网络编程和muduo

网络编程

Conception：

a. 阻塞/非阻塞、同步/异步
b. 5种IO模型
c. IO多路复用
d. select、poll
e. epoll
f. LT、ET

Question：

a. 阻塞/非阻塞、同步/异步？
    - 一个IO接口调用有两个阶段：数据准备和数据读写
    - 数据准备阶段
        - 阻塞：数据没有到来时候阻塞等待数据
        - 非阻塞：不用阻塞，根据返回值进行判断
    - 数据读写阶段
        - 同步：应用程序自己去读取buf中的数据
        - 异步：让操作系统读取，读取完毕后约定通知方式进行通知

b. 5种IO模型？
    - 两个阶段：等待数据、将数据从内核拷贝到用户空间
    - 阻塞IO：发起IO调用后进程被阻塞，等待数据以及数据拷贝到用户空间后返回

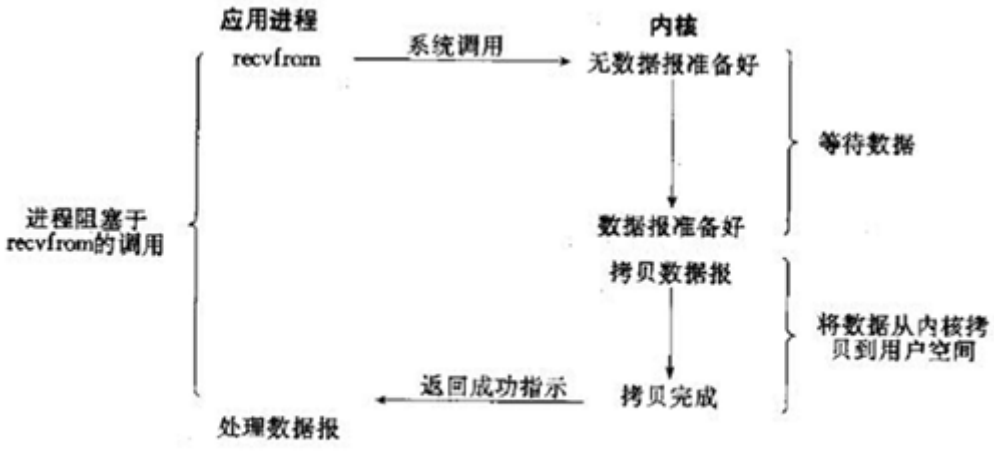


图 6.1 阻塞 I/O 模型

