并行是一种通过使用多处理器以提高速度的能力。所以并发程序可以是并行的，也可以不是。公认的，使用多线程的应用难以做到准确，最主要的问题是内存中的数据共享，它们会被多线程以无法预知的方式进行操作，导致一些无法重现或者随机的结果（称作 竞态）。

不要使用全局变量或者共享内存，它们会给你的代码在并发运算的时候带来危险。
解决之道在于同步不同的线程，对数据加锁，这样同时就只有一个线程可以变更数据。
在 Go 的标准库 sync 中有一些工具用来在低级别的代码中实现加锁。不过过去的软件开发经验告诉我们**这会带来更高的复杂度，更容易使代码出错以及更低的性能，所以这个经典的方法明显不再适合现代多核/多处理器编程：thread-per-connection 模型不够有效。**
Go 更倾向于其他的方式，在诸多比较合适的范式中，有个被称作 Communicating Sequential Processes（顺序通信处理）（CSP, C. Hoare 发明的）还有一个叫做 message passing-model（消息传递）（已经运用在了其他语言中，比如 Erlang）。
在 Go 中，应用程序并发处理的部分被称作 goroutines（协程），它可以进行更有效的并发运算。在协程和操作系统线程之间并无一对一的关系：协程是根据一个或多个线程的可用性，映射（多路复用，执行于）在他们之上的；协程调度器在 Go 运行时很好的完成了这个工作。
协程工作在相同的地址空间中，所以共享内存的方式一定是同步的；这个可以使用 sync 包来实现，不过我们很不鼓励这样做：**Go 使用 channels 来同步协程**

协程是轻量的，比线程更轻。它们痕迹非常不明显（使用少量的内存和资源）：使用 4K 的栈内存就可以在堆中创建它们。因为创建**非常廉价，必要的时候可以轻松创建并运行大量的协程**（在同一个地址空间中 100,000 个连续的协程）。并且**它们对栈进行了分割，从而动态的增加（或缩减）内存的使用；栈的管理是自动的，但不是由垃圾回收器管理的，而是在协程退出后自动释放。**

协程可以运行在多个操作系统线程之间，也可以运行在线程之内，让你可以很小的内存占用就可以处理大量的任务。由于操作系统线程上的协程时间片，你**可以使用少量的操作系统线程就能拥有任意多个提供服务的协程**，而且**Go 运行时可以聪明的意识到哪些协程被阻塞了，暂时搁置它们并处理其他协程。**

存在两种并发方式：**确定性的（明确定义排序）和非确定性的（加锁/互斥从而未定义排序）。**
Go的协程和通道理所当然的支持确定性的并发方式（例如通道具有一个 sender 和一个 receiver）。

协程是通过使用关键字 go 调用（执行）一个函数或者方法来实现的（也可以是匿名或者 lambda 函数）。这样会**在当前的计算过程中开始一个同时进行的函数，在相同的地址空间中并且分配了独立的栈**，比如：go sum(bigArray)，在后台计算总和。
协程的栈会根据需要进行伸缩，不出现栈溢出；开发者不需要关心栈的大小。当协程结束的时候，它会静默退出：用来启动这个协程的函数不会得到任何的返回值。

任何 Go 程序都必须有的 **main() 函数也可以看做是一个协程**，尽管它并没有通过 go 来启动。**协程可以在程序初始化的过程中运行（在 init() 函数中）。**

在一个协程中，比如它需要**进行非常密集的运算，你可以在运算循环中周期的使用 runtime.Gosched()：这会让出处理器，允许运行其他协程；它并不会使当前协程挂起，所以它会自动恢复执行。使用 Gosched() 可以使计算均匀分布，使通信不至于迟迟得不到响应。**

并发与并行

Go 的并发原语提供了良好的并发设计基础：表达程序结构以便表示独立地执行的动作；**所以Go的的重点不在于并行的首要位置：并发程序可能是并行的，也可能不是。并行是一种通过使用多处理器以提高速度的能力。但往往是，一个设计良好的并发程序在并行方面的表现也非常出色。**

在 gc 编译器下（6g 或者 8g）你必须设置 GOMAXPROCS 为一个大于默认值 1 的数值来允许运行时支持使用多于 1 个的操作系统线程，所有的协程都会共享同一个线程除非将 GOMAXPROCS 设置为一个大于 1 的数。当 GOMAXPROCS 大于 1 时，会有一个线程池管理许多的线程。通过 gccgo 编译器 GOMAXPROCS 有效的与运行中的协程数量相等。假设 n 是机器上处理器或者核心的数量。如果你设置环境变量 GOMAXPROCS>=n，或者执行 runtime.GOMAXPROCS(n)，接下来协程会被分割（分散）到 n 个处理器上。更多的处理器并不意味着性能的线性提升。有这样一个**经验法则，对于 n 个核心的情况设置 GOMAXPROCS 为 n-1 以获得最佳性能，也同样需要遵守这条规则：协程的数量 > 1 + GOMAXPROCS > 1。**

