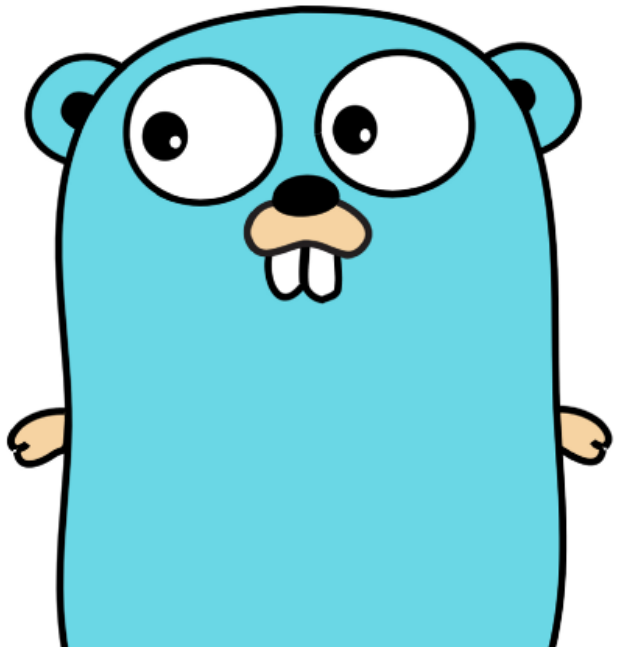
Che Dan 360 Followers About Follow Sign in Get started

You have **2** free member-only stories left this month. Sign up for Medium and get an extra one

一文读懂Wire

Che Dan Jan 1 · 15 min read ★

Master Wire
-- the DI tool in Golang



Wire 是啥

Wire 是一个轻巧的Golang依赖注入工具。它由Go Cloud团队开发，通过自动生成代码的方式在编译期完成依赖注入。

依赖注入是保持软件 “低耦合、易维护” 的重要设计准则之一。

此准则被广泛应用在各种开发平台之中，有很多与之相关的优秀工具。

其中最著名的当属 Spring，Spring IOC 作为框架的核心功能对Spring的发展到今天统治地位起了决定性作用。

事实上， 软件开发 S.O.L.I.D 原则 中的“D”， 就专门指代这个话题。

. . .

Wire 的特点

依赖注入很重要，所以Golang社区中早已有人开发了相关工具， 比如来自Uber 的 dig 、来自Facebook 的 inject 。他们都通过反射机制实现了运行时依赖注入。

为什么Go Cloud团队还要重造一遍轮子呢？ 因为在他们看来上述类库都不符合Go的哲学：

> *Clear is better than clever* ， *Reflection is never clear.*
>
> — *Rob Pike*

作为一个代码生成工具， Wire可以生成Go源码并在编译期完成依赖注入。 它不需要反射机制或 Service Locators 。 后面会看到， Wire 生成的代码与手写无异。 这种方式带来一系列好处：

1. 方便debug，若有依赖缺失编译时会报错
2. 因为不需要 Service Locators， 所以对命名没有特殊要求
3. 避免依赖膨胀。 生成的代码只包含被依赖的代码，而运行时依赖注入则无法作到这一点
4. 依赖关系静态存于源码之中， 便于工具分析与可视化

团队对Wire设计的仔细权衡可以参看 Go Blog 。

虽然目前Wire只发布了 v0.4 .0，但已经比较完备地达成了团队设定的目标。 预计后面不会有什么大变化了。 从团队傲娇的声明中可以看出这一点：

> *It works well for the tasks it was designed to perform, and we prefer to keep it as simple as possible.*
>
> *We'll not be accepting new features at this time, but will gladly accept bug reports and fixes.*
>
> — *Wire team*

. . .

上手使用

安装很简单，运行 `go get github.com/google/wire/cmd/wire` 之后， `wire` 命令行工具 将被安装到 `$GOPATH/bin` 。只要确保 `$GOPATH/bin` 在 `$PATH` 中， `wire` 命令就可以在任何目录调用了。

在进一步介绍之前， 需要先解释 wire 中的两个核心概念： Provider 和 Injector：

Provider: 生成组件的普通方法。这些方法接收所需依赖作为参数，创建组件并将其返回。

组件可以是对象或函数 —— 事实上它可以是任何类型，但单一类型在整个依赖图中只能有单一provider。因此返回 `int` 类型的provider 不是个好主意。 对于这种情况， 可以通过定义类型别名来解决。例如先定义`type Category int` ， 然后让 provider 返回 `Category` 类型

典型provider示例如下：

```
// DefaultConnectionOpt provide default connection option
func DefaultConnectionOpt()*ConnectionOpt{...}


// NewDb provide an Db object
func NewDb(opt *ConnectionOpt)(*Db, error){...}


// NewUserLoadFunc provide a function which can load user
func NewUserLoadFunc(db *Db)(func(int) *User, error){...}
```

实践中， 一组业务相关的provider时常被放在一起组织成 **ProviderSet**，以方便维护与切换。

```
var DbSet = wire.NewSet(DefaultConnectionOpt, NewDb)
```

