



GoFrame框架介绍及设计

×



郭强

成都医联科技
架构师



目 录

-  框架介绍 01
-  模块化设计 02
-  统一框架设计 03
-  代码分层设计 04
-  对象封装设计 05
-  DAO封装设计 06
-  未来发展规划 07

第一部分

框架介绍

- 框架介绍
- 框架架构
- 项目初心

框架介绍

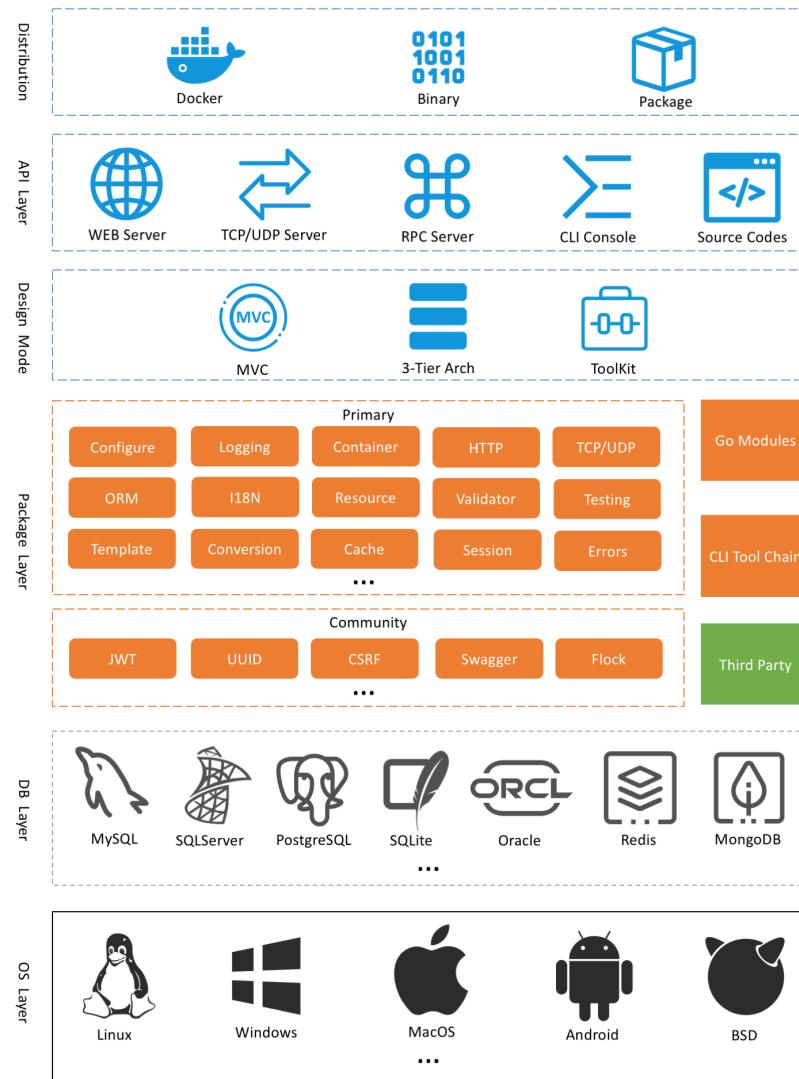


GoFrame是一款模块化、高性能、企业级的Go基础开发框架。

特点

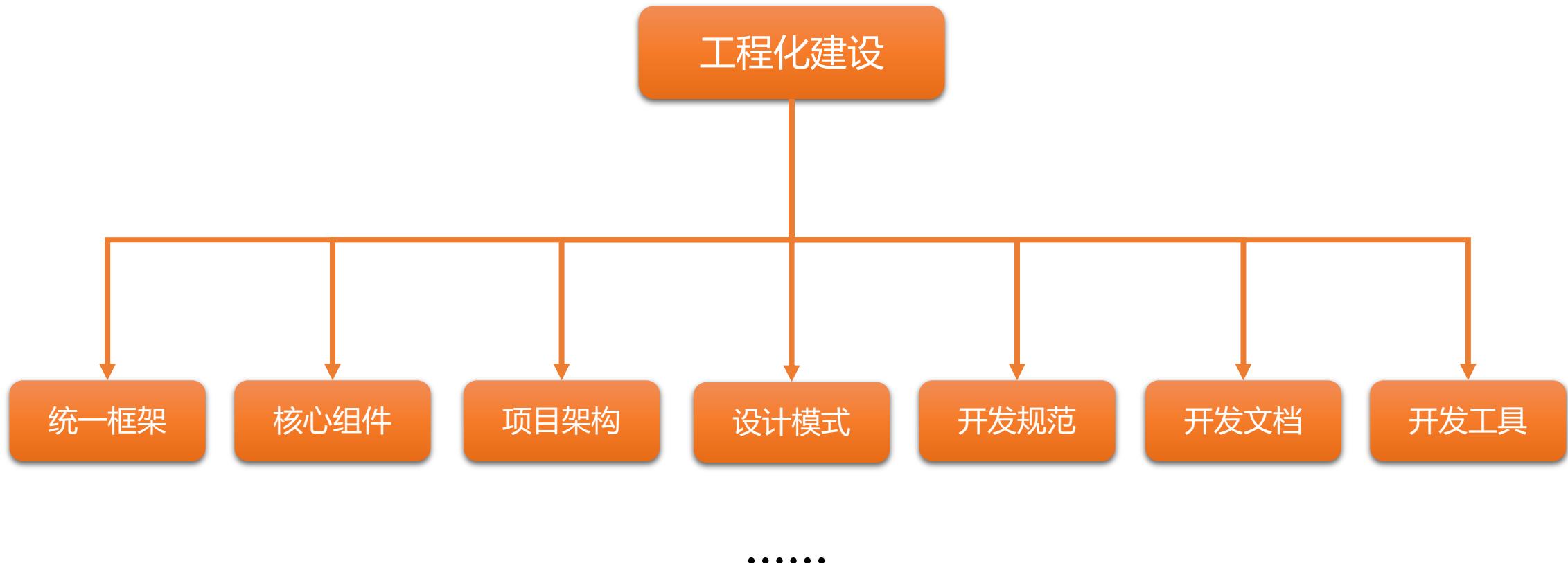
- 模块化、松耦合
- 模块丰富、开箱即用
- 简洁易用、快速接入
- 文档详尽、易于维护
- 自顶向下、体系化设计
- 统一框架、统一组件、降低选择成本
- 开发规范、设计模式、代码分层模型
- 强大便捷的开发工具链
- 完善的本地化文化支持
- 设计为团队及企业使用

框架介绍-框架架构



- 发布方式：Docker、二级制、源码模块
- 应用接口：HTTP/TCP/UDP/RPC Server、终端应用、源码接口
- 设计模式：MVC、三层架构、工具集
- 模块管理：
 - 核心模块、社区模块、三方模块
 - Go Modules管理方式
 - 开发工具链
- 数据库类型：通过标准库驱动接口支持多种数据库类型
- 跨平台性：基于Golang开发语言强大跨平台特性

