C/C++Linux服务器开发

高级架构师课程

三年课程沉淀 | 五次精益升级 | 十年行业积累 | 百个实战项目 | 十万内容受众

讲师:darren/326873713

扫一扫 升职加薪

班主任:柚子/2690491738

讲师介绍--专业来自专注和实力

King老师

系统架构师，曾供职著名创业公司系统架构师，微软亚洲研究院、创维集团全球研发中心。国内第一代商业Paas平台开发者。著有多个软件专利，参与多个开源软件维护。在全球化，高可用的物联网云平台架构与智能硬件设计方面有丰富的研发与实战经验。

Darren老师

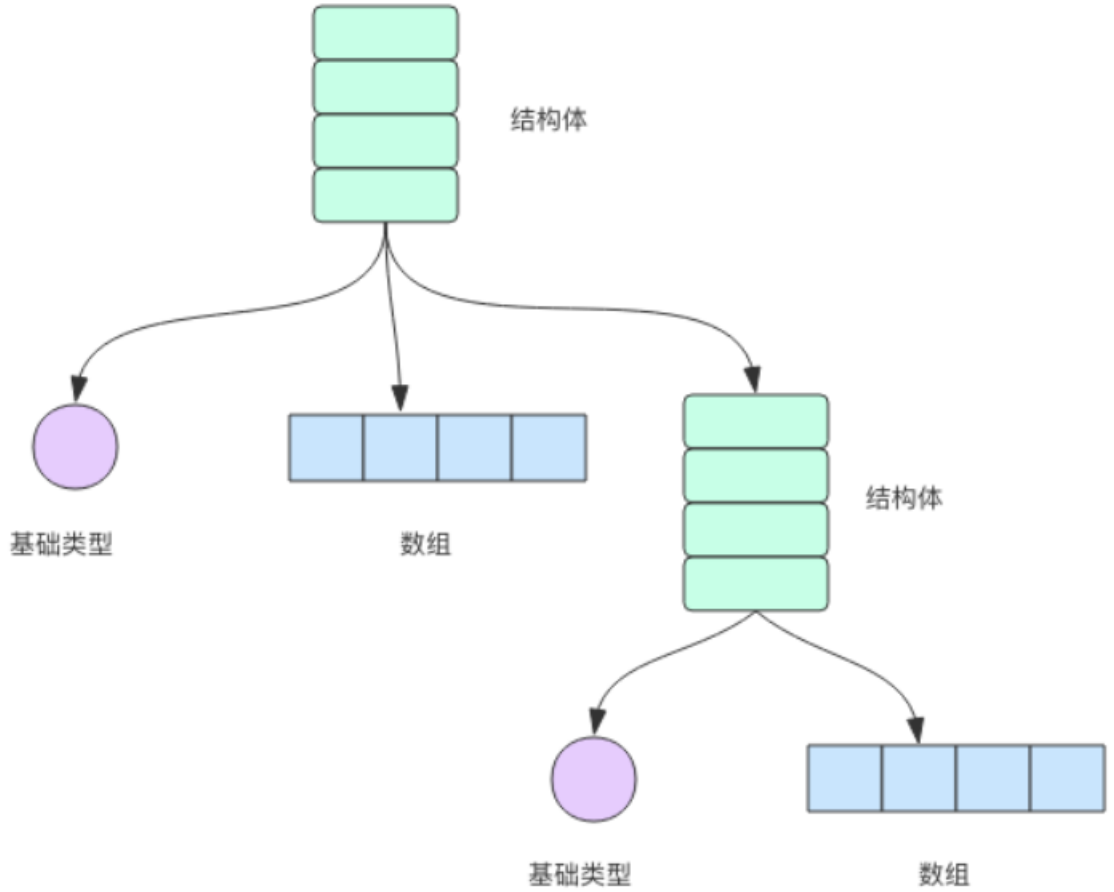
曾供职于国内知名半导体公司（珠海扬智/深圳联发科），曾在某互联网公司担任音视频通话项目经理。主要从事音视频驱动、多媒体中间件、流媒体服务器的开发，开发过即时通讯+音视频通话的大型项目，在音视频、C/C++/GO Linux服务器领域有丰富的实战经验。

Go语言接口与反射

1. 结构
2. 接口
3. 反射

1.1 struct简介

- Go通过结构体struct和interface实现oop(面向对象编程)
- struct的成员(也叫属性或字段)可以是任何类型，如普通类型、复合类型、函数、map、interface、struct等



1.2 struct详解-struct定义

struct定义

```
type struct_variable_type struct {
    member member_type
    member member_type
    .....
    member member_type

 }
//示例
type Student struct {
    name string
    age int
    Class string
}
```

小写私有成员(对外不可见)
首字母大写则该成员为公有成员(对外可见)

1.2 struct详解-声明与初始化

声明与初始化

```
var stu1 Student
var stu2 *Student= &Student{} //简写stu2 := &Student{}
var stu3 *Student = new(Student)  //简写stu3 := new(Student)
```

