Go示例学

itfanr

版权

原作者：https://gobyexample.com/
译者：@jemygraw @everyx @mmcgrana
整理：@itfanr

参考：

- http://gobyexample.everyx.in/
- https://github.com/jemygraw/TechDoc
- https://github.com/jemygraw/GoQuickLearn
- https://github.com/mmcgrana/gobyexample
- https://github.com/everyx/gobyexample

内容

Go 状态协程

Go 状态协程

在上面的例子中，我们演示了如何通过使用mutex来在多个协程之间共享状态。另外一种方法是使用协程内置的同步机制来实现。这种基于通道的方法和Go的通过消息共享内存，保证每份数据为单独的协程所有的理念是一致的。

```
package main

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

// 在这个例子中，将有一个单独的协程拥有这个状态。这样可以
// 保证这个数据不会被并行访问所破坏。为了读写这个状态，其
// 他的协程将向这个协程发送信息并且相应地接受返回信息。
// 这些`readOp`和`writeOp`结构体封装了这些请求和回复
type readOp struct {
    key  int
    resp chan int
}
type writeOp struct {
    key  int
    val  int
    resp chan bool
}

func main() {

    // 我们将计算我们执行了多少次操作
    var ops int64 = 0

    // reads和writes通道将被其他协程用来从中读取或写入数据
    reads := make(chan *readOp)
    writes := make(chan *writeOp)

    // 这个是拥有`state`的协程，`state`是一个协程的私有map
    // 变量。这个协程不断地`select`通道`reads`和`writes`，
    // 当有请求来临的时候进行回复。一旦有请求，首先执行所
    // 请求的操作，然后给`resp`通道发送一个表示请求成功的值。
```

```
    go func() {
        var state = make(map[int]int)
        for {
            select {
            case read := <-reads:
                read.resp <- state[read.key]
            case write := <-writes:
                state[write.key] = write.val
                write.resp <- true
            }
        }
    }()

    // 这里启动了100个协程来向拥有状态的协程请求读数据。
    // 每次读操作都需要创建一个`readOp`，然后发送到`reads`
    // 通道，然后等待接收请求回复
    for r := 0; r < 100; r++ {
        go func() {
            for {
                read := &readOp{
                    key:  rand.Intn(5),
                    resp: make(chan int)}
                reads <- read
                <-read.resp
                atomic.AddInt64(&ops, 1)
            }
        }()
    }

    // 我们开启10个写协程
    for w := 0; w < 10; w++ {
        go func() {
            for {
                write := &writeOp{
                    key:  rand.Intn(5),
                    val:  rand.Intn(100),
                    resp: make(chan bool)}
                writes <- write
                <-write.resp
                atomic.AddInt64(&ops, 1)
            }
        }()
    }

    // 让协程运行1秒钟
    time.Sleep(time.Second)

    // 最后输出操作数量ops的值
    opsFinal := atomic.LoadInt64(&ops)
    fmt.Println("ops:", opsFinal)
}
```

运行结果

```
ops: 880578
```

运行这个程序，我们会看到基于协程的状态管理每秒可以处理800, 000个操作。对于这个例子来讲，基于协程的方法比基于mutex的方法更加复杂一点。当然在某些情况下还是很有用的。例如你有很多复杂的协程，而且管理多个mutex可能导致错误。
当然你可以选择使用任意一种方法，只要你保证这种方法让你觉得很舒服而且也能保证程序的正确性。

Go 字典

Go 字典

字典是Go语言内置的关联数据类型。因为数组是索引对应数组元素，而字典是键对应值。

```
package main

import "fmt"

func main() {

    // 创建一个字典可以使用内置函数make
    // "make(map[键类型]值类型)"
    m := make(map[string]int)

    // 使用经典的"name[key]=value"来为键设置值
    m["k1"] = 7
    m["k2"] = 13

    // 用Println输出字典，会输出所有的键值对
    fmt.Println("map:", m)

    // 获取一个键的值 "name[key]".
    v1 := m["k1"]
    fmt.Println("v1: ", v1)

    // 内置函数返回字典的元素个数
    fmt.Println("len:", len(m))

    // 内置函数delete从字典删除一个键对应的值
    delete(m, "k2")
    fmt.Println("map:", m)

    // 根据键来获取值有一个可选的返回值，这个返回值表示字典中是否
    // 存在该键，如果存在为true，返回对应值，否则为false，返回零值
    // 有的时候需要根据这个返回值来区分返回结果到底是存在的值还是零值
    // 比如字典不存在键x对应的整型值，返回零值就是0，但是恰好字典中有
    // 键y对应的值为0，这个时候需要那个可选返回值来判断是否零值。
    _, ok := m["k2"]
    fmt.Println("ok:", ok)

    // 你可以用 ":=" 同时定义和初始化一个字典
    n := map[string]int{"foo": 1, "bar": 2}
    fmt.Println("map:", n)
}
```

输出结果为

```
map: map[k1:7 k2:13]
v1:  7
len: 2
map: map[k1:7]
ok: false
map: map[foo:1 bar:2]
```

Go 字符串操作函数

Go 字符串操作函数

strings 标准库提供了很多字符串操作相关的函数。这里提供的几个例子是让你先对这个包有个基本了解。

```
package main

import s "strings"
import "fmt"

// 这里给fmt.Println起个别名，因为下面我们会多处使用。
var p = fmt.Println

func main() {

    // 下面是strings包里面提供的一些函数实例。注意这里的函数并不是
    // string对象所拥有的方法，这就是说使用这些字符串操作函数的时候
    // 你必须将字符串对象作为第一个参数传递进去。
    p("Contains:  ", s.Contains("test", "es"))
    p("Count:     ", s.Count("test", "t"))
    p("HasPrefix: ", s.HasPrefix("test", "te"))
    p("HasSuffix: ", s.HasSuffix("test", "st"))
    p("Index:     ", s.Index("test", "e"))
    p("Join:      ", s.Join([]string{"a", "b"}, "-"))
    p("Repeat:    ", s.Repeat("a", 5))
    p("Replace:   ", s.Replace("foo", "o", "0", -1))
    p("Replace:   ", s.Replace("foo", "o", "0", 1))
    p("Split:     ", s.Split("a-b-c-d-e", "-"))
    p("ToLower:   ", s.ToLower("TEST"))
    p("ToUpper:   ", s.ToUpper("test"))
    p()

    // 你可以在strings包里面找到更多的函数

    // 这里还有两个字符串操作方法，它们虽然不是strings包里面的函数，
    // 但是还是值得提一下。一个是获取字符串长度，另外一个是从字符串中
    // 获取指定索引的字符
    p("Len: ", len("hello"))
    p("Char:", "hello"[1])
}
```

运行结果

```
Contains:  true
Count:     2
HasPrefix: true
HasSuffix: true
Index:     1
Join:      a-b
Repeat:    aaaaa
Replace:   f00
Replace:   f0o
Split:     [a b c d e]
ToLower:   test
ToUpper:   TEST

Len:  5
Char: 101
```

Go 字符串格式化

Go 字符串格式化

Go对字符串格式化提供了良好的支持。下面我们看些常用的字符串格式化的例子。

```
package main

import "fmt"
import "os"

type point struct {
    x, y int
}

func main() {

    // Go提供了几种打印格式，用来格式化一般的Go值，例如
    // 下面的%v打印了一个point结构体的对象的值
    p := point{1, 2}
    fmt.Printf("%v\n", p)

    // 如果所格式化的值是一个结构体对象，那么`%+v`的格式化输出
    // 将包括结构体的成员名称和值
    fmt.Printf("%+v\n", p)

    // `%#v`格式化输出将输出一个值的Go语法表示方式。
    fmt.Printf("%#v\n", p)

    // 使用`%T`来输出一个值的数据类型
    fmt.Printf("%T\n", p)
```

```
    // 格式化布尔型变量
    fmt.Printf("%t\n", true)

    // 有很多的方式可以格式化整型，使用`%d`是一种
    // 标准的以10进制来输出整型的方式
    fmt.Printf("%d\n", 123)

    // 这种方式输出整型的二进制表示方式
    fmt.Printf("%b\n", 14)

    // 这里打印出该整型数值所对应的字符
    fmt.Printf("%c\n", 33)

    // 使用`%x`输出一个值的16进制表示方式
    fmt.Printf("%x\n", 456)

    // 浮点型数值也有几种格式化方法。最基本的一种是`%f`
    fmt.Printf("%f\n", 78.9)

    // `%e`和`%E`使用科学计数法来输出整型
    fmt.Printf("%e\n", 123400000.0)
    fmt.Printf("%E\n", 123400000.0)

    // 使用`%s`输出基本的字符串
    fmt.Printf("%s\n", "\"string\"")

    // 输出像Go源码中那样带双引号的字符串，需使用`%q`
    fmt.Printf("%q\n", "\"string\"")

    // `%x`以16进制输出字符串，每个字符串的字节用两个字符输出
    fmt.Printf("%x\n", "hex this")

    // 使用`%p`输出一个指针的值
    fmt.Printf("%p\n", &p)

    // 当输出数字的时候，经常需要去控制输出的宽度和精度。
    // 可以使用一个位于%后面的数字来控制输出的宽度，默认
    // 情况下输出是右对齐的，左边加上空格
    fmt.Printf("|%6d|%6d|\n", 12, 345)

    // 你也可以指定浮点数的输出宽度，同时你还可以指定浮点数
    // 的输出精度
    fmt.Printf("|%6.2f|%6.2f|\n", 1.2, 3.45)

    // To left-justify, use the `-` flag.
    fmt.Printf("|%-6.2f|%-6.2f|\n", 1.2, 3.45)

    // 你也可以指定输出字符串的宽度来保证它们输出对齐。默认
    // 情况下，输出是右对齐的
    fmt.Printf("|%6s|%6s|\n", "foo", "b")

    // 为了使用左对齐你可以在宽度之前加上`-`号
    fmt.Printf("|%-6s|%-6s|\n", "foo", "b")

    // `Printf`函数的输出是输出到命令行`os.Stdout`的，你
```

```
    // 可以用`Sprintf`来将格式化后的字符串赋值给一个变量
    s := fmt.Sprintf("a %s", "string")
    fmt.Println(s)

    // 你也可以使用`Fprintf`来将格式化后的值输出到`io.Writers`
    fmt.Fprintf(os.Stderr, "an %s\n", "error")
}
```

运行结果

```
{1 2}
{x:1 y:2}
main.point{x:1, y:2}
main.point
true
123
1110
!
1c8
78.900000
1.234000e+08
1.234000E+08
"string"
"\"string\""
6865782074686973
0x103a10c0
|    12|   345|
|  1.20|  3.45|
|1.20  |3.45  |
|   foo|     b|
|foo   |b     |
a string
an error
```

Go 自定义排序

Go 自定义排序

有的时候我们希望排序不是仅仅按照自然顺序排序。例如，我们希望按照字符串的长度来对一个字符串数组排序而不是按照字母顺序来排序。这里我们介绍一下Go的自定义排序。

```
package main

import "sort"
import "fmt"

// 为了能够使用自定义函数来排序，我们需要一个
// 对应的排序类型，比如这里我们为内置的字符串
// 数组定义了一个别名ByLength
type ByLength []string

// 我们实现了sort接口的Len，Less和Swap方法
// 这样我们就可以使用sort包的通用方法Sort
// Len和Swap方法的实现在不同的类型之间大致
// 都是相同的，只有Less方法包含了自定义的排序
// 逻辑，这里我们希望以字符串长度升序排序
func (s ByLength) Len() int {
    return len(s)
}
func (s ByLength) Swap(i, j int) {
    s[i], s[j] = s[j], s[i]
}
func (s ByLength) Less(i, j int) bool {
    return len(s[i]) < len(s[j])
}

// 一切就绪之后，我们就可以把需要进行自定义排序
// 的字符串类型fruits转换为ByLength类型，然后使用
// sort包的Sort方法来排序
func main() {
    fruits := []string{"peach", "banana", "kiwi"}
    sort.Sort(ByLength(fruits))
    fmt.Println(fruits)
}
```

输出结果

```
[kiwi peach banana]
```

同样的，对于其他的类型，使用这种方法，我们可以为Go的切片提供任意的排序方法。归纳一下就是：

1. 创建自定义排序类型
2. 实现sort包的接口方法Len,Swap和Less
3. 使用sort.Sort方法来排序

Go Base64编码

Go Base64编码

Go提供了对base64编码和解码的内置支持

```
package main

// 这种导入包的语法将默认的base64起了一个别名b64，这样
// 我们在下面就可以直接使用b64表示这个包，省点输入量
import b64 "encoding/base64"
import "fmt"

func main() {

    // 这里是我们用来演示编码和解码的字符串
    data := "abc123!?$*&()'-=@~"

    // Go支持标准的和兼容URL的base64编码。
    // 我们这里使用标准的base64编码，这个
    // 函数需要一个`[]byte`参数，所以将这
    // 个字符串转换为字节数组
    sEnc := b64.StdEncoding.EncodeToString([]byte(data))
    fmt.Println(sEnc)

    // 解码一个base64编码可能返回一个错误，
    // 如果你不知道输入是否是正确的base64
    // 编码，你需要检测一些解码错误
    sDec, _ := b64.StdEncoding.DecodeString(sEnc)
    fmt.Println(string(sDec))
    fmt.Println()

    // 使用兼容URL的base64编码和解码
    uEnc := b64.URLEncoding.EncodeToString([]byte(data))
    fmt.Println(uEnc)
    uDec, _ := b64.URLEncoding.DecodeString(uEnc)
    fmt.Println(string(uDec))
}
```

运行结果

```
YWJjMTIzIT8kKiYoKSctPUB+
abc123!?$*&()'-=@~

YWJjMTIzIT8kKiYoKSctPUB-
abc123!?$*&()'-=@~
```