Injector: 由`wire`自动生成的函数。函数内部会按根据依赖顺序调用相关privoder 。

为了生成此函数， 我们在 `wire.go` (文件名非强制，但一般约定如此)文件中定义injector函数签名。 然后在函数体中调用`wire.Build` ，并以所需provider作为参数（无须考虑顺序）。

由于`wire.go`中的函数并没有真正返回值，为避免编译器报错， 简单地用`panic`函数包装起来即可。不用担心执行时报错， 因为它不会实际运行，只是用来生成真正的代码的依据。一个简单的wire.go 示例

```
// +build wireinject

package main

import "github.com/google/wire"
```

```
func UserLoader()(func(int)*User, error){
   panic(wire.Build(NewUserLoadFunc, DbSet))
}

var DbSet = wire.NewSet(DefaultConnectionOpt, NewDb)
```

有了这些代码以后，运行 `wire` 命令将生成wire_gen.go文件，其中保存了injector 函数的真正实现。 wire.go 中若有非injector 的代码将被原样复制到 wire_gen.go 中（虽然技术上允许，但不推荐这样作）。 生成代码如下：

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

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

package main

import (
   "github.com/google/wire"
)

// Injectors from wire.go:

func UserLoader() (func(int) *User, error) {
   connectionOpt := DefaultConnectionOpt()
   db, err := NewDb(connectionOpt)
   if err != nil {
      return nil, err
   }
```

```
    v, err := NewUserLoadFunc(db)
    if err != nil {
       return nil, err
    }
    return v, nil
}

// wire.go:

var DbSet = wire.NewSet(DefaultConnectionOpt, NewDb)
```

上述代码有两点值得关注：

1. wire.go 第一行 `// +build wireinject` ，这个 build tag 确保在常规编译时忽略wire.go 文件（因为常规编译时不会指定 `wireinject` 标签）。 与之相对的是 wire_gen.go 中的 `//+build !wireinject` 。两组对立的build tag保证在任意情况下， wire.go 与 wire_gen.go 只有一个文件生效， 避免了“UserLoader方法被重复定义”的编译错误
2. 自动生成的UserLoader 代码包含了 error 处理。 与我们手写代码几乎相同。 对于这样一个简单的初始化过程， 手写也不算麻烦。 但当组件数达到几十、上百甚至更多时， 自动生成的优势就体现出来了。

要触发“生成”动作有两种方式： `go generate` 或 `wire` 。前者仅在 wire_gen.go 已存在的情况下有效（因为wire_gen.go 的第三行

`//go:generate wire`），而后者在任何时候都有可以调用。 并且后者有更多参数可以对生成动作进行微调， 所以建议始终使用 `wire` 命令。

然后我们就可以使用真正的injector了， 例如：

```
package main

import "log"

func main() {
   fn, err := UserLoader()
   if err != nil {
      log.Fatal(err)
   }
   user := fn(123)
   ...
}
```

如果不小心忘记了某个provider， `wire` 会报出具体的错误， 帮忙开发者迅速定位问题。 例如我们修改 `wire.go` ， 去掉其中的`NewDb`

```
// +build wireinject

package main

import "github.com/google/wire"
```

```
func UserLoader()(func(int)*User, error){
   panic(wire.Build(NewUserLoadFunc, DbSet))
}

var DbSet = wire.NewSet(DefaultConnectionOpt) //forgot add Db
provider
```

将会报出明确的错误：“`no provider found for *example.Db`”

```
wire: /usr/example/wire.go:7:1: inject UserLoader: no provider found
for *example.Db
      needed by func(int) *example.User in provider "NewUserLoadFunc"
(/usr/example/provider.go:24:6)
wire: example: generate failed
wire: at least one generate failure
```

同样道理， 如果在wire.go 中写入了未使用的provider , 也会有明确的错误提示。

. . .

高级功能

谈过基本用法以后， 我们再看看高级功能

接口注入

有时需要自动注入一个接口， 这时有两个选择：

1. 较直接的作法是在provider中生成具体类， 然后返回接口类型。 但这不符合Golang代码规范。一般不采用
2. 让provider返回具体类，但在injector声明环节作文章，将类绑定成接口，例如：

```
// FooInf, an interface
// FooClass, an class which implements FooInf
// fooClassProvider, a provider function that provider *FooClass


var set = wire.NewSet(
    fooClassProvider,
    wire.Bind(new(FooInf), new(*FooClass) // bind class to interface
)
```

属性自动注入

有时我们不需什么特定的初始化工作， 只是简单地创建一个对象实例， 为其指定属性赋值，然后返回。当属性多的时候，这种工作会很无聊。

```
// provider.go
```

```
type App struct {
    Foo *Foo
    Bar *Bar
}


func DefaultApp(foo *Foo, bar *Bar)*App{
    return &App{Foo: foo, Bar: bar}
}


// wire.go
...
wire.Build(provideFoo, provideBar, DefaultApp)
...
```

`wire.Struct` 可以简化此类工作， 指定属性名来注入特定属性：

