链滴

Go 笔记之如何防止 goroutine 泄露

作者：xiaoxiezaijia

原文链接：https://ld246.com/article/1564975144459

来源网站：链滴

今天简单谈谈，Go 如何防止 goroutine 泄露。

概述

Go 的并发模型与其他语言不同，虽说它简化了并发程序的开发难度，但如果不了解使用方法，常常遇到 goroutine 泄露的问题。虽然 goroutine 是轻量级的线程，占用资源很少，但如果一直得不到放并且还在不断创建新协程，毫无疑问是有问题的，并且是要在程序运行几天，甚至更长的时间才能现的问题。

对于上面描述的问题，我觉得可以从两方面入手解决，如下：

一是预防，要做到预防，我们就需要了解什么样的代码会产生泄露，以及了解如何写出正确的代码；

二是监控，虽说预防减少了泄露产生的概率，但没有人敢说自己不犯错，因而，通常我们还需要一些控手段进一步保证程序的健壮性；

接下来，我将会分两篇文章分别从这两个角度进行介绍，今天先谈第一点。

如何监控泄露

本文主要集中在第一点上，但为了更好的演示效果，可以先介绍一个最简单的监控方式。通过 runtime NumGoroutine() 获取当前运行中的 goroutine 数量，通过它确认是否发生泄漏。它的使用非常简，就不为它专门写个例子了。

一个简单的例子

语言级别的并发支持是 Go 的一大优势，但这个优势也很容易被滥用。通常我们在开始 Go 并发学习，常常听别人说，Go 的并发非常简单，在调用函数前加上 go 关键词便可启动 goroutine，即一个发单元，但很多人可能只听到了这句话，然后就出现了类似下面的代码：

```
package main

import (
    "fmt"
    "runtime"
    "time"
)

func sayHello() {
    for {
        fmt.Println("Hello gorotine")
        time.Sleep(time.Second)
    }
}

func main() {
    defer func() {
        fmt.Println("the number of goroutines: ", runtime.NumGoroutine())
    }()

    go sayHello()
    fmt.Println("Hello main")
}
```

对 Go 比较熟悉的话，很容易发现这段代码的问题，sayHello 是个死循环，没有如何退出机制，因此就没有任何办法释放创建的 goroutine。我们通过在 main 函数最前面的 defer 实现在函数退出时打

当前运行中的 goroutine 数量，毫无意外，它的输出如下：

```
the number of goroutines: 2
```

不过，因为上面的程序并非常驻，有泄露问题也不大，程序退出后系统会自动回收运行时资源。但如这段代码在常驻服务中执行，比如 http server，每接收到一个请求，便会启动一次 sayHello，时间逝，每次启动的 goroutine 都得不到释放，你的服务将会离奔溃越来越近。

这个例子比较简单，我相信，对 Go 的并发稍微有点了解的朋友都不会犯这个错。

泄露情况分类

前面介绍的例子由于在 goroutine 运行死循环导致的泄露。接下来，我会按照并发的数据同步方式对露的各种情况进行分析。简单可归于两类，即：

channel 导致的泄露

传统同步机制导致的泄露

传统同步机制主要指面向共享内存的同步机制，比如排它锁、共享锁等。这两种情况导致的泄露还是较常见的。go 由于 defer 的存在，第二类情况，一般情况下还是比较容易避免的。

chanel 引起的泄露

先说 channel，如果之前读过官方的那篇并发的文章，翻译版，你会发现 channel 的使用，一个不小就泄露了。我们来具体总结下那些情况下可能导致。

发送不接收

我们知道，发送者一般都会配有相应的接收者。理想情况下，我们希望接收者总能接收完所有发送的据，这样就不会有任何问题。但现实是，一旦接收者发生异常退出，停止继续接收上游数据，发送者会被阻塞。这个情况在 前面说的文章 中有非常细致的介绍。

示例代码：

```
package main

import "time"

func gen(nums ...int) <-chan int {
    out := make(chan int)
    go func() {
        for _, n := range nums {
            out <- n
        }
        close(out)
    }()
    return out
}

func main() {
    defer func() {
        fmt.Println("the number of goroutines: ", runtime.NumGoroutine())
    }()

    // Set up the pipeline.
    out := gen(2, 3)
```

```
    for n := range out {
        fmt.Println(n)              // 2
        time.Sleep(5 * time.Second) // done thing, 可能异常中断接收
        if true { // if err != nil
            break
        }
    }
}
```

例子中，发送者通过 out chan 向下游发送数据，main 函数接收数据，接收者通常会依据接收到的据做一些具体的处理，这里用 Sleep 代替。如果这期间发生异常，导致处理中断，退出循环。gen 函中启动的 goroutine 并不会退出。

如何解决?

