Go 每日一库之 wire


孤雨
李子家的程序猿

关注他

28 人赞同了该文章

简介

之前的一篇文章Go 每日一库之 dig介绍了 uber 开源的依赖注入框架 `dig` 。读了这篇文章后，@overtalk推荐了 Google 开源的 `wire` 工具。所以就有了今天这篇文章，感谢推荐

`wire` 是 Google 开源的一个依赖注入工具。它是一个代码生成器，并不是一个框架。我们只需要在一个特殊的 `go` 文件中告诉 `wire` 类型之间的依赖关系，它会自动帮我们生成代码，帮助我们创建指定类型的对象，并组装它的依赖。

快速使用

先安装工具：

```
$ go get github.com/google/wire/cmd/wire
```

上面的命令会在 `$GOPATH/bin` 中生成一个可执行程序 `wire` ，这就是代码生成器。我个人习惯把 `$GOPATH/bin` 加入系统环境变量 `$PATH` 中，所以可直接在命令行中执行 `wire` 命令。

下面我们在一个例子中看看如何使用 `wire` 。

现在，我们来到一个黑暗的世界，这个世界中有一个邪恶的怪兽。我们用下面的结构表示，同时编写一个创建方法：

```
type Monster struct {
  Name string
}
```

```
func NewMonster() Monster {
  return Monster{Name: "kitty"}
}
```

有怪兽肯定就有勇士，结构如下，同样地它也有创建方法：

```
type Player struct {
  Name string
}

func NewPlayer(name string) Player {
  return Player{Name: name}
}
```

终于有一天，勇士完成了他的使命，战胜了怪兽：

```
type Mission struct {
  Player  Player
  Monster Monster
}

func NewMission(p Player, m Monster) Mission {
  return Mission{p, m}
}

func (m Mission) Start() {
  fmt.Printf("%s defeats %s, world peace!\n", m.Player.Name, m.Monster.Name)
}
```

这可能是某个游戏里面的场景哈，我们看如何将上面的结构组装起来放在一个应用程序中：

```
func main() {
  monster := NewMonster()
  player := NewPlayer("dj")
  mission := NewMission(player, monster)

  mission.Start()
}
```

代码量少，结构不复杂的情况下，上面的实现方式确实没什么问题。但是项目庞大到一定程度，结构之间的关系变得非常复杂的时候，这种手动创建每个依赖，然后将它们组装起来的方式就会变得异常繁琐，并且容易出错。这个时候勇士 wire 出现了！

wire 的要求很简单，新建一个 wire.go 文件（文件名可以随意），创建我们的初始化函数。比如，我们要创建并初始化一个 Mission 对象，我们就可以这样：

```
//+build wireinject

package main

import "github.com/google/wire"

func InitMission(name string) Mission {
  wire.Build(NewMonster, NewPlayer, NewMission)
  return Mission{}
}
```

首先这个函数的返回值就是我们需要创建的对象类型， wire 只需要知道类型， return 后返回什么不重要。然后在函数中，我们调用 wire.Build() 将创建 Mission 所依赖的类型的构造器传进去。例如，需要调用 NewMission() 创建 Mission 类型， NewMission() 接受两个参数一个 Monster 类型，一个 Player 类型。 Monster 类型对象需要调用 NewMonster() 创建， Player 类型对象需要调用 NewPlayer() 创建。所以 NewMonster() 和 NewPlayer() 我们也需要传给 wire 。

文件编写完成之后，执行 wire 命令：

```
$ wire
wire: github.com/darjun/go-daily-lib/wire/get-started/after: \
wrote D:\code\golang\src\github.com\darjun\go-daily-lib\wire\get-started\after\
```

我们看看生成的 wire_gen.go 文件：

```
// Code generated by Wire. DO NOT EDIT.
```

```
//go:generate wire
//+build !wireinject

package main

// Injectors from wire.go:

func InitMission(name string) Mission {
  player := NewPlayer(name)
  monster := NewMonster()
  mission := NewMission(player, monster)
  return mission
}
```