框架介绍-项目初心



第二部分

模块化设计

- 复用原则
- 单仓包设计
- 模块聚合设计
- 常见问题

模块化设计

什么是模块？

模块也称作**组件**，是软件系统中可复用的功能逻辑封装单位。

在不同的软件架构层次，模块的概念会有些不太一样。

在开发框架层面，模块是某一类功能逻辑的最小封装单位。

在Golang代码层面中，我们也可以将package称作模块。

模块化的目标？

软件进行模块化设计的目的，

是为了使得软件功能逻辑尽可能的**解耦**和**复用**，

终极目标也是为了保证软件开发维护的效率和质量。

模块化设计-复用原则

REP 发布等同原则 (Release/Reuse Equivalency Principle)

软件复用的最小粒度应等同于其发布的最小粒度。

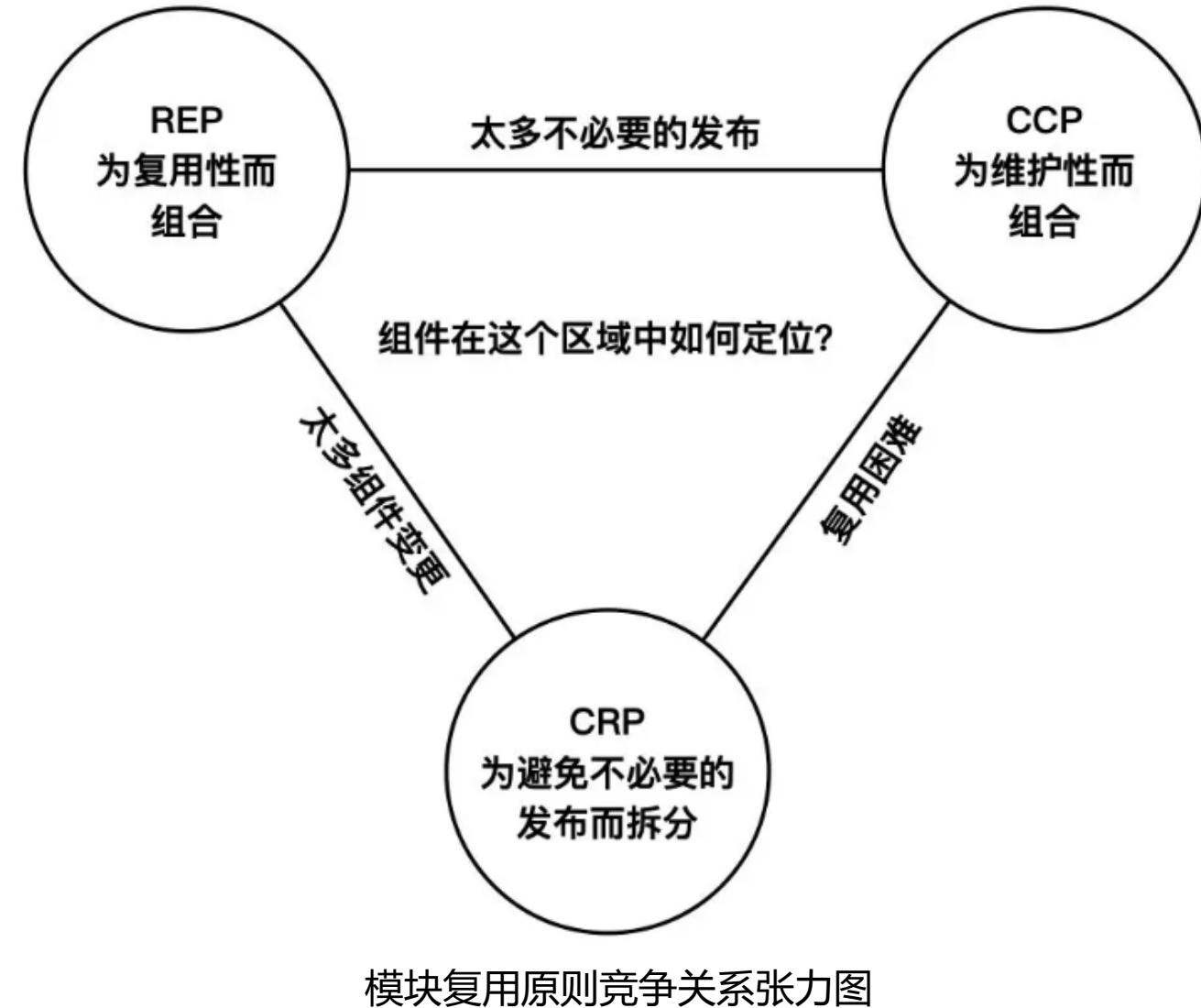
CCP 共同闭包原则 (Common Closure Principle)

为了相同目的而同时修改的类，应该放在同一个模块中。

对大部分应用程序而言，可维护性的重要性远远大于可复用性，由同一个原因引起的代码修改，最好在同一个模块中，如果分散在多个模块中，那么开发、提交、部署的成本都会上升。

CRP 共同复用原则 (Common Reuse Principle)

不要强迫一个模块依赖它不需要的东西。



模块化设计-单仓包设计

```
module business

go 1.16

require (
    business.com/golang/strings v1.0.0
    business.com/golang/config v1.15.0
    business.com/golang/container v1.1.0
    business.com/golang/encoding v1.2.0
    business.com/golang/files v1.2.1
    business.com/golang/cache v1.7.3
    business.com/framework/utils v1.30.1
    github.com/pkg/errors v0.9.0
    github.com/goorm/orm v1.2.1
    github.com/goredis/redis v1.7.4
    github.com/gokafka/kafka v0.1.0
    github.com/gometrics/metrics v0.3.5
    github.com/gotracing/tracing v0.8.2
    github.com/gohttp/http v1.18.1
    github.com/google/grpc v1.16.1
    github.com/smith/env v1.0.2
    github.com/htbj/command v1.1.1
    github.com/kmlevel1/pool v1.1.4
    github.com/analog/logging v1.16.2
    github.com/bgses123/session v1.5.1
    github.com/gomytmp/template v1.3.4
    github.com/govalidation/validate v1.19.2
    github.com/yetme1/goi18n v0.10.0
    github.com/convman/convert v1.20.0
    github.com/google/uuid v1.1.2
    // ...
)
```



痛点：

- 实现相同功能逻辑的模块百花齐放，选择成本过高
- 项目依赖的模块过多，项目整体的稳定性会受到影响
- 项目依赖的模块过多，无从下手是否应当升级模块版本
- 各个模块孤立设计，单独看待每个模块可替换性很高，开发体系难以建立，开发规范难以统一

模块化设计-模块聚合设计

```
module business

go 1.16

require (
    github.com/gogf/gf v1.16.0
    github.com/goorm/orm v1.15.1
    github.com/goredis/redis v1.7.4
    github.com/gokafka/kafka v0.1.0
    github.com/google/grpc v1.16.1
    // ...
)
```

