C/C++Linux服务器开发

高级架构师课程

三年课程沉淀 | 五次精益升级 | 十年行业积累 | 百个实战项目 | 十万内容受众

讲师:darren/326873713

扫一扫 升职加薪

班主任:柚子/2690491738

讲师介绍--专业来自专注和实力

King老师

系统架构师，曾供职著名创业公司系统架构师，微软亚洲研究院、创维集团全球研发中心。国内第一代商业Paas平台开发者。著有多个软件专利，参与多个开源软件维护。在全球化，高可用的物联网云平台架构与智能硬件设计方面有丰富的研发与实战经验。

Darren老师

曾供职于国内知名半导体公司（珠海扬智/深圳联发科），曾在某互联网公司担任音视频通话项目经理。主要从事音视频驱动、多媒体中间件、流媒体服务器的开发，开发过即时通讯+音视频通话的大型项目，在音视频、C/C++/GO Linux服务器领域有丰富的实战经验。

GO语言并发编程

1. Goroutine
2. Channel
3. 线程安全
4. context

cgo go和c混编

1 Go协程 Goroutine

1.1 Goroutine使用
1.2 Goroutine原理

1.1 如何使用Goroutine

在函数或方法调用前面加上关键字go，您将会同时运行一个新的Goroutine

```
func hello() {
    fmt.Println("Hello world goroutine")
}
func main() {
    go hello()
    time.Sleep(1 * time.Second)
    fmt.Println("main function")

}
```

1.2 子协程异常退出的影响

在使用子协程时一定要特别注意保护好每个子协程，确保它们正常安全的运行。因为子协程的异常退出会将异常传播到主协程，直接会导致主协程也跟着挂掉，然后整个程序就崩溃了。

```
fmt.Println("run in main goroutine")
go func() {
    fmt.Println("run in child goroutine")
    go func() {
        fmt.Println("run in grand child goroutine")
        go func() {
            fmt.Println("run in grand grand child goroutine")
            var ptr *int
            *ptr = 0x12345 // 故意制造崩溃
        }()
    }()
}()
time.Sleep(time.Second)
fmt.Println("main goroutine will quit")
```

1-2-panic.go

1.3 协程异常处理-recover

recover 是一个Go语言的内建函数，可以让进入宕机流程中的 goroutine 恢复过来，recover 仅在延迟函数 defer 中有效。
如果当前的 goroutine 陷入恐慌，调用 recover 可以捕获到 panic 的输入值，并且恢复正常的执行。

```
go func() {
    defer func() { // 要在对应的协程里执行
        fmt.Println("执行defer:")
        if err := recover(); err != nil {
            fmt.Println("捕获error:", err)
        }
    }()
    fmt.Println("run in grand grand child goroutine")
    var ptr *int
    *ptr = 0x12345 // 故意制造崩溃
}()
```

1-3-recover.go 1-3-panic-recover.go

1-4 启动百万协程

Go 语言能同时管理上百万的协程

```go
const N = 1000000
func main() {
    fmt.Println("run in main goroutine")
    i := 1
    for {
        go func() {
            for {
                time.Sleep(time.Second)
            }
        }()
        if i%10000 == 0 {
            fmt.Printf("%d goroutine started\n", i)
        }
        i++
        if i== N {
            break
        }
    }
    time.Sleep(time.Second*10)
}
```

1-4-million.go

1–5 死循环

如果有个别协程死循环了会导致其它协程饥饿得到不运行么?

```go
func main() {
    fmt.Println("run in main goroutine")
    n := 3
    for i := 0; i < n; i++ {
        go func() {
            fmt.Println("dead loop goroutine start")
            for {
            } // 死循环
        }()
    }
    for {
        time.Sleep(time.Second)
        fmt.Println("main goroutine running")
    }
}
```

1-5-loop.go

1.6 设置线程数

Go的调度器使用了一个叫做GOMAXPROCS的变量来决定会有多少个操作系统的线程同时执行Go的代码。其默认的值是运行机器上的CPU的核心数，所以在一个有8个核心的机器上时，调度器一次会在8个OS线程上去调度GO代码。

```go
func main() {
    // 读取默认的线程数
    fmt.Println(runtime.GOMAXPROCS(0))
    // 设置线程数为 10
    runtime.GOMAXPROCS(10)
    // 读取当前的线程数
    fmt.Println(runtime.GOMAXPROCS(0))
}
```

