



下载APP



14 | Channel: 透过代码看典型的应用模式

2020-11-11 晁岳攀

Go 并发编程实战课

[进入课程 >](#)**讲述：安晓辉**

时长 20:59 大小 19.23M



你好，我是鸟窝。

前一讲，我介绍了 Channel 的基础知识，并且总结了几种应用场景。这一讲，我将通过实例的方式，带你逐个学习 Channel 解决这些问题的方法，帮你巩固和完全掌握它的用法。

在开始上课之前，我先补充一个知识点：通过反射的方式执行 select 语句，在处理很多的 case clause，尤其是不定长的 case clause 的时候，非常有用。而且，在后面介绍任务编排的实现时，我也会采用这种方法，所以，我先带你具体学习下 Channel 的反射用法。



使用反射操作 Channel

select 语句可以处理 chan 的 send 和 recv, send 和 recv 都可以作为 case clause。如果我们同时处理两个 chan, 就可以写成下面的样子:

[复制代码](#)

```
1  select {
2  case v := <-ch1:
3      fmt.Println(v)
4  case v := <-ch2:
5      fmt.Println(v)
6  }
```

如果需要处理三个 chan, 你就可以再添加一个 case clause, 用它来处理第三个 chan。可是, 如果要处理 100 个 chan 呢? 一万个 chan 呢?

或者是, chan 的数量在编译的时候是不定的, 在运行的时候需要处理一个 slice of chan, 这个时候, 也没有办法在编译前写成字面意义的 select。那该怎么办?

这个时候, 就要“祭”出我们的反射大法了。

通过 reflect.Select 函数, 你可以将一组运行时的 case clause 传入, 当作参数执行。Go 的 select 是伪随机的, 它可以在执行的 case 中随机选择一个 case, 并把选择的这个 case 的索引 (chosen) 返回, 如果没有可用的 case 返回, 会返回一个 bool 类型的返回值, 这个返回值用来表示是否有 case 成功被选择。如果是 recv case, 还会返回接收的元素。Select 的方法签名如下:


[复制代码](#)

```
1 func Select(cases []SelectCase) (chosen int, recv Value, recvOK bool)
```

下面, 我来借助一个例子, 来演示一下, 动态处理两个 chan 的情形。因为这样的方式可以动态处理 case 数据, 所以, 你可以传入几百几千几万的 chan, 这就解决了不能动态处理 n 个 chan 的问题。

首先, createCases 函数分别为每个 chan 生成了 recv case 和 send case, 并返回一个 reflect.SelectCase 数组。

然后，通过一个循环 10 次的 for 循环执行 reflect.Select，这个方法会从 cases 中选择一个 case 执行。第一次肯定是 send case，因为此时 chan 还没有元素，recv 还不可用。等 chan 中有了数据以后，recv case 就可以被选择了。这样，你就可以处理不定数量的 chan 了。

 复制代码

```
1 func main() {
2     var ch1 = make(chan int, 10)
3     var ch2 = make(chan int, 10)
4
5     // 创建SelectCase
6     var cases = createCases(ch1, ch2)
7
8     // 执行10次select
9     for i := 0; i < 10; i++ {
10        chosen, recv, ok := reflect.Select(cases)
11        if recv.IsValid() { // recv case
12            fmt.Println("recv:", cases[chosen].Dir, recv, ok)
13        } else { // send case
14            fmt.Println("send:", cases[chosen].Dir, ok)
15        }
16    }
17 }
18
19 func createCases(chs ...chan int) []reflect.SelectCase {
20     var cases []reflect.SelectCase
21
22
23     // 创建recv case
24     for _, ch := range chs {
25         cases = append(cases, reflect.SelectCase{
26             Dir: reflect.SelectRecv,
27             Chan: reflect.ValueOf(ch),
28         })
29     }
30
31     // 创建send case
32     for i, ch := range chs {
33         v := reflect.ValueOf(i)
34         cases = append(cases, reflect.SelectCase{
35             Dir: reflect.SelectSend,
36             Chan: reflect.ValueOf(ch),
37             Send: v,
38         })
39     }
40
41     return cases
42 }
```

典型的应用场景

了解刚刚的反射用法，我们就解决了今天的基础知识问题，接下来，我就带你具体学习下 Channel 的应用场景。

首先来看消息交流。

消息交流

从 chan 的内部实现看，它是以一个循环队列的方式存放数据，所以，它有时候也会被当成线程安全的队列和 buffer 使用。一个 goroutine 可以安全地往 Channel 中塞数据，另外一个 goroutine 可以安全地从 Channel 中读取数据，goroutine 就可以安全地实现信息交流了。

