深入理解网络编程分享

By 公众号@极客重生

内容提要

- 网络编程全景
- Linux 网络I/O系统演进, Linux网络I/O模型
- Linux socket接口
- Linux网络IO框架（select，poll，epoll，io_uring）
- Linux 网络编程常见问题和异常处理
- 网络高性能编程模型
- 网络编程优化
- 开源网络IO框架学习

网络编程全景

网络应用程序

HTTP gRPC SSH redis mysql ngnix

DHCP QUIC DNS

ping

tcpdump ARP LLDP

iproute2

TCP socket

发送接收TCP数据包

UDP socket

发送接收UDP数据包

unix domain socket

进程间通信

raw socket

socket(AF_INET, SOCK_RAW, IPPROTO_TCP|IPPROTO_UDP|IPPROTO_ICMP)

发送接收ip数据包

packet socket

ETH_P_IP 0x800 只接收发往本机mac的ip类型的数据帧

ETH_P_ARP 0x806 只接受发往本机mac的arp类型的数据帧

ETH_P_ARP 0x8035 只接受发往本机mac的rarp类型的数据帧

sockaddr_ll

发送接收以太网数据帧

netlink socket

和内核通信

协议族或地址族

AF_INET
AF_INET6
AF_PACKET
AF_UNIX
AF_NETLINK

套接字类型

SOCK_STREAM = 1,
SOCK_DGRAM = 2,
SOCK_RAW = 3,

AF_INET
IPPROTO_TCP
IPPROTO_UDP
IPPROTO_ICMP
....

协议

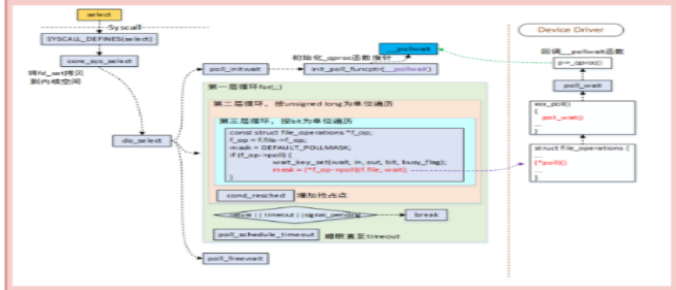
AF_PACKET
ETH_P_IP
ETH_P_ARP
...

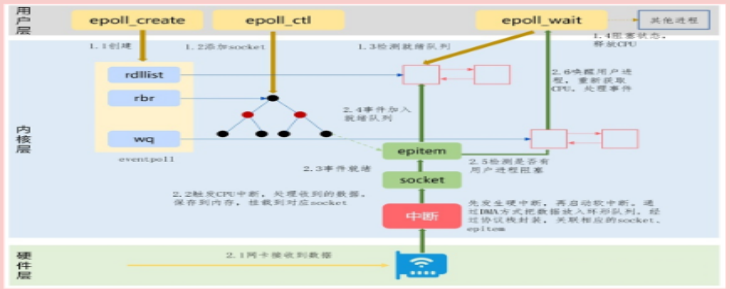
socket错误码

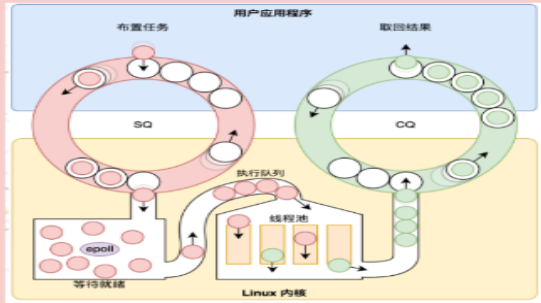
socket核心接口

socket API

Linux网络IO框架







select

poll

epoll

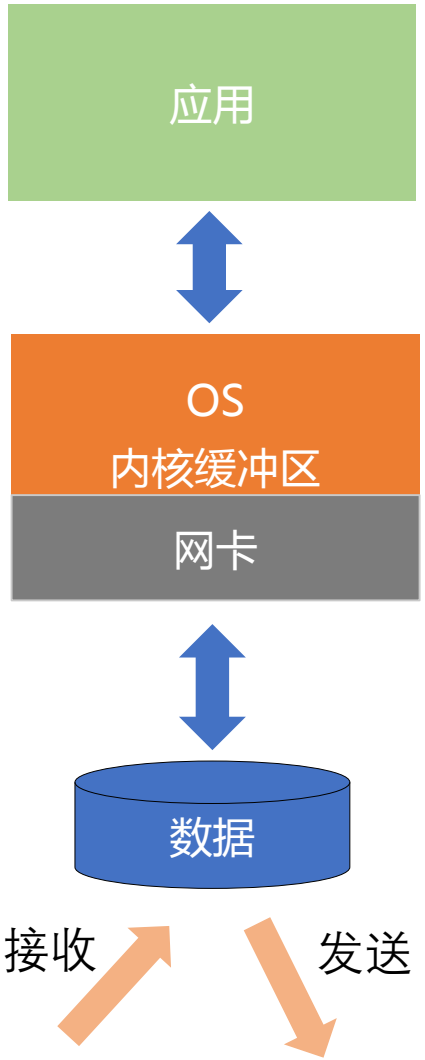
io_uring

网卡驱动

混杂模式
ALL

正常模式
MAC匹配 || 广播

Linux 网络I/O系统演进



网络编程的问题有哪些

- CPU利用率问题
- 数据copy问题
- 上下文切换问题
- C10K问题
- C10M问题

- 阻塞 IO(BIO)
- 非阻塞 IO(NIO)
- IO 多路复用1.0(select)
- IO 多路复用1.5(poll)
- IO 多路复用2.0(epoll)
- Direct I/O：绕过 page cache
- 异步 IO（AIO）
  - 传统 Linux AIO
  - io_uring

