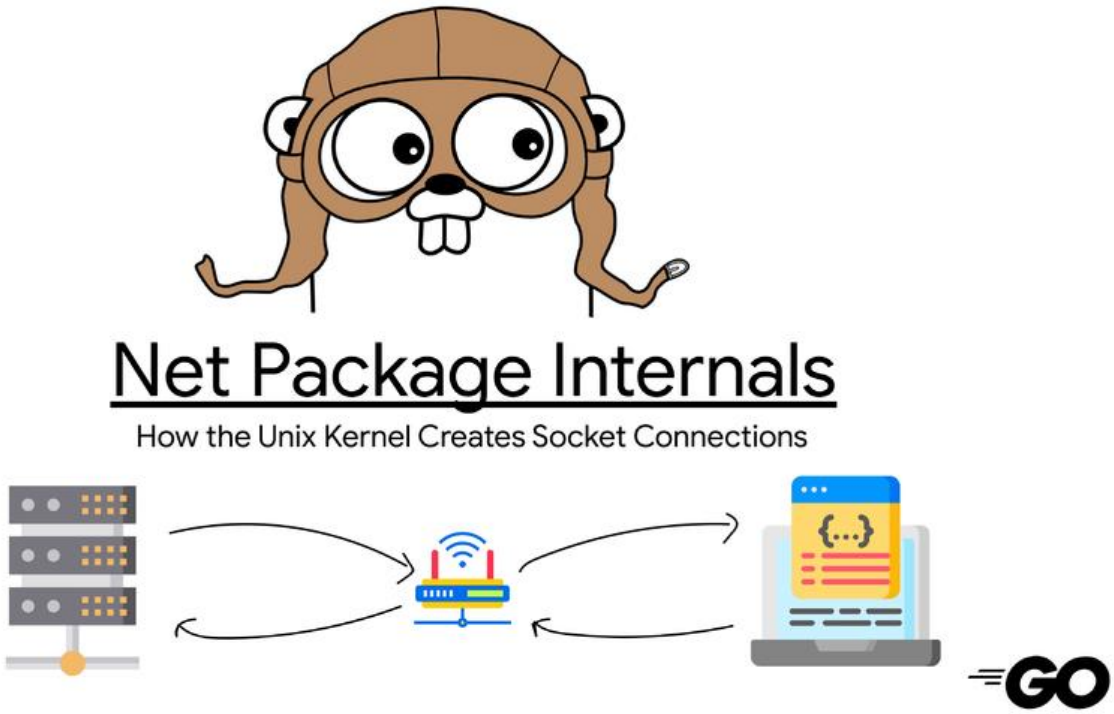
链滴

Go netpoll I/O 多路复用构建原生网络模型之源码深度解析

作者：panjf2000

原文链接：https://ld246.com/article/1573321729430

来源网站：链滴



导言

Go 基于 I/O multiplexing 和 goroutine 构建了一个简洁而高性能的原生网络模型(基于 Go 的 I/O 路复用 netpoll)，提供了 goroutine-per-connection 这样简单的网络编程模式。在这种模式下，发者使用的是同步的模式去编写异步的逻辑，极大地降低了开发者编写网络应用时的心智负担，且借于 Go runtime scheduler 对 goroutines 的高效调度，这个原生网络模型不论从适用性还是性能上足以满足绝大部分的应用场景。

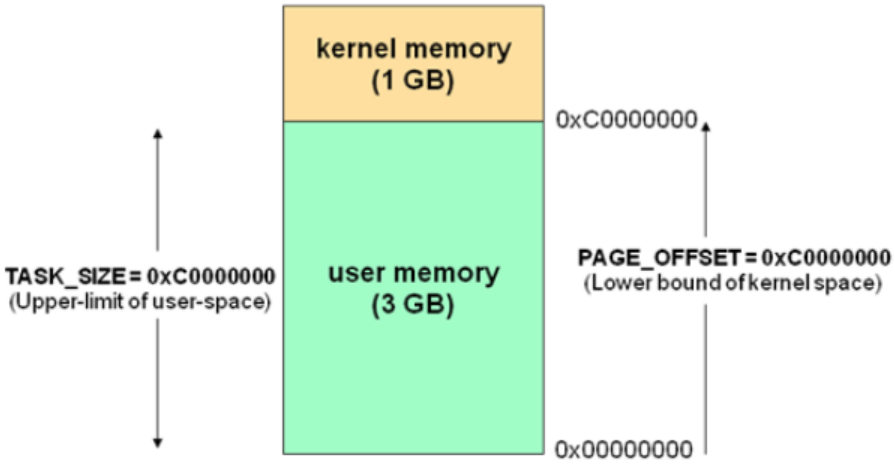
然而，在工程性上能做到如此高的普适性和兼容性，最终暴露给开发者提供接口/模式如此简洁，其层必然是基于非常复杂的封装，做了很多取舍，也有可能放弃了一些『极致』的设计和理念。事实上etpoll 底层就是基于 epoll/kqueue/iocp 这些系统调用来做封装的，最终暴露出 goroutine-per-connection 这样的极简的开发模式给使用者。

Go netpoll 在不同的操作系统，其底层使用的 I/O 多路复用技术也不一样，可以从 Go 源码目录结和对应代码文件了解 Go 在不同平台下的网络 I/O 模式的实现。比如，在 Linux 系统下基于 epoll，freBSD 系统下基于 kqueue，以及 Windows 系统下基于 iocp。

本文将基于 Linux 平台来解析 Go netpoll 之 I/O 多路复用的底层是如何基于 epoll 封装实现的，从码层层推进，全面而深度地解析 Go netpoll 的设计理念和实现原理，以及 Go 是如何利用 netpoll构建它的原生网络模型的。主要涉及到的一些概念：I/O 模式、用户/内核空间、epoll、Linux 源码、oroutine scheduler 等等，我会尽量简单地讲解，如果有对相关概念不熟悉的同学，还是希望能提前悉一下。

用户空间与内核空间

现在操作系统都是采用虚拟存储器，那么对 32 位操作系统而言，它的寻址空间（虚拟存储空间）为G（2 的 32 次方）。操作系统的核心是内核，独立于普通的应用程序，可以访问受保护的内存空间也有访问底层硬件设备的所有权限。为了保证用户进程不能直接操作内核（kernel），保证内核的安，操心系统将虚拟空间划分为两部分，一部分为内核空间，一部分为用户空间。针对 Linux 操作系统言，将最高的 1G 字节（从虚拟地址 0xC0000000 到 0xFFFFFFFF），供内核使用，称为内核空间，将较低的 3G 字节（从虚拟地址 0x00000000 到 0xBFFFFFFF），供各个进程使用，称为用户空间。



I/O 多路复用

在神作《UNIX 网络编程》里，总结归纳了 5 种 I/O 模型，包括同步和异步 I/O：

- 阻塞 I/O (Blocking I/O)
- 非阻塞 I/O (Nonblocking I/O)
- I/O 多路复用 (I/O multiplexing)
- 信号驱动 I/O (Signal driven I/O)
- 异步 I/O (Asynchronous I/O)

操作系统上的 I/O 是用户空间和内核空间的数据交互，因此 I/O 操作通常包含以下两个步骤：

1. 等待网络数据到达网卡(读就绪)/等待网卡可写(写就绪) –> 读取/写入到内核缓冲区
2. 从内核缓冲区复制数据 –> 用户空间(读)/从用户空间复制数据 -> 内核缓冲区(写)

而判定一个 I/O 模型是同步还是异步，主要看第二步：数据在用户和内核空间之间复制的时候是不是阻塞当前进程，如果会，则是同步 I/O，否则，就是异步 I/O。基于这个原则，这 5 种 I/O 模型中只一种异步 I/O 模型：Asynchronous I/O，其余都是同步 I/O 模型。

这 5 种 I/O 模型的对比如下：

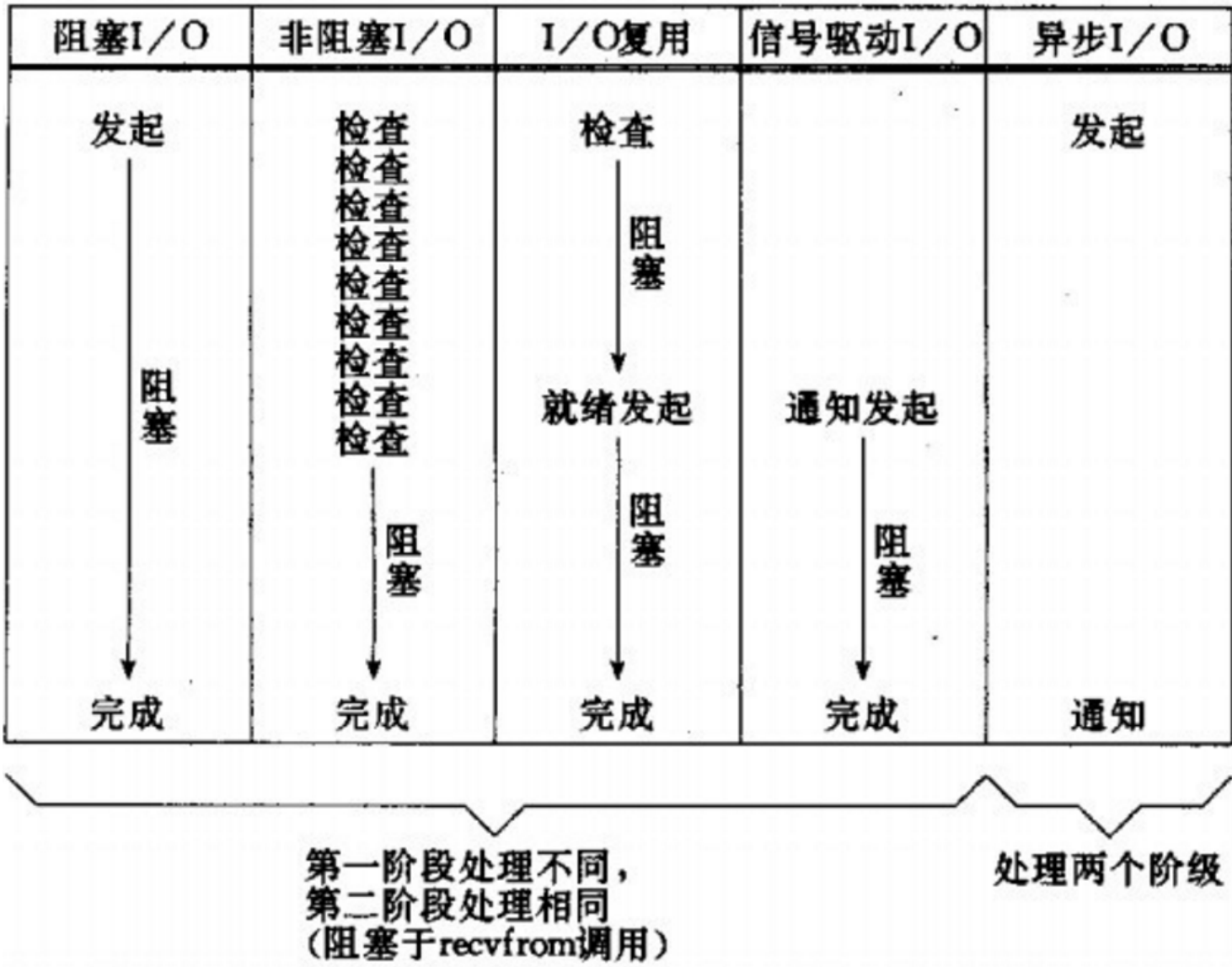


图 6.6　五个 I/O 模型的比较

所谓 I/O 多路复用指的就是 select/poll/epoll 这一系列的多路选择器：支持单一线程同时监听多个件描述符（I/O 事件），阻塞等待，并在其中某个文件描述符可读写时收到通知。 I/O 复用其实复用不是 I/O 连接，而是复用线程，让一个 thread of control 能够处理多个连接（I/O 事件）。

select & poll

```
#include <sys/select.h>

/* According to earlier standards */
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);

// 和 select 紧密结合的四个宏：
void FD_CLR(int fd, fd_set *set);
int FD_ISSET(int fd, fd_set *set);
```