1-6-threads.go

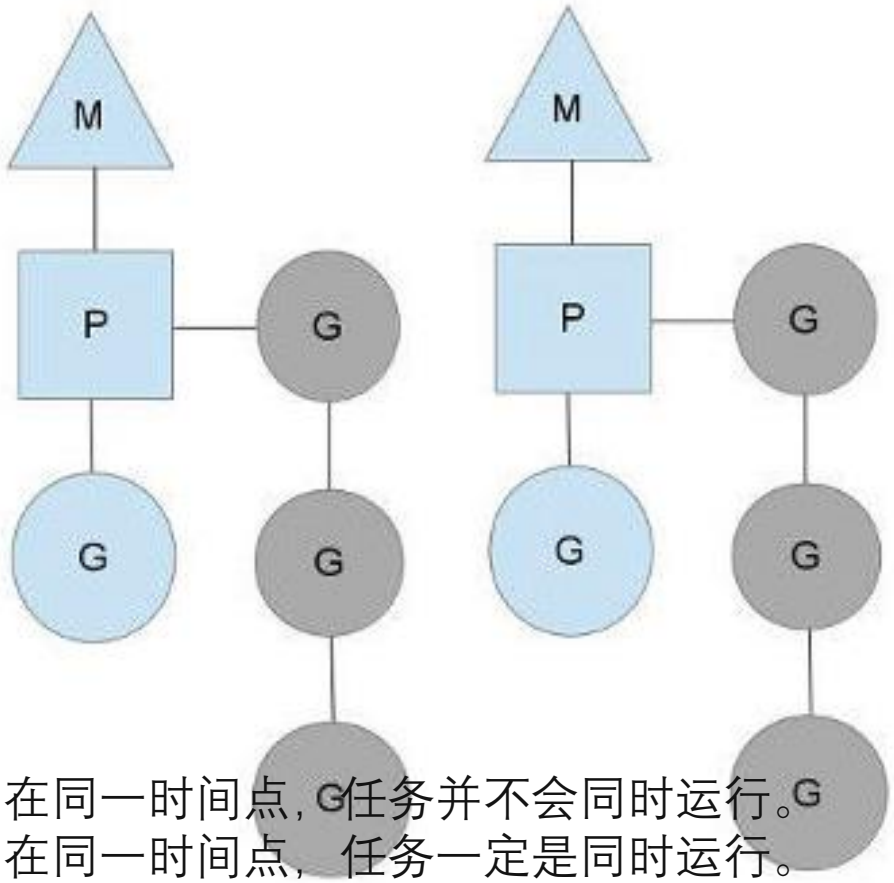
1-7 G-P-M模型-为什么引入协程？

核心原因为goroutine的轻量级，无论是从进程到线程，还是从线程到协程，其核心都是为了使得我们的调度单元更加轻量级。可以轻易得创建几万几十万的goroutine而不用担心内存耗尽等问题。

