TinyWebServer: C++搭建的网络后端服务器

1. 项目综述

- 利用I/O复用技术epoll与线程池实现多线程的Reactor高并发模型；(code/server)
- 利用正则表达式与状态机解析HTTP请求报文，实现处理静态资源的请求，并发送响应报文；(code/http)
- 利用标准库容器封装char，实现自动增长的缓冲区；(code/buffer)
- 基于小根堆实现的定时器，关闭超时的非活动连接；(code/timer)
- 利用单例模式与阻塞队列实现异步的日志系统，记录服务器运行状态；(code/log)
- 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。(code/pool)

2. 项目功能

本项目实现了以下主要功能:

1. 监听端口: 服务器能够监听指定端口，等待客户端的连接请求。
2. 处理 HTTP 请求: 服务器能够解析客户端发送的 HTTP 请求，包括请求方法、URL、请求头、请求体等内容。
3. 处理静态资源: 能够根据客户端请求的 URL，返回相应的静态资源文件，如 HTML 文件、CSS 文件、JavaScript 文件、图片等。
4. HTTP 响应: 根据客户端请求的处理结果，生成相应的 HTTP 响应，包括状态码、响应头、响应体等内容。
5. 并发处理:采用多线程或多路复用技术，实现并发处理客户端请求，提高服务器的并发能力。
6. 连接管理:通过定时器管理客户端连接，及时关闭空闲连接，防止资源浪费和连接泄漏。
7. 日志记录: 记录服务器运行过程中的重要信息和错误日志，方便排查和分析问题。
8. 异常处理: 处理客户端异常断开、请求超时等异常情况，保证服务器的稳定性和可靠性。
9. 优雅关闭: 在关闭服务器时，能够优雅地关闭监听套接字、释放资源并通知客户端连接关闭。
10. 配置管理: 提供配置文件或参数设置接口，方便用户灵活配置服务器的运行参数和行为。
11. 性能优化: 通过优化算法、数据结构和网络通信方式，提高服务器的性能和吞吐量。
12. 安全防护:采取安全措施防范常见的网络攻击，如拒绝服务攻击、SQL注入攻击等。
13. 可扩展性: 设计良好的架构和模块化组件，方便扩展和定制，满足不同需求和业务场景。

3. 实现思路(具体细节见代码注释)

3.1 buffer

这个类的设计源自陈硕大佬的muduo网络库。由于muduo库使用的是非阻塞I/O模型，即每次send()不一定会发送完，没发完的数据要用一个容器进行接收，所以必须要实现应用层缓冲区。

3.1.1 buffer 结构

```
// buffer的结构
// |-----------------buffer_-----------------|
//            readPos_     writePos_
//               ^            ^
// |---已读数据---|---可读数据---|---可写数据---|
```

3.1.2 buffer.h

可以看到

```
std::atomic<std::size_t> readPos_;  // 读的下标
std::atomic<std::size_t> writePos_; // 写的下标
```

C++中原子变量（atomic）是一种多线程编程中常用的同步机制，它能够确保对共享变量的操作在执行时不会被其他线程的操作干扰，从而避免竞态条件（race condition）和死锁（deadlock）等问题。

原子变量可以看作是一种特殊的类型，它具有类似于普通变量的操作，但是这些操作都是原子级别的，即要么全部完成，要么全部未完成。C++标准库提供了丰富的原子类型，包括整型、指针、布尔值等，使用方法也非常简单，只需要通过std::atomic定义一个原子变量即可，其中T表示变量的类型。

在普通的变量中，并发的访问它可能会导致数据竞争，竞争的后果会导致操作过程不会按照正确的顺序进行操作。

3.1.3 buffer.cpp

基于 buffer的结构的扩容、读写buffer等操作(具体实现见代码及其注释)

3.2 log

3.2.1 异步log

日志系统在整个项目中能够帮助调试、错误定位、数据分析。为设计一个日志模块，能顺利写日志但是又不要占用主线程时间去写，所以要设计异步写日志的模块。

- 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。
- 异步日志：将所写的日志内容先存入阻塞队列中，写线程从阻塞队列中取出内容，写入日志。

考虑到文件IO操作是非常慢的，所以采用异步日志就是先将内容存放在内存里，然后日志线程有空的时候再写到文件里。

3.2.2 日志的分级与分文件

分级情况：

- Debug，调试代码时的输出，在系统实际运行时，一般不使用。

- Warn，这种警告与调试时终端的warning类似，同样是调试代码时使用。
- Info，报告系统当前的状态，当前执行的流程或接收的信息等。
- Erro，输出系统的错误信息

分文件情况：

- 按天分，日志写入前会判断当前today是否为创建日志的时间，若为创建日志时间，则写入日志，否则按当前时间创建新的log文件，更新创建时间和行数。
- 按行分，日志写入前会判断行数是否超过最大行限制，若超过，则在当前日志的末尾加lineCount / MAX_LOG_LINES为后缀创建新的log文件。

3.2.3 blockqueue.h

注意锁和信号量的使用

3.2.4 log.h 和 log.cpp

见代码及其注释

3.3 pool

3.3.1 线程池

使用线程池可以减少线程的销毁，而且如果不使用线程池的话，来一个客户端就创建一个线程。比如有1000，这样线程的创建、线程之间的调度也会耗费很多的系统资源，所以采用线程池使程序的效率更高。 线程池就是项目启动的时候，就先把线程池准备好。一般线程池的实现是通过生产者消费者模型来的.

