为什么不建议在go项目中使用init()

更新时间：2021年04月12日 08:51:55 作者：机智的小小帅

前言

`go` 的 `init` 函数给人的感觉怪怪的，我想不明白聪明的 `google` 团队为何要设计出这么一个"鸡肋"的机制。实际编码中，我主张尽量不要使用 `init` 函数。

首先来看看 `init` 函数的作用吧。

init() 介绍

`init()` 与包的初始化顺序息息相关，所以先介绍一个 `go` 中包的初始化顺序吧。（下面的内容部分摘自《The go programinng language》）

大体而言，顺序如下：

- 首先初始化包内声明的变量
- 之后调用 `init` 函数
- 最后调用 `main` 函数

变量的初始化顺序

变量的初始化顺序由他们的依赖关系决定

应该任何强类型语言都是这样子吧。

例如：

```
var a = b + c;
var b = f();    // 需要调用 f() 函数
var c = 1
func f() int{return c + 1;}
```

`a` 依赖 `b` 和 `c`；`b` 依赖 `f()`，`f()` 依赖 `c`。因此，他们的初始化顺序理所当然是 `c -> b -> a`。

graph TB; b-->a c-->a f-->b c-->b

Ps：其实在这里可能引申出一个没用的小技巧。当你有一个函数需要在包被初始化的过程中被调用时，你可以把这个函数赋值给一个包级变量。这样，当包被初始化时就会自动调用这个函数了，这个函数甚至能够在 `init()` 之前被调用！不过话说回来，它既然比 `init()` 更早被调用，那它才是真正的 `init()` 才对；此外你也可以在 `init()` 中调用该函数，这样才更合理一些。

```go
// 笨版
// 函数必须得有一个返回值才行
var _ = func() interface{} {
    fmt.Println("hello")
    return nil
}()

func init() {
    fmt.Println("world")
}

func main() {

}
// Output:
// hello
// world
```

```go
// 更合理的版本
func init() {
    fmt.Println("hello")
    fmt.Println("world")
}

func main() {

}
// Output:
// hello
// world
```

包内变量的初始化顺序

一个包内往往有多个 `go` 文件，这么 `go` 文件的初始化顺序由它们被提交给编译器的顺序决定，顺序和这些文件的名字有关。

`init()`

主角出场了。先来看看它的设计动机吧：

Each variable declared at package level starts life with the value of its initializer expression, if any, but for some variables, like tables of data,an initializer expression may not be the simplest way to set its initial value.In that case,the init function mechanism may be simpler. 《The go pragramming language P44》

这句话的意思是有的包级变量没办法用一条简单的表达式来初始化，这个 时候， `init` 机制就派上用场了。

`init()` 不能被调用，也不能被 reference，它们会在程序启动时自动执行。

同一个 go 文件中 init 函数的调用顺序

一个包内，甚至 `go` 文件内可以包含多个 `init()` ，同一个 `go` 文件中的 `init()` 调用顺序由他们的声明顺序决定 。

```
func init() {
    fmt.Print("a")
}
func init() {
    fmt.Print("b")
}
func init() {
    fmt.Print("c")
}
// Output
// abc
```

同一个包下面不同 `go` 文件中 `init()` 的调用顺序

依旧是由它们的声明顺序决定，同一个包下面的所有 `go` 文件在编译时会被编译器合并成一个"大的 `go` 文件"（并不是真正合并，仅仅是效果类似而已）。合并的顺序由编译器决定。

不要把程序是否能够正常工作寄托在 `init()` 能够按照你期待的顺序被调用上。

不过话说回来，正经人谁在一个包里写很多 `init()` 呀，而且还把这些 `init()` 放在不同文件里，更可恶的是每个文件里还有多个 `init()` 。要是看到这样的代码，我立马：@#$%^&*...balabala...

