# Table of Contents


前言 1.1

Part I：语法讲解 1.2

- 并发编程简介 1.2.1
- goroutine 和 channel 1.2.2
  - 你的第一个 go 并发程序 1.2.2.1
  - 等待 goroutine 结束 1.2.2.2
  - goroutine 通过 channel 通信 1.2.2.3
  - 关闭 channel 1.2.2.4
  - 带缓冲的 channel 1.2.2.5
  - for 和 select 和 channel 的结合 1.2.2.6
- 共享变量和锁机制 1.2.3
  - 竞争条件 1.2.3.1
  - 互斥锁 1.2.3.2
  - 读写锁 1.2.3.3
- 总结 1.2.4

Part II：实例解析 1.3

- go syncMap 是如何实现并发访问的 1.3.1

part III：内部原理 1.4

# go 语言并发编程

这本书主要介绍 go 语言的并发编程，分为基本语法、实战练习和原理剖析三个部分。

基本语法主要介绍 goroutine、sync 标准库 和 channel 等概念的语法和使用；实战练习选择多个适合并发的实际问题，使用 go 并发来解决； 原理解析则从 go 语言实现的角度分析 goroutine 是如何创建、调度和管理的。

## 谁适合阅读这本书

本书属于中高级内容，目标是对 go 语言有一定基础的读者，不会介绍 go 语言的基础语法，而是选择 go 并发这个话题进行深入地分析。 因此需要读者已经拥有以下知识：

- 熟悉 go 语言的基本语法
- 知道如何组织 go 语言代码，了解 `GOPATH` 的使用
- 知道如何编译和运行 go 语言程序

## 联系作者

因为个人能力有限，难免会有错误，如果发现问题，请联系作者：

- 个人网站： http://cizixs.com/about
- Email: cizixs#163.com

# Part I：语法讲解

# 并发编程简介

> Performance is a Feature.

性能一直是软件开发不断追求的目标，因为没有人喜欢和一个缓慢的系统或者应用打交道。有很多研究都会告诉你，如果网站的响应速度越慢，用户的满意度也就越低，更多用户也会因此选择其他产品。

在软件开发领域，提高性能有两种方法：购买更多的硬件，和充分利用已有硬件的计算能力。 两者都是有效的手段，并没有本质上的优劣之分。我们经常能听到，在业务快速发展的阶段， 公司为此大肆购买服务器的事情，这是很多互联网公司的标准做法。购买服务器并不是没有技术 含量的事情，维护服务器集群、保证程序逻辑能够无限水平扩展需要非常复杂的技术。

因为摩尔定律的存在，每过一段时间，硬件性能的性能就会翻翻，同时硬件的价格还会降低。 对于程序来说，每过一段时间，它就能运行在更快 CPU 和 内存、更快更多硬盘的机器上， 可以说，程序的性能会随着时间自动提高。

但是进入到后摩尔定律时代，硬件的性能不再快速提高，价格也不再快速下降，我们也就不能 继续享受免费的午餐。基于此，充分利用硬件的计算能力这个方法变得越来越重要，而并发 编程就是实现这一目标的最有效手段。

换一个角度来说，编程一直都是对现实世界的描述和模拟。过程式编程可以看做我们思考问题和 解决问题的步骤的模拟；面向对象编程可以看做对现实世界物体状态和行为的模拟。 而现实世界是并发运行的：每个生物都同时存在，有独立的行为； 国家之间虽然有巨大差异，但也是互相同时运行良好； 经济行为都是频繁并快速地同时发生……当然，小的个体也会有等待和阻塞，比如去买咖啡要排队、 驾驶汽车会堵车、处理事务需要等待审核等等。但是并发才是常态，这些都是异常的不希望的行为。 而且我们还会在等待的时候做其他事情，比如排队的时候和朋友聊天或者刷手机、堵车的时候听听 音乐、等待审核期间去忙工作等，减少等待行为造成的时间浪费。

因此作为对世界的模拟，程序追求并发运行本身就是非常合理的事情。

# 并发与并行

在编程领域，有时候我们需要区分并发（concurrency）和并行（parallel）的概念。

并行强调的是在同一时间有多个实体在运行，比如代码运行在不同的 CPU 上； 而并发是指同时管理多个任务，并不要求这些任务同时在执行。

> Parallelism is about doing a lot of things at once. Concurrency is about managing a lot of things at one.

举个生活中的例子，我们每天有很多事情要处理：接发邮件、开会、撰写文稿..... 我们会说自己同时忙很多事情，但具体到每个时间点，我们只能在做其中一件事情，只是在它们之间不断切换，这就是并发；公司举办派对，找来很多同事来布置，有些人贴横幅、有些人预定食物、 有些人准备气球，这些事情都是同时进行的，这就是并行。

并发和并行的区别在很多情况下并不需要严格区分，在本书中，如果没有特意说明，我们统一使用并发这个词。

# go 语言和并发编程

go 语言在设计之初就把性能作为很重要的目标，它不仅能让我们更快速地编写代码，也能快速编译代码，还希望 最终的程序运行效率很高。因为对性能的追求，go 把并发实现成了语言本身的特性，这也成为了吸引很多 程序员和公司的一大亮点。

go 语言对并发的支持，让快速开发正确的并发程序变得容易。但这个容易只是相对而言， 相对其他语言和之前的开发过程，要想真正掌握并发编程，我们还是需要深入了解 go 提供的语法和功能、 认真设计程序的逻辑和行为。

这本书就针对 go 语言并发编程这一话题进行深入探讨，希望能帮助大家解锁 go 并发的威力， 有效地开发出性能更好的程序。

# goroutine 和 channel

go 语言提供了 goroutine 作为并发的实体，而不是传统编程语言中的进程和线程机制。goroutine 可以看做轻量级的线程，可以轻松创建成千上万的 goroutine 共同运行，它们的调度 由 go 语言运行时负责，而不是操作系统内核负责。

goroutine 之间通信需要通过 channel，channel 一般翻译成管道，它是一种类似于队列、unix 管道和 socket 的存在， 支持读写两种操作，一个 goroutine 往里面写入数据，另外一个 goroutine 就能从里面读取数据。而且 channel 的读写是 并发安全的，使用起来不用担心数据不一致的问题。

goroutine 和 channel 并发编程中最重要的两个概念，这部分我们就讲解它们两个的使用方法和技巧。

# 你的第一个 go 并发程序

很多介绍 go 语言的文章和书籍都会提到它的并发模型，夸赞用 go 语言编写并发应用是多简单。 并发编程已经成为 go 语言最有标志性的特性，也是很多程序员和技术公司选择它的主要原因。

go 语言提出了 goroutine 的概念，作为程序运行和调度的基本单位。可以把 goroutine 类比于操作系统的 进程和线程，只不过非常轻量（或许你曾看到过蝇量级这个说法），所以可以创建大量的 goroutine，并且它们的调度 也更高效。

创建一个 goroutine 在 go 中非常简单，之前在正常的函数前面加上 `go` 关键词，那么这个函数就会作为一个 goroutine 在后台运行。来看一个并发版本的 hello world 程序：

```
package main

import (
    "fmt"
    "time"
)

func worker(name string) {
    for i:=0; i<10; i++ {
        fmt.Println(name)
        time.Sleep(10 * time.Millisecond)
    }
}

func main() {
    go worker("hello")
    go worker("world")
}
```

我们编写了一个 `worker` 函数，简单地把某个字符创打印 10 遍。然后在 main 函数中启动两个 goroutine，分别打印 `hello` 和 `world` 字符串。运行程序我们期望在终端交替看到两个单词的出现，直到打印完毕。可以看到，和普通的程序相比， 我们不需要依赖额外的库，只是添加了两个 `go` 关键字就把顺序执行的程序变成了并发执行，这也许是很多人说 go 并发编程简单的原因吧。

如果上面的程序保存为 `main.go` ，执行 `go run main.go` 来运行，你会发现终端上什么都没有打印。这是为什么？不是说很简单吗，为什么上来就遇到问题？

虽然 go 语言提供了语言级别特性让并发编程变得简单，但是想要编写正确高效的并发代码并不是件容易的事情，我们还是 要了解 goroutine 的各种特性，并认真设计代码逻辑。对于这里遇到的第一个问题，答案在于 main 函数本身也是一个 goroutine， 创建完两个 goroutine

