



下载APP



## 17 | SingleFlight 和 CyclicBarrier: 请求合并和循环栅栏该怎么用?

2020-11-18 晁岳攀

Go 并发编程实战课

[进入课程 >](#)**讲述: 安晓辉**

时长 16:16 大小 14.91M



你好，我是鸟窝。

这节课，我来给你介绍两个非常重要的扩展并发原语：SingleFlight 和 CyclicBarrier。SingleFlight 的作用是将并发请求合并成一个请求，以减少对下层服务的压力；而 CyclicBarrier 是一个可重用的栅栏并发原语，用来控制一组请求同时执行的数据结构。

其实，它们两个并没有直接的关系，只是内容相对来说比较少，所以我打算用最短的时间带你掌握它们。一节课就能掌握两个“武器”，是不是很高效？



### 请求合并 SingleFlight

SingleFlight 是 Go 开发组提供的一个扩展并发原语。它的作用是，在处理多个 goroutine 同时调用同一个函数的时候，只让一个 goroutine 去调用这个函数，等到这个 goroutine 返回结果的时候，再把结果返回给这几个同时调用的 goroutine，这样可以减少并发调用的数量。

这里我想先回答一个问题：标准库中的 sync.Once 也可以保证并发的 goroutine 只会执行一次函数 f，那么，SingleFlight 和 sync.Once 有什么区别呢？

其实，sync.Once 不是只在并发的時候保证只有一个 goroutine 执行函数 f，而是会保证永远只执行一次，而 SingleFlight 是每次调用都重新执行，并且在多个请求同时调用的时候只有一个执行。它们两个面对的场景是不同的，**sync.Once 主要是用在单次初始化场景中，而 SingleFlight 主要用在合并并发请求的场景中**，尤其是缓存场景。

如果你学会了 SingleFlight，在面对秒杀等大并发请求的场景，而且这些请求都是读请求时，你就可以把这些请求合并为一个请求，这样，你就可以将后端服务的压力从 n 降到 1。尤其是在面对后端是数据库这样的服务的时候，采用 SingleFlight 可以极大地提高性能。那么，话不多说，就让我们开始学习 SingleFlight 吧。

## 实现原理

SingleFlight 使用互斥锁 Mutex 和 Map 来实现。Mutex 提供并发时的读写保护，Map 用来保存同一个 key 的正在处理 (in flight) 的请求。

SingleFlight 的数据结构是 Group，它提供了三个方法。

```
type Group
  ◦ 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)
```

Do: 这个方法执行一个函数，并返回函数执行的结果。你需要提供一个 key，对于同一个 key，在同一时间只有一个在执行，同一个 key 并发的请求会等待。第一个执行的请求返回的结果，就是它的返回结果。函数 fn 是一个无参的函数，返回一个结果或者 error，而 Do 方法会返回函数执行的结果或者是 error，shared 会指示 v 是否返回给多个请求。

DoChan: 类似 Do 方法, 只不过是返回一个 chan, 等 fn 函数执行完, 产生了结果以后, 就能从这个 chan 中接收这个结果。

Forget: 告诉 Group 忘记这个 key。这样一来, 之后这个 key 请求会执行 f, 而不是等待前一个未完成的 fn 函数的结果。

下面, 我们来看具体的实现方法。

首先, SingleFlight 定义一个辅助对象 call, 这个 call 就代表正在执行 fn 函数的请求或者是已经执行完的请求。Group 代表 SingleFlight。

 复制代码

```

1  // 代表一个正在处理的请求, 或者已经处理完的请求
2  type call struct {
3      wg sync.WaitGroup
4
5
6      // 这个字段代表处理完的值, 在waitgroup完成之前只会写一次
7      // waitgroup完成之后就读取这个值
8      val interface{}
9      err error
10
11      // 指示当call在处理时是否要忘掉这个key
12      forgotten bool
13      dups int
14      chans []chan<- Result
15  }
16
17      // group代表一个singleflight对象
18  type Group struct {
19      mu sync.Mutex // protects m
20      m map[string]*call // lazily initialized
21  }
```

