自我介绍：

熟练的编写c++代码，数据结构，基于linux环境的网络编程操作系统，

包括 vim和gdb调试，VSCode 上使用git进行版本控制;

对 Oracle 数据库也有了解，熟悉 SQL 语言，熟悉 PL/SQL Developer 工具，powerdesigner创建表格

追觅已成为智能生活家电领域多品类立体布局的全球化科技公司，先后推出了扫地机器人、智能洗地机、无线吸尘器、高速吹风机四大品类，并持续拓新。目前产品覆盖包括中国、 美国、德国、法国、韩国等120余个国家和地区。

在高速数字马达、智能算法、流体力学及机器人技术等方面，追觅拥有一系列授权专利并处于世界领先地位。其中，“心脏”高速数字马达、“大脑”智能算法和设计是追觅的核心竞争力，也是追觅产品矩阵迭代拓新的核心驱动力

项目介绍：

此项目是基于Linux的轻量级多线程Reactor服务器，利用多路IO复用，可以同时监听多个请求，使用线程池处理请求，主线程负责监听，监听有事件之后，从socket中循环读取数据，然后将读取到的数据封装成一个请求对象放入队列。睡眠在请求队列上的工作线程被唤醒进行处理，使用状态机解析请求报文，实现同步/异步日志系统，记录服务器运行状态，并对系统进行了压力测试。

项目难点：

1、如何提高服务器的并发能力

2、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

3、多线程并发的情况下，保证线程的同步

问题

Visual stdio 调试没问题，在liunux下编译不通过

g++默认不支持c++11，std=c++11

不运行的时候可以调试，运行起来调试Missing separate debuginfos

水平触发模式

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。适用于并发量小的情况

边缘触发模式

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。必须要一次性将数据读取完。ET适用于并发量大的情况。

epoll 是如何实现的

它是通过红黑树和双向链表实现的。 首先通过epoll\_create()创建一个指示epoll内核事件表的文件描述符，其中包括红黑树根节点和双向链表头节点。然后通过epoll\_ctl()向epoll对象的红黑树结构中添加、删除、修改感兴趣的事件，返回0标识成功，返回-1表示失败。最后通过epoll\_wait()系统调用判断双向链表是否为空，如果为空则阻塞。当文件描述符状态改变，fd上的回调函数被调用，该函数将fd加入到双向链表中，此时epoll\_wait函数被唤醒，返回就绪好的事件。

了解C/S模型

当程序执行到recv时，操作系统会将进程 A 从工作队列移动到该socket的等待队列中。工作队列只剩下了进程 B 和 C，依据进程调度，CPU会轮流执行这两个进程的程序，不会执行进程 A 的程序。所以进程 A 被阻塞，不会往下执行代码，也不会占用CPU资源。当socket接收到数据后，操作系统将该socket等待队列上的进程重新放回到工作队列，该进程变成运行状态，继续执行代码。同时由于socket的接收缓冲区已经有了数据，recv可以返回接收到的数据。（就是阻塞的具体实现）

为什么要使用select模型？

答：解决基本C/S模型中，accept()、recv()、send()阻塞的问题

* C/S模型中accept()会阻塞一直傻等socket来链接
* select模型只解决accept()傻等的问题，不解决recv(),send()执行阻塞问题

select模型逻辑

1. 将所有的socket(服务器端+客户端)装进一个数组中
2. 有唤醒
3. 通过select()遍历socket数组
4. 取出有相应的socket放进另一个数组（都是有响应的socket）
5. 对装有响应的socket数组集中处理

使用CS模型时，当链接了一个客户端，执行完了recv，while循环又回到了accept()，傻等着客户端来链接，无法多客户端链接通信。

使用select模型时，把进程A分别加入所有需要监视的socket的等待队列中。当有一个socket接收了数据，进程A被唤醒，程序并不知道哪些socket收到数据，还需要遍历一次。

有缺点，主要是：每次调用select都需要将进程加入到所有监视socket的等待队列，每次唤醒都需要从每个队列中移除。这里涉及了两次遍历，而且每次都要将整个 FDS 列表传递给内核，有一定的开销。