    - 非阻塞IO：发起IO调用后如果数据没有到来则返回错误，有数据则进行拷贝

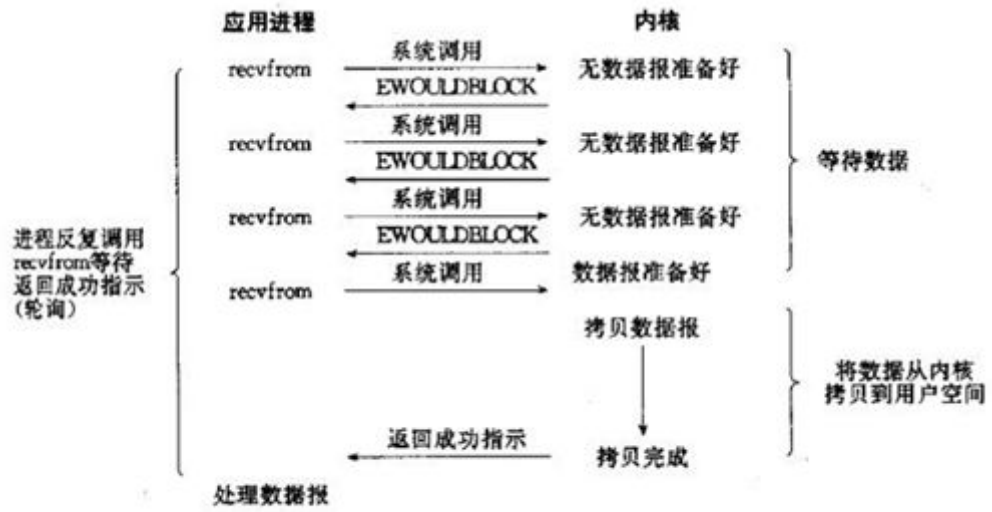


图 6.2 非阻塞 I/O 模型

- IO多路复用：多个IO注册到一个进程上，没有数据可读时，select进行被阻塞，数据到来时候select返回，进行数据拷贝

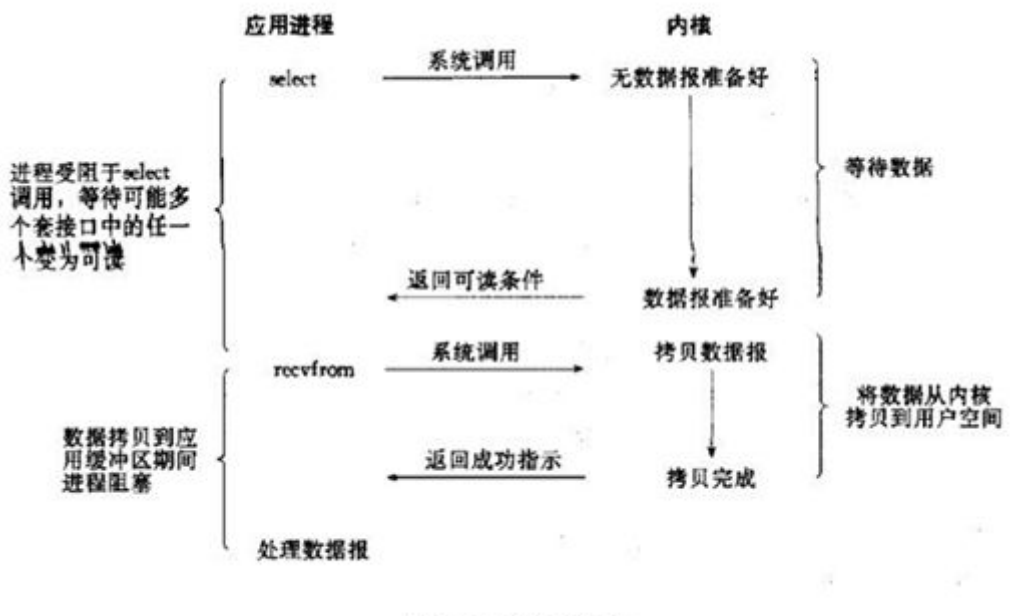


图 6.3 I/O 复用模型

- 信号驱动IO：向内核注册一个信号处理函数，进程不阻塞，有数据发送信号，进行数据拷贝

图 6.4 信号驱动 I/O 模型

- 异步IO：进程不阻塞，数据到来、拷贝到用户空间后进行完成通知

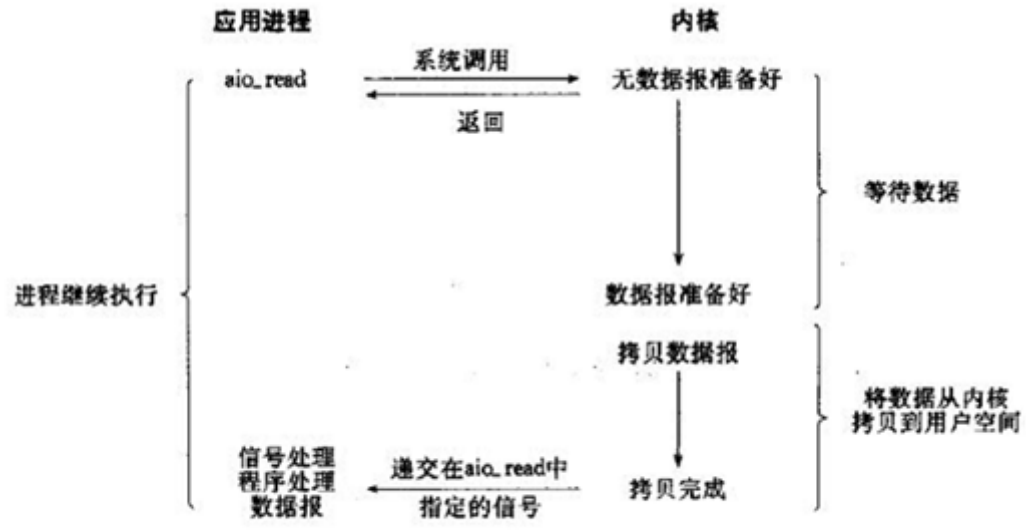
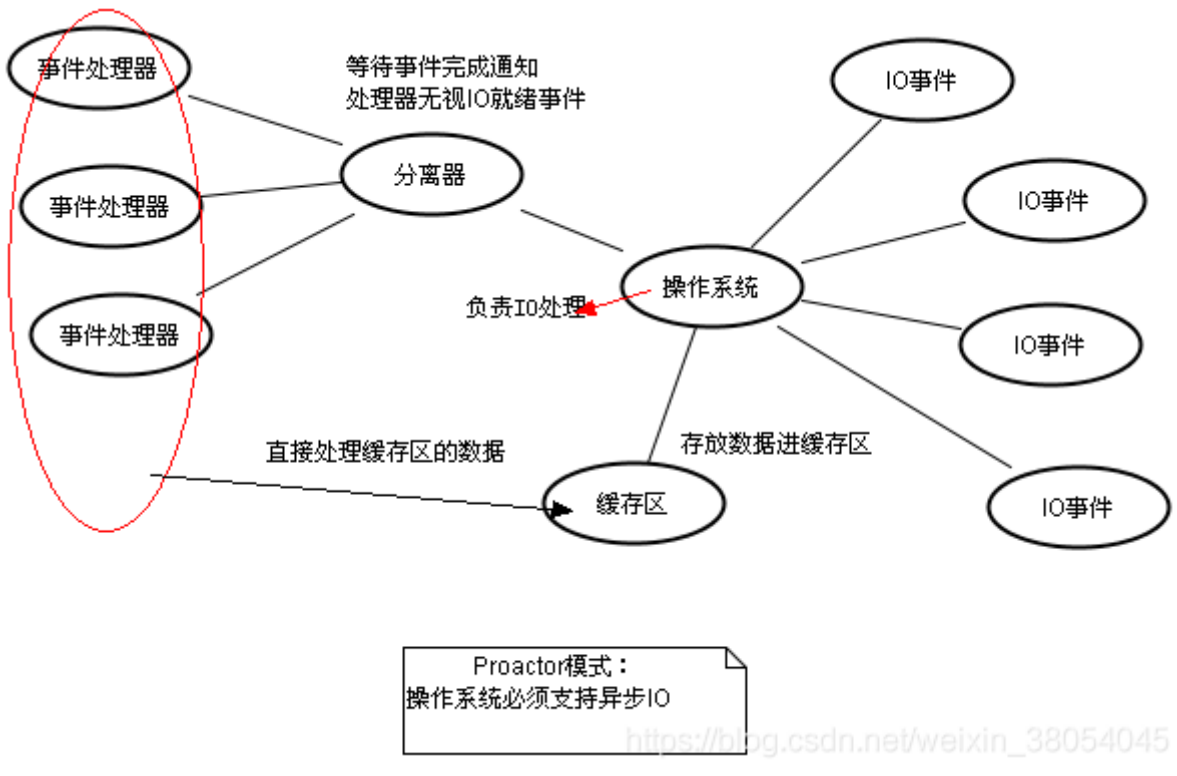


图 6.5　异步 I/O 模型

- 参考

c. Reactor、Proactor模式?

