Go 快速入门

分享内容

- Go语言介绍
- Go基础
- 流程控制
- 错误处理
- 常用包
- 小案例
- QA

一、 Go语言介绍

1.1 Go语言介绍

- Go 即 Golang，是 Google 公司 2009 年 11 月正式对外公开的一门编程语言。
- 根据 Go 语言开发者自述，近 10 多年，从单机时代的 C 语言到现在互联网时代的 Java，都没有令人满意的开发语言，而 C++往往给人的感觉是，花了 100%的经历，却只有 60%的开发效率，产出比太低，Java 和 C#的哲学又来源于 C++。
- 并且，随着硬件的不断升级，这些 语言不能充分的利用硬件及 CPU。
- 因此，一门高效、简洁、开源的语言诞生了。
- Go 语言不仅拥有静态编译语言的安全和高性能，而且又达到了动态语言开发速度和易 维护性。
- 有人形容 Go 语言：Go = C + Python , 说明 Go 语言既有 C 语言程序的运行速度，又能达到 Python 语言的快速开发。
- Go **语言是非常有潜力的语言**，是因为它的应用场景是目前互联网非常热门的几个领域
- 比如 WEB 开发、区块链开发、大型游戏服务端开发、分布式/云计算开发。
- 国内比较知名的B 站就是用 Go 语言开发的，像 Goggle、阿里、京东、百度、腾讯、小米、360 的很多应用也是使用 Go 语言开发的。

