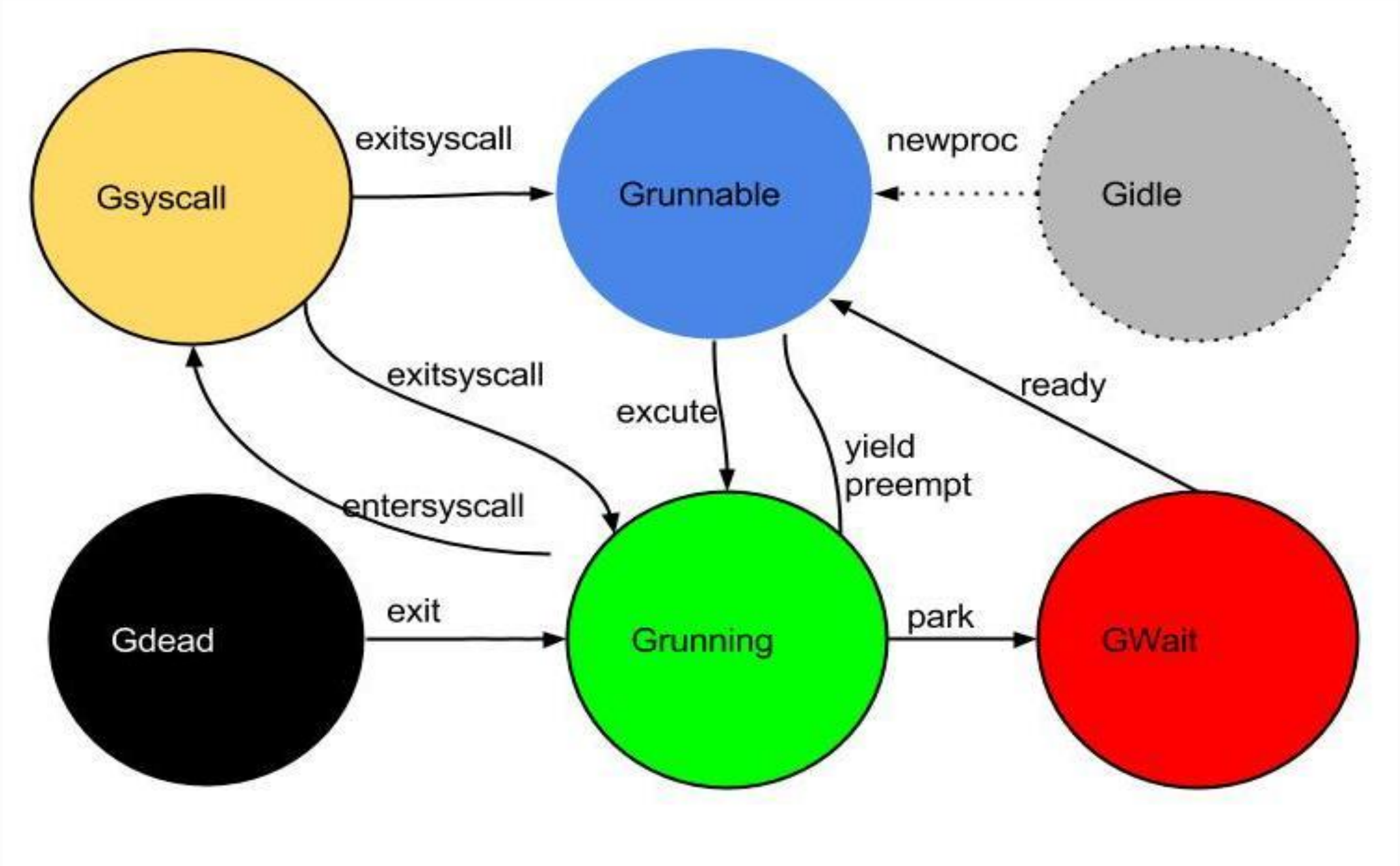
# Golang语言同步实践

Huayulei_2003@hotmail.com
2019/10/15

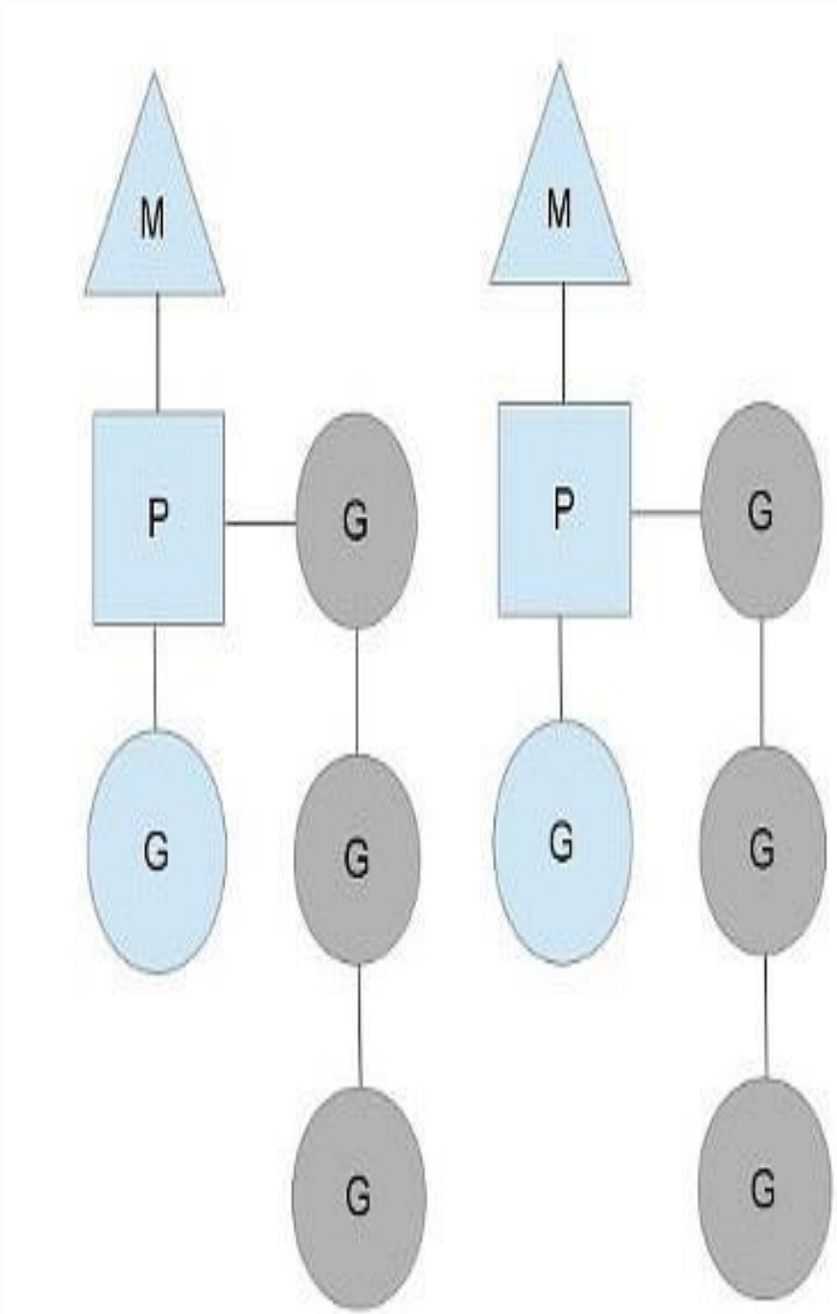
# 目录

- 理解goroutine
- 理解channel
- 理解golang同步机制
- 项目实践
- Q&A

# 理解goroutine：状态机



# 理解goroutine：高级协程




- G: 表示goroutine，存储了goroutine的执行stack信息、goroutine状态以及goroutine的任务函数等；另外G对象是可以重用的。
- P: 表示逻辑processor，P的数量决定了系统内最大可并行的G的数量（前提：系统的物理cpu核数>=P的数量）；P的最大作用还是其拥有的各种G对象队列、链表、一些cache和状态。
- M: M代表着真正的执行计算资源。在绑定有效的p后，进入schedule循环；而schedule循环的机制大致是从各种队列、p的本地队列中获取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。

