Go进阶训练营
第4课
Go 工程化实践

毛剑

目录

Standard Go Project Layout

https://github.com/golang-standards/project-layout/blob/master/README_zh.md

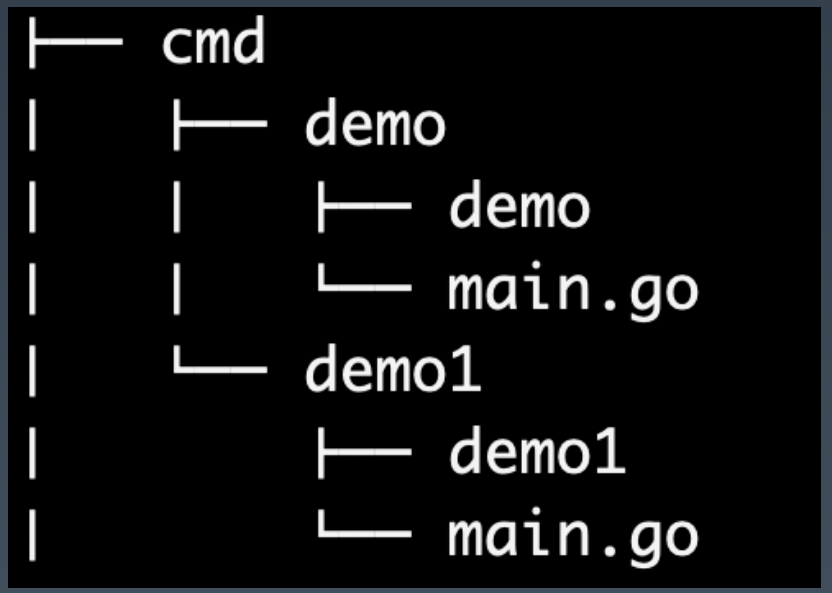
如果你尝试学习 *Go*，或者你正在为自己建立一个 *PoC* 或一个玩具项目，这个项目布局是没啥必要的。从一些非常简单的事情开始(一个 *main.go* 文件绰绰有余)。当有更多的人参与这个项目时，你将需要更多的结构，包括需要一个 *toolkit* 来方便生成项目的模板，尽可能大家统一的工程目录布局。

- */cmd*

  本项目的主干。

每个应用程序的目录名应该与你想要的可执行文件的名称相匹配(例如，/cmd/myapp)。

不要在这个目录中放置太多代码。如果你认为代码可以导入并在其他项目中使用，那么它应该位于 /pkg 目录中。如果代码不是可重用的，或者你不希望其他人重用它，请将该代码放到 /internal 目录中。



Standard Go Project Layout

- /internal

私有应用程序和库代码。这是你不希望其他人在其应用程序或库中导入代码。请注意，这个布局模式是由 Go 编译器本身执行的。有关更多细节，请参阅Go 1.4 release notes。注意，你并不局限于顶级 internal 目录。在项目树的任何级别上都可以有多个内部目录。

你可以选择向 internal 包中添加一些额外的结构，以分隔共享和非共享的内部代码。这不是必需的(特别是对于较小的项目)，但是最好有有可视化的线索来显示预期的包的用途。你的实际应用程序代码可以放在 /internal/app 目录下(例如 /internal/app/myapp)，这些应用程序共享的代码可以放在 /internal/pkg 目录下(例如 /internal/pkg/myprivlib)。

因为我们习惯把相关的服务，比如账号服务，内部有 rpc、job、admin 等，相关的服务整合一起后，需要区分 app。单一的服务，可以去掉 /internal/myapp。

```
├── internal
|   └── demo
|       ├── biz
|       ├── data
|       └── service
```

Standard Go Project Layout

- /pkg

外部应用程序可以使用的库代码(例如 /pkg/mypubliclib)。其他项目会导入这些库，所以在这里放东西之前要三思:-)注意，internal 目录是确保私有包不可导入的更好方法，因为它是由 Go 强制执行的。/pkg 目录仍然是一种很好的方式，可以显式地表示该目录中的代码对于其他人来说是安全使用的好方法。

/pkg 目录内，可以参考 go 标准库的组织方式，按照功能分类。/internla/pkg 一般用于项目内的 跨多个应用的公共共享代码，但其作用域仅在单个项目工程内。

由 Travis Jeffery 撰写的 I'll take pkg over internal 博客文章提供了 pkg 和 internal 目录的一个很好的概述，以及什么时候使用它们是有意义的。

当根目录包含大量非 Go 组件和目录时，这也是一种将 Go 代码分组到一个位置的方法，这使得运行各种 Go 工具变得更加容易组织。

```
pkg
├── cache
|   ├── memcache
|   └── redis
├── conf
|   ├── dsn
|   ├── env
|   ├── flagvar
|   └── paladin
```

```
.
├── docs
├── example
├── misc
├── pkg
├── third_party
└── tool
```