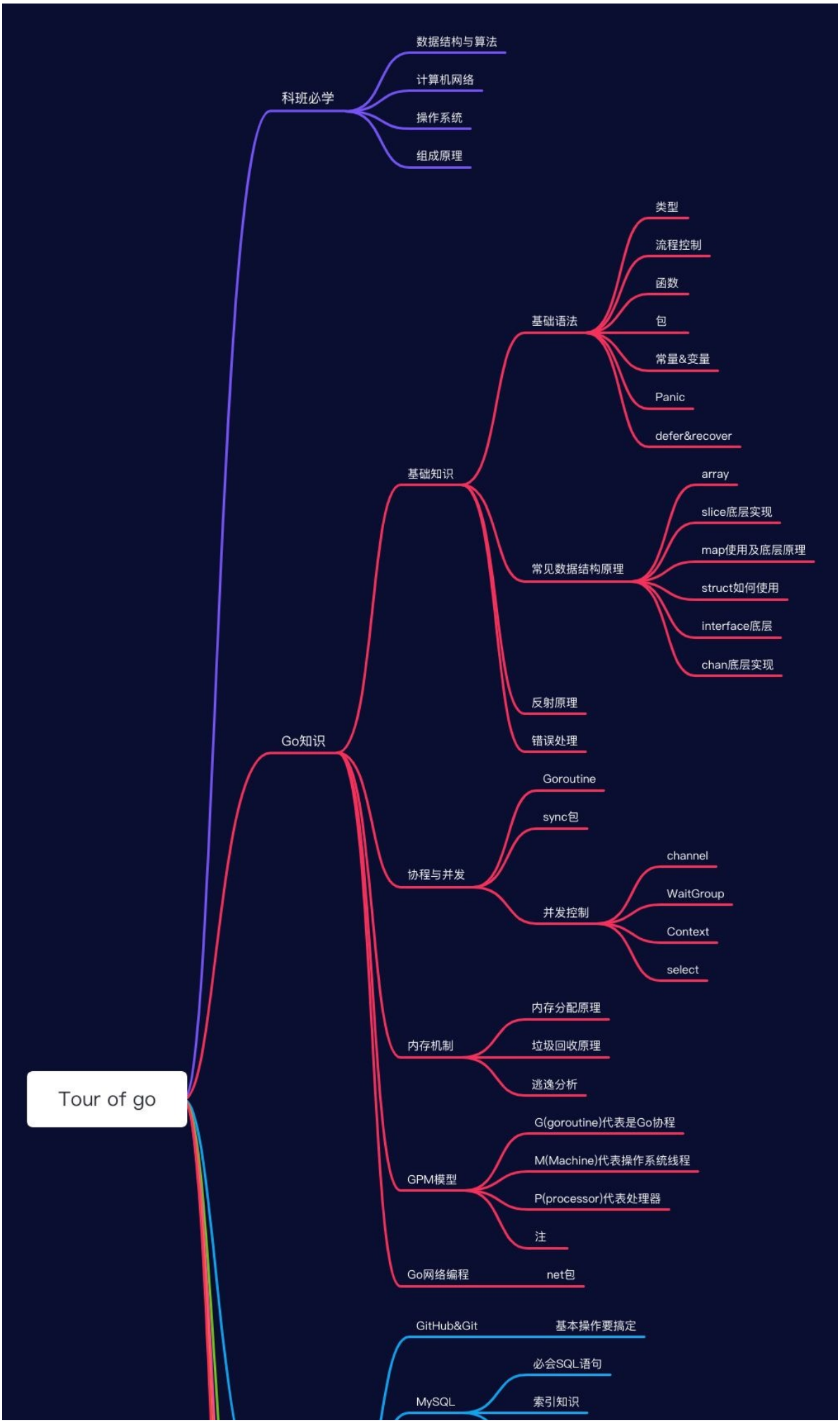
1.2 Go语言解决的问题

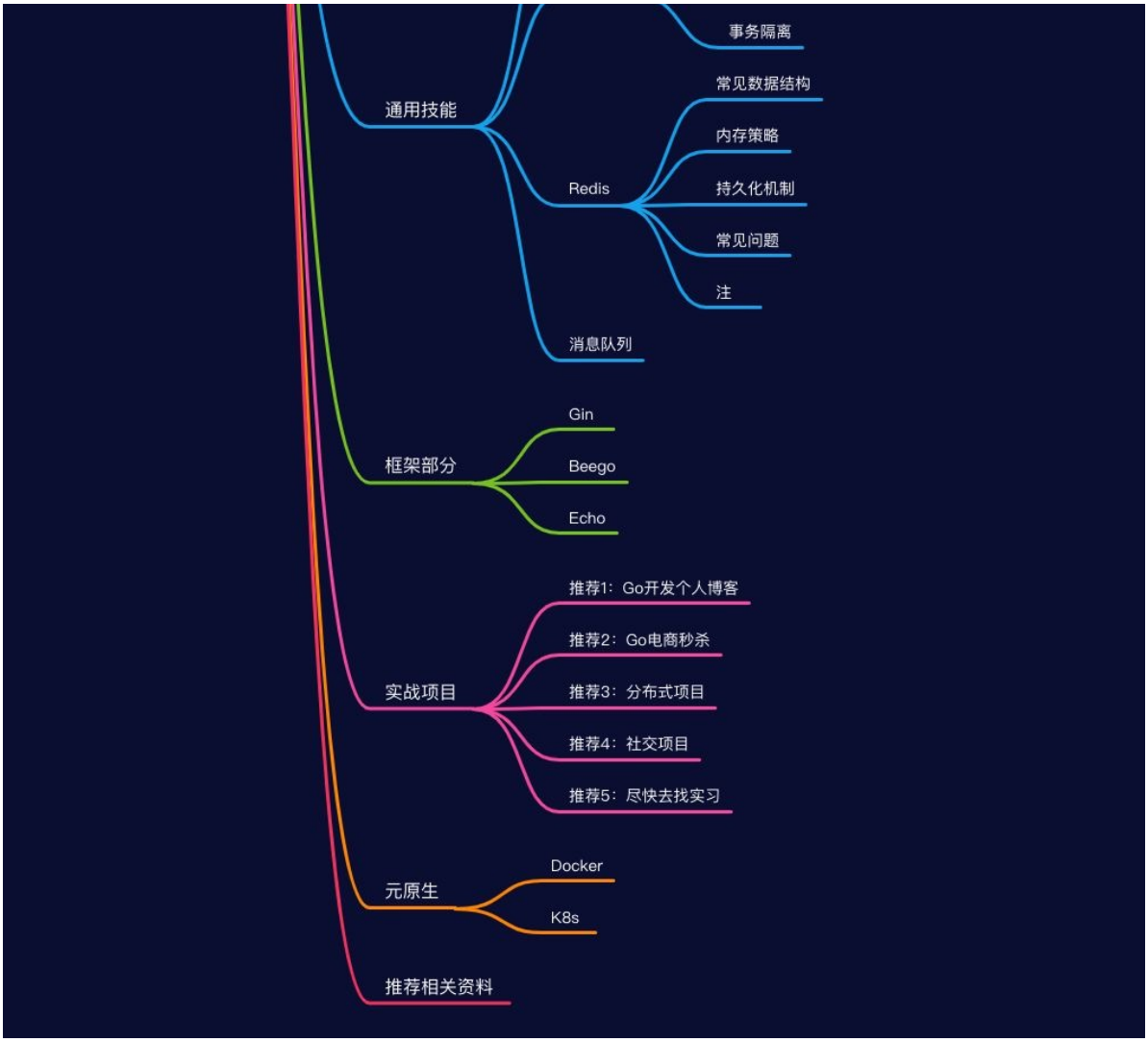
- 多核硬件架构；
- 超大规模分布式计算集群；
- Web 开发模式导致的前所未有的开发规模和更新速度。

