
目录

Introduction	1.1
介绍	1.2
软件架构	1.3
架构说明	1.3.1
技术说明	1.3.2
系统部署	1.4
准备工作	1.4.1
域名	1.4.1.1
云存储	1.4.1.2
阿里云	1.4.1.2.1
七牛云	1.4.1.2.2
微信开放平台第三方平台申请	1.4.1.3
域名备案	1.4.1.4
服务器方案	1.4.1.5
腾讯位置服务	1.4.1.6
小程序	1.4.1.7
EcshopX初始化	1.4.1.8
部署方案	1.4.2
单机部署方案	1.4.2.1
集群部署方案	1.4.2.2
框架	1.5
本地性能调试	1.5.1
文档说明	1.5.2
入门指南	1.5.3
开发环境搭建	1.5.3.1
docker环境	1.5.3.1.1
本地环境	1.5.3.1.2
composer	1.5.3.2
目录结构	1.5.3.3
架构思想	1.5.4
中大型项目架构	1.5.4.1

Repository	1.5.4.2
Service	1.5.4.3
Bundle	1.5.5
请求	1.5.5.1
路由	1.5.5.1.1
控制器	1.5.5.1.2
数据库	1.5.5.2
快速入门	1.5.5.2.1
Repository	1.5.5.2.2
CRUD Repository	1.5.5.2.3
事件系统	1.5.5.3
消息队列	1.5.5.4
规范	1.5.6
git提交规范	1.5.6.1
说明	1.5.6.2
关于「能愿动词」的使用	1.5.6.3
代码规范	1.5.6.4
基本代码规范	1.5.6.4.1
编码风格规范	1.5.6.4.2
Bundle 规范	1.5.6.5
命名规范	1.5.6.5.1
调试	1.5.7
环境变量及配置	1.5.8
测试	1.5.9
快速入门	1.6
运维	1.7
压力测试报告	1.7.1
5机集群	1.7.1.1
API文档	1.8
数据库migration规则	1.9
FAQ	1.10

ecshopex开放文档

介绍

EcshopX实现线上线下融合, 重构消费链接. 通过引入社交玩法突破获客与转化瓶颈, 形成导购与顾客销售的互动,促进线下实体的商业数字化转型. 帮客户重构销售网络.