之后，main 函数就运行结束并退出，而不会等待它创建的 goroutine 运行完成。因此 两个 **woker goroutine** 还没有运行，**main** 函数就退出了，所以我们自然看不到输出。

知道了问题，解决的思路就是让 main 函数等待两个 goroutine 运行结束再退出。提到等待，最容易的想法是在程序中 sleep 一段时间，所以把上面的代码修改一下：

```
package main

import (
    "fmt"
    "time"
)

func worker(name string) {
    for i:=0; i<10; i++ {
        fmt.Println(name)
        time.Sleep(time.Millisecond * 10)
    }
}

func main() {
    go worker("hello")
    go worker("world")

    time.Sleep(time.Second * 2)
}
```

再次执行，你会在终端看到期望的输出结果：hello 和 world 交替出现，而且运行多次会发现输出的顺序也不是不断变化的。

虽然上面的程序得到的期望结果，但还是有一个致命的问题：sleep 的时间应该设置为多少？对于这个简单的程序两秒钟就足够了，但是对于复杂的程序 要评估一个合理的运行时间非常困难。如果设置的时间太短，会导致上面 goroutine 没有运行完成的问题；如果设置的时间太长，在实际中又会导致时间浪费。

理想情况下，我们希望 goroutine 一旦运行完，main 函数得到通知，并立即退出。go 当然提供了对应的解决方案，这正是我们下一节要讲的内容。

# 等待 goroutine 结束

在上一节，我们介绍了如何启动 goroutine，并通过 sleep 一段时间来等待 goroutine 执行完成，并解释了这种方案的缺点。 那么这一节，我们会学习 go 语言提供的方案，来等待 goroutine 执行结束。

go 标准库提供了 `sync` 包来解决各种需要同步的问题，而我们只需要用到 `sync.WaitGroup` 。

从名字来看，可以把 `WaitGroup` 理解成等待（wait）一组（group） goroutine ，它会维护一个计数器，当有 goroutine 运行时，把这个信息告诉 WaitGroup， WaitGroup 就知道有多少个 goroutine 在运行；当某个 goroutine 运行结束时，也告诉 WaitGroup，WaitGroup 就知道有多少 goroutine 运行结束。 当执行等待函数时，WaitGroup 如果发现还有 goroutine 没有执行完成（计数器不是 0 ）就阻塞，如果 goroutine 都已经执行完成就直接返回。

还是拿上一节的例子，来看一下使用 `WaitGroup` 之后的代码：

```
package main

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

func worker(name string, wg *sync.WaitGroup) {
    for i:=0; i<10; i++ {
        fmt.Println(name)
        time.Sleep(time.Millisecond * 10)
    }
    wg.Done()
}

func main() {
    wg := sync.WaitGroup{}

    wg.Add(2)
    go worker("hello", &wg)
    go worker("world", &wg)

    wg.Wait()
}
```

编译运行，结果还是和上一节一样：随机地打印 `hello` 和 `world` 字符串，但区别是程序运行时间会明显小于 2 秒，比如我机器上运行只需要 `0.20s` ：

```
➜  wait-goroutine time go run main.go
world
hello
hello
......

go run main.go  0.20s user 0.15s system 80% cpu 0.436 total
```

这段代码和原来的代码结构相同，只是添加了 WaitGroup 的逻辑：在运行 goroutine 之前，定义了 `WaitGroup` 变量，并调用了 `Add` 方法，因为我们实现知道了会运行多少个 goroutine，所以直接把计数器增加了对应的次数；worker 函数添加了一个 `wg` 参数，并在 goroutine 执行完成时调用 `Done()` 方法，最后是 main 函数调用 `Wait()` 方法等待 goroutine 执行结束。

`WaitGroup` 是一个结构体，定义在 `sync` 库里，它没有对外公开的字段，因此初始化非常简单：`sync.WaitGroup{}`。

`WaitGroup` 一共对外暴露了三个方法：

- `Add()`：接收一个整数作为参数，表示对计数器进行修改，一般为正数，表示有多少个 goroutine 要运行，也可以为负数，表示有多少个 goroutine 运行完成。如果参数导致计数器为负，则会直接 panic
- `Done()`：功能和 `Add(-1)` 相同，表示有一个 goroutine 运行完成，一般在 goroutine 最后调用
- `Wait()`：等待 `WaitGroup` 中注册的 goroutine 全部运行完成，也就是计数器为 0。如果计数器大于 0，这个操作会一直阻塞，一旦发现计数器为 0，就立即返回

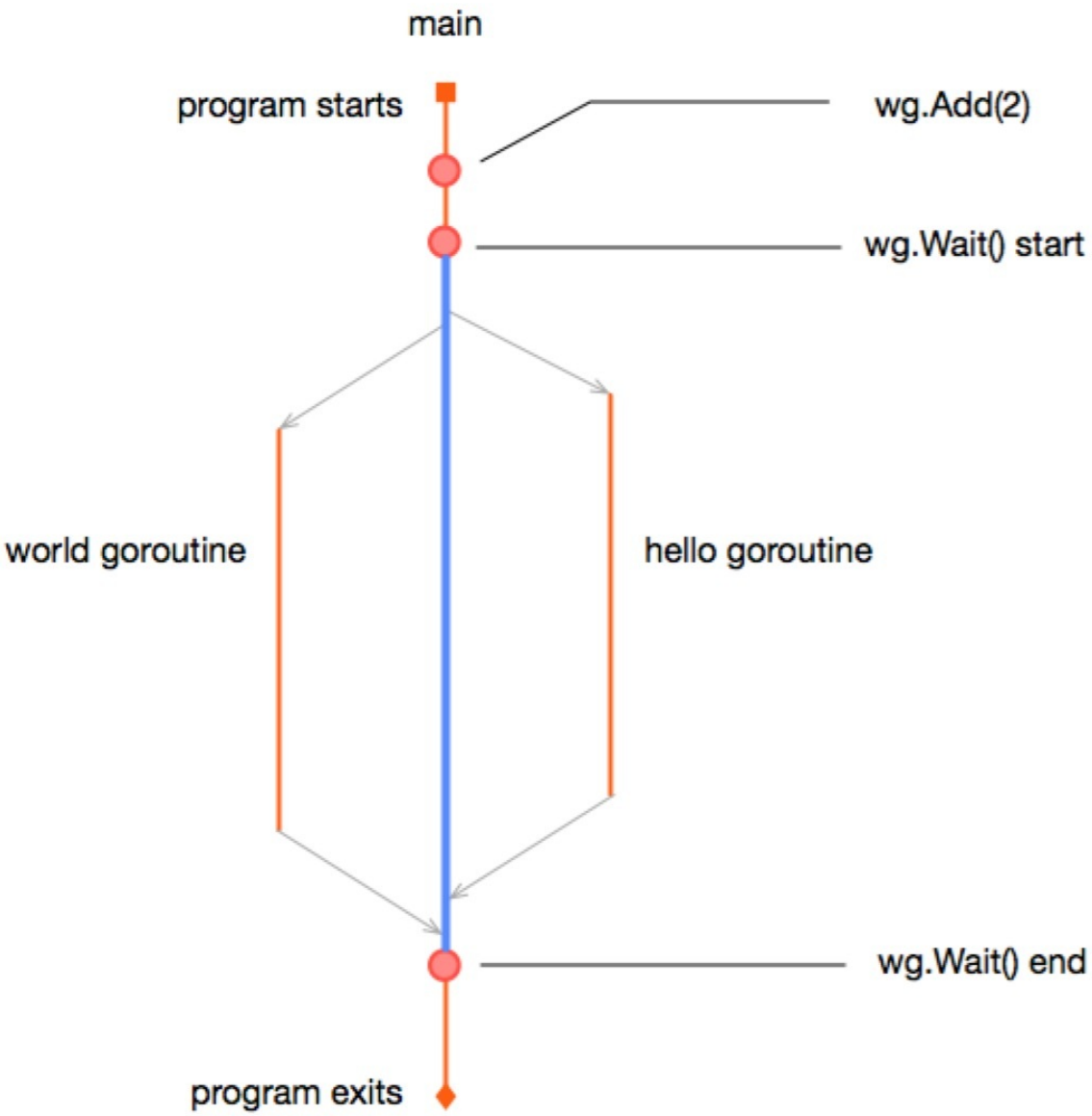
可以看到，WaitGroup 并不会自动感知 goroutine 的运行状态，而是需要我们在执行之前和之后告诉它 goroutine 要运行和运行结束了，它只是保证并发安全地计数，并提供阻塞功能。我们会认为 goroutine 一定会创建成功，并且在创建之前就调用 `Add()` 方法，保证它在 `Wait()` 方法之前运行，而 `Done()` 方法是在 goroutine 运行结束之前立即调用的，正常情况下不要使用 `wg.Add(-3)` 这种把负数传给 `Add` 的用法。