这两种方法都将原数据编码为base64编码，区别在于标准的编码后面是`+`，而兼容URL的编码方式后面是`-`。

Go Defer

Go Defer

Defer 用来保证一个函数调用会在程序执行的最后被调用。通常用于资源清理工作。

```
package main

import "fmt"
import "os"

// 假设我们想创建一个文件，然后写入数据，最后关闭文件
func main() {
    // 在使用createFile得到一个文件对象之后，我们使用defer
    // 来调用关闭文件的方法closeFile，这个方法将在main函数
    // 最后被执行，也就是writeFile完成之后
    f := createFile("/tmp/defer.txt")
    // Windows下面使用这个语句
    // f := createFile("D:\\Temp\\defer.txt")
    defer closeFile(f)
    writeFile(f)
}

func createFile(p string) *os.File {
    fmt.Println("creating")
    f, err := os.Create(p)
    if err != nil {
        panic(err)
    }
    return f
}

func writeFile(f *os.File) {
    fmt.Println("writing")
    fmt.Fprintln(f, "data")

}

func closeFile(f *os.File) {
    fmt.Println("closing")
    f.Close()
}
```

运行结果

```
creating
writing
closing
```

使用defer来调用closeFile函数可以保证在main函数结束之前，关闭文件的操作一定会被执行。

Go Exit.md

Go Exit

使用 os.Exit 可以给定一个状态，然后立刻退出程序运行。

```
package main

import "fmt"
import "os"

func main() {
    // 当使用`os.Exit`的时候defer操作不会被运行，
    // 所以这里的``fmt.Println`将不会被调用
    defer fmt.Println("!")

    // 退出程序并设置退出状态值
    os.Exit(3)
}
```

注意，Go和C语言不同，main函数并不返回一个整数来表示程序的退出状态，而是将退出状态作为 os.Exit 函数的参数。

如果你使用 go run 来运行程序，将会有如下输出

```
exit status 3
```

如果你使用 go build 先编译程序，然后再运行可执行文件，程序将不会有输出。
如果你想查看程序的返回值，*nix系列系统下面使用如下方法:

```
$ ./go_exit
$ echo $?
3
```

Go for循环

Go for循环

for循环是Go语言唯一的循环结构。这里有三个基本的for循环类型。

```
package main

import "fmt"

func main() {

    // 最基本的一种，单一条件循环
    // 这个可以代替其他语言的while循环
    i := 1
    for i <= 3 {
        fmt.Println(i)
        i = i + 1
    }

    // 经典的循环条件初始化/条件判断/循环后条件变化
    for j := 7; j <= 9; j++ {
        fmt.Println(j)
    }

    // 无条件的for循环是死循环，除非你使用break跳出循环或者
    // 使用return从函数返回
    for {
        fmt.Println("loop")
        break
    }
}
```

输出结果

```
1
2
3
7
8
9
loop
```

在后面的例子中，你将会看到其他的循环方式，比如使用range函数循环数组，切片和字典，或者用select函数循环channel通道。

Go if..else if..else 条件判断

Go if..else if..else 条件判断

Go语言的条件判断结构也很简单。

```
package main

import "fmt"

func main() {

    // 基本的例子
    if 7%2 == 0 {
        fmt.Println("7 is even")
    } else {
        fmt.Println("7 is odd")
    }

    // 只有if条件的情况
    if 8%4 == 0 {
        fmt.Println("8 is divisible by 4")
    }

    // if条件可以包含一个初始化表达式，这个表达式中的变量
    // 是这个条件判断结构的局部变量
    if num := 9; num < 0 {
        fmt.Println(num, "is negative")
    } else if num < 10 {
        fmt.Println(num, "has 1 digit")
    } else {
        fmt.Println(num, "has multiple digits")
    }
}
```

条件判断结构中，条件两边的小括号()是可以省略的，但是条件执行语句块两边的大括号{}不可以。

输出结果为

```
7 is odd
8 is divisible by 4
9 has 1 digit
```

在Go里面没有三元表达式"?:"，所以你只能使用条件判断语句。

Go JSON支持

Go JSON支持

Go内置了对JSON数据的编码和解码，这些数据的类型包括内置数据类型和自定义数据类型。

```
package main

import "encoding/json"
import "fmt"
import "os"

// 我们使用两个结构体来演示自定义数据类型的JSON数据编码和解码。
type Response1 struct {
    Page   int
    Fruits []string
}
type Response2 struct {
    Page   int      `json:"page"`
    Fruits []string `json:"fruits"`
}

func main() {

    // 首先我们看一下将基础数据类型编码为JSON数据
    bolB, _ := json.Marshal(true)
    fmt.Println(string(bolB))

    intB, _ := json.Marshal(1)
    fmt.Println(string(intB))

    fltB, _ := json.Marshal(2.34)
    fmt.Println(string(fltB))

    strB, _ := json.Marshal("gopher")
    fmt.Println(string(strB))

    // 这里是将切片和字典编码为JSON数组或对象
    slcD := []string{"apple", "peach", "pear"}
    slcB, _ := json.Marshal(slcD)
    fmt.Println(string(slcB))

    mapD := map[string]int{"apple": 5, "lettuce": 7}
    mapB, _ := json.Marshal(mapD)
    fmt.Println(string(mapB))

    // JSON包可以自动地编码自定义数据类型。结果将只包括自定义
    // 类型中的可导出成员的值并且默认情况下，这些成员名称都作
    // 为JSON数据的键
    res1D := &Response1{
        Page:   1,
        Fruits: []string{"apple", "peach", "pear"}}
    res1B, _ := json.Marshal(res1D)
    fmt.Println(string(res1B))

    // 你可以使用tag来自定义编码后JSON键的名称
    res2D := &Response2{
        Page:   1,
        Fruits: []string{"apple", "peach", "pear"}}
    res2B, _ := json.Marshal(res2D)
```

```
    fmt.Println(string(res2B))

    // 现在我们看看解码JSON数据为Go数值
    byt := []byte(`{"num":6.13,"strs":["a","b"]}`)

    // 我们需要提供一个变量来存储解码后的JSON数据，这里
    // 的`map[string]interface{}`将以Key-Value的方式
    // 保存解码后的数据，Value可以为任意数据类型
    var dat map[string]interface{}

    // 解码过程，并检测相关可能存在的错误
    if err := json.Unmarshal(byt, &dat); err != nil {
        panic(err)
    }
    fmt.Println(dat)

    // 为了使用解码后map里面的数据，我们需要将Value转换为
    // 它们合适的类型，例如我们将这里的num转换为期望的float64
    num := dat["num"].(float64)
    fmt.Println(num)

    // 访问嵌套的数据需要一些类型转换
    strs := dat["strs"].([]interface{})
    str1 := strs[0].(string)
    fmt.Println(str1)

    // 我们还可以将JSON解码为自定义数据类型，这有个好处是可以
    // 为我们的程序增加额外的类型安全并且不用再在访问数据的时候
    // 进行类型断言
    str := `{"page": 1, "fruits": ["apple", "peach"]}`
    res := &Response2{}
    json.Unmarshal([]byte(str), &res)
    fmt.Println(res)
    fmt.Println(res.Fruits[0])

    // 上面的例子中，我们使用bytes和strings来进行原始数据和JSON数据
    // 之间的转换，我们也可以直接将JSON编码的数据流写入`os.Writer`
    // 或者是HTTP请求回复数据。
    enc := json.NewEncoder(os.Stdout)
    d := map[string]int{"apple": 5, "lettuce": 7}
    enc.Encode(d)
}
```

运行结果

```
true
1
2.34
"gopher"
["apple","peach","pear"]
{"apple":5,"lettuce":7}
{"Page":1,"Fruits":["apple","peach","pear"]}
{"page":1,"fruits":["apple","peach","pear"]}
map[num:6.13 strs:[a b]]
6.13
a
&{1 [apple peach]}
apple
{"apple":5,"lettuce":7}
```

Go Line Filters

Go Line Filters

- Go Line Filters
- Go Line Filters

Line Filters翻译一下大概是行数据过滤器。简单一点就是一个程序从标准输入stdin读取数据，然后处理一下，将处理的结果输出到标准输出stdout。grep和sed就是常见的行数据过滤器。
这里有一个行数据过滤器的例子，是把一个输入文本转换为大写的文本。你可以使用这种方式来实现你自己的Go Line Filters。

```
package main

import (
    "bufio"
    "fmt"
    "os"
    "strings"
)

func main() {

    // 使用缓冲scanner来包裹无缓冲的`os.Stdin`可以让我们
    // 方便地使用`Scan`方法，这个方法会将scanner定位到下
    // 一行的位置
    scanner := bufio.NewScanner(os.Stdin)

    for scanner.Scan() {
        // `Text`方法从输入中返回当前行
        ucl := strings.ToUpper(scanner.Text())

        // 输出转换为大写的行
        fmt.Println(ucl)
    }

    // 在`Scan`过程中，检查错误。文件结束不会被当作一个错误
    if err := scanner.Err(); err != nil {
        fmt.Fprintln(os.Stderr, "error:", err)
        os.Exit(1)
    }
}
```

运行结果

```
hello world
HELLO WORLD
how are you
HOW ARE YOU
```

Go Panic

Go Panic

Panic表示的意思就是有些意想不到的错误发生了。通常我们用来表示程序正常运行过程中不应该出现的，或者我们没有处理好的错误。

```
package main

import "os"

func main() {

    // 我们使用panic来检查预期不到的错误
    panic("a problem")

    // Panic的通常使用方法就是如果一个函数
    // 返回一个我们不知道怎么处理的错误的
    // 时候，直接终止执行。
    _, err := os.Create("/tmp/file")
    if err != nil {
        panic(err)
    }
}
```

运行结果

```
panic: a problem

goroutine 1 [running]:
runtime.panic(0x44e060, 0xc0840031b0)
        C:/Users/ADMINI~1/AppData/Local/Temp/2/bindist667667715/go/src/pkg/runtime/panic.c:266
+0xc8
main.main()
        D:/GoDoc/go_panic.go:8 +0x58
exit status 2
```

和其他的编程语言不同的是，Go并不使用exception来处理错误，而是通过函数返回值返回错误代码。

Go range函数

Go range函数

range函数是个神奇而有趣的内置函数，你可以使用它来遍历数组，切片和字典。

当用于遍历数组和切片的时候，range函数返回索引和元素；

当用于遍历字典的时候，range函数返回字典的键和值。

```
package main

import "fmt"

func main() {

    // 这里我们使用range来计算一个切片的所有元素和
    // 这种方法对数组也适用
    nums := []int{2, 3, 4}
    sum := 0
    for _, num := range nums {
        sum += num
    }
    fmt.Println("sum:", sum)

    // range 用来遍历数组和切片的时候返回索引和元素值
    // 如果我们不要关心索引可以使用一个下划线(_)来忽略这个返回值
    // 当然我们有的时候也需要这个索引
    for i, num := range nums {
        if num == 3 {
            fmt.Println("index:", i)
        }
    }

    // 使用range来遍历字典的时候，返回键值对。
    kvs := map[string]string{"a": "apple", "b": "banana"}
    for k, v := range kvs {
        fmt.Printf("%s -> %s\n", k, v)
    }

    // range函数用来遍历字符串时，返回Unicode代码点。
    // 第一个返回值是每个字符的起始字节的索引，第二个是字符代码点，
    // 因为Go的字符串是由字节组成的，多个字节组成一个rune类型字符。
    for i, c := range "go" {
        fmt.Println(i, c)
    }
}
```

输出结果为

```
sum: 9
index: 1
a -> apple
b -> banana
0 103
1 111
```

Go SHA1 散列

Go SHA1 散列

SHA1散列经常用来计算二进制或者大文本数据的短标识值。git版本控制系统用SHA1来标识受版本控制的文件和目录。这里介绍Go中如何计算SHA1散列值。
Go在 `crypto/*` 包里面实现了几个常用的散列函数。

```
package main

import "crypto/sha1"
import "fmt"

func main() {
    s := "sha1 this string"

    // 生成一个hash的模式是`sha1.New()`，`sha1.Write(bytes)`
    // 然后是`sha1.Sum([]byte{})`，下面我们开始一个新的hash
    // 示例
    h := sha1.New()

    // 写入要hash的字节，如果你的参数是字符串，使用`[]byte(s)`
    // 把它强制转换为字节数组
    h.Write([]byte(s))

    // 这里计算最终的hash值，Sum的参数是用来追加而外的字节到要
    // 计算的hash字节里面，一般来讲，如果上面已经把需要hash的
    // 字节都写入了，这里就设为nil就可以了
    bs := h.Sum(nil)

    // SHA1散列值经常以16进制的方式输出，例如git commit就是
    // 这样，所以可以使用`%x`来将散列结果格式化为16进制的字符串
    fmt.Println(s)
    fmt.Printf("%x\n", bs)
}
```

运行结果

```
sha1 this string
cf23df2207d99a74fbe169e3eba035e633b65d94
```

Go String与Byte切片之间的转换

Go String与Byte切片之间的转换

String转换到Byte数组时，每个byte(byte类型其实就是uint8)保存字符串对应字节的数值。

注意Go的字符串是UTF-8编码的，每个字符长度是不确定的，一些字符可能是1、2、3或者4个字节结尾。

示例1：

```
package main

import "fmt"

func main() {

    s1 := "abcd"
    b1 := []byte(s1)
    fmt.Println(b1) // [97 98 99 100]

    s2 := "中文"
    b2 := []byte(s2)
    fmt.Println(b2)  // [228 184 173 230 150 135], unicode，每个中文字符会由三个byte组成

    r1 := []rune(s1)
    fmt.Println(r1) // [97 98 99 100], 每个字一个数值

    r2 := []rune(s2)
    fmt.Println(r2) // [20013 25991], 每个字一个数值

}
```

