这个状态来进行不同部分的解析跳转的

2、状态机的转移图画一下

从状态机负责读取报文的一行，主状态机负责对该行数据进行解析，主状态机内部调用从状态机，从状态机驱动主状态机。
主状态机
三种状态，标识解析位置。
CHECK_STATE_REQUESTLINE，解析请求行
CHECK_STATE_HEADER，解析请求头
CHECK_STATE_CONTENT，解析消息体，仅用于解析 POST 请求
从状态机
三种状态，标识解析一行的读取状态。
LINE_OK，完整读取一行，该条件涉及解析请求行和请求头部
LINE_BAD，报文语法有误
LINE_OPEN，读取的行不完整
处理结果：
NO_REQUEST请求不完整，需要继续读取请求报文数据
GET_REQUEST　　获得了完整的 HTTP 请求
BAD_REQUEST　　HTTP 请求报文有语法错误
INTERNAL_ERROR 服务器内部错误

12、一次完整 HTTP 请求所经历的步骤

（当我们在 web 浏览器的地址栏中输入：www.baidu.com，然后回车，到底发生了什么？）
由域名→ IP 地址 寻找 IP 地址的过程依次经过了浏览器缓存、系统缓存、hosts 文件、路由器缓存、 递归搜索根域名服务器（DNS 解析）。
建立 TCP/IP 连接（三次握手具体过程）。
由浏览器发送一个 HTTP 请求。
经过路由器的转发，通过服务器的防火墙，该 HTTP 请求到达了服务器。
服务器处理该 HTTP 请求，返回一个 HTML 文件。
浏览器解析该 HTML 文件，并且显示在浏览器端。
服务器关闭 TCP 连接（四次挥手具体过程）。

14、HTTP 报文处理流程

· 浏览器端发出 http 连接请求，主线程创建 http 对象接收请求并将所有数据读入对应 buffer，将该对象插入任务队列，工作线程从任务队列中取出一个任务进行处理。
· 工作线程取出任务后，调用 process_read 函数，通过主、从状态机对请求报文进行解析。

· 解析完之后，跳转 do_request 函数生成响应报文，通过 process_write 写入 buffer，返回给浏览器端。

15、 Web 服务器如何接收客户端发来的 HTTP 请求报文

Web 服务器端通过 socket 监听来自用户的请求。远端的很多用户会尝试去 connect()这个 Web Server 上正在 listen 的这个端口，而监听到的这些连接会排队等待被 accept()。由于用户连接请求是随机到达的异步事件，每当监听 socket（listenfd）listen 到新的客户连接并且放入监听队列，我们都需要告诉我们的 Web 服务器有连接来了，accept 这个连接，并分配一个逻辑单元来处理这个用户请求。而且，我们在处理这个请求的同时，还需要继续监听其他客户的请求并分配其另一逻辑单元来处理。这里，服务器通过 epoll 这种 I/O 复用技术（还有 select 和 poll）来实现对监听 socket（listenfd）和连接 socket（客户请求）的同时监听。

定时器相关

1、为什么要用定时器？（优化：定时器处理非活跃连接）

为了定期删除非活跃事件，防止连接资源的浪费。

非活跃，是指浏览器与服务器端建立连接后，长时间不交换数据，一直占用服务器端的文件描述符，导致连接资源的浪费。

定时事件，是指固定一段时间之后触发某段代码，由该段代码处理一个事件，如从内核事件表删除事件，并关闭文件描述符，释放连接资源。

2、说一下定时器的工作原理

定时器利用结构体将定时事件进行封装起来。定时事件，即定期检测非活跃连接。

服务器主循环为每一个连接创建一个定时器，并对每个连接进行定时。另外，利用升序双向链表将所有定时器串联起来，利用 alarm 函数周期性地触发 SIGALRM 信号，信号处理函数利用管道通知主循环，主循环接收到该信号后对升序链表上所有定时器进行处理，若该段时间内没有交换数据，则将该连接关闭，释放所占用的资源。