我们只需要查看一个 Do 方法, DoChan 的处理方法是类似的。


 复制代码

```

1  func (g *Group) Do(key string, fn func() (interface{}, error)) (v interface{
2      g.mu.Lock()
3      if g.m == nil {
4          g.m = make(map[string]*call)
5      }
6      if c, ok := g.m[key]; ok { //如果已经存在相同的key
```

```
7      c.dups++
8      g.mu.Unlock()
9      c.wg.Wait() //等待这个key的第一个请求完成
10     return c.val, c.err, true //使用第一个key的请求结果
11 }
12 c := new(call) // 第一个请求, 创建一个call
13 c.wg.Add(1)
14 g.m[key] = c //加入到key map中
15 g.mu.Unlock()
16
17
18 g.doCall(c, key, fn) // 调用方法
19 return c.val, c.err, c.dups > 0
20 }
```

doCall 方法会实际调用函数 fn:

 复制代码

```
1 func (g *Group) doCall(c *call, key string, fn func() (interface{}, error))
2     c.val, c.err = fn()
3     c.wg.Done()
4
5
6     g.mu.Lock()
7     if !c.forgotten { // 已调用完, 删除这个key
8         delete(g.m, key)
9     }
10    for _, ch := range c.chans {
11        ch <- Result{c.val, c.err, c.dups > 0}
12    }
13    g.mu.Unlock()
14 }
```

在这段代码中, 你要注意下第 7 行。在默认情况下, forgotten==false, 所以第 8 行默认会被调用, 也就是说, 第一个请求完成后, 后续的同个 key 的请求又重新开始新一次的 fn 函数的调用。

Go 标准库的代码中就有一个 SingleFlight 的 [实现](#), 而扩展库中的 SingleFlight 就是在标准库的代码基础上改的, 逻辑几乎一模一样, 我就不多说了。

## 应用场景

了解了 SingleFlight 的实现原理, 下面我们来看看它都应用于什么场景中。

Go 代码库中有两个地方用到了 SingleFlight。

第一个是在 [net/lookup.go](https://golang.org/src/net/lookup.go) 中，如果同时有查询同一个 host 的请求，lookupGroup 会把这些请求 merge 到一起，只需要一个请求就可以了：

[复制代码](#)

```
1 // lookupGroup merges LookupIPAddr calls together for lookups for the same
2 // host. The lookupGroup key is the LookupIPAddr.host argument.
3 // The return values are ([]IPAddr, error).
4 lookupGroup singleflight.Group
```

第二个是 Go 在查询仓库版本信息时，将并发的请求合并成 1 个请求：

[复制代码](#)

```
1 func metaImportsForPrefix(importPrefix string, mod ModuleMode, security web.Se
2     // 使用缓存保存请求结果
3     setCache := func(res fetchResult) (fetchResult, error) {
4         fetchCacheMu.Lock()
5         defer fetchCacheMu.Unlock()
6         fetchCache[importPrefix] = res
7         return res, nil
8     }
9     // 使用 SingleFlight 请求
10    resi, _, _ := fetchGroup.Do(importPrefix, func() (resi interface{}), err er
11    fetchCacheMu.Lock()
12    // 如果缓存中有数据，那么直接从缓存中取
13    if res, ok := fetchCache[importPrefix]; ok {
14        fetchCacheMu.Unlock()
15        return res, nil
16    }
17    fetchCacheMu.Unlock()
18    .....
```

需要注意的是，这里涉及到了缓存的问题。上面的代码会把结果放在缓存中，这也是常用的一种解决缓存击穿的例子。

设计缓存问题时，我们常常需要解决缓存穿透、缓存雪崩和缓存击穿问题。缓存击穿问题是指，在平常高并发的系统中，大量的请求同时查询一个 key 时，如果这个 key 正好过期失效了，就会导致大量的请求都打到数据库上。这就是缓存击穿。



用 SingleFlight 来解决缓存击穿问题再合适不过了。因为，这个时候，只要这些对同一个 key 的并发请求的其中一个到数据库中查询，就可以了，这些并发的请求可以共享同一个结果。因为是缓存查询，不用考虑幂等性问题。

事实上，在 Go 生态圈知名的缓存框架 groupcache 中，就使用了较早的 Go 标准库的 SingleFlight 实现。接下来，我就来给你介绍一下 groupcache 是如何使用 SingleFlight 解决缓存击穿问题的。

groupcache 中的 SingleFlight 只有一个方法：

[复制代码](#)

```
1 func (g *Group) Do(key string, fn func() (interface{}, error)) (interface{}, e
```

