**反射 reflect**

反射在go中是一个包，报名为reflect，其作用是可以查看一个变量的具体类型和值。

反射这个包提供了两种接口类型 Type 和 Value，分别用于记录一个变量的类型和具体值。

下面我们看看一个简单的反射的应用：获取变量的类型

func main(){

var x io.Writer

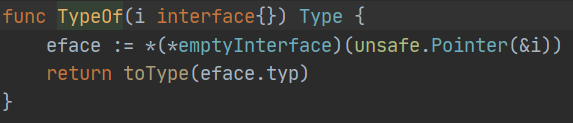
x = os.Stdout

typ := reflect.TypeOf(x)

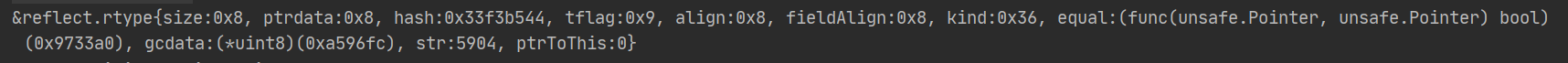
fmt.Println(typ) // \*os.File

}

TypeOf源码如下



我们不用关注他的方法体，而是关注他的返回值，他返回的是一个reflect.Type类型。因此如果我们typ不是一个简单的字符串，而是一个reflect.Type类型，typ的内容如下：



reflect.Type和reflect.Value接口类型变量实现了Stringer接口类型，因此我们可以调用其String()方法得到字符串形式的类型信息：

typ\_str := typ.String()

一个注意点是：reflect.Type记录的不是变量的接口类型而是变量的动态类型，因此typ是"\*os.File" 而不是 "io.Writer"

实际上我们使用下面的方式也一样可以打印出x变量的类型。

fmt.Printf("%T", x)

或者

typ := fmt.Sprintf("%T", x)

fmt.Printf("%#v", typ)

其实使用 %T，其内部还是使用了 reflect.TypeOf() 。当然了 TypeOf() 返回的是reflect.Type类型，而Sprintf(“%T”,x)返回的是字符串类型。

reflect.Value这个对象不仅记录一个变量的值还记录了类型，通过ValueOf方法我们可以得到一个reflect.Value对象。

func main(){

x := 3

x\_val := reflect.ValueOf(x)

x\_type\_str := x\_val.String()

fmt.Println(x\_val) // 3

fmt.Println(x\_type\_str) // <int Value>

x\_type := x\_val.Type()

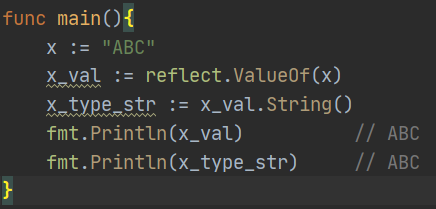
fmt.Println(x\_type) // int, 是reflect.Type类型

fmt.Printf("%#v", x\_type)

}

只需对 Value 对象调用 String() 方法或者 Type() 方法就能得到x的类型，所以说Value这个接口类型不仅记录了变量的具体值，还记录了其类型。

但是如果x是一个字符串，那么无论是 x\_val 还是 x\_val.String() 都是具体值。



reflect.Value.Interface()方法可以将一个reflect.Value类型转变为一个interface{} 空接口类型，它相当于是reflect.ValueOf()的逆操作。

func main(){

var w io.Writer

w = os.Stdout

w\_val := reflect.ValueOf(w)

fmt.Printf("%#v", w\_val)

w\_interface := w\_val.Interface() // 返回一个interface{}空接口类型而不是io.Writer类型

w\_interface.Write() // 报错，因为此时w\_interface的所有方法都被屏蔽，自然也没有Write方法

}

w\_interface变量装着和w\_val以及w一样的具体值，只是类型变成了空接口, 因此这个变量的所有方法都会被隐藏，我们只关心它的具体值就好。

最后，我们回想一下之前学习接口的时候，我们是通过类型断言来判断一个变量的类型。但是美中不足的是这种方式只能够判断一个变量是不是某种类型，但是不能直接得到变量的类型是什么，而reflect很好的解决了这个问题。