原文链接：Go netpoll I/O 多路复用构建原生网络模型之源码深度解析

```
void FD_SET(int fd, fd_set *set);
void FD_ZERO(fd_set *set);
```

select 是 epoll 之前 Linux 使用的 I/O 事件驱动技术。

理解 select 的关键在于理解 fd_set，为说明方便，取 fd_set 长度为 1 字节，fd_set 中的每一 bit 可对应一个文件描述符 fd，则 1 字节长的 fd_set 最大可以对应 8 个 fd。select 的调用过程如下：

1. 执行 FD_ZERO(&set), 则 set 用位表示是　0000,0000

2. 若 fd = 5, 执行 FD_SET(fd, &set); 后 set 变为 0001,0000(第 5 位置为 1)

3. 再加入 fd = 2, fd=1，则 set 变为　0001,0011

4. 执行 select(6, &set, 0, 0, 0) 阻塞等待

5. 若 fd=1, fd=2 上都发生可读事件，则 select 返回，此时 set 变为　0000,0011 (注意：没有事件生的 fd=5 被清空)

基于上面的调用过程，可以得出 select 的特点：

● 可监控的文件描述符个数取决于 sizeof(fd_set) 的值。假设服务器上 sizeof(fd_set) = 512，每 bit 示一个文件描述符，则服务器上支持的最大文件描述符是 512*8=4096。fd_set 的大小调整可参考 原创】技术系列之 网络模型（二） 中的模型 2，可以有效突破 select 可监控的文件描述符上限

● 将 fd 加入 select 监控集的同时，还要再使用一个数据结构 array 保存放到 select 监控集中的 fd 一是用于在 select 返回后，array 作为源数据和 fd_set 进行 FD_ISSET 判断。二是 select 返回后会以前加入的但并无事件发生的 fd 清空，则每次开始 select 前都要重新从 array 取得 fd 逐一加入（FD ZERO 最先），扫描 array 的同时取得 fd 最大值 maxfd，用于 select 的第一个参数

● 可见 select 模型必须在 select 前循环 array（加 fd，取 maxfd），select 返回后循环 array（FD_I SET 判断是否有事件发生）

所以，select 有如下的缺点：

1. 最大并发数限制：使用 32 个整数的 32 位，即 32*32=1024 来标识 fd，虽然可修改，但是有以第 2, 3 点的瓶颈

2. 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

3. 性能衰减严重：每次 kernel 都需要线性扫描整个 fd_set，所以随着监控的描述符 fd 数量增长，其 /O 性能会线性下降

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，poll 解决了最大文件描述符数量限制的问题，但是同样需要从用户态拷贝所有的 fd 到核态，也需要线性遍历所有的 fd 集合，所以它和 select 只是实现细节上的区分，并没有本质上的区 。

epoll

epoll 是 Linux kernel 2.6 之后引入的新 I/O 事件驱动技术，I/O 多路复用的核心设计是 1 个线程处所有连接的 等待消息准备好 I/O 事件，这一点上 epoll 和 select&poll 是大同小异的。但 select&pol 错误预估了一件事，当数十万并发连接存在时，可能每一毫秒只有数百个活跃的连接，同时其余数十连接在这一毫秒是非活跃的。select&poll 的使用方法是这样的： 返回的活跃连接 == select(全部待控的连接) 。

什么时候会调用 select&poll 呢？在你认为需要找出有报文到达的活跃连接时，就应该调用。所以，se ect&poll 在高并发时是会被频繁调用的。这样，这个频繁调用的方法就很有必要看看它是否有效率

因为，它的轻微效率损失都会被 高频 二字所放大。它有效率损失吗？显而易见，全部待监控连接是以十万计的，返回的只是数百个活跃连接，这本身就是无效率的表现。被放大后就会发现，处理并发万个连接时，select&poll 就完全力不从心了。这个时候就该 epoll 上场了，epoll 通过一些新的设计优化，基本上解决了 select&poll 的问题。

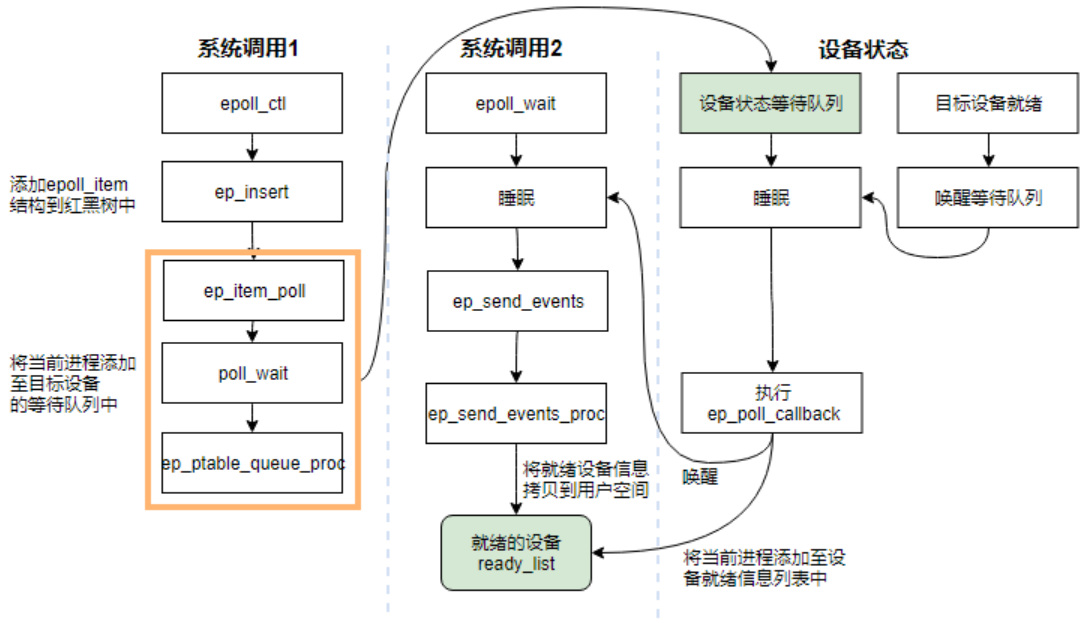
epoll 的 API 非常简洁，涉及到的只有 3 个系统调用：

```
#include <sys/epoll.h>
int epoll_create(int size); // int epoll_create1(int flags);
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
```

其中，epoll_create 创建一个 epoll 实例并返回 epollfd；epoll_ctl 注册 file descriptor 等待的 I/O 件(比如 EPOLLIN、EPOLLOUT 等) 到 epoll 实例上；epoll_wait 则是阻塞监听 epoll 实例上所有的 fi e descriptor 的 I/O 事件，它接收一个用户空间上的一块内存地址 (events 数组)，kernel 会在有 I/O 事件发生的时候把文件描述符列表复制到这块内存地址上，然后 epoll_wait 解除阻塞并返回，最后户空间上的程序就可以对相应的 fd 进行读写了：

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

epoll 的工作原理如下：



与 select&poll 相比，epoll 分清了高频调用和低频调用。例如，epoll_ctl 相对来说就是非频繁调用，而 epoll_wait 则是会被高频调用的。所以 epoll 利用 epoll_ctl 来插入或者删除一个 fd，实现用户到内核态的数据拷贝，这确保了每一个 fd 在其生命周期只需要被拷贝一次，而不是每次调用 epoll_wa t 的时候都拷贝一次。 epoll_wait 则被设计成几乎没有入参的调用，相比 select&poll 需要把全部监的 fd 集合从用户态拷贝至内核态的做法，epoll 的效率就高出了一大截。

在实现上 epoll 采用红黑树来存储所有监听的 fd，而红黑树本身插入和删除性能比较稳定，时间复杂 O(logN)。通过 epoll_ctl 函数添加进来的 fd 都会被放在红黑树的某个节点内，所以，重复添加是没用的。当把 fd 添加进来的时候时候会完成关键的一步：该 fd 会与相应的设备（网卡）驱动程序建立调关系，也就是在内核中断处理程序为它注册一个回调函数，在 fd 相应的事件触发（中断）之后（备就绪了），内核就会调用这个回调函数，该回调函数在内核中被称为： ep_poll_callback ，**这个回函数其实就是把这个 fd 添加到 rdllist 这个双向链表（就绪链表）中**。epoll_wait 实际上就是去检查

dlist 双向链表中是否有就绪的 fd，当 rdlist 为空（无就绪 fd）时挂起当前进程，直到 rdlist 非空时程才被唤醒并返回。

相比于 select&poll 调用时会将全部监听的 fd 从用户态空间拷贝至内核态空间并线性扫描一遍找出绪的 fd 再返回到用户态，epoll_wait 则是直接返回已就绪 fd，因此 epoll 的 I/O 性能不会像 select poll 那样随着监听的 fd 数量增加而出现线性衰减，是一个非常高效的 I/O 事件驱动技术。

由于使用 epoll 的 I/O 多路复用需要用户进程自己负责 I/O 读写，从用户进程的角度看，读写过程阻塞的，所以 select&poll&epoll 本质上都是同步 I/O 模型，而像 Windows 的 IOCP 这一类的异 I/O，只需要在调用 WSARecv 或 WSASend 方法读写数据的时候把用户空间的内存 buffer 提交给 kernel，kernel 负责数据在用户空间和内核空间拷贝，完成之后就会通知用户进程，整个过程不需用户进程参与，所以是真正的异步 I/O。

延伸

另外，我看到有些文章说 epoll 之所以性能高是因为利用了 Linux 的 mmap 内存映射让内核和用户程共享了一片物理内存，用来存放就绪 fd 列表和它们的数据 buffer，所以用户进程在 epoll_wait 返之后用户进程就可以直接从共享内存那里读取/写入数据了，这让我很疑惑，因为首先看 epoll_wait 函数声明：

```
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
```

