Golang channel 三大坑，你踩过了嘛?

Go语言中文网 2022-06-02 08:52 发表于北京

以下文章来源于翔叔架构笔记 ，作者陈翔宇

翔叔架构笔记

专注 Golang 后端开发与架构，分享优质内容与独立音乐。关于翔叔：鹅厂长大的 T10...

忙活了几个月，终于尘埃落定，最近可以好好更文了。

1. 前言

在使用 channel 进行 goroutine 之间的通信时，有时候场面会变得十分复杂，以至于写出难以觉察、难以定位的偶现 bug，而且上线的时候往往跑得好好的，直到某一天深夜收到服务挂了、OOM 了之类的告警……

本文来梳理一下使用 channel 中常见的三大坑：panic、死锁、内存泄漏，做到防患于未然。

2. 死锁

go 语言新手在编译时很容易碰到这个死锁的问题：

```
fatal error: all goroutines are asleep - deadlock!
```

这个就是喜闻乐见的「死锁」了…… 在操作系统中，我们学过，「死锁」就是两个线程互相等待，耗在那里，最后程序不得不终止。go 语言中的「死锁」也是类似的，两个 goroutine 互相等待，导致程序耗在那里，无法继续跑下去。看了很多死锁的案例后，channel 导致的死锁可以归纳为以下几类案例（先讨论 **unbuffered channel** 的情况）：

2.1 只有生产者，没有消费者，或者反过来

channel 的生产者和消费者**必须成对出现**，如果缺乏一个，就会造成死锁，例如：

```
// 只有生产者，没有消费者
```

```
func f1() {
    ch := make(chan int)
    ch <- 1
}
```

或是：

```
// 只有消费者，没有生产者
func f2() {
    ch := make(chan int)
    <-ch
}
```

2.2 生产者和消费者出现在同一个 goroutine 中

除了需要成对出现，还需要出现在不同的 goroutine 中，例如：

```
// 同一个 goroutine 中同时出现生产者和消费者
func f3() {
    ch := make(chan int)
    ch <- 1  // 由于消费者还没执行到，这里会一直阻塞住
    <-ch
}
```

对于 buffered channel 则是：

2.3 buffered channel 已满，且出现上述情况

buffered channel 会将收到的元素先存在 `hchan` 结构体的 `ringbuffer` 中，继而才会发生阻塞。而当发生阻塞时，如果阻塞了主 goroutine，则也会出现死锁

所以实际使用中，推荐尽量使用 buffered channel，使用起来会更安全，在下文的「内存泄漏」相关内容也会提及

3. 内存泄漏

内存泄漏一般都是通过 `OOM(Out of Memory)` 告警或者发布过程中对内存的观察发现的，服务内存往往都是缓慢上升，直到被系统 OOM 掉清空内存再周而复始

在 go 语言中，错误地使用 channel 会导致 goroutine 泄漏，进而导致内存泄漏。

3.1 如何实现 goroutine 泄漏呢?

不会修 bug，我还不会写 bug 吗？让 goroutine 泄漏的核心就是：

> **生产者/消费者 所在的 goroutine 已经退出，而其对应的 消费者/生产者 所在的 goroutine 会永远阻塞住，直到进程退出**

3.2 生产者阻塞导致泄漏

我们一般会用 channel 来做一些超时控制，例如下面这个例子：

```
func leak1() {
    ch := make(chan int)
    // g1
    go func() {
        time.Sleep(2 * time.Second) // 模拟 io 操作
        ch <- 100                   // 模拟返回结果
    }()

    // g2
    // 阻塞住，直到超时或返回
    select {
    case <-time.After(500 * time.Millisecond):
        fmt.Println("timeout! exit...")
    case result := <-ch:
        fmt.Printf("result: %d\n", result)
    }
}
```

这里我们用 goroutine `g1` 来模拟 io 操作，主 goroutine `g2` 来模拟客户端的处理逻辑，

1. 假设客户端超时为 500ms，而实际请求耗时为 2s，则 select 会走到 timeout 的逻辑，这时 `g2` 退出，channel `ch` 没有消费者，会一直在等待状态，输出如下：

```
Goroutine num: 1
timeout! exit...
Goroutine num: 2
```