- 不同：真正的读取和写入操作是谁完成的，Reactor模式中读写数据、接收新连接、处理客户请求都在工作线程中进行；Proactor模式仅仅负责业务逻辑
- Reactor：
  - 同步I/O；
    - 注册读就绪事件和相关处理器；
    - 等待事件发生；
    - 调用对应的处理器；
    - 处理器执行读操作，然后根据读到的内容进行处理
  - 等待就绪再调用write()写入数据，写完数据处理后续逻辑
- Proactor：
  - 异步I/O；
    - 初始化异步读取操作，注册相应的事件处理器，关注的不是读就绪，而是**关注读取完成**事件；
    - 等待事件发生；
    - 调用内核完成读取操作，将读取的内容放入用户传递的**缓冲区**中
    - 激活事件处理器，事件处理器直接从缓存区读，而不需要进行实际读取



- 由内核负责写操作，写完后调用相应的回调函数处理后续逻辑

d. IO多路复用？

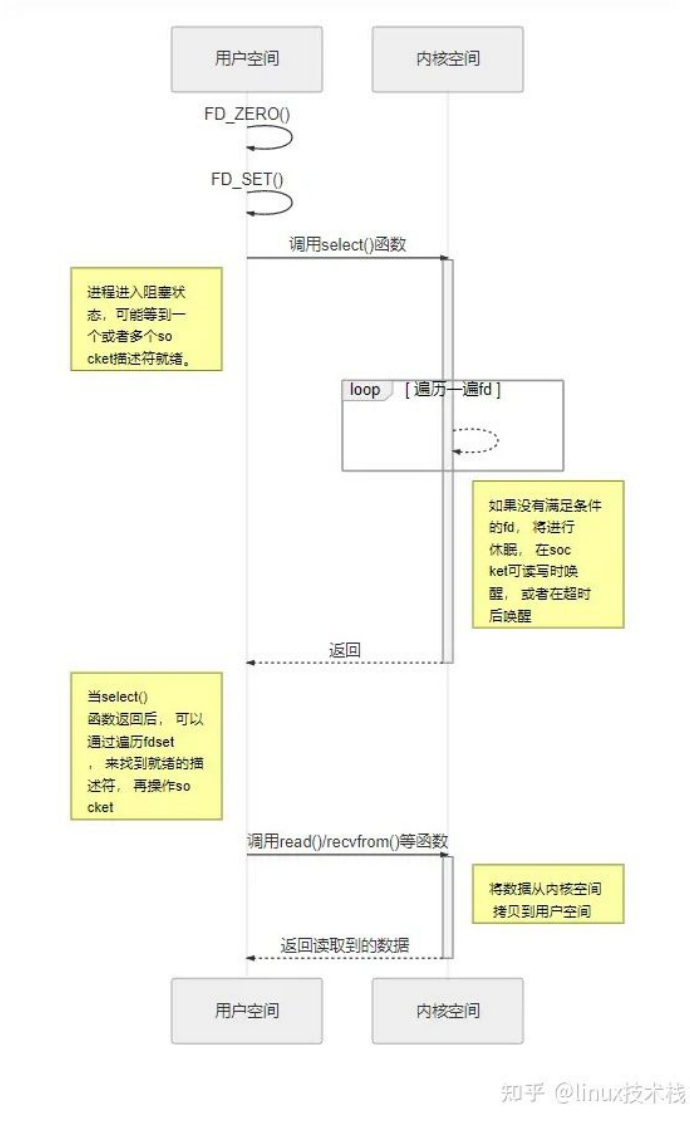
- 一个进程可以监视多个描述符，同时处理多个IO请求，使用select、poll、epoll函数，谁的数据到达就处理谁的请求
- 多路：监听多个socket网络连接
- 复用：复用同一个线程

- 优点：
  - 只需要一个进程就可以处理多个事件，数据共享容易、调试容易
  - 单一进程中，不需要进程线程切换的开销
- 缺点：
  - 逻辑困难
  - 不能充分利用多核处理器

e. select、poll、epoll

- select：bitmap；监听端口32位默认是1024；每次调用select都要将fds拷贝到内核态空间，内核做遍历，用户态进行事件处理

1. 用户线程调用select，将fd_set从用户空间拷贝到内核空间
2. 内核在内核空间对fd_set遍历一遍，检查是否有就绪的socket描述符，如果没有的话，就会进入休眠，直到有就绪的socket描述符
3. 内核返回select的结果给用户线程，即就绪的文件描述符数量
4. 用户拿到就绪文件描述符数量后，再次对fd_set进行遍历，找出就绪的文件描述符
5. 用户线程对就绪的文件描述符进行读写操作