一个包里最多写一个init()（我甚至觉得最好连一个 `init()` 都不要有）

> **不同包内 init 函数的调用顺序**

唯独这个顺序，我们程序员是绝对可控的。它们的调用顺序由包之间的依赖关系决定。假设 `a` 包需要 `import` `b` 包，`b` 包需要 `import` `c` 包，那么很显然他们的调用顺序是，`c` 包的 `init()` 最先被调用，其次是 `b` 包，最后是 `a` 包。

graph LRc-->bb-->a

一个包的init函数最多会被调用一次

道理类似于一个变量最多会被初始化一次。

有的同学会问，一个变量明明可以多次赋值呀，可第二次对这个变量赋值那还能够叫初始化么？

例如有如下的包结构，`B` 包和 `C` 包都分别 `import` `A` 包，D包需要 `import` `B` 包和 `C` 包。

graph TD; A-->B A-->C B-->D C-->D

在 `A` 包中有 `init()`

```
func init() {
    fmt.Println("hello world")
}
```

`D` 包是 `main` 包，最终程序只输出了一句 `hello world` 。

我不喜欢 init 函数的原因

我不喜欢 `init` 函数的一个重要原因是，它会隐藏掉程序的一些细节，它会在没有经过你同意的情况下，偷偷干一些事情。`go` 的函数王国里，所有的函数都需要程序员显示的调用(Call)才会被执行，只有它—— `init()` ，是个例如，你明明没 Call 它，它却偷偷执行了。

有的同学会说，`c++` 里类的构造函数也是在对象被创建时就会默默执行呀。确实是这样，但在 `c++` 里，当你点进这个类的定义时，你就能立马看到它的构造函数和析构函数。在 `go` 里，当你点进某个包时，你能立马看到包内的 `init()` 么？这个包有没有 `init()` 以及有几个 `init()` 完全是个未知数，你需要在包内的所有文件中搜索 `init()` 这个关键字才能摸清包的 `init()` 情况，而大多数人包括我懒得费这个功夫。在 `c++` 中创建对象时，程序员能够很清楚的意识到这个操作会触发这个类的构造函数，这个构造函数的内容也能很快找到；但在 `go` 中，`import` 包时，一切却没那么清晰了。

希望将来 goland 或者 vscode 能够分析包内的 `init()` 情况，这样我对 `init()` 的恶意会减半。

`init()` 给项目维护带来的困难

当你看到这样的 `import` 代码时

```
import(
    _ "pkg"
)
```

你立马能够知道，这个 `import` 的目的就是调用 `pkg` 包的 `int()` 。

当看到

```
import(
    "pkg"
)
```

你却很难知道， `pkg` 包里藏着一个 `init()` ，它被偷偷调用了。

但这还好，你起码知道如果 `pkg` 包有 `init()` 的话，它会在此处被调用。

但当 `pkg` 包，被多个包 `import` 时， `pkg` 包内的 `init()` 何时被调用的，就是一个谜了。你得搞清楚这些包之间的 `import` 先后顺序关系，这是一场噩梦。

使用 `init()` 的时机

先说一下我的结论：我认为 `init()` 应该仅被用来初始化包内变量。

《The go programming language》提供了一个使用 `init` 函数的例子。

```
// pc[i] 是 i 中 bit = 1 的数量
var pc [256]byte

func init() {
    for i := range pc {
        pc[i] = pc[i/2] + byte(i&1)
    }
}

// 返回 x 中等于 1 的 bit 的数量
func PopCount(x uint64) int {
    return int(pc[byte(x>>(0*8))] +
        pc[byte(x>>(1*8))] +
        pc[byte(x>>(2*8))] +
```

```
        pc[byte(x>>(3*8))] +
        pc[byte(x>>(4*8))] +
        pc[byte(x>>(5*8))] +
        pc[byte(x>>(6*8))] +
        pc[byte(x>>(7*8))])
}
```