- 框架核心维护较全面的通用基础模块，降低基础模块选择成本
- 我们只需要维护一个统一的框架版本，而不是数十个模块版本

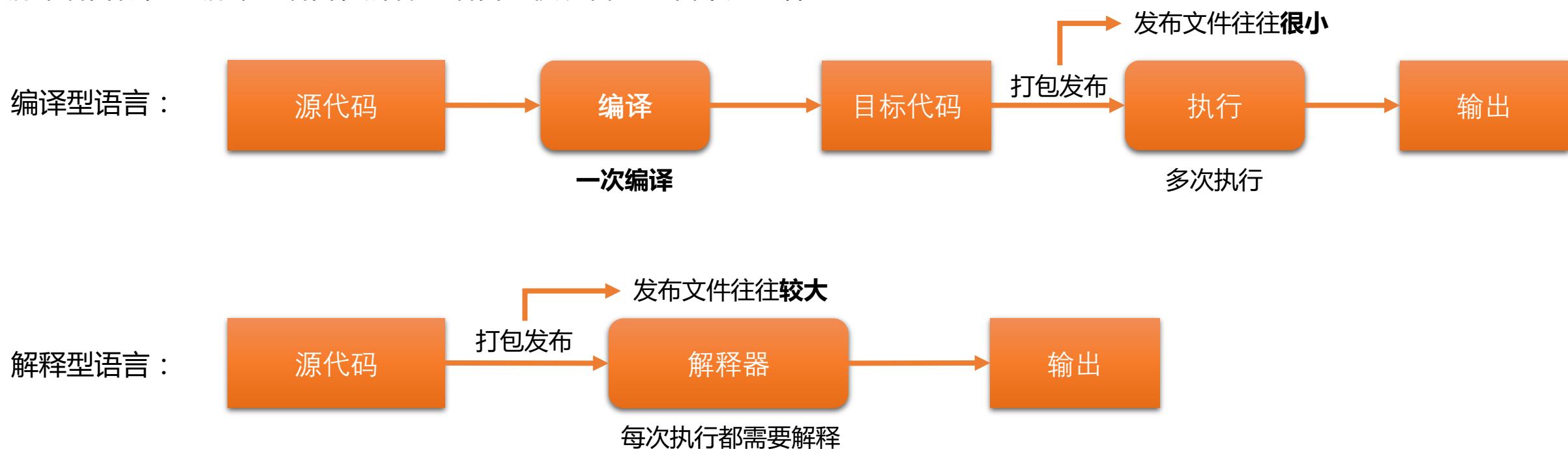
改进：

- 我们只需要了解一个框架的内容变化，而不是数十个模块的内容变化
- 升级的时候只需要升级一个框架版本，而不是数十个模块版本的升级
- 减轻心智负担，提高模块可维护性，更容易保证各业务项目的模块版本一致性

模块化设计-常见问题

虽然框架每一个模块都按照低耦合设计，模块可以选择性引入，但在使用时也得全量下载完整框架代码。

1. 模块低耦设计：文件层面的源文件下载与模块之间的逻辑耦合没有直接关系
2. 框架核心精简：功能强大且代码精简，包含测试与示例代码共8MB容量
3. 编译语言特性：编译型语言和解释型语言的模块管理逻辑不太一样