我们来看几个例子。

第一个例子是 worker 池的例子。Marcio Castilho 在 [使用 Go 每分钟处理百万请求](#) 这篇文章中，就介绍了他们应对大并发请求的设计。他们将用户的请求放在一个 chan Job 中，这个 chan Job 就相当于一个待处理任务队列。除此之外，还有一个 chan chan Job 队列，用来存放可以处理任务的 worker 的缓存队列。

dispatcher 会把待处理任务队列中的任务放到一个可用的缓存队列中，worker 会一直处理它的缓存队列。通过使用 Channel，实现了一个 worker 池的任务处理中心，并且解耦了前端 HTTP 请求处理和后端任务处理的逻辑。

我在讲 Pool 的时候，提到了一些第三方实现的 worker 池，它们全部都是通过 Channel 实现的，这是 Channel 的一个常见的应用场景。worker 池的生产者和消费者的消息交流都是通过 Channel 实现的。

第二个例子是 etcd 中的 node 节点的实现，包含大量的 chan 字段，比如 recvc 是消息处理的 chan，待处理的 protobuf 消息都扔到这个 chan 中，node 有一个专门的 run goroutine 处理这些消息。

```

252     type node struct {
253         propc      chan msgWithResult
254         recvc       chan pb.Message
255         confc       chan pb.ConfChangeV2
256         confstatec  chan pb.ConfState
257         readyc      chan Ready
258         advancec    chan struct{}
259         tickc       chan struct{}
260         done        chan struct{}
261         stop        chan struct{}
262         status      chan chan Status
263
264         rn *RawNode
265     }
266

```


数据传递

“击鼓传花”的游戏很多人都玩过，花从一个人手中传给另外一个人，就有点类似流水线的操作。这个花就是数据，花在游戏者之间流转，这就类似编程中的数据传递。

还记得上节课我给你留了一道任务编排的题吗？其实它就可以用数据传递的方式实现。

有 4 个 goroutine，编号为 1、2、3、4。每秒钟会有一个 goroutine 打印出它自己的编号，要求你编写程序，让输出的编号总是按照 1、2、3、4、1、2、3、4.....这个顺序打印出来。

为了实现顺序的数据传递，我们可以定义一个令牌的变量，谁得到令牌，谁就可以打印一次自己的编号，同时将令牌**传递**给下一个 goroutine，我们尝试使用 chan 来实现，可以看下下面的代码。

 复制代码

```

1  type Token struct{}
2
3  func newWorker(id int, ch chan Token, nextCh chan Token) {
4      for {
5          token := <-ch           // 取得令牌
6          fmt.Println((id + 1)) // id从1开始

```

```
7         time.Sleep(time.Second)
8         nextCh <- token
9     }
10 }
11 func main() {
12     chs := []chan Token{make(chan Token), make(chan Token), make(chan Token),
13
14     // 创建4个worker
15     for i := 0; i < 4; i++ {
16         go newWorker(i, chs[i], chs[(i+1)%4])
17     }
18
19     //首先把令牌交给第一个worker
20     chs[0] <- struct{}{}
21
22     select {}
23 }
```

我来给你具体解释下这个实现方式。

首先，我们定义一个令牌类型（Token），接着定义一个创建 worker 的方法，这个方法会从它自己的 chan 中读取令牌。哪个 goroutine 取得了令牌，就可以打印出自己编号，因为需要每秒打印一次数据，所以，我们让它休眠 1 秒后，再把令牌交给它的下家。

接着，在第 16 行启动每个 worker 的 goroutine，并在第 20 行将令牌先交给第一个 worker。

如果你运行这个程序，就会在命令行中看到每一秒就会输出一个编号，而且编号是以 1、2、3、4 这样的顺序输出的。

这类场景有一个特点，就是当前持有数据的 goroutine 都有一个信箱，信箱使用 chan 实现，goroutine 只需要关注自己的信箱中的数据，处理完毕后，就把结果发送到下一家的信箱中。

信号通知

chan 类型有这样一个特点：chan 如果为空，那么，receiver 接收数据的时候就会阻塞等待，直到 chan 被关闭或者有新的数据到来。利用这个机制，我们可以实现 wait/notify 的设计模式。

