



下载APP



## 10 | 设计方法：怎么写出优雅的 Go 项目？

2021-06-15 孔令飞

《Go 语言项目开发实战》

课程介绍 >



讲述：孔令飞

时长 30:37 大小 28.04M



你好，我是孔令飞，今天我们来聊聊如何写出优雅的 Go 项目。

Go 语言简单易学，对于大部分开发者来说，编写可运行的代码并不是一件难事，但如果想真正成为 Go 编程高手，你需要花很多精力去研究 Go 的编程哲学。

在我的 Go 开发生涯中，我见过各种各样的代码问题，例如：代码不规范，难以阅读；函数共享性差，代码重复率高；不是面向接口编程，代码扩展性差，代码不可测；代码质量低下。究其原因，是因为这些代码的开发者**很少花时间去认真研究如何开发一个优雅的 Go 项目，更多时间是埋头在需求开发中。** ☆

如果你也遇到过以上问题，那么是时候花点时间来研究下如何开发一个优雅的 Go 项目了。只有这样，你才能区别于绝大部分的 Go 开发者，从而在职场上建立自己的核心竞争

力，并最终脱颖而出。

其实，我们之前所学的各种规范设计，也都是为了写出一个优雅的 Go 项目。在这一讲，我又补充了一些内容，从而形成了一套“写出优雅 Go 项目”的方法论。这一讲内容比较多，但很重要，希望你能花点精力认真掌握，掌握之后，能够确保你开发出一个优秀的 Go 项目。

## 如何写出优雅的 Go 项目？

那么，如何写出一个**优雅的 Go 项目**呢？在回答这个问题之前，我们先来看另外两个问题：

1. 为什么是 Go 项目，而不是 Go 应用？
2. 一个优雅的 Go 项目具有哪些特点？

先来看第一个问题。Go 项目是一个偏工程化的概念，不仅包含了 Go 应用，还包含了项目管理和项目文档：

这就来到了第二个问题，一个优雅的 Go 项目，不仅要求我们的 Go 应用是优雅的，还要确保我们的项目管理和文档也是优雅的。这样，我们根据前面几讲学到的 Go 设计规范，很容易就能总结出一个优雅的 Go 应该具备的特点：

符合 Go 编码规范和最佳实践；

易阅读、易理解，易维护；

易测试、易扩展；

代码质量高。

解决了这两个问题，让我们回到这一讲的核心问题：如何写出优雅的 Go 项目？

写出一个优雅的 Go 项目，在我看来，就是用“**最佳实践**”的方式去实现 Go 项目中的**Go 应用、项目管理和项目文档**。具体来说，就是编写高质量的 Go 应用、高效管理项目、编写高质量的项目文档。

为了协助你理解，我将这些逻辑绘制成了下面一张图。

接下来，我们就看看如何根据前面几讲学习的 Go 项目设计规范，实现一个优雅的 Go 项目。我们先从编写高质量的 Go 应用看起。

## 编写高质量的 Go 应用

基于我的研发经验，要编写一个高质量的 Go 应用，其实可以归纳为 5 个方面：代码结构、代码规范、代码质量、编程哲学和软件设计方法，见下图。

接下来，我们详细说说这些内容。

### 代码结构

为什么先说代码结构呢？因为组织合理的代码结构是一个项目的门面。我们可以通过两个手段来组织代码结构。

第一个手段是，组织一个好的目录结构。关于如何组合一个好的目录结构，你可以回顾 [🔗 06 讲](#) 的内容。

第二个手段是，选择一个好的模块拆分方法。做好模块拆分，可以使项目内模块职责分明，做到低耦合高内聚。

那么 Go 项目开发中，如何拆分模块呢？目前业界有两种拆分方法，分别是按层拆分和按功能拆分。

**首先，我们看下按层拆分**，最典型的是 MVC 架构中的模块拆分方式。在 MVC 架构中，我们将服务中的不同组件按访问顺序，拆分成了 Model、View 和 Controller 三层。

每层完成不同的功能：

View（视图）是提供给用户的操作界面，用来处理数据的显示。

Controller ( 控制器 ) , 负责根据用户从 View 层输入的指令 , 选取 Model 层中的数据 , 然后对其进行相应的操作 , 产生最终结果。

Model ( 模型 ) , 是应用程序中用于处理数据逻辑的部分。

我们看一个典型的按层拆分的目录结构 :

