# 万字长文：在 Go 中如何优雅的使用 wire 依赖注入工具提高开发效率？上篇

==========

如果你做过 Java 开发，那么想必一定听说或使用过**依赖注入**。依赖注入是一种软件设计模式，它允许将组件的依赖项外部化，从而使组件本身更加**模块化和可测试**。在 Java 中，依赖注入广泛应用于各种框架中，帮助开发者解耦代码和提高应用的灵活性。本文就来介绍下什么是依赖注入，以及在 Go 语言中如何实践依赖注入，提高 Go 项目的开发效率和可维护性。

### 什么是依赖注入？

正如前文所述，[依赖注入](http://cxyroad.com/”https://zh.wikipedia.org/wiki/%E4%BE%9D%E8%B5%96%E6%B3%A8%E5%85%A5”)（dependency injection，缩写为 DI）是一种软件设计模式。

官方定义比较晦涩，我直接举个例子你就理解了。

在 Web 开发中，我们可以在 `store` 层（有些地方可会将其命名为 `repository`、`repo` 等）来操作数据库进行 CRUD。Go 语言中可以使用 GORM 操作数据库，所以 `store` 依赖 `*gorm.DB`，示例代码如下：

```
type userStore struct {
db *gorm.DB
}

func NewStore() *userStore {
db := NewDB()
return &userStore{db: db}
}

func (u *userStore) Create(ctx context.Context, user *model.UserM)
error {
return u.db.Create(&user).Error
}
```

> NOTE: 如果你对 GORM 不太了解，可以阅读我的另一篇文章[《Go 语言流行 ORM 框架 GORM 使用介绍》](http://cxyroad.com/”https://jianghushinian.cn/2023/05/27/go-popular-orm-framework-gorm-introduction/”)。

针对这一小段示例代码，我们可以按照如下方式创建一个用户：

```
store := NewStore()
store.Create(ctx, user)
```

我们还可以将示例代码修改成这样：

```
type userStore struct {
db *gorm.DB
}

func NewStore(db *gorm.DB) *userStore {
return &userStore{db: db}
}

func (u *userStore) Create(ctx context.Context, user *model.UserM)
error {
return u.db.Create(&user).Error
}
```

修改后示例代码中，我将 `*gorm.DB` 对象 `db` 的实例化过程，移动到了 `NewStore` 函数外面，在调用 `NewStore` 创建 `*userStore` 对象 `store` 时，将其通过参数形式传递进来。

现在，如果要创建一个用户，用法如下：

```
```

```
db := NewDB()
store := NewStore(db)
store.Create(ctx, user)
```

没错，我们已经在使用**依赖注入**了。

我们还是使用 `store.Create(ctx, user)` 创建用户。但构造 `store` 时，`*userStore` **依赖** `*gorm.DB`，我们使用构造函数 `NewStore` 创建 `*userStore` 对象，并且将它的**依赖对象** `*gorm.DB` 通过函数参数的形式**注入**进来，这种编程思想，就叫「依赖注入」。

回想一下，我们平时在编写 Go 代码的过程中，为了方便测试，是不是经常将某个方法的依赖项通过参数传递进来，而非在方法内部实例化，这就是在使用依赖注入编写代码。

我在文章[《在 Go 中如何编写出可测试的代码》](http://cxyroad.com/"https://jianghushinian.cn/2023/07/22/how-to-write-testable-code-in-go/#%E4%BD%BF%E7%94%A8%E4%BE%9D%E8%B5%96%E6%B3%A8%E5%85%A5%E6%9D%A5%E8%A7%A3%E5%86%B3%E5%A4%96%E9%83%A8%E4%BE%9D%E8%B5%96")中就有提到**如何使用依赖注入来解决外部依赖问题**，你可以点击文章进行阅读。

在 Go 中使用依赖注入的核心目的，就是为了解耦代码。这样做的主要好处是：

1. 方便测试。依赖由外部注入，方便使用 `fake object` 来替换依赖项。
2. 每个对象仅需要初始化一次，其他方法都可以复用。比如使用 `db := NewDB()` 初始化得到一个 `*gorm.DB` 对象，在 `NewUserStore(db)` 时可以使用，在 `NewPostStore(db)` 时还可以使用。

> NOTE: 我不太喜欢使用比较官方的话术来讲解技术，因为本来技术就需要理解成本，而官方的定义往往晦涩难懂。为了降低读者的心智负担，我更喜欢用白话讲解。
> 但说到「依赖注入」，定会有人提及「控制反转」。为了不一些让读者产生困惑，这里简单说明下控制反转和依赖注入的关系：[控制反转](http://cxyroad.com/"https://zh.wikipedia.org/wiki/%E6%8E%A7%E5%88%B6%E5%8F%8D%E8%BD%AC")（英语：Inversion of Control，缩写为 IoC），是面向对象

