



下载APP



## 06 | WaitGroup: 协同等待, 任务编排利器

2020-10-23 晁岳攀

Go 并发编程实战课

[进入课程 >](#)**讲述: 安晓辉**

时长 20:31 大小 18.80M



你好, 我是鸟窝。

WaitGroup, 我们以前都多多少少学习过, 或者是使用过。其实, WaitGroup 很简单, 就是 package sync 用来做任务编排的一个并发原语。它要解决的就是并发 - 等待的问题: 现在有一个 goroutine A 在检查点 (checkpoint) 等待一组 goroutine 全部完成, 如果在执行任务的这些 goroutine 还没全部完成, 那么 goroutine A 就会阻塞在检查点, 直到所有 goroutine 都完成后才能继续执行。

我们来看一个使用 WaitGroup 的场景。



比如, 我们要完成一个大的任务, 需要使用并行的 goroutine 执行三个小任务, 只有这三个小任务都完成, 我们才能去执行后面的任务。如果通过轮询的方式定时询问三个小任务

是否完成, 会存在两个问题: 一是, 性能比较低, 因为三个小任务可能早就完成了, 却要等很长时间才被轮询到; 二是, 会有很多无谓的轮询, 空耗 CPU 资源。

那么, 这个时候使用 WaitGroup 并发原语就比较有效了, 它可以阻塞等待的 goroutine。等到三个小任务都完成了, 再即时唤醒它们。

其实, 很多操作系统和编程语言都提供了类似的并发原语。比如, Linux 中的 barrier、Pthread (POSIX 线程) 中的 barrier、C++ 中的 std::barrier、Java 中的 CyclicBarrier 和 CountDownLatch 等。由此可见, 这个并发原语还是一个非常基础的并发类型。所以, 我们要认真掌握今天的内容, 这样就可以举一反三, 轻松应对其他场景下的需求了。

我们还是从 WaitGroup 的基本用法学起吧。

## WaitGroup 的基本用法

Go 标准库中的 WaitGroup 提供了三个方法, 保持了 Go 简洁的风格。

```
1 func (wg *WaitGroup) Add(delta int)
2 func (wg *WaitGroup) Done()
3 func (wg *WaitGroup) Wait()
```

[复制代码](#)

我们分别看下这三个方法:

Add, 用来设置 WaitGroup 的计数值;

Done, 用来将 WaitGroup 的计数值减 1, 其实就是调用了 Add(-1);

Wait, 调用这个方法 goroutine 会一直阻塞, 直到 WaitGroup 的计数值变为 0。

接下来, 我们通过一个使用 WaitGroup 的例子, 来看下 Add、Done、Wait 方法的基本用法。

在这个例子中, 我们使用了以前实现的计数器 struct。我们启动了 10 个 worker, 分别对计数值加一, 10 个 worker 都完成后, 我们期望输出计数器的值。

```
1 // 线程安全的计数器
2 type Counter struct {
3     mu    sync.Mutex
4     count uint64
5 }
6 // 对计数值加一
7 func (c *Counter) Incr() {
8     c.mu.Lock()
9     c.count++
10    c.mu.Unlock()
11 }
12 // 获取当前的计数值
13 func (c *Counter) Count() uint64 {
14     c.mu.Lock()
15     defer c.mu.Unlock()
16     return c.count
17 }
18 // sleep 1秒, 然后计数值加1
19 func worker(c *Counter, wg *sync.WaitGroup) {
20     defer wg.Done()
21     time.Sleep(time.Second)
22     c.Incr()
23 }
24
25 func main() {
26     var counter Counter
27
28     var wg sync.WaitGroup
29     wg.Add(10) // WaitGroup的值设置为10
30
31     for i := 0; i < 10; i++ { // 启动10个goroutine执行加1任务
32         go worker(&counter, &wg)
33     }
34     // 检查点, 等待goroutine都完成任务
35     wg.Wait()
36     // 输出当前计数器的值
37     fmt.Println(counter.Count())
38 }
```

我们一起来分析下这段代码。

第 28 行, 声明了一个 WaitGroup 变量, 初始值为零。

第 29 行, 把 WaitGroup 变量的计数值设置为 10。因为我们需要编排 10 个 goroutine(worker) 去执行任务, 并且等待 goroutine 完成。

第 35 行, 调用 Wait 方法阻塞等待。

第 32 行, 启动了 goroutine, 并把我們定义的 WaitGroup 指针当作参数传递进去。goroutine 完成后, 需要调用 Done 方法, 把 WaitGroup 的计数值减 1。等 10 个 goroutine 都调用了 Done 方法后, WaitGroup 的计数值降为 0, 这时, 第 35 行的主 goroutine 就不再阻塞, 会继续执行, 在第 37 行输出计数值。