M Machine： os线程（即操作系统内核提供的线程）。

G：goroutine，其包含了调度一个协程所需要的堆栈以及instruction pointer（IP指令指针），以及其他一些重要的调度信息。

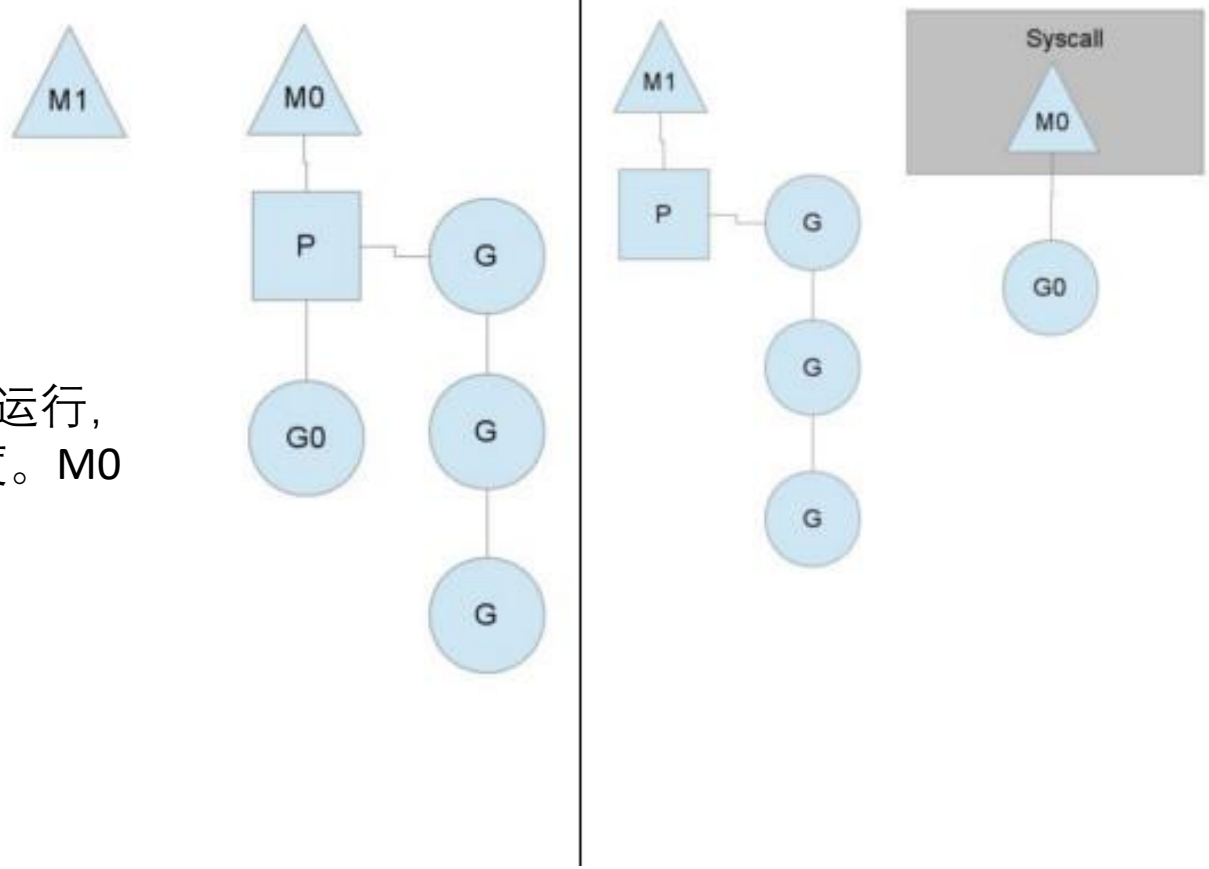
P Process： M与P的中介，实现m:n 调度模型的关键，M必须拿到P才能对G进行调度，P其实限定了golang调度其的最大并发度。
表示一个逻辑处理器一个p绑定一个os线程



并发：把任务在不同的时间点交给处理器进行处理。在同一时间点，G任务并不会同时运行。
并行：把每一个任务分配给每一个处理器独立完成。在同一时间点，任务一定是同时运行。

