管道channel

单纯地将函数并发执行是没有意义的。函数与函数间需要交换数据才能体现并发执行函数的意义。

虽然可以使用共享内存进行数据交换，但是共享内存在不同的 goroutine 中容易发生竞态问题。为了保证数据交换的正确性，很多并发模型中必须使用互斥量对内存进行加锁，这种做法势必造成性能问题。

Go语言采用的并发模型是`CSP（Communicating Sequential Processes）`，提倡**通过通信共享内存**而不是**通过共享内存而实现通信。**

如果说 goroutine 是Go程序并发的执行体，`channel`就是它们之间的连接。`channel`是可以让一个 goroutine 发送特定值到另一个 goroutine 的通信机制。

Go 语言中的通道（channel）是一种特殊的类型。通道像一个传送带或者队列，总是遵循先入先出（First In First Out）的规则，保证收发数据的顺序。每一个通道都是一个具体类型的导管，也就是声明channel的时候需要为其指定元素类型。

channel类型

`channel`是 Go 语言中一种特有的类型。声明通道类型变量的格式如下：

```
var 变量名称 chan 元素类型
```

其中：

- chan：是关键字
- 元素类型：是指通道中传递元素的类型

举几个例子：

```
var ch1 chan int   // 声明一个传递整型的通道
var ch2 chan bool  // 声明一个传递布尔型的通道
var ch3 chan []int // 声明一个传递int切片的通道
```

channel零值

未初始化的通道类型变量其默认零值是`nil`。

```
var ch chan int
fmt.Println(ch) // <nil>
```

初始化channel

声明的通道类型变量需要使用内置的`make`函数初始化之后才能使用。具体格式如下：

```
make(chan 元素类型, [缓冲大小])
```

其中：

- channel的缓冲大小是可选的。

举几个例子：

```
ch4 := make(chan int)
ch5 := make(chan bool, 1)  // 声明一个缓冲区大小为1的通道
```

channel操作

通道共有发送（send）、接收(receive）和关闭（close）三种操作。而发送和接收操作都使用`<-`符号。

现在我们先使用以下语句定义一个通道：

```
ch := make(chan int)
```

发送

将一个值发送到通道中。

```
ch <- 10 // 把10发送到ch中
```

接收

从一个通道中接收值。

```
x := <- ch // 从ch中接收值并赋值给变量x
<-ch       // 从ch中接收值，忽略结果
```

关闭

我们通过调用内置的`close`函数来关闭通道。

```
close(ch)
```

注意：一个通道值是可以被垃圾回收掉的。通道通常由发送方执行关闭操作，并且只有在接收方明确等待通道关闭的信号时才需要执行关闭操作。它和关闭文件不一样，通常在结束操作之后关闭文件是必须要做的，但关闭通道不是必须的。

关闭后的通道有以下特点：

1. 对一个关闭的通道再发送值就会导致 panic。
2. 对一个关闭的通道进行接收会一直获取值直到通道为空。
3. 对一个关闭的并且没有值的通道执行接收操作会得到对应类型的零值。
4. 关闭一个已经关闭的通道会导致 panic。

无缓冲的通道

无缓冲的通道又称为阻塞的通道。我们来看一下如下代码片段。

```
func main() {
    ch := make(chan int)
    ch <- 10
    fmt.Println("发送成功")
}
```

上面这段代码能够通过编译，但是执行的时候会出现以下错误：

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

goroutine 1 [chan send]:
main.main()
        .../main.go:8 +0x54
```

`deadlock`表示我们程序中的 goroutine 都被挂起导致程序死锁了。为什么会出现`deadlock`错误呢？

因为我们使用`ch := make(chan int)`创建的是无缓冲的通道，无缓冲的通道只有在有接收方能够接收值的时候才能发送成功，否则会一直处于等待发送的阶段。同理，如果对一个无缓冲通道执行接收操作时，没有任何向通道中发送值的操作那么也会导致接收操作阻塞。就像田径比赛中的4x100接力赛，想要完成交棒必须有一个能够接棒的运动员，否则只能等待。简单来说就是无缓冲的通道必须有至少一个接收方才能发送成功。

上面的代码会阻塞在`ch <- 10`这一行代码形成死锁，那如何解决这个问题呢？

其中一种可行的方法是创建一个 goroutine 去接收值，例如：