Go Switch语句

Go Switch语句

当条件判断分支太多的时候，我们会使用switch语句来优化逻辑。

```
package main

import "fmt"
import "time"

func main() {

    // 基础的switch用法
    i := 2
    fmt.Print("write ", i, " as ")
    switch i {
    case 1:
        fmt.Println("one")
    case 2:
        fmt.Println("two")
    case 3:
        fmt.Println("three")
    }

    // 你可以使用逗号来在case中分开多个条件。还可以使用default语句，
    // 当上面的case都没有满足的时候执行default所指定的逻辑块。
    switch time.Now().Weekday() {
    case time.Saturday, time.Sunday:
        fmt.Println("it's the weekend")
    default:
        fmt.Println("it's a weekday")
    }

    // 当switch没有跟表达式的时候，功能和if/else相同，这里我们
    // 还可以看到case后面的表达式不一定是常量。
    t := time.Now()
    switch {
    case t.Hour() < 12:
        fmt.Println("it's before noon")
    default:
        fmt.Println("it's after noon")
    }
}
```

输出结果为

```
write 2 as two
it's a weekday
it's before noon
```

Go URL解析

Go URL解析

URL提供了一种统一访问资源的方式。我们来看一下Go里面如何解析URL。

```
package main

import "fmt"
import "net/url"
import "strings"

func main() {

    // 我们将解析这个URL，它包含了模式，验证信息，
    // 主机，端口，路径，查询参数和查询片段
    s := "postgres://user:pass@host.com:5432/path?k=v#f"

    // 解析URL，并保证没有错误
    u, err := url.Parse(s)
    if err != nil {
        panic(err)
    }

    // 可以直接访问解析后的模式
    fmt.Println(u.Scheme)

    // User包含了所有的验证信息，使用
    // Username和Password来获取单独的信息
    fmt.Println(u.User)
    fmt.Println(u.User.Username())
    p, _ := u.User.Password()
    fmt.Println(p)

    // Host包含了主机名和端口，如果需要可以
    // 手动分解主机名和端口
    fmt.Println(u.Host)
    h := strings.Split(u.Host, ":")
    fmt.Println(h[0])
    fmt.Println(h[1])

    // 这里我们解析出路径和`#`后面的片段
    fmt.Println(u.Path)
    fmt.Println(u.Fragment)

    // 为了得到`k=v`格式的查询参数，使用RawQuery。你可以将
    // 查询参数解析到一个map里面。这个map为字符串作为key，
    // 字符串切片作为value。
    fmt.Println(u.RawQuery)
    m, _ := url.ParseQuery(u.RawQuery)
    fmt.Println(m)
    fmt.Println(m["k"][0])
}
```

运行结果

```
postgres
user:pass
user
pass
host.com:5432
host.com
5432
/path
f
k=v
map[k:[v]]
v
```

Go 闭包函数

Go 闭包函数

Go支持匿名函数，匿名函数可以形成闭包。闭包函数可以访问定义闭包的函数定义的内部变量。

示例1：

```
package main

import "fmt"

// 这个"intSeq"函数返回另外一个在intSeq内部定义的匿名函数，
// 这个返回的匿名函数包住了变量i，从而形成了一个闭包
func intSeq() func() int {
    i := 0
    return func() int {
        i += 1
        return i
    }
}

func main() {
    // 我们调用intSeq函数，并且把结果赋值给一个函数nextInt，
    // 这个nextInt函数拥有自己的i变量，这个变量每次调用都被更新。
    // 这里i的初始值是由intSeq调用的时候决定的。
    nextInt := intSeq()

    // 调用几次nextInt，看看闭包的效果
    fmt.Println(nextInt())
    fmt.Println(nextInt())
    fmt.Println(nextInt())

    // 为了确认闭包的状态是独立于intSeq函数的，再创建一个。
    newInts := intSeq()
    fmt.Println(newInts())
}
```

输出结果为

```
1
2
3
1
```

示例2：

```
package main

import "fmt"

func main() {
    add10 := closure(10)//其实是构造了一个加10函数
    fmt.Println(add10(5))
    fmt.Println(add10(6))
    add20 := closure(20)
    fmt.Println(add20(5))
}

func closure(x int) func(y int) int {
    return func(y int) int {
        return x + y
    }

}
```

输出结果为：

```
15
16
25
```

示例3：

```
package main

import "fmt"

func main() {

    var fs []func() int

    for i := 0; i < 3; i++ {

        fs = append(fs, func() int {

            return i
        })
    }
    for _, f := range fs {
        fmt.Printf("%p = %v\n", f, f())
    }
}
```

输出结果：

```
0x401200 = 3
0x401200 = 3
0x401200 = 3
```

示例4：

```
package main

import "fmt"

func adder() func(int) int {
    sum := 0
    return func(x int) int {
        sum += x
        return sum
    }
}

func main() {
    result := adder()
    for i := 0; i < 10; i++ {
        fmt.Println(result(i))
    }
}
```

输出结果为：

```
0
1
3
6
10
15
21
28
36
45
```

Go变量

Go变量

Go是静态类型语言，变量是有明确类型的。编译器会检查函数调用中，变量类型的正确性。

使用 `var` 关键字来定义变量。

Go 的基本类型有：

- bool
- string
- int int8 int16 int32 int64
- uint uint8 uint16 uint32 uint64 uintptr
- byte // uint8 的别名
- rune // int32 的别名 代表一个Unicode码
- float32 float64
- complex64 complex128

看看下面的例子

```
package main

import "fmt"

func main() {
    // `var` 关键字用来定义一个或者多个变量
    var a string = "initial"
    fmt.Println(a)

    // 你一次可以定义多个变量
    var b, c int = 1, 2
    fmt.Println(b, c)

    // Go会推断出具有初始值的变量的类型
    var d = true
    fmt.Println(d)

    //定义变量时，没有给出初始值的变量被默认初始化为零值
    //整型的零值就是0
    var e int
    fmt.Println(e)

    //":=" 语法是同时定义和初始化变量的快捷方式
    f := "short"
    fmt.Println(f)
}
```

输出结果为

```
initial
1 2
true
0
short
```

Go 遍历通道

Go 遍历通道

我们知道range函数可以遍历数组，切片，字典等。这里我们还可以使用range函数来遍历通道以接收通道数据。

```
package main

import "fmt"

func main() {

    // 我们遍历queue通道里面的两个数据
    queue := make(chan string, 2)
    queue <- "one"
    queue <- "two"
    close(queue)

    // range函数遍历每个从通道接收到的数据，因为queue再发送完两个
    // 数据之后就关闭了通道，所以这里我们range函数在接收到两个数据
    // 之后就结束了。如果上面的queue通道不关闭，那么range函数就不
    // 会结束，从而在接收第三个数据的时候就阻塞了。
    for elem := range queue {
        fmt.Println(elem)
    }
}
```

运行结果

```
one
two
```

这个例子同时说明了，即使关闭了一个非空通道，我们仍然可以从通道里面接收到值。

Go 并行功能

Go 并行功能

goroutine是一个轻量级的线程。

```
package main

import "fmt"

func f(from string) {
    for i := 0; i < 3; i++ {
        fmt.Println(from, ":", i)
    }
}

func main() {

    // 假设我们有一个函数叫做f(s)
    // 这里我们使用通常的同步调用来调用函数
    f("direct")

    // 为了能够让这个函数以协程(goroutine)方式
    // 运行使用go f(s)
    // 这个协程将和调用它的协程并行执行
    go f("goroutine")

    // 你也可以为匿名函数开启一个协程运行
    go func(msg string) {
        fmt.Println(msg)
    }("going")

    // 上面的协程在调用之后就异步执行了，所以程序不用等待它们执行完成
    // 就跳到这里来了，下面的Scanln用来从命令行获取一个输入，然后才
    // 让main函数结束
    // 如果没有下面的Scanln语句，程序到这里会直接退出，而上面的协程还
    // 没有来得及执行完，你将无法看到上面两个协程运行的结果
    var input string
    fmt.Scanln(&input)
    fmt.Println("done")
}
```

运行结果

```
direct : 0
direct : 1
direct : 2
goroutine : 0
goroutine : 1
goroutine : 2
going
ok
done
```

Go 并行通道Channel

Go 并行通道Channel

Channel是连接并行协程(goroutine)的通道。你可以向一个通道写入数据然后从另外一个通道读取数据。

```
package main

import "fmt"

func main() {

    // 使用`make(chan 数据类型)`来创建一个Channel
    // Channel的类型就是它们所传递的数据的类型
    messages := make(chan string)

    // 使用`channel <-`语法来向一个Channel写入数据
    // 这里我们从一个新的协程向messages通道写入数据ping
    go func() { messages <- "ping" }()

    // 使用`<-channel`语法来从Channel读取数据
    // 这里我们从main函数所在的协程来读取刚刚写入
    // messages通道的数据
    msg := <-messages
    fmt.Println(msg)
}
```

运行结果

```
ping
```

当我们运行程序的时候，数据ping成功地从一个协程传递到了另外一个协程。
默认情况下，协程之间的通信是同步的，也就是说数据的发送端和接收端必须配对使用。Channel的这种特点使得我们可以不用在程序结尾添加额外的代码也能够获取协程发送端发来的信息。因为程序执行到 `msg:=<-messages` 的时候被阻塞了，直到获得发送端发来的信息才继续执行。

Go常量

Go常量

Go支持定义字符常量，字符串常量，布尔型常量和数值常量。

使用 `const` 关键字来定义常量。

```
package main

import "fmt"
import "math"

// "const" 关键字用来定义常量
const s string = "constant"

func main() {
    fmt.Println(s)

    // "const"关键字可以出现在任何"var"关键字出现的地方
    // 区别是常量必须有初始值
    const n = 500000000

    // 常量表达式可以执行任意精度数学计算
    const d = 3e20 / n
    fmt.Println(d)

    // 数值型常量没有具体类型，除非指定一个类型
    // 比如显式类型转换
    fmt.Println(int64(d))

    // 数值型常量可以在程序的逻辑上下文中获取类型
    // 比如变量赋值或者函数调用。
    // 例如，对于math包中的Sin函数,它需要一个float64类型的变量
    fmt.Println(math.Sin(n))
}
```

输出结果为

```
constant
6e+11
600000000000
-0.28470407323754404
```

Go 超时

Go 超时

超时对那些连接外部资源的程序来说是很重要的，否则就需要限定执行时间。在Go里面实现超时很简单。我们可以使用channel和select很容易地做到。

```
package main

import "time"
import "fmt"

func main() {

    // 在这个例子中，假设我们执行了一个外部调用，2秒之后将结果写入c1
    c1 := make(chan string, 1)
    go func() {
        time.Sleep(time.Second * 2)
        c1 <- "result 1"
    }()

    // 这里使用select来实现超时，`res := <-c1`等待通道结果，
    // `<- Time.After`则在等待1秒后返回一个值，因为select首先
    // 执行那些不再阻塞的case，所以这里会执行超时程序，如果
    // `res := <-c1`超过1秒没有执行的话
    select {
    case res := <-c1:
        fmt.Println(res)
    case <-time.After(time.Second * 1):
        fmt.Println("timeout 1")
    }

    // 如果我们将超时时间设为3秒，这个时候`res := <-c2`将在
    // 超时case之前执行，从而能够输出写入通道c2的值
    c2 := make(chan string, 1)
    go func() {
        time.Sleep(time.Second * 2)
        c2 <- "result 2"
    }()
    select {
    case res := <-c2:
        fmt.Println(res)
    case <-time.After(time.Second * 3):
        fmt.Println("timeout 2")
    }
}
```

运行结果

```
timeout 1
result 2
```

Go 错误处理

Go 错误处理

在Go里面通常采用显式返回错误代码的方式来进行错误处理。这个和Java或者Ruby里面使用异常或者是C里面运行正常返回结果，发生错误返回错误代码的方式不同。Go的这种错误处理的方式使得我们能够很容易看出哪些函数可能返回错误，并且能够像调用那些没有错误返回的函数一样调用。

```
package main

import "errors"
import "fmt"

// Go语言里面约定错误代码是函数的最后一个返回值，
// 并且类型是error，这是一个内置的接口

func f1(arg int) (int, error) {
    if arg == 42 {

        // errors.New使用错误信息作为参数，构建一个基本的错误
        return -1, errors.New("can't work with 42")

    }

    // 返回错误为nil表示没有错误
    return arg + 3, nil
}

// 你可以通过实现error接口的方法Error()来自定义错误
// 下面我们自定义一个错误类型来表示上面例子中的参数错误
type argError struct {
    arg  int
    prob string
}

func (e *argError) Error() string {
    return fmt.Sprintf("%d - %s", e.arg, e.prob)
}

func f2(arg int) (int, error) {
    if arg == 42 {

        // 这里我们使用&argError语法来创建一个新的结构体对象，
        // 并且给它的成员赋值
        return -1, &argError{arg, "can't work with it"}
    }
    return arg + 3, nil
}

func main() {

    // 下面的两个循环例子用来测试我们的带有错误返回值的函数
    // 在for循环语句里面，使用了if来判断函数返回值是否为nil是
```

```
    // Go语言里面的一种约定做法。
    for _, i := range []int{7, 42} {
        if r, e := f1(i); e != nil {
            fmt.Println("f1 failed:", e)
        } else {
            fmt.Println("f1 worked:", r)
        }
    }
    for _, i := range []int{7, 42} {
        if r, e := f2(i); e != nil {
            fmt.Println("f2 failed:", e)
        } else {
            fmt.Println("f2 worked:", r)
        }
    }

    // 如果你需要使用自定义错误类型返回的错误数据，你需要使用类型断言
    // 来获得一个自定义错误类型的实例才行。
    _, e := f2(42)
    if ae, ok := e.(*argError); ok {
        fmt.Println(ae.arg)
        fmt.Println(ae.prob)
    }
}
```