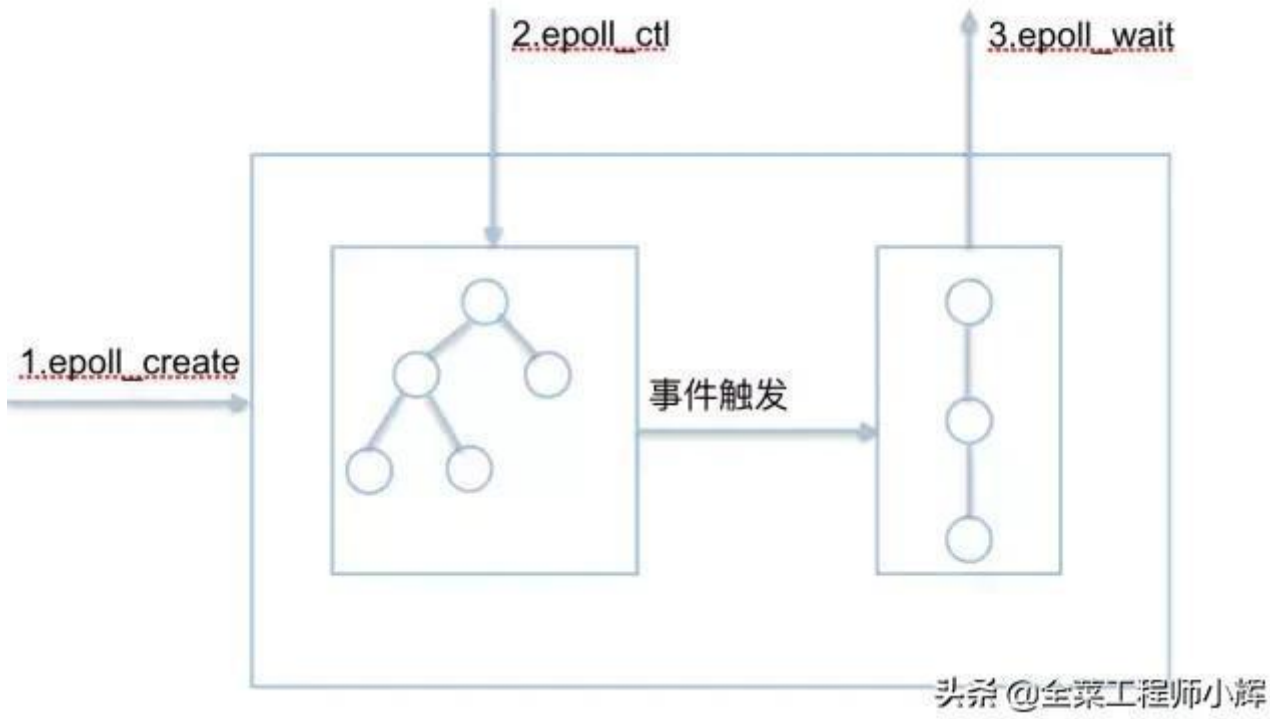
- poll：和select差别不大，数组；监听端口没有上限；
- select、poll缺点：整体要进行**内核空间的拷贝**，不断轮询fd集合，开销随文件描述符数量增加而增加
- epoll：红黑树，等待队列列表、一个就绪链表；epoll_create创建文件句柄，epoll_ctl注册文件描述符，epoll_wait返回已经就绪的文件描述符链表
- select、poll、epoll
- 文章

f. select和epoll应用场景

- 并发量低，socket都比较活跃的情况下，使用select；比如游戏服务器发送心跳检测或者频繁同步，此时用epoll简历红黑树和链表效率反而不高
- 负责大量的客户端链接时候，使用epoll

g. epoll的实现原理?

- epoll_create：创建一个epoll对象epfd，创建红黑树和就绪链表
- epoll_ctl：向epfd上添加/修改/删除事件，在红黑树上进行这些操作，红黑树的每个节点都是一个epitem结构体；向内核函数注册回调事件，中断事件来临时向就绪链表中插入数据
- epoll_wait：检查rdllist双向链表是否有epitem元素；有数据就返回，没有的话就sleep，timeout之后就绪链表没有数据也返回



h. SO_REUSEADDR、SO_REUSEPORT

- SO_REUSEADDR：解决TIME_WAIT问题
- SO_REUSEPORT：允许多个套接字监听同一个IP 和端口组合
- 文章

i. **惊群问题**

- 概念：简单说，惊群是因为多进/线程在同时阻塞等待同一个事件的时候（休眠状态），当事件发生时，就会唤醒所有等待的进/线程。但是事件只能被一个进程或线程处理，而其他进/线程获取失败，只能重新进入休眠状态，这种现象和性能浪费就叫做惊群
- accept惊群
- epoll惊群
- Nginx解决惊群：使用锁

j. **read/write函数**

```
size_t recv(int sockfd,void *buf,int len,int flags)
size_t send(int sockfd,void *buf,int len,int flags)


n > 0: 读取到n个字节;
n == 0: 对端关闭、文件末尾;
n == -1: 表示遇到问题,
    errno == EAGAIN/EWOULDBLOCK: 在非阻塞IO模式下，表示没有数据可读，可忽略本次read操作;
    errno == EINTR: 表示被信号中断，重新读取一次即可。
    其他错误类型。

 EAGAIN: 因为要是非阻塞的，所以要返回EAGAIN
```

k. **LT、ET模式?**

- select、poll只支持LT，epoll支持LT和ET
- LT（水平触发）：
  - 内核会告诉一个fd已经就绪，如果没有立即处理，会持续通知，直到处理
  - LT是阻塞模式或者非阻塞模式都可以

- 优点：LT不会丢失数据或消息，ET只触发一次
- LT比ET少一次系统调用，因为ET的read函数必须读到EAGAIN才能保证所有数据读完

- ET（边缘触发）：
  - 只在就绪时进行通知，之后不会再次通知
  - ET必须是非阻塞模式
  - 优点：系统中有大量**不需要读写**的fd，LT下每次都会返回，效率低
  - 缺点：每次读写事件必须保证把数据读写完，否则第二次可读的时候才会通知，造成积压
- 两者适用场景：
  - **并发量高**，用**LT**，muduo中有应用缓冲区的概念，把数据缓存在应用区，多次保证读完5MB就好；
  - **实时性要求高**，ET只触发一次，所以每次尽可能读完5M，但是可能导致busyloop，因为数据量过大影响连接公平性

l. **ET提醒一次，怎么保证把缓冲区数据都读完？**

- 函数返回-1，EAGAIN/EWOULDBLOCK时候表示读完，EBADF表示失败了
- 上述标志位存在errno中，函数会把错误码设置成相应的值

m. **客户端、服务端建立连接过程？**

- 客户端：socket()、connect()、write()、read()、close()
- 服务端：socket()、bind()、listen()、accept()、read()、write()、close()