```
func recv(c chan int) {
    ret := <-c
    fmt.Println("接收成功", ret)
}

func main() {
    ch := make(chan int)
    go recv(ch) // 创建一个 goroutine 从通道接收值
    ch <- 10
    fmt.Println("发送成功")
}
```

首先无缓冲通道`ch`上的发送操作会阻塞，直到另一个 goroutine 在该通道上执行接收操作，这时数字10才能发送成功，两个 goroutine 将继续执行。相反，如果接收操作先执行，接收方所在的 goroutine 将阻塞，直到 main goroutine 中向该通道发送数字10。

使用无缓冲通道进行通信将导致发送和接收的 goroutine 同步化。因此，无缓冲通道也被称为`同步通道`。

有缓冲的通道

还有另外一种解决上面死锁问题的方法，那就是使用有缓冲区的通道。我们可以在使用 make 函数初始化通道时，可以为其指定通道的容量，例如：

```
func main() {
    ch := make(chan int, 1) // 创建一个容量为1的有缓冲区通道
    ch <- 10
    fmt.Println("发送成功")
}
```

只要通道的容量大于零，那么该通道就属于有缓冲的通道，通道的容量表示通道中最大能存放的元素数量。当通道内已有元素数达到最大容量后，再向通道执行发送操作就会阻塞，除非有从通道执行接收操作。就像你小区的快递柜只有那么个多格子，格子满了就装不下了，就阻塞了，等到别人取走一个快递员就能往里面放一个。

我们可以使用内置的`len`函数获取通道内元素的数量，使用`cap`函数获取通道的容量，虽然我们很少会这么做。

多返回值模式

当向通道中发送完数据时，我们可以通过`close`函数来关闭通道。当一个通道被关闭后，再往该通道发送值会引发`panic`，从该通道取值的操作会先取完通道中的值。通道内的值被接收完后再对通道执行接收操作得到的值会一直都是对应元素类型的零值。那我们如何判断一个通道是否被关闭了呢?

对一个通道执行接收操作时支持使用如下多返回值模式。

```
value, ok := <- ch
```

其中:

- value：从通道中取出的值，如果通道被关闭则返回对应类型的零值。
- ok：通道ch关闭时返回 false，否则返回 true。

下面代码片段中的`f2`函数会循环从通道`ch`中接收所有值，直到通道被关闭后退出。

```
func f2(ch chan int) {
    for {
        v, ok := <-ch
        if !ok {
            fmt.Println("通道已关闭")
            break
        }
        fmt.Printf("v:%#v ok:%#v\n", v, ok)
    }
}

func main() {
    ch := make(chan int, 2)
    ch <- 1
    ch <- 2
    close(ch)
    f2(ch)
}
```

for range接收值

通常我们会选择使用`for range`循环从通道中接收值，当通道被关闭后，会在通道内的所有值被接收完毕后会自动退出循环。上面那个示例我们使用`for range`改写后会很简洁。

```
func f3(ch chan int) {
    for v := range ch {
        fmt.Println(v)
    }
}
```

注意：目前Go语言中并没有提供一个不对通道进行读取操作就能判断通道是否被关闭的方法。不能简单的通过`len(ch)`操作来判断通道是否被关闭。

单向通道

在某些场景下我们可能会将通道作为参数在多个任务函数间进行传递，通常我们会选择在不同的任务函数中对通道的使用进行限制，比如限制通道在某个函数中只能执行发送或只能执行接收操作。想象一下，我们现在有`Producer`和`Consumer`两个函数，其中`Producer`函数会返回一个通道，并且会持续将符合条件的数据发送至该通道，并在发送完成后将该通道关闭。而`Consumer`函数的任务是从通道中接收值进行计算，这两个函数之间通过`Processer`函数返回的通道进行通信。完整的示例代码如下。