线程同步问题涉及到了互斥量、条件变量。 在代码中，将互斥锁、条件变量、关闭状态、工作队列封装到了一起，通过一个共享智能指针来管理这些条件。

3.3.1.1 threadpool.h

```
#ifndef THREADPOOL_H
#define THREADPOOL_H

#include <queue>
#include <mutex>
#include <condition_variable>
#include <functional>
#include <thread>
#include <assert.h>


class ThreadPool
{
public:
    ThreadPool() = default; // 默认构造函数
    ThreadPool(ThreadPool&&) = default; // 移动构造函数
    // 尽量用make_shared代替new，如果通过new再传递给shared_ptr，内存是不连续的，会造成内存碎片化
```

```
    explicit ThreadPool(int threadCount = 8) : pool_(std::make_shared<Pool>())
    { // make_shared:传递右值，功能是在动态内存中分配一个对象并初始化它，返回指向此对象
的shared_ptr
        assert(threadCount > 0);
        for(int i = 0; i < threadCount; i++)
        {
            // 创建线程，线程函数是lambda表达式
            std::thread([this]()
            {
                std::unique_lock<std::mutex> locker(pool_->mtx_);
                while(true) {
                    if(!pool_->tasks.empty()) {
                        auto task = std::move(pool_->tasks.front());    // 左值变
右值,资产转移
                        pool_->tasks.pop();
                        locker.unlock();    // 因为已经把任务取出来了，所以可以提前解
锁了
                        task();
                        locker.lock();      // 马上又要取任务了，上锁
                    }
                    else if(pool_->isClosed)
                    {
                        break;
                    }
                    else
                    {
                        pool_->cond_.wait(locker);    // 等待,如果任务来了就notify
的
                    }
                }
            }).detach();
        }
    }

    ~ThreadPool() {
        if(pool_) {
            std::unique_lock<std::mutex> locker(pool_->mtx_);
            pool_->isClosed = true;
        }
        pool_->cond_.notify_all();  // 唤醒所有的线程
    }

    template<typename T>
    void AddTask(T&& task) {
        std::unique_lock<std::mutex> locker(pool_->mtx_);
        pool_->tasks.emplace(std::forward<T>(task));
        pool_->cond_.notify_one();
    }

private:
    // 用一个结构体封装起来，方便调用
    struct Pool {
        std::mutex mtx_;
        std::condition_variable cond_;
```

```
        bool isClosed;
        std::queue<std::function<void()>> tasks; // 任务队列，函数类型为void()
    };
    std::shared_ptr<Pool> pool_;
};

#endif
```

3.3.2 连接池

3.3.2.1 什么是RAII?

RAII是Resource Acquisition Is Initialization（wiki上面翻译成 “资源获取就是初始化”）的简称，是C++语言的一种管理资源、避免泄漏的惯用法。利用的就是C++构造的对象最终会被销毁的原则。RAII的做法是使用一个对象，在其构造时获取对应的资源，在对象生命期内控制对资源的访问，使之始终保持有效，最后在对象析构的时候，释放构造时获取的资源。

3.3.2.2 为什么要使用RAII?

上面说到RAII是用来管理资源、避免资源泄漏的方法。那么，用了这么久了，也写了这么多程序了，口头上经常会说资源，那么资源是如何定义的？在计算机系统中，资源是数量有限且对系统正常运行具有一定作用的元素。比如：网络套接字、互斥锁、文件句柄和内存等等，它们属于系统资源。由于系统的资源是有限的，就好比自然界的石油，铁矿一样，不是取之不尽，用之不竭的，所以，我们在编程使用系统资源时，都必须遵循一个步骤：

- 1 申请资源；
- 2 使用资源；
- 3 释放资源。

第一步和第三步缺一不可，因为资源必须要申请才能使用的，使用完成以后，必须要释放，如果不释放的话，就会造成资源泄漏。

我们常用的智能指针如unique_ptr、锁lock_guard就是采用了RAII的机制。

在使用多线程时，经常会涉及到共享数据的问题，C++中通过实例化std::mutex创建互斥量，通过调用成员函数lock()进行上锁，unlock()进行解锁。不过这意味着必须记住在每个函数出口都要去调用unlock()，也包括异常的情况，这非常麻烦，而且不易管理。C++标准库为互斥量提供了一个RAII语法的模板类std::lock_guard，其会在构造函数的时候提供已锁的互斥量，并在析构的时候进行解锁，从而保证了一个已锁的互斥量总是会被正确的解锁。

3.3.2.3 为什么要使用连接池?

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。

在连接池的实现中，使用到了信号量来管理资源的数量；而锁的使用则是为了在访问公共资源的时候使用。所以说，无论是条件变量还是信号量，都需要锁。

不同的是，信号量的使用要先使用信号量sem_wait再上锁，而条件变量的使用要先上锁再使用条件变量wait。

3.4 http

3.4.1 http解析

3.4.1.1 httprequest

httprequest.h

```
#ifndef HTTP_REQUEST_H
#define HTTP_REQUEST_H

#include <unordered_map>
#include <unordered_set>
#include <string>
#include <regex>    // 正则表达式
#include <errno.h>
#include <mysql/mysql.h>  //mysql

#include "../buffer/buffer.h"
#include "../log/log.h"
#include "../pool/sqlconnpool.h"

class HttpRequest {
public:
    enum PARSE_STATE // 解析状态
    {
        REQUEST_LINE,
        HEADERS,
        BODY,
        FINISH,
    };

    HttpRequest() { Init(); }
    ~HttpRequest() = default;

    void Init();
    bool parse(Buffer& buff);

    std::string path() const; // 获取路径
    std::string& path();      // 获取路径
    std::string method() const; // 获取方法
    std::string version() const; // 获取版本
    std::string GetPost(const std::string& key) const;  // 获取post请求，重载
    std::string GetPost(const char* key) const; // 获取post请求

    bool IsKeepAlive() const;
```

```
private:
    bool ParseRequestLine_(const std::string& line);    // 处理请求行
    void ParseHeader_(const std::string& line);         // 处理请求头
    void ParseBody_(const std::string& line);           // 处理请求体

    void ParsePath_();                                  // 处理请求路径
    void ParsePost_();                                  // 处理Post事件
    void ParseFromUrlencoded_();                        // 从url种解析编码

    // 从url中解析编码
    static bool UserVerify(const std::string& name, const std::string& pwd, bool
isLogin);  // 用户验证

    PARSE_STATE state_; // 解析状态
    std::string method_, path_, version_, body_;    // 方法，路径，版本，请求体
    std::unordered_map<std::string, std::string> header_;   // 请求头
    std::unordered_map<std::string, std::string> post_;    // post请求

    static const std::unordered_set<std::string> DEFAULT_HTML;  // 默认html
    static const std::unordered_map<std::string, int> DEFAULT_HTML_TAG; // 默认
html标签
    static int ConverHex(char ch);  // 16进制转换为10进制
};

#endif
```

