# reflect - 反射

## 关于静态类型

Go是静态类型语言，比如"int"、"float32"、"[]byte"等等。每个变量都有一个静态类型，且在编译时就确定了。 那么考虑一下如下一种类型声明:

type Myint int  
  
var i int  
var j Myint

Q: i 和j 类型相同吗？

A：i 和j类型是不同的。 二者拥有不同的静态类型，没有类型转换的话是不可以互相赋值的，尽管二者底层类型是一样的。

## 特殊的静态类型interface

interface类型是一种特殊的类型，它代表方法集合。 它可以存放任何实现了其方法的值。

经常被拿来举例的是io包里的这两个接口类型：

// Reader is the interface that wraps the basic Read method.  
type Reader interface {  
 Read(p []byte) (n int, err error)  
}  
  
// Writer is the interface that wraps the basic Write method.  
type Writer interface {  
 Write(p []byte) (n int, err error)  
}

任何类型，比如某struct，只要实现了其中的Read()方法就被认为是实现了Reader接口，只要实现了Write()方法，就被认为是实现了Writer接口，不过方法参数和返回值要跟接口声明的一致。

接口类型的变量可以存储任何实现该接口的值。

## 特殊的interface类型

最特殊的interface类型为空interface类型，即interface {}，前面说了，interface用来表示一组方法集合，所有实现该方法集合的类型都被认为是实现了该接口。那么空interface类型的方法集合为空，也就是说所有类型都可以认为是实现了该接口。

一个类型实现空interface并不重要，重要的是一个空interface类型变量可以存放所有值，记住是所有值，这才是最最重要的。 这也是有些人认为Go是动态类型的原因，这是个错觉。

## 反射三定律

* 反射可以将interface类型变量转换成反射对象

var x float64 = 3.4  
t := reflect.TypeOf(x) //t is reflext.Type  
fmt.Println(t) //输出：float64  
  
v := reflect.ValueOf(x) //v is reflext.Value  
fmt.Println(v) //输出：3.4

* 反射可以将反射对象还原成interface对象

var x float64 = 3.4  
v := reflect.ValueOf(x) //v is reflext.Value  
  
var y float64 = v.Interface().(float64)  
fmt.Println(y) //输出： 3.4

* 反射对象可修改，value值必须是可设置的

var x float64 = 3.4  
v := reflect.ValueOf(&x)  
v.SetFloat(7.1) // Error: will panic.

错误原因即是v是不可修改的。

反射对象是否可修改取决于其所存储的值，回想一下函数传参时是传值还是传址就不难理解上例中为何失败了。

上例中，传入reflect.ValueOf()函数的其实是x的值，而非x本身。即通过v修改其值是无法影响x的，也即是无效的修改，所以golang会报错。

想到此处，即可明白，如果构建v时使用x的地址就可实现修改了，但此时v代表的是指针地址，我们要设置的是指针所指向的内容，也即我们想要修改的是\*v。 那怎么通过v修改x的值呢？

reflect.Value提供了`Elem()`方法，可以获得指针向指向的value。看如下代码：

var x float64 = 3.4  
v := reflect.ValueOf(&x)  
v.Elem().SetFloat(7.1)  
fmt.Println(x) //输出：7.1

