0.1版本下实现了以下功能：

1、使用epoll边沿触发的EPOLLSHONESHOT加非阻塞IO

epoll ET(边沿触发模型)：EPOLLIN | EPOLLET

非阻塞IO：O\_NONBLOCK

2、使用了一个固定线程数的线程池；

thread\_count = 4

queue\_size = 65535

//初始化线程池中的任务队列实际大小

pool->queue = (threadpool\_task\_t \*)*malloc*(sizeof(threadpool\_task\_t) \* queue\_size);

typedef struct

{

void(\*function)(void\*);

void \*argument;

}threadpool\_task\_t;

threadpool\_t \*threadpool\_create(int thread\_count, int queue\_size, int flags)

3、实现了一个任务队列，由条件变量触发通知新任务的到来

//int threadpool\_add(threadpool\_t \*pool, void(\*function)(void \*), void \*argument, int flags)

/\*这个回调函数即为处理请求数据的回调函数 myHandler -》 handleRequest\*/

int rc = threadpool\_add(tp, myHandler, events[i].*data*.*ptr*, 0);->添加任务

pthread\_cond\_wait(&(pool->notify), &(pool->*lock*));->唤醒线程处理任务

4、实现了一个小根堆的定时器及时剔除超时请求，使用了STL的优先级队列来管理定时器

//设置定时器优先级队列 小顶堆 是根据我们所设置的请求过期时间来判断的

*priority\_queue*<mytimer\*, *deque*<mytimer\*>, timerCmp> myTimerQueue;

5、解析了HTTP的get、post请求，支持长短连接

int parse\_URL();

如果要支持长连接需要在客户端和服务端同时设置keep-alive属性

如果支持短连接则如果需要一直访问的资源则对服务器的资源开销压力大 这里我们设计的是静态服务器，因此无须考虑开销

6、mime设计为单例模式

static pthread\_mutex\_t lock;

static *std*::*unordered\_map*<*std*::*string*, *std*::*string*> mime;

static *std*::*string* getMime(const *std*::*string* &suffix);

7、线程的工作分配：

1）主线程负责等待epoll事件，并把到来的事件放进任务队列，在每次循环结束用来剔除超时请求和被设置为删除的时间节点

void handle\_events(int epoll\_fd, int listen\_fd, struct epoll\_event\* events, int events\_num, const *string* &path, threadpool\_t\* tp)

void handle\_expired\_event();

2）工作线程阻塞在条件变量的等待中，新任务到来，某一工作线程会被唤醒，执行具体的IO操作和计算任务。如果需要继续监听，会添加到epoll中.

pthread\_cond\_wait(&(pool->notify), &(pool->*lock*));

线程阻塞在此 等待唤醒 如果可执行线程数为0 且排除是关闭唤醒 则即可执行该语句

8、锁的使用

1）任务队列的添加和取操作，都需要加锁，并配合条件变量，跨越了多个线程

pthread\_mutex\_lock(&(pool->*lock*))

pthread\_mutex\_unlock(&(pool->*lock)*)

2）定时器节点的添加和删除，都需要加锁，主线程和工作线程都需要操作定时器队列

pthread\_mutex\_lock(&qlock);

pthread\_mutex\_unlock(&qlock);

1. 指针可以改成shared\_ptr,不需要delete

shared\_ptr 和 weak\_ptr的区别及应用，在陈硕的linux多线程服务器端编程一书中谈及

1. 可以将某些地方写成单例模式
2. readn 和 writen改成写到不能写后或读到不能读后加入epoll等待，要记录写到哪或读到哪里，维护这样一个状态，比较两种的性能差异
3. 信号处理部分可以将epoll\_wait替换为更加安全的epoll\_pwait。

int epoll\_pwait(int epfd, struct epoll\_event \*events, int maxevents, int timeout, const sigset\_t \*sigmask);

The relationship between epoll\_wait() and epoll\_pwait() is analogous to the relationship between select(2) and pselect(2): like pselect(2), epoll\_pwait() allows an application to safely wait until either a file descriptor becomes ready or until a signal is caught.

The sigmask argument may be specified as NULL, in which case epoll\_pwait() is equivalent

to epoll\_wait().

When successful, epoll\_wait() returns the number of file descriptors ready for the requested I/O, or zero if no file descriptor became ready during the requested timeout milliseconds. When an error occurs, epoll\_wait() returns -1 and errno is set appropriately.