所以**如果在某一时间只有一个协程在执行，不要设置 GOMAXPROCS!**

还有一些通过实验观察到的现象：在一台 1 颗 CPU 的笔记本电脑上，增加 GOMAXPROCS 到 9 会带来性能提升。在一台 32 核的机器上，设置 GOMAXPROCS=8 会达到最好的性能，在测试环境中，更高的数值无法提升性能。如果设置一个很大的 GOMAXPROCS 只会带来轻微的性能下降；设置 GOMAXPROCS=100，使用 top 命令和 H 选项查看到只有 7 个活动的线程。

增加 GOMAXPROCS 的数值对程序进行并发计算是有好处的；

请看 goroutine_select2.go

总结：GOMAXPROCS 等同于（并发的）线程数量，在一台核心数多于1个的机器上，会尽可能有等同于核心数的线程在并行运行。

```
package main

import (
    "fmt"
    "time"
```

```
)

func main() {

    fmt.Println("at the begin of main")
    go short()
    go slow()
    start := time.Now()
    time.Sleep(5 * time.Second)
    fmt.Println("at the end of main after",time.Since(start).Seconds(),"s")
    fmt.Printf("%2.2fs\n", time.Since(start).Seconds())
}
func slow() {
    fmt.Println("begin slow")
    time.Sleep(3 * time.Second)
    fmt.Println("end slow")
}

func short() {
    fmt.Println("begin short")
    time.Sleep(2 * time.Second)
    fmt.Println("end short")
}
```

main()，longWait() 和 shortWait() 三个函数作为独立的处理单元按顺序启动，然后开始并行运行。每一个函数都在运行的开始和结束阶段输出了消息。为了模拟他们运算的时间消耗，我们使用了 time 包中的 Sleep 函数。Sleep() 可以按照指定的时间来暂停函数或协程的执行，这里使用了纳秒（ns，符号 1e9 表示 1 乘 10 的 9 次方，e=指数）。

他们按照我们期望的顺序打印出了消息，几乎都一样，可是我们明白这是模拟出来的，以并行的方式。我们让 main() 函数暂停 10 秒从而确定它会在另外两个协程之后结束。如果不这样（如果我们让 main() 函数停止 4 秒），main() 会提前结束，longWait() 则无法完成。**如果我们不在 main() 中等待，协程会随着程序的结束而消亡。**

当 main() 函数返回的时候，程序退出：它不会等待任何其他非 main 协程的结束。这就是为什么在服务器程序中，每一个请求都会启动一个协程来处理，server() 函数必须保持运行状态。通常使用一个无限循环来达到这样的目的。

协程是独立的处理单元，一旦陆续启动一些协程，**你无法确定他们是什么时候真正开始执行的。你的代码逻辑必须独立于协程调用的顺序。**

但是我们看到在上面的代码案例中 **协程是独立执行的，他们之间没有通信**

Go 协程（goroutines）和协程（coroutines）

（译者注：标题中的“Go协程（goroutines）” 即是 14 章讲的协程指的是 Go 语言中的协程。而“协程（coroutines）”指的是其他语言中的协程概念，仅在本节出现。）
在其他语言中，比如 C#，Lua 或者 Python 都有协程的概念。这个名字表明它和 Go协程有些相似，不过有两点不同：
Go 协程意味着并行（或者可以以并行的方式部署），协程一般来说不是这样的
Go 协程通过通道来通信；协程通过让出和恢复操作来通信
Go 协程比协程更强大，也很容易从协程的逻辑复用到 Go 协程。

goroutine协程间的通信信道 channel
前一个案例中协程是独立执行的，他们之间没有通信。**他们必须通信才会变得更有用：彼此之间发送和接收信息并且协调/同步他们的工作。协程可以使用共享变量来通信，但是很不提倡这样做，因为这种方式给所有的共享内存的多线程都带来了困难。**

