# 26 | 高并发爬虫：模型、控制与冲突检测

2022-12-08 郑建勋 来自北京



《Go进阶 · 分布式爬虫实战》

课程介绍 >

讲述：郑建勋

时长 12:38　大小 11.54M

你好，我是郑建勋。

上一节课，我们看到了协程和调度器的工作原理，协程的特性决定了我们无法保证协程之间的执行顺序，然而在真正的实践中，协程是无法完全隔离的。它们通常需要完成数据的共享，或者说要完成某种通信，而这又会导致数据的并发安全等新的问题。

所以，如何合理地组织大量的协程，协程之间如何进行通信，协程怎么优雅退出，如何保证并发安全，这些是我们在设计高并发的模型时需要考虑的问题。好的并发模型能够提升程序的性能、扩展性与安全性。

这节课，我们就来看看一些富有表现力的高并发模型。让我们先从并发带来的问题说起。

## 数据争用

在 Go 语言中，当两个以上协程同时访问相同的内存空间，并且至少有一个写操作时，就可能会出现并发安全问题，这种现象也被叫做数据争用。在下面这个例子中，两个协程共同访问了全局变量 count。这个程序其实是有数据争用的，因为 count 的最终结果是不明确的。



```
复制代码
1 var count = 0
2 func add() {
3   count++
4 }
5 func main() {
6   go add()
7   go add()
8 }
```

count++ 操作看起来是一条指令，但是对 CPU 来说，需要先读取 count 的值，执行 +1 操作，再将 count 的值写回内存。大部分人期望的操作可能是： R←0 代表读取到 0，w→1 代表写入 count 为 1；协程 1 写入数据 1 后，协程 2 再写入，count 最后的值为 2。




但是当两个协程并行时，情况开始变得复杂了。如果执行的流程如下所示，那么 count 最后的值为 1，这说明如果并发设计不合理会带来不确定性的结果。

数据争用可谓高并发程序中最难排查的问题。因为它的结果是不明确的，而且可能只在特定条件下出错。这就导致很难复现相同的错误，在测试阶段也不一定能测试出问题。而要解决数据争用问题，我们需要一些机制来保证某一时刻只能有一个协程执行特定操作，我们比较熟悉的传统解决手段就是锁，它包括原子锁、互斥锁与读写锁。



## 原子锁

就像我们刚才看到的，即便是简单的像 count++ 这样的操作，在底层也经历了读取数据、 更新 CPU 缓存、存入内存这一系列操作。这些操作如果并发进行可能出现严重错误。这种情况就可以用原子锁来保证并发的安全。

还有一些更加复杂的场景需要用到原子锁。许多编译器（在编译时）和 CPU 处理器（在运行时）通过调整指令顺序进行优化，因此指令执行顺序可能与代码中显示的不同。例如，如果已知有两个内存引用将到达同一位置，并且没有中间写入会影响该位置，那么编译器可能只使用最初获取的值。又如，在编译时，a + b + c 并不能用一条 CPU 指令来执行，所以按照加法结合律，它可能被拆分为 b+c 再 +a 的形式。这种非原子性的操作就可能会遇到并发问题。



```
sum = a + b + c;
=>
sum = b + c;
sum = a + sum;
```

另外，在 CPU 执行过程中，不仅可能出现编译器执行顺序混乱的问题，也可能发生与程序中执行顺序不同的内存访问。例如，许多处理器包含存储缓冲区，这个缓冲区会接收对内存的挂起写操作，写缓冲区基本上是 < 地址，数据 > 的队列。通常，这些写操作可以按顺序执行，但是如果随后的写操作地址已经存在于写缓冲区中了，那可以将此写操作与先前的挂起写操作组合在一起。

还有一种情况是，处理器高速缓存未命中。这时，许多处理器在等待当前指令从主内存中获取数据时，为了最大程度地利用资源， 会继续执行后续指令，导致操作乱序。因此需要有一种机制解决并发访问时数据冲突及内存操作乱序的问题，即提供一种原子性的操作。这通常依赖硬件的支持，例如 X86 指令集中的 LOCK 指令，对应到 Go 语言就是 sync/atomic 包。

下面这个例子使用 atomic.AddInt64 函数将变量增加了 1，这种原子操作不会发生并发时的数据争用问题。

复制代码

```
var count int64 = 0
func add() {
  atomic.AddInt64(&count,1)
}
func main() {
  go add()
  go add()
}
```



sync/atomic 包中还有一个重要的功能：CompareAndSwap，它能够对比并替换元素值。

下面这个例子中，atomic.CompareAndSwapInt64 会判断 flag 变量的值是否为 0，如果是，则将 flag 的值设置为 1。这一系列操作都是原子性的，不会发生数据争用，也不会出现内存操作乱序问题。通过 sync/atomic 包中的原子操作，我们能构建起一种自旋锁，只有获取该锁，才能执行区域中的代码。下面这段代码使用一个 for 循环不断轮询原子操作，直到原子操作成功才获取该锁。

复制代码

```
var flag int64 = 0
var count int64 = 0
func add() {
  for {
    if atomic.CompareAndSwapInt64(&flag, 0, 1) {
      count++
      atomic.StoreInt64(&flag, 0)
      return
    }
  }
}
func main() {
  go add()
  go add()
}
```

这种自旋锁的形式在 Go 源代码中随处可见。其实，原子操作是保证同步的最基本的技术，通过原子操作可以构建起许多同步原语，例如自旋锁、信号量、互斥锁等。

## 互斥锁

通过原子操作构建起的自旋锁，虽然简单高效，但是它并不是万能的。例如，当某一个协程长时间霸占锁的时候，其他协程仍在继续抢占锁，这会导致 CPU 资源持续无意义地被浪费。同时，当有许多协程都在获取锁的时候，可能会有协程始终抢占不到锁。