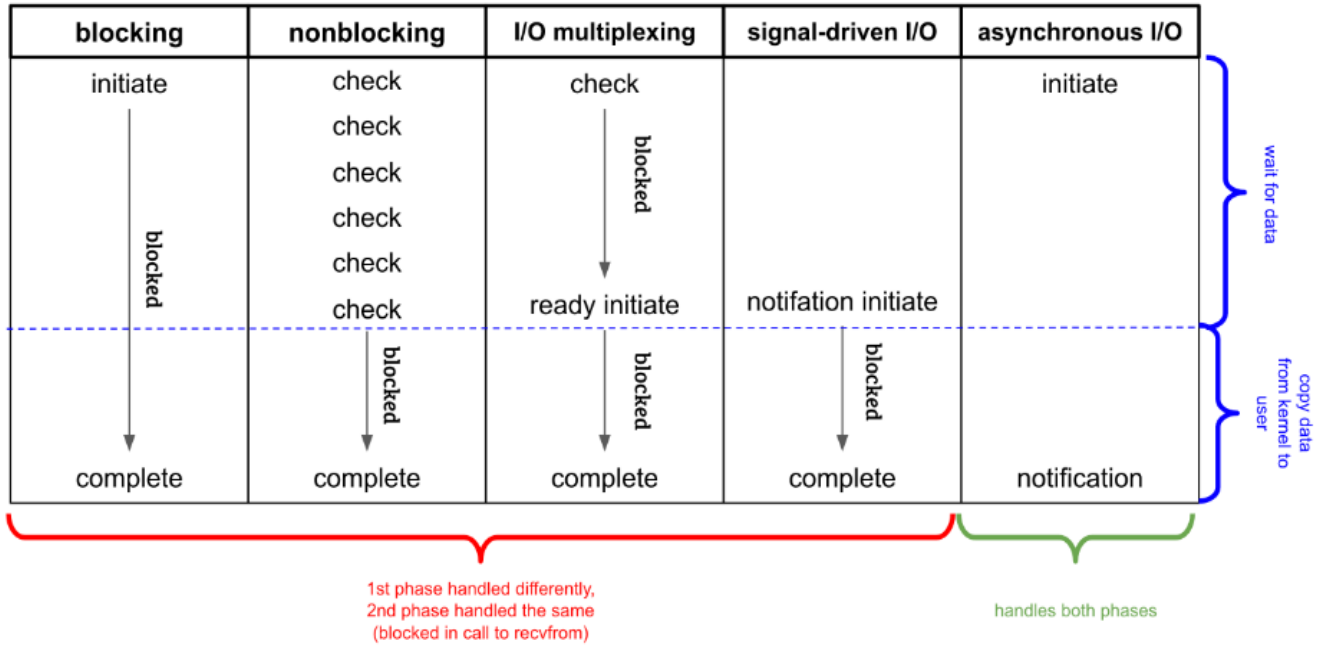
不断改进当前网络IO机制缺陷，不断优化性能

Linux网络I/O模型

一次网络IO可以抽象为两个重要阶段

第一阶段：等待数据从网络中到达，并拷贝到内核中某个缓冲区
Waiting for the data to be ready

第二阶段：把数据从内核态的缓冲区拷贝到用户态的应用进程缓冲区来
Copying the data from the kernel to the process



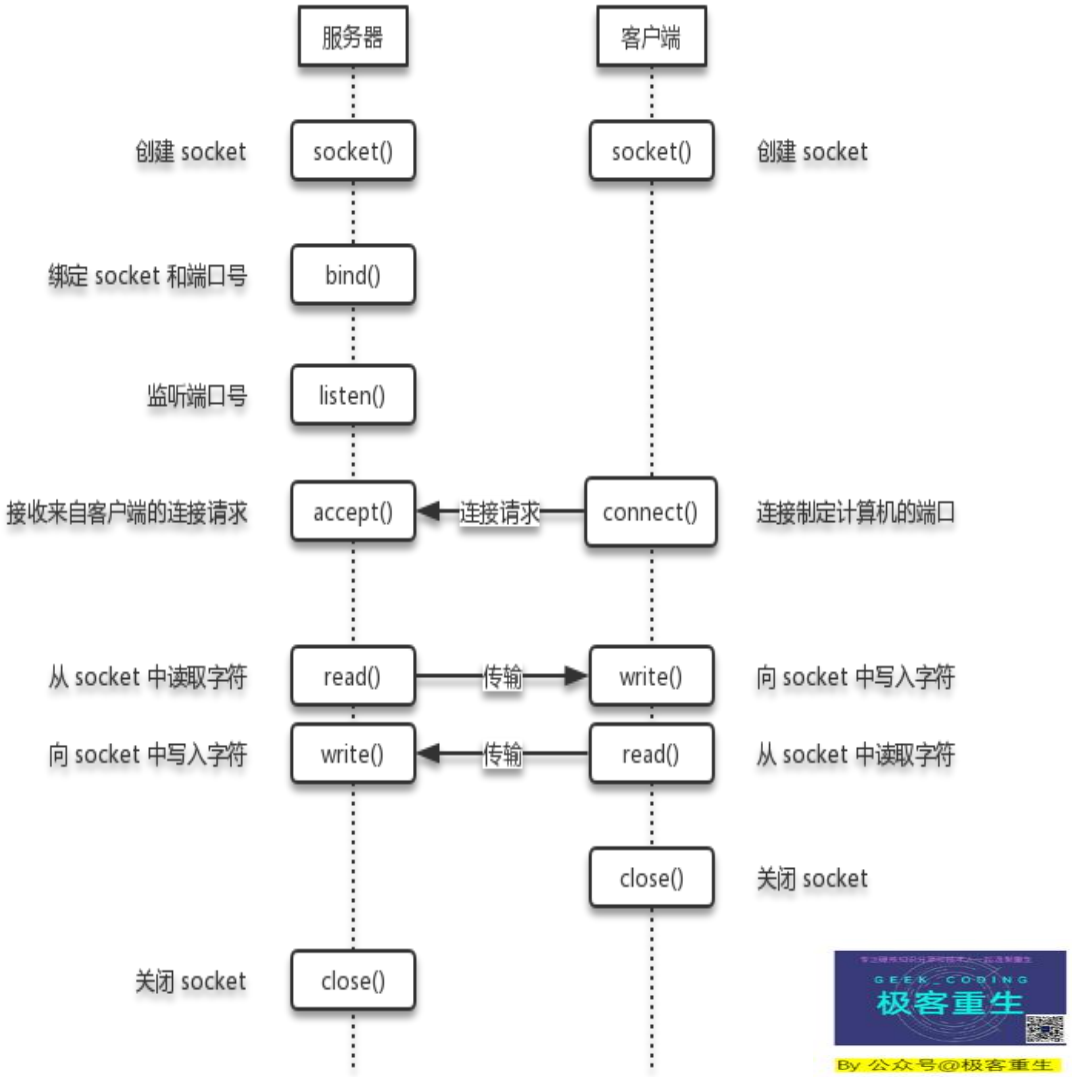
- **阻塞与非阻塞阻塞**：**主要线程内调用，是指进程执行状态**，主要区别在第一阶段等待数据的时候，阻塞会导致进程睡眠直到有数据到来（**进程无法执行，就是阻塞意思**），非阻塞不会，一般非阻塞需要轮训。

- **同步与非同步**：主要线程间调用，调用者与被调用者关系，区别就在于第二个步骤是否阻塞，如果不阻塞，而是操作系统帮你做完IO操作再将结果返回给你，那么就是异步IO，同步IO具有实时性（**阻塞可以是实现同步的一种手段**），但异步IO实现生产者和消费者解耦，具有高吞吐。

五大IO模型：

- **阻塞I/O**(Blocking I/O model) : 同步阻塞I/O，两阶段都阻塞。
- **非阻塞I/O**(Non-Blocking I/O model) **:** 同步非阻塞I/O，第一阶段非阻塞，第二阶段阻塞。
- **I/O 多路复用**(I/O Multiplexing model): 同步 I/O，第一阶段可以阻塞也可以非阻塞，第二阶段阻塞。
- **信号驱动IO(**Signal-driven I/O): 同步信号机制I/O，第二阶段阻塞。
- **异步IO(**Asynchronous I/O) :两阶段都非阻塞。

Linux socket接口



Socket类型和场景

- TCP socket：tcp字节流通信
- UDP socket：udp数据报通信
- Raw socket：基于ip层报文通信
- Packet socket：穿透协议栈报文通信
- Unix Domain socket：进程间通信
- Netlink socket：user/kernel通信

Socket API

- Socket创建:socket
- Socket控制：bind,connect,listen, accept
- 数据收发: write,read,send,recv, sendto,recvfrom
- socket关闭:close,shutdown
- IO多路复用：select/poll/epoll/io_uring
- 网络字节序/大小端转化：htonl,htons
- 设置option: setsockopt
  - SO_SNDBUF/SO_RCVBUF
  - SO_REUSEADDR/SO_REUSEPORT
  - TCP_NODELAY

Linux Socket接口—TCP编程

- connect
  - 非阻塞connect 场景
  - 被中断的connect场景
- accept
  - 新建soceket：多线程处理
  - 阻塞问题：单独线程处理或者设置非阻塞模式
  - 惊群问题： EPOLLEXCLUSIVE 或者 SO_REUSEPORT
- bind
  - SO_REUSEADDR：TIME_WAIT状态的socket复用
  - SO_REUSEPORT：
    - 允许将多个socket绑定到相同的地址和端口
    - 新的客户连接请求(由accept返回)负载均衡到同一地址和端口的socket
- Read
  - 阻塞情况：如果发现没有数据在接收缓冲区中会一直等待，只有有数据就会返回。
  - 对于非阻塞的模式： 如果缓冲区没有数据，调用将立即返回EWOULDBLOCK错误（在Linux上EWOULDBLOCK与EAGAIN等价），
  - 读取完整数据：一般循环读取/MSG_WAITALL优化
  - 粘包问题（数据无边界性）：自定义边界标记
- Write
  - 阻塞的模式：直到全部数据写完内核的发送缓冲区才返回；
  - 对于非阻塞的模式：如果发送缓冲区没有空间，函数将立即返回EWOULDBLOCK错误。
  - write成功返回：返回实际写入字节，buf中的数据被复制到了kernel中的发送缓冲区，数据是否发送网络未定。

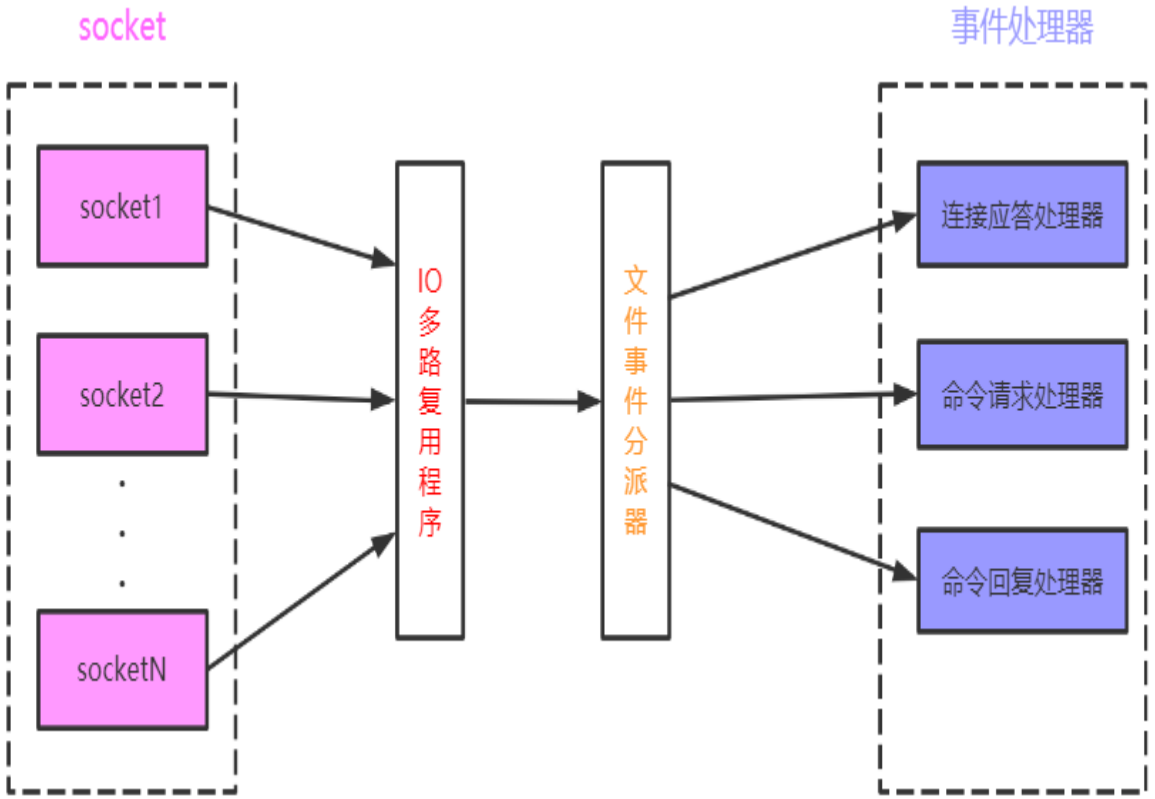
- Listen
  - 参数backlog：accept队列大小

LISTEN 状态:
Recv-Q 表示的当前等待服务端调用 accept 完成三次握手的 listen backlog 数值，也就是说，当客户端通过 connect() 去连接正在 listen() 的服务端时，这些连接会一直处于这个 queue 里面直到被服务端 accept()；
Send-Q 表示的则是最大的 listen backlog 数值，这就就是上面提到的 min(backlog, somaxconn) 的值

Linux Socket接口—UDP编程

- bind
  - 如果client没有bind本地local address，那么在发送UDP数据包的时候，可能是不同的Port
  - UDP端口分配
- connect
  - UDP的"连接性"，提高发送效率
  - 高并发场景：增加系统稳定性
- recvfrom
  - connect + recv vs recvfrom
  - 数据包有界
  - 数据包无序
  - 大包/缓冲区大小
- sendto
  - connect + send vs sendto
  - 大包/缓冲区大小
- option
  - SO_REUSEADDR: 端口复用
  - SO_REUSEPORT
    - UDP的负载均衡
    - 多核并发
    - 已知问题

