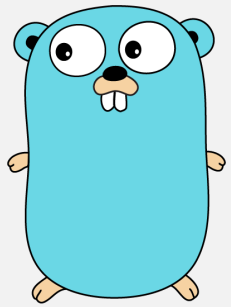
Go 语言网络编程和 gnet

BY 潘少

自我介绍

Tencent 腾讯

amazon



Golang

redis

kubernetes

ISTIO

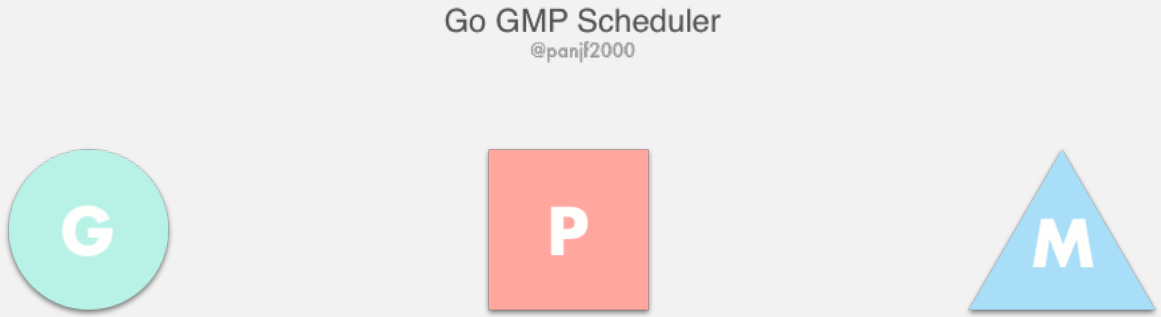
潘建锋, 后端开发。

Go 协程池 ants 和高性能 Go 网络库 gnet 作者，Golang、redis、istio、fasthttp、gin 等知名开源项目的活跃贡献者，专注于系统底层原理、高性能网络编程、架构设计、云原生、分布式。

目　　录

CONTENTS

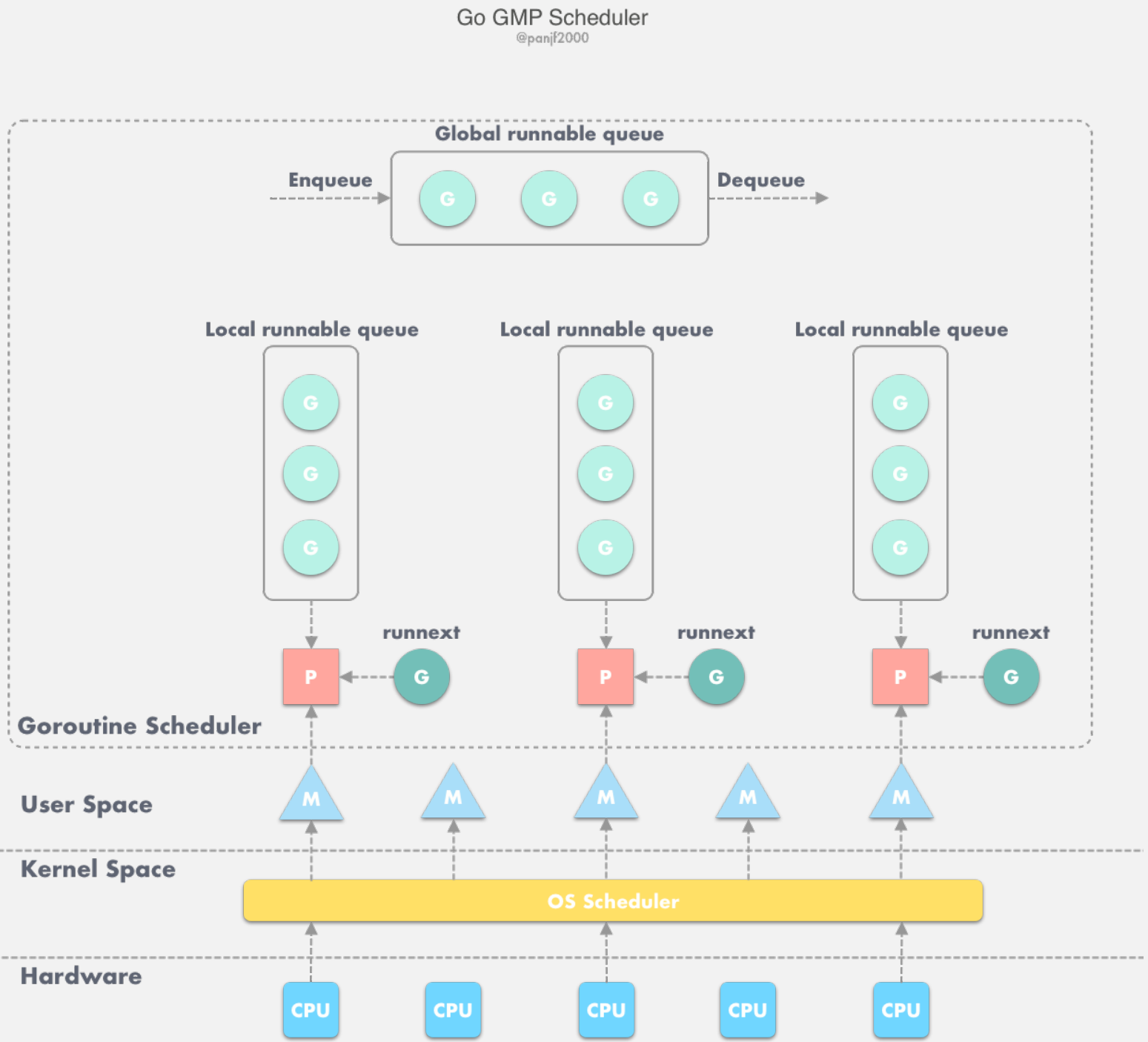
GMP 调度器



G: 表示 Goroutine，每个 Goroutine 对应一个 G 结构体，G 存储 Goroutine 的运行堆栈、状态以及任务函数，可重用。G 并非执行体，每个 G 需要绑定到 P 才能被调度执行。