1-7 G-P-M模型-系统调用

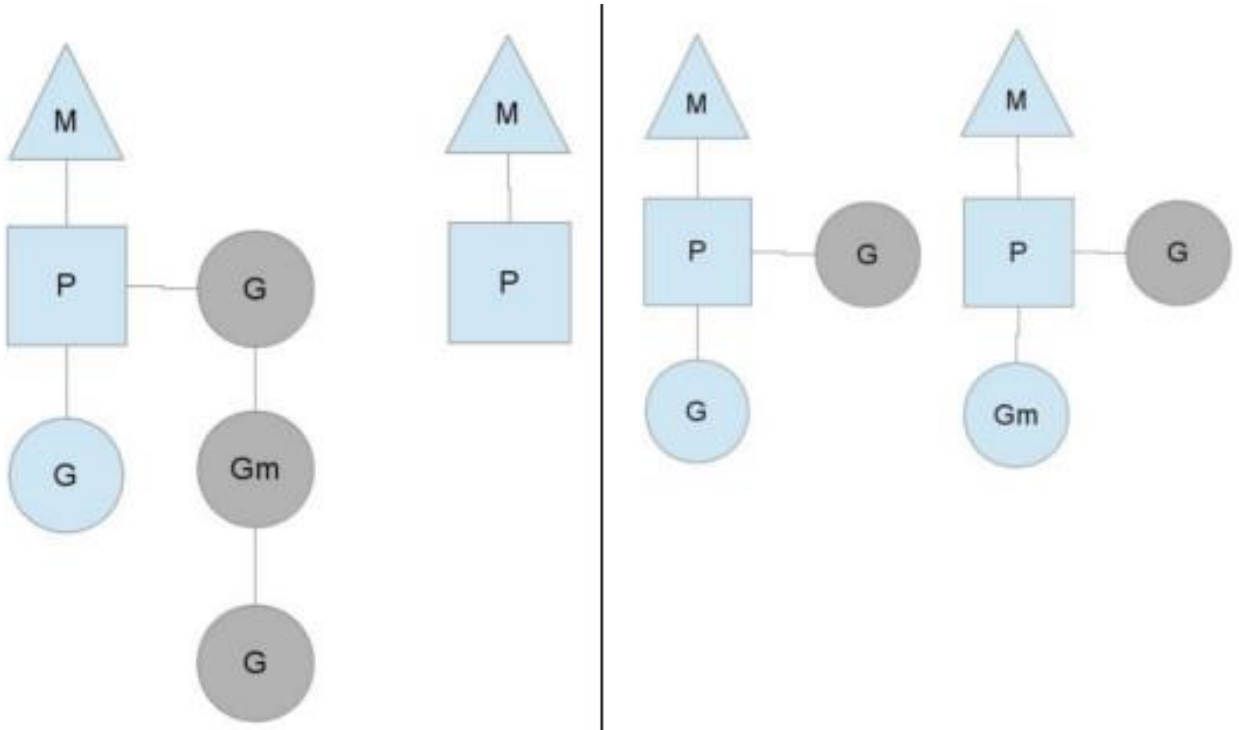
调用system call陷入内核没有返回之前，为保证调度的并发性，golang 调度器在进入系统调用之前从线程池拿一个线程或者新建一个线程，当前P交给新的线程M1执行。



G0返回之后，需要找一个可用的P继续运行，如果没有则将其放在全局队列等待调度。M0待G0返回后退出或放回线程池。

1-7 G-P-M模型-工作流窃取

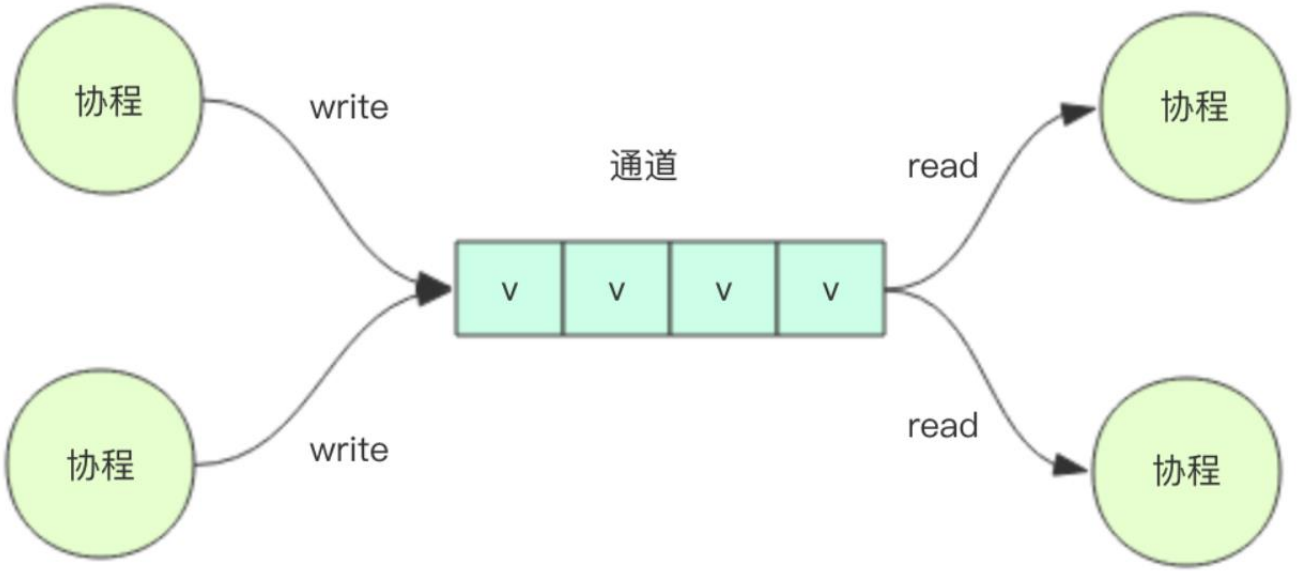
在P队列上的goroutine全部调度完了之后，对应的M首先会尝试从global runqueue中获取goroutine进行调度。如果golbal runqueue中没有goroutine，当前M会从别的M对应P的local runqueue中抢一半的goroutine放入自己的P中进行调度。**具体要看C代码去了。**



