

**译 文**

**原文题目：** A Golang Unit Test Piling Tool

**译文题目：** 一种Golang单元测试打桩工具

**学 院： 计算机科学学院 院**

**专业班级： 软件工程15级01班**

**学生姓名： 孙希望**

**学 号： 41509050120 4**

译文

一种Golang 单元测试打桩工具

周雷

**摘 要**： 介绍一种 Golang 语言的单元测试打桩工具。TDD （测试驱动开发） 中需要先写单元测试再实现功能，但 Golang 自带测试工具 Testing 不能满足 TDD 实践。通过对比 Golang 几个流行的单元测试打桩工具，实例演示了Monkey 打桩完成单元测试。

**关键词**： Golang, TDD, Monkey

# 概述

TDD 是测试驱动开发（ Test Driven Development）的英文简称，是敏捷开发中的一项核心实践和技术， 也是一种设计方法论。TDD 的原理是在开发功能代码之前，先编写单元测试用例代码，测试代码确定需要编写什么产品代码。

TDD 实践都是先写单元测试，从一个simple\_test.go开始测试驱动开发， 在写单元测试的时候，都是先假设这个结构体、函数、对象等已经实现，运行 go test的时候编译和测试都通过不了，需要开发者一边完善代码，一边重复运行 go test，一直到完成开发，这是一个非常好的习惯。

# 问题提出

Go 本身提供了一套轻量级的测试框架。 符合规则的测试代码会在运行测试时被自动识别并执行。 单元测试源文件的命名规则如下： 在需要测试的包下面创建以“\_test” 结尾的 go 文件， 形如 [] \*\_test.go[2]。

Go test 是 Golang 语言自带的测试工具， 只是完成了基本的单元测试框架， 没有提供对被测函数调用的函数打桩的工具。 基于 TDD 开发模式下， 开发者在编写单元测试的时候，希望有一个打桩类似 C++语言 Mock- Cpp 工具。 如果调用函数没有实现，被测函数中使用打桩工具，可顺利地完成测试。

这是来自 《重构--改善既有代码的设计》 的例子，不讨论重构话题， 只用来做示例，“实例非常简单，这是一个影片出租店用的程序，计算每一个顾客的消费金额并打印详情。操作者告诉程序：顾客租了哪些影片、租期多长， 程序便根据租凭时间和影片类型算出费用。影片分为 3 类：普通片、儿童片和新片。除了计算费用，还要为常客计算积分，积分会根据租片种类是否为新片而有不同[1]。”



Movie 1

priceCode:int

Movie

priceCode:int

\*

Movie

priceCode:int

\*

1

图 1 影片出租店 UML 图

拿 customer.go 的 StateMent 方法举例：

func (customer \*Customer)StateMent() string{  
 totalAmount := float64(0.0)  
 frequentRenterPoints:= 0  
 result := "Rental Record for " + customer.GetName()+"\n"  
 fmt.Println(len(customer.rentals))  
 for \_,rental:=range customer.rentals{  
 thisAmount:= float64(0.0)  
 switch rental.GetMovie().GetPriceCode() {  
 case REGULAR:  
 thisAmount+=2  
 if rental.GetDaysRented()>2{   
 thisAmount +=float64((rental.GetDaysRented()-2))\*1.5  
 }  
 case NEW\_RELEASE:  
 thisAmount += float64(rental.GetDaysRented()\*3)  
 case CHILDRENS:  
 thisAmount+=1.5  
 if rental.GetDaysRented()>3{   
 thisAmount+=(float64)(rental.GetDaysRented()-3)\*1.5  
 }  
 }  
 frequentRenterPoints++  
 if (rental.GetMovie ().GetPriceCode () == NEW\_RELEASE)&& (rental.GetMovie().GetPriceCode()>1){  
 frequentRenterPoints++  
 }  
 result+= rental.GetMovie().GetTitle()+":"+str- conv.FormatFloat(thisAmount,'f',-1,64)+"\n"  
 totalAmount +=thisAmount  
 }  
 result += "\nAmount owed is " + strconv.Format- Float(totalAmount,'f',-1,64) +"\n"  
 result +="You earned "+ strconv.FormatInt(int64 (frequentRenterPoints),10)+" frequent renter points"  
 return result  
}

若想对 customer 类的方法 func ( customer \*Cus- tomer) StateMent() string 进行单元测试时， 发现没有实现Rental 类和 Movie 类方法 （代码中黑色字体）， 就没法进行单元测试。

# 解决方法

Golang 语言有没有像 C 和 C++语言的单元测试工具MockCpp 呢？大家普遍了解的是 GoStub 和 GoMock， 在Github 还有一个 Golang 单元测试打桩工具 Go Monkey Patching，简称“Monkey”。