1.3 Go语言应用领域

- 服务端编程
- 中间件
- 网络编程
- 云计算
- 区块链开发
- 分布式系统

1.4 Go语言学习脑图


Tour of go
科班必学
数据结构与算法
计算机网络
操作系统
组成原理
Go知识
基础知识
基础语法
类型
流程控制
函数
包
常量&变量
Panic
defer&recover
常见数据结构原理
array
slice底层实现
map使用及底层原理
struct如何使用
interface底层
chan底层实现
反射原理
错误处理
协程与并发
Goroutine
sync包
并发控制
channel
WaitGroup
Context
select
内存机制
内存分配原理
垃圾回收原理
逃逸分析
GPM模型
G(goroutine)代表是Go协程
M(Machine)代表操作系统线程
P(processor)代表处理器
注
Go网络编程
net包
GitHub&Git
基本操作要搞定
MySQL
必会SQL语句
索引知识



二、Go基础

2.1 基本数据类型

- 布尔型

  bool

- 整数型

  Int8、int16、int32、int64、uint8、uint16、uint32、uint64

- 浮点型

  float32、float64、complex64、complex128

- 其他数字类型

  byte、int、uint、rune、uintptr

- 字符串

  string

2.2 变量的定义

声明变量不赋值，将使用默认值, int 默认值为0

```
var a int
```

声明变量并赋值

```
var b int = 10
```

简化var关键词的写法

```
c := 100
```

根据值自动判断变量类型

```
var d = 10
```

一次性声明多个类型相同的变量

```
var n1, n2, n3 int
```

一次性声明多个变量，且类型不同

```
var age, nickName, addr = 10, "jerry", "北京"
```

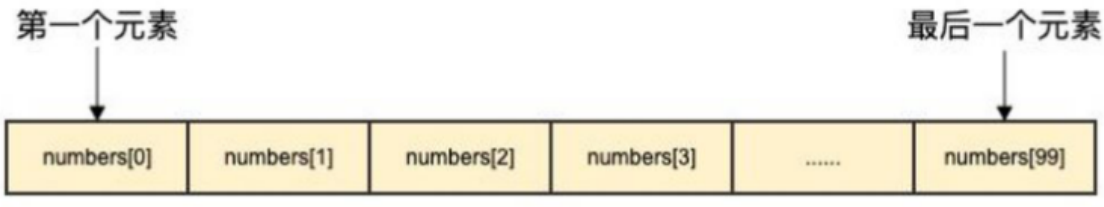
一次性声明多个变量，简写var关键词

```
email, phone, pwd := "1314@qq.com", 9580, "123456"
```

声明多个全局变量时可以直接用一个var包裹

```
var (
    ok_status = "ok"
    ok_code = 200
)
```

2.3 数组



数组在定义时就必须设置大小；声明后长度不可变，元素的值可以变；注意：在函数中传递数组时，进行的是值传递；要和切片区分开。

```go
var arr [6]int
arr[0] = 5
fmt.Println("arr=", arr)

var arr1 [3]int = [3]int{1, 2, 3}
fmt.Println("arr1=", arr1)

var arr2 = [3]int{4, 5, 6}
fmt.Println("arr2=", arr2)

arr3 := [3]int{7, 8, 9}
fmt.Println("arr3=", arr3)
```

