Golang学习

终端命令符

cd 进入目录

cd ..返回上一级目录

dir 查看文件目录

cls 清屏

md name 创建目录

rd name 删除目录

ipconfig 查看本机ip

1. **go build**：
   - 用途：编译指定的代码包或可执行文件。
   - 示例：`go build main.go`
2. **go run**：
   - 用途：编译并运行指定的Go程序。
   - 示例：`go run main.go`
3. **go install**：
   - 用途：编译并安装指定的Go程序或库。
   - 示例：`go install ./...`
4. **go get**：
   - 用途：下载并安装指定的远程依赖。
   - 示例：`go get github.com/example/package`
5. **go test**：
   - 用途：运行测试文件。
   - 示例：`go test ./...`
6. **go fmt**：

- 用途：格式化Go源文件。
- 示例：`go fmt ./...`

7. **go vet**：
   - 用途：静态分析Go源代码中的常见错误。
   - 示例：`go vet ./...`
8. **go mod init**：
   - 用途：初始化一个新的模块（项目）。
   - 示例：`go mod init example.com/myproject`
9. **go mod tidy**：
   - 用途：整理和删除模块文件中未使用的依赖项。
   - 示例：`go mod tidy`
10. **go mod vendor**：
    - 用途：将模块的依赖项复制到 vendor 目录中。
    - 示例：`go mod vendor`
11. **go mod download**：
    - 用途：下载模块的依赖项。
    - 示例：`go mod download`
12. **go mod verify**：
    - 用途：验证模块依赖项的正确性。
    - 示例：`go mod verify`
13. **go clean**：
    - 用途：清理Go编译产生的文件。
    - 示例：`go clean -i -r`
14. **go version**：
    - 用途：显示安装的Go版本。
    - 示例：`go version`
15. **go env**：
    - 用途：显示Go的环境变量。
    - 示例：`go env GOPATH`

变量: variable

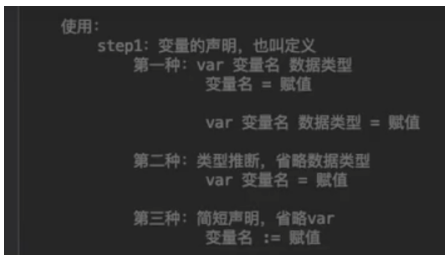
概念:一小块内存,用于存储数据,在运行中可以改变(和C语言差不多);

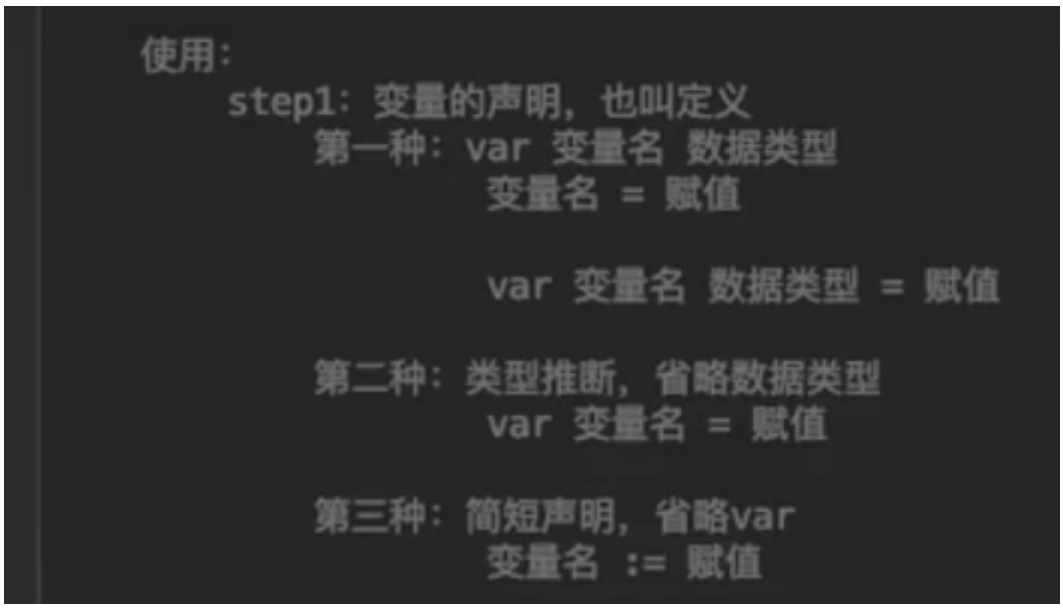
使用输出的时候直接用printf,和C语言一样的用法

%T判断数据类型

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      var nums1 int
7      nums1 = 100
8      fmt.Printf(nums1)
9      fmt.Printf("he%dlle", nums1)
10 }
```

变量的声明





异常强大

// 以后统一使用第三种

注意事项

设置了变量就必须使用;

简短定义不能定义全局变量;

变量默认值为0/ ;

遍历数组

C语言中没有的遍历方法,看起来还不错

```
s1 := []int{123, 546, 67, 6786, 54, 46, 576, 7}
for index, value := range s1 {
    fmt.Printf("%d-->%d \n", index, value)
}
```

切片

```
s2 := make([]int, 1, 20)
//s2:=[]int{12,343,4,354,435}
s2 = append(s2, s1...)
for index, value := range s2 {
    fmt.Printf("%d-->%d \n", index, value)
}
```

append是在末尾添加

当数据大小超过容量后,自动扩容(成倍增长),

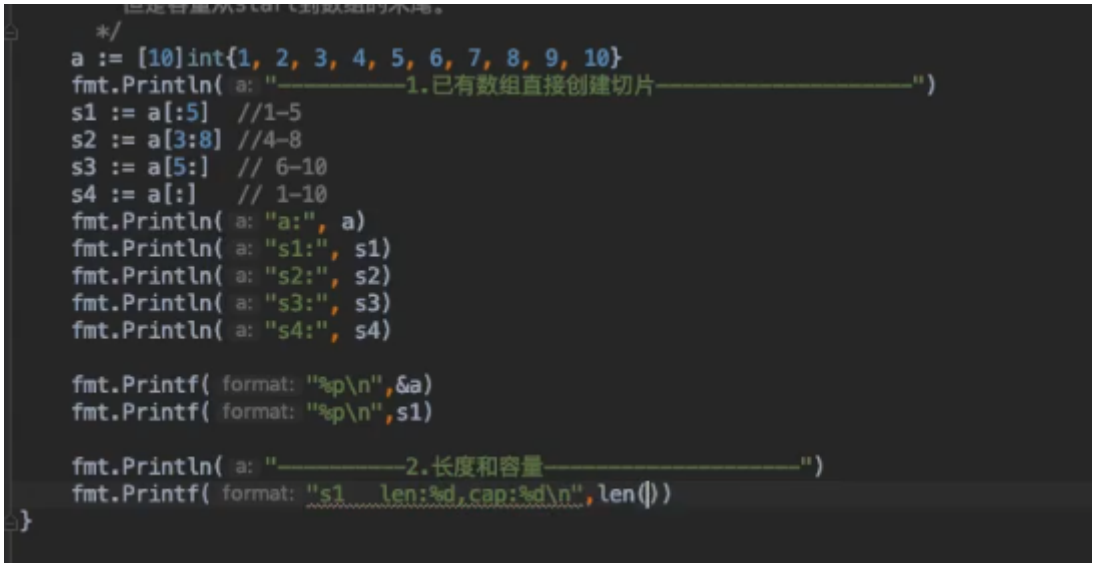
有关切片的操作

```
s1 := []int{1, 3, 2, 4, 5, 656, 5, 65, 6, 54}
fmt.Println(s1[1:5])
```

用[:]类似于数学中的集合

左闭右开[1:7)

在已有数组上可以直接创造切片



```
    */
    a := [10]int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
    fmt.Println( a: "————————1.已有数组直接创建切片————————————————")
    s1 := a[:5]  //1-5
    s2 := a[3:8] //4-8
    s3 := a[5:]  // 6-10
    s4 := a[:]   // 1-10
    fmt.Println( a: "a:", a)
    fmt.Println( a: "s1:", s1)
    fmt.Println( a: "s2:", s2)
    fmt.Println( a: "s3:", s3)
    fmt.Println( a: "s4:", s4)

    fmt.Printf( format: "%p\n",&a)
    fmt.Printf( format: "%p\n",s1)

    fmt.Println( a: "————————2.长度和容量————————————————")
    fmt.Printf( format: "s1   len:%d,cap:%d\n",len())
}
```

切片是引用类型,类似于指针所引用的东西,所以,要注意它的值的修改

深拷贝就是可以直接复制过来,在复制的基础上进行修改不会使原有值发生改变,而浅拷贝复制的地址,所以可以修改原有值.