这就是我們使用 WaitGroup 编排这类任务的常用方式。而“这类任务”指的就是, 需要启动多个 goroutine 执行任务, 主 goroutine 需要等待子 goroutine 都完成后才继续执行。

熟悉了 WaitGroup 的基本用法后, 我们再看看它具体是如何实现的吧。


## WaitGroup 的实现

首先, 我们看看 WaitGroup 的数据结构。它包括了一个 noCopy 的辅助字段, 一个 state1 记录 WaitGroup 状态的数组。

noCopy 的辅助字段, 主要就是辅助 vet 工具检查是否通过 copy 赋值这个 WaitGroup 实例。我会在后面和你详细分析这个字段;

state1, 一个具有复合意义的字段, 包含 WaitGroup 的计数、阻塞在检查点的 waiter 数和信号量。

WaitGroup 的数据结构定义以及 state 信息的获取方法如下:

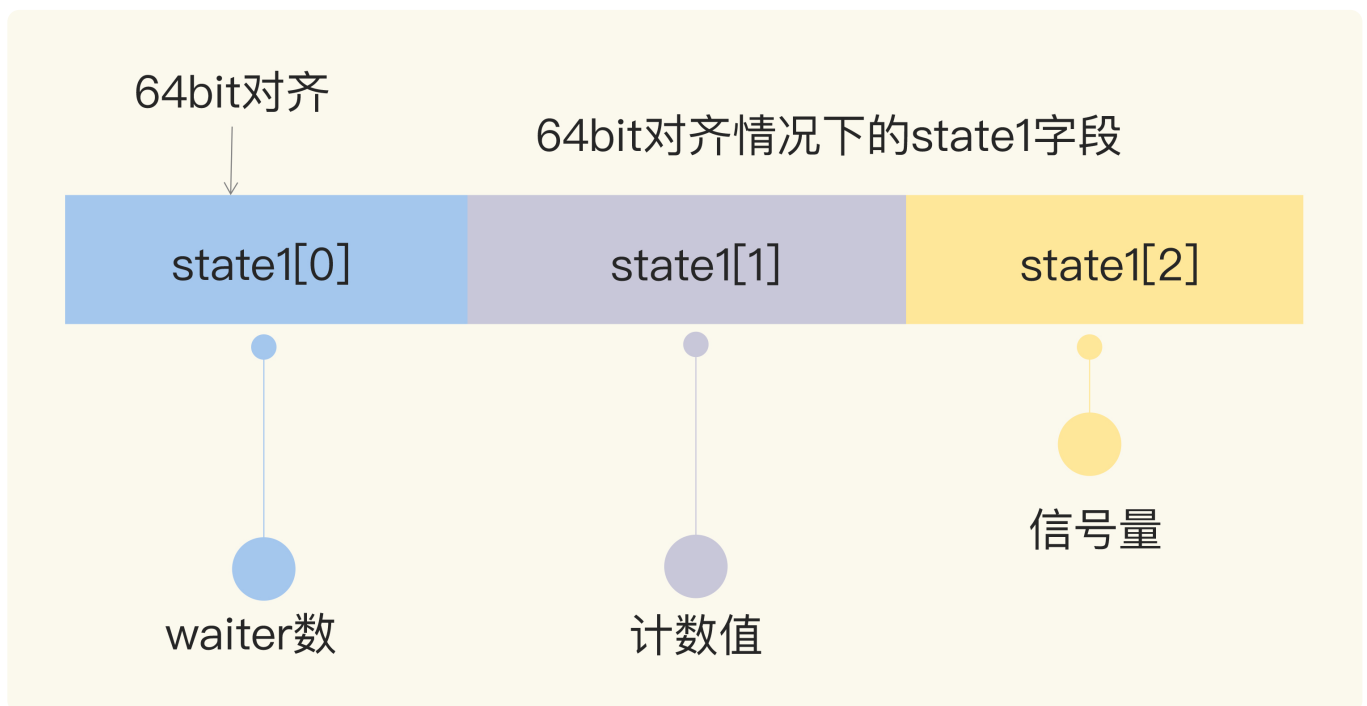
 复制代码

```
1 type WaitGroup struct {
2     // 避免复制使用的一个技巧, 可以告诉vet工具违反了复制使用的规则
3     noCopy noCopy
4     // 64bit(8bytes)的值分成两段, 高32bit是计数值, 低32bit是waiter的计数
5     // 另外32bit是用作信号量的
6     // 因为64bit值的原子操作需要64bit对齐, 但是32bit编译器不支持, 所以数组中的元素在不同的
7     // 总之, 会找到对齐的那64bit作为state, 其余的32bit做信号量
8     state1 [3]uint32
9 }
10
11
12 // 得到state的地址和信号量的地址
13 func (wg *WaitGroup) state() (statep *uint64, semap *uint32) {
14     if uintptr(unsafe.Pointer(&wg.state1))%8 == 0 {
15         // 如果地址是64bit对齐的, 数组前两个元素做state, 后一个元素做信号量
16         return (*uint64)(unsafe.Pointer(&wg.state1)), &wg.state1[2]
```