注意事项: 因为在 worker 执行完成是要调用 `wg.Done()`，所以我们添加了一个参数把 `wg` 传给它。如果要传递 **`WaitGroup`** 的话，一定要使用指针。如果不适用指针，go 会对原来的变量进行值传递，复制一个新的值传递给函数，执行 `Done()` 操作原来的 wg 变量也不会知道。如果把上面代码指针改成值传递，运行会出现死锁，因为 main goroutine `wg` 一直认为还有两个 goroutine 在等待，所以会一直阻塞在 `Done()` 调用上。

虽然，上面的方案已经满足了我们的需求，但是有个不好的地方，那就是修改了原来的 worker 函数，把 WaitGroup 相关的控制逻辑和业务逻辑耦合在一起。利用 go 语言函数的灵活功能，我们可以封装一个匿名函数，把 WaitGroup 的逻辑封装在这层函数里：

```
package main

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

func worker(name string) {
    for i:=0; i<10; i++ {
        fmt.Println(name)
        time.Sleep(time.Millisecond * 10)
    }
}

func main() {
    var wg = sync.WaitGroup

    wg.Add(2)
    go func() {
        worker("hello")
        wg.Done()
    }()

    go func() {
        worker("world")
        wg.Done()
    }()

    wg.Wait()
}
```

程序的运行示意图如下：




可以看到，原来的 worker 函数完全不需要任何改动，这是比较推荐的做法，你会在很多 go 语言的代码库中看到类似的用法。

另外一个需要注意的事项 `wg.Done()` 一般在 goroutine 运行之前就添加，如果不小心把它放到 goroutine 中，可能会遇到 `Add` 还没有执行就运行到 `wg.Wait()`，这时候因为计数器还是 0，main 函数就继续执行并退出了。感兴趣的读者可以把上面例子 `wg.Add(2)` 放到 goroutine 中测试一下，看看会出现什么结果。

另外一个留给读者思考的问题是，如果在调用 `wg.Add` 的时候搞错参数，写成了 `wg.Add(1)`，会出现什么情况呢？为什么？

# goroutine 通过 channel 通信

通过前面几节的学习，我们已经知道如何在 go 语言中启动 goroutine 来实现并发地运行程序逻辑。但是这些并发的函数都是相互独立的，不需要互相之间有任何的交互，但现实生活中以及实际项目中，并发的 goroutine 之间往往需要知道对方的存在，同步事情的进展，发送数据给对方。

go 提供了另外一个概念: channel。channel 一般翻译成管道，或者通道，能够比较形象地阐释数据的发送和接收。对于程序员开发，也可以把 channel 简单类比成队列。和队列一样，goroutine 可以往 channel 中发送数据和读取数据，而且数据能保证先进先出的顺序，但是和队列不同的是，这个队列是 go 运行时维护的，能够保证并发运行的正确性，还提供了一些更复杂的功能。

## channel 基本知识

要定义一个 channel 变量，需要使用 `make` 关键词:

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

`chan` 关键词说明我们是在创建一个 channel，而后面的 `int` 代表 channel 中可以传输的数据类型。这个类型可以是 go 语言自带的类型、命名类型、struct 类型等，也可以是这些类型的指针类型。

既然类似于队列，那么就一定会支持两种基本操作：从 channel 中读取数据和往 channel 中写入数据。go 语言定义了个特殊的符号: `<-`，而没有提供内嵌的函数。

往 channel 中写入数据是这样的：

```
ch <- 42
```

`<-` 右边是要写入的数据，左边是写到的 channel 变量。

而从 channel 中读取数据变量符号两边的内容正好相反，左右是存放数据的变量，右边是 channel，因为是赋值操作，中间还需要 `=` 或者 `:=` 赋值符号：

```
answer := <- ch
```

我们上面创建的 channel 还有一个特别重要的特性：当往里面写数据的时候，如果没有人把它取走，那么这么操作会一直阻塞；反之亦然。 这样的 channel 要求两个 goroutine 必须在这个时间点同步，就像两个人约好在某个地方碰面，一起去看电影。如果其中一个人先到，它 必须在这里等待，一直等到另外一个人来到，才能去看电影。

在后面的小节里，我们会讲解另一种 channel，它能够保存特定数量的数据，这样 goroutine 往里面写数据不用每次都阻塞。

# 使用 **channel** 模拟乒乓游戏

为了说明 channel 的使用，我们写一个模拟打乒乓球的 go 程序。和之前打印字符串的程序不同，打乒乓球需要两个选手参与（goroutine）， 而且他们要等待同一个乒乓球（同步数据），只有乒乓球被打到自己这边的时候才能接球，否则就要一直处于准备状态（等待）。

```go
package main

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

// palyer 函数模拟每个乒乓球队员的行为
// 接球 -> 把球打过去 -> 等待球回来
func player(name string, table chan int){
    for {
        ball := <- table
        fmt.Printf("%d %s\n", ball, name)
        time.Sleep(time.Second)
        ball++
        table <- ball
    }
}

func main() {
    var wg sync.WaitGroup

    // 定义 table 模拟球台，传递乒乓球
    table := make(chan int)
    ball := 1

    wg.Add(2)

    // 第一个乒乓球员上线
    go func(){
        player("ping", table)
        wg.Done()
    }()

    // 第二个乒乓球员上线
    // 使用 Sleep 是为了让第一个球员先接球
    go func(){
        time.Sleep(time.Millisecond)
        player("\tpong", table)
        wg.Done()
    }()

    // 发球
    table <- ball

    wg.Wait()

    // 实际上，这句话并不会执行到
    fmt.Println("Game Over")
}
```

这个简单的程序继续使用 `WaitGroup` 来等待 goroutine 执行完成（但是实际上两个 goroutine 不会自动结束），每个队员在后台作为 goroutine 运行。 player 函数模拟每个乒乓球队员的行为，不断循环以下逻辑：

- 等待球发到自己这里
- 在终端打印自己收到球的信息，并修改击球的次数
- 把球发出去

乒乓球在这里定义为一个整数，它只是记录了游戏中球被击中了几次。这里需要重点说明的是，我们定义了 `table` 各个变量来模拟球台。 因为每次击球都是通过球台把球发送给对方的，因此球台定义为 channel 类型，其中传递的球是 int 类型。

运行程序，可以看到终端依次打印 `ping` 和 `pong` ，以及每次击球的次数，输入 `Ctrl + c` 来结束程序的运行：

```
➜  ping-pong go run main.go
1 ping
2     pong
3 ping
4     pong
5 ping
6     pong
^Csignal: interrupt
```

我们已经让两个 goroutine 进行同步，等待对方完成一个事件后自己才继续执行，这个逻辑非常简单，但是很使用，在实际的代码中会经常遇到。

虽然我们可以通过在顺序执行的程序里依次调用两个队员来实现相同的逻辑， 但是我们的程序明显更符合现实世界的情况：每个队员都是独立地完成自己的动作，并没有统一的控制中心。

这个游戏还有一个明显的缺陷：我们假设两个乒乓球选手永远都不会失误，每次都能准确地把球击中回去。这在实际生活中显然是不可能的，在下一节，我们继续 改进程序，让乒乓游戏能够正常运行结束。

# 关闭 channel

在上一节，我们介绍了 channel 的基本概念，并使用 channel 模拟了打乒乓球的过程，最后留下了一个问题：怎么结束乒乓球游戏？ 这一节，我们将解答这个问题。

不妨我们先思考一下，现实中一场乒乓球是怎么结束的？无非是两种情况：一个是没接住对方的球；另外是接到了球，但是没有打到对方桌面上。 我们可以把这两种情况简化为一种情况：某个选手接到球之后，直接失败了。因为不同选手的能力不同，所以失败的几率也不同， 根据选手的接球成功率，我们可以用简单的算法判断它每次接球是成功还是失败。 一方选手接球失败，另外一方就能直接看到，接收到这个消息，准备下一个回合的比赛，而不是像之前那样傻等着对方继续发球过来。