```go
package main

import (
    "fmt"
)

func main() {
    fmt.Printf("Hello world\n")
    s1 := []int{1, 3, 2, 4, 5, 656, 5, 65, 6, 54}
    s2 := make([]int, 7, 20)
    fmt.Println(s1[1:5])
    fmt.Println(s2)
    copy(s2, s1)
    fmt.Println(s1)
    fmt.Println(s2)
}
```

在此处提供一个函数可以做到切片的深拷贝:

补充

slice的内部定义

```go
type slice struct(
```

```
    array unsafe.Poinert            //这个地方发生变化
    len int
    cap int
)
```

当无特别声明的时候,切片的值默认为0.

切片在进行切割的时候,是取了数组中相应数据的地址,前面的数据是依旧存在的,只不过起始指针位置发生了变化.示例:

```
1
```

MAP

MAP的创建:


```go
package main

import "fmt"

func main() {
    var map1 map[int]string
    var map2 = make(map[int]string)
    var map3 = map[int]string{1: "joker", 2: "king", 3: "queue"}
    fmt.Println(map1)
    fmt.Println(map2)
    fmt.Println(map3)
}
```

第一次写的时候在第三种声明的时候忘记了""

对map的增删改查操作:

```go
package main
```

```go

import "fmt"

func main() {
    var map1 map[int]string
    var map2 = make(map[int]string)
    var map3 = map[int]string{1: "joker", 2: "king", 3: "queue"}
    fmt.Println(map1)
    fmt.Println(map2)
    fmt.Println(map3)

    //存储MAP
    map2[1] = "joker"
    map2[2] = "king"
    map2[3] = "queue"
    //获取MAP
    fmt.Println(map2[1])
    //如果不存在可以通过一个方法去判断
    v1, ok := map2[1]
    if ok != false {
       fmt.Println(v1)
    } else {
       fmt.Printf("NULL")
    }
    //修改数据
    //就是直接赋值操作

    fmt.Println(map2)
    map2[2] = "joker's friend"
    fmt.Println(map2)
    //删除操作
    //使用内置函数
    delete(map2, 3)
    fmt.Println(map2)
}
```

MAP添加到Slice

```go
package main

import "fmt"

func main() {
    map1 := make(map[int]string)
    map1[1] = "joker"
```

```
    map1[2] = "batman"
    map1[3] = "harry"
    map1[4] = "potter"
    fmt.Println(map1)
    s1 := make([]map[int]string, 0, 10)
    fmt.Println(s1)
    s1 = append(s1, map1)
    fmt.Println(s1)
}
```

MAP也同为引用型数据

Go语言三大引用型数据:Slice map chan

字符串

一般来说,中文占据三个字节

```
package main

import "fmt"

func main() {
    str := "hello china"
    str1 := "周五考试"
    fmt.Println(str)
    for i := 0; i < len(str1); i++ {
       fmt.Printf("%c", str1[i])
    }
    fmt.Println(str1)
}
```

这个Go语言打印中文有点奇怪

string包常用函数https://pkg.go.dev/strings#Contains

compare

```
package main

import (
        "fmt"
        "strings"
```

```
)

func main() {
        fmt.Println(strings.Compare("a", "b"))
        fmt.Println(strings.Compare("a", "a"))
        fmt.Println(strings.Compare("b", "a"))
}

```

Contains

```
package main

import (
        "fmt"
        "strings"
)

func main() {
        fmt.Println(strings.Contains("seafood", "foo"))
        fmt.Println(strings.Contains("seafood", "bar"))
        fmt.Println(strings.Contains("seafood", ""))
        fmt.Println(strings.Contains("", ""))
}

```

https://pkg.go.dev/strings#Contains

strconv包

https://pkg.go.dev/strconv

```
package main

import (
        "fmt"
        "strconv"
)

func main() {
        v := "10"
        if s, err := strconv.Atoi(v); err == nil {
```

```
11                  fmt.Printf("%T, %v", s, s)
12          }
13
14  }
```

当时记得在C语言中用过,但是没注意看,现在发现了.

函数func

类似于C语言

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      fmt.Println("请输入求和数字:")
7      n := 0
8      fmt.Scanf("%d", &n)
9      res := getSum(n)
10     fmt.Printf("%d\n", res)
11 }
12
13 func getSum(n int) int {
14     sum := 0
15     for i := 1; i <= n; i++ {
16        sum += i
17     }
18     return sum
19 }
```

但是可以有多个返回值

其次,也不用刻意去声明,或者写在main函数的前面,写后面也可以通过

defer关键字

新关键字,意思为延迟,推迟,就是推迟一个函数的执行,并且遵循栈的特点,先进后出,当遍历到这个关键字的时候使得后置标有关键字的函数执行入栈,在main函数执行完毕之前出栈,便完成了操作

.在执行到defer关键字的时候,参数已经传递过去了

匿名函数

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      fmt.Println("Hello World")
7
8      func() {
9          fmt.Println("Hello nimianghanshu")
10     }()
11
12 }
```

定义一个没有名字的函数,直接调用,如果不用一个

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      fmt.Println("Hello World")
7
8      func() {
9          fmt.Println("Hello nimianghanshu")
10     }()
11
12     func(a, b int) {
13         fmt.Printf("%d %d", a, b)
14     }(1, 2)
15
16 }
```

定义带参数的匿名函数

匿名函数的作用:

1.将匿名函数作为另一个函数的参数,回调函数

2.将匿名函数作为另一个函数的返回值,可以作为闭包结构

高阶函数:

根据Go语言得数据类型特点.将一个函数作为另一个函数得参数

闭包:

在一个外层函数中,有内层函数,该内层函数中,会操作外层函数得局部变量,(外层函数中得参数,或者外层函数中直接定于的变量,)并且该外层函数的返回值就是这个内层函数,这个内层函数和外层函数的局部变量,统称为函数的闭包结构.

但是闭包结构中的外层函数的局部变量并不会随着外层函数的结束而销毁,因为内层函数还要继续使用.

在Go语言中，闭包（closure）是一个函数值，它引用了函数体之外的变量。这个函数可以访问并修改这些外部变量，并且这些变量的生命周期会延续到该闭包的生命周期内。

闭包在Go语言中通常是匿名函数，也就是没有函数名字的函数。闭包可以被用来捕获函数体外部的变量，并在函数体内使用这些变量。这种特性使得闭包非常灵活，可以用来实现一些功能强大的模式，比如函数式编程中的高阶函数、回调函数等。

下面是一个简单的示例，演示了闭包的概念：

```

package main

import "fmt"

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

func main() {// 创建一个 adder 函数闭包
    increment := adder()
// 使用闭包进行累加操作
    fmt.Println(increment(1)) // 输出: 1
    fmt.Println(increment(2)) // 输出: 3
    fmt.Println(increment(3)) // 输出: 6
}
```

在这个示例中，`adder` 函数返回了一个闭包，该闭包能够将传入的参数累加到 `sum` 变量上，并返回累加结果。在 `main` 函数中，我们通过调用 `adder` 函数创建了一个闭包 `increment`，

然后通过多次调用 `increment` 来实现累加操作， `sum` 变量的值在多次调用中被保留，达到了累加的效果。

指针:

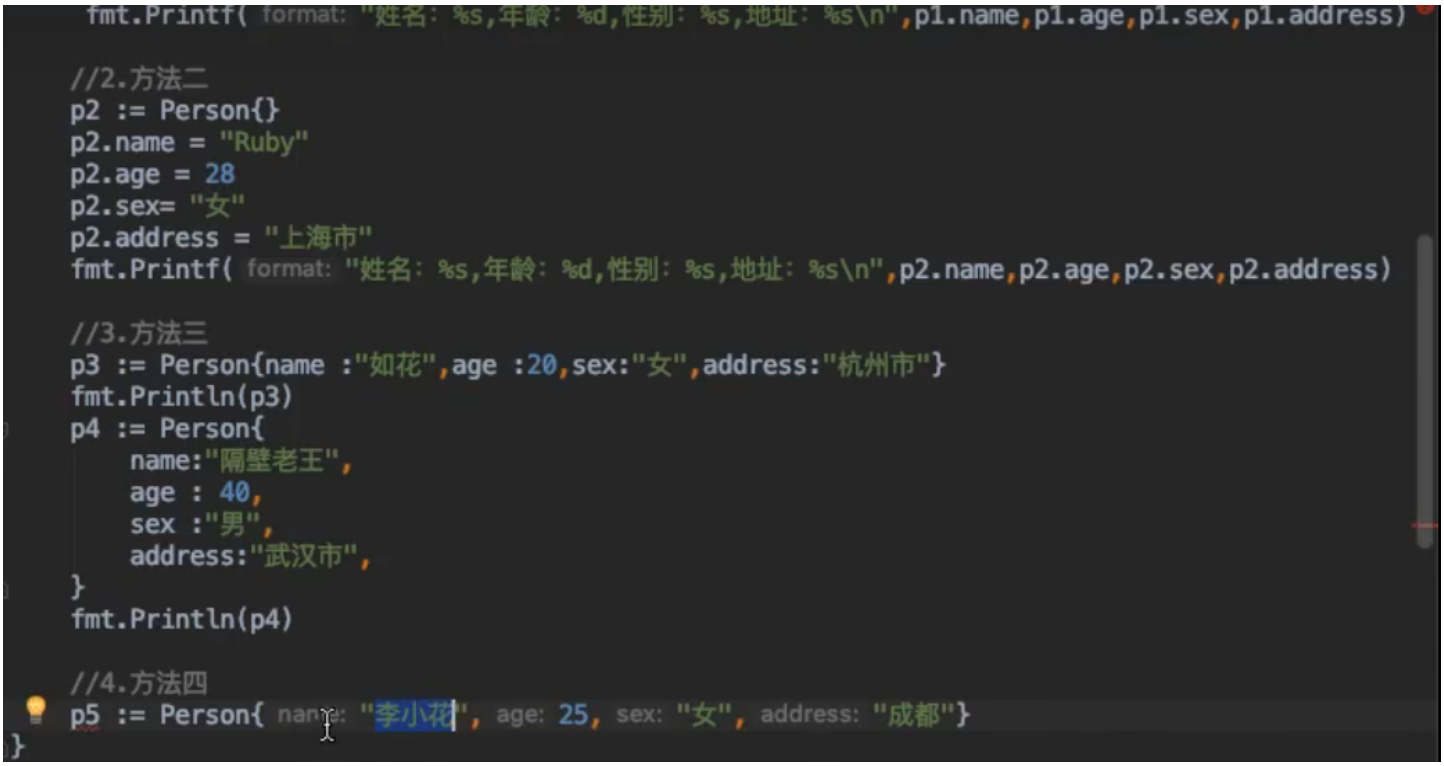
和C语言一样,参考C语言的使用.

结构体:

```go
package main

import "fmt"

type Person struct {
    name string
    age  int
    sex  string
}

func main() {
    user := Person{}
    admin := Person{
       name: "admin",
       age:  20,
       sex:  "man",
    }
    user.name = "gao"
    user.age = 19
    user.sex = "gay"
    fmt.Println(user)
    fmt.Println(admin)
}
```

基本和C语言差不多

其中有一种定义方式比较直接,在创建的时候直接输入数据



常用的几种定义方法

new操作:

new(T)分配了零值填充的T类型的内存空间,并返回其地址,即*T类型的值.即new返回指针.

面向对象编程OOP

在Go语言当中,并没有明确使用面向对象的编程方法,但是可以用结构体去模拟这个类似于Java中的面向对象编程,

特点:封装,继承,多态

Go语言仍然提供了支持面向对象编程的特性，例如：

1. **结构体（struct）**：结构体是一种用户定义的数据类型，可以包含不同类型的字段。在Go语言中，结构体常用于定义对象的属性。
2. **方法（Method）**：方法是与特定类型关联的函数。在Go语言中，可以通过为结构体定义方法来实现对象的行为。
3. **接口（Interface）**：接口是一种抽象类型，它定义了一组方法。任何类型只要实现了接口中定义的所有方法，就被认为是实现了该接口。接口在Go语言中被广泛用于实现多态。

方法

```go
package main

import "fmt"

type Person struct {
    name string
    age  int
    sex  string
}

func (p Person) printName() {
    fmt.Println(p.name, "正在复习高数")
}

func (p Person) sleep(n int) {
    fmt.Println(p.name, "SLEEPING", n)
}

func main() {
    p1 := Person{"liu", 18, "man"}
    p1.printName()
    p1.sleep(1000)
}
```

方法与函数的对比:

意义:

1方法:某个类别的行为功能,需要指定的接收者调用

函数:一段独立功能的代码,可以直接调用

语法;

方法:方法名字可以冲突,需要指定接收者调用

函数:命名不可以冲突

接口interface

接口是一组方法签名,

当某个类型为这个接口中的所有方法提供了方法的实现,它被称为实现接口.

接口和类型的关系,是非嵌入式.

```go
package main

import "fmt"

type USB interface {
    start()
    end()
}

type Mouse struct {
    name string
}

type FlashDisk struct {
    name string
}

func (m Mouse) start() {
    fmt.Println("鼠标开始工作了")
}

func (m Mouse) end() {
    fmt.Println("鼠标停止工作")
}

func (d FlashDisk) start() {
    fmt.Println("U盘已经插入了")
}

func (d FlashDisk) end() {
    fmt.Println("U盘已经拔出来了")
}

func textInterFace(usb USB) {
    usb.start()
    usb.end()
}

func main() {
    m1 := Mouse{"雷蛇"}
    fmt.Println(m1)
    f1 := FlashDisk{"小米64G"}
    fmt.Println(f1)
    textInterFace(m1)
}
```