SingleFlight 的作用是，在加载一个缓存项的时候，合并对同一个 key 的 load 的并发请求：

[复制代码](#)

```
1 type Group struct {
2     .....
3     // loadGroup ensures that each key is only fetched once
4     // (either locally or remotely), regardless of the number of
5     // concurrent callers.
6     loadGroup flightGroup
7     .....
8 }
9
10 func (g *Group) load(ctx context.Context, key string, dest Sink) (value By
11 viewi, err := g.loadGroup.Do(key, func() (interface{}, error) {
12     // 从cache, peer, local尝试查询cache
13     return value, nil
14 })
15 if err == nil {
16     value = viewi.(ByteView)
17 }
18 return
19 }
```

其它的知名项目如 Cockroachdb (小强数据库)、CoreDNS (DNS 服务器) 等都有 SingleFlight 应用，你可以查看这些项目的代码，加深对 SingleFlight 的理解。

总结来说, 使用 SingleFlight 时, 可以通过合并请求的方式降低对下游服务的并发压力, 从而提高系统的性能, 常常用于缓存系统中。最后, 我想给你留一个思考题, 你觉得, SingleFlight 能不能合并并发的写操作呢?

## 循环栅栏 CyclicBarrier

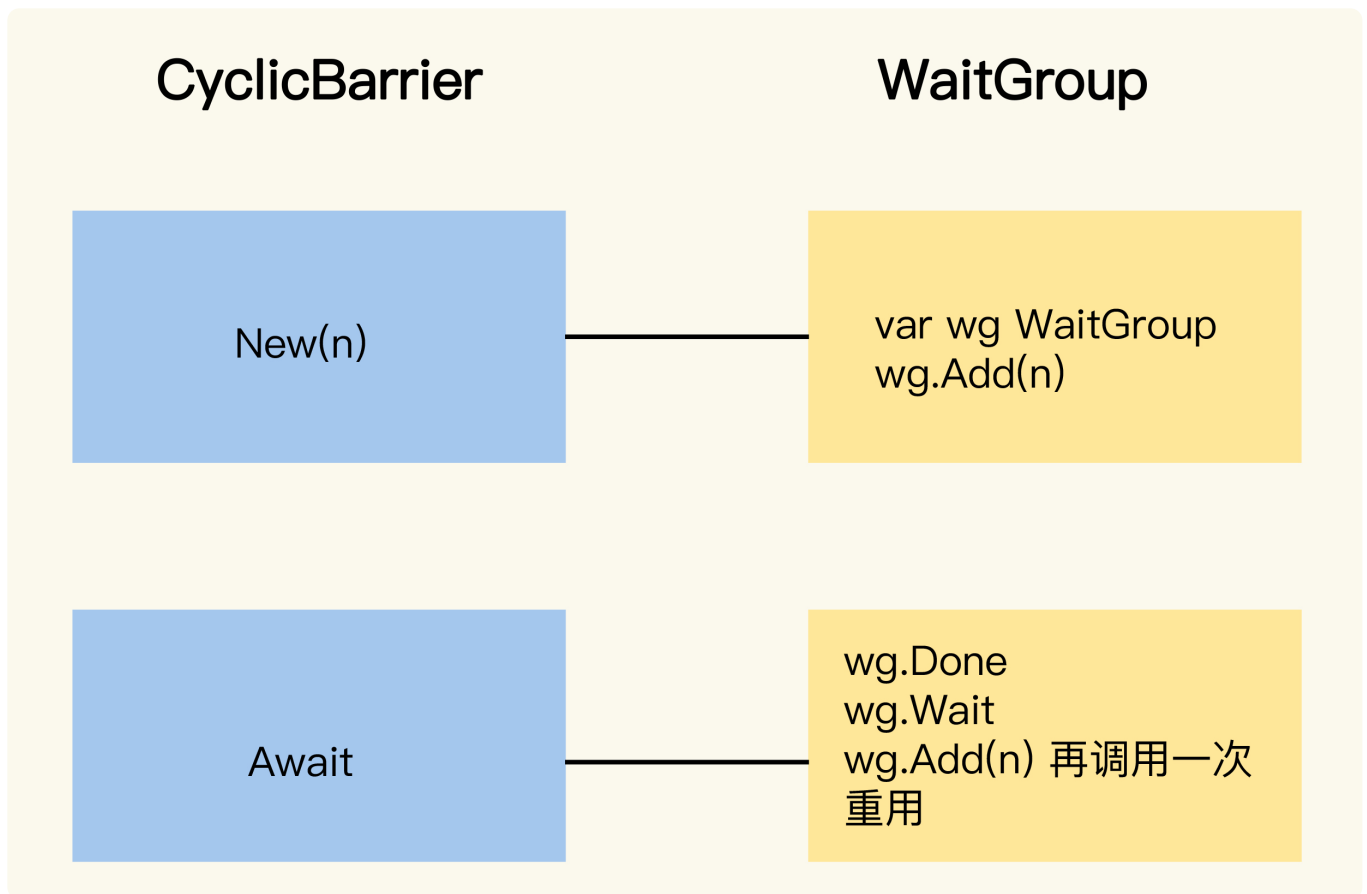
接下来, 我再给你介绍另外一个并发原语: 循环栅栏 (CyclicBarrier), 它常常应用于重复进行一组 goroutine 同时执行的场景中。