`PopCount` 函数的作用数计算数字中等于 `1` 的 `bit` 的数量。例如：

```
var i uint64 = 2
```

变量 i 的二进制表示形式为

> 0000000000000000000000000000000000000000000000000000000000000010

把它传入 `PopCount` 最终得到的结果将为 `1`，因为它只有一个 `bit` 的值为 `1`。

`pc` 是一个表，它的 index 为 x，其中 0 <= x <= 255，value 为 x 中等于 1 的 bit 的数量。

它的初始化思想是：

- 如果一个数x最后的 bit 为 1，那么这个数值为 1 的bit数 = x/2 的值为1的bit数 + 1；
- 如果一个数x最后的 bit 为 0，那么这个数值为 1 的bit数 = x/2 的值为1的bit数；

在 `PopCount` 中把一个 8byte 数拆成了 8 个单 byte 数，分别计算这8个单 byte 数中 `bit` 为 `1` 的数量，最后累加即可。

这里 `pc` 的初始化确实比较复杂，无法直接用

```
var pc = []byte{0, 1, 1,...}
```

这种形式给出。

一个可以替代 `init()` 的方法是：

```
var pc = generatePc()

func generatePc() [256]byte {
    var localPc [256]byte
    for i := range localPc {
        localPc[i] = localPc[i/2] + byte(i&1)
```

```
    }
    return localPc
}
```

我觉得这样子初始化比利用 `init()` 初始化要更好，因为你可以立马知道 `pc` 是怎样得来的，而利用 `init()` 时，你需要利用 ide 来查找 `pc` 的 write reference，之后才能知道，哦，原来它(`pc`)来这里(`init()`) 被初始化了呀。

当包内有多个变量的初始化流程比较复杂时，可能会写出如下代码。

```
var pc = generatePc()
var pc2 = generatePc2()
var pc3 = generatePc3()
// ...
```

有的同学可能不太喜欢这种写法，那么用上 `init()` 后，会写成这样

```
func init() {
    initPc()
    initPc2()
    initPc3()
}
```

我觉得两种写法都说的过去吧，虽然我个人更倾向第一种写法。

使用 `init()` 的时机，仅仅有一个例外，后面说。

不使用 init 函数的时机

`init()` 除了初始化变量，不应该干其他任何事！

有两个原则：

- 一个包的 `init()` 不应该依赖包外的环境
- 一个包的 `init()` 不应该对包外的环境造成影响

设置这两个原则的理由是：任何对外部有依赖或者对外部有影响的代码都有义务显式的让程序员知晓，不应该自己悄咪咪地去做，最好是显式地让程序员自己去调用。

`init()` 的活动范围就应该仅仅被局限在包内，自己和自己玩，不要影响了其他小朋友的游戏体验。

如下几条行为就踩了红线：

- 读取配置（依赖于外部的配置文件，且一般读取配置得到的 obj 会被其他包访问，违反了第一条和第二条）
- 注册路由（因为修改了 `http` 包中的 routeMap，会对 `http` 包造成影响，违反了第二条）
- 连接数据库（连接数据库后一般会得到一个 db 对象给业务层去curd吧？违反了第二条）
- etc... 我暂时只能想到这么多了

一个反面教材 https://github.com/go-sql-driver/mysql

反面教材就是：https://github.com/go-sql-driver/mysql 这个大名鼎鼎的包

当使用这个包时，一个必不可少的语句是：

```
import (
    _ "github.com/go-sql-driver/mysql"
)
```

原因是它里面有个 `init` 函数，会把自己注册到 sql 包里。

```
func init() {
    sql.Register("mysql", &MySQLDriver{})
}
```

按照之前的标准，此处明显不符合规范，因为它影响了标准库的 `sql` 包。

我认为一个更好的方法是，创建一个 `export` 的专门用来做初始化工作的方法：

```
// Package mysql
func Init() {
    sql.Register("mysql", &MySQLDriver{})
}
```

然后在 `main` 包中显式的调用它：

```
// Package main
func main(){
    mysql.Init();
    // other logic
}
```

来比较一下两种方式吧。

- 使用 `Init()`
- 是否需要告诉开发者额外的信息?

需要。

需要告诉开发者：使用这个库时，记得一定要调用 Init() 哦，我在里面做了一些工作。

开发者，点进 `Init()` ，瞬间了然。

- 是否能够阻止开发者不正确的调用?