1.2 struct详解-struct使用

- 访问其成员都使用"."
- struct分配内存使用new，返回的是指针
- struct没有构造函数，但是我们可以自己定义"构造函数"
- struct是我们自己定义的类型，不能和其他类型进行强制转换

```
type Student struct {
    name   string
    age    int
    Class string
}
```

1-1.go

```
var stu1 Student
stu1.age = 34
stu1.name = "darren"
stu1.Class = "class1"
fmt.Println(stu1.name) //darren
var stu2 *Student = new(Student)
stu2.name = "king"
stu2.age = 33
fmt.Println(stu2.name, (*stu2).name) //king
var stu3 *Student = &Student{name: "rose", age: 18, Class: "class3"}
fmt.Println(stu3.name, (*stu3).name) //rose  rose
```

1.2 struct详解-自定义构造函数

- 通过工厂模式自定义构造函数方法

```
func Newstu(name1 string,age1 int,class1 string) *Student {
   return &Student{name:name1,age:age1,Class:class1}
}
func main() {
   stu1 := Newstu(“darren",34,"math")
   fmt.Println(stu1.name) // darren
}
```

1-2.go

1.3 struct tag

- tag可以为结构体的成员添加说明或者标签便于使用,这些说明可以通过反射获取到。
- 结构体中的成员首字母小写对外不可见，但是我们把成员定义为首字母大写这样与外界进行数据交互会带来极大的不便，此时tag带来了解决方法

```
type Student struct {
    Name string      "the name of student"
    Age int          "the age of student"
    Class string     "the class of student"
}
```

1.3 struct tag - 应用场景json示例

应用场景示例，json序列化操作（序列化和反序列化演示）

```go
type Student struct {
    Name string `json:"name"`
    Age  int    `json:"age"`
}

func main() {
    var stu = Student{Name: "darren", Age: 34}
    data, err := json.Marshal(stu)
    if err != nil {
        fmt.Println("json encode failed err:", err)
        return
    }
    fmt.Println(string(data)) //{"name":"darren","age":34}
}
```

1-3-tag-json.go

1.4 struct匿名成员（字段、属性）

- 结构体中，每个成员不一定都有名称，也允许字段没有名字，即匿名成员。
- 匿名成员的一个重要作用，可以用来实现oop中的继承。
- 同一种类型匿名成员只允许最多存在一个。
- 当匿名成员是结构体时，且两个结构体中都存在相同字段时，优先选择最近的字段。

```
type Person struct {
    Name string
    Age  int
}
type Student struct {
    score string
    Age   int
    Person  // 匿名内嵌结构体
}
```

代码：1-4.go

```
func main() {
    var stu = new(Student)
    stu.Age = 34                    //优先选择Student中的Age
    fmt.Println(stu.Person.Age, stu.Age) // 0,34
}
```

1.5 struct-继承、多继承

- 当结构体中的成员也是结构体时，**该结构体就继承了这个结构体**，继承了其所有的方法与属性，当然有多个结构体成员也就是多继承。
- 访问父结构中属性也使用“.”，但是当子结构体中存在和父结构中的字段相同时候，只能使用：**"子结构体.父结构体.字段"**访问父结构体中的属性，如上面示例的stu.Person.Age
- 继承结构体可以使用别名，访问的时候通过别名访问，如下面示例man1.job.Salary：

```
type Person struct {
    Name string
    Age  int
}
type Teacher struct {
    Salary  int
    Classes string
}
```

1-5.go

```
type man struct {
    sex    string
    job    Teacher //别名，继承Teacher  这个时候就不是匿名了
    Person         //继承Person

}

func main() {
    var man1 = new(man)
    man1.Age = 34
    man1.Name = "darren"
    man1.job.Salary = 100000
    fmt.Println(man1, man1.job.Salary) //&{ {8500 } {darren 34}} 8500
}
```

1.6 struct-结构体中的方法

方法是什么

- Go方法是作用在接受者（个人理解成作用对象）上的一个函数，接受者是某种类型的变量，因此方法是一种特殊类型的函数。
- 接收者可以是任何类型，不仅仅是结构体，Go中的基本类型（int，string，bool等）也是可以，或者说数组的别名类型，甚至可以是函数类型。但是，接受者不能是一个接口类型，因为接口是一个抽象的定义，方法是一个具体实现。
- 一个类型加上它的方法等价于面向对象中的一个类。一个重要的区别是：在 Go 中，类型的代码和绑定在它上面的方法的代码可以不放置在一起，它们可以存在在不同的源文件，**唯一的要求是：它们必须是同一个包的**。
- 类型 T（或 *T）上的所有方法的集合叫做类型 T（或 *T）的方法集。
- 因为方法是函数，所以同样的，不允许方法重载，即对于一个类型只能有一个给定名称的方法。但是如果基于接收者类型，是有重载的：具有同样名字的方法可以在 2 个或多个不同的接收者类型上存在，比如在同一个包里这么做是允许的
- 别名类型不能有它原始类型上已经定义过的方法（因为别名类型和原始类型底层是一样的）。