运行结果为

```
f1 worked: 10
f1 failed: can't work with 42
f2 worked: 10
f2 failed: 42 - can't work with it
42
can't work with it
```

Go 打点器

Go 打点器

Timer是让你等待一段时间然后去做一件事情，这件事情只会做一次。而Ticker是让你按照一定的时间间隔循环往复地做一件事情，除非你手动停止它。

```
package main

import "time"
import "fmt"

func main() {

    // Ticker使用和Timer相似的机制，同样是使用一个通道来发送数据。
    // 这里我们使用range函数来遍历通道数据，这些数据每隔500毫秒被
    // 发送一次，这样我们就可以接收到
    ticker := time.NewTicker(time.Millisecond * 500)
    go func() {
        for t := range ticker.C {
            fmt.Println("Tick at", t)
        }
    }()

    // Ticker和Timer一样可以被停止。一旦Ticker停止后，通道将不再
    // 接收数据，这里我们将在1500毫秒之后停止
    time.Sleep(time.Millisecond * 1500)
    ticker.Stop()
    fmt.Println("Ticker stopped")
}
```

输出结果

```
Tick at 2014-02-18 05:42:50.363640783 +0800 CST
Tick at 2014-02-18 05:42:50.863793985 +0800 CST
Tick at 2014-02-18 05:42:51.363532887 +0800 CST
Ticker stopped
```

在这个例子中，我们让Ticker一个独立协程上每隔500毫秒执行一次，然后在main函数所在协程上等待1500毫秒，然后停止Ticker。所以只输出了3次 Ticker at 信息。

Go 递归函数

Go 递归函数

Go语言支持递归函数，这里是一个经典的斐波拉切数列的列子。

```
package main

import "fmt"

// fact函数不断地调用自身，直到达到基本状态fact(0)
func fact(n int) int {
    if n == 0 {
        return 1
    }
    return n * fact(n-1)
}

func main() {
    fmt.Println(fact(7))
}
```

输出结果为

```
5040
```

Go 读取文件

Go 读取文件

读写文件是很多程序的基本任务，下面我们看看Go里面的文件读取。

```
package main

import (
    "bufio"
    "fmt"
    "io"
    "io/ioutil"
    "os"
)

// 读取文件的函数调用大多数都需要检查错误，
// 使用下面这个错误检查方法可以方便一点
func check(e error) {
    if e != nil {
        panic(e)
    }
}

func main() {

    // 最基本的文件读写任务就是把整个文件的内容读取到内存
```

```
    dat, err := ioutil.ReadFile("/tmp/dat")
    check(err)
    fmt.Print(string(dat))

    // 有的时候你想更多地控制到底是读取文件的哪个部分，这个
    // 时候你可以使用`os.Open`打开一个文件获取一个`os.File`
    // 对象
    f, err := os.Open("/tmp/dat")

    // 从这个文件中读取一些字节，并且由于字节数组长度所限，
    // 最多读取5个字节，另外还需要注意实际能够读取的字节
    // 数量
    b1 := make([]byte, 5)
    n1, err := f.Read(b1)
    check(err)
    fmt.Printf("%d bytes: %s\n", n1, string(b1))

    // 你也可以使用`Seek`来跳转到文件中的一个已知位置，并从
    // 那个位置开始读取数据
    o2, err := f.Seek(6, 0)
    check(err)
    b2 := make([]byte, 2)
    n2, err := f.Read(b2)
    check(err)
    fmt.Printf("%d bytes @ %d: %s\n", n2, o2, string(b2))

    // `io`包提供了一些帮助文件读取的函数。例如上面的方法如果
    // 使用方法`ReadAtLeast`函数来实现，将使得程序更健壮
    o3, err := f.Seek(6, 0)
    check(err)
    b3 := make([]byte, 2)
    n3, err := io.ReadAtLeast(f, b3, 2)
    check(err)
    fmt.Printf("%d bytes @ %d: %s\n", n3, o3, string(b3))

    // 没有内置的rewind方法，但是可以使用`Seek(0,0)`来实现
    _, err = f.Seek(0, 0)
    check(err)

    // `bufio`包提供了缓冲读取文件的方法，这将使得文件读取更加
    // 高效
    r4 := bufio.NewReader(f)
    b4, err := r4.Peek(5)
    check(err)
    fmt.Printf("5 bytes: %s\n", string(b4))

    // 最后关闭打开的文件。一般来讲这个方法会在打开文件的时候，
    // 使用defer来延迟关闭
    f.Close()
}
```

在运行程序之前，你需要创建一个 `/tmp/dat` 文件，然后写入一些测试数据。

运行结果

```
hello world
i am jemy
who are you
what do you like
5 bytes: hello
2 bytes @ 6: wo
2 bytes @ 6: wo
5 bytes: hello
```

Go 方法

Go 方法

一般的函数定义叫做函数，定义在结构体上面的函数叫做该结构体的方法。

示例1：

```
package main

import "fmt"

type rect struct {
    width, height int
}

// 这个area方法有一个限定类型*rect，
// 表示这个函数是定义在rect结构体上的方法
func (r *rect) area() int {
    return r.width * r.height
}

// 方法的定义限定类型可以为结构体类型
// 也可以是结构体指针类型
// 区别在于如果限定类型是结构体指针类型
// 那么在该方法内部可以修改结构体成员信息
func (r rect) perim() int {
    return 2*r.width + 2*r.height
}

func main() {
    r := rect{width: 10, height: 5}

    // 调用方法
    fmt.Println("area: ", r.area())
    fmt.Println("perim:", r.perim())

    // Go语言会自动识别方法调用的参数是结构体变量还是
    // 结构体指针，如果你要修改结构体内部成员值，那么使用
    // 结构体指针作为函数限定类型，也就是说参数若是结构体
    //变量，仅仅会发生值拷贝。
    rp := &r
    fmt.Println("area: ", rp.area())
    fmt.Println("perim:", rp.perim())
}
```

输出结果为

```
area:  50
perim: 30
area:  50
perim: 30
```

示例2：

从某种意义上说，方法是函数的“语法糖”。当函数与某个特定的类型绑定，那么它就是一个方法。也证因为如此，我们可以将方法“还原”成函数。

instance.method(args)->(type).func(instance,args)

为了区别这两种方式，官方文档中将左边的称为 Method Value ，右边则是 Method Expression 。Method Value是包装后的状态对象，总是与特定的对象实例关联在一起（类似闭包，拐带私奔），而Method Expression函数将Receiver作为第一个显式参数，调用时需额外传递。

注意：对于Method Expression，T仅拥有T Receiver方法，*T拥有（T+*T）所有方法。

```
package main

import (
    "fmt"
)

func main() {
    p := Person{2, "张三"}

    p.test(1)
    var f1 func(int) = p.test
    f1(2)
    Person.test(p, 3)
    var f2 func(Person, int) = Person.test
    f2(p, 4)

}

type Person struct {
    Id   int
    Name string
}

func (this Person) test(x int) {
    fmt.Println("Id:", this.Id, "Name", this.Name)
    fmt.Println("x=", x)
}
```

输出结果：

```
Id: 2 Name 张三
x= 1
Id: 2 Name 张三
x= 2
Id: 2 Name 张三
x= 3
Id: 2 Name 张三
x= 4
```

示例3：

使用匿名字段，实现模拟继承。即可直接访问匿名字段（匿名类型或匿名指针类型）的方法这种行为类似“继承”。访问匿名字段方法时，有隐藏规则，这样我们可以实现override效果。

```
package main

import (
    "fmt"
)

func main() {
    p := Student{Person{2, "张三"}, 25}
    p.test()

}

type Person struct {
    Id   int
    Name string
}

type Student struct {
    Person
    Score int
}

func (this Person) test() {
    fmt.Println("person test")
}

func (this Student) test() {
    fmt.Println("student test")
}
```

输出结果为：

```
student test
```

Go 工作池

Go 工作池

在这个例子中，我们来看一下如何使用gorouotine和channel来实现工作池。

```
package main

import "fmt"
import "time"

// 我们将在worker函数里面运行几个并行实例，这个函数从jobs通道
// 里面接受任务，然后把运行结果发送到results通道。每个job我们
// 都休眠一会儿，来模拟一个耗时任务。
func worker(id int, jobs <-chan int, results chan<- int) {
    for j := range jobs {
        fmt.Println("worker", id, "processing job", j)
        time.Sleep(time.Second)
        results <- j * 2
    }
}

func main() {

    // 为了使用我们的工作池，我们需要发送工作和接受工作的结果，
    // 这里我们定义两个通道，一个jobs，一个results
    jobs := make(chan int, 100)
    results := make(chan int, 100)

    // 这里启动3个worker协程，一开始的时候worker阻塞执行，因为
    // jobs通道里面还没有工作任务
    for w := 1; w <= 3; w++ {
        go worker(w, jobs, results)
    }

    // 这里我们发送9个任务，然后关闭通道，告知任务发送完成
    for j := 1; j <= 9; j++ {
        jobs <- j
    }
    close(jobs)

    // 然后我们从results里面获得结果
    for a := 1; a <= 9; a++ {
        <-results
    }
```

运行结果

```
worker 1 processing job 1
worker 2 processing job 2
worker 3 processing job 3
worker 1 processing job 4
worker 3 processing job 5
worker 2 processing job 6
worker 1 processing job 7
worker 3 processing job 8
worker 2 processing job 9
```

Go 关闭通道

Go 关闭通道

关闭通道的意思是该通道将不再允许写入数据。这个方法可以让通道数据的接受端知道数据已经全部发送完成了。

```
package main

import "fmt"

// 在这个例子中，我们使用通道jobs在main函数所在的协程和一个数据
// 接收端所在的协程通信。当我们数据发送完成后，我们关闭jobs通道
func main() {
    jobs := make(chan int, 5)
    done := make(chan bool)

    // 这里是数据接收端协程，它重复使用`j, more := <-jobs`来从通道
    // jobs获取数据，这里的more在通道关闭且通道中不再有数据可以接收的
    // 时候为false，我们通过判断more来决定所有的数据是否已经接收完成。
    // 如果所有数据接收完成，那么向done通道写入true
    go func() {
        for {
            j, more := <-jobs
            if more {
                fmt.Println("received job", j)
            } else {
                fmt.Println("received all jobs")
                done <- true
                return
            }
        }
    }()

    // 这里向jobs通道写入三个数据，然后关闭通道
    for j := 1; j <= 3; j++ {
        jobs <- j
        fmt.Println("sent job", j)
    }
    close(jobs)
    fmt.Println("sent all jobs")

    // 我们知道done通道在接收数据的时候会阻塞，所以在所有的数据发送
    // 接收完成后，写入done的数据将在这里被接收，然后程序结束。
    <-done
}
```

运行结果

```
sent job 1
received job 1
sent job 2
sent job 3
sent all jobs
received job 2
received job 3
received all jobs
```

Go 函数定义

Go 函数定义

函数是Go语言的重要内容。

```
package main

import "fmt"

// 这个函数计算两个int型输入数据的和，并返回int型的和
func plus(a int, b int) int {
    // Go需要使用return语句显式地返回值
    return a + b
}

func main() {
    // 函数的调用方式很简单
    // "名称(参数列表)"
    res := plus(1, 2)
    fmt.Println("1+2 =", res)
}
```

输出结果为

```
1+2 = 3
```

Go的函数还有很多其他的特性，其中一个就是多返回值，我们下面会看到。

Go 函数多返回值

Go 函数多返回值

Go语言内置支持多返回值，这个在Go语言中用的很多，比如一个函数同时返回结果和错误信息。

```
package main

import "fmt"

// 这个函数的返回值为两个int
func vals() (int, int) {
    return 3, 7
}

func main() {

    // 获取函数的两个返回值
    a, b := vals()
    fmt.Println(a)
    fmt.Println(b)

    // 如果你只对多个返回值里面的几个感兴趣
    // 可以使用下划线(_)来忽略其他的返回值
    _, c := vals()
    fmt.Println(c)
}
```

输出结果为

```
3
7
7
```

Go 函数回调

Go 函数回调

Go支持函数回调，你可以把函数名称作为参数传递给另外一个函数，然后在别的地方实现这个函数。

```
package main

import "fmt"

type Callback func(x, y int) int

func main() {
    x, y := 1, 2
    fmt.Println(test(x, y, add))
}

//提供一个接口，让外部去实现
func test(x, y int, callback Callback) int {
    return callback(x, y)
}

func add(x, y int) int {
    return x + y
}
```

运行结果

```
3
```

Go 函数命名返回值

Go 函数命名返回值

函数接受参数。在 Go 中，函数可以返回多个“结果参数”，而不仅仅是一个值。它们可以像变量那样命名和使用。

如果命名了返回值参数，一个没有参数的 `return` 语句，会将当前的值作为返回值返回。注意，如果遇到if等代码块和返回值同名，还需要显示写出返回值。

```
package main

import "fmt"

func split(sum int) (x, y int) {
    x = sum * 4 / 9
    y = sum - x
    return
}

func main() {
    fmt.Println(split(17))
}
```