# 关于 reflect 包的详细解析

|  |  |
| --- | --- |
| **函数** | **说明** |
| **reflect.Type** | **func ArrayOf(length int, elem Type) Type**  转换为数组类型，如 ArrayOf(3, Type{int}) --> [3]int  **func ChanOf(dir ChanDir, t Type) Type**  转换为通道类型，如 int --> chan int / chan<- int / <-chan int  **func FuncOf(in, out []Type, variadic bool) Type**  转换为函数类型，如 FuncOf([]Type{int}, []Type{string}, false) --> func(int) string  **func MapOf(key, elem Type) Type**  转换为map类型，如 FuncOf(Type{int}, []Type{string}) --> map[int]string  **func PointerTo(t Type) Type**  **func PtrTo(t Type) Type**  转换为指针类型， 如 int --> \*int  **func SliceOf(t Type) Type**  转换为切片类型，如 int --> []int  **func StructOf(fields []StructField) Type**  转换为结构体类型  **func TypeOf(i any) Type**  将任意类型 interface{} 转为 reflect.Type 类型  fmt.Printf("%T\n", 3) // 相当于输出 reflect.Type.ToString() |
| **reflect.Value** | func Append(s Value, x ...Value) Value  func AppendSlice(s, t Value) Value  func Indirect(v Value) Value  func MakeChan(typ Type, buffer int) Value  func MakeFunc(typ Type, fn func(args []Value) (results []Value)) Value  func MakeMap(typ Type) Value  func MakeMapWithSize(typ Type, n int) Value  func MakeSlice(typ Type, len, cap int) Value  func New(typ Type) Value  func NewAt(typ Type, p unsafe.Pointer) Value  **func Select(cases []SelectCase) (chosen int, recv Value, recvOK bool)**  **func ValueOf(i any) Value**  返回基于 i 值的新 Value 类型  func Zero(typ Type) Value  **func (v Value) Addr() Value**  返回指向 v 的指针值的 Value 类型  **func (v Value) Bool() bool**  返回 v 的布尔值结果。如果 v 的 kind 类型不是boolean 类型则会panic  **func (v Value) Bytes() []byte**  返回 v 的字节数组结果。如果 v 的 kind 类型不是 []byte 类型则会panic  **func (v Value) Call(in []Value) []Value**  返回 v 的函数调用结果。如果 v 的 kind 类型不是 func 类型则会panic  **func (v Value) CallSlice(in []Value) []Value**  返回 v 的可变函数调用结果。如果 v 的 kind 类型不是 func 类型则会panic  **func (v Value) CanAddr() bool**  判断 v 是否可寻址，例如以 slice、array、\*struct、\*p等可寻址的类型的 Value。  如果 v 不可寻址，则调用 v.Addr() 会产生 panic  func (v Value) CanComplex() bool  **func (v Value) CanConvert(t Type) bool**  判断 v 是否可以转换为 Type类型的 t。  如果 v 不不支持 t 类型的转换，则调用 v.Convert(t) 会产生 panic  func (v Value) CanFloat() bool  func (v Value) CanInt() bool  func (v Value) CanInterface() bool  **func (v Value) CanSet() bool**  检查对应的v 是否是可取地址并可被修改  func (v Value) CanUint() bool  **func (v Value) Cap() int**  返回 v 的 cap 容量。如果 v 的 kind 类型不是array、slice、chan，则会产生panic  **func (v Value) Close()**  关闭 v 的通道。如果 v 的 kind 类型不是chan，则会产生panic  func (v Value) Complex() complex128  **func (v Value) Convert(t Type) Value**  将 v 转换为 Type类型为 t 的新 Value 类型  **func (v Value) Elem() Value**  返回v 对应的值或者 v （如果是指针）指向的值，将得到无法取地址的Value  **func (v Value) Field(i int) Value**  返回结构体第 i 个字段。如果 v 的 kind 类型不是结构体或者 i 超过范围，则会产生panic  **func (v Value) FieldByIndex(index []int) Value**  返回结构体的嵌套字段  func (v Value) FieldByIndexErr(index []int) (Value, error)  **func (v Value) FieldByName(name string) Value**  根据结构体的字段名称获取字段  func (v Value) FieldByNameFunc(match func(string) bool) Value  根据回调函数 match 获取匹配到的字段  func (v Value) Float() float64  func (v Value) Index(i int) Value  func (v Value) Int() int64  func (v Value) Interface() (i any)  返回 v 当前的值，类型为interface{}  func (v Value) IsNil() bool  func (v Value) IsValid() bool  func (v Value) IsZero() bool  func (v Value) Kind() Kind  func (v Value) Len() int  func (v Value) MapIndex(key Value) Value  func (v Value) MapKeys() []Value  func (v Value) MapRange() \*MapIter  func (v Value) Method(i int) Value  func (v Value) MethodByName(name string) Value  func (v Value) NumField() int  func (v Value) NumMethod() int  func (v Value) OverflowComplex(x complex128) bool  func (v Value) OverflowFloat(x float64) bool  func (v Value) OverflowInt(x int64) bool  func (v Value) OverflowUint(x uint64) bool  func (v Value) Pointer() uintptr  func (v Value) Recv() (x Value, ok bool)  func (v Value) Send(x Value)  func (v Value) Set(x Value)  func (v Value) SetBool(x bool)  func (v Value) SetBytes(x []byte)  func (v Value) SetCap(n int)  func (v Value) SetComplex(x complex128)  func (v Value) SetFloat(x float64)  func (v Value) SetInt(x int64)  func (v Value) SetIterKey(iter \*MapIter)  func (v Value) SetIterValue(iter \*MapIter)  func (v Value) SetLen(n int)  func (v Value) SetMapIndex(key, elem Value)  func (v Value) SetPointer(x unsafe.Pointer)  func (v Value) SetString(x string)  func (v Value) SetUint(x uint64)  func (v Value) Slice(i, j int) Value  func (v Value) Slice3(i, j, k int) Value  func (v Value) String() string  func (v Value) TryRecv() (x Value, ok bool)  func (v Value) TrySend(x Value) bool  func (v Value) Type() Type  func (v Value) Uint() uint64  func (v Value) UnsafeAddr() uintptr  func (v Value) UnsafePointer() unsafe.Pointer  fmt.Printf("%v\n", v) // 相当于输出 reflect.Value.ToString() |
| reflect.StructField | func VisibleFields(t Type) []StructField  func (f StructField) IsExported() bool |
| reflect.StructTag | func (tag StructTag) Get(key string) string  func (tag StructTag) Lookup(key string) (value string, ok bool) |