[复制代码](#)

```
1 $ tree --noreport -L 2 layers
2 layers
3 |— controllers
4 |   |— billing
5 |   |— order
6 |   |— user
7 |— models
8 |   |— billing.go
9 |   |— order.go
10 |   |— user.go
11 |— views
12 |— layouts
```

在 Go 项目中 , 按层拆分会带来很多问题。最大的问题是循环引用 : 相同功能可能在不同层被使用到 , 而这些功能又分散在不同的层中 , 很容易造成循环引用。

所以 , **你只要大概知道按层拆分是什么意思就够了 , 在 Go 项目中我建议你使用的是按功能拆分的方法 , 这也是 Go 项目中最常见的拆分方法。**

那什么是按功能拆分呢 ? 我给你看一个例子你就明白了。比如 , 一个订单系统 , 我们可以根据不同功能将其拆分成用户 ( user ) 、 订单 ( order ) 和计费 ( billing ) 3 个模块 , 每一个模块提供独立的功能 , 功能更单一 :

下面是该订单系统的代码目录结构 :

[复制代码](#)

```
1 $ tree pkg
2 $ tree --noreport -L 2 pkg
3 pkg
4 |— billing
5 |— order
```

```
6 |   └─ order.go
7 |   └─ user
```

相较于按层拆分，按功能拆分模块带来的好处也很好理解：

不同模块，功能单一，可以实现高内聚低耦合的设计哲学。

因为所有的功能只需要实现一次，引用逻辑清晰，会大大减少出现循环引用的概率。

所以，有很多优秀的 Go 项目采用的都是按功能拆分的模块拆分方式，例如 Kubernetes、Docker、Helm、Prometheus 等。

除了组织合理的代码结构这种方式外，编写高质量 Go 应用的另外一个行之有效的方法，是遵循 Go 语言代码规范来编写代码。在我看来，这也是最容易出效果的方式。

## 代码规范

那我们要遵循哪些代码规范来编写 Go 应用呢？在我看来，其实就两类：编码规范和最佳实践。

**首先，我们的代码要符合 Go 编码规范，这是最容易实现的途径。** Go 社区有很多这类规范可供参考，其中，比较受欢迎的是 [🔗 Uber Go 语言编码规范](#)。

阅读这些规范确实有用，也确实花时间、花精力。所以，我在参考了已有的很多规范后，结合自己写 Go 代码的经验，特地为你整理了一篇 Go 编码规范作为加餐，也就是“特别放送 | 给你一份清晰、可直接套用的 Go 编码规范”。

有了可以参考的编码规范之后，我们需要扩展到团队、部门甚至公司层面。只有大家一起参与、遵守，规范才会变得有意义。其实，我们都清楚，要开发者靠自觉来遵守所有的编码规范，不是一件容易的事儿。这时候，我们可以使用静态代码检查工具，来约束开发者的行为。

有了静态代码检查工具后，不仅可以确保开发者写出的每一行代码都是符合 Go 编码规范的，还可以将静态代码检查集成到 CI/CD 流程中。这样，在代码提交后自动地检查代码，就保证了只有符合编码规范的代码，才会被合入主干。

Go 语言的静态代码检查工具有很多，目前用的最多的是 [golangci-lint](#)，这也是我极力推荐你使用的一个工具。关于这个工具的使用，我会在**第 15 讲**和你详细介绍。

除了遵循编码规范，**要想成为 Go 编程高手，你还得学习并遵循一些最佳实践**。“最佳实践”是社区经过多年探索沉淀下来的、符合 Go 语言特色的经验和共识，它可以帮助你开发出一个高质量的代码。

这里我给你推荐几篇介绍 Go 语言最佳实践的文章，供你参考：

🔗 [Effective Go](#)：高效 Go 编程，由 Golang 官方编写，里面包含了编写 Go 代码的一些建议，也可以理解为最佳实践。

🔗 [Go Code Review Comments](#)：Golang 官方编写的 Go 最佳实践，作为 Effective Go 的补充。

🔗 [Style guideline for Go packages](#)：包含了如何组织 Go 包、如何命名 Go 包、如何写 Go 包文档的一些建议。

## 代码质量

有了组织合理的代码结构、符合 Go 语言代码规范的 Go 应用代码之后，我们还需要通过一些手段来确保我们开发出的是一个高质量的代码，这可以通过单元测试和 Code Review 来实现。