P: Processor，表示逻辑处理器， 对 G 来说，P 相当于 CPU 核，G 只有绑定到 P(在 P 的 local runq 中)才能被调度。对 M 来说，P 提供了相关的执行环境(Context)，如内存分配状态(mcache)，任务队列(G)等，P 的数量决定了系统内最大可并行的 G 的数量（前提：物理 CPU 核数 >= P 的数量），P 的数量由用户设置的 GOMAXPROCS 决定，但是不论 GOMAXPROCS 设置为多大，P 的数量最大为 256。

M: Machine，OS 线程抽象，代表着真正执行计算的资源，在绑定有效的 P 后，进入 schedule 循环；而 schedule 循环的机制大致是从 Global 队列、P 的 Local 队列以及 wait 队列中获取 G，切换到 G 的执行栈上并执行 G 的函数，调用 goexit 做清理工作并回到 M，如此反复。M 并不保留 G 状态，这是 G 可以跨 M 调度的基础，M 的数量是不定的，由 Go Runtime 调整，为了防止创建过多 OS 线程导致系统调度不过来，目前默认最大限制为 10000 个。



GMP scheduler (从 M0 开始)会不断循环调用 runtime.schedule() 去调度 goroutines，而每个 goroutine 执行完成并退出之后，会再次调用 runtime.schedule()，使得调度器回到调度循环去执行其他的 goroutine，不断循环，永不停歇。

runtime.schedule --> runtime.execute --> runtime.gogo --> goroutine code --> runtime.goexit --> runtime.goexit1 --> runtime.mcall --> runtime.goexit0 --> runtime.schedule

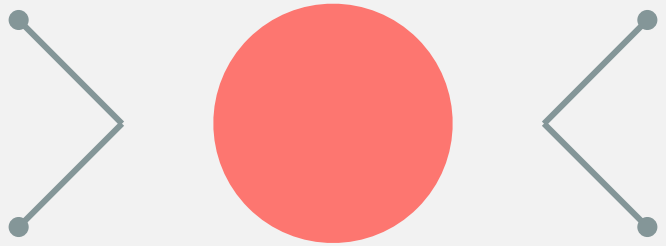
当我们使用 go 关键字启动一个新 goroutine 时，最终会调用 runtime.newproc --> runtime.newproc1，来得到 g，runtime.newproc1 会先从 P 的 gfree 缓存链表中查找可用的 g，若缓存未生效，则会新创建 g 给当前的业务函数，最后这个 g 会被传给 runtime.gogo 去真正执行。

GMP 调度器

轮询 goroutine 的顺序

本地运行队列

全局运行队列



网络轮询器

如何决定下一个要执行的 goroutine?

工作窃取（work stealing）

GMP 调度器

调度流程

当我们通过 go 关键字创建一个 goroutine 时，内部会调用 runtime.newproc --> runtime.newproc1 封存函数上下文信息得到 g；然后先尝试把这个待运行的 g 放到 P 的本地队列里，如果本地队列已经满了则放到全局队列中；

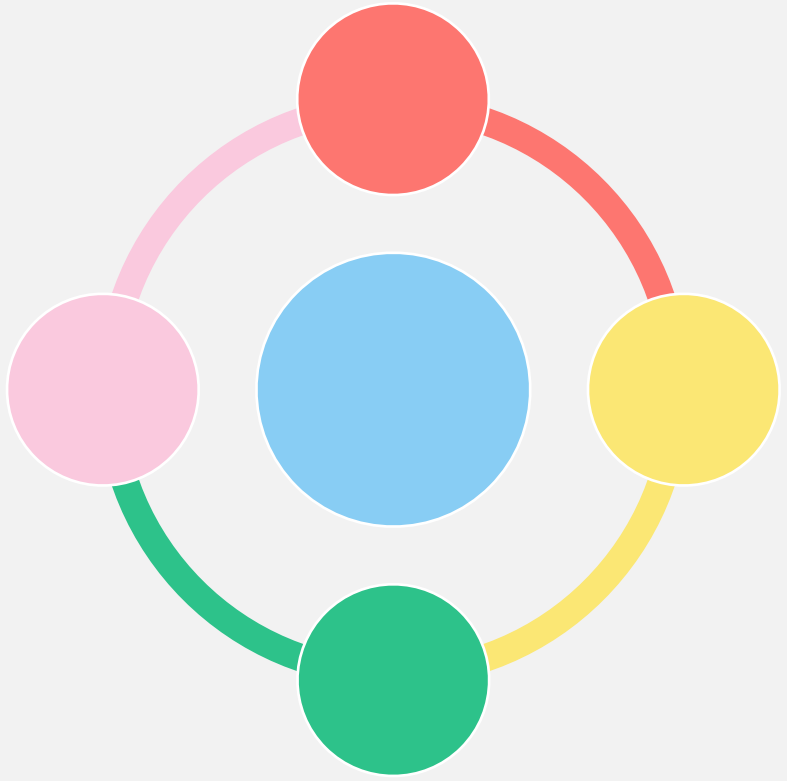
GMP 在启动之后会进入一个调度循环，用一种相对公平的方式查找并执行 g：

1. 首先每执行 61 次 g 之后会从全局队列拿一个 g 出来执行。
2. 然后尝试从 P 本地队列里取 g。
3. 如果这两个队列都为空，则调用 findrunable()，阻塞地取 g。