这里的...是固定写法；最终大小根据后面设置的初始值个数决定+

```go
var arr3 = [...]int{7, 8, 9}
fmt.Println("arr3=", arr3)

var arr4 = [...]int{1: 10, 0: 11, 2: 12}
fmt.Println("arr3=", arr4)
```

二维数组

```go
var arr [6][6]int
arr[1][2] = 1
```

2.4 切片

golang中的切片可以简单的认为就是一个动态扩容的数组。切片在底层分配的内存是连续的。切片的定义和数组的很像，不指定大小

```go
var slice []int
```

注意没有赋初始值的切片，必须通过make方法申请空间

```go
slice := make([]int, 0)
```

赋初始值的切片

```go
var slice2 []string = []string{"tom","jerry"}
```

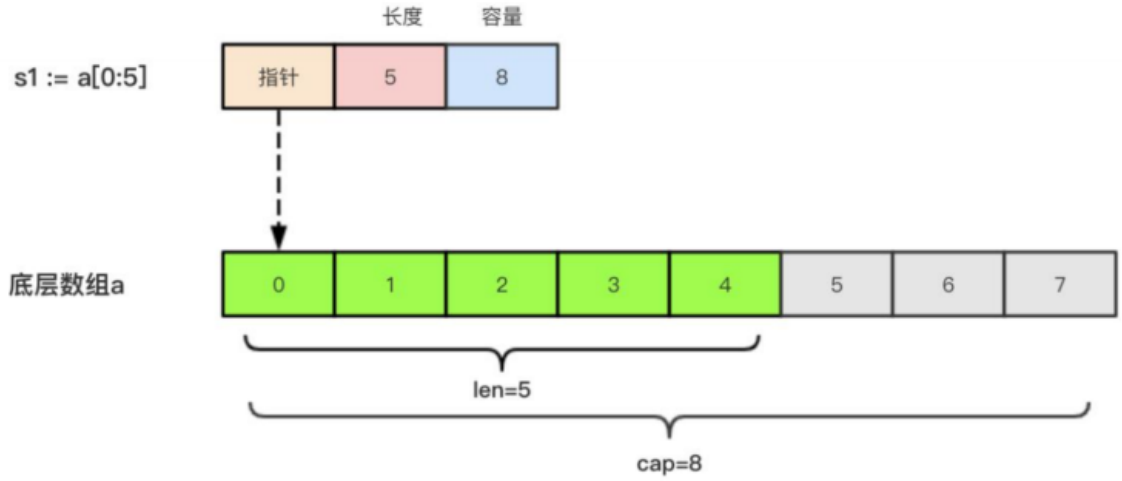
从一个数组或者是切片中截取一段数据

```go
arr := [...]int{1, 2, 3, 4}
```

[1:3] 表示引用arr数组下标1到3的值（不包含3）

```go
slice := arr[1:3]
```

举个例子，现在有一个数组 a := [8]int{0, 1, 2, 3, 4, 5, 6, 7}，切片 s1 := a[:5]，相应示意图如下。



2.5 Map

Map也是Go的一种数据类型，用于记录键值间的映射关系。使用以下代码创建一个map。声明一个map并赋值

```
var map2 = make(map[string]string, 10)
map2["s1"] = "100"
fmt.Println(map2)
```

声明的时候就赋值

```
map3 := map[string]string {
    "address":"成都",
}
fmt.Println(map3)
```

删除map中的值

```
delete(map3, "address")
```

如果想删全部的key，需要一个个的遍历来删除；或者重新开辟个空间

```
for k, _ := range map3 {
    delete(map3, k)
}
```