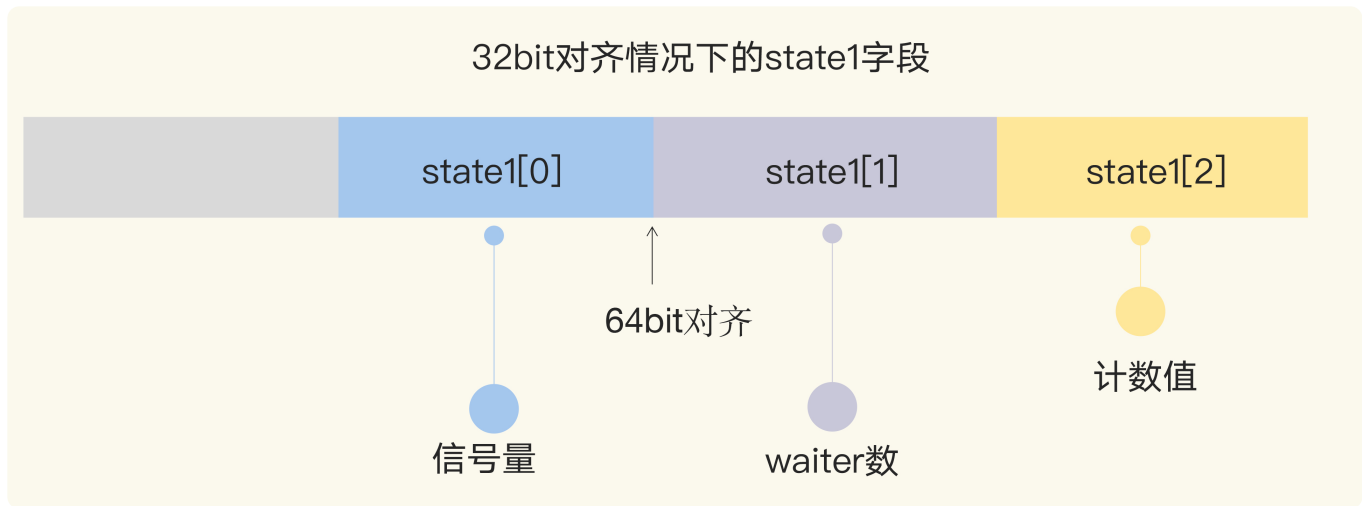
```
17     } else {  
18         // 如果地址是32bit对齐的, 数组后两个元素用来做state, 它可以用来做64bit的原子操作  
19         return (*uint64)(unsafe.Pointer(&wg.state1[1])), &wg.state1[0]  
20     }  
21 }
```

因为对 64 位整数的原子操作要求整数的地址是 64 位对齐的, 所以针对 64 位和 32 位环境的 state 字段的组成是不一样的。

在 64 位环境下, state1 的第一个元素是 waiter 数, 第二个元素是 WaitGroup 的计数值, 第三个元素是信号量。



在 32 位环境下, 如果 state1 不是 64 位对齐的地址, 那么 state1 的第一个元素是信号量, 后两个元素分别是 waiter 数和计数值。



然后，我们继续深入源码，看一下 Add、Done 和 Wait 这三个方法的实现。

在查看这部分源码实现时，我们会发现，除了这些方法本身的实现外，还会有一些额外的代码，主要是 race 检查和异常检查的代码。其中，有几个检查非常关键，如果检查不通过，会出现 panic，这部分内容我会在下一小节分析 WaitGroup 的错误使用场景时介绍。现在，我们先专注在 Add、Wait 和 Done 本身的实现代码上。

我先为你梳理下 **Add 方法的逻辑**。Add 方法主要操作的是 state 的计数部分。你可以为计数值增加一个 delta 值，内部通过原子操作把这个值加到计数值上。需要注意的是，这个 delta 也可以是个负数，相当于为计数值减去一个值，Done 方法内部其实就是通过 Add(-1) 实现的。

它的实现代码如下：

复制代码

```
1 func (wg *WaitGroup) Add(delta int) {
2     statep, semap := wg.state()
3     // 高32bit是计数值v，所以把delta左移32，增加到计数上
4     state := atomic.AddUint64(statep, uint64(delta)<<32)
5     v := int32(state >> 32) // 当前计数值
6     w := uint32(state) // waiter count
7
8     if v > 0 || w == 0 {
9         return
10    }
11
12    // 如果计数值v为0并且waiter的数量w不为0，那么state的值就是waiter的数量
13    // 将waiter的数量设置为0，因为计数值v也是0，所以它们俩的组合*statep直接设置为0即可。此
14    *statep = 0
15    for ; w != 0; w-- {
```



```
16     runtime_Semrelease(semaph, false, 0)
17 }
18 }
19
20
21 // Done方法实际就是计数器减1
22 func (wg *WaitGroup) Done() {
23     wg.Add(-1)
24 }
```