运行结果

```
7 10
```

Go 互斥

Go 互斥

上面的例子中，我们看过了如何在多个协程之间原子地访问计数器，对于更复杂的例子，我们可以使用 `Mutex` 来在多个协程之间安全地访问数据。

```
package main

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

func main() {

    // 这个例子的状态就是一个map
    var state = make(map[int]int)

    // 这个`mutex`将同步对状态的访问
    var mutex = &sync.Mutex{}

    // ops将对状态的操作进行计数
    var ops int64 = 0
    // 这里我们启动100个协程来不断地读取这个状态
```

```
    // 这里我们启动100个协程来不断地读取这个状态
    for r := 0; r < 100; r++ {
        go func() {
            total := 0
            for {

                // 对于每次读取，我们选取一个key来访问，
                // mutex的`Lock`函数用来保证对状态的
                // 唯一性访问，访问结束后，使用`Unlock`
                // 来解锁，然后增加ops计数器
                key := rand.Intn(5)
                mutex.Lock()
                total += state[key]
                mutex.Unlock()
                atomic.AddInt64(&ops, 1)

                // 为了保证这个协程不会让调度器出于饥饿状态，
                // 我们显式地使用`runtime.Gosched`释放了资源
                // 控制权，这种控制权会在通道操作结束或者
                // time.Sleep结束后自动释放。但是这里我们需要
                // 手动地释放资源控制权
                runtime.Gosched()
            }
        }()
    }
    // 同样我们使用10个协程来模拟写状态
    for w := 0; w < 10; w++ {
        go func() {
            for {
                key := rand.Intn(5)
                val := rand.Intn(100)
                mutex.Lock()
                state[key] = val
                mutex.Unlock()
                atomic.AddInt64(&ops, 1)
                runtime.Gosched()
            }
        }()
    }

    // 主协程Sleep，让那10个协程能够运行一段时间
    time.Sleep(time.Second)

    // 输出总操作次数
    opsFinal := atomic.LoadInt64(&ops)
    fmt.Println("ops:", opsFinal)

    // 最后锁定并输出状态
    mutex.Lock()
    fmt.Println("state:", state)
    mutex.Unlock()
}
```

运行结果

```
ops: 3931611
state: map[0:84 2:20 3:18 1:65 4:31]
```

Go 环境变量

Go 环境变量

环境变量是一种很普遍的将配置信息传递给Unix程序的机制。

```
package main

import "os"
import "strings"
import "fmt"
func main() {
    // 为了设置一个key/value对，使用`os.Setenv`
    // 为了获取一个key的value，使用`os.Getenv`
    // 如果所提供的key在环境变量中没有对应的value，
    // 那么返回空字符串
    os.Setenv("FOO", "1")
    fmt.Println("FOO:", os.Getenv("FOO"))
    fmt.Println("BAR:", os.Getenv("BAR"))

    // 使用`os.Environ`来列出环境变量中所有的key/value对
    // 你可以使用`strings.Split`方法来将key和value分开
    // 这里我们打印所有的key
    fmt.Println()
    for _, e := range os.Environ() {
        pair := strings.Split(e, "=")
        fmt.Println(pair[0])
    }
}
```

这里我们设置了FOO环境变量，所以我们取到了它的值，但是没有设置BAR环境变量，所以值为空。另外我们列出了系统的所有环境变量，当然这个输出根据不同的系统设置可能并不相同。

输出结果

```
FOO: 1
BAR:

TERM_PROGRAM
TERM
SHELL
TMPDIR
Apple_PubSub_Socket_Render
OLDPWD
USER
SSH_AUTH_SOCK
__CF_USER_TEXT_ENCODING
__CHECKFIX1436934
PATH
PWD
ITERM_PROFILE
SHLVL
COLORFGBG
HOME
ITERM_SESSION_ID
LOGNAME
LC_CTYPE
GOPATH
_
FOO
```

Go 集合功能

Go 集合功能

我们经常需要程序去处理一些集合数据，比如选出所有符合条件的数据或者使用一个自定义函数将一个集合元素拷贝到另外一个集合。

在一些语言里面，通常是使用泛化数据结构或者算法。但是Go不支持泛化类型，在Go里面如果你的程序或者数据类型需要操作集合，那么通常是为集合提供一些操作函数。

这里演示了一些操作strings切片的集合函数，你可以使用这些例子来构建你自己的函数。注意在有些情况下，使用内联集合操作代码会更清晰，而不是去创建新的帮助函数。

```
package main

import "strings"
import "fmt"

// 返回t在vs中第一次出现的索引，如果没有找到t，返回 - 1
func Index(vs []string, t string) int {
    for i, v := range vs {
```

```
        if v == t {
            return i
        }
    }
    return -1
}

// 如果t存在于vs中，那么返回true，否则false
func Include(vs []string, t string) bool {
    return Index(vs, t) >= 0
}

// 如果使用vs中的任何一个字符串作为函数f的参数可以让f返回true，
// 那么返回true，否则false
func Any(vs []string, f func(string) bool) bool {
    for _, v := range vs {
        if f(v) {
            return true
        }
    }
    return false
}

// 如果分别使用vs中所有的字符串作为f的参数都能让f返回true，
// 那么返回true，否则返回false
func All(vs []string, f func(string) bool) bool {
    for _, v := range vs {
        if !f(v) {
            return false
        }
    }
    return true
}

// 返回一个新的字符串切片，切片的元素为vs中所有能够让函数f
// 返回true的元素
func Filter(vs []string, f func(string) bool) []string {
    vsf := make([]string, 0)
    for _, v := range vs {
        if f(v) {
            vsf = append(vsf, v)
        }
    }
    return vsf
}

// 返回一个bool类型切片，切片的元素为vs中所有字符串作为f函数
// 参数所返回的结果
func Map(vs []string, f func(string) string) []string {
    vsm := make([]string, len(vs))
    for i, v := range vs {
        vsm[i] = f(v)
    }
    return vsm
}
```

```
}

func main() {

    // 来，试试我们的字符串切片操作函数
    var strs = []string{"peach", "apple", "pear", "plum"}

    fmt.Println(Index(strs, "pear"))

    fmt.Println(Include(strs, "grape"))

    fmt.Println(Any(strs, func(v string) bool {
        return strings.HasPrefix(v, "p")
    }))

    fmt.Println(All(strs, func(v string) bool {
        return strings.HasPrefix(v, "p")
    }))

    fmt.Println(Filter(strs, func(v string) bool {
        return strings.Contains(v, "e")
    }))

    // 上面的例子都使用匿名函数，你也可以使用命名函数
    fmt.Println(Map(strs, strings.ToUpper))
}
```

运行结果

```
2
false
true
false
[peach apple pear]
[PEACH APPLE PEAR PLUM]
```

Go 计时器

Go 计时器

我们有的时候希望Go在未来的某个时刻执行或者是以一定的时间间隔重复执行。Go内置的timer和ticker功能使得这些任务变得简单了。我们先看看timer的功能，下一节再看看ticker的功能。

```
package main

import "time"
import "fmt"

func main() {
    // Timer 代表了未来的一个事件，你告诉timer需要等待多久，然后
    // 计时器提供了一个通道，这个通道将在等待的时间结束后得到通知，
    // 这里的timer将等待2秒
    timer1 := time.NewTimer(time.Second * 2)

    // 这里`<-timer1.C`在timer的通道`C`上面阻塞等待，直到有个值发送给该
    // 通道，通知通道计时器已经等待完成。
    // timer.NewTimer方法获取的timer1的结构体定义为
    // type Ticket struct{
    //  C <-chan Time
    //}
    <-timer1.C
    fmt.Println("Timer 1 expired")

    // 如果你仅仅需要等待的话，你可以使用`time.Sleep`，而timer的
    // 独特之处在于你可以在timer等待完成之前取消等待。
    timer2 := time.NewTimer(time.Second)
    go func() {
        <-timer2.C
        fmt.Println("Timer 2 expired")
    }()
    stop2 := timer2.Stop()
    if stop2 {
        fmt.Println("Timer 2 stopped")
    }
}
```

运行结果

```
Timer 1 expired
Timer 2 stopped
```

在上面的例子中，第一个timer将在2秒后等待完成而第二个timer则在等待完成之前被取消了。

Go 接口

Go 接口

接口是一个方法签名的集合。

所谓方法签名，就是指方法的声明，而不包括实现。

```
package main

import "fmt"
import "math"

// 这里定义了一个最基本的表示几何形状的方法的接口
type geometry interface {
    area() float64
    perim() float64
}

// 这里我们要让正方形square和圆形circle实现这个接口
type square struct {
    width, height float64
}
type circle struct {
    radius float64
}

// 在Go中实现一个接口，只要实现该接口定义的所有方法即可
// 下面是正方形实现的接口
func (s square) area() float64 {
    return s.width * s.height
}
func (s square) perim() float64 {
    return 2*s.width + 2*s.height
}

// 圆形实现的接口
func (c circle) area() float64 {
    return math.Pi * c.radius * c.radius
}
func (c circle) perim() float64 {
    return 2 * math.Pi * c.radius
}

// 如果一个函数的参数是接口类型，那么我们可以使用命名接口
// 来调用这个函数
// 比如这里的正方形square和圆形circle都实现了接口geometry，
// 那么它们都可以作为这个参数为geometry类型的函数的参数。
// 在measure函数内部，Go知道调用哪个结构体实现的接口方法。
func measure(g geometry) {
    fmt.Println(g)
    fmt.Println(g.area())
    fmt.Println(g.perim())
}

func main() {
    s := square{width: 3, height: 4}
    c := circle{radius: 5}

    // 这里circle和square都实现了geometry接口，所以
    // circle类型变量和square类型变量都可以作为measure
    // 函数的参数
```

```
    measure(s)
    measure(c)
}
```

输出结果为

```
{3 4}
12
14
{5}
78.53981633974483
31.41592653589793
```

也就是说如果结构体A实现了接口B定义的所有方法，那么A也是B类型。

Go 结构体

Go 结构体

Go语言结构体数据类是将各个类型的变量定义的集合，通常用来表示记录。

```
package main

import "fmt"

// 这个person结构体有name和age成员
type person struct {
    name string
    age  int
}

func main() {

    // 这个语法创建一个新结构体变量
    fmt.Println(person{"Bob", 20})

    // 可以使用"成员:值"的方式来初始化结构体变量
    fmt.Println(person{name: "Alice", age: 30})

    // 未显式赋值的成员初始值为零值
    fmt.Println(person{name: "Fred"})

    // 可以使用&来获取结构体变量的地址
    fmt.Println(&person{name: "Ann", age: 40})

    // 使用点号(.)来访问结构体成员
    s := person{name: "Sean", age: 50}
    fmt.Println(s.name)

    // 结构体指针也可以使用点号(.)来访问结构体成员
    // Go语言会自动识别出来
    sp := &s
    fmt.Println(sp.age)

    // 结构体成员变量的值是可以改变的
    sp.age = 51
    fmt.Println(sp.age)
}
```

输出结果为

```
{Bob 20}
{Alice 30}
{Fred 0}
&{Ann 40}
Sean
50
51
```

Go 进程触发

Go 进程触发

有的时候，我们需要从Go程序里面触发一个其他的非Go进程来执行。

```
package main

import "fmt"
import "io/ioutil"
import "os/exec"

func main() {

    // 我们从一个简单的命令开始，这个命令不需要任何参数
    // 或者输入，仅仅向stdout输出一些信息。`exec.Command`
    // 函数创建了一个代表外部进程的对象
    dateCmd := exec.Command("date")

    // `Output`是另一个运行命令时用来处理信息的函数，这个
    // 函数等待命令结束，然后收集命令输出。如果没有错误发
    // 生的话，`dateOut`将保存date的信息
    dateOut, err := dateCmd.Output()
    if err != nil {
        panic(err)
    }
    fmt.Println("> date")
    fmt.Println(string(dateOut))

    // 下面我们看一个需要从stdin输入数据的命令，我们将
    // 数据输入传给外部进程的stdin，然后从它输出到stdout
    // 的运行结果收集信息
    grepCmd := exec.Command("grep", "hello")

    // 这里我们显式地获取input/output管道，启动进程，
    // 向进程写入数据，然后读取输出结果，最后等待进程结束
    grepIn, _ := grepCmd.StdinPipe()
    grepOut, _ := grepCmd.StdoutPipe()
    grepCmd.Start()
    grepIn.Write([]byte("hello grep\ngoodbye grep"))
    grepIn.Close()
    grepBytes, _ := ioutil.ReadAll(grepOut)
    grepCmd.Wait()

    // 在上面的例子中，我们忽略了错误检测，但是你一样可以
    // 使用`if err!=nil`模式来进行处理。另外我们仅仅收集了
    // `StdoutPipe`的结果，同时你也可以用一样的方法来收集
    // `StderrPipe`的结果
    fmt.Println("> grep hello")
    fmt.Println(string(grepBytes))

    // 注意，我们在触发外部命令的时候，需要显式地提供
    // 命令和参数信息。另外如果你想用一个命令行字符串
    // 触发一个完整的命令，你可以使用bash的-c选项
```

```
    // 触发一个完整的命令，你可以使用bash的-c选项
    lsCmd := exec.Command("bash", "-c", "ls -a -l -h")
    lsOut, err := lsCmd.Output()
    if err != nil {
        panic(err)
    }
    fmt.Println("> ls -a -l -h")
    fmt.Println(string(lsOut))
}
```

所触发的程序的执行结果和我们直接执行这些程序的结果是一样的。
运行结果

```
> date
Wed Oct 10 09:53:11 PDT 2012
> grep hello
hello grep
> ls -a -l -h
drwxr-xr-x  4 mark 136B Oct 3 16:29 .
drwxr-xr-x 91 mark 3.0K Oct 3 12:50 ..
-rw-r--r--  1 mark 1.3K Oct 3 16:28 spawning-processes.go
```

Go 进程执行

Go 进程执行

在上面的例子中，我们演示了一下如何去触发执行一个外部的进程。我们这样做的原因是我们希望从Go进程里面可以访问外部进程的信息。但有的时候，我们仅仅希望执行一个外部进程来替代当前的Go进程。这个时候，我们需要使用Go提供的 `exec` 函数。

```
package main

import "syscall"
import "os"
import "os/exec"

func main() {

    // 本例中，我们使用`ls`来演示。Go需要一个该命令
    // 的完整路径，所以我们使用`exec.LookPath`函数来
    // 找到它
    binary, lookErr := exec.LookPath("ls")
    if lookErr != nil {
        panic(lookErr)
    }
    // `Exec`函数需要一个切片参数，我们给ls命令一些
    // 常见的参数。注意，第一个参数必须是程序名称
    args := []string{"ls", "-a", "-l", "-h"}

    // `Exec`还需要一些环境变量，这里我们提供当前的
    // 系统环境
    env := os.Environ()

    // 这里是`os.Exec`调用。如果一切顺利，我们的原
    // 进程将终止，然后启动一个新的ls进程。如果有
    // 错误发生，我们将获得一个返回值
    execErr := syscall.Exec(binary, args, env)
    if execErr != nil {
        panic(execErr)
    }
}
```