传统的并发原语 Cond 也能实现这个功能，但是，Cond 使用起来比较复杂，容易出错，而使用 chan 实现 wait/notify 模式就方便很多了。

除了正常的业务处理时的 wait/notify，我们经常碰到的一个场景，就是程序关闭的时候，我们需要在退出之前做一些清理（doCleanup 方法）的动作。这个时候，我们经常要使用 chan。

比如，使用 chan 实现程序的 graceful shutdown，在退出之前执行一些连接关闭、文件 close、缓存落盘等一些动作。


 复制代码

```
1 func main() {
2     go func() {
3         ..... // 执行业务处理
4     }()
5
6     // 处理CTRL+C等中断信号
7     termChan := make(chan os.Signal)
8     signal.Notify(termChan, syscall.SIGINT, syscall.SIGTERM)
9     <-termChan
10
11    // 执行退出之前的清理动作
12    doCleanup()
13
14    fmt.Println("优雅退出")
15 }
```

有时候，doCleanup 可能是一个很耗时的操作，比如十几分钟才能完成，如果程序退出需要等待这么长时间，用户是不能接受的，所以，在实践中，我们需要设置一个最长的等待时间。只要超过了这个时间，程序就不再等待，可以直接退出。所以，退出的时候分为两个阶段：

1. closing，代表程序退出，但是清理工作还没做；
2. closed，代表清理工作已经做完。

所以，上面的例子可以改写如下：

 复制代码

```
1 func main() {
2     var closing = make(chan struct{})
3     var closed = make(chan struct{})
4
5     go func() {
6         // 模拟业务处理
7         for {
8             select {
9                 case <-closing:
10                    return
11                default:
12                    // ..... 业务计算
13                    time.Sleep(100 * time.Millisecond)
14            }
15        }
16    }()
17
18    // 处理CTRL+C等中断信号
19    termChan := make(chan os.Signal)
20    signal.Notify(termChan, syscall.SIGINT, syscall.SIGTERM)
21    <-termChan
22
23    close(closing)
24    // 执行退出之前的清理动作
25    go doCleanup(closed)
26
27    select {
28        case <-closed:
29        case <-time.After(time.Second):
30            fmt.Println("清理超时，不等了")
31        }
32    }
33    fmt.Println("优雅退出")
34 }
35
36 func doCleanup(closed chan struct{}) {
37     time.Sleep((time.Minute))
38     close(closed)
```


锁

使用 chan 也可以实现互斥锁。

在 chan 的内部实现中，就有一把互斥锁保护着它的所有字段。从外在表现上，chan 的发送和接收之间也存在着 happens-before 的关系，保证元素放进去之后，receiver 才能读取到（关于 happens-before 的关系，是指事件发生的先后顺序关系，我会在下一讲详细介绍，这里你只需要知道它是一种描述事件先后顺序的方法）。

要想使用 chan 实现互斥锁，至少有两种方式。一种方式是先初始化一个 capacity 等于 1 的 Channel，然后再放入一个元素。这个元素就代表锁，谁取得了这个元素，就相当于获取了这把锁。另一种方式是，先初始化一个 capacity 等于 1 的 Channel，它的“空槽”代表锁，谁能成功地把元素发送到这个 Channel，谁就获取了这把锁。

这是使用 Channel 实现锁的两种不同实现方式，我重点介绍下第一种。理解了这种实现方式，第二种方式也就很容易掌握了，我就不多说了。

 复制代码

```
1 // 使用chan实现互斥锁
2 type Mutex struct {
3     ch chan struct{}
4 }
5
6 // 使用锁需要初始化
7 func NewMutex() *Mutex {
8     mu := &Mutex{make(chan struct{}, 1)}
9     mu.ch <- struct{}{}
10    return mu
11 }
12
13 // 请求锁，直到获取到
14 func (m *Mutex) Lock() {
15     <-m.ch
16 }
17
18 // 解锁
19 func (m *Mutex) Unlock() {
20     select {
21     case m.ch <- struct{}{}:
22     default:
23         panic("unlock of unlocked mutex")
24     }
25 }
26
27 // 尝试获取锁
28 func (m *Mutex) TryLock() bool {
29     select {
30     case <-m.ch:
31         return true
32     default:
33     }
34     return false
35 }
36
37 // 加入一个超时的设置
38 func (m *Mutex) LockTimeout(timeout time.Duration) bool {
```

```
39     timer := time.NewTimer(timeout)
40     select {
41     case <-m.ch:
42         timer.Stop()
43         return true
44     case <-timer.C:
45     }
46     return false
47 }
48
49 // 锁是否已被持有
50 func (m *Mutex) IsLocked() bool {
51     return len(m.ch) == 0
52 }
53
54
55 func main() {
56     m := NewMutex()
57     ok := m.TryLock()
58     fmt.Printf("locked v %v\n", ok)
59     ok = m.TryLock()
60     fmt.Printf("locked %v\n", ok)
61 }
```