编程中的一种设计原则，可以用来减低计算机代码之间的耦合度。其中最常见的方式叫做[依赖注入](http://cxyroad.com/”https://zh.wikipedia.org/wiki/%E4%BE%9D%E8%B5%96%E6%B3%A8%E5%85%A5”)（dependency injection，缩写为 DI）。
> 我们可以简单的将控制反转理解为一种思想，而依赖注入是这一思想的具体实现方式。

### 依赖注入工具 Wire 简介

wire 是一个由 Google 开发的自动依赖注入框架，专门用于 Go 语言。wire 通过**代码生成而非运行时反射**来实现依赖注入，这与许多其他语言中的依赖注入框架不同。这种方法使得注入的代码在编译时就已经确定，从而提高了性能并保证了代码的可维护性。

#### 安装 Wire

wire 分成两部分，一个是在项目中使用的 Go 包，用于在代码中引用 wire 代码；另一个是命令行工具，用于生成依赖注入代码。

* 在项目中导入需要先通过 `go get` 获取 wire 依赖包。

```
$ go get –u github.com/google/wire
```

在 Go 代码中像其他 Go 包一样使用：

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

* 使用 `go install` 可以安装 wire 命令工具。

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

```

安装后通过 `--help` 标志执行 `wire` 命令查看其支持的所有子命令：

```
$ wire --help
Usage: wire <flags> <subcommand> <subcommand args>

Subcommands:
        check            print any Wire errors found
        commands         list all command names
        diff             output a diff between existing wire_gen.go files and
what gen would generate
        flags            describe all known top-level flags
        gen              generate the wire_gen.go file for each package
        help             describe subcommands and their syntax
        show             describe all top-level provider sets
```

由于绝大多数 wire 子命令不常用，所以这部分会放在本文最后再来讲解。

### Wire 快速开始

示例程序 `main.go` 代码如下：

```
package main

import "fmt"

type Message string

func NewMessage() Message {
return Message("Hi there!")
}

type Greeter struct {
Message Message
}
```
```

```
func NewGreeter(m Message) Greeter {
return Greeter{Message: m}
}

func (g Greeter) Greet() Message {
return g.Message
}

type Event struct {
Greeter Greeter
}

func NewEvent(g Greeter) Event {
return Event{Greeter: g}
}

func (e Event) Start() {
msg := e.Greeter.Greet()
fmt.Println(msg)
}
```

示例代码很好理解，定义了 `Message` 类型是 `string` 的类型别名。定义了 `Greeter` 类型及其构造函数 `NewGreeter`，并且接收 `Message` 作为参数，`Greeter.Greet` 方法会返回 `Message` 信息。最后还定义了一个 `Event` 类型，它存储了 `Greeter`，`Greeter` 通过构造函数 `NewEvent` 参数传递进来，`Event.Start` 方法会代理到 `Greeter.Greet` 方法。

定义如下 `main` 函数来执行这个示例程序：

```
func main() {
message := NewMessage()
greeter := NewGreeter(message)
event := NewEvent(greeter)

event.Start()
}
```

执行示例代码，得到如下输出：

```
$ go run main.go
Hi there!
```

可以发现，`main` 函数内部的代码有着明显的依赖关系，`NewEvent` 依赖 `NewGreeter`，`NewGreeter` 又依赖 `NewMessage`。

```
NewEvent –> NewGreeter –> NewMessage
```

我们可以将这部分代码进行抽离，封装到 `InitializeEvent` 函数中，保持入口函数 `main` 足够整洁，修改后代码如下：

```
func InitializeEvent() Event {
message := NewMessage()
greeter := NewGreeter(message)
event := NewEvent(greeter)
return event
}

func main() {
event := InitializeEvent()
event.Start()
}
```

现在是时候让 wire 登场了，在 `main.go` 同级目录创建 `wire.go` 文件（这是一个约定俗称的文件命名，不是强制约束）：

```
//go:build wireinject

package main

import (
```

```
”github.com/google/wire”
)

func InitializeEvent() Event {
wire.Build(NewEvent, NewGreeter, NewMessage)
return Event{}
}
```

我们将 `main.go` 文件中的 `InitializeEvent` 函数迁移过来，并且修改了内部逻辑，不再手动调用每个构造函数，而是将它们依次传递给 `wire.Build` 函数，然后使用 `return` 返回一个空的 `Event{}` 对象。

现在在当前目录下执行 `wire` 命令：

```
$ wire gen .
wire: github.com/jianghushinian/blog-go-example/wire/getting-started: wrote /Users/jianghushinian/projects/blog-go-example/wire/getting-started/wire_gen.go
```

其中：

* `gen` 是 `wire` 的子命令，他会扫描指定包中使用了 `wire.Build` 的代码，然后为其生成一个 `wire_gen.go` 的文件。
* `.` 表示当前目录，用于指定包，不指定的话默认就是当前目录。如果项目下有很多包，可以使用 `./...` 表示全部包，这个参数其实跟我们执行 `go test` 测试时是一个道理。

