## 为什么要做这样一个项目

说实话，这个项目没有非常大的亮点，主要是为了巩固基础，动手实践之前学习的知识，希望能系统的 串联起来。

这个项目虽然代码量不算太多，其中也用到了EPOLL复用，线程池，数据库连接池等技术，webbench 压力测试也可以实现上万的qps[queries per second]。

完成这个项目提高了我对网络编程，网络协议，和多线程的理解。

另外，由于是第一次完整的实现这个项目，过程中遇到了一些问题，比如错误关连接导致的问题，如何优雅关闭连接，writev的手动偏移导致的大文件传输Bug，ET没有通过while接收而导致的接受问题等等...

## 介绍下你的项目

本项目实现了一个轻量级的web服务器，提供用户注册，登录等功能。

1 .在并发模型上，采用epoll边沿触发模式(ET)来实现IO复用，使用同步IO模拟Proactor事件处理模式，实现 半同步/半反应堆线程池（主线程为异步线程，工作线程为同步线程）[图1]，利用工作队列，实现主线程和工作线程的解耦合[解除了主线程和工作线程的耦合关系]

1. 主线程为异步线：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。
2. 工作线程为同步线程：工作线程从请求队列中取出任务并处理任务，任务处理完成后，通知主线程[通过注册EPOLLOUT事件来完成]

图1

2.http解析类：使用有限状态机的设计思想，封装了http连接处理类，以实现解析报文请求和发送。[这里只有主状态机和从状态机]

主状态机在内 部调用从状态机,从状态机将处理状态和数据传给主状态机

支持http长连接[keep-alive]，支持解析GET和POST请

3.定时器：由于非活跃连接占用了连接资源，严重影响服务器的性能，实现了一个基于时间轮的定时器，

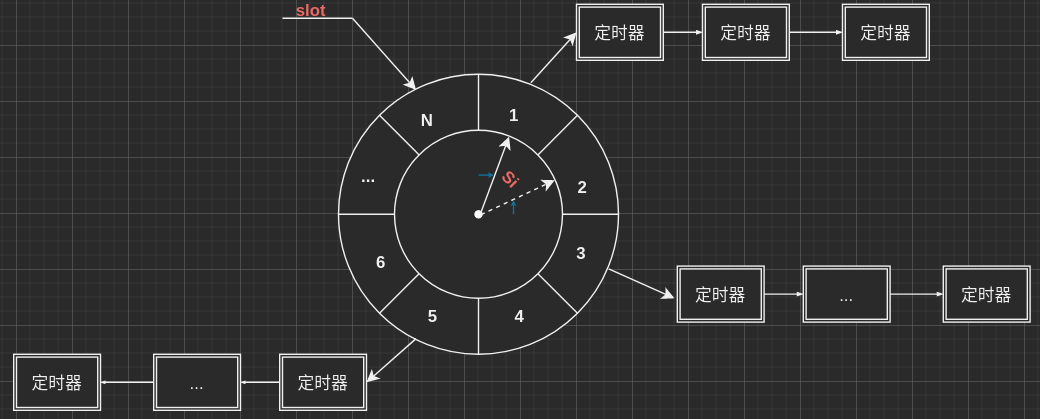
和数据一起封装，用于处理 非活跃连接，释放连接资源。

图2

4.日志系统：使用单例模式创建日志系统，记录服务器运行状态[错误信息和访问数据进行记录]等

* + 1. 日志分类方法：支持按天分类，超行分类功能，
    2. 日志写入方法：同步和异步写入两种方式。其中异步写入方式，
    3. 同步日志： 与工作线程串行执行，但由于写日记涉及到I/O操作，当日志较大时，同步模式会阻塞整个处理流程，导致服务器并发处理能力下降。[有可能成为瓶颈]
    4. 异步写入：类似生产者-消费者模式，用循环数组实现阻塞队列，工作线程将要的日志内容push到队列，写线程从队列中取出日志并写入日志文件。

5.数据库连接池：通过重复使用这些已经建立的数据库连接，解决频繁建立连接的缺点，从而提高系 统性能。

## webbench测压工具原理：

1. 父进程fork若干子进程，每个子进程在指定时间内部对目标web循环发出实际的访问请求
2. 父子进程间采用管道进行通信， 子进程在写端写入若干次请求访问完毕后的记录信息，父进程在 读端读取该信息；子进程在时间到后就全部结束，父进程在所有子进程推出后统计并把结果信息显示给用户然后退出
3. eg： webbench -c 1000 - t 30 127.0.0.1/test.jpg