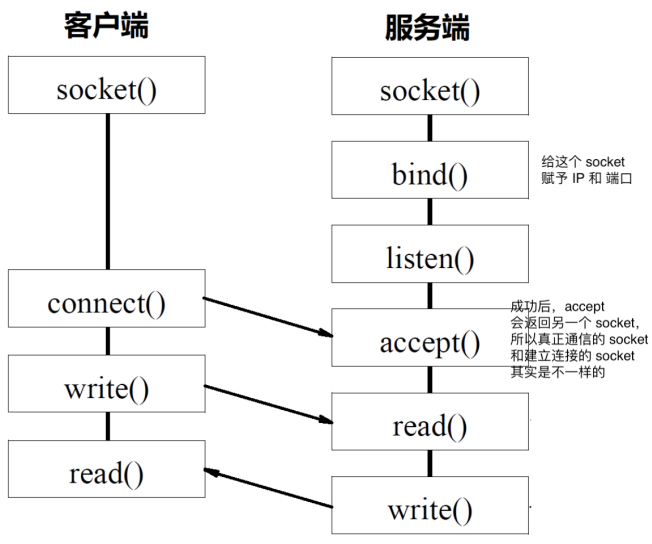
n. **socket为什么要有端口？**

- 源地址，源端口，目的地址，目的端口
- 如果没有端口这个概念，意味着每次只能在源和目的之间建立一条链路

o. **socket是什么？**

- 套接字（socket）是一个抽象层，应用程序可以通过它发送或接收数据，可对其进行像对文件一样的打开、读写和关闭等操作。套接字允许应用程序将I/O插入到网络中，并与网络中的其他应用程序进行通信。网络套接字是IP地址与端口的组合。
- 文件是用fopen打开，一个是用socket打开
- 一个是 fread/fwrite 读写，一个是 recv 和 send （Linux下用read和write，对文件和socket都能读写，只是无法直接设置一些特殊的flag）

p. 监听socket、已连接socket



- 好文
- 酒店的比喻

q. **C10K，C100K、C1000K**

- 极客时间文章
- C10K：2GB+千兆网卡，每个请求占用的内存很少，硬件没有问题，自然是软件的问题
  - C10K以前同步阻塞，10000个请求时候上下文切换和占用内存都会成为瓶颈；
  - 如果解决：一个线程内处理多个请求，I/O多路复用（select、poll、epoll、异步I/O）、一个主线程多个work线程、监听相同端口的多进程模型（SO_REUSERPORT）
- C100K：epoll+线程池
- C1000K：Linux 内核、再到 CPU、内存和网络等各个层次的深度优化，特别是需要借助硬件
- C10M：跳过内核协议栈的冗长路径，把网络包直接发送到应用程序处；DPDK（大页、CPU 绑定、内存对齐、流水线并发）、XDP

muduo

资料

- 项目地址：https://github.com/princeh23/mymuduo
- 参考文章

> 知乎 https://zhuanlan.zhihu.com/p/495016351
> **万字长文梳理Muduo库核心代码及优秀编程细节思想剖析**
> 原文地址： 长文梳理Muduo库核心代码及优秀编程细节剖析_我在地铁站里吃闸机的博客-CSDN博客一、前言： Muduo库是陈硕个人开发的Tcp网络编程库，支持Reactor模型。本人前段时间出于个人学习目的用c++11重构了Muduo…

- 线程模型：https://blog.csdn.net/yolo_yyh/article/details/118367979

Question：

muduo梳理（小的语法点）？

- one loop per thread + thread pool
- noncopyable：C++11中default、delete，protected中的构造和析构函数是为了自身不能被构造，但是子类可以被构造
- Logger：
- Timestamp、InetAddress：使用ex plcit的作用，explict避免被隐式转换（explcit）
- Channel：绑定了fd和感兴趣的事件，tie检测是否被绑定使用了weak_ptr和shared_ptr
  - **此处用到了智能指针：**
  - **Tcpconnection::connectEstablished()继承enable_shared_from_this，提供shared_from_this()，不传递this指针，传递一个shared指针（因为谁也不知道使用this会干什么，delete怎么办），使用share_ptr <this>行吗？（需要析构两次）（enable_shared_from_this）**
  - **tie()传入了一个指向Tcpconnection自身的shared_ptr，传递给channel的tie_（weak_ptr），weak_ptr常用的两个函数expired()和lock()，expired()判断是否过期，lock()返回shared_ptr指针 （智能指针）**

        - **handle里面把weak_ptr再提升为shared_ptr，给handleEventWithGuard处理完之后离开作用域，就计数-1**
    - poller：newDefaultPoller（获取当前具体的IO复用对象，poll还是epoll）单独文件实现，因为基类不能访问派生类
        - 负责文件描述符是否触发以及返回事件的文件描述符以及具体事件
    - epollpoller：epoll_create()、epoll_ctl、epoll_wait
        - channelmap、activechannel
    - gettid：one loop per thread获取当前线程id，__thread和C++11thread_local（虽然是全局变量，但是一个线程中存一份拷贝，当前线程堆变量的更高别的线程是看不到的）
    - EventLoop：
        - main loop：接收新用户连接，打包成channel分发给sub loop
        - 每个loop有一个wakeupfd，如何唤醒：向wakeupfd写一个数据
        - 每个loop绑定一个线程，线程一直执行while循环，调用poll方法进行事件监听；然后进行每个channel对应的事件进行处理
        - **C++11：原子变量存储有没有要执行的回调，互斥锁保护当前loop所有回调操作的线程安全**

```
mutex m

//1.需要手动解锁，会忘
m.lock()
m.unlock()

//2.出作用域自动释放，不支持手动解锁
lock_guard(m)

//3.出作用域自动释放，也能手动解锁
unique_lock(m)

{
    std::unique_lock<std::mutex> lock(mutex_);
}

三种锁讲解
```

    - Acceptor：接受新用户连接并分发连接给SubReactor
        - listen acceptSocket_，accept接收新连接，**负载均衡的选择一个subloop**
    - Socket：
        - setsockopt()
    - TcpConnection：和Acceptor是兄弟类关系，服务于subloop，对套接字fd以及相关方法进行封装，TcpConnection维护一个inputbuffer和outputbuffer

使用C++11的地方

- using代替typedef
- function、bind
- 智能指针 weak_ptr、shared_ptr

- default：声明构造函数为默认构造函数，如果类中有自定义构造函数，编译器就不会隐式生成默认构造函数（不声明时候，不传参数的构造就会报错）
- delete：禁止对象的拷贝和赋值，声明delete

多线程异步日志的实现(muduo中优秀实现1)?