# 理解goroutine：调度器(1/2)

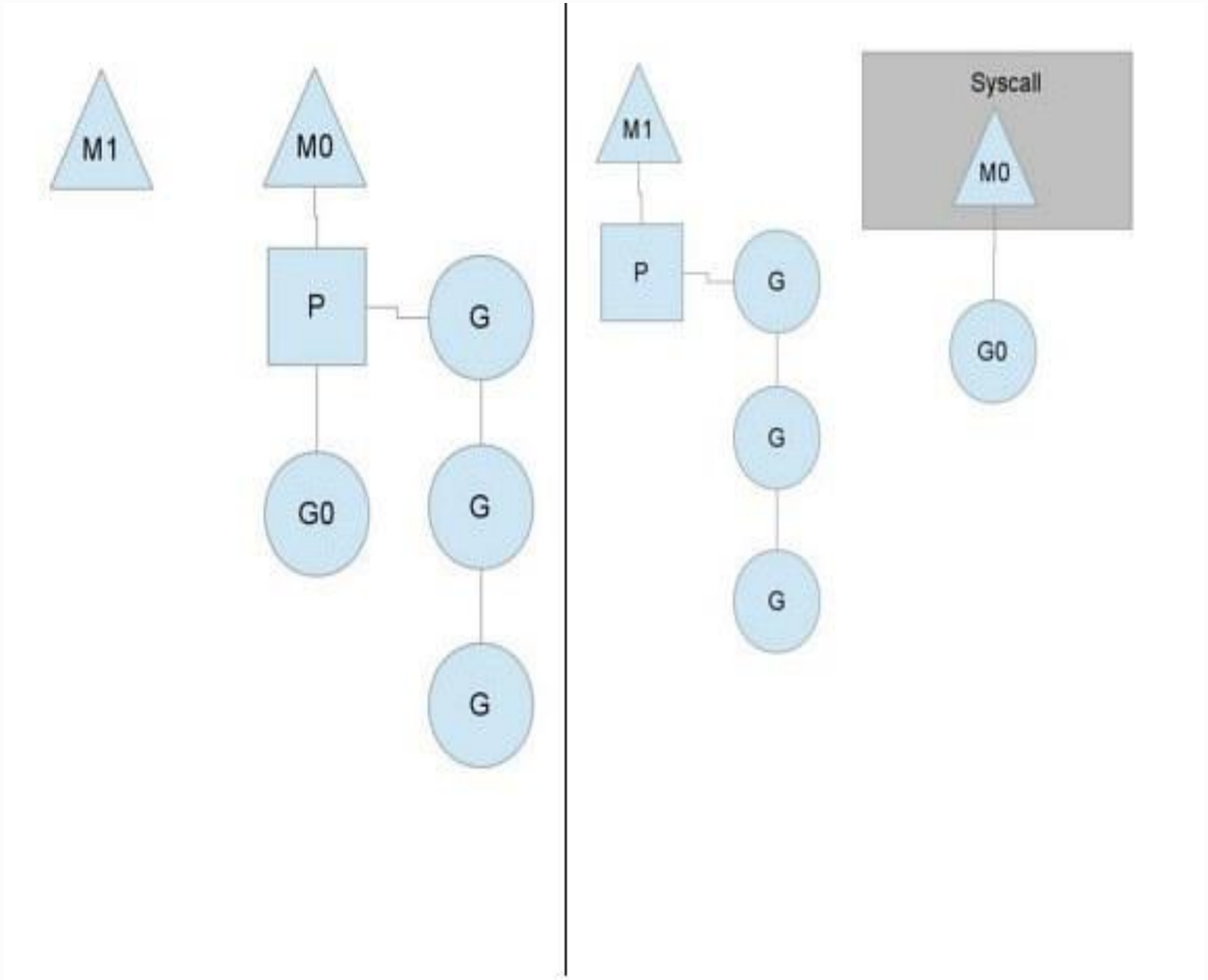
1.抢占式调度的原理是在每个函数或方法的入口，加上一段额外的代码，让runtime有机会检查是否需要执行抢占调度。实现：如果一个G任务运行10ms，sysmon就会认为其运行时间太久而发出抢占式调度的请求。一旦G的抢占标志位被设为true，那么当这个G下一次调用函数或方法时，runtime便可以将G抢占，并移出运行状态，放入P的local runq中，等待下一次被调度

2. Go runtime实现了netpoller，这使得G发起网络I/O操作也不会导致M被阻塞（仅阻塞G），从而不会导致大量M被创建出来。但是对于regular file的I/O操作一旦阻塞，那么M将进入sleep状态，等待I/O返回后被唤醒；这种情况下P将与sleep的M分离，再选择一个idle的M。如果此时没有idle的M，则会新创建一个M，这就是大量I/O操作导致大量Thread被创建的原因。

3. channel阻塞或network I/O情况下的调度：如果G被阻塞在某个channel操作或network I/O操作上时，G会被放置到某个wait队列中，而M会尝试运行下一个runnable的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I/O available或channel操作完成，在wait队列中的G会被唤醒，标记为runnable，放入到某P的队列中，绑定一个M继续执行。