回到我们 goroutine 的代码中，我们可以让某个选手接球失败之后就直接退出 goroutine，但是另外一个 goroutine 并不会自动 知道这个 goroutine 已经退出，还是会执行 `ball := <-table` 的指令，傻傻地等待对方发球过来。

当然，我们可以创建另外一个 channel 单独传递某一方失败的消息，另外一方定时去查看这个 channel 来判断是否比赛已经结束。 但这无疑让整个问题变得很复杂，其实我们可以用 channel 自带的另外一个操作来完成相同的功能，那就是接下来要讲解的关闭 channel。

## channel close 操作

和队列不同的是，channel 还可以关闭，这个有点像 socket，双方通信完成之后，需要关闭连接。 关闭 channel 只需要调用 `close` 函数：

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

那关闭的 channel 会有哪些行为呢？首先，channel 只能关闭一次，如果尝试关闭一个已经关闭的 channel 会报错：

```
panic: close of closed channel
```

尝试往一个已经关闭的 channel 中写入数据也会报错：

```
panic: send on closed channel
```

但是从已经关闭的 channel 中是可以读取数据的，如果 channel 中还有数据（后面介绍有缓存 channel 的时候会更容易理解），可以继续从里面读取数据。如果 channel 中没有写入的数据了，它会返回传输数据类型的默认值，比如如果 channel 中传输的数据类型为 int，则会一直返回 0。

**NOTE:** go 语言中各种类型的默认值请参考相关数据或者文档。

```
package main

import (
    "fmt"
)

func main() {
    ch := make(chan int)
    close(ch)

    for i := 0; i < 10; i++ {
        data := <- ch
        fmt.Printf("%d ", data)
    }
}
```

上面的程序会打印出：`0 0 0 0 0 0 0 0 0 0`，并不会报任何错误。

那么，这就有一个问题：接收方（消费者）怎么知道 channel 已经关闭了呢？还是说 channel 就是在一直发送 0 过来呢？答案是：可以在从 channel 中读取数据的时候提供第二个接收变量，它是一个布尔值。当为真时，表明 channel 是打开状态，当为假时，表明 channel 已经关闭。

```
data, ok := <- ch
if !ok {
    fmt.Printf("channel closed. exit...")
    return
}
fmt.Printf("%d ", data)
```

因为 channel 的这种特性，一般发送方（生产者）在没有数据发送后关闭 channel，并且保证不会再往 channel 中写数据；接收方（消费者）判断 channel 是否关闭，根据请求执行不同的逻辑。

在使用完 channel 之后，不必一定要把它关闭，go 语言会保证未关闭的 channel 不会造成资源泄露。只有当需要通知接收方数据已经发送完毕时，才需要关闭 channel。

# 完整的乒乓球模拟代码

根据 channel 上面的特性，我们来继续改进乒乓球游戏。

首先，定义一个每次接球的成功率，取值是 0-100 之间。 每次接到球，生成一个随机数，结合成功率来决定这次能否接球成功。

如果接球失败，则关闭 channel（通知对方），然后退出函数。每次接球的时候判断 channel 是否已经关闭（对方失败）， 如果已经关闭，则退出函数；否则继续游戏。

完整的代码如下：

```
package main

import (
    "fmt"
    "sync"
    "time"
    "math/rand"
)

func init(){
    // 每次运行用当前时间重置随机数生成器，增加随机性
    rand.Seed(time.Now().UnixNano())
}

type player struct {
    name string
    successRatio int // a number between [0, 100]
}

func play(p *player, table chan int){
    for {
        ball, ok := <- table

        // channel 已经关闭了，只有对方失败了才会关闭 channel，
        // 也就是说，当前队员赢得了游戏。
        if !ok {
            fmt.Printf("%s win!!!\n", p.name)
            return
        }

        // 生成一个 100 以内的随机数
        // 如果这个值大于成功率，则判定接球失败
        r := rand.Intn(100)
        if r > p.successRatio {
            // 失败之后，关闭 channel，然后退出函数
            fmt.Printf("%s lose.\n", p.name)
            close(table)
            return
        }

        fmt.Printf("%d %s\n", ball, p.name)
        time.Sleep(time.Millisecond * 200)
```

```
        ball++
        table <- ball
    }
}

func main() {
    var wg sync.WaitGroup
    table := make(chan int)
    ball := 1

    wg.Add(2)
    go func(){
        play(&player{
            name: "Zhang",
            successRatio: 90,
        }, table)
        wg.Done()
    }()

    go func(){
        time.Sleep(time.Millisecond)
        play(&player{
            name: "cizixs",
            successRatio: 80,
        }, table)
        wg.Done()
    }()

    table <- ball

    wg.Wait()
    fmt.Println("Game Over")
}
```

运行上面的程序，会出现类似下面的结果：

```
1 Zhang
2 cizixs
3 Zhang
4 cizixs
5 Zhang
6 cizixs
Zhang lose.
cizixs win!!!
Game Over
```

# 带缓存的 channel

前面介绍的 channel 作用更多是同步，在乒乓球游戏中，channel 是为了保证两个队员按照顺序执行。但是并发最重要的功能是让多个运行实体（goroutine）能够同时做事情，而不是像打乒乓球那样，一方接球的时候，另外一方只能在那等着什么都不做。

提到并发模型，最经典的就是生产者和消费者问题。这个问题的描述是这样的：有一个生产数据的实体，称为生产者；另外有一个消费数据的实体，称为消费者，它们之间通过固定大小的队列作为缓存来通信。生产者把产生数据，并把数据放到队列中；同时，消费者从队列中取出数据，执行任务。而且，生产者在队列满的时候不会继续往里面放数据，消费者在队列空的时候不能从里面读数据。

和乒乓球游戏不同，生产者和消费者不需要互相等待，因为在实现中，它们根本不知道对方的存在。这个问题更多关注如何保证数据传输的正确性，生产者和消费者是解耦的，而且因为缓存队列的出现，可能有不同的处理速率。

这一节我们看看如何用 go 语言来解决生产者和消费者的问题。

## buffered channel

生产者和消费者之间通过缓存队列通信，不仅使它们功能解耦，不需要知道互相的存在。在并发上还有一个重要的功能：不需要等待（准确的说，避免了大部分情况的等待，因为在缓存队列满或者为空的时候还是会等待）。生产者制造出东西，没必要一定要等到有人来消费才能继续下去。

go 语言支持有缓存和无缓存的 channel，前面几节讲到 channel 是通过 make 关键词创建的：

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

但其实，在创建 channel 的时候可以带上第二个参数，表示 channel 可以缓存多少个数据：

```
// 创建一个能缓存 10 个整数的 channel
ch := make(chan int, 10)
```

缓存 channel 的特点是，当往里面写入数据时，如果缓存还没有满，则会立即写入成功，不会阻塞等待；当从里面读取数据的时候，如果缓存不空，则读取会立即成功，也不会阻塞等待。

无缓存的 channel 只是有缓存 channel 的特例，可以看做缓存长度为 0。也就是说，下面两种定义是等价的：

```
// 创建一个无缓存的 channel
ch := make(chan int)

// 等价于，缓存长度为 0 的 channel
ch := make(chan int, 0)
```

`cap` 函数可以获取有缓存 channel 缓存区的长度：

```
capacity := cap(ch)
```

`len` 函数可以获取有缓存 channel 中当前有多少数据，但是在一个并发的程序中，这个数据是一直处于快速变化的，因此只适合作为参考（比如日志、测试或者调优）。

举个我们取快递的例子，无缓存 channel 是快递员打电话给你，一定要等你亲手签收；有缓存 channel 更像是快递员直接把包裹放到前台或者快递柜，等你有空的时候自己取就行。只有当快递非常重要时（goroutine 之间同步非常重要），才会采取前者；更一般的情况（goroutine 之间只是为了交换数据，对同步不感兴趣），我们倾向于后者。

# channel 的方向性

乒乓球游戏中，双方队员都需要发球和接球，在代码中表现为要对 `table channel` 做读取数据和写入数据两种操作。但是在生产者和消费者中，生产者只会往 channel 中写入数据，消费者只会从 channel 中读取数据，这种情况在实际中很常见，channel 作为参数传递给函数时基本上都能确定它是只读还是只写的。