对比一下几个工具的优缺点：

GoStub 实现打桩时需将变量指向打桩函数和类方法的返回值，但对没有返回值的函数和类方法没法打桩； 当使用时需要对被测代码进行改动，是一种侵入式修改； 也无法对接口进行打桩。 GoMock 生成桩文件需执行命令，接口可以很方便地实现打桩，但对函数、类方法等却难以打桩。 Monkey 虽没有办法对接口进行打桩，但在打桩时是非侵入式修改，无需被测代码做任何修改， 很方便地实现对函数和类方法的打桩。如表 1 所示。

表 1 Golang 单元测试工具对比

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 非侵入式 | 接口 | 函数 | 方法 |
| GoStub | N | N | Y | Y |
| GoMock | Y | Y | N | N |
| Monkey | Y | N | Y | Y |

1. 实际应用

下面演示如何使用 Monkey 对示例中的函数单元测试。按TDD的实践方法， 先写出 StateMent 依赖 Rental 类和 Movie 类方法使单元测试编译通过， 但不对方法具体实现。

然后用 Monkey 对依赖的类方法进行打桩， 黑色字体是用 Monkey 对依依赖的类方法打桩的实现，在打桩以后用 fmt.Println 调用几个方法，可以得到期望的输出

customer\_test.go文件

func Test\_Customer\_Should\_Calc\_By\_Regular\_Rule\_When\_One\_Rental\_Rent\_Regular\_Movie(t \*testing.T) {  
 …  
 guardGetPriceCode : = monkey.PatchInstanceMethod(reflect.TypeOf(customer.rentals[0].GetMovie()), "GetPriceCode",  
 func(movie \*Movie) int {  
 return 0  
 })  
 defer guardGetPriceCode.Unpatch()  
 guardGetTitle := monkey.PatchInstanceMethod (reflect.TypeOf (customer.rentals [0].GetMovie ()), "GetTitle",  
 func(movie \*Movie) string {  
 return "game of chrones"  
 })  
 defer guardGetTitle.Unpatch()  
 guardStart : = monkey.PatchInstanceMethod (reflect.TypeOf (customer.rentals [0]), "GetDaysRented",  
 func(rental \*Rental) int {  
 return 5  
 })  
 defer guardStart.Unpatch()  
 …  
}

1. 结 语

通过对比和实践介绍 Monkey， 可以弥补 Golang 官方单元测试工具 Testing 缺陷， 用打桩保证函数级代码之间的解耦， 是 Golang 实践 TDD 的一种打桩利器。

参考文献

[1] Martin Fowler, 熊杰. 重构--改善代码既有代码的设计. 北京: 人民邮电出版社, 2015.

[2] 许式伟, 吕桂华. Go 语言编程. 北京: 人民邮电出版社， 2016.

A Golang Unit Test Piling Tool

Zhou-Lei

**Absrtact**： This paper introduces a unit test piling tool in Golang language. TDD (Test Driven Development) needs to write unit tests before implementing functions, but Golang's own testing tool Testing can not meet the TDD practice. By comparing several popular unit testing piling tools of Golang, an example demonstrates that Monkey piling completes unit testing.

**Key words**： Golang, TDD, Monkey

# Summary

# TDD is the English abbreviation for Test Driven Development, a core practice and technology in agile development, and a design methodology. The principle of TDD is to write unit test case code, which determines what product code needs to be written before developing the function code.

# TDD practice is to write unit test first, start test drive development from a simple\_test.go, when writing unit tests, first assume that the structure, functions, objects, etc. have been implemented, compile and test when running go test Can't pass, it is a very good habit to require developers to improve the code while running go test repeatedly until the development is completed.

# Questions raised

# Go itself provides a lightweight test framework. Test code that conforms to the rules is automatically recognized and executed when the test is run. The naming rules for unit test source files are as follows: Create a go file ending with "\_test" under the package that needs to be tested, like [] \*\_test.go[2].

# Go test is a test tool that comes with the Golang language. It just completes the basic unit testing framework and does not provide a tool for piling the functions called by the function being tested. Based on the TDD development model, when developers write unit tests, they hope to have a piling similar to the C++ language Mock-Cpp tool. If the calling function is not implemented, the piling tool can be used to successfully complete the test.

# This is an example from Refactoring - Improving the Design of Existing Code. It does not discuss refactoring topics. It is only used as an example. "The example is very simple. This is a program for a movie rental store that calculates each customer's. The amount of consumption is printed and the details are printed. The operator tells the program: which videos the customer rents and how long the rental period is, the program calculates the fee according to the rental time and the type of the movie. The film is divided into three categories: ordinary, children and new. Fees, but also for the frequent customers to calculate points, the points will be different depending on whether the type of rental film is a new film [1]."