这个 InitMission() 函数是不是和我们在 main.go 中编写的代码一毛一样！接下来，我们可以直接在 main.go 调用 InitMission()：

```
func main() {
  mission := InitMission("dj")

  mission.Start()
}
```

细心的童鞋可能发现了， wire.go 和 wire_gen.go 文件头部位置都有一个 +build ，不过一个后面是 wireinject ，另一个是 !wireinject 。 +build 其实是 Go 语言的一个特性。类似 C/C++ 的条件编译，在执行 go build 时可传入一些选项，根据这个选项决定某些文件是否编译。 wire 工具只会处理有 wireinject 的文件，所以我们的 wire.go 文件要加上这个。生成的 wire_gen.go 是给我们来使用的， wire 不需要处理，故有 !wireinject 。

由于现在是两个文件，我们不能用 go run main.go 运行程序，可以用 go run . 运行。运行结果与之前的例子一模一样！

注意，如果你运行时，出现了 InitMission 重定义，那么检查一下你的 //+build wireinject 与 package main 这两行之间是否有空行，这个空行必须要有！见github.com/google/wire/。中招的默默在心里打个 1 好嘛

基础概念

wire 有两个基础概念， Provider （构造器）和 Injector （注入器）。 Provider 实际上就是创建函数，大家意会一下。我们上面 InitMission 就是 Injector 。每个注入器实际上就是一个对象的创建和初始化函数。在这个函数中，我们只需要告诉 wire 要创建什么类型的对象，这个类型的依赖， wire 工具会为我们生成一个函数完成对象的创建和初始化工作。

参数

同样细心的你应该发现了，我们上面编写的 InitMission() 函数带有一个 string 类型的参数。并且在生成的 InitMission() 函数中，这个参数传给了 NewPlayer() 。 NewPlayer() 需要 string 类型的参数，而参数类型就是 string 。所以生成的 InitMission() 函数中，这个参数就被传给了 NewPlayer() 。如果我们让 NewMonster() 也接受一个 string 参数呢?

```
func NewMonster(name string) Monster {
  return Monster{Name: name}
}
```

那么生成的 InitMission() 函数中 NewPlayer() 和 NewMonster() 都会得到这个参数：

```
func InitMission(name string) Mission {
  player := NewPlayer(name)
  monster := NewMonster(name)
  mission := NewMission(player, monster)
  return mission
}
```

实际上， wire 在生成代码时，构造器需要的参数（或者叫依赖）会从参数中查找或通过其它构造器生成。决定选择哪个参数或构造器完全根据类型。如果参数或构造器生成的对象有类型相同的情况，运行 wire 工具时会报错。如果我们想要定制创建行为，就需要为不同类型创建不同的参数结构：

```
type PlayerParam string
type MonsterParam string

func NewPlayer(name PlayerParam) Player {
```

```
  return Player{Name: string(name)}
}

func NewMonster(name MonsterParam) Monster {
  return Monster{Name: string(name)}
}

func main() {
  mission := InitMission("dj", "kitty")
  mission.Start()
}

// wire.go
func InitMission(p PlayerParam, m MonsterParam) Mission {
  wire.Build(NewPlayer, NewMonster, NewMission)
  return Mission{}
}
```

生成的代码如下：

```
func InitMission(m MonsterParam, p PlayerParam) Mission {
  player := NewPlayer(p)
  monster := NewMonster(m)
  mission := NewMission(player, monster)
  return mission
}
```

在参数比较复杂的时候，建议将参数放在一个结构中。

错误

不是所有的构造操作都能成功，没准勇士出山前就死于小人之手：

```
func NewPlayer(name string) (Player, error) {
  if time.Now().Unix()%2 == 0 {
    return Player{}, errors.New("player dead")
  }
```

```
    return Player{Name: name}, nil
}
```

我们使创建**随机失败**，修改注入器 InitMission() 的签名，增加 error 返回值：

```
func InitMission(name string) (Mission, error) {
  wire.Build(NewMonster, NewPlayer, NewMission)
  return Mission{}, nil
}
```