webbench的标准测试可以向我们展示服务器的两项内容：

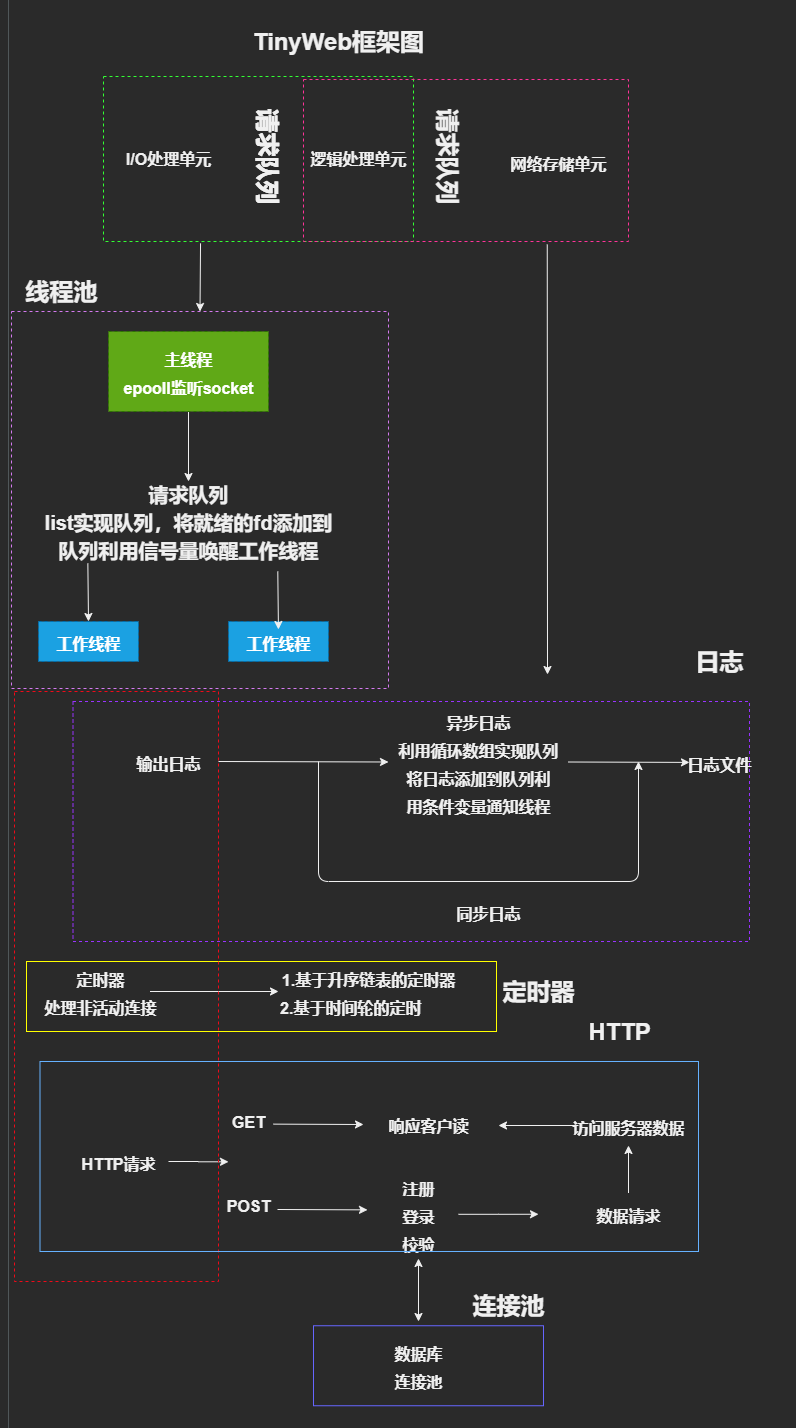
每分钟相应请求数 Speed = 380926 pages/min

每秒钟传输数据量 xxx bytes/sec

webbench是Linux下的一个网站压力测试工具

webbench最多可以模拟3万个并发连接去测试网站的负载能力

TinyWeb框架图：



## 线程的同步机制有哪些

互斥锁、信号量、自旋锁、条件变量、读写锁

**互斥锁（mutex）：**

为了确保同一时间只有一个线程访问数据，在访问共享资源前需要对互斥量上锁。一旦对互斥 量上锁后，任何其他试图再次对互斥量上锁的线程都会被阻塞，即进入等待队列，当其他线程释放互斥 量后，操作系统会激活那个被挂起的线程，让其投入运行。[加锁失败，被阻塞进入等待队列]

**自旋锁(spinclock)：**

自旋锁与互斥量最不同的是：非阻塞锁，它不会被挂起，而是在获取锁之前一直处于忙等，即 不停在消耗cpu，执行循环。适用于多核处理器、临界区无阻塞情况，其中一个 CPU 上的线程进入临界区，另一个 CPU 上的线程尝试获取锁会自旋，因为它不会阻塞，所以只要稍稍等一下下就能进入临界区（预计线程等待锁的时间很短，短到比线程两次上下文切换时间要少的情况下），对于多核 CPU 来说，会提高并发率。[加锁失败，不被阻塞，而是忙等]

**读写锁(read-write-clock)：**

可以多个线程同时读，但是不能多个线程同时写，锁处于读模式时可以线程共享，而锁处于写 模式时只能独占，所以读写锁又叫做共享-独占锁，读写锁比互斥锁更加具有适用性和并行性，最适用于对数据结构的读操作读操作次数多余写操作次数的场合。(适合读多写少的场景)

**信号量(sem)：**

信号量的实现中一般具有两种操作，分别是P操作和V操作。对一个信号量进行P操作时，首先检查其值是否大于0，如果大于0，则将其值减一之后返回进行后续操作，如果值小于等于0，则该进程将进行阻塞。(P，V都是原子操作)

信号量（sem）和互斥锁的区别：互斥锁只允许一个线程进入临界区，而信号量允许多个线程 进入临界区。

**条件变量(cond)：**

当线程在等待满足某些条件时使线程进入睡眠状态，一旦条件满足，就唤醒线程

在条件变量的内部，有一次解锁加锁过程，条件不满足，将本线程加入等待队列，同时将传入的 mutex变量解锁，一旦等待队列中的线程被唤醒，会再次对传入的 mutex变量加锁！[条件不满足时，将线程放入队列中，对mutex变量进行解锁（这两步都是原子操作）一旦线程被唤醒，就对mutex变量进行加锁]

条件变量产生的原因：

互斥量不是万能的，比如某个线程正在等待共享数据内某个条件出现，可能需要重复对数 据对象加锁和解锁（轮询），但是这样轮询非常耗费时间和资源，而且效率非常低，所以互斥锁不太适 合这种情况

## 线程池中的工作线程是一直等待吗？

当工作队列中没有待处理的任务时，线程池的工作线程阻塞进行等待，若有，竞争并获取进行处理，若没有,继续阻塞等待。

## 你的线程池工作处理玩一个任务后的状态时什么？

进行等待，看请求队列中有无请求，无的就阻塞等待，有的话进行处理

工作线程处理完后任务后，继续去查看请求队列，若有任务，则竞争并尝试获取资源，获取后进行处理，若没有，则就阻塞等待。

## 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？

因为这个项目采用的是Proactor，主线程读取完数据才插入任务队列。可以换成普通Reactor， 每个线程去接收数据，进一步的可以使用主从Reactor。

这个TinyWeb采用的是Proactor模式，主线程来完成读I/O操作，然后插入队列，工作线程俩进行去任务进行处理；因此，每个工作线程在同一时间只能做出一个客户请求，适合客户量不多的情况；若客户数量较多，则会导致队列中任务堆积，响应的速度越来越慢。