**单元测试非常重要**。我们开发完一段代码后，第一个执行的测试就是单元测试。它可以保证我们的代码是符合预期的，一些异常变动能够被及时感知到。进行单元测试，不仅需要编写单元测试用例，还需要我们确保代码是可测试的，以及具有一个高的单元测试覆盖率。

接下来，我就来介绍下如何编写一个可测试的代码。

如果我们要对函数 A 进行测试，并且 A 中的所有代码均能够在单元测试环境下按预期被执行，那么函数 A 的代码块就是可测试的。我们来看下一般的单元测试环境有什么特点：

可能无法连接数据库。

可能无法访问第三方服务。

如果函数 A 依赖数据库连接、第三方服务，那么在单元测试环境下执行单元测试就会失败，函数就没法测试，函数是不可测的。

解决方法也很简单：将依赖的数据库、第三方服务等抽象成接口，在被测代码中调用接口的方法，在测试时传入 mock 类型，从而将数据库、第三方服务等依赖从具体的被测函数中解耦出去。如下图所示：

为了提高代码的可测性，降低单元测试的复杂度，对 function 和 mock 的要求是：

要尽可能减少 function 中的依赖，让 function 只依赖必要的模块。编写一个功能单一、职责分明的函数，会有利于减少依赖。

依赖模块应该是易 Mock 的。

为了协助你理解，我们先来看一段不可测试的代码：


[复制代码](#)

```
1 package post
2
3 import "google.golang.org/grpc"
4
5 type Post struct {
6     Name    string
7     Address string
8 }
9
10 func ListPosts(client *grpc.ClientConn) ([]*Post, error) {
11     return client.ListPosts()
12 }
```

这段代码中的 ListPosts 函数是不可测试的。因为 ListPosts 函数中调用了 client.ListPosts() 方法，该方法依赖于一个 gRPC 连接。而我们在做单元测试时，可能因为没有配置 gRPC 服务的地址、网络隔离等原因，导致没法建立 gRPC 连接，从而导致 ListPosts 函数执行失败。


下面，我们把这段代码改成可测试的，如下：



 复制代码

```
1 package main
2
3 type Post struct {
4     Name    string
5     Address string
6 }
7
8 type Service interface {
9     ListPosts() ([]*Post, error)
10 }
11
12 func ListPosts(svc Service) ([]*Post, error) {
13     return svc.ListPosts()
14 }
```

上面代码中，ListPosts 函数入参为 Service 接口类型，只要我们传入一个实现了 Service 接口类型的实例，ListPosts 函数即可成功运行。因此，我们可以在单元测试中可以实现一个不依赖任何第三方服务的 fake 实例，并传给 ListPosts。上述可测代码的单元测试代码如下：

 复制代码

```
1 package main
2
3 import "testing"
4
5 type fakeService struct {
6 }
7
8 func NewFakeService() Service {
9     return &fakeService{}
10 }
11
12 func (s *fakeService) ListPosts() ([]*Post, error) {
13     posts := make([]*Post, 0)
14     posts = append(posts, &Post{
15         Name:    "colin",
16         Address: "Shenzhen",
17     })
18     posts = append(posts, &Post{
19         Name:    "alex",
20         Address: "Beijing",
21     })
22     return posts, nil
23 }
24
25 func TestListPosts(t *testing.T) {
```



```
26     fake := NewFakeService()
27     if _, err := ListPosts(fake); err != nil {
28         t.Fatal("list posts failed")
29     }
30 }
```

当我们的代码可测之后，就可以借助一些工具来 Mock 需要的接口了。常用的 Mock 工具，有这么几个：

🔗 **golang/mock**，是官方提供的 Mock 框架。它实现了基于 interface 的 Mock 功能，能够与 Golang 内置的 testing 包做很好的集成，是最常用的 Mock 工具。golang/mock 提供了 mockgen 工具用来生成 interface 对应的 Mock 源文件。

🔗 **sqlmock**，可以用来模拟数据库连接。数据库是项目中比较常见的依赖，在遇到数据库依赖时都可以用它。

🔗 **httpmock**，可以用来 Mock HTTP 请求。

🔗 **bouk/monkey**，猴子补丁，能够通过替换函数指针的方式来修改任意函数的实现。如果 golang/mock、sqlmock 和 httpmock 这几种方法都不能满足我们的需求，我们可以尝试通过猴子补丁的方式来 Mock 依赖。可以这么说，猴子补丁提供了单元测试 Mock 依赖的最终解决方案。

