Go语言学习(十三)面向对象编程-继承

原创 2016年05月09日 23:34:00

标签：

[go语言](http://so.csdn.net/so/search/s.do?q=go%E8%AF%AD%E8%A8%80&t=blog" \t "http://blog.csdn.net/mchenys/article/details/_blank)

2909

1.匿名组合

Go语言也提供了继承,但是采用了组合的方式,所以我们将其称为匿名组合：

package mainimport "fmt"

//定义基类type Base struct {

Name string

}//基类相关的2个成员方法func (base \*Base) A() {

fmt.Println("Base method A called...")

}func (base \*Base) B() {

fmt.Println("Base method B called...")

}

//定义子类type Son struct {

Base //"继承"基类

}//重写基类的B方法func (son \*Son) B() {

son.Base.B() //调用基类的B方法

fmt.Println("Son method B called...")

}

func main(){

son := Son{Base{"mChenys"}}

son.B() //调用子类的重写至基类的B方法

son.A() //调用子类继承至基类的A方法

}

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

输出结果:

Base method B called...Son method B called...Base method A called...

1

2

3

以上代码定义了一个Base类(实现了A()和B()两个成员方法),然后定义了一个   
Son,该类从Base类“继承”并改写了B()方法(该方法实现时先调用了基类的B()方法).

在“派生类”Son没有改写“基类”Base的成员方法时,相应的方法就被“继承”，例如在   
上面的例子中,调用son.A()和调用son.Base.A()效果一致。   
与其他语言不同，Go语言很清晰地告诉你类的内存布局是怎样的。此外，在Go语言中你还   
可以随心所欲地修改内存布局，如：

type Son struct {

... // 其他成员

Base

}

1

2

3

4

这段代码从语义上来说,和上面给的例子并无不同,但内存布局发生了改变。“基类” Base   
的数据放在了“派生类” Son的最后。

另外，在Go语言中，你还可以以指针方式从一个类型“派生”：   
type Son struct {   
\*Base   
…// 其他成员   
}   
这段Go代码仍然有“派生”的效果,只是Son创建实例的时候,需要外部提供一个Base类   
实例的指针.   
如下所示,匿名组合了一个log.Logger指针：

type Job struct {

Command string

\*log.Logger

}

1

2

3

4

在合适的赋值后，我们在Job类型的所有成员方法中可以很舒适地借用所有log.Logger提   
供的方法。比如如下的写法:   
func (job \*Job)Start() {   
job.Log(“starting now…”) //注意:Log函数的接收者仍然是log.Logger指针   
… // 做一些事情   
job.Log(“started.”)   
}   
对于Job的实现者来说,他甚至根本就不用意识到log.Logger类型的存在,这就是匿名组合的   
魅力所在.在实际工作中,只有合理利用才能最大发挥这个功能的价值。

2.名字冲突问题

我们必须关注一下接口组合中的名字冲突问题，比如如下的组合：

package mainimport(

"fmt"

)type X struct {

Name string

}type Y struct {

X

Name string //相同名字的属性名会覆盖父类的属性

}func main(){

y := Y{X{"XChenys"},"YChenys"}

fmt.Println("y.Name = ",y.Name) //y.Name = YChenys

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

组合的类型和被组合的类型都包含一个Name成员，会不会有问题呢？答案是否定的。所有   
的Y类型的Name成员的访问都只会访问到最外层的那个Name变量，X.Name变量相当于被覆盖了。

文章来源：http://blog.csdn.net/mchenys/article/details/51357346

版权声明：本文为博主原创文章，未经博主允许不得转载。