重新开辟个空间

```
map3 = make(map[string]string)
```

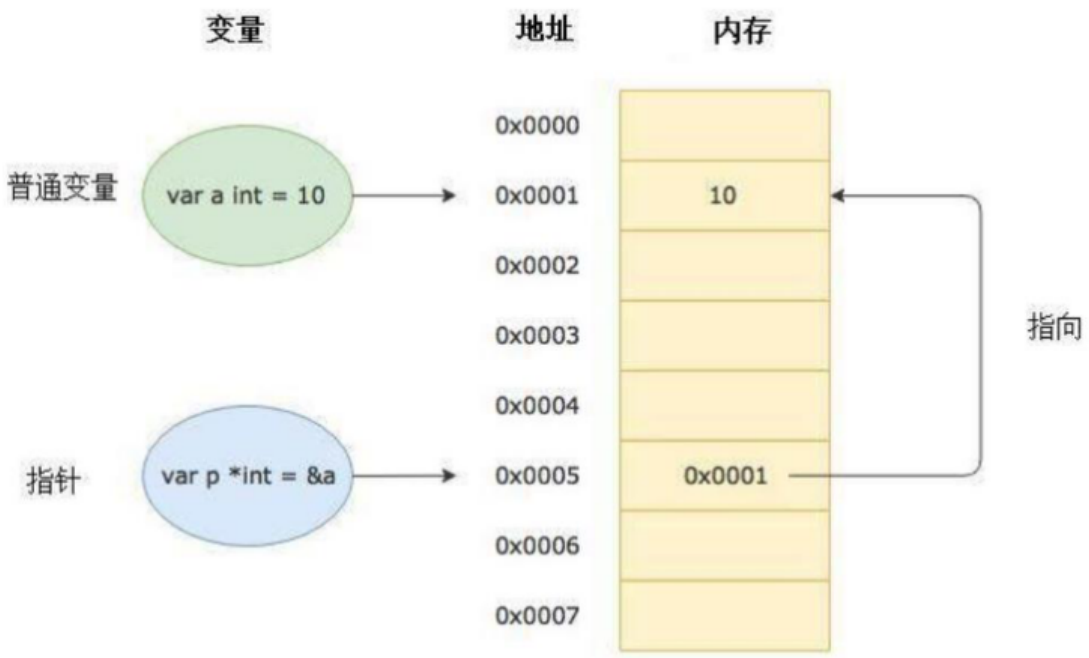
2.7 指针

指针也是一个变量，但它是一种特殊的变量，它存储的数据不是一个普通的值，而是另一个变量的内存地址。

要搞明白 Go 语言中的指针需要先知道 3 个概念：`指针地址、指针类型、指针取值`

- `指针地址（&a）`
- `指针取值（*&a）`
- `指针类型（&a）` —> `*int` 改变数据传指针

```go
package main
import "fmt"
func main() {
    var a = 10
    fmt.Printf("%d \n",&a)      // &a 指针地址 (824633761976)
    fmt.Printf("%d \n",*&a)     // *&a 指针取值 (10)
    fmt.Printf("%T \n",&a)      // %T 指针类型 (*int )
}
```



指针类型赋值

```go
var ap *int
a := 12
ap = &a
fmt.Println(*ap)
```

思考打印结果？

```go
a1 := 1
a2 := a1
a3 := &a1

a1 = 2

fmt.Println(a1,a2,*a3)
fmt.Println(&a1,&a2,a3)
```

以下两种情况，通常优先选用指针：

- 把结构体作为参数传递时。因为值传递会耗费更多内存。
- 声明某类型的方法时。传递指针后，方法/函数可以直接修改指针所指向的值。

思考打印结果？

```
func increment(i *int) {
    *i++
}
func main() {
    i := 10
    increment(&i)
    fmt.Println(i) }
}
```

```
func increment(i int) {
    i++
}
func main() {
    i := 10
    increment(i)
    fmt.Println(i) }
}
```

2.8 函数

在golang中整个程序只能有一个main函数，定义的package类似Java中的类名称，调用其他文件中的方法时需要引入package，然后通过package名称.方法名称；在golang中方法名首字母小写表示只能当前包中自己访问；首字母大小表示其他包中可以调用。

函数的返回值也可以在函数中提前定义

```
func add(a int, b int) (c int) {
    c = a + b
    return
}

func main() {
    fmt.Println(add(2, 1)) //=> 3
}
```

也可以一次返回多个变量

```
func add(a int, b int) (int, string) {
    c := a + b
    return c, "successfully added”
 }

func main() {
    sum, message := add(2, 1)
    fmt.Println(message)
    fmt.Println(sum)
}
```

