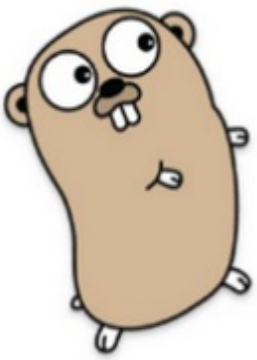
Go 零基础编程入门教程



Go 零基础编程入门教程，带你零基础学习Go语言。

目　录

致谢

当前文档 《Go 零基础编程入门教程》 由 进击的皇虫 使用 书栈网(BookStack.CN) 进行构建，生成于 2020-02-13。

书栈网仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈网难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到书栈网，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到书栈网获取最新的文档，以跟上知识更新换代的步伐。

内容来源：宋佳洋,薛锦 https://github.com/songjiayang/go-basic-courses

文档地址：http://www.bookstack.cn/books/go-basic-courses

书栈官网：https://www.bookstack.cn

书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

Go 零基础编程入门教程

制作、分享高质量 Go 入门教程，包括：

- 视频
  - B 站主页：https://space.bilibili.com/276273794/#/
- GitBook 书籍：https://www.gitbook.com/book/songjiayang/go-basic-courses/details
- GitHub 源码：https://github.com/binatify/importgo
- 欢迎加入 QQ 群：694650181 ，加入请备注：“Go 零基础入门教程”

作者

- 宋佳洋
  - 微博
  - github
  - 个人公众号

- 薛锦
  - 微博
  - github
  - 个人公众号

Go 安装和配置

写在前面

- 教学 Go 版本 1.9.x
- 教学使用 `GOPATH` 为 `~/importgo`

下载并安装

- 下载安装对应版本 https://golang.org/dl/
- 查看 go 安装目录 `/usr/local/go` (Windows 下默认安装到 `c:\Go`)
- 运行 `go version` 命令检查是否安装正确

> 推荐大家使用二进制默认安装方式

项目环境变量

本课程命名为 `importgo`，故添加一个 `IMPORTGOROOT` 的环境变量进行所有的代码开发和演示，具体配置如下：

```
$ vi ~/.profile

export IMPORTGOROOT=$HOME/importgo
export GOPATH=$IMPORTGOROOT # 覆盖 GOPATH 环境变为 importgo
export PATH=$IMPORTGOROOT/bin:$PATH #
```

当我们配置完毕后，可以执行 `source ~/.profile` 更新系统环境变量。

编写我的第一个 Go 程序

首先需要在 GOPATH 文件夹下创建一个 `src` 目录用于存放我们的源代码。

```
$ mkdir -p $GOPATH/src
```

然后我们在 src 目录下面新建 `hello/hello.go` 的文件，内容如下：

```
package main

```

```
3.  import "fmt"
4.
5.  func main() {
6.      fmt.Println("hello, world")
7.  }
```

我们使用 `go run hello.go` 来运行程序，输出为：

```
1.  hello, world
```

Go 命令详解

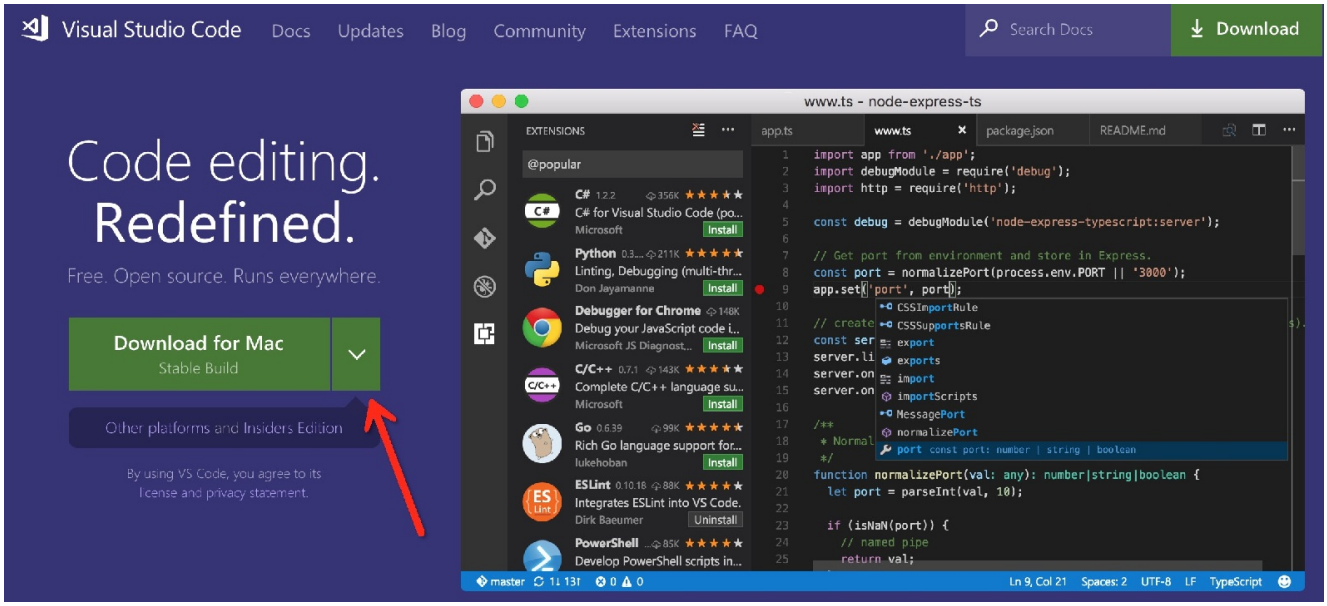
- go run: 运行 Go 源码程序
- go build: 编译 Go 源码
- go install: Go 源码编译并打包到 $GOPATH/bin 目录

为什么选择 VSCode?

- 微软官方出品，更新迭代快
- 强大的插件：代码跳转，自动格式化，错误检测等

下载安装

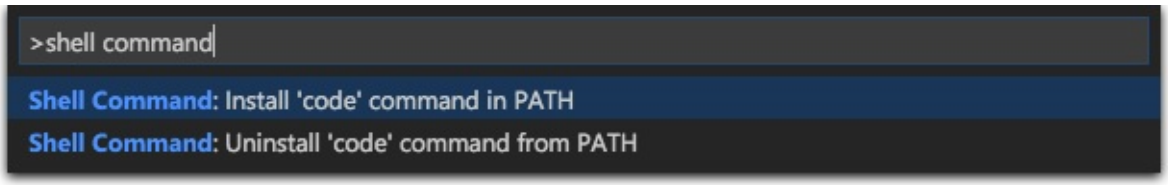
- 登录 vscode 官网：https://code.visualstudio.com
- 根据操作系统选择对应包下载



将 `code` 命令添加到系统 PATH 中

> 效果：在终端输入 `code <filename/foldername>` 就能用 vscode 打开文件或文件夹

- 以 mac 为例：在 vscode 中使用快捷键 `command + shif + p`，
- 输入 shell command，选择 `Shell Command:Install 'code' command in PATH`，如下图：

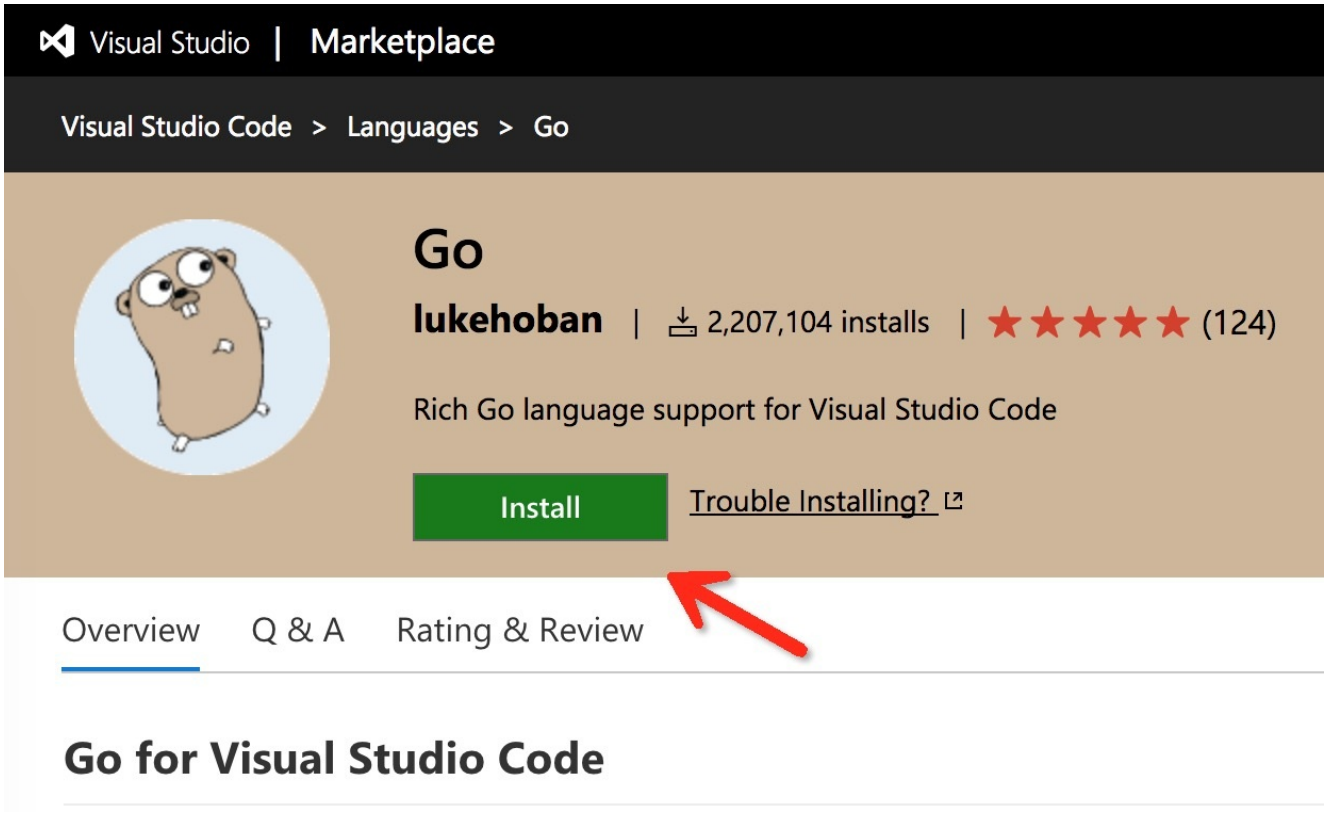


安装 Go 插件

- 登录 vscode 官网 `Extensions` 模

块：https://marketplace.visualstudio.com/VSCode

- 搜索 `go` 插件
- 推荐安装 `lukehoban` 的 go 插件：https://marketplace.visualstudio.com/items?itemName=lukehoban.Go
- 点击 `install` 后就打开 vscode 界面，进行安装



Go 开发相关配置

```
1.  "files.autoSave": "onFocusChange",
2.  "editor.formatOnSave": true,
    "go.gopath":"${workspaceRoot}:/Users/jinxue/golib", // 当前工作空间
3.  ${wordspaceRoot}加上系统 GOPATH 目录
4.  "go.goroot": "/usr/local/Cellar/go/1.9/libexec", // go 的安装目录
5.  "go.formatOnSave": true, //在保存代码时自动格式化代码
    "go.formatTool": "goimports", //使用 goimports 工具进行代码格式化，或者使用
6.  goreturns 和 gofmt
7.  "go.buildOnSave": true, //在保存代码时自动编译代码
8.  "go.lintOnSave": true, //在保存代码时自动检查代码可以优化的地方，并给出建议
9.  "go.vetOnSave": false, //在保存代码时自动检查潜在的错误
10. "go.coverOnSave": false, //在保存代码时执行测试，并显示测试覆盖率
11. "go.useCodeSnippetsOnFunctionSuggest": true, //使用代码片段作为提示
12. "go.gocodeAutoBuild": false //代码自动编译构建
```

基础知识

命名规范

Go 语言中，任何标识符（变量，常量，函数，自定义类型等）都应该满足以下规律：

- 连续的 字符 (unicode*letter* | `` .) 或 数字(“0” … “9”) 组成。
- 以字符或下划线开头。
- 不能和 Go 关键字冲突。

Go 关键字：

```
1.  break        default      func         interface    select
2.  case         defer        go           map          struct
3.  chan         else         goto         package      switch
4.  const        fallthrough  if           range        type
5.  continue     for          import       return       var
```