网络IO框架-I/O Multiplexing（IO多路复用）

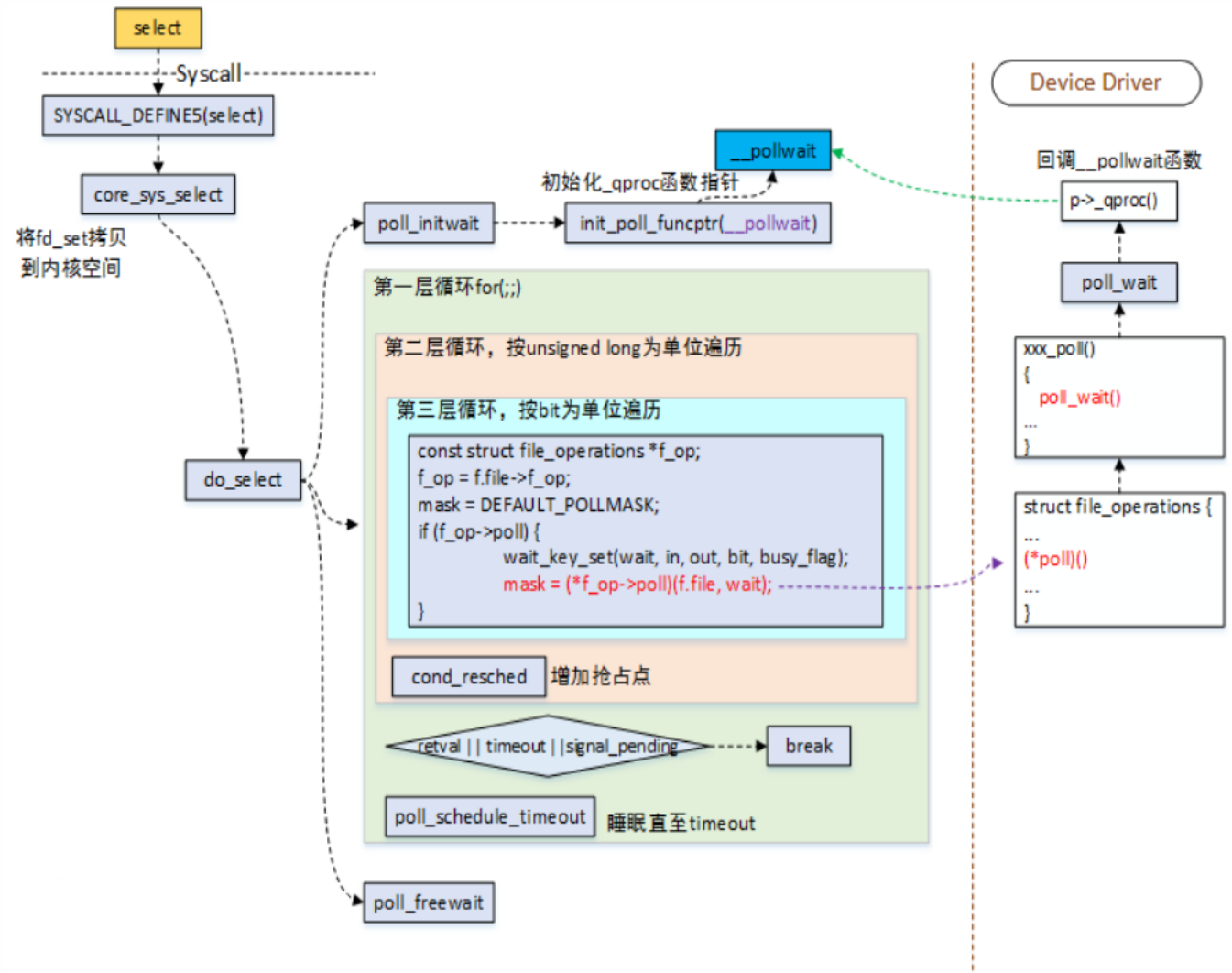


- **事件驱动 (Event-driven IO)**

常见的I/O Multiplexing 实现:

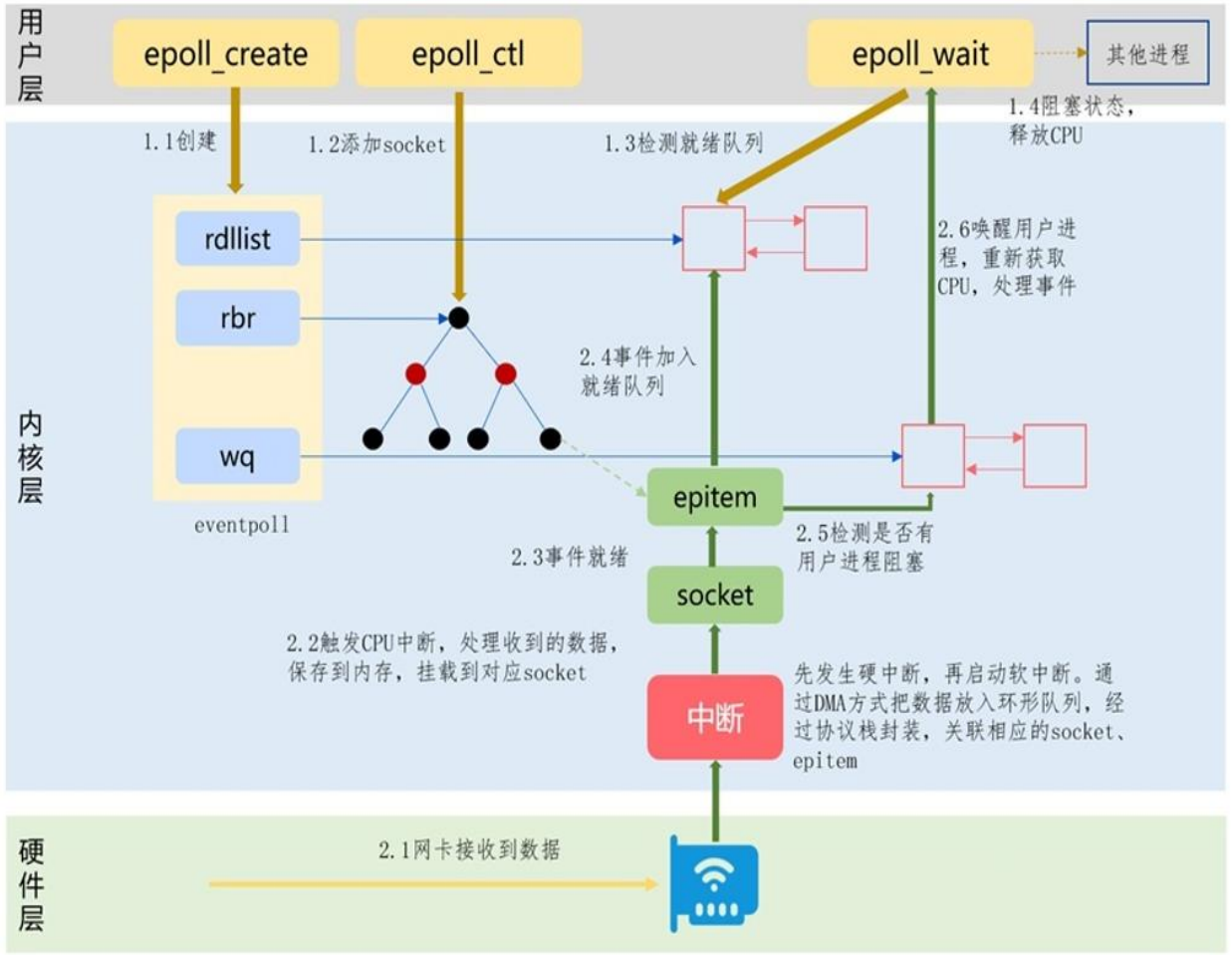
- Linux : **select、poll、epoll，linux-aio，io_uring**
- Mac/FreeBSD : **kqueue**
- Windows : **IOCP (Input/Output Completion Port)**

Linux网络IO框架-select，poll



- 采用轮询方式检测就绪事件，时间复杂度O(n)
- Select最大连接数有限，适合连接数量小且活跃
- select 的timeout 参数精度为微秒，可以作为精确定时器，更加适用于实时性要求比较高的场景
- select中使用的文件描述符集合是采用的固定长度为1024的BitMap结构的fd_set，而poll换成了一个pollfd结构没有固定长度的数组，这样就没有了最大描述符数量的限制（但会受到系统文件描述符限制）

Linux网络IO框架-epoll



- epoll句柄采用红黑树管理，提升socket句柄CURD性能
- 唤醒相关文件句柄睡眠队列的entry，调用其回调
- 就绪事件用rdllist链表记录。
- 唤醒epoll睡眠队列的task，上报就绪socket

编程API：

1、调用 epoll_create 建立一个 epoll 对象（在epoll文件系统中给这个句柄分配资源）；

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 收集发生事件的连接。

Linux网络IO框架-epoll

ET和LT场景理解

ET模式（边缘触发）

- 只有数据到来才触发，不管缓存区中是否还有数据，缓冲区剩余未读尽的数据不会导致epoll_wait返回；
- 边沿触发模式很大程度上降低了同一个epoll事件被重复触发的次数，所以效率更高；
- 对于读写的connfd，边缘触发模式下，必须使用非阻塞IO，并要一次性全部读写完数据。
- ET的编程可以做到更加简洁，某些场景下更加高效，但另一方面容易遗漏事件，容易产生bug；

LT 模式（水平触发，默认）

- 只要有数据都会触发，缓冲区剩余未读尽的数据会导致epoll_wait返回；
- LT比ET多了一个开关EPOLLOUT事件(系统调用消耗，上下文切换)的步骤；
- 对于监听的sockfd，最好使用水平触发模式（参考nginx），边缘触发模式会导致高并发情况下，有的客户端会连接不上，**LT适合处理紧急事件**；
- 对于读写的connfd，水平触发模式下，阻塞和非阻塞效果都一样，不过为了防止特殊情况，还是建议设置非阻塞；
- LT的编程与poll/select接近，符合一直以来的习惯，不易出错；