1. 通过增加线程数量可以解决问题，但工作线程的切换会浪费大量的CPU时间。
2. 可以用采用半同步/半异步的模式：改用Reactor模式，让每个工作线程去接受数据
   1. 主线程只监听socket，若有I/O处理事件，通知工作线程来处理
   2. 连接socket的工作和处理由工作线程完成，此后该socket的任何操作都由被选择的工作线程来完成，执导客户端关闭。

Proactor和Reactor的区别：

Proactor所有I/O操作都由主线程和内核完成，工作线程只负责业务逻辑的处理：

Reactor：主线程只负责监听文件描述符是否有事件发生，若有，就通知工作线程来处理(此后该lsocket的所有均由该工作线程完成直到关闭)： 读写数据，接受新连接，处理请求都在该工作线程完成

## 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?

是由影响的，在这个项目中，线程池是固定的，若某个客户长期占用某线程，则相当于线程可用的线程池的可用的线程数量-1了。

如果没有线程数限制，目前项目是固定线程数，可以改成动态线程池，连接请求过多的时候增加线程， 对线程池进行一个扩充。

## 简单说一下服务器使用的并发模型？

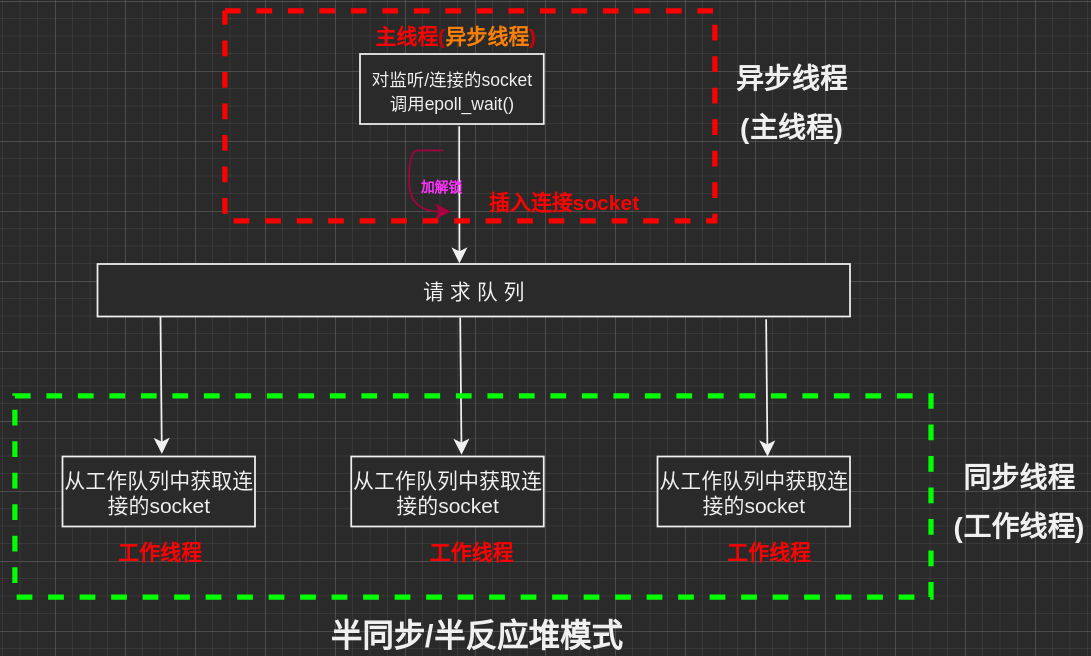
TinyWeb这个项目采用的是 epoll边缘触发(ET)模式来实现IO复用，使用同步IO模拟Proactor事件处理模式，实现 半同步/半反应堆线程池，主线程为异步线程，负责监听文件描述符，接收socket新连接，若当前监听 的socket发生了读写事件，然后将任务插入到请求队列。工作线程从请求队列中取出任务，完成读写数 据的处理，解除了主线程和工作线程的耦合关系

图1

## Reactor、Proactor、主从Reactor模型的区别？

**Reactor模式**中，主线程(**I/O处理单元**)只负责监听文件描述符上是否有事件发生，有的话立即通知 工作线程(**逻辑单元**)，读写数据、接受新连接及处理客户请求均在工作线程中完成。一般由**同步**

**I/O**实现。

**Proactor模式**中，主线程和内核负责处理读写数据、接受新连接等I/O操作，工作线程仅负责业务 逻辑，如处理客户请求。通常由**异步I/O**实现。

主从reactor

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

I/O多路复用: 是一种同步I/O模型，实现一个线程同时监听多个文件描述符的功能，多路指代的是网络连接，复用指的是同一个线程

epoll是Linux特有的I/O复用函数，在实现和使用上与seleclt，poll由很大区别：

1. epoll使用一组函数来完成任务，而不是单个函数(epoll\_create()，epoll\_wait(),epoll\_ctl())
2. epoll把用户关心的文件描述符上事件放在内核里的一个事件表中，避免select和poll每次调用的重复传入，
3. epoll需要一个额外的文件描述符来唯一标识内核中的事件表。

epoll工作过程：

1. **epoll\_wait()调用ep\_poll()**，如果就绪链表**rdlist**为空，则挂起当前进程，直到**rdlist**不为空时被唤醒，这个时候会调用**ep\_send\_events()将实际发生的事件revents**和**data**从内核空间拷贝到用户空间（拷贝调用的是put\_user，并不存在什么共享内存之类的）。
2. 当文件描述符fd的**状态改变**时（buﬀer由不可读变为可读或由不可写变为可写），导致相应fd上的 回调函数**ep\_poll\_callback()** 被调用。
3. **ep\_poll\_callback()将有事件发生的文件描述符（epitem）加入到就绪链表rdlist** 中，这时候就绪链表不为空，**epoll\_wait()** 进程被唤醒。
4. **ep\_send\_events()会扫描就绪链表**，调用每个文件描述符的**poll**函数返回**revents**，之后将