除此之外reflect.Value和reflect.Type还提供了Method方法可以一种类型有哪些方法：

func main(){

var w io.Writer

w = os.Stdout

w\_val := reflect.ValueOf(w)

w\_type := w\_val.Type()

fmt.Printf("%s\n", w\_val.Method(0)) // 返回reflect.Value类型

fmt.Printf("%s\n", w\_val.Method(0).Type())

fmt.Printf("%s\n", w\_type.Method(0)) // 返回reflect.Method类型

fmt.Printf("%s\n", w\_type.Method(0).Type) // 返回reflect.Type类型

}

reflect.Value 和 reflect.Type 这两种类型都有 Method 方法，需要传入一个下标，就可以获取这个下标对应的方法A。Value的Mathod方法返回的是这个方法A的reflect.Value类型表示方法A的具体值，而Type的Method方法返回的是一个reflect.Method类型。

reflect.Method类型的源码如下

type Method struct {

// Name is the method name.

// PkgPath is the package path that qualifies a lower case (unexported)

// method name. It is empty for upper case (exported) method names.

// The combination of PkgPath and Name uniquely identifies a method

// in a method set.

// See https://golang.org/ref/spec#Uniqueness\_of\_identifiers

Name string

PkgPath string

Type Type // method type

Func Value // func with receiver as first argument

Index int // index for Type.Method

}

如果想要获取方法A的方法名和类型，调用Method的Name成员和Type成员即可。

下面我们可以做一个获取某个变量所有方法的函数

func main(){

var w io.Writer

w = os.Stdout

w\_methods := getAllMethod(w)

fmt.Printf("%#v", w\_methods)

}

func getAllMethod(x interface{}) (methods map[string]string){

methods = map[string]string{}

x\_val := reflect.ValueOf(x)

x\_type := x\_val.Type()

// 遍历x的所有方法

for i := 0; i < x\_val.NumMethod(); i++{

x\_method := x\_type.Method(i)

methods[x\_method.Name] = x\_method.Type.String()

}

return methods

}

**慎用反射**

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。

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。

**包 package**

每个包是由一个全局唯一的字符串所标识的导入路径定位。出现在import语句中的导入路径也是字符串。

import (

"fmt"

"math/rand"

"encoding/json"

"golang.org/x/net/html"

"github.com/go-sql-driver/mysql"

)

Go语言的规范并没有指明包的导入路径字符串的具体含义，导入路径的具体含义是由构建工具来解释的。在本章，我们将深入讨论Go语言工具箱的功能，包括大家经常使用的构建测试等功能。当然，也有第三方扩展的工具箱存在。例如，Google公司内部的Go语言码农，他们就使用内部的多语言构建系统，用不同的规则来处理包名字和定位包，用不同的规则来处理单元测试等等，因为这样可以更紧密适配他们内部环境。

如果你计划分享或发布包，那么导入路径最好是全球唯一的。为了避免冲突，所有非标准库包的导入路径建议以**所在组织的互联网域名为前缀**；而且这样也有利于包的检索。例如，上面的import语句导入了Go团队维护的HTML解析器和一个流行的第三方维护的MySQL驱动。

**包声明**

在每个Go语言源文件的开头都必须有包声明语句。包声明语句的主要目的是确定当前包被其它包导入时默认的标识符（也称为包名）。

例如，math/rand包的每个源文件的开头都包含package rand包声明语句，所以当你导入这个包，你就可以用rand.Int、rand.Float64类似的方式访问包的成员。

通常来说，默认的包名就是包导入路径名的最后一段，因此即使两个包的导入路径不同，它们依然可能有一个相同的包名。例如，math/rand包和crypto/rand包的包名都是rand。稍后我们将看到如何同时导入两个有相同包名的包。

关于默认包名一般采用导入路径名的最后一段的约定也有三种例外情况。第一个例外，包对应一个可执行程序，也就是main包，这时候main包本身的导入路径是无关紧要的。名字为main的包是给go build（§10.7.3）构建命令一个信息，这个包编译完之后必须调用连接器生成一个可执行程序。