httprequest.cpp

```
#include "httprequest.h"
using namespace std;

// 默认html
const unordered_set<string> HttpRequest::DEFAULT_HTML{
            "/index", "/register", "/login",
             "/welcome", "/video", "/picture", };

// 默认html标签
const unordered_map<string, int> HttpRequest::DEFAULT_HTML_TAG {
            {"/register.html", 0}, {"/login.html", 1},  };

// 初始化
void HttpRequest::Init() {
    method_ = path_ = version_ = body_ = "";
    state_ = REQUEST_LINE;
    header_.clear();
    post_.clear();
}

// 是否保持连接
bool HttpRequest::IsKeepAlive() const {
    if(header_.count("Connection") == 1) {
```

```
        return header_.find("Connection")->second == "keep-alive" && version_ ==
"1.1";
    }
    return false;
}

// 解析处理
bool HttpRequest::parse(Buffer& buff)
{
    const char CRLF[] = "\r\n";      // 行结束符标志(回车换行)
    if(buff.ReadableBytes() <= 0) { // 没有可读的字节
        return false;
    }
    // 读取数据
    while(buff.ReadableBytes() && state_ != FINISH) {
        // 从buff中的读指针开始到读指针结束，这块区域是未读取得数据并去处"\r\n"，返回有
效数据得行末指针
        const char* lineEnd = search(buff.Peek(), buff.BeginWriteConst(), CRLF,
CRLF + 2);
        // 转化为string类型
        std::string line(buff.Peek(), lineEnd);
        switch(state_)
        {
        /*
            有限状态机，从请求行开始，每处理完后会自动转入到下一个状态
        */
        case REQUEST_LINE:
            if(!ParseRequestLine_(line))
            {
                return false;
            }
            ParsePath_();   // 解析路径
            break;
        case HEADERS:
            ParseHeader_(line);
            if(buff.ReadableBytes() <= 2) {
                state_ = FINISH;
            }
            break;
        case BODY:
            ParseBody_(line);
            break;
        default:
            break;
        }
        if(lineEnd == buff.BeginWrite()) { break; } // 读完了
        buff.RetrieveUntil(lineEnd + 2);        // 跳过回车换行
    }
    LOG_DEBUG("[%s], [%s], [%s]", method_.c_str(), path_.c_str(),
version_.c_str());
    return true;
}

// 解析路径
```

```
void HttpRequest::ParsePath_()
{
    if(path_ == "/")
    {
        path_ = "/index.html";
    }
    else
    {
        for(auto &item: DEFAULT_HTML)
        {
            if(item == path_)
            {
                path_ += ".html";
                break;
            }
        }
    }
}

// 解析请求行
bool HttpRequest::ParseRequestLine_(const string& line)
{
    // 正则表达式，匹配请求行，例如: GET /index.html HTTP/1.1
    regex patten("^([^ ]*) ([^ ]*) HTTP/([^ ]*)$");
    smatch subMatch;
    // 在匹配规则中，以括号()的方式来划分组别 一共三个括号 [0]表示整体
    if(regex_match(line, subMatch, patten)) {      // 匹配指定字符串整体是否符合
        method_ = subMatch[1];
        path_ = subMatch[2];
        version_ = subMatch[3];
        state_ = HEADERS;    // 状态转换为下一个状态
        return true;
    }
    LOG_ERROR("RequestLine Error");
    return false;
}

// 解析请求头
void HttpRequest::ParseHeader_(const string& line)
{
    regex patten("^([^:]*): ?(.*)$");
    smatch subMatch;
    //  匹配请求头，例如: Host: www.baidu.com
    if(regex_match(line, subMatch, patten))
    {
        header_[subMatch[1]] = subMatch[2]; // key-value
    }
    else {
        state_ = BODY;  // 状态转换为下一个状态
    }
}

// 解析请求体
void HttpRequest::ParseBody_(const string& line)
```

```
{
    body_ = line;   // 请求体
    ParsePost_();   // 处理post请求
    state_ = FINISH;    // 状态转换为下一个状态
    LOG_DEBUG("Body:%s, len:%d", line.c_str(), line.size());
}

// 16进制转化为10进制
int HttpRequest::ConverHex(char ch)
{
    if(ch >= 'A' && ch <= 'F') return ch -'A' + 10;
    if(ch >= 'a' && ch <= 'f') return ch -'a' + 10;
    return ch;
}

// 处理post请求
void HttpRequest::ParsePost_()
{
    if(method_ == "POST" && header_["Content-Type"] == "application/x-www-form-
urlencoded") {
        ParseFromUrlencoded_();     // POST请求体示例
        if(DEFAULT_HTML_TAG.count(path_))
        { // 如果是登录/注册的path
            int tag = DEFAULT_HTML_TAG.find(path_)->second;
            LOG_DEBUG("Tag:%d", tag);
            if(tag == 0 || tag == 1) {
                bool isLogin = (tag == 1);  // 为1则是登录
                if(UserVerify(post_["username"], post_["password"], isLogin)) {
                    path_ = "/welcome.html";
                }
                else {
                    path_ = "/error.html";
                }
            }
        }
    }
}

// 从url中解析编码
void HttpRequest::ParseFromUrlencoded_() {
    if(body_.size() == 0) { return; }

    string key, value;
    int num = 0;
    int n = body_.size();
    int i = 0, j = 0;

    // 从body中解析键值对
    for(; i < n; i++) {
        char ch = body_[i];
        switch (ch) {
        // key
        case '=': // 键值对连接符
            key = body_.substr(j, i - j);
```

```
                j = i + 1;
                break;
            // 键值对中的空格换为+或者%20
            case '+':
                body_[i] = ' ';
                break;
            case '%':
                num = ConverHex(body_[i + 1]) * 16 + ConverHex(body_[i + 2]);
                body_[i + 2] = num % 10 + '0';
                body_[i + 1] = num / 10 + '0';
                i += 2;
                break;
            // 键值对连接符
            case '&':
                value = body_.substr(j, i - j);
                j = i + 1;
                post_[key] = value;
                LOG_DEBUG("%s = %s", key.c_str(), value.c_str());
                break;
            default:
                break;
            }
        }
        assert(j <= i);
        if(post_.count(key) == 0 && j < i) {
            value = body_.substr(j, i - j);
            post_[key] = value;
        }
    }

    // 用户验证
    bool HttpRequest::UserVerify(const string &name, const string &pwd, bool isLogin)
    {
        if(name == "" || pwd == "") { return false; }
        LOG_INFO("Verify name:%s pwd:%s", name.c_str(), pwd.c_str());
        MYSQL* sql;
        SqlConnRAII(&sql,  SqlConnPool::Instance());    // 获取数据库连接
        assert(sql);    // 断言

        bool flag = false;
        unsigned int j = 0;
        char order[256] = { 0 };
        MYSQL_FIELD *fields = nullptr;  // 字段
        MYSQL_RES *res = nullptr;       // 结果集

        if(!isLogin) { flag = true; }
        /* 查询用户及密码 */
        snprintf(order, 256, "SELECT username, password FROM user WHERE username='%s'
    LIMIT 1", name.c_str());
        LOG_DEBUG("%s", order);

        // 查询
        if(mysql_query(sql, order))
        {
```

```
            mysql_free_result(res); // 释放结果集
            return false;
        }
        res = mysql_store_result(sql);
        j = mysql_num_fields(res);
        fields = mysql_fetch_fields(res);

        while(MYSQL_ROW row = mysql_fetch_row(res)) {
            LOG_DEBUG("MYSQL ROW: %s %s", row[0], row[1]);
            string password(row[1]);
            /* 注册行为 且 用户名未被使用*/
            if(isLogin) {
                if(pwd == password) { flag = true; }
                else {
                    flag = false;
                    LOG_INFO("pwd error!");
                }
            }
            else {
                flag = !(name == row[0]);
                if(flag == false)LOG_INFO("user used!");
            }
        }
        mysql_free_result(res);

        /* 注册行为 且 用户名未被使用*/
        if(!isLogin && flag == true) {
            LOG_DEBUG("regirster!");
            bzero(order, 256);
            snprintf(order, 256,"INSERT INTO user(username, password)
    VALUES('%s','%s')", name.c_str(), pwd.c_str());
            LOG_DEBUG( "%s", order);
            if(mysql_query(sql, order)) {
                LOG_DEBUG( "Insert error!");
                flag = false;
            }
            flag = true;
        }
        // SqlConnPool::Instance()->FreeConn(sql);
        LOG_DEBUG( "UserVerify success!!");
        return flag;
    }

    std::string HttpRequest::path() const{
        return path_;
    }

    std::string& HttpRequest::path(){
        return path_;
    }
    std::string HttpRequest::method() const {
        return method_;
    }
```

```
std::string HttpRequest::version() const {
    return version_;
}

std::string HttpRequest::GetPost(const std::string& key) const {
    assert(key != "");
    if(post_.count(key) == 1) {
        return post_.find(key)->second;
    }
    return "";
}

std::string HttpRequest::GetPost(const char* key) const {
    assert(key != nullptr);
    if(post_.count(key) == 1) {
        return post_.find(key)->second;
    }
    return "";
}
```