为了解决这种问题，操作系统的锁接口提供了终止与唤醒的机制（例如 Linux 中的 pthread mutex），这就避免了频繁自旋造成的浪费。不过，调用操作系统级别的锁会锁住整个线程使之无法运行，另外锁的抢占还会涉及线程之间的上下文切换。而 Go 语言借助协程实现了一种比传统操作系统级别的锁更加轻量级的互斥锁，它的使用方式如下：

复制代码

```
var count int64 = 0
var m sync.Mutex
func add() {
  m.Lock()
  count++
  m.Unlock()
}
func main() {
  go add()
  go add()
}
```

这里，sync.Mutex 构建起了互斥锁，在同一时刻，只会有一个获取了锁的协程会继续执行任务，其他的协程将陷入等待状态。借助协程的休眠与调度器的调度，这种锁会变得非常轻量。

## 读写锁

当然，互斥锁也并不总是最好的。由于在同一时间内只能有一个协程获取互斥锁并执行操作，那么在多读少写的情况下，如果长时间没有写操作，读取到的会是完全相同的值，使用互斥锁就显得没有必要了。这个时候，使用读写锁则更加恰当。

**读写锁通过两种锁来实现，一种为读锁，另一种为写锁。**当进行读取操作时，需要加读锁，而进行写入操作时则需要加写锁。多个协程可以同时获得读锁并执行。如果此时有协程申请了写锁，那么该协程会等待所有的读锁都释放后，才能获取写锁并执行。如果当前的协程申请读锁时已经存在写锁，那么读锁会等待写锁释放后再获取读锁并执行。

总之，读锁必须能观察到上一次写锁写入的值，写锁则要在之前的读锁释放后才能写入。可以有多个协程获得读锁，但只有一个协程可以获得写锁。

举一个简单的例子，哈希表并不是并发安全的，它只能够并发读取，一旦并发写入就会出现冲突。一种简单的规避方式是，在获取 Map 中的数据时加入 RLock 读锁，在写入数据时使用 Lock 写锁。


复制代码

```
type Stat struct {
  counters map[string]int64
  mutex sync.RWMutex
}
func (s *Stat) getCounter(name string) int64 {
  s.mutex.RLock()
  defer s.mutex.RUnlock()
  return s.counters[name]
}
func (s *Stat) SetCounter(name string){
  s.mutex.Lock()
  defer s.mutex.Unlock()
  s.counters[name]++
}
```

借助原始的并发控制手段，Go 提供了一些好用的并发控制工具，包括了 sync.WaitGroup，sync.Once、sync.Pool、sync.Cond。下面我们分开来看一下。

## Go 并发控制库

### sync.WaitGroup

先来看这样一个场景。在加载配置的过程中，我们希望可以让多个协程同时加载不同的配置文件，但是却希望等到所有协程都加载完毕后才让程序提供服务，这可以加快程序的运行。

这时，一种方案是使用 sleep 休眠，但这是一种低效的解决方法。更高效的方法就是使用 Go 标准库中的`sync.WaitGroup`，它会等待所有的协程执行完毕后退出。

`sync.WaitGroup` 提供了 3 个 API。其中，`WaitGroup.Add` 代表将等待的数量加 1，`WaitGroup.done` 代表将等待的数量减 1，`WaitGroup.Wait` 代表陷入等待，直到等待的数量为 0。所以我们一般在开启协程之前调用 `WaitGroup.Add`，然后开启多个工作协程；在每一个协程结束时延迟调用`WaitGroup.done`， 在末尾调用 `WaitGroup.Wait` 陷入堵塞，等待所有协程执行完毕。示例代码如下：

```go
package main

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

func worker(id int) {
    fmt.Printf("Worker %d starting\\n", id)

    time.Sleep(time.Second)
    fmt.Printf("Worker %d done\\n", id)
}

func main() {

    var wg sync.WaitGroup

    for i := 1; i <= 5; i++ {
        wg.Add(1)

        i := i

        go func() {
            defer wg.Done()
            worker(i)
        }()
    }

    wg.Wait()

}
```




## sync.Once

sync.Once 则可以保证某一个操作只能执行一次，它在实践中的使用也非常广泛。例如，我们希望配置的加载、日志的初始化只在初始化时加载一次。又如在释放资源时，我们希望文件描述符与通道只关闭一次，这时候都可以用到 sync.Once。在下面这个例子中，使用 sync.Once 只允许 MysSQL 数据库打开一次。

```go
var (
    once  sync.Once
)

func DbOnce() (*sql.DB, error) {
```

```
    once.Do(func() {
        fmt.Println("Am called")
        db, dbErr = sql.Open("mysql", "root:test@tcp(127.0.0.1:3306)/test")
        if dbErr != nil {
            return
        }
        dbErr = db.Ping()
    })
    return db, dbErr
}
```



## sync.Cond

sync.Cond 是 Go 提供的一种类似条件变量的同步机制，它能够让协程陷入阻塞，直到某个条件发生后再继续执行。

sync.Cond 包含了 3 个重要的 API：Cond.Wait()、Cond.Signal() 和 Cond.Broadcast() 。其中，Cond.Wait() 表示陷入等待，Cond.Broadcast() 会唤醒所有等待的协程，Cond.Signal() 只唤醒一个最先等待的协程。要注意的是，使用 Cond.Wait() 之前必须要调用Cond.L.Lock()进行加锁，在结束后还需要调用 Cond.L.UnLock() 进行解锁。

一般使用 sync.Cond 的正确姿势是： 协程 A 会用 for 循环判断是否满足 Condition 条件，如果不满足则陷入休眠。协程 B 会在恰当的时候调用 c.Broadcast() 唤醒等待的协程。使用 for 循环是因为当协程被唤醒时，并不能保证当前条件是满足的。这样做也可以实现某种程度上的解耦，消息的发出者并不需要知道具体的 Condition 条件是怎样的。



```
    // 协程A
    c.L.Lock()
    for !condition() {
        c.Wait()
    }
    ...
    c.L.Unlock()


   // 协程B
   ...
   c.Broadcast()
```

