**Golang单元测试入门实践总结**

背景: 之前很长一段时间再写Golang程序时，不会有意识去写单元测试，直到后来写了独立项目后，慢慢才发现给一个功能编写对应的单元测试是多么高效和方便，接下来就再一起复习下Golang中的测试.

#### UnitTest(单元测试)

单元测试是程序开发者适用一段代码来验证另外一段代码写的是否符合预期的一种相对高效的自我测试方法。

还记得最早开始搞运维时，写的程序基本上是通过 **main** 程序去调用具体的功能函数，然后通过具体的输出来主观验证结果是否符合预期，这种方式对于搞正统的软件开发者而言会感觉很傻，但这对于运维领域来说却很实用，很有效，因为通常运维工作中需要的一些开发都不会是逻辑较为复杂的程序，所以没有必要专门去写测试程序去测试另外一个程序是否符合预期。

但是随着工作内容和运维需求的变化，不得不使用一些正规软件工程领域的相关方法来进行测试，因为对于程序开发来说，经过长期的积累和方法总结，单元测试是一种比较好的开发程序验证方式，而且能够提高程序开发的质量。而在 **Golang** 语言中内置了一系列的测试框架，加下来就主要讲讲 **UnitTest** 单元测试的相关知识点。

#### UnitTest的编写

**注意:** 在Golang中，对于单元测试程序来说通常会有一些重要约束，主要如下:

xxx\_test.**go**

Testxxx

t \*testing.T

**package**

# 示例文件

# 假设我们为某段业务逻辑专门写了一个**package**(用来初始化一个矩形，并计算体积)，此时看到到整体结构如下

$ tree -L 2 ./unittest

./unittest

├── area.**go**

└── area\_test.**go**

# 业务逻辑代码(业务逻辑需要和单元测试在一个**package**下)

$ cat ./unittest/area.**go**

**package** unittest

**type** box **struct** {

length **int**

width **int**

height **int**

name **string**

}

// 初始化一个结构体指针对象，后面使用结构体指针方法来设置和获取对象属性

**func** **Newbox**() (\*box) {

**return** &box{}

}

// 给结构体对象设置具体的属性(名称，规格大小)

// 注意: 在如下几个方法中，方法接受者为指针类型，而方法参数为值类型，因此在赋值时可能有人产生疑惑，这里其实是Golang底层做了优化(v.name = name 等同于(\*v).name = name)

**func** (v \*box) **SetName**(name **string**) {

v.name = name

}

**func** (v \*box) **SetSize**(l,w,h **int**) {

v.length = l

v.width = w

v.height = h

}

// 获取对象的一些属性(名称和体积)

**func** (v \*box) **GetName**() (**string**) {

**return** v.name

}

**func** (v \*box) **GetVolume**() (**int**) {

**return** (v.length)\*(v.width)\*(v.height)

}

# 对应业务逻辑的单元测试逻辑

$ cat unittest/area\_test.**go**

**package** unittest

// 必须导入testing模块，并且方法的接受者为(t \*testing.T)

**import** (

"fmt"

"testing"

)

// 测试1: 测试名称是否符合预期

**func** **TestSetSomething**(t \*testing.T) {

box := Newbox()

box.SetName("bgbiao")

**if** box.GetName() == "bgbiao" {

fmt.Println("the rectangular name's result is ok")

}

}

// 测试2: 测试计算出来的体积是否符合预期

**func** **TestGetSomething**(t \*testing.T) {

box := Newbox()

box.SetSize(3,4,5)

**if** box.GetVolume() == 60 {

fmt.Println("the rectangular volume's result is ok")

}

}

# 运行单元测试程序

# 可以看到我们编写的两个单元测试都经过预期测试

$ cd unittest

$ **go** test

the rectangular name's result is ok

the rectangular volume's result is ok

PASS

ok \_/User/BGBiao/unittest 0.005s

#### 单元测试的运行

通过上面那个测试示例，我们都知道了可以使用 **go test** 来对Golang代码进行测试，接下来具体讲解一些 **go test** 的其他用法(其实上面说的那些规则也可以在 **go help test** 帮助文档中找到)

这里主要总结下几个常用的参数:

* -args: 指定一些测试时的参数(可以指定超时时间,cpu绑定,压测等等(go test包含单元测试，压力测试等))
  + -test.v: 是否输出全部的单元测试用例（不管成功或者失败），默认没有加上，所以只输出失败的单元测试用例
  + -test.run pattern: 只跑哪些单元测试用例
  + -test.bench patten: 只跑那些性能测试用例
  + -test.benchmem : 是否在性能测试的时候输出内存情况
  + -test.benchtime t : 性能测试运行的时间，默认是1s
  + -test.cpuprofile cpu.out : 是否输出cpu性能分析文件
  + -test.memprofile mem.out : 是否输出内存性能分析文件
  + -test.blockprofile block.out : 是否输出内部goroutine阻塞的性能分析文件
* -c: 编译测试文件到pkg.test,但是不会运行测试程序
* -exec xprog: 使用xprog参数来运行编译的测试文件(参数类似go run后的参数)
* -i: 安装测试程序中的依赖包，但是不运行测试程序
* -json: 以json格式输出测试结果
* -o file: 指定测试程序编译后生成的文件名

单元测试中常用的命令参数:

# 对当前目录下的全部单元测试程序进行运行测试(也就是所有的xxx\_test.**go**文件中的所有function都会运行)

$ **go** test

the rectangular name's result is ok

the rectangular volume's result is ok

PASS

ok \_/Users/BGBiao/unittest 0.005s

# 查看详细的单元测试结果

# (**go** test -v 等同于**go** test -args -test.v)

$ **go** test -v

=== RUN TestSetSomething

the rectangular name's result is ok

--- PASS: TestSetSomething (0.00s)

=== RUN TestGetSomething

the rectangular volume's result is ok

--- PASS: TestGetSomething (0.00s)

PASS

ok \_/Users/BGBiao/unittest 0.005s

# 指定单元测试function来进行测试

# **go** test -v -run functionname

$ **go** test -v -test.run TestGetSomething

=== RUN TestGetSomething

the rectangular volume's result is ok

--- PASS: TestGetSomething (0.00s)

PASS

ok \_/Users/BGBiao/unittest 0.005s

#### 单元测试注意事项

**注意:** 在单元测试时，一个比较重要的事情就是如何构造测试数据，因为通常我们能够想到的测试数据都是在预期之中的，有些核心逻辑的测试数据往往不能考虑到，因此构造测试数据时可考虑如下几个方面:

* 1. 正常输入: 正常的可预测的测试用例
  + 边界输入: 极端情况下的输入来测试容错性
  + 非法输入: 输入异常数据类型，整个逻辑是否能够正常处理或者捕获
  + 白盒覆盖: 需要设计的测试用例能够覆盖所有代码(语句覆盖、条件覆盖、分支覆盖、分支/条件覆盖、条件组合覆盖)

**注意:** 在写项目时，对于基础的工具层 **util** 的逻辑代码，一定要进行全方位，多场景的进行测试，否则当项目大起来后到处引用可能会造成较大麻烦;其次，我们的代码逻辑通常是更新迭代的，单元测试代码也应该进行定期更新.