剖析muduo网络库核心代码

> 网上课堂地址：**https://fixbug.ke.qq.com/**　　咨询学习问题加QQ：2491016871

目录

目的

1、理解阻塞、非阻塞、同步、异步
2、理解Unix/Linux上的五种IO模型
3、epoll的原理以及优势
4、深刻理解Reactor模型

5、从开源C++网络库，学习优秀的代码设计
6、掌握基于事件驱动和事件回调的epoll+线程池面向对象编程
7、通过深入理解muduo源码，加深对于相关项目的深刻理解
8、改造muduo，不依赖boost，用C++11重构

春哥的程序人生： **https://www.ituring.com.cn/article/504549**

知识储备

1、TCP协议和UDP协议
2、TCP编程和UDP编程步骤
3、IO复用接口编程select、poll、epoll编程
4、Linux的多线程编程pthread、进程和线程模型 C++20标准加入了协程的支持

推荐阅读： 《Linux高性能服务器编程》《UNIX环境高级编程》《鸟哥的Linux私房菜》

阻塞、非阻塞、同步、异步

> 典型的一次IO的两个阶段是什么？ 数据准备 和 数据读写

数据准备：根据系统IO操作的就绪状态

- 阻塞
- 非阻塞

数据读写：根据应用程序和内核的交互方式

- 同步
- 异步

陈硕大神原话：在处理 IO 的时候，阻塞和非阻塞都是同步 IO。只有使用了特殊的 API 才是异步 IO。

同步	IO multiplexing (select/poll/epoll)	
	阻塞	非阻塞

异步	Linux	Windows	.NET
	AIO	IOCP	BeginInvoke/EndInvoke

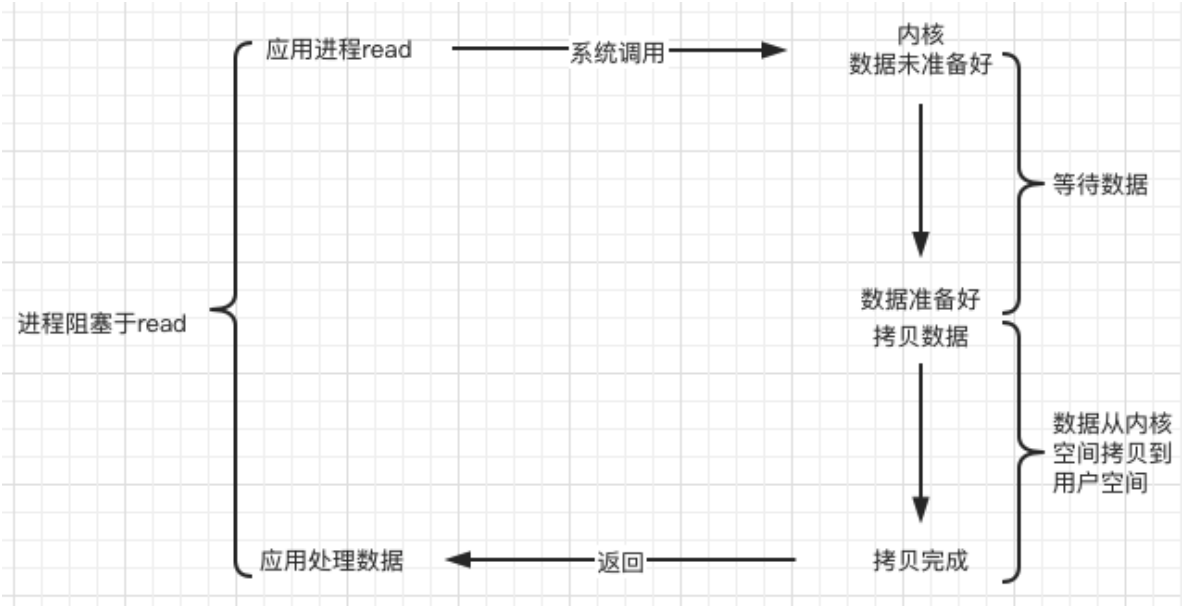
> 一个典型的网络IO接口调用，分为两个阶段，分别是“数据就绪”和“数据读写”，数据就绪阶段分为阻塞和非阻塞，表现得结果就是，阻塞当前线程或是直接返回。
>
> 同步表示A向B请求调用一个网络IO接口时（或者调用某个业务逻辑API接口时），数据的读写都是由请求方A自己来完成的（不管是阻塞还是非阻塞）；异步表示A向B请求调用一个网络IO接口时（或者调用某个业务逻辑API接口时），向B传入请求的事件以及事件发生时通知的方式，A就可以处理其它逻辑了，当B监听到事件处理完成后，会用事先约定好的通知方式，通知A处理结果。