3.4.2 http连接

3.4.2.1 httpresponse

httpresponse.h

```
#ifndef HTTP_RESPONSE_H
#define HTTP_RESPONSE_H

#include <unordered_map>
#include <fcntl.h>       // open
#include <unistd.h>      // close
#include <sys/stat.h>    // stat
#include <sys/mman.h>    // mmap, munmap

#include "../buffer/buffer.h"
#include "../log/log.h"

class HttpResponse
{
public:
    HttpResponse(); // 构造函数
    ~HttpResponse();    // 析构函数

    void Init(const std::string& srcDir, std::string& path, bool isKeepAlive =
false, int code = -1);
    void MakeResponse(Buffer& buff);    // 响应
    void UnmapFile();   // 解除映射
    char* File();       // 文件
    size_t FileLen() const; // 文件长度
```

```
    void ErrorContent(Buffer& buff, std::string message);   // 错误内容
    int Code() const { return code_; }  // 编码

private:
    void AddStateLine_(Buffer &buff);
    void AddHeader_(Buffer &buff);
    void AddContent_(Buffer &buff);

    void ErrorHtml_();
    std::string GetFileType_();

    int code_;
    bool isKeepAlive_;

    std::string path_;
    std::string srcDir_;

    char* mmFile_;
    struct stat mmFileStat_;

    static const std::unordered_map<std::string, std::string> SUFFIX_TYPE;  // 后
缀类型集
    static const std::unordered_map<int, std::string> CODE_STATUS;          // 编
码状态集
    static const std::unordered_map<int, std::string> CODE_PATH;            // 编
码路径集
};


#endif //HTTP_RESPONSE_H
```

httpresponse.cpp

```
#include "httpresponse.h"

using namespace std;

// 后缀类型集
const unordered_map<string, string> HttpResponse::SUFFIX_TYPE =
{
    { ".html",  "text/html" },
    { ".xml",   "text/xml" },
    { ".xhtml", "application/xhtml+xml" },
    { ".txt",   "text/plain" },
    { ".rtf",   "application/rtf" },
    { ".pdf",   "application/pdf" },
    { ".word",  "application/nsword" },
    { ".png",   "image/png" },
    { ".gif",   "image/gif" },
    { ".jpg",   "image/jpeg" },
    { ".jpeg",  "image/jpeg" },
    { ".au",    "audio/basic" },
```

```
    { ".mpeg",  "video/mpeg" },
    { ".mpg",   "video/mpeg" },
    { ".avi",   "video/x-msvideo" },
    { ".gz",    "application/x-gzip" },
    { ".tar",   "application/x-tar" },
    { ".css",   "text/css "},
    { ".js",    "text/javascript "},
};

const unordered_map<int, string> HttpResponse::CODE_STATUS = {
    { 200, "OK" },
    { 400, "Bad Request" },
    { 403, "Forbidden" },
    { 404, "Not Found" },
};

const unordered_map<int, string> HttpResponse::CODE_PATH = {
    { 400, "/400.html" },
    { 403, "/403.html" },
    { 404, "/404.html" },
};

HttpResponse::HttpResponse()
{
    code_ = -1;
    path_ = srcDir_ = "";
    isKeepAlive_ = false;
    mmFile_ = nullptr;
    mmFileStat_ = { 0 };
};

HttpResponse::~HttpResponse()
{
    UnmapFile();
}

// 初始化
void HttpResponse::Init(const string& srcDir, string& path, bool isKeepAlive, int
code)
{
    assert(srcDir != "");
    if(mmFile_) { UnmapFile(); }// 如果文件映射到内存，解除映射
    code_ = code;
    isKeepAlive_ = isKeepAlive;
    path_ = path;
    srcDir_ = srcDir;
    mmFile_ = nullptr;
    mmFileStat_ = { 0 };
}

// 生成响应
void HttpResponse::MakeResponse(Buffer& buff)
{
    /* 判断请求的资源文件 */
```

```
    if(stat((srcDir_ + path_).data(), &mmFileStat_) < 0 ||
S_ISDIR(mmFileStat_.st_mode)) {
        code_ = 404;
    }
    // 没有权限
    else if(!(mmFileStat_.st_mode & S_IROTH))
    {
        code_ = 403;
    }
    // 请求成功
    else if(code_ == -1)
    {
        code_ = 200;
    }
    ErrorHtml_();   // 错误处理
    AddStateLine_(buff);    // 添加状态行
    AddHeader_(buff);    // 添加头部
    AddContent_(buff);  // 添加内容
}

// 获取文件
char* HttpResponse::File()
{
    return mmFile_;
}

// 获取文件大小
size_t HttpResponse::FileLen() const
{
    return mmFileStat_.st_size;
}

// 获取文件类型
void HttpResponse::ErrorHtml_()
{
    if(CODE_PATH.count(code_) == 1)
    {
        path_ = CODE_PATH.find(code_)->second;
        stat((srcDir_ + path_).data(), &mmFileStat_);
    }
}

// 添加状态行
void HttpResponse::AddStateLine_(Buffer& buff)
{
    string status;
    if(CODE_STATUS.count(code_) == 1)
    {
        status = CODE_STATUS.find(code_)->second;
    }
    else
    {
        code_ = 400;
        status = CODE_STATUS.find(400)->second;
```

```
    }
    buff.Append("HTTP/1.1 " + to_string(code_) + " " + status + "\r\n");
}

// 添加头部
void HttpResponse::AddHeader_(Buffer& buff)
{
    buff.Append("Connection: ");
    if(isKeepAlive_) {
        buff.Append("keep-alive\r\n");
        buff.Append("keep-alive: max=6, timeout=120\r\n");
    } else{
        buff.Append("close\r\n");
    }
    buff.Append("Content-type: " + GetFileType_() + "\r\n");
}

// 添加内容
void HttpResponse::AddContent_(Buffer& buff)
{
    int srcFd = open((srcDir_ + path_).data(), O_RDONLY);
    if(srcFd < 0)
    {
        ErrorContent(buff, "File NotFound!");
        return;
    }

    //将文件映射到内存提高文件的访问速度  MAP_PRIVATE 建立一个写入时拷贝的私有映射
    LOG_DEBUG("file path %s", (srcDir_ + path_).data());
    int* mmRet = (int*)mmap(0, mmFileStat_.st_size, PROT_READ, MAP_PRIVATE, srcFd,
0);
    if(*mmRet == -1)
    {
        ErrorContent(buff, "File NotFound!");
        return;
    }
    mmFile_ = (char*)mmRet; // 指向映射的内存地址
    close(srcFd); // 关闭文件描述符
    buff.Append("Content-length: " + to_string(mmFileStat_.st_size) + "\r\n\r\n");
}

// 解除文件映射
void HttpResponse::UnmapFile()
{
    if(mmFile_) {
        munmap(mmFile_, mmFileStat_.st_size);
        mmFile_ = nullptr;
    }
}

// 判断文件类型
string HttpResponse::GetFileType_()
{
    string::size_type idx = path_.find_last_of('.');
```

```
    if(idx == string::npos) {    // 最大值 find函数在找不到指定值得情况下会返回
string::npos
        return "text/plain";
    }
    string suffix = path_.substr(idx);
    if(SUFFIX_TYPE.count(suffix) == 1)
    {
        return SUFFIX_TYPE.find(suffix)->second;
    }
    return "text/plain"; // 默认返回
}

// 错误处理
void HttpResponse::ErrorContent(Buffer& buff, string message)
{
    string body;
    string status;
    body += "<html><title>Error</title>";    // html标题
    body += "<body bgcolor=\"ffffff\">";    // html背景颜色
    if(CODE_STATUS.count(code_) == 1)
    {
        status = CODE_STATUS.find(code_)->second;
    }
    else
    {
        status = "Bad Request";
    }
    body += to_string(code_) + " : " + status  + "\n";
    body += "<p>" + message + "</p>";
    body += "<hr><em>TinyWebServer</em></body></html>";

    buff.Append("Content-length: " + to_string(body.size()) + "\r\n\r\n");
    buff.Append(body);
}
```