多态

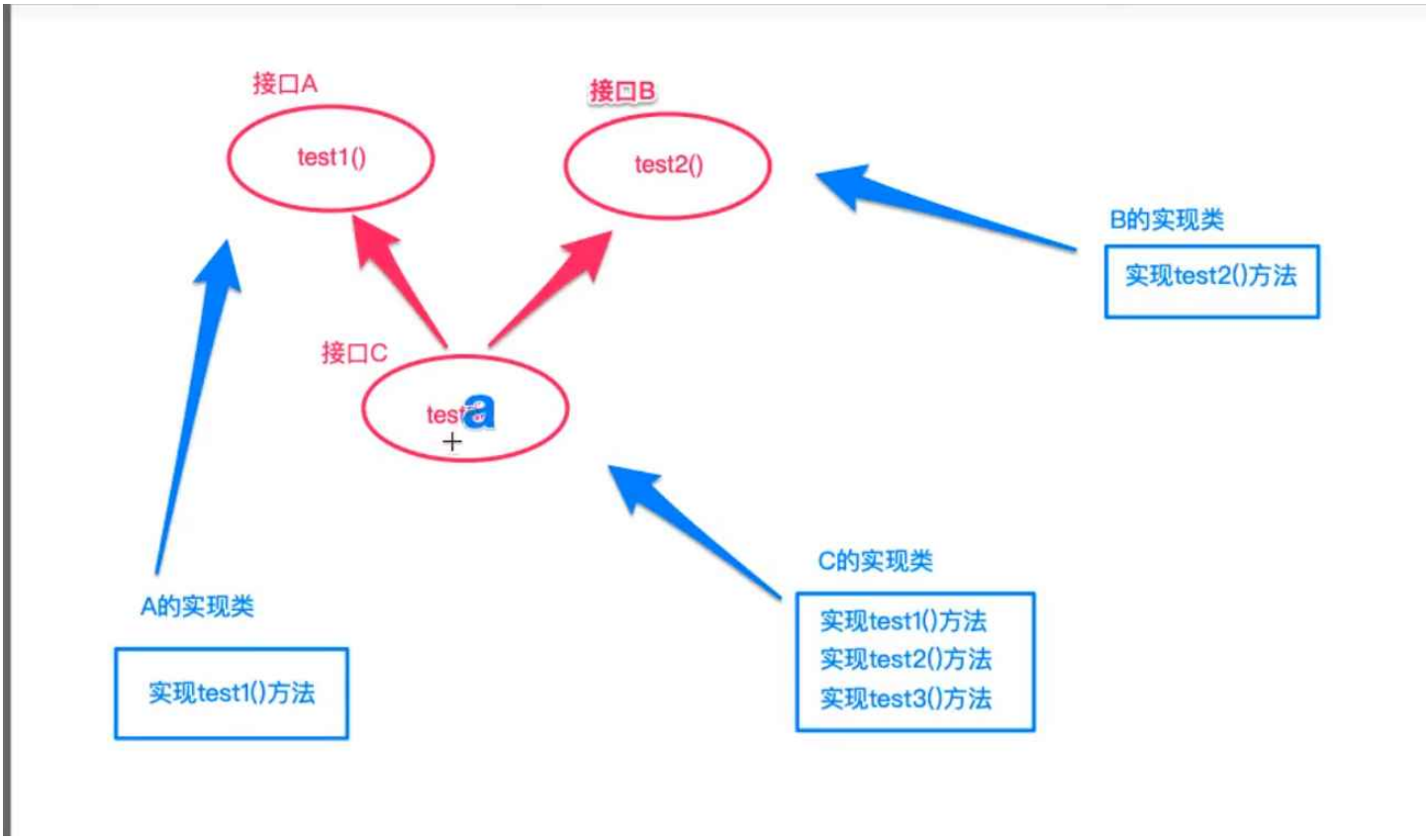
一个事物的多种形态

Go语言可以通过是接口模拟多态

接口嵌套

```go
package main

import "fmt"

type cat struct {
    name  string
    color string
    age   int
}

type CAT interface {
    voice()
    eat()
    sleep()
}

type PET interface {
    CAT
    play()
}

func (c *cat) play() {
    fmt.Println("小猫拆家ing....")
}

func (c *cat) voice() {
    fmt.Println("喵喵喵喵喵...")
}

func (c *cat) eat() {
    fmt.Println("来个罐罐")
}

func (c *cat) sleep() {
    fmt.Println("呼呼呼呼...")
}

```

```go
func main() {
    c1 := cat{"小花", "yellow", 3}
    c2 := cat{"三花", "chao", 3}

    fmt.Println(c1)
    c1.voice()
    c1.eat()
    c1.sleep()

    c2.play()
    c1.play()
}
```




```go
package main

import (
    "fmt"
    "math"
)

type Shape interface {
    peri() float64
    area() float64
```

```go
}

type Triangle struct {
    a float64
    b float64
    c float64
}

type Rectangle struct {
    a float64
    b float64
}

func (r Rectangle) peri() float64 {
    return (r.a + r.b) * 2
}

func (r Rectangle) area() float64 {
    return r.a * r.b
}

func (t *Triangle) peri() float64 {
    return t.a + t.b + t.c
}

func (t *Triangle) area() float64 {
    p := t.peri() / 2
    s := math.Sqrt(p * (p - t.a) * (p - t.b) * (p - t.c))
    return s
}

func main() {
    t1 := Triangle{2, 2, 2}
    fmt.Println(t1.peri())
    fmt.Println(t1.area())
    r1 := Rectangle{5, 5}
    fmt.Println(r1.peri())
    fmt.Println(r1.area())
}
```

type关键字

用于类型定义和类型别名

1.类型定义:type 类型名 Struct

2.类型别名:type 类型名 = Type

起别名

error

Go语言内置的数据类型,内置的接口定义方法

Golang看书所得

token标识符,

Token 原本意思为令牌的意思,在计算机当中意味分隔符

Golang语言全部关键字:

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

引导程序关键字:

```
package     //定义包名的关键字
import      //导入包名关键字
const       //常量声明关键字
var         //变量声明关键字
func        //函数定义关键字
defer       //延迟执行关键字
go          //并发语法糖关键字
return      //函数返回关键字
```

声明符合数据结构:

```
struct      //定义结构类型关键字
interface   //定义接口类型关键字
map         //声明或创建 map 类型关键字
chan        //声明或创建通道类型关键字
```

内置数据结构类型:

```
数值(16个)
    整型(12个)
            byte int int8 int16 int32 int64
            uint unint8 uint16 uint32 uint64 uintprt

    浮点型(2个)
            float32 float64

    复数型(2个)
            complex64 complex128

字符和字符串型(2个)
    string rune

接口型(1个)
    error

布尔型(1个)
    bool
```

注释:

Go语言当中可以设置复数:

```
var v= complex(2.1,3)   //构造一个复数
a := real(v)            //返回复数实部
b := image(v)           //返回复数虚部
```

切片Slice数据结构内部构造:

```
type slice struct{
    array unsafe.Pointer
    len int
    cap int
}
```

Hahahah so funny,,,,

我写了一年代码,今天才发现break用错了.,,,

标签:

不常用的东西,

进程,线程,协程

进程（Process）、线程（Thread）和协程（Coroutine）是计算机科学中重要的概念，它们在并发编程和多任务处理中起着不同的作用。

a. **进程（Process）：**
   - 进程是操作系统进行资源分配和调度的基本单位。每个进程都有自己独立的内存空间、代码和数据。进程之间相互独立，彼此之间不共享内存空间，通信必须通过进程间通信（IPC）的机制来实现。
   - 操作系统负责管理进程的创建、调度、销毁等操作。每个进程都有自己的地址空间，因此进程间的切换开销较大。

b. **线程（Thread）：**
   - 线程是进程中的一个执行单元，是操作系统进行调度的基本单位。一个进程可以包含多个线程，这些线程共享进程的资源，如内存空间、文件句柄等。
   - 线程之间共享进程的内存空间，因此线程间的通信相对容易。但是由于共享资源，需要确保线程同步，避免数据竞争等问题。
   - 线程的切换开销比进程小，因为它们共享同一个进程的地址空间。

c. **协程（ Coroutine）：**
   - 协程是一种用户态的轻量级线程，也被称为“微线程”。它不是由操作系统进行调度，而是由程序员在代码中显式地控制。
   - 协程可以在同一个线程内部进行切换，不需要像线程那样进行内核级别的上下文切换，因此切换开销很小。这使得协程非常适合于处理高并发的任务。
   - 不同于线程，协程通常由用户代码主动挂起和恢复，因此更容易理解和控制。它们通常用于编写异步、非阻塞的程序，例如网络服务器、事件驱动的程序等。

     在总体上，进程提供了更高的隔离性，线程提供了更高的并发性，而协程则提供了更高的可控性和效率。在不同的场景下，可以根据需要选择合适的并发模型来进行开发l'k

串行,并行,并发

串行、并行和并发是计算机科学中描述任务执行方式的重要概念，它们之间有着微妙的区别：

i. **串行（Serial）：**

- 串行指的是任务按照顺序依次执行的方式。在串行执行中，每个任务必须等待前一个任务完成后才能开始执行。因此，串行执行的特点是任务之间不存在重叠，只有一个任务在执行，执行顺序是确定的。

ii. **并行（Parallel）**：

- 并行指的是多个任务同时执行的方式。在并行执行中，多个任务可以同时进行，彼此之间没有先后顺序的依赖关系。并行执行通常需要多个处理单元（例如多核处理器），每个任务可以在不同的处理单元上并发执行。

iii. **并发（Concurrency）**：

- 并发是指多个任务在同一时间段内交替执行的方式。在并发执行中，虽然多个任务同时存在，但是在任意时刻只有一个任务在执行。任务之间通过时间片轮转或者事件驱动等机制进行切换，从而实现看似同时执行的效果。

综上所述：

- 串行是指任务按照顺序依次执行；
- 并行是指多个任务同时执行；
- 并发是指多个任务在同一时间段内交替执行。

并发通常用于解决多任务协作、提高系统资源利用率等问题，而并行通常用于提高计算速度、加速程序执行等需要大量计算的场景。

Goroutine(Coroutine)