Movie 1

priceCode:int

Movie

priceCode:int

\*

Movie

priceCode:int

\*

1

# Picture 1. film rental store UML diagram

# Take the example of the stateMent method of “customer.go”：

# func (customer \*Customer)StateMent() string{ totalAmount := float64(0.0) frequentRenterPoints:= 0 result := "Rental Record for " + customer.GetName()+"\n" fmt.Println(len(customer.rentals)) for \_,rental:=range customer.rentals{ thisAmount:= float64(0.0) switch rental.GetMovie().GetPriceCode() { case REGULAR: thisAmount+=2 if rental.GetDaysRented()>2{ thisAmount +=float64((rental.GetDaysRented()-2))\*1.5 } case NEW\_RELEASE: thisAmount += float64(rental.GetDaysRented()\*3) case CHILDRENS: thisAmount+=1.5 if rental.GetDaysRented()>3{ thisAmount+=(float64)(rental.GetDaysRented()-3)\*1.5 } } frequentRenterPoints++ if(rental.GetMovie().GetPriceCode()==NEW\_RELEASE)&& (rental.GetMovie().GetPriceCode()>1){ frequentRenterPoints++ } result+=rental.GetMovie().GetTitle()+":"+str-conv.FormatFloat(thisAmount,'f',-1,64)+"\n" totalAmount +=thisAmount } result+="\nAmountowed is "+strconv.Format Float(totalAmount,'f',-1,64) +"\n" result+="Youearned"+strconv.FormatInt(int64(frequentRenterPoints),10)+" frequent renter points" return result }

# If you want to unit test the method “func ( customer \*Cus- tomer) StateMent() string” of the customer class, you can't unit test if you don't implement the Rental class and the Movie class method (black font in the code).

# Solution

# Does the Golang language have a unit testing tool MockCpp like C and C++? GoStub and GoMock are commonly known, and there is also a Golang unit test piling tool Go Monkey Patching on Github, referred to as "Monkey".

# Compare the advantages and disadvantages of several tools：

# GoStub needs to point variables to the return value of the piling function and class method when piling, but it can't piling the function and class method without return value; it needs to change the code under test when it is used, which is an intrusive modification; Unable to piling the interface. GoMock generates stub files and executes commands. The interface can be easily piling, but it is difficult to piling for functions and class methods. Although Monkey has no way to piling the interface, it is a non-intrusive modification during piling. It does not require any modification of the code under test. It is convenient to piling the functions and class methods. As shown in Table 1.

Table 1 Golang unit test tool comparison

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Non-invasive | interface | function | method |
| GoStub | N | N | Y | Y |
| GoMock | Y | Y | N | N |
| Monkey | Y | N | Y | Y |

1. Practical application

The following demonstrates how to use Monkey to test a functional unit in an example. According to the TDD practice method, first write StateMent to rely on the Rental class and Movie class methods to make the unit test compile, but not the method implementation.

Then use Monkey to piling the dependent class methods. The black font is implemented by Monkey on the dependent class method. After piling, use fmt.Println to call several methods to get the desired output.

customer\_test.go文件

func Test\_Customer\_Should\_Calc\_By\_Regular\_Rule\_When\_One\_Rental\_Rent\_Regular\_Movie(t \*testing.T) {  
 …  
 guardGetPriceCode : = monkey.PatchInstanceMethod(reflect.TypeOf(customer.rentals[0].GetMovie()), "GetPriceCode",  
 func(movie \*Movie) int {  
 return 0  
 })  
 defer guardGetPriceCode.Unpatch()  
 guardGetTitle := monkey.PatchInstanceMethod (reflect.TypeOf (customer.rentals [0].GetMovie ()), "GetTitle",  
 func(movie \*Movie) string {  
 return "game of chrones"  
 })  
 defer guardGetTitle.Unpatch()  
 guardStart : = monkey.PatchInstanceMethod (reflect.TypeOf (customer.rentals [0]), "GetDaysRented",  
 func(rental \*Rental) int {  
 return 5  
 })  
 defer guardStart.Unpatch()  
 …  
}

1. Conclusion

By comparing and introducing Monkey, you can make up for Golang's official unit testing tool Testing defect, and piling to ensure decoupling between function-level code. It is a piling tool for Golang to practice TDD.

References

[1] Martin Fowler, Xiong-Jie. Refactoring - improving code design for both code. BJ: People Post Press, 2015.

[2] Xu-ShiWei, lv-GuiHua. Go language programming. BJ: People Post Press， 2016.