第二个例外，包所在的目录中可能有一些文件名是以\_test.go为后缀的Go源文件（译注：前面必须有其它的字符，因为以\_或.开头的源文件会被构建工具忽略），并且这些源文件声明的包名也是以\_test为后缀名的。这种目录可以包含两种包：一种是普通包，另一种则是测试的外部扩展包。所有以\_test为后缀包名的测试外部扩展包都由go test命令独立编译，普通包和测试的外部扩展包是相互独立的。测试的外部扩展包一般用来避免测试代码中的循环导入依赖

第三个例外，一些依赖版本号的管理工具会在导入路径后追加版本号信息，例如“gopkg.in/yaml.v2”。这种情况下包的名字并不包含版本号后缀，而是yaml。

**导入包**

可以在一个Go语言源文件包声明语句之后，其它非导入声明语句之前，包含零到多个导入包声明语句。每个导入声明可以单独指定一个导入路径，也可以通过圆括号同时导入多个导入路径。下面两个导入形式是等价的，但是第二种形式更为常见。

import "fmt"

import "os"

import (

"fmt"

"os"

)

导入的包之间可以通过添加空行来分组；通常将来自不同组织的包独自分组。包的导入顺序无关紧要，但是在每个分组中一般会根据字符串顺序排列。（gofmt和goimports工具都可以将不同分组导入的包独立排序。）

import (

"fmt"

"html/template"

"os"

"golang.org/x/net/html"

"golang.org/x/net/ipv4"

)

如果我们想同时导入两个有着名字相同的包，例如math/rand包和crypto/rand包，那么导入声明必须至少为一个同名包指定一个新的包名以避免冲突。这叫做导入包的重命名。

import (

"crypto/rand"

mrand "math/rand" // alternative name mrand avoids conflict

)

如果遇到包循环导入的情况，Go语言的构建工具将报告错误。

**包的匿名导入**

如果只是导入一个包而并不使用导入的包（的变量或者方法）将会导致一个编译错误。

但是有时候我们只是想利用导入包而产生的副作用：它会运行包级变量的初始化表达式和执行导入包的init初始化函数。此时我们可以用下划线\_来重命名导入的包。

import \_ "image/png" // register PNG decoder

**工具**

接下来将讨论Go语言工具箱的具体功能，包括如何下载、格式化、构建、测试和安装Go语言编写的程序。

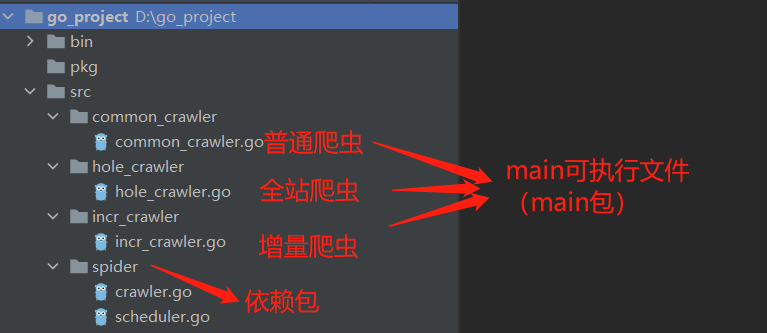
对于大多数的Go语言用户，只需要配置一个名叫GOPATH的环境变量，用来指定当前工作目录即可。当需要切换到不同工作区的时候，只要更新GOPATH就可以了（我理解为是要重新修改GOOTPATH环境变量为新的根目录，而不是追加）。第二个环境变量GOROOT用来指定Go的安装目录，还有它自带的标准库包的位置。

export GOPATH=/root/go\_project

假设你设置了GOPATH为/root/go\_project，那么这个目录就是存放你本个go项目的根目录。

GOPATH对应的工作区目录有三个子目录（src/bin/pkg）。其中src子目录用于存储源代码（可执行文件也要单独放在src下的一个子目录中，不能直接放在src目录下）。pkg子目录用于保存编译后的包的目标文件，bin子目录用于保存编译后的可执行程序。