4. system call阻塞情况下的调度:如果G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也会解绑P，与G一起进入阻塞状态。如果此时有idle的M，则P与其绑定继续执行其他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M。当阻塞在syscall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会被标记为runnable，之前的那个sleep的M将再次进入sleep。

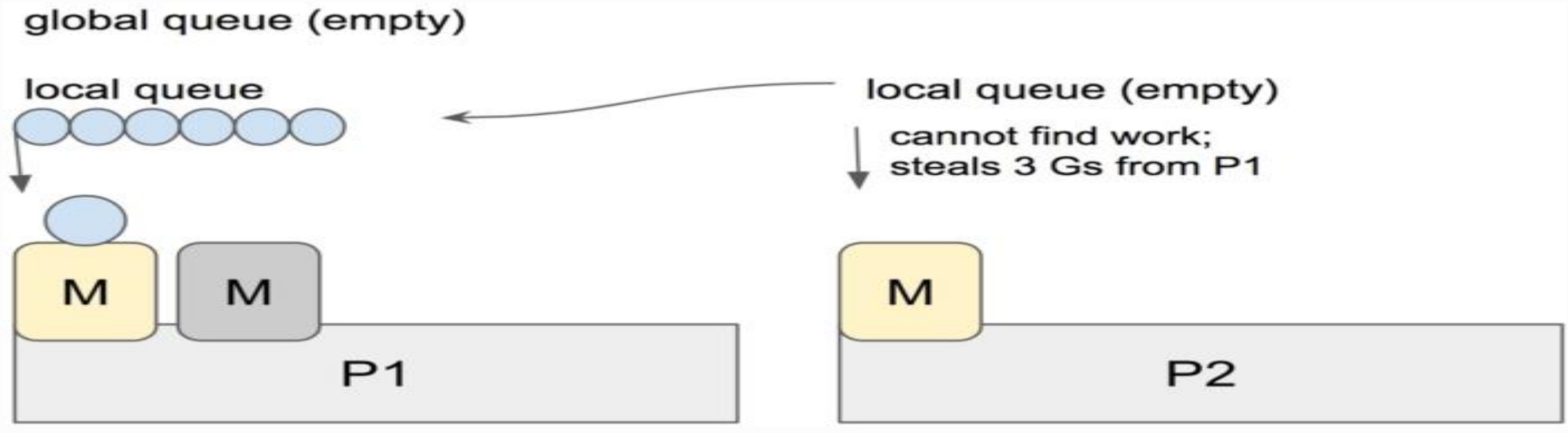
# 理解goroutine：调度器(2/2)




1.当一个OS线程M0陷入阻塞时，P转而在OS线程M1上运行。调度器保证有足够的线程来运行所有的context P。图中的M1可能是被创建，也可能从线程缓存中取出。

2.当M0返回时，它必须尝试取得一个context P来运行goroutine，一般情况下，它会从其他的OS线程那里steal偷一个过来，如果没有偷到，它就把goroutine放在global runqueue里。然后把自己放到线程缓存中或者转入睡眠状态。

# 理解goroutine：G的偷取




```
runtime.schedule() {
// only 1/61 of the time, check the global runnable queue for a G.
 // if not found, check the local queue. // if not found,
// try to steal from other Ps.
// if not, check the global runnable queue.
// if not found, poll network.
}
```

在上图中，P2 这个处理器无法找到任何可执行的 goroutines。因此，它随机选择另一个处理器 P1，并将 3 个 goroutines 偷到自己的局部队列中。P2 将执行这些 goroutines，而调度器也将在多个处理器之间更均衡的调度。

# 理解goroutine:control scheduler

- GOMAXPROCS(n) - set max simultaneously executing CPUS. If n < 1, it does not change the current setting. The number of logical CPUs on the local machine can be queried with NumCPU.
- Goexit() - terminate the calling goroutine, No other goroutine is affected.
- Gosched() - yield processor allowing other goroutines to run
- LockOSThread()/UnlockOSThread() - wire/unwire goroutine to current OS thread. LockOSThread wires the calling goroutine to its current operating system thread. The calling goroutine will always execute in that thread, and no other goroutine will execute in it, until the calling goroutine has made as many calls to UnlockOSThread as to LockOSThread.