架构说明

软件架构

后端开发语言：PHP 7.2

后端技术框架：采用 Lumen 开源框架开发，引入dingo作为接口管理，采用更符合大型项目的 Doctrine ORM 替换系统自带 ORM

前端开发语言：HTML5/CSS3/ES6/小程序开发框架

前端开发框架：VueJS,Taro,Wepy

数据库：支持 MySQL/MariaDB/SqlServer等数据库

队列：支持 Redis/RabbitMQ/数据库 等方案，默认采用 Redis

缓存：支持 Redis/filesystem/memcached，默认采用 Redis

CDN/OSS：支持阿里云/七牛云

图形数据库：Neo4j

搜索：ELK

日志搜集：ELK/Sentry

数据处理能力

支持 MySQL 读写分离模式，支持阿里云PolarDB。

在 5 机集群下支持 2000 每秒并发，其中订单占比 1%，且根据实际业务需求扩容

采用多级缓存机制，可以有效保证API响应时间在 2S 以内

系统具备按照终端，店铺，门店等权限控制

稳定性

系统支持集群模式部署，可按照 web/schedule/worker 等不同角色部署，可有效保证系统高可用

支持阿里云 K8S 部署，可以通过HPA实现 POD 自动扩容，可有效保证在突发流量下，系统的高可用

采用事件机制，可以异步处理耗时业务逻辑，保证实时业务稳定性

采用多级缓存机制，nginx+memcache/php+redis 方式，在redis数据库宕机是，可由 nginx 缓存来替代，保证系统高可用。

持续集成

系统采用 PHP 开发，支持二次开发

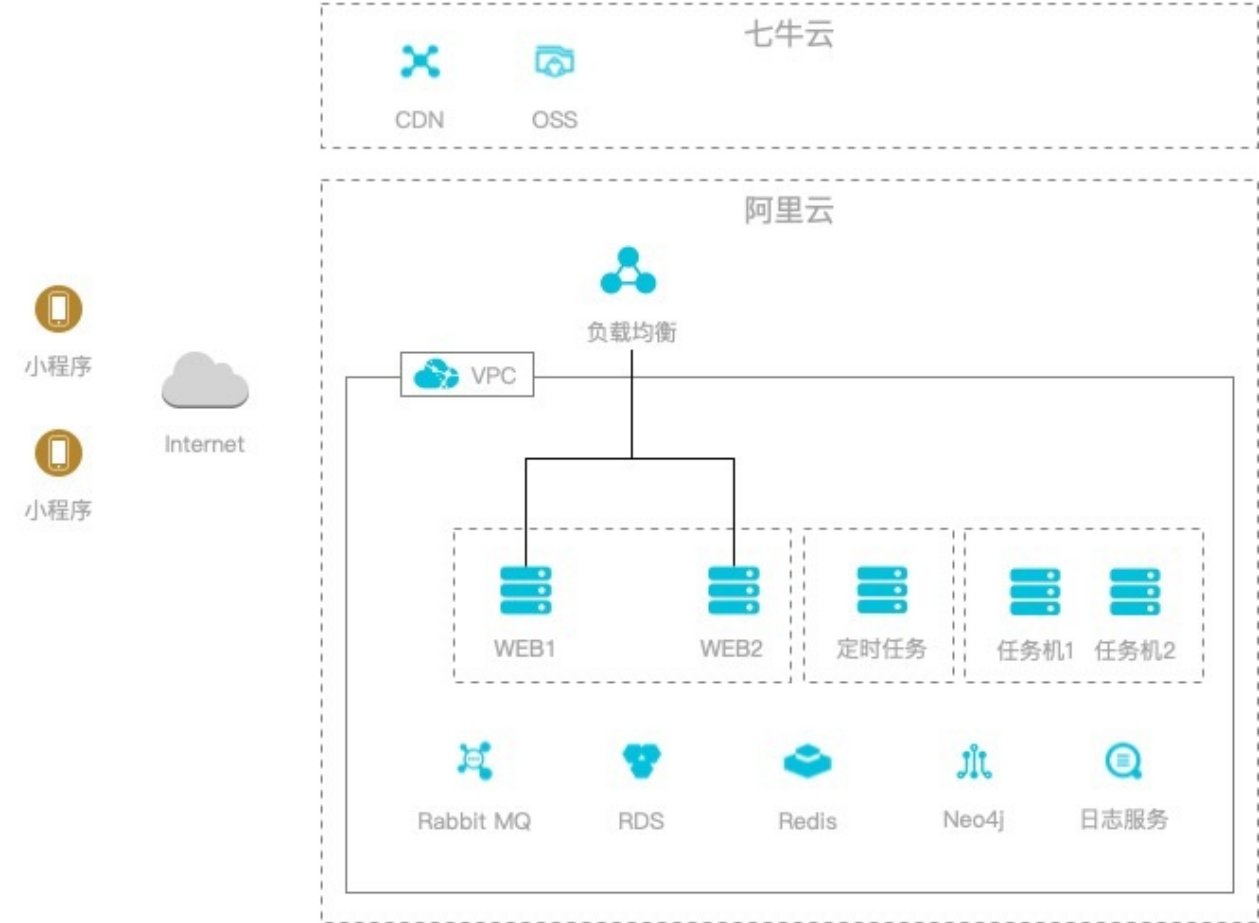
在 ONex中，除了授权部分，其他部分均为源代码

具备完善的开发环境（docker-compose）和测试环境构建脚本，可以快速搭建环境

采用基于 GitLab-CI/K8S/Helm/Helmfile 的持续集成方案

技术说明

整体方案



名称	后端描述
后端开发语言	php
后端开发框架	lumen/doctrine/dingo
前端开发语言	HTML5/CSS3/ES6/小程序开发框架
前端开发框架	VueJS,Taro,Wepy
数据库	mysql
队列	redis/rabbitmq
缓存	redis/cache
cdn/oss	七牛云
图数据库	neo4j
搜索	ELK
日志搜集	Setntry/ELK
持续集成	Git, Gitlib Ci

OSS&CDN方案

支持方案: 七牛, 微信

- 前后端分离
系统前端生成多的css, js
- 图片, 视频

前端技术方案

各端

PC

- vue-router/vuex
- vue+iview组件封装

移动端h5

h5端使用Taro

Taro 是一套遵循 React 语法规范的 多端开发 解决方案。现如今市面上端的形态多种多样，Web、React-Native、微信小程序等各种端大行其道，当业务要求同时在不同的端都要求有所表现的时候，针对不同的端去编写多套代码的成本显然非常高，这时候只编写一套代码就能

够适配到多端的能力就显得极为需要。

使用 Taro，我们可以只书写一套代码，再通过 Taro 的编译工具，将源代码分别编译出可以在不同端（微信/百度/支付宝/字节跳动小程序、H5、React-Native 等）运行的代码。

[Taro官网](#)

微信小程序

- Taro
- Wepy

混合应用

- h5页面内嵌, 与原生应用双向通讯

工具链

- webpack 资源打包/开发/发布
- 基于vue-cli 3的开发框架
 - 命令行工具spx-cli, 用于安装/发布模块包
 - 基于vue-cli 3的插件, 生成样板代码 基础配置 注册模块等

组件库

- 基于vue的pc端基础组件库
- 基于iview的业务组件

前后端协作

- yapi管理接口及文档 mock数据
- spa应用, 通过yapi约定接口, 分别开发

后端技术方案

语言

```
php 7.2
```

框架

前后端分离, 后端脱离传统系统的后端页面渲染. 专心只做API服务端, 因此在技术选型上选择了, 轻量级的lumen框架.

- Lumen
为速度而生的 Laravel 框架. Laravel是一套简洁、优雅的PHP Web开发框架(PHP Web Framework)。它可以让你从面条一样杂乱的代码中解脱出来；它可以帮你构建一个完美的网络APP，而且每行代码都可以简洁、富于表达力。
[laravel官网](#)
[lumen官网](#)
- Dingo
- doctrine orm
 - 支持 master slave
- 开发模式
 - service repository entity

工具链

- 包管理 - composer
- 异常处理 - Sentry
- API - swagger
- doctrine

使用服务

- mysql
- redis
- neo4j
- 队列服务
 - rabbitmq
 - redis
- cdn+oss存储, 目前只支持 七牛云

部署方案

传统部署方案

kubernetes方案

目前支持阿里云 华为云

技术栈

- kubernetes
- docker
- helm
- helmfile
- 持续集成: gitlab-ci

域名

SSL证书

ssl证书, 对网站传输的数据可以进行SSL加密, 包括用户名, 密码等等, 防止窃取和篡改。并且网站安装SSL证书后, 在浏览器地址栏可以显示一个安全锁, 告诉用户你的网站是安全的。SSL证书是数字证书的一种, 类似于驾驶证、护照和营业执照的电子副本。因为配置在服务器上, 也称为SSL服务器证书。

