项目1：基于Linux的多线程服务器开发

Q1: 采用线程池实现多线程，并采用Epoll技术实现I/O多路复用，采用Reactor事件处理模式?

1. **select/poll/epoll的区别?**

   select：*1）每次调用select，都需要把fd集合从用户态拷贝到内核态，这个开销在fd很多时会很大（2）同时每次调用select都需要在内核遍历传递进来的所有fd，这个开销在fd很多时也很大。*

   poll：poll的实现和select非常相似，只是描述fd集合的方式不同，poll使用pollfd结构而不是select的fd_set结构，其他的都差不多,管理多个描述符也是进行轮询，根据描述符的状态进行处理，但是poll没有最大文件描述符数量的限制。它没有最大连接数的限制，原因是它是基于链表来存储的，但是同样有一个缺点：***1、大量的fd的数组被整体复制于用户态和内核地址空间之间，而不管这样的复制是不是有意义。 2、poll还有一个特点是"水平触发"，如果报告了fd后，没有被处理，那么下次poll时会再次报告该fd。***

   epoll：epoll有EPOLL LT和EPOLL ET两种触发模式，LT是默认的模式，ET是"高速"模式。LT模式下，只要这个fd还有数据可读，每次 epoll_wait都会返回它的事件，提醒用户程序去操作，而在ET（边缘触发）模式中，它只会提示一次，直到下次再有数据流入之前都不会再提示了，无 论fd中是否还有数据可读。所以在ET模式下，read一个fd的时候一定要把它的buffer读光，也就是说一直读到read的返回值小于请求值，或者 遇到EAGAIN错误。还有一个特点是，epoll使用"事件"的就绪通知方式，通过epoll_ctl注册fd，一旦该fd就绪，内核就会采用类似callback的回调机制来激活该fd，epoll_wait便可以收到通知。

   没有最大并发连接的限制，能打开的FD的上限远大于1024（1G的内存上能监听约10万个端口）；2、效率提升，不是轮询的方式，不会随着FD数目的增加效率下降。只有活跃可用的FD才会调用callback函数； 即Epoll最大的优点就在于它只管你"活跃"的连接，而跟连接总数无关，因此在实际的网络环境中，Epoll的效率就会远远高于select和poll。 3、 内存拷贝，利用mmap()文件映射内存加速与内核空间的消息传递；即epoll使用mmap减少复制开销。

2. **Proactor和reactor的区别?**

   异步 I/O 好比，你让饭堂阿姨将菜做好并把菜打到饭盒里后，把饭盒送到你面前，整个过程你都不需要任何等待。**很明显，异步 I/O 比同步 I/O 性能更好，因为异步 I/O 在「内核数据准备好」和「数据从内核空间拷贝到用户空间」这两个过程都不用等待。**Proactor 正是采用了异步 I/O 技术，所以被称为异步网络模型。现在我们再来理解 Reactor 和 Proactor 的区别，就比较清晰了。***Reactor* 是非阻塞同步网络模式，感知的是就绪可读写事件。在每次感知到有事件发生（比如可读就绪事件）后，就需要应用进程主动调用 *read* 方法来完成数据的读取，也就是要应用进程主动将 *socket* 接收缓存中的数据读到应用进程内存中，这个过程是同步的，读取完数据后应用进程才能处理数据。*Proactor* 是异步网络模式， 感知的是已完成的读写事件。在发起异步读写请求时，需要传入数据缓冲区的地址（用来存放结果数据）等信息，这样系统内核才可以自动帮我们把数据的读写工作完成，这里的读写工作全程由操作系统来做，并不需要像 *Reactor* 那样还需要应用进程主动发起 *read/write* 来读写数据，操作系统完成读写工作后，就会通知应用进程直接处理数据。 因此，*Reactor* 可以理解为「来了事件操作系统通知应用进程，让应用进程来处理」，而 *Proactor* 可以理解为「来了事件操作系统来处理，处理完再通知应用进程」。**这里的「事件」就是有新连接、有数据可读、有数据可写的这些 I/O 事件这里的「处理」包含从驱动读取到内核以及从内核读取到用户空间。举个实际生活中的例子，Reactor 模式就是快递员在楼下，给你打电话告诉你快递到你家小区了，你需要自己下楼来拿快递。而在 Proactor 模式下，快递员直接将快递送到你家门口，然后通知你。无论是 Reactor，还是 Proactor，都是一种基于「事件分发」的网络编程模式，区别在于 Reactor 模式是基于「待完成」的 I/O 事件，而 Proactor 模式则是基于「已完成」的 I/O 事件。

