Go 并发元件

黎相敏

上海观源信息科技有限公司
上海市闵行区紫竹科技园 4 号楼 303B

12/01/2019

Focus

大纲

1. sync 包
2. 通道 `channel`

sync 包

基本元件

sync 包维护着用于同步底层内存访问的元件，包括

- WaitGroup
- Mutex 和 RWMutex
- Cond
- Once
- Pool

通道 channel

用途和种类

- 用途: 以线程安全的方式实现不同协程之间的通信
- 通道在声明时指定其数据流向
  - 读写　默认，例如，`xStream := make(chan int)`
  - 只读　将 <-箭头放在chan关键字的左边，例如，
    `yStream := make(<-chan int)`
  - 只写　将 <-箭头放在chan关键字的右边，例如，
    `zStream := make(chan<- int)`

温馨提示

go 会在必要时隐式地将单向通道 (只读/只写) 转为双向通道 (读写)

数据类型与传递

- 通道传递的数据类型可以被指定
- 对于一个可读通道`c`，可以利用表达式`c <- x`将值`x`放到通道里
- 对于一个可写通道`c`，可以利用表达式`y <- c`从通道里取出值`y`

代码片段 1: 通道数据传递样例

```
1 stringStream := make(chan string)
2 go func() {
3   stringStream <- "Hello channels!"
4 }()
5 fmt.Println(<-stringStream)
6
7 // 输出:
8 // Hello channels!
```

导致编译错误的操作

- 对只读通道的写操作

```
1 readStream := make(<- chan interface{})
2 readStream <- struct{}{}
3
4 // 输出:
5 // invalid operation: readStream <- struct {} literal
6 // (send // to receive-only type <-chan interface {})
```

- 对只写通道的读操作

```
1 writeStream := make(chan<- interface{})
2 <- writeStream
3
4 // 输出:
5 // invalid operation: <-writeStream (receive from
6 // send-only type chan<- interface {})
```

阻塞性

- 任何尝试向容量已满的通道发送数据的协程会等待直至通道有剩余空间
- 任何尝试向空通道读数据的协程都会等待直至有一个值进入通道

读操作的两种方式

- 只返回一个值: 从通道传输下来的数值
- 返回一个值x和一个布尔值closed
  - x表示从通道传输下来的数据
  - closed=false表示x是由于通道已关闭其容量为空而返回的所声明类型的默认值
  - closed=false则表示x是从别处写入通道的值

代码片段 2: 通道的两种读取操作

```
1  stringStream := make(chan string)
2  go func() {
3    stringStream <- "Hello channels!"
4  }()
5
6  salutation, ok := <-stringStream
7  fmt.Printf("(%v): %v", ok, salutation)
8
9  // 输出:
10 // (true): Hello channels!
```

通道的关闭

- `close(c)`函数调用可关闭通道`c`
- 可读通道能够满足无限的读请求 (即使通道已被关闭)

设计模式 1：FOR...RANGE遍历

range和for关键字组合可以通道作为实参，并在通道关闭时自动跳出循环

```
1  intStream := make(chan int)
2
3  go func() {
4      defer close(intStream)
5      for i := 1; i <= 5; i++ {
6      intStream <- i
7      }
8  }()
9
10 for integer := range intStream {
11     fmt.Printf("%v ", integer)
12 }
13
14 // 输出：
15 // 1 2 3 4 5
```

设计模式 2：作为取消信号

以通道的关闭为信号通知多个协程

```
begin := make(chan interface{})
var wg sync.WaitGroup
for i := 0; i < 3; i++ {
    wg.Add(1)
    go func(i int) {
        defer wg.Done()
        <-begin
        fmt.Printf("%v has begun\n", i)
    }(i)
}
fmt.Println("Unblocking goroutines...")
close(begin)
wg.Wait()

// Unblocking goroutines...
// 1 has begun
// 0 has begun
// 2 has begun
```

容量

- 通道根据容量可划分为 2 种类型：非缓存型和缓存型
- 非缓存型：声明时不指定容量或指定容量为 0
- 缓存型：通道在初始化时被赋予大于 0 的容量
  - 给定容量为n，对通道c可同时支持n个写操作
  - 可作为一个内存型消息队列实现并发进程间的通信
  - 通道为空且有挂起的等待协程时，新值的传入会绕过通道直接传给等待协程
  - 这类通道容易导致过早优化问题和隐藏死锁问题

通道的满和空都是相对与通道的容量而言的
在预知通道需要支持的并发写操作次数n时，容量大小为n的缓存型通道可以使得写操作尽可能快地执行，例如，缓存型通道作为消息队列

NIL 通道

给定nil通道c

- 对c的读操作会阻塞进程 (尽管并不一定会导致死锁??)

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

- 对c的写操作也会阻塞进程

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

- 关闭c会触发panic

```
1    panic: close of nil channel
```

对通道不同操作的后果

Table 3-2. Result of channel operations given a channel's state

Operation	Channel state	Result
Read	`nil`	Block
	Open and Not Empty	Value
	Open and Empty	Block
	Closed	<default value>, false
	Write Only	Compilation Error
Write	`nil`	Block
	Open and Full	Block
	Open and Not Full	Write Value
	Closed	**panic**
	Receive Only	Compilation Error
`close`	`nil`	**panic**
	Open and Not Empty	Closes Channel; reads succeed until channel is drained, then reads produce default value
	Open and Empty	Closes Channel; reads produces default value
	Closed	**panic**
	Receive Only	Compilation Error

图：对通道不同操作的后果

由表可知，

- 阻塞协程的操作有 3 种
- 导致协程恐慌的操作有 3 种

上下文

通道的正确使用需要有合理的上下文环境，而为通道赋予正确的所有权协助创造这种环境

通道的所有权应该属于负责初始化、写入数据和关闭通道的协程

通过这种规则限定

- 所有者持有对通道的读写权限，并负责
  - 初始化通道
  - 执行写操作或将通道所有权转移给其他协程
  - 关闭通道
  - 将以上 3 项封装为一个只读通道暴露给消费者
- 消费者则对通道只有读权限，只需关心
  - 监听通道的关闭
  - 负责处理任何情况下的阻塞

基于通道实现的所有者—消费者样例

```go
chanOwner := func() <-chan int {
    resultStream := make(chan int, 5)
    go func() {
        defer close(resultStream)
        for i := 0; i <= 5; i++ {
        resultStream <- i
        }
    }()
    return resultStream
}

resultStream := chanOwner()
for result := range resultStream {
    fmt.Println("Received:", result)
}
fmt.Println("Done!")
```

```
// 输出:
// Received: 0
// Received: 1
// Received: 2
// Received: 3
// Received: 4
// Received: 5
// Done!
```