而**Go有一个特殊的类型，通道（channel），像是通道（管道），可以通过它们发送类型化的数据在协程之间通信，可以避开所有内存共享导致的坑；通道的通信方式保证了同步性。数据通过通道：同一时间只有一个协程可以访问数据：所以不会出现数据竞争，设计如此。数据的归属（可以读写数据的能力）被传递。**

goroutine 允许你让一段代码相对于其他并行执行。 但是为了让它有用，还有一些先决条件 —— 我们得能够把数据传递到运行中的进程中并且还要可以在运行中的程序把数据创建好时能获取出来。 channel 提供了实现方式，并且它独立于 goroutine 工作。
一个 channel 可以理解为一个有指定大小和容积的管道或者传送带。 在一端可以向上放东西而在另一端可以取到。

```
package main

import (
    "fmt"
    "strconv"
    "time"
)

func main() {
    cs := make(chan string)
    for i := 0; i < 3; i++ {
        go makeCakeAndSend(cs)
        go receiveCakeAndPack(cs)
        time.Sleep(1 * time.Second)
    }

}

var i int
```

```
func makeCakeAndSend(ch chan string) {
    i++
    cakeName := "alishi-cake-" + strconv.Itoa(i)
    fmt.Println("make a cake and sending...", cakeName)
    ch <- cakeName
}

func receiveCakeAndPack(ch chan string) {
    cs := <-ch
    fmt.Println("packing received cake", cs)
}
```

我们在每次循环中使用的 time.Sleep() 产生了一个暂停，让我们的制作和打包可以一个接一个地进行。 **由于我们的 channel 是同步的，每次只允许有一个，在产生一个新蛋糕并放到 channel 上之前必须先从 channel 上移走一个蛋糕并打包。**

修改一下代码，让它更像我们通常使用的代码。 **典型的 goroutine 可以在其中重复运行一段代码**，运行操作，通过 channel 和其他 goroutine 交换数据。

```go
package main

import (
"fmt"
"strconv"
"time"
)
func main() {
cs := make(chan string)
go makeCakeAndSend(cs)
go receiveCakeAndPack(cs)
time.Sleep(1 * time.Second)
}

func makeCakeAndSend(ch chan string) {
for i := 1; i <= 3; i++ {
cakeName := "alishi-cake-" + strconv.Itoa(i)
fmt.Println("make a cake and sending...", cakeName)
ch <- cakeName
}
}
```

func receiveCakeAndPack(ch chan string) {
for i := 1; i <= 3; i++ {
cs := <-ch
fmt.Println("packing received cake", cs)
}

}

需要重点理解的是上面我们看到的输出不是 channel 上数据发送和接收正确反应。 **发送和接收是同步的** — 一次一个蛋糕。 **但是由于打印消息和 channel 上的数据发送和传输之间有延时，所以输出似乎不是正确的顺序**。 所以实际上发送的顺序是：

make 1
receive 1
make 2
receive 2
make 3
receive 3
```
所以在使用打印日志来分析 goroutine 和 channel 代码的时候一定要注意。

数据接收者在**判断什么时候停止接收数据**上有问题。 是否还有更多数据会到来还是所有数据已经完了？ 是该等呢还是继续前进呢？ 一种方法是不断地测试数据源检查 channel 是否关闭，但是这样可能效率不高。 **Go 语言提供了 range 关键字，当用在 channel 上时，将会一直等待直到 channel 关闭。**
**通道服务于通信的两个目的：值的交换，同步的，保证了两个计算（协程）任何时候都是可知状态。**

注意错误用法
```go
/* for range ch { 每次循环相当于 cs:=<-ch
cs := <-ch
fmt.Println("packing received cake", cs)
}*/
```

```go
for s := range ch {
    fmt.Println("packing received cake", s)
}
```
```

```go
package main

import (
    "fmt"
    "strconv"
    "time"
)

func main() {
    cs := make(chan string)
    go makeCakeAndSend(cs, 5)
    go receiveCakeAndPack(cs)
    time.Sleep(2 * time.Second)
}

func makeCakeAndSend(ch chan string, count int) {
    for i := 1; i <= count; i++ {
        cakeName := "alishi-cake-" + strconv.Itoa(i)
        fmt.Println("make a cake and sending...", cakeName)
        ch <- cakeName
    }
}

func receiveCakeAndPack(ch chan string) {
    /*for i := 1; i <= 3; i++ {
        cs := <-ch
        fmt.Println("packing received cake", cs)
    }*/
    /*  for range ch {
            cs := <-ch
            fmt.Println("packing received cake", cs)
        }*/

    for s := range ch {
        fmt.Println("packing received cake", s)
    }
}
```

select和channel

在多个 channel 上使用 select 关键字是一种在不同 channel 之间的 「你准备好了吗」探测机制。 case 可以是发送或者接收数据 —— 当一个发送者或者接收者使用 <- 开始，那么这个 channel 就是准备好了。 也可以包含一个 default 块，意味着总是准备好的。 select 关键字的工