接下来，我们再一起看看**如何提高我们的单元测试覆盖率**。

当我们编写了可测试的代码之后，接下来就需要编写足够的测试用例，用来提高项目的单元测试覆盖率。这里我有以下两个建议供你参考：

使用 gotests 工具自动生成单元测试代码，减少编写单元测试用例的工作量，将你从重复的劳动中解放出来。

定期检查单元测试覆盖率。你可以通过以下方法来检查：

```
1 $ go test -race -cover -coverprofile=./coverage.out -timeout=10m -short -v ./
2 $ go tool cover -func ./coverage.out
```

📄 复制代码

执行结果如下：

在提高项目的单元测试覆盖率时，我们可以先提高单元测试覆盖率低的函数，之后再检查项目的单元测试覆盖率；如果项目的单元测试覆盖率仍然低于期望的值，可以再次提高单元测试覆盖率低的函数的覆盖率，然后再检查。以此循环，最终将项目的单元测试覆盖率优化到预期的值为止。

这里要注意，对于一些可能经常会变动的函数单元测试，覆盖率要达到 100%。

说完了单元测试，我们再看看**如何通过 Code Review 来保证代码质量**。

Code Review 可以提高代码质量、交叉排查缺陷，并且促进团队内知识共享，是保障代码质量非常有效的手段。在我们的项目开发中，一定要建立一套持久可行的 Code Review 机制。

但在我的研发生涯中，发现很多团队没有建立有效的 Code Review 机制。这些团队都认可 Code Review 机制带来的好处，但是因为流程难以遵守，慢慢地 Code Review 就变成了形式主义，最终不了了之。其实，建立 Code Review 机制很简单，主要有 3 点：

首先，确保我们使用的代码托管平台有 Code Review 的功能。比如，GitHub、GitLab 这类代码托管平台都具备这种能力。

接着，建立一套 Code Review 规范，规定如何进行 Code Review。

最后，也是最重要的，每次代码变更，相关开发人员都要去落实 Code Review 机制，并形成习惯，直到最后形成团队文化。

**到这里我们可以小结一下：组织一个合理的代码结构、编写符合 Go 代码规范的代码、保证代码质量，在我看来都是编写高质量 Go 代码的外功。那内功是什么呢？就是编程哲学和软件设计方法。**

## 编程哲学

那编程哲学是什么意思呢？在我看来，编程哲学，其实就是要编写符合 Go 语言设计哲学的代码。Go 语言有很多设计哲学，对代码质量影响比较大的，我认为有两个：面向接口编程和面向“对象”编程。

我们先来看下面向接口编程。

Go 接口是一组方法的集合。任何类型，只要实现了该接口中的方法集，那么就属于这个类型，也称为实现了该接口。

接口的作用，其实就是为不同层级的模块提供一个定义好的中间层。这样，上游不再需要依赖下游的具体实现，充分地对上下游进行了解耦。很多流行的 Go 设计模式，就是通过面向接口编程的思想来实现的。

我们看一个面向接口编程的例子。下面这段代码定义了一个Bird接口，Canary 和 Crow 类型均实现了Bird接口。

 复制代码

```
1 package main
2
3 import "fmt"
4
5 // 定义了一个鸟类
6 type Bird interface {
7     Fly()
8     Type() string
9 }
10
11 // 鸟类: 金丝雀
12 type Canary struct {
13     Name string
14 }
15
16 func (c *Canary) Fly() {
17     fmt.Printf("我是%s, 用黄色的翅膀飞\n", c.Name)
18 }
19 func (c *Canary) Type() string {
20     return c.Name
21 }
22
23 // 鸟类: 乌鸦
24 type Crow struct {
25     Name string
26 }
27
28 func (c *Crow) Fly() {
29     fmt.Printf("我是%s, 我用黑色的翅膀飞\n", c.Name)
30 }
31
32 func (c *Crow) Type() string {
33     return c.Name
34 }
35
```

```
36 // 让鸟类飞一下
37 func LetItFly(bird Bird) {
38     fmt.Printf("Let %s Fly!\n", bird.Type())
39     bird.Fly()
40 }
41
42 func main() {
43     LetItFly(&Canary{"金丝雀"})
44     LetItFly(&Crow{"乌鸦"})
45 }
```

这段代码中，因为 Crow 和 Canary 都实现了 Bird 接口声明的 Fly、Type 方法，所以可以说 Crow、Canary 实现了 Bird 接口，属于 Bird 类型。在函数调用时，可以传入 Bird 类型，并在函数内部调用 Bird 接口提供的方法，以此来解耦 Bird 的具体实现。