**revents**和**data**从内核空间拷贝到用户空间。如果是**ET模式**, epitem是**不会再进入到就绪链表，除非fd再次发生了状态改变**, **ep\_poll\_callback**被调用。如果是**LT模式**，不但会将对应的数据返回给用户，并且会**将当前的epitem再次加入到rdllist中**。这样如果下次再次被唤醒就会给用户空间再 次返回事件。

select poll epoll区别：

调用函数、文件描述符数量、将文件描述符从用户传给内核、内核判断就绪的文件描述符、应用程序索 引就绪文件描述符、工作模式、应用场景

* 调用函数

select，poll都是一个函数，epoll是一组函数

* 文件描述符数量

1. select底层是数组形式，文件描述符有上限，一般是1024，可以修改，但可能导致不可预期的后果。[可以修改源码，重新编译内核，不推荐]
2. poll底层是链表，用nfds参数指定监听的文件描述符，最大可以打开文件的数目65535
3. epoll底层是红黑树，同时维持一个list(就绪list)，用maxevents参数指定文件描述符，最大可以打开文件的数目65535，可以通过命令ulimit -n number修改，仅对当前终端有效

* 将文件描述符从用户传给内核

1. select需要在参数中指定需要监听的事件，它只能监听可读，可写和异常事件，且每次若有事件发生，内核都是直接堆fd\_set集合进行修改，因此，每次调用都需要重置fd\_set集并拷贝到内核态。获取就绪文件描述符需要遍历才能获取，因此为O(n)
2. poll把文件描述符和事都定义在pollfd结构体，任何事件都被统一处理，内核每次修改的是pollfd结构体的revents，而events保持不变，从而避免每次重置事件集参数。但获取就绪文件描述符仍然需要遍历才能获取，因此为O(n)
3. epoll通过epoll\_create建立一棵红黑树，通过epoll\_ctl将要监听的文件描述符注册到红黑树上，epoll在内核中维护一个事件表，并提供epoll\_ctl()函数进行事件的增加，删除和修改，每次epoll\_wait()调用都是直接从内核表取地用户的注册事件[就绪事件list]，从而，可用直接获取就绪文件描述符O(1)

* 内核判断就绪的文件描述符

1. select和poll只是返回了发生事件的文件描述符的个数，若需要找到哪个发生了事件，select和pol则l需要遍历文件描述符集合，判断哪个文件描述符上有事件发生,因此找到就绪的文件描述的时间复杂度是O(n)[select和poll都需要轮询]
2. epoll的文件描述符过程:内核检测到就绪的fd时，将触发回调函数，回调函数将该fd上对应的事件插入内核建立的就绪事件列表list中，在适当的实际将就绪队列的内容拷贝到用户空间。[采用回调的方式],epoll返回发生了事件的个数和结构体数组[就绪list的元素]，结构体包含了socket的信息，可以直接获取到哪个发生了事件。
3. 调用 epoll\_create()函数，内核首先在epoll文件系统中创建一红黑树用于存储之后epoll\_ctl()传入的fd，同时建立一个链表list，用于存储准备就绪的事件
4. 若内核表中的fd有事件发生，则调用该fd上的回调函数[中断函数]判断，若是就绪事件,则插入就绪的list中。[只有发生发生了事件的socket才会主动调用callback()函数，其他空闲的socket则不会]
5. 调用epoll\_wait()时，只是查看这个list链表中是否有数据可读。因此为O(1)

* 工作模式
  + 1. select和poll都只能工作在LT模式下[相对低效]
    2. epoll支持LT和ET[高效模式]两种工作模式，并且epoll还支持EPOLLONESHOT事件，该事件能进一步减 少可读、可写和异常事件被触发的次数。
* 应用场景

1. 当所有的fd都是活跃连接，使用epoll需要建立文件系统，红黑书和链表对于此来说，效率 反而不高，不如selece和poll
2. 当监测的fd数目较小，且各个fd都比较活跃，建议使用select或者poll
3. 当监测的fd数目非常大，成千上万，且单位时间只有一部分fd是活跃连接，这个时 候使用epoll能够明显提升性能

## ET、LT、EPOLLONESHOT

* LT水平触发模式[Level Trigger]

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，应用程序可以不立即处理 该事件;当下一次调用epoll\_wait时，epoll\_wait还会再次向应用程序报告此事件，直至被处理[一次就绪多次触发]

* ET边缘触发模式[Edge Trigger]

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 事件且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** 。ET在一定程度上降低同一个epoll事件被重复触发的次数。

* EPOLLONESHOT

一个线程读取某个socket上的数据后开始处理数据，在处理过程中该socket上又有新数据可 读，此时另一个线程被唤醒读取，此时出现两个线程处理同一个socket[连接的完整性];我们期望的是一个socket连接在任一时刻都只被一个线程处理，通过epoll\_ctl对该文件描述符注册**epolloneshot事件**，一个线程处理socket时，其他线程将无法处理，当该线程处理完后,需要通过epoll\_ctl重置epolloneshot事件

## 用了状态机啊，为什么要用？

服务器可以根据不同的转套和信息类型进行响应的处理逻辑，从而让程序逻辑清晰易懂。

主状态机内部调用从状态机，从状态机驱动主状态机。

### 前提知识点：

在HTTP报文中，每一行的数据由\r\n作为结束字符，空行则是仅仅是字符\r\n。因此，可以通过查找

\r\n将报文拆解成单独的行进行解析，项目中便是利用了这一点。

在TinyWeb这个项目中，使用了主状态机和从状态机：主状态机在内部调用从状态机，从状态机驱动主状态机。主状态机内部调用从状态机，从状态机内部完成check\_state的状态改变，反过来驱动主状态机调用相应的函数，从而实现状态转移。

从状态机parse\_line()函数，主要是对行数据进行解析;