Kit Project Layout

每个公司都应当为不同的微服务建立一个统一的 kit 工具包项目(基础库/框架) 和 app 项目。

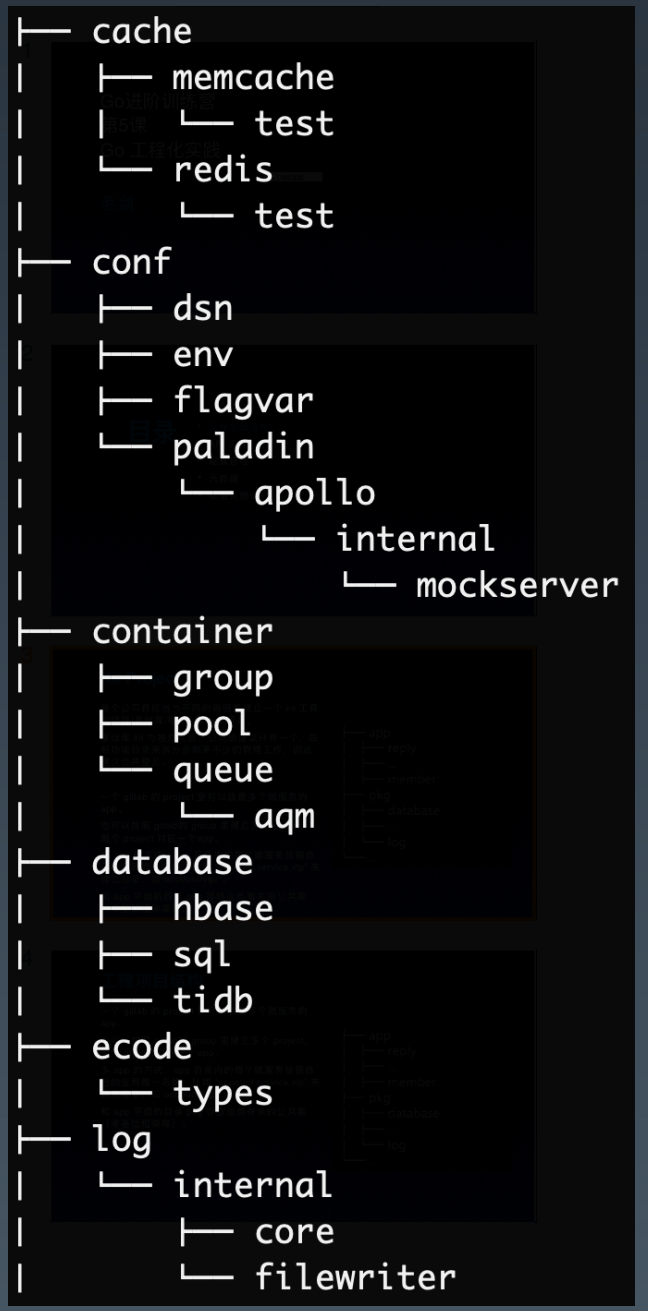
基础库 kit 为独立项目，公司级建议只有一个，按照功能目录来拆分会带来不少的管理工作，因此建议合并整合。

by Package Oriented Design

"To this end, the Kit project is not allowed to have a vendor folder. If any of packages are dependent on 3rd party packages, they must always build against the latest version of those dependences."

kit 项目必须具备的特点:

- *统一*
- *标准库方式布局*
- *高度抽象*
- *支持插件*



Service Application Project Layout

- /api

  API 协议定义目录，xxapi.proto protobuf 文件，以及生成的 go 文件。我们通常把 api 文档直接在 proto 文件中描述。

- /configs

  配置文件模板或默认配置。

- /test

  额外的外部测试应用程序和测试数据。你可以随时根据需求构造 /test 目录。对于较大的项目，有一个数据子目录是有意义的。例如，你可以使用 /test/data 或 /test/testdata (如果你需要忽略目录中的内容)。请注意，Go 还会忽略以“.”或“_”开头的目录或文件，因此在如何命名测试数据目录方面有更大的灵活性。

不应该包含：/src

有些 Go 项目确实有一个 src 文件夹，但这通常发生在开发人员有 Java 背景，在那里它是一种常见的模式。不要将项目级别 src 目录与 Go 用于其工作空间的 src 目录。

```
.
├── README.md
├── api
├── cmd
├── configs
├── go.mod
├── go.sum
├── internal
└── test
```