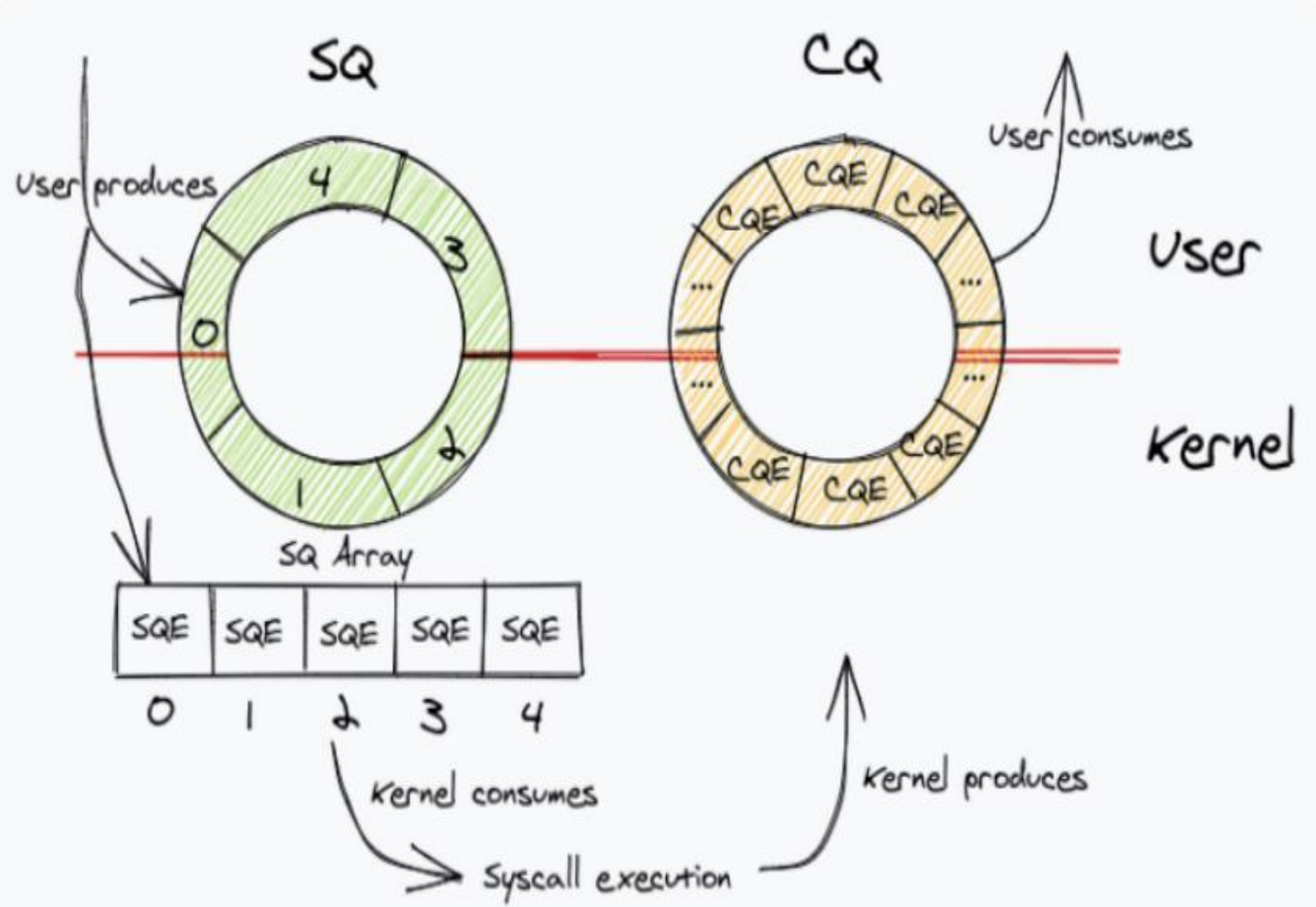
理解epoll不足之处

1.定时的精度不够，只到5ms级别，select可以到0.1ms；
2.当连接数少并且连接都十分活跃的情况下，select和poll的性能可能比epoll好；
3.epoll_ctrl每次只能够修改一个fd（kevent可以一次改多个，每次修改，epoll需要一个系统调用，不能 batch 操作，可能会影响性能）。
4.可能会在定时到期之前返回，导致还需要下一个epoll_wait调用。

其他编程需要注意地方：

- 需要深入理解epoll LT和ET方式下的读写差别，怎么优雅地处理各种错误；
- 需要关注多线程负载均衡，惊群效应等问题，要用 epoll 实现负载均衡并且避免数据竞争，必须掌握好，下面这两个标志：
  - EPOLLONESHOT ：一个事件发生并读取后，文件自动不再监控
  - EPOLLEXCLUSIVE：解决epoll引起的accept惊群, 和SO_REUSEPORT对比

Linux网络异步IO框架-io_uring



io_uring是 2019 年 **Linux 5.1** 内核首次引入的高性能**异步 I/O 框架**

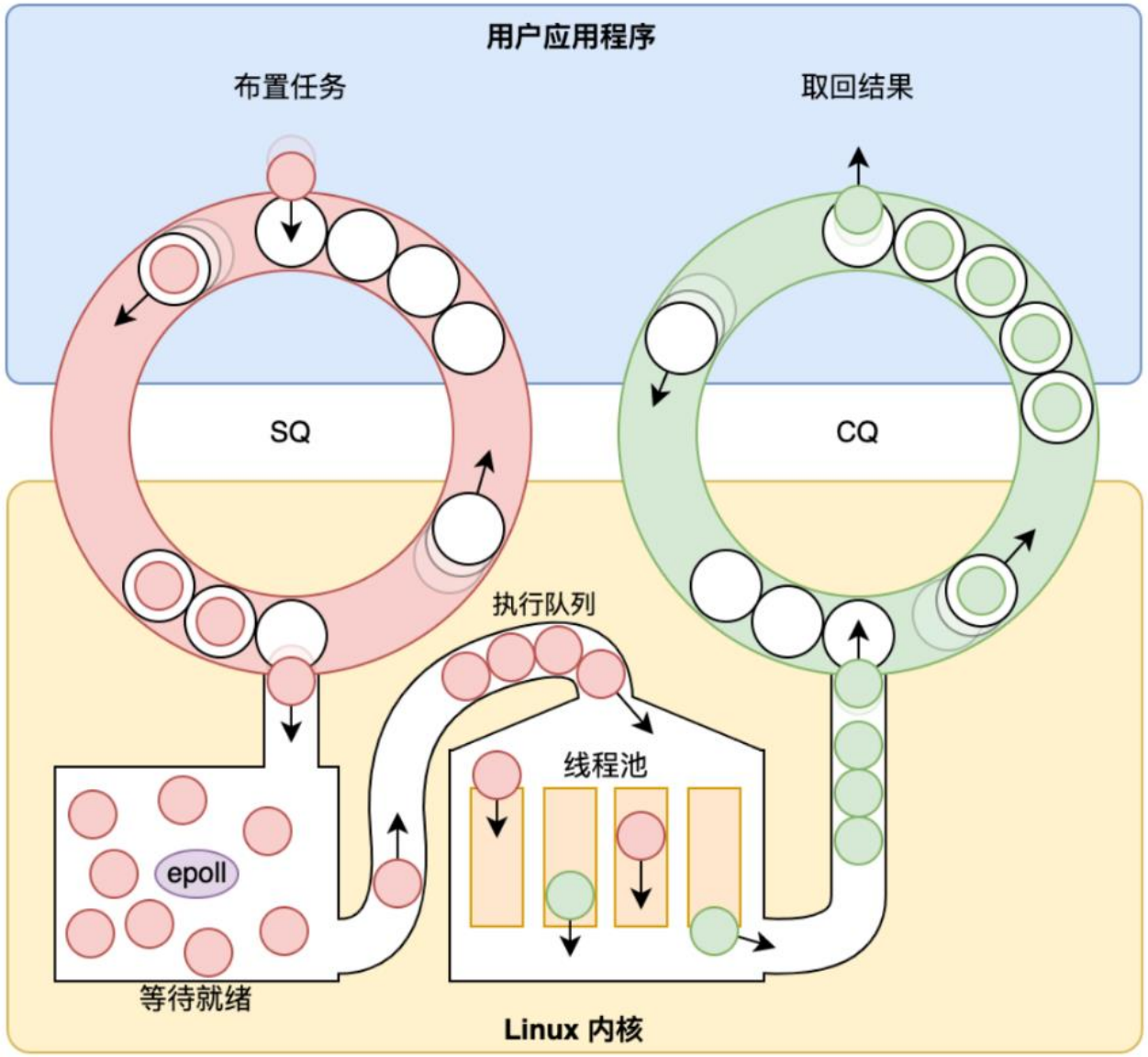
核心数据结构

SQ - Submission Queue：提交队列，这是在内核态的一整块连续的内存空间存储的环形队列。用于存放将执行的操作数据项。
CQ - Completion Queue：完成队列，这是在内核态的一整块连续的内存空间存储的环形队列。用于存放执行完成后的结果。
SQE - Submission Queue Entry：提交队列项，这是储存在SQ中的数据项。
CQE - Completion Queue Entry：完成队列项，这是储存在CQ中的数据项。
Ring - Ring：环：SQ和CQ都是环形队列结构，Ring用来代表一个io_uring的实体。

核心设计：

- 应用程序用内存映射（mmap）的方式拿到两条与内核共享的环状队列。
- 减少上下文切换，减少网络IO的系统调用，初步测试，io_uring能比普通epoll快出5%至40%。
- 对比传统Linux AIO只支持 direct I/O 模式的存储IO，主要用在数据库这一细分领域。io_uring 支持存储IO和网络IO，也支持更多的异步系统调用，在设计上是真正的异步 I/O。
- io_uring 提供了新的系统调用和用户空间 API，因此需要应用程序做改造

Linux网络异步IO框架-io_uring



IORING_FEAT_FAST_POLL
网络编程新利器，向 epoll 等传统基于事件驱动的网络编程模型发起挑战，向用户态提供真正的异步编程API。

系统调用 API

- **io_uring_setup**
  用于建立一个io_uring的实体
- **io_uring_enter**
  用于将SQE(Submission Queue Entry)提交到SQ(Submission Queue)中去。
- **io_uring_register**
  用于预注册读写文件

Linux网络IO框架对比

网络IO框架	select	poll	epoll	Linux-aio	io_uring
底层数据结构	数组	链表	红黑树与链表	环形队列	双环形队列
事件处理	采用轮询方式检测就绪事件，时间复杂度O(n)	采用轮询方式检测就绪事件，时间复杂度O(n)	采用回调方式检测就绪事件，时间复杂度O(1)	异步通知，时间复杂度O(1)	异步通知，时间复杂度O(1)
最大连接数	1024/2048(64位)	无限制	无限制	无限制	无限制
工作模式	LT	LT	LT和ET	LT和ET	LT和ET
运行模式	同步	同步	同步	大部分异步	完全异步
适用场景	• 连接数量小且活跃； • select 的timeout参数精度为微秒，更加适用于实时性要求比较高的场景 • 定时器	连接数量多，平台对实时性要求不高	连接数量多，且活跃的连接数量少，互联网后台服务器场景	Direct I/O，数据库异步IO	通用高性能异步IO编程 RDMA，持久内存等高速场景

网络编程健壮性--异常情况处理

网络编程异常处理

connect函数返回状态及其原因

- **ETIMEOUT**: 服务器繁忙， 或者网络不通，或者网络质量很差，导致三次握手失败。
- **ECONNREFUSED**，表示服务端在我们指定的端口没有进程等待与之连接，ip地址存在，并无对应的监听端口进程
- **EHOSTUNREACH,ENETUNREACH**：表示目标主机不可达 ， 是个软错误（路由器x跳以后找不到能到达的路由，路由返回不可达）
- **EINPROGRESS /EAGAIN** (unix domain)：在一个 TCP 套接字被设置为非阻塞之后调用 connect ， connect 会立即返回 **EINPROGRESS** 错误，表示连接操作正在进行中，但是仍未完成
- **EINTR** ：阻塞式套接口上调用 connect ， 在 TCP 的三次握手操作完成之前被捕获信号中断了，返回 EINTR。

recv函数返回状态及其原因

- EAGAIN/EWOULDBLOCK : 非阻塞，没有数据返回。
- EINTR : 操作完成之前被捕获信号中断了，返回 EINTR。
- 返回值<0时并且(errno == EINTR || errno == EWOULDBLOCK || errno == EAGAIN)的情况下认为连接是正常的，继续接收否则认为连接异常，需要关闭。
- >0 :接收到数据大小（UDP是一个完整报文）。
- = 0：对端关闭连接。（UDP收到空包）

send函数返回状态及其原因

- 阻塞模式与非阻塞模式下是否send返回值 < 0 && (errno == EINTR || errno == EWOULDBLOCK || errno == EAGAIN)表示暂时发送失败，需要重试。
- 如果send返回值 <= 0, && errno != EINTR && errno != EWOULDBLOCK && errno != EAGAIN时，连接异常，需要关闭。
- 如果send返回值 > 0则表示发送了多少数据到内核缓冲区
- send返回值=0 连接关闭或者发送空包

epoll_wait返回状态中的错误处理

- EPOLLRDHUP ：对端关闭连接或者shutdown写入半连接
- EPOLLIN/ EPOLLOUT ：文件可读可写
- EPOLLERR： 文件上发上了一个错误。
- EPOLLHUP：通常情况下EPOLLHUP表示的是本端挂断，造成这种事件出现的原因有很多。
- 出现EPOLLHUP、EPOLLERR，连接都应该关闭
- 返回-1，errno为EINTR，忽略这种错误,重新epoll_wait