好了，我们总结下使用接口的好处吧：

代码扩展性更强了。例如，同样的 Bird，可以有不同的实现。在开发中用的更多的是，将数据库的 CURD 操作抽象成接口，从而可以实现同一份代码对接不同数据库的目的。

可以解耦上下游的实现。例如，LetItFly 不用关注 Bird 是如何 Fly 的，只需要调用 Bird 提供的方法即可。

提高了代码的可测性。因为接口可以解耦上下游实现，我们在单元测试需要依赖第三方系统 / 数据库的代码时，可以利用接口将具体实现解耦，实现 fake 类型。

代码更健壮、更稳定了。例如，如果要更改 Fly 的方式，只需要更改相关类型的 Fly 方法即可，完全影响不到 LetItFly 函数。

所以，我建议你，在 Go 项目开发中，一定要多思考，那些可能有多种实现的地方，要考虑使用接口。

接下来，我们再来看下面向“对象”编程。

面向对象编程（OOP）有很多优点，例如可以使我们的代码变得易维护、易扩展，并能提高开发效率等，所以一个高质量的 Go 应用需要时，也应该采用面向对象的方法去编程。那什么叫“在需要时”呢？就是我们在开发代码时，如果一个功能可以通过接近于日常生活和自然的思考方式来实现，这时候就应该考虑使用面向对象的编程方法。

Go 语言不支持面向对象编程，但是却可以通过一些语言级的特性来实现类似的效果。

面向对象编程中，有几个核心特性：类、实例、抽象，封装、继承、多态、构造函数、析构函数、方法重载、this 指针。在 Go 中可以通过以下几个方式来实现类似的效果：

类、抽象、封装通过结构体来实现。

实例通过结构体变量来实现。

继承通过组合来实现。这里解释下什么叫组合：一个结构体嵌到另一个结构体，称作组合。例如一个结构体包含了一个匿名结构体，就说这个结构体组合了该匿名结构体。

多态通过接口来实现。

至于构造函数、析构函数、方法重载和 this 指针等，Go 为了保持语言的简洁性去掉了这些特性。

Go 中面向对象编程方法，见下图：


我们通过一个示例，来具体看下 Go 是如何实现面向对象编程中的类、抽象、封装、继承和多态的。代码如下：

 复制代码

```
1 package main
2
3 import "fmt"
4
5 // 基类: Bird
6 type Bird struct {
7     Type string
8 }
9
10 // 鸟的类别
11 func (bird *Bird) Class() string {
12     return bird.Type
13 }
14
15 // 定义了一个鸟类
16 type Birds interface {
17     Name() string
18     Class() string
19 }
```

```
20
21 // 鸟类: 金丝雀
22 type Canary struct {
23     Bird
24     name string
25 }
26
27 func (c *Canary) Name() string {
28     return c.name
29 }
30
31 // 鸟类: 乌鸦
32 type Crow struct {
33     Bird
34     name string
35 }
36
37 func (c *Crow) Name() string {
38     return c.name
39 }
40
41 func NewCrow(name string) *Crow {
42     return &Crow{
43         Bird: Bird{
44             Type: "Crow",
45         },
46         name: name,
47     }
48 }
49
50 func NewCanary(name string) *Canary {
51     return &Canary{
52         Bird: Bird{
53             Type: "Canary",
54         },
55         name: name,
56     }
57 }
58
59 func BirdInfo(birds Birds) {
60     fmt.Printf("I'm %s, I belong to %s bird class!\n", birds.Name(), birds.Class)
61 }
62
63 func main() {
64     canary := NewCanary("CanaryA")
65     crow := NewCrow("CrowA")
66     BirdInfo(canary)
67     BirdInfo(crow)
68 }
```

将上述代码保存在 oop.go 文件中，执行以下代码输出如下：

 复制代码

```
1 $ go run oop.go
2 I'm CanaryA, I belong to Canary bird class!
3 I'm CrowA, I belong to Crow bird class!
```

在上面的例子中，分别通过 Canary 和 Crow 结构体定义了金丝雀和乌鸦两种类别的鸟，其中分别封装了 name 属性和 Name 方法。也就是说通过结构体实现了类，该类抽象了鸟类，并封装了该鸟类的属性和方法。