2.9 Struct 结构体

结构体包含不同类型的字段，可用来对数据进行分组。例如，如果我们要对Person类型的数据进行分组，那么可以定义一个人的各种属性，包括姓名，年龄，性别等。

```
type person struct {
    name string
    age int
    gender string
}

//方法 1: 指定参数和值
p := person{name: "Bob", age: 42, gender: "Male"}

//方法 2: 仅指定值
p = person{"Bob", 42, "Male"}

//可以使用.来获取一个对象的参数
p.name
//=> Bob
p.age
//=> 42
p.gender
//=> Male

//也可以通过结构体的指针对象来获取参数。
pp = &person{name: "Bob", age: 42, gender: "Male"}

pp.name
//=> Bob

```

2.10 方法

方法是一种带有接收器的函数。接收器可以是一个值或指针。我们可以把刚刚创建的Person类型作为接收器来创建方法：

```
package main
import "fmt"

// 定义结构体
type person struct {
    name   string
    age    int
    gender string
}

// 定义方法
func (p *person) describe() {
    fmt.Printf("%v is %v years old.", p.name, p.age)
}
func (p *person) setAge(age int) {
    p.age = age
}
```

```go
func (p person) setName(name string) {
    p.name = name
}
func main() {
    pp := &person{name: "Bob", age: 42, gender: "Male"}

    // 使用 . 来调用方法
    pp.describe()
    // => Bob is 42 years old
    pp.setAge(45)
    fmt.Println(pp.age)
    //=> 45
    pp.setName("Hari")
    fmt.Println(pp.name)
    //=> Bob
}
```

三、流程控制

3.1 条件语句

```go
//if-else语句用于条件判断
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")
}

//在上面代码赋值后，是否能打印出来?
fmt.Println(num)

// switch-case用于组织多个条件语句
i := 2
switch i {
    case 1:
    fmt.Println("one")
    case 2:
    fmt.Println("two")
    default:
    fmt.Println("none")
}
```

3.2 循环语句

```go
// 普通的使用
for i := 1; i<10; i++ {
    fmt.Printf("这是第%v个\n", i)
}

// 只写条件：实现类似while的效果
```

```go
var j = 10;
for j>0 {
    fmt.Printf("this is %v\n", j)
    j--
}

// 什么条件都不写，有内部进行判断跳出循环的时机
for {               // 这里等价于 for ; ; {}
    fmt.Printf("this is %v\n", j)
    if(j<1){
        break   //跳出循环
    }
}

// 遍历字符串中的字符
var str string = "ancdefg"

// 使用for-range遍历字符串
for index, val := range str {
    fmt.Printf("%d %c ", index, val)
}

//遍历map
var amap = make(map[string]int)
amap["a"] = 1
amap["b"] = 2
// 使用for-range遍历map
for key, val := range amap {
    fmt.Printf("%s %d ", key, val)
}

```

四、错误处理

4.1 Error处理

大部分的内置包或者外部包，都有自己的报错处理机制。因此我们使用的任何函数可能报错，这些报错都不应该被忽略，而是在调用函数的地方，优雅地处理报错。

```go
package main

import (
    "fmt"
    "net/http"
)

func main(){
    resp, err := http.Get("http://example.com/")
    if err != nil {
        fmt.Println(err)  //打印报错
        return
    }
    fmt.Println(resp)
```

```
}
```

```
package main

import (
    "fmt"
    "errors"
)

func Increment(n int) (int, error) {
    if n < 0 {
        // return error object
        return nil, errors.New(“math: cannot process negative number”)  //自定义报错
    }
    return (n + 1), nil
}

func main() {
    num := 5

    if inc, err := Increment(num); err != nil {
        fmt.Printf("Failed Number: %v, error message: %v", num, err)
    }else {
        fmt.Printf("Incremented Number: %v", inc)
    }
}
```

4.2 Panic异常

当程序在运行过程中，突然遇到了未处理的报错，就会导致panic。在Go中，更推荐使用error对象，而不是panic来处理异常。发生panic后，程序会停止运行，但会运行defer语句代码，Defer适用于 需要在函数最后执行某些操作的场景，比如关闭文件。

```
package main

import "fmt"

func main() {
    f()
    fmt.Println("Returned normally from f.")
}

func f() {
    defer func() {
        if r := recover(); r != nil {
            fmt.Println("Recovered in f", r)
        }
    }()
}

fmt.Println("Calling g.")
      g(0)
      fmt.Println("Returned normally from g.")
```

```
}

func g(i int) {
    if i > 3 {
        fmt.Println("Panicking!")
        panic(fmt.Sprintf("%v", i))
    }
    defer fmt.Println("Defer in g", i)
    fmt.Println("Printing in g", i)
    g(i + 1)
}
```