🔗CyclicBarrier 允许一组 goroutine 彼此等待, 到达一个共同的执行点。同时, 因为它可以被重复使用, 所以叫循环栅栏。具体的机制是, 大家都在栅栏前等待, 等全部都到齐了, 就抬起栅栏放行。

事实上, 这个 CyclicBarrier 是参考 🔗Java CyclicBarrier 和 🔗C# Barrier 的功能实现的。Java 提供了 CountdownLatch (倒计时器) 和 CyclicBarrier (循环栅栏) 两个类似的用于保证多线程到达同一个执行点的类, 只不过前者是到达 0 的时候放行, 后者是到达某个指定的数的时候放行。C# Barrier 功能也是类似的, 你可以查看链接, 了解它的具体用法。

你可能会觉得, CyclicBarrier 和 WaitGroup 的功能有点类似, 确实是这样。不过, CyclicBarrier 更适合用在“固定数量的 goroutine 等待同一个执行点”的场景中, 而且在放行 goroutine 之后, CyclicBarrier 可以重复利用, 不像 WaitGroup 重用的时候, 必须小心翼翼避免 panic。

处理可重用的多 goroutine 等待同一个执行点的场景的时候, CyclicBarrier 和 WaitGroup 方法调用的对应关系如下:



可以看到，如果使用 WaitGroup 实现的话，调用比较复杂，不像 CyclicBarrier 那么清爽。更重要的是，如果想重用 WaitGroup，你还要保证，将 WaitGroup 的计数值重置到 n 的时候不会出现并发问题。

WaitGroup 更适合用在“一个 goroutine 等待一组 goroutine 到达同一个执行点”的场景中，或者是不需要重用的场景中。

好了，了解了 CyclicBarrier 的应用场景和功能，下面我们来学习下它的具体实现。


## 实现原理

CyclicBarrier 有两个初始化方法：

1. 第一个是 New 方法，它只需要一个参数，来指定循环栅栏参与者的数量；
2. 第二个方法是 NewWithAction，它额外提供一个函数，可以在每一次到达执行点的时候执行一次。具体的时间点是在最后一个参与者到达之后，但是其它的参与者还未被放行之前。我们可以利用它，做放行之前的一些共享状态的更新等操作。


这两个方法的签名如下：



 复制代码

```
1 func New(parties int) CyclicBarrier
2 func NewWithAction(parties int, barrierAction func() error) CyclicBarrier
```

CyclicBarrier 是一个接口，定义的方法如下：

 复制代码

```
1 type CyclicBarrier interface {
2     // 等待所有的参与者到达，如果被ctx.Done()中断，会返回ErrBrokenBarrier
3     Await(ctx context.Context) error
4
5     // 重置循环栅栏到初始化状态。如果当前有等待者，那么它们会返回ErrBrokenBarrier
6     Reset()
7
8     // 返回当前等待者的数量
9     GetNumberWaiting() int
10
11     // 参与者的数量
12     GetParties() int
13
14     // 循环栅栏是否处于中断状态
15     IsBroken() bool
16 }
```

循环栅栏的使用也很简单。循环栅栏的参与者只需调用 Await 等待，等所有的参与者都到达后，再执行下一步。当执行下一步的时候，循环栅栏的状态又恢复到初始的状态了，可以迎接下一轮同样多的参与者。

