云原生训练营

模块二：Go 语言进阶

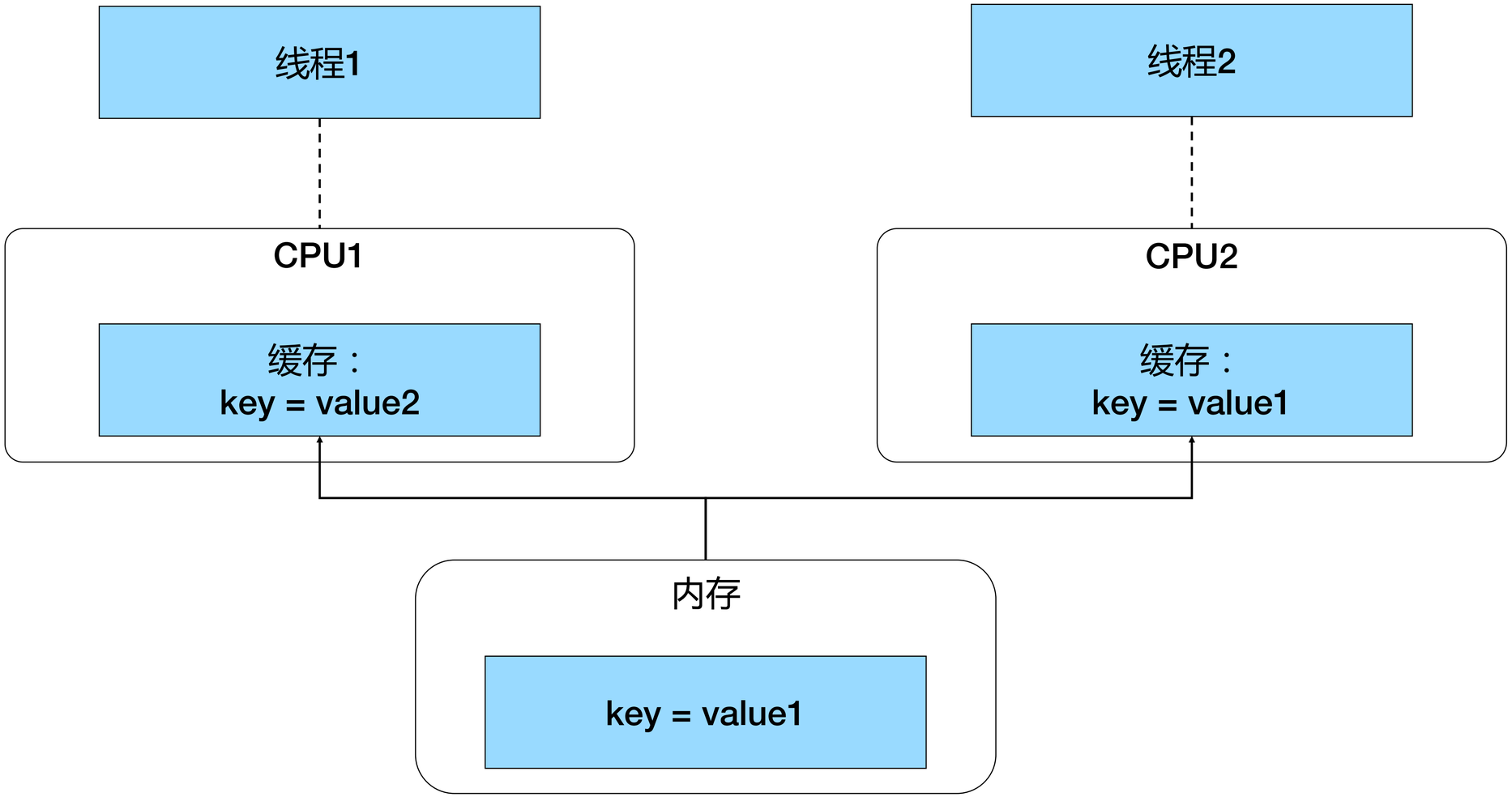
孟凡杰

eBay 资深架构师

目录

1. 线程加锁

理解线程安全



锁

极客时间 训练营

- Go 语言不仅仅提供基于 CSP 的通讯模型，也支持基于共享内存的多线程数据访问
- Sync 包提供了锁的基本原语
- sync.Mutex 互斥锁
    - Lock()加锁，Unlock()解锁
- sync.RWMutex 读写分离锁
    - 不限制并发读，只限制并发写和并发读写
- sync.WaitGroup
    - 等待一组 goroutine 返回
- sync.Once
    - 保证某段代码只执行一次
- sync.Cond
    - 让一组 goroutine 在满足特定条件时被唤醒

Mutex 示例

极客时间 训练营

Kubernetes 中的 informer factory

```
// Start initializes all requested informers.
func (f *sharedInformerFactory) Start(stopCh <-chan struct{}) {
    f.lock.Lock()
    defer f.lock.Unlock()
    for informerType, informer := range f.informers {
        if !f.startedInformers[informerType] {
            go informer.Run(stopCh)
            f.startedInformers[informerType] = true
        }
    }
}
```

WaitGroup 示例

极客时间 训练营

```
// CreateBatch create a batch of pods. All pods are created before
waiting.
func (c *PodClient) CreateBatch(pods []*v1.Pod) []*v1.Pod {
    ps := make([]*v1.Pod, len(pods))
    var wg sync.WaitGroup
    for i, pod := range pods {
        wg.Add(1)
        go func(i int, pod *v1.Pod) {
            defer wg.Done()
            defer GinkgoRecover()
            ps[i] = c.CreateSync(pod)
        }(i, pod)
    }
    wg.Wait()
    return ps
}
```

Cond 示例

Kubernetes 中的队列，标准的生产者消费者模式

cond:　　sync.NewCond(&sync.Mutex{}),

```
// Add marks item as needing processing.
func (q *Type) Add(item interface{}) {
	q.cond.L.Lock()
	defer q.cond.L.Unlock()
	if q.shuttingDown {
		return
	}
	if q.dirty.has(item) {
		return
	}
	q.metrics.add(item)
	q.dirty.insert(item)
	if q.processing.has(item) {
		return
	}
	q.queue = append(q.queue, item)
	q.cond.Signal()
}
```

Cond 示例

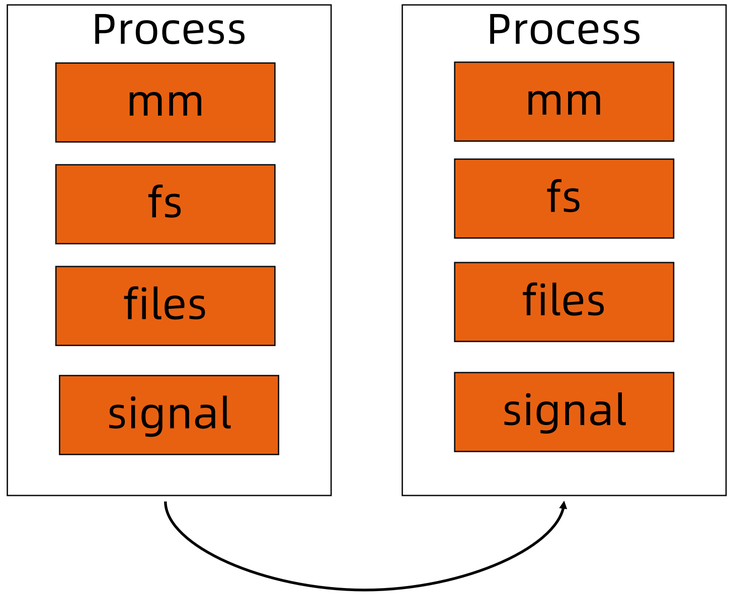
极客时间 训练营

```
// Get blocks until it can return an item to be processed. If shutdown = true,
// the caller should end their goroutine. You must call Done with item when you
// have finished processing it.
func (q *Type) Get() (item interface{}, shutdown bool) {
	q.cond.L.Lock()
	defer q.cond.L.Unlock()
	for len(q.queue) == 0 && !q.shuttingDown {
		q.cond.Wait()
	}
	if len(q.queue) == 0 {
	// We must be shutting down.
	return nil, true

	}

	item, q.queue = q.queue[0], q.queue[1:]

	q.metrics.get(item)

	q.processing.insert(item)

	q.dirty.delete(item)

	return item, false

}
```

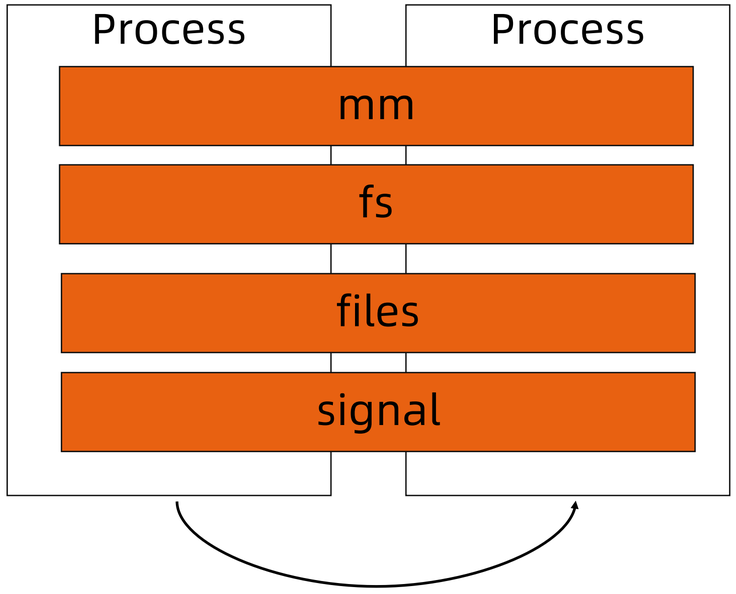
2. 线程调度

深入理解 Go 语言线程调度

- 进程：资源分配的基本单位
- 线程：调度的基本单位
- 无论是线程还是进程，在 linux 中都以 task_struct 描述，从内核角度看，与进程无本质区别
- Glibc 中的 pthread 库提供 NPTL（Native POSIX Threading Library）支持