生成的代码，会将 NewPlayer() 返回的错误，作为 InitMission() 的返回值：

```
func InitMission(name string) (Mission, error) {
  player, err := NewPlayer(name)
  if err != nil {
    return Mission{}, err
  }
  monster := NewMonster()
  mission := NewMission(player, monster)
  return mission, nil
}
```

wire 遵循**fail-fast**的原则，错误必须被处理。如果我们的注入器不返回错误，但构造器返回错误， wire 工具会报错！

高级特性

下面简单介绍一下 wire 的高级特性。

ProviderSet

有时候可能多个类型有相同的依赖，我们每次都将相同的构造器传给 wire.Build() 不仅繁琐，而且不易维护，一个依赖修改了，所有传入 wire.Build() 的地方都要修改。为此， wire 提供

了一个 ProviderSet （构造器集合），可以将多个构造器打包成一个集合，后续只需要使用这个集合即可。假设，我们有关勇士和怪兽的故事有两个结局：

```
type EndingA struct {
  Player  Player
  Monster Monster
}

func NewEndingA(p Player, m Monster) EndingA {
  return EndingA{p, m}
}

func (p EndingA) Appear() {
  fmt.Printf("%s defeats %s, world peace!\n", p.Player.Name, p.Monster.Name)
}

type EndingB struct {
  Player  Player
  Monster Monster
}

func NewEndingB(p Player, m Monster) EndingB {
  return EndingB{p, m}
}

func (p EndingB) Appear() {
  fmt.Printf("%s defeats %s, but become monster, world darker!\n", p.Player.Nar
}
```

编写两个注入器：

```
func InitEndingA(name string) EndingA {
  wire.Build(NewMonster, NewPlayer, NewEndingA)
  return EndingA{}
}

func InitEndingB(name string) EndingB {
  wire.Build(NewMonster, NewPlayer, NewEndingB)
  return EndingB{}
}
```

我们观察到两次调用 wire.Build() 都需要传入 NewMonster 和 NewPlayer 。两个还好，如果很多的话写起来就麻烦了，而且修改也不容易。这种情况下，我们可以先定义一个 ProviderSet ：

```
var monsterPlayerSet = wire.NewSet(NewMonster, NewPlayer)
```

后续直接使用这个 set ：

```
func InitEndingA(name string) EndingA {
  wire.Build(monsterPlayerSet, NewEndingA)
  return EndingA{}
}

func InitEndingB(name string) EndingB {
  wire.Build(monsterPlayerSet, NewEndingB)
  return EndingB{}
}
```

而后如果要添加或删除某个构造器，直接修改 set 的定义处即可。

结构构造器

因为我们的 EndingA 和 EndingB 的字段只有 Player 和 Monster ，我们就不需要显式为它们提供构造器，可以直接使用 wire 提供的**结构构造器**（Struct Provider）。结构构造器创建某个类型的结构，然后用参数或调用其它构造器填充它的字段。例如上面的例子，我们去掉 NewEndingA() 和 NewEndingB() ，然后为它们提供结构构造器：

```
var monsterPlayerSet = wire.NewSet(NewMonster, NewPlayer)

var endingASet = wire.NewSet(monsterPlayerSet, wire.Struct(new(EndingA), "Playe
var endingBSet = wire.NewSet(monsterPlayerSet, wire.Struct(new(EndingB), "Playe

func InitEndingA(name string) EndingA {
  wire.Build(endingASet)
  return EndingA{}
}
```

```
func InitEndingB(name string) EndingB {
  wire.Build(endingBSet)
  return EndingB{}
}
```

结构构造器使用 wire.Struct 注入，第一个参数固定为 new(结构名)，后面可接任意多个参数，表示需要为该结构的哪些字段注入值。上面我们需要注入 Player 和 Monster 两个字段。或者我们也可以使用通配符 * 表示注入所有字段：

```
var endingASet = wire.NewSet(monsterPlayerSet, wire.Struct(new(EndingA), "*"))
var endingBSet = wire.NewSet(monsterPlayerSet, wire.Struct(new(EndingB), "*"))
```

wire 为我们生成正确的代码，非常棒：

```
func InitEndingA(name string) EndingA {
  player := NewPlayer(name)
  monster := NewMonster()
  endingA := EndingA{
    Player:  player,
    Monster: monster,
  }
  return endingA
}
```