第二个参数：就绪事件列表，是需要在用户空间分配内存然后再传给 epoll_wait 的，如果内核会用 map 设置共享内存，直接传递一个指针进去就行了，根本不需要在用户态分配内存，多此一举。其，内核和用户进程通过 mmap 共享内存是一件极度危险的事情，内核无法确定这块共享内存什么时会被回收，而且这样也会赋予用户进程直接操作内核数据的权限和入口，非常容易出现大的系统漏洞因此一般极少会这么做。所以我很怀疑 epoll 是不是真的在 Linux kernel 里用了 mmap，我就去看下最新版本（5.3.9）的 Linux kernel 源码：

```
/*
 * Implement the event wait interface for the eventpoll file. It is the kernel
 * part of the user space epoll_wait(2).
 */
static int do_epoll_wait(int epfd, struct epoll_event __user *events,
             int maxevents, int timeout)
{
    // ...

    /* Time to fish for events ... */
    error = ep_poll(ep, events, maxevents, timeout);
}

// 如果 epoll_wait 入参时设定 timeout == 0, 那么直接通过 ep_events_available 判断当前是否有户感兴趣的事件发生，如果有则通过 ep_send_events 进行处理
// 如果设置 timeout > 0，并且当前没有用户关注的事件发生，则进行休眠，并添加到 ep->wq 等队列的头部；对等待事件描述符设置 WQ_FLAG_EXCLUSIVE 标志
// ep_poll 被事件唤醒后会重新检查是否有关注事件，如果对应的事件已经被抢走，那么 ep_poll 会续休眠等待
static int ep_poll(struct eventpoll *ep, struct epoll_event __user *events, int maxevents, long t meout)
{
    // ...

    send_events:
```

```
    /*
     * Try to transfer events to user space. In case we get 0 events and
     * there's still timeout left over, we go trying again in search of
     * more luck.
     */

    // 如果一切正常, 有 event 发生, 就开始准备数据 copy 给用户空间了
    // 如果有就绪的事件发生，那么就调用 ep_send_events 将就绪的事件 copy 到用户态内存中，
    // 然后返回到用户态，否则判断是否超时，如果没有超时就继续等待就绪事件发生，如果超时就返
用户态。
    // 从 ep_poll 函数的实现可以看到，如果有就绪事件发生，则调用 ep_send_events 函数做进一
处理
    if (!res && eavail &&
            !(res = ep_send_events(ep, events, maxevents)) && !timed_out)
        goto fetch_events;

    // ...
}

// ep_send_events 函数是用来向用户空间拷贝就绪 fd 列表的，它将用户传入的就绪 fd 列表内存简
封装到
// ep_send_events_data 结构中，然后调用 ep_scan_ready_list 将就绪队列中的事件写入用户空间
内存；
// 用户进程就可以访问到这些数据进行处理
static int ep_send_events(struct eventpoll *ep,
                struct epoll_event __user *events, int maxevents)
{
    struct ep_send_events_data esed;

    esed.maxevents = maxevents;
    esed.events = events;
    // 调用 ep_scan_ready_list 函数检查 epoll 实例 eventpoll 中的 rdllist 就绪链表，
    // 并注册一个回调函数 ep_send_events_proc，如果有就绪 fd，则调用 ep_send_events_proc
行处理
    ep_scan_ready_list(ep, ep_send_events_proc, &esed, 0, false);
    return esed.res;
}

// 调用 ep_scan_ready_list 的时候会传递指向 ep_send_events_proc 函数的函数指针作为回调函数

// 一旦有就绪 fd，就会调用 ep_send_events_proc 函数
static __poll_t ep_send_events_proc(struct eventpoll *ep, struct list_head *head, void *priv)
{
    // ...

    /*
     * If the event mask intersect the caller-requested one,
     * deliver the event to userspace. Again, ep_scan_ready_list()
     * is holding ep->mtx, so no operations coming from userspace
     * can change the item.
     */
    revents = ep_item_poll(epi, &pt, 1);
    // 如果 revents 为 0，说明没有就绪的事件，跳过，否则就将就绪事件拷贝到用户态内存中
    if (!revents)
```

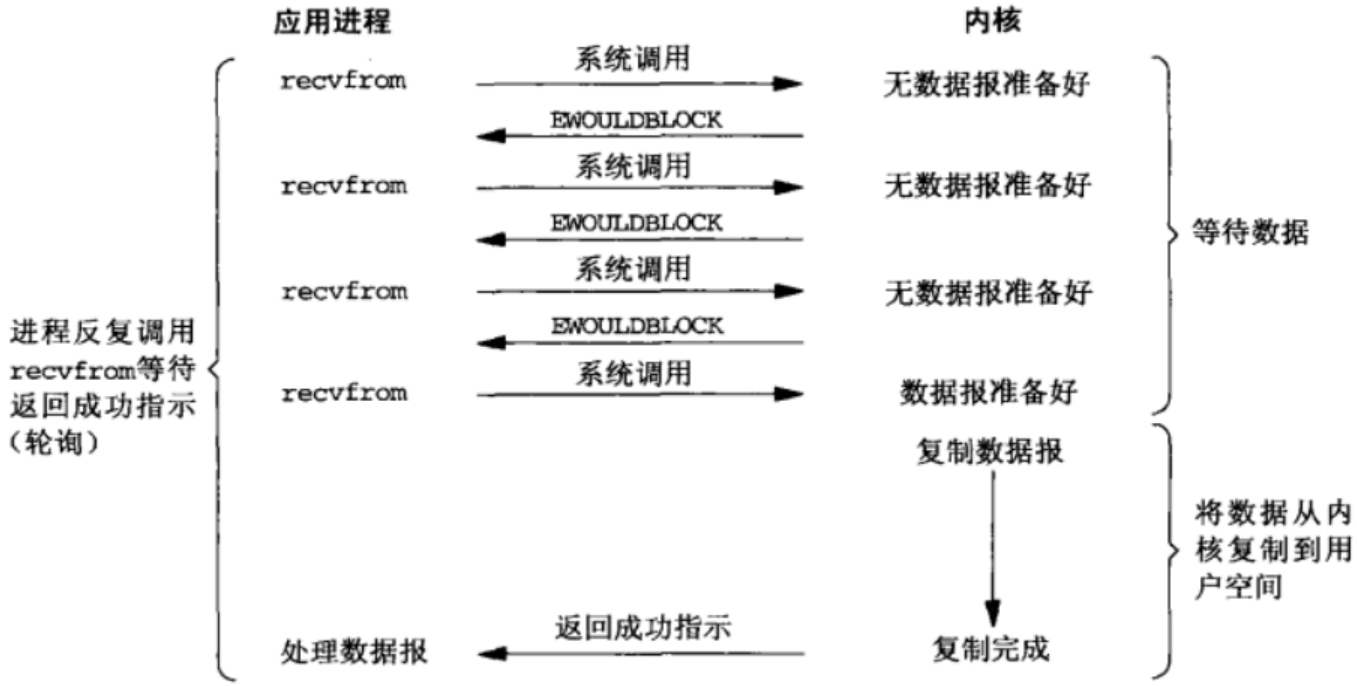
```
            continue;
        // 将当前就绪的事件和用户进程传入的数据都通过 __put_user 拷贝回用户空间,
        // 也就是调用 epoll_wait 之时用户进程传入的 fd 列表的内存
        if (__put_user(revents, &uevent->events) || __put_user(epi->event.data, &uevent->data)) {
            list_add(&epi->rdllink, head);
            ep_pm_stay_awake(epi);
            if (!esed->res)
                esed->res = -EFAULT;
            return 0;
        }

        // ...
}
```

从 do_epoll_wait 开始层层跳转，我们可以很清楚地看到最后内核是通过 __put_user 函数把就绪 fd 表和事件返回到用户空间，而 __put_user 正是内核用来拷贝数据到用户空间的标准函数。此外，我没有在 Linux kernel 的源码中和 epoll 相关的代码里找到 mmap 系统调用做内存映射的逻辑，所以本可以得出结论：epoll 在 Linux kernel 里并没有使用 mmap 来做用户空间和内核空间的内存共享所以那些说 epoll 使用了 mmap 的文章都是误解。

Non-blocking I/O

什么叫非阻塞 I/O，顾名思义就是：所有 I/O 操作都是立刻返回而不会阻塞当前用户进程。I/O 多路用通常情况下需要和非阻塞 I/O 搭配使用，否则可能会产生意想不到的问题。比如，epoll 的 ET(边触发) 模式下，如果不使用非阻塞 I/O，有极大的概率会导致阻塞 event-loop 线程，从而降低吞吐量甚至导致 bug。

Linux 下，我们可以通过 fcntl 系统调用来设置 O_NONBLOCK 标志位，从而把 socket 设置成 non-blocking。当对一个 non-blocking socket 执行读操作时，流程是这个样子：



当用户进程发出 read 操作时，如果 kernel 中的数据还没有准备好，那么它并不会 block 用户进程而是立刻返回一个 EAGAIN error。从用户进程角度讲 ，它发起一个 read 操作后，并不需要等待，是马上就得到了一个结果。用户进程判断结果是一个 error 时，它就知道数据还没有准备好，于是它

以再次发送 read 操作。一旦 kernel 中的数据准备好了，并且又再次收到了用户进程的 system call 那么它马上就将数据拷贝到了用户内存，然后返回。

所以，non-blocking I/O 的特点是用户进程需要不断的主动询问 kernel 数据好了没有。

Go netpoll

一个典型的 Go TCP server:

```
package main

import (
    "fmt"
    "net"
)

func main() {
    listen, err := net.Listen("tcp", ":8888")
    if err != nil {
        fmt.Println("listen error: ", err)
        return
    }

    for {
        conn, err := listen.Accept()
        if err != nil {
            fmt.Println("accept error: ", err)
            break
        }

        // start a new goroutine to handle the new connection
        go HandleConn(conn)
    }
}
func HandleConn(conn net.Conn) {
    defer conn.Close()
    packet := make([]byte, 1024)
    for {
        // 如果没有可读数据，也就是读 buffer 为空，则阻塞
        _, _ = conn.Read(packet)
        // 同理，不可写则阻塞
        _, _ = conn.Write(packet)
    }
}
```

上面是一个基于 Go 原生网络模型（基于 netpoll）编写的一个 TCP server，模式是 goroutine-per-onnection ，在这种模式下，开发者使用的是同步的模式去编写异步的逻辑而且对于开发者来说 I/O 否阻塞是无感知的，也就是说开发者无需考虑 goroutines 甚至更底层的线程、进程的调度和上下文换。而 Go netpoll 最底层的事件驱动技术肯定是基于 epoll/kqueue/iocp 这一类的 I/O 事件驱动技，只不过是把这些调度和上下文切换的工作转移到了 runtime 的 Go scheduler，让它来负责调度 go outines，从而极大地降低了程序员的心智负担！

Go netpoll 核心

Go netpoll 通过在底层对 epoll/kqueue/iocp 的封装，从而实现了使用同步编程模式达到异步执行效果。总结来说，所有的网络操作都以网络描述符 netFD 为中心实现。netFD 与底层 PollDesc 结构定，当在一个 netFD 上读写遇到 EAGAIN 错误时，就将当前 goroutine 存储到这个 netFD 对应的 PllDesc 中，同时调用 gopark 把当前 goroutine 给 park 住，直到这个 netFD 上再次发生读写事件才将此 goroutine 给 ready 激活重新运行。显然，在底层通知 goroutine 再次发生读写等事件的方就是 epoll/kqueue/iocp 等事件驱动机制。

接下来我们通过分析最新的 Go 源码（v1.13.4），解读一下整个 netpoll 的运行流程。

上面的示例代码中相关的在源码里的几个数据结构和方法：

```
// TCPListener is a TCP network listener. Clients should typically
// use variables of type Listener instead of assuming TCP.
type TCPListener struct {
    fd *netFD
    lc ListenConfig
}

// Accept implements the Accept method in the Listener interface; it
// waits for the next call and returns a generic Conn.
func (l *TCPListener) Accept() (Conn, error) {
    if !l.ok() {
        return nil, syscall.EINVAL
    }
    c, err := l.accept()
    if err != nil {
        return nil, &OpError{Op: "accept", Net: l.fd.net, Source: nil, Addr: l.fd.laddr, Err: err}
    }
    return c, nil
}

func (ln *TCPListener) accept() (*TCPConn, error) {
    fd, err := ln.fd.accept()
    if err != nil {
        return nil, err
    }
    tc := newTCPConn(fd)
    if ln.lc.KeepAlive >= 0 {
        setKeepAlive(fd, true)
        ka := ln.lc.KeepAlive
        if ln.lc.KeepAlive == 0 {
            ka = defaultTCPKeepAlive
        }
        setKeepAlivePeriod(fd, ka)
    }
    return tc, nil
}

// TCPConn is an implementation of the Conn interface for TCP network
// connections.
type TCPConn struct {
    conn
}
```

```
// Conn
type conn struct {
    fd *netFD
}

type conn struct {
    fd *netFD
}

func (c *conn) ok() bool { return c != nil && c.fd != nil }

// Implementation of the Conn interface.

// Read implements the Conn Read method.
func (c *conn) Read(b []byte) (int, error) {
    if !c.ok() {
        return 0, syscall.EINVAL
    }
    n, err := c.fd.Read(b)
    if err != nil && err != io.EOF {
        err = &OpError{Op: "read", Net: c.fd.net, Source: c.fd.laddr, Addr: c.fd.raddr, Err: err}
    }
    return n, err
}

// Write implements the Conn Write method.
func (c *conn) Write(b []byte) (int, error) {
    if !c.ok() {
        return 0, syscall.EINVAL
    }
    n, err := c.fd.Write(b)
    if err != nil {
        err = &OpError{Op: "write", Net: c.fd.net, Source: c.fd.laddr, Addr: c.fd.raddr, Err: err}
    }
    return n, err
}
```