从安全性角度考虑，应该秉承最小权限原则，让生产者不能从 channel 中读取数据，让消费者不能从 channel 中写入数据。为此，go 语言定义了单向 channel 类型，只暴露读写操作中的一个。比如 `chan<- int` 是 int 类型的写 channel，只允许往里面写入数据，不允许从里面读取数据；相反的，`<-chan int` 是 int 类型的读 channel，只允许从里面读取数据，不允许往里面写入数据。

**NOTE**：在编译的时候，go 就会检测单向 channel 是否被正常使用，避免了运行时可能产生的错误。

另外，因为 close 的作用是保证不会再往 channel 中写入数据（还记得吧，我们可以从已经关闭的 channel 中读取数据），所以只有写 channel 才能调用 close 函数，关闭只读 channel 会导致编译错误。

细心的读者可能发现，一个完全只读或者完全只写的 channel 是没有实际意义的。在实际应用中，更多的是创建一个双向的 channel，然后根据使用情况把它转换成只读或者只写的 channel。

# 生产者-消费者 go 语言解决方案

跳过一个生产者 VS 一个消费者，以及一个生产者 VS 多个消费者，和一个消费者 VS 多个生产者的情况， 我们直接来看多个生产者和多个消费者的情况，完整的代码如下：

```
package main

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

const (
    // NumOfProducers 表示要运行多少个生产者
    NumOfProducers = 3

    // NumOfConsumers  表示要运行多少消费者
    NumOfConsumers = 5
)

// Producer 生产者结构体，只有一个字段，用来标识生产者的 ID
type Producer struct {
    producerID int
}

// 构建函数，返回一个生产者指针对象
func newProducer(ID int) *Producer {
    return &Producer{
        producerID: ID,
    }
}

// Run 就是生产者的核心逻辑，产生数据，并放到 channel 中
func (p* Producer) Run(ch chan<- string){
    for i:=0; i<5; i++{
        fmt.Printf("producer %d put data %d\n", p.producerID, i)
        data := fmt.Sprintf("data %d from producer %d", i, p.producerID)
        ch <- data

        // 休息一段时间模拟生产者工作时间花费
        time.Sleep(time.Microsecond * 10)
    }
}

// Consumer  消费者结构体，也有一个标识消费者身份的 ID
type Consumer struct {
    consumerID int
}

func newConsumer(ID int) *Consumer {
```

```
    return &Consumer{
        consumerID: ID,
    }
}

// Run 消费者的核心逻辑：不断从 channel 中读取数据进行处理
func (c *Consumer) Run(ch <-chan string){
    for {
        data, ok := <- ch
        if !ok {
            fmt.Printf("consumer %d: detect channel close\n", c.consumerID)
            return
        }

        fmt.Printf("consumer %d got: %s\n", c.consumerID, data)
        time.Sleep(time.Microsecond * 10)
    }
}

func main(){
    buffer := make(chan string, 10)

    // 以 goroutine 运行多个生产者
    prodWg := sync.WaitGroup{}
    prodWg.Add(NumOfProducers)
    for i:=0; i<NumOfProducers;i++ {
        go func(ID int){
            p := newProducer(ID)
            p.Run(buffer)
            prodWg.Done()
        }(i+1)
    }

    // 以 goroutine 运行多个消费者
    consumerWg := sync.WaitGroup{}
    consumerWg.Add(NumOfConsumers)
    //run consumers
    for i:=0; i<NumOfConsumers; i++{
        go func(ID int){
            c := newConsumer(ID)
            c.Run(buffer)
            consumerWg.Done()
        }(i+1)
    }

    // 等生产者都运行完成，则关闭 channel
    prodWg.Wait()
    close(buffer)

    // 消费者任务完成后，则退出程序
    consumerWg.Wait()
    fmt.Printf("exit...\n")
}
```

这段程序中，我们分别把生产者和消费者都封装为一个结构体，每个结构体有一个用来标识身份的 ID。然后各自定义了 `Run` 方法接受 channel 作为参数，运行核心的逻辑。

在实际的程序中，生产者和消费者很可能会一直运行，不会自动退出，除非程序被手动结束。而在我们的例子中，为了让运行结束，我们让每个生产者只生产特定数量的数据；然后关闭 channel，当消费者读取数据发现 channel 关闭了，就知道已经没有数据要处理，因此也会退出。

可以看到，我们只创建一个 channel，然后传递给不同的 goroutine 使用，而且这个 channel 是有缓存的，可以 存放 10 个数据：

```
buffer := make(chan string, 10)
```

其次，为了等待生产者和消费者 goroutine 运行完成，我们封装了匿名函数。需要注意的是，匿名函数接受 ID 作为参数，每次调用会把 `i+1` 传递过去，这是函数闭包的特性；如果直接使用下面的代码：

```
for i:=0; i<NumOfProducers;i++ {
    go func(){
        p := newProducer(i+1)
        p.Run(buffer)
        prodWg.Done()
    }()
}
```

不使用参数传递，而是直接使用 `i+1`，那么所有的生产者内部变量 `producerID` 其实都指向同一个值，也就是说它们的 ID 将会一样。

最后，我们在 main 函数中创建的 channel 是双向的，但是生产者接收的 channel 是只写的，而消费者接受的 channel 是只读的。这是因为在做参数传递时，go 会自动做类型转换。

运行以上程序，可能会得到类似下面的结果：

```
producer 2 put data 0
producer 1 put data 0
producer 3 put data 0
consumer 2 got: data 0 from producer 3
consumer 1 got: data 0 from producer 2
consumer 5 got: data 0 from producer 1
producer 1 put data 1
consumer 3 got: data 1 from producer 1
producer 3 put data 1
consumer 4 got: data 1 from producer 3
producer 2 put data 1
consumer 2 got: data 1 from producer 2
producer 3 put data 2
producer 1 put data 2
consumer 1 got: data 2 from producer 3
producer 2 put data 2
consumer 3 got: data 2 from producer 2
consumer 5 got: data 2 from producer 1
producer 3 put data 3
consumer 4 got: data 3 from producer 3
producer 1 put data 3
consumer 2 got: data 3 from producer 1
producer 2 put data 3
consumer 1 got: data 3 from producer 2
producer 3 put data 4
producer 1 put data 4
consumer 5 got: data 4 from producer 1
consumer 3 got: data 4 from producer 3
producer 2 put data 4
consumer 4 got: data 4 from producer 2
consumer 2: detect channel close
consumer 3: detect channel close
consumer 5: detect channel close
consumer 1: detect channel close
consumer 4: detect channel close
exit...
```

因为并发程序的特定，每次运行的结果不一定完全相同。

充当缓存队列的 channel 有效解决了生产者和消费者之间数据同步的问题，而且也能缓解速率不平衡的问题。但是后面这个问题并没有那么简单，如果生产者生产数据的速率明显大于消费者，缓存队列会经常是满的，那么生产者就需要等待；反之如果消费者消费数据的速率明显大于生产者，缓存队列会经常为空，消费者需要等待。一般情况下，应该事先预估两者的速率，然后配置合适的缓存大小以及生产者和消费者的数量。

# for 和 select

## for...range

从 goroutine 中读取数据，一个常见的模式是用 for 循环不断从 channel 中读数据，直到 channel 关闭才推出循环。 在前一节生产者和消费者模型中，消费者的逻辑就是如此，对应的代码当时是这样写的：

```
for {
    data, ok := <- ch
    if !ok {
        fmt.Printf("consumer %d: detect channel close\n", c.consumerID)
        return
    }

    fmt.Printf("consumer %d got: %s\n", c.consumerID, data)
    time.Sleep(time.Microsecond * 10)
}
```

因为这种用法非常普遍，所以 go 语言提供了快捷的方法：for...range。 如果熟悉 go 的语法，会知道 for...range 一般用来遍历 slice 或者 map，但是它也可以用来读取 channel 中的内容。 当 range 后面跟着的是可读 channel 时，go 语言会每次从 channel 中读取一个数据; 如果 channel 中没有数据可读则阻塞在这里，直到能读到数据；如果 channel 关闭，则退出循环逻辑。

所以上面的代码可以用 for...range 修改成：

```
for data := range ch {
    fmt.Printf("consumer %d got: %s\n", c.consumerID, data)
    time.Sleep(time.Microsecond * 10)
}
fmt.Printf("consumer %d: detect channel close\n", c.consumerID)
```

是不是精简了很多！

## select

另外一个常见的需求是：从多个 channel 中读取数据，只要任意一个 channel 有数据就执行对应的逻辑。 我们不能一次去循环这些 channel，因为第一个执行的逻辑只有运行完成才会继续运行后面的逻辑，而不是 预期的从多个 channel 中选择。