举例说明：

```
1.  foo  #合法
2.  foo1 #合法
3.  _foo #合法
4.  变量 #合法
5.  变量1 #合法
6.  _变量 合法
7.
8.  1foo #不合法
9.  1 #不合法
10. type #不合法
11. go #不合法
```

变量声明

类型声明基本语法

在 Go 语言中，采用的是后置类型的声明方式，形如：

```
<命名> <类型>
```

例如：

```
x int // 定义 x 为整数类型
```

这么定义不是为了凸显与众不同，而是为了让声明更加清晰易懂，具体可以参考文章gos-declaration-syntax

变量声明

在 Go 语言中通常我们使用关键字 `var` 来声明变量，例如

```
var x int // 表示声明一个名为 x 的整数变量
var b int = 1 // 表示声明一个名为 b 的整数变量，并且附上初始值为 1
var b = 1
```

如果有多个变量同时声明，我们可以采用 `var` 加括号批量声明的方式：

```
var (
    a, b int  // 同时声明 a, b 的整数
    c float64
)
```

简短声明方式

变量在声明的时候如果有初始值，我们可以使用 `:=` 的简短声明方式：

```
a := 1 // 声明 a 为 1 的整数
b := int64(1)  // 声明 b 为 1 的 64 位整数
```

常量定义

常量是指值不能改变的定义，它必须满足如下规则：

- 定义的时候，必须指定值
- 指定的值类型主要有三类： 布尔，数字，字符串， 其中数字类型包含（rune, integer, floating-point, complex), 它们都属于基本数据类型。
- 不能使用 `:=`

例子：

```
1.  const a = 64 // 定义常量值为 64 的值
2.
3.  const (
4.    b = 4
5.    c = 0.1
6.  )
```

基本数据类型

Go 语言中的基本数据类型包含：

```
bool      the set of boolean (true, false)

uint8      the set of all unsigned  8-bit integers (0 to 255)
uint16      the set of all unsigned 16-bit integers (0 to 65535)
uint32      the set of all unsigned 32-bit integers (0 to 4294967295)
uint64      the set of all unsigned 64-bit integers (0 to 18446744073709551615)

int8      the set of all signed  8-bit integers (-128 to 127)
int16      the set of all signed 16-bit integers (-32768 to 32767)
int32      the set of all signed 32-bit integers (-2147483648 to 2147483647)
int64      the set of all signed 64-bit integers (-9223372036854775808 to
9223372036854775807)

float32      the set of all IEEE-754 32-bit floating-point numbers
float64      the set of all IEEE-754 64-bit floating-point numbers

complex64      the set of all complex numbers with float32 real and imaginary
parts
complex128      the set of all complex numbers with float64 real and imaginary
parts

byte      alias for uint8
rune      alias for int32
uint      either 32 or 64 bits
int      same size as uint
uintptr      an unsigned integer large enough to store the uninterpreted bits
of a pointer value

string      the set of string value (eg: "hi")
```

我们可以将基本类型分为三大类：

- 布尔类型
- 数字类型
- 字符串类型

布尔类型

一般我们用于判断条件，它的取值范围为 `true`，`false`，声明如下：

```
var a bool
var a = true
a := true

const a = true
```

数字类型：

数字类型主要分为有符号数和无符号数，有符号数可以用来表示负数，除此之外它们还有位数的区别，不同的位数代表它们实际存储占用空间，以及取值的范围。

例如：

```
var (
  a uint8 = 1
  b int8 = 1
)

var (
    c int = 64
)
```

注意：

- 每种数字类型都有取值范围，超过了取值范围，出现 `overflows` 的错误。
- int，uint 的长度由操作系统的位数决定，在 32 位系统里面，它们的长度未 32 bit，64 位系统，长度为 64 bit。

字符串

```
var a = "hello" //单行字符串
var c = "\"" // 转义符

var b = `hello` //原样输出
var d = `line3  //多行输出
line1
line2
`

var str = "hello, 世界"
```

```
11.
12.  b := str[0]  //b is a uint8 type, like byte
13.  fmt.Println(b)  // 104
14.  fmt.Println(string(b)) // h
15.
16.  fmt.Println(len(str)) //12, 查看字符串有多少个字节
17.  fmt.Println(len([]rune(str))) // 8 查看有多少个字符
```

特殊类型

- byte，uint8 别名，用于表示二进制数据的 bytes
- rune，int32 别名， 用于表示一个符号

```
1.  var str = "hello, 世界"
2.
3.  for _, char := range str {
4.    fmt.Printf("%T", char)
5.  }
```

高级数据类型

- 数组
- 切片
- Map
- 自定义类型
- 结构体
- 函数
- 方法
- 接口

数组

- 定义：由若干相同类型的元素组成的序列
- 数组的长度是固定的，声明后无法改变
- 数组的长度是数组类型的一部分，eg：元素类型相同但是长度不同的两个数组是不同类型的
- 需要严格控制程序所使用内存时，数组十分有用，因为其长度固定，避免了内存二次分配操作

示例

```
1.  package main
2.
3.  import "fmt"
4.
5.  func main() {
6.      // 定义长度为 5 的数组
7.      var arr1 [5]int
8.      for i := 0; i < 5; i++ {
9.          arr1[i] = i
10.     }
11.     printHelper("arr1", arr1)
12.
13.     // 以下赋值会报类型不匹配错误，因为数组长度是数组类型的一部分
14.     // arr1 = [3]int{1, 2, 3}
15.     arr1 = [5]int{2, 3, 4, 5, 6} // 长度和元素类型都相同，可以正确赋值
16.
17.     // 简写模式，在定义的同时给出了赋值
18.     arr2 := [5]int{0, 1, 2, 3, 4}
19.     printHelper("arr2", arr2)
20.
21.     // 数组元素类型相同并且数组长度相等的情况下，数组可以进行比较
22.     fmt.Println(arr1 == arr2)
23.
24.     // 也可以不显式定义数组长度，由编译器完成长度计算
25.     var arr3 = [...]int{0, 1, 2, 3, 4}
26.     printHelper("arr3", arr3)
27.
28.     // 定义前四个元素为默认值 0，最后一个元素为 -1
29.     var arr4 = [...]int{4: -1}
30.     printHelper("arr4", arr4)
31.
32.     // 多维数组
```

```
33.     var twoD [2][3]int
34.     for i := 0; i < 2; i++ {
35.         for j := 0; j < 3; j++ {
36.             twoD[i][j] = i + j
37.         }
38.     }
39.     fmt.Println("twoD: ", twoD)
40. }
41. 
42. func printHelper(name string, arr [5]int) {
43.     for i := 0; i < 5; i++ {
44.         fmt.Printf("%v[%v]: %v\n", name, i, arr[i])
45.     }
46. 
47.     // len 获取长度
48.     fmt.Printf("len of %v: %v\n", name, len(arr))
49. 
50.     // cap 也可以用来获取数组长度
51.     fmt.Printf("cap of %v: %v\n", name, cap(arr))
52. 
53.     fmt.Println()
54. }
```

切片

切片组成要素：

- 指针：指向底层数组
- 长度：切片中元素的长度，不能大于容量
- 容量：指针所指向的底层数组的总容量

常见初始化方式

- 使用 `make` 初始化

```
slice := make([]int, 5)     // 初始化长度和容量都为 5 的切片
slice := make([]int, 5, 10) // 初始化长度为 5, 容量为 10 的切片
```

- 使用简短定义

```
slice := []int{1, 2, 3, 4, 5}
```

- 使用数组来初始化切片

```
arr := [5]int{1, 2, 3, 4, 5}
slice := arr[0:3] // 左闭右开区间，最终切片为 [1,2,3]
```

- 使用切片来初始化切片

```
sliceA := []int{1, 2, 3, 4, 5}
sliceB := sliceA[0:3] // 左闭右开区间，sliceB 最终为 [1,2,3]
```

长度和容量

```
package main

import (
    "fmt"
)

func main() {
    slice := []int{1, 2, 3, 4, 5}
    fmt.Println("len: ", len(slice))
```

```
    fmt.Println("cap: ", cap(slice))

    //改变切片长度
    slice = append(slice, 6)
    fmt.Println("after append operation: ")
    fmt.Println("len: ", len(slice))
    fmt.Println("cap: ", cap(slice)) //注意，底层数组容量不够时，会重新分配数组空间，通常为两倍
}
```

以上代码，预期输出如下：

```
len:  5
cap:  5
after append operation:
len:  6
cap:  12
```

注意点

- 多个切片共享一个底层数组的情况

> 对底层数组的修改，将影响上层多个切片的值

```
package main

import (
    "fmt"
)

func main() {
    slice := []int{1, 2, 3, 4, 5}
    newSlice := slice[0:3]
    fmt.Println("before modifying underlying array:")
    fmt.Println("slice: ", slice)
    fmt.Println("newSlice: ", newSlice)
    fmt.Println()

    newSlice[0] = 6
    fmt.Println("after modifying underlying array:")
    fmt.Println("slice: ", slice)
    fmt.Println("newSlice: ", newSlice)
```

```
}
```

以上代码预期输出如下：

```
before modify underlying array:
slice:  [1 2 3 4 5]
newSlice:  [1 2 3]

after modify underlying array:
slice:  [6 2 3 4 5]
newSlice:  [6 2 3]
```

- 使用 `copy` 方法可以避免共享同一个底层数组

> 示例代码如下：

```go
package main

import (
    "fmt"
)

func main() {
    slice := []int{1, 2, 3, 4, 5}
    newSlice := make([]int, len(slice))
    copy(newSlice, slice)
    fmt.Println("before modifying underlying array:")
    fmt.Println("slice: ", slice)
    fmt.Println("newSlice: ", newSlice)
    fmt.Println()

    newSlice[0] = 6
    fmt.Println("after modifying underlying array:")
    fmt.Println("slice: ", slice)
    fmt.Println("newSlice: ", newSlice)
}
```

以上代码预期输出如下：

```
before modifying underlying array:
slice:  [1 2 3 4 5]
newSlice:  [1 2 3 4 5]
```

```
4.
5.  after modifying underlying array:
6.  slice:  [1 2 3 4 5]
7.  newSlice:  [6 2 3 4 5]
```

小练习

如何使用 `copy` 函数进行切片部分拷贝？

```
1.  // 假设切片 slice 如下:
2.  slice := []int{1, 2, 3, 4, 5}
3.
4.  // 如何使用 copy 创建切片 newSlice, 该切片值为 [2, 3, 4]
5.  newSlice = copy(?,?)
```

Map

在 Go 语言里面，map 一种无序的键值对，它是数据结构 hash 表的一种实现方式，类似 Python 中的字典。

语法

使用关键字 map 来声明形如：

```
map[KeyType]ValueType
```

注意点：

- 必须指定 key，value 的类型，插入的纪录类型必须匹配。
- key 具有唯一性，插入纪录的 key 不能重复。
- KeyType 可以为基础数据类型（例如 bool，数字类型，字符串），不能为数组，切片，map，它的取值必须是能够使用 `==` 进行比较。
- ValueType 可以为任意类型。
- 无序性。
- 线程不安全，一个 goroutine 在对 map 进行写的时候，另外的 goroutine 不能进行读和写操作，Go 1.6 版本以后会抛出 runtime 错误信息。

声明和初始化

- 使用 var 声明

```
var cMap map[string]int  // 只定义，此时 cMap 为 nil
fmt.Println(cMap == nil)
cMap["北京"] = 1  // 报错，因为 cMap 为 nil
```

- 使用 `make`

```
cMap := make(map[string]int)
cMap["北京"] = 1

// 指定初始容量
cMap = make(map[string]int, 100)
cMap["北京"] = 1
```

说明：在使用 make 初始化 map 的时候，可以指定初始容量，这在能预估 map key 数量的情况

下，减少动态分配的次数，从而提升性能。

- 简短声明方式

```
cMap := map[string]int{"北京": 1}
```

map 基本操作

```
cMap := map[string]int{}

cMap["北京"] = 1 //写

code := cMap["北京"] // 读
fmt.Println(code)

code = cMap["广州"]  // 读不存在 key
fmt.Println(code)

code, ok = cMap["广州"]  // 检查 key 是否存在
if ok {
  fmt.Println(code)
} else {
  fmt.Println("key not exist")
}

delete(cMap, "北京") // 删除 key
fmt.Println("北京")
```

循环和无序性

```
cMap := map[string]int{"北京": 1, "上海": 2, "广州": 3, "深圳": 4}

for city, code := range cMap {
  fmt.Printf("%s:%d", city, code)
  fmt.Println()
}
```