Epoll模型逻辑

epoll 的使用非常简单，只有下面 3 个系统调用。

epollcreate  负责创建一个监控和管理句柄 fd 的池子；

epollctl  负责管理这个池子里的 fd 增、删、改；

epollwait  负责打盹的，让出 CPU 调度，但是只要有“事”，立马会从这里唤醒；

1.创建epoll 池，内部实现用红黑树的结构体来管理这些注册进程来的句柄 fd。

2.int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event); event 结构体指定监听事件类型，读写或者其他。

3.通过回调，一旦fd读写（event设定的读写）就绪了，请底层硬件自动把这个 fd 相关的结构体放到放入就绪队列中，并且把 epoll 从 epoll\_wait 出唤醒。

Linux 下另外两种典型的 fd ，常常也会放到 epoll 池里。

eventfd：eventfd 实现非常简单，故名思义就是专门用来做事件通知用的。使用系统调用 eventfd 创建，这种文件 fd 无法传输数据，只用来传输事件，常常用于生产消费者模式的事件实现；

T imerfd：这是一种定时器 fd，使用 timerfd\_create 创建，到时间点触发可读事件；

Select、Poll与Epoll的区别？

内核实现机制的不同

select模型，使用一个数组保存所有的文件描述符，内核需要在每次调用时遍历整个数组，检查每个文件描述符的状态。当文件描述符数量较多时，性能会大幅下降（时间复杂度为O(n)），因为每次调用都会扫描整个列表。

epoll模型，采用了事件驱动机制，内核会通过回调机制维护一个就绪事件列表，只返回已经就绪的文件描述符。用户态和内核态之间的交互只涉及就绪的文件描述符。

效率的提升

水平触发（LT）与边缘触发（ET）

epoll支持边缘触发模式（ET），相比传统的水平触发（LT），可以减少重复的事件通知和系统调用次数，从而提高性能。select不支持类似的模式，只能处理水平触发。

事件管理机制

select模型，每次调用都需要重新传入所有需要监控的文件描述符，重复构造监控集合。

epoll模型，文件描述符通过epoll\_ctl进行注册，之后无需重复传递监控集合，内核会自动维护，从而减少了不必要的系统调用。

epoll通过高效的事件通知机制、支持大规模文件描述符管理、以及更灵活的触发模式，显著提升了性能，尤其在高并发场景下优势更加明显。相比之下，select由于设计较为简单，更适合小规模、低并发的场景。

Select、Poll与Epoll的区别？

文件描述符数量

select通过线性表描述文件描述符集合，文件描述符有上限，一般是1024

epoll通过红黑树描述，最大可以打开文件的数目，可以通过命令修改。

将文件描述符从用户传给内核

select和poll通过将所有文件描述符拷贝到内核态，每次调用都需要拷贝

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内核判断就绪的文件描述符

select 和 poll 每次调用都会对连接进行线性遍历，所以随着FD的增加会造成遍历速度慢的“线性下降性能问题”

epoll 因为epoll内核实现是根据每个fd上的callback函数来实现的，只有活跃的socket才会主动调用callback,所以在活跃socket较少的情况下，使用epoll没有前面两者的线性下降的性能问题，但是所有socket都很活跃，可能有性能问题。

应用程序索引就绪文件描述符

select/poll只返回发生了事件的文件描述符的个数，若知道是哪个发生了事件，同样需要遍历

epoll返回的发生了事件的个数和结构体数组，结构体包含socket的信息，因此直接处理返回的数组即可

工作模式

select和poll都只能工作在相对低效的LT模式下，epoll则可以工作在ET高效模式，

应用场景

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

当监测的fd数目非常大，成千上万，且单位时间只有其中的一部分fd处于就绪状态，这个时候使用epoll能够明显提升性能

并发：

在处理这个请求的同时，还需要继续监听其他客户的请求并分配其另一逻辑单元来处理。通过epoll 这种I/O复用技术来实现对监听socket和连接socket的同时监听。通过多线程并发，用线程池来实现并发，为每个就绪的文件描述符分配一个逻辑单元(线程)来处理。

为什么ET模式下一定要设置非阻塞？

因为边沿触发模式下是无限循环读，直到出现错误，读取错误表示socket为空，不用再读了，然后就停止循环了。如果是阻塞，循环读在socket为空的时候就会阻塞到那里，主线程的read（）函数一旦阻塞住，当再有其他监听事件过来就没办法读了，给其他事情造成了影响，所以必须要设置为非阻塞。