Service Application Project

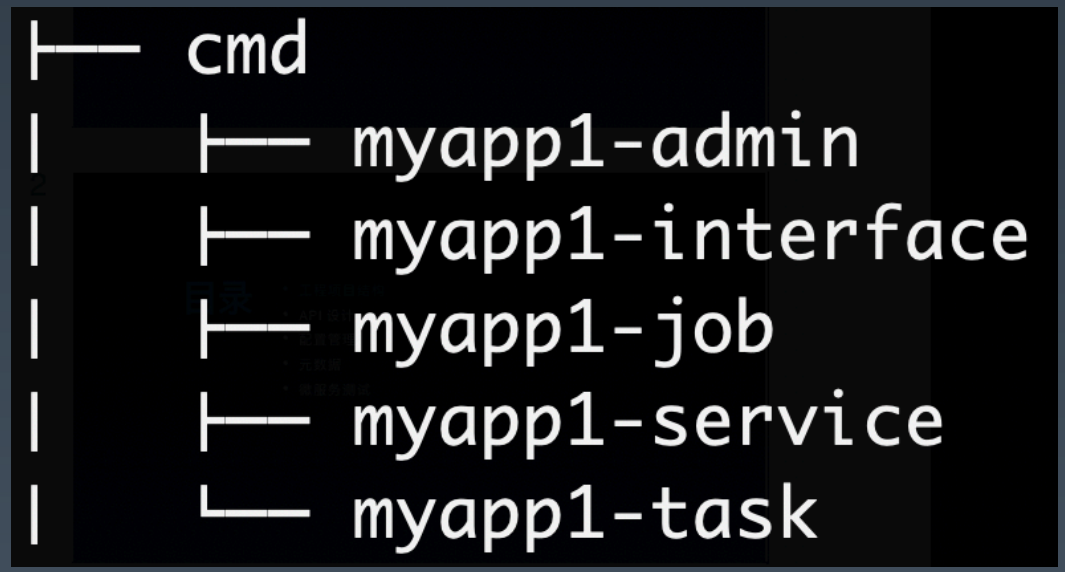
一个 gitlab 的 project 里可以放置多个微服务的 app(类似 monorepo)。也可以按照 gitlab 的 group 里建立多个 project，每个 project 对应一个 app。

- *多 app 的方式，app 目录内的每个微服务按照自己的全局唯一名称，比如 “account.service.vip” 来建立目录，如: account/vip/*。*
- *和 app 平级的目录 pkg 存放业务有关的公共库（非基础框架库）。如果应用不希望导出这些目录，可以放置到 myapp/internal/pkg 中。*

Service Application Project

微服务中的 app 服务类型分为4类：interface、service、job、admin。

- *interface: 对外的 BFF 服务，接受来自用户的请求，比如暴露了 HTTP/gRPC 接口。*
- *service: 对内的微服务，仅接受来自内部其他服务或者网关的请求，比如暴露了gRPC 接口只对内服务。*
- *admin：区别于 service，更多是面向运营测的服务，通常数据权限更高，隔离带来更好的代码级别安全。*
- *job: 流式任务处理的服务，上游一般依赖 message broker。*
- *task: 定时任务，类似 cronjob，部署到 task 托管平台中。*



