```
func main() {
    catChan := make(chan struct{})
    dogChan := make(chan struct{})
    fishChan := make(chan struct{})

    go func(catChan, dogChan chan struct{}) {
        for {
            <-catChan
            fmt.Println("cat")
            dogChan <- struct{}{}
        }
    }(catChan, dogChan)

    go func(dogChan, fishChan chan struct{}) {
        for {
            <-dogChan
            fmt.Println("dog")
            fishChan <- struct{}{}
        }
    }(dogChan, fishChan)

    go func(fishChan, catChan chan struct{}) {
        for {
            <-fishChan
            fmt.Println("fish")
            catChan <- struct{}{}
        }
    }(fishChan, catChan)

    catChan <- struct{}{}

    time.Sleep(1 * time.Millisecond)
}
```

```
package main

import (
    "fmt"
    "time"
)

func main() {
    catChan := make(chan struct{})
    dogChan := make(chan struct{})
    fishChan := make(chan struct{})

    go printCat(catChan, dogChan)
    go printDog(dogChan, fishChan)
    go printFish(fishChan, catChan)
    defer func() {
        close(catChan)
        close(dogChan)
        close(fishChan)
    }()
    //先启动所有 Goroutine，然后再发送信号
```

```
    //这样可以确保 Goroutine 已经准备好接收信号，不会导致信号丢失
    catChan <- struct{}{}

    time.Sleep(10 * time.Second)
}

func printCat(catChan, dogChan chan struct{}) {
    for {
        <-catChan
        fmt.Println("cat")
        time.Sleep(1 * time.Second)
        dogChan <- struct{}{}
    }
}

func printDog(dogChan, fishChan chan struct{}) {
    for {
        <-dogChan
        fmt.Println("dog")
        time.Sleep(1 * time.Second)
        fishChan <- struct{}{}
    }
}

func printFish(fishChan, catChan chan struct{}) {
    for {
        <-fishChan
        fmt.Println("fish")
        time.Sleep(1 * time.Second)
        catChan <- struct{}{}
    }
}
```

2.实现两个协程轮流输出A 1 B 2 C 3 ... Z 26

```
package main

import (
    "fmt"
    "time"
)

func main() {
    letterChan := make(chan struct{})
    NumberChan := make(chan struct{})

    go func() {
        for i := 'A'; i <= 'Z'; i++ {
            <-letterChan
            fmt.Printf("%c ", i)
            NumberChan <- struct{}{}
        }
    }()
    go func() {
        for i := 1; i <= 26; i++ {
```

```
            <-NumberChan
            fmt.Printf("%d ", i)
            if i < 26 {
                letterChan <- struct{}{}
            }
        }
    }()

    letterChan <- struct{}{}

    time.Sleep(1 * time.Second)
}
```

```
package main

import (
    "fmt"
    "time"
)

func main() {
    letterChan := make(chan struct{})
    numberChan := make(chan struct{})

    go func() {
        for i := 'A'; i <= 'Z'; i++ {
            <-letterChan
            fmt.Printf("%c ", i)
            numberChan <- struct{}{}
        }
    }()
    go func() {
        for i := 1; i <= 26; i++ {
            <-numberChan
            fmt.Printf("%d ", i)
            if i < 26 {
                letterChan <- struct{}{}
            }
        }
    }()
    defer func() {
        close(letterChan)
        close(numberChan)
    }()

    letterChan <- struct{}{}

    time.Sleep(1 * time.Second)
}
```