有一道非常经典的并发编程的题目，非常适合使用循环栅栏，下面我们来看一下。

## 并发趣题：一氧化二氢制造工厂

题目是这样的：

有一个名叫大自然的搬运工的工厂，生产一种叫做一氧化二氢的神秘液体。这种液体的分子是由一个氧原子和两个氢原子组成的，也就是水。

这个工厂有多条生产线，每条生产线负责生产氧原子或者是氢原子，每条生产线由一个 goroutine 负责。

这些生产线会通过一个栅栏，只有一个氧原子生产线和两个氢原子生产线都准备好，才能生成出一个水分子，否则所有的生产线都会处于等待状态。也就是说，一个水分子必须由三个不同的生产线提供原子，而且水分子是一个一个按照顺序产生的，每生产一个水分子，就会打印出 HHO、HOH、OHH 三种形式的其中一种。HHH、OOH、OHO、HOO、OOO 都是不允许的。


生产线中氢原子的生产线为  $2N$  条，氧原子的生产线为  $N$  条。

你可以先想一下，我们怎么来实现呢？

首先，我们来定义一个 H2O 辅助数据类型，它包含两个信号量的字段和一个循环栅栏。

1. semaH 信号量：控制氢原子。一个水分子需要两个氢原子，所以，氢原子的空槽数资源数设置为 2。
2. semaO 信号量：控制氧原子。一个水分子需要一个氧原子，所以资源数的空槽数设置为 1。
3. 循环栅栏：等待两个氢原子和一个氧原子填补空槽，直到任务完成。

我们来看下具体的代码：

 复制代码

```
1 package water
2 import (
3     "context"
4     "github.com/marusama/cyclicbarrier"
5     "golang.org/x/sync/semaphore"
6 )
7 // 定义水分子合成的辅助数据结构
8 type H2O struct {
9     semaH *semaphore.Weighted // 氢原子的信号量
10    semaO *semaphore.Weighted // 氧原子的信号量
11    b      cyclicbarrier.CyclicBarrier // 循环栅栏，用来控制合成
12 }
13 func New() *H2O {
14     return &H2O{
15         semaH: semaphore.NewWeighted(2), // 氢原子需要两个
16         semaO: semaphore.NewWeighted(1), // 氧原子需要一个
17         b:      cyclicbarrier.New(3),    // 需要三个原子才能合成
18     }
19 }
```

接下来，我们看看各条流水线的处理情况。

流水线分为氢原子处理流水线和氧原子处理流水线，首先，我们先看一下氢原子的流水线：如果有可用的空槽，氢原子的流水线的处理方法是 `hydrogen`，`hydrogen` 方法就会占用一个空槽 (`h2o.semaH.Acquire`)，输出一个 H 字符，然后等待栅栏放行。等其它的 goroutine 填补了氢原子的另一个空槽和氧原子的空槽之后，程序才可以继续进行。

[复制代码](#)

```
1 func (h2o *H2O) hydrogen(releaseHydrogen func()) {
2     h2o.semaH.Acquire(context.Background(), 1)
3
4     releaseHydrogen() // 输出H
5     h2o.b.Await(context.Background()) //等待栅栏放行
6     h2o.semaH.Release(1) // 释放氢原子空槽
7 }
```

然后是氧原子的流水线。氧原子的流水线处理方法是 `oxygen`，`oxygen` 方法是等待氧原子的空槽，然后输出一个 O，就等待栅栏放行。放行后，释放氧原子空槽位。

[复制代码](#)

```
1 func (h2o *H2O) oxygen(releaseOxygen func()) {
2     h2o.semaO.Acquire(context.Background(), 1)
3
4     releaseOxygen() // 输出O
5     h2o.b.Await(context.Background()) //等待栅栏放行
6     h2o.semaO.Release(1) // 释放氢原子空槽
7 }
```