在 Canary 和 Crow 结构体中，都有一个 Bird 匿名字段，Bird 字段为 Canary 和 Crow 类的父类，Canary 和 Crow 继承了 Bird 类的 Class 属性和方法。也就是说通过匿名字段实现了继承。

在 main 函数中，通过 NewCanary 创建了 Canary 鸟类实例，并将其传给 BirdInfo 函数。也就是说通过结构体变量实现实例。

在 BirdInfo 函数中，将 Birds 接口类型作为参数传入，并在函数中调用了 birds.Name，birds.Class 方法，这两个方法会根据 birds 类别的不同而返回不同的名字和类别，也就是说通过接口实现了多态。

## 软件设计方法

接下来，我们继续学习编写高质量 Go 代码的第二项内功，也就是让编写的代码遵循一些业界沉淀下来的，优秀的软件设计方法。

优秀的软件设计方法有很多，其中有两类方法对我们代码质量的提升特别有帮助，分别是设计模式 ( Design pattern ) 和 SOLID 原则。

在我看来，设计模式可以理解为业界针对一些特定的场景总结出来的最佳实现方式。它的特点是解决的场景比较具体，实施起来会比较简单；而 SOLID 原则更侧重设计原则，需要我们彻底理解，并在编写代码时多思考和落地。



关于设计模式和 SOLID 原则，我是这么安排的：在**第 11 讲**，我会带你学习 Go 项目常用的设计模式；至于 SOLID 原则，网上已经有很多高质量的文章了，所以我会简单告诉你这个原则是啥，然后给你推荐一篇介绍文章。

我们先了解下有哪些设计模式。

在软件领域，沉淀了一些比较优秀的设计模式，其中最受欢迎的是 GOF 设计模式。GOF 设计模式中包含了 3 大类（创建型模式、结构型模式、行为型模式），共 25 种经典的、可以解决常见软件设计问题的设计方案。这 25 种设计方案同样也适用于 Go 语言开发的项目。

这里，我将这 25 种设计模式总结成了一张图，你可以先看看，有个大概的印象，对于一些在 Go 项目开发中常用的设计模式，我会在**第 11 讲**详细介绍。

**如果说设计模式解决的是具体的场景，那么 SOLID 原则就是我们设计应用代码时的指导方针。**

SOLID 原则，是由罗伯特·C·马丁在 21 世纪早期引入的，包括了面向对象编程和面向对象设计的五个基本原则：

遵循 SOLID 原则可以确保我们设计的代码是易维护、易扩展、易阅读的。SOLID 原则同样也适用于 Go 程序设计。

如果你需要更详细地了解 SOLID 原则，可以参考下 [🔗 SOLID 原则介绍](#) 这篇文章。

到这里，我们就学完了“编写高质量的 Go 应用”这部分内容。接下来，我们再来学习下如何高效管理 Go 项目，以及如何编写高质量的项目文档。这里面的大部分内容，之前我们都有学习过，因为它们是“如何写出优雅的 Go 项目”的重要组成部分，所以，这里我仍然会简单介绍下它们。

## 高效管理项目

一个优雅的 Go 项目，还需要具备高效的项目管理特性。那么如何高效管理我们的项目呢？

不同团队、不同项目会采用不同的方法来管理项目，在我看来比较重要的有 3 点，分别是制定一个高效的开发流程、使用 Makefile 管理项目和将项目管理自动化。我们可以通过自动生成代码、借助工具、对接 CI/CD 系统等方法来将项目管理自动化。具体见下图：

## 高效的开发流程

高效管理项目的第一步，就是要有一个高效的开发流程，这可以提高开发效率、减少软件维护成本。你可以回想一下设计开发流程的知识，如果印象比较模糊了，一定要回去复习下 **08 讲**的内容，因为这部分很重要。

## 使用 Makefile 管理项目

为了更好地管理项目，除了一个高效的开发流程之外，使用 Makefile 也很重要。Makefile 可以将项目管理的工作通过 Makefile 依赖的方式实现自动化，除了可以提高管理效率之外，还能够减少人为操作带来的失误，并统一操作方式，使项目更加规范。

