相比于0.4版本0.5版本下做了以下修改：

问题：

1. 客户端一直刷新请求资源 一段时间过后 服务器出现段错误
2. 在目录请求未完成时 继续请求则出现段错误
3. 一直都存在的问题是客户端存在一直在请求无法析构请求的情况

修改了一下bug，主要是一直刷新出现的段错误，稍微调整了类的结构

封装了条件变量

1、封装了条件变量类

class Condition : noncopyable

{

public:

//避免隐式转换

explicit Condition(MutexLock &\_mutex) : mutex(\_mutex)

{

pthread\_cond\_init(cond, *NULL*);

}

~Condition()

{

pthread\_cond\_destroy(cond);

}

void wait()

{

/\*int pthread\_cond\_wait(pthread\_cond\_t \*restrict cond, pthread\_mutex\_t \*restrict mutex);

函数作用：

1. 阻塞等待条件变量cond（参1）满足

2. 释放已掌握的互斥锁（解锁互斥量）相当于pthread\_mutex\_unlock(&mutex);

1.2.两步为一个原子操作。

3. 当被唤醒，pthread\_cond\_wait函数返回时，解除阻塞并重新申请获取互斥锁pthread\_mutex\_lock(&mutex);

\*/

pthread\_cond\_wait(&cond, mutex.get());

}

void notify()

{

//利用条件变量的方式进行唤醒 随机唤醒至少一个

//如果要全部唤醒阻塞等待的线程 应该使用广播 pthread\_cond\_broadcast(&cond);

pthread\_cond\_signal(&cond);

}

void notifyAll()

{

//利用广播的方式就行唤醒 全部唤醒阻塞在条件变量中的线程

pthread\_cond\_broadcast(&cond);

}

private:

MutexLock &mutex;

pthread\_cond\_t cond;

};

2、以前是用hash表进行存储fd -> request 现在我们直接使用数组进行存储，在后续进行删除的时候不需要再使用 fd2req.erase(fd\_ite);进行删除，这样我们就可以继续重复利用该类型，只需要reset即可，而实际上该类型变量还是存储在空间中

//将智能指针加入类型的变量 加入到hash表中 该hash表中的存储是 fd(套接字) --> 对应请求

//static std::unordered\_map<int, SP\_ReqData> fd2req;

/\*用数组的方式来存储请求 shared\_ptr<RequestData>类型的请求\*/

static SP\_ReqData fd2req[MAXFDS];

/\*auto fd\_ite = fd2req.find(fd);

if (fd\_ite != fd2req.end())

{

fd2req.erase(fd\_ite);

}\*/

3、重写封装了read方法和write方法，直接对字符串进行读写操作

ssize\_t readn(int fd, *std*::*string* &inBuffer)

{

ssize\_t nread = 0;

ssize\_t readSum = 0;

while (true)

{

char buff[MAX\_BUFF];

if ((nread = *read*(fd, buff, MAX\_BUFF)) < 0)

{

if (*errno* == *EINTR*)

continue;

else if (*errno* == *EAGAIN*)

{

return readSum;

}

else

{

*perror*("read error");

return -1;

}

}

else if (nread == 0)

break;

//printf("before inBuffer.size() = %d\n", inBuffer.size());

//printf("nread = %d\n", nread);

readSum += nread;

//buff += nread;

/\*每次将buff中读到的数据传递给inBuffer, 最后inBuffer传出的数据就是读到数据\*/

inBuffer += *std*::*string*(buff, buff + nread);

//printf("after inBuffer.size() = %d\n", inBuffer.size());

}

return readSum;

}

ssize\_t writen(int fd, *std*::*string* &sbuff)

{

*size\_t* nleft = sbuff.*size*();

ssize\_t nwritten = 0;

ssize\_t writeSum = 0;

const char \*ptr = sbuff.*c\_str*();

while (nleft > 0)

{

if ((nwritten = *write*(fd, ptr, nleft)) <= 0)

{

if (nwritten < 0)

{

if (*errno* == *EINTR*)

{

nwritten = 0;

continue;

}

else if (*errno* == *EAGAIN*)

break;

else

return -1;

}

}

writeSum += nwritten;

nleft -= nwritten;

ptr += nwritten;