- 同步阻塞 int size = recv(fd, buf, 1024, 0)
- 同步非阻塞 int size = recv(fd, buf, 1024, 0)

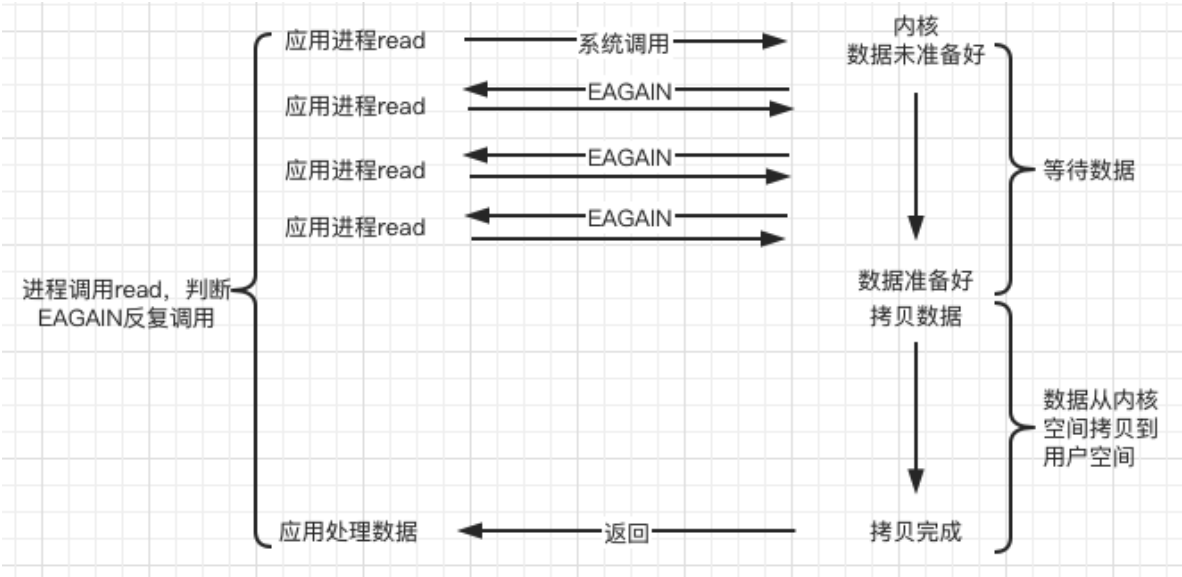
- 异步阻塞
- 异步非阻塞

Unix/Linux上的五种IO模型

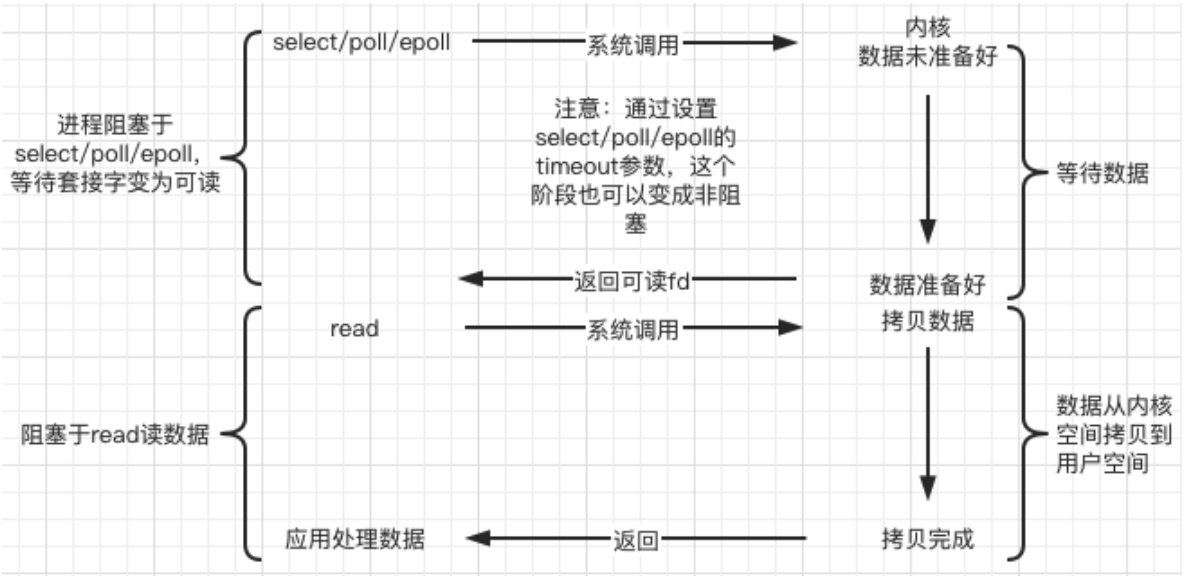
阻塞 blocking



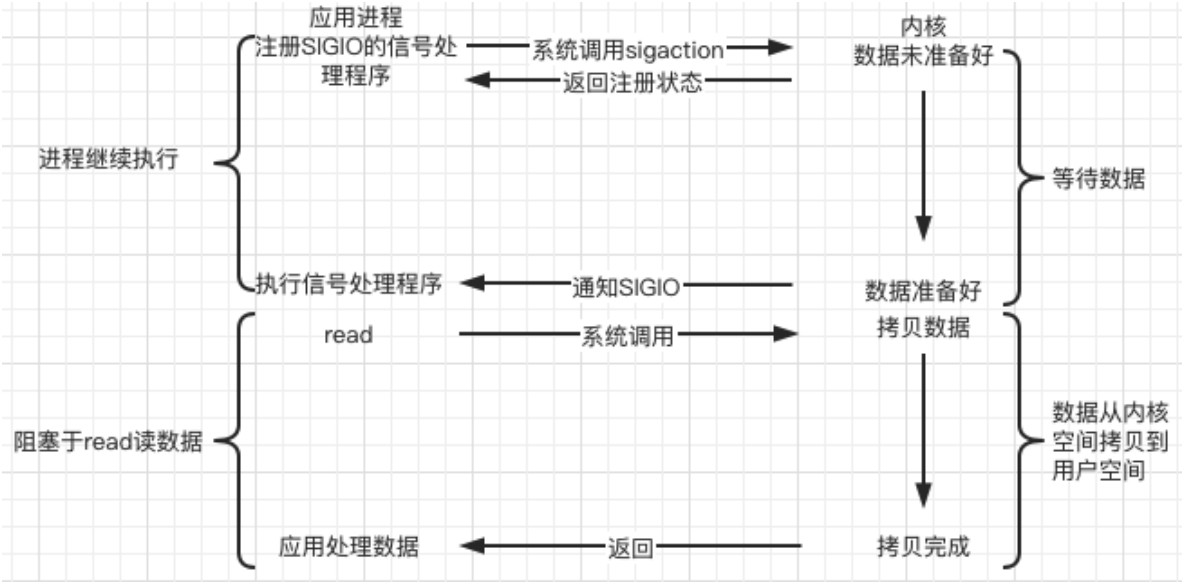
非阻塞 non-blocking



IO复用（IO multiplexing）

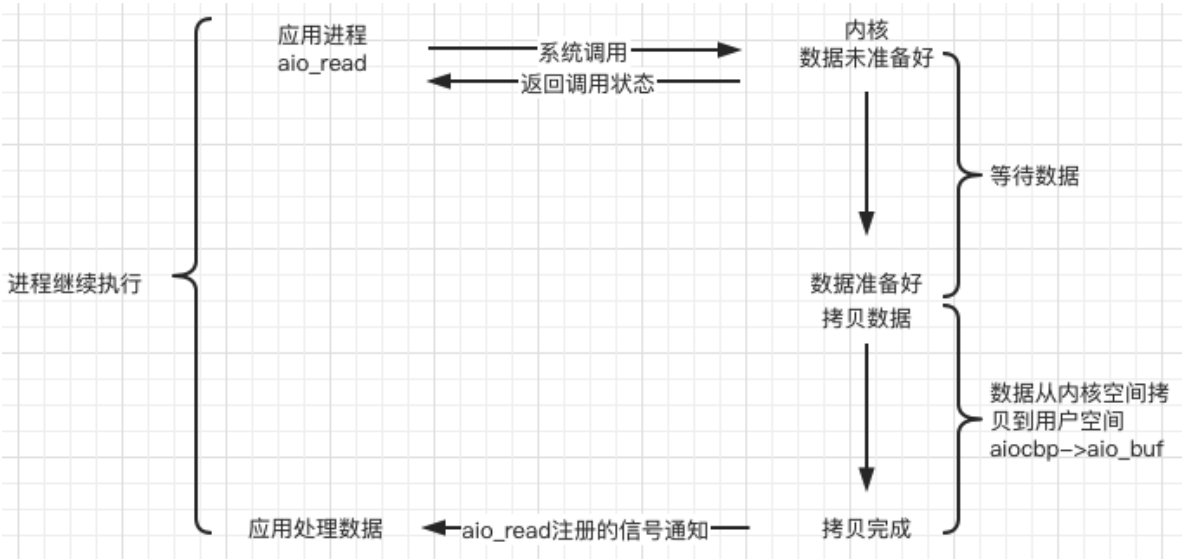


信号驱动（signal-driven）



内核在第一个阶段是异步，在第二个阶段是同步；与非阻塞IO的区别在于它提供了消息通知机制，不需要用户进程不断的轮询检查，减少了系统API的调用次数，提高了效率。