（信号处理函数仅仅发送信号通知程序主循环，将信号对应的处理逻辑放在程序主循环中，由主循环执行信号对应的逻辑代码。）

信号通知的逻辑：创建管道，其中管道写端写入信号值，管道读端通过 I/O 复用系统监测读事件

3、定时任务处理函数的逻辑

使用统一事件源, SIGALRM 信号每次被触发, 主循环中调用一次定时任务处理函数, 处理链表容器中到期的定时器。

（1）链表容器是升序排列，当前时间小于定时器的超时时间，后面的定时器也没有到期

（2）当前定时器到期，则调用回调函数，执行定时事件

（3）将处理后的定时器从链表容器中删除，并重置头结点

若有数据传输，则将定时器往后延迟 3 个单位

（回调函数用来方便删除定时器时将对应的 HTTP 连接关闭）

日志相关

1、说下你的日志系统的运行机制？

使用单例模式创建日志系统，对服务器运行状态、错误信息和访问数据进行记录，该系统可以实现按天分类，超行分类功能，可以根据实际情况分别使用同步和异步写入两种方式。

其中异步写入方式, 将生产者-消费者模型封装为阻塞队列, 创建一个写线程, 工作线程将要写的内容 push 进队列，写线程从队列中取出内容，写入日志文件。

2、为什么要异步？和同步的区别是什么？

同步方式写入日志时会产生比较多的系统调用，若是某条日志信息过大，会阻塞日志系统，造成系统瓶颈。异步方式采用生产者-消费者模型，具有较高的并发能力。

生产者-消费者模型，并发编程中的经典模型。

以多线程为例，为了实现线程间数据同步，生产者线程与消费者线程共享一个缓冲区，其中生产者线程往缓冲区中 push 消息，消费者线程从缓冲区中 pop 消息。

阻塞队列，将生产者-消费者模型进行封装，使用循环数组实现队列，作为两者共享的缓冲区。

异步日志，将所写的日志内容先存入阻塞队列，写线程从阻塞队列中取出内容，写入日志。可以提高系统的并发性能。

同步日志，日志写入函数与工作线程串行执行，由于涉及到 I/O 操作，当单条日志比较大的时候，同步模式会阻塞整个处理流程，服务器所能处理的并发能力将有所下降，尤其是在峰值的时候，写日志可能成为系统的瓶颈。

写入方式通过初始化时是否设置队列大小（表示在队列中可以放几条数据）来判断，若队列大小为 0，则为同步，否则为异步。
若异步,则将日志信息加入阻塞队列,同步则加锁向文件中写

4、关于该项目用到的设计模式

（1）单例模式：单例对象的类只能允许一个实例存在，并提供一个访问它的全局访问点，该实例被所有程序模块共享。主要解决一个全局使用的类频繁的创建和销毁的问题，是一种创建型模式，提供了一种创建对象的最佳方式。

（2）单例模式三要素：

1）单例类只能有一个实例。

2）单例类必须自己创建自己的唯一实例。

3）单例类必须给所有其他对象提供这一实例。

（3）单例设计模式的优缺点

优点：

1 ）单例模式可以保证内存里 只有一个实例 ， 减少了内存的开销 。

2 ）可以避免对资源的 多重占用 。 （比如写文件操作）

3 ）单例模式设置全局访问点，可以优化和共享资源的访问。

缺点：

1 ）单例模式一般没有接口， 不能继承， 扩展困难 。 如果要扩展，则除了修改原来的代码，没有第二种途径，违背开闭原则。

2 ）在并发测试中，单例模式 不利于代码调试 。 在调试过程中，如果单例中的代码没有执行完，也不能模拟生成一个新的对象。

3 ）单例模式的功能代码通常写在一个类中， 如果功能设计不合理，则很容易违背单一职责原则。

（4）C++ 单例设计模式的实现 两步骤

1 ） 私有化 构造函数 ，这样别处的代码就无法通过调用该类的构造函数来实例化该类的对象，只 有通过该类提供的静态方法来得到 该类的唯一实例 ；

2 ） 通过局部静态变量，利用其只初始化一次的特点，返回静态对象成员。

（5） 单例设计模式的种类

1 ）懒汉式：获取该类的对象 时 才创建该类的实例

2 ）饿汉式：获取该类的对象之 前 已经创建好该类的实例

（6）手撕单例模式

（懒汉模式）

```
class single{
private:
    single(){}    // 私有化构造函数
    ~single(){}
public:
    // 公有静态方法获取实例
    static single* getinstance();
};
single* single::getinstance(){
    static single obj;
    return &obj;
}
```