netFD

net.Listen("tcp", ":8888") 方法返回了一个 *TCPListener，它是一个实现了 net.Listener 接口的 struct，而通过 listener.Accept() 接收的新连接 *TCPConn 则是一个实现了 net.Conn 接口的 struct，它嵌了 net.conn struct。仔细阅读上面的源码可以发现，不管是 Listener 的 Accept 还是 Conn 的 Read/Write 方法，都是基于一个 netFD 的数据结构的操作， netFD 是一个网络描述符，类似于 Linux 文件描述符的概念，netFD 中包含一个 poll.FD 数据结构，而 poll.FD 中包含两个重要的数据结构 Sysd 和 pollDesc，前者是真正的系统文件描述符，后者对是底层事件驱动的封装，所有的读写超时等操都是通过调用后者的对应方法实现的。

netFD 和 poll.FD 的源码：

```
// Network file descriptor.
type netFD struct {
    pfd poll.FD

    // immutable until Close
```

```
    family      int
    sotype      int
    isConnected bool // handshake completed or use of association with peer
    net         string
    laddr       Addr
    raddr       Addr
}

// FD is a file descriptor. The net and os packages use this type as a
// field of a larger type representing a network connection or OS file.
type FD struct {
    // Lock sysfd and serialize access to Read and Write methods.
    fdmu fdMutex

    // System file descriptor. Immutable until Close.
    Sysfd int

    // I/O poller.
    pd pollDesc

    // Writev cache.
    iovecs *[]syscall.Iovec

    // Semaphore signaled when file is closed.
    csema uint32

    // Non-zero if this file has been set to blocking mode.
    isBlocking uint32

    // Whether this is a streaming descriptor, as opposed to a
    // packet-based descriptor like a UDP socket. Immutable.
    IsStream bool

    // Whether a zero byte read indicates EOF. This is false for a
    // message based socket connection.
    ZeroReadIsEOF bool

    // Whether this is a file rather than a network socket.
    isFile bool
}
```

pollDesc

前面提到了 pollDesc 是底层事件驱动的封装，netFD 通过它来完成各种 I/O 相关的操作，它的定义下：

```
type pollDesc struct {
    runtimeCtx uintptr
}
```

这里的 struct 只包含了一个指针，而通过 pollDesc 的 init 方法，我们可以找到它具体的定义是在 runtime.pollDesc 这里：

```
func (pd *pollDesc) init(fd *FD) error {
```

```
    serverInit.Do(runtime_pollServerInit)
    ctx, errno := runtime_pollOpen(uintptr(fd.Sysfd))
    if errno != 0 {
        if ctx != 0 {
            runtime_pollUnblock(ctx)
            runtime_pollClose(ctx)
        }
        return syscall.Errno(errno)
    }
    pd.runtimeCtx = ctx
    return nil
}

// Network poller descriptor.
//
// No heap pointers.
//
//go:notinheap
type pollDesc struct {
    link *pollDesc // in pollcache, protected by pollcache.lock

    // The lock protects pollOpen, pollSetDeadline, pollUnblock and deadlineimpl operations.
    // This fully covers seq, rt and wt variables. fd is constant throughout the PollDesc lifetime.
    // pollReset, pollWait, pollWaitCanceled and runtime·netpollready (IO readiness notification

    // proceed w/o taking the lock. So closing, everr, rg, rd, wg and wd are manipulated
    // in a lock-free way by all operations.
    // NOTE(dvyukov): the following code uses uintptr to store *g (rg/wg),
    // that will blow up when GC starts moving objects.
    lock    mutex // protects the following fields
    fd      uintptr
    closing bool
    everr   bool    // marks event scanning error happened
    user    uint32  // user settable cookie
    rseq    uintptr // protects from stale read timers
    rg      uintptr // pdReady, pdWait, G waiting for read or nil
    rt      timer   // read deadline timer (set if rt.f != nil)
    rd      int64   // read deadline
    wseq    uintptr // protects from stale write timers
    wg      uintptr // pdReady, pdWait, G waiting for write or nil
    wt      timer   // write deadline timer
    wd      int64   // write deadline
}
```

runtime.pollDesc 包含自身类型的一个指针，用来保存下一个 runtime.pollDesc 的地址，以此来实链表，可以减少数据结构的大小，所有的 runtime.pollDesc 保存在 runtime.pollCache 结构中，定如下：

```
type pollCache struct {
  lock  mutex
  first *pollDesc
  // PollDesc objects must be type-stable,
  // because we can get ready notification from epoll/kqueue
  // after the descriptor is closed/reused.
```

```
    // Stale notifications are detected using seq variable,
    // seq is incremented when deadlines are changed or descriptor is reused.
}
```

net.Listen

调用 net.Listen 之后，底层会通过 Linux 的系统调用 socket 方法创建一个 fd 分配给 listener，并用来初始化 listener 的 netFD，接着调用 netFD 的 listenStream 方法完成对 socket 的 bind&listen 作以及对 netFD 的初始化（主要是对 netFD 里的 pollDesc 的初始化），相关源码如下：

```
// 调用 linux 系统调用 socket 创建 listener fd 并设置为为阻塞 I/O
s, err := socketFunc(family, sotype|syscall.SOCK_NONBLOCK|syscall.SOCK_CLOEXEC, proto)
// On Linux the SOCK_NONBLOCK and SOCK_CLOEXEC flags were
// introduced in 2.6.27 kernel and on FreeBSD both flags were
// introduced in 10 kernel. If we get an EINVAL error on Linux
// or EPROTONOSUPPORT error on FreeBSD, fall back to using
// socket without them.

socketFunc        func(int, int, int) (int, error)  = syscall.Socket

// 用上面创建的 listener fd 初始化 listener netFD
if fd, err = newFD(s, family, sotype, net); err != nil {
    poll.CloseFunc(s)
    return nil, err
}

// 对 listener fd 进行 bind&listen 操作，并且调用 init 方法完成初始化
func (fd *netFD) listenStream(laddr sockaddr, backlog int, ctrlFn func(string, string, syscall.Ra
Conn) error) error {
    // ...

    // 完成绑定操作
    if err = syscall.Bind(fd.pfd.Sysfd, lsa); err != nil {
        return os.NewSyscallError("bind", err)
    }

    // 完成监听操作
    if err = listenFunc(fd.pfd.Sysfd, backlog); err != nil {
        return os.NewSyscallError("listen", err)
    }

    // 调用 init，内部会调用 poll.FD.Init，最后调用 pollDesc.init
    if err = fd.init(); err != nil {
        return err
    }
    lsa, _ = syscall.Getsockname(fd.pfd.Sysfd)
    fd.setAddr(fd.addrFunc()(lsa), nil)
    return nil
}

// 使用 sync.Once 来确保一个 listener 只持有一个 epoll 实例
var serverInit sync.Once

// netFD.init 会调用 poll.FD.Init 并最终调用到 pollDesc.init，
```

```
// 它会创建 epoll 实例并把 listener fd 加入监听队列
func (pd *pollDesc) init(fd *FD) error {
    // runtime_pollServerInit 内部调用了 netpollinit 来创建 epoll 实例
    serverInit.Do(runtime_pollServerInit)

    // runtime_pollOpen 内部调用了 netpollopen 来将 listener fd 注册到
    // epoll 实例中，另外，它会初始化一个 pollDesc 并返回
    ctx, errno := runtime_pollOpen(uintptr(fd.Sysfd))
    if errno != 0 {
        if ctx != 0 {
            runtime_pollUnblock(ctx)
            runtime_pollClose(ctx)
        }
        return syscall.Errno(errno)
    }
    // 把真正初始化完成的 pollDesc 实例赋值给当前的 pollDesc 代表自身的指针，
    // 后续使用直接通过该指针操作
    pd.runtimeCtx = ctx
    return nil
}

var (
    // 全局唯一的 epoll fd，只在 listener fd 初始化之时被指定一次
    epfd int32 = -1 // epoll descriptor
)

// netpollinit 会创建一个 epoll 实例，然后把 epoll fd 赋值给 epfd，
// 后续 listener 以及它 accept 的所有 sockets 有关 epoll 的操作都是基于这个全局的 epfd
func netpollinit() {
    epfd = epollcreate1(_EPOLL_CLOEXEC)
    if epfd >= 0 {
        return
    }
    epfd = epollcreate(1024)
    if epfd >= 0 {
        closeonexec(epfd)
        return
    }
    println("runtime: epollcreate failed with", -epfd)
    throw("runtime: netpollinit failed")
}

// netpollopen 会被 runtime_pollOpen 调用，注册 fd 到 epoll 实例，
// 同时会利用万能指针把 pollDesc 保存到 epollevent 的一个 8 位的字节数组 data 里
func netpollopen(fd uintptr, pd *pollDesc) int32 {
    var ev epollevent
    ev.events = _EPOLLIN | _EPOLLOUT | _EPOLLRDHUP | _EPOLLET
    *(**pollDesc)(unsafe.Pointer(&ev.data)) = pd
    return -epollctl(epfd, _EPOLL_CTL_ADD, int32(fd), &ev)
}
```

我们前面提到的 epoll 的三个基本调用，Go 在源码里实现了对那三个调用的封装：

```
#include <sys/epoll.h>
```

```
int epoll_create(int size);
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
int epoll_wait(int epfd, struct epoll_event * events, int maxevents, int timeout);

// Go 对上面三个调用的封装
func netpollinit()
func netpollopen(fd uintptr, pd *pollDesc) int32
func netpoll(block bool) gList
```

netFD 就是通过这三个封装来对 epoll 进行创建实例、注册 fd 和等待事件操作的。

Listener.Accept()

netpoll accept socket 的工作流程如下：

1. 服务端的 netFD 在 listen 时会创建 epoll 的实例，并将 listenerFD 加入 epoll 的事件队列
2. netFD 在 accept 时将返回的 connFD 也加入 epoll 的事件队列
3. netFD 在读写时出现 syscall.EAGAIN 错误，通过 pollDesc 的 waitRead 方法将当前的 goroutine park 住，直到 ready，从 pollDesc 的 waitRead 中返回