修改1：改变了程序执行的工作路径 如果需要通过键入程序执行路径和端口也可以，只需要在主函数参数中加入int arg，int\* args即可，该版本暂时没有涉及此功能。

加入的方式很简单，只需要用两个变量来接受端口和访问文件夹目录即可

修改2：加入访问端口即展示文件目录：

面临的问题1：如果请求的是文件夹 应该取消默认跳转为首页（2019.8.29）

首先解决第一个问题再解决第二个问题

处理问题：程序退出才能显示点击文件夹中的内容 说明控制循环出现问题

面临的问题2：编码和解码（2019.8.30）

首先解决的是编码的问题，通过编码我们把请求地址进行重新编码

其次是要解决解码的问题，通过解码可以让浏览器对已经进行的汉子编码进行重新解码认识

面临的问题3：浏览器不同所带来的访问不同-主要是体现在浏览器一直显示在访问资源(该版本未解决)

360浏览器在访问过程中一经刷新直接完成资源访问，也就是request析构函数执行

火狐浏览器访问时析构函数有时需要很久才能执行，因此会显示一直在获取资源，事实上资源已经获取完毕

Google浏览器同样也出现类似火狐这样的情况，在请求文件夹时，虽然资源已经获取完全但是网页却一直在显示获取资源当中。

1、POST和GET请求的区别

GET在浏览器回退时是无害的，而POST会再次提交请求。

GET产生的URL地址可以被Bookmark，而POST不可以。

GET请求会被浏览器主动cache，而POST不会，除非手动设置。

GET请求只能进行url编码，而POST支持多种编码方式。

GET请求参数会被完整保留在浏览器历史记录里，而POST中的参数不会被保留。

GET请求在URL中传送的参数是有长度限制的，而POST么有。

对参数的数据类型，GET只接受ASCII字符，而POST没有限制。

GET比POST更不安全，因为参数直接暴露在URL上，所以不能用来传递敏感信息。

GET参数通过URL传递，POST放在Request body中。

2、长连接和短连接的区别（Connection: Keep-alive）

第一个区别是决定的方式，一个TCP连接是否为长连接，是通过设置HTTP的Connection Header来决定的，而且是需要两边都设置才有效。而一种轮询方式是否为长轮询，是根据服务端的处理方式来决定的，与客户端没有关系。

第二个区别就是实现的方式，连接的长短是通过协议来规定和实现的。而轮询的长短，是服务器通过编程的方式手动挂起请求来实现的。

3、keep-alive中的timeout时间设置的问题

//在HTTP 1.1版本后，默认都开启Keep-Alive模式，只有加入 Connection: close才关闭连接，当然也可以设置Keep-Alive模式的属性，

//例如 Keep-Alive: timeout=5, max=100，表示这个TCP通道可以保持5秒，max=100，表示这个长连接最多接收100次请求就断开。

4、content-length的值的设置：

1、Content-Length如果存在并且有效的话，则必须和消息内容的传输长度完全一致。（经过测试，如果过短则会截断，过长则会导致超时。）  
2、如果存在Transfer-Encoding（重点是chunked），则在header中不能有Content-Length，有也会被忽视。  
3、如果采用短连接，则直接可以通过服务器关闭连接来确定消息的传输长度。（这个很容易懂）  
结合HTTP协议其他的特点，比如说Http1.1之前的不支持keep alive。那么可以得出以下结论：  
1、在Http 1.0及之前版本中，content-length字段可有可无。  
2、在http1.1及之后版本。如果是keep alive，则content-length和chunk必然是二选一。若是非keep alive，则和http1.0一样。content-length可有可无。

5、对EPOLLONESHOT的误解，原以为当epoll\_wait监听到相应的事件触发后，epoll会把与事件关联的fd从epoll描述符中禁止掉并且彻底删除，实际上并不会删除，man手册上的解释如下：

EPOLLONESHOT (since Linux 2.6.2)

Sets the one-shot behavior for the associated file descriptor.

This means that after an event is pulled out with

epoll\_wait(2) the associated file descriptor is internally

disabled and no other events will be reported by the epoll

interface. The user must call epoll\_ctl() with EPOLL\_CTL\_MOD

to rearm the file descriptor with a new event mask.

即使使用ET模式,一个socket上的某个事件还是可能被触发多次.这在并发程序中会引发一个问题. 比如一个线程在读取完某个socket上的数据后开始处理这些数据,而在数据的处理过程中,该socket上又有新数据可读(EPOLLIN再次被触发),此时另外一个线程被唤醒来读取这些新的数据.于是出现了两个线程同时操作一个socket的局面.我们期望的是一个socket连接在任何一个时刻只能被一个线程处理.对于注册了EPOLLONESHOT事件的文件描述符,操作系统最多只触发其上注册的一个可读,可写,异常事件,且只触发一次.除非我们使用epoll\_ctl函数重置该文件描述符上注册的EPOLLONESHOT事件.