Wait 方法的实现逻辑是：不断检查 state 的值。如果其中的计数值变为了 0，那么说明所有的任务已完成，调用者不必再等待，直接返回。如果计数值大于 0，说明此时还有任务没完成，那么调用者就变成了等待者，需要加入 waiter 队列，并且阻塞住自己。

其主干实现代码如下：

[复制代码](#)

```
1 func (wg *WaitGroup) Wait() {
2     statep, semaph := wg.state()
3
4     for {
5         state := atomic.LoadUint64(statep)
6         v := int32(state >> 32) // 当前计数值
7         w := uint32(state) // waiter的数量
8         if v == 0 {
9             // 如果计数值为0，调用这个方法的goroutine不必再等待，继续执行它后面的逻辑即可
10            return
11        }
12        // 否则把waiter数量加1。期间可能有并发调用Wait的情况，所以最外层使用了一个for循环
13        if atomic.CompareAndSwapUint64(statep, state, state+1) {
14            // 阻塞休眠等待
15            runtime_Semacquire(semaph)
16            // 被唤醒，不再阻塞，返回
17            return
18        }
19    }
20 }
```

## 使用 WaitGroup 时的常见错误

在分析 WaitGroup 的 Add、Done 和 Wait 方法的实现的时候，为避免干扰，我删除了异常检查的代码。但是，这些异常检查非常有用。

我们在开发的时候, 经常会遇见或看到误用 WaitGroup 的场景, 究其原因就是没有弄明白这些检查的逻辑。所以接下来, 我们就通过几个小例子, 一起学习下在开发时绝对要避免的 3 个问题。


## 常见问题一: 计数器设置为负值

WaitGroup 的计数器的值必须大于等于 0。我们在更改这个计数值的时候, WaitGroup 会先做检查, 如果计数值被设置为负数, 就会导致 panic。

一般情况下, 有两种方法会导致计数器设置为负数。

第一种方法是: **调用 Add 的时候传递一个负数**。如果你能保证当前的计数器加上这个负数后还是大于等于 0 的话, 也没有问题, 否则就会导致 panic。

比如下面这段代码, 计数器的初始值为 10, 当第一次传入 -10 的时候, 计数值被设置为 0, 不会有啥问题。但是, 再紧接着传入 -1 以后, 计数值就被设置为负数了, 程序就会出现 panic。

 复制代码

```
1 func main() {  
2     var wg sync.WaitGroup  
3     wg.Add(10)  
4  
5     wg.Add(-10) //将-10作为参数调用Add, 计数值被设置为0  
6  
7     wg.Add(-1) //将-1作为参数调用Add, 如果加上-1计数值就会变为负数。这是不对的, 所以会触发  
8 }
```

第二个方法是: **调用 Done 方法的次数过多, 超过了 WaitGroup 的计数值**。

**使用 WaitGroup 的正确姿势是, 预先确定好 WaitGroup 的计数值, 然后调用相同次数的 Done 完成相应的任务**。比如, 在 WaitGroup 变量声明之后, 就立即设置它的计数值, 或者在 goroutine 启动之前增加 1, 然后在 goroutine 中调用 Done。

如果你没有遵循这些规则, 就很可能导致 Done 方法调用的次数和计数值不一致, 进而造成死锁 (Done 调用次数比计数值少) 或者 panic (Done 调用次数比计数值多)。



比如下面这个例子中, 多调用了一次 Done 方法后, 会导致计数值为负, 所以程序运行到这一行会出现 panic。

[复制代码](#)

```
1 func main() {
2     var wg sync.WaitGroup
3     wg.Add(1)
4
5     wg.Done()
6
7     wg.Done()
8 }
```

## 常见问题二: 不期望的 Add 时机

在使用 WaitGroup 的时候, 你一定要遵循的原则就是, **等所有的 Add 方法调用之后再调用 Wait**, 否则就可能导致 panic 或者不期望的结果。


我们构造这样一个场景: 只有部分的 Add/Done 执行完后, Wait 就返回。我们看一个例子: 启动四个 goroutine, 每个 goroutine 内部调用 Add(1) 然后调用 Done(), 主 goroutine 调用 Wait 等待任务完成。