runtime.findrunnable() 函数的查找逻辑如下：

1. 先从 P 本地队列里查找。
2. 然后从全局队列里找。
3. 如果这两个队列都是空的，则从网络轮询器里查找。
4. 最后如果还是没找到 g，则去偷其他 P 的 g，优先尝试偷 timer。

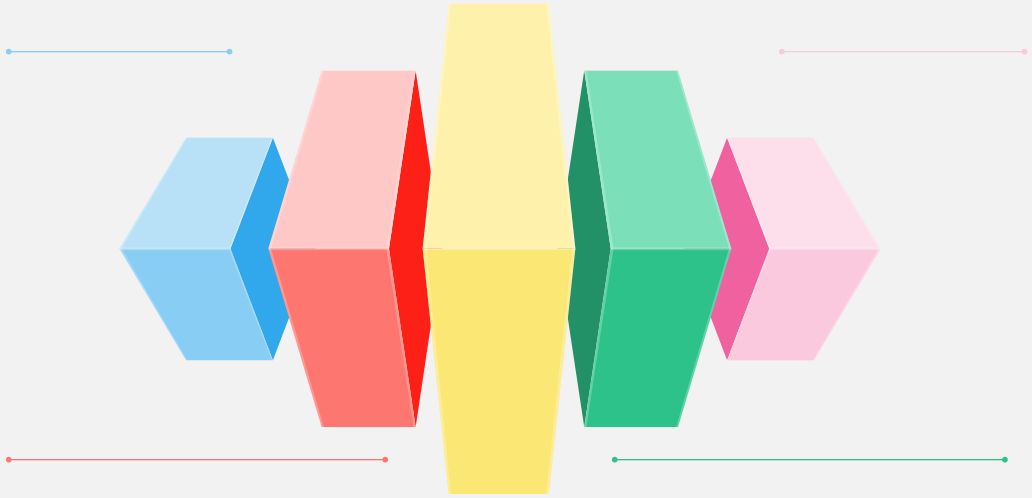
总之，你可以认为 findrunnable() 肯定会返回一个可运行的 g 给调度器，如果完全