异步（asynchronous）



```
struct aiocb {
   int                 aio_fildes
   off_t               aio_offset
   volatile void   *aio_buf
   size_t              aio_nbytes
   int                 aio_reqprio
   struct sigevent aio_sigevent
   int                 aio_lio_opcode
}
```

典型的异步非阻塞状态，Node.js采用的网络IO模型。

好的网络服务器设计

在这个多核时代，服务端网络编程如何选择线程模型呢？ 赞同libev作者的观点：one loop per thread is usually a good model，这样多线程服务端编程的问题就转换为如何设计一个高效且易于使用的event loop，然后每个线程run一个event loop就行了（当然线程间的同步、互斥少不了，还有其它的耗时事件需要起另外的线程来做）。

event loop 是 non-blocking 网络编程的核心，在现实生活中，non-blocking 几乎总是和 IO-multiplexing 一起使用，原因有两点：

- 没有人真的会用轮询 (busy-pooling) 来检查某个 non-blocking IO 操作是否完成，这样太浪费CPU资源了。
- IO-multiplex 一般不能和 blocking IO 用在一起，因为 blocking IO 中 read()/write()/accept()/connect() 都有可能阻塞当前线程，这样线程就没办法处理其他 socket 上的 IO 事件了。

所以，当我们提到 non-blocking 的时候，实际上指的是 non-blocking + IO-multiplexing，单用其中任何一个都没有办法很好的实现功能。

