let's GoLang(二)： 面向对象

标记

Richard
full stack, gopher, rustacean

这是GoLang系列文章的第二篇：面向对象。上一篇是Let's GoLang(一): 反射。

严格来说，说到面向对象，那么一定会谈到Java，Go并没有Java中那种面向对象的概念，但是跟JS基于原型的面向对象一样，可以通过组合Go的一些特性来实现面向对象的功能。

说到面向对象，那么一定少不了**继承、封装、多态。**

封装

首先我们来了解一下Go中的封装。对于封装而言，一个对象它所**封装的是自己的属性和方法**，所以它是不需要依赖其他对象就可以完成自己的操作。使用封装的好处是：**良好的封装能够减少耦合，类内部的结构可以自由修改；可以对成员进行更精确的控制。隐藏信息，实现细节。**在Go中，封装是基于可见性原则来实现的，我们来看个例子：

```
package oop

type User struct {
        Name string
        age int
}


func (user *User) SetAge(age int) *User  {
        user.age = age
        return user
}

func (user *User) GetAge () int  {
        return user.age
}
```

```
package main

import (
        "fmt"
        "github.com/anymost/oop"
```

```
)

func main()  {
        user := oop.User{
                Name: "jack",
        }
        user.SetAge(20)

        fmt.Println(
```

在Golang中，基于可见性原则，**如果一个变量是以大写字母开头的，那么在包外面可以访问到；如果是以小写字母开头的，在包内可见，在包外不可见。**

我们可以看到上面的例子，在一个oop的包内定义了一个User的结构体，那么我们能在包外访问到这个结构体。User结构体中，Name字段是以大写字母开头的，可以直接在包外访问；age字段是以小写字段，在包外不可见，但是我们可以提供了以大写字母开头的GetAge和SetAge方法，同样能够读取和修改age字段。这就是Go里面的封装，很好理解。

继承

通常来说，**继承是使用已存在的类的定义作为基础建立新类的技术**，新类的定义可以增加新的数据或新的功能，也可以用父类的功能，但不能选择性地继承父类。通过使用继承我们能够非常方便地复用以前的代码，能够大大的提高开发的效率。在Java中，子类继承父类，那么能够通过子类来访问到父类上的属性和方法。

在Go中，继承是基于组合的方式来实现的，我们来看个例子：

```
package oop

type Car struct {
        Engine string
        Tire string
}

type SUV struct {
        Car
        Brand string
}
```

```
package main

import (
        "fmt"
        "github.com/anymost/oop"
)

func main()  {
        suv := oop.SUV{Car: oop.Car{Engine: "good", Tire: "fine"}, Brand: "BMW"}
        fmt.Println(suv.Engine)
```

```
        fmt.Println(suv.Brand)
}
```

可以从上面的代码看到，我们首先定义了一个Car的struct，然后定义了一个SUV的struct，其中SUV包含Car，能够通过SUV来访问到Car中的属性和方法等，这就是struct的组合。不仅能够组合struct，还能组合interface：

```
package oop

type Dancer interface {
        Dance()
}

type Singer interface {
        Sing()
}

type Artist interface {
        Dancer
        Singer
}
```

上面分别定义了Dancer，Singer，Artist三个interface，如果要满足Artist类型，就需要定义Dance和Sing两个方法

多态

接下来，我们理解一下多态。 在Java中基于继承的实现机制主要表现在父类和继承该父类的一个或多个子类对某些方法的重写，**多个子类对同一方法的重写可以表现出不同的行为。**基于继承实现的多态可以总结如下：对于引用子类的父类类型，在处理该引用时，它适用于继承该父类的所有子类，子类对象的不同，对方法的实现也就不同，执行相同动作产生的行为也就不同。如果父类是抽象类，那么子类必须要实现父类中所有的抽象方法，这样该父类所有的子类一定存在统一的对外接口，但其内部的具体实现可以各异。这样我们就可以使用顶层类提供的统一接口来处理该层次的方方法。

在Go中，多态是基于interface来实现的，我们还是举一个简单的例子:

```
package oop

import "fmt"

type Animal interface {
        Live()
}

type Cat struct {
        Name string
}

type Dog struct {
        Name string
```

```
}

func (cat *Cat) Live()  {
        fmt.Println(fmt.Sprintf("%s eat fish", cat.Name))
}

func (dog *Dog) Live()  {
        fmt.Println(fmt.Sprintf("%s eat bone", dog.Name))
}
```

```
package main

import "github.com/anymost/oop"

func showLive(animals []oop.Animal)  {
        for _, animal := range animals {
                animal.Live()
        }
}

func main()  {
        animals := []oop.Animal{
                &oop.Cat{Name: "cat"},
                &oop.Dog{Name:"dog"},
        }
        showLive(animals)
}
```

我们来分析上面的代码：

1. 首先定义了一个名为Animal的接口，其中包含了Live方法
2. 然后分别定义了名为Cat、Dog 的两个struct，在它们上定义了Livefangfa
3. 然后定义了名为showLive的函数，参数为Animal类型的数组
4. 最后定义了一个animals的数组，其成员分别为Cat，Dog的实例

总结一下：在Go中，只要在一个struct上定义了某个interface的所有方法，那么这个struct就是可以认为是这个interface的类型，可以在所有使用到这个interface类型的地方使用该struct，就像上面的showLive方法一样。同时，Cat，Dog虽然是Animal类型的，但是又拥有自己的逻辑，这就是多态的强大。那么问题来了，我们怎么来判断一个interface的变量具体是哪个类型的呢？这就要用到我们前面提到的反射了：

```
package main

import (
        "fmt"
        "github.com/anymost/oop"
)

func checkType(animals []oop.Animal)  {
        for _, animal := range animals {
                switch animal.(type) {
```

```
                case *oop.Cat:
                        fmt.Println("type is Cat")
                case *oop.Dog:
                        fmt.Println("type is Dog")
                default:
                        fmt.Println("type is others")
                }
        }
}

func main()  {
        animals := []oop.Animal{
                &oop.Cat{Name: "cat"},
                &oop.Dog{Name:"dog"},
        }

        checkType(animals)
}
```

我们可以看到，借助反射，能够在运行时来动态的判断类型。而且，Go中还有类似于Java中的Object类型，那就是空接口 interface{}，所有类型的变量都可以认为是interface{}类型的。

发布于 2019-01-19 19:38

Go 语言

后端技术

编程语言