IAM 项目的所有操作均是通过 Makefile 来完成的，具体 Makefile 完成了如下操作：

 复制代码

```
1  build                Build source code for host platform.
2  build.multiarch      Build source code for multiple platforms. See option PLAT
3  image               Build docker images for host arch.
4  image.multiarch     Build docker images for multiple platforms. See option PL
5  push                Build docker images for host arch and push images to regi
6  push.multiarch      Build docker images for multiple platforms and push image
7  deploy              Deploy updated components to development env.
8  clean               Remove all files that are created by building.
9  lint                Check syntax and styling of go sources.
10 test                Run unit test.
11 cover               Run unit test and get test coverage.
12 release              Release iam
13 format               Gofmt (reformat) package sources (exclude vendor dir if e
14 verify-copyright     Verify the boilerplate headers for all files.
15 add-copyright        Ensures source code files have copyright license headers.
16 gen                  Generate all necessary files, such as error code files.
17 ca                   Generate CA files for all iam components.
18 install              Install iam system with all its components.
19 swagger              Generate swagger document.
```

```
20  serve-swagger      Serve swagger spec and docs.
21  dependencies      Install necessary dependencies.
22  tools              install dependent tools.
23  check-updates      Check outdated dependencies of the go projects.
24  help               Show this help info.
```

## 自动生成代码

低代码的理念现在越来越流行。虽然低代码有很多缺点，但确实有很多优点，例如：

自动化生成代码，减少工作量，提高工作效率。

代码有既定的生成规则，相比人工编写代码，准确性更高、更规范。

目前来看，自动生成代码现在已经成为趋势，比如 Kubernetes 项目有很多代码都是自动生成的。我认为，想写出一个优雅的 Go 项目，你也应该认真思考哪些地方的代码可以自动生成。在这门课的 IAM 项目中，就有大量的代码是自动生成的，我放在这里供你参考：

错误码、错误码说明文档。

自动生成缺失的 doc.go 文件。

利用 gotests 工具，自动生成单元测试用例。

使用 Swagger 工具，自动生成 Swagger 文档。

使用 Mock 工具，自动生成接口的 Mock 实例。

## 善于借助工具

在开发 Go 项目的过程中，我们也要善于借助工具，来帮助我们完成一部分工作。利用工具可以带来很多好处：


解放双手，提高工作效率。

利用工具的确定性，可以确保执行结果的一致性。例如，使用 golangci-lint 对代码进行检查，可以确保不同开发者开发的代码至少都遵循 golangci-lint 的代码检查规范。

有利于实现自动化，可以将工具集成到 CI/CD 流程中，触发流水线自动执行。

那么，Go 项目中，有哪些工具可以为我們所用呢？这里，我给你整理了一些有用的工具：

所有这些工具都可以通过下面的方式安装。

 复制代码

```
1 $ cd $IAM_ROOT
2 $ make tools.install
```

IAM 项目使用了上面这些工具的绝大部分，用来尽可能提高整个项目的自动化程度，提高项目维护效率。

## 对接 CI/CD

代码在合并入主干时，应该有一套 CI/CD 流程来自动化地对代码进行检查、编译、单元测试等，只有通过后的代码才可以并入主干。通过 CI/CD 流程来保证代码的质量。当前比较流行的 CI/CD 工具有 Jenkins、GitLab、Argo、Github Actions、JenkinsX 等。在**第 51 讲**和**第 52 讲**中，我会详细介绍 CI/CD 的原理和实战。

## 编写高质量的项目文档

最后，一个优雅的项目，还应该有良好的文档。例如 README.md、安装文档、开发文档、使用文档、API 接口文档、设计文档等等。这些内容在**第 04 讲**的文档规范部分有详细介绍，你可以去复习下。

## 总结

使用 Go 语言做项目开发，核心目的其实就是**开发一个优雅的 Go 项目**。那么如何开发一个优雅的 Go 项目呢？Go 项目包含三大内容，即 Go 应用、项目管理、项目文档，因此开发一个优雅的 Go 项目，其实就是**编写高质量的 Go 应用、高效管理项目**和**编写高质量的项目文档**。针对每一项，我都给出了一些实现方式，这些方式详见下图：

## 课后练习

1. 在工作中，你还有哪些方法，来帮助你开发一个优雅的 Go 项目呢？
2. 在你的当前项目中有哪些可以接口化的代码呢？找到它们，并尝试用面向接口的编程哲学去重写这部分代码吧。

期待在留言区看到你的思考和答案，我们下一讲见！

分享给需要的人，Ta 订阅后你可得 **24 元现金奖励**

👍 赞 3

💡 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 09 | 研发流程设计（下）：如何管理应用的生命周期？