阻塞调度

阻塞系统调用

使用阻塞式的系统调用，如读写操作



网络 I/O

读写网络接口

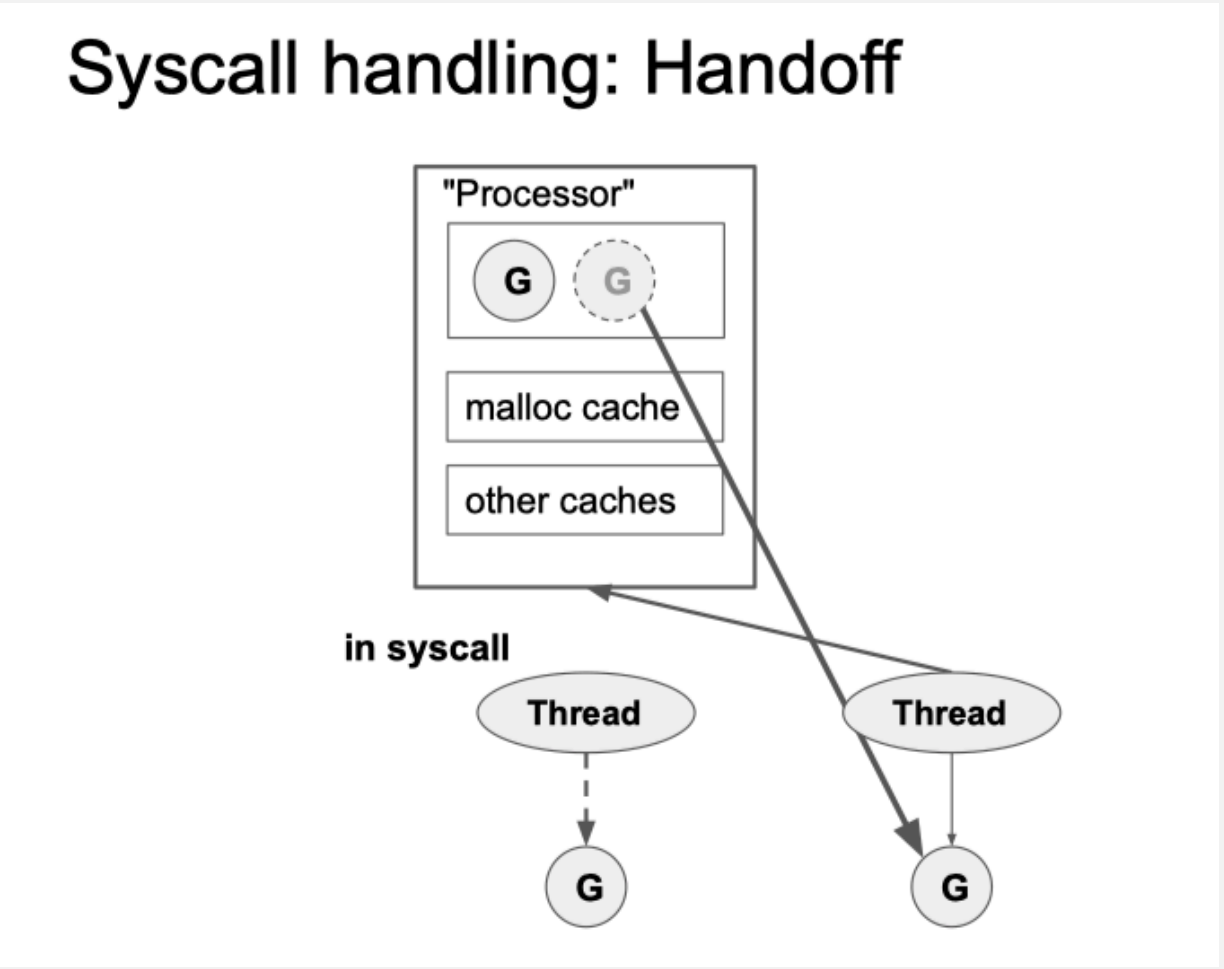
管道操作

golang channel 读写

内存同步原语

比如标准库 sync 提供的 API

系统调用阻塞



当 G 被阻塞在某个系统调用上时，此时 G 会阻塞在 _Gsyscall 状态，M 也处于 block on syscall 状态，此时的 M 可被(sysmon 线程)抢占调度：执行该 G 的 M 会与 P 解绑，而 P 则尝试与其它 idle 的 M 绑定，继续执行其它 G。如果没有其它 idle 的 M，但 P 的 Local 队列中仍然有 G 需要执行，则创建一个新的 M；当系统调用完成后，G 会重新尝试获取一个 idle 的 P 进入它的 Local 队列恢复执行，如果没有 idle 的 P，G 会被标记为 runnable 加入到 Global 队列。

用户态阻塞

当 goroutine 因为 channel 操作或者 network I/O 而阻塞时(实际上 golang 已经用 netpoller 实现了 goroutine 网络 I/O 阻塞不会导致 M 被阻塞，仅阻塞 G，这里仅仅是举个栗子)，对应的 G 会被放置到某个 wait 队列(如 channel 的 waitq 和 sendq)，该 G 的状态由 _Gruning 变为 _Gwaitting ，而 M 会跳过该 G 尝试获取并执行下一个 G，如果此时没有 runnable 的 G 供 M 运行，那么 M 将解绑 P，并进入 sleep 状态；当阻塞的 G 被另一端的 G2 唤醒时(比如 channel 的可读/写通知)，G 被标记为 runnable，尝试加入 G2 所在 P 的 runnext，然后再是 P 的 Local 队列和 Global 队列。

调度方式

协作式和抢占式

协作式

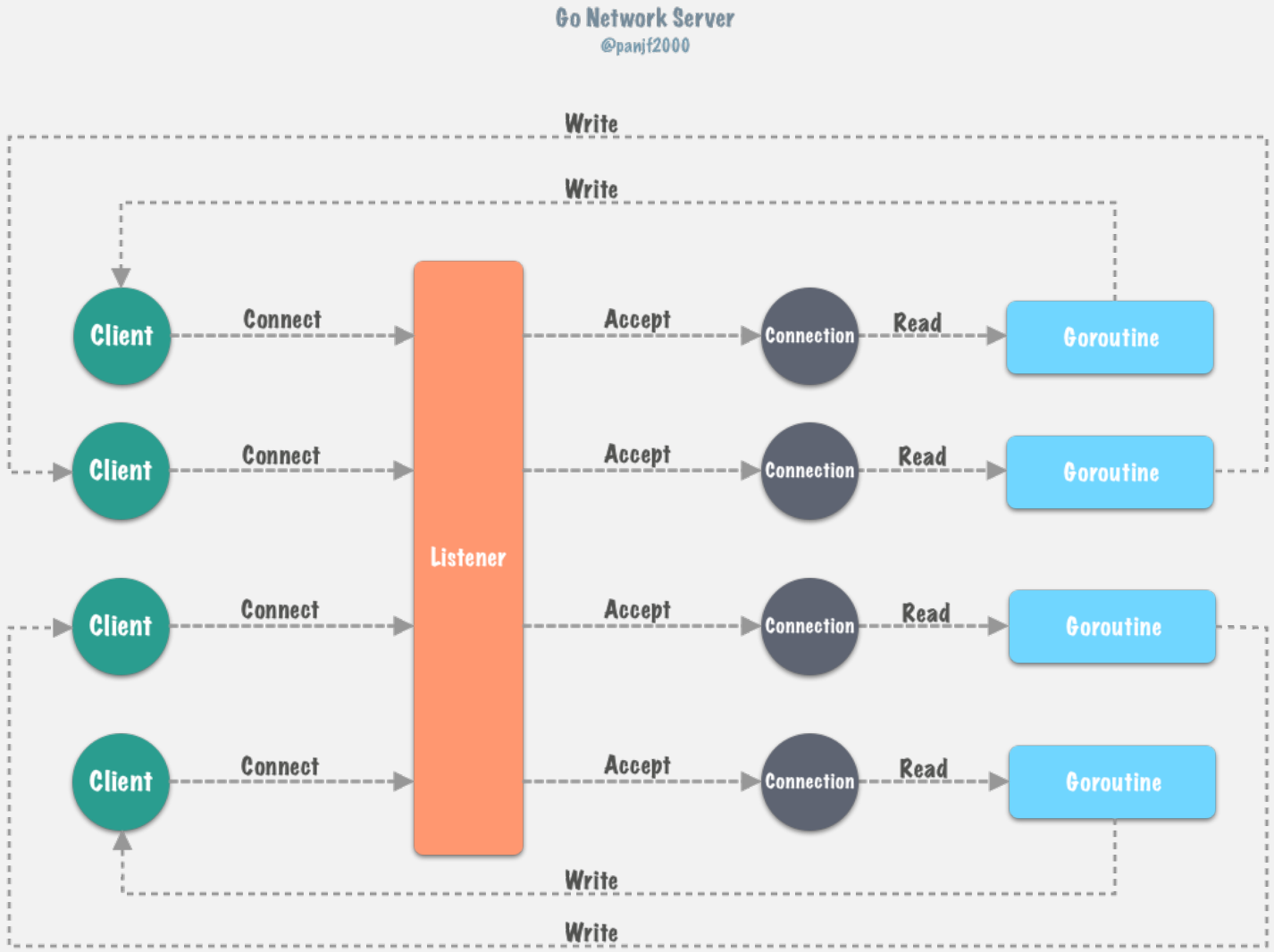
- runtime.GoSched()
- time.Sleep()
- Channel
- Mutex