## **1. 为何需要反射?**

有时候我们需要编写一个函数能够处理一类并不满足普通公共接口的类型的值，也可能是因为它们并没有确定的表示方式，或者是在我们设计该函数的时候这些类型可能还不存在。

一个大家熟悉的例子是fmt.Fprintf函数提供的字符串格式化处理逻辑，它可以用来对任意类型的值格式化并打印，甚至支持用户自定义的类型。让我们也来尝试实现一个类似功能的函数。为了简单起见，我们的函数只接收一个参数，然后返回和fmt.Sprint类似的格式化后的字符串。我们实现的函数名也叫Sprint。

我们首先用switch类型分支来测试输入参数是否实现了String方法，如果是的话就调用该方法。然后继续增加类型测试分支，检查这个值的动态类型是否是string、int、bool等基础类型，并在每种情况下执行相应的格式化操作。

func Sprint(x interface{}) string {  
 type stringer interface {  
 String() string  
 }  
 switch x := x.(type) { //  
 case stringer:  
 return x.String()  
 case string:  
 return x  
 case int:  
 return strconv.Itoa(x)  
 // ...similar cases for int16, uint32, and so on...  
 case bool:  
 if x {  
 return "true"  
 }  
 return "false"  
 default:  
 // array, chan, func, map, pointer, slice, struct  
 return "???"  
 }  
}

但是我们如何处理其它类似[]float64、map[string][]string等类型呢？我们当然可以添加更多的测试分支，但是这些组合类型的数目基本是无穷的。还有如何处理类似url.Values这样的具名类型呢？即使类型分支可以识别出底层的基础类型是map[string][]string，但是它并不匹配url.Values类型，因为它们是两种不同的类型，而且switch类型分支也不可能包含每个类似url.Values的类型，这会导致对这些库的依赖。

没有办法来检查未知类型的表示方式，我们被卡住了。这就是我们为何需要反射的原因。

## **2. reflect.Type 和 reflect.Value**

反射是由 reflect 包提供的。它定义了两个重要的类型，Type 和 Value。一个 Type 表示一个Go类型。它是一个接口，有许多方法来区分类型以及检查它们的组成部分，例如一个结构体的成员或一个函数的参数等。唯一能反映 reflect.Type 实现的是接口的类型描述信息（§7.5），也正是这个实体标识了接口值的动态类型。

函数 reflect.TypeOf 接受任意的 interface{} 类型，并以 reflect.Type 形式返回其动态类型：

t := reflect.TypeOf(3) // a reflect.Type  
fmt.Println(t.String()) // "int"  
fmt.Println(t) // "int"

其中 TypeOf(3) 调用将值 3 传给 interface{} 参数。回到 7.5节 的将一个具体的值转为接口类型会有一个隐式的接口转换操作，它会创建一个包含两个信息的接口值：操作数的动态类型（这里是 int）和它的动态的值（这里是 3）。