假如，go\_project项目是一个爬虫项目，这个项目存放了多个可执行go文件用以对应多种类型的爬虫（普通爬虫，增量爬虫，全站爬虫等）,那么这多个可执行go文件应该分别放在src的不同目录下。这些爬虫所依赖的不同包也应该按照功能存放在不同的src下的目录中。如下所示：



这里要注意，这3个main可执行文件不能直接放在src下，而是要在src下用一个子目录包起来。

编译的时候，我们可以在任何目录执行 go build common\_crawler 进行编译（go会在环境变量找到GOPATH/src，然后再拼接common\_crawler，所以编译的时候会找到$GOPATH/src/common\_crawle，而无需 go build /root/go\_projects/src/common\_crawler写这么长）。在寻找依赖包的时候，go会在 GOPATH/src 和 GOROOT/src下找依赖包。

Go的编译不是以go文件为最小单位的，而是以包为最小单位的，也就是说go build只需跟要编译的目录名即可（它会把这个目录下所有的go文件编译为1个可执行二进制文件），不用也不能接某个具体的go文件。

编译后生成的二进制文件就放在了你执行go命令时的目录。因此我们一般都会先进入GOPATH/bin目录，再执行go build common\_crawler，这样这个二进制可执行文件就会生成再bin目录下。

编译一个普通爬虫的main包（目录），我们可以这样写：

cd anywhere

go build common\_crawler

或者

cd $GOPATH/src/common\_crawler

go build

或者

cd $GOPATH

go build ./src/common\_crawler

但是下面这种方式是错误的cd $GOPATH

go build src/common\_crawler

go env命令用于查看Go语言工具涉及的所有环境变量的值，包括未设置环境变量的默认值。

go get可以下载一个单一的包或者用...下载整个子目录里面的每个包。如果指定-u命令行标志参数，go get命令将确保所有的包和依赖的包的版本都是最新的，然后重新编译和安装它们。如果不包含该标志参数的话，而且如果包已经在本地存在，那么代码将不会被自动更新。

go build命令编译命令行参数指定的每个包。如果包是一个非main的库（目录），则忽略输出结果，这可以用于检测包是可以正确编译的(例如上例中，我们可以 go build spider来检验spider这个依赖包是否有编译错误，你会发现执行了这条命令后没有任何二进制文件，因为spider包只是依赖包而不是可执行的main包)。如果包的名字是main，go build将调用链接器在当前目录创建一个可执行程序，以导入路径的最后一段作为可执行程序的名字（例如上例中执行 go build common\_crawler）。

go run可以快速测试某个可执行的go文件。这个命令结合了go build（编译） 和运行这两步。和 go build不同的是，go build只能接包名，也就是只能接一个目录不能接go文件。而go run不仅可以接包名，也能接go文件，但是go run后面如果接包不能接依赖包只能接可运行的main包。

go run 也是会遵循go build 的寻找项目目录和依赖包目录的规则。例如，按照上面的例子，想运行common\_crawler可以在任意目录执行： go run common\_crawler 也可以 go run common\_crawler/common\_crawler.go

go install命令和go build命令很相似，但是它会保存每个包的编译成果，而不是将它们都丢弃。被编译的包会被保存到$GOPATH/pkg目录下，目录路径和 src目录路径对应，可执行程序被保存到$GOPATH/bin目录（在任意的一个目录运行go install 都会把编译后的可执行程序保存到$GOPATH/bin目录，这是和 go build不同之处）。

go install命令和go build命令都不会重新编译没有发生变化的包，这可以使后续构建更快捷。为了方便编译依赖的包，go build -i命令将安装每个目标所依赖的包。

PS：其实go build 和 go install 只要加上 -i 参数都会安装依赖包到pkg并且编译二进制执行文件（不加 -i 就都不会安装依赖包，而只是生成二进制文件），而且无论在哪个目录下执行go build 和 go install，依赖包的安装都会$GOPATH/pkg下。不同之处在于，go build 的二进制文件会生成在执行命令时的目录，而 go install 则是无论在哪里执行都把二进制文件生成到 $GOPATH/bin 下。