******源源客需要购买通配型ssl证书(泛域名型ssl证书)。

域名规划域名规划

为了表述方便, 以域名**b.yuanyuanke.cn**做例子。

- [https://b.yuanyuanke.cn api/](https://b.yuanyuanke.cn/api/)商家后台域名
- <https://h5.yuanyuanke.cn> h5访问域名
- <https://pc.yuanyuanke.cn> pc访问域名
- <https://b-import-cdn.yuanyuanke.cn> 上传文件cdn域名
- <https://b-cdn.yuanyuanke.cn> 后台前端的相关 css js 资源cdn域名
- <https://b-img-cdn.yuanyuanke.cn> 图片cdn域名

FAQ

- 为什么必须要使用SSL证书
其一SSL证书作为电商领域保证客户信息安全的重要手段. 其二微信小程序 公众号 苹果商店等对于安全性的要求越来越高, 微信小程序是必须SSL才可以, 其它场景也陆续增加限制. 其三, C端用户对于安全的意识也越来越强, 有些客户对于没有小锁头的电商网站有抵触.
- 是否可以选择不收费的SSL?
尽量避免使用免费的SSL, 免费的SSL通常不支持**通配符型SSL证书/泛域名SSL证书**, 本文档相关部署以支持通配符型SSL证书为准.
- 对于收费的SSL证书应该如何选择?
- 云盾证书服务
- SSL去哪里买?
主流的云供应商都有, 阿里的云盾

云存储

概念

cdn

CDN的全称是Content Delivery Network，即内容分发网络。CDN是构建在网络之上的内容分发网络，依靠部署在各地的边缘服务器，通过中心平台的负载均衡、内容分发、调度等功能模块，使用户就近获取所需内容，降低网络拥塞，提高用户访问响应速度和命中率。CDN的关键技术主要有内容存储和分发技术。

开放存储服务 OSS

开放存储服务（OpenStorageService，简称OSS），是云对外提供的海量，安全，低成本，高可靠的云存储服务。用户可以通过简单的API（REST方式的接口），在任何时间、任何地点、任何互联网设备上进行数据上传和下载 保存js, ccs, 图片等静态资源

存储设计

在 ecshopx 中，抽象了三种存储方式：图片、视频、文件。在使用云存储时，需要建立三个不同的存储空间，详细设置如下：

espier-vue

用来存储前端的 css js 及静态图片等

- 存储空间(bucket): espier-vue
- CDN 域名: <https://b-cdn.yuanyuanke.cn>
- 访问控制: 公开空间

espier-import-files

用来存储系统导入 导出的相关数据

- 存储空间(bucket): espier-import-files
- CDN 域名: <https://b-import-cdn.yuanyuanke.cn>
- 访问控制: 私有空间

espier-image

用来存储商品图片

- 存储空间(bucket): espier-images
- CDN 加速域名: <https://b-img-cdn.yuanyuanke.cn>
- 访问控制: 公开空间

支持的云存储供应商

阿里云

存储空间配置, 参见 [阿里云 oss 配置说明](#)

在创建完成存储空间后, 需要在 `.env` 中配置详细参数, 实例如下:

```
DISK_DRIVER=oss

OSS_ACCESS_KEY=your KEY
OSS_SECRET_KEY=your SECRET

OSS_FILE_ENDPOINT=https://oss-cn-hangzhou.aliyuncs.com
OSS_FILE_BUCKET=espier-import-files
OSS_FILE_IS_CNAME=false

OSS_IMAGE_ENDPOINT=https://oss-cn-hangzhou.aliyuncs.com
OSS_IMAGE_BUCKET=espier-image
OSS_IMAGE_IS_CNAME=false
OSS_IMAGE_DOMAIN=https://b-img-cdn.yuanyuanke.cn

OSS_VIDEO_ENDPOINT=https://oss-cn-shanghai.aliyuncs.com
OSS_VIDEO_BUCKET=espier-image
OSS_VIDEO_IS_CNAME=false
OSS_VIDEO_DOMAIN=https://b-img-cdn.yuanyuanke.cn
```

如何获取 [阿里云 Access Key](#) 和 [Secret Key](#)

七牛云

具体配置, 参见 [七牛云 配置说明](#)

在创建完成存储空间后, 需要在 `.env` 中配置详细参数, 实例如下:

```
DISK_DRIVER=qiniu

QINIU_ACCESS_KEY=your KEY
QINIU_SECRET_KEY=your SECRET

QINIU_IMAGE_DOMAIN=b-img-cdn.yuanyuanke.cn
QINIU_IMAGE_NAME=espier-image
QINIU_IMAGE_REGION=z2

QINIU_VIDEO_DOMAIN=b-video-cdn.yuanyuanke.cn
QINIU_VIDEO_NAME=espier-videos
QINIU_VIDEO_REGION=z2

QINIU__FILE_DOMAIN=https://b-import-cdn.yuanyuanke.cn
QINIU_FILE_NAME=espier-file
QINIU_FILE_REGION=z2
```

[怎么获取或者找到 七牛 Access Key 和 Secret Key](#)

