Go语言学习(十五)面向对象编程-接口

原创 2016年05月13日 23:15:15 标签：go语言 965

1.其他语言的接口

Go语言的接口并不是其他语言（C++、Java、C#等）中所提供的接口概念。

在Go语言出现之前，接口主要作为不同组件之间的契约存在。对契约的实现是强制的，你

必须声明你的确实现了该接口。为了实现一个接口，你需要从该接口继承：

interface IFoo {

void Bar();

}

class Foo implements IFoo { // Java语法

// ...

}

1

2

3

4

5

6

即使另外有一个接口IFoo2实现了与IFoo有完全一样的接口方法甚至名字也叫IFoo只不过位

于不同的名字空间(包名)下,编译器也会认为上面的类Foo只实现了IFoo而没有实现IFoo2接口。

这类接口我们称为侵入式接口。“侵入式”的主要表现在于实现类需要明确声明自己实现了

某个接口。这种强制性的接口继承是面向对象编程思想发展过程中一个遭受相当多置疑的特性。

2.非侵入式接口

在Go语言中，一个类只需要实现了接口要求的所有函数,我们就说这个类实现了该接口，

例如：

type File struct {

// ...

}

func (f \*File) Read(buf []byte) (n int, err error)

func (f \*File) Write(buf []byte) (n int, err error)

func (f \*File) Seek(off int64, whence int) (pos int64, err error)

func (f \*File) Close() error

1

2

3

4

5

6

7

这里我们定义了一个File类并实现有Read()、Write()、Seek()、Close()等方法。设

想我们有如下接口：

type IFile interface {

Read(buf []byte) (n int, err error)

Write(buf []byte) (n int, err error)

Seek(off int64, whence int) (pos int64, err error)

Close() error

}

type IReader interface {

Read(buf []byte) (n int, err error)

}

type IWriter interface {

Write(buf []byte) (n int, err error)

}

type ICloser interface {

Close() error

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

尽管File类并没有从这些接口继承,甚至可以不知道这些接口的存在,只要File类

有包含这些接口的所有方法,就可以如下赋值使用.

var file1 IFile = new(File)

var file2 IReader = new(File)

var file3 IWriter = new(File)

var file4 ICloser = new(File)

1

2

3

4

是不是很强大,最大的特点就是不需要明确(显式)声明自己实现了某个接口.

Go语言的非侵入式接口,看似只是做了很小的文法调整,实则影响深远。

其一,Go语言的标准库,再也不需要绘制类库的继承树图.

在Go中,类的继承树并无意义,你只需要知道这个类实现了哪些方法,每个方法是啥含义

就足够了。

其二,实现类的时候,只需要关心自己应该提供哪些方法,不用再纠结接口需要拆得多细才

合理。接口由使用方按需定义,而不用事前规划。

其三,不用为了实现一个接口而导入一个包,因为多引用一个外部的包,就意味着更多的耦

合。接口由使用方按自身需求来定义,使用方无需关心是否有其他模块定义过类似的接口。

3.接口的赋值

接口赋值在Go语言中分为如下两种情况：

1)将对象实例赋值给接口；

2)将一个接口赋值给另一个接口。

先讨论将某种类型的对象实例赋值给接口，这要求该对象实例实现了接口要求的所有方法，

例如之前我们作过一个 Integer 类型，如下：

type Integer int

func (a Integer) Less(b Integer) bool {

return a < b

}

func (a \*Integer) Add(b Integer) {

\*a += b

}

1

2

3

4

5

6

7

相应地，我们定义接口 LessAdder ，如下：

type LessAdder interface {

Less(b Integer) bool

Add(b Integer)

}

1

2

3

4

现在有个问题：假设我们定义一个Integer类型的对象实例,怎么将其赋值给LessAdder

接口呢？应该用下面的语句(1)，还是语句(2)呢？

var a Integer = 1

var b LessAdder = &a ...(1)

var b LessAdder = a ... (2)

1

2

3

答案是应该用语句(1)。原因在于,Go语言可以根据下面的函数：

func (a Integer) Less(b Integer) bool

自动生成一个新的Less()方法：

func (a \*Integer) Less(b Integer) bool {

return (\*a).Less(b)

}

1

2

3

这样,类型\*Integer就既存在Less()方法,也存在Add()方法,满足LessAdder接口的所有方法。

为了进一步证明以上的推理,我们不妨再定义一个Lesser接口,如下：

type Lesser interface {

Less(b Integer) bool

}

1

2

3

然后定义一个Integer类型的对象实例,将其赋值给Lesser接口：

var a Integer = 1

var b1 Lesser = &a ...//(1)

var b2 Lesser = a ... //(2)

1

2

3

正如我们所料的那样,语句(1)和语句(2)均可以编译通过。

我们再来讨论另一种情形：

将一个接口赋值给另一个接口。在Go语言中,只要两个接口拥

有相同的方法列表(次序不同不要紧),那么它们就是等同的,可以相互赋值。

例如:

package one

type ReadWriter interface {

Read(buf []byte) (n int, err error)

Write(buf []byte) (n int, err error)

}

1

2

3

4

5

第二个接口位于另一个包中：

package two

type IStream interface {

Write(buf []byte) (n int, err error)