Linux 网络编程常见问题

网络编程常见问题

- 大小端问题：协议字段处理，htonl或htons
- sigpipe信号
- 缓冲区大小问题
- 非阻塞编程和阻塞编程注意事项
- 多进程问题：父子进程网络资源管理/SOCK_CLOEXEC
- 多线程问题，锁问题，并发优化
- 超时情况处理
- 长连接和短连接问题
- 异步编程问题
- 性能优化问题

高性能网络编程模型

网络编程模型

单进程/线程 Reactor
- redis

单 Reactor 多线程模型
- Java NIO
- Boost.Asio

多Reactor多线程模型
- Memcached
- netty
- Gnet

多 Reactor 多进程
- Nginx

Proactor模型
- 异步io_uring
- Boost.Asio
- 异步io + 协程

Reactor模型和Proactor模型

Reactor模型
- Reactor 模式本质上指的是使用 I/O 多路复用(I/O multiplexing) + 非阻塞 I/O(non-blocking I/O) 的模式。
- 业界实例：
  - Nginx
  - Java NIO Netty
  - redis
  - Memcached

Proactor模型
- 异步IO
- Boost.Asio

网络编程优化

网络IO优化

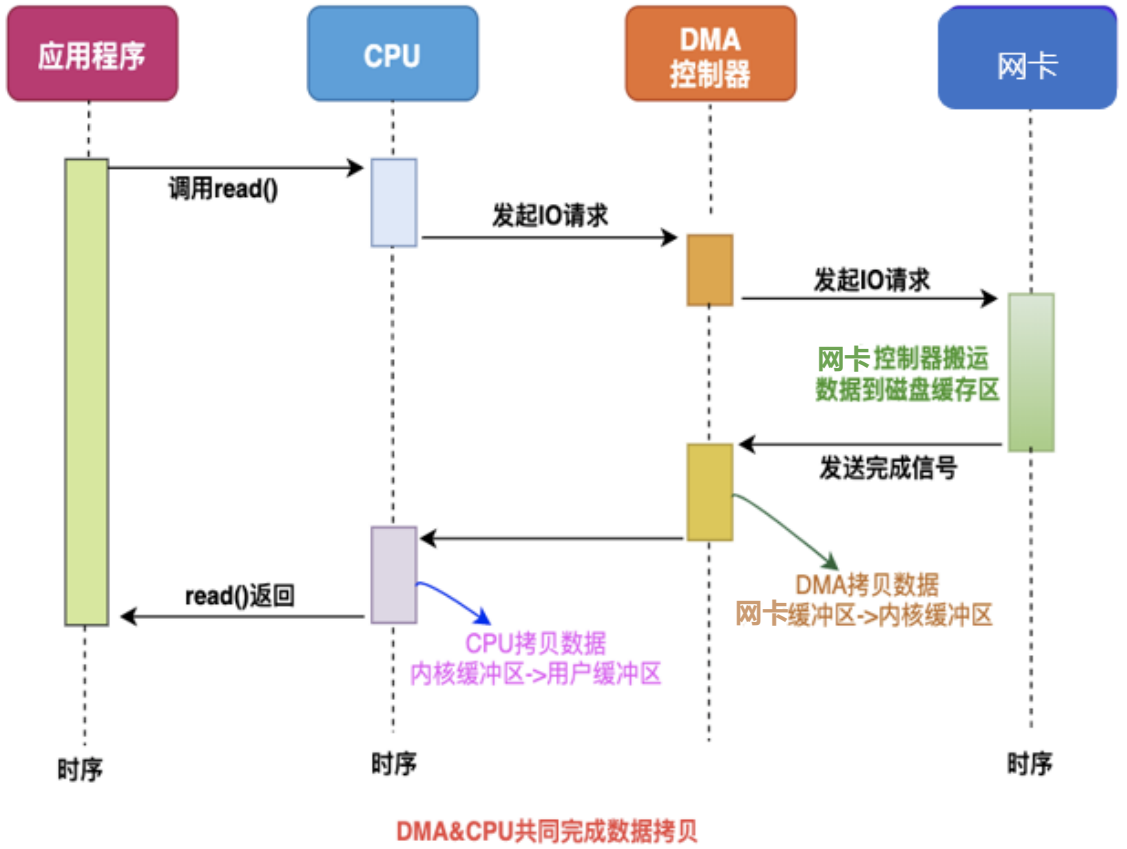
IO加速
- Bypass-kernel：
  - 向上offload：DPDK/SPDK：f-stack等
  - 向下offload：RDMA
- 硬件
  - FPGA/P4

CPU并发优化
- 多进程， 多线程， 协程
- 多核编程，绑核独占
- 无锁/per CPU设计

减少cachemiss
- 预取/分支预测/亲和/局部性原理
- 工具：PCM/perf
- 大页

Linux网络系统优化：
- 网卡offload优化：cksum/GRO/GSO
- 并行优化: RSS/RPS/RFS/aRFS/XPS/SO_REUSEPOR
- IO框架：select/poll/epoll/io_uring
- 内核调参：
  - 调整驱动队列backlog
  - 调整tcp队列大小
  - 调整sock缓冲区大小
  - 调整tcp状态优化

减少重复操作
- 池化技术：内存池, 线程池，连接池等
- 缓存技术
- 零拷贝优化



DMA&CPU共同完成数据拷贝

The C10K/C10M problem, 高性能网络应用

开源网络框架学习

网络IO开源框架

C语言
- Libevent
- Redis
- nginx

C++语言
- boost::asio

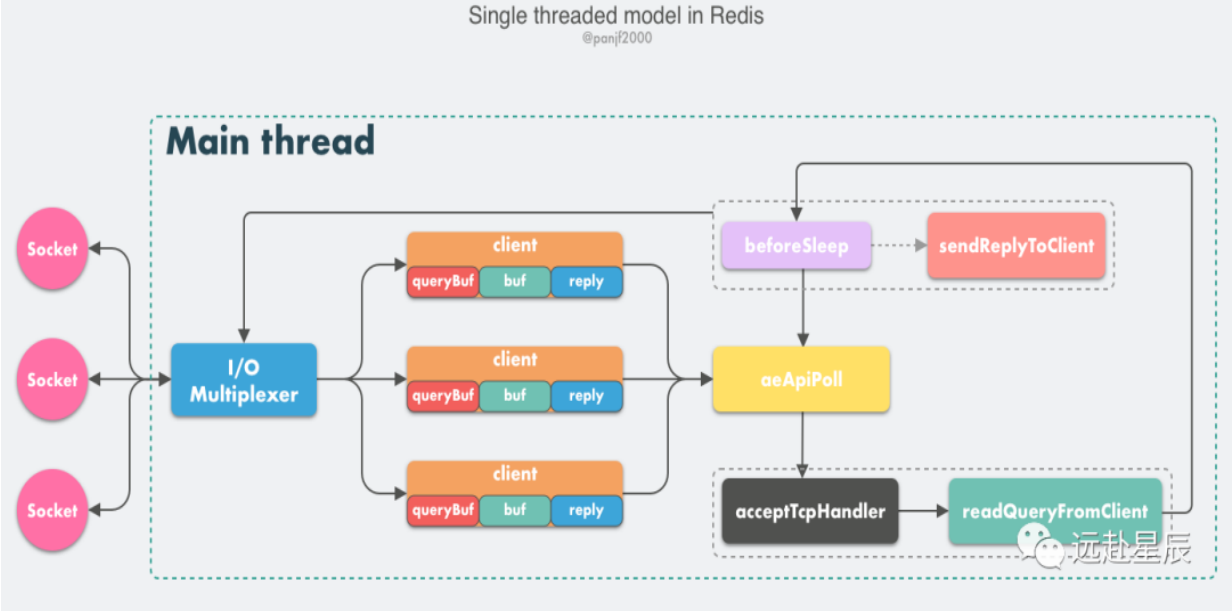
Java语言
- netty

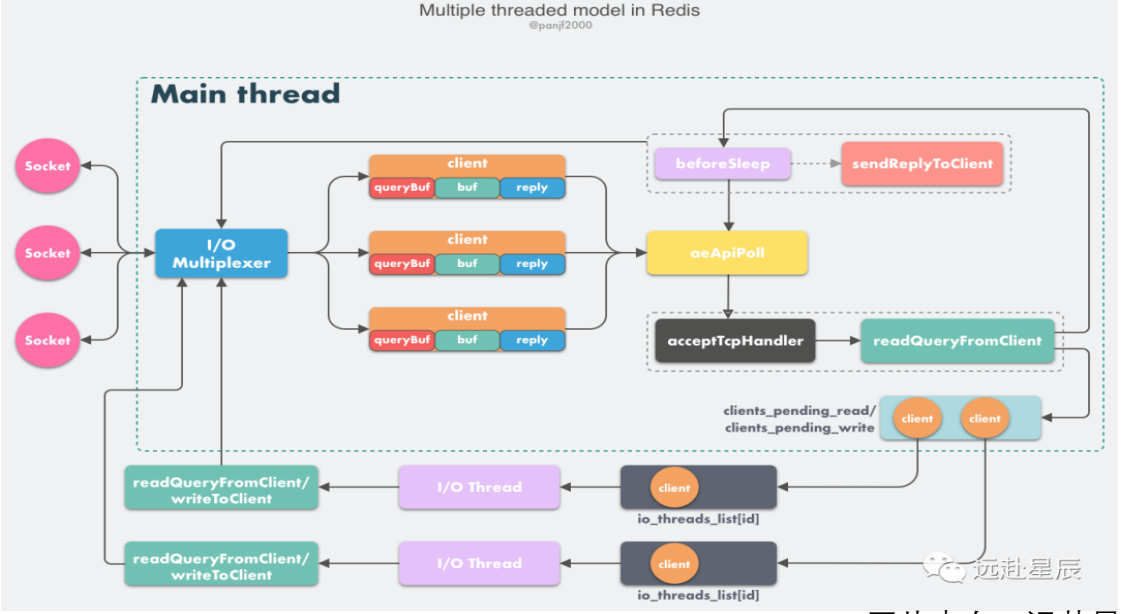
Go语言
- gnet
- netpoll

开源网络框架学习

开源网络框架	语言	网络模型	网络IO	核心技术
redis	C	单进程/线程Reactor	epoll/kqueue/iocp	
nginx	C	多 Reactor 多进程	epoll/kqueue/iocp	
boost.asio	C++	Reactor多线程模型/Proactor模型	epoll/kqueue/iocp	
netty	Java	多Reactor多线程模型	epoll/kqueue/iocp	
gnet	Go	多Reactor多线程模型	epoll kqueue io_uring	
netpoll	Go	多Reactor多线程模型	epoll kqueue	