因为 reflect.TypeOf 返回的是一个动态类型的接口值，它总是返回具体的类型。因此，下面的代码将打印 "\*os.File" 而不是 "io.Writer"。稍后，我们将看到能够表达接口类型的 reflect.Type。

var w io.Writer = os.Stdout  
fmt.Println(reflect.TypeOf(w)) // "\*os.File"

要注意的是 reflect.Type 接口是满足 fmt.Stringer 接口的。因为打印一个接口的动态类型对于调试和日志是有帮助的， fmt.Printf 提供了一个缩写 %T 参数，内部使用 reflect.TypeOf 来输出：

fmt.Printf("%T\n", 3) // "int"

reflect 包中另一个重要的类型是 Value。一个 reflect.Value 可以装载任意类型的值。函数 reflect.ValueOf 接受任意的 interface{} 类型，并返回一个装载着其动态值的 reflect.Value。和 reflect.TypeOf 类似，reflect.ValueOf 返回的结果也是具体的类型，但是 reflect.Value 也可以持有一个接口值。

v := reflect.ValueOf(3) // a reflect.Value  
fmt.Println(v) // "3"  
fmt.Printf("%v\n", v) // "3"  
fmt.Println(v.String()) // NOTE: "<int Value>"

和 reflect.Type 类似，reflect.Value 也满足 fmt.Stringer 接口，但是除非 Value 持有的是字符串，否则 String 方法只返回其类型。而使用 fmt 包的 %v 标志参数会对 reflect.Values 特殊处理。

对 Value 调用 Type 方法将返回具体类型所对应的 reflect.Type：

t := v.Type() // a reflect.Type  
fmt.Println(t.String()) // "int"

reflect.ValueOf 的逆操作是 reflect.Value.Interface 方法。它返回一个 interface{} 类型，装载着与 reflect.Value 相同的具体值：

v := reflect.ValueOf(3) // a reflect.Value  
x := v.Interface() // an interface{}  
i := x.(int) // an int  
fmt.Printf("%d\n", i) // "3"

reflect.Value 和 interface{} 都能装载任意的值。所不同的是，一个空的接口隐藏了值内部的表示方式和所有方法，因此只有我们知道具体的动态类型才能使用类型断言来访问内部的值（就像上面那样），内部值我们没法访问。相比之下，一个 Value 则有很多方法来检查其内容，无论它的具体类型是什么。让我们再次尝试实现我们的格式化函数 format.Any。

我们使用 reflect.Value 的 Kind 方法来替代之前的类型 switch。虽然还是有无穷多的类型，但是它们的 kinds 类型却是有限的：

* Bool、String 和 所有数字类型的基础类型；
* Array 和 Struct 对应的聚合类型；
* Chan、Func、Ptr、Slice 和 Map 对应的引用类型；
* interface 类型；
* 还有表示空值的 Invalid 类型。（空的 reflect.Value 的 kind 即为 Invalid。）

*gopl.io/ch12/format*

package format  
import (  
 "reflect"  
 "strconv"  
)  
// Any formats any value as a string.  
func Any(value interface{}) string {  
 return formatAtom(reflect.ValueOf(value))  
}  
// formatAtom formats a value without inspecting its internal structure.  
func formatAtom(v reflect.Value) string {  
 switch v.Kind() {  
 case reflect.*Invalid*:  
 return "invalid"  
 case reflect.*Int*, reflect.*Int8*, reflect.*Int16*,  
 reflect.*Int32*, reflect.*Int64*:  
 return strconv.FormatInt(v.Int(), 10)  
 case reflect.*Uint*, reflect.*Uint8*, reflect.*Uint16*,  
 reflect.*Uint32*, reflect.*Uint64*, reflect.*Uintptr*:  
 return strconv.FormatUint(v.Uint(), 10)  
 // ...floating-point and complex cases omitted for brevity...  
 case reflect.*Bool*:  
 return strconv.FormatBool(v.Bool())  
 case reflect.*String*:  
 return strconv.Quote(v.String())  
 case reflect.*Chan*, reflect.*Func*, reflect.*Ptr*, reflect.*Slice*, reflect.*Map*:  
 return v.Type().String() + " 0x" +  
 strconv.FormatUint(uint64(v.Pointer()), 16)  
 default: // reflect.Array, reflect.Struct, reflect.Interface  
 return v.Type().String() + " value"  
 }  
}