不过，在实践中并不经常使用 sync.Cond ，因为在很多场景下，我们都可以使用更为强大的通道。不过为了更透彻地讲解 sync.Cond，我们再来看几个可能会用到 sync.Cond 的例子。



第一个场景是这样的。我们设计的营销策略希望当在线用户达到 100 人之后，对前 10 位用户进行奖励。代码如下所示。

这里的判断条件就是，用户是否达到 100 人。如果用户没有达到 100 人，执行就会陷入堵塞。 而另一个程序，每个用户上线后都会发送通知信号，唤醒等待的协程。



```
// 协程A
cond.L.Lock()
for len(users) < 100 {
    cond.Wait()
}
givePrizes(users[:10])
cond.L.Unlock()

// 协程B
cond.L.Lock()
users = append(users, newUser)
cond.L.Unlock()
cond.Signal()
```

另外，如果程序收到了终止信号（例如开发者按下了 Ctrl+C）, 我们也希望程序能够通知所有协程关闭资源并退出。这时，我们需要增加判断条件，只有当在线用户小于 100 人并且程序没有终止时才会陷入堵塞。代码修改如下：



```
cond.L.Lock()
for len(users) < 100 && !shutdown {
    cond.Wait()
}
if shutdown {
    cond.L.Unlock()
    return
}
givePrizes(users[:10])
cond.L.Unlock()
```

sync.Cond 有堵塞与唤醒的语义，并且可以将通知者与等待者解耦，通知者不必知道具体的条件细节，所以程序会更加灵活。如果我们遇到了类似的场景，可以在合适的情况下使用 sync.Cond 。 不过我们也要小心，一旦忘记了释放锁或者忘记了唤醒协程，sync.Cond 可能遇到死锁问题。



我们还可以参考 Go 源码对 sync.Cond 的使用。例如 Go 在构建内存管道时使用了 sync.Cond。其中，pipe.Read 方法会循环读取管道中的数据，如果没有数据，则陷入到等待中。

复制代码

```
func (p *pipe) Read(d []byte) (n int, err error) {
  p.mu.Lock()
  defer p.mu.Unlock()
  for {
    ...
    if p.b != nil && p.b.Len() > 0 {
      return p.b.Read(d)
    }
    p.c.Wait()
  }
}
```

而 pipe.Write 则会在管道另一端写入数据后，唤醒第一个等待读取的协程。

复制代码

```
func (p *pipe) Write(d []byte) (n int, err error) {
  p.mu.Lock()
  defer p.mu.Unlock()
  if p.c.L == nil {
    p.c.L = &p.mu
  }
  defer p.c.Signal()
  if p.err != nil {
    return 0, errClosedPipeWrite
  }
  if p.breakErr != nil {
    p.unread += len(d)
    return len(d), nil // discard when there is no reader
  }
  return p.b.Write(d)
}
```