# 理解Channel:概述

- Channel是goroutine-safe的

hchan 中的 lock mutex

- 可存储、传递值，FIFO(先入先出)

通过 hchan 中的环形缓冲区来实现

- 导致 goroutine 的阻塞和恢复

hchan 中的 sendq和recvq，也就是 sudog 结构的链表队列

调用运行时调度器 (gopark(), goready())

**本质是指向hchan 结构的堆内存空间的指针**

# 理解Channel:基础回顾

```
// G1
func main() {
  ...
  for _, task := range tasks {
    ch <- task
  }
  ...
}
// G2
func worker(ch chan Task) {
  for {
    task :=<-ch
    process(task)
  }
}
```

G1 中的 ch <- taskN具体流程：

① 获取锁

② enqueue(taskN)（这里是内存复制taskN）

③ 释放锁

G2 的 t:= <- ch读取数据流程：

① 获取锁

② t = dequeue()（同样，这里也是内存复制）

③ 释放锁

所有通讯的数据都是内存拷贝，核心思想：

Do not communicate by sharing memory;instead, share memory by communicating.

# 理解Channel:写入满阻塞和恢复

**写goroutine 被阻塞的具体过程：**

当 ch <- taskn 执行的时候，channel 中已经满了，需要 pause G1。这个时候，：

① G1 会调用运行时的 gopark，

② 然后 Go 的运行时调度器就会接管

③ 将 G1 的状态设置为 wait状态

④ 断开 G1 和 M 之间的关系（switch out)，因此 G1 脱离 M，换句话说，M 空闲了，可以执行别的任务了。

⑤ 从 P 的运行队列中，取得一个runnable的 goroutine G

⑥ 建立新的 G 和 M 的关系（Switch in)，因此 新G 就准备好运行了。

⑦ 当调度器返回的时候，新的 G 就开始运行了，而 G1 则不会运行，也就是 block 了。

对于 goroutine 来说，G1 被阻塞了，新的 G 开始运行了；而对于操作系统线程 M 来说，则根本没有被阻塞。

当 G2 调用 t := <- ch 的时候，channel 的缓冲是满的，而且还有一个 G1 在等候发送队列里，然后 G2 执行下面的操作：

① G2 先执行 dequeue() 从缓冲队列中取得 task1 给 t

② G2 从 sendq 中弹出一个等候发送的 sudog

③ 将弹出的 sudog 中的 elem 的值 enqueue() 到 buf 中

④ 将弹出的 sudog 中的 goroutine，也就是 G1，状态从 waiting 改为 runnable

⑤ 然后，G2 需要通知调度器 G1 已经可以进行调度了，因此调用 goready(G1)。

⑥ 调度器将 G1 的状态改为 runnable

⑦ 调度器将 G1 压入 P 的运行队列，因此在将来的某个时刻调度的时候，G1 就会开始恢复运行。

⑧ 返回到 G2

由 G2 来负责将 G1 的 elem 压入 buf 的，这是一个优化。这样将来 G1 恢复运行后，就不必再次获取锁、enqueue()、释放锁了。这样就避免了多次锁的开销

# 理解Channel:读取空阻塞和恢复

读goroutine阻塞的流程：

如果 G2 先执行了 t := <- ch，此时 buf 是空的，因此 G2 会被阻塞，他的流程是：

① G2 给自己创建一个 sudog 结构变量。其中 g 是自己，也就是 G2，而 elem 则指向 t

② 将这个 sudog 变量压入 recvq 等候接收队列

③ G2 需要告诉 goroutine，自己需要 pause 了，于是调用 gopark(G2)

④ 和之前一样，调度器将其 G2 的状态改为 waiting

⑤ 断开 G2 和 M 的关系

⑥ 从 P 的运行队列中取出其他 goroutine

建立新的 goroutine 和 M 的关系

① 返回，开始继续运行新的 goroutine

当 G1 开始发送数据的流程：

G1 可以将 enqueue(task)，然后调用 goready(G2)。

根据 hchan 结构的状态，已经知道 task 进入 buf 后，G2 恢复运行后，会读取其值，复制到 t 中。G1 不走 buf，G1 直接把数据给 G2。

Goroutine 通常都有自己的栈，互相之间不会访问对方的栈内数据，除了 channel。由于已经知道了 t 的地址（通过 elem指针），而且由于 G2 不在运行，所以可以很安全的直接赋值。当 G2 恢复运行的时候，既不需要再次获取锁，也不需要对 buf 进行操作。从而节约了内存复制、以及锁操作的开销。