到目前为止，我们的函数将每个值视作一个不可分割没有内部结构的物品，因此它叫 formatAtom。对于聚合类型（结构体和数组）和接口，只是打印值的类型，对于引用类型（channels、functions、pointers、slices 和 maps），打印类型和十六进制的引用地址。虽然还不够理想，但是依然是一个重大的进步，并且 Kind 只关心底层表示，format.Any 也支持具名类型。例如：

var x int64 = 1var d time.Duration = 1 \* time.*Nanosecond*fmt.Println(format.Any(x)) // "1"  
fmt.Println(format.Any(d)) // "1"  
fmt.Println(format.Any([]int64{x})) // "[]int64 0x8202b87b0"  
fmt.Println(format.Any([]time.Duration{d})) // "[]time.Duration 0x8202b87e0"

## **3. 通过reflect.Value修改值**

到目前为止，反射还只是程序中变量的另一种读取方式。然而，在本节中我们将重点讨论如何通过反射机制来修改变量。

回想一下，Go语言中类似x、x.f[1]和\*p形式的表达式都可以表示变量，但是其它如x + 1和f(2)则不是变量。一个变量就是一个可寻址的内存空间，里面存储了一个值，并且存储的值可以通过内存地址来更新。

对于reflect.Values也有类似的区别。有一些reflect.Values是可取地址的；其它一些则不可以。考虑以下的声明语句：

x := 2 // value type variable?  
a := reflect.ValueOf(2) // 2 int no  
b := reflect.ValueOf(x) // 2 int no  
c := reflect.ValueOf(&x) // &x \*int no  
d := c.Elem() // 2 int yes (x)

其中a对应的变量不可取地址。因为a中的值仅仅是整数2的拷贝副本。b中的值也同样不可取地址。c中的值还是不可取地址，它只是一个指针&x的拷贝。实际上，所有通过reflect.ValueOf(x)返回的reflect.Value都是不可取地址的。但是对于d，它是c的解引用方式生成的，指向另一个变量，因此是可取地址的。我们可以通过调用reflect.ValueOf(&x).Elem()，来获取任意变量x对应的可取地址的Value。

我们可以通过调用reflect.Value的CanAddr方法来判断其是否可以被取地址：

fmt.Println(a.CanAddr()) // "false"  
fmt.Println(b.CanAddr()) // "false"  
fmt.Println(c.CanAddr()) // "false"  
fmt.Println(d.CanAddr()) // "true"

每当我们通过指针间接地获取的reflect.Value都是可取地址的，即使开始的是一个不可取地址的Value。在反射机制中，所有关于是否支持取地址的规则都是类似的。例如，slice的索引表达式e[i]将隐式地包含一个指针，它就是可取地址的，即使开始的e表达式不支持也没有关系。以此类推，reflect.ValueOf(e).Index(i)对应的值也是可取地址的，即使原始的reflect.ValueOf(e)不支持也没有关系。

要从变量对应的可取地址的reflect.Value来访问变量需要三个步骤。第一步是调用Addr()方法，它返回一个Value，里面保存了指向变量的指针。然后是在Value上调用Interface()方法，也就是返回一个interface{}，里面包含指向变量的指针。最后，如果我们知道变量的类型，我们可以使用类型的断言机制将得到的interface{}类型的接口强制转为普通的类型指针。这样我们就可以通过这个普通指针来更新变量了：

x := 2  
d := reflect.ValueOf(&x).Elem() // d refers to the variable x  
px := d.Addr().Interface().(\*int) // px := &x  
\*px = 3 // x = 3  
fmt.Println(x) // "3"

或者，不使用指针，而是通过调用可取地址的reflect.Value的reflect.Value.Set方法来更新对应的值：

d.Set(reflect.ValueOf(4))  
fmt.Println(x) // "4"

Set方法将在运行时执行和编译时进行类似的可赋值性约束的检查。以上代码，变量和值都是int类型，但是如果变量是int64类型，那么程序将抛出一个panic异常，所以关键问题是要确保改类型的变量可以接受对应的值：

d.Set(reflect.ValueOf(int64(5))) // panic: int64 is not assignable to int

同样，对一个不可取地址的reflect.Value调用Set方法也会导致panic异常：

x := 2  
b := reflect.ValueOf(x)  
b.Set(reflect.ValueOf(3)) // panic: Set using unaddressable value

