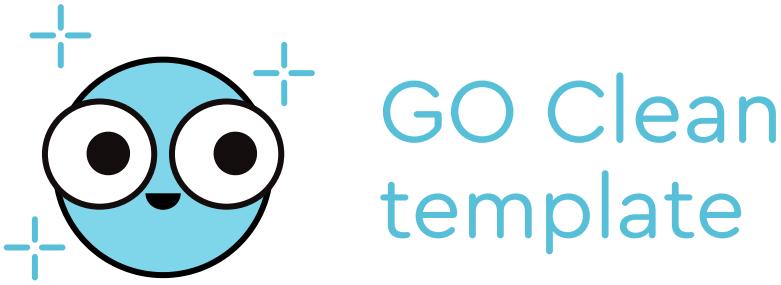
# Go整洁架构模版，建议收藏

mp.weixin.qq.com/s/co-ul3zTwbEWKGZJede7jg

以下文章来源于Go招聘 ，作者ixugo


**Go招聘 .**
Golang 相关求职和招聘，以及面试题、经验分享，Go 语言其他知识和职场也是值得分享的。

本文翻译自 https://github.com/evrone/go-clean-template，由于本人翻译水平有限，翻译不当之处烦请指出。希望大家看了这篇文章能有所帮助。感谢捧场。




概括

模板的作用：

- 如何组织项目并防止它变成一坨意大利面条式的代码。
- 在哪里存放业务逻辑，使其保持独立，整洁和可扩展。
- 如何在微服务扩展时不失控

模版使用了 Robert Martin ( 也叫 Bob 叔叔 ) 的原则[1]。

Go-clean-template[2] 此仓库由 Evrone[3] 创建及维护。

## 目录内容



## 快速开始

本地开发

```
# Postgres, RabbitMQ
$ make compose-up
# Run app with migrations
$ make run
```

集成测试 ( 可以在 CI 中运行 )

```
# DB, app + migrations, integration tests
$ make compose-up-integration-test
```

# 项目结构

```
├── cmd
│   └── app
│       └── main.go
├── config
│   ├── config.go
│   └── config.yml
├── docs
│   ├── docs.go
│   ├── swagger.json
│   └── swagger.yaml
├── go.mod
├── go.sum
├── integration-test
│   ├── Dockerfile
│   └── integration_test.go
├── internal
│   ├── app
│   │   ├── app.go
│   │   └── migrate.go
│   ├── delivery
│   │   ├── amqp_rpc
│   │   │   ├── router.go
│   │   │   └── translation.go
│   │   └── http
│   │       └── v1
│   │           ├── error.go
│   │           ├── router.go
│   │           └── translation.go
│   ├── domain
│   │   └── translation.go
│   └── service
│       ├── interfaces.go
│       ├── repo
│       │   └── translation_postgres.go
│       ├── translation.go
│       └── webapi
│           └── translation_google.go
├── migrations
│   ├── 20210221023242_migrate_name.down.sql
│   └── 20210221023242_migrate_name.up.sql
└── pkg
    ├── httpserver
    │   ├── options.go
    │   └── server.go
    ├── logger
    │   ├── interface.go
    │   ├── logger.go
    │   └── zap.go
    ├── postgres
    │   ├── options.go
    │   └── postgres.go
    └── rabbitmq
        └── rmq_rpc
            ├── client
            │   ├── client.go
            │   └── options.go
            ├── connection.go
            ├── errors.go
            └── server
```

```
        ├── options.go
        └── server.go
```

### `cmd/app/main.go`

**配置和日志实例的初始化**，`main` 函数中调用 `internal/app/app.go` 文件中 的 Run 函数，main 函数将会在此 "延续"。

### `config`

**配置**。首先读取 `config.yml` ，然后用环境变量覆盖相匹配的 yaml 配置。配置的结构体在 `config.go` 文件中。`env-required: true` 结构体标签强制您指定一个值 ( 在 yaml 或在环境变量中 )。

对于配置读取，我们选择 cleanenv[4] 库。它在 GitHub 上没有很多 star，但很简单且满足所有的需求。

从 yaml 中读取配置违背了12 要素，但在实践中，它比从环境变量中读取整个配置更方便。假设默认值定义在 yaml 中，敏感的变量定义在环境变量中。

### `docs`

**Swagger 文档**。可以由 swag[5] 库自动生成。而你不需要自己改正任何事情。

### `integration-test`

**集成测试**。它们作为单独的容器启动，紧挨着应用程序容器。使用 go-hit[6] 测试 REST API 非常方便。

### `internal/app`

在 `app.go` 文件中一般会有一个 Run 函数，它“延续”了main函数。

这是创建所有主要对象的地方。依赖注入通过“ New...”构造函数 ( 参见依赖注入 ) 。这种技术允许我们使用依赖注入原则对应用程序进行分层，使得业务逻辑独立于其他层。