不能。

因为是 `export` 的，所以开发者可以想到哪儿调用就到哪儿调用，想调用多少次就调用多少次。

因此需要额外告诉开发者：请您务必只调用一次，之后就不要调用了。且必须在用到 `sql` 包之前调用，一般而言都是在 `main()` 的第一句调用。

- 使用 `init()`
- 是否需要告诉开发者额外的信息?

需要

依旧需要告诉开发者，一定要用 `_ "github.com/go-sql-driver/mysql"` 这个语句显式的导入包哦，因为我利用 `init()` 在里面做一些工作。

开发者：那你做了什么工作

库：亲，请您点进 `mysql` 包，在目录下搜索 `init()` 关键字，慢慢找哦。

开发者：.....

- 是否能够阻止开发者不正确的调用?

勉强可以吧。

因为 `init()` 只会被调用一次，不可能被调用多次，这从根本上杜绝了开发者调用多次的可能性。

可你管不了开发者的 `import` 时机，假设开发者在其他地方 `import` 了，导致你在 `sql.Open()` 时， `mysql` 的 `driver` 没有被正常注册，你还是拿开发者没有办法。只能哀叹一声：我累了，毁灭吧。

我觉得作为库的提供者，最主要的是提供完善的机制，在用户使用你的库时，能利用你提供的机制，写出无bug 的代码。而不是像保姆一样，想方设法避免用户出错。

所以可能使用 `init()` 为了的优势就是减少了代码量吧。

使用 `Init()` 时，需要两句代码

```
import (
    "github.com/go-sql-driver/mysql"    // 这句
)

func main(){
    mysql.Init()                  // 这句
}
```

但是使用 `init` 时，却只需要一句代码

```
import (
    _ "github.com/go-sql-driver/mysql"  // 这句
)
```

oh yeah，足足少写了一句代码！

一个例外 单元测试

可能使用 `init` 的唯一例外就是写单元测试的时候了吧。

假设我现在需要需要对 `dao` 层的增删改查逻辑的写一个单元测试。

```
func TestCURDPlayer(t *testing.T) {
    // 测试 curd 玩家信息
}

func TestCURDStore(t *testing.T) {
    // 测试 curd 商店信息
}

func TestCURDMail(t *testing.T) {
    // 测试 curd 邮件信息
}
```

很显然，这些测试都是依赖数据库的，因此为了正常的测试，必须初始化数据库

```go
func TestCURDPlayer(t *testing.T) {
    // 测试 curd 玩家信息
    initdb()
    // balabala
}

func TestCURDStore(t *testing.T) {
    // 测试 curd 商店信息
    initdb()
    // balabala
}

func TestCURDMail(t *testing.T) {
    // 测试 curd 邮件信息
    initdb()
    // balabala
}

func initdb(){
    // sql.Open()...
}
```

难道我每次新增一个单元测试，都要在单元测试的代码中加一个 `initdb()` 么，这也太麻烦了吧。

这个时候 `init()` 就派上用场了。可以这样

```go
func TestCURDPlayer(t *testing.T) {
    // 测试 curd 玩家信息
    // balabala
}

func TestCURDStore(t *testing.T) {
    // 测试 curd 商店信息
    // balabala
}

func TestCURDMail(t *testing.T) {
    // 测试 curd 邮件信息
    // balabala
}

func init(){
    initdb()
}

```

```
func initdb(){
    // sql.Open()...
}
```

这样，当对这个文件进行单元测试时，可以确保在执行每个 `TestXXX` 函数时， `db` 肯定是被正确初始化了的。

那为什么这个地方可以利用 `init()` 来初始化数据库呢?

理由之一是它的影响范围很小，仅仅在 `xxx_test.go` 文件中生效，在 `go run` 时不会起作用，在 `go test` 时才会起作用。

理由之二是我懒。。。

总结

`init` 更像是一个语法糖，它会让开发者对代码的追踪能力变弱，所以能不用就最好不用。

原文链接：https://www.cnblogs.com/XiaoXiaoShuai-/p/14642055.html

go init