[复制代码](#)

```
1 func main() {
2     var wg sync.WaitGroup
3     go dosomething(100, &wg) // 启动第一个goroutine
4     go dosomething(110, &wg) // 启动第二个goroutine
5     go dosomething(120, &wg) // 启动第三个goroutine
6     go dosomething(130, &wg) // 启动第四个goroutine
7
8     wg.Wait() // 主goroutine等待完成
9     fmt.Println("Done")
10 }
11
12 func dosomething(millisecs time.Duration, wg *sync.WaitGroup) {
13     duration := millisecs * time.Millisecond
14     time.Sleep(duration) // 故意sleep一段时间
15
16     wg.Add(1)
17     fmt.Println("后台执行, duration:", duration)
18     wg.Done()
19 }
```


在这个例子中, 我们原本设想的是, 等四个 goroutine 都执行完毕后输出 Done 的信息, 但是它的错误之处在于, 将 WaitGroup.Add 方法的调用放在了子 goroutine 中。等主 goroutine 调用 Wait 的时候, 因为四个任务 goroutine 一开始都休眠, 所以可能 WaitGroup 的 Add 方法还没有被调用, WaitGroup 的计数还是 0, 所以它并没有等待四个子 goroutine 执行完毕才继续执行, 而是立刻执行了下一步。

导致这个错误的原因是, 没有遵循先完成所有的 Add 之后才 Wait。要解决这个问题, 一个方法是, 预先设置计数值:

 复制代码

```
1 func main() {
2     var wg sync.WaitGroup
3     wg.Add(4) // 预先设定WaitGroup的计数值
4
5     go dosomething(100, &wg) // 启动第一个goroutine
6     go dosomething(110, &wg) // 启动第二个goroutine
7     go dosomething(120, &wg) // 启动第三个goroutine
8     go dosomething(130, &wg) // 启动第四个goroutine
9
10    wg.Wait() // 主goroutine等待
11    fmt.Println("Done")
12 }
13
14 func dosomething(millisecs time.Duration, wg *sync.WaitGroup) {
15     duration := millisecs * time.Millisecond
16     time.Sleep(duration)
17
18     fmt.Println("后台执行, duration:", duration)
19     wg.Done()
20 }
21
```

另一种方法是在启动子 goroutine 之前才调用 Add:

 复制代码

```
1 func main() {
2     var wg sync.WaitGroup
3
4     dosomething(100, &wg) // 调用方法, 把计数值加1, 并启动任务goroutine
5     dosomething(110, &wg) // 调用方法, 把计数值加1, 并启动任务goroutine
6     dosomething(120, &wg) // 调用方法, 把计数值加1, 并启动任务goroutine
7     dosomething(130, &wg) // 调用方法, 把计数值加1, 并启动任务goroutine
8
9 }
```

```
9     wg.Wait() // 主goroutine等待, 代码逻辑保证了四次Add(1)都已经执行完了
10     fmt.Println("Done")
11 }
12
13 func dosomething(millisecs time.Duration, wg *sync.WaitGroup) {
14     wg.Add(1) // 计数值加1, 再启动goroutine
15
16     go func() {
17         duration := millisecs * time.Millisecond
18         time.Sleep(duration)
19         fmt.Println("后台执行, duration:", duration)
20         wg.Done()
21     }()
22 }
23
```

可见, 无论是怎么修复, 都要保证所有的 Add 方法是在 Wait 方法之前被调用的。

### 常见问题三: 前一个 Wait 还没结束就重用 WaitGroup

“前一个 Wait 还没结束就重用 WaitGroup” 这一点似乎不太好理解, 我借用田径比赛的例子和你解释下吧。在田径比赛的百米小组赛中, 需要把选手分成几组, 一组选手比赛完之后, 就可以进行下一组了。为了确保两组比赛时间上没有冲突, 我们在模型化这个场景的时候, 可以使用 WaitGroup。

WaitGroup 等一组比赛的所有选手都跑完后 5 分钟, 才开始下一组比赛。下一组比赛还可以使用这个 WaitGroup 来控制, 因为 **WaitGroup 是可以重用的**。只要 WaitGroup 的计数值恢复到零值的状态, 那么它就可以被看作是新创建的 WaitGroup, 被重复使用。

但是, 如果我们在 WaitGroup 的计数值还没有恢复到零值的时候就重用, 就会导致程序 panic。我们看一个例子, 初始设置 WaitGroup 的计数值为 1, 启动一个 goroutine 先调用 Done 方法, 接着就调用 Add 方法, Add 方法有可能和主 goroutine 并发执行。

 复制代码

```
1 func main() {
2     var wg sync.WaitGroup
3     wg.Add(1)
4     go func() {
5         time.Sleep(time.Millisecond)
6         wg.Done() // 计数器减1
7         wg.Add(1) // 计数值加1
8     }()
9     wg.Wait() // 主goroutine等待, 有可能和第7行并发执行
```

```
10 }
```