2 通道channel

如果说goroutine是Go语音程序的并发体的话，那么channels它们之间的通信机制。

作为协程的输出，通道是一个容器，它可以容纳数据。
作为协程的输入，通道是一个生产者，它可以向协程提供数据。
通道作为容器是有限定大小的，满了就写不进去，空了就读不出来。
通道有它自己的类型，它可以限定进入通道的数据的类型。



是否可以用锁?

2.1 创建通道

创建通道**只有一种语法，使用make 函数**

有两种通道类型:

「缓冲型通道」 var bufferedChannel = **make**(**chan int(这里是类型，什么类型都行)**, 1024)

「非缓冲型通道」 var unbufferedChannel = make(chan int)

两个相同类型的channel可以使用==运算符比较。如果两个channel引用的是相通的对象，那么比较的结果为真。一个channel也可以和nil进行比较。

```
ch <- x  // a send statement
x = <-ch // a receive expression in an assignment statement
<-ch     // a receive statement; result is discarded
```

2.2 读写通道

Go 语言为通道的读写设计了特殊的箭头语法糖 <-，让我们使用通道时非常方便。把箭头写在通道变量的右边就是写通道，把箭头写在通道的左边就是读通道。一次只能读写一个元素

```go
var ch chan int = make(chan int, 4)
for i := 0; i < cap(ch); i++ {
    ch <- i // 写通道
}
for len(ch) > 0 {
    var value int = <-ch // 读通道
    fmt.Println(value)
}
```

2-2-channel-rw.go

通道作为容器，它可以像切片一样，使用 cap() 和 len() 全局函数获得通道的容量和当前内部的元素个数。

2-3 读写阻塞

通道满了，写操作就会阻塞，协程就会进入休眠，直到有其它协程读通道挪出了空间，协程才会被唤醒。如果有多个协程的写操作都阻塞了，一个读操作只会唤醒一个协程。

2-3-channel-block.go

```
for {
    value := <-ch
    fmt.Printf("recv %d\n", value)
    time.Sleep(time.Second)
}
```

2.4 关闭通道

Go 语言的通道有点像文件，不但支持读写操作， 还支持关闭。读取一个已经关闭的通道会立即返回通道类型的「零值」，而写一个已经关闭的通道会抛异常。如果通道里的元素是整型的，读操作是不能通过返回值来确定通道是否关闭的。

```
var ch = make(chan int, 4)
ch <- 1
ch <- 2
close(ch)

value := <-ch
fmt.Println(value)
value = <-ch
fmt.Println(value)
value = <-ch
fmt.Println(value)
ch <- 3
```

2-5 通道写安全

向一个已经关闭的通道执行写操作会抛出异常，这意味着我们在写通道时一定要确保通道没有被关闭。

2-5-channel-close-safe.go

```go
func send(ch chan int) {
    ch <- 1
    ch <- 2
    ch <- 3
    ch <- 4
    close(ch)
}
```

多人写入怎么办?

2-6 WaitGroup

在写端关闭channel对单写的程序有效，但是多写的时候呢?

使用到内置 sync 包提供的 WaitGroup 对象，它使用计数来等待指定事件完成。

2-6-channel-waitgroup.go

```go
var wg = new(sync.WaitGroup)
wg.Add(2)       // 增加计数值
go send(ch, wg) // 写
go send(ch, wg) // 写
```

```go
func send(ch chan int, wg *sync.WaitGroup) {
    defer wg.Done() // 计数值减一
    i := 0
    for i < 4 {
        i++
        ch <- i
    }
}
```

2-7 多路通道

在真实的世界中，还有一种消息传递场景，那就是消费者有多个消费来源，只要有一个来源生产了数据，消费者就可以读这个数据进行消费。这时候可以将多个来源通道的数据汇聚到目标通道，然后统一在目标通道进行消费。