定义方法

```
func(recv recevier_type(结构体))methodName(parameter_list)(return_value_list){}
```

1.6 struct-结构体中的方法-示例

代码：1-6.go

```
type Person struct {
    Name string
    Age  int
}

func (p Person) Getname() string { //p代表结构体本身的实列，类似python中的self,这里p可以写为self
    fmt.Println(p.Name)
    return p.Name
}

func main() {
    var person1 = new(Person)
    person1.Age = 34
    person1.Name = "darren"
    person1.Getname() // darren
}
```

1.6 struct-结构体中的方法-示例2

代码：1-6-2.go

结构体的指针方法

```go
func (c Circle) expand() {
    c.Radius *= 2
}

func (c *Circle) expand2() {
    c.Radius *= 2
}
```

1.6 struct-结构体中的方法-示例3

代码：1-6.go

```
type Person struct {
    Name string
    Age  int
}

func (p Person) Getname() string { //p代表结构体本身的实列，类似python中的self,这里p可以写为self
    fmt.Println(p.Name)
    return p.Name
}

func main() {
    var person1 = new(Person)
    person1.Age = 34
    person1.Name = "darren"
    person1.Getname() // darren
}
```

1.6 struct-结构体中的方法-方法和函数的区别

- 方法只能被其接受者调用
- 接收者是指针时，方法可以改变接受者的值（或状态），函数也能做到
- 接受者和方法必须在同一个包内

1.6 struct-结构体中的方法-方法的接受者是值或者指针的区别

- 当接受者是一个值的时候，这个值是该类型实例的拷贝
- 如果想要方法改变接受者的数据，就在接受者的指针类型上定义该方法。否则，就在普通的值类型上定义方法。

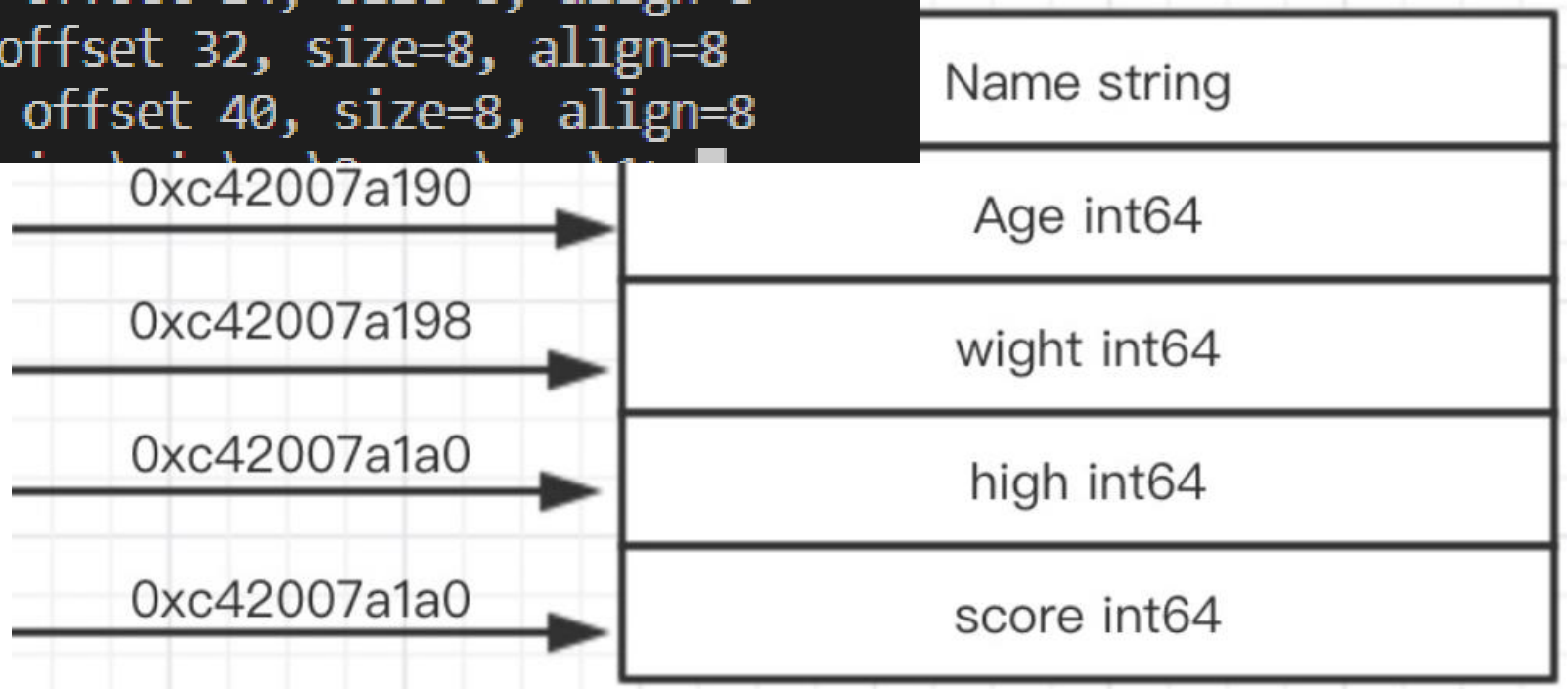
总结：指针方法和值方法都可以在指针或者非指针上被调用。也就是说，方法接收者是指针类型时，指针类型的值也是调用这个方法，反之亦然。