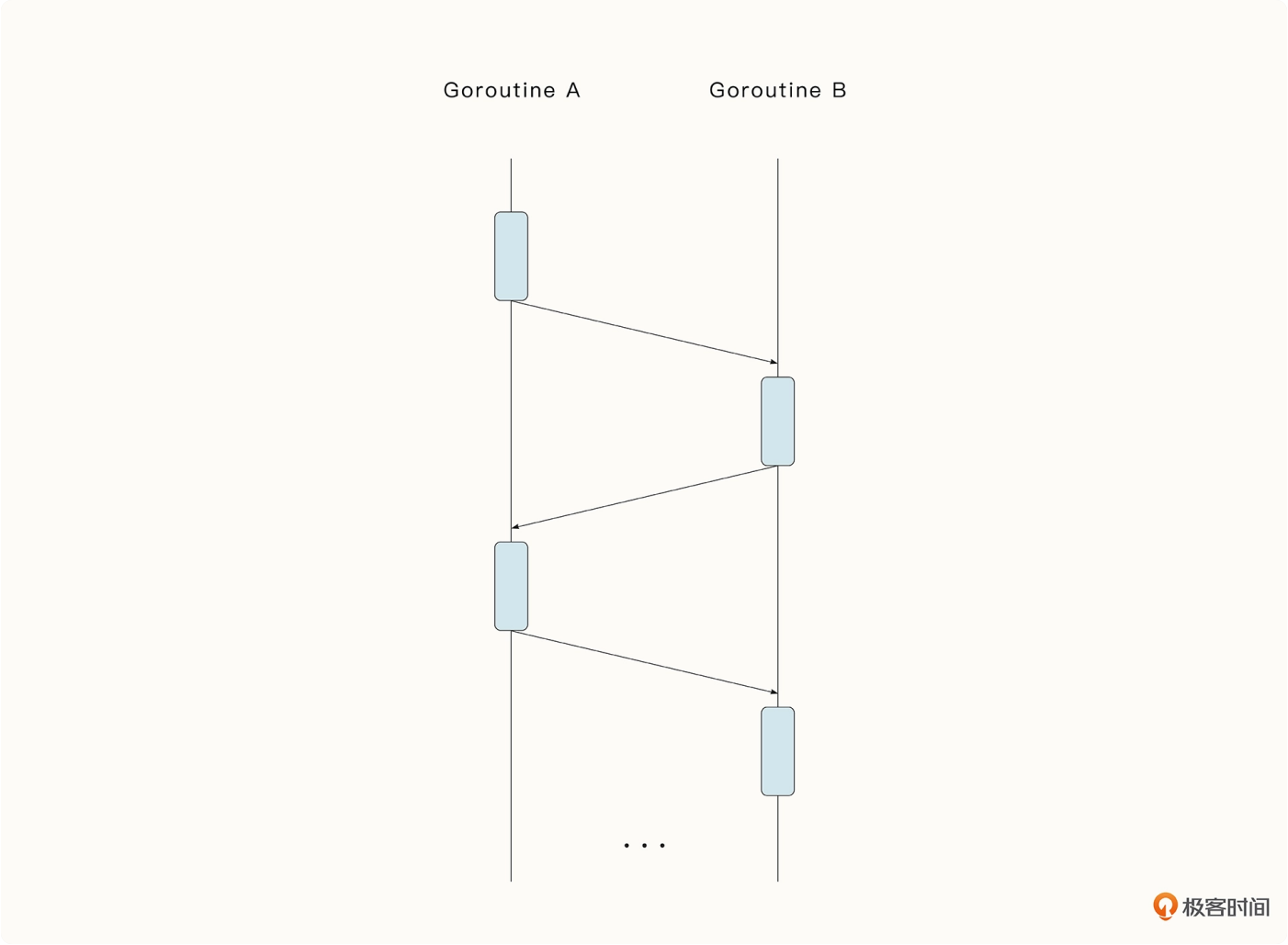
## Go 并发模型

之前我们讲了很多传统的同步模式，但是我们在实践中去协调协程时，使用得最多的还是通道。 就像 Go 语言圈子中的名言所说的那样：**不要通过共享内存来通信，通过通信来共享内存。**

通道的厉害之处在于，在通信的过程中完成了数据所有权的转移。数据只可能在某一个协程中执行，这就在无形中消除了并发访问数据的问题，数据争用问题仍然存在，例如通道内部仍然需要使用锁，但 Go 语言已经为我们屏蔽了底层锁实现的细节。借助通道，我们可以创造出许多有表现力的高并发模型。

## ping-pong 模式

ping-pong 模式即乒乓球模式，它比较形象地呈现了数据之间一来一回的关系。收到数据的协程可以在不加锁的情况下对数据进行处理，而不必担心有并发冲突。




实例代码如下所示。两个协程 player 就相当于两个球员，而通道 table 则类似于球桌。

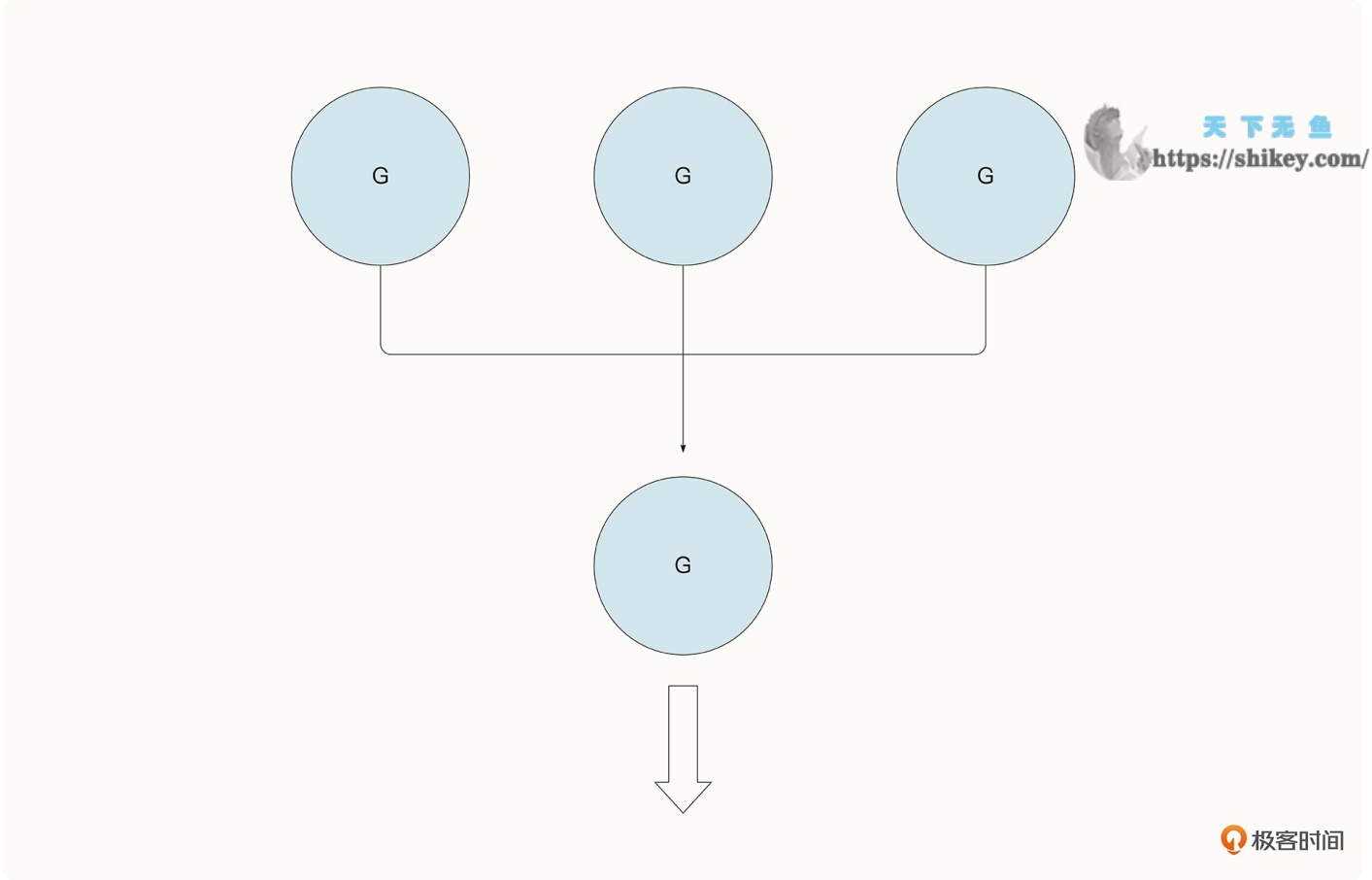
```
func main(){

  var Ball int

  table:= make(chan int)

  go player(table)
  go player(table)

  table<-Ball

  time.Sleep(1*time.Second)
  <-table

}
func player(table chan int) {
  for{
    ball:=<-table
    ball++
    time.Sleep(100*time.Millisecond)
    table<-ball
  }
}
```