根据输出结果可以发现，`wire` 命令为我们在当前目录下生成了 `wire_gen.go` 文件，内容如下：

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

//go:generate go run -mod=mod github.com/google/wire/cmd/wire
//go:build !wireinject
// +build !wireinject
```

```
package main

// Injectors from wire.go:

func InitializeEvent() Event {
message := NewMessage()
greeter := NewGreeter(message)
event := NewEvent(greeter)
return event
}
```

神奇的事情发生了，`wire` 为我们生成了 `InitializeEvent` 函数的代码，并且跟我们自己实现的代码一模一样。

这就是 wire 的威力，它可以为我们自动生成依赖注入代码，只需要我们将所有依赖项（这里是几个构造函数）传给 `wire.Build` 即可。

由于现在当前目录下存在 3 个 `.go` 文件：

```
$ tree
.
├── go.mod
├── go.sum
├── main.go
├── wire.go
└── wire_gen.go
```

所以不能再使用 `go run main.go` 来执行示例代码了，可以使用 `go run .` 来执行：

```
$ go run .
Hi there!
```

细心的你可能会觉得疑惑，代码中有两处 `InitializeEvent` 函数的定义，程序编译执行的时候不会报错吗?

我们在 `wire.go` 中定义了 `InitializeEvent` 函数：

```
func InitializeEvent() Event {
wire.Build(NewEvent, NewGreeter, NewMessage)
return Event{}
}
```

然后 `wire` 命令帮我们在 `wire_gen.go` 中生成了新的 `InitializeEvent` 函数：

```
func InitializeEvent() Event {
message := NewMessage()
greeter := NewGreeter(message)
event := NewEvent(greeter)
return event
}
```

而且这二者都是在同一个包下。

程序没有编译报错，主要取决于 `wire.go` 和 `wire_gen.go` 文件中的 `//go:build` 注释。

在 `wire.go` 文件中，注释为：

```
//go:build wireinject
```

首先 `//go:build` 叫[构建约束](http://cxyroad.com/

”https://pkg.go.dev/go/build#hdr-Build_Constraints”）（build constraint）或构建标记（build tag），是一个必须放在 `.go` 文件最开始的注释代码。有了它之后，我们就可以告诉 `go build` 如何来构建代码。

其次，`wireinject` 是传递给构建约束的选项。选项就相当于一个 `if` 判断条件，可以根据选项来定制构建时如何处理 Go 文件。

这个构建约束有两个作用：

* 将此文件标记文件为 `wire` 处理的目标：`//go:build wireinject` 告诉 `wire` 工具及开发者，该文件包含使用 `wire` 进行依赖注入的设置。即这通常意味着文件中包含了 `wire.Build` 函数调用。有了它，文件才会被 `wire` 识别。
* 条件编译：确保在正常的构建过程中，带有这个构建约束的文件不会被编译进最终的可执行文件中。它只有在使用 `wire` 工具生成依赖注入代码时才被处理。这也是为什么代码不会编译报错，其实 `wire.go` 文件只是给 `wire` 命令用的，`go run .` 执行的是 `main.go` 和 `wire_gen.go` 两个文件，会忽略 `wire.go`。

注意：`//go:build wireinject` 和 `package main` 之间需要保留一个空行，否则程序会报错。你记住就行，不必过于纠结于此，这个问题在 wire 仓库的 [issues 117](http://cxyroad.com/”https://github.com/google/wire/issues/117”) 中也有提及。

我们再来看 `wire_gen.go` 文件，注释为：

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

//go:generate go run -mod=mod github.com/google/wire/cmd/wire
//go:build !wireinject
// +build !wireinject
```

第一行注释仅作为提示用，无特殊用途。

`//go:generate` 这行注释是一个 [go generate](http://cxyroad.com/”https://go.dev/blog/generate”) 指令。`go generate` 是一个由 Go 工具链提供的命令，用于在编译前自动执行生成代码的命令。这个特定的生成指令告诉 Go 在执行 `go generate` 命令时，运行 `wire` 工具来自动生成或更新

`wire_gen.go` 文件。

`go run -mod=mod github.com/google/wire/cmd/wire` 这部分指令运行 `wire` 命令，其中 `-mod=mod` 确保使用的是项目的 `go.mod` 文件中指定的依赖版本。

我们也可以验证下：

```
$ go generate
wire: github.com/jianghushinian/blog-go-example/wire/getting-started: wrote /Users/jianghushinian/projects/blog-go-example/wire/getting-started/wire_gen.go
```

执行 `go generate` 确实会自动执行 `wire` 命令。

注释 `//go:build !wireinject` 同样是一个构建约束。与 `wire.go` 中的约束不同，这里的 `!wireinject` 多了一个 `!`，`!` 在编程中通常是取反的意思，所以它用来告诉 `wire` 忽略此文件。因为这个文件是最终执行的代码，`wire` 并不需要知道此文件的存在。