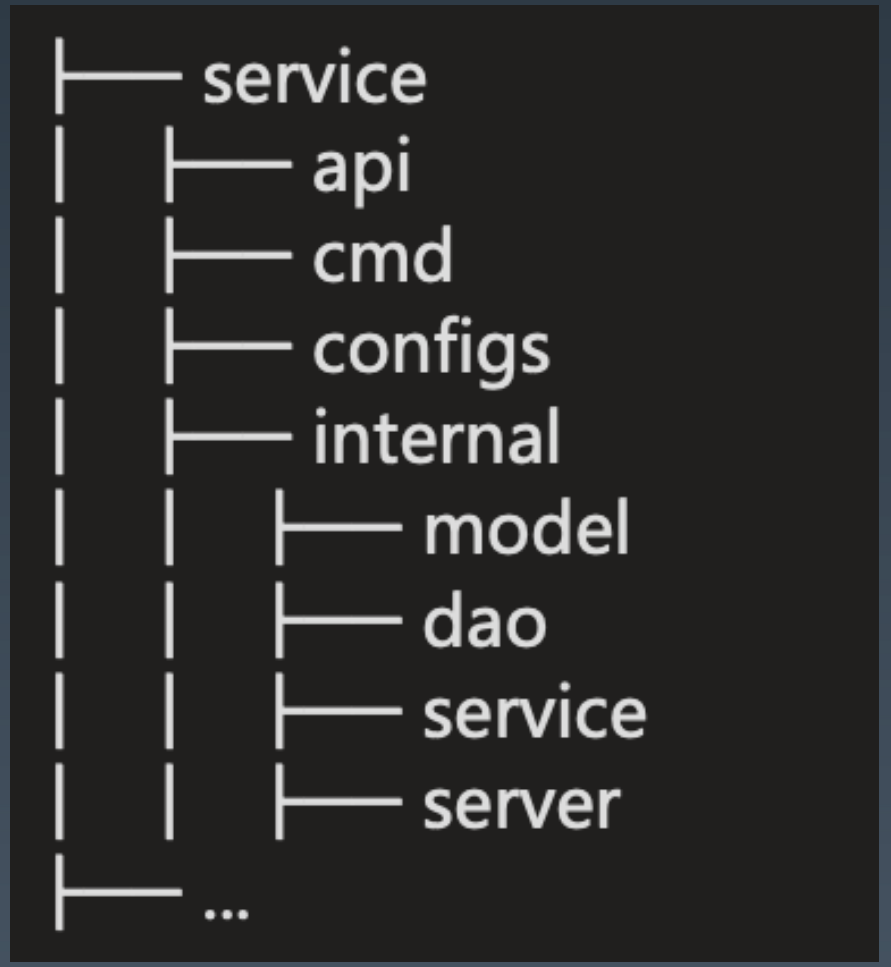
cmd 应用目录负责程序的: 启动、关闭、配置初始化等。

Service Application Project - v1

我们老的布局，app 目录下有 api、cmd、configs、internal 目录，目录里一般还会放置 README、CHANGELOG、OWNERS。

- *api: 放置 API 定义(protobuf)，以及对应的生成的 client 代码，基于 pb 生成的 swagger.json。*
- *configs: 放服务所需要的配置文件，比如database.yaml、redis.yaml、application.yaml。*
- *internal: 是为了避免有同业务下有人跨目录引用了内部的 model、dao 等内部 struct。*
- *server: 放置 HTTP/gRPC 的路由代码，以及 DTO 转换的代码。*

DTO(Data Transfer Object)：数据传输对象，这个概念来源于 J2EE 的设计模式。但在这里，泛指用于展示层/API 层与服务层(业务逻辑层)之间的数据传输对象。



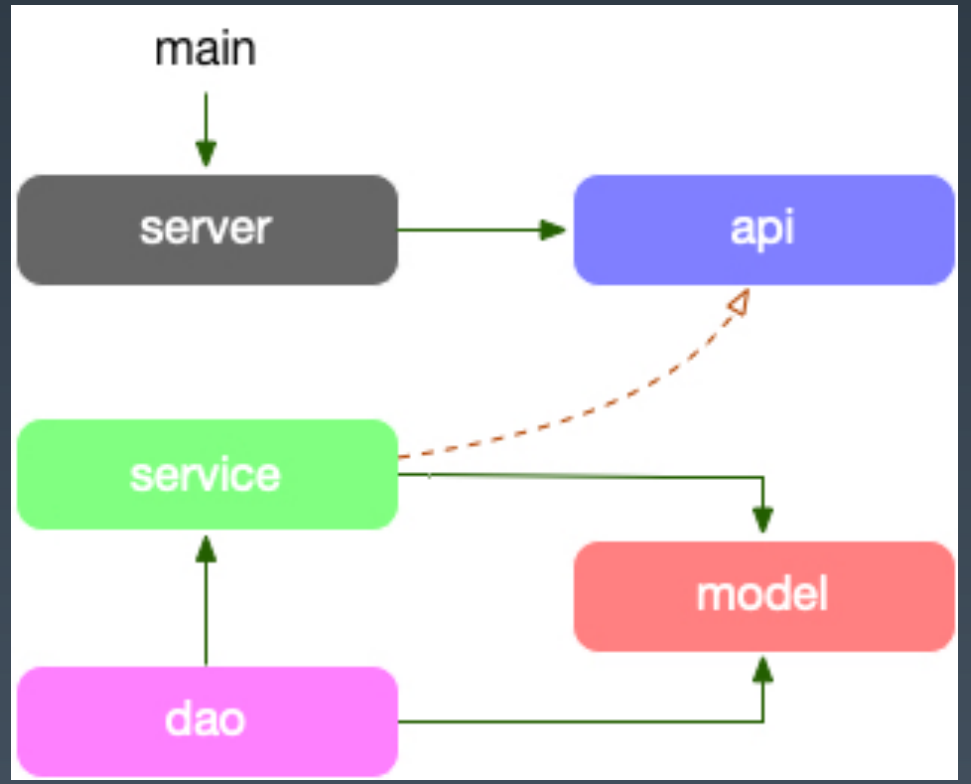
Service Application Project - v1

项目的依赖路径为: model -> dao -> service -> api，model struct 串联各个层，直到 api 需要做 DTO 对象转换。

- *model: 放对应“存储层”的结构体，是对存储的一一隐射。*
- *dao: 数据读写层，数据库和缓存全部在这层统一处理，包括 cache miss 处理。*
- *service: 组合各种数据访问来构建业务逻辑。*
- *server: 依赖 proto 定义的服务作为入参，提供快捷的启动服务全局方法。*
- *api: 定义了 API proto 文件，和生成的 stub 代码，它生成的 interface，其实现者在 service 中。*