接下来，为了优雅的完成，我们启动服务并在select中等待特定的信号。如果 `app.go` 代码越来越多，可以将其拆分为多个文件。

对于大量的注入，可以使用 wire[7] 库 ( wire 是一个代码生成工具，它使用依赖注入自动连接组件)。

`migrate.go` 文件用于数据库自动迁移。如果指定了 migrate 标签的参数，则会包含它。例如：

```
$ go run -tags migrate ./cmd/app
```

### `internal/delivery`

**服务的handler层** ( MVC 控制器 )。模板展示了两个服务:

- RPC ( RabbitMQ 用于传递消息 )
- REST HTTP ( GIN 框架 )

服务的路由也以同样的风格编写 :

- Handlers按照应用领域分组 ( 按公共基础 )
- 对于每个组，都创建自己的路由结构，以及处理接口路径的方法
- 业务逻辑的结构被注入到路由结构中，由handlers处理调用

### `internal/delivery/http`

**简单的 REST 版本控制**。对于 v2，我们需要添加具有相同内容的 `http/v2` 文件夹。
在 `internal/app` 程序文件中添加以下行 :

```
handler := gin.New()
v1.NewRouter(handler, translationService)
v2.NewRouter(handler, translationService)
```

你可以使用任何其他的 HTTP 框架甚至是标准的 net/http 库来代替 Gin。

在 `v1/router.go` 和上面的 handler 方法中，有一些注释是用 swag库来生成 swagger 文档的。

### `internal/domain`

**业务逻辑的实体** ( 模型 ) 可以在任何层中使用。也可以有方法，例如，用于验证。

### `internal/service`

**业务逻辑**

- 方法按应用领域分组 ( 在公共的基础上 )
- 每个组都有自己的结构
- 一个文件一个结构

Repositories、 webapi、 rpc 和其他业务逻辑结构被注入到业务逻辑结构中 ( 见依赖注入 )。

### `internal/service/repo`

repository 是业务逻辑使用的**抽象存储** ( 数据库 )。

### `internal/service/webapi`

它是业务逻辑使用的**抽象 web API**。例如，它可能是业务逻辑通过 REST API 访问的另一个微服务。包的名称根据用途而变化。

### `pkg/rabbitmq`

RabbitMQ RPC 模式 :

- RabbitMQ 内部没有路由

- 使用Exchange fanout 广播模式，将1 个独立队列绑定到其中，这是最高效的配置。
- 重新连接断开丢失的连接

## 依赖注入

为了消除业务逻辑对外部包的依赖，使用了依赖注入。

例如，通过 NewService 构造函数，我们将依赖注入到业务逻辑的结构中。这使得业务逻辑独立 ( 便于移植 )。我们可以重写接口的实现，而不需要对 `service` 包进行更改。

```
package service

import (
// Nothing!
)



type Repository interface {
    Get()
}

type Service struct {
    repo Repository
}

func NewService(r Repository) *Service{
return &Service{r}
}

    func (s *Service) Do()  {
    s.repo.Get()
}
```