### 从状态机三种状态：从的状态机parse\_line()函数，主要是对行数据进行解析

### line\_ok 读取到完整的一行

line\_bad 读取的行有语法错误

line\_open 读取的行不完整

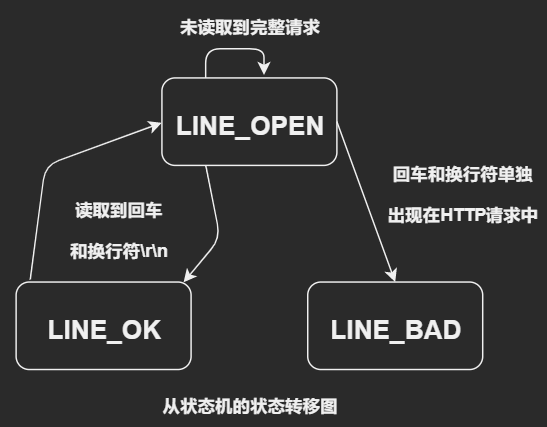
从状态级的状态及其转移过程如图5所示：

图5

从状态机负责读取buﬀer中的数据，将每行数据末尾的\r\n置为\0\0，并更新从状态机在buﬀer中读取 的位置m\_checked\_idx，以此来驱动主状态机解析。

### 主状态机

使用 check\_state 记录主状态机的状态，标识解析位置。check\_state 三种状态：

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

## 状态机的转移图画一下

主状态机根据状态调用不同的函数对请求进行解析，并设置状态机的状态。主状态机的内部过程如图6所示：

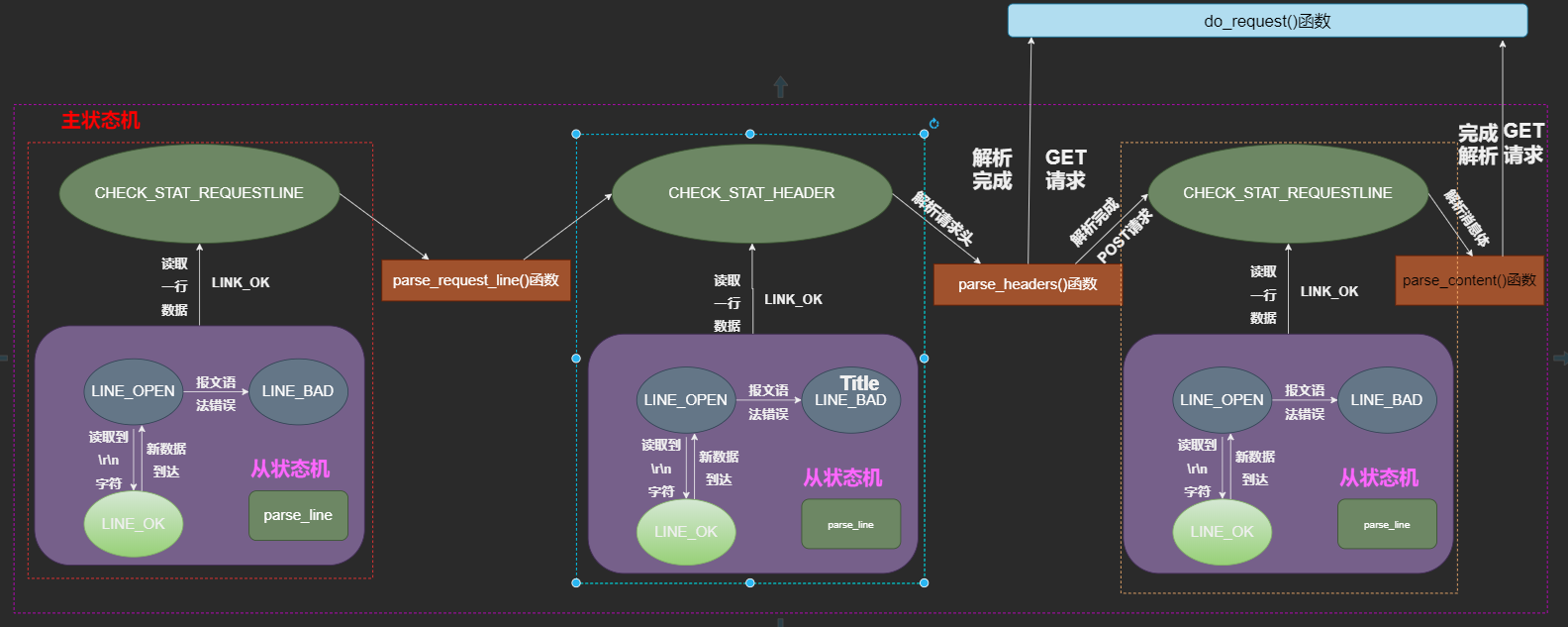


图6

## https协议为什么安全？

非对称加密算法（公钥和私钥）交换对称密钥+数字证书验证身份（验证公钥是否是伪造的）+利用对称 密钥加解密后续传输的数据

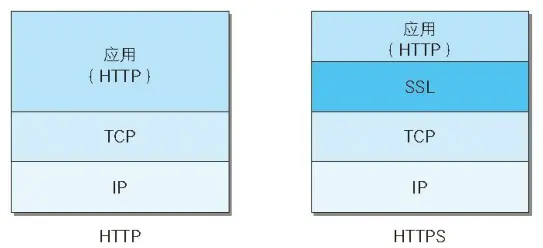
http 工作在应用层，明文传输，导致不安全：

窃听风险： 使用明文传输，内容可能被窃听

篡改风险： 报文的完整性无法保证，可能被篡改了

冒充风险：没有验证通讯方的身份，因此存在对方可能冒充的风险

https是针对http存在的以上问题进行改进的，https不是一种新的协议，是在http通信接口部分用SSL和TLS协议代替了。[HTTPS = HTTP + TLS/SSL]



1. 对于窃听风险：则采用对传输信息加密处理：混合加密

https采用的是混合加密，用了对称加密算法和非对称加密算法

在建立通信时中用非对称加密算法,保证安全性

连接建立完成后，用对称加密算法对传输的信息加密[对称加密的解密效率较高]