抢占式

基于标记的抢占和基于信号的抢占，Go1.14 引入

Go 网络并发模型

一个典型的 go 网络服务器



Go 原生网络模型(netpoller)，编程模式是 goroutine-per-connection ，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 是否阻塞是无感知的，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切换。

而 Go netpoller 最底层的事件驱动技术肯定是基于 epoll/kqueue/iocp 这一类的 I/O 事件驱动技术，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 goroutines，从而极大地降低了程序员的心智负担！

Go 网络并发模型

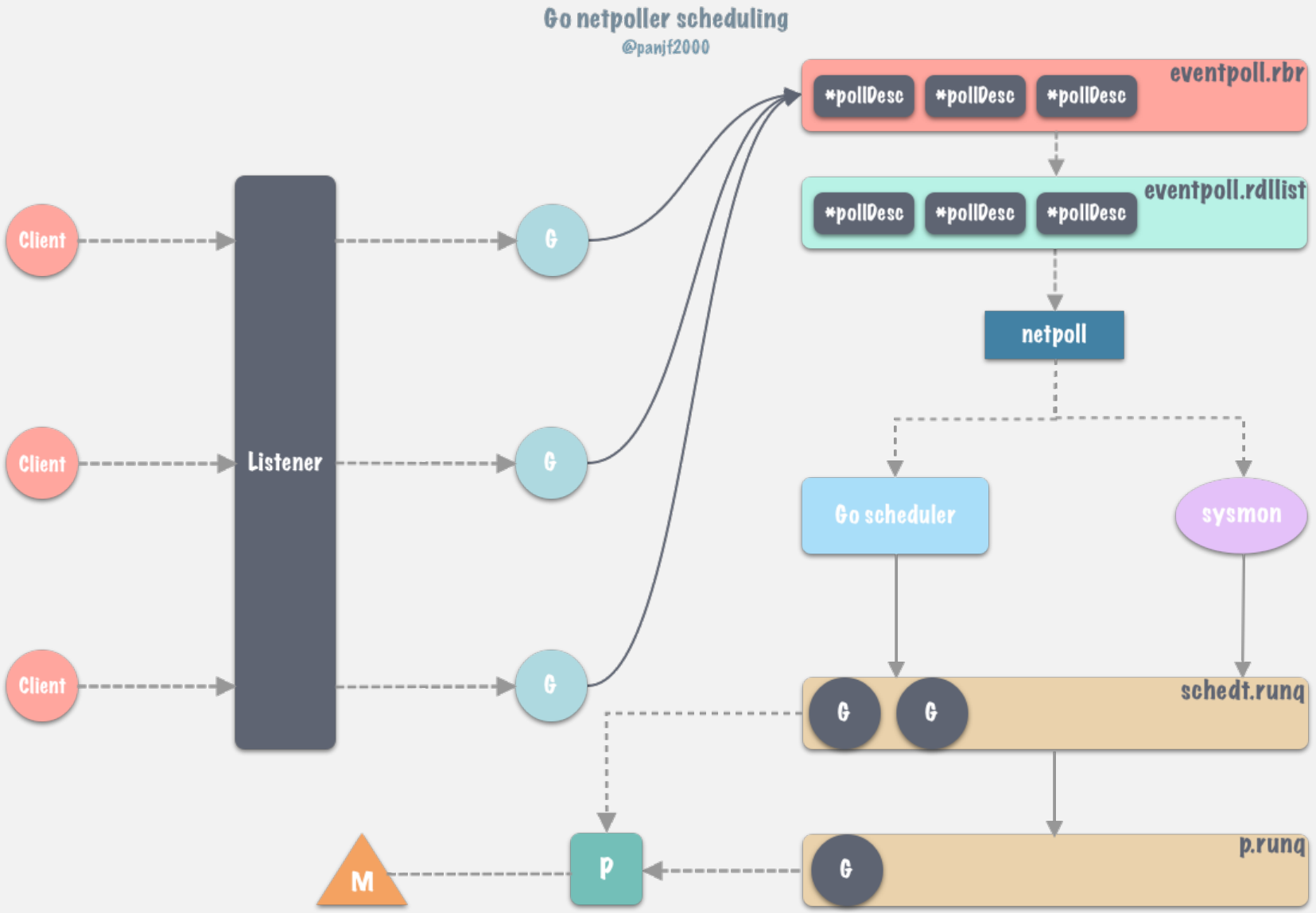
底层原理

首先，client 连接 server 的时候，listener 通过 accept 调用接收新 connection，每一个新 connection 都启动一个 goroutine 处理，accept 调用会把该 connection 的 fd 连带所在的 goroutine 上下文信息封装注册到 epoll 的监听列表里去，当 goroutine 调用 conn.Read 或者 conn.Write 等需要阻塞等待的函数时，会被 gopark 给封存起来并使之休眠，让 P 去执行本地调度队列里的下一个可执行的 goroutine，往后 Go scheduler 会在循环调度的 runtime.schedule() 函数以及 sysmon 监控线程中调用 runtime.netpoll 以获取可运行的 goroutine 列表并通过调用 injectglist 把剩下的 g 放入全局调度队列或者当前 P 本地调度队列去重新执行。