先上代码,.这样好理解一点把

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    go printNums()
    for i := 0; i < 100; i++ {
       fmt.Printf("\t主Goroutine中打印字母:%d\n", i)
    }
    time.Sleep(1 * time.Second)
    fmt.Println("Game Over")
}

func printNums() {
```

```
    for i := 0; i < 100; i++ {
        fmt.Printf("子Goroutine中打印数字:%d\n", i)
    }
}
```

1.2.2 主goroutine

封装main函数的goroutine称为主goroutine。

主goroutine所做的事情并不是执行main函数那么简单。它首先要做的是：设定每一个goroutine所能申请的栈空间的最大尺寸。在32位的计算机系统中此最大尺寸为250MB，而在64位的计算机系统中此尺寸为1GB。如果有某个goroutine的栈空间尺寸大于这个限制，那么运行时系统就会引发一个栈溢出(stack overflow)的运行时恐慌。随后，这个go程序的运行也会终止。

此后，主goroutine会进行一系列的初始化工作，涉及的工作内容大致如下：

1. 创建一个特殊的defer语句，用于在主goroutine退出时做必要的善后处理。因为主goroutine也可能非正常的结束
2. 启动专用于在后台清扫内存垃圾的goroutine，并设置GC可用的标识
3. 执行mian包中的init函数
4. 执行main函数

   执行完main函数后，它还会检查主goroutine是否引发了运行时恐慌，并进行必要的处理。最后主goroutine会结束自己以及当前进程的运行。

后续相关包的使用:

time包

```
package main

import (
    "fmt"
    "time"
)

func main() {
    t1 := time.Now()
    //get now time

    fmt.Println(t1)
    f1 := t1.Format("2006年1月2日 15时4分5秒")
    fmt.Println(f1)
    //format the time

    timestr := "2011-11-11 11:11:11"
    parsedtime, err := time.Parse("2006-01-02 15:04:05", timestr)
```

```
    if err != nil {
        fmt.Println("解析失败:", err)
    }
    fmt.Println(parsedtime)
    //parse the time

    time.Sleep(5 * time.Second)
    endTime := time.Now()
    duration := endTime.Sub(t1)
    fmt.Println("Duration:", duration)
    //caculate the time duration

    timer := time.NewTimer(2 * time.Second)
    <-timer.C
    fmt.Println("定时器触发")
    //timer
}
```

获取当前时间,字符串转时间,时间睡眠

file操作

runtime包

init函数

```
package main

import (
        "fmt"
)

func init() {
        fmt.Println("init() 函数被执行")
}

func main() {
        fmt.Println("main() 函数被执行")
}
```

`init()` 函数是 Go 语言中的一个特殊函数，用于在程序执行开始时自动执行一些初始化操作。它没有参数，也没有返回值。`init()` 函数的特点如下：

a. `init()` 函数在包（package）被导入时自动执行，且执行顺序与导入顺序相反，即先导入的包先执行 `init()` 函数。

b. 一个包可以包含多个 `init()` 函数，它们按照定义顺序依次执行。

c. `init()` 函数不能被显式调用，而是在程序运行时由 Go 运行时系统自动调用。

d. `init()` 函数主要用于执行一些初始化操作，如初始化变量、加载配置、注册资源等。

l

并发编程临界资源问题

在并发编程中，临界资源安全问题是指多个并发执行的线程（或 Goroutine）同时访问临界资源（如共享变量、内存区域等）时可能出现的问题。这些问题包括：

1. **竞态条件（Race Condition）：** 当多个线程同时修改共享变量时，由于执行顺序不确定，导致最终结果依赖于线程执行的顺序，从而产生不确定的结果。这可能导致程序出现不一致的行为。
2. **数据竞争（Data Race）：** 这是一种特殊的竞态条件，指多个线程同时访问同一内存位置，并且至少其中一个是写操作。如果没有适当的同步机制保护，这可能导致未定义的行为。
3. **死锁（Deadlock）：** 当多个线程相互等待对方持有的资源时，导致所有线程都无法继续执行的情况。这通常发生在每个线程都试图获取其他线程持有的资源时。
4. **活锁（Livelihood）：** 类似于死锁，但是线程并没有被阻塞，而是在不断重试获取资源，导致系统无法继续进行下去。

为了避免这些问题，需要采取适当的同步机制来保护临界资源，例如：

- **互斥锁（Mutex）：** 通过在访问临界资源前获取锁，然后在访问结束后释放锁，来确保同一时间只有一个线程可以访问临界资源。
- **信号量（Semaphore）：** 用于控制对共享资源的访问，通过计数器来控制允许同时访问的线程数量。
- **条件变量（Condition Variable）：** 用于线程间的通信和同步，当某个条件满足时，线程可以继续执行，否则等待。
- **原子操作（Atomic Operation）：** 提供了一种在不需要锁的情况下进行并发访问的方式，通常用于对简单数据类型的操作，如增加、减少等。

正确地使用这些同步机制可以避免临界资源安全问题，确保程序的正确性和稳定性。

```
1 package main
2
3 import (
```

```
    "fmt"
    "math/rand"
    "time"
)

var tickets = 10

func main() {
    go saleTicket("售票口1")
    go saleTicket("售票口2")
    go saleTicket("售票口3")
    go saleTicket("售票口4")
    time.Sleep(3 * time.Second)
}

func saleTicket(name string) {
    rand.Seed(time.Now().UnixNano())
    for {
       if tickets > 0 {
          time.Sleep(time.Duration(rand.Intn(1000)) * time.Millisecond)
          fmt.Println(name, "售出", tickets)
          tickets--
       } else {
          fmt.Println(" 卖完了")
          break
       }
    }
}
```

在这个多线程当中,就会抢占临界资源,导致了数据的错误.

```
售票口2 售出 10
售票口3 售出 10
售票口4 售出 10
售票口1 售出 10
售票口1 售出 6
售票口2 售出 5
售票口4 售出 4
售票口3 售出 3
售票口4 售出 2
售票口1 售出 1
 卖完了
售票口3 售出 0
 卖完了
售票口2 售出 -1
```

```
15   卖完了
16 售票口4 售出 -2
17   卖完了
18
```

导致结果出现了不合理的地方

sync(同步)包

waitgroup(等待组)

```go
import "sync"

 var wg sync.WaitGroup

 func worker() {
    defer wg.Done()
    // 执行一些任务
 }

 func main() {
    // 启动多个worker
    for i := 0; i < 10; i++ {
        wg.Add(1)
        go worker()
    }

    // 等待所有worker完成
    wg.Wait()
 }
```

```go
package main

import (
    "fmt"
    "sync"
)

var wg sync.WaitGroup

func main() {
    wg.Add(2)
```

```
12
13      go fun1()
14      go fun2()
15
16      fmt.Println("main 阻塞")
17      wg.Wait() //main goroutine阻塞 ,其余的Goroutine执行
18      fmt.Println("解除!!!!")
19  }
20
21  func fun1() {
22      for i := 0; i < 10; i++ {
23          fmt.Println(i)
24      }
25      wg.Add(-1)
26  }
27
28  func fun2() {
29      defer wg.Add(-1)
30      for i := 0; i < 10; i++ {
31          fmt.Println("fucking", i)
32      }
33      //wg.Add(-1)
34  }
```

定义

```
1  var wg sync.Waitgroup
```

Add增加线程

Add(-1)减少线程

当线程==0,开始主线程,

当大于0,会造成问题,出现死锁(deadlock)

Mutex(互斥锁)

```
1  package main
2
3  import (
4      "fmt"
5      "sync"
6  )
```

```
7
8  var tickets = 10
9  var mu sync.Mutex
10 var wg sync.WaitGroup
11
12 func main() {
13     wg.Add(4)
14
15     go saleTicket("saler1")
16     go saleTicket("saler2")
17     go saleTicket("saler3")
18     go saleTicket("saler4")
19
20     wg.Wait()
21     fmt.Println("tickets done")
22
23 }
24
25 func saleTicket(name string) {
26     defer wg.Done()
27     for {
28        mu.Lock()
29        if tickets > 0 {
30           fmt.Println(name, "sale out ", tickets)
31           tickets--
32        } else {
33           mu.Unlock()
34           fmt.Println("without ticket")
35           break
36        }
37        mu.Unlock()
38     }
39 }
```

运行结果:

```
1 saler1 sale out  10
2 saler1 sale out  9
3 saler1 sale out  8
4 saler1 sale out  7
5 saler1 sale out  6
6 saler1 sale out  5
7 saler1 sale out  4
8 saler1 sale out  3
9 saler1 sale out  2
```

```
saler1 sale out  1
without ticket
without ticket
without ticket
without ticket
tickets done
```

很简单,上锁,解锁

和上古卷轴挺像的

RWMutex(读写锁)

```go
import "sync"

var mu sync.RWMutex
var count int

// 读操作
func read() {
    mu.RLock()
    defer mu.RUnlock()
    // 读取count值
}

// 写操作
func write() {
    mu.Lock()
    defer mu.Unlock()
    // 修改count值
}

```

视频所写代码:

```go
package main

import (
    "fmt"
    "sync"
)

```

```
var wg sync.WaitGroup
var rwm sync.RWMutex

func main() {
    wg.Add(4)

    go readData(1)
    go readData(2)
    go writeData(3)
    go writeData(4)

    wg.Wait()
    fmt.Println("program exited")
}

func readData(i int) {
    defer wg.Done()
    fmt.Println("start reading")
    rwm.RLock()
    fmt.Println("reading", i)
    rwm.RUnlock()
    fmt.Println("read over")
}

func writeData(i int) {
    defer wg.Done()
    fmt.Println("start writing")
    rwm.Lock()
    fmt.Println("writing", i)
    rwm.Unlock()
    fmt.Println("write over")
}
```

读取可以多个同时读取,但是写入只能有一个

```
package main

import (
    "fmt"
    "sync"
)

var (
    wg  sync.WaitGroup
    rwm sync.RWMutex
```

```
)

func main() {
    wg.Add(4)

    go readData(1)
    go readData(2)
    go writeData(3)
    go writeData(4)

    wg.Wait()
    fmt.Println("program exited")
}

func readData(i int) {
    defer wg.Done()
    //fmt.Println("start reading")
    rwm.RLock()
    fmt.Println("start reading")
    fmt.Println("reading", i)
    fmt.Println("read over")
    rwm.RUnlock()
    //fmt.Println("read over")
    fmt.Print("\n")
}

func writeData(i int) {
    defer wg.Done()
    //fmt.Println("start writing")
    rwm.Lock()
    fmt.Println("start writing")
    fmt.Println("writing", i)
    fmt.Println("write over")
    rwm.Unlock()
    //fmt.Println("write over")
    fmt.Print("\n")
}
```

ok,完美输出读写内容

channel(通道)chan

Channel 渠道,信道,频道

```
package main
```

```go

import "fmt"

func main() {
    ch1 := make(chan int)

    go sendData(ch1)
    /*
       for {
          v, ok := <-ch1
          if !ok {
             fmt.Println("all the data have been sent")
             break
          } else {
             fmt.Println("sendData is ", v)
          }
       }
    */
    for v := range ch1 {
       fmt.Println("Accepted data:", v)
    }
}

func sendData(ch1 chan int) {
    for i := 0; i < 10; i++ {
       ch1 <- i
    }
    close(ch1)
}
```

```
Accepted data: 0
Accepted data: 1
Accepted data: 2
Accepted data: 3
Accepted data: 4
Accepted data: 5
Accepted data: 6
Accepted data: 7
Accepted data: 8
Accepted data: 9
```

缓冲区通道

```go
package main

import (
    "fmt"
    "strconv"
)

func main() {
    ch1 := make(chan string, 4)
    go seadData(ch1)
    for {
       msg, ok := <-ch1
       if !ok {
          fmt.Println("all the data is over")
          break
       } else {
          fmt.Println("accepted data:", msg)
       }
    }
    fmt.Println("all the data is ", len(ch1))
}

func seadData(ch chan string) {
    for i := 0; i < 10; i++ {
       ch <- "hello Friday" + strconv.Itoa(i)
       fmt.Println("writing the data:", i)
    }
    close(ch)
}
```

单向通道

```go
ch1 := make(chan string)    //双向通道
ch2:=make(chan <- int)      //只能写
ch3:=make(<-chan int)       //只能读
```

select语句

当使用 Go 语言编写并发程序时，通常会使用 `select` 语句来处理多个 channel 的操作，它类似于其他语言中的 `switch` 语句，但是用于处理 channel 操作。

```go
package main
```

```
2
3  import (
4      "fmt"
5      "time"
6  )
7
8  func main() {
9      ch1 := make(chan string)
10     ch2 := make(chan string)
11
12     go func() {
13         time.Sleep(2 * time.Second)
14         ch1 <- "one"
15     }()
16
17     go func() {
18         time.Sleep(1 * time.Second)
19         ch2 <- "two"
20     }()
21
22     for i := 0; i < 2; i++ {
23         select {
24         case msg1 := <-ch1:
25             fmt.Println("Received from ch1:", msg1)
26         case msg2 := <-ch2:
27             fmt.Println("Received from ch2:", msg2)
28         }
29     }
30 }
```

反射reflect

```
1  package main
2
3  import (
4      "fmt"
5      "reflect"
6  )
7
8  type Person struct {
9      name string
10     age  int
11     sex  string
12 }
```

```
13
14 func (p *Person) say(msg string) {
15     fmt.Println("haha,what can i say ,manba out")
16 }
17
18 func (p *Person) info(age int) {
19     fmt.Println("we miss you laoda,", p.name, p.age, p.sex)
20 }
21
22 func getInfo(input interface{}) {
23     getType := reflect.TypeOf(input)
24     fmt.Println("getType:", getType.Name())
25     fmt.Println("getType:", getType.Kind())
26     getValue := reflect.ValueOf(input)
27     fmt.Println(getValue)
28     fmt.Println("------------------------------------------------------------
   ")
29     for i := 0; i < getType.NumField(); i++ {
30        field := getType.Field(i)
31        //value := getValue.Field(i).Interface()
32        fmt.Println(field.Name, field.Type)
33     }
34 }
35
36 func main() {
37     p1 := Person{"joker", 23, "man"}
38     getInfo(p1)
39     fmt.Println(p1)
40 }
```

反射（reflection）是 Go 语言的一项强大特性，它允许程序在运行时检查和修改变量、结构体、函数等程序结构的类型和值信息。反射可以用于编写通用的代码，使代码更加灵活和易于扩展。

反射包 `reflect` 提供了一组函数和类型，用于获取和操作类型、值、方法等信息。本教程将介绍反射的基础知识和常用技巧，帮助读者更好地理解和使用反射功能。

1. **反射的基础概念**

在 Go 语言中，每个变量都有一个静态类型（static type），表示变量的数据类型，如 `int`、`string`、`struct` 等。静态类型在编译时就已经确定，因此无法在运行时进行修改。

但是，Go 语言提供了一种机制，允许程序在运行时动态地获取和处理类型和值的信息，这就是反射。通过反射，我们可以在运行时获取一个变量的类型信息和值信息，或者动态地创建、操作对象。

在反射中，有几个重要的概念需要了解：

1.1 类型和值

在反射中，类型和值是两个核心概念。类型（type）表示一个变量的数据类型，如 `int`、`string`、`struct` 等；值（value）则表示一个变量的具体数值。

在 Go 语言中，类型和值都可以用 `reflect.Type` 和 `reflect.Value` 类型表示。`reflect.Type` 表示类型信息，包括类型的名称、大小、对齐方式、字段、方法等；`reflect.Value` 则表示值信息，包括值的类型、大小、具体数值等。

1.2 接口和反射对象

在反射中，还有两个重要的概念：接口（interface）和反射对象（reflect object）。

在 Go 语言中，接口是一种特殊的类型，它定义了一组方法，表示一个对象所具有的行为。接口可以用于实现多态性，使不同类型的对象能够统一地进行操作。

反射对象是指 `reflect.Type` 或 `reflect.Value` 类型的实例，它们用于表示变量的类型和值信息。反射对象可以通过反射函数获取，也可以通过接口类型转换得到。

1.3 可寻址性

在反射中，变量的可寻址性是一个重要的概念。一个变量是否可寻址，决定了我们是否能够通过反射对变量进行修改。

在 Go 语言中，一个变量是否可寻址，取决于它的类型和具体情况。一般来说，只有通过指针或接口类型，才能访问变量的地址。

2. **反射的常用函数和用法**

在 Go 语言中，反射包 `reflect` 提供了一组函数和类型，用于获取和操作类型、值、方法等信息。下面介绍几个常用的反射函数和用法。

2.1 reflect.TypeOf 和 reflect.ValueOf

`reflect.TypeOf` 函数用于获取一个变量的类型信息，返回值为 `reflect.Type` 类型；`reflect.ValueOf` 函数用于获取一个变量的值信息，返回值为 `reflect.Value` 类型。

例如：