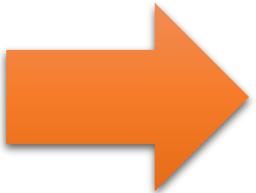


第三部分

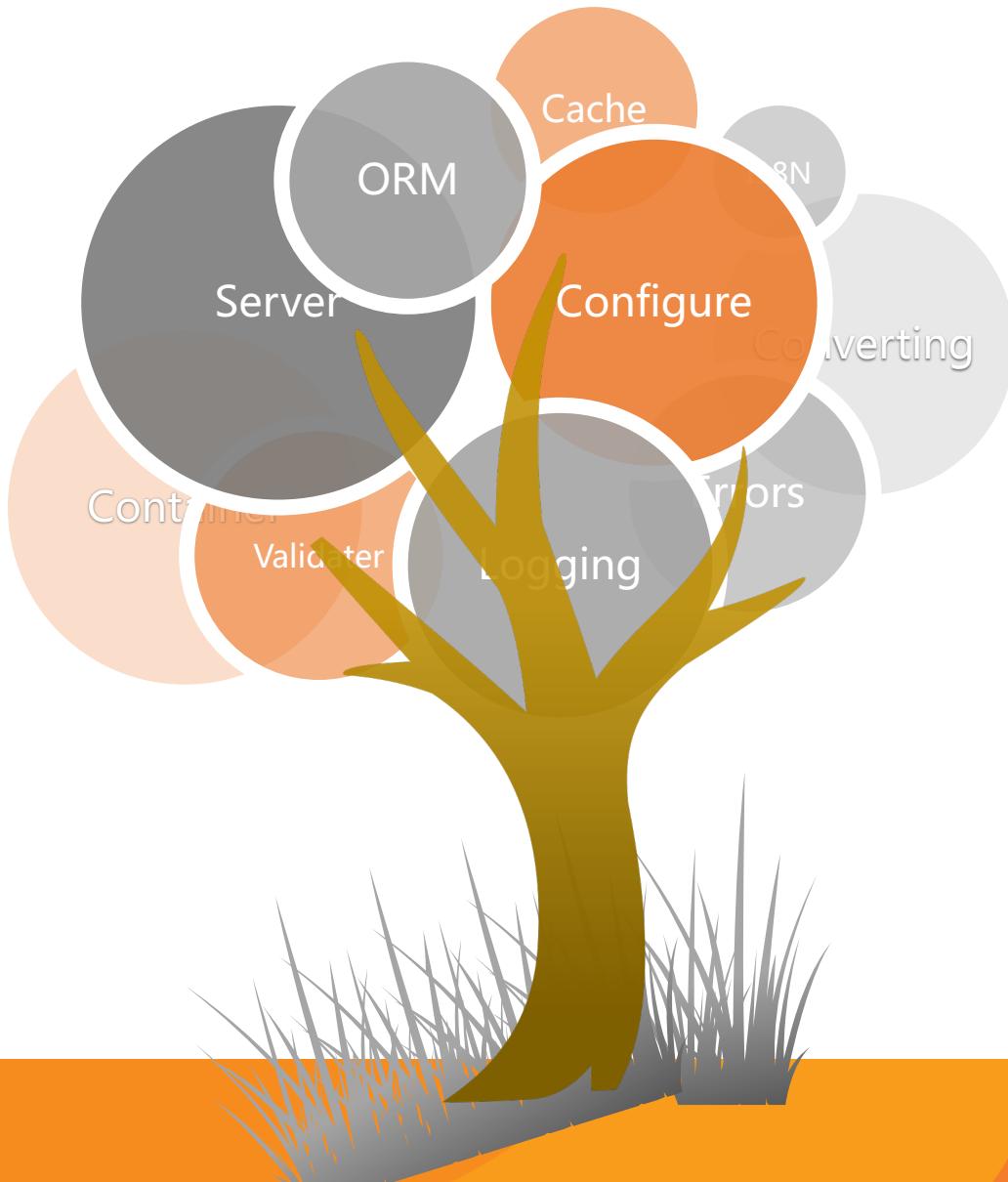
统一框架设计

- 技术体系化
- 开发规范化
- 组件统一化
- 形成技术沉淀
- 避免资源浪费

统一框架设计



统一框架设计-技术体系化



这里的体系化是指微观层面的代码开发框架自顶向下统一设计，使得整个框架的设计思想是一体的，而不是分散的。

体系化更关注的是框架整体战斗力，而不是每个模块本身。

自顶向下设计，保证框架各个组件高效的组织协调性，避免重复功能逻辑，提高执行效率和易用性。

自顶向下设计，保证框架各个组件的设计思想及风格高度一致，使得开发者能够快速学习和认知框架行为，降低接入成本、提高维护效率。

统一框架设计-开发规范化



统一的框架设计，将会使得所有的业务项目按照统一的代码设计进行编码，形成统一的开发规范。同时，框架的开发工具链也会使得开发规范更容易快速推广和落地实施。

统一框架设计-组件统一化



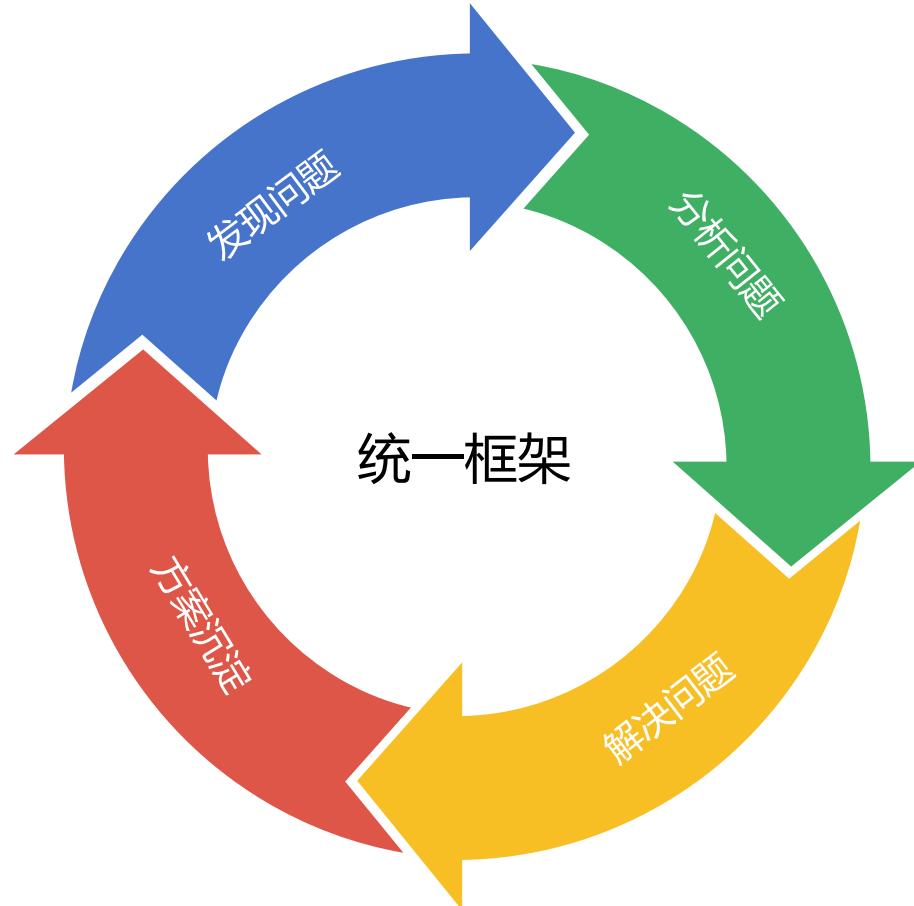
痛点：

- 实现相同功能逻辑的模块百花齐放，选择成本增加
- 项目依赖的模块过多，项目整体的稳定性会受到影响
- 项目依赖的模块过多，项目无从下手是否应当升级这些模块版本
- 各个模块孤立设计，单独看待每个模块可替换性很高，开发体系难以建立，开发规范难以统一

改进：

- 框架核心维护较全面的通用基础模块，降低基础模块选择成本
- 我们只需要维护一个统一的框架版本，而不是数十个模块版本
- 我们只需要了解一个框架的内容变化，而不是数十个模块的内容变化
- 升级的时候只需要升级一个框架版本，而不是数十个模块版本的升级
- 统一的模块化设计可以减少不必要的逻辑实现，提高模块性能及易用性
- 减轻开发人员的心智负担，提高模块可维护性，更容易保证各业务项目的模块版本一致性

统一框架设计-形成技术沉淀



基于统一的开发框架，更容易形成技术沉淀，企业与社区形成良性循环。

统一框架设计-避免资源浪费



当每个团队都在试图自己创造轮子时，不仅无法形成统一的开发规范，而且会出现非常多的资源浪费。

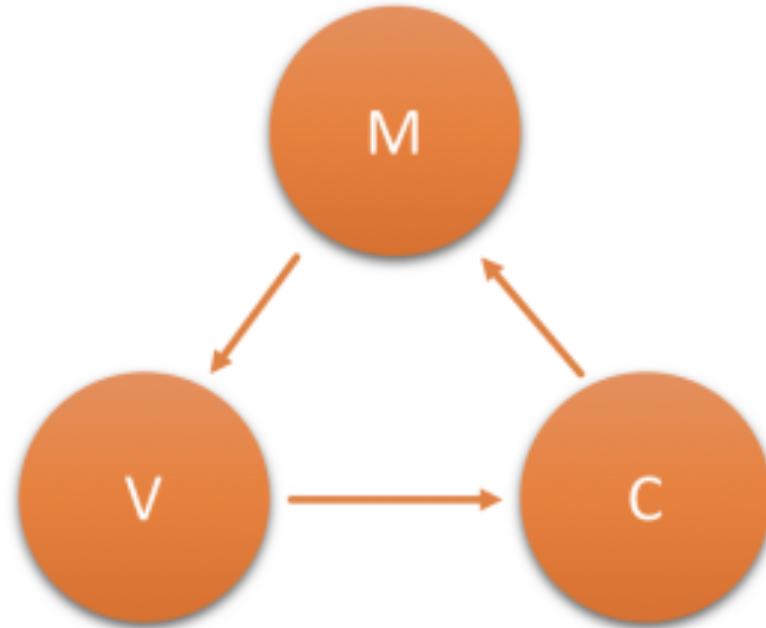
让项目组把精力更多的投入到业务中，相信这是大多数技术公司的共识。使用统一的开发架构，可以把共性的技术问题提炼出来，并形成通用的解决方案。避免每个项目都独自去解决遇到的各种各样的技术难题，有效的把精力释放出来。