此处的主要问题在于，当接收者停止工作，发送者并不知道，还在傻傻地向下游发送数据。故而，我需要一种机制去通知发送者。我直接说答案吧，就不循渐进了。Go 可以通过 channel 的关闭向所有接收者发送广播信息。

修改后的代码:

```
package main

import "time"

func gen(done chan struct{}, nums ...int) <-chan int {
    out := make(chan int)
    go func() {
        defer close(out)
        for _, n := range nums {
            select {
            case out <- n:
            case <-done:
                return
            }
        }
    }()
    return out
}

func main() {
    defer func() {
        time.Sleep(time.Second)
        fmt.Println("the number of goroutines: ", runtime.NumGoroutine())
    }()

    // Set up the pipeline.
    done := make(chan struct{})
    defer close(done)

    out := gen(done, 2, 3)

    for n := range out {
        fmt.Println(n) // 2
```

```
            time.Sleep(5 * time.Second) // done thing, 可能异常中断接收
            if true { // if err != nil
                break
            }
        }
    }
```

函数 gen 中通过 select 实现 2 个 channel 的同时处理。当异常发生时，将进入 <-done 分支，实现 goroutine 退出。这里为了演示效果，保证资源顺利释放，退出时等待了几秒保证释放完成。

执行后的输出如下：

```
the number of goroutines:  1
```

在只有主 goroutine 存在。

接收不发送

发送不接收会导致发送者阻塞，反之，接收不发送也会导致接收者阻塞。直接看示例代码，如下：

```
package main

func main() {
    defer func() {
        time.Sleep(time.Second)
        fmt.Println("the number of goroutines: ", runtime.NumGoroutine())
    }()

    var ch chan struct{}
    go func() {
        ch <- struct{}{}
    }()
}
```

运行结果显示：

```
the number of goroutines:  2
```

当然，我们正常不会遇到这么傻的情况发生，现实工作中的案例更多可能是发送已完成，但是发送者没有关闭 channel，接收者自然也无法知道发送完毕，阻塞因此就发生了。

解决方案是什么？那当然就是，发送完成后一定要记得关闭 channel。

nil channel

向 nil channel 发送和接收数据都将会导致阻塞。这种情况可能在我们定义 channel 时忘记初始化的候发生。

示例代码：

```
func main() {
    defer func() {
        time.Sleep(time.Second)
        fmt.Println("the number of goroutines: ", runtime.NumGoroutine())
    }()
```

```
    var ch chan int
    go func() {
        <-ch
        // ch<-
    }()
}
```

两种写法：<-ch 和 ch<- 1，分别表示接收与发送，都将会导致阻塞。如果想实现阻塞，通过 nil channel 和 done channel 结合实现阻止 main 函数的退出，这或许是可以一试的方法。

```
func main() {
    defer func() {
        time.Sleep(time.Second)
        fmt.Println("the number of goroutines: ", runtime.NumGoroutine())
    }()

    done := make(chan struct{})

    var ch chan int
    go func() {
        defer close(done)
    }()

    select {
    case <-ch:
    case <-done:
        return
    }
}
```

在 goroutine 执行完成，检测到 done 关闭，main 函数退出。

真实的场景

真实的场景肯定不会像案例中的简单，可能涉及多阶段 goroutine 之间的协作，某个 goroutine 可即使接收者又是发送者。但归根接底，无论什么使用模式。都是把基础知识组织在一起的合理运用。

传统同步机制

虽然，一般推荐 Go 并发数据的传递，但有些场景下，显然还是使用传统同步机制更合适。Go 中提传统同步机制主要在 sync 和 atomic 两个包。接下来，我主要介绍的是锁和 WaitGroup 可能导致 goroutine 的泄露。