对称加密算法：加密和解密使用同一密钥，只要拿到密钥就可以进行加密和解密

发送密文时，必须将密钥也同时发给对方，因此存在安全问题。 非对称加密算法：公钥用来加密，私钥用来解密，发送秘文时，将公钥一起发送出去，对方使用公钥进行加密，自己收到密文后再用私钥进行解密，从而保证安全。

2. 对于篡改风险：则采用完整性校验: 数字签名（Digital Signature）

使用数字签名确保数据的完整性：

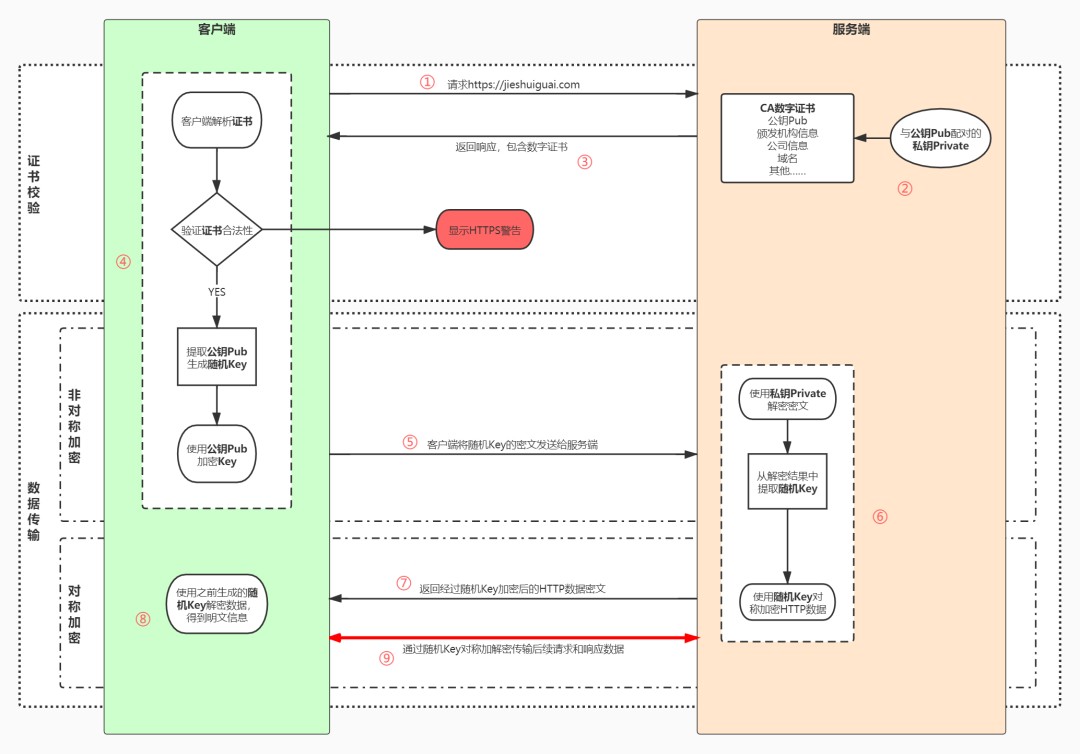
验证传输的内容是不是真实服务器发送的数据

发送的数据有没有被篡改过

3. 对于冒充风险：采用身份验证：数字证书（Certificate Authority）

使用数字证书[CA]确保通信方的身份

## HTTPS的SSL连接过程[有点麻烦]



## GET和POST的区别：

我们可以这样理解：一个URL地址，它用于描述一个网络上的资源，而HTTP中的GET，POST，PUT，

DELETE就对应着对这个资源的查，改，增，删4个操作。

GET一般用于获取/查询资源信息，而POST一 般用于更新资源信息。

GET是等幂的，多次操作结果相同，且不会改变服务器上的资源[查]；POST是非等幂的,操作会影响服务器上的资源[改]

GET从指定的资源请求数据并把传输的数据放在url中，大多数的俩长度在2K个字节；POST是向指定的服务器提交被处理的数据，将数据放在body中 ，可以存放更多的数据

GET - 从指定的资源请求数据,把传送的数据放在url中，大多数浏览器url长度在2K个字节，大多数服务器能处理最长64K字节的url，只产生一个TCP包，浏览器会把http header和data一并发送出去，服务器响应200（返回数据）；

POST - 向指定的资源提交要被处理的数据，把传送的数据放在body中，可以存放的数据多，浏览器先发送header，服务器响应100 continue，浏览器再发送data，服务器响应200 ok（返回数据）但是不同浏览器不一定是产生两个TCP包



## 数据库登录注册相关

**登录说一下？**

在初始化时，创建一个数据库的连接池，创建多个数据库连接，且集中管理。利用连接池完成登录和注册的校验功能。工作线程从连接池中获取一个连接，访问数据库的资源，访问完毕后，将连接交换给连接池。

使用POST请求完成注册和登录的校验工作：服务器端解析浏览器的请求报文[解析用户名和密码]，当解析为POST请求时，cgi 标志位设置为1，并将请求报文的消息体赋值给m\_string，进而提取出用户名和密码。

登录：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。

注册：判断获取的用户名是否在数据库中，若在，注册失败，提示已有该用户，否则注册成功，跳转到登录界面

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

没有进行保存，借助cookie跟session机制：

没有保存状态，http是无状态的协议，如果服务器端需要记录用户的状态，就需要用其他机制来追踪用户。比如Session机制和Cookie机制：

Session是在服务端保存的一种机制[数据结构]，用来跟踪用的状态，可以保存在集群，数据库，文件等中[默认保存在服务器一个文件中。[session保存在服务器端]

Cookie是客户端保存用户信息的一种机制，用来记录用户的相关消息。[cookie保存在客户端]

Session和Cookie通过sessionID来识别和追踪用户。

1．当一个用户登录后，服务器端为的用户创建对应的session，session中sessionID用于唯一表示这个用户。在响应客户请求时，将sessionID附加响应报文发给用户。Session保存在服务器端，可以保存在文件，内存，数据库等中