它还允许我们自动生成模拟参数 ( 例如使用 mockery[8] ) 和轻松地编写单元测试。

> 我们可以不受特定实现的约束，来将一个组件更改为另一个组件。如果新组件实现了该接口，则业务逻辑中不需要进行任何更改。

## 整洁架构之道

### 关键点

程序员在编写了大量代码后才意识到应用程序的最佳架构。

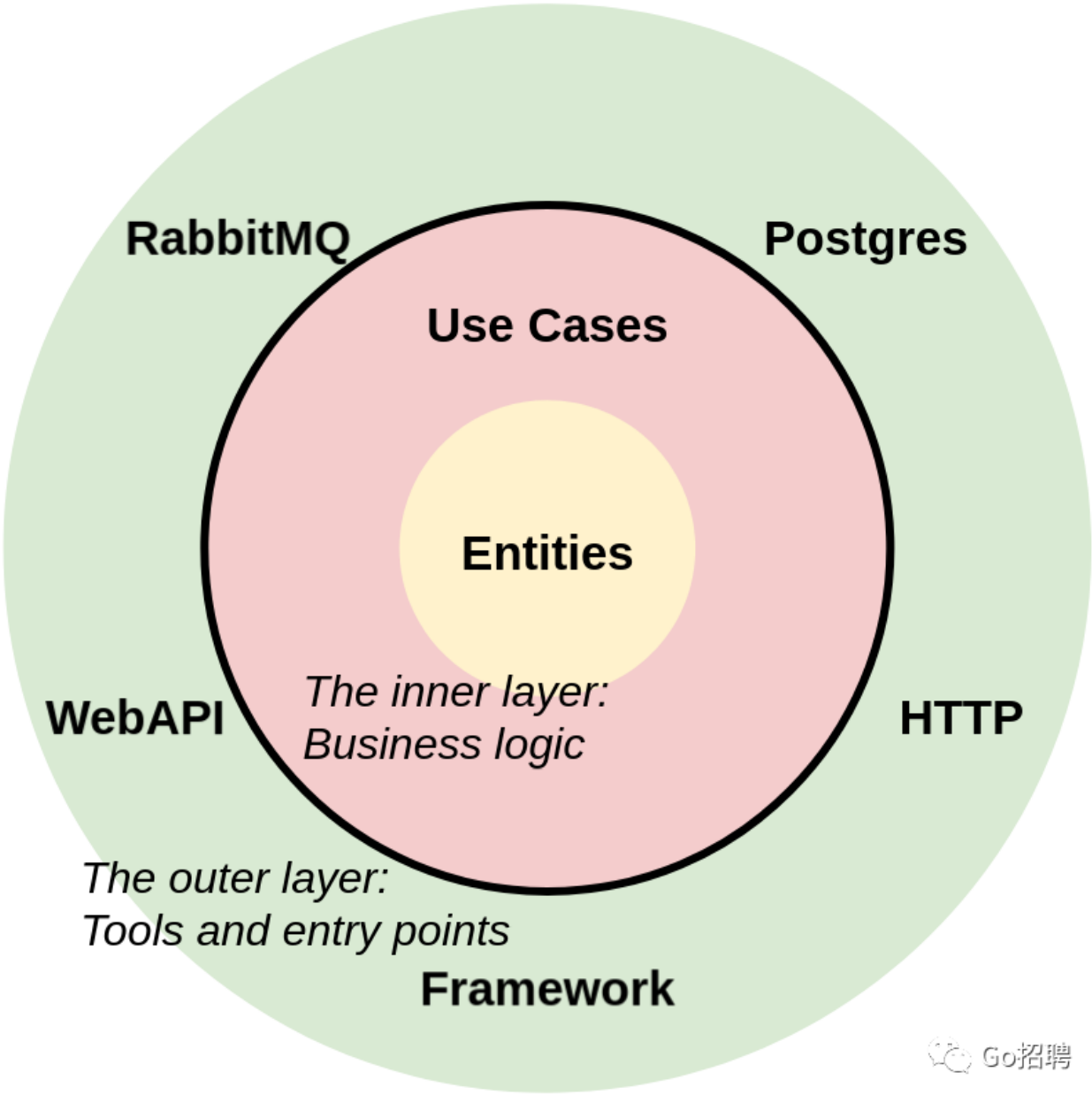
> 一个好的架构允许尽可能推迟决策。

### 主要原则

Dependency Inversion ( 与 SOLID 相同 ) 是依赖倒置的原则。依赖关系的方向是从外层到内层。由于这个原因，业务逻辑和实体仍然独立于系统的其他部分。