作方式大约是这样的

先看下之前方式的例子,代码比较冗余，下一版本进行重构
```go
package main

import (
"fmt"
"strconv"
"time"
)

func main() {
ch := make(chan string)
go makeStrawCakeAndSend(ch,2)
go makeChocoCakeAndSend(ch,2)

```
go receiveCakeAndPack(ch)

time.Sleep(3*time.Second)
```

}

func makeStrawCakeAndSend(ch chan string, count int) {
cakeName := ""
for i := 1; i <= count; i++ {
cakeName = "strawCake-"+strconv.Itoa(i)
fmt.Println("make strawberry cake and sending...", cakeName)
ch <- cakeName
}
}

func makeChocoCakeAndSend(ch chan string, count int) {
cakeName := ""
for i := 1; i <= count; i++ {
cakeName = "chocoCake-"+strconv.Itoa(i)
fmt.Println("make chocolate cake and sending...", cakeName)
ch <- cakeName
}
}
```

func receiveCakeAndPack(ch chan string) {
for s := range ch {
fmt.Println("receiving cake and packing...", s)
}
}

使用select case的方式进行重构

```go
package main

import (
    "fmt"
    "strconv"
    "time"
)

func makeCakeAndSend(favor string, cs chan string, count int) {
    cakeName := ""
    for i := 1; i <= count; i++ {
        cakeName = favor + strconv.Itoa(i)
        //fmt.Printf("make %s and sending...\n", cakeName)
        cs <- cakeName
    }

    //卡克2
    //close(cs)
}

func receiveCakeAndPack(straw_cs chan string, choco_cs chan string) {
    straw_cs_closed, choco_cs_closed := false, false
    for {
        if straw_cs_closed && choco_cs_closed { //卡克3
            return
        }
        fmt.Println("waiting a new cake...")
        select {
        case cakeName, straw_ok := <-straw_cs:
            if !straw_ok {
                straw_cs_closed = true
                fmt.Println("Strawberry channel closed")
            } else {
                fmt.Println("received from straw channel,Now packing", cakeN
ame)
            }
        case cakeName, choco_ok := <-choco_cs:
```

```
            if !choco_ok {
                choco_cs_closed = true
                fmt.Println("ChocoCake chanel closed")
            } else {
                fmt.Println("received from choco chanel,Now packing", cakeNa
me)
            }
            //default:
                //fmt.Println("default")
        }
    }
}

func main() {
    straw_cs := make(chan string)
    choco_cs := make(chan string)

    go makeCakeAndSend("straw_cake", straw_cs, 3)
    go makeCakeAndSend("choco_cake", choco_cs, 3)

    //go receiveCakeAndPack() //卡克1
    go receiveCakeAndPack(straw_cs, choco_cs)

    time.Sleep(2*1e9)
    //time.Sleep(3 * time.Second)
}

/**
waiting a new cake...
received from choco chanel,Now packing choco_cake1
waiting a new cake...
received from straw channel,Now packing straw_cake1
waiting a new cake...
received from straw channel,Now packing straw_cake2
waiting a new cake...
received from straw channel,Now packing straw_cake3
waiting a new cake...
received from choco chanel,Now packing choco_cake2
waiting a new cake...
received from choco chanel,Now packing choco_cake3
waiting a new cake...
ChocoCake chanel closed
waiting a new cake...
Strawberry channel closed

Process finished with exit code 0
*/
```

```
package main

import (
    "fmt"
    "strconv"
    "time"
)

func makeCakeAndSend(favor string, cs chan string, count int) {
    cakeName := ""
    for i := 1; i <= count; i++ {
        cakeName = favor + strconv.Itoa(i)
        //fmt.Printf("make %s and sending...\n", cakeName)
        cs <- cakeName
        fmt.Println(<-cs) //接收消息回执 需要对称
    }

    //卡克2
    //close(cs)
}

func receiveCakeAndPack(straw_cs chan string, choco_cs chan string) {
    straw_cs_closed, choco_cs_closed := false, false
    for {
        if straw_cs_closed && choco_cs_closed { //卡克3
            return
        }
        fmt.Println("waiting a new cake...")
        select {
        case cakeName, straw_ok := <-straw_cs:
            if !straw_ok {
                straw_cs_closed = true
                fmt.Println("Strawberry channel closed")
            } else {
                fmt.Println("received from straw channel,Now packing", cakeN
ame)
                straw_cs <- cakeName+"您的消息已经被处理喽~~~~~~"
            }
        case cakeName, choco_ok := <-choco_cs:
            if !choco_ok {
                choco_cs_closed = true
                fmt.Println("ChocoCake chanel closed")
            } else {
                fmt.Println("received from choco chanel,Now packing", cakeNa
me)
                choco_cs <- cakeName+"您的消息已经被处理喽~~~~~~"
            }
        //default:
```

```
            //  fmt.Println("default")
            }
        }
    }

    func main() {
        straw_cs := make(chan string)
        choco_cs := make(chan string)

        go makeCakeAndSend("straw_cake", straw_cs, 3)
        go makeCakeAndSend("choco_cake", choco_cs, 3)

        //go receiveCakeAndPack() //卡克1
        go receiveCakeAndPack(straw_cs, choco_cs)

        //time.Sleep(2*1e9)
        time.Sleep(2 * time.Second)
    }

    /**
    waiting a new cake...
    received from choco chanel,Now packing choco_cake1
    choco_cake1您的消息已经被处理喽~~~~~~
    waiting a new cake...
    received from straw channel,Now packing straw_cake1
    straw_cake1您的消息已经被处理喽~~~~~~
    waiting a new cake...
    received from choco chanel,Now packing choco_cake2
    choco_cake2您的消息已经被处理喽~~~~~~
    waiting a new cake...
    received from choco chanel,Now packing choco_cake3
    choco_cake3您的消息已经被处理喽~~~~~~
    waiting a new cake...
    received from straw channel,Now packing straw_cake2
    straw_cake2您的消息已经被处理喽~~~~~~
    waiting a new cake...
    received from straw channel,Now packing straw_cake3
    straw_cake3您的消息已经被处理喽~~~~~~
    waiting a new cake...
    */
```