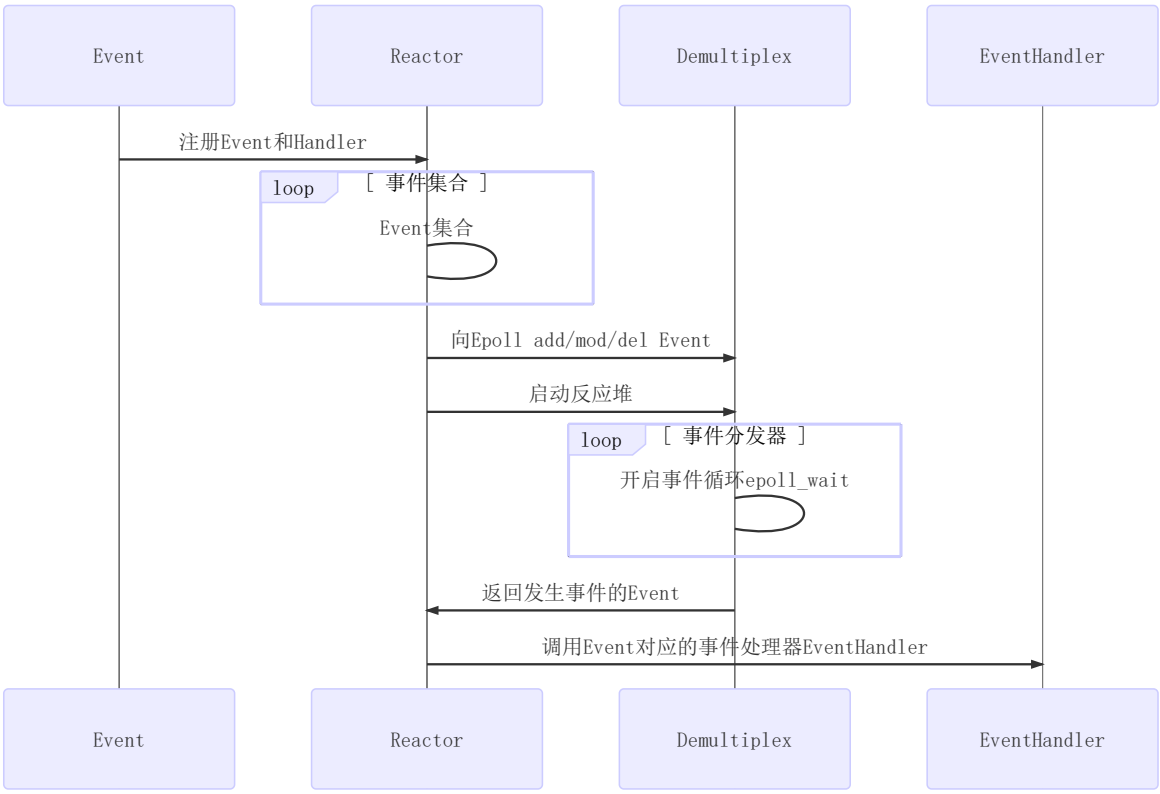
> epoll + fork不如epoll + pthread?

强大的nginx服务器采用了epoll+fork模型作为网络模块的架构设计，实现了简单好用的负载算法，使各个fork网络进程不会忙的越忙、闲的越闲，并且通过引入一把乐观锁解决了该模型导致的**服务器惊群**现象，功能十分强大。