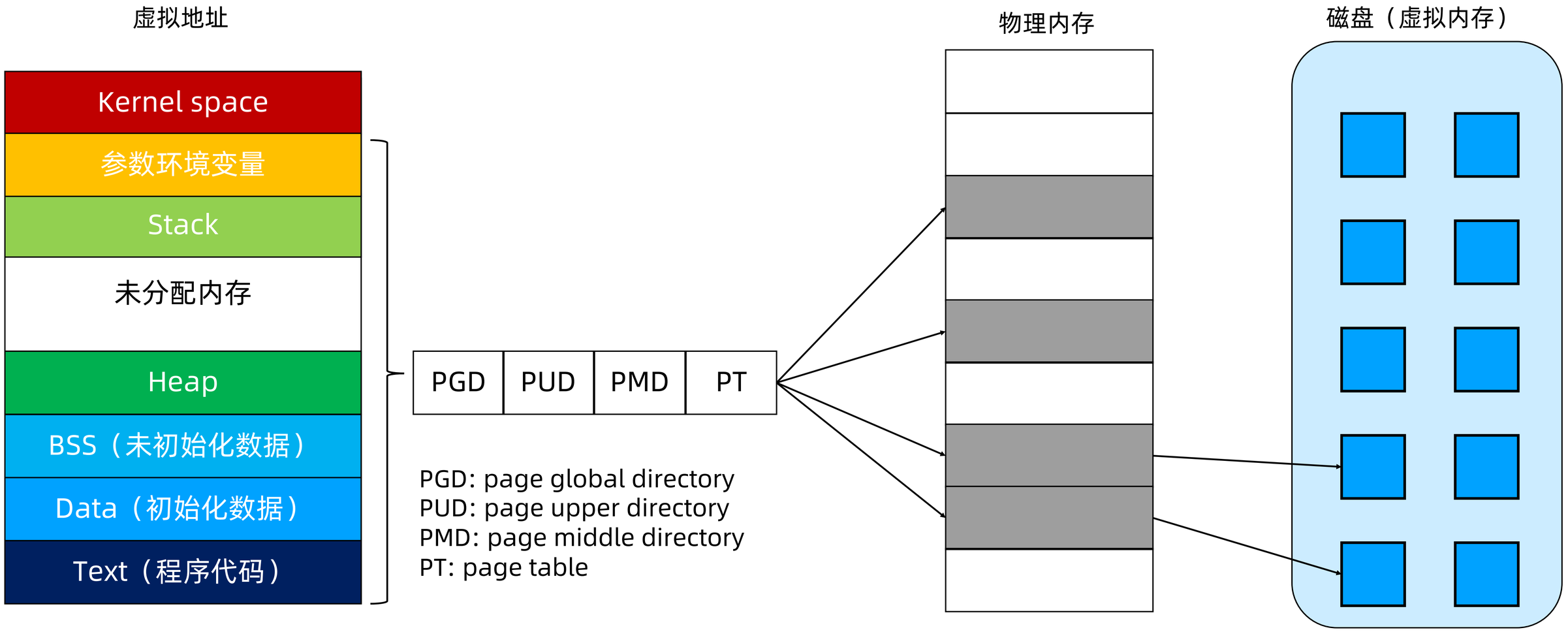


创建进程：fork



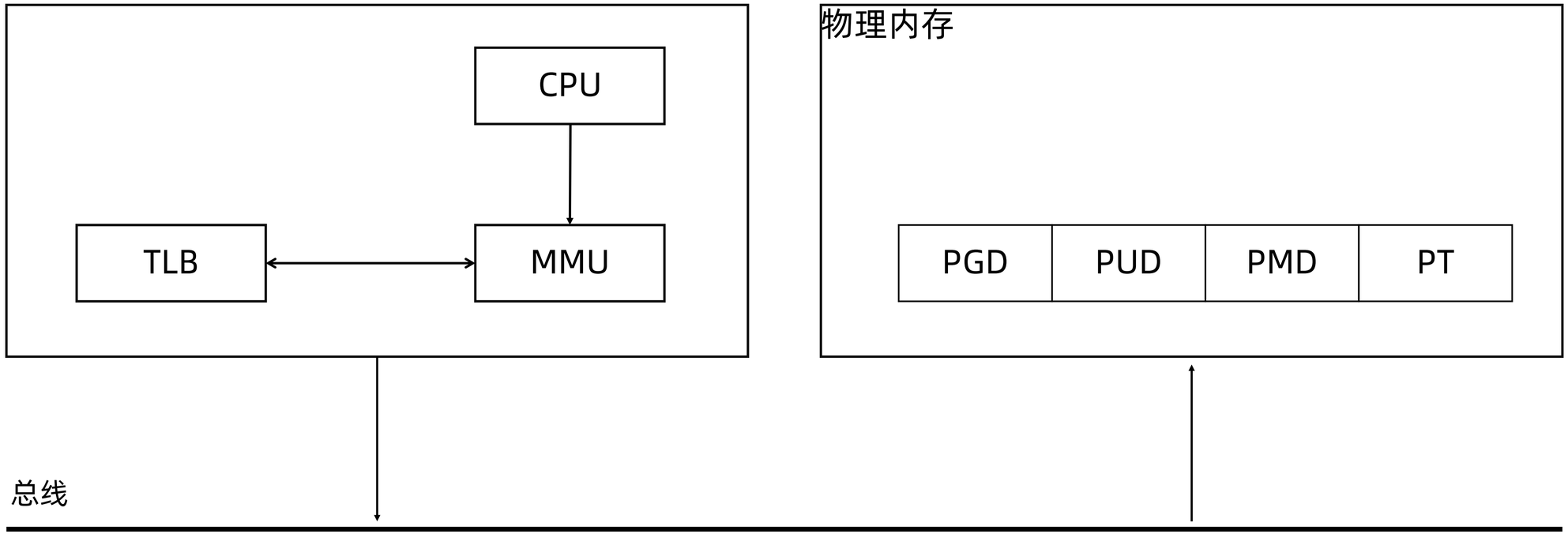
创建线程：pthread_create

Linux 进程的内存使用



CPU 对内存的访问

- CPU 上有个 Memory Management Unit（MMU） 单元
- CPU 把虚拟地址给 MMU，MMU 去物理内存中查询页表，得到实际的物理地址
- CPU 维护一份缓存 Translation Lookaside Buffer（TLB），缓存虚拟地址和物理地址的映射关系



进程切换开销

极客时间 训练营

- 直接开销
  - 切换页表全局目录（PGD）
  - 切换内核态堆栈
  - 切换硬件上下文（进程恢复前，必须装入寄存器的数据统称为硬件上下文）
  - 刷新 TLB
  - 系统调度器的代码执行
- 间接开销
  - CPU 缓存失效导致的进程需要到内存直接访问的 IO 操作变多

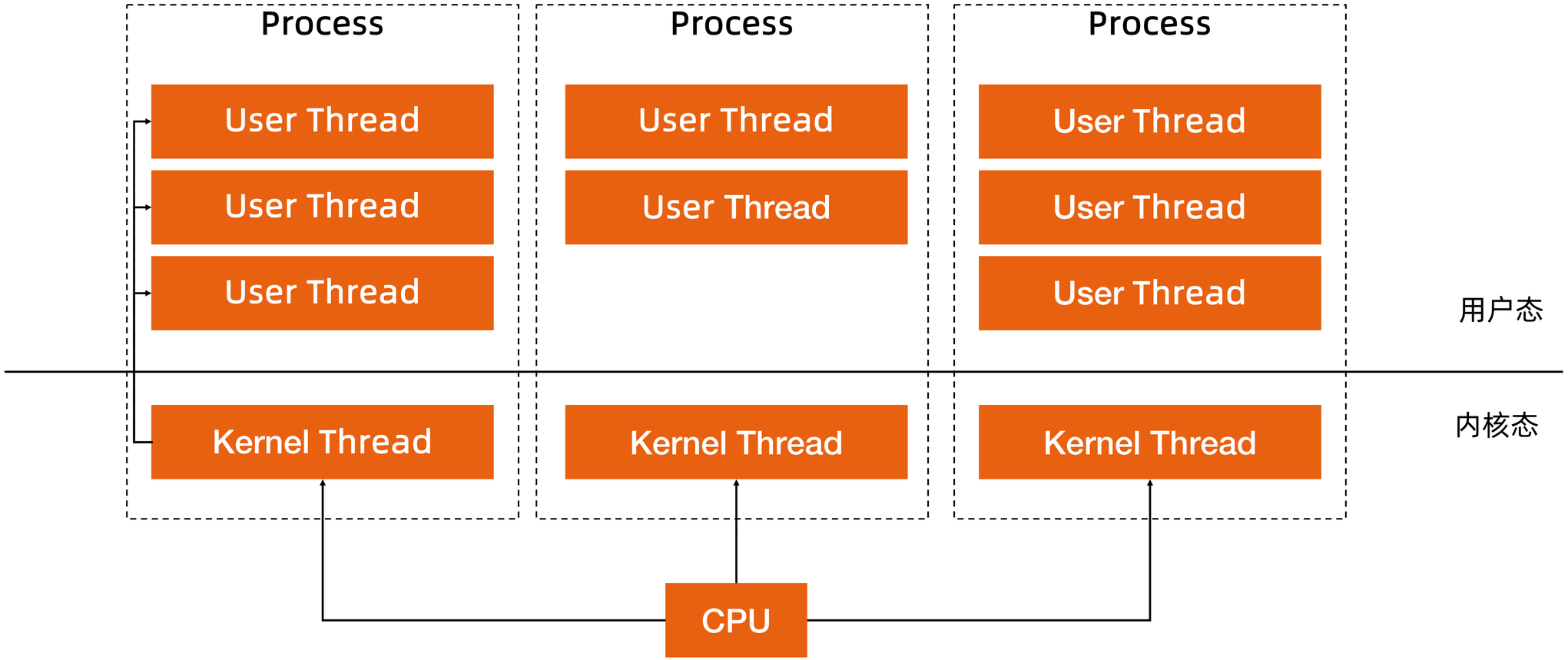
线程切换开销

极客时间训练营

- 线程本质上只是一批共享资源的进程，线程切换本质上依然需要内核进行进程切换
- 一组线程因为共享内存资源，因此一个进程的所有线程共享虚拟地址空间，线程切换相比进程切换，主要节省了虚拟地址空间的切换

用户线程

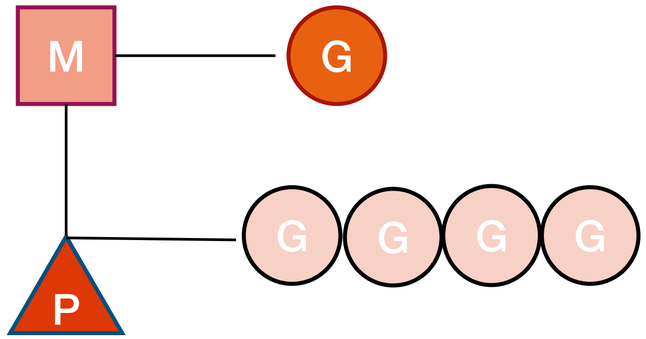
无需内核帮助，应用程序在用户空间创建的可执行单元，创建销毁完全在用户态完成。



Goroutine

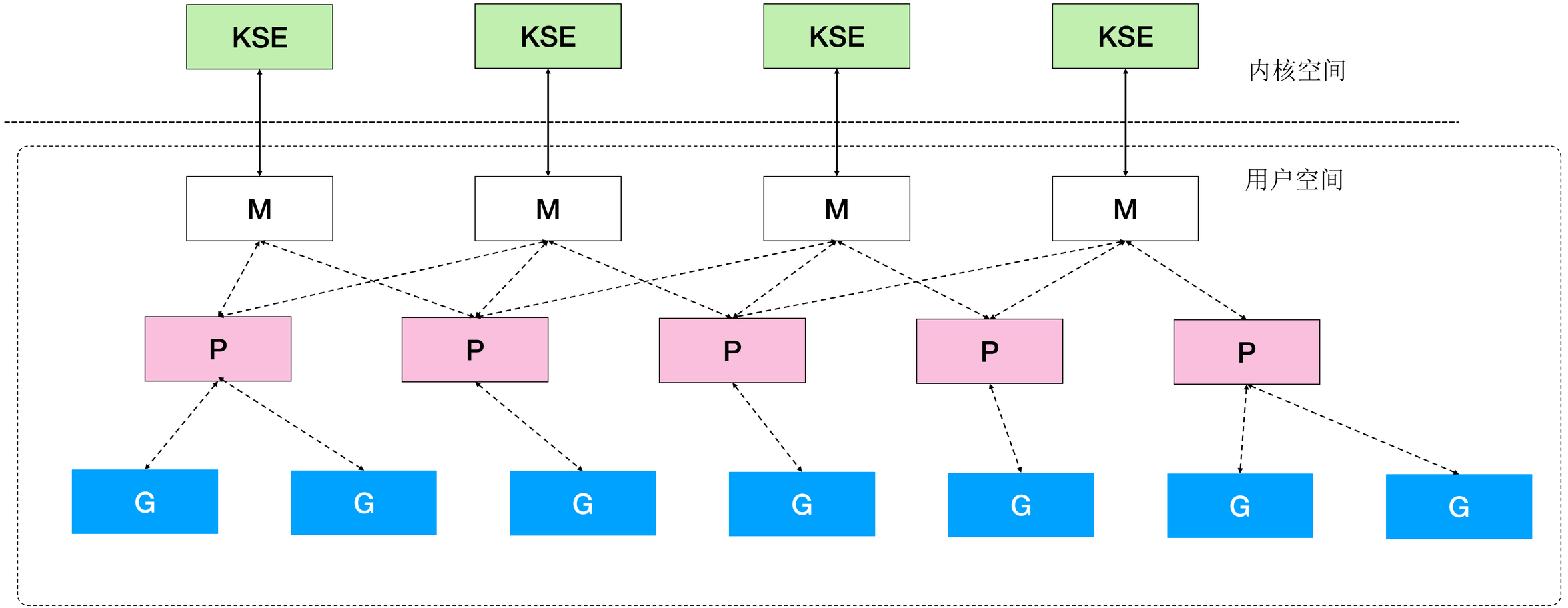
Go 语言基于 GMP 模型实现用户态线程

- Goroutine：表示 goroutine，每个 goroutine 都有自己的栈空间，定时器，初始化的栈空间在 2k 左右，空间会随着需求增长。
- Machine：抽象化代表内核线程，记录内核线程栈信息，当 goroutine 调度到线程时，使用该 goroutine 自己的栈信息。
- Process：代表调度器，负责调度 goroutine，维护一个本地 goroutine 队列，M 从 P 上获得 goroutine 并执行，同时还负责部分内存的管理。