Read(buf []byte) (n int, err error)

}

1

2

3

4

5

这里我们定义了两个接口,一个叫 one.ReadWriter,一个叫two.Istream,两者都定义

了Read()、Write()方法,只是定义次序相反,one.ReadWriter先定义了Read()再定义了

Write(),而two.IStream反之。

在Go语言中,这两个接口实际上并无区别,因为：

1)任何实现了one.ReadWriter接口的类,均实现了two.IStream;

2)任何实现了one.ReadWriter接口的类,均实现了two.IStream;

3)在任何地方使用one.ReadWriter接口与使用two.IStream并无差异。

以下这些代码均可编译通过：

var file1 two.IStream = new(File)

var file2 one.ReadWriter = file1

var file3 two.IStream = file2

1

2

3

接口赋值并不要求两个接口必须等价,如果接口A的方法列表是接口B的方法列表的子集，

那么接口B可以赋值给接口A。例如,假设我们有Writer接口：

type Writer interface {

Write(buf []byte) (n int, err error)

}

1

2

3

就可以将上面的one.ReadWriter和two.IStream接口的实例赋值给Writer接口：

var file1 two.IStream = new(File)

var file4 Writer = file1 //赋值给子集接口

1

2

但是反过来并不成立：

var file1 Writer = new(File)

var file5 two.IStream = file1 // 编译不能通过

1

2

这段代码无法编译通过,原因是显然的:file1并没有Read()方法。

4.接口查询

有办法判断Writer接口是否可以转换为two.IStream接口呢?

有,那就是我们即将讨论的接口查询语法,代码如下：

var file1 Writer = ...

if file5, ok := file1.(two.IStream); ok {

...

}

1

2

3

4

这个if语句检查file1接口指向的对象实例是否实现了two.IStream接口,如果实现了,则执

行特定的代码。

类似的java语法中instanceof,比如查询一个对象的类型是否继承自某个类型(基类查询)，

或者是否实现了某个接口(接口派生查询),但是它们的动态查询与Go的动态查询很不一样。

5.类型查询

在Go语言中,还可以更加直截了当地询问接口指向的对象实例的类型,例如：

var v1 interface{} = ...

switch v := v1.(type) {

case int: // 现在v的类型是int

case string: // 现在v的类型是string

...

}

1

2

3

4

5

6

就像现实生活中物种多得数不清一样,语言中的类型也多得数不清,所以类型查询并不经常

使用。它更多是个补充,需要配合接口查询使用,例如：

package main

import (

"fmt"

)

type Stringer interface { //定义接口

String() string

}

type MyStringer struct{ //定义结构体

}

func (\*MyStringer) String()string{ //实现接口的方法

return "this is a method implement from Stringer"

}

func main(){

Println(123)

Println("mChenys")

var myStringer Stringer = new(MyStringer)

Println(myStringer)

}

func Println(args ...interface{}) {//不定参数

for \_, arg := range args {

switch v := arg.(type) {//判断类型

case int:

fmt.Println("现在",v,"的类型是int")

case string:

fmt.Println("现在",v,"的类型是string")

default:

if v, ok := arg.(Stringer); ok { // 现在v的类型是Stringer

val := v.String()

fmt.Println(val)

} else {

fmt.Println("其它类型")

}

}

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

//输出结果:

现在 123 的类型是int

现在 mChenys 的类型是string

this is a method implement from Stringer

6.接口组合(嵌套)

像之前介绍的类型(结构体)组合一样,Go语言同样支持接口组合,例如:

// ReadWriter接口将基本的Read和Write接口组合起来

type ReadWriter interface {

Reader

Writer

}

1

2

3

4

5

这个接口组合了Reader和Writer两个接口,它完全等同于如下写法：

type ReadWriter interface {

Read(p []byte) (n int, err error)//Reader的方法

Write(p []byte) (n int, err error)//Writer的方法

}

1

2

3

4

因为这两种写法的表意完全相同:ReadWriter接口既能做Reader接口的所有事情,又能做

Writer接口的所有事情.在Go语言包中,还有众多类似的组合接口,比如ReadWriteCloser、

ReadWriteSeeker、ReadSeeker和WriteCloser等。

可以认为接口组合是类型匿名组合的一个特定场景,只不过接口只包含方法,而不包含任何

成员变量。

7.Any类型

由于Go语言中任何对象实例都满足空接口interface{},所以interface{}看起来像是可

以指向任何对象的Any类型,如下：

var v1 interface{} = 1 //将int类型赋值给interface{}

var v2 interface{} = "abc" //将string类型赋值给interface{}

var v3 interface{} = &v2 //将\*interface{}类型赋值给interface{}

var v4 interface{} = struct{ X int }{1}

var v5 interface{} = &struct{ X int }{1}

1

2

3

4

5

当函数可以接受任意的对象实例时,我们会将其声明为interface{},最典型的例子是标

准库fmt中PrintXXX系列的函数,例如：

func Printf(fmt string, args ...interface{})

func Println(args ...interface{})

...

1

2

3

文章来源：http://blog.csdn.net/mchenys/article/details/51399536

版权声明：本文为博主原创文章，未经博主允许不得转载。