最后一个注释 `// +build !wireinject` 其实还是一个构建约束，只不过这是旧版本的条件编译标记（在 Go 1.17 版本之前使用）。它的作用与 `//go:build !wireinject` 相同，确保向后兼容性。这意味着在较老的 Go 版本中，编译条件也能被正确处理。

### 为什么要使用依赖注入工具?

前文讲解了依赖注入思想，以及通过[快速开始](http://cxyroad.com/”#Wire-%E5%BF%AB%E9%80%9F%E5%BC%80%E5%A7%8B”)的示例程序，我们极速入门了 wire 依赖注入工具的使用。

不过直到到现在我们都还没有讨论过为什么要使用依赖注入工具?

其实通过前文的示例，我们应该已经体会到，wire 最大的作用就是解放双手，提高生产力。

示例程序中，依赖链只有 3 个对象，一个中大型项目，依赖对象可能有几十个，wire 的作用会愈加明显。

使用依赖注入思想可以有效的解耦代码，那么使用依赖注入工具则进一步提高了生产力。我们无需手动实例化所有的依赖对象，仅需要编写函数声明，将依赖项扔给 `wire.Build`，`wire` 命令就能自动生成代码，可见 wire 是我们偷懒的利器，毕竟懒才是程序员的第一驱动力 :)。

如果你对依赖注入工具的作用还存在质疑，请接着往下看！

### Wire 核心概念

我们已经通过[快速开始](http://cxyroad.com/ "#Wire-%E5%BF%AB%E9%80%9F%E5%BC%80%E5%A7%8B")示例演示了 wire 的核心能力，现在是时候正式介绍下 wire 中的概念了。

在 wire 中，有两个核心概念：`providers`（提供者）和 `injectors`（注入器）。

这两个概念也很好理解，前文中的 `NewEvent`、`NewGreeter`、`NewMessage` 都是一个 `provider`。简单一句话：`provider` 就是一个可以**产生值的函数**，这些函数都是**普通的 Go 函数**。

值得注意的是，`provider` 必须是可导出的函数，即函数名称首字母大写。

而 `InitializeEvent` 实际上就是一个 `injector`。`injector` 是一个按依赖顺序调用 `provider` 的函数，该函数声明的主体是对 `wire.Build` 的调用。使用 wire 时，我们仅需编写 `injector` 的签名，然后由 `wire` 命令生成函数体。

### Wire 高级用法

wire 还有很多高级用法，值得介绍一下。

#### injector 函数参数和返回错误

首先是 `injector` 函数支持传参和返回 `error`。

修改示例程序代码如下：

```
type Message string

// 接收参数作为消息内容
func NewMessage(phrase string) Message {
return Message(phrase)
}

type Greeter struct {
Message Message
}

func NewGreeter(m Message) Greeter {
return Greeter{Message: m}
}

func (g Greeter) Greet() Message {
return g.Message
}

type Event struct {
Greeter Greeter
}

// 增加返回错误信息
func NewEvent(g Greeter) (Event, error) {
// 模拟创建 Event 报错
if time.Now().Unix()%2 == 0 {
return Event{}, errors.New("new event error")
}
return Event{Greeter: g}, nil
}

func (e Event) Start() {
msg := e.Greeter.Greet()
fmt.Println(msg)
}
```

这里主要修改了两处代码，`NewMessage` 接收一个字符串类型的参数作为消息内容，在 `NewEvent` 内部模拟了创建 `Event` 出错的场景，并将错误返回。

现在 `InitializeEvent` 函数定义如下：

```
func InitializeEvent(phrase string) (Event, error) {
wire.Build(NewEvent, NewMessage, NewGreeter)
return Event{}, nil
}
```

这次传给 `wire.Build` 的 3 个构造函数顺序不同，可见顺序并不重要。但为了代码可维护性，我建议还是要按照依赖顺序依次传入 `provider`。

这里返回值增加了 `error`，所以 `return` 的值增加了一个 `nil`，其实我们返回什么并不重要，只要返回的类型正确即可（确保编译通过），因为最终生成的代码返回值是由 `wire` 生成的。

> NOTE: 为了逻辑清晰，我只贴出核心代码，并且 `wire.go` 文件也不再贴出 `//go:build wireinject` 相关代码，后文也是如此，你在实践时不要忘记。完整代码详见文末给出的 [GitHub 地址](http://cxyroad.com/”https://github.com/jianghushinian/blog-go-example/tree/main/wire”)。

使用 `wire` 生成代码如下：

```
func InitializeEvent(phrase string) (Event, error) {
message := NewMessage(phrase)
greeter := NewGreeter(message)
event, err := NewEvent(greeter)
if err != nil {
return Event{}, err
}
return event, nil
}
```

```

现在我们执行示例代码，可能出现两种情况：