Q2:文件的读写缓冲区采用vector容器封装，实现缓冲区大小的自动增长

建立二级备用缓存，根据写指针确定最终的容器大小。涉及到扩大容量、移动数据的过程。类似于vector底层的扩容机制

Q3:利用状态机和正则表达式解析HTTP请求报文，并生成HTTP响应报文，可以实现GET/POST两种请求解析

1. **使用有限状态机解析http的过程**

   使用状态机完成对HTTP请求的读取和分析

   主状态机作用(parse_content())：读取请求行与头部行数据
   状态：请求行、头部行
   从状态机作用(parse_line)：读取每一行数据
   状态：读取成功、失败、需要继续读取
   parse_request():读取请求行，并将主状态机状态更改为头部行
   parse_header():读取头部行

   主状态机初始状态为请求行，
   循环调用parse_line，若为成功状态继续（失败退出返回失败原因）
   并且判断主状态机状态
   请求->parse_request()->继续循环（失败退出返回失败原因）
   头部->parse_header()->成功返回好的请求结果（失败退出返回失败原因）

2. **get/post请求的区别?**

   GET与POST的请求区别 【面试题】
   1.传参区别
   GET请求传参会将表单中的信息使用URL拼接的方法显示到浏览器的地址栏中，不安全。
   post传参不会暴露信息
   GET方法属于URL参数传递信息 以?进行拼接，多个参数使用&形式连接
   post会将信息放在请求主体中，这样的信息是不会暴露的。
   2.长度限制
   get传参有长度限制，超过限制后直接报错。60KB
   post传参长度不受限制。
   3.缓存
   get方式传递参数浏览器会直接缓存数据，无论多重要的信息浏览器都会缓存。
   POST方式浏览不会直接缓存到，比较安全。
   Post：安全 幂等
   get唯一的好处就是比post快。

   浏览器其他的请求方式

HTTP协议的8种请求类型介绍

HTTP协议的8种请求类型介绍

HTTP协议中共定义了八种方法或者叫"动作"来表明对Request-URI指定的资源的不同操作方式，具体介绍如下：

OPTIONS：返回服务器针对特定资源所支持的HTTP请求方法。也可以利用向Web服务器发送'*'的请求来测试服务器的功能性。
HEAD：向服务器索要与GET请求相一致的响应，只不过响应体将不会被返回。这一方法可以在不必传输整个响应内容的情况下，就可以获取包含在响应消息头中的元信息。
GET：向特定的资源发出请求。
POST：向指定资源提交数据进行处理请求（例如提交表单或者上传文件）。数据被包含在请求体中。POST请求可能会导致新的资源的创建和/或已有资源的修改。
PUT：向指定资源位置上传其最新内容。
DELETE：请求服务器删除Request-URI所标识的资源。
TRACE：回显服务器收到的请求，主要用于测试或诊断。

CONNECT：HTTP/1.1协议中预留给能够将连接改为管道方式的代理服务器。

虽然HTTP的请求方式有8种，但是我们在实际应用中常用的也就是get和post，其他请求方式也都可以通过这两种方式间接的来实现。

3. http状态码

> HTTP 常见的状态码，有哪些?

五大类 HTTP 状态码

	具体含义	常见的状态码
1××	提示信息，表示目前是协议处理的中间状态，还需要后续的操作；	
2××	成功，报文已经收到并被正确处理；	200、204、206
3××	重定向，资源位置发生变动，需要客户端重新发送请求；	301、302、304
4××	客户端错误，请求报文有误，服务器无法处理；	400、403、404
5××	服务器错误，服务器在处理请求时内部发生了错误。	500、501、502、503

Q4:定时器是由底层用vector实现的小根堆来实现，用来关闭超时的非活动连接

二、时间堆

1. 概念简述

由于定时器的触发是由于时间到了，因此只有时间最短的定时器会首先被触发，通过这个原理，我们可以采用最小堆，将按时间顺序排序，堆顶元素是时间最短的定时器，因此只要判断堆顶元素是否被触发即可。只有堆顶定时器的时间到了，才会到其他时间较晚的定时器的时间。