```
1 go复制代码
2 var x int = 123
3 t := reflect.TypeOf(x)  // 获取变量 x 的类型信息
4 v := reflect.ValueOf(x) // 获取变量 x 的值信息
```

上述代码中，变量 `t` 是 `reflect.Type` 类型，表示变量 `x` 的类型信息；变量 `v` 是 `reflect.Value` 类型，表示变量 `x` 的值信息。

2.2 Type 和 Value 方法

`reflect.Type` 和 `reflect.Value` 类型都提供了一些方法，用于获取类型和值的详细信息。

`reflect.Type` 类型提供的常用方法包括：

- `Name()`：获取类型的名称，仅适用于命名类型。
- `Kind()`：获取类型的基础类型，返回值为 `reflect.Kind` 类型，如 `reflect.Int`、`reflect.String` 等。
- `Size()`：获取类型的大小，以字节为单位。
- `NumField()`：获取结构体类型的字段数量。
- `Field(i int)`：获取结构体类型的第 `i` 个字段信息，返回值为 `reflect.StructField` 类型。
- `NumMethod()`：获取类型的方法数量。
- `Method(i int)`：获取类型的第 `i` 个方法信息，返回值为 `reflect.Method` 类型。

`reflect.Value` 类型提供的常用方法包括：

- `Interface()`：将值转换为 `interface{}` 类型，返回对应的接口值。
- `Bool()`、`Int()`、`Float()`、`String()` 等方法：将值转换为对应的基本类型，并返回该类型的值。
- `CanSet()`：判断值是否可寻址，即能否通过反射对其进行修改。
- `Elem()`：获取指针或接口指向的值，如果不是指针或接口类型，则返回原值。
- `Field(i int)`：获取结构体类型的第 `i` 个字段值，返回值为 `reflect.Value` 类型。
- `MethodByName(name string)`：根据方法名获取方法值，返回值为 `reflect.Value` 类型。

例如：

```
1
2  type Person struct {
3      Name string
4      Age  int
5  }
6  p := Person{"Alice", 18}
7  v := reflect.ValueOf(p)
8  fmt.Println(v.Kind())      // 输出: struct
9  fmt.Println(v.NumField())  // 输出: 2
10 name := v.Field(0).String()
11 age := v.Field(1).Int()
12 fmt.Println(name, age)     // 输出: Alice 18
```

上述代码中，通过 `reflect.ValueOf` 函数获取结构体 `Person` 的值信息，并通过 `Value` 类型的方法获取结构体的字段信息。其中， `Field` 方法返回的是 `reflect.Value` 类型，需要再次调用 `String` 或 `Int` 等方法将其转换成实际类型。

2.3 reflect.New 和 reflect.MakeFunc

`reflect.New` 函数用于动态创建一个新对象，返回值为 `reflect.Value` 类型；`reflect.MakeFunc` 函数用于动态创建一个新函数，返回值为 `reflect.Value` 类型。

例如：

```
go复制代码
type Person struct {
    Name string
    Age  int
}
t := reflect.TypeOf(Person{})
p := reflect.New(t)  // 创建 Person 类型的新对象
v := p.Elem()        // 获取对象的值
v.FieldByName("Name").SetString("Bob")
v.FieldByName("Age").SetInt(20)
fmt.Println(v.Interface())  // 输出: {Bob 20}
```

上述代码中，首先使用 `reflect.TypeOf` 函数获取 `Person` 类型的元信息，然后使用 `reflect.New` 函数动态创建一个新的 `Person` 类型对象，最后使用 `Elem` 方法获取对象的值。通过 `FieldByName` 方法和 `SetString`、 `SetInt` 等方法可以对对象的字段进行修改。

另一个常用的函数是 `reflect.MakeFunc`，它可以用于动态创建一个新的函数对象，返回值为 `reflect.Value` 类型。使用 `MakeFunc` 函数时，需要传入一个函数类型和一个函数实现作为参数，然后利用反射机制创建一个新的函数对象。

例如：

```
go复制代码
func add(a, b int) int {return a + b
}
fType := reflect.TypeOf(add) // 获取函数类型
fValue := reflect.ValueOf(add) // 获取函数值
newFunc := reflect.MakeFunc(fType, func(args []reflect.Value) []reflect.Value {
    x := args[0].Int()
    y := args[1].Int()
    ret := fValue.Call([]reflect.Value{reflect.ValueOf(x+1),
reflect.ValueOf(y+1)})return ret
})
```

```
addPlusOne := newFunc.Interface().(func(int, int) int)
result := addPlusOne(1, 2)
fmt.Println(result) // 输出: 5
```

上述代码中，首先使用 `reflect.TypeOf` 和 `reflect.ValueOf` 函数获取函数 `add` 的类型信息和值信息，然后利用 `reflect.MakeFunc` 函数和一个函数实现创建了一个新的函数对象 `newFunc`。最后使用 `Interface` 方法将其转换为实际类型，并调用该函数。

2.4 反射的类型转换

在反射中，类型转换是一个重要的问题。由于 `reflect.Type` 和 `reflect.Value` 类型都是接口类型，因此需要进行类型转换才能获取实际类型。

反射提供了以下几种类型转换方式：

- `reflect.Value.Interface()`：将 `reflect.Value` 类型转换为 `interface{}` 类型，返回对应的接口值。
- `reflect.Value.Convert()`：将 `reflect.Value` 类型的值转换为指定类型的值。
- `reflect.ValueOf(interface{})`：将 `interface{}` 类型的值转换为 `reflect.Value` 类型。

例如：

```
go复制代码
var x interface{} = 123
v := reflect.ValueOf(x)

if v.Kind() == reflect.Int {
    fmt.Println(v.Int()) // 输出: 123
}
y := v.Interface().(int)
fmt.Println(y) // 输出: 123
```

上述代码中，首先将一个 `interface{}` 类型的变量 `x` 转换为 `reflect.Value` 类型的变量 `v`，然后通过 `Kind` 方法判断 `v` 的类型是否为 `int`。接着使用 `Interface` 方法将 `v` 转换为 `interface{}` 类型，并使用类型断言将其转换为 `int` 类型的变量 `y`。

3. 总结

反射是 Go 语言的一项强大特性，它允许程序在运行时检查和修改变量、结构体、函数等程序结构的类型和值信息。在反射中，类型和值是两个核心概念，接口和反射对象是重要的概念之一。反射包 `reflect` 提供了一组函数和类型，用于获取和操作类型、值、方法等信息。反射的使用需要注意可寻址性、类型转换等问题。

Golang核心编程读书

函数签名:就是函数的名字,只不过标出了类型

defer后面必须根方法或者函数,不然会报错

defer先注册后执行

扇入扇出

扇入

```go
package main

import "fmt"

func main() {
    ch1 := make(chan string)
    ch2 := make(chan string)
    out := make(chan string)

    go func() {
       for {
          select {
          case msg := <-ch1:
             out <- msg
          case msg := <-ch2:
             out <- msg
          }
       }
    }()

    go func() {
       ch1 <- "Hello"
       ch2 <- "World"
    }()

    result := <-out
    fmt.Println(result) // 输出: Hello
}
```

Go通道退出通知机制

- 读取已经关闭的通道不会引起阻塞，也不会导致panic，而是立即返回该通道存储类型的零值。
- 关闭select 监听的某个通道能使select立即感知此种通知，并能够进行相应的处理。