对于这种情况，go 提供了 select...case 语句，select...case 和 switch...case 结构类似，都是从多个分支中 选择一个执行。但是区别在于，select...case 从多个 channel 操作中选择一个可执行的，switch...case 是从多个 语句中选择一个值为真或者值匹配的。

select...case 的结构大致是这样的：

```
select {
    case <- ch1:
        // do something if read from ch1
    case x := <-ch2:
        // do something if read from ch2, and assign value to x
    case ch3 <- y:
        // do something if can write to ch3
    default:
        // do something if none of the above happens
}
```

select 可以跟多个 case 语句，以及可选的 default 语句。每个 case 后面是一个通信操作（从 channel 中读数据，或者往 channel 中写数据）， 下面跟着一个代码块。从 channel 中读数据可以丢弃读到的值（第一种情况），或者把读到的值赋给某个变量（第二种情况）。

如果没有 default 语句，select 会阻塞，直到某个通信操作可以执行，go 执行它的通信操作，以及下面跟着的代码逻辑块，其他的 select 语句不会执行。 如果有 default 语句，运行到这里时，select 如果发现有 case 语句可以执行，则执行相关逻辑；如果没有可以执行的 case，也不会阻塞，而是直接运行 default 下面跟着的代码块。

如果有多个 case 可以执行，go 会随机选择一个，我们不应该对它的顺序有什么期望。

比如要执行一个很耗时的任务，我们希望打印出进展，那么可以使用 select 语句：

```
package main

import (
    "fmt"
    "time"
)

func worker(done chan<-struct{}){
    time.Sleep(5 * time.Second)

    done <- struct{}{}
}

func main(){
    done := make(chan struct{})
    tick := time.NewTicker(1 * time.Second)

    go worker(done)

    for {
        select {
        case <- tick.C:
            fmt.Printf(".")
        case <- done:
            fmt.Printf("\nwork done.\n")
            return
        }
    }
}
```

上面的代码中，我们运行一个 woker，用 time.Sleep 来模拟它运行需要的时间，在后台以 goroutine 方式运行它，当运行完成后会往 `done` channel 中发送一个数据。 同时，我们创建了一个 `time.Ticker` 对象，它是一个可读 channel，会每秒钟发送一个数据。

我们在 main 函数中使用 `select` 从 `tick.C` 和 `done` 中选择一个读取数据，在 worker 执行完成之前，一定是 `tick.C` 中能读到数据，终端会打印一个点 `.` 表示程序还在运行；当 worker 执行完成时，`done` 中能够读到数据，我们就打印程序完成的消息，然后退出。

执行上面的代码，可以下面的结果：

```
$ go run main.go
.....
work done.
```

# 共享变量和锁机制

除了 channel 之外，go 语言还提供了另外一种并发读写变量的方法，那就是传统编程的锁机制。

当多个 goroutine 要并发地读写某个共享变量时，如果不采取额外的处理，会出现竞争条件。多次执行会出现不同的结果，最经典的问题就是银行转账，我们希望不管如何操作，结果都是预期的。 不管账户上多出了金额，还是少了金额，都是不可以接受的行为。通过加锁，我们可以实现变量的安全读写。

锁会影响并发的性能，因此锁的粒度要尽量小，以减少额外的开销。

我们还会介绍为什么 go 语言要提供两种并发控制机制，以及它们各自适合的使用场景。

# 竞争条件（race condition）

## 问题描述

在介绍锁机制之前，我们先来解释一下 goroutine 在读写共享变量的时候会出现什么问题，我们使用最经典的银行转账作为例子来进行说明。

假如 Alice 在银行开始了一个账户，并每个月往里面存钱。简单模拟这个场景的话，可以定义一个 `Account` 结构体表示个人银行账户，提供 `Deposit()` 和 `Balance()` 两个方法，分别代表存钱，和查看账户余额。

```
// Account 代表某个人的银行账户，有用户名和余额两个字段
type Account struct {
    name string
    amount uint32
}

// Deposit 往账户里面存特定数量的钱
func (a *Account) Deposit(amount uint32) {
    a.amount = a.amount + amount
}

// Balance 返回账户里还有多少余额
func (a *Account) Balance() uint32 {
    return a.amount
}
```

在 main 入口函数中，创建出来一个账户，运行两个 worker 一直往账户里存钱（这里省略了等待 goroutine 运行完成的逻辑）：

```
a := &Account{name: “Alice”, amount: 0}

go worker(a)
go worker(a)
```

worker 的功能很简单，就一个 for 循环，每次往账户里面存 1 块钱：

```
func worker(a *Account){
    for i:=0; i<100000; i++ {
        a.Deposit(1)
    }
}
```

整个代码很简单，逻辑非常清晰，完整的代码如下：

```
package main

import (
    "fmt"
    "sync"
)

type Account struct {
    name string
    amount uint32
}

func (a *Account) Deposit(amount uint32) {
    a.amount = a.amount + amount
}

func (a *Account) Balance() uint32 {
    return a.amount
}

func worker(a *Account){
    for i:=0; i<100000; i++ {
        a.Deposit(1)
    }
}

func main() {
    a := &Account{name: "cizixs", amount: 0}
    var wg sync.WaitGroup

    wg.Add(2)
    go func(){
        worker(a)
        wg.Done()
    }()

    go func(){
        worker(a)
        wg.Done()
    }()

    wg.Wait()

    fmt.Println(a.Balance())
}
```

对于 Alice 来说，辛辛苦苦每次存 1 块钱，最后希望账户里有 200,000 元。但是运行上面的程序，你会发现，最终看到的结果很可能少于 20 万，在我的机器上多次运行的结果如下：

```
➜  race git:(master) ✗ go run main.go
110155
➜  race git:(master) ✗ go run main.go
101187
➜  race git:(master) ✗ go run main.go
100287
➜  race git:(master) ✗ go run main.go
101189
```

不仅少，而且少得可怜，账户里最终只有大约一半的钱。 那么，我们要帮 Alice 查一下，钱怎么就没了呢？

# 数据竞争

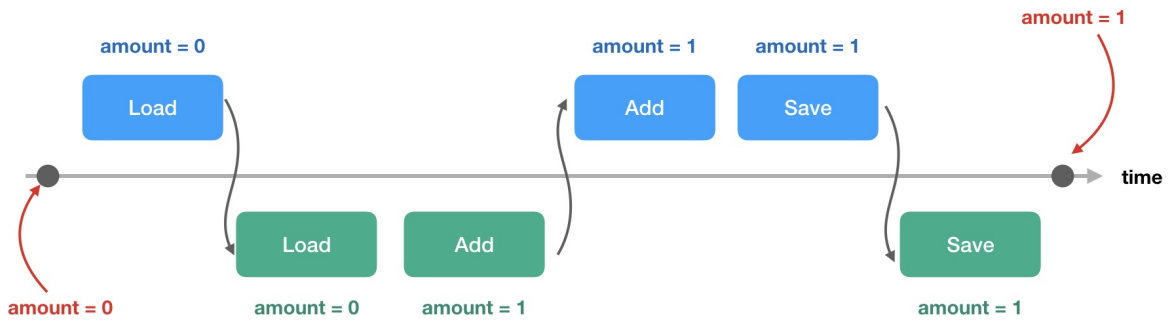
首先在代码中，只有一个地方对账户中的金额进行了修改：

```
a.amount = a.amount + amount
```

这句话只有一行，就是把账户余额加上刚存进去的钱。虽然有两个 goroutine 在运行，按照直觉， 不管它们怎么调度，每个 goroutine 执行到都会把钱加上去，即使调度回去， 另外一个 goroutine 就能看到刚加进去的余额了。

问题在于，对于代码只有一句话，但对于计算机来说要执行的指令却有多条。 要完成存钱的逻辑，计算机首先会读取 `a.amount` 的值，然后执行加法运行， 最后把结果复制给 `a.amount`，并不是一步完成的。 如果一个 goroutine 在执行中间某个步骤时，go runtime 进行了调度， 把 CPU 让给了另外一个 goroutine，就会出现余额减少的情况。

为了容易理解，我们再把事情简化一下，只考虑最开始账户余额为 0 时， 两个 goroutine 运行过程中发生调度， 而且只有一个 CPU 的情况（不会出现两个 gorotuine 同时运行在不同 CPU 的情况）。如下图所示：