那么当 I/O 事件发生之后，netpoller 是通过什么方式唤醒那些在 I/O wait 的 goroutine 的？答案是通过 runtime.netpoll。

Go 网络并发模型

基本流程



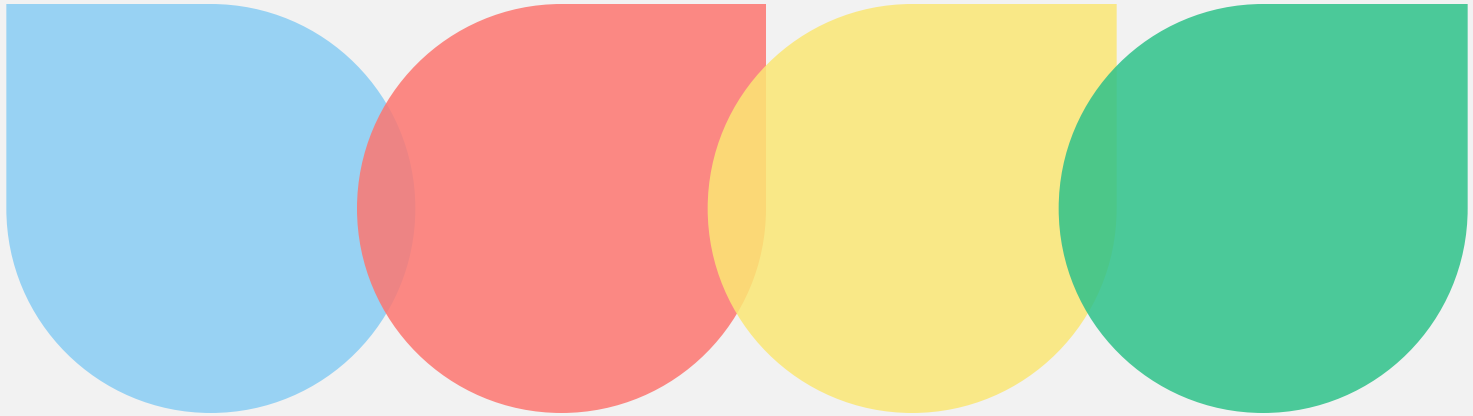
Go 网络并发模型

价值

netpoll 通过使用非阻塞 I/O，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 (长时间) 陷入内核态，当 G 被阻塞在某个 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 调用 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 available，在 epoll 的 eventpoll.rdr 中等待的 G 会被放到 eventpoll.rdllist 链表里并通过 netpoll 中的 epoll_wait 系统调用返回放置到全局调度队列或者 P 的本地调度队列，标记为 _Grunnable ，等待 P 绑定 M 恢复执行。

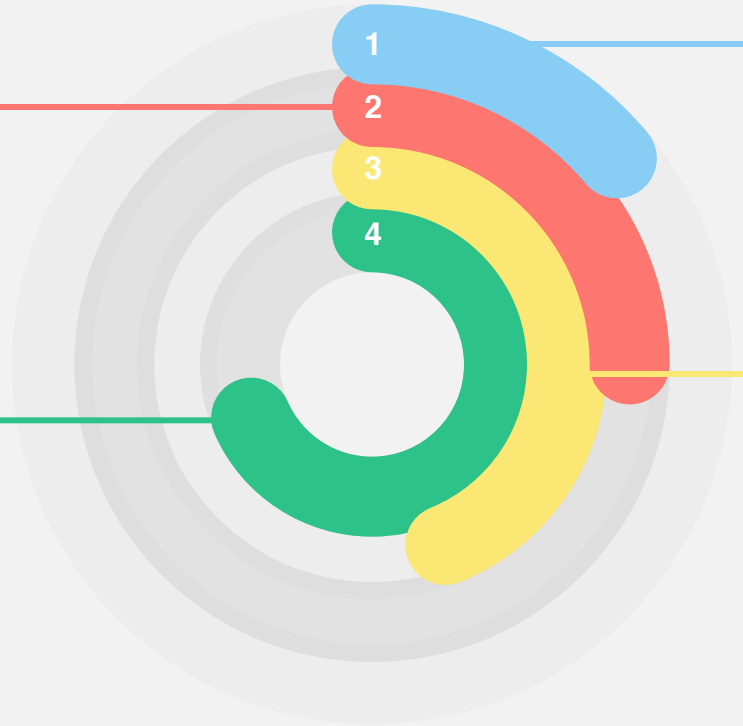
Go 网络模型的问题



Go netpoller 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字：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，虽然 Go scheduler 内部做了 g 的缓存链表，可以一定程度上缓解高频创建销毁 goroutine 的压力，但是对于瞬时性暴涨的长连接场景就无能为力了，大量的 goroutines 会被不断创建出来，从而对 Go runtime scheduler 造成极大的调度压力和侵占系统资源，然后资源被侵占又反过来影响 Go scheduler 的调度，进而导致性能下降。