如果这是在 server 代码中，这个请求处理完后， `g1` 就会挂起、发生泄漏了，就等着 OOM 吧 =。=

2. 假设客户端超时调整为 5000ms，实际请求耗时 2s，则 select 会进入获取 result 的分支，输出如下：

```
Goroutine num: 1
result: 100
Goroutine num: 1
```

3.3 消费者阻塞导致泄漏

如果生产者不继续生产，消费者所在的 goroutine 也会阻塞住，不会退出，例如：

```
func leak2() {
    ch := make(chan int)

    // 消费者 g1
    go func() {
        for result := range ch {
            fmt.Printf("result: %d\n", result)
        }
    }()

    // 生产者 g2
    ch <- 1
    ch <- 2
    time.Sleep(time.Second)  // 模拟耗时
    fmt.Println("main goroutine g2 done...")
}
```

这种情况下，只需要增加 `close(ch)` 的操作即可， `for-range` 操作在收到 close 的信号后会退出、goroutine 不再阻塞，能够被回收。

3.4 如何预防内存泄漏?

预防 goroutine 泄漏的核心就是：

> **创建 goroutine 时就要想清楚它什么时候被回收**

具体到执行层面，包括：

- 当 goroutine 退出时，需要考虑它使用的 channel 有没有可能阻塞对应的生产者、消费者的 goroutine
- 尽量使用 `buffered channel` 使用 `buffered channel` 能减少阻塞发生、即使疏忽了一些极端情况，也能降低 goroutine 泄漏的概率

4. panic

panic 就更刺激了，一般是测试的时候没发现，上线之后偶现，程序挂掉，服务出现一个超时毛刺后触发告警。channel 导致的 panic 一般是以下几个原因：

4.1 向已经 close 掉的 channel 继续发送数据

先举一个简单的栗子：

```
func p1() {
    ch := make(chan int, 1)
    close(ch)
    ch <- 1
}
// panic: send on closed channel
```

在实际开发过程中，处理多个 goroutine 之间协作时，可能存在一个 goroutine 已经 close 掉 channel 了，另外一个不知道，也去 close 一下，就会 panic 掉，例如：

```
func p1() {
    ch := make(chan int, 1)
    done := make(chan struct{}, 1)
```

```
    go func() {
        <- time.After(2*time.Second)
        println("close2")
        close(ch)
        close(done)
    }()
    go func() {
        <- time.After(1*time.Second)
        println("close1")
        ch <- 1
        close(ch)
    }()

    <-done
}
```

万恶之源就是在 go 语言里，你是无法知道一个 channel 是否已经被 close 掉的，所以在尝试做 close 操作的时候，就应该做好会 panic 的准备……

4.2 多次 close 同一个 channel

同上，在尝试往 channel 里发送数据时，就应该考虑

- 这个 channel 已经关了吗?
- 这个 channel 什么时候、在哪个 goroutine 里关呢?
- 谁来关呢? 还是干脆不关?

5. 如何优雅地 close channel

5.1 我们需要检查 channel 是否关闭吗?

刚遇到上面说的 panic 问题时，我也试过去找一个内置的 `closed` 函数来检查关闭状态，结果发现，并没有这样一个函数……

那么，如果有这样的函数，真能彻底解决 panic 的问题么? 答案是不能。因为 channel 是在一个并发的环境下去做收发操作，就算当前执行 `closed(ch)` 得到的结果是 false，还是不能直接去关，例如如下 yy 出来的代码:

```
if !closed(ch) {  // 返回 false
```

```
    // 在这中间出了幺蛾子！
    close(ch)  // 还是 panic 了……
}
```

遵循 less is more 的原则，这个 `closed` 函数是要不得了

5.2 需要 close 吗？为什么？

> 结论：除非**必须**关闭 chan，否则不要主动关闭。关闭 chan 最优雅的方式，就是不要关闭 chan~

当一个 chan 没有 sender 和 receiver 时，即不再被使用时，GC 会在一段时间后标记、清理掉这个 chan。那么什么时候必须关闭 chan 呢？比较常见的是将 close 作为一种通知机制，尤其是生产者与消费者之间是 1:M 的关系时，通过 close 告诉下游：我收工了，你们别读了。