这里有很多用于基本数据类型的Set方法：SetInt、SetUint、SetString和SetFloat等。

d := reflect.ValueOf(&x).Elem()  
d.SetInt(3)  
fmt.Println(x) // "3"

从某种程度上说，这些Set方法总是尽可能地完成任务。以SetInt为例，只要变量是某种类型的有符号整数就可以工作，即使是一些命名的类型、甚至只要底层数据类型是有符号整数就可以，而且如果对于变量类型值太大的话会被自动截断。但需要谨慎的是：对于一个引用interface{}类型的reflect.Value调用SetInt会导致panic异常，即使那个interface{}变量对于整数类型也不行。

x := 1  
rx := reflect.ValueOf(&x).Elem()  
rx.SetInt(2) // OK, x = 2  
rx.Set(reflect.ValueOf(3)) // OK, x = 3  
rx.SetString("hello") // panic: string is not assignable to int  
rx.Set(reflect.ValueOf("hello")) // panic: string is not assignable to int  
var y interface{}  
ry := reflect.ValueOf(&y).Elem()  
ry.SetInt(2) // panic: SetInt called on interface Value  
ry.Set(reflect.ValueOf(3)) // OK, y = int(3)  
ry.SetString("hello") // panic: SetString called on interface Value  
ry.Set(reflect.ValueOf("hello")) // OK, y = "hello"

当我们用Display显示os.Stdout结构时，我们发现反射可以越过Go语言的导出规则的限制读取结构体中未导出的成员，比如在类Unix系统上os.File结构体中的fd int成员。然而，利用反射机制并不能修改这些未导出的成员：

stdout := reflect.ValueOf(os.Stdout).Elem() // \*os.Stdout, an os.File var  
fmt.Println(stdout.Type()) // "os.File"  
fd := stdout.FieldByName("fd")  
fmt.Println(fd.Int()) // "1"  
fd.SetInt(2) // panic: unexported field

一个可取地址的reflect.Value会记录一个结构体成员是否是未导出成员，如果是的话则拒绝修改操作。因此，CanAddr方法并不能正确反映一个变量是否是可以被修改的。另一个相关的方法CanSet是用于检查对应的reflect.Value是否是可取地址并可被修改的：

fmt.Println(fd.CanAddr(), fd.CanSet()) // "true false"

## **4. 获取结构体字段标签**

对于一个web服务，大部分HTTP处理函数要做的第一件事情就是展开请求中的参数到本地变量中。我们定义了一个工具函数，叫params.Unpack，通过使用结构体成员标签机制来让HTTP处理函数解析请求参数更方便。

首先，我们看看如何使用它。下面的search函数是一个HTTP请求处理函数。它定义了一个匿名结构体类型的变量，用结构体的每个成员表示HTTP请求的参数。其中结构体成员标签指明了对于请求参数的名字，为了减少URL的长度这些参数名通常都是神秘的缩略词。Unpack将请求参数填充到合适的结构体成员中，这样我们可以方便地通过合适的类型类来访问这些参数。

*gopl.io/ch12/search*

import (  
 "fmt"  
 "gopl.io/ch12/params"  
 "net/http"  
)  
// search implements the /search URL endpoint.  
func search(resp http.ResponseWriter, req \*http.Request) {  
 var data struct {  
 Labels []string `http:"l"`  
 MaxResults int `http:"max"`  
 Exact bool `http:"x"`  
 }  
 data.MaxResults = 10 // set default  
 if err := params.Unpack(req, &data); err != nil {  
 http.Error(resp, err.Error(), http.*StatusBadRequest*) // 400  
 return  
 }  
 // ...rest of handler...  
 fmt.Fprintf(resp, "Search: %+v\n", data)  
}

下面的Unpack函数主要完成三件事情。第一，它调用req.ParseForm()来解析HTTP请求。然后，req.Form将包含所有的请求参数，不管HTTP客户端使用的是GET还是POST请求方法。

下一步，Unpack函数将构建每个结构体成员有效参数名字到成员变量的映射。如果结构体成员有成员标签的话，有效参数名字可能和实际的成员名字不相同。reflect.Type的Field方法将返回一个reflect.StructField，里面含有每个成员的名字、类型和可选的成员标签等信息。其中成员标签信息对应reflect.StructTag类型的字符串，并且提供了Get方法用于解析和根据特定key提取的子串，例如这里的http:"..."形式的子串。