在这个例子中, 第 6 行虽然让 WaitGroup 的计数恢复到 0, 但是因为第 9 行有个 waiter 在等待, 如果等待 Wait 的 goroutine, 刚被唤醒就和 Add 调用 (第 7 行) 有并发执行的冲突, 所以就会出现 panic。

总结一下: WaitGroup 虽然可以重用, 但是是有一个前提的, 那就是必须等到上一轮的 Wait 完成之后, 才能重用 WaitGroup 执行下一轮的 Add/Wait, 如果你在 Wait 还没执行完的时候就调用下一轮 Add 方法, 就有可能出现 panic。

## noCopy: 辅助 vet 检查

我们刚刚在学习 WaitGroup 的数据结构时, 提到了里面有一个 noCopy 字段。你还记得它的作用吗? 其实, 它就是指示 vet 工具在做检查的时候, 这个数据结构不能做值复制使用。更严谨地说, 是不能在第一次使用之后复制使用 (must not be copied after first use)。

你可能会说了, 为什么要把 noCopy 字段单独拿出来讲呢? 一方面, 把 noCopy 字段穿插到 waitgroup 代码中讲解, 容易干扰我们对 WaitGroup 整体的理解。另一方面, 也是非常重要的原因, noCopy 是一个通用的计数技术, 其他并发原语中也会用到, 所以单独介绍有助于你以后在实践中使用这个技术。

我们在 [第 3 讲](#) 学习 Mutex 的时候用到了 vet 工具。vet 会对实现 Locker 接口的数据类型做静态检查, 一旦代码中有复制使用这种数据类型的情况, 就会发出警告。但是, WaitGroup 同步原语不就是 Add、Done 和 Wait 方法吗? vet 能检查出来吗?

其实是可以的。通过给 WaitGroup 添加一个 noCopy 字段, 我们就可以为 WaitGroup 实现 Locker 接口, 这样 vet 工具就可以做复制检查了。而且因为 noCopy 字段是未输出类型, 所以 WaitGroup 不会暴露 Lock/Unlock 方法。

noCopy 字段的类型是 noCopy, 它只是一个辅助的、用来帮助 vet 检查用的类型:

```
1 type noCopy struct{}  
2
```

[复制代码](#)

```
3 // Lock is a no-op used by -copylocks checker from `go vet`.
4 func (*noCopy) Lock() {}
5 func (*noCopy) Unlock() {}
6
```

如果你想要自己定义的数据结构不被复制使用, 或者说, 不能通过 `vet` 工具检查出复制使用的报警, 就可以通过嵌入 `noCopy` 这个数据类型来实现。

## 流行的 Go 开发项目中的坑

接下来又到了喝枸杞红枣茶的时间了。你可以稍微休息一下, 心态放轻松地跟我一起围观下知名项目犯过的错, 比如 `copy Waitgroup`、`Add/Wait` 并发执行问题、遗漏 `Add` 等 Bug。

有网友在 Go 的 [issue 28123](#) 中提了以下的例子, 你能发现这段代码有什么问题吗?

[复制代码](#)

```
1 type TestStruct struct {
2     Wait sync.WaitGroup
3 }
4
5 func main() {
6     w := sync.WaitGroup{}
7     w.Add(1)
8     t := &TestStruct{
9         Wait: w,
10    }
11
12    t.Wait.Done()
13    fmt.Println("Finished")
14 }
```

这段代码最大的一个问题, 就是第 9 行 `copy` 了 `WaitGroup` 的实例 `w`。虽然这段代码能执行成功, 但确实是违反了 `WaitGroup` 使用之后不要复制的规则。在项目中, 我们可以通过 `vet` 工具检查出这样的错误。

Docker [issue 28161](#) 和 [issue 27011](#), 都是因为在重用 `WaitGroup` 的时候, 没等前一次的 `Wait` 结束就 `Add` 导致的错误。Etcd [issue 6534](#) 也是重用 `WaitGroup` 的 Bug, 没有等前一个 `Wait` 结束就 `Add`。

Kubernetes [issue 59574](#) 的 Bug 是忘记 Wait 之前增加计数了, 这就属于我们通常认为几乎不可能出现的 Bug。

```

1 test/e2e/scalability/density.go
363 // Stop apiserver CPU profile gatherer and gather memory allocations profile.
364 close(profileGathererStopCh)
365 wg := sync.WaitGroup{}
366 wg.Add(1)
367 framework.GatherApiserverMemoryProfile(&wg, "density")
368 wg.Wait()
369

```

```

1 test/e2e/scalability/load.go
@@ -104,6 +104,7 @@ var _ = SIGDescribe("Load capacity", func() {
104 // Stop apiserver CPU profile gatherer and gather memory allocations profile.
105 close(profileGathererStopCh)
106 wg := sync.WaitGroup{}
107 wg.Add(1)
108 framework.GatherApiserverMemoryProfile(&wg, "load")
109 wg.Wait()
110