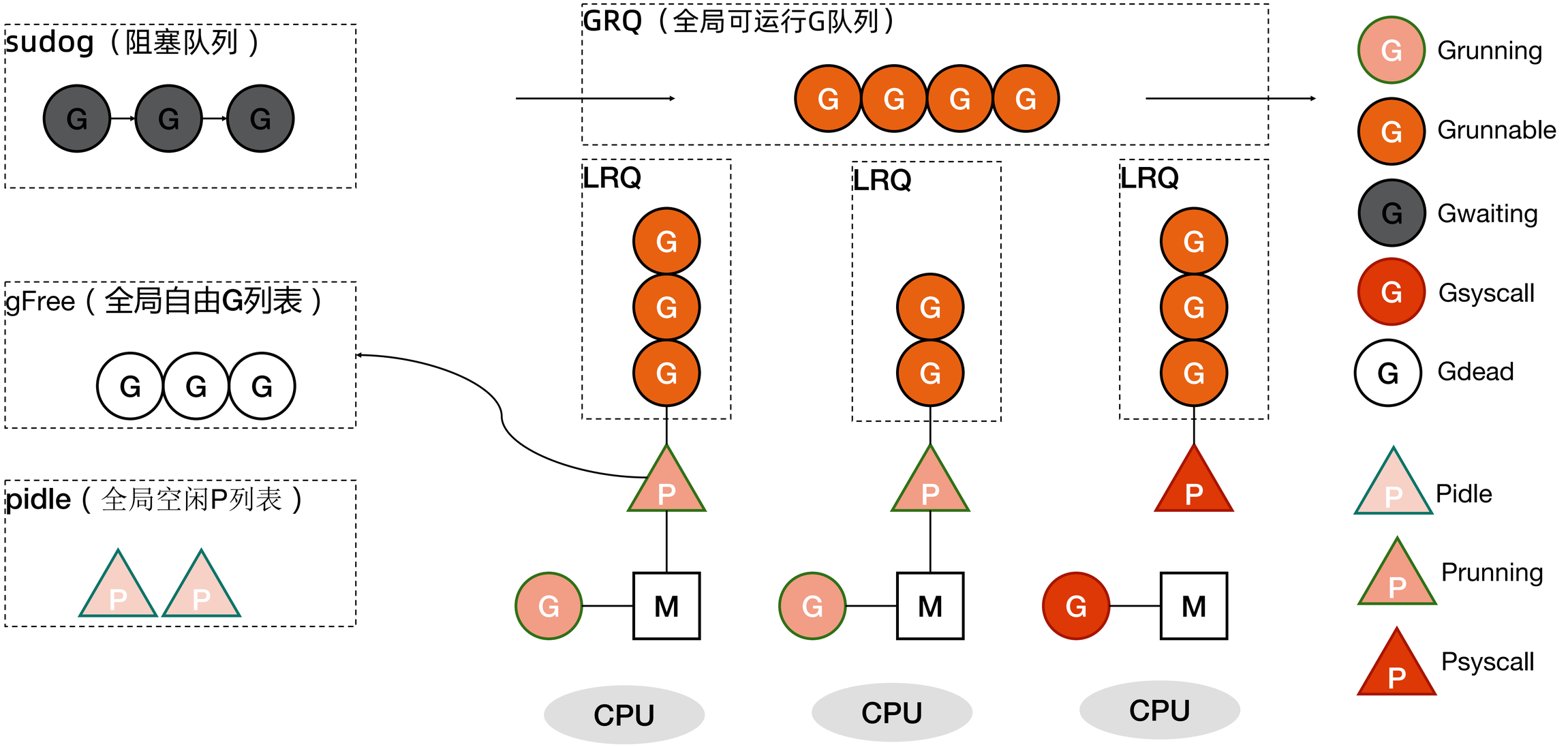


MPG的对应关系

KSE: Kernel Scheduling Entity



GMP 模型细节


sudog（阻塞队列）
G
G
G
GRQ（全局可运行G队列）
G
G
G
G
gFree（全局自由G列表）
G
G
G
pidle（全局空闲P列表）
P
P
LRQ
G
G
G
P
G
M
CPU
LRQ
G
G
P
G
M
CPU
LRQ
G
G
G
P
G
M
CPU
G
Grunning
G
Grunnable
G
Gwaiting
G
Gsyscall
G
Gdead
P
Pidle
P
Prunning
P
Psyscall

P 的状态

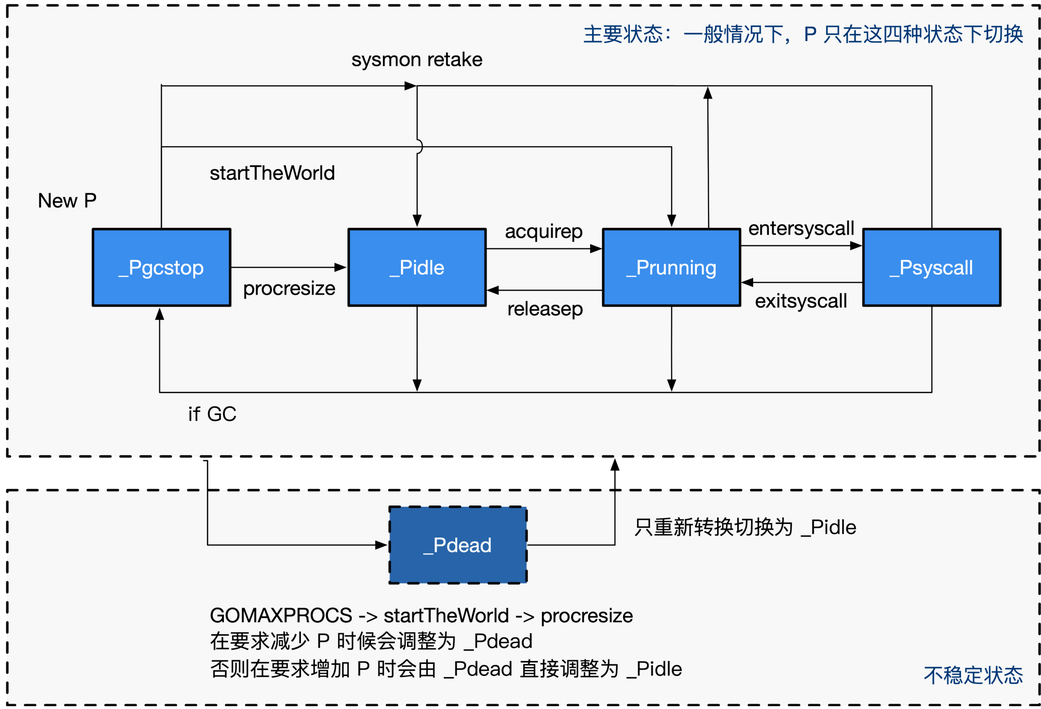
_Pidle：处理器没有运行用户代码或者调度器，被空闲队列或者改变其状态的结构持有，运行队列为空

_Prunning：被线程 M 持有，并且正在执行用户代码或者调度器

_Psyscall：没有执行用户代码，当前线程陷入系统调用

_Pgcstop：被线程 M 持有，当前处理器由于垃圾回收被停止

_Pdead：当前处理器已经不被使用



G 的状态

极客时间 训练营

_Gidle：刚刚被分配并且还没有被初始化，值为0，为创建 goroutine 后的默认值

_Grunnable： 没有执行代码，没有栈的所有权，存储在运行队列中，可能在某个P的本地队列或全局队列中(如上图)。

_Grunning： 正在执行代码的 goroutine，拥有栈的所有权

_Gsyscall：正在执行系统调用，拥有栈的所有权，与 P 脱离，但是与某个 M 绑定，会在调用结束后被分配到运行队列

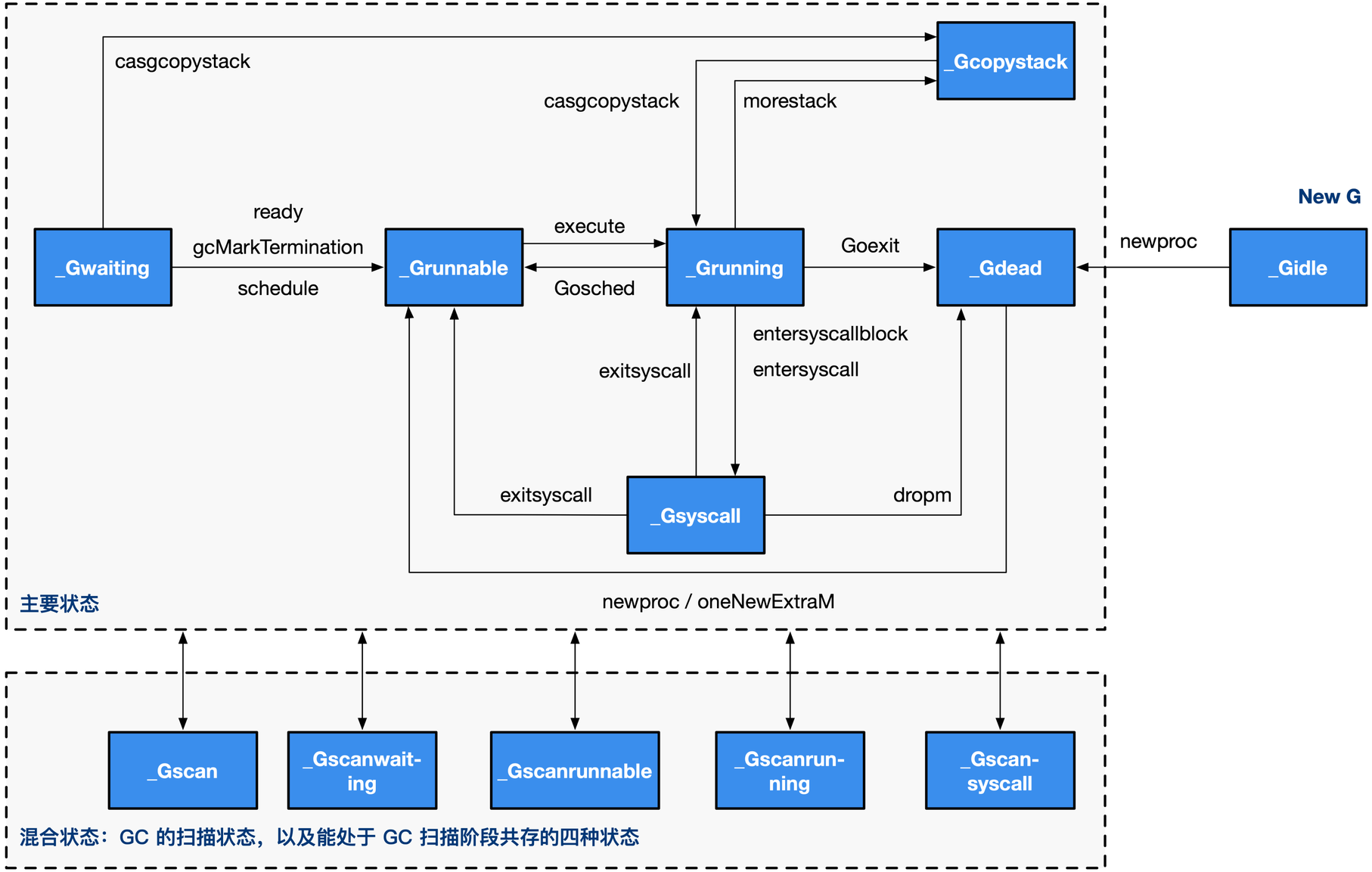
_Gwaiting：被阻塞的 goroutine，阻塞在某个 channel 的发送或者接收队列

_Gdead： 当前 goroutine 未被使用，没有执行代码，可能有分配的栈，分布在空闲列表 gFree，可能是一个刚刚初始化的 goroutine，也可能是执行了 goexit 退出的 goroutine

_Gcopystac：栈正在被拷贝，没有执行代码，不在运行队列上，执行权在

_Gscan： GC 正在扫描栈空间，没有执行代码，可以与其他状态同时存在

G的状态转换图


casgcopystack
_Gcopystack
casgcopystack
morestack
New G
ready
gcMarkTermination
schedule
execute
Gosched
Goexit
newproc
_Gwaiting
_Grunnable
_Grunning
_Gdead
_Gidle
entersyscallblock
entersyscall
exitsyscall
exitsyscall
dropm
_Gsyscall
主要状态
newproc / oneNewExtraM
_Gscan
_Gscanwait-ing
_Gscanrunnable
_Gscanrun-ning
_Gscan-syscall
混合状态：GC 的扫描状态，以及能处于 GC 扫描阶段共存的四种状态

G 所处的位置

- 进程都有一个全局的 G 队列
- 每个 P 拥有自己的本地执行队列
- 有不在运行队列中的 G
    - 处于 channel 阻塞态的 G 被放在 sudog
    - 脱离 P 绑定在 M 上的 G，如系统调用
    - 为了复用，执行结束进入 P 的 gFree 列表中的 G

Goroutine 创建过程

- 获取或者创建新的 Goroutine 结构体
    - 从处理器的 gFree 列表中查找空闲的 Goroutine
    - 如果不存在空闲的 Goroutine，会通过 runtime.malg 创建一个栈大小足够的新结构体
- 将函数传入的参数移到 Goroutine 的栈上
- 更新 Goroutine 调度相关的属性，更新状态为_Grunnable
- 返回的 Goroutine 会存储到全局变量 allgs 中

将 Goroutine 放到运行队列上

- Goroutine 设置到处理器的 runnext 作为下一个处理器执行的任务
- 当处理器的本地运行队列已经没有剩余空间时(256)，就会把本地队列中的一部分 Goroutine 和待加入的 Goroutine 通过 runtime.runqputslow 添加到调度器持有的全局运行队列上

调度器行为

极客时间 训练营

- 为了保证公平，当全局运行队列中有待执行的 Goroutine 时，通过 schedtick 保证有一定几率(1/61)会从全局的运行队列中查找对应的 Goroutine
- 从处理器本地的运行队列中查找待执行的 Goroutine
- 如果前两种方法都没有找到 Goroutine，会通过 runtime.findrunnable 进行阻塞地查找 Goroutine
    - 从本地运行队列、全局运行队列中查找
    - 从网络轮询器中查找是否有 Goroutine 等待运行
    - 通过 runtime.runqsteal 尝试从其他随机的处理器中窃取一半待运行的 Goroutine