通道实际上是类型化消息的队列：使数据得以传输。它是先进先出（FIFO）的结构所以可以保证发送给他们的元素的顺序（有些人知道，通道可以比作 Unix shells 中的双向管道（tow-way pipe））。通道也是引用类型，所以我们使用 make() 函数来给它分配内存。这里先声明了一个字符串通道 ch1，然后创建了它（实例化）：

```
go
```

```
var ch1 chan string
ch1 = make(chan string)
```

当然可以更短： ch1 := make(chan string)。
这里我们构建一个int通道的通道： chanOfChans := make(chan int)。
或者**函数通道**： `funcChan := chan func()`

所以通道是对象的第一类型：可以存储在变量中，作为函数的参数传递，从函数返回以及通过通道发送它们自身。另外它们是类型化的，允许类型检查，比如尝试使用整数通道发送一个指针。

<- ch 可以单独调用获取通道的（下一个）值，当前值会被丢弃，但是可以用来验证，所以以下代码是合法的：

```
go
if <- ch != 1000{
...
}
```

通道的发送和接收操作都是自动的：它们通常一气呵成。

我们发现协程之间的同步非常重要：

main() 等待了 1 秒让两个协程完成，如果不这样，sendData() 就没有机会输出。
getData() 使用了无限循环：它随着 sendData() 的发送完成和 ch 变空也结束了。
如果我们移除一个或所有 go 关键字，程序无法运行，Go 运行时会抛出 panic：
---- Error run E:/Go/Goboek/code examples/chapter 14/goroutine2.exe with code Crashed ----
Program exited with code -2147483645: panic: all goroutines are asleep-deadlock!
为什么会这样？运行时会检查所有的协程（也许只有一个是这种情况）是否在等待（可以读取或者写入某个通道），意味着程序无法处理。这是死锁（deadlock）形式，运行时可以检测到这种情况。

注意：**不要使用打印状态来表明通道的发送和接收顺序：由于打印状态和通道实际发生读写的时间延迟会导致和真实发生的顺序不同。**

```go
package main

import "fmt"

func main() {
    out := make(chan int) // 无缓冲的协程是同步的，会阻塞，直到有其他的协程来消费
    //out := make(chan int,1) // 带缓冲的协程是异步执行的，知道缓冲满，变为阻塞
    out <- 2
```

```
    fmt.Println(<-out)
}
```

golang死锁问题 (fatal error: all goroutines are asleep - deadlock!)
由 songleo 在 2016-11-18 18:19 发布 602 次点击
这段代码运行没有问题，循环输出：call f1...

```
package main
import (
    "fmt"
)
func f1() {
    for {
        fmt.Println("call f1...")
    }
}
func f2() {
    fmt.Println("call f2...")
}
func main() {
    go f1()
    go f2()
    ch := make(chan int)
    <-ch
}
```

若注释掉go f1(),代码运行时提示死锁：

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

代码如下：

```
package main
import (
    "fmt"
)
func f1() {
    for {
        fmt.Println("call f1...")
    }
}
func f2() {
    fmt.Println("call f2...")
}
func main() {
    // go f1()
    go f2()
    ch := make(chan int)
```

```
    <-ch
}
```
为什么同是调用协程，调用f1不会产生死锁，而调用f2会产生死锁。是因为f1有死循环么？按说没有协程往通道ch写，在main中读取ch应该都会产生死锁，是不是我理解错了，求解答。谢谢！