你可以用 buffer 等于 1 的 chan 实现互斥锁，在初始化这个锁的时候往 Channel 中先塞入一个元素，谁把这个元素取走，谁就获取了这把锁，把元素放回去，就是释放了锁。元素在放回到 chan 之前，不会有 goroutine 能从 chan 中取出元素的，这就保证了互斥性。

在这段代码中，还有一点需要我们注意下：利用 select+chan 的方式，很容易实现 TryLock、Timeout 的功能。具体来说就是，在 select 语句中，我们可以使用 default 实现 TryLock，使用一个 Timer 来实现 Timeout 的功能。

任务编排

前面所说的消息交流的场景是一个特殊的任务编排的场景，这个“击鼓传花”的模式也被称为流水线模式。

在🔗第 6 讲，我们学习了 WaitGroup，我们可以利用它实现等待模式：启动一组 goroutine 执行任务，然后等待这些任务都完成。其实，我们也可以使用 chan 实现 WaitGroup 的功能。这个比较简单，我就不举例子了，接下来我介绍几种更复杂的编排模式。

这里的编排既指安排 goroutine 按照指定的顺序执行，也指多个 chan 按照指定的方式组合处理的方式。goroutine 的编排类似“击鼓传花”的例子，我们通过编排数据在 chan 之间的流转，就可以控制 goroutine 的执行。接下来，我来重点介绍下多个 chan 的编排方式，总共 5 种，分别是 Or-Done 模式、扇入模式、扇出模式、Stream 和 Map-Reduce。


Or-Done 模式

首先来看 Or-Done 模式。Or-Done 模式是信号通知模式中更宽泛的一种模式。这里提到了“信号通知模式”，我先来解释一下。

我们会使用“信号通知”实现某个任务执行完成后的通知机制，在实现时，我们为此任务定义一个类型为 `chan struct{}` 类型的 `done` 变量，等任务结束后，我们就可以 `close` 这个变量，然后，其它 receiver 就会收到这个通知。

这是有一个任务的情况，如果有多个任务，只要有任意一个任务执行完，我们就想获得这个信号，这就是 Or-Done 模式。

比如，你发送同一个请求到多个微服务节点，只要任意一个微服务节点返回结果，就算成功，这个时候，就可以参考下面的实现：

 复制代码

```
1 func or(channels ...<-chan interface{}) <-chan interface{} {
2     // 特殊情况，只有零个或者1个chan
3     switch len(channels) {
4     case 0:
5         return nil
6     case 1:
7         return channels[0]
8     }
9
10    orDone := make(chan interface{})
11    go func() {
12        defer close(orDone)
13
14        switch len(channels) {
15        case 2: // 2个也是一种特殊情况
16            select {
17            case <-channels[0]:
18            case <-channels[1]:
19            }
20
```

```
21     default: //超过两个, 二分法递归处理
22         m := len(channels) / 2
23         select {
24             case <-or(channels[:m]...):
25             case <-or(channels[m:]...):
26         }
27     }
28 }()
29
30 return orDone
```

我们可以写一个测试程序测试它：

 复制代码

```
1 func sig(after time.Duration) <-chan interface{} {
2     c := make(chan interface{})
3     go func() {
4         defer close(c)
5         time.Sleep(after)
6     }()
7     return c
8 }
9
10
11 func main() {
12     start := time.Now()
13
14     <-or(
15         sig(10*time.Second),
16         sig(20*time.Second),
17         sig(30*time.Second),
18         sig(40*time.Second),
19         sig(50*time.Second),
20         sig(01*time.Minute),
21     )
22
23     fmt.Printf("done after %v", time.Since(start))
24 }
```