课后练习 2.1

- 将练习 1.2 中的生产者消费者模型修改成为多个生产者和多个消费者模式

3. 内存管理

关于内存管理的争论

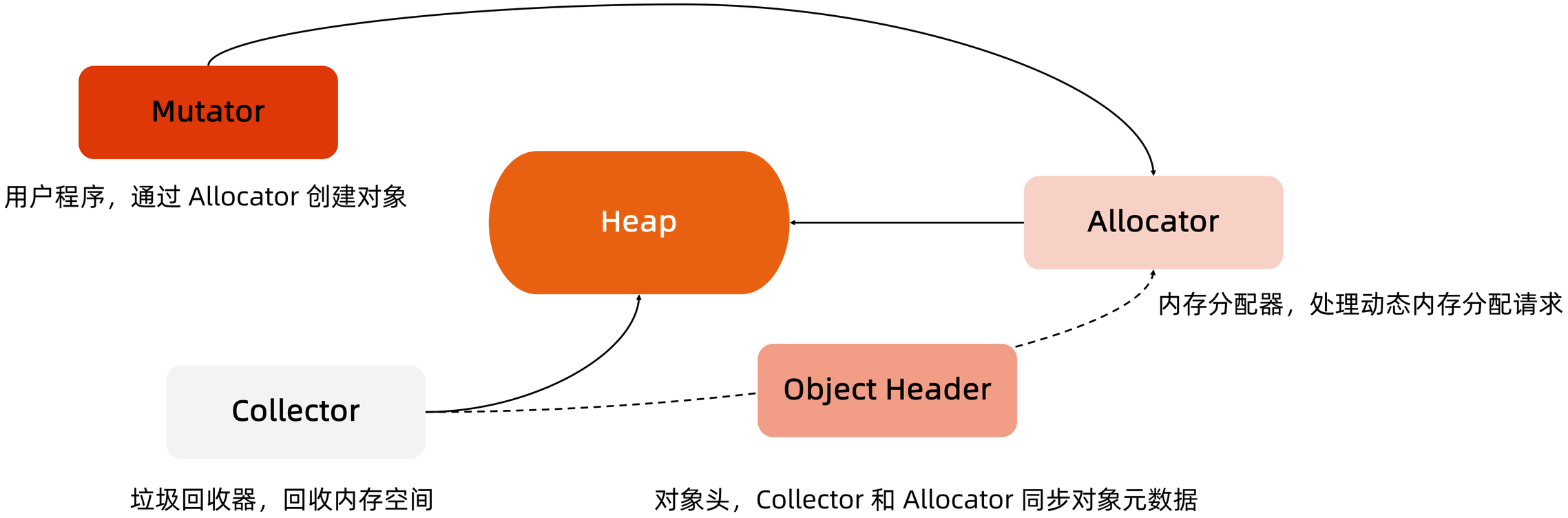
内存管理太重要了！手动管理麻烦且容易出错，所以我们应该交给机器去管理！

Java/golang

内存管理太重要了！所以如果交给机器管理我不能放心！

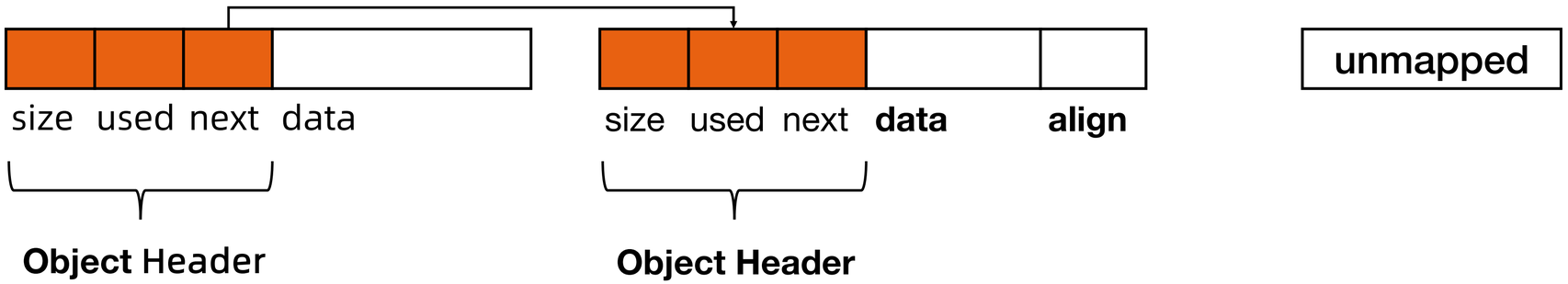
c/c++

堆内存管理

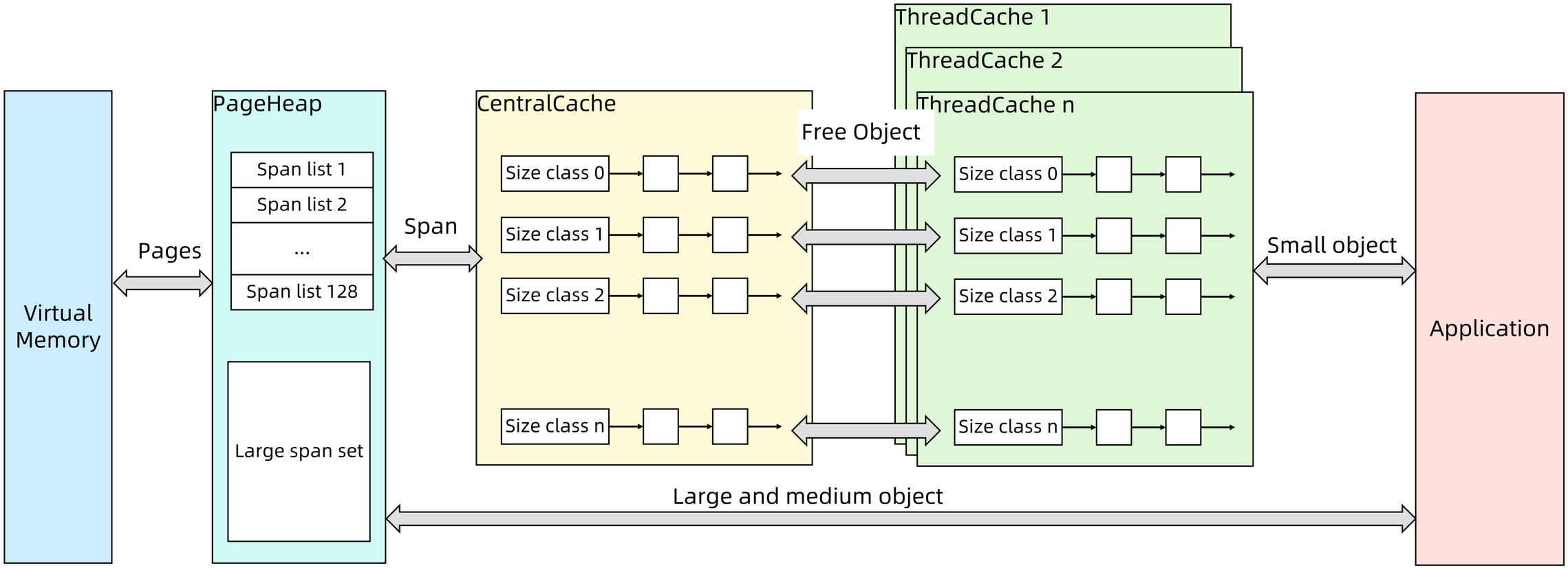

Mutator
用户程序，通过 Allocator 创建对象
Heap
Allocator
内存分配器，处理动态内存分配请求
Collector
垃圾回收器，回收内存空间
Object Header
对象头，Collector 和 Allocator 同步对象元数据

堆内存管理

- 初始化连续内存块作为堆
- 有内存申请的时候，Allocator 从堆内存的未分配区域分割小内存块
- 用链表将已分配内存连接起来
- 需要信息描述每个内存块的元数据：大小，是否使用，下一个内存块的地址等



TCMalloc 概览



Span list 1: 1 page
Span list 2: 2 pages
Span list 128: 128 pages = 1M

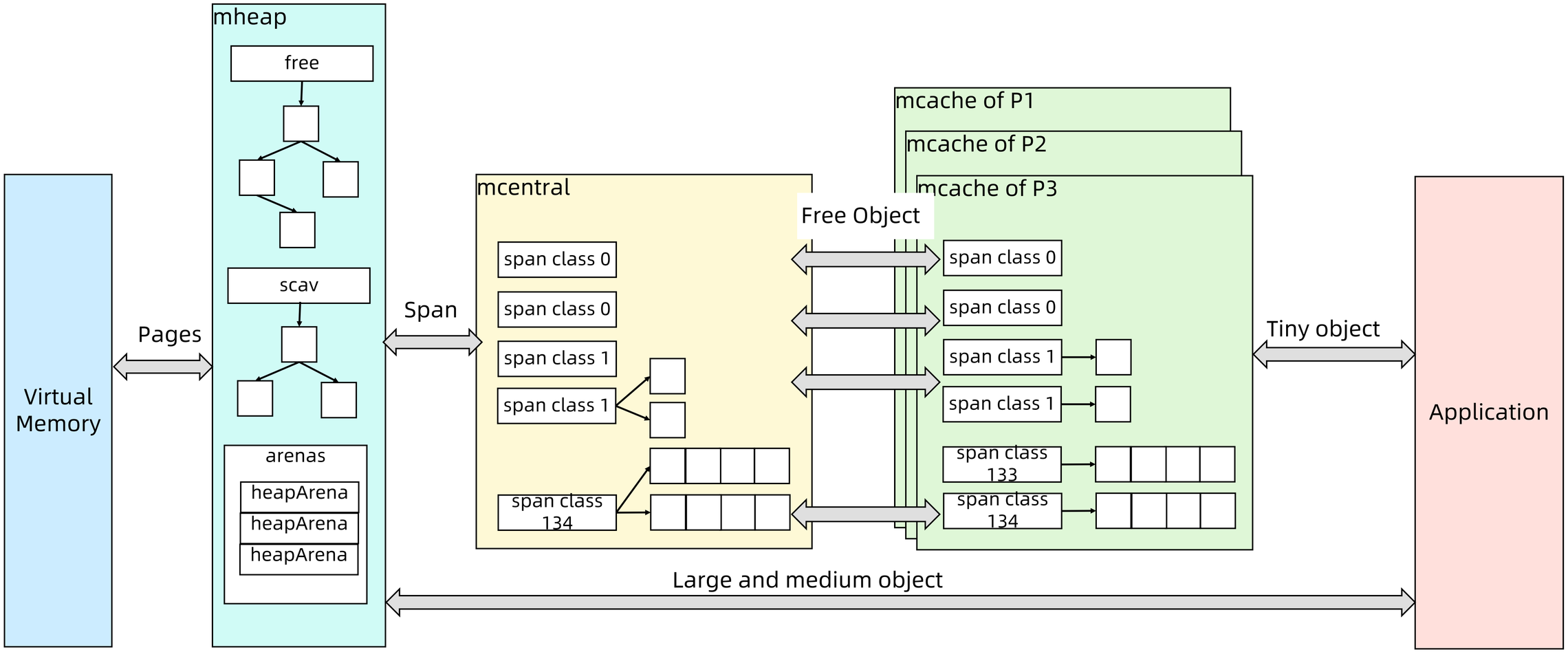
TCMalloc

- page:内存页，一块 8K 大小的内存空间。Go 与操作系统之间的内存申请和释放，都是以 page 为单位的
- span: 内存块，一个或多个连续的 page 组成一个 span
- sizeclass：空间规格，每个 span 都带有一个 sizeclass，标记着该 span 中的 page 应该如何使用
- object：对象，用来存储一个变量数据内存空间，一个 span 在初始化时，会被切割成一堆等大的 object；假设 object 的大小是 16B，span 大小是 8K，那么就会把 span 中的 page 就会被初始化 8K / 16B = 512 个 object，所谓内存分配，就是分配一个 object 出去

TCMalloc

- 对象大小定义
    - 小对象大小：0~256KB
    - 中对象大小：256KB~1MB
    - 大对象大小：>1MB
- 小对象的分配流程
    - ThreadCache -> CentralCache -> HeapPage，大部分时候，ThreadCache 缓存都是足够的，不需要去访问 CentralCache 和 HeapPage，无系统调用配合无锁分配，分配效率是非常高的
- 中对象分配流程
    - 直接在 PageHeap 中选择适当的大小即可，128 Page 的 Span 所保存的最大内存就是 1MB
- 大对象分配流程
    - 从 large span set 选择合适数量的页面组成 span，用来存储数据

Go 语言内存分配


mheap
free
scav
arenas
heapArena
heapArena
heapArena
Virtual Memory
Pages
Span
mcentral
span class 0
span class 0
span class 1
span class 1
span class 134
mcache of P1
mcache of P2
mcache of P3
Free Object
span class 0
span class 0
span class 1
span class 1
span class 133
span class 134
Tiny object
Application
Large and medium object

Go 语言内存分配

- mcache：小对象的内存分配直接走
  - size class 从 1 到 66，每个 class 两个 span
  - Span 大小是 8KB，按 span class 大小切分
- mcentral
  - Span 内的所有内存块都被占用时，没有剩余空间继续分配对象，mcache 会向 mcentral 申请1个 span，mcache 拿到 span 后继续分配对象
  - 当 mcentral 向 mcache 提供 span 时，如果没有符合条件的 span，mcentral 会向 mheap 申请 span
- mheap
  - 当 mheap 没有足够的内存时，mheap 会向 OS 申请内存
  - Mheap 把 Span 组织成了树结构，而不是链表
  - 然后把 Span 分配到 heapArena 进行管理，它包含地址映射和 span 是否包含指针等位图
    - 为了更高效的分配、回收和再利用内存