阿里云

进入 <https://oss.console.aliyun.com/bucket> 创建bucket。

创建 espier-image 和 espier-vue（公共读）

一、创建 Bucket



二、填写详细信息

创建 Bucket

创建存储空间

注意：Bucket 创建成功后，您所选择的 存储类型、区域 不支持变更。

Bucket 名称

espier-image

Bucket 名称

12/63

区域

华东2（上海）

根据需求选择区域

Endpoint

oss-cn-shanghai.aliyuncs.com

存储类型

标准存储

低频访问

归档存储

标准：高可靠、高可用、高性能，数据会经常被访问到。

如何选择适合您的存储类型？

同城冗余存储

启用

关闭

OSS 将您的数据以冗余的方式存储在单一区域（Region）的 3 个可用区（Zone）中。提供机房级容灾能力。更多详情请参见 同城冗余存储。

同城冗余存储能提高您的数据可用性，同时会采用相对较高的计费标准。请查看 价格详情。同城冗余存储属性开启后，将不支持关闭。

版本控制

开通

不开通

开启版本控制特性后，针对数据的覆盖和删除操作将会以历史版本的形式保存下来。了解 版本控制。

读写权限

私有

公共读

公共读写

公共读：对文件写操作需要进行身份验证；可以对文件进行匿名读。

确定

取消

三、进入详情页面配置

17

espier-image

读写权限 公共读 类型 标准存储（本地冗余） 区域 华东2（上海） 创建时间 2020年3月17日 18:35

基础数据

基础数据统计平均延迟 1-2 小时。不作为计量数据，仅作参考。

存储用量	总用量（不含 ECS 快照）	本月流量	外网流出流量	本月请求次数	读请求	文件数量	文件碎片
0 Byte		0 Byte		0		0	0
月同比 - 日环比 -		上月外网流出流量: 0 Byte		上月请求次数: 0			

访问域名

访问域名	EndPoint（地域节点）	Bucket 域名	HTTPS
外网访问	oss-cn-shanghai.aliyuncs.com	espier-image.oss-cn-shanghai.aliyuncs.com	支持
ECS 的经典网络访问（内网）	oss-cn-shanghai-internal.aliyuncs.com	espier-image.oss-cn-shanghai-internal.aliyuncs.com	支持
ECS 的 VPC 网络访问（内网）	oss-cn-shanghai-internal.aliyuncs.com	espier-image.oss-cn-shanghai-internal.aliyuncs.com	支持
传输加速域名（全地域上传下载加速）	未开启	开启	支持

基础设置

读写权限	公共读	服务器端加密	无
跨域访问	未开启	防盗链	未开启
镜像回源	未开启	生命周期	未开启

将此信息给到运维人员

进入，添加一条跨域规则

四、创建跨域规则

对象存储 / espier-image / 权限管理 / 跨域访问

espier-image

创建规则 清除全部规则 刷新

来源 允许 Methods 允许 Headers

创建跨域规则

来源 *

允许 Methods * ☐ GET ☒ POST ☐ PUT ☐ DELETE ☐ HEAD

允许 Headers

暴露 Headers

缓存时间（秒） 0

返回 Vary: Origin ☐

确定 取消

创建完成如下：



五、配置 CDN

进入 cdn 管理界面 <https://cdn.console.aliyun.com/domain/list>

← 添加域名

加速域名 * 此处配置自己的 cdn 域名

支持添加泛域名如 "*.test.com"。了解更多

资源分组

业务类型

源站信息 *

☒ OSS域名

☐ IP

☐ 源站域名

☐ 函数计算域名

域名

OSS作为源站，可为您节省更多回源流量费用。

端口 *

☐ 80端口

☒ 443端口

加速区域 *

☒ 仅中国大陆

☐ 全球

☐ 全球（不包含中国大陆）

i 加速区域含中国大陆时，加速域名必须备案。什么是ICP备案?

不同加速区域价格有差别，请按您的实际需求选择。 [价格详情](#)

六、域名解析

CDN添加完成后，会在域名管理界面生成相应的CNAME，然后在域名解析后台添加相应的解析

CDN / 域名管理

域名管理

添加域名

全部业务类型 ▼

选择标签 ▼

请输入

☐	域名	CNAME ?	状态
☐	b-cdn.esign.cn	 b-cdn.esign.cn.w.kunlunca.com	 正常运行
☐	b-img-cdn.esign.cn	 b-img-cdn.esign.cn.w.kunlung.com	 正常运行

创建 espier-import-files （私有）

一、创建

创建 Bucket

创建存储空间

注意：Bucket 创建成功后，您所选择的 **存储类型**、**区域** 不支持变更。

Bucket 名称

espier-import-files19/63

区域

华北2（北京）

相同区域内的产品内网可以互通；订购后不支持更换区域，请谨慎选择。

您在该区域下没有可用的 **存储包**、**流量包**。建议您购买资源包享受更多优惠，[点击 购买](#)。

Endpoint

oss-cn-beijing.aliyuncs.com

存储类型

标准存储

低频访问

归档存储

标准：高可靠、高可用、高性能，数据会经常被访问到。

[如何选择适合您的存储类型？](#)