开源网络框架学习-redis





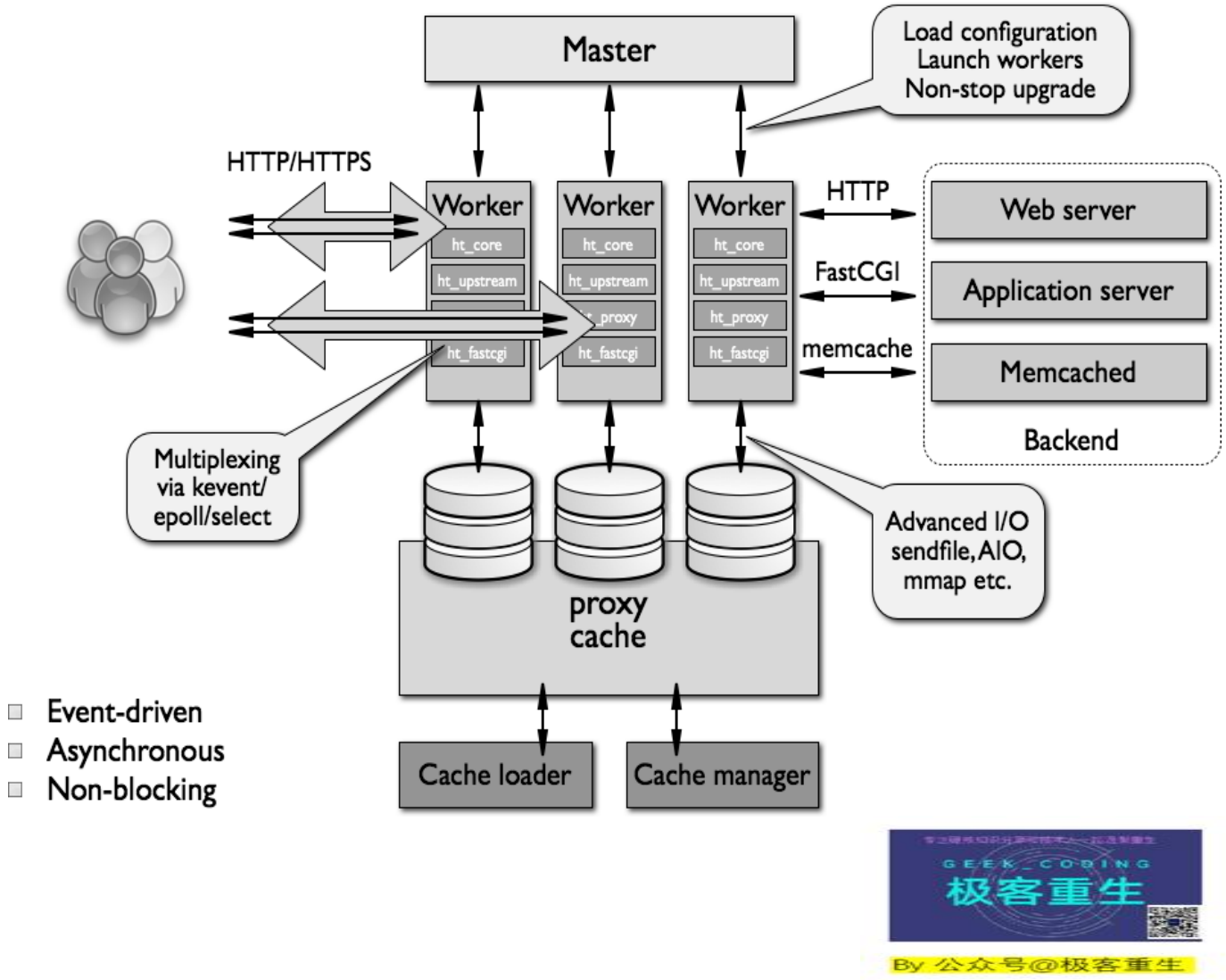
图片来自：远赴星辰

- **In-memory**：基于内存实现，而非磁盘
- **数据结构**：基于不同业务场景的高效数据结构
- **线程模型**：单线程来执行的，避免了不必要的上下文切换和锁竞争
- **I/O 模型**：基于I/O多路复用模型(epoll)，非阻塞的I/O模型
- **数据编码**：根据实际数据类型，选择合理的数据编码
- **Hash结构**：Redis 本身是一个全局哈希表，时间复杂度是O(1)，另外为了防止

- **数据结构**：优化数据结构底层实现，Hash，List， Zset的底层数据结构用listpack替换了ziplist。
- **线程模型**：核心的线程（Execute Command）还是单线程，多线程是指网络IO（socket）读写的多线程化
- **性能优化：**各种小细节的优化，降低内存使用，降低延迟时间

Redis 5.0之前 → **Redis 6.0之后**

开源网络框架学习-nginx



Master负责管理worker进程，worker进程负责处理网络事件。**整个框架被设计为一种依赖事件驱动、异步、非阻塞的模式**。

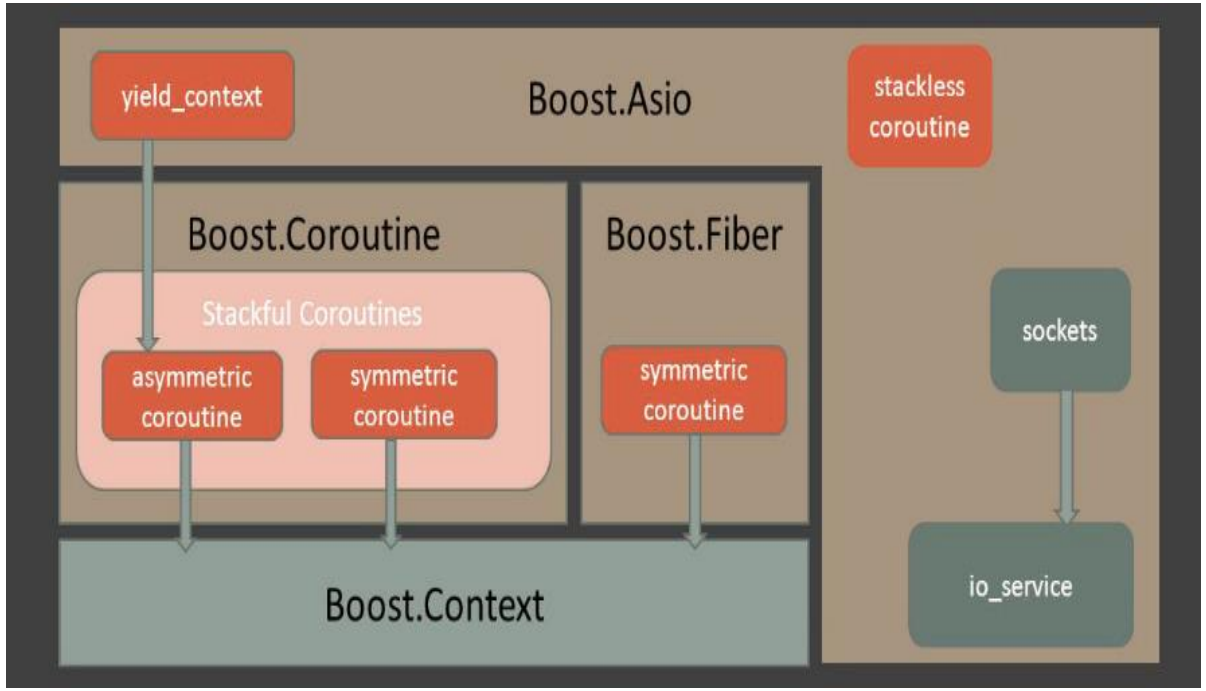
- **并发技术**
  - 可以充分利用多核机器，增强并发处理能力。
  - 多worker间可以实现负载均衡。
- **进程池**

  Master监控并统一管理worker行为。在worker异常后，可以主动拉起worker进程，从而提升了系统的可靠性。并且由Master进程控制服务运行中的程序升级、配置项修改等操作，从而增强了整体的动态可扩展与热更的能力。
- **高性能IO框架**

  Worker进程在处理网络事件时，依靠epoll模型，来管理并发连接，实现了事件驱动、异步、非阻塞等特性
- **无锁设计**

  采用多进程，可以让互相之间不会影响。一个进程异常崩溃，其他进程的服务不会中断，提升了架构的可靠性。进程之间不共享资源，不需要加锁，所以省掉了锁带来的开销。进程数已经等于核心数，减少切换代价。