3.4.2.2 httpconn

httpconn.h

```
#ifndef HTTP_CONN_H
#define HTTP_CONN_H

#include <sys/types.h>
#include <sys/uio.h>     // readv/writev
#include <arpa/inet.h>   // sockaddr_in
#include <stdlib.h>      // atoi()
#include <errno.h>

#include "../log/log.h"
```

```
#include "../buffer/buffer.h"
#include "httprequest.h"
#include "httpresponse.h"
/*
进行读写数据并调用httprequest 来解析数据以及httpresponse来生成响应
*/
class HttpConn {
public:
    HttpConn();
    ~HttpConn();

    void init(int sockFd, const sockaddr_in& addr); // 初始化
    ssize_t read(int* saveErrno);   // 读
    ssize_t write(int* saveErrno);  // 写
    void Close();                   // 关闭
    int GetFd() const;          // 获取文件描述符
    int GetPort() const;    // 获取端口
    const char* GetIP() const;  // 获取IP
    sockaddr_in GetAddr() const;    // 获取地址
    bool process(); // 处理请求

    // 写的总长度
    int ToWriteBytes()
    {
        return iov_[0].iov_len + iov_[1].iov_len;
    }

    bool IsKeepAlive() const {
        return request_.IsKeepAlive();
    }

    static bool isET;
    static const char* srcDir;
    static std::atomic<int> userCount;  // 原子，支持锁

private:

    int fd_;
    struct  sockaddr_in addr_;

    bool isClose_;

    int iovCnt_;
    struct iovec iov_[2];

    Buffer readBuff_; // 读缓冲区
    Buffer writeBuff_; // 写缓冲区

    HttpRequest request_;
    HttpResponse response_;
};

#endif
```