将一个特别的构建注释参数放在包声明语句之前可以提供更多的构建过程控制。例如，文件中可能包含下面的注释：

// +build linux darwin

该构建注释参数告诉go build只在编译程序对应的目标操作系统是Linux或Mac OS X时才编译这个文件

// +build ignore

该构建注释则表示不编译这个文件

更多细节，可以参考go/build包的构建约束部分的文档。

**包测试 go test**

Go语言的测试技术是相对低级的。它依赖一个go test测试命令和一组按照约定方式编写的测试函数，测试命令可以运行这些测试函数。编写相对轻量级的纯测试代码是有效的（但是用 go test 测试一个大型项目可能就不太好使了）。

在包目录内，所有以\_test.go为后缀名的源文件在执行go build时不会被构建成包的一部分，它们是go test测试的一部分。

在\*\_test.go文件中，有三种类型的函数：测试函数、基准测试（benchmark）函数、示例函数。一个测试函数是以Test为函数名前缀的函数，用于测试程序的一些逻辑行为是否正确；go test命令会调用这些测试函数并报告测试结果是PASS或FAIL。基准测试函数是以Benchmark为函数名前缀的函数，它们用于衡量一些函数的性能；go test命令会多次运行基准测试函数以计算一个平均的执行时间。示例函数是以Example为函数名前缀的函数，提供一个由编译器保证正确性的示例文档。

**测试函数**

我们看一下，一个标准的测试函数应该怎么写。

下面的函数IsPalindrome用于检查一个字符串是否是对称的：

// Package word provides utilities for word games.

package word

// IsPalindrome reports whether s reads the same forward and backward.

// (Our first attempt.)

func IsPalindrome(s string) bool {

for i := range s {

if s[i] != s[len(s)-1-i] {

return false

}

}

return true

}

这是一个依赖包文件而不是main文件

在相同的目录下，word\_test.go测试文件中包含了TestPalindrome和TestNonPalindrome两个测试函数。每一个都是测试IsPalindrome是否给出正确的结果，并使用t.Error报告失败信息：

package word

import "testing"

func TestPalindrome(t \*testing.T) {

if !IsPalindrome("detartrated") {

t.Error(`IsPalindrome("detartrated") = false`)

}

if !IsPalindrome("kayak") {

t.Error(`IsPalindrome("kayak") = false`)

}

}

func TestNonPalindrome(t \*testing.T) {

if IsPalindrome("palindrome") {

t.Error(`IsPalindrome("palindrome") = true`)

}

}

每一个测试文件名都应该以\_test.go为后缀结尾。每个测试函数都要以Test为前缀，每隔测试文件都要引入testing包。

使用 go test 进行包测试的时候，

go test 包名

或者

cd 某个包目录下 && go test

go test的规则和go build 的一样

go test 这个命令会运行要测试的包下所有的\_test.go文件的所有 Test开头的函数。如果只想要指定运行某一个或者多个Test开头的函数，可以用 -run 参数后面接一个正则：

例如有3个测试函数包含不同的测试文件，这3个函数分别是：

TestFrenchPalindrome

TestCanalPalindrome

TestChinaPalindrome

go test -run=”French|Canal”

满足上面这个正则的函数名是TestFrenchPalindrome 和TestCanalPalindrome，所以只有这两个测试函数会被测试。

-v可用于打印每个测试函数的名字和运行时间

$ go test -v

=== RUN TestPalindrome

--- PASS: TestPalindrome (0.00s)

=== RUN TestNonPalindrome

--- PASS: TestNonPalindrome (0.00s)

=== RUN TestFrenchPalindrome

--- FAIL: TestFrenchPalindrome (0.00s)

word\_test.go:28: IsPalindrome("été") = false

=== RUN TestCanalPalindrome

--- FAIL: TestCanalPalindrome (0.00s)

word\_test.go:35: IsPalindrome("A man, a plan, a canal: Panama") = false

FAIL

exit status 1

FAIL gopl.io/ch11/word1 0.017s

比较常见的测试方式是以表格的形式存放测试用例，然后对这些例子一一测试：

func TestIsPalindrome(t \*testing.T) {

var tests = []struct {

input string

want bool

}{

{"", true},

{"a", true},

{"aa", true},

{"ab", false},

{"kayak", true},

{"detartrated", true},

{"A man, a plan, a canal: Panama", true},

{"Evil I did dwell; lewd did I live.", true},

{"Able was I ere I saw Elba", true},

{"été", true},

{"Et se resservir, ivresse reste.", true},

{"palindrome", false}, // non-palindrome

{"desserts", false}, // semi-palindrome

}

for \_, test := range tests {

if got := IsPalindrome(test.input); got != test.want {

t.Errorf("IsPalindrome(%q) = %v", test.input, got)