线程不安全

```
cMap := make(map[string]int)
```

```
2.
3.  var wg sync.WaitGroup
4.  wg.Add(2)
5.
6.  go func() {
7.      cMap["北京"] = 1
8.      wg.Done()
9.  }()
10.
11. go func() {
12.     cMap["上海"] = 2
13.     wg.Done()
14. }()
15.
16. wg.Wait()
```

在 Go 1.6 之后的版本，多次运行此段代码，你将遇到这样的错误信息：

```
1.  fatal error: concurrent map writes
2.
3.  goroutine x [running]:
4.  runtime.throw(0x10c64b6, 0x15)
5.  .....
```

解决之道：

- 对读写操作加锁
- 使用 security map，例如 `sync.map`

map 嵌套

```
1.  provinces := make(map[string]map[string]int)
2.
3.  provinces["北京"] = map[string]int{
4.    "东城区": 1,
5.    "西城区": 2,
6.    "朝阳区": 3,
7.    "海淀区": 4,
8.  }
9.
10. fmt.Println(provinces["北京"])
```

自定义类型

前面我们已经学习了不少基础和高级数据类型，在 Go 语言里面，我们还可以通过自定义类型来表示一些特殊的数据结构和业务逻辑。

使用关键字 `type` 来声明:

```
type NAME TYPE
```

声明语法

- 单次声明

```
type City string
```

- 批量声明

```
type (
    B0 = int8
    B1 = int16
    B2 = int32
    B3 = int64
)

type (
    A0 int8
    A1 int16
    A2 int32
    A3 int64
)
```

简单示例

```
package main

import "fmt"

type City string

```

```
func main() {
    city := City("上海")
    fmt.Println(city)
}
```

基本操作

```
package main

import "fmt"

type City string
type Age int

func main() {
    city := City("北京")
    fmt.Println("I live in", city + " 上海")  //  字符串拼接
    fmt.Println(len(city))  // len 方法

    middle := Age(12)

    if middle >= 12 {
        fmt.Println("Middle is bigger than 12")
    }
}
```

总结： 自定义类型的原始类型的所有操作同样适用。

函数参数

```
package main

import "fmt"

type Age int

func main() {
    middle := Age(12)
    printAge(middle)
}

```

```
12. func printAge(age int) {
13.     fmt.Println("Age is", age)
14. }
```

当我们运行代码的时候会出现 `./main.go:11:10: cannot use middle (type Age) as type int in argument to printAge` 的错误。

因为 `printAge` 方法期望的是 `int` 类型，但是我们传入的参数是 `Age` ，他们虽然具有相同的值，但为不同的类型。

我们可以采用显式的类型转换（ `printAge(int(primary))` ）来修复。

不同自定义类型间的操作

```
1.  package main
2.
3.  import "fmt"
4.
5.  type Age int
6.  type Height int
7.
8.  func main() {
9.      age := Age(12)
10.     height := Height(175)
11.
12.     fmt.Println(height / age)
13. }
```

当我们运行代码会出现 `./main.go:12:21: invalid operation: height / age (mismatched types Height and Age)` 错误，修复方法使用显式转换：

```
1.  fmt.Println(int(height) / int(age))
```

结构体

数组、切片和 Map 可以用来表示同一种数据类型的集合，但是当我们要表示不同数据类型的集合时就需要用到结构体。

> 结构体是由零个或多个任意类型的值聚合成的实体

关键字 `type` 和 `struct` 用来定义结构体：

```
type StructName struct{
    FieldName type
}
```

简单示例：定义一个学生结构体

```
package main

import "fmt"

type Student struct {
    Age     int
    Name    string
}

func main() {
    stu := Student{
        Age:     18,
        Name:    "name",
    }
    fmt.Println(stu)

    // 在赋值的时候，字段名可以忽略
    fmt.Println(Student{20, "new name"})

    return
}
```

> 通常结构体中一个字段占一行，但是类型相同的字段，也可以放在同一行，例如：

```
type Student struct{
    Age           int
```

```
    Name, Address string
}
```

> 一个结构体中的字段名是唯一的，例如一下代码，出现了两个 `Name` 字段，是错误的：

```
type Student struct{
    Name string
    Name string
}
```

> 结构体中的字段如果是小写字母开头，那么其他 package 就无法直接使用该字段，例如：

```
// 在包 pk1 中定义 Student 结构体
package pk1
type Student struct{
    Age  int
    name string
}
```

```
// 在另外一个包 pk2 中调用 Student 结构体
package pk2

func main(){
    stu := Student{}
    stu.Age = 18         //正确
    stu.name = "name"  // 错误，因为`name` 字段为小写字母开头，不对外暴露
}
```

结构体中可以内嵌结构体

> 但是需要注意的是：如果嵌入的结构体是本身，那么只能用指针。请看以下例子。

```
package main

import "fmt"

type Tree struct {
    value       int
    left, right *Tree
}

```

```
func main() {
    tree := Tree{
        value: 1,
        left: &Tree{
            value: 1,
            left:  nil,
            right: nil,
        },
        right: &Tree{
            value: 2,
            left:  nil,
            right: nil,
        },
    }

    fmt.Printf(">>> %#v\n", tree)
}
```

结构体是可以比较的

> 前提是结构体中的字段类型是可以比较的

```
package main

import "fmt"

type Tree struct {
    value       int
    left, right *Tree
}

func main() {
    tree1 := Tree{
        value: 2,
    }

    tree2 := Tree{
        value: 1,
    }

    fmt.Printf(">>> %#v\n", tree1 == tree2)
}
```

结构体内嵌匿名成员

> 声明一个成员对应的数据类型而不指名成员的名字；这类成员就叫匿名成员

```
1.  package main
2.
3.  import "fmt"
4.
5.  type Person struct {
6.      Age  int
7.      Name string
8.  }
9.
10. type Student struct {
11.     Person
12. }
13.
14. func main() {
15.     per := Person{
16.         Age:  18,
17.         Name: "name",
18.     }
19.
20.     stu := Student{Person: per}
21.
22.     fmt.Println("stu.Age: ", stu.Age)
23.     fmt.Println("stu.Name: ", stu.Name)
24. }
```

函数

> 函数是语句序列的集合，能够将一个大的工作分解为小的任务，对外隐藏了实现细节

- 函数组成
  - 函数名
  - 参数列表(parameter-list)
  - 返回值(result-list)
  - 函数体(body)

```
func name(parameter-list) (result-list){
    body
}
```

- 单返回值函数

```
func plus(a, b int) (res int){
    return a + b
}
```

- 多返回值函数

```
func multi()(string, int){
    return "name", 18
}
```

- 命名返回值

```
// 被命名的返回参数的值为该类型的默认零值
// 该例子中 name 默认初始化为空字符串, height 默认初始化为 0
func namedReturnValue()(name string, height int){
    name = "xiaoming"
    height = 180
    return
}
```

- 参数可变函数

```
func sum(nums ...int)int{
    fmt.Println("len of nums is : ", len(nums))
```

```
    res := 0
    for _, v := range nums{
        res += v
    }
    return res
}

func main(){
    fmt.Println(sum(1))
    fmt.Println(sum(1,2))
    fmt.Println(sum(1,2,3))
}
```

- 匿名函数

```
func main(){
    func(name string){
       fmt.Println(name)
    }("禾木课堂")
}
```

- 闭包

```
func main() {
    addOne := addInt(1)
    fmt.Println(addOne())
    fmt.Println(addOne())
    fmt.Println(addOne())

    addTwo := addInt(2)
    fmt.Println(addTwo())
    fmt.Println(addTwo())
    fmt.Println(addTwo())
}

func addInt(n int) func() int {
    i := 0
    return func() int {
        i += n
        return i
    }
}
```

- 函数作为参数

```
func sayHello(name string) {
    fmt.Println("Hello ", name)
}

func logger(f func(string), name string) {
    fmt.Println("start calling method sayHello")
    f(name)
    fmt.Println("end calling method sayHellog")
}

func main() {
    logger(sayHello, "禾木课堂")
}
```

- 传值和传引用

```
func sendValue(name string) {
    name = "hemuketang"
}

func sendAddress(name *string) {
    *name = "hemuketang"
}

func main() {
    // 传值和传引用
    str := "禾木课堂"
    fmt.Println("before calling sendValue, str : ", str)
    sendValue(str)
    fmt.Println("after calling sendValue, str : ", str)

    fmt.Println("before calling sendAddress, str : ", str)
    sendAddress(&str)
    fmt.Println("after calling sendAddress, str: ", str)
}
```

方法

方法主要源于 OOP 语言，在传统面向对象语言中（例如 C++），我们会用一个“类”来封装属于自己的数据和函数，这些类的函数就叫做方法。

虽然 Go 不是经典意义上的面向对象语言，但是我们可以在一些接收者（自定义类型，结构体）上定义函数，同理这些接收者的函数在 Go 里面也叫做方法。

声明

方法（method）的声明和函数很相似，只不过它必须指定接收者：

```
1.  func (t T) F() {}
```

注意：

- 接收者的类型只能为用关键字 `type` 定义的类型，例如自定义类型，结构体。
- 同一个接收者的方法名不能重复（没有重载），如果是结构体，方法名还不能和字段名重复。
- 值作为接收者无法修改其值，如果有更改需求，需要使用指针类型。

简单例子

```
1.  package main
2.
3.  type T struct{}
4.
5.  func (t T) F()  {}
6.
7.  func main() {
8.      t := T{}
9.      t.F()
10. }
```

接收者类型不是任意类型

例如：

```
1.  package main
2.
3.  func (t int64) F()  {}
```

```
4.
5.  func main() {
6.      t := int64(10)
7.      t.F()
8.  }
```

当运行以下代码会得到 `cannot define new methods on non-local type int64` 类似错误信息，我们可以使用自定义类型来解决：

```
1.  package main
2.
3.  type T int64
4.  func (t T) F()  {}
5.
6.  func main() {
7.      t := T(10)
8.      t.F()
9.  }
```

> 小结：接收者不是任意类型，它只能为用关键字 `type` 定义的类型（例如自定义类型，结构体）。

命名冲突

a. 接收者定义的方法名不能重复，例如：

```
1.  package main
2.
3.  type T struct{}
4.
5.  func (T) F()         {}
6.  func (T) F(a string) {}
7.
8.  func main() {
9.      t := T{}
10.     t.F()
11. }
```

运行代码我们会得到 `method redeclared: T.F` 类似错误。

b. 结构体方法名不能和字段重复，例如：

```
1.  package main
2.
3.  type T struct{
4.    F string
5.  }
6.
7.  func (T) F(){}
8.
9.  func main() {
10.     t := T{}
11.     t.F()
12. }
```

运行代码我们会得到 `: type T has both field and method named F` 类似错误。

> 小结： 同一个接收者的方法名不能重复（没有重载）；如果是结构体，方法名不能和字段重复。

接收者可以同时为值和指针

在 Go 语言中，方法的接收者可以同时为值或者指针，例如：

```
1.  package main
2.
3.  type T struct{}
4.
5.  func (T) F()  {}
6.  func (*T) N() {}
7.
8.  func main() {
9.      t := T{}
10.     t.F()
11.     t.N()
12.
13.     t1 := &T{} // 指针类型
14.     t1.F()
15.     t1.N()
16. }
```

可以看到无论值类型 `T` 还是指针类型 `&T` 都可以同时访问 `F` 和 `N` 方法。

值和指针作为接收者的区别

同样我们先看一段代码：

```
package main

import "fmt"

type T struct {
    value int
}

func (m T) StayTheSame() {
    m.value = 3
}

func (m *T) Update() {
    m.value = 3
}

func main() {
    m := T{0}
    fmt.Println(m) // {0}

    m.StayTheSame()
    fmt.Println(m) // {0}

    m.Update()
    fmt.Println(m) // {3}
}
```

运行代码输出结果为：

```
{0}
{0}
{3}
```

> 小结：值作为接收者（ `T` ） 不会修改结构体值，而指针 `*T` 可以修改。

接口

接口类型是一种抽象类型，是方法的集合，其他类型实现了这些方法就是实现了这个接口。

```
/* 定义接口 */
type interface_name interface {
   method_name1 [return_type]
   method_name2 [return_type]
   method_name3 [return_type]
   ...
   method_namen [return_type]
}
```