你可以想一想，如果我们把两个 player 扩展为多个 player，是不是就有点像很多人在踢毽子了。当我们遇到类型的问题，可以用这一简单的模式来进行抽象。

## fan-in 模式

fan-in 模式又叫扇入模式，意思是多个协程把数据写入到通道中，但只有一个协程等待读取通道数据。




这种模式在实践中有很多应用场景。举个例子，我们想查找某一个文件夹中有没有特殊的关键字。当文件数量很多时，我们可以用并发的方式去查找，找到结果后输出到相同的通道中打印出来。

复制代码

```
func search(ch chan string, msg string) {
  var i int
  for {
    // 模拟找到了关键字
    ch <- fmt.Sprintf("get %s %d", msg, i)
    i++
    time.Sleep(1000 * time.Millisecond)
  }
}

func main() {
  ch := make(chan string)
  go search(ch, "jonson")
  go search(ch, "olaya")

  for i := range ch {
    fmt.Println(i)
  }
}
```

不过，fan-in 模式在读取数据时，并不总是只有一个通道。它也可以同时读取多个通道的内容，以多路复用的形式存在。让我们把上面的例子改造一下，现在 search 函数会返回一个新的通道，并新建协程把数据写入到这个通道中。在读取数据时，我们要监听 ch1、ch2 两个协程，并使用 select 来实现多路复用。



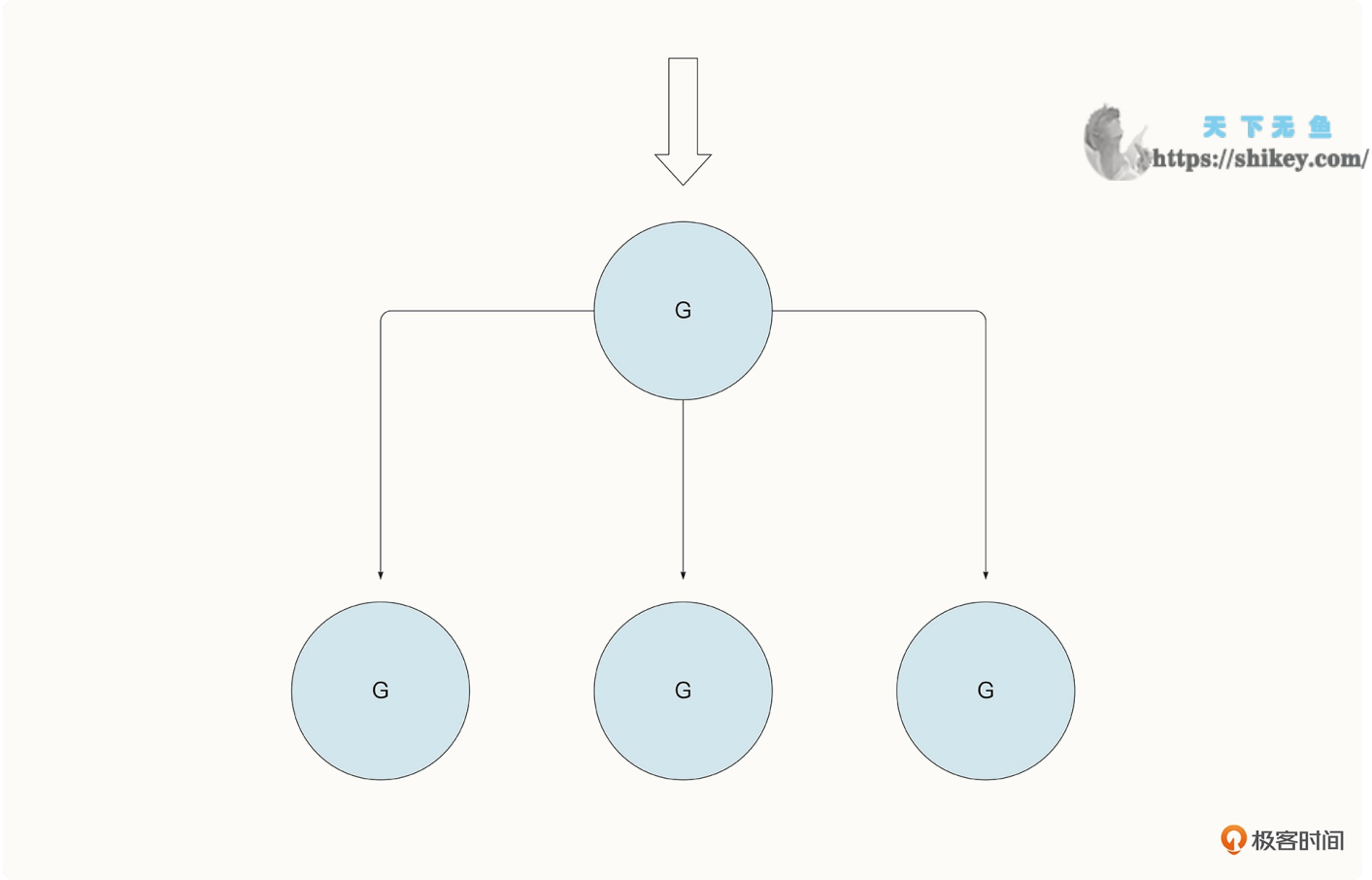


```
func search(msg string) chan string {
  var ch = make(chan string)
  go func() {
    var i int
    for {
      ch <- fmt.Sprintf("get %s %d", msg, i)
      i++
      time.Sleep(100 * time.Millisecond)
    }
  }()
  return ch
}

func main() {
  ch1 := search("jonson")
  ch2 := search("olaya")

  for {
    select {
    case msg := <-ch1:
      fmt.Println(msg)
    case msg := <-ch2:
      fmt.Println(msg)
    }
  }
}
```