在读取数据的时候怎么知道读取完了呢？

ET模式下是无限循环读，直到read函数返回值为-1且函数调用发生系统错误标志为EAGAIN或者EWOULDBLOCK，这两个错误表示socket为空，不用再读了，然后就停止循环了。

怎么知道客户端连接断开了？

1. 客户端断开会触发epoll的读事件，epoll\_wait唤醒之后，处理返回的事件列表中可以发现断开的客户端。
2. 当处理客户端发送的信号，如果read函数的返回值为0，客户端断开。（返回值）

//新客户端一旦连接，会被添加到epoll中。一旦断开连接，epoll\_wait唤醒，会有一个//数组，存储了活跃的socket相关的结构体，结构体中包含了socket相关的数据

什么时候用ET，什么时候用LT？

LT适用于并发量小的情况，ET适用于并发量大的情况。ET在通知用户之后，就会将fd从就绪链表中删除，而LT不会，它会一直保留，这就会导致随着fd增多，就绪链表越大，每次都要从头开始遍历找到对应的fd，所以并发量越大效率越低。ET因为会删除所以效率比较高。

什么是线程池，为什么使用线程池？

在处理外部的某种请求的时候，一般我们都是来了任务，我们就会去创建一个线程，然后让线程去处理任务——这种方式是没有问题的！但是来了任务才去创建线程！这样的后果就是会让我们的效率降低！因为创建线程也是有成本的也要花时间和花空间！

内存池的存在就是尽量的减少系统调用，一次性创建足够的线程，预先把线程创建好，当有任务的时候，这些线程再去竞争这些任务。

线程就是没有任务的时候休眠，有任务的时候唤醒，唤醒线程成本相比创建线程成本更低！

任务从外部传过来，服务端将收到的任务push到任务队列里面就不管了，让线程们去处理任务

任务队列+一堆的线程就是线程池——其实这个模型就是典型的生产消费模型

这就是池化技术的本质——预先使用更大的成本，先申请一部分资源！当我们要用的时候能够直接拿出来！

线程池还有一个优势——就是可以控制线程总数。如果我们控制了线程池总数只有4个线程！不管有多少任务到来，永远都是4个线程线程。可以让过多的客户端去等待，从而维护服务器的稳定。

主线程选择哪个子线程来为新任务服务方式：

随机算法和轮流选取算法。

主进程和所有子进程通过一个工作队列(list 单链表)来同步，子线程都睡眠在该工作队列上。

线程池实现细节？

线程池，就是一个pthread\_t类型的普通数组，通过lambda函数直接向数组中添加指定个数的线程，通过detach()将线程设置成脱离态后，当这一线程运行结束时，它的资源会被系统自动回收，而不再需要在其它线程中对其进行 join() 操作。 消息队列的大小由机器硬件来决定，本实验环境选取10000

I/O操作的速度远没有CPU计算速度快，所以让程序阻塞于I/O操作将浪费大量的CPU时间。

操作工作队列一定要加锁（locker），因为它被所有线程共享。我们用信号量来标识请求队列中的请求数，通过m\_queuestat.wait();来等待一个请求队列中待处理的请求，然后交给线程池中的空闲线程来处理。

设置成脱离态的目的？

为了在使用线程的时候，避免线程的资源得不到正确的释放，从而导致了内存泄漏的问题。所以要确保进程为可分离的的状态，否则要进行线程等待已回收他的资源。

线程池中线程的数量如何确定？

针对不同的任务性质而言：CPU密集型任务应配置尽可能小的线程，如配置CPU个数+1的线程数，IO密集型任务应配置尽可能多的线程，因为IO操作不占用CPU，不要让CPU闲下来，应加大线程数量，如配置两倍CPU个数+1。如果需要等待的时间越长，则CPU空闲的时间越长，那么线程数量应设置得越大，才能更好的利用CPU。

线程的同步机制有哪些？

介绍一下几种典型的锁？

读写锁

多个读者可以同时进行读

写者必须互斥（只允许一个写者写，也不能读者写者同时进行）

写者优先于读者（一旦有写者，则后续读者必须等待，唤醒时优先考虑写者）

互斥锁

一次只能一个线程拥有互斥锁，其他线程只有等待

互斥锁是在抢锁失败的情况下主动放弃CPU进入睡眠状态直到锁的状态改变时再唤醒，而操作系统负责线程调度，为了实现锁的状态发生改变时唤醒阻塞的线程或者进程，需要把锁交给操作系统管理。