简单示例：打印不同几何图形的面积和周长

```
package main

import (
    "fmt"
    "math"
)

type geometry interface {
    area() float32
    perim() float32
}

type rect struct {
    len, wid float32
}

func (r rect) area() float32 {
    return r.len * r.wid
}

func (r rect) perim() float32 {
    return 2 * (r.len + r.wid)
}

type circle struct {
    radius float32
```

```
}

func (c circle) area() float32 {
    return math.Pi * c.radius * c.radius
}

func (c circle) perim() float32 {
    return 2 * math.Pi * c.radius
}

func show(name string, param interface{}) {
    switch param.(type) {
    case geometry:
        // 类型断言
        fmt.Printf("area of %v is %v \n", name, param.(geometry).area())
        fmt.Printf("perim of %v is %v \n", name, param.(geometry).perim())
    default:
        fmt.Println("wrong type!")
    }
}

func main() {
    rec := rect{
        len: 1,
        wid: 2,
    }
    show("rect", rec)

    cir := circle{
        radius: 1,
    }
    show("circle", cir)

    show("test", "test param")
}
```

接口中可以内嵌接口

对上述例子做以下修改：

- 首先添加 `tmp` 接口，该接口定义了 `area()` 方法
- 将 `tmp` 作为 `geometry` 接口中的匿名成员，并且将 `geometry` 接口中原本定义的

`area()` 方法删除

> 完成以上两步后，`geometry` 接口将会拥有 `tmp` 接口所定义的所有方法。运行结果和上述例子相同。

```
type tmp interface{
    area() float32
}

type geometry interface {
    // area() float32
    tmp
    perim() float32
}
```

流程控制

分支循环

在编写 Go 程序的时候，我们不仅会用前面学到的数据结构来存储数据，还会用到 `if` 、 `switch` 、 `for` 来进行条件判断和流程控制，今天我们就来一起学习下它们。

if

`if` 主要用于条件判断，语法为：

```
if 条件 {
  # 业务代码
}
```

先看一个简单例子：

```
package main

import "fmt"

func main() {
    age := 7

    if age > 6 {
        fmt.Println("It's primary school")
    }
}
```

我们可以在条件中使用 `&` 或 `||` 来进行组合判断：

```
package main

import "fmt"

func main() {
    age := 7

    if age > 6 && age <= 12 {
        fmt.Println("It's primary school")
    }
}
```

我们还可以使用 `if` .. `else if` .. `else` 来实现多分支的条件判断：

```
1.  package main
2.
3.  import "fmt"
4.
5.  func main() {
6.      age := 13
7.
8.      if age > 6 && age <= 12 {
9.          fmt.Println("It's primary school")
10.     } else if age > 12 && age <= 15 {
11.         fmt.Println("It's middle school")
12.     } else {
13.         fmt.Println("It's high school")
14.     }
15. }
```

switch

如果我们的条件分支太多，可以考虑使用 `switch` 替换 `if` ，例如：

```
1.  package main
2.
3.  import "fmt"
4.
5.  func main() {
6.      age := 10
7.
8.      switch age {
9.      case 5:
10.         fmt.Println("The age is 5")
11.     case 7:
12.         fmt.Println("The age is 7")
13.     case 10:
14.         fmt.Println("The age is 10")
15.     default:
16.         fmt.Println("The age is unkown")
17.     }
18. }
```

注意：在 Go 中 `switch` 只要匹配中了就会中止剩余的匹配项，这和 `Java` 很大不一样，它

需要使用 `break` 来主动跳出。

`switch` 的 `case` 条件可以是多个值，例如：

```
1.  package main
2.
3.  import "fmt"
4.
5.  func main() {
6.      age := 7
7.
8.      switch age {
9.      case 7, 8, 9, 10, 11, 12:
10.         fmt.Println("It's primary school")
11.     case 13, 14, 15:
12.         fmt.Println("It's middle school")
13.     case 16, 17, 18:
14.         fmt.Println("It's high school")
15.     default:
16.         fmt.Println("The age is unkown")
17.     }
18. }
```

注意： 同一个 case 中的多值不能重复。

`switch` 还可以使用 `if..else` 作为 `case` 条件，例如：

```
1.  package main
2.
3.  import "fmt"
4.
5.  func main() {
6.      age := 7
7.
8.      switch {
9.      case age >= 6 && age <= 12:
10.         fmt.Println("It's primary school")
11.     case age >= 13 && age <= 15:
12.         fmt.Println("It's middle school")
13.     case age >= 16 && age <= 18:
14.         fmt.Println("It's high school")
15.     default:
16.         fmt.Println("The age is unkown")
```

```
    }
}
```

小技巧： 使用 `switch` 对 `interface{}` 进行断言，例如：

```
package main

import "fmt"

func checkType(i interface{}) {
    switch v := i.(type) {
    case int:
        fmt.Printf("%v is an in\n", v)
    case string:
        fmt.Printf("%v is a string\n", v)
    default:
        fmt.Printf("%v's type is unkown\n", v)
    }
}

func main() {
    checkType(8)
    checkType("hello, world")
}
```

for

使用 `for` 来进行循环操作，例如：

```
package main

import "fmt"

func main() {
    for i := 0; i < 2; i++ {
        fmt.Println("loop with index", i)
    }
}
```

使用 `for..range` 对数组、切片、map、 字符串等进行循环操作，例如：

```go
package main

import "fmt"

func main() {
    numbers := []int{1, 2, 3}

    for i, v := range numbers {
        fmt.Printf("numbers[%d] is %d\n", i, v)
    }
}
```

注意：这里的 `i` 、 `v` 是切片元素的位置索引和值。

```go
package main

import "fmt"

func main() {
    cityCodes := map[string]int{
        "北京": 1,
        "上海": 2,
    }

    for i, v := range cityCodes {
        fmt.Printf("%s is %d\n", i, v)
    }
}
```

注意：这里的 `i` 、 `v` 是 `map` 的一组键值对的键和值。

使用 `continue` 和 `break` 对循环进行控制，例如：

```go
package main

import "fmt"

func main() {
    numbers := []int{1, 2, 3, 4, 5}

    for i, v := range numbers {
        if v == 4 {
```

```
            break
        }

        if v%2 == 0 {
            continue
        }

        fmt.Printf("numbers[%d] is %d\n", i, v)
    }
}
```

注意：

- `break` 会结束所有循环。
- `continue` 会跳过当前循环直接进入下一次循环。

异常处理

defer

上一节课介绍了常见的流程控制，这一节课将介绍一个 Go 特有的流程控制语句： defer

defer 通常用于延迟调用指定的函数。例如：

```
1.  func outerFunc() {
2.      defer fmt.Printf(" World!\n")
3.
4.      fmt.Print("Hello")
5.  }
```

上例最终输出的结果是： “Hello World!”.

这是因为： `defer` 会在 `outerFunc` 退出之前执行打印操作，因此被 `defer` 调用的函数也称为“延迟函数”。

defer 常用场景

defer语句经常被用于处理成对的操作，如打开和关闭，连接和断开连接，加锁和释放锁。恰当使用 defer 能够保证资源正确释放。以下是几个例子：

```
1.  // 使用 defer 关闭 http 请求响应体的 Body
2.  func closeBody(url string) error {
3.      resp, err := http.Get(url)
4.      defer resp.Body.Close()
5.      // ... do more stuff ...
6.
7.      return err
8.  }
```

```
1.  // 使用 defer 关闭文件句柄
2.  func closeFile(filename string) error{
3.      f, err := os.Open(filename)
4.      defer f.Close()
5.      // ... do more stuff ...
6.
7.      return err
8.  }
```

```
// 使用 defer 解锁
func BillCustomer(c *Customer)  {
    c.mutex.Lock()
    defer c.mutex.Unlock()
    // ... do more stuff ...

    return
}
```

defer 使用中一些注意点

- 例子1

请看以下例子，猜下输出结果是？

```
func printNumber() {
    for i := 0; i<5; i++{
        defer func(){
            fmt.Println(i)
        }()
    }
}
```

最终输出的结果是 `5 5 5 5 5` 。 这是因为 defer 所调用的函数是延迟执行的。等到执行 defer 所调用的函数时，i 已经是 5 了。接着看下面这个例子：

```
func printNum() {
    for i := 0; i < 5; i++ {
        defer func(v int) {
            fmt.Println(v)
        }(i)
    }
}
```

这个例子最终输出的是: `4 3 2 1 0` 。具体是什么原因留作大家思考。

- 例子2

```
func testDefer() (i int) {
    defer func() {
        fmt.Println(i)
```

```
        i = 4
    }()

    return 2
}
```

以上例子，最终返回的是 4。因为 `return 2` 执行后，变量 `i` 赋值为 2， 但是随后执行了 defer 函数， `i` 被赋值为4，所以最终返回结果为4。

panic

当程序遇到致命错误导致无法继续运行时就会出发 `panic` ，例如：数组越界，空指针等。

以下代码将会出发数组越界异常。

```
s := []int{1, 2, 3}
for i := 0; i <= 4; i++ {
    fmt.Println(s[i])
}
```

上述例子中因为数组越界，触发了 runtime 异常，导致程序退出。在实际开发中，也可以主动调用 `panic` 函数达到同样效果。

```
func panicFunc() {
    panic(errors.New("this is test for panic"))
}
```

recover

顾名思义，recover 函数能使当前程序从 panic 中恢复。recover 能够拦截 panic 事件，使得程序不会因为意外而触发 panic 事件而完全退出。

recover 函数返回的是一个 interfac{} 类型的结果，如果捕获到了 panic 事件，该结果就为非 nil。见下例:

```
func panicFunc() {
    defer func() {
        if p := recover(); p != nil {
            fmt.Println("recover panic")
        }
    }()

```

```
8.      panic(errors.New("this is test for panic"))
9.  }
10. func main() {
11.     fmt.Println("before panic")
12.
13.     panicFunc()
14.
15.     fmt.Println("after panic")
16. }
```

课后练习

- 思考以下代码的输出，并解释为什么

```
1.  func printNumbers(){
2.      for i:=0; i<5; i++{
3.          defer func(n int){
4.              fmt.Printf(n)
5.          }(i * 2)
6.      }
7.  }
```

并发

- Goroutine
- Channel
- 锁的使用
- 原子操作

Goroutine （并发）

并发指的是多个任务被（一个）cpu 轮流切换执行，在 Go 语言里面主要用 goroutine （协程）来实现并发，类似于其他语言中的线程（绿色线程）。

语法

```
go f(x, y, z)
```

具体例子

首先我们看一个例子：

```
package main

import (
    "log"
    "time"
)

func doSomething(id int) {
    log.Printf("before do job:(%d) \n", id)
    time.Sleep(3 * time.Second)
    log.Printf("after do job:(%d) \n", id)
}

func main() {
    doSomething(1)
    doSomething(2)
    doSomething(3)
}
```

输出结果为：

```
2018/03/16 12:13:20 before do job:(1)
2018/03/16 12:13:23 after do job:(1)
2018/03/16 12:13:23 before do job:(2)
2018/03/16 12:13:26 after do job:(2)
2018/03/16 12:13:26 before do job:(3)
2018/03/16 12:13:29 after do job:(3)
```

可以看到执行完结果总共耗时 9 秒，每个任务是阻塞的。

我们可以使用 goroutine 并发执行任务，从而整体加快速度，下面是使用 goroutine 改进的代码：

```
1.  package main
2.
3.  import (
4.      "log"
5.      "time"
6.  )
7.
8.  func doSomething(id int) {
9.      log.Printf("before do job:(%d) \n", id)
10.     time.Sleep(3 * time.Second)
11.     log.Printf("after do job:(%d) \n", id)
12. }
13.
14. func main() {
15.     go doSomething(1)
16.     go doSomething(2)
17.     go doSomething(3)
18. }
```

当运行代码的时候，会发现没有任何输出。

这是因为我们的 `main()` 函数其实也是在一个 goroutine 中执行，但是 `main()` 执行完毕后，其他三个 goroutine 还没开始执行，所以就无法看到输出结果。

为了看到输出结果，我们可以使用 `time.Sleep()` 方法让 `main()` 函数延迟结束，例如：

```
1.  package main
2.
3.  import (
4.      "log"
5.      "time"
6.  )
7.
8.  func doSomething(id int) {
9.      log.Printf("before do job:(%d) \n", id)
10.     time.Sleep(3 * time.Second)
11.     log.Printf("after do job:(%d) \n", id)
```

```
}

func main() {
    go doSomething(1)
    go doSomething(2)
    go doSomething(3)
    time.Sleep(4 * time.Second)
}
```