fan-in 模式比较清晰，在实际中也是很常见的。例如我们之后在项目中会看到，通过 fan-in 模式来整合爬取到的数据，并存储起来。

## fan-out 模式

fan-out 模式与 fan-in 模式相反，它描述的是一个协程完成数据的写入，但是多个协程抢夺同一个通道中的数据的场景。




**Fan-out 模式通常会用在任务的分配中。**比方说，程序消费 Kafka、NATS 等中间件的数据，多个协程就会监听同一个通道中的数据，读到数据后立即进行后续的处理，处理完毕后再继续读取，循环往复。

以下面的代码为例。多个 worker 监听同一个协程，而 `tasksCh <- i` 会把任务分配到 worker 中去。fan-out 模式使 worker 得到了充分的利用，并且任务的分配也实现了负载均衡，哪一个 worker 闲下来了就会自动去领取新的任务（注意，示例代码中的 sync.WaitGroup 只是为了防止 main 函数提前退出）：

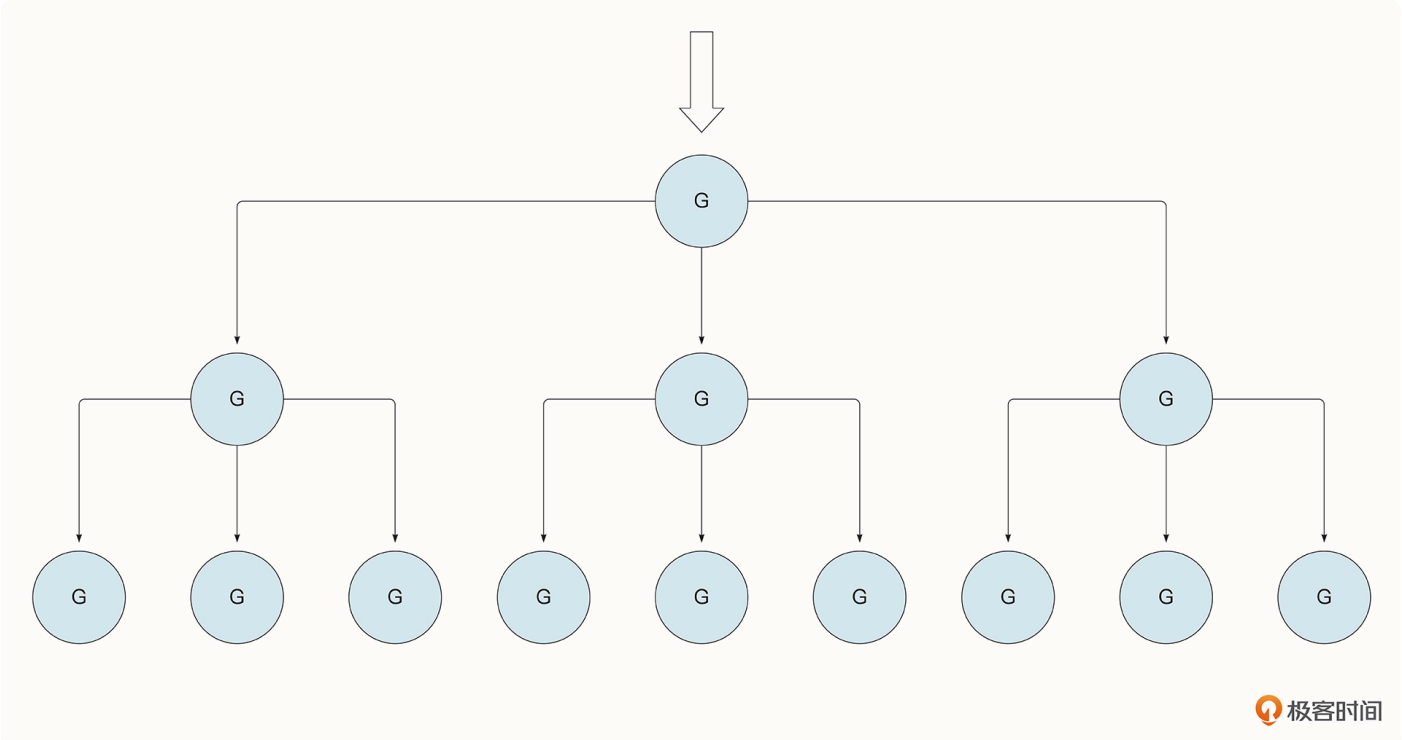
复制代码

```
func worker(tasksCh <-chan int, wg *sync.WaitGroup) {
  defer wg.Done()
  for {
    task, ok := <-tasksCh
    if !ok {
      return
    }
    d := time.Duration(task) * time.Millisecond
    time.Sleep(d)
    fmt.Println("processing task", task)
  }
}

```

```go
func pool(wg *sync.WaitGroup, workers, tasks int) {
  tasksCh := make(chan int)

  for i := 0; i < workers; i++ {
    go c(tasksCh, wg)
  }

  for i := 0; i < tasks; i++ {
    tasksCh <- i
  }

  close(tasksCh)
}

func main() {
  var wg sync.WaitGroup
  wg.Add(36)
  go pool(&wg, 36, 50)
  wg.Wait()
}
```



在生产实践中，我们还可以在上面这个例子的基础上构建出更复杂的模型，例如每一个 worker 中还可以分出多个 subwoker。




接下来我们就尝试在前例的基础上构建出具有 subworker 的并发模式。

如下所示，worker 也变成了类似调度的模式，worker 创建出了多个 subworker 的工作线程，并通过 subtasks <- task1 将任务分发到了 subworker 中。