这里的实现使用了一个巧妙的方式，当 **chan** 的数量大于 2 时，使用递归的方式等待信号。

在 **chan** 数量比较多的情况下，递归并不是一个很好的解决方式，根据这一讲最开始介绍的反射的方法，我们也可以实现 Or-Done 模式：

```
1 func or(channels ...<-chan interface{}) <-chan interface{} {
2     //特殊情况, 只有0个或者1个
3     switch len(channels) {
4     case 0:
5         return nil
6     case 1:
7         return channels[0]
8     }
9
10    orDone := make(chan interface{})
11    go func() {
12        defer close(orDone)
13        // 利用反射构建SelectCase
14        var cases []reflect.SelectCase
15        for _, c := range channels {
16            cases = append(cases, reflect.SelectCase{
17                Dir:  reflect.SelectRecv,
18                Chan: reflect.ValueOf(c),
19            })
20        }
21
22        // 随机选择一个可用的case
23        reflect.Select(cases)
24    }()
25
26
27    return orDone
28 }
```

这是递归和反射两种方法实现 Or-Done 模式的代码。反射方式避免了深层递归的情况，可以处理有大量 chan 的情况。其实最笨的一种方法就是为每一个 Channel 启动一个 goroutine，不过这会启动非常多的 goroutine，太多的 goroutine 会影响性能，所以不太常用。你只要知道这种用法就行了，不用重点掌握。

扇入模式

扇入借鉴了数字电路的概念，它定义了单个逻辑门能够接受的数字信号输入最大量的术语。一个逻辑门可以有多个输入，一个输出。

在软件工程中，模块的扇入是指有多少个上级模块调用它。而对于我们这里的 Channel 扇入模式来说，就是指有多个源 Channel 输入、一个目的 Channel 输出的情况。扇入比就是源 Channel 数量比 1。

每个源 Channel 的元素都会发送给目标 Channel，相当于目标 Channel 的 receiver 只需要监听目标 Channel，就可以接收所有发送给源 Channel 的数据。

扇入模式也可以使用反射、递归，或者是最笨的每个 goroutine 处理一个 Channel 的方式来实现。

这里我列举下递归和反射的方式，帮你加深一下对这个技巧的理解。

反射的代码比较简短，易于理解，主要就是构造出 SelectCase slice，然后传递给 reflect.Select 语句。

[复制代码](#)

```
1 func fanInReflect(chans ...<-chan interface{}) <-chan interface{} {
2     out := make(chan interface{})
3     go func() {
4         defer close(out)
5         // 构造SelectCase slice
6         var cases []reflect.SelectCase
7         for _, c := range chans {
8             cases = append(cases, reflect.SelectCase{
9                 Dir:  reflect.SelectRecv,
10                Chan: reflect.ValueOf(c),
11            })
12        }
13
14        // 循环，从cases中选择一个可用的
15        for len(cases) > 0 {
16            i, v, ok := reflect.Select(cases)
17            if !ok { // 此channel已经close
18                cases = append(cases[:i], cases[i+1:]...)
19                continue
20            }
21            out <- v.Interface()
22        }
23    }()
24    return out
25 }
```

递归模式也是在 Channel 大于 2 时，采用二分法递归 merge。

[复制代码](#)

```
1 func fanInRec(chans ...<-chan interface{}) <-chan interface{} {
```



```
2     switch len(chans) {
3     case 0:
4         c := make(chan interface{})
5         close(c)
6         return c
7     case 1:
8         return chans[0]
9     case 2:
10        return mergeTwo(chans[0], chans[1])
11    default:
12        m := len(chans) / 2
13        return mergeTwo(
14            fanInRec(chans[:m]...),
15            fanInRec(chans[m:]...))
16    }
17
```

这里有一个 mergeTwo 的方法，是将两个 Channel 合并成一个 Channel，是扇入形式的一种特例（只处理两个 Channel）。下面我来借助一段代码帮你理解下这个方法。

[复制代码](#)

```
1 func mergeTwo(a, b <-chan interface{}) <-chan interface{} {
2     c := make(chan interface{})
3     go func() {
4         defer close(c)
5         for a != nil || b != nil { //只要还有可读的chan
6             select {
7                 case v, ok := <-a:
8                     if !ok { // a 已关闭, 设置为nil
9                         a = nil
10                        continue
11                    }
12                    c <- v
13                case v, ok := <-b:
14                    if !ok { // b 已关闭, 设置为nil
15                        b = nil
16                        continue
17                    }
18                    c <- v
19            }
20        }
21    }()
22    return c
23 }
```

扇出模式

有扇入模式，就有扇出模式，扇出模式是和扇入模式相反的。

扇出模式只有一个输入源 Channel，有多个目标 Channel，扇出比就是 1 比目标 Channel 数的值，经常用在设计模式中的 [观察者模式](#) 中（观察者设计模式定义了对象间的一种一对多的组合关系。这样一来，一个对象的状态发生变化时，所有依赖于它的对象都会得到通知并自动刷新）。在观察者模式中，数据变动后，多个观察者都会收到这个变更信号。