输出结果为：

```
2018/03/16 12:24:23 before do job:(2)
2018/03/16 12:24:23 before do job:(1)
2018/03/16 12:24:23 before do job:(3)
2018/03/16 12:24:26 after do job:(3)
2018/03/16 12:24:26 after do job:(2)
2018/03/16 12:24:26 after do job:(1)
```

可以看到，执行完所有任务从原本的 9 秒下降到 3 秒，大大提高了我们的效率，根据打印输出结果还可以看出：

- 多个 goroutine 的执行是随机。
- 对于 IO 密集型任务特别有效，比如文件，网络读写。

使用 `sync.WaitGroup` 实现同步

上面例子中，其实我们还可以使用 `sync.WaitGroup` 来等待所有的 goroutine 结束，从而实现并发的同步，这比使用 `time.Sleep()` 更加优雅，例如：

```
package main

import (
    "log"
    "sync"
    "time"
)

func doSomething(id int, wg *sync.WaitGroup) {
    defer wg.Done()

    log.Printf("before do job:(%d) \n", id)
    time.Sleep(3 * time.Second)
```

```
    log.Printf("after do job:(%d) \n", id)
}

func main() {
    var wg sync.WaitGroup
    wg.Add(3)

    go doSomething(1, &wg)
    go doSomething(2, &wg)
    go doSomething(3, &wg)

    wg.Wait()
    log.Printf("finish all jobs\n")
}
```

运行代码输出结果为:

```
2018/03/16 13:56:09 before do job:(1)
2018/03/16 13:56:09 before do job:(3)
2018/03/16 13:56:09 before do job:(2)
2018/03/16 13:56:12 after do job:(1)
2018/03/16 13:56:12 after do job:(2)
2018/03/16 13:56:12 after do job:(3)
2018/03/16 13:56:12 finish all jobs
```

一个小坑

我们一起来猜猜，下面一段代码运行结果是什么？

```
package main

import (
    "fmt"
    "time"
)

func main() {
    for i := 0; i < 3; i++ {
        go func() {
            fmt.Println(i)
        }()
    }
```

```
14.
15.     time.Sleep(1 * time.Second)
16. }
```

运行代码，输出结果为：

```
1. 3
2. 3
3. 3
```

我们想要的是随机打印 `0,1,2` ，但实际输出结果和我们预期不一致，这是原因：

1. 所有 goroutine 代码片段中的 `i` 是同一个变量，待循环结束的时候，它的值为 `3` 。
2. `main()` 循环结束后才开始并发执行的新生成的 goroutine。

修复方法：

```
1. package main
2.
3. import (
4.     "fmt"
5.     "time"
6. )
7.
8. func main() {
9.     for i := 0; i < 3; i++ {
10.         go func(v int) {
11.             fmt.Println(v)
12.         }(i)
13.     }
14.
15.     time.Sleep(1 * time.Second)
16. }
```

我们可以通过方法传参的方式，将 `i` 的值拷贝到新的变量 `v` 中，而在每一个 goroutine 都对应了一个属于自己作用域的 `v` 变量，所以最终打印结果为随机的 `0,1,2` 。

channel

channel 简介

goroutine 是 Go 中实现并发的重要机制，channel 是 goroutine 之间进行通信的重要桥梁。

使用内建函数 make 可以创建 channel，举例如下：

```
1.  ch := make(chan int)  // 注意： channel 必须定义其传递的数据类型
```

也可以用 var 声明 channel，如下：

```
1.  var ch chan int
```

以上声明的 channel 都是双向的，意味着可以该 channel 可以发送数据，也可以接收数据。

“发送”和“接收”是 channel 的两个基本操作。

```
1.  ch <- x // channel 接收数据 x
2.
3.  x <- ch // channel 发送数据并赋值给 x
4.
5.  <- ch // channel 发送数据，忽略接受者
```

channel buffer

上文提到，可以通过 `make(chan int)` 创建channel，此类 channel 称之为非缓冲通道。事实上 channel 可以定义缓冲大小，如下：

```
1.  chInt := make(chan int)       // unbuffered channel  非缓冲通道
2.  chBool := make(chan bool, 0)  // unbuffered channel  非缓冲通道
3.  chStr := make(chan string, 2) // bufferd channel     缓冲通道
```

需要注意的是，程序中必须同时有不同的 goroutine 对非缓冲通道进行发送和接收操作，否则会造成阻塞。

以下是一个错误的使用示例：

```
1.  func main() {
2.      ch := make(chan string)
```

```
3.
4.      ch <- "ping"
5.
6.      fmt.Println(<-ch)
7.  }
```

这一段代码运行后提示错误： `fatal error: all goroutines are asleep - deadlock!` 。

因为 main 函数是一个 goroutine，在这一个 goroutine 中发送了数据给非缓冲通道，但是却没有另外一个 goroutine 从非缓冲通道中里读取数据，所以造成了阻塞或者称为死锁。

在以上代码中添加一个 goroutine 从非缓冲通道中读取数据，程序就可以正常工作。如下所示：

```
1.  func main() {
2.      ch := make(chan string)
3.
4.      go func() {
5.          ch <- "ping"
6.      }()
7.
8.      fmt.Println(<-ch)
9.  }
```

与非缓冲通道不同，缓冲通道可以在同一个 goroutine 内接收容量范围内的数据，即便没有另外的 goroutine 进行读取操作，如下代码可以正常执行：

```
1.  func main() {
2.      ch := make(chan int, 2)
3.
4.      ch <- 1
5.
6.      ch <- 2
7.  }
```

单向 channel

单向通道即限定了该 channel 只能接收或者发送数据，单向通道通常作为函数的参数，如下例所示：

```
1.  func receive(receiver chan<- string, msg string) {
2.      receiver <- msg
3.  }
4.
```

```
5.  func send(sender <-chan string, receiver chan<- string) {
6.      msg := <-sender
7.      receiver <- msg
8.  }
9.
10. func main() {
11.     ch1 := make(chan string, 1)
12.     ch2 := make(chan string, 1)
13.
14.     receive(ch1, "pass message")
15.     send(ch1, ch2)
16.
17.     fmt.Println(<-ch2)
18. }
```

需要注意的是，在变量声明中是不应该出现单向通道的，因为通道本来就是为了通信而生，只能接收或者只能发送数据的通道是没有意义的。请看下面这个例子：

```
1.  func main() {
2.      ch := make(chan <- string , 1)
3.      ch <- "str"
4.  }
```

这个例子中定义了一个只能用来接收数据的通道，从语法上来看没有错误，但这是一种糟糕的实践。

channel 遍历和关闭

close() 函数可以用于关闭 channel，关闭后的 channel 中如果有缓冲数据，依然可以读取，但是无法再发送数据给已经关闭的channel。

```
1.  func main() {
2.      ch := make(chan int, 10)
3.      for i := 0; i < 10; i++ {
4.          ch <- i
5.      }
6.      close(ch)
7.
8.      res := 0
9.      for v := range ch {
10.         res += v
11.     }
12.
```

```
    fmt.Println(res)
}
```

select 语句

select 专门用于通道发送和接收操作，看起来和 switch 很相似，但是进行选择和判断的方法完全不同。

在下述例子中，通过 select 的使用，保证了 worker 中的事务可以执行完毕后才退出 main 函数

```
func strWorker(ch chan string) {
    time.Sleep(1 * time.Second)
    fmt.Println("do something with strWorker...")
    ch <- "str"
}

func intWorker(ch chan int) {
    time.Sleep(2 * time.Second)
    fmt.Println("do something with intWorker...")
    ch <- 1
}

func main() {
    chStr := make(chan string)
    chInt := make(chan int)

    go strWorker(chStr)
    go intWorker(chInt)

    for i := 0; i < 2; i++ {
        select {
        case <-chStr:
            fmt.Println("get value from strWorker")

        case <-chInt:
            fmt.Println("get value from intWorker")

        }
    }
}
```

> 思考： 如果上述例子中，没有这个 select ，那么 worker 函数是否有机会执行？

通过 channel 实现同步机制

一个经典的例子如下，main 函数中起了一个 goroutine，通过非缓冲队列的使用，能够保证在 goroutine 执行结束之前 main 函数不会提前退出。

```
1.  func worker(done chan bool){
2.      fmt.Println("start working...")
3.      done <- true
4.      fmt.Println("end working...")
5.  }
6.
7.  func main() {
8.      done := make(chan bool, 1)
9.
10.     go worker(done)
11.
12.     <- done
13. }
```

思考：如果把上述例子中的 `<-done` 注释掉，运行结果会如何？

锁

在前面我们已经讲了如何使用 channel 在多个 goroutine 之间进行通信，其实对于并发还有一种较为常用通信方式，那就是共享内存。

首先我们来看一个例子：

```
1.  package main
2.
3.  import (
4.      "log"
5.      "time"
6.  )
7.
8.  var name string
9.
10. func main() {
11.     name = "小明"
12.
13.     go printName()
14.     go printName()
15.
16.     time.Sleep(time.Second)
17.
18.     name = "小红"
19.
20.     go printName()
21.     go printName()
22.
23.     time.Sleep(time.Second)
24. }
25.
26. func printName() {
27.     log.Println("name is", name)
28. }
```

运行程序，可以得到类似输出结果：

```
1.  2018/03/23 14:53:28 name is 小明
2.  2018/03/23 14:53:28 name is 小明
3.  2018/03/23 14:53:29 name is 小红
```

```
4.  2018/03/23 14:53:29 name is 小红
```

可以看到在两个 goroutine 中我们都可以访问 `name` 这个变量，当修改它后，在不同的 goroutine 中都可以同时获取到最新的值。

这就是一个最简单的通过共享变量（内存）的方式在多个 goroutine 进行通信的方式。

下面再来看一个例子：

```
1.  package main
2.
3.  import (
4.      "fmt"
5.      "sync"
6.  )
7.
8.  func main() {
9.      var (
10.         wg      sync.WaitGroup
11.         numbers []int
12.     )
13.
14.     for i := 0; i < 10; i++ {
15.         wg.Add(1)
16.         go func(i int) {
17.             numbers = append(numbers, i)
18.             wg.Done()
19.         }(i)
20.     }
21.
22.     wg.Wait()
23.
24.     fmt.Println("The numbers is", numbers)
25. }
```

多次运行代码，可以得到类似输出：

```
1.  The numbers is [0 1 5 4 7]
2.  The numbers is [0 5 7]
```

可以看到当我们并发对同一个切片进行写操作的时候，会出现数据不一致的问题，这就是一个典型的共享变量的问题。

针对这个问题我们可以使用 Lock（锁）来修复，从而保证数据的一致性，例如：

```
package main

import (
    "fmt"
    "sync"
)

func main() {
    var (
        wg      sync.WaitGroup
        numbers []int

        mux sync.Mutex
    )

    for i := 0; i < 10; i++ {
        wg.Add(1)
        go func(i int) {
            mux.Lock()
            numbers = append(numbers, i)
            mux.Unlock()

            wg.Done()
        }(i)
    }

    wg.Wait()

    fmt.Println("The numbers is", numbers)
}
```

修改过后，我们再次运行代码，可以看到最后的 numbers 都会包含 0~9 这个10个数字。

`sync.Mutex` 是互斥锁，只有一个信号标量；在 Go 中还有一种读写锁 `sync.RWMutex`，对于我们的共享对象，如果可以分离出读和写两个互斥信号的情况，可以考虑使用它来提高读的并发性能。

例如代码：

```
package main

```

```
import (
    "fmt"
    "sync"
    "sync/atomic"
    "time"
)

func main() {
    var (
        mux    sync.Mutex
        state1 = map[string]int{
            "a": 65,
        }
        muxTotal uint64

        rw     sync.RWMutex
        state2 = map[string]int{
            "a": 65,
        }
        rwTotal uint64
    )

    for i := 0; i < 10; i++ {
        go func() {
            for {
                mux.Lock()
                _ = state1["a"]
                mux.Unlock()
                atomic.AddUint64(&muxTotal, 1)
            }
        }()
    }

    for i := 0; i < 10; i++ {
        go func() {
            for {
                rw.RLock()
                _ = state2["a"]
                rw.RUnlock()
                atomic.AddUint64(&rwTotal, 1)
            }
        }()
```

```
45.     }
46.
47.     time.Sleep(time.Second)
48.
49.     fmt.Println("sync.Mutex readOps is", muxTotal)
50.     fmt.Println("sync.RWMutex readOps is", rwTotal)
51. }
```

运行代码可以得到如下结果：

```
1. sync.Mutex readOps is 1561870
2. sync.RWMutex readOps is 15651069
```