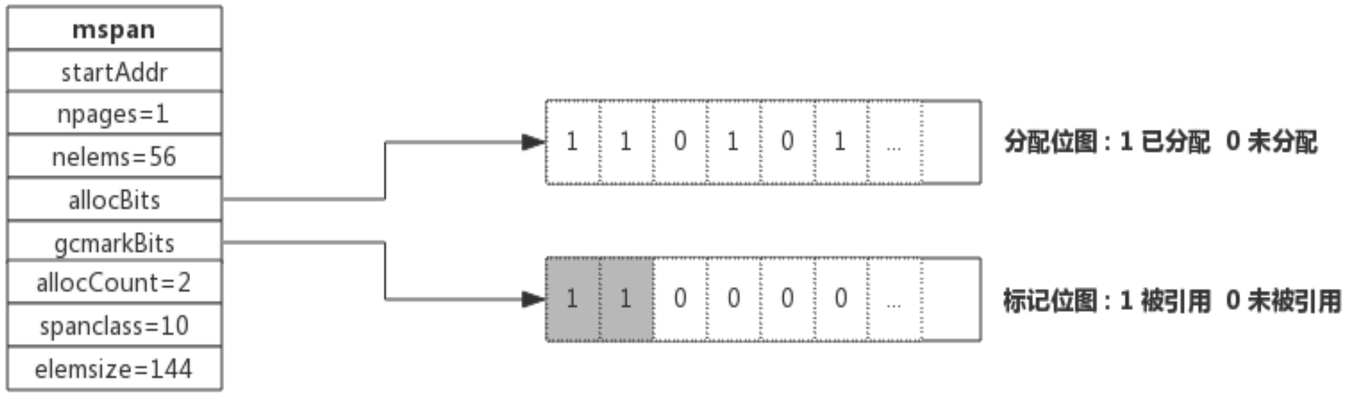
内存回收

极客时间 训练营

- 引用计数（Python，PHP，Swift）
    - 对每一个对象维护一个引用计数，当引用该对象的对象被销毁的时候，引用计数减 1，当引用计数为 0 的时候，回收该对象
    - 优点：对象可以很快的被回收，不会出现内存耗尽或达到某个阀值时才回收
    - 缺点：不能很好的处理循环引用，而且实时维护引用计数，有也一定的代价
- 标记-清除（Golang）
    - 从根变量开始遍历所有引用的对象，引用的对象标记为"被引用"，没有被标记的进行回收
    - 优点：解决引用计数的缺点
    - 缺点：需要 STW（stop the word），即要暂停程序运行
- 分代收集（Java）
    - 按照生命周期进行划分不同的代空间，生命周期长的放入老年代，短的放入新生代，新生代的回收频率高于老年代的频率

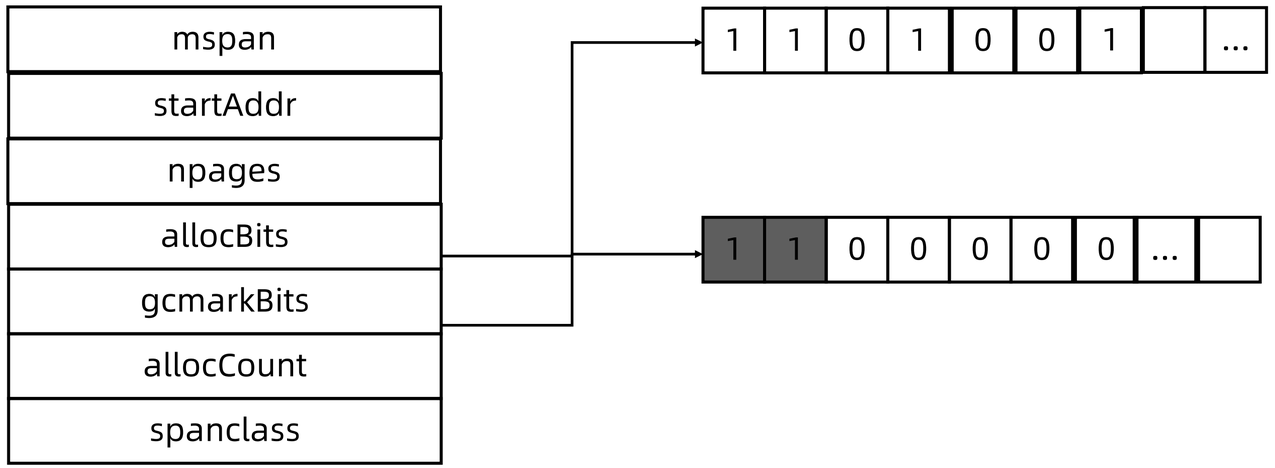
mspan

- allocBits
  - 记录了每块内存分配的情况
- gcmarkBits
  - 记录了每块内存的引用情况，标记阶段对每块内存进行标记，有对象引用的内存标记为1，没有的标记为 0



mspan

- 这两个位图的数据结构是完全一致的，标记结束则进行内存回收，回收的时候，将 allocBits 指向 gcmarkBits，标记过的则存在，未进行标记的则进行回收



GC 工作流程

Golang GC 的大部分处理是和用户代码并行的

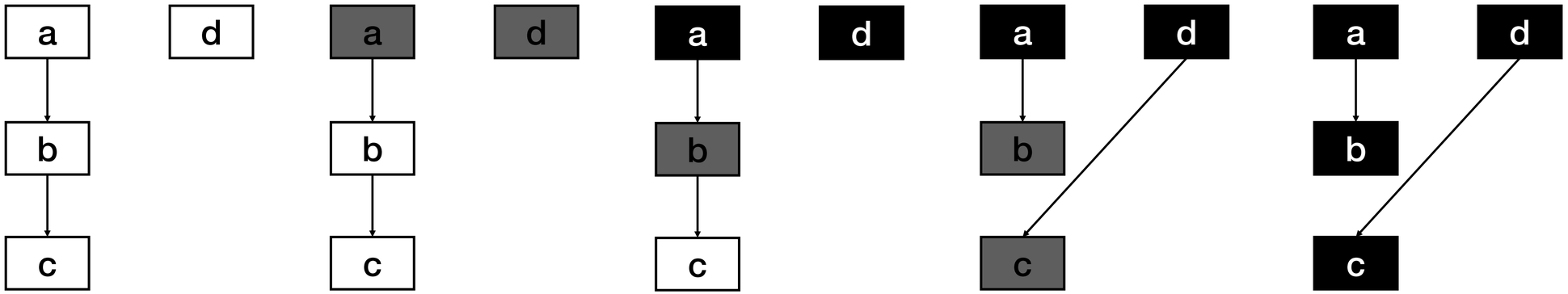
- Mark:
    - Mark Prepare: 初始化 GC 任务，包括开启写屏障 (write barrier) 和辅助 GC(mutator assist)，统计root对象的任务数量等。这个过程需要STW
    - GC Drains: 扫描所有 root 对象，包括全局指针和 goroutine(G) 栈上的指针（扫描对应 G 栈时需停止该 G)，将其加入标记队列(灰色队列)，并循环处理灰色队列的对象，直到灰色队列为空。该过程后台并行执行
- Mark Termination：完成标记工作，重新扫描(re-scan)全局指针和栈。因为 Mark 和用户程序是并行的，所以在 Mark 过程中可能会有新的对象分配和指针赋值，这个时候就需要通过写屏障（write barrier）记录下来，re-scan 再检查一下，这个过程也是会 STW 的
- Sweep：按照标记结果回收所有的白色对象，该过程后台并行执行
- Sweep Termination：对未清扫的 span 进行清扫, 只有上一轮的 GC 的清扫工作完成才可以开始新一轮的 GC

GC 工作流程

关闭			• GC关闭 • 写操作是正常的赋值
栈扫描	开启写屏障	STW	• 开启写屏障等准备工作，短暂开启STW • 从全局空间和 goroutine 栈空间上收集变量
标记			• 三色标记法，直到没有灰色对象
标记结束		STW	• 开启 STW，回头重新扫描 root 区域新变量，对他们进行标记
清除			• 关闭 STW 和 写屏障，对白色对象进行清除
关闭			• 循环结束，重启下一阶段GC

三色标记

- GC 开始时，认为所有 object 都是 白色，即垃圾
- 从 root 区开始遍历，被触达的 object 置成 灰色
- 遍历所有灰色 object，将他们内部的引用变量置成 灰色，自身置成 黑色
- 循环第 3 步，直到没有灰色 object 了，只剩下了黑白两种，白色的都是垃圾
- 对于黑色 object，如果在标记期间发生了写操作，写屏障会在真正赋值前将新对象标记为 灰色
- 标记过程中，mallocgc 新分配的 object，会先被标记成 黑色 再返回



垃圾回收触发机制

极客时间 训练营

- 内存分配量达到阀值触发 GC
    - 每次内存分配时都会检查当前内存分配量是否已达到阀值，如果达到阀值则立即启动 GC。
        - 阀值 = 上次 GC 内存分配量 * 内存增长率
        - 内存增长率由环境变量 GOGC 控制，默认为 100，即每当内存扩大一倍时启动 GC。
- 定期触发 GC
    - 默认情况下，最长 2 分钟触发一次 GC，这个间隔在 src/runtime/proc.go:forcegcperiod 变量中被声明
- 手动触发
    - 程序代码中也可以使用 runtime.GC()来手动触发 GC。这主要用于 GC 性能测试和统计。

4. 包引用与依赖管理

Go 语言依赖管理的演进

极客时间 训练营

- 回顾 GOPATH
  - 通过环境变量设置系统级的 Go 语言类库目录
  - GOPATH 的问题？
    - 不同项目可能依赖不同版本
    - 代码被 clone 以后需要设置 GOPATH 才能编译
- vendor
  - 自 1.6 版本，支持 vendor 目录，在每个 Go 语言项目中，创建一个名叫 vendor 的目录，并将依赖拷贝至该目录。
  - Go 语言项目会自动将 vendor 目录作为自身的项目依赖路径
  - 好处？
    - 每个项目的 vendor目录是独立的，可以灵活的选择版本
    - Vendor 目录与源代码一起 check in 到 github，其他人 checkout 以后可直接编译
    - 无需在编译期间下载依赖包，所有依赖都已经与源代码保存在一起

vendor 管理工具

通过声明式配置，实现 vendor 管理的自动化

- 在早期，Go 语言无自带依赖管理工具，社区方案鱼龙混杂比较出名的包括
    - Godeps, Glide
- Go 语言随后发布了自带的依赖管理工具 Gopkg
- 很快用新的工具 gomod 替换掉了 gopkg
    - 切换 mod 开启模式：export GO111MODULE=on/off/auto
    - Go mod 相比之前的工具更灵活易用，以基本统一了 Go 语言依赖管理

思考：用依赖管理工具的目的？

- 版本管理
- 防篡改

Go mod 使用

- 创建项目
- 初始化 Go 模块
    - go mod init
- 下载依赖包
    - go mod download（下载的依赖包在$GOPATH/pkg，如果没有设置 GOPATH，则下载在项目根目录/pkg）
    - 在源代码中使用某个依赖包，如 github.com/emicklei/go-restful
- 添加缺少的依赖并为依赖包瘦身
    - go mod tidy
- 把 Go 依赖模块添加到 vendor 目录
    - go mod vendor

配置细节会被保存在项目根目录的 go.mod 中

可在 require 或者 replacement 中指定版本

go.mod sample

```
module k8s.io/apiserver
go 1.13
require (
  github.com/evanphx/json-patch v4.9.0+incompatible
  github.com/go-openapi/jsonreference v0.19.3 // indirect
  github.com/go-openapi/spec v0.19.3
  github.com/gogo/protobuf v1.3.2
  golang.org/x/crypto master
  github.com/google/gofuzz v1.1.0
  k8s.io/apimachinery v0.0.0-20210518100737-44f1264f7b6b
)

replace (
  golang.org/x/crypto => golang.org/x/crypto v0.0.0-20200220183623-bac4c82f6975
  golang.org/x/image => github.com/golang/image
  k8s.io/api => k8s.io/api v0.0.0-20210518101910-53468e23a787
  k8s.io/apimachinery => k8s.io/apimachinery v0.0.0-20210518100737-44f1264f7b6b
  k8s.io/client-go => k8s.io/client-go v0.0.0-20210518104342-fa3acefe68f3
  k8s.io/component-base => k8s.io/component-base v0.0.0-20210518111421-67c12a31a26a
)
```

GOPROXY 和 GOPRIVATE

- GOPROXY
  - 为拉取 Go 依赖设置代理
    - export GOPROXY=https://goproxy.cn
- 在设置 GOPROXY 以后，默认所有依赖拉取都需要经过 proxy 连接 git repo，拉取代码，并做 checksum 校验
- 某些私有代码仓库是 goproxy.cn 无法连接的，因此需要设置 GOPRIVATE 来声明私有代码仓库