Listener.Accept() 接收来自客户端的新连接，具体还是调用 netFD.accept 方法来完成这个功能：

```
// Accept implements the Accept method in the Listener interface; it
// waits for the next call and returns a generic Conn.
func (l *TCPListener) Accept() (Conn, error) {
    if !l.ok() {
        return nil, syscall.EINVAL
    }
    c, err := l.accept()
    if err != nil {
        return nil, &OpError{Op: "accept", Net: l.fd.net, Source: nil, Addr: l.fd.laddr, Err: err}
    }
    return c, nil
}

func (ln *TCPListener) accept() (*TCPConn, error) {
    fd, err := ln.fd.accept()
    if err != nil {
        return nil, err
    }
    tc := newTCPConn(fd)
    if ln.lc.KeepAlive >= 0 {
        setKeepAlive(fd, true)
        ka := ln.lc.KeepAlive
        if ln.lc.KeepAlive == 0 {
            ka = defaultTCPKeepAlive
        }
        setKeepAlivePeriod(fd, ka)
    }
    return tc, nil
}

func (fd *netFD) accept() (netfd *netFD, err error) {
```

```
    // 调用 poll.FD 的 Accept 方法接受新的 socket 连接，返回 socket 的 fd
    d, rsa, errcall, err := fd.pfd.Accept()
    if err != nil {
        if errcall != "" {
            err = wrapSyscallError(errcall, err)
        }
        return nil, err
    }
    // 以 socket fd 构造一个新的 netFD，代表这个新的 socket
    if netfd, err = newFD(d, fd.family, fd.sotype, fd.net); err != nil {
        poll.CloseFunc(d)
        return nil, err
    }
    // 调用 netFD 的 init 方法完成初始化
    if err = netfd.init(); err != nil {
        fd.Close()
        return nil, err
    }
    lsa, _ := syscall.Getsockname(netfd.pfd.Sysfd)
    netfd.setAddr(netfd.addrFunc()(lsa), netfd.addrFunc()(rsa))
    return netfd, nil
}
```

netFD.accept 方法里会再调用 poll.FD.Accept ，最后会使用 Linux 的系统调用 accept 来完成新连的接收，并且会把 accept 的 socket 设置成非阻塞 I/O 模式：

```
// Accept wraps the accept network call.
func (fd *FD) Accept() (int, syscall.Sockaddr, string, error) {
    if err := fd.readLock(); err != nil {
        return -1, nil, "", err
    }
    defer fd.readUnlock()

    if err := fd.pd.prepareRead(fd.isFile); err != nil {
        return -1, nil, "", err
    }
    for {
        // 使用 linux 系统调用 accept 接收新连接，创建对应的 socket
        s, rsa, errcall, err := accept(fd.Sysfd)
        // 因为 listener fd 在创建的时候已经设置成非阻塞的了，
        // 所以 accept 方法会直接返回，不管有没有新连接到来；如果 err == nil 则表示正常建立新
接，直接返回
        if err == nil {
            return s, rsa, "", err
        }
        // 如果 err != nil，则判断 err == syscall.EAGAIN，符合条件则进入 pollDesc.waitRead 方法
        switch err {
        case syscall.EAGAIN:
            if fd.pd.pollable() {
                // 如果当前没有发生期待的 I/O 事件，那么 waitRead 会通过 park goroutine 让逻辑 blo
k 在这里
                if err = fd.pd.waitRead(fd.isFile); err == nil {
                    continue
                }
```

```
            }
        case syscall.ECONNABORTED:
            // This means that a socket on the listen
            // queue was closed before we Accept()ed it;
            // it's a silly error, so try again.
            continue
        }
        return -1, nil, errcall, err
    }
}

// 使用 linux 的 accept 系统调用接收新连接并把这个 socket fd 设置成非阻塞 I/O
ns, sa, err := Accept4Func(s, syscall.SOCK_NONBLOCK|syscall.SOCK_CLOEXEC)
// On Linux the accept4 system call was introduced in 2.6.28
// kernel and on FreeBSD it was introduced in 10 kernel. If we
// get an ENOSYS error on both Linux and FreeBSD, or EINVAL
// error on Linux, fall back to using accept.

// Accept4Func is used to hook the accept4 call.
var Accept4Func func(int, int) (int, syscall.Sockaddr, error) = syscall.Accept4
```

pollDesc.waitRead 方法主要负责检测当前这个 pollDesc 的上层 netFD 对应的 fd 是否有『期待的』/O 事件发生，如果有就直接返回，否则就 park 住当前的 goroutine 并持续等待直至对应的 fd 上发可读/可写或者其他『期待的』I/O 事件为止，然后它就会返回到外层的 for 循环，让 goroutine 继执行逻辑。

poll.FD.Accept() 返回之后，会构造一个对应这个新 socket 的 netFD，然后调用 init() 方法完成初化，这个 init 过程和前面 net.Listen() 是一样的，调用链：netFD.init() --> poll.FD.Init() --> poll.pol Desc.init()，最终又会走到这里：

```
var serverInit sync.Once

func (pd *pollDesc) init(fd *FD) error {
    serverInit.Do(runtime_pollServerInit)
    ctx, errno := runtime_pollOpen(uintptr(fd.Sysfd))
    if errno != 0 {
        if ctx != 0 {
            runtime_pollUnblock(ctx)
            runtime_pollClose(ctx)
        }
        return syscall.Errno(errno)
    }
    pd.runtimeCtx = ctx
    return nil
}
```

然后把这个 socket fd 注册到 listener 的 epoll 实例的事件队列中去，等待 I/O 事件。

Conn.Read/Conn.Write

我们先来看看 Conn.Read 方法是如何实现的，原理其实和 Listener.Accept 是一样的，具体调用链是首先调用 conn 的 netFD.Read ，然后内部再调用 poll.FD.Read ，最后使用 Linux 的系统调用 rea : syscall.Read 完成数据读取：

```
// Implementation of the Conn interface.

// Read implements the Conn Read method.
func (c *conn) Read(b []byte) (int, error) {
    if !c.ok() {
        return 0, syscall.EINVAL
    }
    n, err := c.fd.Read(b)
    if err != nil && err != io.EOF {
        err = &OpError{Op: "read", Net: c.fd.net, Source: c.fd.laddr, Addr: c.fd.raddr, Err: err}
    }
    return n, err
}

func (fd *netFD) Read(p []byte) (n int, err error) {
    n, err = fd.pfd.Read(p)
    runtime.KeepAlive(fd)
    return n, wrapSyscallError("read", err)
}

// Read implements io.Reader.
func (fd *FD) Read(p []byte) (int, error) {
    if err := fd.readLock(); err != nil {
        return 0, err
    }
    defer fd.readUnlock()
    if len(p) == 0 {
        // If the caller wanted a zero byte read, return immediately
        // without trying (but after acquiring the readLock).
        // Otherwise syscall.Read returns 0, nil which looks like
        // io.EOF.
        // TODO(bradfitz): make it wait for readability? (Issue 15735)
        return 0, nil
    }
    if err := fd.pd.prepareRead(fd.isFile); err != nil {
        return 0, err
    }
    if fd.IsStream && len(p) > maxRW {
        p = p[:maxRW]
    }
    for {
        // 尝试从该 socket 读取数据，因为 socket 在被 listener accept 的时候设置成
        // 了非阻塞 I/O，所以这里同样也是直接返回，不管有没有可读的数据
        n, err := syscall.Read(fd.Sysfd, p)
        if err != nil {
            n = 0
            // err == syscall.EAGAIN 表示当前没有期待的 I/O 事件发生，也就是 socket 不可读
            if err == syscall.EAGAIN && fd.pd.pollable() {
                // 如果当前没有发生期待的 I/O 事件，那么 waitRead
                // 会通过 park goroutine 让逻辑 block 在这里
                if err = fd.pd.waitRead(fd.isFile); err == nil {
                    continue
                }
            }
```

```
            // On MacOS we can see EINTR here if the user
            // pressed ^Z.  See issue #22838.
            if runtime.GOOS == "darwin" && err == syscall.EINTR {
                continue
            }
        }
        err = fd.eofError(n, err)
        return n, err
    }
}
```

conn.Write 和 conn.Read 的原理是一致的，它也是通过类似 pollDesc.waitRead 的 pollDesc.wait rite 来 park 住 goroutine 直至期待的 I/O 事件发生才返回，而 pollDesc.waitWrite 的内部实现原和 pollDesc.waitRead 是一样的，都是基于 runtime_pollWait ，这里就不再赘述。

pollDesc.waitRead

pollDesc.waitRead 内部调用了 runtime_pollWait 来达成无 I/O 事件时 park 住 goroutine 的目的：

```
//go:linkname poll_runtime_pollWait internal/poll.runtime_pollWait
func poll_runtime_pollWait(pd *pollDesc, mode int) int {
    err := netpollcheckerr(pd, int32(mode))
    if err != 0 {
        return err
    }
    // As for now only Solaris, illumos, and AIX use level-triggered IO.
    if GOOS == "solaris" || GOOS == "illumos" || GOOS == "aix" {
        netpollarm(pd, mode)
    }
    // 进入 netpollblock 并且判断是否有期待的 I/O 事件发生，
    // 这里的 for 循环是为了一直等到 io ready
    for !netpollblock(pd, int32(mode), false) {
        err = netpollcheckerr(pd, int32(mode))
        if err != 0 {
            return err
        }
        // Can happen if timeout has fired and unblocked us,
        // but before we had a chance to run, timeout has been reset.
        // Pretend it has not happened and retry.
    }
    return 0
}

// returns true if IO is ready, or false if timedout or closed
// waitio - wait only for completed IO, ignore errors
func netpollblock(pd *pollDesc, mode int32, waitio bool) bool {
    // gpp 保存的是 goroutine 的数据结构 g，这里会根据 mode 的值决定是 rg 还是 wg
    // 后面调用 gopark 之后，会把当前的 goroutine 的抽象数据结构 g 存入 gpp 这个指针
    gpp := &pd.rg
    if mode == 'w' {
        gpp = &pd.wg
    }
```

```
    // set the gpp semaphore to WAIT
    // 这个 for 循环是为了等待 io ready 或者 io wait
    for {
        old := *gpp
        // gpp == pdReady 表示此时已有期待的 I/O 事件发生，
        // 可以直接返回 unblock 当前 goroutine 并执行响应的 I/O 操作
        if old == pdReady {
            *gpp = 0
            return true
        }
        if old != 0 {
            throw("runtime: double wait")
        }
        // 如果没有期待的 I/O 事件发生，则通过原子操作把 gpp 的值置为 pdWait 并退出 for 循环
        if atomic.Casuintptr(gpp, 0, pdWait) {
            break
        }
    }

    // need to recheck error states after setting gpp to WAIT
    // this is necessary because runtime_pollUnblock/runtime_pollSetDeadline/deadlineimpl
    // do the opposite: store to closing/rd/wd, membarrier, load of rg/wg

    // waitio 此时是 false，netpollcheckerr 方法会检查当前 pollDesc 对应的 fd 是否是正常的，
    // 通常来说  netpollcheckerr(pd, mode) == 0 是成立的，所以这里会执行 gopark
    // 把当前 goroutine 给 park 住，直至对应的 fd 上发生可读/可写或者其他『期待的』I/O 事件为
,
    // 然后 unpark 返回，在 gopark 内部会把当前 goroutine 的抽象数据结构 g 存入
    // gpp(pollDesc.rg/pollDesc.wg) 指针里，以便在后面的 netpoll 函数取出 pollDesc 之后，
    // 把 g 添加到链表里返回，接着重新调度 goroutine
    if waitio || netpollcheckerr(pd, mode) == 0 {
        // 注册 netpollblockcommit 回调给 gopark，在 gopark 内部会执行它，保存当前 goroutine
到 gpp
        gopark(netpollblockcommit, unsafe.Pointer(gpp), waitReasonIOWait, traceEvGoBlockNet,
5)
    }
    // be careful to not lose concurrent READY notification
    old := atomic.Xchguintptr(gpp, 0)
    if old > pdWait {
        throw("runtime: corrupted polldesc")
    }
    return old == pdReady
}

// gopark 会停住当前的 goroutine 并且调用传递进来的回调函数 unlockf，从上面的源码我们可以
道这个函数是
// netpollblockcommit
func gopark(unlockf func(*g, unsafe.Pointer) bool, lock unsafe.Pointer, reason waitReason, tra
eEv byte, traceskip int) {
    if reason != waitReasonSleep {
        checkTimeouts() // timeouts may expire while two goroutines keep the scheduler busy
    }
    mp := acquirem()
    gp := mp.curg
```

```
    status := readgstatus(gp)
    if status != _Grunning && status != _Gscanrunning {
        throw("gopark: bad g status")
    }
    mp.waitlock = lock
    mp.waitunlockf = unlockf
    gp.waitreason = reason
    mp.waittraceev = traceEv
    mp.waittraceskip = traceskip
    releasem(mp)
    // can't do anything that might move the G between Ms here.
  // gopark 最终会调用 park_m，在这个函数内部会调用 unlockf，也就是 netpollblockcommit，
    // 然后会把当前的 goroutine，也就是 g 数据结构保存到 pollDesc 的 rg 或者 wg 指针里
    mcall(park_m)
}

// park continuation on g0.
func park_m(gp *g) {
    _g_ := getg()

    if trace.enabled {
        traceGoPark(_g_.m.waittraceev, _g_.m.waittraceskip)
    }

    casgstatus(gp, _Grunning, _Gwaiting)
    dropg()

    if fn := _g_.m.waitunlockf; fn != nil {
        // 调用 netpollblockcommit，把当前的 goroutine，
        // 也就是 g 数据结构保存到 pollDesc 的 rg 或者 wg 指针里
        ok := fn(gp, _g_.m.waitlock)
        _g_.m.waitunlockf = nil
        _g_.m.waitlock = nil
        if !ok {
            if trace.enabled {
                traceGoUnpark(gp, 2)
            }
            casgstatus(gp, _Gwaiting, _Grunnable)
            execute(gp, true) // Schedule it back, never returns.
        }
    }
    schedule()
}

// netpollblockcommit 在 gopark 函数里被调用
func netpollblockcommit(gp *g, gpp unsafe.Pointer) bool {
    // 通过原子操作把当前 goroutine 抽象的数据结构 g，也就是这里的参数 gp 存入 gpp 指针，
    // 此时 gpp 的值是 pollDesc 的 rg 或者 wg 指针
    r := atomic.Casuintptr((*uintptr)(gpp), pdWait, uintptr(unsafe.Pointer(gp)))
    if r {
        // Bump the count of goroutines waiting for the poller.
        // The scheduler uses this to decide whether to block
        // waiting for the poller if there is nothing else to do.
        atomic.Xadd(&netpollWaiters, 1)
```

```
    }
    return r
}
```