可以看到使用 `sync.RWMutex` 的读的并发能力大概是 `sync.Mutex` 的十倍，从而大大提高了其并发能力。

总结：

- 我们可以通过共享内存的方式实现多个 goroutine 中的通信。
- 多个 goroutine 对于共享的内存进行写操作的时候，可以使用 `Lock` 来避免数据不一致的情况。
- 对于可以分离为读写操作的共享数据可以考虑使用 `sync.RWMutex` 来提高其读的并发能力。

原子操作

> 本文讲解 golang 中 sync.atomic 的常见操作

atomic 提供的原子操作能够确保任一时刻只有一个goroutine对变量进行操作，善用 atomic 能够避免程序中出现大量的锁操作。

atomic常见操作有:

- 增减
- 载入
- 比较并交换
- 交换
- 存储

下面将分别介绍这些操作。

增减操作

atomic 包中提供了如下以Add为前缀的增减操作:

- func AddInt32(addr *int32, delta int32) (new int32)
- func AddInt64(addr *int64, delta int64) (new int64)
- func AddUint32(addr *uint32, delta uint32) (new uint32)
- func AddUint64(addr *uint64, delta uint64) (new uint64)
- func AddUintptr(addr *uintptr, delta uintptr) (new uintptr)

> 需要注意的是，第一个参数必须是指针类型的值，通过指针变量可以获取被操作数在内存中的地址，确保同一时间只有一个goroutine能够进行操作。

先来看一个例子： 分别用“锁”和原子操作来实现多个 goroutine 对同一个变量进行累加操作。

使用锁实现

```
1.  package main
2.
3.  import (
4.      "fmt"
5.      "sync"
6.      "time"
7.  )
8.
```

```
func main() {
    var (
        mux   sync.Mutex
        total = 0
    )

    for i := 0; i < 10; i++ {
        go func() {
            for {
                mux.Lock()
                total += 1
                mux.Unlock()
                time.Sleep(time.Millisecond)
            }
        }()
    }

    time.Sleep(time.Second)
    fmt.Println("The total number is", atomic.LoadUint64(&total))
}
```

运行代码可以得到类似输出：

```
The total number is 7770
```

使用 atomic 实现

```
package main

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

func main() {
    var total int64

    for i := 0; i < 10; i++ {
        go func() {
            for {
                atomic.AddInt64(&total, 1)
```

```
16.                 time.Sleep(time.Millisecond)
17.             }
18.         }()
19.     }
20.
21.     time.Sleep(time.Second)
22.     fmt.Println("The total number is", total)
23. }
```

运行代码可以得到类似输出：

```
1.  The total number is 7701
```

载入操作

atomic 包中提供了如下以Load为前缀的载入操作：

- func LoadInt32(addr *int32) (val int32)
- func LoadInt64(addr *int64) (val int64)
- func LoadPointer(addr *unsafe.Pointer) (val unsafe.Pointer)
- func LoadUint32(addr *uint32) (val uint32)
- func LoadUint64(addr *uint64) (val uint64)
- func LoadUintptr(addr *uintptr) (val uintptr)

载入操作能够保证原子的读变量的值，当读取的时候，任何其他 goroutine 都无法对该变量进行读写.

比较并交换

该操作简称 CAS(Compare And Swap)。 这类操作的前缀为 `CompareAndSwap` ：

- func CompareAndSwapInt32(addr *int32, old, new int32) (swapped bool)
- func CompareAndSwapInt64(addr *int64, old, new int64) (swapped bool)
- func CompareAndSwapPointer(addr *unsafe.Pointer, old, new unsafe.Pointer) (swapped bool)
- func CompareAndSwapUint32(addr *uint32, old, new uint32) (swapped bool)
- func CompareAndSwapUint64(addr *uint64, old, new uint64) (swapped bool)
- func CompareAndSwapUintptr(addr *uintptr, old, new uintptr) (swapped bool)

该操作在进行交换前首先确保变量的值未被更改，即仍然保持参数 `old` 所记录的值，满足此前提下才进行交换操作。CAS的做法类似操作数据库时常见的乐观锁机制。需要注意的是，当有大量的

goroutine 对变量进行读写操作时，可能导致CAS操作无法成功，这时可以利用for循环多次尝试。

```
var value int64

func atomicAddOp(tmp int64) {
    for {
        oldValue := value
        if atomic.CompareAndSwapInt64(&value, oldValue, oldValue+tmp) {
            return
        }
    }
}
```

交换

此类操作的前缀为 `Swap` :

- func SwapInt32(addr *int32, new int32) (old int32)
- func SwapInt64(addr *int64, new int64) (old int64)
- func SwapPointer(addr *unsafe.Pointer, new unsafe.Pointer) (old unsafe.Pointer)
- func SwapUint32(addr *uint32, new uint32) (old uint32)
- func SwapUint64(addr *uint64, new uint64) (old uint64)
- func SwapUintptr(addr *uintptr, new uintptr) (old uintptr)

相对于CAS，明显此类操作更为暴力直接，并不管变量的旧值是否被改变，直接赋予新值然后返回背替换的值。

存储

此类操作的前缀为 `Store` :

- func StoreInt32(addr *int32, val int32)
- func StoreInt64(addr *int64, val int64)
- func StorePointer(addr *unsafe.Pointer, val unsafe.Pointer)
- func StoreUint32(addr *uint32, val uint32)
- func StoreUint64(addr *uint64, val uint64)
- func StoreUintptr(addr *uintptr, val uintptr)

在原子地存储某个值的过程中，任何 goroutine 都不会进行针对同一个值的读或写操作。

结论

- `atomic` 和锁的方式其性能没有太大区别。
- `atomic` 写法相较于使用锁，更简单。

读文件

在日常开发中我们少不了对文件读取，今天我们就从三部分来讲解：全量读，带缓冲区读，任意位置读。

全量读

我们先来看看一个简单的例子：

```
1.  package main
2.
3.  import (
4.      "fmt"
5.      "io/ioutil"
6.  )
7.
8.  func main() {
9.      dat, err := ioutil.ReadFile("./main.go")
10.     fmt.Println(err)
11.     fmt.Println(string(dat))
12. }
```

运行程序可以打印整个 `main.go` 文件内容，如果我们将 `./main.go` 修改为 `./main.go1`，程序将出现 `no such file or directory` 的错误，所以在文件读取的时候一定要注意检查 `err`。

带缓冲区读

```
1.  package main
2.
3.  import (
4.      "fmt"
5.      "os"
6.  )
7.
8.  func main() {
9.      f, _ := os.Open("./main.go")
10.     defer f.Close()
11.
12.     buf := make([]byte, 16)
```

```
    f.Read(buf)

    fmt.Println(string(buf))
}
```

运行程序会输出 `main.go` 的前 16 个字节内容，具体为：

```
package main

im
```

任意位置读

有些时候我们想在一个文件特定地方读取特定长度的内容，那我们有什么方法可以使用呢？

第一种： f.Seek + f.Read

```
package main

import (
    "fmt"
    "os"
)

func main() {
    f, _ := os.Open("./main.go")
    defer f.Close()

    b1 := make([]byte, 2)

    f.Seek(5, 0)
    f.Read(b1)
    fmt.Println(string(b1))
}
```

运行代码输出结果为：

```
ge
```

第二种：使用 `f.ReadAt`

```
package main

import (
    "fmt"
    "os"
)

func main() {
    f, _ := os.Open("./main.go")
    defer f.Close()

    b1 := make([]byte, 2)

    f.ReadAt(b1, 5)
    fmt.Println(string(b1))
}
```

运行结果同样为：

```
ge
```

但注意：

第一种方式是非并发安全的，例如：

```
package main

import (
    "fmt"
    "os"
    "time"
)

func main() {
    f, _ := os.Open("./main.go")
    defer f.Close()

    for i := 0; i < 5; i++ {
        go func() {
            b1 := make([]byte, 2)

            f.Seek(5, 0)
```

```
            f.Read(b1)
            fmt.Println(string(b1))

            f.Seek(2, 0)
            f.Read(b1)
            fmt.Println(string(b1))
        }()
    }

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

输出结果为：

```
ge
ge
ge
ck
ck
ai
ck
 m
ck
ck
```

第二种 `f.ReadAt` 是并发安全的，例如：

```
package main

import (
    "fmt"
    "os"
    "time"
)

func main() {
    f, _ := os.Open("./main.go")
    defer f.Close()

    for i := 0; i < 5; i++ {
        go func() {
            b1 := make([]byte, 2)
```

```
            f.ReadAt(b1, 5)
            fmt.Println(string(b1))

            f.ReadAt(b1, 2)
            fmt.Println(string(b1))
        }()
    }

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

输出结果为：

```
ge
ge
ck
ck
ge
ge
ck
ck
ge
ck
```

项目实战

实战项目一： 如何使用 `buf` 实现 `ioutil.ReadFile` 类似效果：

```
package main

import (
    "fmt"
    "io"
    "os"
)

func main() {
    f, _ := os.Open("./main.go")
    defer f.Close()

    content := make([]byte, 0)
    buf := make([]byte, 16)
```

```
15.
16.     for {
17.         n, err := f.Read(buf)
18.         if err == io.EOF {
19.             break
20.         }
21.
22.         if n == 16 {
23.             content = append(content, buf...)
24.         } else {
25.             content = append(content, buf[0:n]...)
26.         }
27.     }
28.
29.     fmt.Println(string(content))
30. }
```

实战项目二： 使用 `bufio` 实现行统计：

```
1.  package main
2.
3.  import (
4.      "bufio"
5.      "fmt"
6.      "io"
7.      "os"
8.  )
9.
10. func main() {
11.     f, _ := os.Open("./main.go")
12.     defer f.Close()
13.
14.     br := bufio.NewReader(f)
15.     totalLine := 0
16.
17.     for {
18.         _, isPrefix, err := br.ReadLine()
19.
20.         if !isPrefix {
21.             totalLine += 1
22.         }
23.
```

```
        if err == io.EOF {
            break
        }
    }

    fmt.Println("total lines is:", totalLine)
}
```

运行结果为：

```
total lines is: 31
```

写文件

上一节课介绍了 Go 读文件的常用操作，本章节将介绍 Go 写文件的相关操作，包括：

- 创建文件
- 在文件指定位置写入内容
- 通过 Buffered Writter 写文件

创建文件

- 使用 os.Create(name string) 方法创建文件
- 使用 os.Stat(name string) 方法获取文件信息

简单示例：

```
1.  package main
2.
3.  import (
4.      "fmt"
5.      "io/ioutil"
6.      "os"
7.  )
8.
9.  func checkErr(err error) {
10.     if err != nil {
11.         panic(err)
12.     }
13. }
14.
15. func main() {
16.     path := "test.txt"
17.
18.     newFile, err := os.Create(path)
19.     checkErr(err)
20.     defer newFile.Close()
21.
22.     fileInfo, err := os.Stat(path)
23.     if err != nil {
24.         if os.IsNotExist(err) {
25.             fmt.Println("file doesn't exist!")
26.             return
```

```
        }
    }
    fmt.Println("file does exist, file name : ", fileInfo.Name())
}
```

除了上述方法外，还可以通过 `ioutil.WriteFile` 一步完成文件创建和写入操作。假如该文件之前已经存在，那么将会覆盖掉原来的内容，写入新的内容。

示例如下：

```
package main

import (
    "fmt"
    "io/ioutil"
)

func checkErr(err error) {
    if err != nil {
        panic(err)
    }
}

func readFile(path string) {
    data, err := ioutil.ReadFile(path)
    checkErr(err)
    fmt.Println("file content: ", string(data))
}

func main() {
    path := "test.txt"
    str := "hello"

    err := ioutil.WriteFile(path, []byte(str), 0644)
    checkErr(err)
    readFile(path)
}
```

在文件指定位置写入内容

使用 writeAt 可以在文件指定位置写入内容

```go
package main

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

func checkErr(err error) {
    if err != nil {
        panic(err)
    }
}

func readFile(path string) {
    data, err := ioutil.ReadFile(path)
    checkErr(err)
    fmt.Println("file content: ", string(data))
}

func main() {
    path := "test.txt"
    str := "hello"
    newStr := "world"

    newFile, err := os.Create(path)
    checkErr(err)

    n1, err := newFile.WriteString(str)
    checkErr(err)
    fmt.Println("n1: ", n1)
    readFile(path)

    n2, err := newFile.WriteAt([]byte(newStr), 6)
    checkErr(err)
    fmt.Println("n2: ", n2)
    readFile(path) // the file content should be "helloworld"

    n3, err := newFile.WriteAt([]byte(newStr), 0)
    checkErr(err)
    fmt.Println("n3: ", n3)
    readFile(path) // the file content should be "worldworld"
```

```
43. }
```