1.7 struct-内存分布

go中的结构体内存布局和

```
type Student s
    Name  stri
    Age   int64
    wight int64
    high  int64
    score int64
}
```

```
Struct is 48 bytes long
Name at offset 0, size=16, align=8
Age at offset 16, size=1, align=1
wight at offset 24, size=8, align=8
high at offset 32, size=8, align=8
score at offset 40, size=8, align=8
```



1-7.go 反射获取

2.1 interface简介

interface(接口)是golang最重要的特性之一，Interface类型可以定义一组方法，但是这些不需要实现。并且interface不能包含任何变量。

- interface 是方法的集合
- interface是一种类型，并且是指针类型
- interface的 **更重要的作用在于多态实现**
- **interface 不能包含任何变量**

2.2 interface定义

```
type  接口名称  interface {
    method1 (参数列表) 返回值列表
    method2 (参数列表) 返回值列表
…
}
```

2.3 interface使用

- 接口的使用不仅仅针对结构体，自定义类型、变量等等都可以实现接口。
- 如果一个接口没有任何方法，我们称为空接口，由于空接口没有方法，任意结构体都隐式地实现了空接口。
- 要实现一个接口，**必须实现该接口里面的所有方法**。

```go
//定义接口
type Skills interface {
    Running()
    Getname() string
}

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

```go
// 实现接口
func (p Student) Getname() string { //实现Getname方法
    fmt.Println(p.Name)
    return p.Name
}

func (p Student) Running() { // 实现 Running方法
    fmt.Printf("%s running", p.Name)
}
```

```go
// 使用接口
func main() {
    var skill Skills
    var stu1 Student
    stu1.Name = "darren"
    stu1.Age = 34
    skill = stu1
    skill.Running() //调用接口
}
```

2-3-0.go 2-3-1.go

2.4 interface多态

- go语言中interface是实现多态的一种形式，所谓多态，就是一种事物的多种形态
- 同一个interface，不同的类型实现，都可以进行调用，它们都按照统一接口进行操作

2-4.go

2.5 interface接口嵌套

- go语言中的接口可以嵌套，可以理解为继承，子接口拥有父接口的所有方法
- 如果使用该子接口，必须将父接口和子接口的所有方法都实现

2-5.go

```
type Skills interface {
    Running()
    Getname() string
}

type Test interface {
    sleeping()
    Skills //继承Skills
}
```

2.6 interface接口组合

- 接口的定义也支持组合继承

```go
// 可以闻
type Smellable interface {
    smell()
}

// 可以吃
type Eatable interface {
    eat()
}

type Fruitable interface {
    Smellable
    Eatable
}
```

2-6.go

2.7 interface类型转换

由于接口是一般类型，当我们使用接口时候可能不知道它是那个类型实现的，
基本数据类型我们有对应的方法进行类型转换，当然接口类型也有类型转换。

```
var s int
var x interface

x = s
y , ok := x.(int)  //将interface 转为int,ok可省略 但是省略以后转换失败会报错，
true转换成功，false转换失败，并采用默认值
```

2.8 interface类型判断

```go
func TestType(items ...interface{}) {
    for k, v := range items {
        switch v.(type) {
        case string:
            fmt.Printf("type is string, %d[%v]\n", k, v)
        case bool:
            fmt.Printf("type is bool, %d[%v]\n", k, v)
        case int:
            fmt.Printf("type is int, %d[%v]\n", k, v)
        case float32, float64:
            fmt.Printf("type is float, %d[%v]\n", k, v)
        case Student:
            fmt.Printf("type is Student, %d[%v]\n", k, v)
        case *Student:
            fmt.Printf("type is Student, %d[%p]\n", k, v)
        }
    }
}
```

2-8.go

2.9 指向指针的接口变量