2. 实现细节

- 堆顶节点的删除：将堆顶节点删除，就会留有一个空位置，因此可以将最后一个节点放到堆顶位置，再对堆顶节点进行下滤，就可以确保构成最小堆。
- 使用数组来模拟堆的实现，相比于链表而言，不仅节省空间，而且更容易实现堆的插入、删除操作。如上面图片中的最小堆，可以用数组表示为：
- 由于非叶子结点有`N/2 - 1`个，因此只要保证这些节点构成的子树具有堆性质，就能保证整棵树具有堆性质。（因为非叶子结点的下滤会将叶子节点也变的具有堆性质）

Q5: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

1.**单例模式的两种实现：**

```
// 饿汉式
//singleton.h
// version 1.3
class Singleton
{
private:
    static Singleton instance;
private:
    Singleton();
    ~Singleton();
    Singleton(const Singleton&);
    Singleton& operator=(const Singleton&);
public:
    static Singleton& getInstance() {
        return instance;
    }
};

// initialize defaultly
Singleton Singleton::instance;


// 懒汉式
//singleton.h
class Singleton
{
private:
    Singleton() { };
    ~Singleton() { };
    Singleton(const Singleton&);
    Singleton& operator=(const Singleton&);
public:
    static Singleton& getInstance()
        {
        static Singleton instance;
        return instance;
    }
};
```

2.**异步日志系统?**

异步日志，按我的理解就是主线程的日志打印接口仅负责生产日志数据（作为日志的生产者），而日志的落地操作留给另一个后台线程去完成（作为日志的消费者），这是一个典型的生产-消费问题，如此一来会使得：

主线程调用日志打印接口成为非阻塞操作，同步的磁盘IO从主线程中剥离出来，有助于提高性能

对于异步日志，我们很容易借助队列来一个实现方式：主线程写日志到队列，队列本身使用条件变量、或者管道、eventfd等通知机制，当有数据入队列就通知消费者线程去消费日志。

简单说 主线程接受数据，放到队列中，子线程从队列中取数据，往文件中写。队列用mutex和condition_variable做同步。其实就是简单的一生产者一消费者模型。子线程处理能力不足的话可以简单增加线程数量，扩展成一生产者多消费者模型。

Q6:利用RAII机制实现了数据库连接池，用来实现用户注册登录功能

1.**什么是RAII机制?**

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。

C++面向对象编程时，创建变量会执行构造函数，销毁对象时执行析构函数，若进程堆资源在构造函数中获取，在析构函数中释放资源，即实现了RAII，理论上该技术可以以用于任何的系统资源。

在使用时通过调用GetInstance()生成数据库连接池，将数据库连接池永恒放在静态栈中，代码如下所示：

```
m_connPool = connection_pool::GetInstance();
m_connPool->init("localhost", m_user, m_passWord, m_databaseName, 3306,
m_sql_num, m_close_log);
//初始化数据库读取表
users->initmysql_result(m_connPool);
```

当程序出该作用域时mysqlcon函数自动析构，释放该条数据库的连接。

```
connectionRAII mysqlcon(&request->mysql, m_connPool);
request->process();
```

```
// SqlHandler.h
#include "SqlConnPool.h"

class SqlHandler {
public:
    SqlHandler(SqlConnPool *connpool);
    SqlHandler(MYSQL **sql, SqlConnPool *connpool);
    ~SqlHandler();

private:
    MYSQL *m_sql;
    SqlConnPool *m_connpool;
};

// SqlHandler.cpp
#include "SqlHandler.h"

SqlHandler::SqlHandler(SqlConnPool *connpool) {
    if (connpool) {
        m_sql = connpool->GetConnObj();
        m_connpool = connpool;
    }
}
```

```
SqlHandler::SqlHandler(MYSQL **sql, SqlConnPool *connpool) {
    if (connpool) {
        *sql = connpool->GetConnObj();
        m_sql = *sql;
        m_connpool = connpool;
    }
}

SqlHandler::~SqlHandler() {
    if (m_sql) {
        m_connpool->FreeConnObj(m_sql);
    }
}
```