httpconn.cpp

```
#include "httpconn.h"
using namespace std;

const char* HttpConn::srcDir;
std::atomic<int> HttpConn::userCount;
bool HttpConn::isET;

HttpConn::HttpConn()
{
    fd_ = -1;   // 文件描述符
    addr_ = { 0 };  // 地址
    isClose_ = true; // 是否关闭
};

HttpConn::~HttpConn()
{
    Close();
};

// 初始化，传入文件描述符和地址
void HttpConn::init(int fd, const sockaddr_in& addr)
{
    // assert fd > 0,表示文件描述符有效
    assert(fd > 0);
    userCount++;    // 增加用户数
    addr_ = addr;
    fd_ = fd;
    writeBuff_.RetrieveAll();   // 清空写缓冲区
    readBuff_.RetrieveAll();    // 清空读缓冲区
    isClose_ = false;   // 未关闭
    LOG_INFO("Client[%d](%s:%d) in, userCount:%d", fd_, GetIP(), GetPort(),
(int)userCount);
}

// 关闭
void HttpConn::Close()
{
    response_.UnmapFile();
    if(isClose_ == false)
    {
        isClose_ = true;
        userCount--;    // 减少用户数
        close(fd_);
        LOG_INFO("Client[%d](%s:%d) quit, UserCount:%d", fd_, GetIP(), GetPort(),
(int)userCount);
    }
}

int HttpConn::GetFd() const
```

```
{
    return fd_;
};

struct sockaddr_in HttpConn::GetAddr() const
{
    return addr_;
}

const char* HttpConn::GetIP() const
{
    return inet_ntoa(addr_.sin_addr);
}

int HttpConn::GetPort() const
{
    return addr_.sin_port;
}

// 读取数据
ssize_t HttpConn::read(int* saveErrno) {
    ssize_t len = -1;
    do {
        len = readBuff_.ReadFd(fd_, saveErrno);
        if (len <= 0) {
            break;
        }
    } while (isET); // ET:边沿触发要一次性全部读出
    return len;
}

// 主要采用writev连续写函数
ssize_t HttpConn::write(int* saveErrno) {
    ssize_t len = -1;
    do
    {
        len = writev(fd_, iov_, iovCnt_);   // 将iov的内容写到fd中
        if(len <= 0) {
            *saveErrno = errno;
            break;
        }
        if(iov_[0].iov_len + iov_[1].iov_len  == 0) { break; } /* 传输结束 */
        else if(static_cast<size_t>(len) > iov_[0].iov_len)
        {
            // iov_[1].iov_base指向第二个iov的剩余内容
            iov_[1].iov_base = (uint8_t*) iov_[1].iov_base + (len -
iov_[0].iov_len);
            // iov_[1].iov_len更新为剩余长度
            iov_[1].iov_len -= (len - iov_[0].iov_len);

            // 如果iov_[0].iov_len不为0,则表示第一个iov的内容已经全部写入
            if(iov_[0].iov_len)
            {
                writeBuff_.RetrieveAll();
```

```
                iov_[0].iov_len = 0;
            }
        }
        // len <= iov_[0].iov_len
        else
        {
            // 同理，更新iov_[0]的指针和长度，指向未写入的内容
            iov_[0].iov_base = (uint8_t*)iov_[0].iov_base + len;
            iov_[0].iov_len -= len;
            writeBuff_.Retrieve(len);
        }
    } while(isET || ToWriteBytes() > 10240);
    return len;
}

// 判断是否保持连接
bool HttpConn::process()
{
    request_.Init();
    if(readBuff_.ReadableBytes() <= 0)
    {
        return false;
    }
    else if(request_.parse(readBuff_))
    {    // 解析成功
        LOG_DEBUG("%s", request_.path().c_str());
        response_.Init(srcDir, request_.path(), request_.IsKeepAlive(), 200);
    }
    else
    {
        response_.Init(srcDir, request_.path(), false, 400);
    }

    response_.MakeResponse(writeBuff_); // 生成响应报文放入writeBuff_中
    // 响应头
    iov_[0].iov_base = const_cast<char*>(writeBuff_.Peek());
    iov_[0].iov_len = writeBuff_.ReadableBytes();
    iovCnt_ = 1;

    // 文件
    if(response_.FileLen() > 0  && response_.File())
    {
        iov_[1].iov_base = response_.File();
        iov_[1].iov_len = response_.FileLen();
        iovCnt_ = 2;
    }
    LOG_DEBUG("filesize:%d, %d  to %d", response_.FileLen() , iovCnt_,
ToWriteBytes());
    return true;
}
```

3.5 timer

网络编程中除了处理IO事件之外，定时事件也同样不可或缺，如定期检测一个客户连接的活动状态、游戏中的技能冷却倒计时以及其他需要使用超时机制的功能。我们的服务器程序中往往需要处理众多的定时事件，因此有效的组织定时事件，使之能在预期时间内被触发且不影响服务器主要逻辑，对我们的服务器性能影响特别大。

我们的web服务器也需要这样一个时间堆，定时剔除掉长时间不动的空闲用户，避免他们占着耗费服务器资源。

一般的做法是将每个定时事件封装成定时器，并使用某种容器类数据结构将所有的定时器保存好，实现对定时事件的统一管理。常用方法有排序链表、红黑树、时间堆和时间轮。这里使用的是时间堆。

时间堆的底层实现是由小根堆实现的。小根堆可以保证堆顶元素为最小的。

3.5.1 heaptimer.h

```
#ifndef HEAP_TIMER_H
#define HEAP_TIMER_H

#include <queue>
#include <unordered_map>
#include <time.h>
#include <algorithm>
#include <arpa/inet.h>
#include <functional>
#include <assert.h>
#include <chrono>
#include "../log/log.h"

typedef std::function<void()> TimeoutCallBack;
typedef std::chrono::high_resolution_clock Clock;
typedef std::chrono::milliseconds MS;
typedef Clock::time_point TimeStamp;

struct TimerNode {
    int id;
    TimeStamp expires;  // 超时时间点
    TimeoutCallBack cb; // 回调function<void()>
    bool operator<(const TimerNode& t) {    // 重载比较运算符
        return expires < t.expires;
    }
    bool operator>(const TimerNode& t) {    // 重载比较运算符
        return expires > t.expires;
    }
};
class HeapTimer {
public:
    HeapTimer() { heap_.reserve(64); }  // 保留（扩充）容量
    ~HeapTimer() { clear(); }

    void adjust(int id, int newExpires);
```

```
    void add(int id, int timeOut, const TimeoutCallBack& cb);
    void doWork(int id);
    void clear();
    void tick();
    void pop();
    int GetNextTick();

private:
    void del_(size_t i);
    void siftup_(size_t i);
    bool siftdown_(size_t i, size_t n);
    void SwapNode_(size_t i, size_t j);

    std::vector<TimerNode> heap_;
    // key:id value:vector的下标
    std::unordered_map<int, size_t> ref_;   // id对应的在heap_中的下标，方便用heap_
的时候查找
};

#endif //HEAP_TIMER_H
```