*gopl.io/ch12/params*

import (  
 "fmt"  
 "net/http"  
 "reflect"  
 "strings"  
)  
  
// Unpack populates the fields of the struct pointed to by ptr// from the HTTP request parameters in req.  
func Unpack(req \*http.Request, ptr interface{}) error {  
 if err := req.ParseForm(); err != nil {  
 return err  
 }  
  
 // Build map of fields keyed by effective name.  
 fields := make(map[string]reflect.Value)  
 v := reflect.ValueOf(ptr).Elem() // the struct variable  
 for i := 0; i < v.NumField(); i++ {  
 fieldInfo := v.TypeOf().Field(i) // a reflect.StructField  
 tag := fieldInfo.Tag // a reflect.StructTag  
 name := tag.Get("http")  
 if name == "" {  
 name = strings.ToLower(fieldInfo.Name)  
 }  
 fields[name] = v.Field(i)  
 }  
  
 // Update struct field for each parameter in the request.  
 for name, values := range req.Form {  
 f := fields[name]  
 if !f.IsValid() {  
 continue // ignore unrecognized HTTP parameters  
 }  
 for \_, value := range values {  
 if f.Kind() == reflect.*Slice* {  
 elem := reflect.New(f.Type().Elem()).Elem()  
 if err := populate(elem, value); err != nil {  
 return fmt.Errorf("%s: %v", name, err)  
 }  
 f.Set(reflect.Append(f, elem))  
 } else {  
 if err := populate(f, value); err != nil {  
 return fmt.Errorf("%s: %v", name, err)  
 }  
 }  
 }  
 }  
 return nil  
}

最后，Unpack遍历HTTP请求的name/valu参数键值对，并且根据更新相应的结构体成员。回想一下，同一个名字的参数可能出现多次。如果发生这种情况，并且对应的结构体成员是一个slice，那么就将所有的参数添加到slice中。其它情况，对应的成员值将被覆盖，只有最后一次出现的参数值才是起作用的。

populate函数小心用请求的字符串类型参数值来填充单一的成员v（或者是slice类型成员中的单一的元素）。目前，它仅支持字符串、有符号整数和布尔型。其中其它的类型将留做练习任务。

func populate(v reflect.Value, value string) error {  
 switch v.Kind() {  
 case reflect.*String*:  
 v.SetString(value)  
 case reflect.*Int*:  
 i, err := strconv.ParseInt(value, 10, 64)  
 if err != nil {  
 return err  
 }  
 v.SetInt(i)  
 case reflect.*Bool*:  
 b, err := strconv.ParseBool(value)  
 if err != nil {  
 return err  
 }  
 v.SetBool(b)  
 default:  
 return fmt.Errorf("unsupported kind %s", v.Type())  
 }  
 return nil  
}

如果我们上上面的处理程序添加到一个web服务器，则可以产生以下的会话：

$ go build gopl.io/ch12/search  
$ ./search &  
$ ./fetch 'http://localhost:12345/search'  
Search: {Labels:[] MaxResults:10 Exact:false}  
$ ./fetch 'http://localhost:12345/search?l=golang&l=programming'  
Search: {Labels:[golang programming] MaxResults:10 Exact:false}  
$ ./fetch 'http://localhost:12345/search?l=golang&l=programming&max=100'  
Search: {Labels:[golang programming] MaxResults:100 Exact:false}  
$ ./fetch 'http://localhost:12345/search?x=true&l=golang&l=programming'  
Search: {Labels:[golang programming] MaxResults:10 Exact:true}  
$ ./fetch 'http://localhost:12345/search?q=hello&x=123'  
x: strconv.ParseBool: parsing "123": invalid syntax  
$ ./fetch 'http://localhost:12345/search?q=hello&max=lots'  
max: strconv.ParseInt: parsing "lots": invalid syntax