因此，应用程序分为内部和外部两个层次：

- **业务逻辑**（使用 Go 标准库）
- **工具**（数据库、其他服务、消息代理、任何其他包和框架）




Clean Architecture

**业务逻辑的内层**应该是整洁的，它应该：

- 没有从外层导入的包
- 只使用标准库的功能
- 通过接口调用外层！

业务逻辑对 Postgres 或详细的 web API 一无所知。业务逻辑应该具有一个用于处理抽象数据库或抽象 web API 的接口。

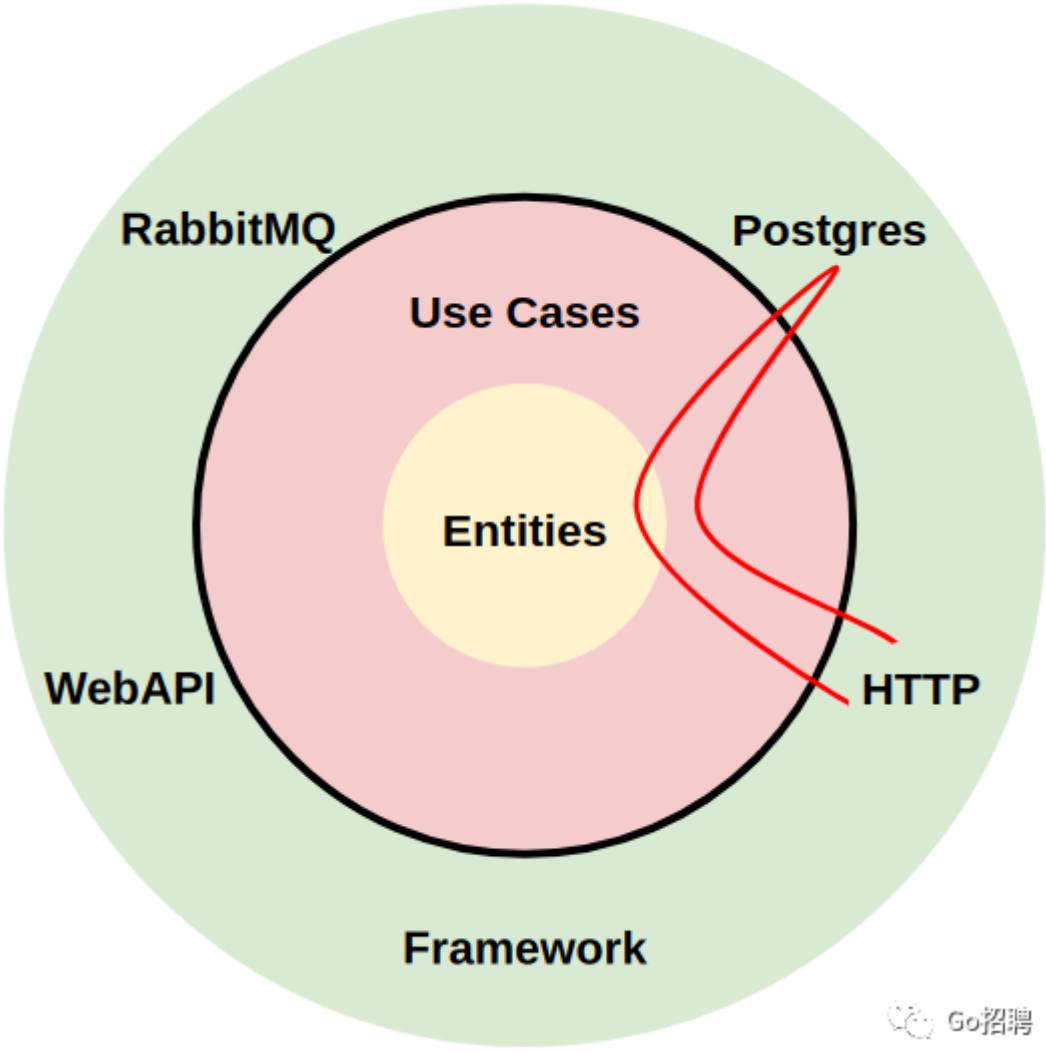
**外层**还有其他限制：

- 这一层的所有组成部分都不知道彼此的存在。如何从一个工具调用另一个工具？不是直接，而是只能通过内层的业务逻辑来调用。
- 对内层的所有调用都是通过接口来完成的
- 数据以便于业务逻辑的格式传输（ `internal/domain` ）

例如，你需要从 HTTP（控制器）访问数据库。HTTP 和数据库都在外层，这意味着它们对彼此一无所知。它们之间的通信是通过 `service`（业务逻辑）进行的：

```
HTTP > service
       service > repository (Postgres)
       service < repository (Postgres)
HTTP < service
```