条件变量

互斥锁一个明显的缺点是他只有两种状态：锁定和非锁定。而条件变量是让线程阻塞，等待另一个线程发送唤醒信号的方法弥补了互斥锁的不足，常和互斥锁一起使用，以免出现竞态条件。当条件不满足时，线程先解开互斥锁并阻塞线程然后等待条件发生变化。一旦其他的某个线程改变了条件变量，则唤醒一个或多个正被此条件变量阻塞的线程。

总的来说互斥锁是线程间互斥的机制，条件变量则是同步机制。

线程池中的工作线程是一直等待吗？

线程池中的工作线程是处于一直阻塞等待的模式下的。在run函数中，我们为了能够处理高并发的问题，将线程池中的工作线程都设置为阻塞等待在请求队列是否不为空的条件上，因此项目中线程池中的工作线程是处于一直阻塞等待的模式下的。

线程池工作线程处理完一个任务后的状态是什么？

（1） 当处理完任务后如果请求队列为空时，则这个线程重新回到阻塞等待的状态

（2） 当处理完任务后如果请求队列不为空时，那么这个线程将处于与其他线程竞争资源的状态，谁获得锁谁就获得了处理事件的资格。

如果同时有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？

本项目是通过对子线程循环调用来**解决高并发**的问题的。创建线程的同时就调用了pthread\_detach将线程进行分离，不用单独对工作线程进行回收，资源自动回收。

通过子线程一直运行在while循环中，让每一个线程池中的线程永远都不会停终止，访问请求被封装到请求队列(list)中，如果没有任务线程就一直阻塞等待，有任务线程就抢占处理，直到请求队列为空，表示任务全部处理完成。

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?

会影响接下来的客户请求，因为线程池内线程的数量时有限的，如果客户请求占用线程时间过久的话会影响到处理请求的效率，当请求处理过慢时会造成后续接受的请求只能在请求队列中等待被处理，从而影响接下来的客户请求。

什么是虚假唤醒？

一句话：唤醒的线程想要拿到的任务被其他刚执行完任务的线程拿走了

举个例子，我们现在有一个生产者-消费者队列和三个线程。

1号线程从队列中获取了一个元素，此时队列变为空。这时，3号线程将一个元素入队，并调用cond.notify()唤醒条件变量。处于等待状态的2号线程接收到3号线程的唤醒信号，便准备解除阻塞状态，执行接下来的任务(获取队列中的元素)。此时1号线程刚好执行完之前的元素操作，返回再去请求队列中的元素，1号线程便获得队列的锁，检查到队列非空，就获取到了3号线程刚刚入队的元素，然后释放队列锁。等到2号线程获得队列锁，判断发现队列仍为空，2号线程而言，这次唤醒就是“虚假”的，它需要再次等待队列非空。

在wait成功之后，资源就一定可以被使用么？

如果同时有两个或者两个以上的线程正在等待此资源，wait返回后，资源可能已经被使用了。

有可能多个线程都在等待这个资源可用的信号，信号发出后只有一个资源可用，但是有A，B两个线程都在等待，B比较速度快，获得互斥锁，然后加锁，消耗资源，然后解锁，之后A获得互斥锁，但A回去发现资源已经被使用了，它便有两个选择，一个是去访问不存在的资源，另一个就是继续等待，那么继续等待下去的条件就是使用while，要不然使用if的话pthread\_cond\_wait返回后，就会顺序执行下去。所以，在这种情况下，应该使用while而不是if

连接过程：

客户端：

创建客户端的socket int sockfd = socket(AF\_INET,SOCK\_STREAM,0);

向服务器发起连接。 if (connect(sockfd,(struct sockaddr \*)&servaddr,sizeof(servaddr))==-1)

与服务端通讯

发送报文

(iret=send(sockfd,buffer,strlen(buffer),0))

接收服务端的回应报文，如果服务端没有发送回应报文，recv()函数将阻塞等待。

if ( (iret=recv(sockfd,buffer,sizeof(buffer),0))<=0)

关闭socket，释放资源。 close(sockfd);

