从 bug 中学习：六大开源项目告诉你 go 并发编程的那些坑

理查的天空 腾讯技术工程 2021-03-08 18:00

作者：richardyao，腾讯 CSIG 后台开发工程师

> 并发编程中，go 不仅仅支持传统的通过共享内存的方式来通信，更推崇通过channel来传递消息，这种新的并发编程模型会出现不同于以往的bug。从 bug 中学习，《Understanding Real-World Concurrency Bugs in Go》这篇 paper 在分析了六大开源项目并发相关的bug之后，为我们总结了go并发编程中常见的坑。别往坑里跳，编程更美妙。

在 go 中，创建 goroutine 非常简单，在函数调用前加 go 关键字，这个函数的调用就在一个单独的 goroutine 中执行了；go 支持匿名函数，让创建 goroutine 的操作更加简洁。另外，在并发编程模型上，go 不仅仅支持传统的通过共享内存的方式来通信，更推崇通过 channel 来传递消息：

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

这种新的并发编程模型会带来新类型的 bug，从 bug 中学习，《Understanding Real-World Concurrency Bugs in Go》这篇 paper 在 Docker、Kubernetes、etcd、gRPC、CockroachDB、BoltDB 六大开源项目的 commit log 中搜索等关键字：

"race"、"deadlock"、"synchronization"、"concurrency"、"lock"、"mutex"、"atomic"、"compete"、"context"、"once"、"goroutine leak"，找出这六大项目中并发相关的 bug，然后归类这些 bug，总结出了 go 并发编程中常见的一些坑。通过学习这些坑，可以让我们在以后的项目里防范类似的错误，或者遇到类似问题的时候可以帮助指导快速定位排查。

unbuffered channel 由于 receiver 退出导致 sender 侧 block

如下面一个 bug 的例子：

```
func finishReq(timeout time.Duration) ob {
```

```
    ch := make(chan ob)
    go func() {
        result := fn()
        ch <- result // block
    }()
    select {
    case result = <-ch:
        return result
    case <-time.After(timeout):
        return nil
    }
}
```

本意是想调用 fn()时加上超时的功能，如果 fn()在超时时间没有返回，则返回 nil。但是当超时发生的时候，针对代码中第二行创建的 ch 来说，由于已经没有 receiver 了，第 5 行将会被 block 住，导致这个 goroutine 永远不会退出。

> If the capacity is zero or absent, the channel is unbuffered and communication succeeds only when both a sender and receiver are ready. Otherwise, the channel is buffered and communication succeeds without blocking if the buffer is not full (sends) or not empty (receives).

这个 bug 的修复方式也是非常的简单，把 unbuffered channel 修改成 buffered channel。

```
func finishReq(timeout time.Duration) ob {
    ch := make(chan ob, 1)
    go func() {
        result := fn()
        ch <- result // block
    }()
    select {
    case result = <-ch:
        return result
    case <-time.After(timeout):
        return nil
    }
}
```

> 思考：在上面的例子中，虽然这样不会 block 了，但是 channel 一直没有被关闭，channel 保持不关闭是否会导致资源的泄漏呢？

WaitGroup 误用导致阻塞

下面是一个 WaitGroup 误用导致阻塞的一个 bug 的例子：
https://github.com/moby/moby/pull/25384

```
var group sync.WaitGroup
group.Add(len(pm.plugins))
for _, p := range pm.plugins {
    go func(p *plugin) {
        defer group.Done()
    }(p)
    group.Wait()
}
```

当 len(pm.plugins)大于等于 2 时，第 7 行将会被卡住，因为这个时候只启动了一个异步的 goroutine，group.Done()只会被调用一次，group.Wait()将会永久阻塞。修复如下：

```
var group sync.WaitGroup
group.Add(len(pm.plugins))
for _, p := range pm.plugins {
    go func(p *plugin) {
        defer group.Done()
    }(p)
}
group.Wait()
```

context 误用导致资源泄漏

如下面的代码所示：

```
hctx, hcancel := context.WithCancel(ctx)
if timeout > 0 {
    hctx, hcancel = context.WithTimeout(ctx, timeout)
}
```

第一行 context.WithCancel(ctx)有可能会创建一个 goroutine，来等待 ctx 是否 Done，如果 parent 的 ctx.Done()的话，cancel 掉 child 的 context。也就是说 hcancel 绑定了一定的资源，不能直接覆盖。

> Canceling this context releases resources associated with it, so code should call cancel as soon as the operations running in this Context complete.