第四部分

代码分层设计

- 经典MVC
- 三层架构模式
- 项目代码结构

代码分层设计-经典MVC

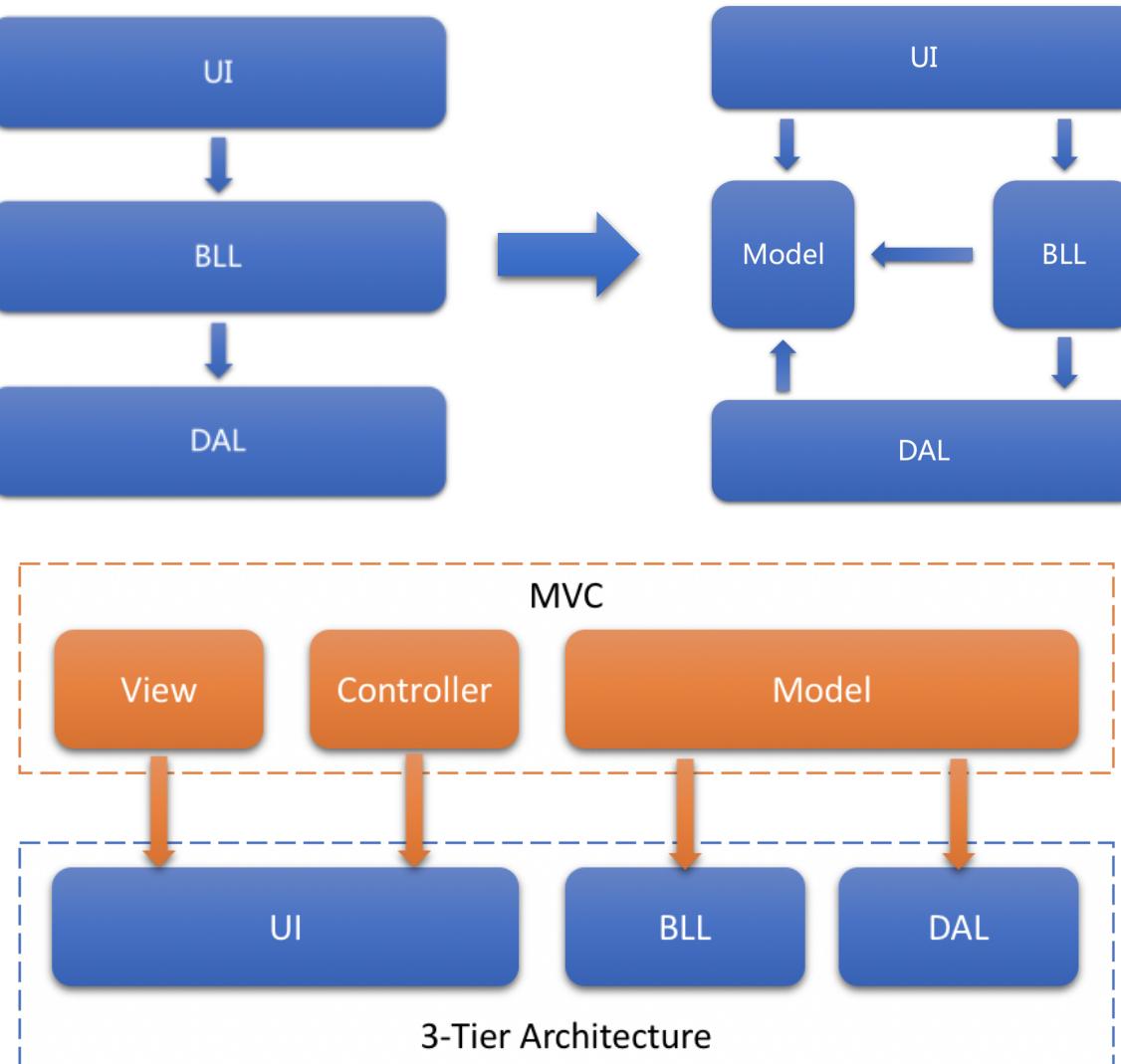


经典MVC模式

痛点：

- 视图展示与数据操作方式的进一步剥离，特别是移动端的发展，前端MVVM框架的发展，我们大多数场景下已不再需要服务端渲染View
- Model层级的代码既维护着数据，也封装着业务逻辑，随着业务逻辑变得越来越复杂，这一层功能逻辑会变得越来越臃肿不易维护
- Controller和Model的职责边界模糊，对于开发人员写好代码的要求会比较高

代码分层设计-三层架构模式



表示层 - UI

位于三层构架的最上层，与用户直接接触，主要是B/S中的 WEB页面，也可以是API接口。

业务逻辑层 - BLL

对具体问题进行逻辑判断与执行操作。

数据访问层 - DAL

实现数据的增删改查等操作，并将操作结果反馈到业务逻辑层 BLL。

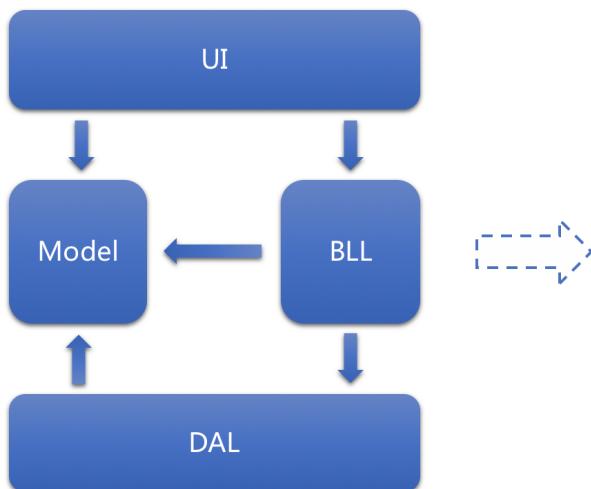
模型定义层 - Model

也常用Entity实体对象来表示，主要用于数据库表的映射对象。

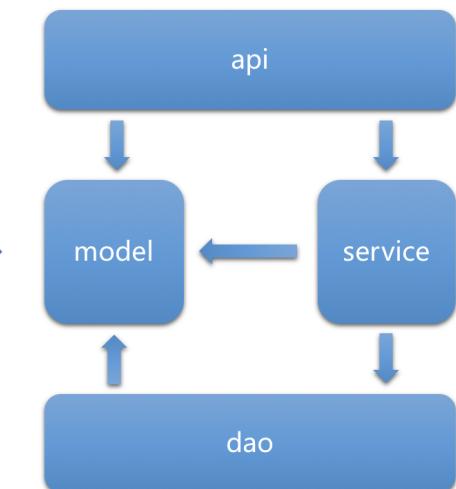
代码分层设计-项目代码结构

```
/  
└── app  
    ├── api  
    ├── dao  
    ├── model  
    └── service  
└── boot  
└── config  
└── docker  
└── document  
└── i18n  
└── library  
└── packed  
└── public  
└── router  
└── template  
└── Dockerfile  
└── go.mod  
└── main.go
```

三层架构设计



框架代码分层



目录/文件名称	说明	描述
app	业务逻辑层	所有的业务逻辑存放目录。
- api	业务接口	接收/解析用户输入参数的入口/接口层。
- dao	数据访问	数据库的访问操作，仅包含最基础的数据库 CURD 方法
- model	结构模型	数据结构管理模块，管理数据实体对象，以及输入与输出数据结构定义。
- service	逻辑封装	业务逻辑封装管理，特定的业务逻辑实现和封装。
boot	初始化包	用于项目初始化参数设置，往往作为 <code>main.go</code> 中第一个被 <code>import</code> 的包。
config	配置管理	所有的配置文件存放目录。
docker	镜像文件	Docker 镜像相关依赖文件，脚本文件等等。
document	项目文档	Documentation项目文档，如：设计文档、帮助文档等等。
i18n	I18N国际化	I18N国际化配置文件目录。
library	公共库包	公共的功能封装包，往往不包含业务需求实现。
packed	打包目录	将资源文件打包的 Go 文件存放在里面， <code>boot</code> 包初始化时会自动调用。
public	静态目录	仅有该目录下的文件才能对外提供静态服务访问。
router	路由注册	用于路由统一的注册管理。
template	模板文件	MVC 模板文件存放的目录。
Dockerfile	镜像描述	云原生时代用于编译生成Docker镜像的描述文件。
go.mod	依赖管理	使用 Go Module 包管理的依赖描述文件。
main.go	入口文件	程序入口文件。