通过 Buffered Writer 写文件

使用 Buffered Writer 可以避免太多次的磁盘 IO 操作。写入的内容首先是存在内存中，当调用 `Flush()` 方法后才会写入磁盘。

```
1.  package main
2.
3.  import (
4.      "bufio"
5.      "fmt"
6.      "io/ioutil"
7.      "os"
8.  )
9.
10. func checkErr(err error) {
11.     if err != nil {
12.         panic(err)
13.     }
14. }
15.
16. func readFile(path string) {
17.     data, err := ioutil.ReadFile(path)
18.     checkErr(err)
19.     fmt.Println("file content: ", string(data))
20. }
21.
22. func main() {
23.     path := "test.txt"
24.     str := "hello"
25.
26.     newFile, err := os.Create(path)
27.     checkErr(err)
28.     defer newFile.Close()
29.
30.     bufferWriter := bufio.NewWriter(newFile)
31.
32.     for _, v := range str {
33.         written, err := bufferWriter.WriteString(string(v))
34.         checkErr(err)
35.         fmt.Println("written: ", written)
```

```
36.     }
37.
        readFile(path) // NOTE: you'll read nothing here because without Flush()
38. operation
39.
40.     // let's check how much is stored in buffer
41.     unflushSize := bufferWriter.Buffered()
42.     fmt.Println("unflushSize: ", unflushSize)
43.
44.     // write memory buffer to disk
45.     bufferWriter.Flush()
46.
47.     readFile(path) // now you can get content from file
48. }
```

- xml
- json

xml 序列化和反序列化

这一节课将介绍 xml 的序列化和反序列化。

首先看下 xml 标签常见用法：

- `xml:"xxx,omitempty"` 代表如果这个字段值为空，则序列化时忽略该字段
- `xml:"xxx,attr"` 代表该字段为 xml 标签的属性说明
- `xml:"-"` 代表序列化时忽略该字段
- `xml:"a>b>c"` 代表 xml 标签嵌套模式

以下例子，演示了

- xml 序列化，包含 xml 标签的不同用法
- 写 xml 文件
- 读 xml 文件
- xml 反序列化

```
1.  package main
2.
3.  import (
4.      "encoding/xml"
5.      "fmt"
6.      "io/ioutil"
7.  )
8.
9.  type Student struct {
10.     Name    string `xml:"name"`               // xml 标签
11.     Address string `xml:"address,omitempty"` // 如果该字段为空就过滤掉
12.     Hobby   string `xml:"-"`                  // 进行 xml 序列化的时候忽略该字段
13.     Father  string `xml:"parent>father"`      // xml 标签嵌套模式
14.     Mother  string `xml:"parent>mother"`      // xml 标签嵌套模式
15.     Note    string `xml:"note,attr"`          // xml 标签属性
16. }
17.
18. func checkErr(err error) {
19.     if err != nil {
20.         panic(err)
21.     }
22. }
23.
```

```xml
func main() {
    stu1 := Student{
        Name:  "haha",
        Hobby: "basketball",
    }

    // data, _ := xml.Marshal(s)
    // fmt.Println(string(data))

    // xml 序列化
    newData, err := xml.MarshalIndent(stu1, " ", "    ")
    checkErr(err)
    fmt.Println(string(newData))

    // 写 xml 文件
    err = ioutil.WriteFile("stu.xml", newData, 0644)
    checkErr(err)

    // 读 xml 文件
    content, err := ioutil.ReadFile("stu.xml")
    stu2 := &Student{}

    // xml 反序列化
    err = xml.Unmarshal(content, stu2) // 注意：第二个参数必须是指针
    checkErr(err)

    fmt.Printf("stu2: %#v\n", stu2)
}
```

JSON 序列化和反序列化

上一节我们介绍了 `xml` 的使用，其实对于数据的序列化和反序列化还有一种更为常见的方式，那就是 JSON，尤其是在 http, rpc 的微服务调用中。

基础语法

在 Go 中我们主要使用官方的 encoding/json 包对 JSON 数据进行序列化和反序列化，主要使用方法有:

- 序列化:

```
1.  func Marshal(v interface{}) ([]byte, error)
```

- 反序列化:

```
1.  func Unmarshal(data []byte, v interface{}) error
```

简单例子:

```
1.  package main
2.
3.  import (
4.      "encoding/json"
5.      "fmt"
6.  )
7.
8.  func main() {
9.      var (
10.         data  = `1`
11.         value int
12.     )
13.
14.     err1 := json.Unmarshal([]byte(data), &value)
15.
16.     fmt.Println("Unmarshal error is:", err1)
17.     fmt.Printf("Unmarshal value is: %T, %d \n", value, value)
18.
19.     value2, err2 := json.Marshal(value)
20.
```

```
    fmt.Println("Marshal error is:", err2)
    fmt.Printf("Marshal value is: %s \n", string(value2))
}
```

当我们运行代码的时候可以得到如下输出结果：

```
Unmarshal error is: <nil>
Unmarshal value is: int, 1
Marshal error is: <nil>
Marshal value is: 1
```

在这个例子中，我们使用 `Unmarshal` 和 `Marshal` 将一个整数的 JSON 二进制转化为 go `int` 数据。

> 注意：在实际应用中，我们在序列化和反序列化的时候，需要检查函数返回的 err，如果 err 不为空，表示数据转化失败。

例如：我们把上面例子中 `value` 类型由 `int` 修改为 `string` 后再次运行代码，你将得到 `Unmarshal error is: json: cannot unmarshal number into Go value of type string` 的错误提醒。

数据对应关系

JSON 和 Go 数据类型对照表：

类型	JSON	Go
bool	true, false	true, false
string	“a”	string(“a”)
整数	1	int(1), int32(1), int64(1) …
浮点数	3.14	float32(3.14), float64(3.14) …
数组	[1,2]	[2]int{1,2}, []int{1, 2}
对象 Object	{“a”: “b”}	map[string]string, struct
未知类型	…	interface{}

例如：

```
package main

import (
    "encoding/json"
```

```
    "fmt"
)

func main() {
    var (
        d1 = `false`
        v1 bool
    )

    json.Unmarshal([]byte(d1), &v1)
    printHelper("d1", v1)

    var (
        d2 = `2`
        v2 int
    )

    json.Unmarshal([]byte(d2), &v2)
    printHelper("d2", v2)

    var (
        d3 = `3.14`
        v3 float32
    )

    json.Unmarshal([]byte(d3), &v3)
    printHelper("d3", v3)

    var (
        d4 = `[1,2]`
        v4 []int
    )

    json.Unmarshal([]byte(d4), &v4)
    printHelper("d4", v4)

    var (
        d5 = `{"a": "b"}`
        v5 map[string]string
        v6 interface{}
    )

```

```
    json.Unmarshal([]byte(d5), &v5)
    printHelper("d5", v5)

    json.Unmarshal([]byte(d5), &v6)
    printHelper("d5(interface{})", v6)
}

func printHelper(name string, value interface{}) {
    fmt.Printf("%s Unmarshal value is: %T, %v \n", name, value, value)
}
```

运行代码我们可以得到如下输出结果：

```
d1 Unmarshal value is: bool, false
d2 Unmarshal value is: int, 2
d3 Unmarshal value is: float32, 3.14
d4 Unmarshal value is: []int, [1 2]
d5 Unmarshal value is: map[string]string, map[a:b]
d5(interface{}) Unmarshal value is: map[string]interface {}, map[a:b]
```

自定义数据类型

除了使用上面基础数据外，对于那些比较复杂的数据集合（Object），我们还可以使用自定义数据类型 struct 来转化。

- Go 中关于 JSON 转化字段名的对应语法为：

```
Field int `json:"myName"`
```

- 如果我们想忽略那些空值的字段，我们可以使用 `omitempty` 选项：

```
Field int `json:"myName,omitempty"`
```

- 如果我们想忽略特定字段：

```
Field int `json:"-"`
```

组合示例：

```
type A struct {
    A int     `json:"k"`
```

```
    B string  `json:"b,omitempty"`
    C float64 `json:"-"`
}
```

实际例子练习

假如我们有这样一段 JSON 数据，它表示一个学生的考试成绩，下面我们就来看看在 Go 中如何序列化和反序列化。

- 数据准备：

```
# data.json
{
    "id": 1,
    "name": "小红",
    "results": [
        {
            "name": "语文",
            "score": 90
        },
        {
            "name": "数学",
            "score": 100
        }
    ]
}
```

- 反序列化：

```
package main

import (
    "encoding/json"
    "fmt"
    "io/ioutil"
)

type Result struct {
    Name  string  `json:"name"`
    Score float64 `json:"score"`
}

```

```
type Student struct {
    Id int `json:"id"`

    Name    string   `json:"name"`
    Results []Result `json:"results"`
}

func main() {
    dat, _ := ioutil.ReadFile("data.json")

    var s Student
    json.Unmarshal(dat, &s)
    fmt.Printf("Student's result is: %v\n", s)
}
```

运行代码输出结果为：

```
Student's result is: {1 小红 [{语文 90} {数学 100}]}
```

- 序列化：

```
package main

import (
    "encoding/json"
    "io/ioutil"
)

type Result struct {
    Name  string  `json:"name"`
    Score float64 `json:"score"`
}

type Student struct {
    Id int `json:"id"`

    Name    string   `json:"name"`
    Results []Result `json:"results"`
}

func main() {
    s := Student{
```

```
        Id:   1,
        Name: "小红",
        Results: []Result{
            Result{
                Name:  "语文",
                Score: 90,
            },
            Result{
                Name:  "数学",
                Score: 100,
            },
        },
    }

    dat, _ := json.Marshal(s)
    ioutil.WriteFile("data2.json", dat, 0755)
}
```

当我们运行代码后，打开 `data2.json` 文件，将看到如下内容：

```
{
    "id": 1,
    "name": "小红",
    "results": [
        {
            "name": "语文",
            "score": 90
        },
        {
            "name": "数学",
            "score": 100
        }
    ]
}
```

- socket
- http
- websocket

Socket

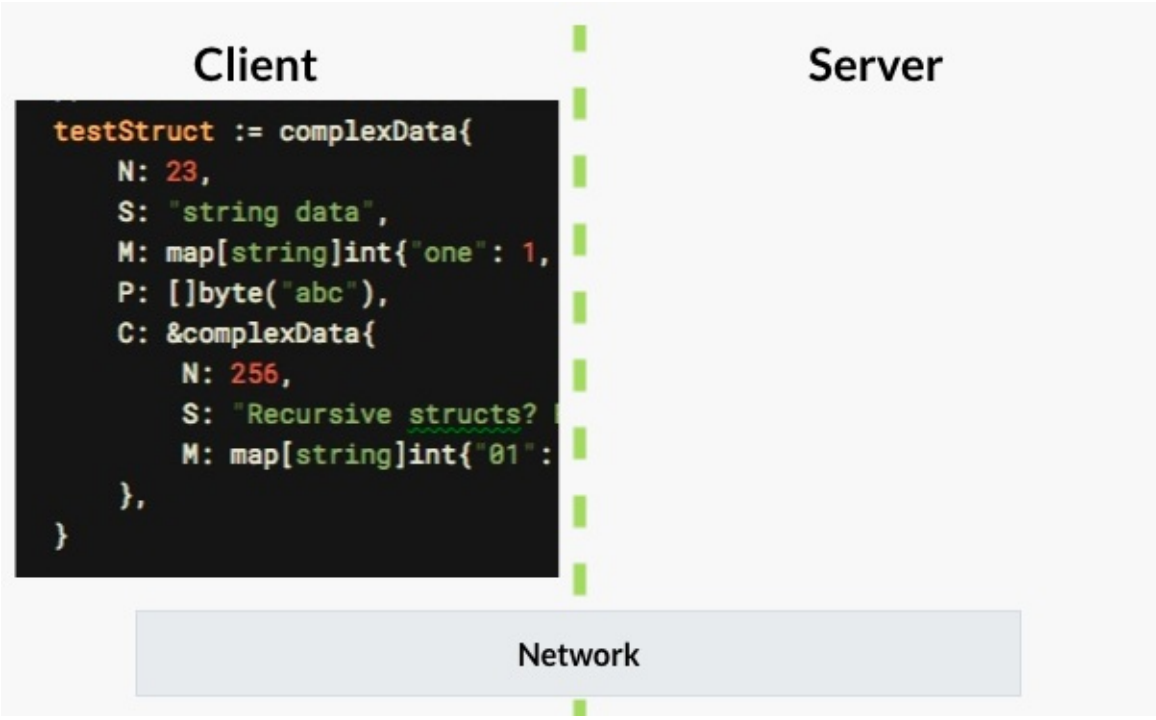
Socket是网络编程的一个抽象概念，通常我们用一个 Socket 表示 “打开了一个网络连接”，在 Go 中主要使用 net 包。