```
$ go run .
Hello World!
```

或者：

```
$ go run .
new event error
```

#### 使用 ProviderSet 进行分组

wire 为我们提供了 `provider sets`，顾名思义，它可以包含一组 `providers`。使用 `wire.NewSet` 函数可以将多个 `provider` 添加到一个集合中。

我们把 `NewMessage`、`NewGreeter` 两个构造函数合并成一个 `provider sets`：

```
var providerSet wire.ProviderSet = wire.NewSet(NewMessage, NewGreeter)
```

`wire.NewSet` 接收不定长参数，并将它们组装成一个 `wire.ProviderSet` 类型返回。

`wire.Build` 可以直接接收 `wire.ProviderSet` 类型，现在我们只需要给它传递两个参数即可：
```

```
func InitializeEvent(phrase string) (Event, error) {
wire.Build(NewEvent, providerSet)
return Event{}, nil
}
```

使用 `wire` 生成代码如下：

```
func InitializeEvent(phrase string) (Event, error) {
message := NewMessage(phrase)
greeter := NewGreeter(message)
event, err := NewEvent(greeter)
if err != nil {
return Event{}, err
}
return event, nil
}
```

与之前生成的代码一模一样。

分组后，代码会更加清晰，每个 `provider sets` 仅包含一组关联的 `providers`，下文中实践部分你还能够看到 `provider sets` 更具有意义的用法。

#### 使用 Struct 定制 Provider

有时候一个 `struct` 比较简单，我们通常不会为其定义一个构造函数，此时我们可以使用 `wire.Struct` 作为 `provider`。

为了演示此功能，我们修改 `Message` 定义如下：

```
```

```
type Message struct {
Content string
Code    int
}
```

修改 `InitializeEvent` 代码如下：

```
func InitializeEvent(phrase string, code int) (Event, error) {
wire.Build(NewEvent, NewGreeter, wire.Struct(new(Message), "Content"))
return Event{}, nil
}
```

这里使用 `wire.Struct(new(Message), "Content")` 替代了原来的 `NewMessage` 作为一个 `provider`。

`wire.Struct` 函数签名如下：

```
func Struct(structType interface{}, fieldNames ...string) StructProvider
```

`structType` 就是我们要使用的 `struct`，`fieldNames` 用来控制哪些字段会被赋值。

正常来说 `InitializeEvent` 的 `phrase` 参数会传给 `Message` 的 `Content` 字段，`code` 参数会传给 `Code`。wire 根据**参数类型**来判断应该将参数传给谁。

由于我们在这里仅显式指定了 `Content` 字段，所以最终只有 `Content` 字段会被赋值。

使用 `wire` 生成代码如下：

```
func InitializeEvent(phrase string, code int) (Event, error) {
message := Message{
Content: phrase,
}
greeter := NewGreeter(message)
event, err := NewEvent(greeter)
if err != nil {
return Event{}, err
}
return event, nil
}
```

从生成的代码可以发现，确实没给 `Code` 赋值。

如果我们想给 `Code` 赋值，可以这样写：

```
wire.Build(NewEvent, NewGreeter, wire.Struct(new(Message), ”Content”, ”Code”))
```

也可以这样写：

```
wire.Build(NewEvent, NewGreeter, wire.Struct(new(Message), ”*”))
```

`*` 表示通配符，即使用 `Message` 所有字段。

使用 `wire.Struct` 的好处是少定义一个构造函数，并且可以定制使用字段。

#### 使用 Struct 字段作为 Provider

我们还可以指定 `struct` 的具体某个字段作为一个 `provider`。

这需要用到 `wire.FieldsOf`，函数签名如下：

```
func FieldsOf(structType interface{}, fieldNames ...string) StructFields
```

可以发现它跟 `wire.Struct` 函数参数一样，区别是返回值不同。

现在示例代码如下：

```
type Content string

type Message struct {
Content Content
Code    int
}

// NewMessage 注意，这里返回的是指针类型
func NewMessage(content string, code int) *Message {
return &Message{
Content: Content(content),
Code:    code,
}
}
```

`Message` 的 `Content` 字段修改为 `string` 的别名类型 `Content`，**这是有用意的，稍后讲解**。

为了演示更多种情况，这里采用日常开发中更加常用的场景，即构造函数返回 `struct` 的指针类型。

修改 `InitializeEvent` 代码如下：

```
func InitializeMessage(phrase string, code int) Content {
wire.Build(NewMessage, wire.FieldsOf(new(*Message), "Content"))
return Content("")
}
```

因为示例代码中 `NewMessage` 返回 `*Message` 类型，而非 `Message` 类型，所以传递给 `wire.FieldsOf` 必须是 `new(*Message)` 而不是 `new(Message)`。

`InitializeMessage` 函数返回 `Content` 类型，而非 `Message` 类型。

使用 `wire` 生成代码如下：

```
func InitializeMessage(phrase string, code int) Content {
message := NewMessage(phrase, code)
content := message.Content
return content
}
```