绑定值

有时候，我们需要为某个类型绑定一个值，而不想依赖构造器每次都创建一个新的值。有些类型天生就是单例，例如配置，数据库对象（ sql.DB ）。这时我们可以使用 wire.Value 绑定值，使用 wire.InterfaceValue 绑定接口。例如，我们的怪兽一直是一个 Kitty ，我们就不用每次都去创建它了，直接绑定这个值就 ok 了：

```
var kitty = Monster{Name: "kitty"}

func InitEndingA(name string) EndingA {
```

```
  wire.Build(NewPlayer, wire.Value(kitty), NewEndingA)
  return EndingA{}
}

func InitEndingB(name string) EndingB {
  wire.Build(NewPlayer, wire.Value(kitty), NewEndingB)
  return EndingB{}
}
```

注意一点，这个值每次使用时都会拷贝，需要确保拷贝无副作用：

```
// wire_gen.go
func InitEndingA(name string) EndingA {
  player := NewPlayer(name)
  monster := _wireMonsterValue
  endingA := NewEndingA(player, monster)
  return endingA
}

var (
  _wireMonsterValue = kitty
)
```

结构字段作为构造器

有时候我们编写一个构造器，只是简单的返回某个结构的一个字段，这时可以使用 `wire.FieldsOf` 简化操作。现在我们直接创建了 `Mission` 结构，如果想获得 `Monster` 和 `Player` 类型的对象，就可以对 `Mission` 使用 `wire.FieldsOf`：

```
func NewMission() Mission {
  p := Player{Name: "dj"}
  m := Monster{Name: "kitty"}

  return Mission{p, m}
}

// wire.go
```

```
func InitPlayer() Player {
  wire.Build(NewMission, wire.FieldsOf(new(Mission), "Player"))
}

func InitMonster() Monster {
  wire.Build(NewMission, wire.FieldsOf(new(Mission), "Monster"))
}

// main.go
func main() {
  p := InitPlayer()
  fmt.Println(p.Name)
}
```

同样的，第一个参数为 new(结构名)，后面跟多个参数表示将哪些字段作为构造器， * 表示全部。

清理函数

构造器可以提供一个清理函数，如果后续的构造器返回失败，前面构造器返回的清理函数都会调用：

```
func NewPlayer(name string) (Player, func(), error) {
  cleanup := func() {
    fmt.Println("cleanup!")
  }
  if time.Now().Unix()%2 == 0 {
    return Player{}, cleanup, errors.New("player dead")
  }
  return Player{Name: name}, cleanup, nil
}

func main() {
  mission, cleanup, err := InitMission("dj")
  if err != nil {
    log.Fatal(err)
  }

  mission.Start()
  cleanup()
```

```
}

// wire.go
func InitMission(name string) (Mission, func(), error) {
  wire.Build(NewMonster, NewPlayer, NewMission)
  return Mission{}, nil, nil
}
```

一些细节

首先，我们调用 `wire` 生成 `wire_gen.go` 之后，如果 `wire.go` 文件有修改，只需要执行 `go generate` 即可。 `go generate` 很方便，我之前一篇文章写过 `generate` ，感兴趣可以看看深入理解Go之generate。

总结

`wire` 是随着 `go-cloud` 的示例 `guestbook` 一起发布的，可以阅读 `guestbook` 看看它是怎么使用 `wire` 的。与 `dig` 不同， `wire` 只是生成代码，不使用 `reflect` 库，性能方面是不用担心的。因为它生成的代码与你自己写的基本是一样的。如果生成的代码有性能问题，自己写大概率也会有。

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue

参考

wire GitHub：github.com/google/wire

Go 每日一库 GitHub：github.com/darjun/go-da

我

我的博客

欢迎关注我的微信公众号【GoUpUp】，共同学习，一起进步~

weixin.qq.com/r/9y-s9Of (二维码自动识别)