service 的方法签名因为实现了 API 的 接口定义，DTO 直接在业务逻辑层直接使用了，更有 dao 直接使用，最简化代码。

DO(Domain Object): 领域对象，就是从现实世界中抽象出来的有形或无形的业务实体。缺乏 DTO -> DO 的对象转换。



Service Application Project - v2

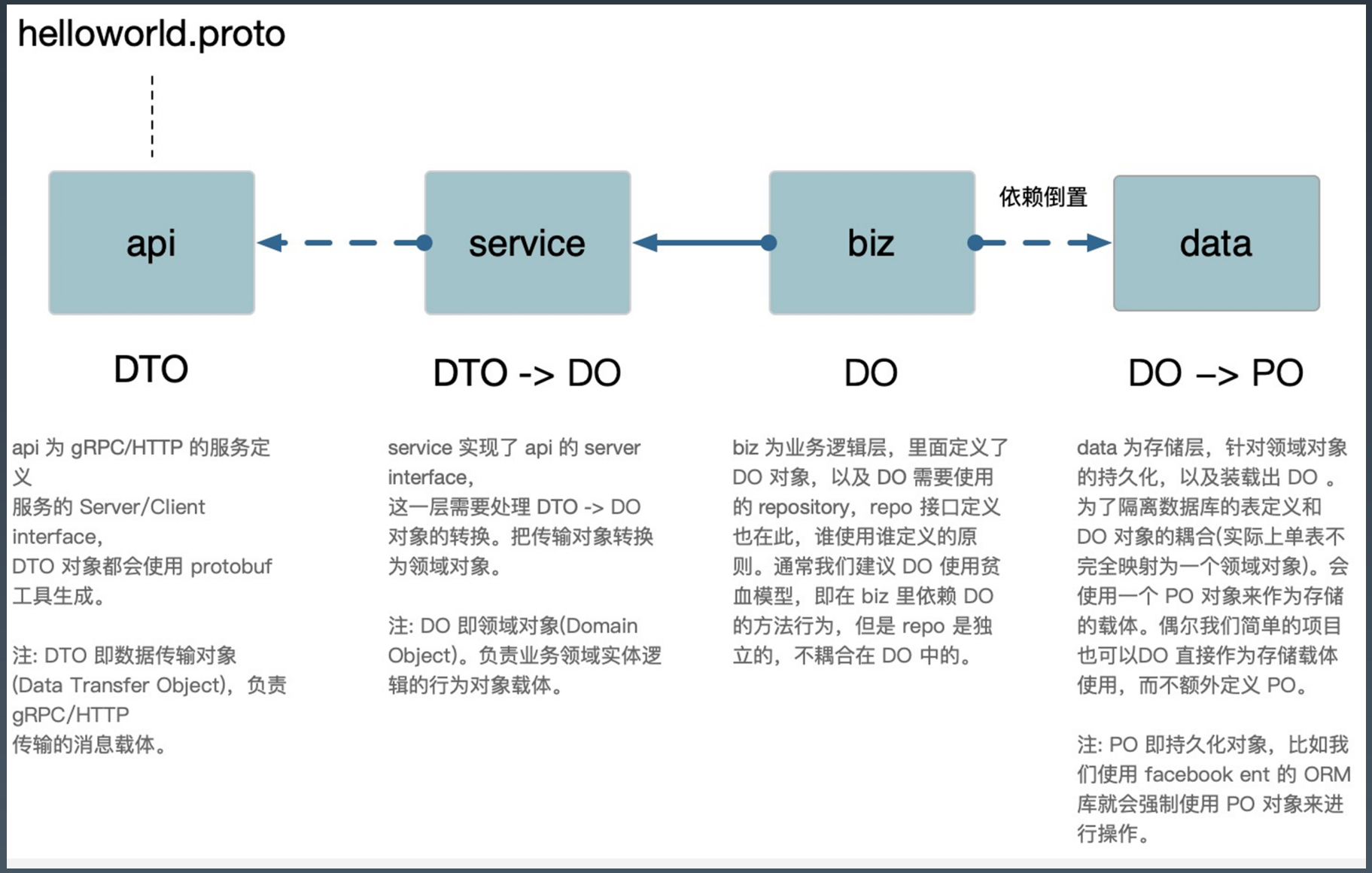
app 目录下有 api、cmd、configs、internal 目录，目录里一般还会放置 README、CHANGELOG、OWNERS。

- *internal: 是为了避免有同业务下有人跨目录引用了内部的 biz、data、service 等内部 struct。*
  - *biz: 业务逻辑的组装层，类似 DDD 的 domain 层，data 类似 DDD 的 repo，repo 接口在这里定义，使用依赖倒置的原则。*
  - *data: 业务数据访问，包含 cache、db 等封装，实现了 biz 的 repo 接口。我们可能会把 data 与 dao 混淆在一起，data 偏重业务的含义，它所要做的是将领域对象重新拿出来，我们去掉了 DDD 的 infra 层。*
  - *service: 实现了 api 定义的服务层，类似 DDD 的 application 层，处理 DTO 到 biz 领域实体的转换(DTO -> DO)，同时协同各类 biz 交互，但是不应处理复杂逻辑。*

PO(Persistent Object): 持久化对象，它跟持久层（通常是关系型数据库）的数据结构形成一一对应的映射关系，如果持久层是关系型数据库，那么数据表中的每个字段（或若干个）就对应 PO 的一个（或若干个）属性。https://github.com/facebook/ent

```
.
├── CHANGELOG
├── OWNERS
├── README
├── api
├── cmd
│   ├── myapp1-admin
│   ├── myapp1-interface
│   ├── myapp1-job
│   ├── myapp1-service
│   └── myapp1-task
├── configs
├── go.mod
└── internal
    ├── biz
    ├── data
    ├── pkg
    └── service
```