netpoll

前面已经从源码的角度分析完了 netpoll 是如何通过 park goroutine 从而达到阻塞 Accept/Read/Write 的效果，而通过调用 gopark，goroutine 会被放置在某个等待队列中(如 channel 的 waitq ，此时 G 的状态由 _Grunning 为 _Gwaitting)，因此 G 必须被手动唤醒(通过 goready)，否则会丢失任务应用层阻塞通常使用这种方式。

所以，最后还有一个非常关键的问题是：当 I/O 事件发生之后，netpoll 是通过什么方式唤醒那些在 IO wait 的 goroutine 的？答案是通过 epoll_wait ，在 Go 源码中的 src/runtime/netpoll_epoll.go 件中有一个 func netpoll(block bool) gList 方法，它会内部调用 epoll_wait 获取就绪的 fd 列表，**将每个 fd 对应的 goroutine 添加到链表返回**：

```
// polls for ready network connections
// returns list of goroutines that become runnable
func netpoll(block bool) gList {
    if epfd == -1 {
        return gList{}
    }
    waitms := int32(-1)
    // 是否以阻塞模式调用 epoll_wait
    if !block {
        waitms = 0
    }
    var events [128]epollevent
retry:
    // 获取就绪的 fd 列表
    n := epollwait(epfd, &events[0], int32(len(events)), waitms)
    if n < 0 {
        if n != -_EINTR {
            println("runtime: epollwait on fd", epfd, "failed with", -n)
            throw("runtime: netpoll failed")
        }
        goto retry
    }
    // toRun 是一个 g 的链表，存储要恢复的 goroutines，最后返回给调用方
    var toRun gList
    for i := int32(0); i < n; i++ {
        ev := &events[i]
        if ev.events == 0 {
            continue
        }
        var mode int32
        // 判断发生的事件类型，读类型或者写类型
        if ev.events&(_EPOLLIN|_EPOLLRDHUP|_EPOLLHUP|_EPOLLERR) != 0 {
            mode += 'r'
        }
        if ev.events&(_EPOLLOUT|_EPOLLHUP|_EPOLLERR) != 0 {
            mode += 'w'
        }
        if mode != 0 {
```

```
            // 取出保存在 epollevent 里的 pollDesc
            pd := *(**pollDesc)(unsafe.Pointer(&ev.data))
            pd.everr = false
            if ev.events == _EPOLLERR {
                pd.everr = true
            }
            // 调用 netpollready，传入就绪 fd 的 pollDesc，把 fd 对应的 goroutine 添加到链表 toRun
中
            netpollready(&toRun, pd, mode)
        }
    }
    if block && toRun.empty() {
        goto retry
    }
    return toRun
}

// netpollready 调用 netpollunblock 返回就绪 fd 对应的 goroutine 的抽象数据结构 g
func netpollready(toRun *gList, pd *pollDesc, mode int32) {
    var rg, wg *g
    if mode == 'r' || mode == 'r'+'w' {
        rg = netpollunblock(pd, 'r', true)
    }
    if mode == 'w' || mode == 'r'+'w' {
        wg = netpollunblock(pd, 'w', true)
    }
    if rg != nil {
        toRun.push(rg)
    }
    if wg != nil {
        toRun.push(wg)
    }
}

// netpollunblock 会依据传入的 mode 决定从 pollDesc 的 rg 或者 wg 取出当时 gopark 之时存入

// goroutine 抽象数据结构 g 并返回
func netpollunblock(pd *pollDesc, mode int32, ioready bool) *g {
    // mode == 'r' 代表当时 gopark 是为了等待读事件，而 mode == 'w' 则代表是等待写事件
    gpp := &pd.rg
    if mode == 'w' {
        gpp = &pd.wg
    }

    for {
        // 取出 gpp 存储的 g
        old := *gpp
        if old == pdReady {
            return nil
        }
        if old == 0 && !ioready {
            // Only set READY for ioready. runtime_pollWait
            // will check for timeout/cancel before waiting.
            return nil
```

```
        }
        var new uintptr
        if ioready {
            new = pdReady
        }
        // 重置 pollDesc 的 rg 或者 wg
        if atomic.Casuintptr(gpp, old, new) {
            if old == pdReady || old == pdWait {
                old = 0
            }
            // 通过万能指针还原成 g 并返回
            return (*g)(unsafe.Pointer(old))
        }
    }
}
```

而 Go 在多种场景下都可能会调用 netpoll 检查文件描述符状态。寻找到 I/O 就绪的 socket fd，并到这些 socket fd 对应的轮询器中附带的信息，根据这些信息将之前等待这些 socket fd 就绪的 goroutine 状态修改为 _Grunnable 。执行完 netpoll 之后，会返回一个就绪 fd 列表对应的 goroutine 列，接下来将就绪的 goroutine 加入到调度队列中，等待调度运行。

首先，在 Go runtime scheduler 正常调度 goroutine 之时就有可能会调用 netpoll 获取到已就绪的 d 对应的 goroutine 来调度执行：

```
// One round of scheduler: find a runnable goroutine and execute it.
// Never returns.
func schedule() {
    // ...

  if gp == nil {
        gp, inheritTime = findrunnable() // blocks until work is available
    }

    // ...
}

// Finds a runnable goroutine to execute.
// Tries to steal from other P's, get g from global queue, poll network.
func findrunnable() (gp *g, inheritTime bool) {
  // ...

  // Poll network.
    // This netpoll is only an optimization before we resort to stealing.
    // We can safely skip it if there are no waiters or a thread is blocked
    // in netpoll already. If there is any kind of logical race with that
    // blocked thread (e.g. it has already returned from netpoll, but does
    // not set lastpoll yet), this thread will do blocking netpoll below
    // anyway.
    if netpollinited() && atomic.Load(&netpollWaiters) > 0 && atomic.Load64(&sched.lastpoll)
!= 0 {
        if list := netpoll(false); !list.empty() { // non-blocking
            gp := list.pop()
            injectglist(&list)
            casgstatus(gp, _Gwaiting, _Grunnable)
```

```
            if trace.enabled {
                traceGoUnpark(gp, 0)
            }
            return gp, false
        }
    }

    // ...
}
```

Go scheduler 的核心方法 schedule 里会调用一个叫 findrunable() 的方法获取可运行的 goroutine 来执行，而在 findrunable() 方法里就调用了 netpoll 获取已就绪的 fd 列表对应的 goroutine 列表。

另外， sysmon 监控线程也可能会调用到 netpoll ：

```
// Always runs without a P, so write barriers are not allowed.
//
//go:nowritebarrierrec
func sysmon() {
        // ...
        now := nanotime()
        if netpollinited() && lastpoll != 0 && lastpoll+10*1000*1000 < now {
            atomic.Cas64(&sched.lastpoll, uint64(lastpoll), uint64(now))
            // 以非阻塞的方式调用 netpoll 获取就绪 fd 列表
            list := netpoll(false) // non-blocking - returns list of goroutines
            if !list.empty() {
                // Need to decrement number of idle locked M's
                // (pretending that one more is running) before injectglist.
                // Otherwise it can lead to the following situation:
                // injectglist grabs all P's but before it starts M's to run the P's,
                // another M returns from syscall, finishes running its G,
                // observes that there is no work to do and no other running M's
                // and reports deadlock.
                incidlelocked(-1)
                // 将其插入调度器的runnable列表中（全局），等待被调度执行
                injectglist(&list)
                incidlelocked(1)
            }
        }
        // retake P's blocked in syscalls
        // and preempt long running G's
        if retake(now) != 0 {
            idle = 0
        } else {
            idle++
        }
        // check if we need to force a GC
        if t := (gcTrigger{kind: gcTriggerTime, now: now}); t.test() && atomic.Load(&forcegc.idle)
= 0 {
            lock(&forcegc.lock)
            forcegc.idle = 0
            var list gList
            list.push(forcegc.g)
            injectglist(&list)
```

```
            unlock(&forcegc.lock)
        }
        if debug.schedtrace > 0 && lasttrace+int64(debug.schedtrace)*1000000 <= now {
            lasttrace = now
            schedtrace(debug.scheddetail > 0)
        }
    }
}
```

Go runtime 在程序启动的时候会创建一个独立的 M 作为监控线程，叫 sysmon ，这个线程为系统的 daemon 线程，无需 P 即可运行， sysmon 每 20us~10ms 运行一次。 sysmon 中以轮询的方式行以下操作（如上面的代码所示）：

1. 以非阻塞的方式调用 runtime.netpoll ，从中找出能从网络 I/O 中唤醒的 G，并调用 injectglist ，将其插入调度器的 runnable 列表中（全局），调度触发时，有可能从这个全局 runnable 列表获取 G。然后再循环调用 startm ，直到所有 P 都不处于 _Pidle 状态。

2. 调用 retake ，抢占长时间处于 _Psyscall 状态的 P。

综上，Go 借助于 epoll/kqueue/iocp 和 runtime scheduler 等的帮助，设计出了自己的 I/O 多路用 netpoll，成功地让 Listener.Accept / conn.Read / conn.Write 等方法从开发者的角度看来是同模式。

Go netpoll 的价值

通过前面对源码的分析，我们现在知道 Go netpoll 依托于 runtime scheduler，为开发者提供了一强大的同步网络编程模式；然而，Go netpoll 存在的意义却远不止于此，Go netpoll I/O 多路复用配 Non-blocking I/O 而打造出来的这个原生网络模型，它最大的价值是把网络 I/O 的控制权牢牢掌在 Go 自己的 runtime 里，关于这一点我们需要从 Go 的 runtime scheduler 说起，Go 的 G-P-M 度模型如下：

G 在运行过程中如果被阻塞在某个 system call 操作上，那么不光 G 会阻塞，执行该 G 的 M 也会解 P(实质是被 sysmon 抢走了)，与 G 一起进入 sleep 状态。如果此时有 idle 的 M，则 P 与其绑定继执行其他 G；如果没有 idle M，但仍然有其他 G 要去执行，那么就会创建一个新的 M。当阻塞在 system call 上的 G 完成 syscall 调用后，G 会去尝试获取一个可用的 P，如果没有可用的 P，那么 G 会