****练习 12.11：**** 编写相应的Pack函数，给定一个结构体值，Pack函数将返回合并了所有结构体成员和值的URL。

****练习 12.12：**** 扩展成员标签以表示一个请求参数的有效值规则。例如，一个字符串可以是有效的email地址或一个信用卡号码，还有一个整数可能需要是有效的邮政编码。修改Unpack函数以检查这些规则。

****练习 12.13：**** 修改S表达式的编码器（§12.4）和解码器（§12.6），采用和encoding/json包（§4.5）类似的方式使用成员标签中的sexpr:"..."字串。

## **5. 显示一个类型的方法集**

我们的最后一个例子是使用reflect.Type来打印任意值的类型和枚举它的方法：

*gopl.io/ch12/methods*

// Print prints the method set of the value x.  
func Print(x interface{}) {  
 v := reflect.ValueOf(x)  
 t := v.Type()  
 fmt.Printf("type %s\n", t)  
 for i := 0; i < v.NumMethod(); i++ {  
 methType := v.Method(i).Type()  
 fmt.Printf("func (%s) %s%s\n", t, t.Method(i).Name,  
 strings.TrimPrefix(methType.String(), "func"))  
 }  
}

reflect.Type和reflect.Value都提供了一个Method方法。每次t.Method(i)调用将一个reflect.Method的实例，对应一个用于描述一个方法的名称和类型的结构体。每次v.Method(i)方法调用都返回一个reflect.Value以表示对应的值（§6.4），也就是一个方法是帮到它的接收者的。使用reflect.Value.Call方法（我们这里没有演示），将可以调用一个Func类型的Value，但是这个例子中只用到了它的类型。

这是属于time.Duration和\*strings.Replacer两个类型的方法：

methods.Print(time.*Hour*)  
// Output:  
// type time.Duration  
// func (time.Duration) Hours() float64  
// func (time.Duration) Minutes() float64  
// func (time.Duration) Nanoseconds() int64  
// func (time.Duration) Seconds() float64  
// func (time.Duration) String() string  
  
methods.Print(new(strings.Replacer))  
// Output:// type \*strings.Replacer  
// func (\*strings.Replacer) Replace(string) string  
// func (\*strings.Replacer) WriteString(io.Writer, string) (int, error)

## **6. 几点忠告**

虽然反射提供的API远多于我们讲到的，我们前面的例子主要是给出了一个方向，通过反射可以实现哪些功能。反射是一个强大并富有表达力的工具，但是它应该被小心地使用，原因有三。

第一个原因是，基于反射的代码是比较脆弱的。对于每一个会导致编译器报告类型错误的问题，在反射中都有与之相对应的误用问题，不同的是编译器会在构建时马上报告错误，而反射则是在真正运行到的时候才会抛出panic异常，可能是写完代码很久之后了，而且程序也可能运行了很长的时间。

以前面的readList函数（§12.6）为例，为了从输入读取字符串并填充int类型的变量而调用的reflect.Value.SetString方法可能导致panic异常。绝大多数使用反射的程序都有类似的风险，需要非常小心地检查每个reflect.Value的对应值的类型、是否可取地址，还有是否可以被修改等。

避免这种因反射而导致的脆弱性的问题的最好方法，是将所有的反射相关的使用控制在包的内部，如果可能的话避免在包的API中直接暴露reflect.Value类型，这样可以限制一些非法输入。如果无法做到这一点，在每个有风险的操作前指向额外的类型检查。以标准库中的代码为例，当fmt.Printf收到一个非法的操作数时，它并不会抛出panic异常，而是打印相关的错误信息。程序虽然还有BUG，但是会更加容易诊断。

fmt.Printf("%d %s\n", "hello", 42) // "%!d(string=hello) %!s(int=42)"

反射同样降低了程序的安全性，还影响了自动化重构和分析工具的准确性，因为它们无法识别运行时才能确认的类型信息。

避免使用反射的第二个原因是，即使对应类型提供了相同文档，但是反射的操作不能做静态类型检查，而且大量反射的代码通常难以理解。总是需要小心翼翼地为每个导出的类型和其它接受interface{}或reflect.Value类型参数的函数维护说明文档。

第三个原因，基于反射的代码通常比正常的代码运行速度慢一到两个数量级。对于一个典型的项目，大部分函数的性能和程序的整体性能关系不大，所以当反射能使程序更加清晰的时候可以考虑使用。测试是一个特别适合使用反射的场景，因为每个测试的数据集都很小。但是对于性能关键路径的函数，最好避免使用反射。