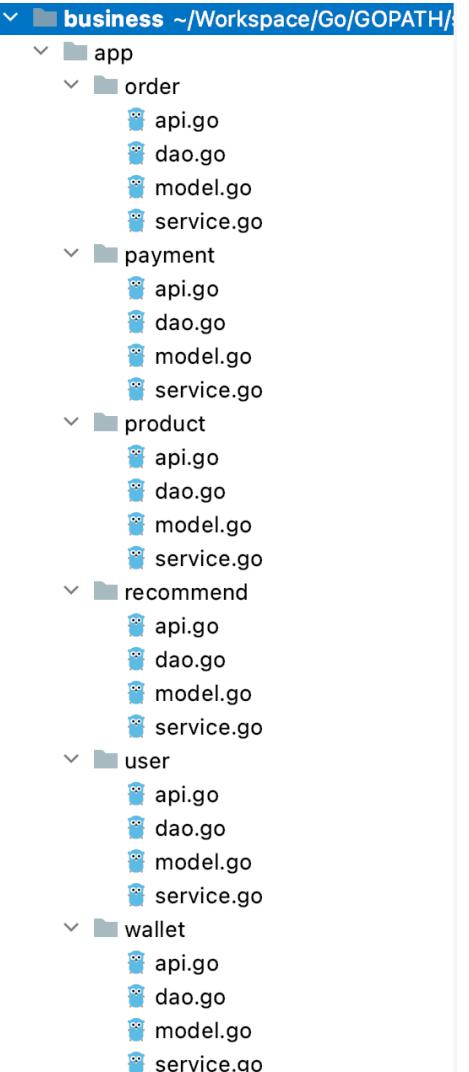
第五部分

对象封装设计

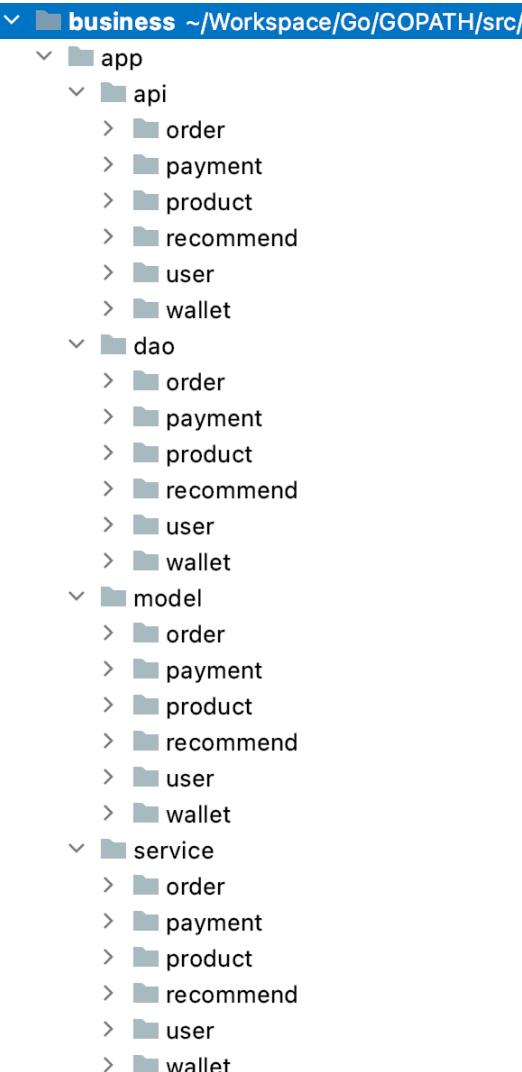
- 包与对象封装
- 资源明规范
- 对象封装示例
- 对象访问安全

对象封装设计-包与对象封装

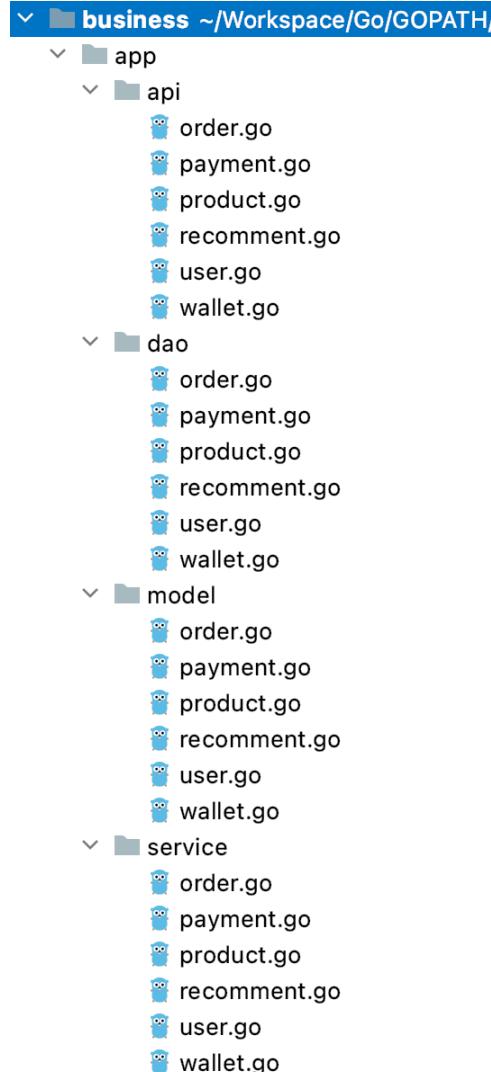
常见项目架构示例1



常见项目架构示例2



对象封装改进



痛点：

- 没有代码分层设计，代码耦合高
- 单包职责过大，维护成本高
- 包命名困难，易出现重复包名，使用困惑，开发效率低
- 包管理困难，容易出现 cycle import 循环引用问题

改进：

- 代码分层设计，降低代码耦合
- 功能逻辑按照对象封装，细化封装粒度，降低单包职责
- 规避了同名包名的问题
- 规避了 cycle import 问题

对象封装设计-资源命名规范

```
1 package api
2
3 import (
4     "focus/app/model"
5     "focus/app/system/index/internal/define"
6     "focus/app/system/index/internal/service"
7     "github.com/gogf/gf/net/ghttp"
8 )
9
10 var Ask = askApi{}
11
12 type askApi struct{}
13
14 // @summary 展示问答首页
15 // @tags 前台-问答
16 // @produce html
17 // @param cate query int    false "栏目ID"
18 // @param page query int    false "分页号码"
19 // @param size query int    false "分页数量"
20 // @param sort query string false "排序方式"
21 // @router /ask [GET]
22 // @success 200 {string} html "页面HTML"
23 func (a *askApi) Index(r *ghttp.Request) {
24     var (
25         data *define.ContentServiceGetListReq
26     )
27     if err := r.Parse(&data); err != nil {
28         service.View.Render500(r, model.View{
```

在三层架构设计模式下，我们的业务包命名仅会有 api、 dao、 model、 service四个包。

每个业务包仅对外暴露 **实例化的对象** 用于该业务领域的具体功能逻辑封装，同一层级下不同的业务领域逻辑通过不同文件来分别管理。

包对外的公开对象采用 **业务领域名称** 来命名，包内部的数据结构定义采用 **业务领域名称+分层名称** 来命名。

对象封装设计-对象封装示例1

api访问service层对象

```
service/user.go x index.go x article.go x middleware.go x ask.go x interact.go x view_buildin.go x version.go x
23 func (a *articleApi) Index(r *ghttp.Request) {
24     var (
25         data *define.ContentServiceGetListReq
26     )
27     if err := r.Parse(&data); err != nil {
28         service.View.Render500(r, model.View{
29             Error: err.Error(),
30         })
31     }
32     data.Type = model.ContentTypeArticle
33     service.
34     if ge v User
35         s v Tag
36         v Middleware
37     } else {
38         s v Content
39             v File
40             v Menu
41             v Reply
42             v Session
43             v Setting
44     }
45     ta); err != nil {
46     userSe
47     tagSe
48     middlew
49     interactS
50     captchaS
51     categoryS
52     contentSe
53     fileSe
54     menuSe
55     replySe
56     sessionSe
57     settingSe
58     wServiceGetTitleReq{
```