- 问题1：一个线程来写日志速度慢
- 问题2：多线程日志写到多个地方，存在线程安全问题，可以使用全局mutex/每个线程写一个日志，但是存在抢占一个mutex/阻塞在写磁盘操作上

- 用一个背景线程负责收集日志信息，写入日志文件，其他线程往这个日志线程发送日志消息（异步日志，非阻塞日志），写到缓冲区中

- 生产者消费者缺点：写文件频繁

- 采用多缓冲机制
- 前端往buffer里填充数据，后端负责将buffer里的内容写进磁盘
- 前端（线程往缓冲写）：来消息往currentBuffer里写/拷贝（拷贝一条日志信息开销不是很大），满了存入buffers，写nextBuffer（线程写一条日志的时候会加mutex）
- 后端（交换之后从缓冲往磁盘写）：预先准备好buffer，超时/写满时候加互斥锁进行和前端buffer的交换（交换的时候会加mutex）
- 条件变量：前端通知后端数据写完

- 问题以及处理方法：日志消息堆积muduo直接丢掉多余的buffer

- 改进：全局用了一个互斥锁，java中的一个实现方法，根据线程id放在不同的bucket中，问题是后端实现比较复杂

Buffer缓冲区(muduo中优秀的实现2)?

- 原因1：非阻塞+IO多路复用；非阻塞中的轮询方法太耗费CPU，不能单独使用非阻塞；阻塞IO+IO多路复用一般不能用，阻塞IO中的read/write/accept/connect可能会阻塞，线程没有办法处理其他的IO事件
- 原因2：非阻塞IO的**核心**在于避免阻塞在read/write函数上，IO线程只能阻塞在IO多路复用的select、poll、epoll函数上，这样需要buffer

- output buffer：想发出去的还没有发完，100kb发了80kb剩下20kb
- input buffer：无边界字节流协议，可能拆包到达
- input/output buffer是不是**线程安全**的，input来说，回调只发生在tcpconnection的onmessage()上，应用程序只用onmessage()，保证其仅在当前l线程，不暴露给其他线程；output应用程序不会直接操作，调用TcpConnection::send保证在当前线程中执行
- TcpConnection::send()：操作outbuffer，如果在当前线程，就在当前线程进行操作；如果不在当前线程，把sendinloop转移到runinloop，保证在当前线程中

- 数据结构：预留头部(8字节) + 可读部分(writeIndex) + 可写部分(writeIndex)（整个是一个vector大小，几个下标是**整数**而不是指针，重新分配会导致指针失效）
- 设计点1：writeIndex和readIndex重合时候移到开头，避免前面的空间没有被用
- 设计点2：若干次读写后头部越来越长，前面够的话会把原有数据移到前面
- 设计点3：自动增长，且发送完不会回收，利用vector的capacity
- 设计点4：前方添加，聊天服务器在序列化完成后，在前方添加长度（粘包、拆包问题），简化客户代码，以空间换时间
- 设计点5：readv()函数可以分散读，读到分散的内存中，对于TCP缓冲区的数据量有多少是**未知的**，可能会溢出，再用一个extrabuf，先存起来之后再进行扩容

定时器(muduo中优秀的实现3)?

- 作用：每个loop绑定一个定时器，用于执行一些定时任务（心跳检测等等）
- 用的是Linux下的timerfd，将一个loop和一个fd绑定给channel，超时之后触发回调函数handleRead

- Timestamp：当前时间相关计算
- Timer：回调函数、超时时间、时间间隔、是否重复执行
- TimerId：每个Timer对应的ID
- TimerQueue：每个loop绑定的定时器，用set红黑树来存储
- 问题：裸指针管理，用shared_ptr小题大做；std::unique_ptr是不能 cpoy 的，只能 move，因此包含std::unique_ptr的std::pair<>也是如此。但由于 std::set的 key 是不可修改的，只能 copy。所以总会编译失败

One Loop Per Thread?

- 充分利用多核CPU的能力
- 每个loop绑定一个线程，线程一直执行while循环，调用poll方法进行事件监听
- muduo如何保证One Loop Per Thread：
  - 如果当前线程调用的就是loop绑定的线程中，就执行函数回调操作
  - 当前loop和thread不对应，放进队列中，唤醒所在loop（向wakeupfd中写入数据即为唤醒），把可调用对象放到pendingLoop中，等到这个loop对应的线程启动后dopendingloop()进行消耗（不应该当前错的线程执行，而不是当前loop错了，这个函数回调就应该对应这个loop）

项目中的问题?

- chatserver使用muduo时候发现，用户发送消息后，好友只能接受到部分消息

```
/**
* 主要是else里面  如果不在当前线程中执行的这个函数，需要对buf数据进行一个拷贝
* 因为使用runinloop注册会造成send操作有一定的延迟，因为它不是立马执行的，而是注册给这个线程让它待会来执行的。
* 在延迟处理的这段时间send过来的数据buf可能会被销毁
* 因为这个函数的buf参数是传的引用进来的，不能保证资源的是否销毁。
  // 调用send函数之前，是把缓冲区里的内容提出来，上一个调用函数的msg离开作用域已经被销毁了
```

```
  // 值传递可以解决问题
  // 传引用为了提高效率，绝大多数都是thread和loop对应
*/
```

- 涉及到：buffer
- 缓冲区相关：input/output buffer是不是**线程安全**的，input来说，回调只发生在tcpconnection的onmessage()上，应用程序只用onmessage()，保证其仅在当前l线程，不暴露给其他线程；output应用程序不会直接操作，调用TcpConnection::send保证在当前线程中执行
- TcpConnection::send()：操作outbuffer，如果在当前线程，就在当前线程进行操作；如果不在当前线程，把sendinloop转移到runinloop，此时在非当前loop中执行cb，需要唤醒loop所在线程，此时有延迟，buf参数传的是引用，缓冲区失效，实际内容消失