```go
type Rect struct {
    Width  int
    Height int
}

func main() {
    var a interface{}
    var r = Rect{50, 50}
    a = &r // 指向了结构体指针

    var rx = a.(*Rect) // 转换成指针类型
    r.Width = 100
    r.Height = 100
    fmt.Println("r:", r)
    fmt.Println("rx:", rx)
    fmt.Printf("rx:%p, r:%p\n", rx, &r)
}
```

2-9.go

3 reflect反射是什么，为什么需要反射

反射定义：在计算机科学中，反射是指计算机程序在运行时（Run time）可以访问、检测和修改它本身状态或行为的一种能力。用比喻来说，反射就是程序在运行的时候能够“观察”并且修改自己的行为。

GO 反射的意义：Go 语言的 ORM 库离不开它，Go 语言的 json 序列化库离不开它， fmt包字符串格式化离不开它，Go 语言的运行时更是离不开它。

反射的目标:

1. 获取变量的类型信息，例如这个类型的名称、占用字节数、所有的方法列表、所有的内部字段结构、它的底层存储类型等等。
2. 动态的修改变量内部字段值。比如 json 的反序列化，你有的是对象内部字段的名称和相应的值，你需要把这些字段的值循环填充到对象相应的字段里

3 reflect反射

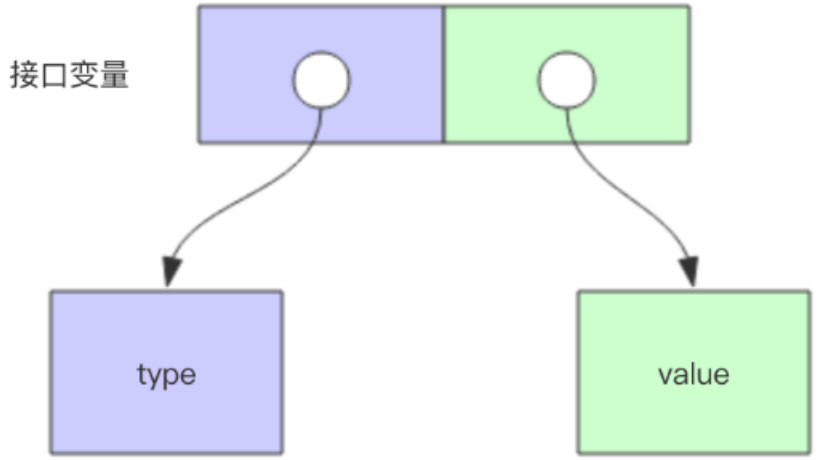
- go语言中的反射通过refect包实现，reflect包实现了运行时反射，允许程序操作任意类型的对象
- reflect包中的两个关键数据类Type和Value

```
func TypeOf(v interface{}) Type        // 返回类型 实际是接口
func ValueOf(v interface{}) Value      // 返回值 结构体
```

```go
var s int = 42
fmt.Println(reflect.TypeOf(s))
fmt.Println(reflect.ValueOf(s))
```

3-0.go

3 reflect反射- Type和Value



TypeOf() 方法返回变量的类型信息得到的是一个类型为 reflect.Type 的变量,
ValueOf() 方法返回变量的值信息得到的是一个类型为 reflect.Value 的变量。

3 reflect反射-利弊

反射的好处

1. 为了降低多写代码造成的bug率，做更好的归约和抽象
2. 为了灵活、好用、方便，做动态解析、调用和处理
3. 为了代码好看、易读、提高开发效率，补足与动态语言之间的一些差别

反射的弊端

1. 与反射相关的代码，经常是难以阅读的。在软件工程中，代码可读性也是一个非常重要的指标。
2. Go 语言作为一门静态语言，编码过程中，编译器能提前发现一些类型错误，但是对于反射代码是无能为力的。所以包含反射相关的代码，很可能会运行很久，才会出错，这时候经常是直接 panic，可能会造成严重的后果。
3. 反射对性能影响还是比较大的，比正常代码运行速度慢一到两个数量级。所以，对于一个项目中处于运行效率关键位置的代码，尽量避免使用反射特性。

3.1 reflect反射-Type

Type：Type类型用来表示一个go类型。

不是所有go类型的Type值都能使用所有方法。请参见每个方法的文档获取使用限制。在调用有分类限定的方法时，应先使用Kind方法获知类型的分类。调用该分类不支持的方法会导致运行时的panic。

获取Type对象的方法：

```
func TypeOf(i interface{}) Type
```

```
str := "darren"
res_type := reflect.TypeOf(str)
fmt.Println(res_type) //string

int1 := 1
res_type2 := reflect.TypeOf(int1)
fmt.Println(res_type2) //int
```

3-1.go

3.1 reflect反射-Type续-reflect.Type通用方法