在栅栏放行之前，只有两个氢原子的空槽位和一个氧原子的空槽位。只有等栅栏放行之后，这些空槽位才会被释放。栅栏放行，就意味着一个水分子组成成功。

这个算法是不是正确呢？我们来编写一个单元测试检测一下。

[复制代码](#)

```
1 package water
2
3
```


```
4 import (
5     "math/rand"
6     "sort"
7     "sync"
8     "testing"
9     "time"
10 )
11
12
13 func TestWaterFactory(t *testing.T) {
14     //用来存放水分子结果的channel
15     var ch chan string
16     releaseHydrogen := func() {
17         ch <- "H"
18     }
19     releaseOxygen := func() {
20         ch <- "O"
21     }
22
23     // 300个原子, 300个goroutine,每个goroutine并发的产生一个原子
24     var N = 100
25     ch = make(chan string, N*3)
26
27
28     h2o := New()
29
30     // 用来等待所有的goroutine完成
31     var wg sync.WaitGroup
32     wg.Add(N * 3)
33
34     // 200个氢原子goroutine
35     for i := 0; i < 2*N; i++ {
36         go func() {
37             time.Sleep(time.Duration(rand.Intn(100)) * time.Millisecond)
38             h2o.hydrogen(releaseHydrogen)
39             wg.Done()
40         }()
41     }
42     // 100个氧原子goroutine
43     for i := 0; i < N; i++ {
44         go func() {
45             time.Sleep(time.Duration(rand.Intn(100)) * time.Millisecond)
46             h2o.oxygen(releaseOxygen)
47             wg.Done()
48         }()
49     }
50
51     //等待所有的goroutine执行完
52     wg.Wait()
53
54     // 结果中肯定是300个原子
55     if len(ch) != N*3 {
```

```
56         t.Fatalf("expect %d atom but got %d", N*3, len(ch))
57     }
58
59     // 每三个原子一组, 分别进行检查。要求这一组原子中必须包含两个氢原子和一个氧原子, 这样才能
60     var s = make([]string, 3)
61     for i := 0; i < N; i++ {
62         s[0] = <-ch
63         s[1] = <-ch
64         s[2] = <-ch
65         sort.Strings(s)
66
67
68         water := s[0] + s[1] + s[2]
69         if water != "HHO" {
70             t.Fatalf("expect a water molecule but got %s", water)
71         }
72     }
73 }
```

## 总结

每一个并发原语都有它存在的道理, 也都有它应用的场景。

如果你没有学习 CyclicBarrier, 你可能只会想到, 用 WaitGroup 来实现这个水分子制造工厂的例子。

 复制代码

```
1  type H2O struct {
2      semaH *semaphore.Weighted
3      semaO *semaphore.Weighted
4      wg    sync.WaitGroup //将循环栅栏替换成WaitGroup
5  }
6
7  func New() *H2O {
8      var wg sync.WaitGroup
9      wg.Add(3)
10
11     return &H2O{
12         semaH: semaphore.NewWeighted(2),
13         semaO: semaphore.NewWeighted(1),
14         wg:    wg,
15     }
16 }
17
18
19 func (h2o *H2O) hydrogen(releaseHydrogen func()) {
20     h2o.semaH.Acquire(context.Background(), 1)
```

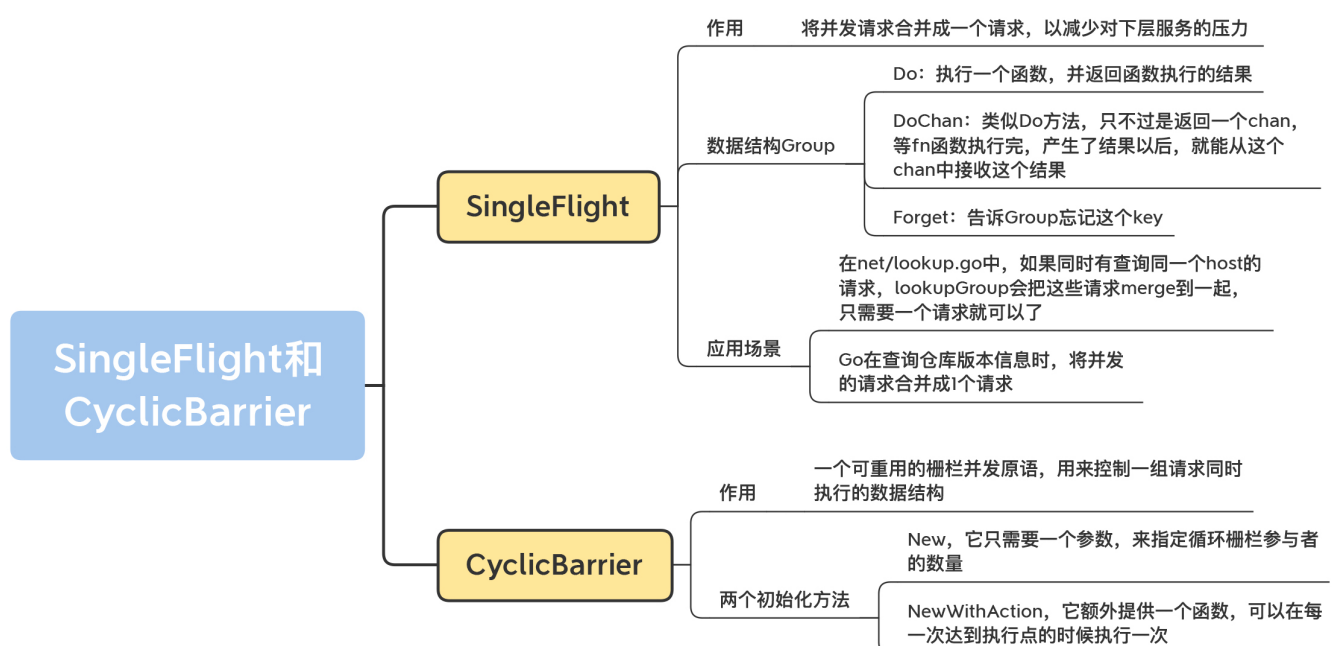
```

21     releaseHydrogen()
22
23     // 标记自己已达到, 等待其它goroutine到达
24     h2o.wg.Done()
25     h2o.wg.Wait()
26
27     h2o.semaH.Release(1)
28 }
29
30 func (h2o *H2O) oxygen(releaseOxygen func()) {
31     h2o.sema0.Acquire(context.Background(), 1)
32     releaseOxygen()
33
34     // 标记自己已达到, 等待其它goroutine到达
35     h2o.wg.Done()
36     h2o.wg.Wait()
37     //都到达后重置wg
38     h2o.wg.Add(3)
39
40     h2o.sema0.Release(1)
41 }