}

if (writeSum == sbuff.*size*())

sbuff.*clear*();

else

sbuff = sbuff.*substr*(writeSum);

return writeSum;

}

1. 将handleRequest拆成了handleRead和handleWrite以及handleConn

将outbuffer中的内容读给fd套接字

void RequestData::handleWrite()

{

if (!error)

{

if (writen(fd, outBuffer) < 0)

{

*perror*("written");

events = 0;

error = true;

}

else if (outBuffer.*size*() > 0)

{

events |= EPOLLOUT;

}

}

}

处理连接

void RequestData::handleConn()

{

if (!error)

{

if (events != 0)

{

//一定要先加时间信息，否则可能会出现刚加进去，下一个in触发来了，然后分离失败后，又加入队列，最后超时被删除，

//然后正在线程中进行的任务出错，double free错误

int timeout = 2000;

if (keep\_alive)

{

//以毫秒计数，表示如果keep\_live = true时，则表示超时时间设置为5分钟

timeout = 5 \* 60 \* 1000;

}

isAbleRead = false;

isAbleWrite = false;

Epoll::add\_timer(*shared\_from\_this*(), timeout);

if ((events & EPOLLIN) && (events & EPOLLOUT))

{

events = \_\_uint32\_t(0);

events |= EPOLLOUT;

}

//设置边沿触发模式 并且一次触发

events |= (EPOLLET | EPOLLONESHOT);

\_\_uint32\_t \_events = events;

events = 0;

if (Epoll::epoll\_mod(fd, *shared\_from\_this*(), \_events) < 0)

{

*printf*("Epoll::epoll\_mod error\n");

}

}

else if (keep\_alive)

{

events |= (EPOLLIN | EPOLLOUT | EPOLLONESHOT);

int timeout = 5 \* 60 \* 1000;

isAbleRead = false;

isAbleWrite = false;

Epoll::add\_timer(*shared\_from\_this*(), timeout);

\_\_uint32\_t \_events = events;

events = 0;

if (Epoll::epoll\_mod(fd, *shared\_from\_this*(), \_events) < 0)

{

*printf*("Epoll::epoll\_mod error\n");

}

}

}

}

1. 当代码堆好之后，以前一直刷新出现段错误的问题得到了解决，但是由于使用outbuff来存header和body出现了浏览器无法显示图片和.mp3文件等信息

V0.4版以前 服务端给客户端发消息的流程是：先发头文件，在发body内容

V0.5开始 将read读的内容和write的内容 分别放在了inbuffer和outbuffer中，同时在inbuffer和outbuffer的开始部分放的是header头文件的内容标签等信息

服务端发数据的方式是服务端调用handlewrite()方法，将outbuffer中的数据通过套接字传给客户端

1、检查浏览器时发现头部header的内容基本上没有问题

2、通过先发头文件，再发body内容的方式也无法解决

解决方式：（但是有个巨大的改变就是该版本本来设计了outBuffer缓冲区进行装内容 但是只适合文本文件 却不适合除开文本文件以外的其他文件）

由于使用outBuffer统一发送时出现指定.mp3文件无法读取的问题

//先发头文件再发body文件

writen(fd, header);

// 发送文件并校验完整性

int send\_len = writen(fd, src\_addr, sbuf.*st\_size*);

if (send\_len != sbuf.*st\_size*)

{

*perror*("Send file failed");

return ANALYSIS\_ERROR;

}

1. 关于无法读取到favion.ico文件的解决/请求文件时，浏览器一直处于请求状态

通过查看浏览器开发者工具 发现存在一个大的问题就是在请求文件夹的时候直到我们停止程序时才出现状态码，请求头，首先对该问题进行修改：

经过此次修改，该问题依旧没有解决

前端控制台报错，初步猜想是由于发送时所检测的目录大小和最后发送的目录大小不相同，通过查看原来成功的代码案例尝试找到该问题的解决方案：

GET http://192.168.242.129:8888/ net::ERR\_CONTENT\_LENGTH\_MISMATCH 200 (OK)：

通过将文件的content-length值设置为-1改变了这一问题，但是在google浏览器中提示的是favion.ico文件无法获取，在linux火狐浏览器中只有通过ctr+c结束系统才可以在客户端浏览器显示，并且客户端浏览器请求完毕。

该版本始终没有解决该问题