复制代码

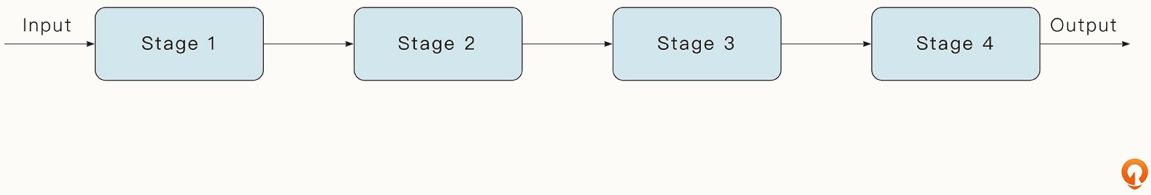
```
const (
  WORKERS    = 5
  SUBWORKERS = 3
  TASKS      = 20
  SUBTASKS   = 10
)

func subworker(subtasks chan int) {
  for {
    task, ok := <-subtasks
    if !ok {
      return
    }
    time.Sleep(time.Duration(task) * time.Millisecond)
    fmt.Println(task)
  }
}

func worker(tasks <-chan int, wg *sync.WaitGroup) {
  defer wg.Done()
  for {
    task, ok := <-tasks
    if !ok {
      return
    }

    subtasks := make(chan int)
    for i := 0; i < SUBWORKERS; i++ {
      go subworker(subtasks)
    }
    for i := 0; i < SUBTASKS; i++ {
      task1 := task * i
      subtasks <- task1
    }
    close(subtasks)
  }
}

func main() {
  var wg sync.WaitGroup
  wg.Add(WORKERS)
  tasks := make(chan int)

  for i := 0; i < WORKERS; i++ {
    go worker(tasks, &wg)
  }
```

```
  for i := 0; i < TASKS; i++ {
    tasks <- i
  }

  close(tasks)
  wg.Wait()
}
```



## pipeline 模式

pipeline 模式即管道模式，指的是由通道连接的一系列连续的阶段，以类似流的形式进行计算。每个阶段是一组执行特定任务的协程，每个阶段的协程都会通过通道获取从上游传递过来的值，经过处理后，再把新的值发送给下游。






其实我们平时的四则运算就很像一个管道。举个例子，我们要计算 2*(2*number+1) 这串数字就可以用下面的方式实现。可以看到，`multiply(v, 2)` 首先被计算出来，计算的结果会紧接着被送入 add 函数中执行加 1 操作。之后，生成的结果将继续作为 multiply 函数的参数被处理。

复制代码

```
func main() {
  multiply := func(value, multiplier int) int {
    return value * multiplier
  }

  add := func(value, additive int) int {
    return value + additive
  }

  ints := []int{1, 2, 3, 4}
  for _, v := range ints {
    fmt.Println(multiply(add(multiply(v, 2), 1), 2))
  }
}
```

《Concurrency in Go》 这本书中给出了将上例的算术操作转换为 pipeline 模式的例子，如下所示，我们梳理一下这段代码。



在这里，generator、multiply、add 是三个函数，代表管道的不同阶段。每个阶段会返回一个通道供下一个阶段消费。其中，multiply 代表乘法操作；add 代表加法操作；generator 是管道的第一个阶段，代表数据的产生。而在代码的最后，`for v := range pipeline` 代表管道的最后一个阶段，消费最后产生的结果。通道 done 则是为了实现协程的退出而设计的。

复制代码

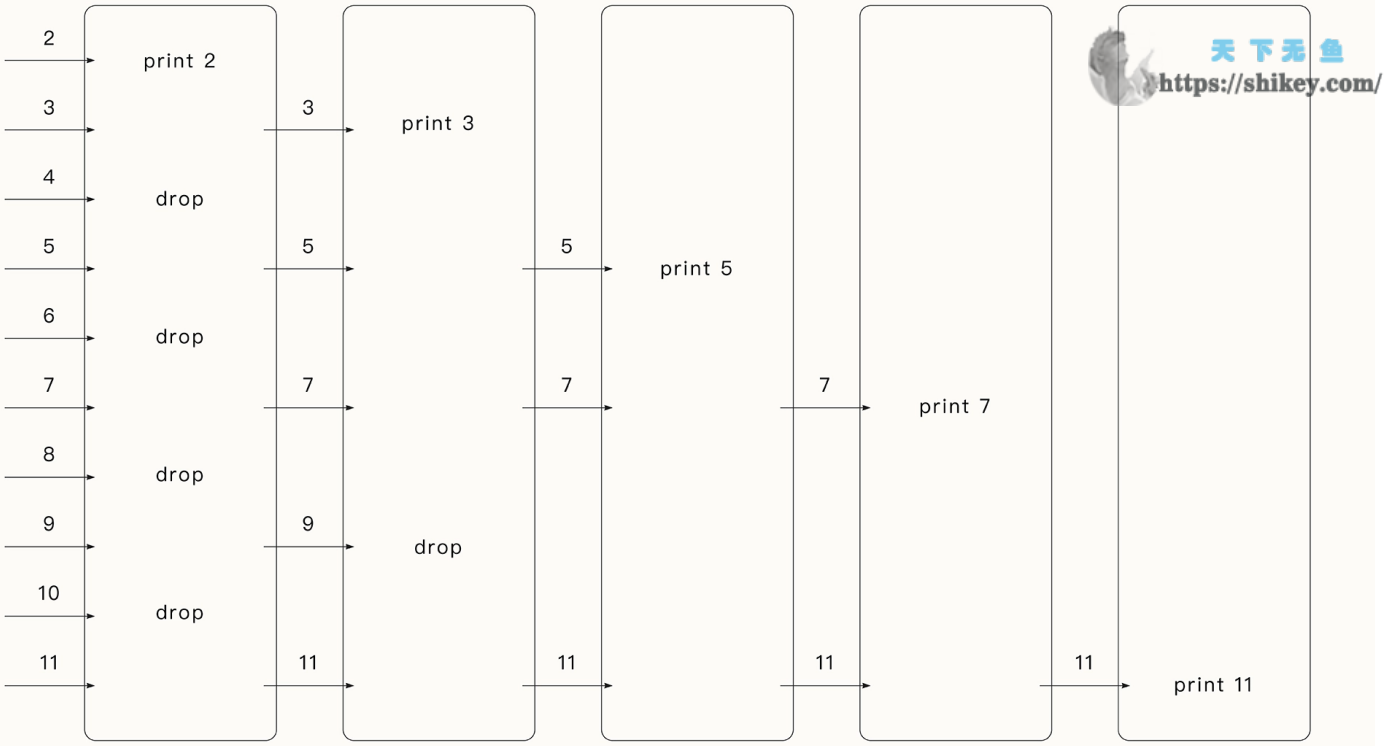
```
func main() {
  generator := func(done <-chan interface{}, integers ...int) <-chan int {
    intStream := make(chan int)
    go func() {
      defer close(intStream)
      for _, i := range integers {
        select {
        case <-done:
          return
        case intStream <- i:
        }
      }
    }()
    return intStream
  }

  multiply := func(
    done <-chan interface{},
    intStream <-chan int,
    multiplier int,
  ) <-chan int {
    multipliedStream := make(chan int)
    go func() {
      defer close(multipliedStream)
      for i := range intStream {
        select {
        case <-done:
          return
        case multipliedStream <- i * multiplier:
        }
      }
    }()
    return multipliedStream
  }

  add := func(
    done <-chan interface{},
    intStream <-chan int,
    additive int,
```

```
  ) <-chan int {
    addedStream := make(chan int)
    go func() {
      defer close(addedStream)
      for i := range intStream {
        select {
        case <-done:
          return
        case addedStream <- i + additive:
        }
      }
    }()
    return addedStream
  }

  done := make(chan interface{})
  defer close(done)

  intStream := generator(done, 1, 2, 3, 4)
  pipeline := multiply(done, add(done, multiply(done, intStream, 2), 1), 2)

  for v := range pipeline {
    fmt.Println(v)
  }
}
```