同城冗余存储

Hot

启用

关闭

OSS 将您的数据以冗余的方式存储在同一区域（Region）的 3 个可用区（Zone）中。提供机房级容灾能力。更多详情请参见 [同城冗余存储](#)。

同城冗余存储能提高您的数据可用性，同时会采用相对较高的计费标准。请查看 [价格详情](#)。同城冗余存储属性开启后，将不支持关闭。

版本控制

Hot

开通

不开通

开启版本控制特性后，针对数据的覆盖和删除操作将会以历史版本的形式保存下来。了解 [版本控制](#)。

读写权限

私有

公共读

公共读写

私有：对文件的所有访问操作需要进行身份验证。

确定

取消

这里选择私有

私有存储不需要配置 cdn 和跨域

七牛云

进入 <https://portal.qiniu.com/kodo/bucket> 创建 bucket

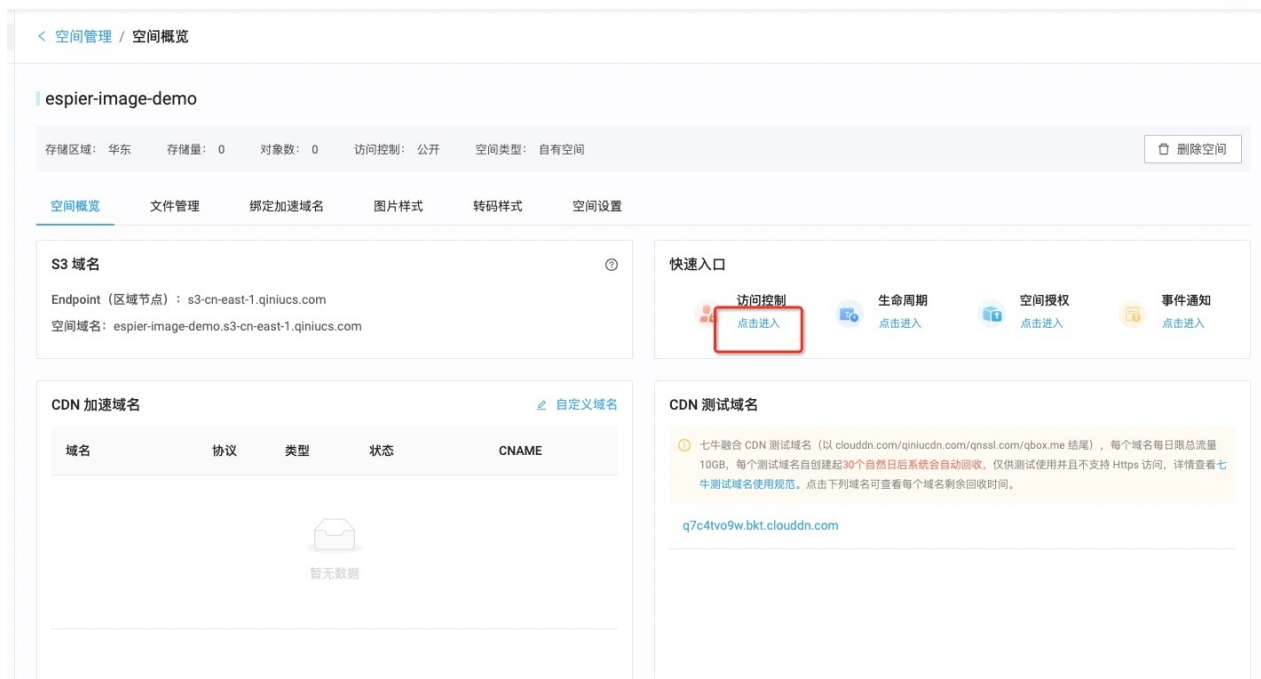
创建 espier-image 和 espier-vue（公共读）

一、创建 bucket



二、设置跨域

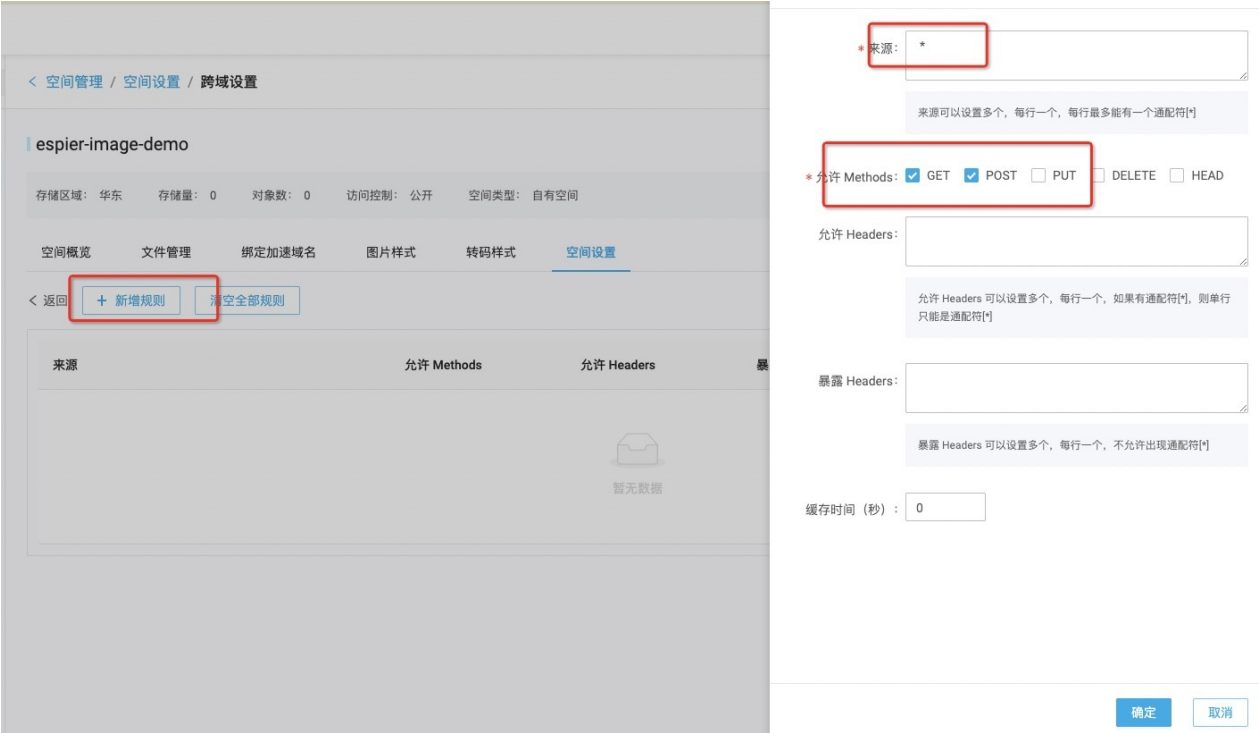
进入详情：



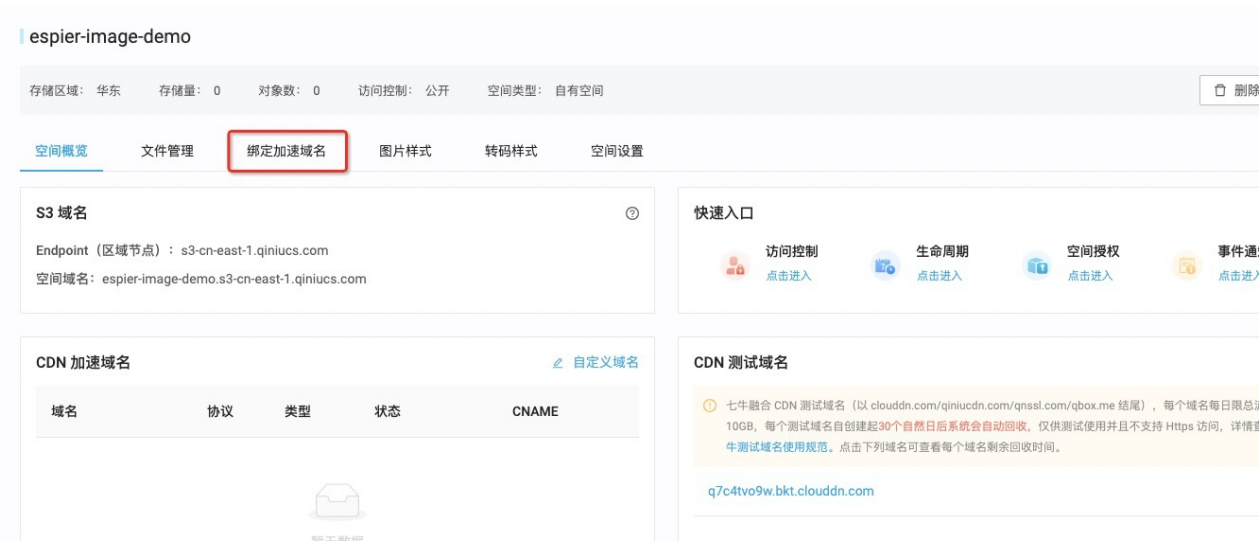
点击访问控制，下拉找到跨域设置



设置规则



三、绑定域名





创建 espiers-import-files （私有）

一、创建

新建存储空间

X

* 存储空间名称：

espier-import-files-demo

存储空间名称不允许重复，遇到冲突请更换名称。
名称格式为 3 ~ 63 个字符，可以包含小写字母、数字、短划线，且必须以小写字母或者数字开头和结尾

* 存储区域：☒ 华东 ☐ 华北 ☐ 华南 ☐ 北美 ☐ 东南亚

空间一旦创建，区域无法修改，请谨慎选择。

* 访问控制：☐ 公开 ☒ 私有

公开和私有仅对 Bucket 的读文件生效，修改、删除、写入等对 Bucket 的操作均需要拥有者的授权才能进行操作。