3.n个goroutine顺序打印数字

```
package main

import (
    "fmt"
    "time"
)

func main() {
    const N = 10

    channels := make([]chan struct{}, N)
    for i := range N {
        channels[i] = make(chan struct{})
    }

    for i := range N {
        go func(i int) {
            for {
                <-channels[i]
                fmt.Printf("Goroutine %d 正在打印: %d\n", i, i+1)
                channels[(i+1)%N] <- struct{}{}
            }
        }(i)
    }
    defer func() {
        for i := range N {
            close(channels[i])
        }
    }()

    channels[0] <- struct{}{}

    time.Sleep(1 * time.Second)
}
```

```
package main

import (
    "fmt"
    "time"
)

const N = 10

func main() {
    channels := make([]chan struct{}, N)
    for i := range channels {
        channels[i] = make(chan struct{})
    }

    for i := range N {
        go printNumber(i, channels[i], channels[(i+1)%N])
```

```
    }
    defer func() {
        for i := range N {
            close(channels[i])
        }
    }()

    channels[0] <- struct{}{}

    time.Sleep(1 * time.Millisecond)
}

func printNumber(i int, curChan, nextChan chan struct{}) {
    for {
        <-curChan
        fmt.Printf("Goroutine %d 正在打印: %d\n", i, i+1)
        nextChan <- struct{}{}
    }
}
```

4.下面函数执行结果是啥

第一次调用 doAppend(s[:4])

s[:4] 和 s 共享同一个底层数组

append 操作没有触发扩容，直接在底层数组的第 5 个位置添加了 1

因此，外部的 s 也被修改了

第二次调用 doAppend(s)

s 的长度和容量都是 8，append 操作触发了扩容

扩容后，doAppend 函数中的 s 指向一个新的底层数组，而外部的 s 仍然指向原来的底层数组

因此，外部的 s 没有被修改

```
package main

import "fmt"

func main() {
    s := make([]int, 8, 8)
    //s[:4]创建了s的一个切片，该切片从s的开始到索引4（不包括索引4），因此它包含s的前4个元素
    //由于切片操作并没有指定容量，所以新切片的容量仍然是8（即原slice的容量）
    //由于其容量允许，增加的元素被放置在原始数组的第五个位置上
    //这个操作改变了底层数组，s的内容也被改变了
    doAppend(s[:4])
    //可以看到之前doAppend函数中的更改
    printLengthAndCapacity(s)
    //会创建一个新的底层数组，并将原数组的内容复制过去，然后添加新的元素1
    //这一次append操作的结果是新的slice，它包含了原数组的所有元素加上新元素
    doAppend(s)
    //上条语句的创建的新slice并没有赋值回到外部的s变量
    printLengthAndCapacity(s)
}
```

```
func doAppend(s []int) {
    s = append(s, 1)
    printLengthAndCapacity(s)
}

func printLengthAndCapacity(s []int) {
    fmt.Println(s)
    fmt.Printf("len=%d cap=%d\n", len(s), cap(s))
}
-----------------------------------------
[0 0 0 0 1]
len=5 cap=8
[0 0 0 0 1 0 0 0]
len=8 cap=8
[0 0 0 0 1 0 0 0 1]
len=9 cap=16
[0 0 0 0 1 0 0 0]
len=8 cap=8
```

5.下面代码的输出是啥

runnext优先级更高

减少延迟：最新创建的 G 通常是用户最关心的任务，优先执行可以减少延迟。

提高局部性：最新创建的 G 可能仍然在 CPU 缓存中，优先执行可以提高性能。

避免饥饿：如果 runq 中有大量 G，新创建的 G 可能会被长时间阻塞。runnext 确保新 G 能够尽快执行

```
package main

import (
    "runtime"
    "sync"
)

func main() {
    //设置只有一个P可以工作
    runtime.GOMAXPROCS(1)

    var wg sync.WaitGroup
    var n = 10

    //启动n个goroutine打印，哪个goroutine最先打印
    wg.Add(n)

    // Go/src/runtime/proc.go newproc(fn *funcval)
    for i := 1; i <= n; i++ {
        go func(i int) {
            defer wg.Done()
            println("I am goroutine ", i)
        }(i)
    }

    wg.Wait()
```

```
}
--------------------------------
I am goroutine  10
I am goroutine  1
I am goroutine  2
I am goroutine  3
I am goroutine  4
I am goroutine  5
I am goroutine  6
I am goroutine  7
I am goroutine  8
I am goroutine  9
```

