Go是面向对象的吗？Yes and no!

Go 语言不是一门面向对象的语言，但是它通过组合的方式变相地支持面向对象。

面向对象是业务抽象、代码复用方面比较成功的一个编程范式。开发者在初次接触 Go 语言的时候，对于 Go 语言相对松散的面向对象实现可能会感到不适应。本文围绕 Go 语言基于面向对象的特性进行阐述，为你提供一些不同的视角。

概述

封装、继承、多态是面向对象的三大特性。一门编程语言只要实现了这三个方面，就可以说是具备了面向对象的能力。

Go 语言中没有类和对象，但是它的结构体类型可以承载这些概念。结构体类型可以定义属性，保存对象的数据。同时也可以为结构体绑定一个方法，从而定义对象的行为。

封装

封装的目的是实现信息隐藏。对外隐藏内部实现，仅提供公开的方法。

可见性

在传统的面向对象语言中的可见性约定：

关键词	任意范围	包	类	子类
Public	Y	Y	Y	Y
Protected	N	Y	Y	Y
Private	N	N	Y	N

Go 语言通过变量和方法的首字母大小写来标记是否公开。公开的字段或者方法可以在任意范围访问。首字母大写对应 `public` 关键字，首字母小写只能在包内可见写对应 `private` 。由于

Go 语言不支持继承，因此没有对应的 `protected` 概念。因为 Go 语言是通过组合实现的代码复用，因此也不需要 `extends` 关键字。

由于 Go 语言中的包并没有复杂的可见性层级，不适合构建多层级的封装。因此你会发现 Go 语言中的包往往是很小的，也是职责很单一的。

继承

Go 语言是通过类型组合来实现代码复用的。

类型嵌套

结构体类型嵌套是 Go 语言模拟面向对象继承的方法。

一个结构体 A 如果把另一个结构体 B 以嵌入的方式引入，那么这个这个结构体 A 就拥有了结构体 B 的属性和方法。如果子类的功能跟父类已实现的功能存在冲突的地方，Go 语言也支持通过重写方法的方式覆盖父类的实现。如果需要在父类的基础上进行扩展，那么可以在子类的调用父类的方法，然后在子类实现差异化的部分。

由于 Go 并不是面向对象的语言，没有 `this` 指针以及 `super` 等关键字功能，因此需要显式的调用父类的方法。

继承是代码的垂直扩展和复用，而组合是可以水平扩展的。在继承中，子类的功能会受到父类的功能限制。在不支持多重继承的面向对象语言中，对象的继承关系往往是一个树形结构。继承可以很好地描述 is a 关系的逻辑，但是对于 has a 类型的关系无法描述。比如一辆车由轮子、发动机、车座、车身组成，这样一种场景，继承的方式就描述不了。Go 语言天然的就支持组合。

类型嵌套示例：

Go

```go
// 父类
type Logger struct{}

func NewLogger() *Logger {
    return &Logger{}
}

func (l *Logger) write(msg string) {
    println(msg)
}

func (l *Logger) Debug(msg string) {
    debugInfo := ""
    l.write(debugInfo + msg)
}

// 子类继承
type FileLogger struct {
    *Logger
    FileName string
}

// 构造函数嵌套，等价于super.constructor()
func NewFileLogger(file string) *FileLogger {
    return &FileLogger{Logger: NewLogger(), FileName: file}
}

func (f *FileLogger) write(msg string) {
    ioutil.WriteFile(f.FileName, []byte(msg), fs.ModeAppend)
}

// Debug 同名方法覆盖
func (f *FileLogger) Debug(msg string) {
    debugInfo := ""
    f.write(debugInfo + msg)
}
```

直接使用类型嵌套有一个问题，它会暴露父类的方法可见性。因此我们推荐类型实例的嵌套。

两种嵌套方式的对比如下：

改进前：

Go

```go
package main

// 父类
type Logger struct{}

func (l *Logger) Lock() {}

func (l *Logger) Unlock() {}

// 子类继承
type FileLogger struct {
    *Logger
    FileName string
}

func main() {
    f := FileLogger{}
    // 暴露了Logger的Unlock方法，这里有问题
    f.Unlock()
}