下面是一个扇出模式的实现。从源 Channel 取出一个数据后，依次发送给目标 Channel。在发送给目标 Channel 的时候，可以同步发送，也可以异步发送：

[复制代码](#)

```
1 func fanOut(ch <-chan interface{}, out []chan interface{}, async bool) {
2     go func() {
3         defer func() { //退出时关闭所有的输出chan
4             for i := 0; i < len(out); i++ {
5                 close(out[i])
6             }
7         }()
8
9         for v := range ch { // 从输入chan中读取数据
10             v := v
11             for i := 0; i < len(out); i++ {
12                 i := i
13                 if async { //异步
14                     go func() {
15                         out[i] <- v // 放入到输出chan中,异步方式
16                     }()
17                 } else {
18                     out[i] <- v // 放入到输出chan中, 同步方式
19                 }
20             }
21         }
22     }()
23 }
```

你也可以尝试使用反射的方式来实现，我就不列相关代码了，希望你课后可以自己思考下。

Stream

这里我来介绍一种把 Channel 当作流式管道使用的方式，也就是把 Channel 看作流 (Stream)，提供跳过几个元素，或者是只取其中的几个元素等方法。

首先，我们提供创建流的方法。这个方法把一个数据 slice 转换成流：

[复制代码](#)

```
1 func asStream(done <-chan struct{}, values ...interface{}) <-chan interface{}
2     s := make(chan interface{}) //创建一个unbuffered的channel
3     go func() { // 启动一个goroutine, 往s中塞数据
4         defer close(s) // 退出时关闭chan
5         for _, v := range values { // 遍历数组
6             select {
7                 case <-done:
8                     return
9                 case s <- v: // 将数组元素塞入到chan中
10            }
11        }
12    }()
13    return s
14 }
```

流创建好以后，该咋处理呢？下面我再给你介绍下实现流的方法。

1. takeN：只取流中的前 n 个数据；
2. takeFn：筛选流中的数据，只保留满足条件的数据；
3. takeWhile：只取前面满足条件的数据，一旦不满足条件，就不再取；
4. skipN：跳过流中前几个数据；
5. skipFn：跳过满足条件的数据；
6. skipWhile：跳过前面满足条件的数据，一旦不满足条件，当前这个元素和以后的元素都会输出给 Channel 的 receiver。

这些方法的实现很类似，我们以 takeN 为例来具体解释一下。

[复制代码](#)

```
1 func takeN(done <-chan struct{}, valueStream <-chan interface{}, num int) <-ch
2     takeStream := make(chan interface{}) // 创建输出流
3     go func() {
4         defer close(takeStream)
5         for i := 0; i < num; i++ { // 只读取前num个元素
6             select {
7                 case <-done:
8                     return
```

```
9         case takeStream <- <-valueStream: //从输入流中读取元素
10     }
11 }
12 }()
13 return takeStream
14
```

Map-Reduce


map-reduce 是一种处理数据的方式，最早是由 Google 公司研究提出的一种面向大规模数据处理的并行计算模型和方法，开源的版本是 hadoop，前几年比较火。

不过，我要讲的并不是分布式的 map-reduce，而是单机单进程的 map-reduce 方法。

map-reduce 分为两个步骤，第一步是映射（map），处理队列中的数据，第二步是规约（reduce），把列表中的每一个元素按照一定的处理方式处理成结果，放入到结果队列中。

就像做汉堡一样，map 就是单独处理每一种食材，reduce 就是从每一份食材中取一部分，做成一个汉堡。

我们先来看下 map 函数的处理逻辑:

 复制代码

```
1 func mapChan(in <-chan interface{}, fn func(interface{}) interface{}) <-chan i
2     out := make(chan interface{}) //创建一个输出chan
3     if in == nil { // 异常检查
4         close(out)
5         return out
6     }
7
8     go func() { // 启动一个goroutine,实现map的主要逻辑
9         defer close(out)
10        for v := range in { // 从输入chan读取数据，执行业务操作，也就是map操作
11            out <- fn(v)
12        }
13    }()
14
15    return out
16 }
```

reduce 函数的处理逻辑如下:

[复制代码](#)

```
1 func reduce(in <-chan interface{}, fn func(r, v interface{}) interface{}) interface{} {
2     if in == nil { // 异常检查
3         return nil
4     }
5
6     out := <-in // 先读取第一个元素
7     for v := range in { // 实现reduce的主要逻辑
8         out = fn(out, v)
9     }
10
11     return out
12 }
```

我们可以写一个程序, 这个程序使用 map-reduce 模式处理一组整数, map 函数就是为每个整数乘以 10, reduce 函数就是把 map 处理的结果累加起来:

[复制代码](#)

```
1 // 生成一个数据流
2 func asStream(done <-chan struct{}) <-chan interface{} {
3     s := make(chan interface{})
4     values := []int{1, 2, 3, 4, 5}
5     go func() {
6         defer close(s)
7         for _, v := range values { // 从数组生成
8             select {
9                 case <-done:
10                    return
11                 case s <- v:
12
13             }
14         }()
15     }()
16     return s
17 }
18 func main() {
19     in := asStream(nil)
20
21     // map操作: 乘以10
22     mapFn := func(v interface{}) interface{} {
23         return v.(int) * 10
24     }
25
26     // reduce操作: 对map的结果进行累加
```

```
27     reduceFn := func(r, v interface{}) interface{} {  
28         return r.(int) + v.(int)  
29     }  
30  
31     sum := reduce(mapChan(in, mapFn), reduceFn) //返回累加结果  
32     fmt.Println(sum)  
33
```