GOPRIVATE=*.corp.example.com

GOPROXY=proxy.example.com

GONOPROXY=myrepo.corp.example.com

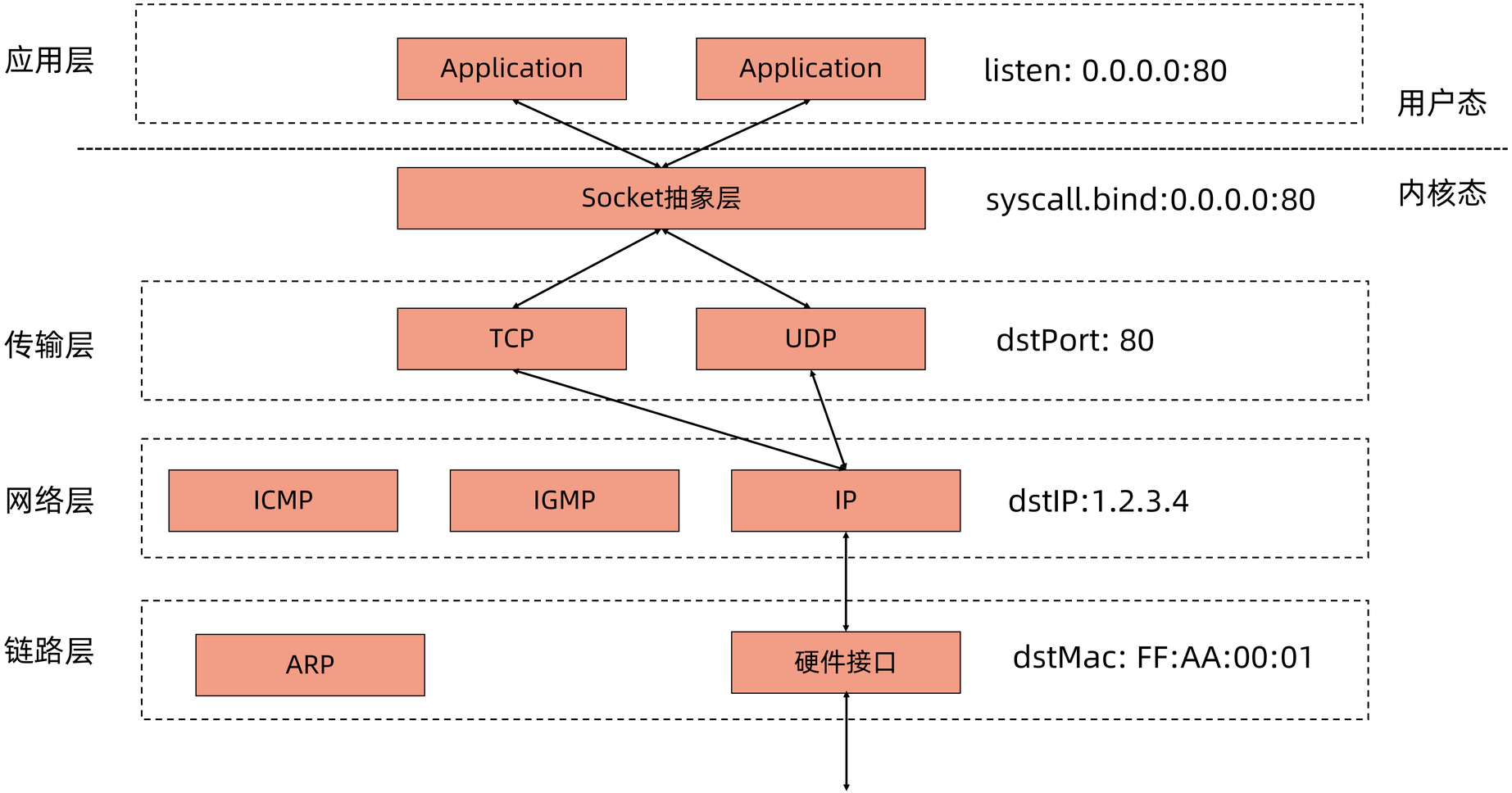
5. Makefile

Go 语言项目多采用 Makefile 组织项目编译


```
root:
	export ROOT=github.com/cncamp/golang;
.PHONY: root

release:
	echo "building httpserver binary"
	mkdir -p bin/amd64
	CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -o bin/amd64 .
.PHONY: release
```

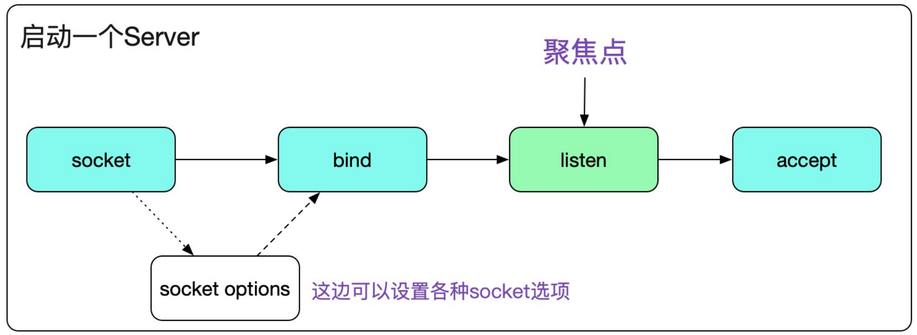
6. 动手编写一个 HTTP Server

理解网络协议层


应用层
Application
Application
listen: 0.0.0.0:80
用户态
Socket抽象层
syscall.bind:0.0.0.0:80
内核态
传输层
TCP
UDP
dstPort: 80
网络层
ICMP
IGMP
IP
dstIP:1.2.3.4
链路层
ARP
硬件接口
dstMac: FF:AA:00:01

理解 Socket

- socket 被翻译为“套接字”，它是计算机之间进行通信的一种约定或一种方式
- Linux 中的一切都是文件
- 为了表示和区分已经打开的文件，UNIX/Linux 会给每个文件分配一个 ID，这个 ID 就是一个整数，被称为文件描述符
- 网络连接也是一个文件，它也有文件描述符
- 服务器端先初始化 Socket，然后与端口绑定（bind），对端口进行监听（listen），调用 accept 阻塞，等待客户端连接
- 在这时如果有个客户端初始化一个 Socket，然后连接服务器（connect），如果连接成功，这时客户端与服务器端的连接就建立了
- 服务端的 Accept 接收到请求以后，会生成连接 FD，借助这个 FD 我们就可以使用普通的文件操作函数来传输数据了，例如：
    - 用 read() 读取从远程计算机传来的数据
    - 用 write() 向远程计算机写入数据



理解 net.http 包

- 注册 handle 处理函数

```
http.HandleFunc("/healthz", healthz)
//Use the default DefaultServeMux.
```

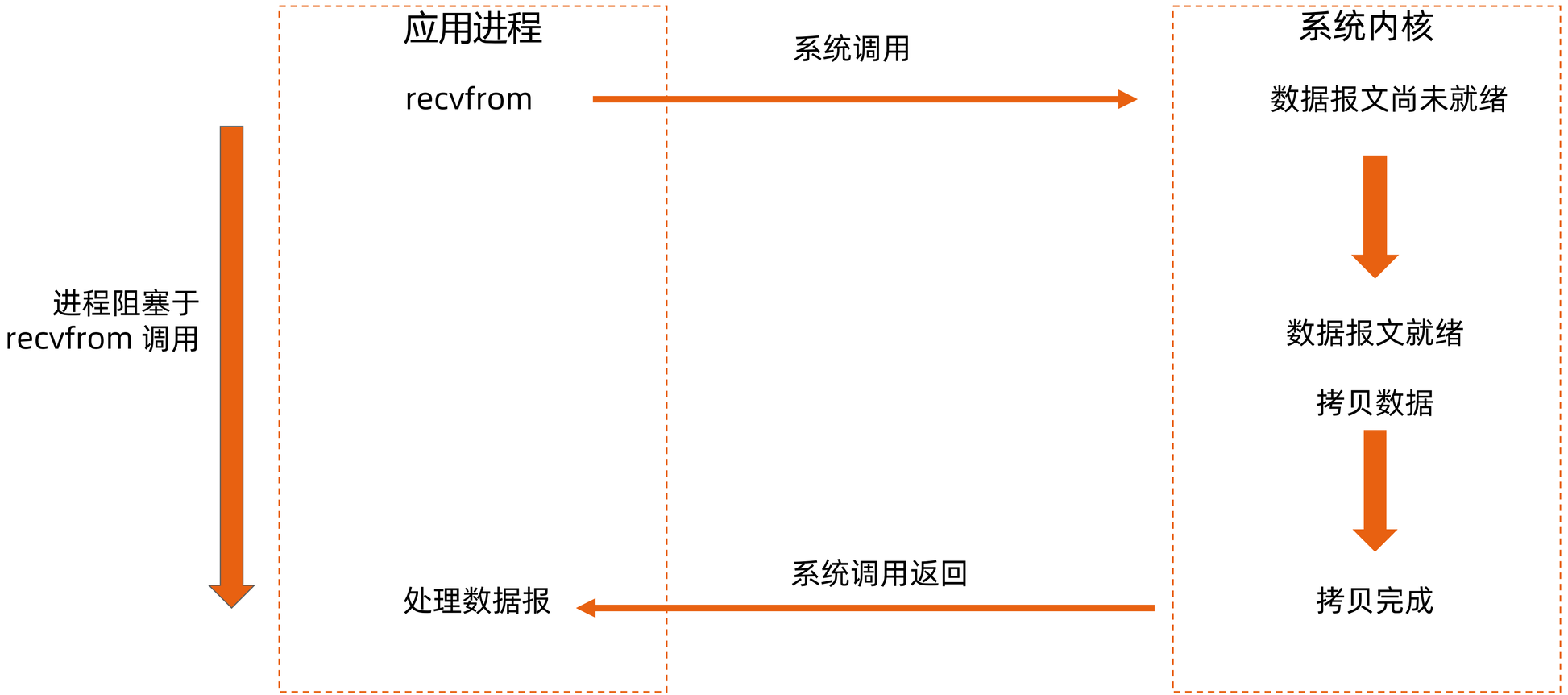
- ListenAndService

```
err := http.ListenAndServe(":80", nil)
if err != nil {
log.Fatal(err)
}
```

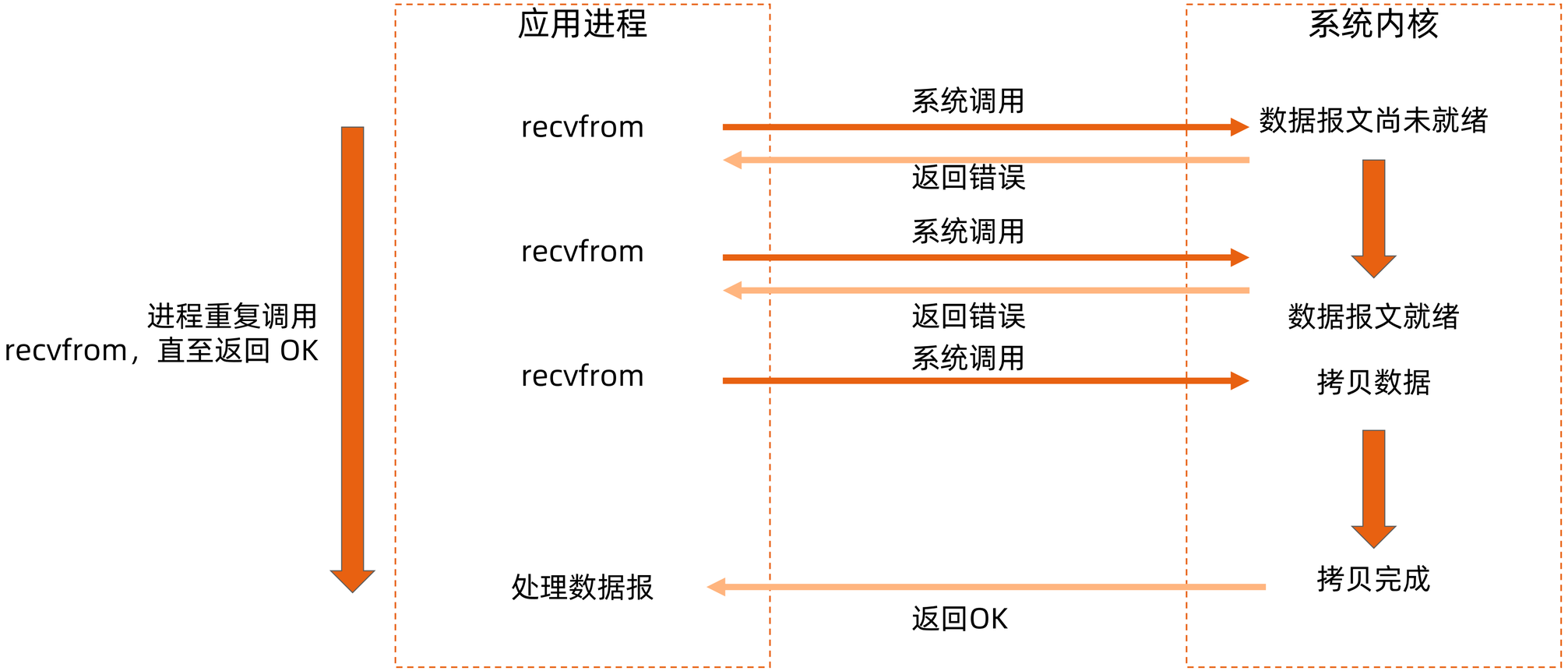
- 定义 handle 处理函数

```
func healthz(w http.ResponseWriter, r
*http.Request) {
io.WriteString(w, "ok")
}
```