}

}

}

需要注意的是：t.Errorf不会引发panic终止程序，因此引发了一个t.Errorf 后，测试还会继续。如果我们真的需要停止测试，或许是因为初始化失败或可能是早先的错误导致了后续错误等原因，我们可以使用t.Fatal或t.Fatalf替代t.Errorf。它们必须在和测试函数同一个goroutine内调用。

测试失败的信息一般的形式是“f(x) = y, want z”，其中f(x)解释了失败的操作和对应的输入，y是实际的运行结果，z是期望的正确的结果。(虽然不强制这样写，但是这是规范)

一般来说，测试文件都是用来测试一个依赖包，而不是一个main包，例如写了一个爬虫，我们可能很难说用 go test 来测试一个能沿着uri路径爬取整站的爬虫的运行是否错误。而是测试这个爬虫的一些组件的包，例如 download 下载函数，crawl 爬取单个页面函数，saveData 保存数据函数等等这种小块功能的测试。

当然我们也可以测试main包下的可执行文件，和测试依赖包没什么不同，不过，我们需要将main包中的代码给模块化，因为测试文件无法测试main文件的main函数，如下面的例子：

// Echo prints its command-line arguments.

package main

import (

"flag"

"fmt"

"io"

"os"

"strings"

)

var (

n = flag.Bool("n", false, "omit trailing newline")

s = flag.String("s", " ", "separator")

)

var out io.Writer = os.Stdout // modified during testing

func main() {

flag.Parse()

if err := echo(!\*n, \*s, flag.Args()); err != nil {

fmt.Fprintf(os.Stderr, "echo: %v\n", err)

os.Exit(1)

}

}

func echo(newline bool, sep string, args []string) error {

fmt.Fprint(out, strings.Join(args, sep))

if newline {

fmt.Fprintln(out)

}

return nil

}

这是一个main文件，里面有main和echo两个函数，但是测试文件测的时候只能测echo这个函数，无法去调用main函数，当然啦包级的变量的初始化在测试时还是会运行的。

下面就是echo\_test.go文件中的测试代码

package main

import (

"bytes"

"fmt"

"testing"

)

func TestEcho(t \*testing.T) {

var tests = []struct {

newline bool

sep string

args []string

want string

}{

{true, "", []string{}, "\n"},

{false, "", []string{}, ""},

{true, "\t", []string{"one", "two", "three"}, "one\ttwo\tthree\n"},

{true, ",", []string{"a", "b", "c"}, "a,b,c\n"},

{false, ":", []string{"1", "2", "3"}, "1:2:3"},

}

for \_, test := range tests {

descr := fmt.Sprintf("echo(%v, %q, %q)",

test.newline, test.sep, test.args)

out = new(bytes.Buffer) // captured output

if err := echo(test.newline, test.sep, test.args); err != nil {

t.Errorf("%s failed: %v", descr, err)

continue

}

got := out.(\*bytes.Buffer).String()

if got != test.want {

t.Errorf("%s = %q, want %q", descr, got, test.want)

}

}

}

个人觉得关于测试方面的内容到这里就差不多够用了，如果希望了解go圣经作者基准测试函数和示例函数的内容可以到下方go圣经链接查看。

那么到此为止，本博客网站的《Go入门系列》文章已经结束，该系列文章参考go 圣经加上自己的理解添加了一些本人自己写的示例，最终将这个系列的文章给展示出来。

该系列文章只是介绍了Go圣经 中主要和重要的知识点，如果想要查看更多 go圣经 的内容，可以参考原书链接：

https://books.studygolang.com/gopl-zh/