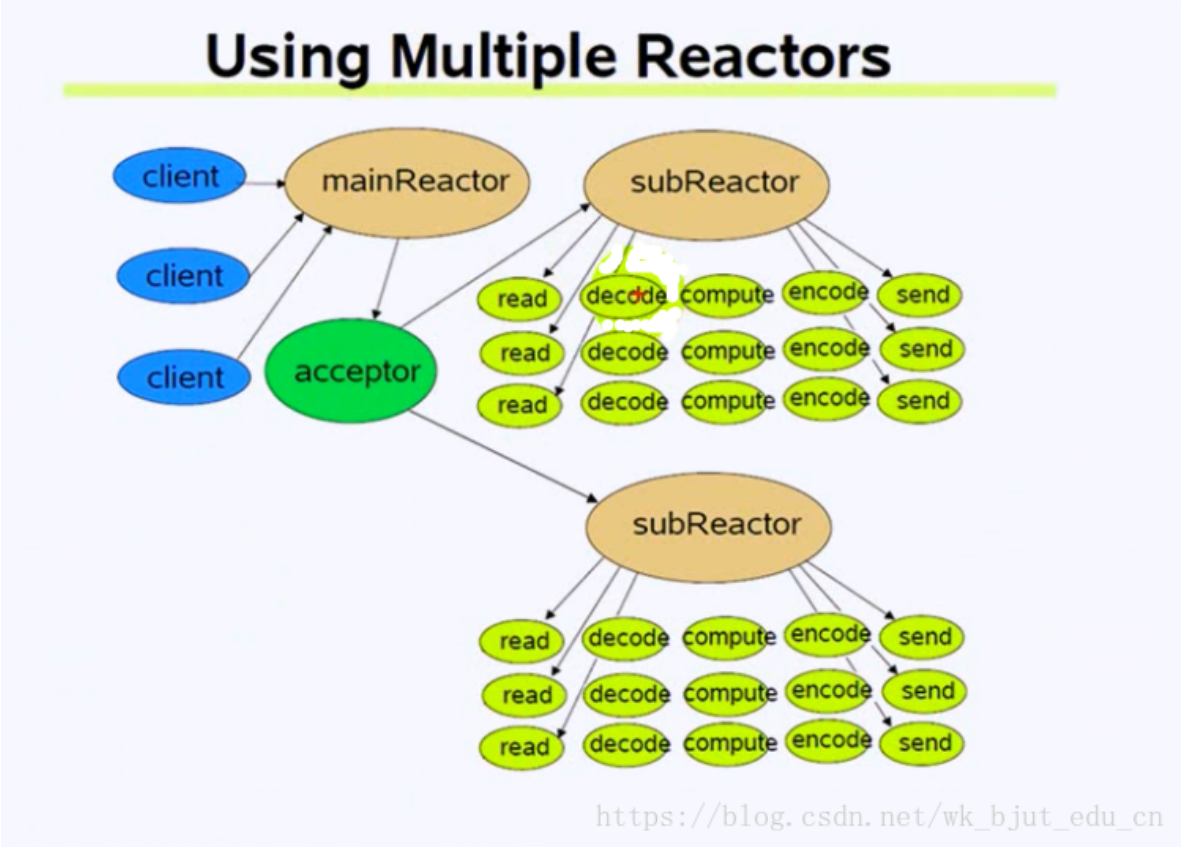
Reactor模型

> The reactor design pattern is an event handling pattern for handling service requests delivered concurrently to a service handler by one or more inputs. The service handler then demultiplexes the incoming requests and dispatches them synchronously to the associated request handlers.

重要组件：**Event事件、Reactor反应堆、Demultiplex事件分发器、Evanthandler事件处理器**



muduo库的Multiple Reactors模型如下：



epoll

select和poll的缺点

select的缺点：

1、单个进程能够监视的文件描述符的数量存在最大限制，通常是1024，当然可以更改数量，但由于select采用轮询的方式扫描文件描述符，文件描述符数量越多，性能越差；(在linux内核头文件中，有这样的定义：#define __FD_SETSIZE　1024

2、内核 / 用户空间内存拷贝问题，select需要复制大量的句柄数据结构，产生巨大的开销

3、select返回的是含有整个句柄的数组，应用程序需要遍历整个数组才能发现哪些句柄发生了事件

4、select的触发方式是水平触发，应用程序如果没有完成对一个已经就绪的文件描述符进行IO操作，那么之后每次select调用还是会将这些文件描述符通知进程

相比select模型，poll使用链表保存文件描述符，因此没有了监视文件数量的限制，但其他三个缺点依然存在。

以select模型为例，假设我们的服务器需要支持100万的并发连接，则在__FD_SETSIZE 为1024的情况下，则我们至少需要开辟1k个进程才能实现100万的并发连接。除了进程间上下文切换的时间消耗外，从内核/用户空间大量的句柄结构内存拷贝、数组轮询等，是系统难以承受的。因此，基于select模型的服务器程序，要达到100万级别的并发访问，是一个很难完成的任务。

epoll原理以及优势

epoll的实现机制与select/poll机制完全不同，它们的缺点在epoll上不复存在。

设想一下如下场景：有100万个客户端同时与一个服务器进程保持着TCP连接。而每一时刻，通常只有几百上千个TCP连接是活跃的(事实上大部分场景都是这种情况)。如何实现这样的高并发?