五、常用库

5.1 序列化和反序列化

序列化是指把对象转换为字节序列的过程，而反序列化是指把字节序列恢复为对象的过程；

```
// 序列化
package main

import (
  "fmt"
  "encoding/json"
)

func main(){
  mapA := map[string]int{"apple": 5, "orange": 7}
  mapB, _ := json.Marshal(mapA)
  fmt.Println(string(mapB))
}
```

```
// 反序列化
package main

import (
  "fmt"
  "encoding/json"
)

type response struct {
  PageNumber int `json:"page"`
  Fruits []string `json:"fruits"`
}

func main(){
  str := `{"page": 1, "fruits": ["apple", "peach"]}`
  res := response{}
  json.Unmarshal([]byte(str), &res)
  fmt.Println(res.PageNumber)
}
//=> 1
```

5.2 文件I/O操作

文件操作常用包：os、ioutil、bufio等

```go
//打开、读取文件
file, err :=  os.Open("/doc/addGoPath.bat)
if err != nil {
    fmt.Println("open file err=", err)
}
defer file.Close()

// NewReader 返回的buffer缓冲区大小为4096
reader := bufio.NewReader(file)
for {
    // 读到一个换行就结束
    str, err := reader.ReadString('\n')
    if err == io.EOF {
        // io.EOF表示文件末尾
        break
    }
    fmt.Println(str)
}
```

```go
//一次性读取文件内容
path := "/doc/addGoPath.bat"
// 这里返回的content是一个切片
content, err := ioutil.ReadFile(path)
if err != nil {
    fmt.Printf("read file err=%v\n", err)
    return
}

fmt.Printf("%v", string(content))
```

```go
//创建、写入文件
fileName := "/doc/demo4.txt"
// 对应参数说明：文件名称，读取操作配置，权限（仅在linux、Unix系统下生效）
file, err := os.OpenFile(fileName, os.O_WRONLY | os.O_CREATE, 0666)
if err != nil {
    fmt.Printf("read file err=%v\n", err)
    return
}
str := "hello, golang\r\n"
// 获取写缓存
writer := bufio.NewWriter(file)

for i := 0; i<5; i++ {
    // 注意这样是直接覆盖的写
    writer.WriteString(str)
}

// 由于使用的是缓存，因此需要调用刷新函数将其刷新到磁盘上去
writer.Flush()
```

5.3 命令行参数

在golang中有很多方法来处理命令行参数，简单情况下可以不使用任何库，直接使用os.Args；但是golang标准库提供了flag包来处理命令行参数；还有第三方提供的处理命令行参数的库cobra、cli。

```
//os.Args获取命令行参数
fmt.Println("命令行的参数有", len(os.Args))

for i, v := range os.Args {
    fmt.Printf("args[%v]=%v\n", i, v)
}
```

```
//  go run 024.go -pwd 23456 -port 8087;
// 使用flag工具包读取命令行信息
var pwd string
flag.StringVar(&pwd, "pwd", "123456", "这是说明")

var port int
flag.IntVar(&port, "port", 8080, "端口号，默认为8080")

// 必须调用这个方法才会将数据转换出来
flag.Parse()
fmt.Println("pwd=", pwd)
fmt.Println("port=", port)

```

5.4 HTTP

一个简单的http接口

```
package main

import (
    "fmt"
    "net/http"
)

// 请求来了后的处理方法
func handler(w http.ResponseWriter, r *http.Request) {
    res := "welcome use grpc"
    url := r.URL.Path
    if "/api" == url {
        res = "api server is ok"
    }
    w.WriteHeader(200)
    w.Write([]byte(res))
}

func main() {
    // 配置路由监听
    http.HandleFunc("/", handler)
    // 启动服务并监听8081端口
    err := http.ListenAndServe(":8081", nil)