根据生成的代码可以发现，在通过 `NewMessage` 函数创建 `*Message` 对象以后，会将 `message.Content` 字段提取出来并返回。

前文在讲解 [使用 Struct 定制 Provider](http://cxyroad.com/"#%E4%BD%BF%E7%94%A8-Struct-%E5%AD%97%E6%AE%B5%E4%BD%9C%E4%B8%BA-Provider")时我提到过「wire 根据**参数类型**来判断应该将参数传给谁」。

其实不仅仅是参数，wire 规定 `injector` 函数的参数和返回值类型都必须唯一。不然 wire 无法对应上哪个值该给谁，这也是为什么我专门定义了 `Content` 类型作为 `Message` 的字段，因为 `InitializeMessage` 的参数 `phrase` 已经是 `string` 类型了，所以其返回值就不能是 `string` 类型了。

现在就来演示一下 `injector` 函数的参数和返回值类型出现重复的情况，我们可以尝试把 `Message` 改回去：

```
type Message struct {
Content string
Code    int
}

// NewMessage 注意，这里返回的是指针类型
func NewMessage(content string, code int) *Message {
return &Message{
Content: content,
Code:    code,
}
}

```

`InitializeEvent` 函数返回值也改为 `string`：

```
func InitializeMessage(phrase string, code int) string {
wire.Build(NewMessage, wire.FieldsOf(new(*Message), "Content"))
return ""
}

```

现在使用 `wire` 生成代码，会得到类似如下错误：

```
$ wire gen .
wire: wire.go:10:2: multiple bindings for string
        current:
        <- wire.FieldsOf (structfields.go:8:2)
        previous:
        <- argument phrase to injector function InitializeMessage
(wire.go:7:1)
wire: github.com/jianghushinian/blog-go-example/wire/getting-
started/advanced/structfields: generate failed
wire: at least one generate failure

```

> NOTE: 注意，这里为了展示清晰，我将输出的文件绝对路径进行了修改，去掉了路径部分，只保留了文件名，不影响输出语义。后文中可能也会如此。

根据错误信息 `multiple bindings for string` 可知，wire 不支持函数的参数和返回值类型出现重复。

所以，当遇到 `injector` 函数出现参数或返回值类型重复的情况，可以通过给类型定义别名来解决。

#### 绑定「值」作为 Provider

可以直接将一个**值**作为参数传给 `wire.Value` 来构造一个 `provider`。

定义 `Message`：

```
type Message struct {
Message string
Code    int
}
```

我们可以直接实例化这个 `struct`，然后将其传给 `wire.Value`：

```
func InitializeMessage() Message {
// 假设没有提供 NewMessage，可以直接绑定值并返回
wire.Build(wire.Value(Message{
Message: ”Binding Values”,
Code:    1,
}))
return Message{}
}
```

使用 `wire` 生成代码如下：

```
func InitializeMessage() Message {
message := _wireMessageValue
return message
}

var (
_wireMessageValue = Message{
Message: ”Binding Values”,
Code:    1,
}
)
```

可以发现，实际上 wire 为我们定义了一个变量，并将这个变量作为 `InitializeMessage` 函数返回值。

这种拿来即用的方式，提供了非常大的便利。

`wire.Value` 接收任何**值类型**，所以不止 `struct`，一个普通的 `int`、`string` 等类型都可以，就交给你自己去尝试了。

#### 绑定「接口」作为 Provider

`provider` 依赖项或返回值并不总是**值**，很多时候是一个**接口**。

我们可以接将一个**接口**作为参数传给 `wire.InterfaceValue` 来构造一个 `provider`。

创建一个 `Write` 函数，它依赖一个 `io.Writer` 接口：

```
func Write(w io.Writer, value any) {
n, err := fmt.Fprintln(w, value)
fmt.Printf(”n: %d, err: %v\n”, n, err)
}
```

```

同 `wire.Value` 用法类似，我们可以使用 `wire.InterfaceValue` 绑定接口：

```
func InitializeWriter() io.Writer {
wire.Build(wire.InterfaceValue(new(io.Writer), os.Stdout))
return nil
}
```

使用 `wire` 生成代码如下：

```
func InitializeWriter() io.Writer {
writer := _wireFileValue
return writer
}

var (
_wireFileValue = os.Stdout
)
```

生成代码套路跟 `wire.Value` 没什么区别。

#### 绑定结构体到接口

有时候我们可能会写出如下代码：

```
type Message struct {
Content string
Code    int
}

type Store interface {
```
```

```
Save(msg *Message) error
}

type store struct{}

// 确保 store 实现了 Store 接口
var _ Store = (*store)(nil)

func New() *store {
return &store{}
}

func (s *store) Save(msg *Message) error {
return nil
}

func SaveMessage(s Store, msg *Message) error {
fmt.Printf("save message: %+v\n", msg)
return s.Save(msg)
}

func RunStore(msg *Message) error {
s := New()
return SaveMessage(s, msg)
}
```