```
wire.Build(provideFoo, provideBar, wire.Struct(new(App),"Foo","Bar")
```

如果要注入全部属性，则有更简化的写法：

```
wire.Build(provideFoo, provideBar, wire.Struct(new(App), "*")
```

如果struct 中有个别属性不想被注入，那么可以修改 struct 定义：

```
type App struct {
    Foo *Foo
    Bar *Bar
    NoInject int `wire:"-"`
}
```

这时 `NoInject` 属性会被忽略。与常规 provider相比， `wire.Struct` 提供一项额外的灵活性： 它能适应指针与非指针类型，根据需要自动调整生成的代码。

大家可以看到`wire.Struct`的确提供了一些便利。但它要求注入属性可公开访问， 这导致对象暴露本可隐藏的细节。

好在这个问题可以通过上面提到的“接口注入”来解决。用 `wire.Struct` 创建对象，然后将其类绑定到接口上。 至于在实践中如何权衡便利性和封装程度，则要具体情况具体分析了。

值绑定

虽不常见，但有时需要为基本类型的属性绑定具体值， 这时可以使用 `wire.Value`：

```
// provider.go
type Foo struct {
    X int
}


// wire.go
...
wire.Build(wire.Value(Foo{X: 42}))
...
```

为接口类型绑定具体值，可以使用 `wire.InterfaceValue`：

```
wire.Build(wire.InterfaceValue(new(io.Reader), os.Stdin))
```

把对象属性用作Provider

有时我们只是需要用某个对象的属性作为Provider，例如

```
// provider
func provideBar(foo Foo)*Bar{
    return foo.Bar
}


// injector
```

```
...
wire.Build(provideFoo, provideBar)
...
```

这时可以用 `wire.FieldsOf` 加以简化，省掉啰嗦的 provider:

```
wire.Build(provideFoo, wire.FieldsOf(new(Foo), "Bar"))
```

与 `wire.Struct` 类似， `wire.FieldsOf` 也会自动适应指针/非指针的注入请求

清理函数

前面提到若provider 和 injector 函数有返回错误， 那么wire会自动处理。除此以外，wire还有另一项自动处理能力： 清理函数。

所谓清理函数是指型如 `func()` 的闭包， 它随provider生成的组件一起返回， 确保组件所需资源可以得到清理。

清理函数典型的应用场景是文件资源和网络连接资源，例如：

```
```

```
type App struct {
   File *os.File
   Conn net.Conn
}

func provideFile() (*os.File, func(), error) {
   f, err := os.Open("foo.txt")
   if err != nil {
      return nil, nil, err
   }
   cleanup := func() {
      if err := f.Close(); err != nil {
         log.Println(err)
      }
   }
   return f, cleanup, nil
}

func provideNetConn() (net.Conn, func(), error) {
   conn, err := net.Dial("tcp", "foo.com:80")
   if err != nil {
      return nil, nil, err
   }
   cleanup := func() {
      if err := conn.Close(); err != nil {
         log.Println(err)
      }
   }
   return conn, cleanup, nil
}
```

上述代码定义了两个 provider 分别提供了文件资源和网络连接资源

wire.go

```
// +build wireinject

package main

import "github.com/google/wire"

func NewApp() (*App, func(), error) {
   panic(wire.Build(
      provideFile,
      provideNetConn,
      wire.Struct(new(App), "*"),
   ))
}
```

注意由于provider 返回了清理函数， 因此injector函数签名也必须返回，否则将会报错

wire_gen.go

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

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

package main

// Injectors from wire.go:

func NewApp() (*App, func(), error) {
   file, cleanup, err := provideFile()
```

```
    if err != nil {
        return nil, nil, err
    }
    conn, cleanup2, err := provideNetConn()
    if err != nil {
        cleanup()
        return nil, nil, err
    }
    app := &App{
        File: file,
        Conn: conn,
    }
    return app, func() {
        cleanup2()
        cleanup()
    }, nil
}
```

生成代码中有两点值得注意：

1. 当 `provideNetConn` 出错时会调用 `cleanup()`， 这确保了即使后续处理出错也不会影响前面已分配资源的清理。
2. 最后返回的闭包自动组合了 `cleanup2()` 和 `cleanup()` 。 意味着无论分配了多少资源， 只要调用过程不出错，他们的清理工作就会被集中到统一的清理函数中。 最终的清理工作由injector的调用者负责

可以想像当几十个清理函数的组合在一起时， 手工处理上述两个场景是非常繁琐且容易出错的。 wire 的优势再次得以体现。

然后就可以使用了：

```
func main() {
    app, cleanup, err := NewApp()
    if err != nil {
        log.Fatal(err)
    }
    defer cleanup()
    ...
}
```

注意main函数中的 `defer cleanup()` ，它确保了所有资源最终得到回收

. . .

总结

以上详细介绍了wire的概念、特点、使用方法及各种高级特性。

希望能帮你掌握这个小巧却强大的工具。

全文完。

p.s. 2020 新年快乐 ^-^

Golang Go Dependency Injection

Medium

About Help Legal