```

即使是开发 Go 语言的开发者自己, 在使用 WaitGroup 的时候, 也可能会犯错。比如 [issue 12813](#), 因为 defer 的使用, Add 方法可能在 Done 之后才执行, 导致计数负值的 panic。

```

2 cmd/coordinator/coordinator.go
@@ -2033,7 +2033,7 @@ func (st *buildStatus) runTests(helpers <-chan *buildlet.Client) (remoteErr, err
2033 go func() {
2034 defer buildletActivity.Done() // for the per-goroutine Add(2) above
2035 for helper := range helpers {
2036 defer buildletActivity.Add(1)
2037 buildletActivity.Add(1)
2038 go func(bc *buildlet.Client) {
2039 defer buildletActivity.Done() // for the per-helper Add(1) above
2040 defer st.logEventTime("closed_helper", bc.Name())

```

## 总结

学完这一讲, 我们知道了使用 WaitGroup 容易犯的错, 是不是有些手脚被束缚的感觉呢? 其实大可不必, 只要我们不是特别复杂地使用 WaitGroup, 就不用有啥心理负担。



而关于如何避免错误使用 WaitGroup 的情况, 我们只需要尽量保证下面 5 点就可以了:

不重用 WaitGroup。新建一个 WaitGroup 不会带来多大的资源开销, 重用反而更容易出错。

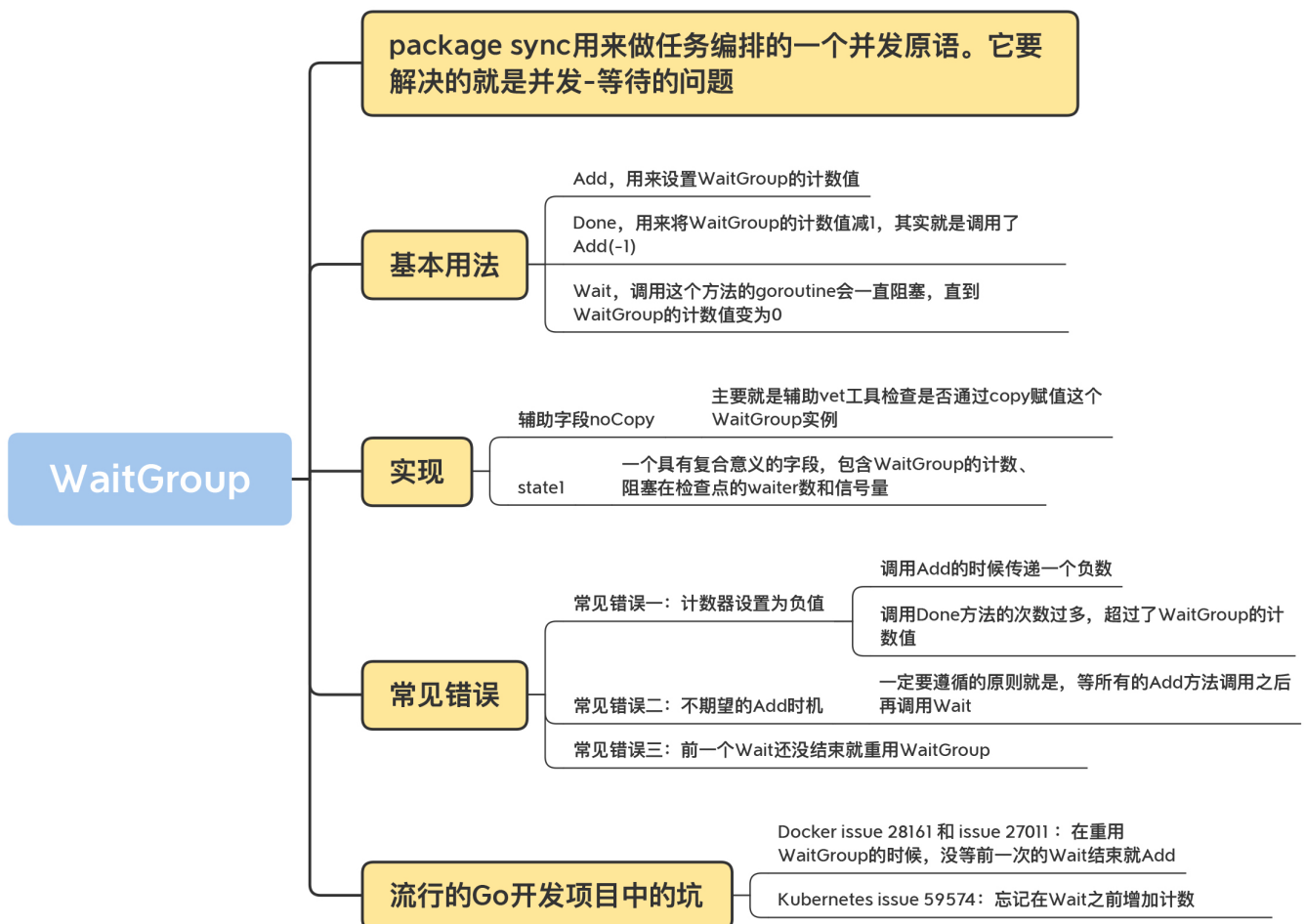
保证所有的 Add 方法调用都在 Wait 之前。

不传递负数给 Add 方法, 只通过 Done 来给计数值减 1。

不做多余的 Done 方法调用, 保证 Add 的计数值和 Done 方法调用的数量是一样的。

不遗漏 Done 方法的调用, 否则会导致 Wait hang 住无法返回。

这一讲我们详细学习了 WaitGroup 的相关知识, 这里我整理了一份关于 WaitGroup 的知识地图, 方便你复习。



## 思考题

通常我们可以把 WaitGroup 的计数值, 理解为等待要完成的 waiter 的数量。你可以试着扩展下 WaitGroup, 来查询 WaitGroup 的当前的计数值吗?

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 05 | RWMutex: 读写锁的实现原理及避坑指南

下一篇 07 | Cond: 条件变量的实现机制及避坑指南

#### 精选留言 (12)



那一刻

写留言



2020-10-23

关于64位对齐, 这个帖子讲的也不错, 分享一下。 <https://go101.org/article/memory-layout.html>



2

**橙子888**

2020-10-23

issue 12813 按照 defer 后进先出的原则, Done 一定会在 Add 之前执行吧, 为啥是“可能”呢?