2．Session的运行依赖于sessionID， sessionID用来唯一标识和追踪用户，sessionID也存在客户端的Cookie，

3. Cookie是服务器发送客户端的特殊的消息，通过HTTP的响应报文中的set-cookie字段标识是否设置cookie。

4.客户端之后再每次发送请求时，发请求时都会自动带上cookie的信息。服务端通过cookie中的sessionID找到对应的session，从而识别用户。

## 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？

缓存 + 数据库

可以用缓存结合数据库的方法。

## 用的mysql啊，redis了解吗？用过吗？

Redis是一个内存数据库，它的数据存储在内存中，那么读写数据就在内存中完成，速度很快

Redis是一个KV内存数据库，内部构建了一个哈希表，根据key值 访问value,在O(1)的时间复杂度下找到对应数据。

Redis的数据类型：

Redis在3.0之前都采用单线程，可以在单线程中监听多个sokcet请求，任意一个可读/可写，就会采取对应措施

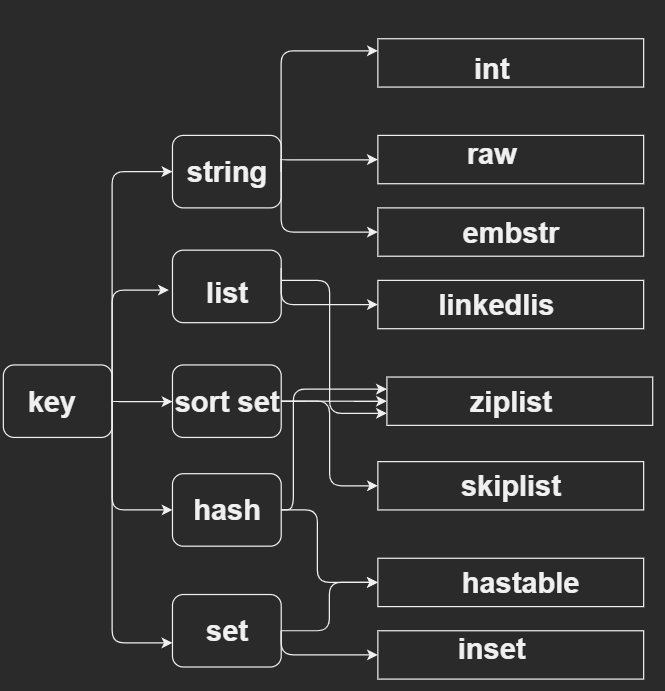
Redis采用单线程优势： 没有多线程上下文切换的性能开销，没有访问共享资源的加锁性能损耗， 开发和调试友好

Redis6.0引入了多线程完成数据协议的解析。

Redis采用IO多路复用技术和非阻塞IO

Redis非CPU密集型任务

Redis数据类型：



## 定时器相关

## 为什么要用定时器？

客户端（这里是浏览器）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，需要对这种占用资源的客户端进行剔除。

客户端[这里浏览器]与服务器建立连接后，若尝时间不交换数据，一直占用服务器的文件描述符，会导致连接资源的浪费，因此使用定时器定时对这种占用资源而不交换数据的客户端连接定期清理。

定时器主要谜底是是定期检查一个客户连接的活动状态。

## 说一下定时器的工作原理

链表：

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，利用升序时间链表容器将 所有定时器串联起来，若主循环接收到定时通知，则在链表中依次执行定时任务。具体来说，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，遍历定时器升序链表容器，从头结点开始依次处理每个定时器，直 到遇到尚未到期的定时器，对到期的定时器执行回调函数，关闭连接，释放资源

时间轮：

用循环数组实现一个时间轮，时间轮上的每一个槽【slot】指向每个槽指向一条定时器链表。

时间轮以恒定的速度顺时针转动，每转动一步就指向下一个槽。每次转动称位一个滴答tick，一个滴答的时间称位时间轮的槽间隔si[slot interval],共有N个槽，转动一次时间为si。

每条链表上的定时器具有相同的特征[它们的定时器时间相差N\*si的整数倍]，时间轮利用这种关系将定时器散列到不同的链表中, 链表将定时器串联起来。

时间轮将定时器散列到不同链表上，假设现在在 槽cs，若要添加一个定时时间为ti的定时器，则该定时将被添加到 ts的槽对应的链表：

ts = （cs + ti/si）%N

主循环收到定时通知，就去时间轮对应的槽中找到执行定时任务【怎么找到到期任务并执行】

由alarm函数周期性的触发SIGALRM信号，在主循环中对信号采用统一事件源[将信号当作IO事件]的方法，也即是信号理函数利用管通知主循环，主循环接收到信号后，对时间轮上的定时器进行处理，若时间到期，就对到期的定时器执行tcik()函数[回调函数],回调函数关闭连接，释放资源。

## 双向链表啊，删除和添加的时间复杂度说一下？还可以优化吗？

链表：添加时的效率偏低，添加O(n), 删除O(1) 执行O(1)

添加定时器时时间复杂度是O(n),删除是O(1)，从头开始遍历的直到遍历到未超时的。

优化：使用时间堆，用最小堆来构建，堆顶的元素就是最小的，将所有定时器中的超时时间最小的定时 器的超时值作为一个tick，他到时间以后就将它提出，在堆中找当前最小的。

定时器：添加事件O(1),删除时间O(1)，执行时间O(n)[实际上执行一个定时器任务比O(n)好很多,因为时间轮加你个所有定时器散列到不同链表上，时间轮的槽越多，每条链表的定时器数量越少]。类似 散列 + 链表

## 最小堆优化？说一下时间复杂度和工作原理

添加是log(n) 上浮操作，删除是log(1)，但是有个下沉的操作是log(n)