使用函数内的局部静态对象无需加锁和解锁，因为 C++11 后编译器可以保证内部静态变量的线程安全性

（懒汉模式）加锁版本

```
class single{
private:
	static pthread_mutex_t lock;
	single(){
		pthread_mutex_init(&lock, NULL);
	}
	~single(){}

public:
	static single* getinstance();

};
pthread_mutex_t single::lock;
single* single::getinstance(){
	pthread_mutex_lock(&lock);
	static single obj;
	pthread_mutex_unlock(&lock);
	return &obj;
}
```

（饿汉模式）

```
class single{
private:
	static single* p;
	single(){}
	~single(){}
public:
	static single* getinstance();
};

single* single::p = new single();
single* single::getinstance(){
	return p;
}
```

饿汉模式不需要用锁，就可以实现线程安全。原因在于，在程序运行时就定义了对象，并对其初始化。之后，不管哪个线程调用成员函数 getinstance()，都只是返回一个对象的指针

5、现在你要监控一台服务器的状态，输出监控日志，请问如何将该日志分发到不同的机器上？（消息队列）

数据库登录注册相关

1、 什么是数据库连接池，为什么要创建连接池？

（1）池是资源的容器，这组资源在服务器启动之初就被完全创建好并初始化，本质上是对资源的复用。

当系统开始处理客户请求的时候，如果它需要相关的资源，可以直接从池中获取，无需动态分配；当服务器处理完一个客户连接后,可以把相关的资源放回池中，无需执行系统调用释放资源。

（2）若系统需要频繁访问数据库，则需要频繁创建和断开数据库连接，而创建数据库连接是一个很耗时的操作，也容易对数据库造成安全隐患。

在程序初始化的时候，集中创建多个数据库连接，并把他们集中管理，供程序使用，可以保证较快的数据库读写速

度，更加安全可靠。
（3）使用单例模式和链表创建数据库连接池，实现对数据库连接资源的复用。
连接池的功能主要有：初始化，获取连接、释放连接，销毁连接池
连接池中的多线程使用信号量进行通信，使用互斥锁进行同步。

数据库连接的获取与释放通过 RAII 机制封装，避免手动释放。
RAII 机制
RAII 全称是“Resource Acquisition is Initialization”，直译过来是“资源获取即初始化”.
RAII 的核心思想是将资源或者状态与对象的生命周期绑定，通过 C++的语言机制，实现资源和状态的安全管理，智能指针是 RAII 最好的例子
具体来说：构造函数的时候初始化获取资源，析构函数释放资源

2、 获取释放连接、销毁对象池
获取链接
容器有空闲连接，直接用
容器无空闲
未达上限，自己创建
达上限，报错打回等待
释放连接
放回容器
目前暂无较好的销毁连接策略
销毁对象池
关闭销毁池中连接
释放连接池对象
完成释放

3、登录说一下？（登录注册是 POST 请求）
将数据库中的用户名和密码载入到服务器的 map 中来，map 中的 key 为用户名，value 为密码
服务器端解析浏览器的请求报文，当解析为 POST 请求时，提取出请求报文的消息体的用户名和密码。
POST 请求中最后是用户名和密码，用&隔开。分隔符&，前是用户名，后是密码。
登录：将浏览器输入的用户名和密码在数据库中查找，直接判断。
注册：往数据库中插入数据，需要判断是否有重复的用户名。
最后进行页面跳转
通过 m_url 定位/所在位置，根据/后的第一个字符，使用分支语句实现页面跳转。具体的，
0 — 跳转注册页面，GET
1 — 跳转登录页面，GET
5 — 显示图片页面，POST
6 — 显示视频页面，POST
7 — 显示关注页面，POST