作者回复: Done是在另外的goroutine执行的。保证不了先后顺序



1

1

**Junes**

2020-10-23

思路: 主要难点在于32位和64位时的处理, 具体逻辑就参考sync.WaiterGroup源码中的state()方法, 跟老师的示例图互相映证。

实现:

```
func getStateAndWait(wgp *sync.WaitGroup) (uint32, uint32) {...
```

展开 ▾



1

**moooofly**

2020-10-26

没理解错的话, waiter 数量对应的应该是调用 Wait() 的 goroutine 的数量吧, 文中的示例代码都只是在 main goroutine 中调用一次, 所以 waiter 数量都只是 1, 没错吧

作者回复: 对

**linxs**

2020-10-26

为什么32bit系统的处理上, state1的元素排列和64bit的不同呢

64bit : waiter,counter,sem

32bit : sem,waiter,counter

作者回复: 因为32bit上是32bit的对齐的, state1  
地址不一定正好是8byte对齐

 1

**约书亚**

2020-10-25

如果有一个goroutine先调用了runtime\_Semrelease, 之后另一个goroutine调用runtime\_Semacquire, 应该不会阻塞而是立即返回吧? 考虑到如下场景: Wait方法假如执行完13行, 此时Done方法整个执行完成, 之后Wait方法执行15行。

展开 ∨



**地下城勇士**

2020-10-23

「可以阻塞等待的 goroutine。等到三个小任务都完成了, 再即时唤醒它们」

---

老师, 没有理解这句话是什么意思? 什么叫阻塞等待? 还有完成了, 为什么还要唤醒?

展开 ∨

作者回复: 就是这个goroutine被设置成waiting,让渡出M。相当于进入休眠状态, 需要别人唤醒它, 它才能继续工作



**锋**

2020-10-23

老师好, 有两个疑问, 谢谢。

1.wg.Add(-1)

这个方法是在Done中调用的。但是我没太理解 -1的时候, 是怎么减去的。我在Add中的代码中看到...

展开 ∨

作者回复: 1补码, 看godoc

2复制一份代码, 重用



**橙子888**

2020-10-23

先打卡, 然后慢慢消化.....

展开 ▾

**oCupJS**

2020-10-23

打卡

展开 ▾

**那一刻**

2020-10-23

请问老师, waitgroup里state1里提到对 64 位整数的原子操作要求整数的地址是 64 位对齐的, 不是很理解, 能否提供例子说明下呢?

另外在32位对齐的图例里, 也有个64位对齐指示, 是什么含义呢?

展开 ▾

作者回复: 1.那个原子操作要求64位对齐

2.那段逻辑是如果发现state1不是64对齐的, 那么可以推断跳过32位后就是64位对齐了



1

**新味道**

2020-10-23

// 阻塞休眠等待

runtime\_Semacquire(semaph)

-----

没理解『阻塞休眠等待』的意思, 能否再详细讲一下。

展开 ▾

作者回复: 这个是运行时的实现, 用来阻塞当前goroutine. 它会把当前g放入队列, 标记成waitin g,让渡m