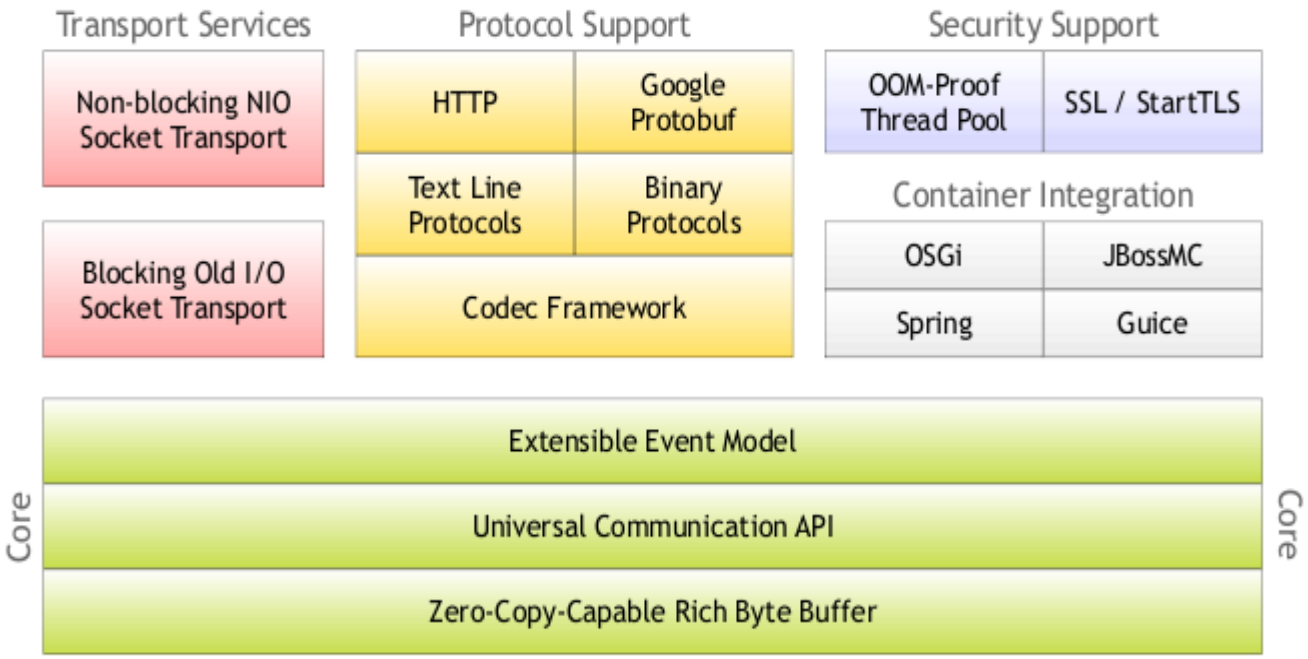
开源网络框架学习-boost.asio

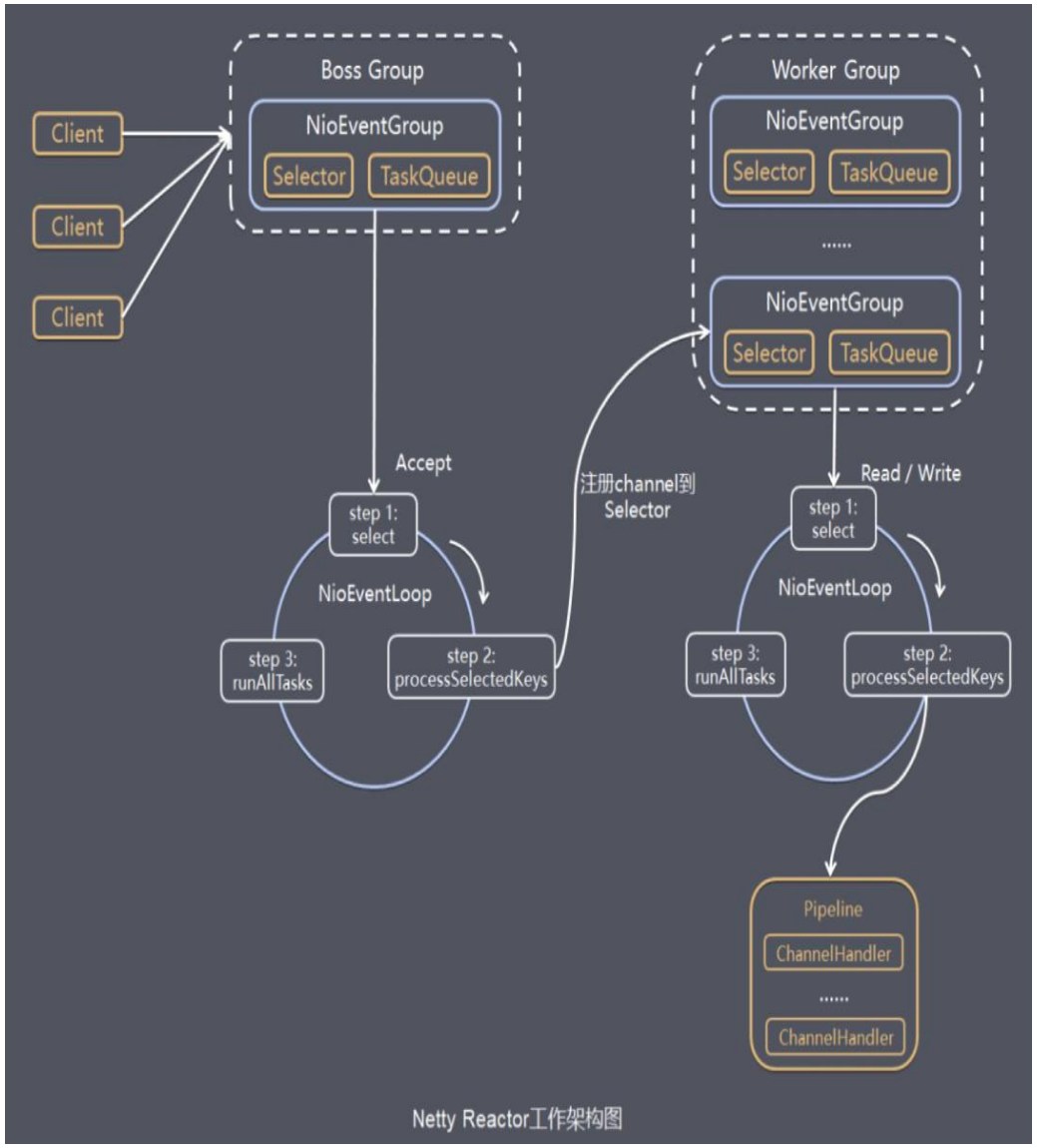


boost.asio是Boost库中非常著名的I/O组件，是用于网络和低层IO编程的跨平台C++库，为开发者提供了C++环境下稳定的异步模型。其在性能、移植性、扩展性等方面均为人称道，甚至被很多业内人士称为“网络神器”。asio是目前唯一有希望进入C++标准库以弥补标准库在网络方面的缺失的C++网络库，因此对asio的学习在某种意义上可以说是学习C++网络编程的必修课。

- 支持同步和异步IO模型，可以在不支持真正异步IO平台上以Reactor的形式实现 Proactor设计模式，统一异步IO编程模型。
- 池化技术：
  - 对象池，对象池可以避免对象频繁创建和销毁
  - 线程池设计，多线程无锁接口Strands
  - 内存Buffer池，支持scatter-gather批量操作。
- 提供有栈和无栈，对称和非对称的协程支持
- 提供流读写快捷操作接口
- 自定义内存分配算法
- 良好的跟踪调试
- boost库asio 为了适应不同平台使用采用了策略模式
- 减少虚函数使用（比如采用CRTP-静态多态），提高性能。

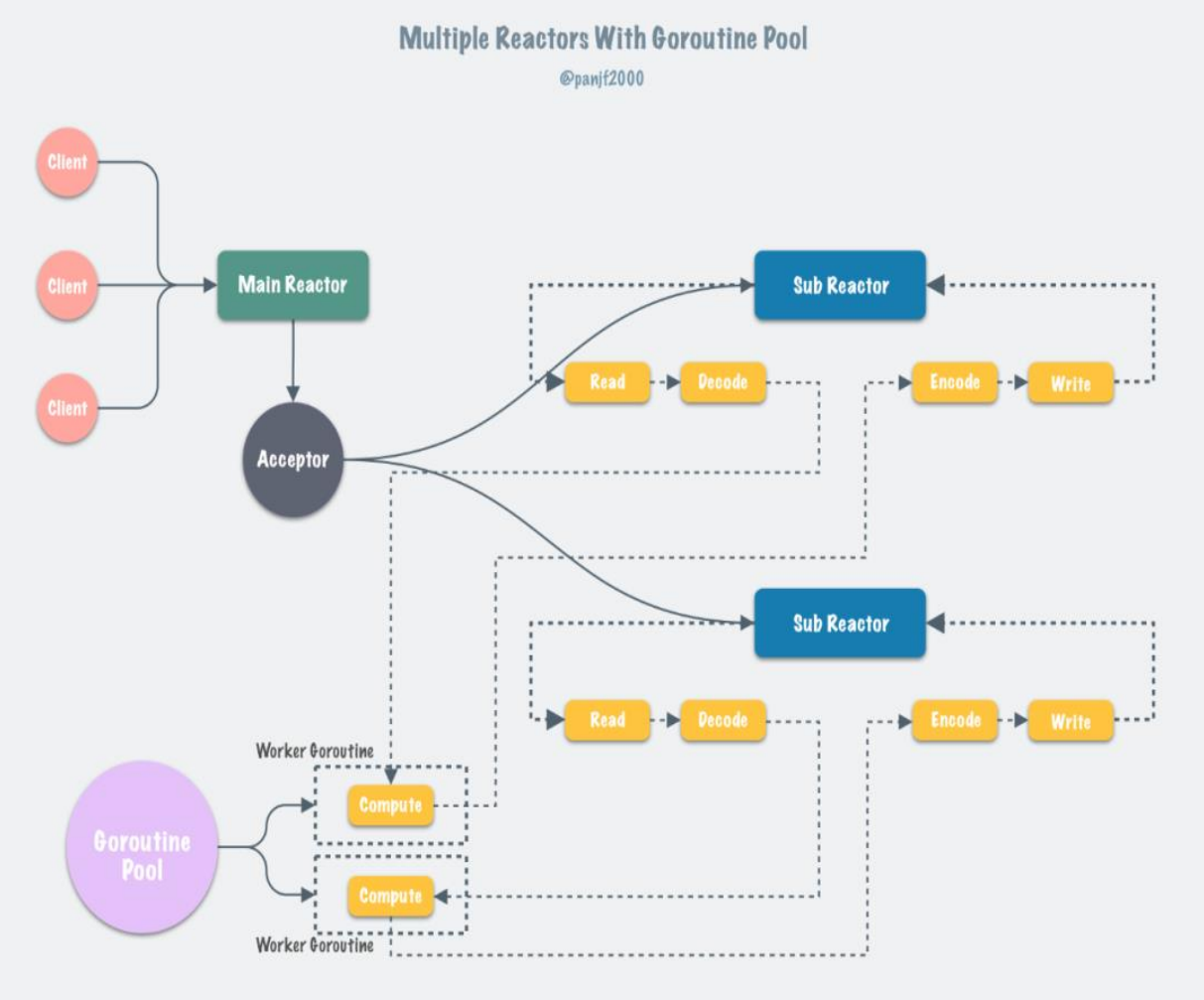
开源网络框架学习-netty





- **高效的Reactor线程模型**：主从Reactor多线程模型
- **高效的程序处理能力**：
  - 控制轮训时间，减少无效轮训CPU时间。
  - 使用责任链机制实现无锁化的串行设计ChannelPipeline
  - 使用并发库，无锁编程
- **高性能的序列化框架**：灵活选择序列化框架，比如Protobuff，Thrift的压缩二进制编解码框架
- **池化技术**： Executor线程池，ByteBuf内存池
- **高性能的ByteBuf**:
  - 支持堆外内存读写
  - 零拷贝
  - 内存分配算法/动态扩容
  - 引用计数器与资源管理复用
- **异步非阻塞通信**：Channel（epoll + 非阻塞）