服务端：

创建客户端的socket

int listenfd = socket(AF\_INET,SOCK\_STREAM,0);

把服务端用于通信的IP和端口绑定到socket上。

if (bind(listenfd,(struct sockaddr \*)&servaddr,sizeof(servaddr))==-1)

把socket设置为可连接（监听）的状态。

if (listen(listenfd,5) == -1 )

受理客户端的连接请求，没有客户端连上来，accept()函数将阻塞等待。（select可解决）

int clientfd=accept(listenfd,0,0);

与客户端通信。

接收客户端的请求报文，如果客户端没有发送请求报文，recv()函数将阻塞等待。

if ( (iret=recv(clientfd,buffer,sizeof(buffer),0))<=0)

向客户端发送回应报文。

if ( (iret=send(clientfd,buffer,strlen(buffer),0))<=0)

关闭socket，释放资源。

close(listenfd); // 关闭服务端用于监听的socket。

close(clientfd); // 关闭客户端连上来的socket。可以关闭客户端

子进程调试

Shell中输入 ps -ef|grep demo 查找所有与demo相关的进程

三次握手

服务端调用listen()函数后进入监听（等待连接）状态，这时候，客户端就可以调用connect()函数发起TCP连接请求，connect()函数会触发三次握手，三次握手完成后，客户端和服务端将建立一个双向的传输通道。

四次挥手（握手）

断开一个TCP连接时，客户端和服务端需要相互总共发送四个包以确认连接的断开。在socket编程中，这一过程由客户端或服务端任一方执行close()函数触发。

细节：

1）主动断开的端在四次挥手后，socket的状态为TIME\_WAIT，该状态将持续2MSL（30秒/1分钟/2分钟）。 MSL（Maximum Segment Lifetime）报文在网络上存在的最长时间，超过这个时间报文将被丢弃。

2）如果是客户端主动断开，TIME\_WAIT状态的socket几乎不会造成危害。a）客户端程序的socket很少，服务端程序的socket很多（成千上万）；b）客户端的端口是随机分配的，不存在重用的问题。

3）如果是服务端主动断开，有两方面的危害：a）socket没有立即释放；b）端口号只能在2MSL后才能重用。

在服务端程序中，用setsockopt()函数设置socket的属性（一定要放在bind()之前）。

int opt = 1;

setsockopt(m\_listenfd,SOL\_SOCKET,SO\_REUSEADDR,&opt,sizeof(opt));

TCP缓存

系统为每个socket创建了发送缓冲区和接收缓冲区，应用程序调用send()/write()函数发送数据的时候，内核把数据从应用进程拷贝socket的发送缓冲区中；应用程序调用recv()/read()函数接收数据的时候，内核把数据从socket的接收缓冲区拷贝应用进程中。

发送数据即把数据放入发送缓冲区中。接收数据即从接收缓冲区中取数据。

如果自己的发送缓冲区和对端的接收缓冲区都满了，send()函数会阻塞。

向socket中写入数据后，如果关闭了socket，对端还能接收到数据吗？能，存在缓冲区了

TCP粘包和分包

Tcp接收到数据存放在接收缓冲区，数据之间不存在分隔符，如果数据没能及时取走，数据会看上去粘在一起。Tcp报文缺省大小是1460字节，如果发送缓冲区中的数据超过1460字节，将分成多个包发送，如果接收方及时取走数据，看上去收到多个报文。

解决办法：

1. 采用固定长度的报文。
2. 报文之前加上报文长度，报文头部（4个字节整数）+报文内容
3. 报文之间加分隔符。即http的分隔符\r\n\r\n

为什么需要接收缓冲区（程序里面的缓冲区，不是socket的）

采用了边沿触发模式，所以需要一次性把socket里面的数据读完，否则会反复触发pollin事件。读出来的数据可能不是完整的，需要先把数据存在buffer中，等到形成一条完整的信息再通知程序的业务逻辑。

为什么需要接收缓冲区（程序里面的缓冲区，不是socket的）

Tcp的发送缓冲区会被填满引起阻塞，我们不能原地等。需要一个程序中的发送缓冲区outbuffer，程序只管把数据放进outbuffer中，然后注册pollout事件，一旦socket可写了，就立刻发送数据。数据发送完需要立刻停止关注pollout事件。