符号 > 和 < 通过接口显示层与层边界的交集，如图所示：

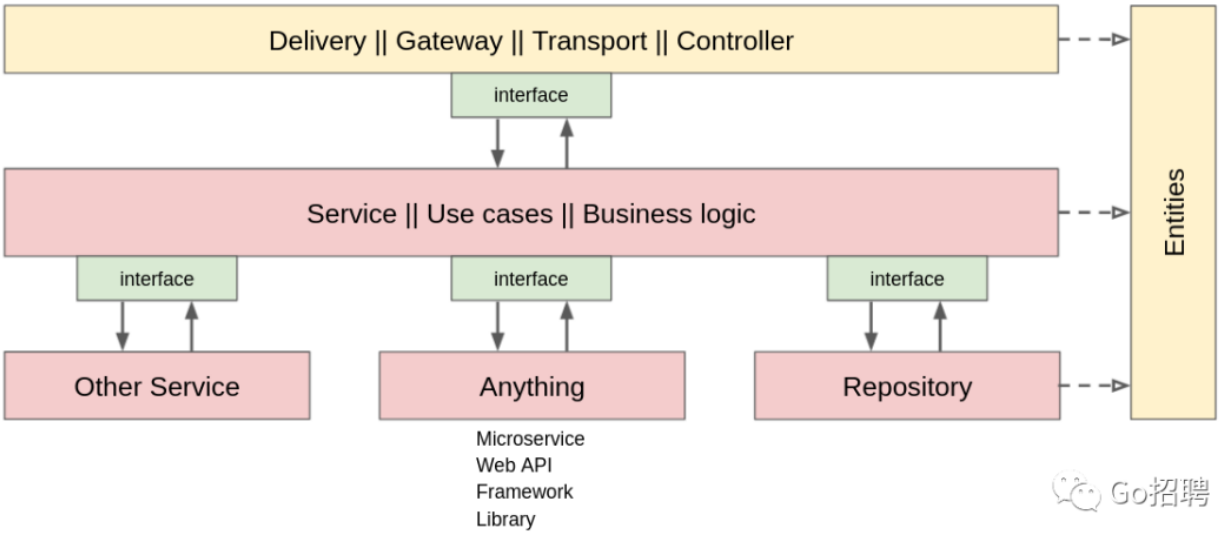



Example

或者更复杂的业务逻辑：

```
HTTP > service
       service > repository
       service < repository
       service > webapi
       service < webapi
       service > RPC
       service < RPC
       service > repository
       service < repository
HTTP < service
```

## 层级




Example

整洁架构的术语

- **实体是业务逻辑操作的结构**。它们位于 `internal/domain` 文件夹中。Domain 暗示我们坚持 DDD ( 领域驱动设计 ) 的原则，这在一定程度上是正确的。在 MVC 术语中，实体就是模型。
- **用例**是位于 `internal/service` 中的业务逻辑。从整洁架构的角度来看，调用业务逻辑使用 `service` 一词不是习惯的用法，但是对于一个包名称来说，使用一个单词 ( service ) 比使用两个单词 ( use case ) 更方便。

业务逻辑直接交互的层通常称为基础设施层。它们可以是存储库 `internal/service/repo`、web API `internal/service/webapi`、任何pkg，以及其他微服务。在模板中，_ infrastructure 包位于 `internal/service` 中。

你可以根据需要去选择如何调用入口点。选项如下：

- delivery (in our case)
- controllers
- transport
- gateways
- entrypoints

- primary
- input

### 附加层

经典版本的 整洁架构之道[9] 是为构建大型单体应用程序而设计的，它有4层。

在最初的版本中，外层被分为两个以上的层，两层之间也存在相互依赖关系倒置 ( 定向内部 )，并通过接口进行通信。

在逻辑复杂的情况下，内层也分为两个( 接口分离 )。

复杂的工具可以被划分成更多的附加层，但你应该在确实需要时再添加层。

### 替代方法

除了整洁架构之道，洋葱架构和六边形架构 ( 端口适配器模式 ) 是类似的。两者都是基于依赖倒置的原则。端口和适配器模式非常接近于整洁架构之道，差异主要在术语上。

## 类似的项目

- https://github.com/bxcodec/go-clean-arch
- https://github.com/zhashkevych/courses-backend

## 扩展阅读链接

- 整洁架构之道[10]
- 12 要素[11]

### 参考资料


[1]
原则: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

[2]
Go-clean-template: https://evrone.com/go-clean-template?utm_source=github&utm_campaign=go-clean-template

[3]
Evrone: https://evrone.com/?utm_source=github&utm_campaign=go-clean-template

[4]
cleanenv: https://github.com/ilyakaznacheev/cleanenv

[5]
swag: https://github.com/swaggo/swag

[6]
go-hit: https://github.com/Eun/go-hit

[7]
wire: https://github.com/google/wire

[8]
mockery: https://github.com/vektra/mockery

[9]
整洁架构之道: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

[10]
整洁架构之道: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

[11]
12 要素: https://12factor.net/ru/

---