线程池：

- 执行任务的工作线程、任务队列、
- 互斥锁所有线程公用、每个线程都有wait状态进入休眠，等待notify_one唤醒
- 工作线程不断从任务队列取任务；没有任务就挂起休眠；任务队列有新任务就notify_one任意线程
- ThreadPool实现 讲解

为什么用LT模式？

- ET只触发一次，LT不会丢失数据或消息
- 并发量高，用LT，muduo中有应用缓冲区的概念，把数据缓存在应用区，多次保证读完5MB就好；实时性要求高，ET只触发一次，所以每次尽可能读完5M，但是可能导致busyloop，只，因为数据量过大影响连接公平性
- LT比ET少一次系统调用，因为ET的read函数必须读到EAGAIN才能保证所有数据读完

如果内核写缓冲区充足，epoll的LT模式会反复的触发可写事件，怎么解决？

- 开始时候不把注册fd的EPOLLOUT事件加入epoll，需要向socket写数据时候，直接调用write/send()函数，根据差值计算剩余数据量，如果还有未发送的数据，放到缓冲区之后，注册epollout事件，在epoll驱动下，发送完剩余数据，再注销掉EPOLLOUT读事件（muduo->TcpConnection->sendinloop）
- 使用ET模式（边沿触发），这样socket有可写事件，只会触发一次。
- 在epoll_ctl()使用**EPOLLONESHOT**标志，当事件触发以后，socket会被禁止再次触发。

如果都不要IO多路复用（select、poll、epoll），套接字会怎么样？

- 阻塞和非阻塞

多线程+阻塞可以实现吗？多线程的开销来自哪里？

- 创建销毁线程
- 线程同步、互斥

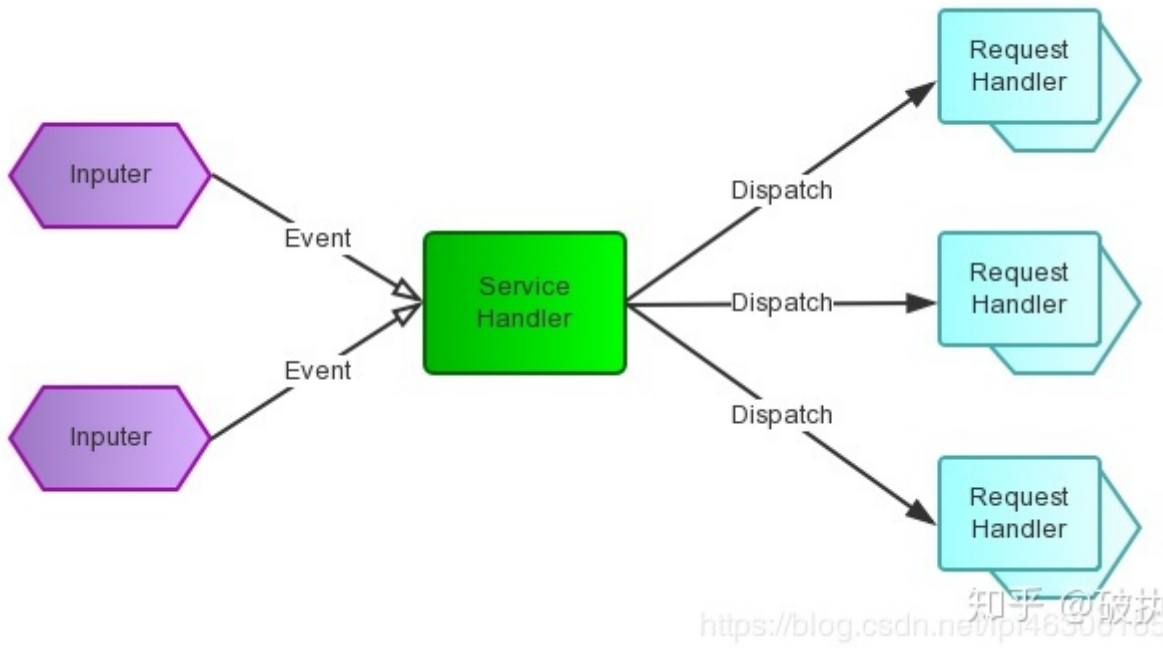
boost库缺点

- boost是试验版，冒险版，激进版的stl
- stl是稳定版，保守版，工业版的boost
- boost就是STL的孵化器

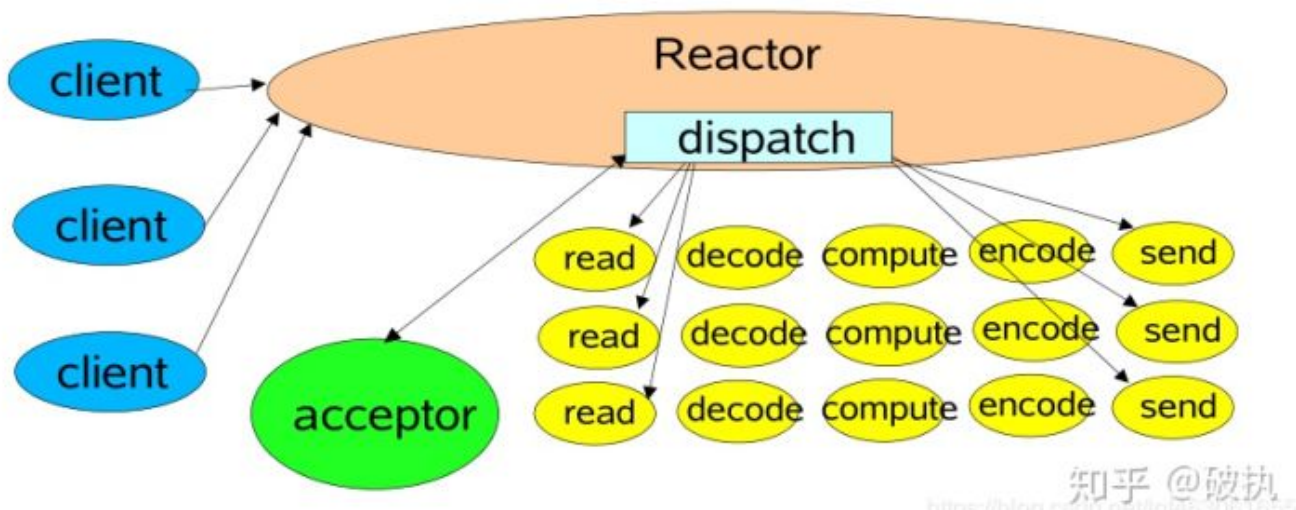
- bind function shared_ptr shared_lock unique_lock
- any：类型安全的void*，C++17加入了std::any
- boost库太重，需要编译很久，宏太多代码分析起来比较困难。