真实的网络服务

连接的状态



短连接

瞬时连接，可能通信一次或几次就销毁的链接。

长连接

长时间保持，复用连接，不销毁。

闲置连接

在某一个时间窗口内闲置的链接。

活跃连接

在某一个时间窗口内通信频繁的链接。

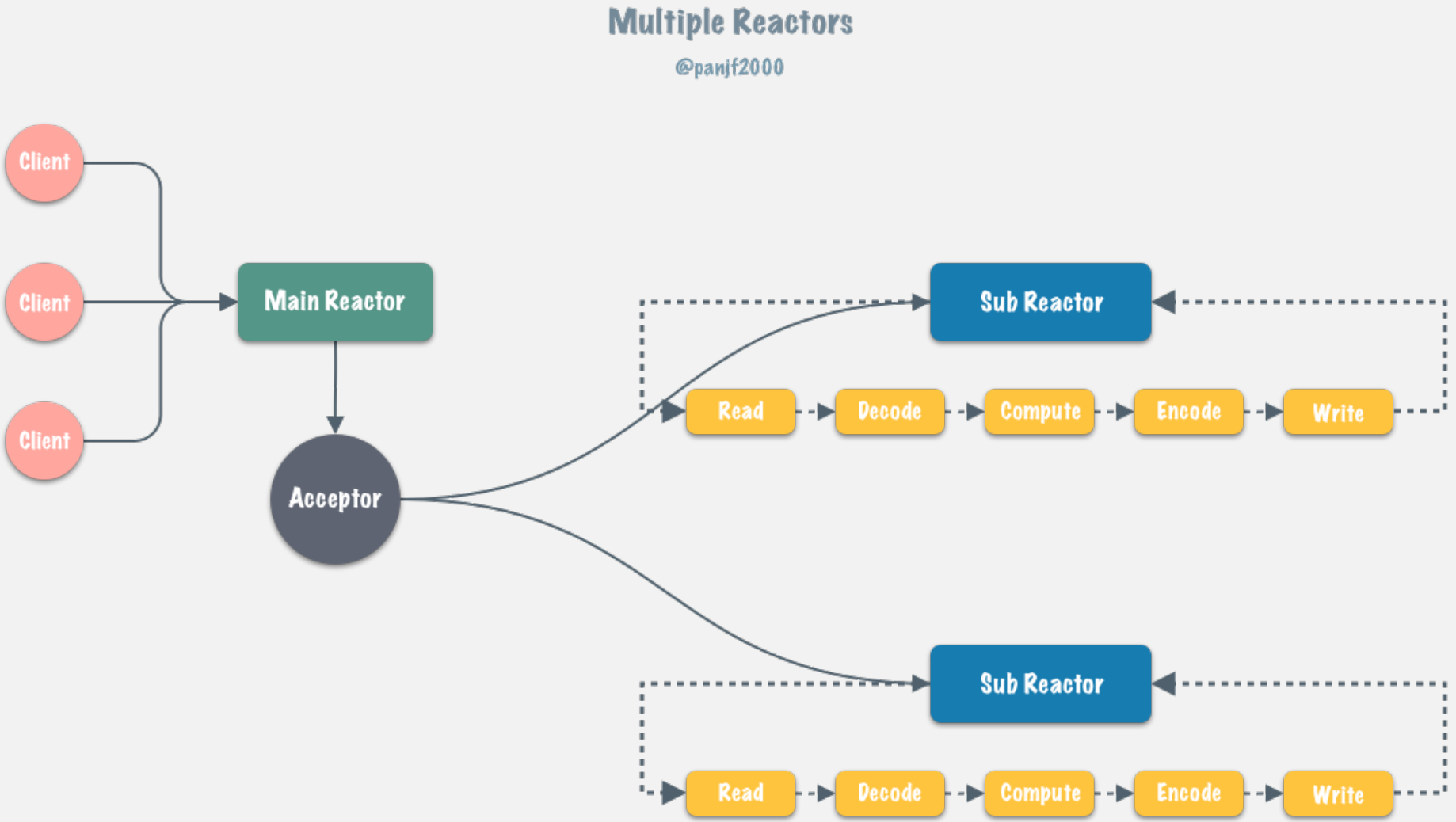
Reactor 网络并发模型

目前 Linux 平台上主流的高性能网络库/框架中，大都采用 Reactor 模式，比如 netty、libevent、libev、ACE，POE(Perl)、Twisted(Python)等。