这个 bug 的修复方式是：

```
var hctx context.Context
var hcancel context.CancelFunc
if timeout > 0 {
    hctx, hcancel = context.WithTimeout(ctx, timeout)
} else {
    hctx, hcancel = context.WithCancel(ctx)
}
```

或者

```
hctx, hcancel := context.WithCancel(ctx)
if timeout > 0 {
    hcancel.Cancel()
    hctx, hcancel = context.WithTimeout(ctx, timeout)
}
```

多个 goroutine 同时读写共享变量导致的 bug

如下面的例子：

```
for i := 17; i <= 21; i++ { // write
    go func() { /* Create a new goroutine */
        apiVersion := fmt.Sprintf("v1.%d", i) // read
    }()
}
```

第二行中的匿名函数形成了一个闭包(closures)，在闭包内部可以访问定义在外面的变量，如上面的例子中，第 1 行在写 i 这个变量，在第 3 行在读 i 这个变量。这里的关键的问题是对同一个变量的读写是在两个 goroutine 里面同时进行的，因此是不安全的。

> Function literals are closures: they may refer to variables defined in a surrounding function. Those variables are then shared between the surrounding function and the function literal, and they survive as long as they are accessible.

可以修改成：

```
for i := 17; i <= 21; i++ { // write
    go func(i int) { /* Create a new goroutine */
        apiVersion := fmt.Sprintf("v1.%d", i) // read
    }(i)
}
```

通过 passed by value 的方式规避了并发读写的问题。

channel 被关闭多次引发的 bug

https://github.com/moby/moby/pull/24007/files

```
select {
case <-c.closed:
default:
    close(c.closed)
}
```

上面这块代码可能会被多个 goroutine 同时执行，这段代码的逻辑是，case 这个分支判断 closed 这个 channel 是否被关闭了，如果被关闭的话，就什么都不做；如果 closed 没有被关闭的话，就执行 default 分支关闭这个 channel，多个 goroutine 并发执行的时候，有可能会导致 closed 这个 channel 被关闭多次。

> For a channel c, the built-in function close(c) records that no more values will be sent on the channel. It is an error if c is a receive-only channel. Sending to or closing a closed channel causes a run-time panic.

这个 bug 的修复方式是：

```
Once.Do(func() {
    close(c.closed)
})
```

把整个 select 语句块换成 Once.Do，保证 channel 只关闭一次。

timer 误用产生的 bug

如下面的例子：

```
timer := time.NewTimer(0)
if dur > 0 {
    timer = time.NewTimer(dur)
}
select {
case <-timer.C:
case <-ctx.Done():
    return nil
}
```

原意是想 dur 大于 0 的时候，设置 timer 超时时间，但是 timer := time.NewTimer(0)导致 timer.C 立即触发。修复后：

```
var timeout <-chan time.Time
if dur > 0 {
    timeout = time.NewTimer(dur).C
}
select {
case <-timeout:
case <-ctx.Done():
    return nil
}
```

> A nil channel is never ready for communication.

上面的代码中第一个 case 分支 timeout 有可能是个 nil 的 channel，select 在 nil 的 channel 上，这个分支不会被触发，因此不会有问题。

读写锁误用引发的 bug

go 语言中的 RWMutex，write lock 有更高的优先级：

> If a goroutine holds a RWMutex for reading and another goroutine might call Lock, no goroutine should expect to be able to acquire a read lock until the initial read lock is released. In particular, this prohibits recursive read locking. This is to ensure that the lock eventually becomes available; a blocked Lock call excludes new readers from acquiring the lock.

如果一个 goroutine 拿到了一个 read lock，然后另外一个 goroutine 调用了 Lock，第一个 goroutine 再调用 read lock 的时候会死锁，应予以避免。

参考资料

- https://songlh.github.io/paper/go-study.pdf
- https://golang.org/ref/spec
- https://golang.org/doc/effective_go.html
- https://golang.org/pkg/