中间横轴为时间线，时间线上下蓝色和绿色的方框分别代表两个 goroutine 运行存钱逻辑的不同阶段：读取当前余额、把余额加一、和把计算值保存会余额三个动作。

- 最开始，账户余额为 0 元
- goroutine A 执行了读取操作，得到的值为 0
- 然后 goroutine 发生调度，切换到另外一个 goroutine B 运行，它读取的余额也是 0，并执行了加一操作，此时它保存了 amount = 1 的数据在自己的状态
- 调度再次发生，切换到 goroutine A，它之前读取的余额值为 0，执行加一操作变成 1，然后保存到账户余额中，此时余额为 1
- 又发生调度，goroutine B 把数据之前保存的余额为 1 写回到账户，账户的余额还是 1

可以看到，虽然两个 goroutine 执行了两次加一操作，但是最终的账户余额只增加了一元。这就是 Alice 账号上没有期望金额的原因，在两个 goroutine 执行大量并发存钱时，每次出现上面的情况账户就会少 1 元钱。

这种现象被称为数据竞争（data race），当两个或者多个 goroutine 并发地去操作一个共享的变量，而且至少一个 goroutine 的操作为写时，就会发生。如果没有正确处理并发读写，数据竞争会导致以下可能的后果：

- 更新丢失（lost updates）：某个 goroutine 的更新被其他 goroutine 覆盖，因此更新的数据丢失。上面银行账户的例子就是这种情况
- 脏读（dirty reads）：某个 goroutine 读到还没有更新完全的数据，比如数据还有部分没有提交，或者因为错误需要回退
- 不可重复读(Nonrepeatable Read)：同一个事务里，多次读到某个数据的值不一样

# 解决方案

对于数据竞争的问题，解决方案分为三种，下面就分别来讲一下。

## 1. 不要写变量

既然数据并发需要至少一个 goroutine 对数据执行写操作，在 goroutine 运行的时候不进行写数据的操作就行了。只要把写操作放在 goroutine 运行之前就执行完，在 goroutine 中只执行读操作，就不会出现问题了。

这种方法使用的情况比较少，但是思想却很重要。如果某些操作可以在初始化的时候做掉，那么就不会放到 goroutine 中并发地去运行。

## 2. 不要在多个 goroutine 中读写变量

数据竞争是由于多个 goroutine 并发地对某个数据执行读写操作到处的，另外一个可以想到的方案就是不要把数据共享。把数据的归属权放到某个特定的 goroutine 中，如果另外的 goroutine 要对变量进行读写，就通过 channel 把数据传递过去，其实这就是我们之前一直讲述的 channel 方式，也是 go 社区那句著名的“不要通过共享内存来通信，而是通过通信来共享内存”思想。

> Do not communicate by sharing memory, sharing memory by communicating.

感兴趣的读者可以考虑，如何把银行转账的例子用这种方式来实现。

## 3. 运行多个 goroutine 读写变量，但每次只有一个 goroutine 能操作

最后一种解决方案是这样的，既然数据竞争是因为多个 goroutine 交替执行关键指令导致的，那只要某个 goroutine 执行指令不中断，也就不会出现问题了。比如上面银行账户的问题，如果每个 goroutine 都是执行完 load、add、save 操作才发生调度，结果也不会出现混乱。

这就是我们接下来要讲的锁机制，对某些操作（多个指令）加锁，只有某个 goroutine 执行完加锁的内容，其他 goroutine 才能运行。

# 互斥锁

锁机制是传统的编程语言（比如 Java、C 等）对于并发程序的解决方案，线程读写数据区之前先进行加锁操作，只能加锁成功才能执行读写逻辑，执行完成后需要释放锁， 以供其他线程能够使用。当某个线程执行加锁动作，其他想要执行相同加锁操作的线程只能等待，等到锁解开后才能继续运行。

需要注意的是，加锁其实并没有锁住任何东西，更像是在门上贴上“此门已锁”的标语，解锁就是把这个标语撕掉。 如果大家都主动去读标语，并遵守标语的规定，那么并发读写就是安全的。但是如果一个线程对关键区加锁，但是另外一个线程完全不关心 锁的事情，直接去读写关键区的数据，也是能读写成功的，并没有任何机制会阻止它这么做，当然这么做会导致并发处理的数据有问题。 所以，一定要对所有可能并发读写的地方进行加锁和解锁的操作，漏掉任何一个地方都会让程序出问题。

加锁和解锁的动作必须是成对出现的，如果某个线程只执行加锁操作，但是忘记执行解锁操作，那么所有要读写关键区变量的进程都会 一直处于阻塞的状态，这被称为死锁，在后面的章节我们会介绍死锁的检测。

go 语言也提供了锁机制，只不过不是在语言本身的语法中，而是出现在 `sync` 标准函数库。我们在之前介绍过 `sync.WaitGroup`，这一节介绍另外一个类型。 `sync.Mutex` 是 go 语言提供的互斥锁类型，它不包含任何对外公开的字段，因此声明一个该类型的变量（zero value）就能直接使用，代表着一个未被锁住的 mutex。

mutex 提供两个方法：`Lock()` 和 `Unlock()`，都不接受任何的参数，从名字中可以看出它们的作用分别是锁住当前 mutex，以及解锁当前 mutex。 多次加锁只有其中一个会成功，其他加锁的 goroutine 会处于阻塞的状态，直到锁被解开；如果对一个未被加锁的 mutex 执行解锁操作，会触发程序的 运行时错误。另外，加锁和解锁操作是独立的，完全可以由不同的 goroutine 来执行，也就是说一个 goroutine 对 mutex 加锁， 另外一个 goroutine 对同一个 mutex 解锁的行为完全是 ok 的。尽管可以这么做，我们还是不要这么做，而是尽量让 mutex 加锁和解锁操作在同一个 goroutine 执行，并且让 mutex 只出现在同一个结构体里，不要对 mutex 进行复制和参数传递。

因为加锁和解锁的行为是成对出现的，而且不成对出现会导致错误，所以推荐在加锁之后使用 `defer` 跟着解锁的动作，这样可以减少因为人工失误导致的错误。 这种方法的锁定区域是加锁的地方一直到函数结束，如果需要对加锁的区域进行精细的控制，只能抛弃 `defer`，使用手动在需要的地方解锁的方式。

了解了这些，使用 mutex 来改进银行账户的例子就非常简单了。在 `Account` 结构体中新加一个 `sync.Mutex` 类型的字段 `mu`，它用来保护账户余额的读写操作：

```
type Account struct {
    name   string
    amount uint32
    mu     sync.Mutex
}
```

然后在存钱和查询余额的时候，分别执行加锁和解锁的操作：

```
func (a *Account) Deposit(amount uint32) {
    a.mu.Lock()
    defer a.mu.Unlock()
    a.amount = a.amount + amount
}

func (a *Account) Balance() uint32 {
    a.mu.Lock()
    defer a.mu.Unlock()
    return a.amount
}
```

其他不需要改动，对外的接口还是一样的，改动完之后运行程序，每次执行的结果都是一样的：

```
➜  exclusive-lock git:(master) ✗ go run main.go
200000
➜  exclusive-lock git:(master) ✗ go run main.go
200000
➜  exclusive-lock git:(master) ✗ go run main.go
200000
```

加锁保证了程序的正确性，但是却影响了程序的性能，因为加锁之后，其他所有需要读写数据的 goroutine 只能等待，什么事情都做不了。 为了证明 mutex 锁机制导致性能降低，我们对程序进行 benchmark 测试。因为第一个程序的写操作是有问题的，所以和加锁之后的版本进行性能测试没有什么可比性， 所以我们只测试了读取账户余额的方法，对应的 benchmark 测试用例的代码如下：

```
package main

import (
    "testing"
)

func BenchmarkAccountRead(b *testing.B) {
    a := Account{name: "cizixs", amount: 0}
    for i := 0; i < b.N; i++ {
        a.Balance()
    }
}
```

代码很短，就是基本的 go 语言 Benchmark 的例子，在没使用锁的版本运行结果如下：

```
➜  race git:(master) ✗ go test -test.bench=".*" .
goos: darwin
goarch: amd64
pkg: github.com/cizixs/playground/lock/race
BenchmarkAccountRead-4          2000000000               0.37 ns/op
PASS
ok      github.com/cizixs/playground/lock/race  0.780s
```