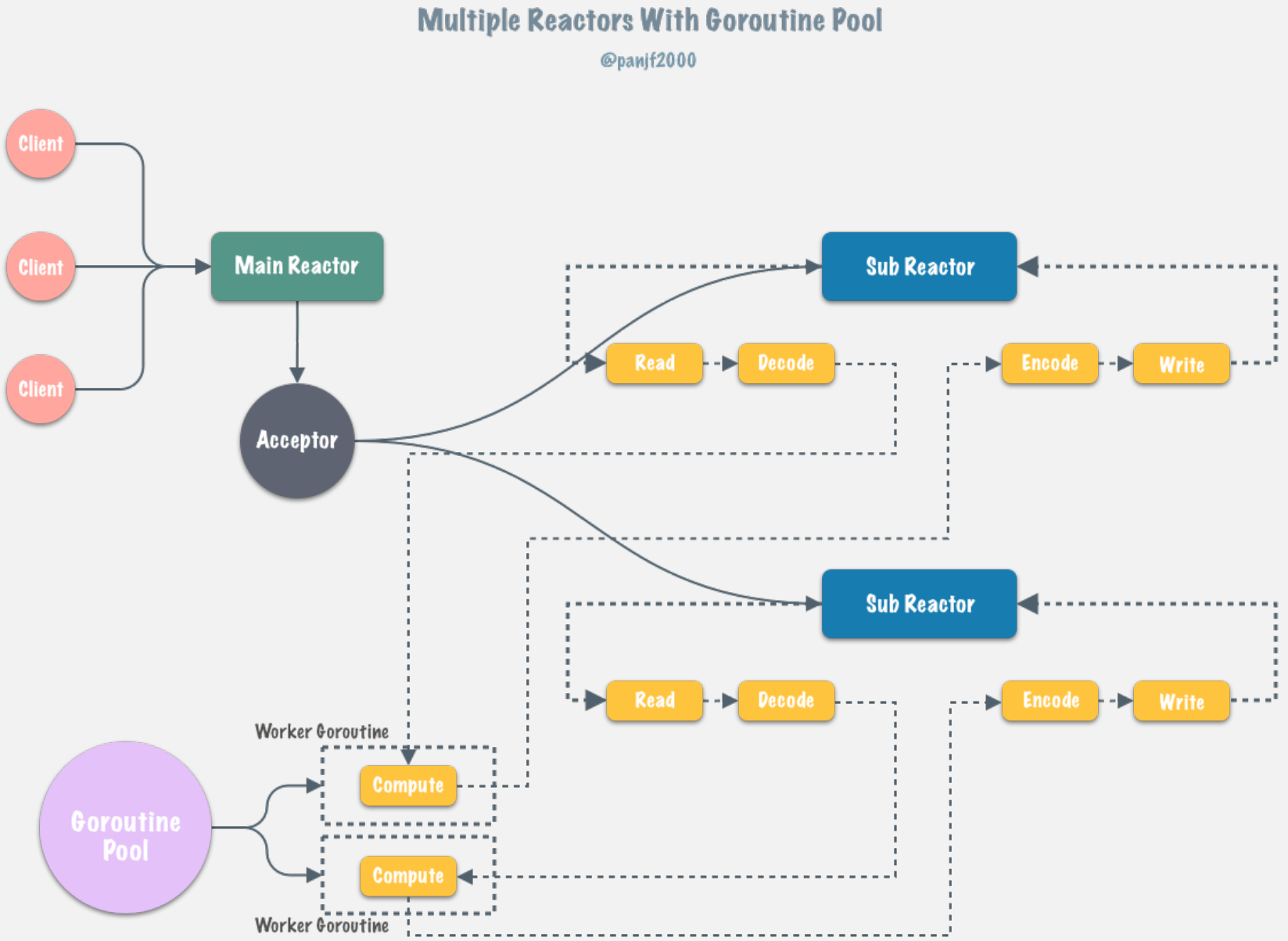
Reactor 模式本质上指的是使用 I/O 多路复用(I/O multiplexing) + 非阻塞 I/O(non-blocking I/O) 的模式。

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 等系统调用监听 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塞在 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，让一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 网络并发模型



Reactor 网络并发模型



Reactor 网络并发模型

Go 网络并发模型

功能特性

* 高性能 的基于多线程/Go程网络模型的 event-loop 事件驱动
* 内置 goroutine 池，由开源库 ants 提供支持
* 内置 bytes 内存池，由开源库 bytebufferpool 提供支持
* 整个生命周期是无锁的
* 简单易用的 APIs
* 高效、可重用而且自动伸缩的环形内存 buffer
* 支持多种网络协议/IPC 机制：TCP、UDP 和 Unix Domain Socket
* 支持多种负载均衡算法：Round-Robin(轮询)、Source-Addr-Hash(源地址哈希) 和 Least-Connections(最少连接数)
* 支持两种事件驱动机制：Linux 里的 epoll 以及 FreeBSD/DragonFly/Darwin 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

Milestone

gnet 发展历程

2019.9

发布首个 RC 版本

2019 年 9 月份，gnet 开始开发并于当月发布首个 RC 版本。

2019.10

发布首个 v1.0.0 RC 版本

2019 年 10 月，发布首个 v1.0.0 RC 版本。

这期间大幅度修改源码，做了大量的优化。

2020

gnet v1.0.0 正式版发布

发布 v1.0.0 正式版，内部基本架构定型。后续一直在快速迭代，添加新功能，提升框架性能。

此一年间，陆陆续续有一些小公司开始使用 gnet。

2021

继续发展

开始有一些大型互联网公司开始试用 gnet，渐渐在生产环境部署，目前有腾讯、腾讯游戏、爱奇艺、百度等等。

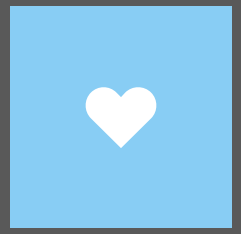
20XX

继续向前

gnet 会一直持续维护，持续迭代，满足用户的现实需求，致力于成为最好的 Go 语言 Reactor 网络框架。

感谢您的观看

Thank you for coming



Social Network

Address

Email