下一篇 11 | 设计模式：Go 常用设计模式概述

## 更多课程推荐

# 容器实战高手课

在实战中深入理解容器技术的本质

李程远

eBay 总监级工程师  
云平台架构师



涨价倒计时 🕒

今日订阅 **¥69**，7月20日涨价至 **¥129**

## 精选留言 (15)

💬 写留言

pedro

2021-06-15

一节值回票价！！！！

展开 ▾



15

**daz2yy**

2021-06-16

很完善的内容！不过，道理其实都懂，或者多多少少看过一些，如果能结合一些实际落地的例子就更好了；也正如老师说的，要求人去按照规范做很难，需要靠工具实现约束；还有比如 CR 的具体操作细节，花费的时间，参与人，CR不通过与通过的情况等。

作者回复: 项目中会有实战那俩



4

**A**

2021-06-18

<https://readme.so/editor> readme模板

展开 ▾

作者回复: 好网站，感谢分享！



2

**狮盔银甲**

2021-07-15

关于项目架构 相比 老师介绍的 按层拆分(水平拆分)和按功能拆分(垂直拆分) 在 <<clean architecture>> 和 领域驱动设计 是不是都推荐六边形架构呢 在这块比较疑惑

展开 ▾

作者回复: 按功能拆分和六边形架构不冲突。看了下六边形架构介绍，很cool，IAM项目也有六边形架构的影子，老哥完全可以按六边形架构来设计系统。



1

1

**惟新**

2021-06-15

我来 催更了。

展开 ▾



1

**莫林**

2021-07-21

谢谢老师详细的讲解。关于目录结构有两点比较疑惑：

1. 按功能拆分的时候，如果两个功能模块需要相互调用怎么办？例如 User、Article 和 Comment.
  2. 每个功能模块里面是不是也要按照分层来：domain、repository、service?
  3. 对于公共组件，如 mysql 连接池。需要如何处理，在何时注入。起 server 的时候注...
- 展开 ∨

**timidsmile**

2021-06-29

代码结构这一小节，有点不太理解，求指点~~~

1. 目录结构和分层是二选一吗？还是先按照【目录结构】规范，然后在具体的目录下再按照【分层】来实现呢？
2. 分层建议按功能拆分，那么一个功能模块里还需要考虑分层吗？比如订单模块，里面会包含业务逻辑，db、api、redis调用，这些都放到一个模块里需要考虑分层吗？

展开 ∨

作者回复: 需要按功能拆分，而非分层



2021-06-22

想问问像 changelog 的生成是以为个 维度划分？如 日期？分支提交，还是以功能发布为准？

作者回复: 以版本发布为准

**demon**


2021-06-21

你好，请问单元测试文件应该放在哪个文件夹下？

展开 ∨

作者回复: 建议跟源码文件放一个目录中





**nio**  
2021-06-17

我也来催更了

展开

**helloworld**  
2021-06-17

赞

展开

**夏夜星语**  
2021-06-15

golangci-lint 安装失败，换版本为v1.14.0也不行：

```
(base) [cody@dev iam]$ go get -u github.com/golangci/golangci-lint/cmd/golangci-lint
```

go: downloading github.com/tommy-muehle/go-mnd v1.3.1-0.20200224220436...

展开

作者回复: 直接官网下载二进制文件吧，最新版可能有问题

**不明真相的群众**  
2021-06-15

我来催更了

展开

**旋风**  
2021-06-15

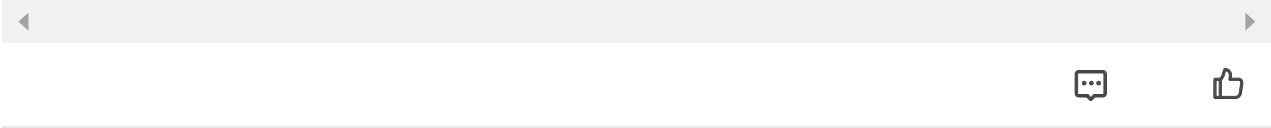
在mac下安装工具，有两个安装出错了：

1. golangci-lint

出错信息：

github.com/golangci/golangci-lint@v1.40.1/pkg/golinters/gosimple.go:11:40: can not use simple.Analyzers (type []\*"honnef.co/go/tools/analysis/lint".Analyzer) as ...  
展开

作者回复: 最新版本，可能有问题，可以直接下载github上编译好的二进制文件。或者clone下来自己编译



**YangYi乐1990**  
2021-06-15

期待下一讲 ~ 🤔🤔🤔  
展开