`Store` 接口定义了一个 `Save` 方法用来保存 `Message`，定义了 `store` 结构体，结构体的指针 `*store` 实现了 `Store` 接口，所以 `store` 的构造函数 `New` 返回 `*store`。

我们还定义了 `SaveMessage` 方法，它接收两个参数，分别是 `Store` 接口以及 `*Message`。

最终定义的 `RunStore` 方法接收 `*Message`，并在内部创建 `*store`，然后将这两个变量传给 `SaveMessage` 保存消息。

假如我们想使用 `wire` 命令来生成 `RunStore` 函数，定义如下：

```
func WireRunStore(msg *Message) error {
wire.Build(SaveMessage, New)
```

```
return nil
}
```

使用 `wire` 生成代码将得到报错：

```
$ wire gen .
wire: wire.go:7:1: inject WireRunStore: no provider found for github.com/jianghushinian/blog-go-example/wire/getting-started/advanced/bindingstruct.Store
        needed by error in provider "SaveMessage" (bindingstruct.go:29:6)
wire: github.com/jianghushinian/blog-go-example/wire/getting-started/advanced/bindingstruct: generate failed
wire: at least one generate failure
```

这是因为 **wire 的构建依靠参数类型，但不支持接口类型**。而 `SaveMessage` 的 `s Store` 参数就是接口。

此时，我们可以使用 `wire.Bind` 告诉 `wire` 工具，将一个结构体绑定到接口：

```
func WireRunStore(msg *Message) error {
// new(Store) 接口无需使用指针
wire.Build(SaveMessage, New, wire.Bind(new(Store), new(*store)))
return nil
}
```

这样，`wire` 就知道 `New` 创建得到的 `*store` 类型需要传递给 `SaveMessage` 的 `s Store` 参数了。

使用 `wire` 生成代码如下：

```
```

```
func WireRunStore(msg *Message) error {
bindingstructStore := New()
error2 := SaveMessage(bindingstructStore, msg)
return error2
}
```

没有问题。

#### 清理函数

有时候我们的函数返回值可能包含一个清理函数，用来释放资源，示例代码如下：

```
func OpenFile(path string) (*os.File, func(), error) {
f, err := os.Open(path)
if err != nil {
return nil, nil, err
}

cleanup := func() {
fmt.Println("cleanup...")
if err := f.Close(); err != nil {
fmt.Println(err)
}
}

return f, cleanup, nil
}

func ReadFile(f *os.File) (string, error) {
b := make([]byte, 1024)
_, err := f.Read(b)
if err != nil {
return "", err
}
return string(b), nil
}
```

`OpenFile` 函数接收一个文件路径作为参数，其内部会打开这个文件，并返回

文件对象 `*os.File`。除此以外，还会返回一个清理函数和 `error`，清理函数内部会调用 `f.Close()` 关闭文件对象。

`ReadFile` 函数依赖 `*os.File` 文件对象，可以读取并返回其内容。

我们可以定义如下 `injector` 函数：

```
func InitializeFile(path string) (*os.File, func(), error) {
wire.Build(OpenFile)
return nil, nil, nil
}
```

使用 `wire` 生成代码如下：

```
func InitializeFile(path string) (*os.File, func(), error) {
file, cleanup, err := OpenFile(path)
if err != nil {
return nil, nil, err
}
return file, func() {
cleanup()
}, nil
}
```

可以发现，wire 能够正确处理这种情况。

不过，wire 规定清理函数签名只能为 `func()`。而 `InitializeFile` 函数的返回值，也是我们工作中使用 wire 的典型场景：`injector` 函数返回 3 个值，分别是对象、清理函数以及 `error`。

示例代码使用方式如下：

```
f, cleanup, err := InitializeFile("testdata/multi.txt")
if err != nil {
fmt.Println(err)
}
content, err := ReadFile(f)
if err != nil {
fmt.Println(err)
}
fmt.Println(content)
cleanup()
```

还有一种情况，假如我们传递给的 `wire.Build` 多个 `provider` 都存在清理函数，这时候 `wire` 命名生成的代码会是什么样呢?

这个就当做作业留给你自己去尝试了。

> NOTE: 如果你懒得尝试，其实我也写好了例子，你可以点击 [GitHub 地址](http://cxyroad.com/ "https://github.com/jianghushinian/blog-go-example/tree/main/wire/getting-started/advanced/cleanupfunctions/multi")进行查看。篇幅所限，我就不贴代码了，感兴趣可以点进去查看。

#### 备用注入器语法，给语法加点糖

前文讲过，`injector` 函数返回值并不重要，只要我们写在 `return` 后面的返回值类型，跟函数签名一致即可。因为 wire 会忽略它们，所以上面很多示例返回值我都使用 `nil` 来替代。

那么，既然返回值没什么用，我们是否可以偷个懒，不写 `return` 呢?