```go
package main

import (
    "fmt"
    "math/rand/v2"
    "runtime"
)

func main() {
    done := make(chan struct{})
    ch := GA(done)
    fmt.Println(<-ch)
    fmt.Println(<-ch)
    close(done)
    fmt.Println(<-ch)
    fmt.Println(<-ch)
    println("Number of goroutines:", runtime.NumGoroutine())
}

func GA(done chan struct{}) chan int {
    ch := make(chan int)
    text := rand.IntN(100)
    fmt.Println(text)
    go func() {
    Lable:
       for {
          select {
          case ch <- rand.IntN(100):
          case <-done:
             break Lable
          }
       }
       close(ch)
    }()
    return ch
}
```

权限

当我们谈论权限时，通常是指在计算机系统中对资源（如文件、目录、数据库记录等）进行访问控制的过程。权限控制是确保系统安全性和数据保护的重要步骤。下面是一些常见的权限概念和教学：

1. **用户和组：**
   - **用户（User）：** 是系统中的一个实体，可以是一个人、一个程序或一个服务。
   - **组（Group）：** 将用户组织在一起，方便对一组用户设置相同的权限。
2. **权限类型：**
   - **读取权限（Read）：** 允许用户查看资源内容。
   - **写入权限（Write）：** 允许用户修改或添加资源内容。
   - **执行权限（Execute）：** 允许用户执行资源（如程序文件）。
   - **所有者权限（Owner）：** 资源的所有者拥有最高权限，可以自由地修改权限设置。
   - **组权限（Group）：** 指定属于资源所在组的用户的权限。
   - **其他用户权限（Others）：** 指定除所有者和组成员之外的其他用户的权限。
3. **权限表示方法：**
   - **符号表示法（Symbolic Notation）：** 使用符号（如 rwx）表示权限，例如 `rwxr-xr--` 表示所有者具有读、写、执行权限，组成员具有读、执行权限，其他用户只有读权限。
   - **数字表示法（Numeric Notation）：** 使用数字表示权限，每种权限用一个数字表示（读=4、写=2、执行=1），将所有权限加起来得到一个三位数，例如 `764` 表示所有者具有读、写、执行权限，组成员有读、写权限，其他用户有读权限。
4. **权限管理命令：**
   - **chmod：** 用于改变文件或目录的权限。
   - **chown：** 用于改变文件或目录的所有者。
   - **chgrp：** 用于改变文件或目录的所属组。
   - **ls -l：** 显示文件或目录的详细信息，包括权限信息。
5. **权限实践：**
   - **为文件设置权限：** 使用 `chmod` 命令为文件或目录设置不同用户的权限。
   - **为用户和组分配权限：** 使用 `chown` 和 `chgrp` 命令为文件或目录指定所有者和所属组。
   - **查看文件权限：** 使用 `ls -l` 命令查看文件的权限信息。

终端文件操作

操作终端相关的文件句柄常量
os.Stdin:标准输入
os.Stdout:标准输出
os.Stderr:标准错误输出

最基础的输入输出

```go
package main

import "fmt"

var (
    firstName,lastName,s string
    i int
    f float32
    input = "56.12/5212/Go"
    format = "%f/%d/%s"
)

func main() {
```

```
    fmt.Println("please enter your full name:")
    //fmt.Scanln(&firstName,&lastName)
    fmt.Scanf("%s %s",&firstName,&lastName) //这一句和上一句是相同的
    fmt.Printf("Hi %s %s!\n",firstName,lastName)
    fmt.Sscanf(input,format,&f,&i,&s)
    fmt.Println("From the string we read:",f,i,s)

}
```

带缓冲区的读写

可以读入一行的数据

```
package main

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

var inputReader *bufio.Reader
var input string
var err error


func main(){
    inputReader = bufio.NewReader(os.Stdin)
    fmt.Println("please enter some input:")
    // 下面这个表示我们读取一行，最后是以\n 为分割的，\n表示换行
    input,err = inputReader.ReadString('\n')
    if err != nil{
        fmt.Println(err)
        return
    }
    fmt.Printf("the input was:%s\n",input)
}


---------------------------------------
please enter some input:
haha what can i say? mayba out
the input was:haha what can i say? mayba out
```

文件读写(一行一行读)

会读取计算机本地文件中的内容,并将其打印在终端

```
package main

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

func main(){
    file,err:=
os.Open("/Users/zhaofan/go_project/src/go_dev/07/ex13/main.go")
    if err!= nil{
        fmt.Println("open file is err:",err)
        return
    }
    //这里切记在打开文件的时候跟上defer 关闭这个文件，防止最后忘记关闭
    defer file.Close()
    inputReader := bufio.NewReader(file)
    for {
        inputString,readError:= inputReader.ReadString('\n')
        // 这里readError == io.EOF表示已经读到文件末尾
        if readError ==io.EOF{
            return
        }
        // 下面会把每行的内容进行打印
        fmt.Printf("the input is:%s",inputString)
    }

}

---------------------------------------
the input is:jdowajoid
the input is:weadaiy
the input is:fefsef
the input is:efrdgt
the input is:rgtf
the input is:htfy
the input is:htf
the input is:htfht
the input is:fh
the input is:g
```

```
the input is:
the input is:
the input is:
the input is:
the input is:     ddawd
the input is:      awd
the input is:
the input is:
the input is:       wdawd
the input is:
the input is:       fefidesofh
the input is:
the input is:@#343435
```

文件读写(整体读)

如果文件大,就最好不要用

```go
package main

import (
    "io/ioutil"
    "fmt"
)

func main(){
    buf,err := ioutil.ReadFile("/Users/zhaofan/go_project/src/go_dev/07/ex13/main.go")
    if err != nil{
        fmt.Println("read file is err:",err)
        return
    }
    fmt.Printf("%s\n",string(buf))

}
```

将终端内容变为文件

```
os.OpenFile("filename",os.O_WRONLY|os.O_CREATE,066)


```

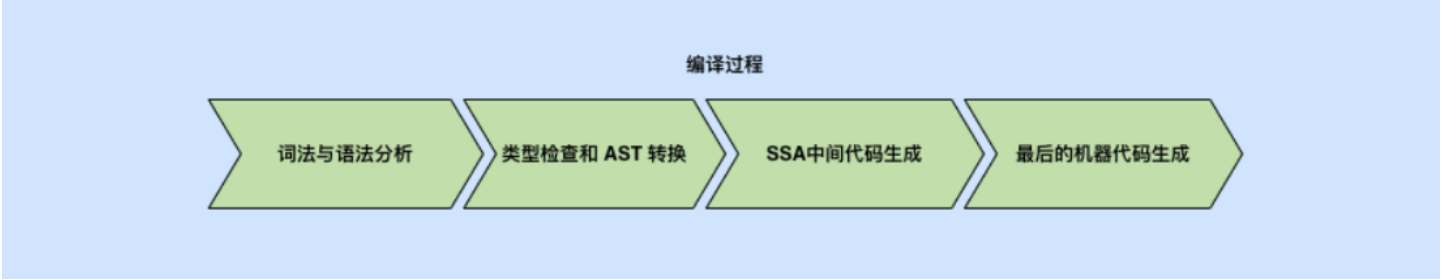
第一个参数是文件名
第二个参数是文件的打开模式：
os.O_WRONLY:只写
os.O_CREATE:创建文件
os.O_RDONLY:只读
os.O_RDWR:读写
os.O_TRUNC:清空

第三个参数：权限控制
r-->4
w-->2
x-->1

```go
package main

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

func testWriteBuf(){
    file,err:= os.OpenFile("mylog.txt",os.O_WRONLY|os.O_CREATE,0666)
    if err!= nil{
        fmt.Println("open file failed:",err)
        return
    }
    defer file.Close()

    bufioWrite := bufio.NewWriter(file)
    for i:=0;i<10;i++{
        bufioWrite.WriteString("hello\n")
    }
    bufioWrite.Flush()
}

func main(){
    testWriteBuf()
}
```

go编译

go的编译分为四个阶段



词法和语法分析

类型检查和AST转换

SSA中间代码生成

最后机器代码生成