在使用锁机制的代码运行结果如下：

```
➜  exclusive-lock git:(master) ✗ go test -test.bench=".*" .
goos: darwin
goarch: amd64
pkg: github.com/cizixs/playground/lock/exclusive-lock
BenchmarkAccountRead-4          20000000                75.9 ns/op
PASS
ok      github.com/cizixs/playground/lock/exclusive-lock        1.604s
```

具体结果会因为机器的配置以及每次运行时系统的负载不太相同，但是可以从上面两个简单的结果看出，加锁之后每次操作的的时间从 `0.37ns` 变成了 `75.9 ns`，如此简单的例子就能带来这么大的性能差距，在实际上更负责的代码中，锁机制带来的性能损失可能会更严重，是我们必须要考虑中的事情。在下一节中，我们将介绍如何使用读写锁来减少某些情况下锁机制带来的性能损耗。

# 读写锁

互斥锁的核心思想在于每次只有一个并发实体能够访问共享的变量，不管是执行读操作还是写操作。 虽然能达到并发安全的需求，但是却给性能带来很大的影响，在上节的最后我们对此进行过测试。 只要使用锁就避免不了性能的损耗，除了控制锁的粒度尽可能小之外，还有一种办法可以减缓这种问题， 那就是这节要讲的读写锁。

在竞争条件那部分我们讲到数据竞争的条件是多个 goroutine 并发操作共享变量，并且至少一个操作为写。 后面这个条件非常关键，因为并发地读取操作并不会出现数据不一致的问题。可以利用这个特性把读操作 的锁和写操作的锁分开，从而提升整个系统的性能。这就是读写锁的思想。

读写锁允许多个读操作同时进行，但是每次只允许一个写操作（不支持多个写操作，也不支持读操作和写操作同时进行）。

如果说互斥锁是通过加锁实现并发读写操作串行化：


那么，读写锁就是通过读锁和写锁分离来达到并发读操作的性能优化：

从上面两张对比图可以看出，原来多个读需要等待前面一个读解锁之后才能继续，它们只能串行运行；使用读写锁之后，多个读操作可以同时进行，从而减少了整个的运行时间。不难知道，读写锁适用于读多写少的场景，而且读写比例差距越大，性能优化越明显。反过来，如果是读少写多，那么性能改进并不明显，极端情况下，写操作比读操作频繁很多，读写锁和互斥锁性能基本没有太大差别。

# go 语言读写锁

读写锁在 go 语言中是通过 `sync.RWMutex` 实现的，从名字上也可以看出，它是在上一节讲到的互斥锁 `Mutex` 前面加上 `RW`（Read-Write 缩写）前缀。这个结构体一共提供了下面几种方法:

```
type RWMutex
    func (rw *RWMutex) Lock()      // 获取写锁，如果系统中读锁或者写锁已经在使用中，那么该操作会一直阻塞，直到写锁可用
    func (rw *RWMutex) Unlock()    // 释放写锁，如果写锁没有被加锁，则会报 runtime error

    func (rw *RWMutex) RLock()     // 获取读锁, 只要系统中写锁没有在使用中，就能获取成功。也就是说允许多个 goroutine 获取读锁
    func (rw *RWMutex) RUnlock()   // 释放读锁，如果读锁没有被加锁，则会报 runtime error

    func (rw *RWMutex) RLocker() Locker // 返回一个 `Locker` 接口实现，它的 `Lock()` 和 `Unlock()` 方法就是调用 `rw.RLock()` 和 `rw.RUnlock()`.
                                        // 换句话说，这只是一个快捷操作
```

也就是说，对写操作部分使用和之前一样，只是额外增加了两个方法用来为读操作加锁和解锁而已。

使用读写锁，系统的状态可以分成三种：

- 空闲状态：没有任何的读操作或者写操作
- 读状态：系统中有一个或者多个读操作在执行
- 写状态：系统中有一个写操作在执行

当处于空闲状态或者读状态时，获取读锁的操作是可以成功的，因为读写锁允许多个读操作；当处于空闲状态时，获取写锁的操作是可以成功的，因为系统中写操作是互斥的，只能存在一个，而且不能和读操作并存。

# 银行账户重写

这部分，我们用读写锁重写前一节银行账户的例子，以提升其性能。

因为银行账户的例子读写就是分开的，存钱是写操作，查看余额是读操作，因此改成读写锁改动的地方很少。首先是把 `sync.Mutex` 改成 `sync.RWMutex`，其次是查看余额的时候加锁和解锁的对象是读写锁中的读锁，使用的方法是 `mu.RLock()` 和 `mu.RUnlock()`，而写锁不需要改动，依旧是 `mu.Lock()` 和 `mu.Unlock()`。

```
type Account struct {
    name   string
    amount uint32
    mu     sync.RWMutex
}

func (a *Account) Deposit(amount uint32) {
    a.mu.Lock()
    defer a.mu.Unlock()
    a.amount = a.amount + amount
}

func (a *Account) Balance() uint32 {
    a.mu.RLock()
    defer a.mu.RUnlock()
    return a.amount
}
```

最后再来看看性能测试的结果，我们不能直接使用之前的 benchmark 代码，因为里面只有读取余额的行为，使用互斥锁还是读写锁并没有明显的差别。我们这次使用的 benchmark 代码如下：

```
func BenchmarkAccount(b *testing.B) {
    a := Account{name: "cizixs", amount: 0}

    var wg sync.WaitGroup

    // 启动 100 个 goroutine，并发读取账户里的余额
    // 每个 goroutine 操作次数是 b.N，也就是 benchmark 设定的一个很大的一个数值
    wg.Add(100)
    for i := 0; i < 100; i++ {
        go func() {
            for i := 0; i < b.N; i++ {
                a.Balance()
            }
            wg.Done()
        }()
    }

    wg.Add(10)
    // 启动 10 个 goroutine，并发往账户里存钱
    // 每个 goroutine 操作的次数是 b.N/10000，比读操作少很多
    for i := 0; i < 10; i++ {
        go func() {
            for j := 0; j < b.N/10000; j++ {
                a.Deposit(1)
            }
            wg.Done()
        }()
    }

    wg.Wait()
}
```

代码会分别启动读 goroutine 和 写 goroutine，读操作的 goroutine 不仅数量多，而且每次的操作次数更多，也就是说我们在模拟一个读多写少的场景。

互斥锁实现的程序性能测试结果如下，需要关注的数据是每次操作耗时，这次是 `21423 ns/op`：

```
➜  exclusive-lock git:(master) ✗ go test -test.bench=".*" .
goos: darwin
goarch: amd64
pkg: github.com/cizixs/playground/lock/exclusive-lock
BenchmarkAccount-4        100000             21423 ns/op
PASS
ok      github.com/cizixs/playground/lock/exclusive-lock        2.377s
```

读写锁实现的程序性能测试结果如下，单次操作耗时为 `7497 ns/op`：

```
➜  read-write-lock git:(master) ✗ go test -test.bench=".*" .
goos: darwin
goarch: amd64
pkg: github.com/cizixs/playground/lock/read-write-lock
BenchmarkAccount-4        300000              7497 ns/op
PASS
ok      github.com/cizixs/playground/lock/read-write-lock       2.322s
```

可以看到读写锁的性能大概是互斥锁实现的三倍左右，确实和期望一样。

# 总结

- goroutine 是 go 语言中可以并发运行的实体
  - 创建 goroutine 只需要在正常的函数前面加上 `go` 关键字就行
  - 使用 `sync.WaitGroup` 可以等待一组 goroutine 运行结束
- channel 可以用来在 goroutine 之间传输数据
  - channel 可以读数据，也可以写数据，go 语言会保证数据的顺序和安全
  - 无缓存（unbuffered）channel 需要读写数据的 goroutine 同时进行才能完成；缓存（buffered）channel 允许 goroutine 异步写入数据，不是每次都要等待
  - channel 是有方向的，分为只读或者只写。一般在创建一个双向 channel 之后，通过类型转换把不同方向的 channel 交给不同的处理函数
  - 关闭 channel 之后，往里面写数据会报错，从里面读数据会返回数据类型的空值
  - 可以使用 `for ... range` 循环地从 channel 中读取数据；`select` 可以从多个 channel 读写操作中选择一个可用的
- go 语言还提供了锁机制

# Part II：实例解析

# go syncMap 是如何实现并发访问的

# part III：内部原理