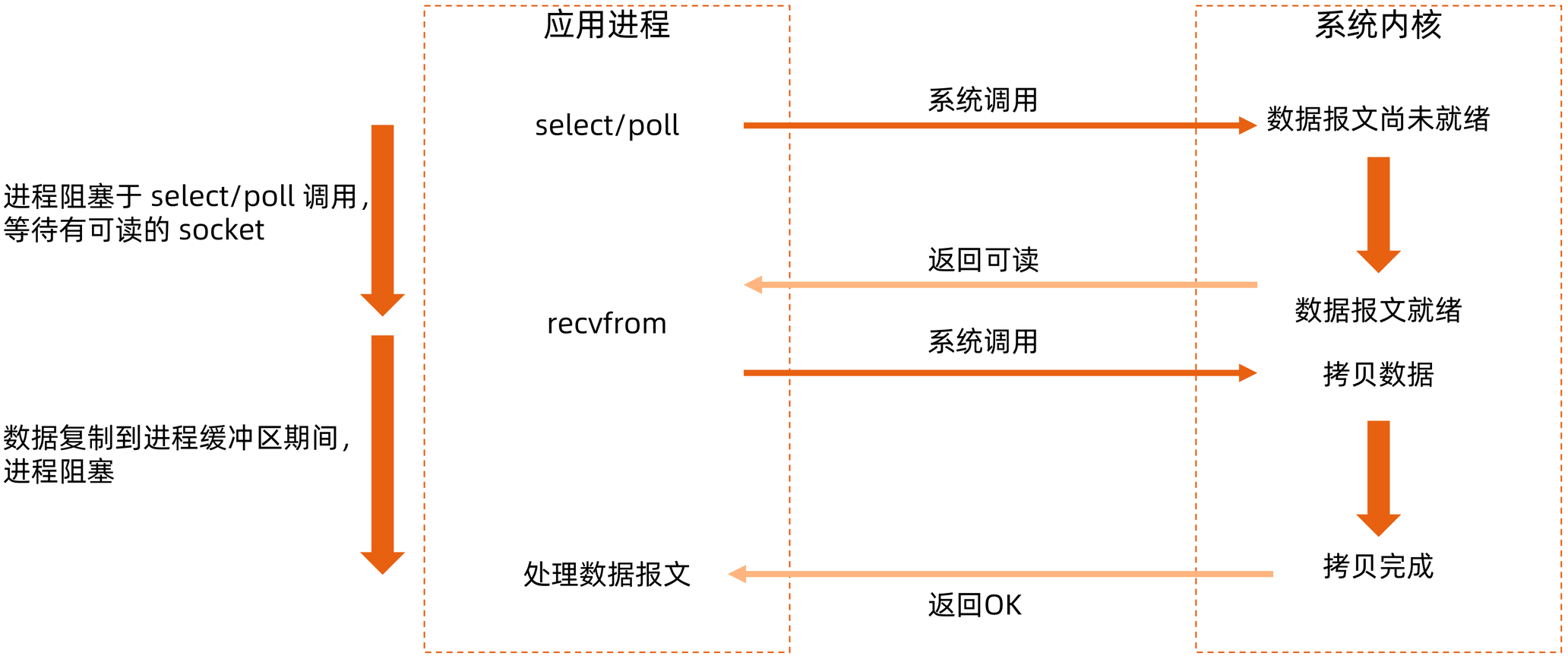
阻塞 IO 模型


应用进程
系统调用
系统内核
recvfrom
数据报文尚未就绪
进程阻塞于
recvfrom 调用
数据报文就绪
拷贝数据
系统调用返回
处理数据报
拷贝完成

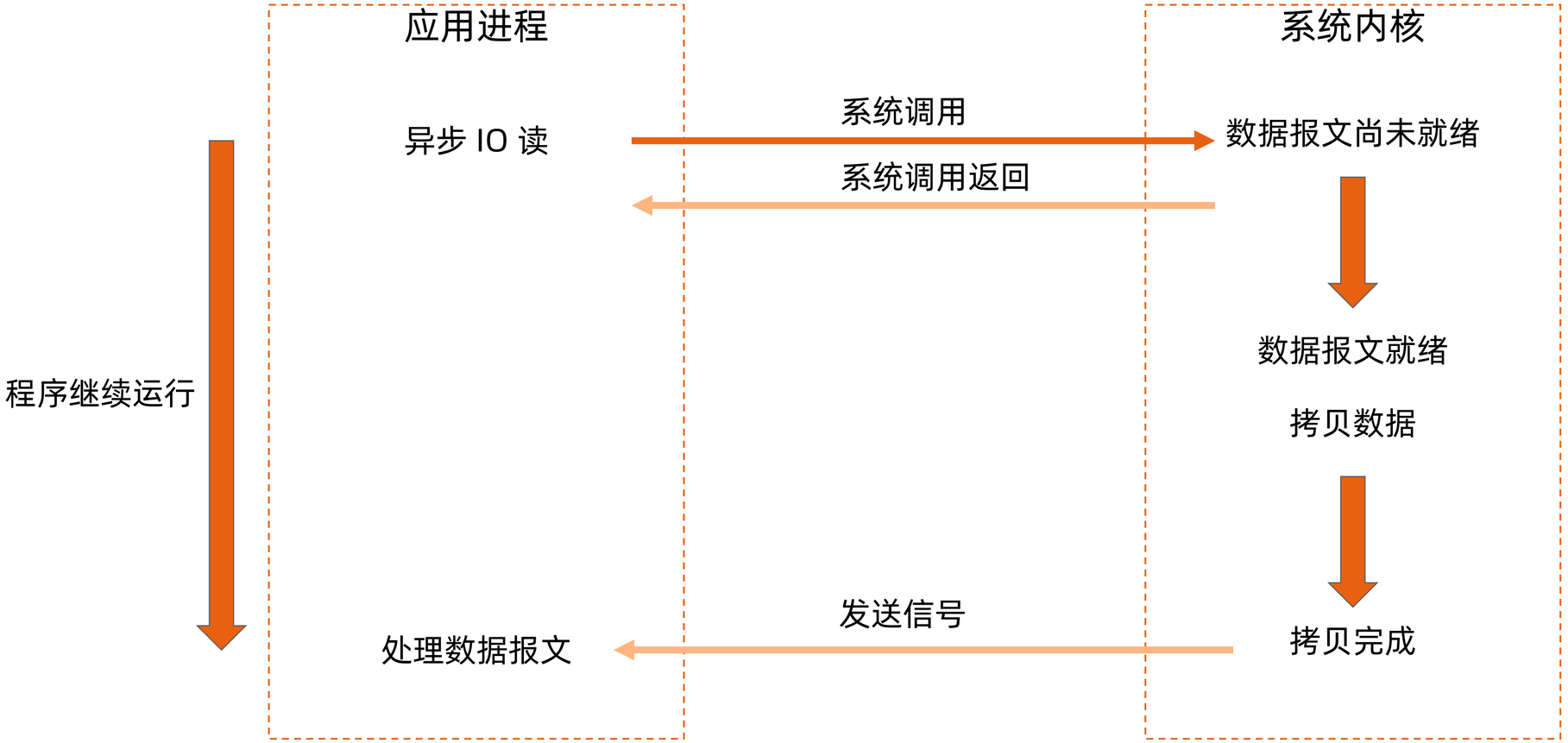
非阻塞 IO 模型


应用进程
系统内核
recvfrom
系统调用
数据报文尚未就绪
返回错误
recvfrom
系统调用
返回错误
数据报文就绪
recvfrom
系统调用
拷贝数据
进程重复调用
recvfrom，直至返回 OK
拷贝完成
处理数据报
返回OK

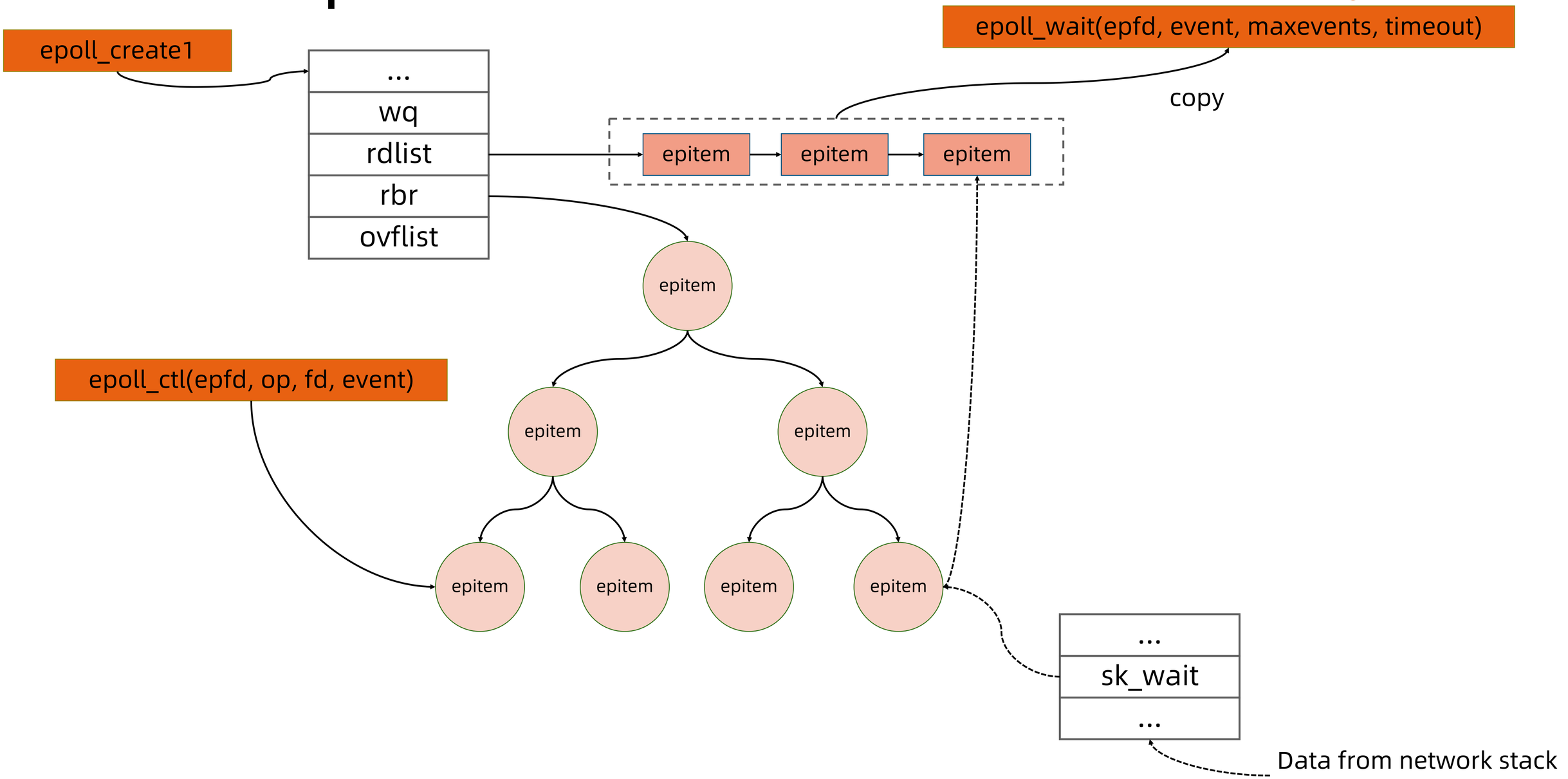
IO 多路复用


应用进程
系统内核
select/poll
系统调用
数据报文尚未就绪
进程阻塞于 select/poll 调用，
等待有可读的 socket
返回可读
数据报文就绪
recvfrom
系统调用
拷贝数据
数据复制到进程缓冲区期间，
进程阻塞
拷贝完成
处理数据报文
返回OK

异步 IO


应用进程
系统内核
异步 IO 读
系统调用
数据报文尚未就绪
系统调用返回
数据报文就绪
拷贝数据
程序继续运行
发送信号
处理数据报文
拷贝完成

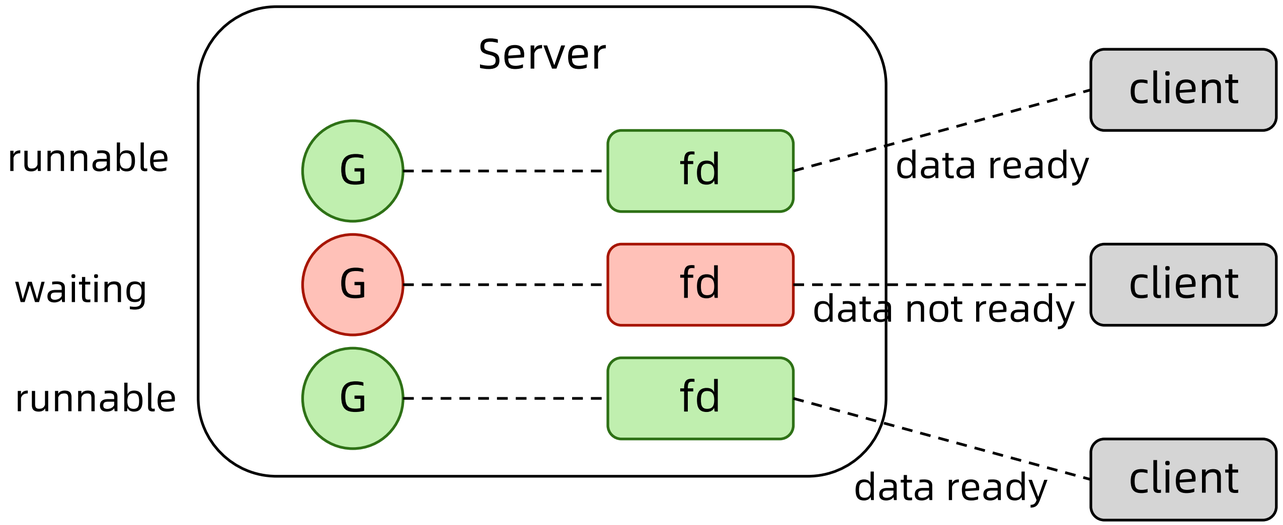
Linux epoll


epoll_create1
...
wq
rdlist
rbr
ovflist
epoll_wait(epfd, event, maxevents, timeout)
copy
epitem
epitem
epitem
epoll_ctl(epfd, op, fd, event)
epitem
epitem
epitem
epitem
epitem
epitem
...
sk_wait
...
Data from network stack