在select/poll时代，服务器进程每次都把这100万个连接告诉操作系统（从用户态复制句柄数据结构到内核态），让操作系统内核去查询这些套接字上是否有事件发生，轮询完成后，再将句柄数据复制到用户态，让服务器应用程序轮询处理已发生的网络事件，这一过程资源消耗较大，因此，select/poll一般只能处理几千的并发连接。

epoll的设计和实现与select完全不同。epoll通过在Linux内核中申请一个简易的文件系统（文件系统一般用什么数据结构实现？B+树，磁盘IO消耗低，效率很高）。把原先的select/poll调用分成以下3个部分：

1）调用epoll_create()建立一个epoll对象（在epoll文件系统中为这个句柄对象分配资源）

2）调用epoll_ctl向epoll对象中添加这100万个连接的套接字

3）调用epoll_wait收集发生的事件的fd资源

如此一来，要实现上面说是的场景，只需要在进程启动时建立一个epoll对象，然后在需要的时候向这个epoll对象中添加或者删除事件。同时，epoll_wait的效率也非常高，因为调用epoll_wait时，并没有向操作系统复制这100万个连接的句柄数据，内核也不需要去遍历全部的连接。

epoll_create在内核上创建的eventpoll结构如下：

```
struct eventpoll{
    ....
     /*红黑树的根节点，这颗树中存储着所有添加到epoll中的需要监控的事件*/
     struct rb_root   rbr;
     /*双链表中则存放着将要通过epoll_wait返回给用户的满足条件的事件*/
     struct list_head rdlist;
    ....
};
```

LT模式

内核数据没被读完，就会一直上报数据。

ET模式

内核数据只上报一次。

muduo采用的是LT

- 不会丢失数据或者消息
  - 应用没有读取完数据，内核是会不断上报的
- 低延迟处理
  - 每次读数据只需要一次系统调用；照顾了多个连接的公平性，不会因为某个连接上的数据量过大而影响其他连接处理消息
- 跨平台处理
  - 像select一样可以跨平台使用

muduo网络库的核心代码模块

Channel

fd、events、revents、callbacks　两种channel　listenfd-acceptorChannel　connfd-connectionChannel

Poller和EPollPoller - Demultiplex

std::unordered_map<int, Channel*> channels

EventLoop - Reactor

ChannelList activeChannels_;

std::unique_ptr poller_;

int wakeupFd;　-> loop
std::unique_ptr wakeupChannel;

Thread和EventLoopThread

EventLoopThreadPool

getNextLoop()：通过轮询算法获取下一个subloop　baseLoop

一个thread对应一个loop　=> one loop per thread

Socket

Acceptor

主要封装了listenfd相关的操作　socket　bind　listen　baseLoop

Buffer

缓冲区　应用写数据 -> 缓冲区 -> Tcp发送缓冲区 -> send

prependable　readeridx　writeridx

TcpConnection

一个连接成功的客户端对应一个TcpConnection　Socket　Channel　各种回调　发送和接收缓冲区

TcpServer

Acceptor　EventLoopThreadPool

ConnectionMap connections_;

扩展

1、TcpClient编写客户端类

2、支持定时事件TimerQueue　　链表/队列　　时间轮（libevent）　nginx定时器（红黑树）

3、DNS、HTTP、RPC

4、丰富的使用示例examples目录

5、服务器性能测试 - QPS　　涉及linux上进程socketfd的设置相关

- wrk　　linux上，需要单独编译安装　只能测试http服务的性能
- Jmeter　JDK　Jmeter 测试http服务　tcp服务　生成聚合报告