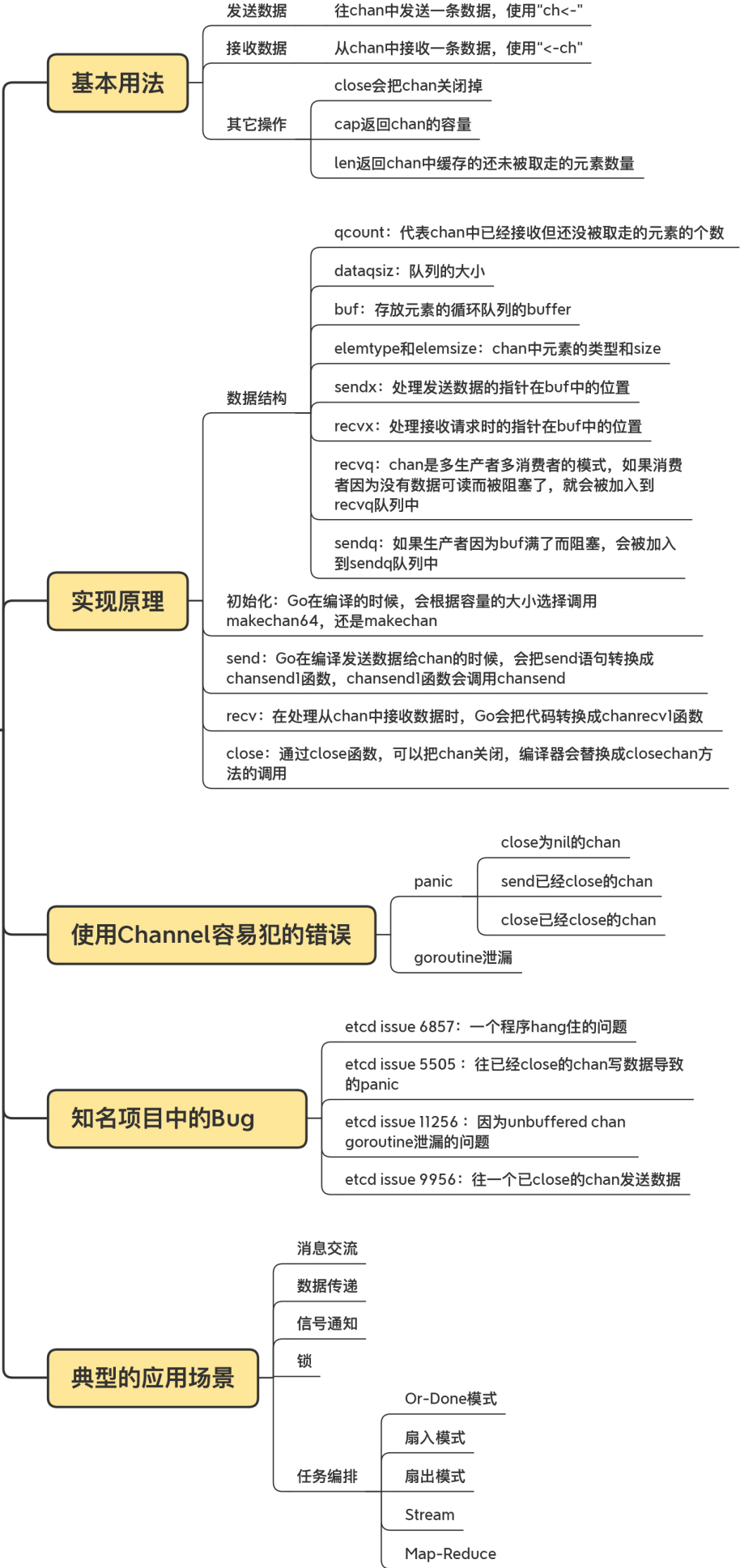
总结

这节课，我借助代码示例，带你学习了 Channel 的应用场景和应用模式。这几种模式不是我们学习的终点，而是学习的起点。掌握了这几种模式之后，我们可以延伸出更多的模式。

虽然 Channel 最初是基于 CSP 设计的用于 goroutine 之间的消息传递的一种数据类型，但是，除了消息传递这个功能之外，大家居然还演化出了各式各样的应用模式。我不确定 Go 的创始人在设计这个类型的时候，有没有想到这一点，但是，我确实被各位大牛利用 Channel 的各种点子折服了，比如有人实现了一个基于 TCP 网络的分布式的 Channel。

在使用 Go 开发程序的时候，你也不妨多考虑考虑是否能够使用 chan 类型，看看你是不是也能创造出别具一格的应用模式。

Channel



思考题

想一想，我们在利用 chan 实现互斥锁的时候，如果 buffer 设置的不是 1，而是一个更大的值，会出现什么状况吗？能解决什么问题吗？

欢迎在留言区写下你的思考和答案，我们一起交流讨论。如果你觉得有所收获，也欢迎你把今天的内容分享给你的朋友或同事。

提建议

Go 并发编程实战课

鸟窝带你攻克并发编程难题

晁岳攀 (鸟窝)

前微博技术专家

知名微服务框架 rpcx 的作者



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 13 | Channel: 另辟蹊径，解决并发问题

下一篇 15 | 内存模型：Go如何保证并发读写的顺序？

精选留言 (6)

写留言



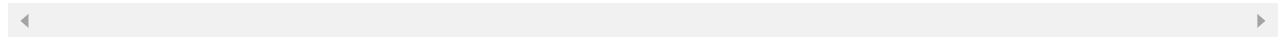
润豪

2020-11-14

channel 来实现互斥锁，优势是 trylock，timeout 吧，因为mutex 没有这些功能。否则的话，是不是用回 mutex 呢

展开 ∨

作者回复: 对。如果不需要这些特性，我的建议是使用mutex



1



myrfy

2020-11-11

老师好，我有两个问题

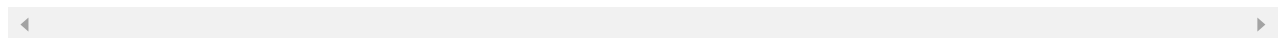
1、关于or done或者fan in模式，我之前在sof上看到过类似的问题，其中的高赞回答是说，启动与ch数量相等的goroutine，每个goroutine监听一个ch并把读到的结果放入一个收集ch的模式效率要比反射高，并且给出了测评数据，现在手机码字，不太好找到。但想和老师确认一下是不是后面go某个版本对反射做了优化呢？...

展开 ∨

作者回复: 我并没有benchmark结果列在这里，凭经验我们也知道反射的效率很低。Francesc Campoy有一篇文章专门做了测试，你可以搜一下。

analyzing the performance of go functions with benchmarks.

异步的方式并不是你所说的目的，而是避免一个out chan阻塞的时候影响其他out



1



党

2020-11-14

反射chan第一次知道，理解起来是有点困难

展开 ∨



Chris

2020-11-11

reflect性能是比较差的，贴一下压测结果：

BenchmarkFanIn-4 382776 3255 ns/op 131 B/op 2 allocs/op

BenchmarkFanInReflect-4 1000000 13168 ns/op 6974 B/op 90 allocs/op

BenchmarkFanInRec-4 280599 5524 ns/op 1009 B/op 27 allocs/op

展开 ▾



青生先森

2020-11-11

对channel有更深入的理解

展开 ▾



那一刻

2020-11-11

在利用 chan 实现互斥锁的时候，如果 buffer 设置的不是 1，而是一个更大的值N

允许最多N个goroutine同时拥有锁，类似Semaphore作用

展开 ▾