标记为 _Grunnable 并把它放入全局的 runqueue 中等待调度，之前的那个 sleep 的 M 将再次进入 sleep。

现在清楚为什么 netpoll 为什么一定要使用非阻塞 I/O 了吧？就是为了避免让操作网络 I/O 的 goroutine 陷入到系统调用从而进入内核态，因为一旦进入内核态，整个程序的控制权就会发生转移(到内核)不再属于用户进程了，那么也就无法借助于 Go 强大的 runtime scheduler 来调度业务程序的并发了而有了 netpoll 之后，借助于非阻塞 I/O，G 就再也不会因为系统调用的读写而陷入内核态，当 G 阻塞在某个 network I/O 操作上时，实际上它不是因为陷入内核态被阻塞住了，而是被 Go runtime 用 gopark 给 park 住了，此时 G 会被放置到某个 wait queue 中，而 M 会尝试运行下一个 _Grunnable 的 G，如果此时没有 _Grunnable 的 G 供 M 运行，那么 M 将解绑 P，并进入 sleep 状态。当 I/O vailable，在 wait queue 中的 G 会被唤醒，标记为 _Grunnable，放入某个可用的 P 的 local 队列，绑定一个 M 恢复执行。

Go netpoll 的问题

Go netpoll 的设计不可谓不精巧、性能也不可谓不高，配合 goroutine 开发网络应用的时候就一个：爽。因此 Go 的网络编程模式是及其简洁高效的。然而，没有任何一种设计和架构是完美的，goroutine-per-connection 这种模式虽然简单高效，但是在某些极端的场景下也会暴露出问题：goroutine 虽然非常轻量，它的自定义栈内存初始值仅为 2KB，后面按需扩容；海量连接的业务场景下，goroutine-per-connection，此时 goroutine 数量以及消耗的资源就会呈线性趋势暴涨，首先给 Go runtim scheduler 造成极大的压力和侵占系统资源，然后资源被侵占又反过来影响 runtime 的调度，导致能大幅下降；此外，我们通过源码可以知道，Go netpoll 会通过 sync.Once 确保只初始化一个 epoll 实例，也就是说它是 single event-loop 模式，接受新连接和处理 I/O 事件是全部放在一个 thread 的，所以在海量连接同时又高频创建和销毁连接的业务场景下有可能会导致性能瓶颈。

Reactor 模式

目前在 Linux 平台下构建的高性能网络程序中，大都使用 Reactor 模式，比如 netty、libevent、libe、ACE，POE(Perl)、Twisted(Python)等。

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) 模式。

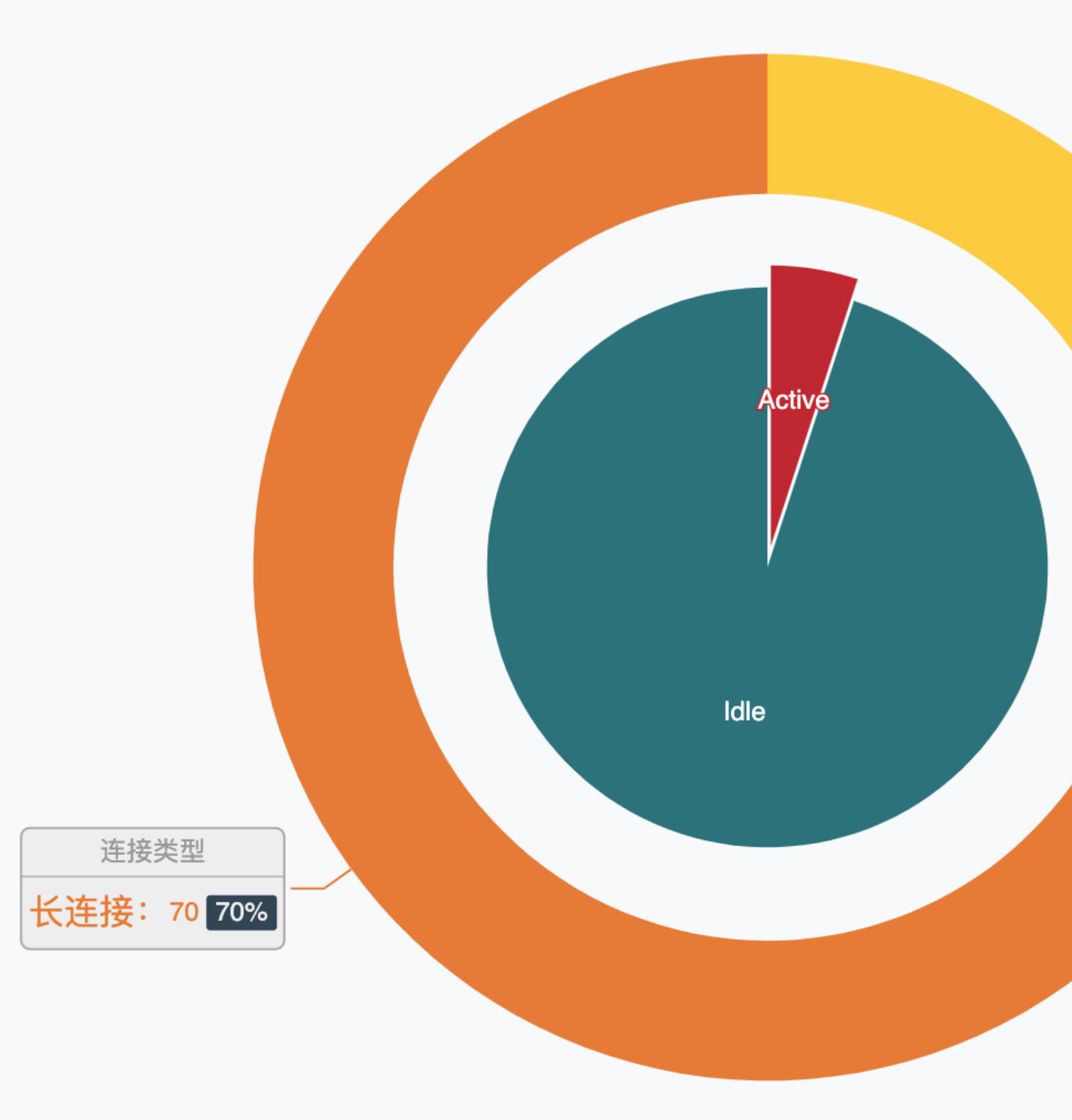
通常设置一个主线程负责做 event-loop 事件循环和 I/O 读写，通过 select/poll/epoll_wait 等系统用监听 I/O 事件，业务逻辑提交给其他工作线程去做。而所谓『非阻塞 I/O』的核心思想是指避免阻在 read() 或者 write() 或者其他的 I/O 系统调用上，这样可以最大限度的复用 event-loop 线程，让个线程能服务于多个 sockets。在 Reactor 模式中，I/O 线程只能阻塞在 I/O multiplexing 函数上（select/poll/epoll_wait）。

Reactor 模式通常的工作流程如下：

- Server 端完成在 bind&listen 之后，将 listenfd 注册到 epollfd 中，最后进入 event-loop 事件环。循环过程中会调用 select/poll/epoll_wait 阻塞等待，若有在 listenfd 上的新连接事件则解除阻返回，并调用 socket.accept 接收新连接 connfd，并将 connfd 加入到 epollfd 的 I/O 复用（监听队列。
- 当 connfd 上发生可读/可写事件也会解除 select/poll/epoll_wait 的阻塞等待，然后进行 I/O 读操作，这里读写 I/O 都是非阻塞 I/O，这样才不会阻塞 event-loop 的下一个循环。然而，这样容易裂业务逻辑，不易理解和维护。
- 调用 read 读取数据之后进行解码并放入队列中，等待工作线程处理。
- 工作线程处理完数据之后，返回到 event-loop 线程，由这个线程负责调用 write 把数据写回 client。

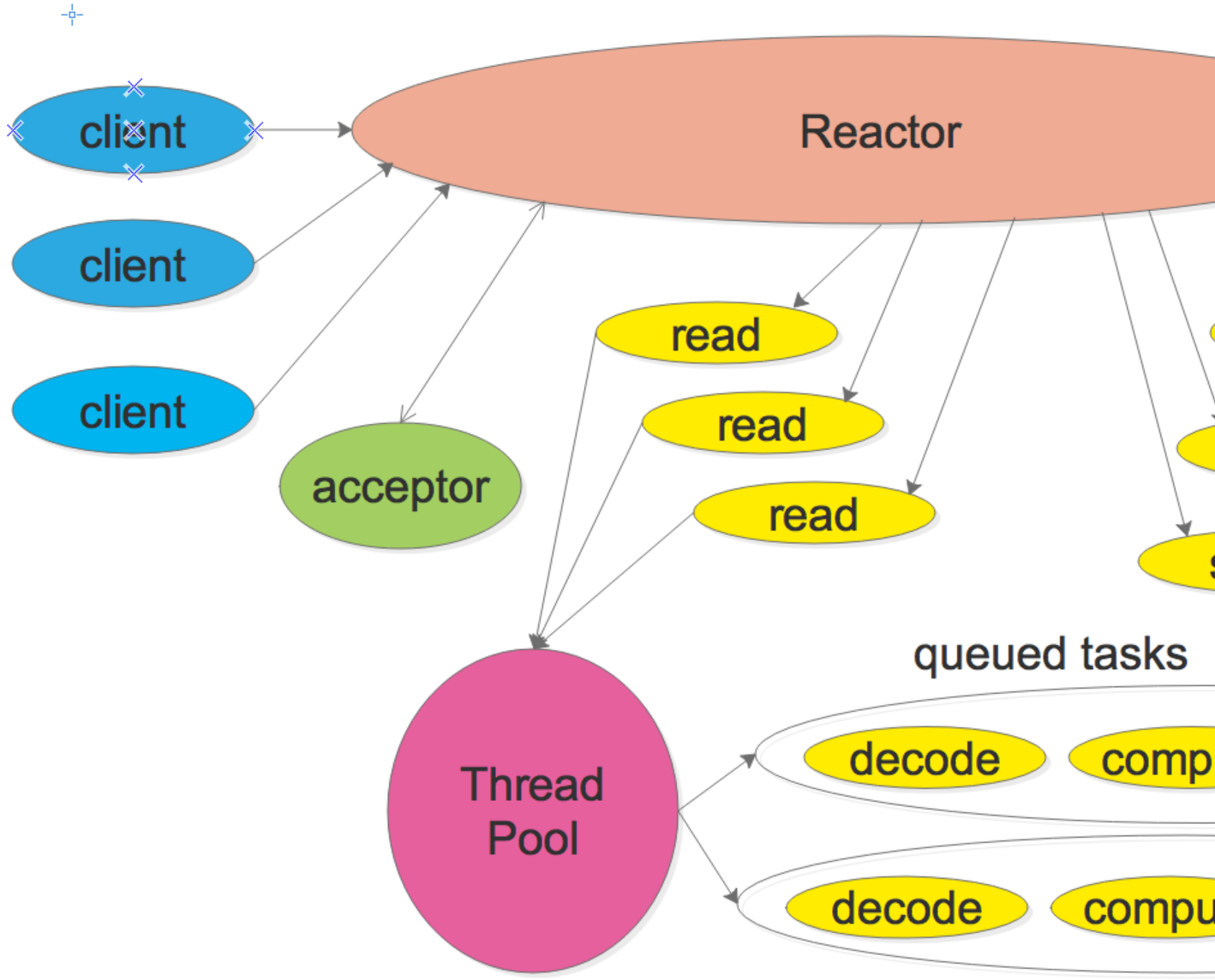
accept 连接以及 conn 上的读写操作若是在主线程完成，则要求是非阻塞 I/O，因为 Reactor 模式条最重要的原则就是：I/O 操作不能阻塞 event-loop 事件循环。**实际上 event loop 可能也可以是线程的，只是一个线程里只有一个 select/poll/epoll_wait。**