确定

取消

私有存储不需要配置 cdn 和跨域

微信开放平台第三方平台申请

注册微信平台账号

[注册微信平台注册](#)

开发者资质认证

[第三方平台申请流程](#)



点击 [开发者资质认证](#)，并进行认证.

开发者资质认证介绍 微信开放平台帐号的开发者资质认证提供更安全、更严格的真实性认证、也能够更好的保护企业及用户的合法权益

开发者资质认证通过后，微信开放平台帐号下的应用，将获得微信登录、智能接口、第三方平台开发等高级能力

审核费用：中国大陆地区：300元，非中国大陆地区：99美元

认证时间大概2-3天, 已微信实际认证时间为准

创建第三方平台

管理中心->第三方平台

微信 · 开放平台

首页 资源中心 管理中心 数据中心 帐号中心

移动应用 网站应用 公众帐号 小程序 [第三方平台](#)

创建第三方平台

定制化开发服务商：上限5个，还可创建5个
平台型服务商：上限5个，还可创建4个

第三方平台	服务商类型	状态
-------	-------	----

点击 [创建第三方平台](#)

输入基本信息

平台类型选择 平台型服务商

管理中心 / 创建第三方平台

1

输入基本信
息

2

选择权限

3

填写开发资
料

4

等待审核

1

基本信息将显示在授权页被用户看到，请认真填写

平台类型

定制化开发服务商

平台型服务商

可以一键授权给第三方平台，通过第三方平台来完成业务的服务商，[查看详情](#)。

平台名称

需在2-20个字符之间（应尽量简短），不允许以“微信”开头，不允许包含“测试”和“test”等文案，请通过[测试白名单](#)进行测试

业务标签

+ 添加标签

平台简介

平台简介将显示在平台详情页，需在4到80个字符之间

官方网站

请确保用户可以在官方网站了解你的业务和功能，请勿输入http等协议头

平台图标

请上传高清图片作为平台图标
108*108像素，仅支持PNG格式，大小不超过300KB。

选择文件

参考示例
(不允许直接作为平台图标)