```

```
    if err != nil {
        fmt.Println("服务启动失败")
    }
}
```

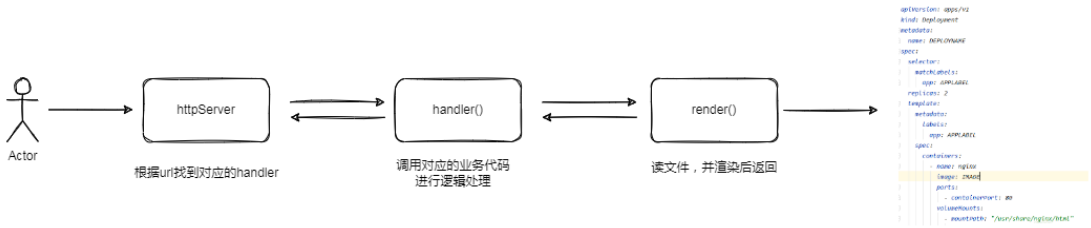
使用http包发起请求，先生成http.client -> 再生成 http.request -> 之后提交请求：client.Do(request) -> 处理返回结果，每一步的过程都可以设置一些具体的参数，下面是一个最朴素最基本的例子：

```
package main

import (
    "fmt"
    "io"
    "net/http"
    "os"
)

func main() {
    //生成client 参数为默认
    client := &http.Client{}

    //生成要访问的url
    url := "http://www.baidu.com"

    //提交请求
    reqest, err := http.NewRequest("GET", url, nil)

    if err != nil {
        panic(err)
    }
    //处理返回结果
    response, _ := client.Do(reqest)

    //返回的状态码
    //status := response.StatusCode
    defer response.Body.Close()

    body,err := ioutil.ReadAll(response.Body)
    if err != nil {
        fmt.Println(err)
    }
    fmt.Println(string(body))
}
```

六、小案例

编写API接口和命令行脚本，将YAML文件的内容替换后返回

知识点：flag、http、文件i/o、err、panic



- API接口

```go
package main

import (
    "io/ioutil"
    "net/http"
    "strings"
)

var (
    filename = "F:\\goproject\\go-demo\\deployment.yaml"
    deployName = "nginx-deploy"
    appLabel = "nginx"
    image = "daocloud.io/library/nginx:latest"
)

func main() {
    http.HandleFunc("/api/yaml", handler)
    err := http.ListenAndServe(":8081", nil)
    if err != nil {
        panic(err)
    }
}

func handler(w http.ResponseWriter, r *http.Request) {
    deploymentStr, err := render(filename)
    if err != nil {
        panic(err)
    }
    w.WriteHeader(200)
    w.Write([]byte(deploymentStr))
}

func render(filname string) (string, error) {
    deployment, err := ioutil.ReadFile(filename)
    if err != nil {
        return "", err
    }
    valueStr := strings.Replace(string(deployment), "DEPLOYNAME",
deployName, -1)
    valueStr = strings.Replace(valueStr, "APPLABEL", appLabel, -1)
    valueStr = strings.Replace(valueStr, "IMAGE", image, -1)

    return valueStr, nil
```

```
}
```

- 命令行脚本

```
package main

import (
    "flag"
    "fmt"
    "io/ioutil"
    "strings"
)

var (
    filename string
    deployName = "nginx-deploy"
    appLabel = "nginx"
    image = "daocloud.io/library/nginx:latest"
)

func main() {
    flag.StringVar(&filename, "filename", "", "yaml文件路径")
    // 必须调用这个方法才会将数据转换出来
    flag.Parse()
    fmt.Println(filename)
    deploymentStr, err := render(filename)
    if err != nil {
        panic(err)
    }
    fmt.Println("------获取deployment yaml成功------")
    fmt.Println(deploymentStr)
}

func render(filname string) (string, error) {
    deployment, err := ioutil.ReadFile(filname)
    if err != nil {
        return "", err
    }
    valueStr := strings.Replace(string(deployment), "DEPLOYNAME",
deployName, -1)
    valueStr = strings.Replace(valueStr, "APPLABEL", appLabel, -1)
    valueStr = strings.Replace(valueStr, "IMAGE", image, -1)

    return valueStr, nil
}

```

七、QA

- 运维人员是否有必要学go？
- 运维体系从哪里开始？