答案是可以的，我们可以直接 `panic`，这样程序依然可以通过编译。

示例代码如下:

```
```

```
type Message string

func NewMessage(phrase string) Message {
return Message(phrase)
}
```

这里直接在 `injector` 函数中使用 `panic` 来简化代码：

```
func InitializeMessage(phrase string) Message {
panic(wire.Build(NewMessage))
}
```

使用 `wire` 生成代码如下：

```
func InitializeMessage(phrase string) Message {
message := NewMessage(phrase)
return message
}
```

没有任何问题。

这种方式少写了一行 `return`，算是 wire 给我们提供的一个“语法糖”。Kratos 框架[文档](http://cxyroad.com/ "https://go-kratos.dev/docs/guide/wire/")中也是这么写的，你可以点击查看。

至于到底选用 `return` 还是 `panic`，社区并没有一致的规范，看个人喜好就好。我目前更喜欢使用 `return`，毕竟谁都不希望自己程序出现 `panic`，占位也不行 :)。

至此，终于将 wire 的常用功能全部讲解完毕。

下篇就进入 wire 生产实践讲解了，敬请期待！

**延伸阅读**

* Compile-time Dependency Injection With Go Cloud's Wire: [go.dev/blog/wire](http://cxyroad.com/ "https://go.dev/blog/wire")
* Wire README: [github.com/google/wire...](http://cxyroad.com/ "https://github.com/google/wire/blob/main/README.md")
* Wire Documentation: [pkg.go.dev/github.com/...](http://cxyroad.com/ "https://pkg.go.dev/github.com/google/wire/internal/wire")
* Wire 源码: [github.com/google/wire](http://cxyroad.com/ "https://github.com/google/wire")
* onex usercenter: [github.com/superproj/o...](http://cxyroad.com/ "https://github.com/superproj/onex/tree/master/internal/usercenter")
* Go Dependency Injection with Wire: [blog.drewolson.org/go-dependen...](http://cxyroad.com/ "https://blog.drewolson.org/go-dependency-injection-with-wire/")
* Golang Dependency Injection Using Wire: [clavinjune.dev/en/blogs/go...](http://cxyroad.com/ "https://clavinjune.dev/en/blogs/golang-dependency-injection-using-wire/")
* Dependency Injection in Go using Wire: [www.mohitkhare.com/blog/go-dep...](http://cxyroad.com/ "https://www.mohitkhare.com/blog/go-dependency-injection/")
* Wire 依赖注入: [go-kratos.dev/docs/guide/...](http://cxyroad.com/ "https://go-kratos.dev/docs/guide/wire/")
* Dependency Injection with Dig: [www.jetbrains.com/guide/go/tu...](http://cxyroad.com/ "https://www.jetbrains.com/guide/go/tutorials/dependency_injection_part_one/di_with_dig/")
* inject Documentation: [pkg.go.dev/github.com/...](http://cxyroad.com/ "https://pkg.go.dev/github.com/facebookgo/inject")
* Build Constraints: [pkg.go.dev/go/build#hd...](http://cxyroad.com/ "https://pkg.go.dev/go/build#hdr-Build_Constraints")
* 控制反转: [zh.wikipedia.org/wiki/控制反转](http://cxyroad.com/ "https://zh.wikipedia.org/wiki/%E6%8E%A7%E5%88%B6%E5%8F%8D%E8%BD%AC")
* 依赖注入: [zh.wikipedia.org/wiki/依赖注入](http://cxyroad.com/ "https://zh.wikipedia.org/wiki/%E4%BE%9D%E8%B5%96%E6%B3%A8%E5%85%A5")
* SOLID (面向对象设计): [zh.wikipedia.org/wiki/SOLID_...](http://cxyroad.com/ "https://zh.wikipedia.org/wiki/SOLID_(%E9%9D%A2%E5%90%91%E5%AF%B9%E8%B1%A1%E8%AE%BE%E8%AE%A1)")
* 设计模式之美 —— 19 | 理论五：控制反转、依赖反转、依赖注入，这三者有何区别和联系？: [time.geekbang.org/column/arti...](http://cxyroad.com/ "https://time.geekbang.org/column/article/177444")

* 本文 GitHub 示例代码：[github.com/jianghushin…](http://cxyroad.com/”https://github.com/jianghushinian/blog-go-example/tree/main/wire”)

**联系我**

* 公众号：[Go编程世界](http://cxyroad.com/”https://jianghushinian.cn/about”)
* ：jianghushinian
* 邮箱：[jianghushinian007@outlook.com](http://cxyroad.com/”mailto:jianghushinian007@outlook.com”)
* 博客：[jianghushinian.cn](http://cxyroad.com/”https://jianghushinian.cn”)
原文链接: https://juejin.cn/post/7379897208533221430