访问service层对象具体操作

```
23 func (a *articleApi) Index(r *ghttp.Request) {
24     var (
25         data *define.ContentServiceGetListReq
26     )
27     if err := r.Parse(&data); err != nil {
28         service.View.Render500(r, model.View{
29             Error: err.Error(),
30         })
31     }
32     data.Type = model.ContentTypeArticle
33     service.Content.
34     if getListRes m Update(ctx context.Context, r *define.ContentServiceUpdateReq) →
35         service.V m Search(ctx context.Context, r *define.ContentServiceSearchReq) →
36             Error
37         } else {
38             service.V
39                 Content
40                 Data:
41                 dereference
42                 Title
43                 C
44                 C
45                 d
46                 Dereference
47                 nil
48             })
```

对象封装设计-对象封装示例2

路由注册时访问api对象

```
service/user.go x index.go x middleware.go x ask.go x interact.go x content.go x view_buildin.go x version.go
37 // 前台系统路由注册
38 s.Group("/", func(group *ghhttp.RouterGroup) {
39     group.Middleware(service.Middleware.Ctx)
40     group.ALLMap(g.Map{
41         "/":           api.Index,          // 首页
42         "/login":      api.Login,        // 登录
43         "/register":   api.Register,    // 注册
44         "/category":   api.Category,    // 栏目
45         "/topic":      api.Topic,        // 主题
46         "/topic/:id":  api.Topic.Detail, // 主题 - 详情
47         "/ask":        api.Ask,          // 问答
48         "/ask/:id":    api.Ask.Detail,  // 问答 - 详情
49         "/article":    api.Article,     // 文章
50         "/article/:id": api.Article.Detail, // 文章 - 详情
51         "/reply":      api.Reply,        // 回复
52         "/search":    api.Search,       // 搜索
53         "/captcha":   api.Captcha,     // 验证码
54         "/user/:id":   api.,          // 用户 - 主页
55     })
56     // 权限控制路由
57     group.Group("/", func(group *ghhttp.RouterGroup) {
58         group.Middleware(service.Middleware.Ctx)
59         group.ALLMap(g.Map{
60             "/user":        api.User,
61             "/content":    api.Content,
62             "/interact":   api.Interact,
63             "/file":       api.File,
64         })
65     })
66 }
67
68 func Init() {
69     func(group *ghhttp.RouterGroup) {
70         group.Middleware(service.Middleware.Ctx)
71         group.ALLMap(g.Map{
72             "/user":        api.User,
73             "/content":    api.Content,
74             "/interact":   api.Interact,
75             "/file":       api.File,
76             "/login":      api.Login,
77             "/register":   api.Register,
78             "/category":   api.Category,
79             "/topic":      api.Topic,
80             "/topic/:id":  api.Topic.Detail,
81             "/ask":        api.Ask,
82             "/ask/:id":    api.Ask.Detail,
83             "/article":    api.Article,
84             "/article/:id": api.Article.Detail,
85             "/reply":      api.Reply,
86             "/search":    api.Search,
87             "/captcha":   api.Captcha,
88             "/user/:id":   api.User,
89         })
90     })
91 }
```

路由注册具体业务领域对象的方法

```
group.ALLMap(g.Map{
    "/":           api.Index,          // 首页
    "/login":      api.Login,         // 登录
    "/register":   api.Register,      // 注册
    "/category":   api.Category,      // 栏目
    "/topic":       api.Topic,         // 主题
    "/topic/:id":   api.Topic.Detail, // 主题 - 详情
    "/ask":         api.Ask,           // 问答
    "/ask/:id":    api.Ask.Detail,    // 问答 - 详情
    "/article":    api.Article,        // 文章
    "/article/:id": api.Article.Detail, // 文章 - 详情
    "/reply":       api.Reply,          // 回复
    "/search":     api.Search,         // 搜索
    "/captcha":    api.Captcha,        // 验证码
    "/user/:id":   api.User.,         // 用户 - 主页
})
// 权限控制路由
group.Group("/", func(group.Group{
    group.Middleware(service.Middleware)
    group.ALLMap(g.Map{
        "/user":      api.U...
        "/content":   api.C...
        "/interact":  api.I...
        "/file":      api.F...
    })
})
func(group *ghttp.RouterGroup)
```



对象封装设计-对象封装示例3

model数据结构命名

```
content.go x
38 // API执行修改内容
39 type ContentApiDoUpdateReq struct {
40     ContentApiCreateUpdateBase
41     Id uint `v:"min:1#请选择需要修改的内容"` // 修改时ID不能为空
42 }
43 // API执行删除内容
44 type ContentApiDoDeleteReq struct {
45     Id uint `v:"min:1#请选择需要删除的内容"` // 删除时ID不能为空
46 }
47
48 // =====
49 // Service
50 // =====
51 // Service查询列表
52 // Service查询列表结果
53 type ContentServiceGetListReq struct {
54     Type      string // 内容模型
55     CategoryId uint   `p:"cate"`           // 栏目ID
56     Page      int    `d:"1" v:"min:0#分页号码错误"` // 分页号码
57     Size      int    `d:"10" v:"max:50#分页数量最大50条"` // 分页数量，最大50
58     Sort      int    // 排序类型(0:最新, 默认。1:活跃, 2:热度)
59     UserId    uint    // 要查询的用户ID
60 }
61
62 // Service查询列表结果
63 type ContentServiceGetListRes struct {
64     List []*ContentServiceGetListResItem `json:"list"` // 列表
65     Page  int                           `json:"page"` // 分页码

```

service调用dao对象示例

```
category.go x content.go x file.go x reply.go x dao/setting.go x dao/user.go x service/user.go x interact.go x internal/set
35
36 // 执行登录
37 func (s *userService) Login(ctx context.Context, loginReq *define.UserServiceLoginReq) error {
38     userEntity, err := s.GetUserByPassportAndPassword(
39         loginReq.Passport,
40         s.EncryptPassword(loginReq.Passport, loginReq.Password),
41     )
42     if err != nil {
43         return err
44     }
45     if userEntity == nil {
46         return gerror.New("账号或密码错误")
47     }
48     if err := Session.SetUser(ctx, userEntity); err != nil {
49         return err
50     }
51     dao.| / User userDao
52     |s| Category categoryDao
53     |s| Content contentDao
54     |s| File fileDao
55     |s| Interact interactDao
56     |s| Reply replyDao
57     |s| Setting settingDao
58     |s| par (expr)
59     |s| Press ⇧ to insert, → to replace Next Tip
60 }
61
62 *userService.Login(ctx context.Context, loginReq *define.UserServiceLoginReq) error