运行结果

```
total 16
drwxr-xr-x  4 mark 136B Oct 3 16:29 .
drwxr-xr-x 91 mark 3.0K Oct 3 12:50 ..
-rw-r--r--  1 mark 1.3K Oct 3 16:28 execing-processes.go
```

注意，Go没有提供Unix下面经典的fork函数。通常这也不是一个问题，因为进程触发和进程执行已经覆盖了fork的大多数功能。

Go hello world

Go 经典hello world

我们的第一个例子是打印经典的“hello world”信息，我们先看下代码。

```
package main

import "fmt"

func main() {
    fmt.Println("hello world")
}
```

输出结果为：

```
$ ls
el_01_hello_world.go
$ go build el_01_hello_world.go
$ ls
el_01_hello_world    el_01_hello_world.go
$ ./el_01_hello_world
hello world
```

为了使一个 go文件 能够 编译 为 可执行文件 ，包名必须是 main ，然后我们导入提供格式化输出的 fmt 包，该程序的执行入口是 func main() 函数，在函数里面，我们使用 fmt 包提供的 Println 函数来输出"hello world"字符串。

为了运行这个程序，我们可以使用 go run el_01_hello_world.go 来运行这个例子，这样是直接输出运行结果而不会产生任何中间文件。但是有的时候我们希望能够将程序编译为二进制文件保存起来，我们可以像上面一样使用 go build el_01_hello_world.go 来将源代码编译为二进制可执行文件。然后我们可以直接运行这个二进制可执行文件。

好了，第一个例子就这样结束了。很简单。

Go 可变长参数列表

Go 可变长参数列表

支持可变长参数列表的函数可以支持任意个传入参数，比如fmt.Println函数就是一个支持可变长参数列表的函数。

```
package main

import "fmt"

// 这个函数可以传入任意数量的整型参数
func sum(nums ...int) {
    fmt.Print(nums, " ")
    total := 0
    for _, num := range nums {
        total += num
    }
    fmt.Println(total)
}

func main() {

    // 支持可变长参数的函数调用方法和普通函数一样
    // 也支持只有一个参数的情况
    sum(1, 2)
    sum(1, 2, 3)

    // 如果你需要传入的参数在一个切片中，像下面一样
    // "func(slice...)"把切片打散传入
    nums := []int{1, 2, 3, 4}
    sum(nums...)
}
```

输出结果为

```
[1 2] 3
[1 2 3] 6
[1 2 3 4] 10
```

需要注意的是，可变长参数应该是函数定义的最右边的参数，即最后一个参数。

Go 命令行参数

Go 命令行参数

命令行参数是一种指定程序运行初始参数的常用方式。比如 `go run hello.go` 使用 `run` 和 `hello.go` 参数来执行程序。

```
package main

import "os"
import "fmt"

func main() {

    // `os.Args`提供了对命令行参数的访问，注意该
    // 切片的第一个元素是该程序的运行路径，而
    // `os.Args[1:]`则包含了该程序的所有参数
    argsWithProg := os.Args
    argsWithoutProg := os.Args[1:]

    // 你可以使用索引的方式来获取单个参数
    arg := os.Args[3]

    fmt.Println(argsWithProg)
    fmt.Println(argsWithoutProg)
    fmt.Println(arg)
}
```

在运行该程序的时候，需要首先用 go build 将代码编译为可执行文件，然后提供足够数量的参数。例如

```
$ go build command-line-arguments.go
$ ./command-line-arguments a b c d
[./command-line-arguments a b c d]
[a b c d]
c
```

Go 命令行参数标记

Go 命令行参数标记

命令行参数标记是为命令行程序指定选项参数的常用方法。例如，在命令 `wc -l` 中， `-l` 就是一个命令行参数标记。

Go提供了 `flag` 包来支持基本的命令行标记解析。我们这里将要使用这个包提供的方法来实现带选项的命令行程序。

```
package main

import "flag"
import "fmt"

func main() {

    // 基础的标记声明适用于string，integer和bool型选项。
    // 这里我们定义了一个标记`word`，默认值为`foo`和一
    // 个简短的描述。`flag.String`函数返回一个字符串指
    // 针（而不是一个字符串值），我们下面将演示如何使
    // 用这个指针
    wordPtr := flag.String("word", "foo", "a string")

    // 这里定义了两个标记，一个`numb`，另一个是`fork`，
    // 使用和上面定义`word`标记相似的方法
    numbPtr := flag.Int("numb", 42, "an int")
    boolPtr := flag.Bool("fork", false, "a bool")

    // 你也可以程序中任意地方定义的变量来定义选项，只
    // 需要把该变量的地址传递给flag声明函数即可
    var svar string
    flag.StringVar(&svar, "svar", "bar", "a string var")

    // 当所有的flag声明完成后，使用`flag.Parse()`来分
    // 解命令行选项
    flag.Parse()

    // 这里我们仅仅输出解析后的选项和任何紧跟着的位置
    // 参数，注意我们需要使用`*wordPtr`的方式来获取最
    // 后的选项值
    fmt.Println("word:", *wordPtr)
    fmt.Println("numb:", *numbPtr)
    fmt.Println("fork:", *boolPtr)
    fmt.Println("svar:", svar)
    fmt.Println("tail:", flag.Args())
}
```

为了运行示例，你需要先将程序编译为可执行文件。

```
go build command-line-flags.go
```

下面分别看看给予该命令行程序不同选项参数的例子：

(1) 给所有的选项设置一个参数

```
$ ./command-line-flags -word=opt -numb=7 -fork -svar=flag
word: opt
numb: 7
fork: true
svar: flag
tail: []
```

(2) 如果你不设置flag，那么它们自动采用默认的值

```
$ ./command-line-flags -word=opt
word: opt
numb: 42
fork: false
svar: bar
tail: []
```

(3) 尾部的位置参数可以出现在任意一个flag后面

```
$ ./command-line-flags -word=opt a1 a2 a3
word: opt
numb: 42
fork: false
svar: bar
tail: [a1 a2 a3]
```

(4) 注意flag包要求所有的flag都必须出现在尾部位置参数的前面，否则这些flag将被当作位置参数处理

```
$ ./command-line-flags -word=opt a1 a2 a3 -numb=7
word: opt
numb: 42
fork: false
svar: bar
trailing: [a1 a2 a3 -numb=7]
```

(5) 使用 `-h` 或者 `--help` 这两个flag来自动地生成命令行程序的帮助信息

```
$ ./command-line-flags -h
Usage of ./command-line-flags:
  -fork=false: a bool
  -numb=42: an int
  -svar="bar": a string var
  -word="foo": a string
```

(6) 如果你提供了一个程序不支持的flag，那么程序会打印一个错误信息和帮助信息

```
$ ./command-line-flags -wat
flag provided but not defined: -wat
Usage of ./go_cmd_flag:
  -fork=false: a bool
  -numb=42: an int
  -svar="bar": a string var
  -word="foo": a string
```

Go 排序

Go 排序

Go的sort包实现了内置数据类型和用户自定义数据类型的排序功能。我们先看看内置数据类型的排序。

```
package main

import "fmt"
import "sort"

func main() {

    // 这些排序方法都是针对内置数据类型的。
    // 这里的排序方法都是就地排序，也就是说排序改变了
    // 切片内容，而不是返回一个新的切片
    strs := []string{"c", "a", "b"}
    sort.Strings(strs)
    fmt.Println("Strings:", strs)

    // 对于整型的排序
    ints := []int{7, 2, 4}
    sort.Ints(ints)
    fmt.Println("Ints:   ", ints)

    // 我们还可以检测切片是否已经排序好
    s := sort.IntsAreSorted(ints)
    fmt.Println("Sorted: ", s)
}
```

输出结果

```
Strings: [a b c]
Ints:    [2 4 7]
Sorted:  true
```

Go 切片

Go 切片

切片是Go语言的关键类型之一，它提供了比数组更多的功能。

示例1：

```
package main

import "fmt"

func main() {

    // 和数组不同的是，切片的长度是可变的。
    // 我们可以使用内置函数make来创建一个长度不为零的切片
    // 这里我们创建了一个长度为3，存储字符串的切片，切片元素
    // 默认为零值，对于字符串就是""。
    s := make([]string, 3)
    fmt.Println("emp:", s)

    // 可以使用和数组一样的方法来设置元素值或获取元素值
    s[0] = "a"
    s[1] = "b"
    s[2] = "c"
    fmt.Println("set:", s)
    fmt.Println("get:", s[2])

    // 可以用内置函数len获取切片的长度
    fmt.Println("len:", len(s))

    // 切片还拥有一些数组所没有的功能。
    // 例如我们可以使用内置函数append给切片追加值，然后
    // 返回一个拥有新切片元素的切片。
    // 注意append函数不会改变原切片，而是生成了一个新切片，
    // 我们需要用原来的切片来接收这个新切片
    s = append(s, "d")
    s = append(s, "e", "f")
    fmt.Println("apd:", s)

    // 另外我们还可以从一个切片拷贝元素到另一个切片
    // 下面的例子就是创建了一个和切片s长度相同的新切片
    // 然后使用内置的copy函数来拷贝s的元素到c中。
    c := make([]string, len(s))
    copy(c, s)
    fmt.Println("cpy:", c)

    // 切片还支持一个取切片的操作 "slice[low:high]"
    // 获取的新切片包含元素"slice[low]"，但是不包含"slice[high]"
    // 下面的例子就是取一个新切片，元素包括"s[2]"，"s[3]"，"s[4]"。
```

```
    l := s[2:5]
    fmt.Println("sl1:", l)

    // 如果省略low，默认从0开始，不包括"slice[high]"元素
    l = s[:5]
    fmt.Println("sl2:", l)

    // 如果省略high，默认为len(slice)，包括"slice[low]"元素
    l = s[2:]
    fmt.Println("sl3:", l)

    // 我们可以同时声明和初始化一个切片
    t := []string{"g", "h", "i"}
    fmt.Println("dcl:", t)

    // 我们也可以创建多维切片，和数组不同的是，切片元素的长度也是可变的。
    twoD := make([][]int, 3)
    for i := 0; i < 3; i++ {
        innerLen := i + 1
        twoD[i] = make([]int, innerLen)
        for j := 0; j < innerLen; j++ {
            twoD[i][j] = i + j
        }
    }
    fmt.Println("2d: ", twoD)
}
```

输出结果为

```
emp: [  ]
set: [a b c]
get: c
len: 3
apd: [a b c d e f]
cpy: [a b c d e f]
sl1: [c d e]
sl2: [a b c d e]
sl3: [c d e f]
dcl: [g h i]
2d:  [[0] [1 2] [2 3 4]]
```

数组和切片的定义方式的区别在于 `[]` 之中是否有 `固定长度` 或者推断长度标志符 `...` 。

示例2：

```
package main

import "fmt"

func main() {
    s1 := make([]int, 0)
    test(s1)
    fmt.Println(s1)
}

func test(s []int) {
    s = append(s, 3)
    //因为原来分配的空间不够，所以在另外一个地址又重新分配了空间，所以原始地址的数据没有变
}
```

输出结果为：

```
[]
```

若改为：

```
package main

import "fmt"

func main() {
    s1 := make([]int, 0)
    s1 = test(s1)
    fmt.Println(s1)
}

func test(s []int) []int {
    s = append(s, 3)
    return s
}
```

输出结果为：

```
[3]//正确结果
```

示例3：

cap是slice的最大容量，append函数添加元素，如果超过原始slice的容量，会重新分配底层数组。

```
package main

import "fmt"

func main() {
    s1 := make([]int, 3, 6)
    fmt.Println("s1= ", s1, len(s1), cap(s1))
    s2 := append(s1, 1, 2, 3)
    fmt.Println("s1= ", s1, len(s1), cap(s1))
    fmt.Println("s2= ", s2, len(s2), cap(s2))
    s3 := append(s2, 4, 5, 6)
    fmt.Println("s1= ", s1, len(s1), cap(s1))
    fmt.Println("s2= ", s2, len(s2), cap(s2))
    fmt.Println("s3= ", s3, len(s3), cap(s3))

}
```

输出结果为：

```
s1=  [0 0 0] 3 6
s1=  [0 0 0] 3 6
s2=  [0 0 0 1 2 3] 6 6
s1=  [0 0 0] 3 6
s2=  [0 0 0 1 2 3] 6 6
s3=  [0 0 0 1 2 3 4 5 6] 9 12
```

示例4：

指向同一底层数组的slice之间copy时，允许存在重叠。copy数组时，受限于src和dst数组的长度最小值。

```
package main

import "fmt"

func main() {
    s1 := []int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
    s2 := make([]int, 3, 20)
    var n int
    n = copy(s2, s1)
    fmt.Println(n, s2, len(s2), cap(s2))

    s3 := s1[4:6]
    fmt.Println(n, s3, len(s3), cap(s3))

    n = copy(s3, s1[1:5])
    fmt.Println(n, s3, len(s3), cap(s3))
}
```