每个P都是长度为256的队列（runq），还有长度为1的指针（runnext）指向下个要执行的G

创建g1到runnext

g1放到runq，创建g2到runnext

g2放到runq，创建g3到runnext

……

g9放到runq，创建g10到runnext

全部创建完

执行runnext中的g10

执行runq中的g1~g9

```
runnext: g10
runq: g1 g2 g3 g4 g5 g6 g7 g8 g9
```

```
func newproc(fn *funcval) {
    // 获取当前执行的goroutine
    gp := getg()

    // 获取调用者的程序计数器地址（即获取到调用newproc函数的地方的返回地址）
    pc := getcallerpc()

    // 使用systemstack切换到系统栈执行以下函数
    // 因为创建goroutine涉及底层数据结构的更改，需要在系统栈上操作以避免影响用户栈
    systemstack(func() {
        // 调用newproc1，实际上完成创建goroutine的工作
        // fn是新goroutine将要执行的函数
        // gp是当前goroutine的指针
        // pc是程序计数器的值
        // false表示这个goroutine不是系统goroutine
        // waitReasonZero是等待原因的初始状态
        newg := newproc1(fn, gp, pc, false, waitReasonZero)

        // 获取当前M对应的P的指针
        pp := getg().m.p.ptr()

        // 把新创建的goroutine放入本地可运行队列
        runqput(pp, newg, true)
```

```
        // 如果main函数已经启动，则可能需要唤醒一个线程(M)来执行新的goroutine
        if mainStarted {
            wakep()
        }
    })
}
```

6.有一个数组，用两个协程，一个打印所有偶数的和，一个打印所有奇数的和，要用for channel机制（腾讯一面）

```
package main

import (
    "fmt"
    "sync"
)

func main() {
    var wg sync.WaitGroup
    arr := []int{1, 2, 3, 4, 5, 6, 7, 8, 9}
    evenCh := make(chan int) // 偶数通道
    oddCh := make(chan int)  // 奇数通道


    go func() {
        // 关闭通道不会影响通道中已存在的数据，只是通知接收方“不会再有新数据发送”
        defer close(evenCh)
        defer close(oddCh)
        for _, num := range arr {
            if num%2 == 0 {
                evenCh <- num
            } else {
                oddCh <- num
            }
        }
    }()

    // 偶数求和 Goroutine
    wg.Add(1)
    go func() {
        defer wg.Done()
        evenSum := 0
        for num := range evenCh {
            evenSum += num
        }
        fmt.Println("偶数和:", evenSum) // 2 + 4 + 6 + 8 = 20
    }()

    // 奇数求和 Goroutine
    wg.Add(1)
    go func() {
        defer wg.Done()
        oddSum := 0
```

```
        for num := range oddCh {
            oddSum += num
        }
        fmt.Println("奇数和:", oddSum) // 1 + 3 + 5 + 7 + 9 = 25
    }()

    wg.Wait()
}
```

7.10个生产者，5个消费者，生产者总共生产1000个消费物料（编号1-1000），5个消费者并行消费

```
package main

import (
    "fmt"
    "sync"
)

func main() {
    const (
        numProducers = 10
        numConsumers = 5
        totalJobs    = 1000
    )

    jobs := make(chan int, totalJobs)          // 生产任务通道（缓冲大小为总任务数）
    materials := make(chan int, numProducers) // 物料传递通道（带缓冲）

    // 1、主协程预先生成1-1000的任务编号
    go func() {
        for i := 1; i <= totalJobs; i++ {
            jobs <- i
        }
        close(jobs) // 任务生成完毕后关闭通道
    }()

    // 2.启动10个生产者协程
    var wgProducers sync.WaitGroup
    wgProducers.Add(numProducers)
    for i := 0; i < numProducers; i++ {
        go func() {
            defer wgProducers.Done()
            for job := range jobs {
                // 从jobs通道读取任务
                materials <- job // 将物料发送到materials通道
            }
        }()
    }

    // 3.等待所有生产者结束，关闭materials通道
    go func() {
        wgProducers.Wait()
        close(materials)
```

```
    }()

    // 4.启动5个消费者协程
    var wgConsumers sync.WaitGroup
    wgConsumers.Add(numConsumers)
    for i := 0; i < numConsumers; i++ {
        go func() {
            defer wgConsumers.Done()
            for material := range materials { // 从materials通道读取物料
                fmt.Printf("消费者处理物料：%d\n", material)
            }
        }()
    }

    // 5.等待所有消费者结束
    wgConsumers.Wait()
    fmt.Println("所有物料处理完毕")
}
```