# 理解Channel:无缓冲和select

无缓冲 channel

无缓冲的 channel 行为与前面说的直接发送的一样：

① 接收方阻塞 → 发送方直接写入接收方的栈

② 发送方阻塞 → 接收方直接从发送方的 sudog 中读取

select

① 先把所有需要操作的 channel 上锁

② 给自己创建一个 sudog，然后添加到所有 channel 的 sendq或recvq（取决于是发送还是接收）

③ 把所有的 channel 解锁，然后 pause 当前调用 select 的 goroutine（gopark()，进入wait状态）

④ 然后当有任意一个 channel 可用时，select 的这个 goroutine 就会被调度执行。

⑤ resuming mirrors the pause sequence

**设计上:更倾向于带锁的队列，而不是无锁的实现。**

“性能提升不是凭空而来的，是随着复杂度增加而增加的。” – dvyokov

**simplicity vs performance 的权衡后的结果**

# 理解Channel:使用总结

发生 panic 的情况有三种：

- 向一个关闭的 channel 进行写操作
- 关闭一个 nil 的 channel
- 重复关闭一个 channel

操作 channel 的结果：

| 操作 | nil channel | closed channel | not nil, not closed channel |
| --- | --- | --- | --- |
| close | panic | panic | 正常关闭 |
| 读 <- ch | 阻塞 | 读到对应类型的零值 | 阻塞或正常读取数据。缓冲型 channel 为空或非缓冲型 channel 没有等待发送者时会阻塞 |
| 写 ch <- | 阻塞 | panic | 阻塞或正常写入数据。非缓冲型 channel 没有等待接收者或缓冲型 channel buf 满时会被阻塞 |

# Golang同步机制：基础知识

- 使用channel
- 使用WaitGroup
- 使用Mutex、Once、Atomic
- 使用Mutext andCond
- 使用Context
- 子仓库 x/sync 中的包：ErrGroup、Semaphore 和 SingleFlight
- 其他：文件锁，定时器等

# Golang同步机制:使用channel

例子： stop.go

```
func main() {
        dataCh := make(chan int, 100)
        stopCh := make(chan struct{})

        // senders
        for i := 0; i < 5; i++ {
                go func() {
                        for {
                                select {
                                case <-stopCh:
                                        fmt.Println("stop")
                                        return
                                case dataCh <- rand.Intn(5):
                                }
                        }
                }()
        }
        for value := range dataCh {
                if value == 1 {
                        fmt.Println("send stop signal to senders.")
                        close(stopCh)
                        break
                }
                fmt.Println(value)
        }
        time.Sleep(5 * time.Second)
}
```

# Golang同步机制:使用WaitGroup

例子： wait_group.go

```
func main() {
        var wg sync.WaitGroup
        var urls = []string{
                "http://www.baidu.org/",
                "http://www.sohu.com/",
                "http://www.163.com/",
        }
        for _, url := range urls {
                // Increment the WaitGroup counter.
                wg.Add(1)
                // Launch a goroutine to fetch the URL.
                go func(url string) {
                        // Decrement the counter when the goroutine completes.
                        defer wg.Done()
                        // Fetch the URL.
                        http.Get(url)
                }(url)
        }
        // Wait for all HTTP fetches to complete.
        wg.Wait()
}
```

# Golang同步机制: Mutex Once Atomic

- Mutex：A Mutex is a mutual exclusion lock. The zero value for a Mutex is an unlocked mutex.

*func (m *Mutex) Lock()*

*func (m *Mutex) Unlock()*

- RWMutex：A RWMutex is a reader/writer mutual exclusion lock. The lock can be held by an arbitrary number of readers or a single writer. The zero value for a RWMutex is an unlocked mutex.

*func (rw *RWMutex) Lock()*

*func (rw *RWMutex) RLock()*

*func (rw *RWMutex) RUnlock()*

*func (rw *RWMutex) Unlock()*

- Once：Once is an object that will perform exactly one action.

*func (o *Once) Do(f func())*

Do calls the function f if and only if Do is being called for the first time for this instance of Once.if once.Do(f) is called multiple times, only the first call will invoke f, even if f has a different value in each invocation.

- Atomic：Package atomic provides low-level atomic memory primitives useful for implementing synchronization algorithms.

# Golang同步机制: Mutex and Cond

Cond:Cond implements a condition variable, a rendezvous point for goroutines waiting for or announcing the occurrence of an event.

*func (c *Cond) Broadcast()*

*func (c *Cond) Signal()*

*func (c *Cond) Broadcast()*

例子： cond_broad.go and cond_signal.go

和Linux下Pthread库类似.