上面提到了 Go netpoll 在某些场景下可能因为创建太多的 goroutine 而过多地消耗系统资源，而在实世界的网络业务中，服务器持有的海量连接中在极短的时间窗口内只有极少数是 active 而大多数是 idle，就像这样（非真实数据，仅仅是为了比喻）：



那么为每一个连接指派一个 goroutine 就显得太过奢侈了，而 Reactor 模式这种利用 I/O 多路复用而只需要使用少量线程即可管理海量连接的设计就可以在这样网络业务中大显身手了：

原文链接：Go netpoll I/O 多路复用构建原生网络模型之源码深度解析



在绝大部分应用场景下，我推荐大家还是遵循 Go 的 best practices，以这种 netpoll 模式来构建自的网络应用。然而，在某些极度追求性能、压榨系统资源以及技术栈必须是原生 Go （不考虑 C/C++ 写中间层而 Go 写业务层）的业务场景下，我们可以考虑自己构建 Reactor 网络模型。

gnet

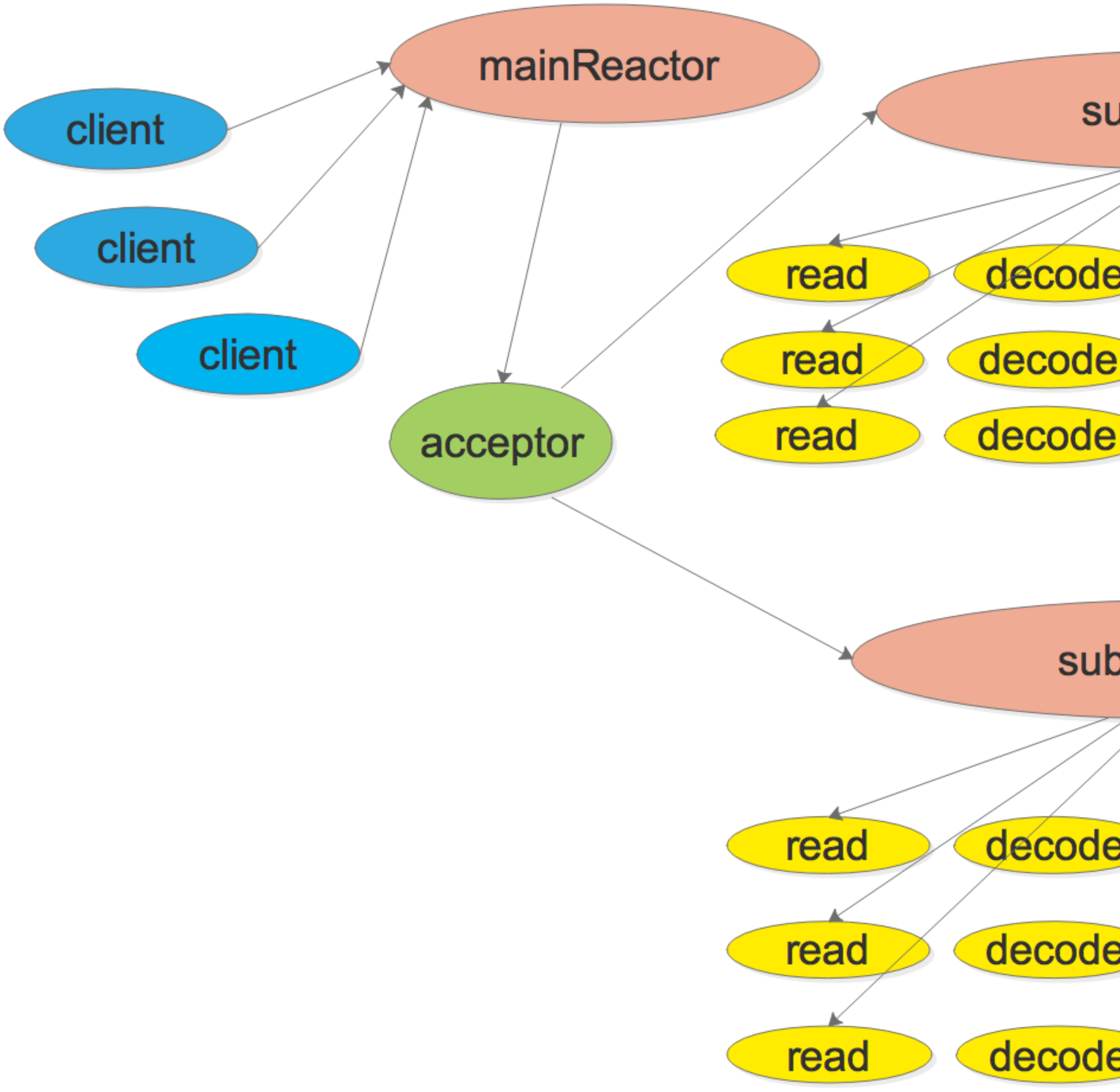
gnet 是一个基于事件驱动的高性能和轻量级网络框架，支持多种协议：TCP/UDP/Unix-Socket。它接使用 epoll 和 kqueue 系统调用而非标准 Golang 网络包：net 来构建网络应用，它的工作原理类两个开源的网络库：netty 和 libuv。

gnet 的亮点在于它是一个高性能、轻量级、非阻塞的纯 Go 实现的传输层（TCP/UDP/Unix-Socket 网络框架，开发者可以使用 gnet 来实现自己的应用层网络协议，从而构建出自己的应用层网络应用比如在 gnet 上实现 HTTP 协议就可以创建出一个 HTTP 服务器 或者 Web 开发框架，实现 Redis 协就可以创建出自己的 Redis 服务器等等。

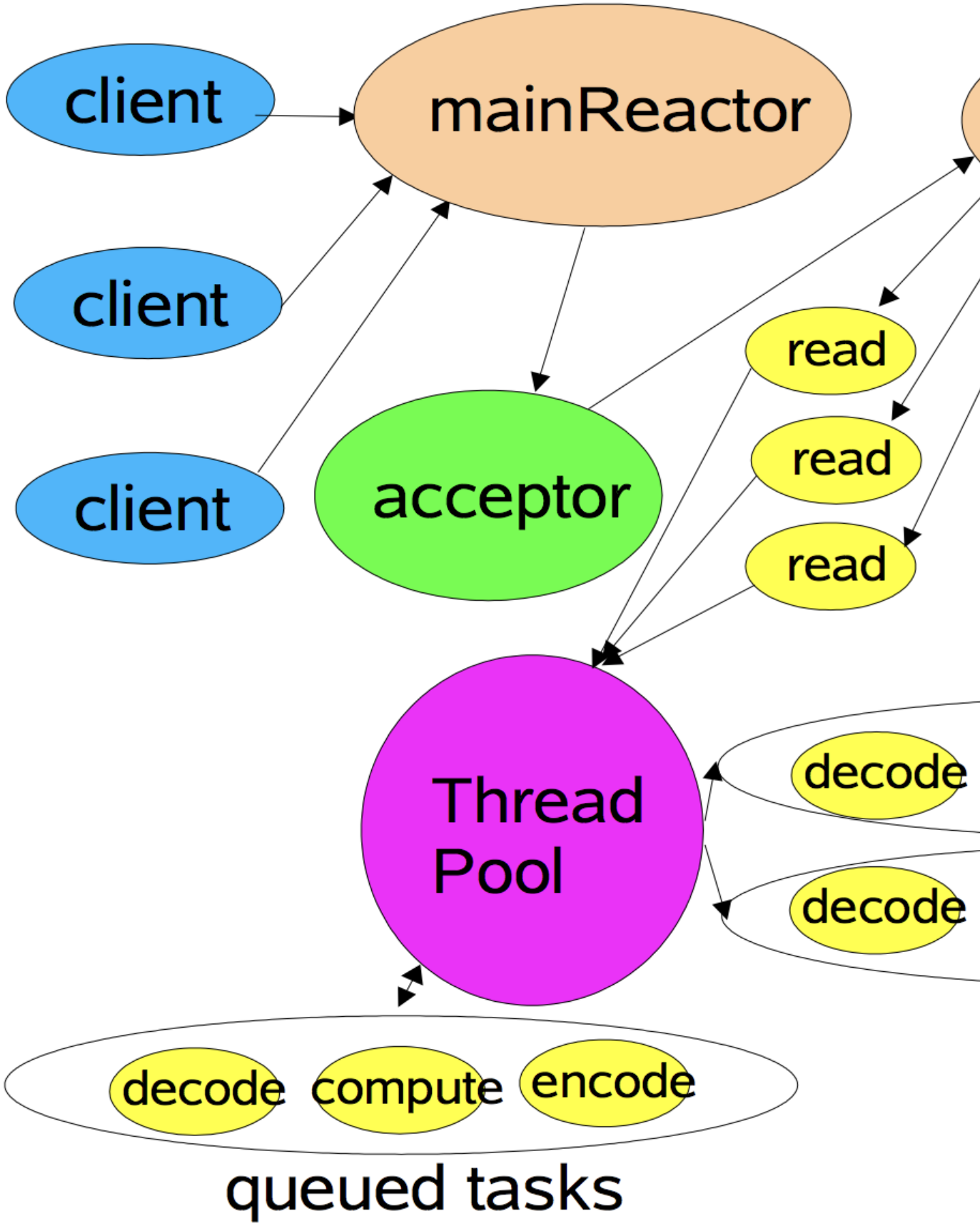
gnet，在某些极端的网络业务场景，比如海量连接、高频创建销毁连接等等场景，gnet 在性能和资占用上都远超 Go 原生的 net 包（基于 netpoll）。

原文链接：Go netpoll I/O 多路复用构建原生网络模型之源码深度解析

gnet 已经实现了 Multi-Reactors 和 Multi-Reactors + Goroutine Pool 两种网络模型，也得益于些网络模型，使得 gnet 成为一个高性能和低损耗的 Go 网络框架：



原文链接：Go netpoll I/O 多路复用构建原生网络模型之源码深度解析


client
client
client
mainReactor
acceptor
read
read
read
Thread Pool
decode
decode
decode
compute
encode
queued tasks

rocket **功能**

- 高性能 的基于多线程/Go 程网络模型的 event-loop 事件驱动
- 内置 Round-Robin 轮询负载均衡算法
- 内置 goroutine 池，由开源库 ants 提供支持
- 内置 bytes 内存池，由开源库 pool 提供支持
- 简洁的 APIs
- 基于 Ring-Buffer 的高效内存利用
- 支持多种网络协议：TCP、UDP、Unix Sockets
- 支持两种事件驱动机制：Linux 里的 epoll 以及 FreeBSD 里的 kqueue
- 支持异步写操作
- 灵活的事件定时器
- SO_REUSEPORT 端口重用
- 内置多种编解码器，支持对 TCP 数据流分包：LineBasedFrameCodec, DelimiterBasedFrameCodec, FixedLengthFrameCodec 和 LengthFieldBasedFrameCodec，参考自 netty codec，而且支持定制编解码器
- 支持 Windows 平台，基于 I~~OCP 事件驱动机制~~ Go 标准网络库
- 加入更多的负载均衡算法：随机、最少连接、一致性哈希等等
- 支持 TLS
- 实现 gnet 客户端

参考

- Linux I/O模式及 select、poll、epoll详解
- IO多路复用与Go网络库的实现
- 聊聊 Linux IO
- Go 调度模型
- linux 用户空间与内核空间——高端内存详解
- Goroutine 并发调度模型深度解析之手撸一个高性能协程池
- Go语言实现(2)：调度
- 关于select函数中timeval和fd_set重新设置的问题
- 也谈goroutine调度器