8.通过协程(goroutine)和通道(channel)实现多个协程执行随机数加法，最后输出其中最大值

```
package main

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

func main() {
    // 设置随机数种子，确保每次运行结果不同
    rand.Seed(time.Now().UnixNano())

    const numGoroutines = 10 // 定义协程数量
    resultChan := make(chan int)

    // 启动多个协程生成随机数并计算加法
    for i := 0; i < numGoroutines; i++ {
        go func() {
            a := rand.Intn(100) // 生成0-99的随机数
            b := rand.Intn(100)
            sum := a + b
            resultChan <- sum // 将结果发送到通道
        }()
    }

    // 主协程收集结果并计算最大值
    // 在 Go 中，main() 函数是程序的入口点，主协程从这里开始执行
    max := 0
    for i := 0; i < numGoroutines; i++ {
        current := <-resultChan
        if current > max {
            max = current
```

```
        }
    }

    fmt.Printf("最大值为: %d\n", max)
}
```

9.使用两个goroutine交替打印1-100之间的奇数和偶数，输出时按照从小到大输出

```
package main

import (
    "fmt"
    "sync"
)

func main() {
    // 创建两个无缓冲通道，用于控制交替执行
    oddChan := make(chan struct{})  // 奇数协程信号通道
    evenChan := make(chan struct{}) // 偶数协程信号通道
    var wg sync.WaitGroup
    wg.Add(2)

    // 奇数协程
    go func() {
        defer wg.Done()
        for i := 1; i <= 99; i += 2 {
            <-oddChan
            fmt.Println("奇数:", i)  // 打印奇数
            evenChan <- struct{}{} // 通知偶数协程
        }
    }()

    // 偶数协程
    go func() {
        defer wg.Done()
        for i := 2; i <= 100; i += 2 {
            <-evenChan
            fmt.Println("偶数:", i) // 打印偶数
            if i < 100 {
                oddChan <- struct{}{} // 通知奇数协程（最后一次无需通知）
            }
        }
    }()

    oddChan <- struct{}{} // 初始触发奇数协程
    wg.Wait()             // 等待两个协程完成
}
```

10.创建10个goroutine,id分别是0,1,2,3...9，每个goroutine只能打印最后一位是自己id号的数字，例如: 3号只能打印3,13,23,33...编写一个程序，依次打印1-10000

```go
package main

import (
    "fmt"
    "sync"
)

func main() {
    const (
        maxNumber = 10000
        workers   = 10
    )

    numberChan := make(chan int, 100)
    var wg sync.WaitGroup

    // 启动 10 个 worker goroutine
    for id := 0; id < workers; id++ {
        wg.Add(1)
        go func(workerID int) {
            defer wg.Done()
            for num := range numberChan {
                if num%10 == workerID {
                    fmt.Printf("%5d (worker %d)\n", num, workerID) // 格式化输出
                }
            }
        }(id)
    }

    // 发送数字到通道
    for i := 1; i <= maxNumber; i++ {
        numberChan <- i
    }
    close(numberChan)

    wg.Wait()
    fmt.Println("所有数字打印完成")
}
```