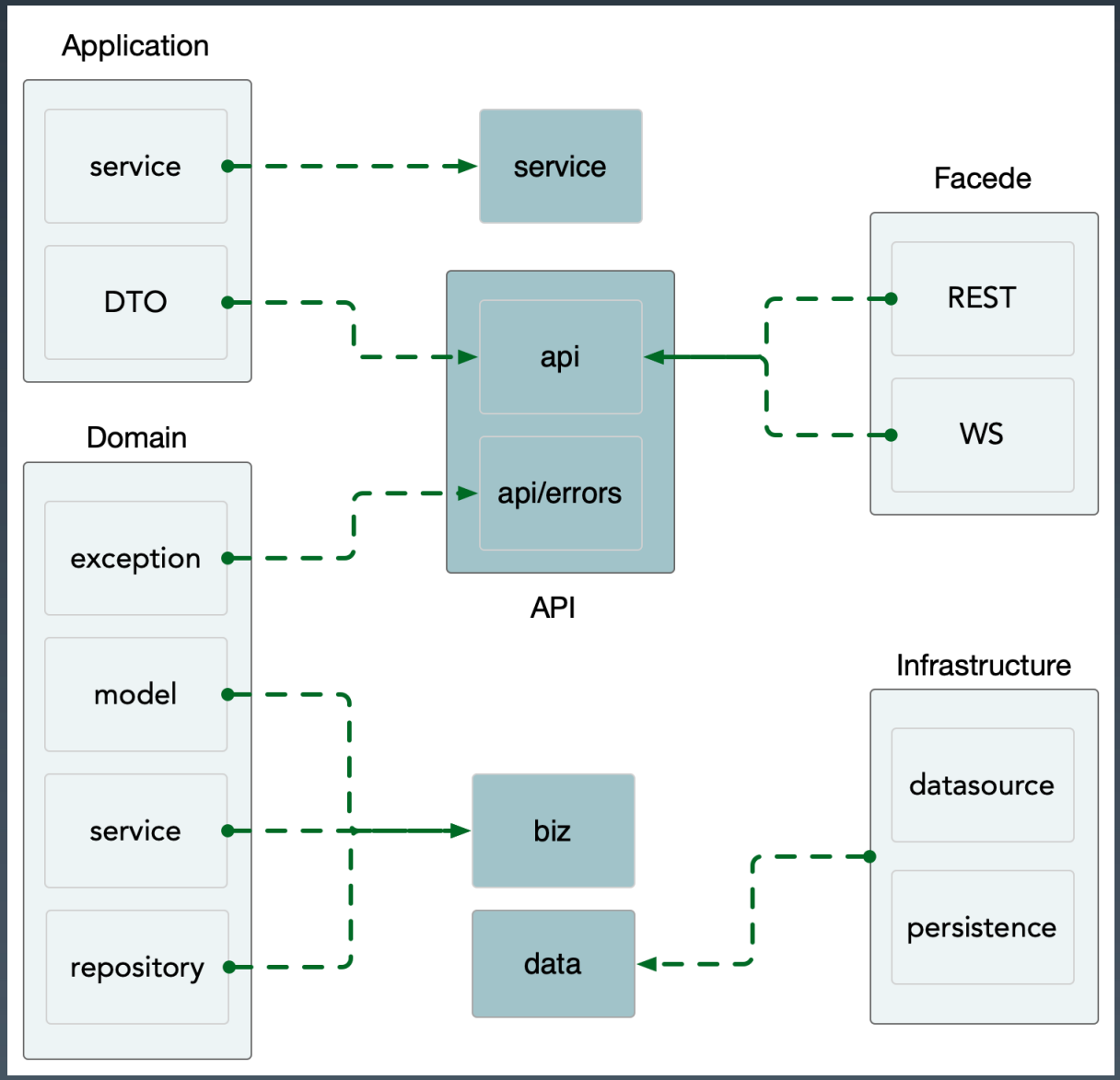
Service Application Project - v2



Service Application Project - v2

我们将 DDD 设计中的一些思想和工程结构做了一些简化，映射到 api、service、biz、data 各层。

在 review PPT 的时候，发现之前 kratos 设计中关于 bussiness error 的定义是放在了 api 层。实际业务错误属于归属于业务领域层。比较合理的应该调整 errros 定义在 biz 中。只需要处理好业务错误到传输（HTTP/gRPC）的转换就行。