项目2：基于Linux的网络通信引擎架构与实现

Q1:建立服务端和客户端通信的框架。基于select多路复用技术，使用C++TCP进行网络通信，实现了一套支持高并发的网络通信引擎

Q2:为客户端加入线程、实现跨平台、处理粘包、服务端和客户端升级为生产者消费者模式

手写生产者消费者模式

```
// 版本1
# include<iostream>
using namespace std;
# include<thread>
# include<vector>
# include<mutex>
# include<condition_variable>
# include<queue>
#include <windows.h>
//生产者数量
# define PRODUCT_SIZE 20
//消费者数量
# define CUSTOMER_SIZE 1
//最大产品数量
#define MAX_SIZE 10
//互斥锁
mutex mut;
//条件变量
condition_variable con;
//队列，模拟缓冲区
queue<int> que;
void Producter()
{

    while (true)
    {
        Sleep(10);
```

```
        std::unique_lock <std::mutex> lck(mut);
        while(que.size()> MAX_SIZE)
        {

            con.wait(lck);
        }
        int data = rand();
         que.push(data);
        cout << this_thread::get_id() << "生产了产品：" << data<<endl;
        con.notify_all();

    }
}
void Customer()
{
    while (true)
    {
        std::unique_lock <std::mutex> lck(mut);
        while (que.empty())
        {
            con.wait(lck);
        }
        cout << this_thread::get_id() << "消费了产品：" << que.front() << endl;
        que.pop();
        con.notify_all();
    }
}
int main()
{
    vector<thread> threadPoll;
    //创建生产者和消费者
    for (int i = 0; i < PRODUCT_SIZE; ++i)
    {
        threadPoll.push_back(thread(Producter));
    }
    for (int i = 0; i < PRODUCT_SIZE+ CUSTOMER_SIZE; ++i)
    {
        threadPoll.push_back(thread(Customer));
    }


    for (int i = 0; i < PRODUCT_SIZE + CUSTOMER_SIZE; ++i)
    {
        threadPoll[i].join();
    }

    return 0;
}

// 版本2
#include "stdafx.h"
#include <thread>
#include <mutex>
#include <deque>
#include <vector>
#include <condition_variable>

class CThreadDemo
```

```
{
private:
    std::deque<int> m_data;
    std::mutex m_mtx; // 全局互斥锁.
    std::condition_variable m_cv; // 全局条件变量.
    int       m_nGen;

private:
    void ProductThread(){
        while (true){
            std::unique_lock <std::mutex> lck(m_mtx);
            m_nGen = ++m_nGen % 1000;
            printf("product %d\n", m_nGen);
            m_data.push_back(m_nGen);
            lck.unlock();
            m_cv.notify_all();

            /* 等待1S */
            std::chrono::milliseconds dura(1000);
            std::this_thread::sleep_for(dura);
        }
    }

    void ConsumeThread(){
        while (true){
            std::unique_lock <std::mutex> lck(m_mtx);
            while (m_data.empty()){
                m_cv.wait(lck);
            }
            int nData = m_data.front();
            m_data.pop_front();
            printf("consume %d\n", nData);
            lck.unlock();

            /* 等待2S */
            std::chrono::milliseconds dura(2000);
            std::this_thread::sleep_for(dura);
        }
    }
public:
    CThreadDemo(){
        m_data.clear();
        m_nGen = 0;
    }

    void Start(){
        std::vector<std::thread> threads;
        threads.clear();
        for (int i = 0; i < 5; i++){/* 生产者线程 */
            threads.push_back(std::thread(&CThreadDemo::ProductThread, this));
        }
        for (int i = 5; i < 10; i++){/* 消费者线程 */
            threads.push_back(std::thread(&CThreadDemo::ConsumeThread, this));
        }
        for (auto& t : threads){/* 等待所有线程的退出 */
            t.join();
        }
    }
```

```
};


int _tmain(int argc, _TCHAR* argv[])
{
    CThreadDemo test;
    test.Start();
    return 0;
}
```

Q3:解决收发数据的瓶颈、添加发送缓冲区以实现定时定量发送数据、添加心跳检测和简易日志系统

Q4: 对服务器进行优化。使用内存池技术减少内存碎片的产生，使用对象池技术避免创建和删除大量对象，以支持程序长效稳定运行；同时将服务端升级为epoll来处理多客户端