#2pathletboy • 2016-11-19 11:26
就如报错提示
all goroutines are asleep
当所有存活的协程处于asleep状态时候，就发生了死锁，因为这个时候没有任何内部或者外部的信号使之继续运行。

#3racoon • 2016-11-19 13:00
f2 是吃瓜群众，啥都不影响

如果程序所有 goroutine 都退出了，<-ch 这里就绝望了，不可能等到任何东西

joeonly
#4joeonly • 2016-11-19 23:21
f1是死循环。
这个示例的**chan**相关处理不完整，只有消费者，没有生产者。

已经提示很明确了，all goroutines are asleep，所有协程都在睡觉！
所以**go**就认为是死锁了。如果有f1的话，至少有一个协程是在干活的。

对于死锁的检测非常麻烦，或许**go**就采用了这种比较简单粗暴的方法。算是一个小bug吧，但是关系不大。
2016年11月18日回答 · 2016年11月18日更新 1 评论 编辑

flybywind
947 声望
答案对人有帮助，有参考价值0 答案没帮助，是错误的答案，答非所问
f1是死循环阻塞主线程，调用f1的时候还没执行ch := make(**chan** int) <-ch ，但是调用f2的时候 一次执行完毕，等待<-ch输出,没有载体接数据，所以就死锁了我的理解

go语言中**chan**的死锁问题。

多线程 golang qc1iu 2015年07月27日提问
关注 5 关注
收藏 0 收藏，1.9k 浏览
问题对人有帮助，内容完整，我也想知道答案0 问题没有实际价值，缺少关键内容，没有改进余地

```
package main

import "fmt"

func main() {
    out := make(chan int)
```

```
    out <- 2

    fmt.Println(<-out)
}
```

上面的代码一运行就有错

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

如果将**chan**的初始化改为

```
out:=make(chan int, 1)
```

就可以。

为什么?

2015年07月27日提问 评论 邀请回答 编辑 更多

已采纳

因为默认无缓冲的channel是阻塞的，在

```
out <- 2
```

这一句会阻塞等待其它goroutine从out读取

然而并没有其他goroutine

2015年07月27日回答 3 评论 编辑

武可

993 声望

答案对人有帮助，有参考价值2 答案没帮助，是错误的答案，答非所问

无缓冲的channel，如题主的代码。

当在向一个无缓冲的channel发送数据时，代码会阻塞直到有另一个goroutine接收这个channel。

楼上的解释很形象，就像你打电话，你拨过去就等着人接，那边没人接你就一直等，所以阻塞了。

所以改进方法就是，先开启一个goroutine去电话那头等着接电话，然后你再拨号，这样两边就通了，

然后故事继续

```
func haha(ch chan int) {
    num := <-ch
    fmt.Println(num)
}

func main() {
    ch := make(chan int)
    ch <- 2
    //.....
}
```

2015年10月05日回答 1 评论 编辑

Deen

914 声望

答案对人有帮助，有参考价值1 答案没帮助，是错误的答案，答非所问

我的理解是，**chan**不同于变量

变量是你可以先放进去，再取出来。

但是**chan**在是一个通信工具，如果没人取，你就放不进去。

比如电话，没人接你就打不通。但是比如微信你就可以先录进去对方以后再听。

```
package main

import (
    "fmt"
    "time"
)

func main() {
    ch := make(chan string)
    //没有go关键字，同步执行方法，进入到独立的(独立的啊)方法体中，都是同步执行的，没有其他的协程接收，也会阻塞，所以必然死锁
    //sendData(ch)
    //getData(ch)

    //1. 只有发送数据的函数有go修饰，而数据通过go的协程以及channel发送出去了，receive端是个无限循环来接收
    // 而异常恰恰出现在receive端，说明阻塞了，原因是没有其他协程发送数据，而其又在等数据接收
    //2. 试想如果将getData内部改为5次循环呢？经过实验，果然是可以的!
    //3. 那么此时能从侧面证明 "[go] getData" 没有和有的区别是什么？有go则明，底层会告知什么时候可以没有数据了，不用傻等了。
    //4. 协程中接收端只需要保证发送端有数据来就好说。
    //5. go关键字就像两边的生产者和消费者的对象，是可以看到有没有数据需要发送和接收的。有了go，就不需要说明循环固定的5次，go关键字知道对面的数据已经没有了。
    //go sendData(ch)
    //getData(ch)

    //那这种情况上来就没有go，没有go的协程，怎么通过channel发送数据，肯定也一样发数据的时候同步阻塞，注意发送的时候一定要有人来接(无缓冲的时候)，而这个时候channel也没有架设呢
    //sendData(ch)
    //go getData(ch)

    go sendData(ch)
    go getData(ch)

    time.Sleep(1e9)
}

func sendData(ch chan string) {
    ch <- "Washington"
    ch <- "Tripoli"
    ch <- "London"
    ch <- "Beijing"
    ch <- "Tokio"
}
```

```
func getData(ch chan string) {
    var input string
    // time.Sleep(2e9)
    for {
        input = <-ch
        fmt.Printf("%s ", input)
    }

    //果然是可以的
    /*for i := 0; i < 5; i++ {
        input = <-ch
        fmt.Printf("%s ", input)
    }*/
}
```