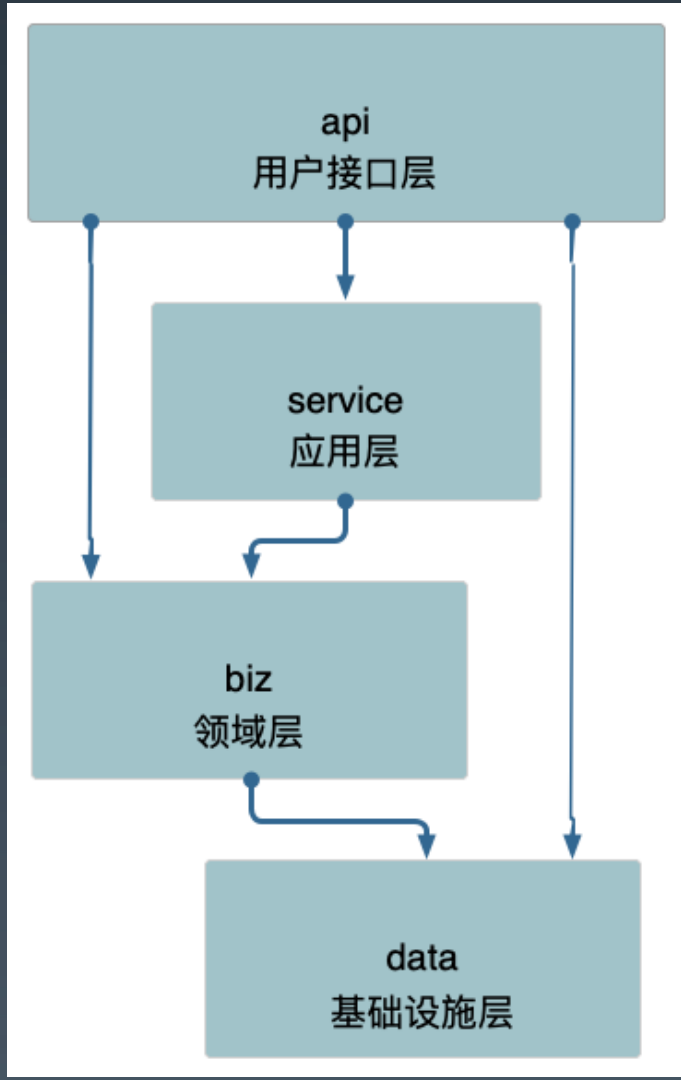
Service Application Project - v2

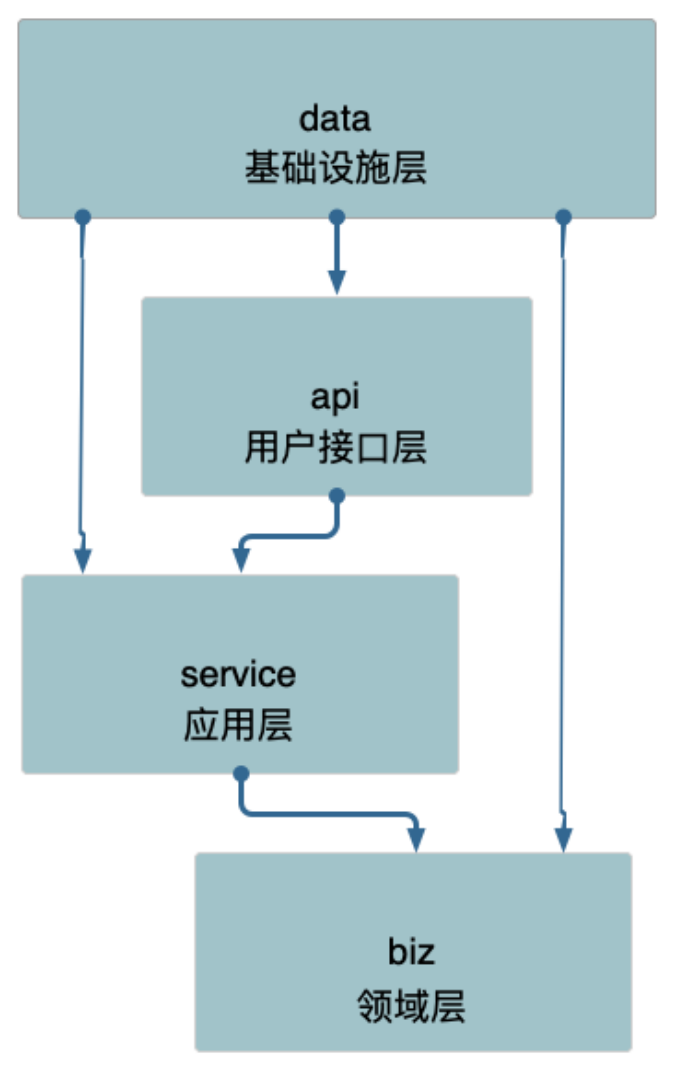
松散分层架构（Relaxed Layered System）

层间关系不那么严格。每层都可能使用它下面所有层的服务，而不仅仅是下一层的服务。每层都可能是半透明的，这意味着有些服务只对上一层可见，而有些服务对上面的所有层都可见。

同时在领域驱动设计（DDD）中也采用了继承分层架构（Layering Through Inheritance），高层继承并实现低层接口。我们需要调整一下各层的顺序，并且将基础设施层移动到最高层。