```go
package main

import (
    "fmt"
)

// Producer 返回一个通道
// 并持续将符合条件的数据发送至返回的通道中
// 数据发送完成后会将返回的通道关闭
func Producer() chan int {
    ch := make(chan int, 2)
    // 创建一个新的goroutine执行发送数据的任务
    go func() {
        for i := 0; i < 10; i++ {
            if i%2 == 1 {
                ch <- i
            }
        }
        close(ch) // 任务完成后关闭通道
    }()

    return ch
}

// Consumer 从通道中接收数据进行计算
func Consumer(ch chan int) int {
    sum := 0
    for v := range ch {
        sum += v
    }
    return sum
}

func main() {
    ch := Producer()

    res := Consumer(ch)
    fmt.Println(res) // 25

}
```

从上面的示例代码中可以看出正常情况下`Consumer`函数中只会对通道进行接收操作，但是这不代表不可以在`Consumer`函数中对通道进行发送操作。作为`Producer`函数的提供者，我们在返回通道的时候可能只希望调用方拿到返回的通道后只能对其进行接收操作。但是我们没有办法阻止在`Consumer`函数中对通道进行发送操作。

Go语言中提供了**单向通道**来处理这种需要限制通道只能进行某种操作的情况。

```
<- chan int // 只接收通道，只能接收不能发送
chan <- int // 只发送通道，只能发送不能接收
```

其中，箭头`<-`和关键字`chan`的相对位置表明了当前通道允许的操作，这种限制将在编译阶段进行检测。另外对一个只接收通道执行close也是不允许的，因为默认通道的关闭操作应该由发送方来完成。

我们使用单向通道将上面的示例代码进行如下改造。

```
// Producer2 返回一个接收通道
func Producer2() <-chan int {
    ch := make(chan int, 2)
    // 创建一个新的goroutine执行发送数据的任务
    go func() {
        for i := 0; i < 10; i++ {
            if i%2 == 1 {
                ch <- i
            }
        }
        close(ch) // 任务完成后关闭通道
    }()

    return ch
}

// Consumer2 参数为接收通道
func Consumer2(ch <-chan int) int {
    sum := 0
    for v := range ch {
        sum += v
    }
    return sum
}

func main() {
    ch2 := Producer2()

    res2 := Consumer2(ch2)
    fmt.Println(res2) // 25
}
```

这一次，`Producer`函数返回的是一个只接收通道，这就从代码层面限制了该函数返回的通道只能进行接收操作，保证了数据安全。很多读者看到这个示例可能会觉着这样的限制是多余的，但是试想一下如果`Producer`函数可以在其他地方被其他人调用，你该如何限制他人不对该通道执行发送操作呢？并且返回限制操作的单向通道也会让代码语义更清晰、更易读。

在函数传参及任何赋值操作中全向通道（正常通道）可以转换为单向通道，但是无法反向转换。

```go
var ch3 = make(chan int, 1)
ch3 <- 10
close(ch3)
Consumer2(ch3) // 函数传参时将ch3转为单向通道

var ch4 = make(chan int, 1)
ch4 <- 10
var ch5 <-chan int // 声明一个只接收通道ch5
ch5 = ch4          // 变量赋值时将ch4转为单向通道
<-ch5
```

总结

下面的表格中总结了对不同状态下的通道执行相应操作的结果。

通道操作结果表				
操作 \ 状态	nil	没值	有值	满
发送	阻塞	发送成功	发送成功	阻塞
接收	阻塞	阻塞	接收成功	接收成功
关闭	panic	关闭成功	关闭成功	关闭成功

注意：对已经关闭的通道再执行 close 也会引发 panic。

select多路复用

在某些场景下我们可能需要同时从多个通道接收数据。通道在接收数据时，如果没有数据可以被接收那么当前 goroutine 将会发生阻塞。你也许会写出如下代码尝试使用遍历的方式来实现从多个通道中接收值。