go协程和channel的搭配使用理解

```
package main

import (
    "fmt"
    "time"
)

func main() {
    ch := make(chan string)
    //没有go关键字，同步执行方法，进入到独立的(独立的啊)方法体中，都是同步执行的，没有其他的协程接收，也会阻塞，所以必然死锁
    //sendData(ch)
    //getData(ch)

    //1. 只有发送数据的函数有go修饰，而数据通过go的协程以及channel发送出去了，receive端是个无限循环来接收
    // 而异常恰恰出现在receive端，说明阻塞了，原因是没有其他协程发送数据，而其又在等数据接收
    //2. 试想如果将getData内部改为5次循环呢？经过实验，果然是可以的!
    //3. 那么此时能从侧面证明 "[go] getData" 没有和有的区别是什么？有go则明，底层会告知什么时候可以没有数据了，不用傻等了。
    //4. 协程中接收端只需要保证发送端有数据来就好说。
    //5. go关键字就像两边的生产者和消费者的对象，是可以看到有没有数据需要发送和接收的。有了go，就不需要说明循环固定的5次，go关键字知道对面的数据已经没有了。
    //go sendData(ch)
    //getData(ch)
```

```
    //那这种情况上来就没有go，没有go的协程，怎么通过channel发送数据，肯定也一样发数据的时候同步阻塞，注意发送的时候一定要有人来接(无缓冲的时候)，而这个时候channel的发送端也没有架设呢，最起码保证发送端go协程 channel嫁接起来，至于接收端使用channel嫁接，有go则更智能，没go就傻点不过可以完成任务。
    //sendData(ch)
    //go getData(ch)

    go sendData(ch)
    go getData(ch)

    time.Sleep(1e9)
}

func sendData(ch chan string) {
    ch <- "Washington"
    ch <- "Tripoli"
    ch <- "London"
    ch <- "Beijing"
    ch <- "Tokio"
}

func getData(ch chan string) {
    var input string
    // time.Sleep(2e9)
    for {
        input = <-ch
        fmt.Printf("%s ", input)
    }

    //果然是可以的
    /*for i := 0; i < 5; i++ {
        input = <-ch
        fmt.Printf("%s ", input)
    }*/
}
```

以下这段代码，循环次数10 几乎不会死循环
但是次数多了100，1000 就很容易造成死循环。 原因是c永远达不到100 只能是99，因为并发执行
```go
package main

import (
"sync"
```

"fmt"
"runtime"
"strconv"
)

var count int = 0
func counter(lock *sync.Mutex) {
//lock.Lock()

count++
fmt.Println("count==" + strconv.Itoa(count))
//lock.Unlock()
}

func main() {
lock := &sync.Mutex{}
for i := 0; i < 100; i++ {
go counter(lock)
}

```
    for {
            //lock.Lock() //为了安全
            c:=count
            fmt.Println("c=="+strconv.Itoa(c))
            //lock.Unlock()
            runtime.Gosched()
            if c >= 100 {
                    break
             }
    }
```

}

使用了锁变量（属于一种共享内存）来同步协程，事实上Go语言主要使用消息机制（channel）来作为通信模型。

> 默认情况下，通信是同步且无缓冲的：在有接受者接收数据之前，发送不会结束。可以想象一个无缓冲的通道在没有空间来保存数据的时候：必须要一个接收者准备好接收通道的数据然后发送者可以直接把数据发送给接收者。所以通道的发送/接收操作在对方准备好之前是阻塞的：
>1）对于同一个通道，发送操作（协程或者函数中的），在接收者准备好之前是阻塞的：如果ch中的数

据无人接收，就无法再给通道传入其他数据：新的输入无法在通道非空的情况下传入。所以发送操作会等待 ch 再次变为可用状态：就是通道值被接收时（可以传入变量）。
>2）对于同一个通道，接收操作是阻塞的（协程或函数中的），直到发送者可用：如果通道中没有数据，接收者就阻塞了。
>尽管这看上去是非常严格的约束，实际在大部分情况下工作的很不错。