2-7-channel-multi.go

2-8 多路复用select

```go
func recv(ch1 chan int, ch2 chan int) {
    for {
        select {
        case v := <-ch1:
            fmt.Printf("recv %d from ch1\n", v)
        case v := <-ch2:
            fmt.Printf("recv %d from ch2\n", v)
        }
    }
}
```

读取

2-8-channel-select .go

2-9 非阻塞读写

通道的非阻塞读写。当通道空时，读操作不会阻塞，当通道满时，写操作也不会阻塞。非阻塞读写需要依靠 select 语句的 default 分支。当 select 语句所有通道都不可读写时，如果定义了 default 分支，那就会执行 default 分支逻辑，这样就起到了不阻塞的效果。

```go
func send(ch1 chan int, ch2 chan int) {
    i := 0
    for {
        i++
        select {
        case ch1 <- i:
            fmt.Printf("send ch1 %d\n", i)
        case ch2 <- i:
            fmt.Printf("send ch2 %d\n", i)
        default:
            fmt.Printf("ch block\n")
            time.Sleep(2 * time.Second) // 这里只是为了演示
        }
    }
}
```

写入

2-9-channel-select-noblock.go

2-10 生产者、消费者模型

生产者消费模型

2-10-channel-producer-consumer.go

3-1 线程安全-互斥锁

竞态检查工具是基于运行时代码检查，而不是通过代码静态分析来完成的。这意味着那些没有机会运行到的代码逻辑中如果存在安全隐患，它是检查不出来的。
需要加上-race 执行

3-1-unsafe.go

```go
func write(d map[string]int) {
    d["fruit"] = 2
}

func read(d map[string]int) {
    fmt.Println(d["fruit"])
}
```

3-2 避免锁复制

sync.Mutex 是一个结构体对象，这个对象在使用的过程中要避免被复制 —— 浅拷贝。复制会导致锁被「分裂」了，也就起不到保护的作用。所以在平时的使用中要尽量使用它的指针类型。读者可以尝试将上面的类型换成非指针类型，然后运行一下竞态检查工具，会看到警告信息再次布满整个屏幕。锁复制存在于结构体变量的赋值、函数参数传递、方法参数传递中，都需要注意。

3-2-lock.go

```
func (d *SafeDict) Put(key string, value int) (int, bool) {
    d.mutex.Lock()
    defer d.mutex.Unlock()
    old_value, ok := d.data[key]
    d.data[key] = value
    return old_value, ok
}
```

3-3 使用匿名锁字段

在结构体章节，我们知道外部结构体可以自动继承匿名内部结构体的所有方法。如果将上面的 SafeDict 结构体进行改造，将锁字段匿名，就可以稍微简化一下代码。

3-3-anonymous_lock.go

```go
func (d *SafeDict) Put(key string, value int) (int, bool) {
    d.Lock()
    defer d.Unlock()
    old_value, ok := d.data[key]
    d.data[key] = value
    return old_value, ok
}
```

3-4 使用读写锁

日常应用中，大多数并发数据结构都是读多写少的，对于读多写少的场合，可以将互斥锁换成读写锁，可以有效提升性能。sync 包也提供了读写锁对象 RWMutex，不同于互斥锁只有两个常用方法 Lock() 和 Unlock()，读写锁提供了四个常用方法，分别是写加锁 Lock()、写释放锁 Unlock()、读加锁 RLock() 和读释放锁 RUnlock()。写锁是排他锁，加写锁时会阻塞其它协程再加读锁和写锁，读锁是共享锁，加读锁还可以允许其它协程再加读锁，但是会阻塞加写锁。

```
type SafeDict struct {
    data map[string]int
    *sync.RWMutex
}
```

3-4-rw-lock.go

3.5 发布订阅模型

综合前面学的

支持过滤器设置主题

3-5-pubsub.go

```
golang := p.SubscribeTopic(func(v interface{}) bool {
    if s, ok := v.(string); ok {
        return strings.Contains(s, "golang")
    }
    return false
})
```

3.6 sync.Once初始化

sync.Once.Do(f func())是一个挺有趣的东西,能保证once只执行一次，无论你是否更换once.Do(xx)这里的方法,这个sync.Once块只会执行一次。