已阅读并同意 [《微信开放平台公众号平台开发协议》](#)

下一步

选择权限

公众号权限

27

- ☒ 公众号权限
如有请选择

- 28

☐ 好物圈权限

帮助公众号将物品、订单、收藏等信息同步至好物圈，方便用户进行推荐 [详情](#)

☐ 微信一物一码权限

帮助公众号使用微信一物一码功能 [详情](#)

小程序权限

☒ 小程序权限

如有请选择

☒ 帐号管理权限

帮助小程序获取二维码，进行帐号管理 [详情](#)

☒ 开发管理与数据分析权限

帮助小程序进行功能开发与数据分析 [详情](#)

☒ 客服消息管理权限

帮助小程序接收和发送客服消息 [详情](#)

☒ 开放平台帐号管理权限

帮助小程序绑定开放平台帐号，实现用户身份打通 [详情](#)

☐ 小程序基本信息设置权限

帮助小程序设置名称、头像、简介、类目等基本信息 [详情](#)

☐ 小程序认证权限

帮助小程序申请认证 [详情](#)

☐ 微信卡路里权限

为小程序提供用户卡路里同步、授权查询、兑换功能 [详情](#)

☐ 附近地点权限

帮助小程序创建附近地点，可设置小程序展示在“附近的小程序”入口中 [详情](#)

☐ 插件管理权限

用于代小程序管理插件的添加和使用 [详情](#)

☐ 好物圈权限

帮助小程序将物品、订单、收藏等信息同步至好物圈，方便用户进行推荐 [详情](#)

☐ 物流助手权限

帮助有物流需求的开发者，快速高效对接多家物流公司。对接后用户可通过微信服务通知接收实时物流状态，提升用户体验 [详情](#)

填写开发资料

为了表述方便, 下文将以源源客正式站站为例.

websocket地址: **b-websocket.yuanyuanke.cn**

授权流程相关

- 授权发起页域名: **b.yuanyuanke.cn**

必须从本域名内网页跳转到登录授权页，才可完成登录授权。无需填写 `http://`等域名协议前缀。

- 授权测试公众号列表: **gh_c61c423a93b4**

在全网发布之前，仅该列表内公众号才可进行授权(包含测试小程序)，以便测试。请填写公众号的原始ID（可在公众平台网站的公众号设置页找到），最多10个，以英文“,”隔开。

- 授权事件接收URL: **b.yuanyuanke.cn/wechatAuth/events**

用于接收取消授权通知、授权成功通知、授权更新通知，也用于接收ticket，ticket是验证平台方的重要凭据。

授权后实现业务

- 消息校验Token: qwn2995ktj024

开发者在代替公众号或小程序接收到消息时，用此Token来校验消息。

消息校验Token前后, 绝不能有空格

- 消息加解密Key: skljkfgeiawwwqvn65997wj01ndnfoqpe8x2k0nck

在代替公众号或小程序收发消息过程中使用。必须是长度为43位的字符串，只能是字母和数字。

消息加解密Key前后, 绝不能有空格. 长度必须43位

- 消息与事件接收URL: b.yuanyuanke.cn/wechatAuth/callback/\$APPID\$

通过该URL接收公众号或小程序消息和事件推送，该参数按规则填写（需包含/\$APPID\$，如www.abc.com/\$APPID\$/callback），实际接收消息时\$APPID\$将被替换为公众号或小程序AppId。

- 公众号开发域名: b.yuanyuanke.cn

第三方平台在代公众号做网页授权、调用JS SDK等网页开发工作时所用的域名，以;隔开。为了满足开发者管理需要，符合以下要求的下级域名也将生效：

\$APPID\$.wx.abc.com（\$APPID\$为公众号的AppID的替换符）

请下载校验文件，并将文件放置在域名根目录下，例如wx.qq.com，并确保可以访问该文件。

每月可提交修改申请3次，本月有3次机会

- 小程序服务器域名: b.yuanyuanke.cn;b-websocket.yuanyuanke.cn;b-img-cdn.yuanyuanke.cn;mmbiz.qpic.cn;wx.qlogo.cn;

第三方平台旗下授权的小程序，只可配置本平台服务器域名列表中的域名，以;隔开。

每月可提交修改申请3次，本月有2次机会

需要访问的小程序服务器域名, 同时需要后端接口b.yuanyuanke.cn 和 websocket 服务: b-websocket.yuanyuanke.cn

- 小程序业务域名: 无需填写

请下载校验文件，并将文件放置在域名根目录下，例如wx.qq.com，并确保可以访问该文件。

每月可提交修改申请3次，本月有3次机会

其他

- 白名单IP地址列表: 47.115.75.92;47.115.146.58;47.100.201.135

仅当开发者IP地址在该列表中时，才被允许调用相关接口。最多填写100个IP地址，以英文“;”隔开。

服务器出口ip地址, 如果是vpc. 否则访问微信接口会被拒

获取APPSECRET

获取appsecret(微信开放平台管理中心的第三方平台详情页中的AppID和AppSecret)

绑定开发者小程序

微信开放平台管理中心的第三方平台详情页的开发配置里添加开发小程序

同时在对应的小程序添加开发者

开放平台绑定小程序

微信开放平台管理中心的小程序中去绑定小程序.