*#include <pthread.h>*

*int pthread_cond_destroy(pthread_cond_t *cond);*

*int pthread_cond_init(pthread_cond_t *restrict cond, const pthread_condattr_t *restrict attr);*

*int pthread_cond_broadcast(pthread_cond_t *cond);*

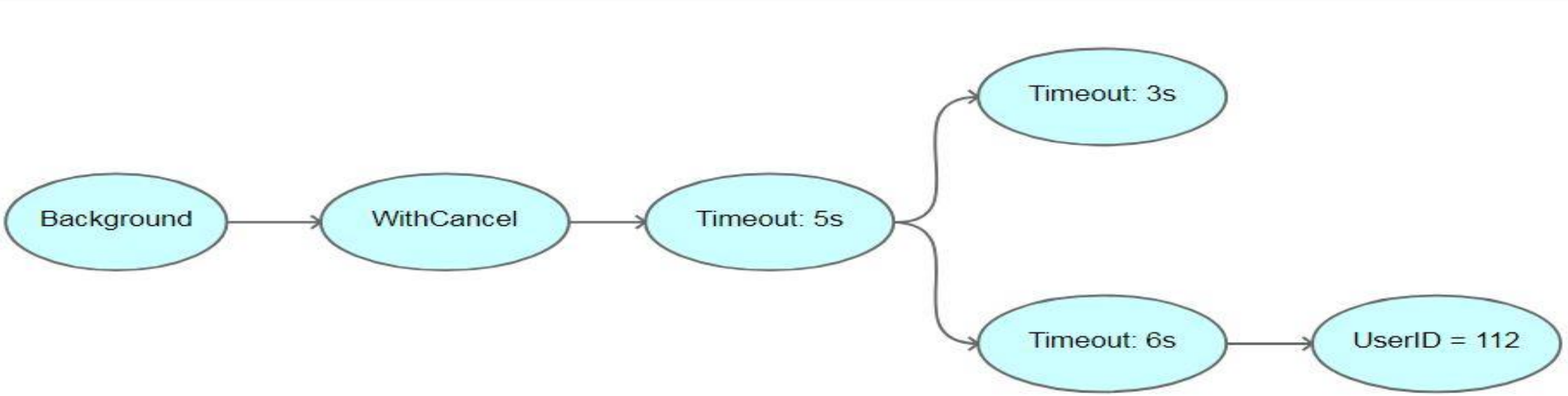
*int pthread_cond_signal(pthread_cond_t *cond);*

*int pthread_cond_timedwait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex, const struct timespec *restrict abstime);*

*int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex);*

# Golang同步机制:例子

```
func tree() {
  ctx1 := context.Background()
  ctx2, _ := context.WithCancel(ctx1)
  ctx3, _ := context.WithTimeout(ctx2, time.Second * 5)
  ctx4, _ := context.WithTimeout(ctx3, time.Second * 3)
  ctx5, _ := context.WithTimeout(ctx3, time.Second * 6)
  ctx6 := context.WithValue(ctx5, "userID", 12)
}
```



# Golang同步机制:Context概述

context.Context API：

```
type Context interface {
  Deadline() (deadline time.Time, ok bool)
  Done() <-chan struct{}
  Err() error
  Value(key interface{}) interface{}
}
```

两类操作：

3个函数用于限定什么时候子节点退出；

1个函数用于设置请求范畴的变量

什么时候应该使用 Context：

1. 每一个 调用都应该有超时退出的场景
2. 不仅仅是超时，还需要去结束那些不再需要操作的行为
3. 任何函数可能被阻塞，或者需要很长时间来完成的，都应该有个 context.Context

如何创建 Context:

1. 在 调用 开始的时候，使用 context.Background()
2. 请求来了后从创建context, 每个请求都有自己的 context.Background() 。
3. 如果你没有 context，却需要调用一个 context 的函数的话，用 context.TODO()
4. 如果某步操作需要自己的超时设置的话，给它一个独立的 sub-context

# Golang同步机制:使用Context

Context的使用：

1. 如果有 Context，将其作为第一个变量。如 func (d* Dialer) DialContext(ctx context.Context, network, address string) (Conn, error)
2. 将其作为可选的方式，用 request 结构体方式。如：func (r *Request) WithContext(ctx context.Context) *Request
3. 不要把 Context 存储到一个 struct 里，除非你使用的是像 http.Request 中的 request 结构体的方式，并且request 结构体应该以 Request 结束为生命终止
4. 要养成关闭 Context 的习惯，特别是超时的Context