# 日志相关

## 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。

流程：

日志文件

局部变量的懒汉模式获取实例

生成日志文件，并判断同步和异步写入方式

同步

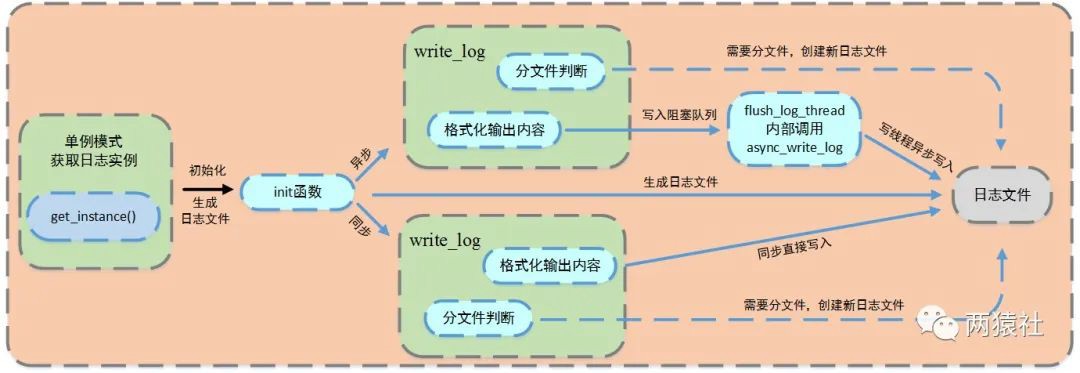
判断是否分文件

直接格式化输出内容，将信息写入日志文件

异步

判断是否分文件

格式化输出内容，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



## 为什么要异步？和同步的区别是什么？

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。

## 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）

使用消息队列的发布订阅

# 压测相关

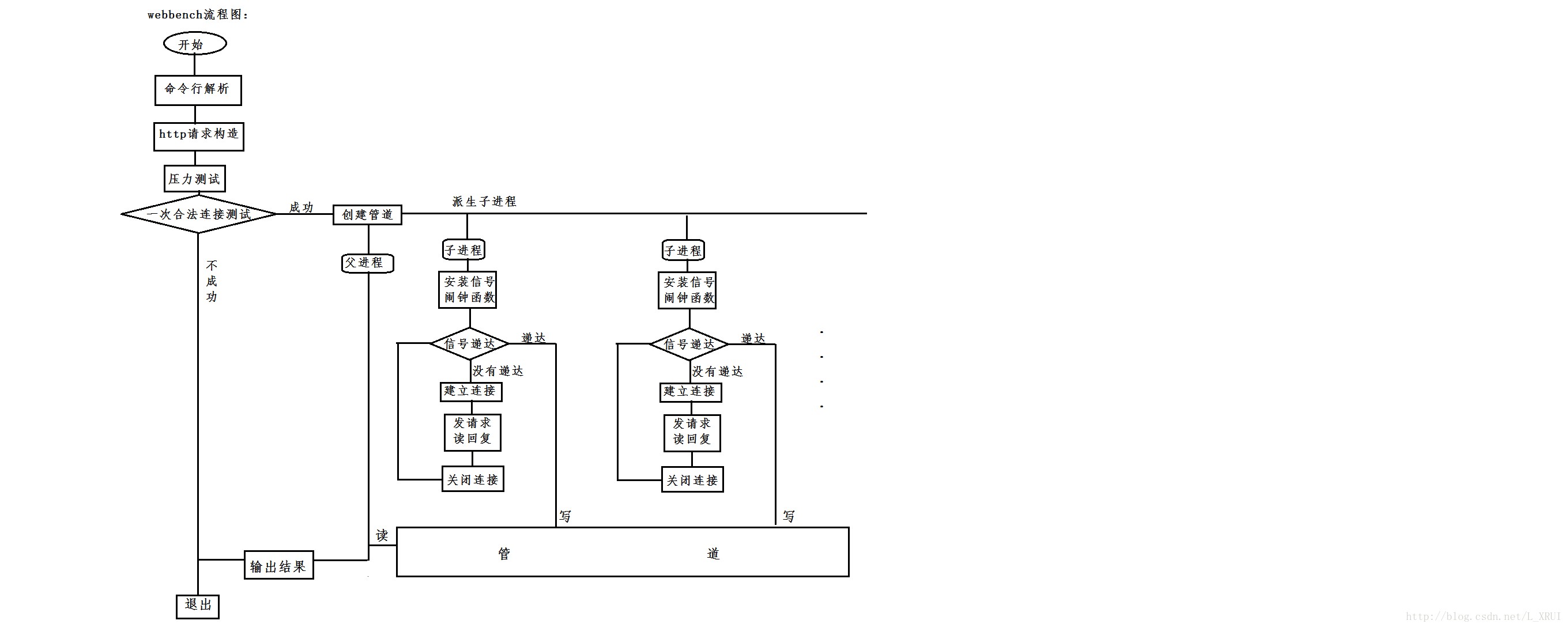
## 服务器并发量测试过吗？怎么测试的？

有的，使用webbench，在关闭日志后，使用Webbench对服务器进行压力测试，对listenfd和connfd 分别采用ET和LT模式，均可实现上万的并发连接。

webbench -c 10500 -t 5 h ttp://121.48.164.15:9006/

## webbench是什么？介绍一下原理

Webench是一款轻量级的网站测压工具，最多可以对网站模拟3w左右的并发请求，可以控制时间、是 否使用缓存、是否等待服务器回复等等，且对中小型网站有明显的效果，基本上可以测出中小型网站的 承受能力，对于大型的网站，如百度、淘宝这些巨型网站没有意义，因为其承受能力非常大。同时测试 结果也受自身网速、以及自身主机的性能与内存的限制，性能好、内存大的主机可以模拟的并发就明显 要多。

基本原理：父进程fork若干个子进程，每个子进程在用户要求时间或默认的时间内对目标web循环发出 实际访问请求，父子进程通过管道进行通信，子进程通过管道写端向父进程传递在若干次请求访问完毕 后记录到的总信息，父进程通过管道读端读取子进程发来的相关信息，子进程在时间到后结束，父进程 在所有子进程退出后统计并给用户显示最后的测试结果，然后退出。

## 测试的时候有没有遇到问题？

大文件、epoll accept问题