3.5.2 heaptimer.cpp

```
#include "heaptimer.h"

void HeapTimer::SwapNode_(size_t i, size_t j) {
    assert(i >= 0 && i <heap_.size());
    assert(j >= 0 && j <heap_.size());
    swap(heap_[i], heap_[j]);
    ref_[heap_[i].id] = i;    // 结点内部id所在索引位置也要变化
    ref_[heap_[j].id] = j;
}

void HeapTimer::siftup_(size_t i) {
    assert(i >= 0 && i < heap_.size());
    size_t parent = (i-1) / 2;
    while(parent >= 0) {
        if(heap_[parent] > heap_[i]) {
            SwapNode_(i, parent);
            i = parent;
            parent = (i-1)/2;
        } else {
            break;
        }
    }
}

// false: 不需要下滑  true: 下滑成功
bool HeapTimer::siftdown_(size_t i, size_t n) {
    assert(i >= 0 && i < heap_.size());
```

```
    assert(n >= 0 && n <= heap_.size());    // n:共几个结点
    auto index = i;
    auto child = 2*index+1;
    while(child < n) {
        if(child+1 < n && heap_[child+1] < heap_[child]) {
            child++;
        }
        if(heap_[child] < heap_[index]) {
            SwapNode_(index, child);
            index = child;
            child = 2*child+1;
        }
        break;  // 需要跳出循环
    }
    return index > i;
}

// 删除指定位置的结点
void HeapTimer::del_(size_t index) {
    assert(index >= 0 && index < heap_.size());
    // 将要删除的结点换到队尾，然后调整堆
    size_t tmp = index;
    size_t n = heap_.size() - 1;
    assert(tmp <= n);
    // 如果就在队尾，就不用移动了
    if(index < heap_.size()-1) {
        SwapNode_(tmp, heap_.size()-1);
        if(!siftdown_(tmp, n)) {
            siftup_(tmp);
        }
    }
    ref_.erase(heap_.back().id);
    heap_.pop_back();
}

// 调整指定id的结点
void HeapTimer::adjust(int id, int newExpires) {
    assert(!heap_.empty() && ref_.count(id));
    heap_[ref_[id]].expires = Clock::now() + MS(newExpires);
    siftdown_(ref_[id], heap_.size());
}

void HeapTimer::add(int id, int timeOut, const TimeoutCallBack& cb) {
    assert(id >= 0);
    // 如果有，则调整
    if(ref_.count(id)) {
        int tmp = ref_[id];
        heap_[tmp].expires = Clock::now() + MS(timeOut);
        heap_[tmp].cb = cb;
        if(!siftdown_(tmp, heap_.size())) {
            siftup_(tmp);
        }
    } else {
        size_t n = heap_.size();
```

```
        ref_[id] = n;
        // 这里应该算是结构体的默认构造?
        heap_.push_back({id, Clock::now() + MS(timeOut), cb});  // 右值
        siftup_(n);
    }
}

// 删除指定id, 并触发回调函数
void HeapTimer::doWork(int id) {
    if(heap_.empty() || ref_.count(id) == 0) {
        return;
    }
    size_t i = ref_[id];
    auto node = heap_[i];
    node.cb();  // 触发回调函数
    del_(i);
}

void HeapTimer::tick() {
    /* 清除超时结点 */
    if(heap_.empty()) {
        return;
    }
    while(!heap_.empty()) {
        TimerNode node = heap_.front();
        if(std::chrono::duration_cast<MS>(node.expires - Clock::now()).count() >
0) {
            break;
        }
        node.cb();
        pop();
    }
}

void HeapTimer::pop() {
    assert(!heap_.empty());
    del_(0);
}

void HeapTimer::clear() {
    ref_.clear();
    heap_.clear();
}

int HeapTimer::GetNextTick() {
    tick();
    size_t res = -1;
    if(!heap_.empty()) {
        res = std::chrono::duration_cast<MS>(heap_.front().expires -
Clock::now()).count();
        if(res < 0) { res = 0; }
    }
    return res;
```

```
}
```

在提供的代码中，`cb` 是一个类型为 `TimeoutCallBack` 的变量。根据代码的上下文，可以猜测 `TimeoutCallBack` 是一个函数指针类型或者是一个可调用对象（比如函数对象、Lambda 表达式等），用于表示定时器到期时需要执行的回调函数。

在 `HeapTimer` 类中，当定时器到期时，会执行存储在定时器结构体中的回调函数 `cb()`。这样可以使得定时器到期时能够执行用户指定的操作，比如通知、处理事件等。

3.6 server

3.6.1 Epoller

Epoll的解释资料:

https://zhuanlan.zhihu.com/p/367591714

https://xiaolincoding.com/os/8_network_system/selete_poll_epoll.html#%E6%9C%80%E5%9F%BA%E6%9C%AC%E7%9A%84-socket-%E6%A8%A1%E5%9E%8B

3.6.1.1 Epoll api接口

```
#include <sys/epoll.h>
// 创建一个新的epoll实例。在内核中创建了一个数据，这个数据中有两个比较重要的数据，一个是需要检测的文件描述符的信息（红黑树），还有一个是就绪列表，存放检测到数据发送改变的文件描述符信息（双向链表）。
int epoll_create(int size);
    - 参数：
        size ： 目前没有意义了。随便写一个数，必须大于0
    - 返回值：
        -1 ： 失败
        > 0 ： 文件描述符，操作epoll实例的

typedef union epoll_data {
    void *ptr;
    int fd;
    uint32_t u32;
    uint64_t u64;
} epoll_data_t;

struct epoll_event {
    uint32_t events; /* Epoll events */
    epoll_data_t data; /* User data variable */
};

// 对epoll实例进行管理：添加文件描述符信息，删除信息，修改信息
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
    - 参数：
        - epfd ： epoll实例对应的文件描述符
```

```
        - op : 要进行什么操作
            EPOLL_CTL_ADD: 添加
            EPOLL_CTL_MOD: 修改
            EPOLL_CTL_DEL: 删除
        - fd : 要检测的文件描述符
        - event : 检测文件描述符什么事情

// 检测函数
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
    - 参数:
        - epfd : epoll实例对应的文件描述符
        - events : 传出参数，保存了发生了变化的文件描述符的信息
        - maxevents : 第二个参数结构体数组的大小
        - timeout : 阻塞时间
            - 0 : 不阻塞
            - -1 : 阻塞，直到检测到fd数据发生变化，解除阻塞
            - > 0 : 阻塞的时长（毫秒）
    - 返回值:
        - 成功，返回发送变化的文件描述符的个数 > 0
        - 失败 -1
```

3.6.1.2 常见的Epoll检测事件

EAGAIN: 我们时常会用一个while(true)死循环去接收缓冲区中客户端socket的连接，如果这个时候我们设置socket状态为非阻塞，那么accept如果在某个时间段没有接收到客户端的连接，因为是非阻塞的IO，accept函数会立即返回，并将errno设置为EAGAIN

下面是一些检测事件:

```
POLLIN : 表示对应的文件描述符可以读（包括对端 SOCKET 正常关闭）；
EPOLLOUT: 表示对应的文件描述符可以写;
EPOLLPRI: 表示对应的文件描述符有紧急的数据可读（这里应该表示有带外数据到来）；
EPOLLERR: 表示对应的文件描述符发生错误;
EPOLLHUP: 表示对应的文件描述符被挂断;
EPOLLET: 将EPOLL设为边缘触发(Edge Triggered)模式，这是相对于水平触发(Level Triggered)来说的。
EPOLLONESHOT: 只监听一次事件，当监听完这次事件之后，如果还需要继续监听这个socket的话，需要再次把这个socket加入到EPOLL队列里。
1. 客户端直接调用close，会触犯EPOLLRDHUP事件
2. 通过EPOLLRDHUP属性，来判断是否对端已经关闭，这样可以减少一次系统调用。
```