在管道中还有一个经典的案例：求素数。对于一个大于 1 的自然数，除了 1 和它自身外，不能被其他自然数整除的数就叫做素数。那我们怎么利用管道来计算前 1 万个素数呢？

我们可以在管道的每一个阶段都进行筛选。第一个阶段为数字的生成器，第二个阶段我们首先找到第 1 个素数 2，在这个阶段过滤出所有能够被 2 整除的数，这样我们就过滤出了 4、6、8 等偶数。这样我们也就能找到第一个不能被 2 整除的数字 3，可以推断出它一定是素数。因此第三个阶段，我们要排除所有能够被 3 整除的数字，这样就能够排除 9、15 等数字，而下一个不能被 3 整除的数是 5，它也一定是素数。把它作为第四个阶段筛选的依据，以此类推。




这个过程我们要怎么用代码实现呢？这段代码非常巧妙，我建议你细细品味一下：

复制代码

```
// 第一个阶段，数字的生成器
func Generate(ch chan<- int) {
  for i := 2; ; i++ {
    ch <- i // Send 'i' to channel 'ch'.
  }
}

// 筛选，排除不能够被prime整除的数
func Filter(in <-chan int, out chan<- int, prime int) {
  for {
    i := <-in // 获取上一个阶段的
    if i%prime != 0 {
      out <- i // Send 'i' to 'out'.
    }
  }
}

func main() {
  ch := make(chan int)
  go Generate(ch)
  for i := 0; i < 100000; i++ {
    prime := <-ch  // 获取上一个阶段输出的第一个数，其必然为素数
    fmt.Println(prime)
    ch1 := make(chan int)
```

```
25      go Filter(ch, ch1, prime)
26      ch = ch1 // 前一个阶段的输出作为后一个阶段的输入。
27    }
28  }
```


前面我们讲解了许多经典的并发模型。实际上，充满创意的并发模式还有很多，例如 or-channel 模式、or-done-channel 模式、tee-channel 模式、bridge-channel 模式等。在实际生产中也可能存在多种模型的组合。受到篇幅限制我就不再做过多讲解了，如果你对这方面有兴趣也可以继续深挖，相信一定会有所启发。

## 并发检测

Go 1.1 版本之后提供了强大的检查工具 race 来排查数据争用问题。race 可以被用在多个 Go 指令中，当检测器在程序中找到数据争用时，会打印报告。这个报告会包含发生 race 冲突的协程栈，以及此时正在运行的协程栈。

复制代码

```
1 $ go test -race mypkg
2 $ go run -race mysrc.go
3 $ go build -race mycmd
4 $ go install -race mypkg
```

我在后面的课程中还会介绍 race 的用法。

## 总结

好了，这节课的内容比较多，总结一下。

并发协程间的数据通信让我们不得不考虑并发安全的问题。解决这一问题的方式有很多。我们可以使用传统的同步手段，例如原子锁、互斥锁、读写锁，或者是 sync 标准库中的 sync.Once、sync.Cond、sync.WaitGroup 工具。当然，在 Go 语言中使用最多的还是通道，我们可以借助通道来实现不需要加锁的并发模型，例如 fan-in、fan-out、pipeline 等，你可以根据自己的实际需求进行灵活的组合。

## 课后题

学完这节课，请你思考下面这个问题：

你认为什么时候应该使用锁，什么时候应该使用通道？

欢迎你在留言区与我交流讨论，我们下节课见！




上一篇 25 | 运筹帷幄: 协程的运行机制与调度器原理

下一篇 27丨掘地三尺：实战深度与广度优先搜索算法

## 精选留言 (2)

写留言

**Realm**
2022-12-08 来自浙江

数据需要传递的时候用channel；
数据不动的时候，如获取、修改状态，用锁；

**shuff1e**
2022-12-08 来自北京

func pool(wg *sync.WaitGroup, workers, tasks int) { tasksCh := make(chan int) for i := 0; i < workers; i++ { go c(tasksCh, wg) } for i := 0; i < tasks; i++ { tasksCh <- i } close(tasksCh)}

go c改成go worker？