输出结果：

```
3 [0 1 2] 3 20
3 [4 5] 2 6
2 [1 2] 2 6
```

Go 请求处理频率控制

Go 请求处理频率控制

频率控制是控制资源利用和保证服务高质量的重要机制。Go可以使用goroutine，channel和ticker来以优雅的方式支持频率控制。

```
package main

import "time"
import "fmt"

func main() {

    // 首先我们看下基本的频率限制。假设我们得控制请求频率，
    // 我们使用一个通道来处理所有的这些请求，这里向requests
    // 发送5个数据，然后关闭requests通道
    requests := make(chan int, 5)
    for i := 1; i <= 5; i++ {
        requests <- i
    }
    close(requests)

    // 这个limiter的Ticker每隔200毫秒结束通道阻塞
    // 这个limiter就是我们频率控制处理器
    limiter := time.Tick(time.Millisecond * 200)

    // 通过阻塞从limiter通道接受数据，我们将请求处理控制在每隔200毫秒
    // 处理一个请求，注意`<-limiter`的阻塞作用。
    for req := range requests {
        <-limiter
        fmt.Println("request", req, time.Now())
    }

    // 我们可以保持正常的请求频率限制，但也允许请求短时间内爆发
    // 我们可以通过通道缓存来实现，比如下面的这个burstyLimiter
    // 就允许同时处理3个事件。
    burstyLimiter := make(chan time.Time, 3)

    // 填充burstyLimiter，先发送3个数据
    for i := 0; i < 3; i++ {
        burstyLimiter <- time.Now()
    }
```

```
    // 然后每隔200毫秒再向burstyLimiter发送一个数据，这里是不断地
    // 每隔200毫秒向burstyLimiter发送数据
    go func() {
        for t := range time.Tick(time.Millisecond * 200) {
            burstyLimiter <- t
        }
    }()

    // 这里模拟5个请求，burstyRequests的前面3个请求会连续被处理，
    // 因为burstyLimiter被先连续发送3个数据的的缘故，而后面两个
    // 则每隔200毫秒处理一次
    burstyRequests := make(chan int, 5)
    for i := 1; i <= 5; i++ {
        burstyRequests <- i
    }
    close(burstyRequests)
    for req := range burstyRequests {
        <-burstyLimiter
        fmt.Println("request", req, time.Now())
    }
}
```

运行结果

```
request 1 2014-02-21 14:20:05.2696437 +0800 CST
request 2 2014-02-21 14:20:05.4696637 +0800 CST
request 3 2014-02-21 14:20:05.6696837 +0800 CST
request 4 2014-02-21 14:20:05.8697037 +0800 CST
request 5 2014-02-21 14:20:06.0697237 +0800 CST
request 1 2014-02-21 14:20:06.0697237 +0800 CST
request 2 2014-02-21 14:20:06.0697237 +0800 CST
request 3 2014-02-21 14:20:06.0707238 +0800 CST
request 4 2014-02-21 14:20:06.2707438 +0800 CST
request 5 2014-02-21 14:20:06.4707638 +0800 CST
```

我们从输出的结果上可以看出最后的5个输出结果中，前三个的时间是连续的，而后两个的时间是隔了200毫秒。

Go 时间

Go 时间

Go提供了对时间和一段时间的支持。这里有一些例子。

```
package main

import "fmt"
import "time"

func main() {
    p := fmt.Println

    // 从获取当前时间开始
    now := time.Now()
    p(now)

    // 你可以提供年，月，日等来创建一个时间。当然时间
    // 总是会和地区联系在一起，也就是时区
    then := time.Date(2009, 11, 17, 20, 34, 58, 651387237, time.UTC)
    p(then)

    // 你可以获取时间的各个组成部分
    p(then.Year())
    p(then.Month())
    p(then.Day())
    p(then.Hour())
    p(then.Minute())
    p(then.Second())
    p(then.Nanosecond())
    p(then.Location())

    // 输出当天是周几，Monday-Sunday中的一个
    p(then.Weekday())

    // 下面的几个方法判断两个时间的顺序，精确到秒
    p(then.Before(now))
    p(then.After(now))
    p(then.Equal(now))

    // Sub方法返回两个时间的间隔(Duration)
    diff := now.Sub(then)
    p(diff)

    // 可以以不同的单位来计算间隔的大小
    p(diff.Hours())
    p(diff.Minutes())
    p(diff.Seconds())
    p(diff.Nanoseconds())

    // 你可以使用Add方法来为时间增加一个间隔
    // 使用负号表示时间向前推移一个时间间隔
    p(then.Add(diff))
    p(then.Add(-diff))
}
```

运行结果

```
2014-03-02 22:54:40.561698065 +0800 CST
2009-11-17 20:34:58.651387237 +0000 UTC
2009
November
17
20
34
58
651387237
UTC
Tuesday
true
false
false
37578h19m41.910310828s
37578.328308419674
2.2546996985051804e+06
1.3528198191031083e+08
135281981910310828
2014-03-02 14:54:40.561698065 +0000 UTC
2005-08-05 02:15:16.741076409 +0000 UTC
```

Go 时间戳

Go 时间戳

程序的一个通常需求是计算从Unix起始时间开始到某个时刻的秒数，毫秒数，微秒数等。
我们来看看Go里面是怎么做的。

```
package main

import "fmt"
import "time"

func main() {

    // 使用Unix和UnixNano来分别获取从Unix起始时间
    // 到现在所经过的秒数和微秒数
    now := time.Now()
    secs := now.Unix()
    nanos := now.UnixNano()
    fmt.Println(now)

    // 注意这里没有UnixMillis方法，所以我们需要将
    // 微秒手动除以一个数值来获取毫秒
    millis := nanos / 1000000
    fmt.Println(secs)
    fmt.Println(millis)
    fmt.Println(nanos)

    // 反过来，你也可以将一个整数秒数或者微秒数转换
    // 为对应的时间
    fmt.Println(time.Unix(secs, 0))
    fmt.Println(time.Unix(0, nanos))
}
```

运行结果

```
2014-03-02 23:11:31.118666918 +0800 CST
1393773091
1393773091118
1393773091118666918
2014-03-02 23:11:31 +0800 CST
2014-03-02 23:11:31.118666918 +0800 CST
```

Go 时间格式化和解析

Go 时间格式化和解析

Go使用模式匹配的方式来支持日期格式化和解析。

```
package main

import "fmt"
import "time"

func main() {
    p := fmt.Println

    // 这里有一个根据RFC3339来格式化日期的例子
    t := time.Now()
    p(t.Format("2006-01-02T15:04:05Z07:00"))

    // Format 函数使用一种基于示例的模式匹配方式，
    // 它使用已经格式化的时间模式来决定所给定参数
    // 的输出格式
    p(t.Format("3:04PM"))
    p(t.Format("Mon Jan _2 15:04:05 2006"))
    p(t.Format("2006-01-02T15:04:05.999999-07:00"))

    // 对于纯数字表示的时间来讲，你也可以使用标准
    // 的格式化字符串的方式来格式化时间
    fmt.Printf("%d-%02d-%02dT%02d:%02d:%02d-00:00\n",
        t.Year(), t.Month(), t.Day(),
        t.Hour(), t.Minute(), t.Second())

    // 时间解析也是采用一样的基于示例的方式
    withNanos := "2006-01-02T15:04:05.999999999-07:00"
    t1, e := time.Parse(
        withNanos,
        "2012-11-01T22:08:41.117442+00:00")
    p(t1)
    kitchen := "3:04PM"
    t2, e := time.Parse(kitchen, "8:41PM")
    p(t2)

    // Parse将返回一个错误，如果所输入的时间格式不对的话
    ansic := "Mon Jan _2 15:04:05 2006"
    _, e = time.Parse(ansic, "8:41PM")
    p(e)

    // 你可以使用一些预定义的格式来格式化或解析时间
    p(t.Format(time.Kitchen))
}
```

运行结果

```
2014-03-03T22:39:31+08:00
10:39PM
Mon Mar  3 22:39:31 2014
2014-03-03T22:39:31.647077+08:00
2014-03-03T22:39:31-00:00
2012-11-01 22:08:41.117442 +0000 +0000
0000-01-01 20:41:00 +0000 UTC
parsing time "8:41PM" as "Mon Jan _2 15:04:05 2006": cannot parse "8:41PM" as "Mon"
10:39PM
```

Go 数值

Go数值

Go有很多种数据类型，包括字符串类型，整型，浮点型，布尔型等等，这里有几个基础的例子。

```
package main

import "fmt"

func main() {

    // 字符串可以使用"+"连接
    fmt.Println("go" + "lang")

    //整型和浮点型
    fmt.Println("1+1 =", 1+1)
    fmt.Println("7.0/3.0 =", 7.0/3.0)

    // 布尔型的几种操作符
    fmt.Println(true && false)
    fmt.Println(true || false)
    fmt.Println(!true)
}
```

输出结果为

```
golang
1+1 = 2
7.0/3.0 = 2.3333333333333335
false
true
false
```

Go 数字解析

Go 数字解析

从字符串解析出数字是一个基本的而且很常见的任务。

Go内置的 strconv 提供了数字解析功能。

```
package main

import "strconv"
import "fmt"

func main() {
    // 使用ParseFloat解析浮点数，64是说明使用多少位
    // 精度来解析
    f, _ := strconv.ParseFloat("1.234", 64)
    fmt.Println(f)

    // 对于ParseInt函数，0 表示从字符串推断整型进制，
    // 则表示返回结果的位数
    i, _ := strconv.ParseInt("123", 0, 64)
    fmt.Println(i)

    // ParseInt能够解析出16进制的数字
    d, _ := strconv.ParseInt("0x1c8", 0, 64)
    fmt.Println(d)

    // 还可以使用ParseUint函数
    u, _ := strconv.ParseUint("789", 0, 64)
    fmt.Println(u)

    // Atoi是解析10进制整型的快捷方法
    k, _ := strconv.Atoi("135")
    fmt.Println(k)

    // 解析函数在遇到无法解析的输入时，会返回错误
    _, e := strconv.Atoi("wat")
    fmt.Println(e)
}
```

运行结果

```
1.234
123
456
789
135
strconv.ParseInt: parsing "wat": invalid syntax
```

Go 数组

Go 数组

- 数组是一个具有 相同数据类型 的元素组成的 固定长度 的 有序集合 。
- 在Go语言中，数组是值类型，长度是类型的组成部分，也就是说"`[10]int`"和“`[20]int`”是完全不同的两种数组类型。
- 同类型的两个数组支持"=="和"!="比较，但是不能比较大小。
- 数组作为参数时，函数内部不改变数组内部的值，除非是传入数组的指针。
- 数组的指针：*[3]int
- 指针数组：[2]*int

示例1：

```
package main

import "fmt"

func main() {

    // 这里我们创建了一个具有5个元素的整型数组
    // 元素的数据类型和数组长度都是数组的一部分
    // 默认情况下，数组元素都是零值
    // 对于整数，零值就是0
    var a [5]int
    fmt.Println("emp:", a)

    // 我们可以使用索引来设置数组元素的值，就像这样
    // "array[index] = value"  或者使用索引来获取元素值，
    // 就像这样"array[index]"
    a[4] = 100
    fmt.Println("set:", a)
    fmt.Println("get:", a[4])

    // 内置的len函数返回数组的长度
    fmt.Println("len:", len(a))

    // 这种方法可以同时定义和初始化一个数组
    b := [5]int{1, 2, 3, 4, 5}
    fmt.Println("dcl:", b)

    // 数组都是一维的，但是你可以把数组的元素定义为一个数组
    // 来获取多维数组结构
    var twoD [2][3]int
    for i := 0; i < 2; i++ {
        for j := 0; j < 3; j++ {
            twoD[i][j] = i + j
        }
    }
    fmt.Println("2d: ", twoD)
}
```

输出结果为

```
emp: [0 0 0 0 0]
set: [0 0 0 0 100]
get: 100
len: 5
dcl: [1 2 3 4 5]
2d:  [[0 1 2] [1 2 3]]
```

`拥有固定长度` 是数组的一个特点，但是这个特点有时候会带来很多不便，尤其在一个集合元素个数不固定的情况下。这个时候我们更多地使用 `切片` 。

示例2：

可以用new创建数组，并返回数组的指针

```
package main

import "fmt"

func main() {
    var a = new([5]int)
    test(a)
    fmt.Println(a, len(a))
}

func test(a *[5]int) {
    a[1] = 5
}
```

输出结果：

```
&[0 5 0 0 0] 5
```

示例3：

```
package main

import "fmt"

func main() {
    a := [...]User{
        {0, "User0"},
        {8, "User8"},
    }
    b := [...]*User{
        {0, "User0"},
        {8, "User8"},
    }
    fmt.Println(a, len(a))
    fmt.Println(b, len(b))

}

type User struct {
    Id   int
    Name string
}
```

输出结果：

```
[{0 User0} {8 User8}] 2
[0x1f216130 0x1f216140] 2
```

Go 随机数

Go 随机数

Go的 math/rand 包提供了伪随机数的生成。

```
package main

import "fmt"
import "math/rand"

func main() {

    // 例如`rand.Intn`返回一个整型随机数n，0<=n<100
    fmt.Print(rand.Intn(100), ",")
    fmt.Print(rand.Intn(100))
    fmt.Println()

    // `rand.Float64` 返回一个`float64` `f`,
    // `0.0 <= f < 1.0`
    fmt.Println(rand.Float64())

    // 这个方法可以用来生成其他数值范围内的随机数，
    // 例如`5.0 <= f < 10.0`
    fmt.Print((rand.Float64()*5)+5, ",")
    fmt.Print((rand.Float64() * 5) + 5)
    fmt.Println()

    // 为了使随机数生成器具有确定性，可以给它一个seed
    s1 := rand.NewSource(42)
    r1 := rand.New(s1)

    fmt.Print(r1.Intn(100), ",")
    fmt.Print(r1.Intn(100))
    fmt.Println()

    // 如果源使用一个和上面相同的seed，将生成一样的随机数
    s2 := rand.NewSource(42)
    r2 := rand.New(s2)
    fmt.Print(r2.Intn(100), ",")
    fmt.Print(r2.Intn(100))
    fmt.Println()
}
```