开源网络框架学习-gnet

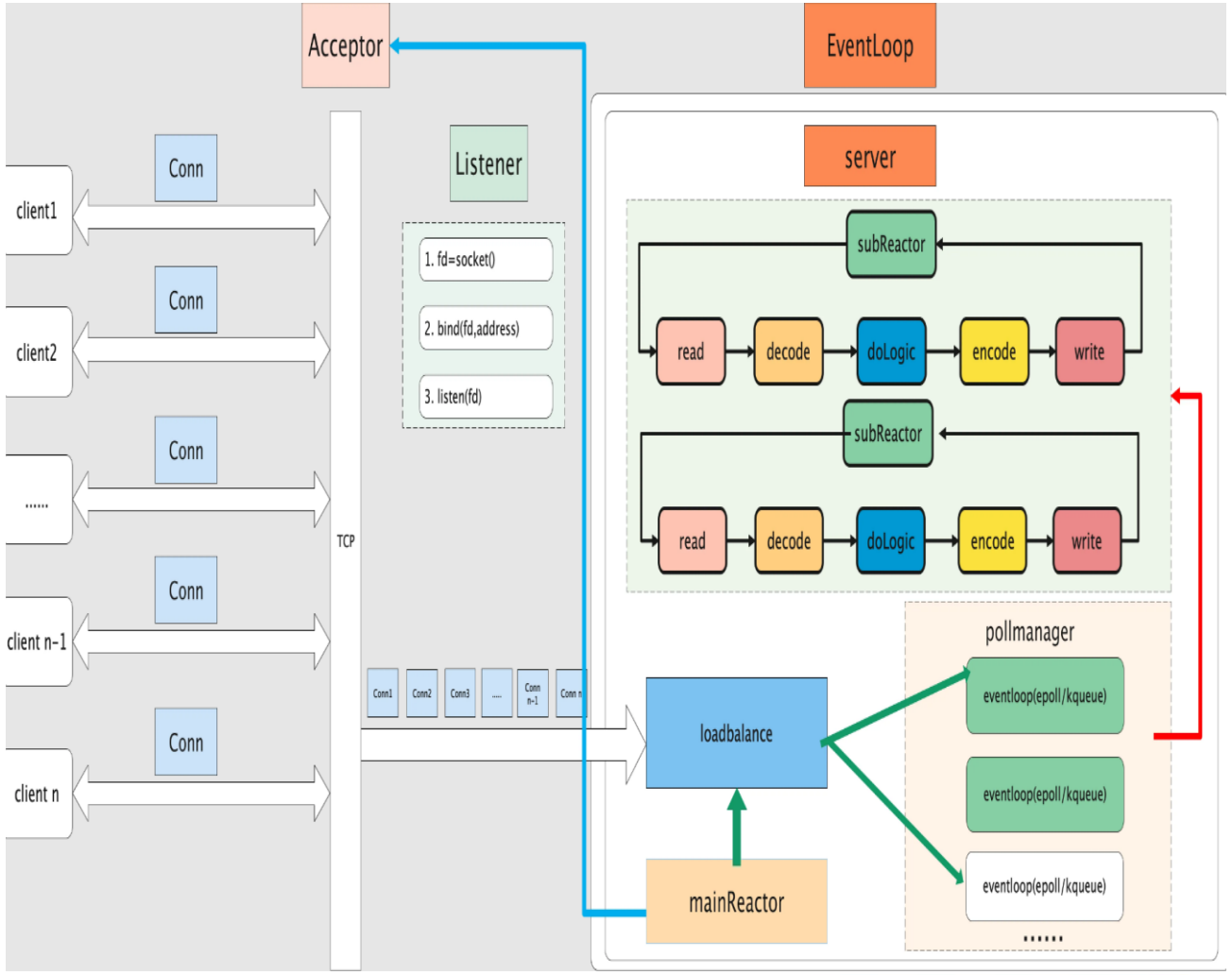


主从 Reactors + Goroutine Pool 模型

gnet 是一个基于事件驱动的高性能和轻量级网络框架。它直接使用 epoll 和 kqueue 系统调用而非标准 Go 网络包：net 来构建网络应用，它的工作原理类似两个开源的网络库：netty 和 libuv，这也使得 gnet 达到了一个远超 Go net 的性能表现。

- **高性能IO模型:**
  主从 Reactors + Goroutine Pool 模型
- **无锁设计:**
  无锁环形Ring-Buffer
- **协程/协程池:**
  由开源库 ants 提供支持协程池
- **内存Buffer池:**
  开源库 pool 提供内存池
- **高性能网络IO:**
  支持epoll/kqueue/ IOCP/io_uring
  SO_REUSEPORT

开源网络框架学习-netpoll

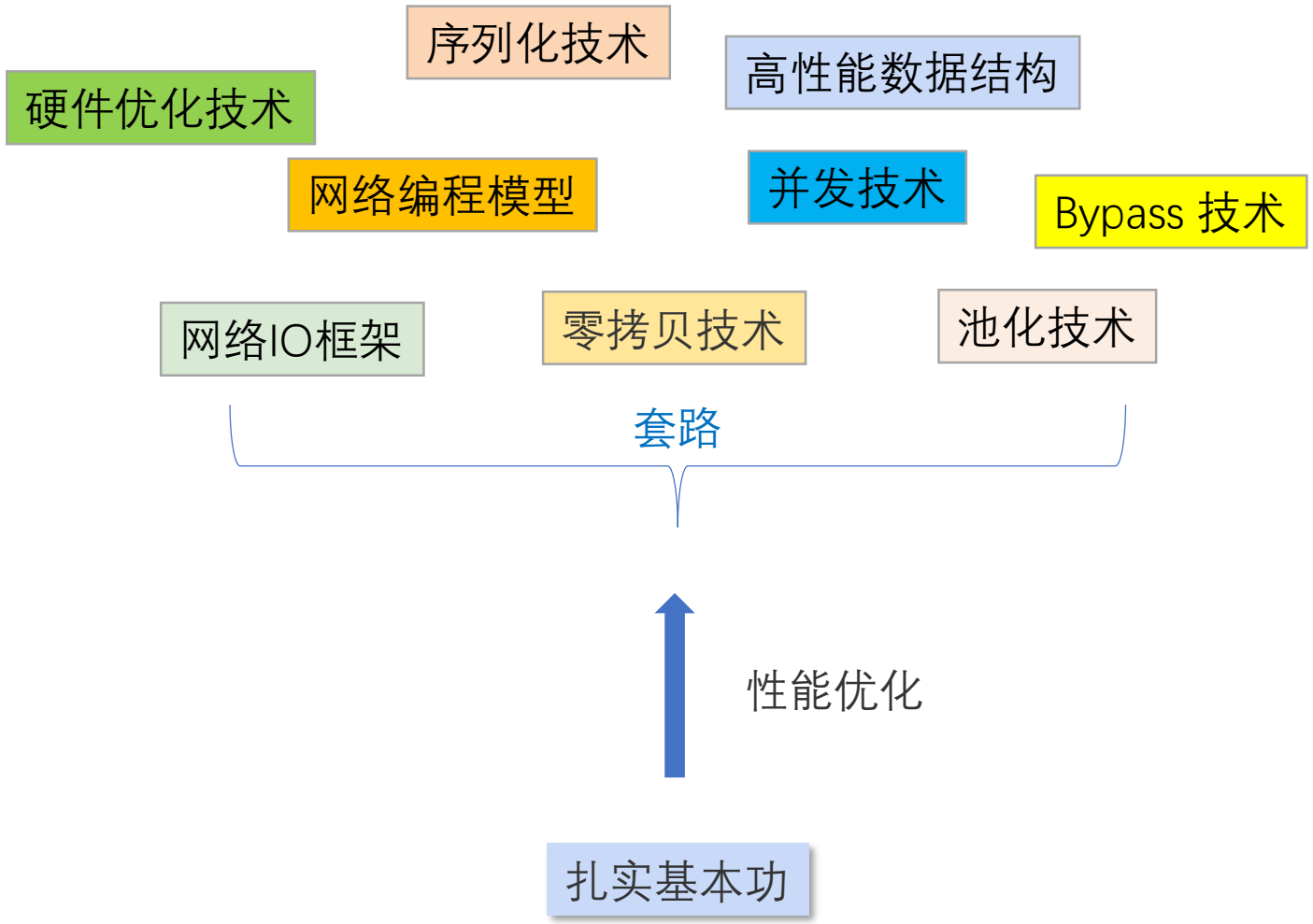


Multi-Reactor模型

netpoll 是由字节跳动开发的高性能 NIO(Non-blocking I/O) 网络库，专注于 RPC 场景另一方面，开源社区目前缺少专注于 RPC 方案的 Go 网络库。类似的项目如：evio , gnet 等，均面向 Redis, HAProxy 这样的场景。因此 Netpoll 应运而生，它借鉴了 evio 和 netty 的优秀设计，具有出色的 性能，更适用于微服务架构。 同时，Netpoll 还提供了一些特性，推荐在RPC设计中替代net 。基于 Netpoll 开发的 RPC 框架 Kitex 和 HTTP框架Hertz，性能均业界领先。

- 经典的Multi-Reactor模型
- 无锁设计:串行调度 I/O，适用于纯计算
- 协程池：gopool
- 内存Buffer池：mcache
- 优化系统调用： RawSyscall，批量系统调用
- 网络IO优化：支持epoll/io_uring，采用LT的编程思路。
- 零拷贝技术：
  - Shared Memory IPC
  - Nocopy Buffer：无锁访问，零拷贝和高效扩缩容，复用减少GC
  - ZeroCopy API(MSG_ZEROCOPY)
- 连接多路复用： Nocopy Buffer，分片锁， thrift header protocol 协议。

项目训练-如何实现一个高网络IO框架



- 参考业界开源设计
- 支持100w并发连接
- 单机百万~千万的QPS
- 排名网站性能跑分

项目训练-如何实现一个高网络IO框架

核心技术

- 高性能数据结构
  - 跳跃表
  - 哈希表
  - Nocopy Buffer
  - 无锁环形Ring-Buffer
  - Batch Buffer
- 序列化技术
  - Protostuff
  - Thrift
  - rapidjson
  - yyjson/simdjson
  - sonic-cpp
- 网络编程模型
  - 单Reactor模型
  - 多Reactor模型
  - Proactor模型
  - 异步IO+协程模型
- 网络IO框架
  - select/poll
  - epoll/kqueue
  - libaio
  - IOCP/io_uring

项目训练-如何实现一个高网络IO框架

核心技术

- 池化技术
  - 线程/协程池
  - 连接池
  - 报文/Buff池
  - 对象/内存池
- 零拷贝技术
  - 共享内存mmap
  - Direct I/O
  - Sendfile
  - ZeroCopy api
- 并发技术
  - Per thread/CPU设计
  - 锁/RCU/无锁优化
  - 绑核独占
  - 调度优化(优先级，nice值，调度算法调整)
- Bypass 技术
  - 消除runtime：直接调用底层接口
  - 消除多余抽象封装：cgo，嵌入汇编
  - Bypass内核：DPDK/RDMA
- 硬件优化技术
  - SIMD指令
  - GPU优化
  - 硬件offload

总结

- 掌握Linux 网络IO演进路线，问题和解决方案
- 掌握socket接口特殊用法和异常处理，深入理解API接口各个方面。
- 理解Linux 网络IO框架原理：select/poll/epoll/io_uring
- 理解网络IO常见的编程模型reactor和proactor等延伸的扩展模型
- 掌握常见网络IO性能优化的套路
- 熟悉常见开源网络IO框架的设计和实现，了解其目标场景，针对性的性能优化方法
- 学会根据业务实际情况进行网络IO编程的设计和实现