```
func (t *rtype) String() string // 获取 t 类型的字符串描述，不要通过 String 来判断两种类型是否一致。
func (t *rtype) Name() string // 获取 t 类型在其包中定义的名称，未命名类型则返回空字符串。
func (t *rtype) PkgPath() string // 获取 t 类型所在包的名称，未命名类型则返回空字符串。
func (t *rtype) Kind() reflect.Kind // 获取 t 类型的类别。
func (t *rtype) Size() uintptr // 获取 t 类型的值在分配内存时的大小，功能和 unsafe.SizeOf 一样。
func (t *rtype) Align() int  // 获取 t 类型的值在分配内存时的字节对齐值。
func (t *rtype) FieldAlign() int  // 获取 t 类型的值作为结构体字段时的字节对齐值。
func (t *rtype) NumMethod() int  // 获取 t 类型的方法数量。
func (t *rtype) Method() reflect.Method  // 根据索引获取 t 类型的方法，如果方法不存在，则 panic。
// 如果 t 是一个实际的类型，则返回值的 Type 和 Func 字段会列出接收者。
// 如果 t 只是一个接口，则返回值的 Type 不列出接收者，Func 为空值。
func (t *rtype) MethodByName(string) (reflect.Method, bool) // 根据名称获取 t 类型的方法。
func (t *rtype) Implements(u reflect.Type) bool // 判断 t 类型是否实现了 u 接口。
func (t *rtype) ConvertibleTo(u reflect.Type) bool // 判断 t 类型的值可否转换为 u 类型。
func (t *rtype) AssignableTo(u reflect.Type) bool // 判断 t 类型的值可否赋值给 u 类型。
func (t *rtype) Comparable() bool // 判断 t 类型的值可否进行比较操作
//注意对于：数组、切片、映射、通道、指针、接口
func (t *rtype) Elem() reflect.Type // 获取元素类型、获取指针所指对象类型，获取接口的动态类型
```

3-1-1.go

3.1 reflect反射-Type续-reflect.Type其他方法

```
// 数值
func (t *rtype) Bits() int  // 获取数值类型的位宽，t 必须是整型、浮点型、复数型

// 数组
func (t *rtype) Len() int  // 获取数组的元素个数

// 映射
func (t *rtype) Key() reflect.Type // 获取映射的键类型

// 通道
func (t *rtype) ChanDir() reflect.ChanDir // 获取通道的方向

// 结构体
func (t *rtype) NumField() int  // 获取字段数量
func (t *rtype) Field(int) reflect.StructField  // 根据索引获取字段
func (t *rtype) FieldByName(string) (reflect.StructField, bool)  // 根据名称获取字段
func (t *rtype) FieldByNameFunc(match func(string) bool) (reflect.StructField, bool)  // 根据指定的匹配函数 math 获取字段
func (t *rtype) FieldByIndex(index []int) reflect.StructField  // 根据索引链获取嵌套字段

// 函数
func (t *rtype) NumIn() int // 获取函数的参数数量
func (t *rtype) In(int) reflect.Type // 根据索引获取函数的参数信息
func (t *rtype) NumOut() int // 获取函数的返回值数量
func (t *rtype) Out(int) reflect.Type // 根据索引获取函数的返回值信息
func (t *rtype) IsVariadic() bool  // 判断函数是否具有可变参数。
```

3-1-2.go

3.1 reflect反射-Type结构

```
// 基础类型 rtype 实现了 Type 接口
type rtype struct {
   size uintptr // 占用字节数
     ptrdata uintptr
   hash uint32 // 类型的hash值
     ...
   kind uint8 // 元类型
     ...
}

// 切片类型
type sliceType struct {
   rtype
   elem *rtype // 元素类型
}

// 结构体类型
type structType struct {
   rtype
   pkgPath name  // 所在包名
     fields []structField  // 字段列表
}
```

，rtype 实现了 Type 接口的所有方法。剩下的不同的部分信息各种特殊类型结构体都不一样。可以将 rtype 理解成父类，特殊类型的结构体是子类，会有一些不一样的字段信息。

源码路径：C:\go\src\reflect\type.go

3.1 reflect反射- reflect.Value方法

reflect.Value.Kind()：获取变量类别，返回常量

```
const (
    Invalid Kind = iota //不存在的类型
    Bool
    Int
    Int8
    Int16
    Int32
    Int64
    Uint
    Uint8
    Uint16
    Uint32
    Uint64
    Uintptr // 指针的整数类型
    Float32
    Float64
    Complex64
    Complex128
    Array
    Chan
    Func
    Interface
    Map
    Ptr
    Slice
    String
    Struct
    UnsafePointer
)
```

3-1-3.go

```
str := "darren"
val := reflect.ValueOf(str).Kind()
fmt.Println(val) //string
```

3.2 reflect反射- reflect.Value方法

获取值方法:

```
func (v Value) Int() int64 // 获取int类型值，如果 v 值不是有符号整型，则 panic。
func (v Value) Uint() uint64 // 获取unit类型的值，如果 v 值不是无符号整型（包括 uintptr），则 panic。
func (v Value) Float() float64 // 获取float类型的值，如果 v 值不是浮点型，则 panic。
func (v Value) Complex() complex128 // 获取复数类型的值，如果 v 值不是复数型，则 panic。
func (v Value) Bool() bool // 获取布尔类型的值，如果 v 值不是布尔型，则 panic。
func (v Value) Len() int // 获取 v 值的长度，v 值必须是字符串、数组、切片、映射、通道。
func (v Value) Cap() int  // 获取 v 值的容量，v 值必须是数值、切片、通道。
func (v Value) Index(i int) reflect.Value // 获取 v 值的第 i 个元素，v 值必须是字符串、数组、切片，i 不能超出范围。
func (v Value) Bytes() []byte // 获取字节类型的值，如果 v 值不是字节切片，则 panic。
func (v Value) Slice(i, j int) reflect.Value // 获取 v 值的切片，切片长度 = j - i，切片容量 = v.Cap() - i。
// v 必须是字符串、数值、切片，如果是数组则必须可寻址。i 不能超出范围。
func (v Value) Slice3(i, j, k int) reflect.Value  // 获取 v 值的切片，切片长度 = j - i，切片容量 = k - i。
// i、j、k 不能超出 v 的容量。i <= j <= k。
// v 必须是字符串、数值、切片，如果是数组则必须可寻址。i 不能超出范围。
func (v Value) MapIndex(key Value) reflect.Value // 根据 key 键获取 v 值的内容，v 值必须是映射。
// 如果指定的元素不存在，或 v 值是未初始化的映射，则返回零值（reflect.ValueOf(nil)）
func (v Value) MapKeys() []reflect.Value // 获取 v 值的所有键的无序列表，v 值必须是映射。
// 如果 v 值是未初始化的映射，则返回空列表。
func (v Value) OverflowInt(x int64) bool // 判断 x 是否超出 v 值的取值范围，v 值必须是有符号整型。
func (v Value) OverflowUint(x uint64) bool  // 判断 x 是否超出 v 值的取值范围，v 值必须是无符号整型。
func (v Value) OverflowFloat(x float64) bool  // 判断 x 是否超出 v 值的取值范围，v 值必须是浮点型。
func (v Value) OverflowComplex(x complex128) bool // 判断 x 是否超出 v 值的取值范围，v 值必须是复数型。
```

3.2 reflect反射- reflect.Value方法

设置值方法:

```
func (v Value) SetInt(x int64)  //设置int类型的值
func (v Value) SetUint(x uint64)  // 设置无符号整型的值
func (v Value) SetFloat(x float64) // 设置浮点类型的值
func (v Value) SetComplex(x complex128) //设置复数类型的值
func (v Value) SetBool(x bool) //设置布尔类型的值
func (v Value) SetString(x string) //设置字符串类型的值
func (v Value) SetLen(n int)  // 设置切片的长度，n 不能超出范围，不能为负数。
func (v Value) SetCap(n int) //设置切片的容量
func (v Value) SetBytes(x []byte) //设置字节类型的值
func (v Value) SetMapIndex(key, val reflect.Value) //设置map的key和value，前提必须是初始化以后，存在覆盖、不存在添加
```

3.2 reflect反射- reflect.Value方法

其他方法:

```
//结构体相关:
func (v Value) NumField() int // 获取结构体字段（成员）数量
func (v Value) Field(i int) reflect.Value  //根据索引获取结构体字段
func (v Value) FieldByIndex(index []int) reflect.Value // 根据索引链获取结构体嵌套字段
func (v Value) FieldByName(string) reflect.Value // 根据名称获取结构体的字段，不存在返回reflect.ValueOf(nil)
func (v Value) FieldByNameFunc(match func(string) bool) Value // 根据匹配函数 match 获取字段,如果没有匹配的字段，则返回零值（reflect.ValueOf(nil)）

//通道相关:
func (v Value) Send(x reflect.Value)// 发送数据（会阻塞），v 值必须是可写通道。
func (v Value) Recv() (x reflect.Value, ok bool) // 接收数据（会阻塞），v 值必须是可读通道。
func (v Value) TrySend(x reflect.Value) bool // 尝试发送数据（不会阻塞），v 值必须是可写通道。
func (v Value) TryRecv() (x reflect.Value, ok bool) // 尝试接收数据（不会阻塞），v 值必须是可读通道。
func (v Value) Close() // 关闭通道

//函数相关
func (v Value) Call(in []Value) (r []Value) // 通过参数列表 in 调用 v 值所代表的函数（或方法）。函数的返回值存入 r 中返回。
// 要传入多少参数就在 in 中存入多少元素。
// Call 即可以调用定参函数（参数数量固定），也可以调用变参函数（参数数量可变）。
func (v Value) CallSlice(in []Value) []Value // 调用变参函数
```

3.2 reflect反射- reflect.Value 结构体

```
type Value struct {
    typ *rtype  // 变量的类型结构体
    ptr unsafe.Pointer // 数据指针
    flag uintptr // 标志位
}
```

3.3 Go 语言官方的反射三大定律1

官方对 Go 语言的反射功能做了一个抽象的描述，总结出了三大定律

1. Reflection goes from interface value to reflection object.
2. Reflection goes from reflection object to interface value.
3. To modify a reflection object, the value must be settable.

第一个定律的意思是反射将接口变量转换成反射对象 Type 和 Value

```
func TypeOf(v interface{}) Type
func ValueOf(v interface{}) Value
```

第二个定律的意思是反射可以通过反射对象 Value 还原成原先的接口变量，这个指的就是 Value 结构体提供的 Interface() 方法。