使用 net 的 `func Dial(network, address string) (Conn, error)` 函数就可轻松建立一个 Socket 连接。Socket 创建成功后，我们可以对其进行 I/O 操作，最后不要记得对其进行关闭操作。

本章将从 TCP， UDP， Unix 入手，带领大家全面了解 Socket 在 Go 中的应用。

基本知识

Socket 连接又分为客户端和服务端，如图：



核心步骤包括：

- 创建连接：

```
1.  Dial(network, address string) (Conn, error)
```

注意，这里的 network 可以为：

```
1.  "tcp", "tcp4", "tcp6"
2.  "udp", "udp4", "udp6"
```

```
"ip", "ip4", "ip6"
"unix", "unixgram", "unixpacket"
```

- 通过连接发送数据：

```
conn.Write([]byte("GET / HTTP/1.0\r\n\r\n"))
```

- 通过连接读取数据：

```
buf := make([]byte, 256)
conn.Read(buf)
```

- 关闭连接：

```
conn.Close()
```

> 注意：`conn` 是一个 IO 对象，我们主要使用 IO 相关的帮助方法来进行读写操作。

实际例子之 google 首页访问

```
package main

import (
    "fmt"
    "io/ioutil"
    "log"
    "net"
)

func main() {
    // 尝试与 google.com:80 建立 tcp 连接
    conn, err := net.Dial("tcp", "google.com:80")
    if err != nil {
        log.Fatal(err)
    }

    defer conn.Close() // 退出关闭连接

    // 通过连接发送 GET 请求，访问首页
```

```
    _, err = fmt.Fprintf(conn, "GET / HTTP/1.0\r\n\r\n")
    if err != nil {
        log.Fatal(err)
    }

    dat, err := ioutil.ReadAll(conn)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println(string(dat))
}
```

当运行代码，可以得到 `google.com` 的首页内容，如下：

```
HTTP/1.0 200 OK
Date: Tue, 05 Jun 2018 14:45:30 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
P3P: CP="This is not a P3P policy! See g.co/p3phelp for more info."
Server: gws
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN
Set-Cookie: 1P_JAR=2018-06-05-14; expires=Thu, 05-Jul-2018 14:45:30 GMT;
path=/; domain=.google.com
Set-Cookie: NID=131=mqkJocXSsDCD6zdcMyc12DCUqt3X19HIoS0HGTsAzsiuvFx56rBsliga5Uj22
FvzCLdYYY9tT0KYtVv; expires=Wed, 05-Dec-2018 14:45:30 GMT; path=/; domain=.google
Accept-Ranges: none
Vary: Accept-Encoding
....
```

说明：google.com 网站后端是一个 HTTP server，因为 HTTP 建立在 TCP 协议基础上，所以我们这里可以使用 TCP 协议来进行访问。

TCP 操作

在这个例子中，我们先使用 net 包创建一个 TCP Server ，然后尝试连接 Server，最后再通过客户端发送 `hello` 到 Server，同时 Server 响应 `word` 。

我们来看完整例子：

- server/main.go

```
1.  package main
2.
3.  import (
4.      "bufio"
5.      "fmt"
6.      "log"
7.      "net"
8.  )
9.
10. func main() {
11.     l, err := net.Listen("tcp", "127.0.0.1:8888")
12.     if err != nil {
13.         log.Fatal(err)
14.     }
15.     log.Printf("Start server with: %s", l.Addr())
16.
17.     defer l.Close()
18.
19.     for {
20.         conn, err := l.Accept()
21.         if err != nil {
22.             log.Fatal(err)
23.         }
24.
25.         go handleConnection(conn)
26.     }
27. }
28.
29. func handleConnection(conn net.Conn) {
30.     reader := bufio.NewReader(conn)
31.
32.     for {
33.         dat, _, err := reader.ReadLine()
34.         if err != nil {
35.             log.Println(err)
36.             return
37.         }
38.
39.         fmt.Println("client:", string(dat))
40.
```

```
41.         _, err = conn.Write([]byte("word\n"))
42.         if err != nil {
43.             log.Println(err)
44.             return
45.         }
46.     }
47. }
```

注意：

```
1. 1. 通过 `net.Listen("tcp", "127.0.0.1:8888")` 新建一个 TCP Server。
2. 2. 通过 `l.Accept()` 获取创建的连接。
3. 3. 通过 `go handleConnection(c)` 新建的 goroutine 来处理连接。
```

- client/main.go

```
1.
2.  package main
3.
4.  import (
5.      "bufio"
6.      "fmt"
7.      "log"
8.      "net"
9.      "time"
10. )
11.
12. func main() {
13.     conn, err := net.Dial("tcp", "127.0.0.1:8888")
14.     if err != nil {
15.         log.Fatal(err)
16.     }
17.
18.     defer conn.Close()
19.
20.     reader := bufio.NewReader(conn)
21.     for {
22.         _, err := conn.Write([]byte("hello\n"))
23.         if err != nil {
24.             log.Fatal(err)
25.         }
26.
```

```
27.         dat, _, err := reader.ReadLine()
28.         if err != nil {
29.             log.Fatal(err)
30.         }
31.         fmt.Println("sever:", string(dat))
32.
33.         time.Sleep(5 * time.Second)
34.     }
35. }
```

注意：

```
1. 1. 通过 `net.DialTCP("tcp", nil, addr)` 尝试创建到 TCP Sever 的连接。
2. 2. 通过 `conn.Write([]byte("hello\n"))` 向服务端发送数据。
3. 3. 通过 `reader.ReadLine()` 读取服务端响应数据。
```

当我们运行代码的时候，可以在终端看到如下输入：

```
1. go run server/main.go
2.
3. 2018/06/08 08:12:23 Start server with: 127.0.0.1:8888
4. client: hello
5. client: hello
```

```
1. go run client/main.go
2.
3. 2018/06/08 08:12:23 Start server with: 127.0.0.1:8888
4. sever: word
5. sever: word
```

UDP 操作

UDP 相较于 TCP 简单的多，它具有以下特点：

- 无连接的
- 要求系统资源较少
- UDP 程序结构较简单
- 基于数据报模式(UDP)
- UDP 可能丢包
- UDP 不保证数据顺序性

下面我们通过一个统计服务在线人数的例子来了解它：

- server/main.go

```
1.  package main
2.
3.  import (
4.      "log"
5.      "net"
6.      "time"
7.  )
8.
9.  func main() {
10.     // listen to incoming udp packets
11.     pc, err := net.ListenPacket("udp", "127.0.0.1:8888")
12.     if err != nil {
13.         log.Fatal(err)
14.     }
15.
16.     log.Printf("Start server with: %s", pc.LocalAddr())
17.
18.     defer pc.Close()
19.
20.     clients := make([]net.Addr, 0)
21.
22.     go func() {
23.         for {
24.             for _, addr := range clients {
25.                 _, err := pc.WriteTo([]byte("pong\n"), addr)
26.                 if err != nil {
27.                     log.Println(err)
28.                 }
29.             }
30.
31.             time.Sleep(5 * time.Second)
32.         }
33.     }()
34.
35.     for {
36.         buf := make([]byte, 256)
37.         n, addr, err := pc.ReadFrom(buf)
38.         if err != nil {
```

```
            log.Println(err)
            continue
        }

        clients = append(clients, addr)

        log.Println(string(buf[0:n]))
        log.Println(addr.String(), "connecting...", len(clients), "connected")
    }
}
```

注意：

1. 监听本地 UDP `127.0.0.1:8888`。
2. 使用 `pc.ReadFrom(buf)` 方法读取客户端发送的消息。
3. 使用 `clients` 来保存所有连上的客户端连接。
4. 通过 `pc.WriteTo([]byte("pong\n"), addr)` 向所有客户端发送消息。

- client/main.go

```
package main

import (
    "bufio"
    "log"
    "net"
)

func main() {
    conn, err := net.Dial("udp", "127.0.0.1:8888")
    if err != nil {
        log.Fatal(err)
    }

    defer conn.Close()

    _, err = conn.Write([]byte("ping..."))
    if err != nil {
        log.Fatal(err)
    }

    reader := bufio.NewReader(conn)
```

```
    for {
        dat, _, err := reader.ReadLine()
        if err != nil {
            log.Fatal(err)
        }

        log.Println(string(dat))
    }
}
```

当我运行代码可以得到如下输出：

```
# 执行命令
go run server/main.go

# 输出
2018/06/08 14:36:13 Start server with: 127.0.0.1:8888
2018/06/08 14:36:15 ping...
2018/06/08 14:36:15 127.0.0.1:61790 connecting... 1 connected
2018/06/08 14:36:18 ping...
2018/06/08 14:36:18 127.0.0.1:59989 connecting... 2 connected
```

```
# 启动 client1
go run client/main.go

# 输出
2018/06/08 14:36:18 pong
2018/06/08 14:36:23 pong
```

```
# 启动 client2
go run client/main.go

# 输出
2018/06/08 14:37:58 pong
2018/06/08 14:38:03 pong
```

Unix 操作

Unix 和 TCP 很相似，只不过监听的地址是一个 Socket 文件，例如：

```
l, err := net.Listen("unix", "/tmp/echo.sock")
```

下面我们就通过一个实际的例子来练习：

- server/main.go

```
1.  package main
2.
3.  import (
4.      "log"
5.      "net"
6.  )
7.
8.  func main() {
9.      l, err := net.Listen("unix", "/tmp/unix.sock")
10.     if err != nil {
11.         log.Fatal("listen error:", err)
12.     }
13.
14.     for {
15.         conn, err := l.Accept()
16.         if err != nil {
17.             log.Fatal("accept error:", err)
18.         }
19.
20.         go helloServer(conn)
21.     }
22. }
23.
24. func helloServer(c net.Conn) {
25.     for {
26.         buf := make([]byte, 512)
27.         nr, err := c.Read(buf)
28.         if err != nil {
29.             return
30.         }
31.
32.         data := buf[0:nr]
33.         log.Println(string(data))
34.
35.         _, err = c.Write([]byte("hello"))
36.         if err != nil {
37.             log.Fatal("Write: ", err)
```

```
        }
    }
}
```

说明:

```
- 使用 `net.Listen("unix", "/tmp/unix.sock")` 启动一个 Server。
- 使用 `conn, err := l.Accept()` 来接受客户端的连接。
- 使用 `go helloServer(conn)` 来处理客户端连接，并读取客户端发送的数据 `hi` 并返回 `hello`。
```

- client/main.go

```
package main

import (
    "io"
    "log"
    "net"
    "time"
)

func reader(r io.Reader) {
    buf := make([]byte, 512)
    for {
        n, err := r.Read(buf[:])
        if err != nil {
            return
        }
        log.Println(string(buf[0:n]))
    }
}

func main() {
    c, err := net.Dial("unix", "/tmp/unix.sock")
    if err != nil {
        log.Fatal(err)
    }
    defer c.Close()

    go reader(c)
    for {
```

```
30.         _, err := c.Write([]byte("hi"))
31.         if err != nil {
32.             log.Fatal("write error:", err)
33.             break
34.         }
35.         time.Sleep(3 * time.Second)
36.     }
37. }
```

注意：

```
1. - 使用 `c, err := net.Dial("unix", "/tmp/unix.sock")` 来连接服务端。
2. - 使用 `c.Write([]byte("hi"))` 向服务端发送 `hi` 消息。
3. - 使用 `r.Read(buf)` 读取客户端发送的消息。
```

当运行代码可以得到如下输出：

```
1. # go run server/main.go
2.
3. 2018/06/09 20:42:14 hi
4. 2018/06/09 20:42:16 hi
5. 2018/06/09 20:42:17 hi
```

```
1. # go run client/main.go
2.
3. 2018/06/09 20:41:47 hello
4. 2018/06/09 20:41:50 hello
5. 2018/06/09 20:41:53 hello
```

WebSocket

- MySQL
- MongoDB

项目工程