```

你一看代码就知道了, 使用 WaitGroup 非常复杂, 而且, 重用和 Done 方法的调用有并发的问題, 程序可能 panic, 远远没有使用循环栅栏更加简单直接。

所以, 我建议你多了解一些并发原语, 甚至是从其它编程语言、操作系统中学习更多的并发原语, 这样可以让你知识库更加丰富, 在面对并发场景的时候, 你也能更加游刃有余。





## 思考题

如果大自然的搬运工工厂生产的液体是双氧水（双氧水分子是两个氢原子和两个氧原子），你又该怎么实现呢？

欢迎在留言区写下你的思考和答案，我们一起交流讨论。如果你觉得有所收获，也欢迎你把今天的内容分享给你的朋友或同事。

提建议

# Go 并发编程实战课

## 鸟窝带你攻克并发编程难题

晁岳攀 (鸟窝)

前微博技术专家

知名微服务框架 rpcx 的作者



新版升级：点击「🔗 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 16 | Semaphore：一篇文章搞懂信号量

下一篇 18 | 分组操作：处理一组子任务，该用什么并发原语？

## 精选留言 (7)

[写留言](#)**杜鑫**

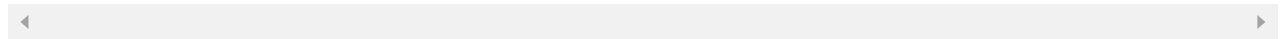
2020-11-19

谢谢老师的文章！第一次知道有这两种并发原语，不管之后工作能不能用到，这都是对个人眼界的开阔！

ps:老师的文章几乎都是一篇顶两篇，刚订阅的时候还担心课程内容太少，现在一看担心完全是多余的，这段时间读了老师的文章之后，个人对go并发知识了解更深刻了，谢谢老师！

[展开](#)

作者回复: 加油!



3

**虫子樱桃**

2020-11-19

写了singleFlight的例子辅助思考。

```
package main
```

```
import (  
    "log"...
```

[展开](#)

1

**那一刻**

2020-11-18

SingleFlight 能不能合并并发的写操作呢？

我觉得得分情况讨论，如果多个写请求是对于同一个对象相同的写操作，比如把某条记录的一个字段设置为某一个值，这样的话可以合并。

如果写操作是对于对象增减操作，涉及幂等行操作不太合适合并。

[展开](#)

1

**伟伟**

2020-11-23

```
package main
```

```
import (
```

"context"

...

展开

作者回复:

1



**chapin**  
2020-11-21

这一节在实际项目中都直接用到。

展开

1



**Linuxer**  
2020-11-20

感觉自己Go刚刚入门，我确实也才学习一周左右，看着挺爽，写起来还是不顺手，还得多练习，老师能否给我们这些打算转Go的新手一些建议，谢谢

展开

作者回复: 多实践，在项目中锻炼，很快就顺手了



**橙子888**  
2020-11-18

打卡。

展开