运行结果

```
81,87
0.6645600532184904
7.1885709359349015,7.123187485356329
5,87
5,87
```

Go 通道的同步功能

Go通道的同步功能

我们使用通道来同步协程之间的执行。

下面的例子是通过获取同步通道数据来阻塞程序执行的方法来等待另一个协程运行结束的。

也就是说main函数所在的协程在运行到 `<-done` 语句的时候将一直等待worker函数所在的协程执行完成，向通道写入数据才会（从通道获得数据）继续执行。

```
package main

import "fmt"
import "time"

// 这个worker函数将以协程的方式运行
// 通道`done`被用来通知另外一个协程这个worker函数已经执行完成
func worker(done chan bool) {
    fmt.Print("working...")
    time.Sleep(time.Second)
    fmt.Println("done")

    // 向通道发送一个数据，表示worker函数已经执行完成
    done <- true
}

func main() {

    // 使用协程来调用worker函数，同时将通道`done`传递给协程
    // 以使得协程可以通知别的协程自己已经执行完成
    done := make(chan bool, 1)
    go worker(done)

    // 一直阻塞，直到从worker所在协程获得一个worker执行完成的数据
    <-done
}
```

运行结果

```
working...done
```

如果我们从main函数里面移除`<-done`语句，那么main函数在worker协程开始运行之前就结束了。

Go 通道方向

Go通道方向

当使用通道作为函数的参数时，你可以指定该通道是只读的还是只写的。这种设置有时候会提高程序的参数类型安全。

```
package main

import "fmt"

// 这个ping函数只接收能够发送数据的通道作为参数，试图从这个通道接收数据
// 会导致编译错误，这里只写的定义方式为`chan<- string`表示这个类型为
// 字符串的通道为只写通道
func ping(pings chan<- string, msg string) {
    pings <- msg
}

// pong函数接收两个通道参数，一个是只读的pings，使用`<-chan string`定义
// 另外一个是只写的pongs，使用`chan<- string`来定义
func pong(pings <-chan string, pongs chan<- string) {
    msg := <-pings
    pongs <- msg
}

func main() {
    pings := make(chan string, 1)
    pongs := make(chan string, 1)
    ping(pings, "passed message")
    pong(pings, pongs)
    fmt.Println(<-pongs)
}
```

运行结果

```
passed message
```

其实这个例子就是把信息首先写入pings通道里面，然后在pong函数里面再把信息从pings通道里面读出来再写入pongs通道里面，最后在main函数里面将信息从pongs通道里面读出来。
在这里，pings和pongs事实上是可读且可写的，不过作为参数传递的时候，函数参数限定了通道的方向。

不过pings和pongs在ping和pong函数里面还是可读且可写的。只是ping和pong函数调用的时候把它们当作了只读或者只写。

Go 通道缓冲

Go通道缓冲

默认情况下，通道是不带缓冲区的。
发送端发送数据，同时必须又接收端相应的接收数据。
而带缓冲区的通道则允许发送端的数据发送和接收端的数据获取处于异步状态，就是说发送端发送的数据可以放在缓冲区里面，可以等待接收端去获取数据，而不是立刻需要接收端去获取数据。
不过由于缓冲区的大小是有限的，所以还是必须有接收端来接收数据的，否则缓冲区一满，数据发送端就无法再发送数据了。

```
package main

import "fmt"

func main() {

    // 这里我们定义了一个可以存储字符串类型的带缓冲通道
    // 缓冲区大小为2
    messages := make(chan string, 2)

    // 因为messages是带缓冲的通道，我们可以同时发送两个数据
    // 而不用立刻需要去同步读取数据
    messages <- "buffered"
    messages <- "channel"

    // 然后我们和上面例子一样获取这两个数据
    fmt.Println(<-messages)
    fmt.Println(<-messages)
}
```

运行结果

```
buffered
channel
```

Go 通道选择Select

Go 通道选择Select

Go的select关键字可以让你同时等待多个通道操作，将协程（goroutine），通道（channel）和select结合起来构成了Go的一个强大特性。

```
package main

import "time"
import "fmt"

func main() {

    // 本例中，我们从两个通道中选择
    c1 := make(chan string)
    c2 := make(chan string)

    // 为了模拟并行协程的阻塞操作，我们让每个通道在一段时间后再写入一个值
    go func() {
        time.Sleep(time.Second * 1)
        c1 <- "one"
    }()
    go func() {
        time.Sleep(time.Second * 2)
        c2 <- "two"
    }()

    // 我们使用select来等待这两个通道的值，然后输出
    for i := 0; i < 2; i++ {
        select {
        case msg1 := <-c1:
            fmt.Println("received", msg1)
        case msg2 := <-c2:
            fmt.Println("received", msg2)
        }
    }
}
```

输出结果

```
received one
received two
```

如我们所期望的，程序输出了正确的值。对于select语句而言，它不断地检测通道是否有值过来，一旦发现有值过来，立刻获取输出。

Go 写入文件

Go 写入文件

Go将数据写入文件的方法和上面介绍过的读取文件的方法很类似。

```
package main

import (
    "bufio"
    "fmt"
    "io/ioutil"
    "os"
)

func check(e error) {
    if e != nil {
        panic(e)
    }
}

func main() {

    // 首先看一下如何将一个字符串写入文件
    d1 := []byte("hello\ngo\n")
    err := ioutil.WriteFile("/tmp/dat1", d1, 0644)
    check(err)

    // 为了实现细颗粒度的写入，打开文件后再写入
    f, err := os.Create("/tmp/dat2")
    check(err)

    // 在打开文件后通常应该立刻使用defer来调用
    // 打开文件的Close方法，以保证main函数结束
    // 后，文件关闭
    defer f.Close()

    // 你可以写入字节切片
    d2 := []byte{115, 111, 109, 101, 10}
    n2, err := f.Write(d2)
    check(err)
    fmt.Printf("wrote %d bytes\n", n2)

    // 也可以使用`WriteString`直接写入字符串
    n3, err := f.WriteString("writes\n")
    fmt.Printf("wrote %d bytes\n", n3)

    // 调用Sync方法来将缓冲区数据写入磁盘
    f.Sync()

    // `bufio`除了提供上面的缓冲读取数据外，还
    // 提供了缓冲写入数据的方法
    w := bufio.NewWriter(f)
    n4, err := w.WriteString("buffered\n")
```

```
    fmt.Printf("wrote %d bytes\n", n4)

    // 使用Flush方法确保所有缓冲区的数据写入底层writer
    w.Flush()
}
```

运行结果

```
wrote 5 bytes
wrote 7 bytes
wrote 9 bytes
```

Go 信号处理

Go 信号处理

有的时候我们希望Go能够智能地处理Unix信号。例如我们希望一个server接收到一个SIGTERM的信号时，能够自动地停止；或者一个命令行工具接收到一个SIGINT信号时，能够停止接收输入。现在我们来看下如何使用channel来处理信号。

```
package main

import "fmt"
import "os"
import "os/signal"
import "syscall"

func main() {

    // Go信号通知通过向一个channel发送``os.Signal`来实现。
    // 我们将创建一个channel来接受这些通知，同时我们还用
    // 一个channel来在程序可以退出的时候通知我们
    sigs := make(chan os.Signal, 1)
    done := make(chan bool, 1)

    // `signal.Notify`在给定的channel上面注册该channel
    // 可以接受的信号
    signal.Notify(sigs, syscall.SIGINT, syscall.SIGTERM)

    // 这个goroutine阻塞等待信号的到来，当信号到来的时候，
    // 输出该信号，然后通知程序可以结束了
    go func() {
        sig := <-sigs
        fmt.Println()
        fmt.Println(sig)
        done <- true
    }()

    // 程序将等待接受信号，然后退出
    fmt.Println("awaiting signal")
    <-done
    fmt.Println("exiting")
}
```

当运行程序的时候，程序将阻塞等待信号的到来，我们可以使用 `CTRL+C` 来发送一个 `SIGINT` 信号，这样程序就会输出interrupt后退出。

```
awaiting signal

interrupt
exiting
```

Go 原子计数器

Go 原子计数器

Go里面的管理协程状态的主要机制就是通道通讯。这些我们上面的例子介绍过。这里还有一些管理状态的

机制，下面我们看看多协程原子访问计数器的例子，这个功能是由sync/atomic包提供的函数来实现的。

```
package main

import "fmt"
import "time"
import "sync/atomic"
import "runtime"

func main() {

    // 我们使用一个无符号整型来代表一个永远为正整数的counter
    var ops uint64 = 0

    // 为了模拟并行更新，我们使用50个协程来每隔1毫秒来
    // 增加一下counter值，注意这里的50协程里面的for循环，
    // 也就是说如果主协程不退出，这些协程将永远运行下去
    // 所以这个程序每次输出的值有可能不一样
    for i := 0; i < 50; i++ {
        go func() {
            for {
                // 为了能够保证counter值增加的原子性，我们使用
                // atomic包中的AddUint64方法，将counter的地址和
                // 需要增加的值传递给函数即可
                atomic.AddUint64(&ops, 1)

                // 允许其他的协程来处理
                runtime.Gosched()
            }
        }()
    }

    //等待1秒中，让协程有时间运行一段时间
    time.Sleep(time.Second)

    // 为了能够在counter仍被其他协程更新值的同时安全访问counter值，
    // 我们获取一个当前counter值的拷贝，这里就是opsFinal，需要把
    // ops的地址传递给函数`LoadUint64`
    opsFinal := atomic.LoadUint64(&ops)
    fmt.Println("ops:", opsFinal)
}
```

我们多运行几次，结果如下:

```
ops: 7499289
ops: 7700843
ops: 7342417
```

Go 正则表达式

Go 正则表达式

Go内置了对正则表达式的支持，这里是一般的正则表达式常规用法的例子。

```
package main

import "bytes"
import "fmt"
import "regexp"

func main() {

    // 测试模式是否匹配字符串，括号里面的意思是
    // 至少有一个a - z之间的字符存在
    match, _ := regexp.MatchString("p([a-z]+)ch", "peach")
    fmt.Println(match)

    // 上面我们直接使用了字符串匹配的正则表达式，
    // 但是对于其他的正则匹配任务，你需要使用
    // `Compile`来使用一个优化过的正则对象
    r, _ := regexp.Compile("p([a-z]+)ch")

    // 正则结构体对象有很多方法可以使用，比如上面的例子
    // 也可以像下面这么写
    fmt.Println(r.MatchString("peach"))

    // 这个方法检测字符串参数是否存在正则所约束的匹配
    fmt.Println(r.FindString("peach punch"))

    // 这个方法查找第一次匹配的索引，并返回匹配字符串
    // 的起始索引和结束索引，而不是匹配的字符串
    fmt.Println(r.FindStringIndex("peach punch"))

    // 这个方法返回全局匹配的字符串和局部匹配的字符，比如
    // 这里会返回匹配`p([a-z]+)ch`的字符串
    // 和匹配`([a-z]+)`的字符串
    fmt.Println(r.FindStringSubmatch("peach punch"))

    // 和上面的方法一样，不同的是返回全局匹配和局部匹配的
    // 起始索引和结束索引
    fmt.Println(r.FindStringSubmatchIndex("peach punch"))

    // 这个方法返回所有正则匹配的字符，不仅仅是第一个
    fmt.Println(r.FindAllString("peach punch pinch", -1))

    // 这个方法返回所有全局匹配和局部匹配的字符串起始索引
    // 和结束索引
    fmt.Println(r.FindAllStringSubmatchIndex("peach punch pinch", -1))

    // 为这个方法提供一个正整数参数来限制匹配数量
    fmt.Println(r.FindAllString("peach punch pinch", 2))
```

```
    //上面我们都是用了诸如`MatchString`这样的方法，其实
    // 我们也可以使用`[]byte`作为参数，并且使用`Match`
    // 这样的方法名
    fmt.Println(r.Match([]byte("peach")))

    // 当使用正则表达式来创建常量的时候，你可以使用`MustCompile`
    // 因为`Compile`返回两个值
    r = regexp.MustCompile("p([a-z]+)ch")
    fmt.Println(r)

    // regexp包也可以用来将字符串的一部分替换为其他的值
    fmt.Println(r.ReplaceAllString("a peach", "<fruit>"))

    // `Func`变量可以让你将所有匹配的字符串都经过该函数处理
    // 转变为所需要的值
    in := []byte("a peach")
    out := r.ReplaceAllFunc(in, bytes.ToUpper)
    fmt.Println(string(out))
}
```

运行结果

```
true
true
peach
[0 5]
[peach ea]
[0 5 1 3]
[peach punch pinch]
[[0 5 1 3] [6 11 7 9] [12 17 13 15]]
[peach punch]
true
p([a-z]+)ch
a <fruit>
a PEACH
```

Go 指针

Go 指针

Go支持指针，可以用来给函数传递变量的引用。

```
package main

import "fmt"

// 我们用两个不同的例子来演示指针的用法
// zeroval函数有一个int类型参数，这个时候传递给函数的是变量的值
func zeroval(ival int) {
    ival = 0
}

// zeroptr函数的参数是int类型指针，这个时候传递给函数的是变量的地址
// 在函数内部对这个地址所指向的变量的任何修改都会反映到原来的变量上。
func zeroptr(iptr *int) {
    *iptr = 0
}

func main() {
    i := 1
    fmt.Println("initial:", i)

    zeroval(i)
    fmt.Println("zeroval:", i)

    // &操作符用来取得i变量的地址
    zeroptr(&i)
    fmt.Println("zeroptr:", i)

    // 指针类型也可以输出
    fmt.Println("pointer:", &i)
}
```

输出结果为

```
initial: 1
zeroval: 1
zeroptr: 0
pointer: 0xc084000038
```