```

改进后：

Go

```go
package main

// 父类
type Logger struct{}

func (l *Logger) Lock() {}

func (l *Logger) Unlock() {}

// 子类继承
type FileLogger struct {
    logger   Logger
    FileName string
}

func main() {
    f := FileLogger{}
    // 此时父类的方法不可见
    //  f.Unlock() ！编译错误
}
```

多态

多态（英语：polymorphism）指为不同数据类型的实体提供统一的接口，或使用一个单一的符号来表示多个不同的类型

从一个父类派生出多个子类，可以使子类之间有不同的行为，这种行为称之为多态。通俗来说，子类重写父类的方法，使子类具有不同的方法实现。多态性依赖于继承。但是 Go 不支持继承，而是通过接口的方式支持多态。

接口模拟多态：

Go

```go
package main

type LoggerInterface interface {
    Debug(msg string)
    Info(msg string)
}

type ConsoleLogger struct{}

func (c *ConsoleLogger) Debug(msg string) {}

func (c *ConsoleLogger) Info(msg string) {}

type FileLogger struct{}

func (c *FileLogger) Debug(msg string) {}

func (c *FileLogger) Info(msg string) {}

func doSomething(example string, logger LoggerInterface) {
    // 不同的实现有不同的表现
    logger.Debug(example)
}

func main() {
    var logger LoggerInterface

    // 动态切换不同的实现对象
    logger = &ConsoleLogger{}
    doSomething("console", logger)

    // 动态切换不同的实现对象
    logger = &FileLogger{}
    doSomething("file", logger)
}

```

编译时多态

Go 语言在构建时，可以通过指定编译标签来动态调整编译行为（类似于 C 语言的条件编译）。

例如，Go 语言内置的 json 序列化包性能会稍差一些，我们想替换成其他的包（jsoniter 等）。那我们我们只需要自定义一个包，然后为不同实现的文件指定不同的构建标签。在我们编译的时候通过指定不同的 tags，来控制编译器构建不同的代码。

Go

```go
//go:build jsoniter

package json

import jsoniter "github.com/json-iterator/go"

var (
    // 导出需要的方法
    json          = jsoniter.ConfigCompatibleWithStandardLibrary
    Marshal       = json.Marshal
    Unmarshal     = json.Unmarshal
    MarshalIndent = json.MarshalIndent
    NewDecoder    = json.NewDecoder
    NewEncoder    = json.NewEncoder
)

```

重载与默认参数

函数重载似的同一个函数可以完成不同的功能。动态语言一般是通过参数默认值来达到函数重载的效果。Go 语言不支持函数重载，但是通过函数式 `option pattern` 实现了同样的效果。当然也可以使用建造者模式进行定制化的对象构造，然后再使用。

Option 模式的示例：

Go

```go
type Logger struct {
    level         string
    flushInterval int
}

type Option func(*Logger)

func WithLevel(level string) Option {
    return func(l *Logger) {
        l.level = level
    }
}

func WithFlushInterval(interval int) Option {
    return func(l *Logger) {
        l.flushInterval = interval
    }
}

// NewLogger
// 使用默认值开箱即用
// logger := NewLogger()
//
// 各种不同场景定制
// logger := NewLogger(WithLevel("info"))
func NewLogger(opts ...Option) *Logger {
    l := &Logger{}
    // 默认配置
    l.level = "debug"
    l.flushInterval = 123

    // 用户自定义配置
    for _, apply := range opts {
        apply(l)
    }
    return l
}
```

总结

Go是面向对象的语言吗？Go语言官方是这么理解的：

> Is Go an object-oriented language?
> Yes and no. Although Go has types and methods and allows an object-oriented style of programming, there is no type hierarchy. The concept of “interface” in Go provides a different approach that we believe is easy to use and in some ways more general. There are also ways to embed types in other types to provide something analogous—but not identical—to subclassing. Moreover, methods in Go are more general than in C++ or Java: they can be defined for any sort of data, even built-in types such as plain, “unboxed” integers. They are not restricted to structs (classes).
> Also, the lack of a type hierarchy makes “objects” in Go feel much more lightweight than in languages such as C++ or Java.

Go语言是不是面向对象语言，这个问题就如同薛定谔的猫一样：*Yes and no.*。

当我们说Go语言是面向对象时，是因为它支持面向对象的主要特性。只要一个东西能像鸭子一样叫，我们就可以认为它是鸭子。

当我们说Go语言不是面向对象语言时，是因为它不支持继承。