5.3 谁来关？

chan 关闭的原则：

> 1. Don't close a channel from the receiver side 不要在消费者端关闭 chan
> 2. Don't close a channel if the channel has multiple concurrent senders 有多个并发写的生产者时也别关

只要我们遵循这两条原则，就能避免两种 panic 的场景，即：向 closed chan 发送数据，或者是 close 一个 closed chan。

按照生产者和消费者的关系可以拆解成以下几类情况：

1. 一写一读：生产者关闭即可
2. 一写多读：生产者关闭即可，关闭时下游全部消费者都能收到通知
3. 多写一读：多个生产者之间需要引入一个协调 channel 来处理信号
4. 多写多读：与 3 类似，核心思路是引入一个中间层以及使用 `try-send` 的套路来处理非阻塞的写入，例如：

```
func main() {
```

```
rand.Seed(time.Now().UnixNano())
log.SetFlags(0)

const Max = 100000
const NumReceivers = 10
const NumSenders = 1000

wgReceivers := sync.WaitGroup{}
wgReceivers.Add(NumReceivers)

dataCh := make(chan int)
stopCh := make(chan struct{})
    // stopCh 是额外引入的一个信号 channel.
    // 它的生产者是下面的 toStop channel,
    // 消费者是上面 dataCh 的生产者和消费者
toStop := make(chan string, 1)
    // toStop 是拿来关闭 stopCh 用的，由 dataCh 的生产者和消费者写入
    // 由下面的匿名中介函数(moderator)消费
    // 要注意，这个一定要是 buffered channel （否则没法用 try-send 来处理了）

var stoppedBy string

// moderator
go func() {
    stoppedBy = <-toStop
    close(stopCh)
}()

// senders
for i := 0; i < NumSenders; i++ {
    go func(id string) {
        for {
            value := rand.Intn(Max)
            if value == 0 {
                // try-send 操作
                // 如果 toStop 满了，就会走 default 分支啥也不干，也不会阻塞
                select {
                case toStop <- "sender#" + id:
                default:
                }
                return
            }
```

```
                // try-receive 操作，尽快退出
                // 如果没有这一步，下面的 select 操作可能造成 panic
                select {
                case <- stopCh:
                    return
                default:
                }

                // 如果尝试从 stopCh 取数据的同时，也尝试向 dataCh
                // 写数据，则会命中 select 的伪随机逻辑，可能会写入数据
                select {
                case <- stopCh:
                    return
                case dataCh <- value:
                }
            }
        }(strconv.Itoa(i))
    }

    // receivers
    for i := 0; i < NumReceivers; i++ {
        go func(id string) {
            defer wgReceivers.Done()

            for {
                // 同上
                select {
                case <- stopCh:
                    return
                default:
                }

                // 尝试读数据
                select {
                case <- stopCh:
                    return
                case value := <-dataCh:
                    if value == Max-1 {
                        select {
                        case toStop <- "receiver#" + id:
                        default:
                        }
                        return
```

```
                    }

                    log.Println(value)
                }
            }
        }(strconv.Itoa(i))
    }

    wgReceivers.Wait()
    log.Println("stopped by", stoppedBy)
}
```

本用例来自参考资料中的《How to Gracefully Close Channels》，go 101 系列非常不错~

参考资料

1. Golang channel 死锁的几种情况以及例子
2. 老手也常误用！详解 Go channel 内存泄漏问题
3. 深入解析 Goroutine 泄露的场景：channel 发送者
4. How to Gracefully Close Channels

推荐阅读

- 多图详解Go中的Channel源码

福利

我为大家整理了一份从入门到进阶的Go学习资料礼包，包含学习建议：入门看什么，进阶看什么。关注公众号 「polarisxu」，回复 **ebook** 获取；还可以回复「**进群**」，和数万 Gopher 交流学习。