



链滴

Go 边看边练 - 《Go 学习笔记》系列（十三）

作者: [88250](#)

原文链接: <https://ld246.com/article/1438938175118>

来源网站: [链滴](#)

许可协议: [署名-相同方式共享 4.0 国际 \(CC BY-SA 4.0\)](#)

ToC

- [Go 边看边练 - 《Go 学习笔记》系列 \(一\) - 变量、常量](#)
 - [Go 边看边练 - 《Go 学习笔记》系列 \(二\) - 类型、字符串](#)
 - [Go 边看边练 - 《Go 学习笔记》系列 \(三\) - 指针](#)
 - [Go 边看边练 - 《Go 学习笔记》系列 \(四\) - 控制流1](#)
 - [Go 边看边练 - 《Go 学习笔记》系列 \(五\) - 控制流2](#)
 - [Go 边看边练 - 《Go 学习笔记》系列 \(六\) - 函数](#)
 - [Go 边看边练 - 《Go 学习笔记》系列 \(七\) - 错误处理](#)
 - [Go 边看边练 - 《Go 学习笔记》系列 \(八\) - 数组、切片](#)
 - [Go 边看边练 - 《Go 学习笔记》系列 \(九\) - Map、结构体](#)
 - [Go 边看边练 - 《Go 学习笔记》系列 \(十\) - 方法](#)
 - [Go 边看边练 - 《Go 学习笔记》系列 \(十一\) - 表达式](#)
 - [Go 边看边练 - 《Go 学习笔记》系列 \(十二\) - 接口](#)
 - [Go 边看边练 - 《Go 学习笔记》系列 \(十三\) - Goroutine](#)
 - [Go 边看边练 - 《Go 学习笔记》系列 \(十四\) - Channel](#)
-

7.1 Goroutine

Go 在语言层面对并发编程提供支持, 一种类似协程, 称作 **goroutine** 的机制。

只需在函数调用语句前添加 **go** 关键字, 就可创建并发执行单元。开发人员无需了解任何执行细节, 调度器会自动将其安排到合适的系统线程上执行。**goroutine** 是一种非非常轻量级的实现, 可在单个程序里执行成千上万的并发任务。

事实上, 入口函数 **main** 就以 **goroutine** 运行。另有与之配套的 **channel** 类型, 用以实现 "以通讯来享内存" 的 **CSP** 模式。相关实现细节可参考本书第二部分的源码剖析。

```
go func() {  
    println("Hello, World!")  
}
```

调度器不能保证多个 **goroutine** 执行次序, 且进程退出时不会等待它们结束。

默认情况下, 进程启动后仅允许一个系统线程服务于 **goroutine**。可使用环境变量或标准库函数 **runtime.GOMAXPROCS** 修改, 让调度器用多个线程实现多核并行, 而不仅仅是并发。

```
func sum(id int) {  
    var x int64
```

```

    for i := 0; i < math.MaxUint32; i++ {
        x += int64(i)
    }

    println(id, x)
}

func main() {
    wg := new(sync.WaitGroup)
    wg.Add(2)

    for i := 0; i < 2; i++ {
        go func(id int) {
            defer wg.Done()
            sum(id)
        }(i)
    }

    wg.Wait()
}

```

输出：

```

$ go build -o test
$ time -p ./test

```

```

0 9223372030412324865
1 9223372030412324865

```

```

real 7.70 // 程序开始到结束时间差 (非 CPU 时间)
user 7.66 // 用户态所使用 CPU 时间片 (多核累加)
sys 0.01 // 内核态所使用 CPU 时间片

```

```

$ GOMAXPROCS=2 time -p ./test

```

```

0 9223372030412324865
1 9223372030412324865

```

```

real 4.18
user 7.61 // 虽然总时间差不多，但由 2 个核并行行，real 时间自自然然少了许多。
sys 0.02

```

调用 `runtime.Goexit` 将立即终止当前 `goroutine` 执行，调度器确保所有已注册 `defer` 延迟调用被执行。

<https://wide.b3log.org/playground/83499692f329caf416974c5b5d2d473.go?embed=true>

和协程 `yield` 作用类似，`Gosched` 让出底层线程，将当前 `goroutine` 暂停，放回队列等待下次被调执行。

<https://wide.b3log.org/playground/9689fab1b228ea611e5fee79435989b.go?embed=true>

7.2 Channel

引用类型 `channel` 是 `CSP` 模式的具体实现，用于多个 `goroutine` 通讯。其内部实现了同步，确保安全。

默认为同步模式，需要发送和接收配对。否则会被阻塞，直到另一方准备好后被唤醒。

`<iframe style="border:1px solid" src="https://wide.b3log.org/playground/15183c59def4d5240c1490785075c42.go?embed=true" width="99%" height="630"></iframe>`

异步方式通过判断缓冲区来决定是否阻塞。如果缓冲区已满，发送被阻塞；缓冲区为空，接收被阻塞。

通常情况下，异步 `channel` 可减少排队阻塞，具备更高的效率。但应该考虑使用指针规避大对象拷贝将多个元素打包，减小缓冲区大小等。

```
func main() {
    data := make(chan int, 3) // 缓冲区可以存储 3 个元素
    exit := make(chan bool)

    data <- 1 // 在缓冲区未滿前，不会阻塞。
    data <- 2
    data <- 3

    go func() {
        for d := range data { // 在缓冲区未空前，不会阻塞。
            fmt.Println(d)
        }

        exit <- true
    }()

    data <- 4 // 如果缓冲区已滿，阻塞。
    data <- 5
    close(data)

    <-exit
}
```

缓冲区是内部属性，并非类型构成要素。

```
var a, b chan int = make(chan int), make(chan int, 3)
```

除用 `range` 外，还可用 `ok-idiom` 模式判断 `channel` 是否关闭。

```
for {
    if d, ok := <-data; ok {
        fmt.Println(d)
    } else {
        break
    }
}
```

向 **closed channel** 发送数据引发 **panic** 错误，接收立即返回零值。而 **nil channel**，无论收发都会被塞。

内置函数 **len** 返回未被读取的缓冲元素数量，**cap** 返回缓冲区大小。

```
d1 := make(chan int)
d2 := make(chan int, 3)

d2 <- 1

fmt.Println(len(d1), cap(d1)) // 0 0
fmt.Println(len(d2), cap(d2)) // 1 3
```

下一篇：<https://hacpai.com/article/1439194647152>

-
- 本系列是基于雨痕的《Go 学习笔记》（第四版）整理汇编而成，非常感谢雨痕的辛勤付出与分享！
 - 转载请注明：文章转载自：**黑客与画家的社区** [<https://hacpai.com>]
 - 如果你觉得本章节做得不错，请在下面打赏一下吧~
-

社区小贴士

- 关注标签 [golang] 可以方便查看 Go 相关帖子
- 关注作者后如有新帖将会收到通知