Mutex

和其他语言类似，Go 中存在两种锁，排它锁和共享锁，关于它们的使用就不作介绍了。我们以排它为例进行分析。

示例如下：

```
func main() {
    total := 0

    defer func() {
        time.Sleep(time.Second)
        fmt.Println("total: ", total)
```

```
        fmt.Println("the number of goroutines: ", runtime.NumGoroutine())
    }()

    var mutex sync.Mutex
    for i := 0; i < 2; i++ {
        go func() {
            mutex.Lock()
            total += 1
        }()
    }
}
```

执行结果如下：

```
total: 1
the number of goroutines: 2
```

这段代码通过启动两个 goroutine 对 total 进行加法操作，为防止出现数据竞争，对计算部分做了加保护，但并没有及时的解锁，导致 i = 1 的 goroutine 一直阻塞等待 i = 0 的 goroutine 释放锁。可看到，退出时有 2 个 goroutine 存在，出现了泄露，total 的值为 1。

怎么解决？因为 Go 有 defer 的存在，这个问题还是非常容易解决的，只要记得在 Lock 的时候，记住 defer Unlock 即可。

示例如下：

```
mutex.Lock()
defer mutext.Unlock()
```

其他的锁与这里其实都是类似的。

WaitGroup

WaitGroup 和锁有所差别，它类似 Linux 中的信号量，可以实现一组 goroutine 操作的等待。使用时候，如果设置了错误的任务数，也可能会导致阻塞，导致泄露发生。

一个例子，我们在开发一个后端接口时需要访问多个数据表，由于数据间没有依赖关系，我们可以并访问，示例如下：

```
package main

import (
    "fmt"
    "runtime"
    "sync"
    "time"
)

func handle() {
    var wg sync.WaitGroup

    wg.Add(4)

    go func() {
        fmt.Println("访问表1")
```

```
        wg.Done()
    }()

    go func() {
        fmt.Println("访问表2")
        wg.Done()
    }()

    go func() {
        fmt.Println("访问表3")
        wg.Done()
    }()

    wg.Wait()
}

func main() {
    defer func() {
        time.Sleep(time.Second)
        fmt.Println("the number of goroutines: ", runtime.NumGoroutine())
    }()

    go handle()
    time.Sleep(time.Second)
}
```

执行结果如下：

```
the number of goroutines: 2
```

出现了泄露。再看代码，它的开始部分定义了类型为 sync.WaitGroup 的变量 wg，设置并发任务数为 4，但是从例子中可以看出只有 3 个并发任务。故最后的 wg.Wait() 等待退出条件将永远无法满足，handle 将会一直阻塞。

怎么防止这类情况发生？

我个人的建议是，尽量不要一次设置全部任务数，即使数量非常明确的情况。因为在开始多个并发任之间或许也可能出现被阻断的情况发生。最好是尽量在任务启动时通过 wg.Add(1) 的方式增加。

示例如下：

```
    wg.Add(1)
    go func() {
        fmt.Println("访问表1")
        wg.Done()
    }()

    wg.Add(1)
    go func() {
        fmt.Println("访问表2")
        wg.Done()
    }()

    wg.Add(1)
    go func() {
```

```
        fmt.Println("访问表3")
        wg.Done()
    }()
```

总结

大概介绍完了我认为的所有可能导致 goroutine 泄露的情况。总结下来，其实无论是死循环、channel 阻塞、锁等待，只要是会造成阻塞的写法都可能产生泄露。因而，如何防止 goroutine 泄露就变成了何防止发生阻塞。为进一步防止泄露，有些实现中会加入超时处理，主动释放处理时间太长的 goroutine。

本篇主要从如何写出正确代码的角度来介绍如何防止 goroutine 的泄露。下篇将会介绍如何实现更好监控检测，以帮助我们发现当前代码中已经存在的泄露。

参考资料

Concurrency In Go

Goroutine leak

Leaking-Goroutines

Go Concurrency Patterns: Context

Go Concurrency Patterns: Pipelines and cancellation

make goroutine stay running after returning from function

Never start a goroutine without knowing how it will stop