>goroutine是Go并行设计的核心。goroutine说到底其实就是协程,但是它比线程更小,十几个goroutine可能体现在底层就是五六个线程,Go语言内部帮你实现了这些goroutine之间的内存共享。执行goroutine只需极少的栈内存(大概是4~5KB),当然会根据相应的数据伸缩。也正因为如此,可同时运行成千上万个并发任务。goroutine比thread更易用、更高效、更轻便。

goroutine是go runtime管理的一个**线程管理器**。
通过`go`关键字实现goroutine。通过`go`关键字 后跟函数(必须跟函数)，就启动了一个goroutine。

```go
go hello(a,b,c)
```

```go
package main
import(
    "fmt"
    "runtime"
)

func say(msg string) {
    for i:=0; i<5; i++ {
        runtime.Gosched()
        fmt.Println(msg)
        // time.sleep(1*time.Second)
    }
}

func main() {
    go say("world") //开一个新的Goroutines执行
    say("hello") //当前Goroutines执行
}
//每次输出的结果不同，但是hello每次都可以保证输出5次，不论顺序。而world极端情况没有输出，是因为在另外一个goroutines中输出。
```

> 可以看到go关键字很方便的就实现了并发编程。 上面的多个goroutine运行在同一个进程里面,共享内存数据,不过设计上我们要遵循:不要通过共享来通信,而要通过通信来共享。

go routine

goroutine运行在相同的地址空间,因此访问共享内存必须做好同步。那么goroutine之间如何进行数据的通 信呢,Go提供了一个很好的通信机制channel。channel可以与Unix shell 中的双向管道做类比:可以通过 它发送或者接收值。这些值只能是特定的类型:channel类型。定义一个channel时,也需要定义发送到 channel的值的类型。注意,**必须使用make创建channel**

```
ci := make(chan int)
cs := make(chan string)
cf := make(chan interface)
```

channel通过<-接收和发送数据，接收和发送取决于chan的的位置
ch <- v //发送v到channel ch
v := <-ch // 从ch中接收数据，并赋值给v

```
package main

import "fmt"

func sum(a []int, ch chan int) {
    sum := 0
    for _, v := range a {
        sum += v
    }
    ch <- sum
}

func main() {
    ch := make(chan int)
    s := []int{1, -4, -9, 20, 4, 9}
    go sum(s[:len(s)/2], ch)
    go sum(s[len(s)/2:], ch)
    //x, y := <-ch, ch
    x, y := <-ch, <-ch
    fmt.Printf("(%d)+(%d)=%d\n", x, y, x+y)
}
```

> 默认情况下,channel接收和发送数据都是阻塞的,除非另一端已经准备好,这样就使得Goroutines同步变的更加的简单,而不需要显式的lock。所谓阻塞,也就是如果读取(value := <-ch)它将会被阻塞,直到有数据接收。其次,任何发送(ch<-5)将会被阻塞,直到数据被读出。无缓冲channel是在多个goroutine之 间同步很棒的工具。

```
package main
import "fmt"
func main() {
```

```
    ch := make(chan int, 2)
    ch <- 1
    ch <- 2

    fmt.Println(<- ch)
    fmt.Println(<- ch)
}
```

将buff值修改为0，1都会
```
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan send]:
main.main()
/Users/fqc/work/src/run.go:8 +0x9c
```

使用range读取channel数据，for range channel专门读取channel的数据，无需关注索引。

```
package main

import "fmt"

func fibonacci(n int, ch chan int) {
    x, y := 1, 1
    for i := 0; i < n; i++ {
        ch <- x
        x, y = y, x+y
    }
    close(ch)
}

func main() {
    ch := make(chan int,10)
    go fibonacci(cap(ch), ch)
    for  v := range ch {
        fmt.Println(v)
    }
}
```

> for i := range c能够不断的读取channel里面的数据,直到该channel被显式的关闭。上面代码我们看 到可以显式的关闭channel,生产者通过内置函数 close 关闭channel。关闭channel之

后就无法再发送任 何数据了,在消费方可以通过语法 v, ok := <-ch 测试channel是否被关闭。如果ok返回false,那么说channel已经没有任何数据并且已经被关闭。
记住应该在生产者的地方关闭channel,而不是消费的地方去关闭它,这样容易引起panic,另外记住一点的就是channel不像文件之类的,不需要经常去关闭,只有当你确实没有任何发送数据了,或者你想显式的结束range循环之类的

如果去除close(ch)，也将会dead lock

reference

《build web app with go》
《the way to go》