3-6-once.go

```go
func main() {
    for i, v := range make([]string, 10) {
        once.Do(onces)
        fmt.Println("count:", v, "---", i)
    }
    for i := 0; i < 5; i++ {

        go func() {
            once.Do(onced)
            fmt.Println("213")
        }()
    }
    time.Sleep(4000)
}
func onces() {
    fmt.Println("执行onces")
}
func onced() {
    fmt.Println("执行onced")
}
```

4 Go语言Context

为什么需要 Context

- 每一个**处理**都应该有个**超时限制**
- 需要在调用中传递这个超时
  - 比如开始处理请求的时候我们说是 3 秒钟超时
  - 那么在函数调用中间，这个超时还剩多少时间了？
  - 需要在什么地方存储这个信息，这样请求处理中间可以停止

Context是协程安全的。代码中可以将单个Context传递给任意数量的goroutine，并在取消该Context时可以将信号传递给所有的goroutine。

4.1 Context接口

```
type Context interface {
    Deadline() (deadline time.Time, ok bool)
    Done() <-chan struct{}
    Err() error
    Value(key interface{}) interface{}
}
```

- Deadline方法是获取设置的截止时间的意思，第一个返回式是截止时间，到了这个时间点，Context会自动发起取消请求；第二个返回值ok==false时表示没有设置截止时间，如果需要取消的话，需要调用取消函数进行取消
- Done方法返回一个只读的chan，类型为struct{}，我们在goroutine中，如果该方法返回的chan可以读取，则意味着parent context已经发起了取消请求，我们通过Done方法收到这个信号后，就应该做清理操作，然后退出goroutine，释放资源
- Err方法返回取消的错误原因，因为什么Context被取消。
- Value方法获取该Context上绑定的值，是一个键值对，所以要通过一个Key才可以获取对应的值，这个值一般是线程安全的

4.1 Background()和TODO()

- Go语言内置两个函数：Background() 和 TODO()，这两个函数分别返回一个实现了 Context 接口的 background 和 todo。
- Background() 主要用于 main 函数、初始化以及测试代码中，作为 Context 这个树结构的最顶层的 Context，也就是根 Context。
- TODO()，它目前还不知道具体的使用场景，在不知道该使用什么 Context 的时候，可以使用这个。
- background 和 todo 本质上都是 emptyCtx 结构体类型，是一个不可取消，没有设置截止时间，没有携带任何值的 Context。

4.2 Context的结

```go
func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc) {
    return WithDeadline(parent, time.Now().Add(timeout))
}
```

```go
func WithCancel(parent Context) (ctx Context, cancel CancelFunc)
func WithDeadline(parent Context, deadline time.Time) (Context, CancelFunc)
func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc)
func WithValue(parent Context, key interface{}, val interface{}) Context
```

四个With函数，接收的都有一个partent参数，就是父Context，我们要基于这个父Context创建出子Context的意思

- WithCancel函数，传递一个父Context作为参数，返回子Context，**以及一个取消函数用来取消Context**
- WithDeadline函数，和WithCancel差不多，它会多传递一个截止时间参数，意味着到了这个时间点，会自动取消Context，当然我们也可以不等到这个时候，可以提前通过取消函数进行取消
- WithTimeout和WithDeadline基本上一样，**这个表示是超时自动取消，是多少时间后自动取消Context的意思，只是传参数不一样。**
- WithValue函数和取消Context无关，它是为了生成一个绑定了一个键值对数据的Context，这个绑定的数据可以**通过Context.Value方法访问到**

4.3 Context使用原则

- 不要把Context放在结构体中，要以参数的方式进行传递
- 以 Context 作为参数的函数方法，应该把 **Context 作为第一个参数**
- 给一个函数方法传递Context的时候，不要传递nil，如果不知道传递什么，就使用context.TODO
- Context 的 Value 相关方法应该传递请求域的必要数据，不应该用于传递可选参数;
- Context 是线程安全的，可以放心的在多个 Goroutine 中传递。

4.4 Derived contexts派生上下文

Context包提供了从现有Context值派生新Context值的函数。这些值形成一个树:当一个Context被取消时，从它派生的所有Context也被取消。

4-4-derived.go

推荐教程

https://geektutu.com/post/geecache-day1.html

Golang 如何正确使用 Context
https://studygolang.com/articles/23247?fr=sidebar