```go
for{
    // 尝试从ch1接收值
    data, ok := <-ch1
    // 尝试从ch2接收值
    data, ok := <-ch2
    …
}
```

这种方式虽然可以实现从多个通道接收值的需求，但是程序的运行性能会差很多。Go 语言内置了`select`关键字，使用它可以同时响应多个通道的操作。

Select 的使用方式类似于之前学到的 switch 语句，它也有一系列 case 分支和一个默认的分支。每个 case 分支会对应一个通道的通信（接收或发送）过程。select 会一直等待，直到其中的某个 case 的通信操作完成时，就会执行该 case 分支对应的语句。具体格式如下：

```
select {
case <-ch1:
    //...
case data := <-ch2:
    //...
case ch3 <- 10:
    //...
default:
    //默认操作
}
```

Select 语句具有以下特点。

- 可处理一个或多个 channel 的发送/接收操作。
- 如果多个 case 同时满足，select 会**随机**选择一个执行。
- 对于没有 case 的 select 会一直阻塞，可用于阻塞 main 函数，防止退出。

下面的示例代码能够在终端打印出10以内的奇数，我们借助这个代码片段来看一下 select 的具体使用。

```
package main

import "fmt"

func main() {
    ch := make(chan int, 1)
    for i := 1; i <= 10; i++ {
        select {
        case x := <-ch:
            fmt.Println(x)
        case ch <- i:
        }
    }
}
```

上面的代码输出内容如下。

```
1
3
5
7
9
```

示例中的代码首先是创建了一个缓冲区大小为1的通道 ch，进入 for 循环后：

- 第一次循环时 i = 1，select 语句中包含两个 case 分支，此时由于通道中没有值可以接收，所以`x := <-ch` 这个 case 分支不满足，而 `ch <- i` 这个分支可以执行，会把1发送到通道中，结束本次 for 循环；
- 第二次 for 循环时，i = 2，由于通道缓冲区已满，所以`ch <- i` 这个分支不满足，而`x := <-ch` 这个分支可以执行，从通道接收值1并赋值给变量 x ，所以会在终端打印出 1；

- 后续的 for 循环以此类推会依次打印出3、5、7、9。

通道误用示例

接下来，我们将展示两个因误用通道导致程序出现 bug 的代码片段，希望能够加深读者对通道操作的印象。

示例1

各位读者可以查看以下示例代码，尝试找出其中存在的问题。

```
// demo1 通道误用导致的bug
func demo1() {
    wg := sync.WaitGroup{}

    ch := make(chan int, 10)
    for i := 0; i < 10; i++ {
        ch <- i
    }
    close(ch)

    wg.Add(3)
    for j := 0; j < 3; j++ {
        go func() {
            for {
                task := <-ch
                // 这里假设对接收的数据执行某些操作
                fmt.Println(task)
            }
            wg.Done()
        }()
    }
    wg.Wait()
}
```

将上述代码编译执行后，匿名函数所在的 goroutine 并不会按照预期在通道被关闭后退出。因为 `task := <- ch` 的接收操作在通道被关闭后会一直接收到零值，而不会退出。此处的接收操作应该使用 `task, ok := <- ch`，通过判断布尔值 `ok` 为假时退出；或者使用select 来处理通道。

示例2

阅读下方代码片段，尝试找出其中存在的问题。

```
// demo2 通道误用导致的bug
func demo2() {
    ch := make(chan string)
    go func() {
        // 这里假设执行一些耗时的操作
        time.Sleep(3 * time.Second)
        ch <- "job result"
    }()

    select {
    case result := <-ch:
        fmt.Println(result)
```

```
    case <-time.After(time.Second): // 较小的超时时间
        return
    }
}
```

上述代码片段可能导致 goroutine 泄露（goroutine 并未按预期退出并销毁）。由于 select 命中了超时逻辑，导致通道没有消费者（无接收操作），而其定义的通道为无缓冲通道，因此 goroutine 中的 `ch <- "job result"` 操作会一直阻塞，最终导致 goroutine 泄露。