3.6.1.3 epoller.h 和 epoller.cpp

epoller.h

```
#ifndef EPOLLER_H
#define EPOLLER_H
```

```
#include <sys/epoll.h> // epoll_ctl()
#include <unistd.h> // close()
#include <assert.h> // assert()
#include <vector>
#include <errno.h>

class Epoller {
public:
    // 构造函数，maxEvent 参数默认为 1024
    explicit Epoller(int maxEvent = 1024);

    // 析构函数
    ~Epoller();

    // 添加文件描述符到 epoll 实例
    bool AddFd(int fd, uint32_t events);

    // 修改文件描述符的事件
    bool ModFd(int fd, uint32_t events);

    // 删除文件描述符
    bool DelFd(int fd);

    // 等待事件发生
    int Wait(int timeoutMs = -1);

    // 获取事件的文件描述符
    int GetEventFd(size_t i) const;

    // 获取事件的属性
    uint32_t GetEvents(size_t i) const;

private:
    int epollFd_; // epoll 实例的文件描述符
    std::vector<struct epoll_event> events_; // 存储事件的容器
};

#endif // EPOLLER_H
```

epoller.cpp

```
#include "epoller.h"

// 构造函数
Epoller::Epoller(int maxEvent):epollFd_(epoll_create(512)), events_(maxEvent){
    // 确保 epoll 文件描述符有效，并且事件向量的大小为正数
    assert(epollFd_ >= 0 && events_.size() > 0);
}

// 析构函数
Epoller::~Epoller() {
```

```
    // 关闭 epoll 文件描述符
    close(epollFd_);
}

// 添加文件描述符到 epoll 实例
bool Epoller::AddFd(int fd, uint32_t events) {
    // 检查文件描述符是否有效
    if(fd < 0) return false;
    // 初始化 epoll 事件结构体
    epoll_event ev = {0};
    ev.data.fd = fd;
    ev.events = events;
    // 将文件描述符添加到 epoll 实例
    return 0 == epoll_ctl(epollFd_, EPOLL_CTL_ADD, fd, &ev);
}

// 修改文件描述符的事件
bool Epoller::ModFd(int fd, uint32_t events) {
    // 检查文件描述符是否有效
    if(fd < 0) return false;
    // 初始化 epoll 事件结构体
    epoll_event ev = {0};
    ev.data.fd = fd;
    ev.events = events;
    // 修改文件描述符的事件
    return 0 == epoll_ctl(epollFd_, EPOLL_CTL_MOD, fd, &ev);
}

// 删除文件描述符
bool Epoller::DelFd(int fd) {
    // 检查文件描述符是否有效
    if(fd < 0) return false;
    // 删除文件描述符
    return 0 == epoll_ctl(epollFd_, EPOLL_CTL_DEL, fd, 0);
}

// 等待事件发生
int Epoller::Wait(int timeoutMs) {
    // 调用 epoll_wait 函数等待事件发生
    return epoll_wait(epollFd_, &events_[0], static_cast<int>(events_.size()),
timeoutMs);
}

// 获取事件的文件描述符
int Epoller::GetEventFd(size_t i) const {
    // 确保索引在有效范围内
    assert(i < events_.size() && i >= 0);
    // 返回事件的文件描述符
    return events_[i].data.fd;
}

// 获取事件的属性
uint32_t Epoller::GetEvents(size_t i) const {
    // 确保索引在有效范围内
```

```
    assert(i < events_.size() && i >= 0);
    // 返回事件的属性
    return events_[i].events;
}
```

3.6.2 webserver

webserver.h

```
#ifndef WEBSERVER_H
#define WEBSERVER_H

#include <unordered_map>
#include <fcntl.h>       // fcntl()
#include <unistd.h>      // close()
#include <assert.h>
#include <errno.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include "epoller.h"
#include "../timer/heaptimer.h"

#include "../log/log.h"
#include "../pool/sqlconnpool.h"
#include "../pool/threadpool.h"

#include "../http/httpconn.h"

class WebServer {
public:
    WebServer(
        int port, int trigMode, int timeoutMS, bool OptLinger,
        int sqlPort, const char* sqlUser, const  char* sqlPwd,
        const char* dbName, int connPoolNum, int threadNum,
        bool openLog, int logLevel, int logQueSize); // 构造函数: 设置服务器参数
+   初始化定时器 / 线程池 / 反应堆 / 连接队列


    ~WebServer(); //析构函数: 关闭listenFd_,   销毁  连接队列/定时器 / 线程池 / 反应堆
    void Start();
    /*
    创建端口, 绑定端口, 监听端口,   创建epoll反应堆,   将监听描述符加入反应堆

    等待事件就绪

    连接事件 - - > DealListen()

    写事件 - - > DealWrite()
```

```
    读事件 - - > DealRead()

    事件处理完毕，修改反应堆，再跳到 2 处循环执行
    */

private:
    bool InitSocket_();  // 初始化Socket连接
    void InitEventMode_(int trigMode); // 初始化事件模式
    void AddClient_(int fd, sockaddr_in addr);

    void DealListen_(); // 新初始化一个HttpConnection对象
    void DealWrite_(HttpConn* client); // DealWrite:   对应连接对象进行处理 - - > 若处理成功，则监听事件转换成  读  事件
    void DealRead_(HttpConn* client); // DealRead:    对应连接对象进行处理 - - > 若处理成功，则监听事件转换成  写  事件

    void SendError_(int fd, const char*info);
    void ExtentTime_(HttpConn* client);
    void CloseConn_(HttpConn* client);

    void OnRead_(HttpConn* client);
    void OnWrite_(HttpConn* client);
    void OnProcess(HttpConn* client);

    static const int MAX_FD = 65536;

    static int SetFdNonblock(int fd);

    int port_;
    bool openLinger_; // 优雅关闭选项
    int timeoutMS_;  /* 毫秒MS */  // 定时器的默认过期时间
    bool isClose_; // 服务启动标志
    int listenFd_; // 监听文件描述符
    char* srcDir_; // 需要获取的路径

    uint32_t listenEvent_;  // 监听事件 初始监听描述符监听设置
    uint32_t connEvent_;    // 连接事件 初始连接描述符监听设置

    std::unique_ptr<HeapTimer> timer_; // 定时器
    std::unique_ptr<ThreadPool> threadpool_; // 线程池
    std::unique_ptr<Epoller> epoller_; // 反应堆
    std::unordered_map<int, HttpConn> users_; //连接队列
};

#endif //WEBSERVER_H
```

3.7 main.cpp

```
#include <unistd.h>
#include "server/webserver.h"

int main()
{
    // 守护进程 后台运行
    WebServer server(
        1316, 3, 60000, false,             /* 端口 ET模式 timeoutMs 优雅退出  */
        3306, "root", "Zlx0613@", "webserver", /* Mysql配置 */
        12, 6, true, 1, 1024);             /* 连接池数量 线程池数量 日志开关 日志等级
日志异步队列容量 */
    server.Start();
}
```

4. 后续改进

cookie(unfinished)

redis(unfinished)

5. reference:

https://github.com/zgzhhw/WebServer https://blog.csdn.net/weixin_51322383/article/details/130464403