   *ctx, cancel := context.WithTimeout(parentCtx, time.Second * 2)*

   *defer cancel()*

   使用 Timeout 会导致内部使用 time.AfterFunc，从而会导致 context 在计时器到时之前都不会被垃圾回收。在建立之后，应立即 defer cancel() 。
5. 可以用errgroup 类似的实现，使用在以下场景：同时有很多并发请求，并需要集中处理超时、出错终止其它并发任务，如下面例子：

# Golang同步机制: errgroup

例子:

```
Import   eg “golang.org/x/sync/errgroup”
func DoTwoRequestsAtOnce(ctx context.Context) error {
  eg, egCtx := errgroup.WithContext(ctx)
  var resp1, resp2 *http.Response
  f := func(loc string, respIn **http.Response) func() error {
    return func() error {
      reqCtx, cancel := context.WithTimeout(egCtx, time.Second)
      defer cancel()
      req, _ := http.NewRequest("GET", loc, nil)
      var err error
      *respIn, err = http.DefaultClient.Do(req.WithContext(reqCtx))
      if err == nil && (*respIn).StatusCode >= 500 {
        return errors.New("unexpected!")
      }
      return err
    }
  }
  eg.Go(f("http://localhost:8080/fast_request", &resp1))
  eg.Go(f("http://localhost:8080/slow_request", &resp2))
  return eg.Wait()
}
```

Errgroup：在一组 Goroutine 中提供了同步、错误传播以及上下文取消的功能

- 主要functions:

  *func WithContext(ctx context.Context) (*Group, context.Context)*

  *func (g *Group) Go(f func() error)*

  *func (g *Group) Wait() error*

- Doc and example:
  https://godoc.org/golang.org/x/sync/errgroup
  https://godoc.org/golang.org/x/sync/errgroup#example-Group--JustErrors

*在这个例子中，同时发起了两个调用，当任何一个调用超时或者出错后，会终止另一个调用。这里就是利用前面讲到的 errgroup 来实现的，应对有很多并非请求，并需要集中处理超时、出错终止其它并发任务的时候，这个 pattern 使用起来很方便。*

# Golang同步机制: semaphore

Semaphore是带权重的信号量，我们可以按照不同的权重对资源的访问进行管理。主要functions：

*func NewWeighted(n int64) *Weighted*

*func (s *Weighted) Acquire(ctx context.Context, n int64) error*

*func (s *Weighted) Release(n int64)*

*func (s *Weighted) TryAcquire(n int64) bool*

注意事项

- Acquire 和 TryAcquire 方法都可以用于获取资源，前者用于同步获取会等待锁的释放，后者会在无法获取锁时直接返回；
- Release 方法会按照 FIFO 的顺序唤醒可以被唤醒的 Goroutine；
- 如果一个 Goroutine 获取了较多地资源，由于 Release 的释放策略可能会等待比较长的时间；

例子： https://godoc.org/golang.org/x/sync/semaphore#example-package--WorkerPool

# Golang同步机制: singleflight

Singleflight提供了一个重复的函数调用抑制机制，对于同一个 Key 最终只会进行一次函数调用。主要functions：

*func (g *Group) Do(key string, fn func() (interface{}, error)) (v interface{}, err error, shared bool)*

*func (g *Group) DoChan(key string, fn func() (interface{}, error)) <-chan Result*

*func (g *Group) Forget(key string)*

注意事项

- Do 和 DoChan 一个用于同步阻塞调用传入的函数，一个用于异步调用传入的参数并通过 Channel 接受函数的返回值；
- Forget 方法可以通知 singleflight 在持有的映射表中删除某个键，接下来对该键的调用就会直接执行方法而不是等待前面的函数返回；
- 多Goroutine调用模式：一旦调用的函数返回了错误，所有在等待的 Goroutine 也都会接收到同样的错误；

例子： singleflight.go

# 项目实践:Channel Feed

// Feed implements one-to-many subscriptions where the carrier of events is a channel.

// Values sent to a Feed are delivered to all subscribed channels simultaneously.

//

// Feeds can only be used with a single type. The type is determined by the first Send or

// Subscribe operation. Subsequent calls to these methods panic if the type does not

// match.

参考项目:

https://github.com/ethereum/go-ethereum/blob/master/event/feed.go

# 项目实践:Worker Pool

使用Context和WaitGroup同步多协程

参考项目:
https://github.com/yuleihua/logstatd/blob/master/work_pool.go

# Q&A

Thank you very much !