muduo in C++98	muduo in C++11	muduo in C++17
gcc 4.4+	gcc 4.7+	gcc 7.1+
`boost::function`	`std::function`	same as C++11
`boost::bind`	`std::bind` / lambda	same as C++11
`boost::shared_ptr`	`std::shared_ptr`	same as C++11
`boost::scoped_ptr`	`std::unique_ptr`	same as C++11
`boost::noncopyable`	`muduo::noncopyable`	same as C++11
`boost::any`	`boost::any`	`std::any`
`muduo::StringPiece`	`muduo::StringPiece`	`std::string_view`
`boost::circular_buffer`	`boost::circular_buffer`	`std::deque`
`boost::ptr_vector<T>`	`std::vector<std::unique_ptr<T>>`	same as C++11
`STATIC_ASSERT`	`static_assert`	same as C++11
`boost::type_traits`	`std::type_traits`	same as C++11

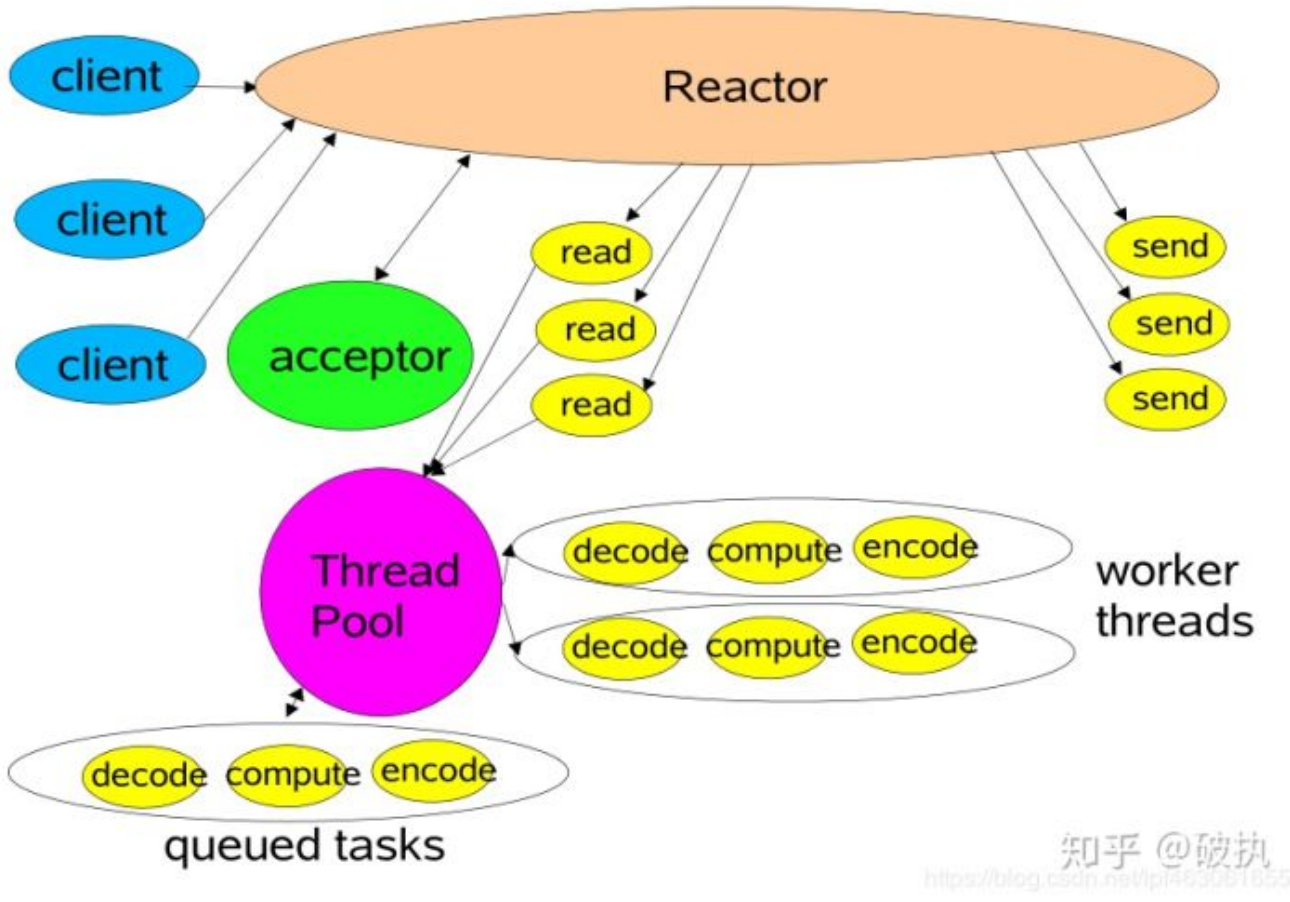
Reactor模式?



- 单线程模式



- 工作者线程池模式



- 多线程模式：mainReactor负责新连接建立，subReactor复杂读写网络数据