4、登录与注册，服务器如何校验
CGI 校验（通用网关接口），它是一个运行在 Web 服务器上的程序，在编译的时候将相应的.cpp 文件编程成.cgi 文件并在主程序中调用即可。这些 CGI 程序通常通过客户在其浏览器上点击一个 button 时运行。这些程序通常用来执行一些信息搜索、存储等任务，而且通常会生成一个动态的 HTML 网页来响应客户的 HTTP 请求。

CGI 程序，将用户请求中的用户名和密码保存在一个 id_passwd.txt 文件中，通过将数据库中的用户名和密码存到一个 map 中用于校验。在主程序中通过 execl(m_real_file, &flag, name, password, NULL);这句命令来执行这个 CGI 文件，这里 CGI 程序仅用于校验，并未直接返回给用户响应。这个 CGI 程序的运行通过多进程来实现，根据其返回结果判断校验结果（使用 pipe 进行父子进程的通信，子进程将校验结果写到 pipe 的写端，父进程在读端读取）。

5、你这个保存状态了吗？如果要保存，你会怎么做？（cookie 和 session）

可以利用 session 或者 cookie 的方式进行状态的保存。

cookie 其实就是服务器给客户分配了一串“身份标识”，比如“123456789happy”这么一串字符串。每次客户发送数据时，都在 HTTP 报文附带上这个字符串，服务器就知道你是谁了；

session 是保存在服务器端的状态，每当一个客户发送 HTTP 报文过来的时候，服务器会在自己记录的用户数据中去找，类似于核对名单；

6、登录中的用户名和密码你是 load 到本地，然后使用 map 匹配的，如果有 10 亿数据，即使 load 到本地后 hash，也是很耗时的，你要怎么优化？

这个问题的关键在于大数据量情况下的用户登录验证怎么进行？将所有的用户信息加载到内存中耗时耗利，对于大数据最遍历的方法就是进行 hash，利用 hash 建立多级索引的方式来加快用户验证。具体操作如下：

首先，将 10 亿的用户信息，利用大致缩小 1000 倍的 hash 算法进行 hash，这时就获得了 100 万的 hash 数据，每一个 hash 数据代表着一个用户信息块（一级）；

而后，再分别对这 100 万的 hash 数据再进行 hash，例如最终剩下 1000 个 hash 数据（二级）。

在这种方式下，服务器只需要保存 1000 个二级 hash 数据，当用户请求登录的时候，先对用户信息进行一次 hash，找到对应信息块（二级），在读取其对应的一级信息块，最终找到对应的用户数据

压测相关

3、测试的时候有没有遇到问题？

Bug：使用 Webbench 对服务器进行压力测试，创建 1000 个客户端，并发访问服务器 10s，正常情况下有接近 8 万个 HTTP 请求访问服务器。

结果显示仅有 7 个请求被成功处理，0 个请求处理失败，服务器也没有返回错误。此时，从浏览器端访问服务器，发现该请求也不能被处理和响应，必须将服务器重启后，浏览器端才能访问正常。

解决办法：

排查：

通过查询服务器运行日志，对服务器接收 HTTP 请求连接，HTTP 处理逻辑两部分进行排查。

日志中显示，7 个请求报文为:GET / HTTP/1.0 的 HTTP 请求被正确处理和响应，排除 HTTP 处理逻辑错误。重点放在接收 HTTP 请求连接部分。其中，服务器端接收 HTTP 请求的连接步骤为 socket -> bind -> listen -> accept

错误原因：错误使用 epoll 的 ET 模式。

ET 边缘触发模式

epoll_wait 检测到文件描述符有事件发生，则将其通知给应用程序，应用程序必须立即处理该事件。

必须要一次性将数据读取完，使用非阻塞 I/O，读取到出现 eagain。

当连接较少时，队列不会变满，即使 listenfd 设置成 ET 非阻塞，不使用 while 一次性读取完，也不会出现 Bug。

若此时 1000 个客户端同时对服务器发起连接请求，连接过多会造成 established 状态的连接队列变满。但 accept 并没有使用 while 一次性读取完，只读取一个。因此，连接过多导致 TCP 就绪队列中剩下的连接都得不到处理，同时新的连接也不会到来。

解决方案

将 listenfd 设置成 LT 阻塞，或者 ET 非阻塞模式下 while 包裹 accept 即可解决问题。