```



对象封装设计-对象访问安全

各分层中的封装对象都是以“可变变量”的形式对外暴露使用，存在被修改的安全风险。

```
api/user.go x service/user.go x
10 "github.com/gotr/tr/crypto/gmac"
11 "github.com/gof/gf/errors/gerror"
12 "github.com/gof/gf/frame/g"
13 "github.com/gof/gf/os/gfile"
14 "github.com/gof/gf/util/gconv"
15 "github.com/oeglg/govatar"
16 )
17 // 用户管理服务
18 var User = &userService{
19     AvatarUploadPath: g.Cfg().GetString(`upload.path`) + `/avatar`,
20     AvatarUploadUrlPrefix: `/upload/avatar`,
21 }
22 )
23 type userService struct {
24     AvatarUploadPath string // 头像上传路径
25     AvatarUploadUrlPrefix string // 头像上传对应的URL前缀
26 }
27 )
28 func init() {
29     // 启动时创建头像存储目录
30     if !gfile.Exists(User.AvatarUploadPath) {
31         gfile.Mkdir(User.AvatarUploadPath)
32     }
33 }
34 )
35 // 执行登录
36 func (s *userService) Login(ctx context.Context, loginReq *define.UserServiceLoginReq) error {
37 }
```

安全风险：外部可直接修改为 nil
安全风险：外部可直接修改公开成员变量

```
middleware.go x user.go x index.go x article.go x .gitignore x config.toml x reply.go x setting.go x category.go x
18 // 用户管理服务
19 var User = userService{
20     avatarUploadPath: g.Cfg().GetString(`upload.path`) + `/avatar`,
21     avatarUploadUrlPrefix: `/upload/avatar`,
22 }
23
24 type userService struct {
25     avatarUploadPath string // 头像上传路径
26     avatarUploadUrlPrefix string // 头像上传对应的URL前缀
27 }
28
29 func init() {
30     // 启动时创建头像存储目录
31     if !gfile.Exists(User.avatarUploadPath) {
32         gfile.Mkdir(User.avatarUploadPath)
33     }
34 }
35
36 // 获取头像上传路径
37 func (s *userService) GetAvatarUploadPath() string {
38     return s.avatarUploadPath
39 }
40
41 // 获取头像上传URL前缀
42 func (s *userService) GetAvatarUploadUrlPrefix() string {
43     return s.avatarUploadUrlPrefix
44 }
```

普通对象形式，外部无法修改为 nil
内部私有成员变量，外部无法修改
通过公开方法访问私有成员变量

第六部分

DAO封装设计

DAO封装设计-痛点举例

```
func (s *Service) GetDoctorInfo(ctx context.Context, r *api.GetDoctorInfoReq) (res *api.GetDoctorInfoRes, err error) {
    res = &api.GetDoctorInfoRes{
        UserInfo: new(api.UserInfo),
        DoctorInfo: new(api.DoctorInfo),
    }
    var (
        userInfoModel = &model.UserInfo{}
        doctorInfoModel = &model.DoctorInfo{}
    )
    // 查询医生信息
    err = s.daoOrmDB.Table("doctorUser").Where("userId=?", r.Id).First(doctorInfoModel).Error
    if err != nil {
        return
    }
    err = copier.Copy(res.DoctorInfo, doctorInfoModel)
    if err != nil {
        return
    }
    // 查询基本用户信息
    err = s.daoOrmDB.Table("user").Where("id=?").First(userInfoModel).Error
    if err != nil {
        return
    }
    err = copier.Copy(res.UserInfo, userInfoModel)
    return
}
```

需要提前初始化返回对象，无论是否查询到数据
内部必须定义特定 tag 关联表结构与 struct 属性
需要创建中间查询结果对象执行赋值转换
原始 DB 对象使用，非 DAO 设计
SELECT * 操作，没有字段筛选和过滤
太多的字符串硬编码，例如表名和字段名称硬编码
多余的的数据赋值转换，浪费性能
不支持链路跟踪、SQL 日志记录

一个简单的GRPC接口示例

1. 必须定义 tag 关联表结构与 struct 属性
2. 不支持通过返回对象指定需要查询的字段
3. 无法对输入对象属性名称进行自动字段过滤
4. 需要创建中间查询结果对象执行赋值转换
5. 需要提前初始化返回对象，不管有无查询到数据
6. 没有 DAO 对象封装操作
7. 太多的字符串硬编码，例如表名和字段的硬编码
8. 不支持链路跟踪
9. 不支持 SQL 日志输出

DAO封装设计-改进方案

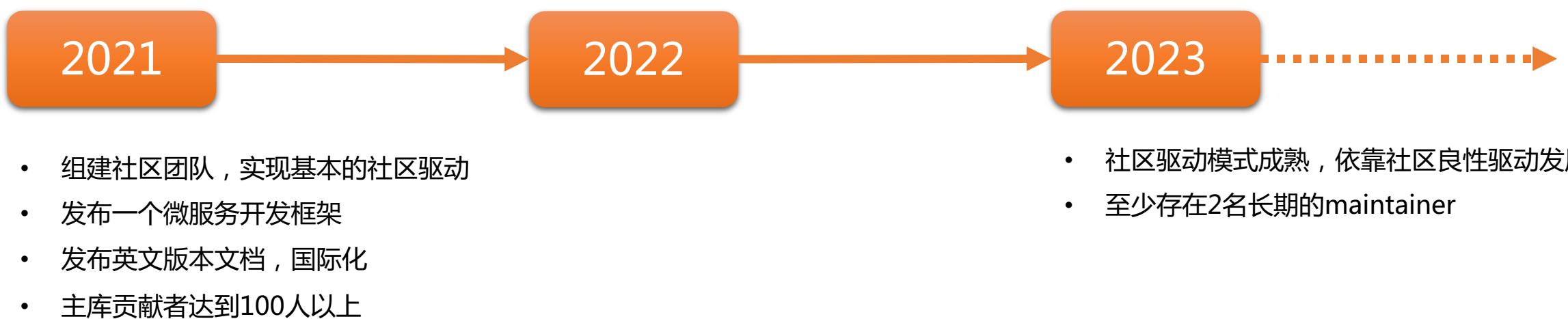
1. 查询结果对象无需特殊标签定义
2. 支持根据指定对象自动识别查询字段，而不是全部SELECT *
3. 支持根据指定对象自动过滤不存在的字段内容
4. 使用DAO对象封装代码设计，通过对象方式操作数据表
5. DAO对象将关联的表名及字段名进行封装，避免字符串硬编码
6. 无需提前定义实体对象接受返回结果，无需创建中间实体对象用于接口返回对象的赋值转换
7. 查询结果对象无需提前初始化，查询到数据时才会自动创建
8. 支持context输入，以便支持链路跟踪
9. 支持SQL日志输出能力，支持开关功能
10. 数据模型、数据操作、业务逻辑解耦，支持Dao及Model代码工具化自动生成，提高开发效率，便于规范落地

```
9 // 查询医生信息
10 func (s *Service) GetDoctorInfo(ctx context.Context, req *api.GetDoctorInfoReq) (res *api.GetDoctorInfoRes, err error) {
11     res = &api.GetDoctorInfoRes{} ← 内部返回对象无需预初始化
12     // 查询医生信息
13     err = dao.DoctorUser.Ctx(ctx).Fields(res.DoctorInfo).Where(dao.DoctorUser.Columns.UserId, req.Id).Scan(&res.DoctorInfo) ← 查询到数据则自动创建对象,
14     if err != nil { ← 否则什么也不做
15         return ← DAO数据对象访问设计
16     }
17     // 查询基础用户信息
18     err = dao.User.Ctx(ctx).Fields(res.DoctorInfo).Where(dao.User.Columns.Id, req.Id).Scan(&res.UserInfo) ← 非字符串硬编码
19     return res, err ← 支持链路跟踪
20 }
21 }
```

第七部分

未来发展规划

未来发展规划



Question?

Thanks!

QQ交流群



微信交流群