Go 语言高性能 httpserver 的实现细节

极客时间 训练营

- Go 语言将协程与 fd 资源绑定
  - 一个 socket fd 与一个协程绑定
  - 当 socket fd 未就绪时，将对应协程设置为 Gwaiting 状态，将 CPU 时间片让给其他协程
  - Go 语言 runtime 调度器进行调度唤醒协程时，检查 fd 是否就绪，如果就绪则将协程置为 Grunnable 并加入执行队列
  - 协程被调度后处理 fd 数据



代码实现细节

https://pouncing-waterfall-7c4.notion.site/http-server-socket-detail-e1f350d63c7c4d9f86ce140949bd90c2

7. 调试

debug

极客时间 训练营

- gdb:
  - Gccgo 原生支持 gdb，因此可以用 gdb 调试 Go 语言代码，但 dlv 对 Go 语言 debug 的支持比 gdb 更好
  - Gdb 对 Go 语言的栈管理，多线程支持等方面做的不够好，调试代码时可能有错乱现象
- dlv:
  - Go 语言的专有 debugger

dlv 的配置

极客时间 训练营

- 配置
    - 在 vscode 中配置 dlb
    - 菜单：View -> Command Palette
    - 选择 Go : Install/Update Tools，选择安装
    - 安装完后，从改入口列表中可以看到 dlv 和 dlv-dap 已经安装好
- Debug 方法
    - 在代码中设置断点
    - 菜单中选择 Run -> Start Debugging 即可进入调试

更多 debug 方法

极客时间 训练营

- 添加日志
  - 在关键代码分支中加入日志
  - 基于fmt包将日志输出到标准输出 stdout
    - fmt.Println()
  - fmt 无日志重定向，无日志分级
- 即与日志框架将日志输出到对应的 appender
  - 比如可利用 glog 进行日志输出
    - 可配置 appender，将标准输出转至文件
    - 支持多级日志输出，可修改配置调整日志等级
    - 自带时间戳和代码行，方便调试

Glog 使用方法示例


```
import (
"flag"

"github.com/golang/glog"
)
func main() {
     flag.Set("v", "4")
     glog.V(2).Info("Starting http server...")
     mux := http.NewServeMux()
     mux.HandleFunc("/", rootHandler)
     err := http.ListenAndServe(":80", mux)
     if err != nil {
          log.Fatal(err)
     }
}
```

性能分析（Performance Profiling）

极客时间训练营

- CPU Profiling: 在代码中添加 CPUProfile 代码，runtime/pprof 包提供支持

```
var cpuprofile = flag.String("cpuprofile", "", "write cpu profile to file")
func main() {
    flag.Parse()
    if *cpuprofile != "" {
        f, err := os.Create(*cpuprofile)
            if err != nil {
                log.Fatal(err)
            }
        pprof.StartCPUProfile(f)
    defer pprof.StopCPUProfile()
    }
}
```

分析 CPU 瓶颈

极客时间 训练营

- 运行 cpuprofilie 代码后，会在 /tmp/cpuprofile 中记录 cpu 使用时间
- 运行 go tool pprof /tmp/cpuprofile 进入分析模式
- 运行 top10 查看 top 10线程，显示 30ms 花费在 main.main

```
Showing nodes accounting for 30ms, 100% of 30ms total
      flat  flat%   sum%        cum   cum%
      30ms   100%   100%      30ms   100%  main.main
         0     0%   100%      30ms   100%  runtime.main
```

- (pprof) list main.main 显示 30 毫秒都花费在循环上

```
Total: 30ms
30ms      30ms (flat, cum)   100% of Total
      20ms      20ms     21:      for i := 0; i < 100000000; i++ {
      10ms      10ms     22:              result += I
```

- 可执行 web 命令生成 svg 文件，在通过浏览器打开 svg 文件查看图形化分析结果

其他可用 profiling 工具分析的问题

极客时间 训练营

- CPU profile
    - 程序的 CPU 使用情况，每 100 毫秒采集一次 CPU 使用情况
- Memory Profile
    - 程序的内存使用情况
- Block Profiling
    - 非运行态的 goroutine 细节，分析和查找死锁
- Goroutine Profiling
    - 所有 goroutines 的细节状态，有哪些 goroutine，它们的调用关系是怎样的

针对 http 服务的 pprof

极客时间 训练营

- net/http/pprof 包提供支持
- 如果采用默认 mux handle，则只需 import

_ "net/http/pprof”

- 如果采用自定义 mux handle，则需要注册 pprof handler

```
Import (
"net/http/pprof"
)

func startHTTP(addr string, s *tnetd.Server) {
    mux := http.NewServeMux()
    mux.HandleFunc( “/debug/pprof/” , pprof.Index)
    mux.HandleFunc( “/debug/pprof/profile” , pprof.Profile)
    mux.HandleFunc( “/debug/pprof/symbol” , pprof.Symbol)
    mux.HandleFunc( “/debug/pprof/trace” , pprof.Trace)
    server := &http.Server{
        Addr: addr,
        Handler: mux,
    }
    server.ListenAndServe()
}
```

分析 go profiling 结果

在运行了开启 pprof 的服务器以后，可以通过访问对应的 URL 获得 profile 结果

- allocs: A sampling of all past memory allocations
- block: Stack traces that led to blocking on synchronization primitives
- cmdline: The command line invocation of the current program
- goroutine: Stack traces of all current goroutines
- heap: A sampling of memory allocations of live objects. You can specify the gc GET parameter to run GC before taking the heap sample.

分析 go profiling 结果

- mutex: Stack traces of holders of contended mutexes
- profile: CPU profile. You can specify the duration in the seconds GET parameter. After you get the profile file, use the go tool pprof command to investigate the profile.
- threadcreate: Stack traces that led to the creation of new OS threads
- trace: A trace of execution of the current program. You can specify the duration in the seconds GET parameter. After you get the trace file, use the go tool trace command to investigate the trace.

结果分析示例

极客时间 训练营

- 分析 goroutine 运行情况
    - curl localhost/debug/pprof/goroutine?debug=2
- 分析堆内存使用情况
    - curl localhost/debug/pprof/heap?debug=2

8. Kubernetes 中常用代码解读

Rate Limit Queue

```
func (r *ItemExponentialFailureRateLimiter) When(item interface{}) time.Duration {

    r.failuresLock.Lock()
    defer r.failuresLock.Unlock()

    exp := r.failures[item]
    r.failures[item] = r.failures[item] + 1

    // The backoff is capped such that 'calculated' value never overflows.
    backoff := float64(r.baseDelay.Nanoseconds()) * math.Pow(2, float64(exp))
    if backoff > math.MaxInt64 {
        return r.maxDelay
    }

    calculated := time.Duration(backoff)
    if calculated > r.maxDelay {
        return r.maxDelay
    }
    return calculated
}
```

9. Kubernetes 日常运维中的代码调试场景

案例1：空指针

- 问题描述

Kubenetes 调度器在调度有外挂存储需求的 pod 的时候，在获取节点信息失败时会异常退出

panic: runtime error: invalid memory address or nil pointer dereference

[signal SIGSEGV: segmentation violation code=0x1 addr=0x0 pc=0x105e283]

案例1：空指针

- 根因分析

nil pointe 是 Go 语言中最常出现的一类错误，也最容易判断，通常在 call stack 中就会告诉你哪行代码有问题

在调度器 csi.go 中的如下代码，当 node 为 nil 的时候，对 node 的引用 node.Name 就会引发空指针

```
node := nodeInfo.Node()
    if node == nil {
    return framework.NewStatus(framework.Error, fmt.Sprintf("node not found: %s", node.Name))
    }
```

- 解决办法

当指针为空时，不要继续引用。

https://github.com/kubernetes/kubernetes/pull/102229

案例2：Map 的读写冲突

极客时间 训练营

- 问题描述：

程序在遍历 Kubernetes 对象的 Annotation 时异常退出

- 根因分析

Kubernetes 对象中 Label 和 Annotation 是 map[string]string

经常有代码需要修改这两个 Map

同时可能有其他线程 for...range 遍历

- 解决方法：
    - 用 sync.RWMutex 加锁
    - 使用线程安全 Map，比如 sync.Map{}

案例3：kube-proxy 消耗 10 个 CPU

极客时间 训练营

- 问题描述

客户汇报问题，kube-proxy 消耗了主机 10 个 CPU

- 根因分析
    - 登录问题机器，执行 top 命令查看 cpu 消耗，可以看到 kube-proxy 的 cpu 消耗和 pid 信息
    - 对 kube-proxy 进程运行 System profiling tool，发现 10 个 CPU 中，超过 60% 的 CPU 都在做垃圾回收，这说明 GC 需要回收的对象太多了，说明程序创建了大量可回收对象。
        - perf top -p <pid>

Overhead Shared Obj Symbol

26.48% kube-proxy [.] runtime.gcDrain

13.86% kube-proxy [.] runtime.greyobject

10.71% kube-proxy [.] runtime.(*lfstack).pop

10.04% kube-proxy [.] runtime.scanobject

案例3：kube-proxy 消耗 10 个 CPU

通过 pprof 分析内存占用情况

curl 127.0.0.1:10249/debug/pprof/heap?debug=2

1: 245760 [301102: 73998827520] @ 0x11ddcda 0x11f306e 0x11f35f5 0x11fbdce 0x1204a8a 0x114ed76 0x114eacb 0x11

0x11ddcd9 k8s.io/kubernetes/vendor/github.com/vishvananda/netlink.(*Handle).RouteListFiltered+0x679

0x11f306d k8s.io/kubernetes/pkg/proxy/ipvs.(*netlinkHandle).GetLocalAddresses+0xed

0x11f35f4 k8s.io/kubernetes/pkg/proxy/ipvs.(*realIPGetter).NodeIPs+0x64

0x11fbdcd k8s.io/kubernetes/pkg/proxy/ipvs.(*Proxier).syncProxyRules+0x47dd

案例3：kube-proxy 消耗 10 个

- heap dump 分析
  - GetLocalAddresses 函数调用创建了 301102 个对象，占用内存 73998827520
  - 如此多的对象被创建，显然会导致 kube-proxy 进程忙于 GC，占用大量 CPU
  - 对照代码分析 GetLocalAddresses 的实现，发现该函数的主要目的是获取节点本机 IP 地址，获取的方法是通过 ip route 命令获得当前节点所有 local 路由信息并转换成 go struct 并过滤掉 ipvs0网口上的路由信息
  - ip route show table local type local proto kernel
  - 因为集群规模较大，该命令返回 5000 条左右记录，因此每次函数调用都会有数万个对象被生成
  - 而 kube-proxy 在处理每一个服务的时候都会调用该方法，因为集群有数千个服务，因此，kube-proxy在反复调用该函数创建大量临时对象
- 修复方法
  - 函数调用提取到循环外
  - https://github.com/kubernetes/kubernetes/pull/79444

案例4：线程池耗尽

极客时间 训练营

- 问题描述：

在 Kubernetes 中有一个控制器，叫做 endpoint controller，该控制器符合生产者消费者模式，默认有5个 worker 线程作为消费者。该消费者在处理请求时，可能调用的 LBaaS 的 API 更新负载均衡配置。我们发现该控制器会时不时不工作，具体表现为，该做的配置变更没发生，相关日志也不打印了。

- 根因分析：

通过 pprof 打印出该进程的所有 go routine 信息，发现 worker 线程都卡在 http 请求调用处。

当worker线程调用 LBaaS API 时，底层是 net/http 包调用，而客户端在发起连接请求时，未设置客户端超时时间。这导致当出现某些网络异常时，客户端会永远处于等待状态。

- 解决方法：

修改代码加入客户端超时控制。

课后练习 2.2

- 编写一个 HTTP 服务器，此练习为正式作业需要提交并批改
- 鼓励群里讨论，但不建议学习委员和课代表发满分答案，给大家留一点思考空间
- 大家视个人不同情况决定完成到哪个环节，但尽量把1都做完

    1. 接收客户端 request，并将 request 中带的 header 写入 response header
    2. 读取当前系统的环境变量中的 VERSION 配置，并写入 response header
    3. Server 端记录访问日志包括客户端 IP，HTTP 返回码，输出到 server 端的标准输出
    4. 当访问 localhost/healthz 时，应返回200

THANKS