```
func (v Value) Interface() interface{}
```

第三个定律的功能不是很好理解，它的意思是想用反射功能来修改一个变量的值，前提是这个值可以被修改。

3.3 Go 语言官方的反射三大定律2

值类型的变量是不可以通过反射来修改，因为在反射之前，传参的时候需要将值变量转换成接口变量，值内容会被浅拷贝，反射对象 Value 指向的数据内存地址不是原变量的内存地址，而是拷贝后的内存地址。这意味着如果值类型变量可以通过反射功能来修改，那么修改操作根本不会影响到原变量的值，那就白白修改了。所以 reflect 包就直接禁止了通过反射来修改值类型的变量。

3-3.go

```
var s int = 42
var v = reflect.ValueOf(s)
v.SetInt(43)
fmt.Println(s)
```

```
var s int = 42
// 反射指针类型
var v = reflect.ValueOf(&s)
// 要拿出指针指向的元素进行修改
v.Elem().SetInt(43)
fmt.Println(s)
```

两者区别

4 Go map实战

- go中的map是hash表的一个引用，类型写为：map[key]value，其中的key, value分别对应一种数据类型，如map[string]string
- 要求所有的key的数据类型相同，所有value数据类型相同(注：key与value可以有不同的数据类型，如果想不同则使用interface作为value)

map中的key的数据类型

- map中的每个key在keys的集合中是唯一的，而且需要支持 == or != 操作
- key的常用类型：int, rune, string, 结构体(每个元素需要支持 == or != 操作), 指针, 基于这些类型自定义的类型

float32/64 类型从语法上可以作为key类型，但是实际一般不作为key，因为其类型有误差

4.1 Go map实战-key的几种数据类型举例

见范例4-1.go

4. 2 map基本操作

map创建

两种创建的方式：一是通过字面值；二是通过make函数

4-2.go

map增删改查

增加，修改：
```
m["c"] = "11"
```
查：
```
v1 := m["x"]
v2, ok2 := m["x"]
```
删：
```
delete(m, "x")
```

map遍历

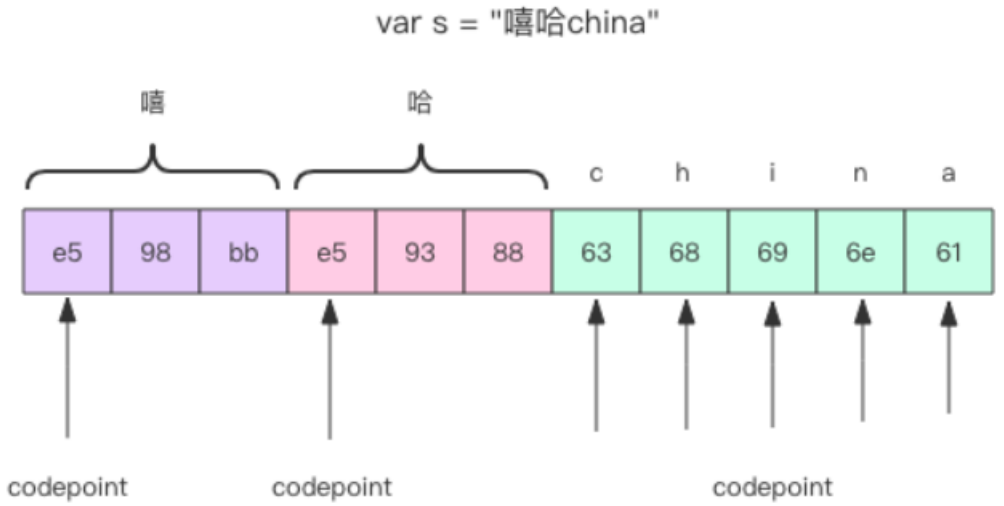
- 遍历的顺序是随机的
- 使用for range遍历的时候，k,v使用的同一块内存，这也是容易出现错误的地方

```
for k, v := range m { fmt.Printf("k:[%v].v:[%v]\n", k, v) //输出k,v值 }
```

5 Go string字符串

字符串通常有两种设计，一种是「字符」串，一种是「字节」串。「字符」串中的每个字都是定长的，而「字节」串中每个字是不定长的。Go 语言里的字符串是「字节」串，英文字符占用 1 个字节，非英文字符占多个字节。

type rune int32



其中 codepoint 是每个「字」的其实偏移量。Go 语言的字符串采用 utf8 编码，中文汉字通常需要占用 3 个字节，英文只需要 1 个字节。len() 函数得到的是字节的数量，通过下标来访问字符串得到的是「字节」。

5.1 Go string字符串-遍历

按字节遍历

```go
var s = "嘻哈china"
for i := 0; i < len(s); i++ {
    fmt.Printf("%x ", s[i])
}
```

按字符 rune 遍历

```go
var s = "嘻哈china"
for codepoint, runeValue := range s {
    fmt.Printf("[%d]: %x", codepoint, int32(runeValue))
}
```

5-2 Go string字节串的内存表示和操作