注意：继承分层架构依然是单向依赖，这也意味着领域层、应用层、表现层将不能依赖基础设施层，相反基础设施层可以依赖它们。





Service Application Project - v2

- 失血模型

模型仅仅包含数据的定义和 getter/setter 方法，业务逻辑和应用逻辑都放到服务层中。这种类在 Java 中叫 POJO，在 .NET 中叫 POCO。

- 贫血模型

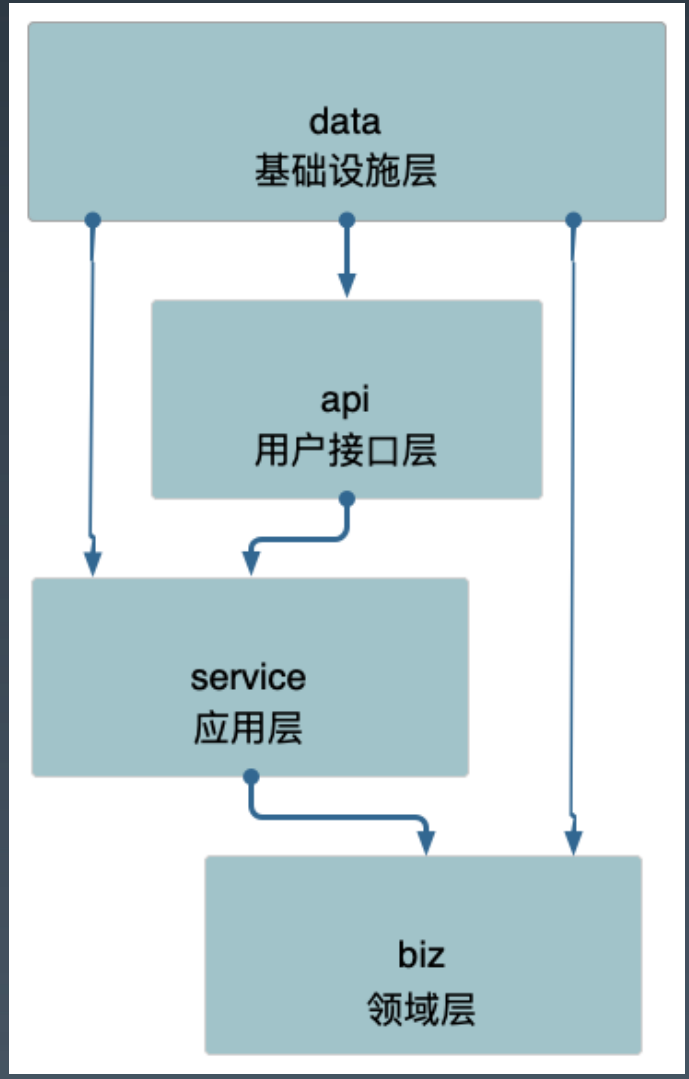
贫血模型中包含了一些业务逻辑，但不包含依赖持久层的业务逻辑。这部分依赖于持久层的业务逻辑将会放到服务层中。可以看出，贫血模型中的领域对象是不依赖于持久层的。

- 充血模型

充血模型中包含了所有的业务逻辑，包括依赖于持久层的业务逻辑。所以，使用充血模型的领域层是依赖于持久层，简单表示就是 UI层->服务层->领域层<->持久层。

- 胀血模型

胀血模型就是把和业务逻辑不想关的其他应用逻辑（如授权、事务等）都放到领域模型中。我感觉胀血模型反而是另外一种的失血模型，因为服务层消失了，领域层干了服务层的事，到头来还是什么都没变。



Lifecycle

Lifecycle 需要考虑服务应用的对象初始化以及生命周期的管理，所有 HTTP/gRPC 依赖的前置资源初始化，包括 data、biz、service，之后再启动监听服务。我们使用 https://github.com/google/wire ，来管理所有资源的依赖注入。为何需要依赖注入？

```python
class RedisList:
    def __init__(self, host, port, password):
        self._client = redis.Redis(host, port, password)

    def push(self, key, val):
        self._client.lpush(key, val)

l = RedisList(host, port, password)
```

依赖翻转之后是这样的：

```python
class RedisList:
    def __init__(self, redis_client)
        self._client = redis_client

    def push(self, key, val):
        self._client.lpush(key, val)

redis_client = get_redis_client(...)
l = RedisList(redis_client)
```

核心是为了：1、方便测试；2、单次初始化和复用；

```go
package main

import (
    "context"
    kratos "go-project-layout"
    "go-project-layout/server/http"
    "log"
)

func main() {
    svr := http.NewServer()
    app := kratos.New()
    app.Append(kratos.Hook{
        OnStart: func(ctx context.Context) error {
            return svr.Start()
        },
        OnStop: func(ctx context.Context) error {
            return svr.Shutdown(ctx)
        },
    })

    if err := app.Run(); err != nil {
        log.Printf("app failed: %v\n", err)
        return
    }
}
```

Wire

https://blog.golang.org/wire

手撸资源的初始化和关闭是非常繁琐，容易出错的。上面提到我们使用依赖注入的思路 DI，结合 google wire，静态的 go generate 生成静态的代码，可以在很方便诊断和查看，不是在运行时利用 reflection 实现。

```
type Message string

type Greeter struct {
    // ... TBD
}

type Event struct {
    // ... TBD
}
```

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

    event.Start()
}
```

```
// wire_gen.go

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

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

```
// wire.go

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

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

type Greeter struct {
    Message Message // <- adding a Message field
}
```