域名备案

根据《互联网信息服务管理办法》以及《非经营性互联网信息服务备案管理办法》，国家对非经营性互联网信息服务实行备案制度，对经营性互联网信息服务实行许可制度。未取得许可或者未履行备案手续的，不得从事互联网信息服务。即所有对中国大陆提供服务的网站都必须先进行 ICP 备案，才可开通服务。阿里云 ICP 代备案系统为您提供申请备案、修改注销备案信息、认领备案等功能。

如果由商派提供采购阿里云服务器, 可以选用我们的备案服务.

阿里云备案

[阿里云备案](#)

腾讯云备案

[腾讯云备案](#)

微软云(中国)

[微软云备案](#)

华为云备案

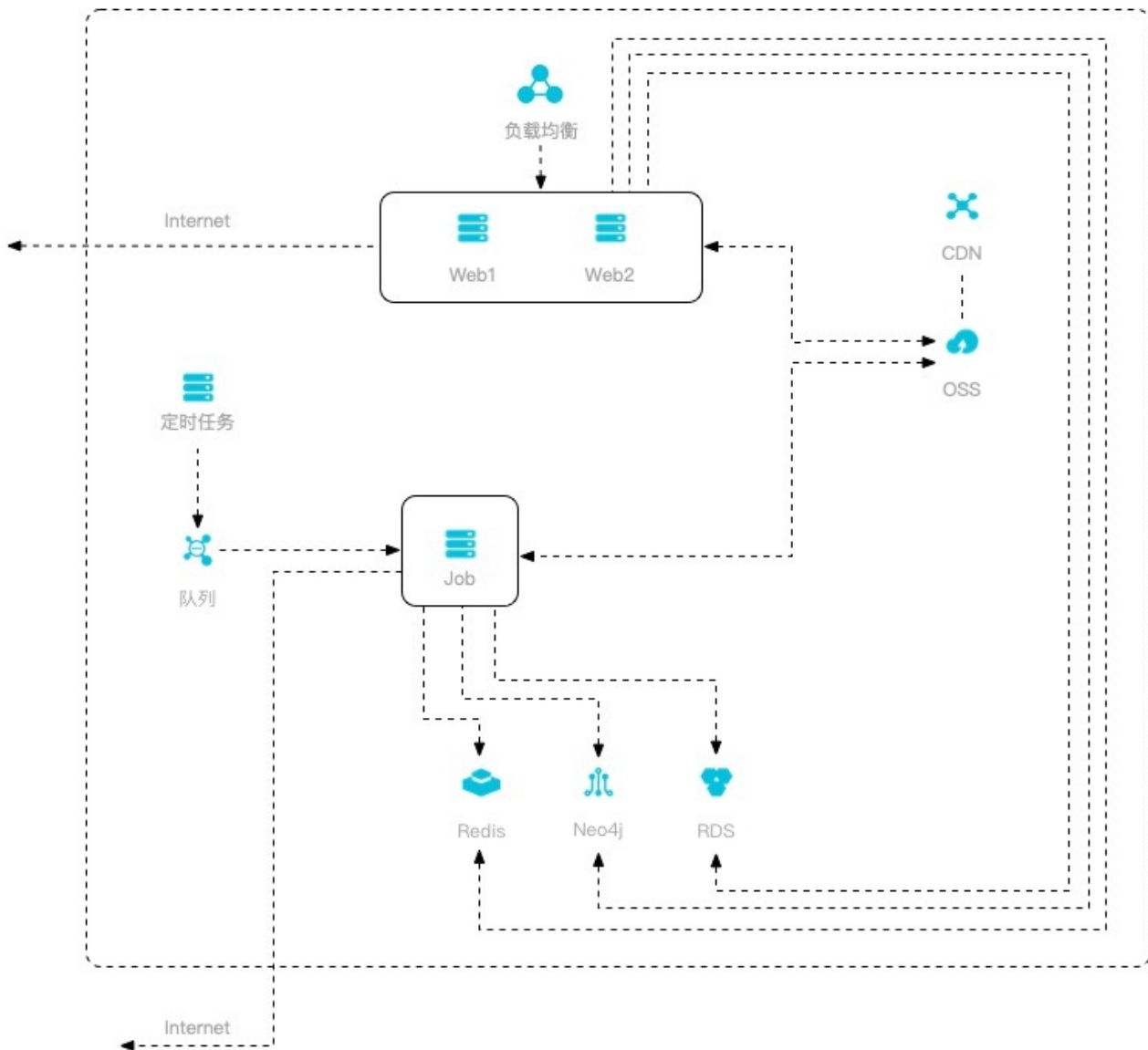
[华为云备案](#)

网易云备案

[网易云备案](#)

服务器方案

独立部署服务器方案



- 角色定义
 - 负载均衡: Load Balance
 - Web服务:
API服务
 - 队列:
可使用redis/rabbitmq
 - scheduler:
定时任务, 例如凌晨1点产生统计任务, 加入队列.
 - job:
处理队列中的任务

- neo4j:
图数据库
- redis:
Key-Value数据库
- OSS:
对象存储, 存储文件/图片/视频
- CDN:
CDN的全称是Content Delivery Network, 即内容分发网络

声明

- 本文方案所指皆为以阿里云, 其它云方案可以此为基准.
- 数据库建议使用RDS. 如果不使用RDS, 请做好备份方案, 以免误操作造成数据丢失.

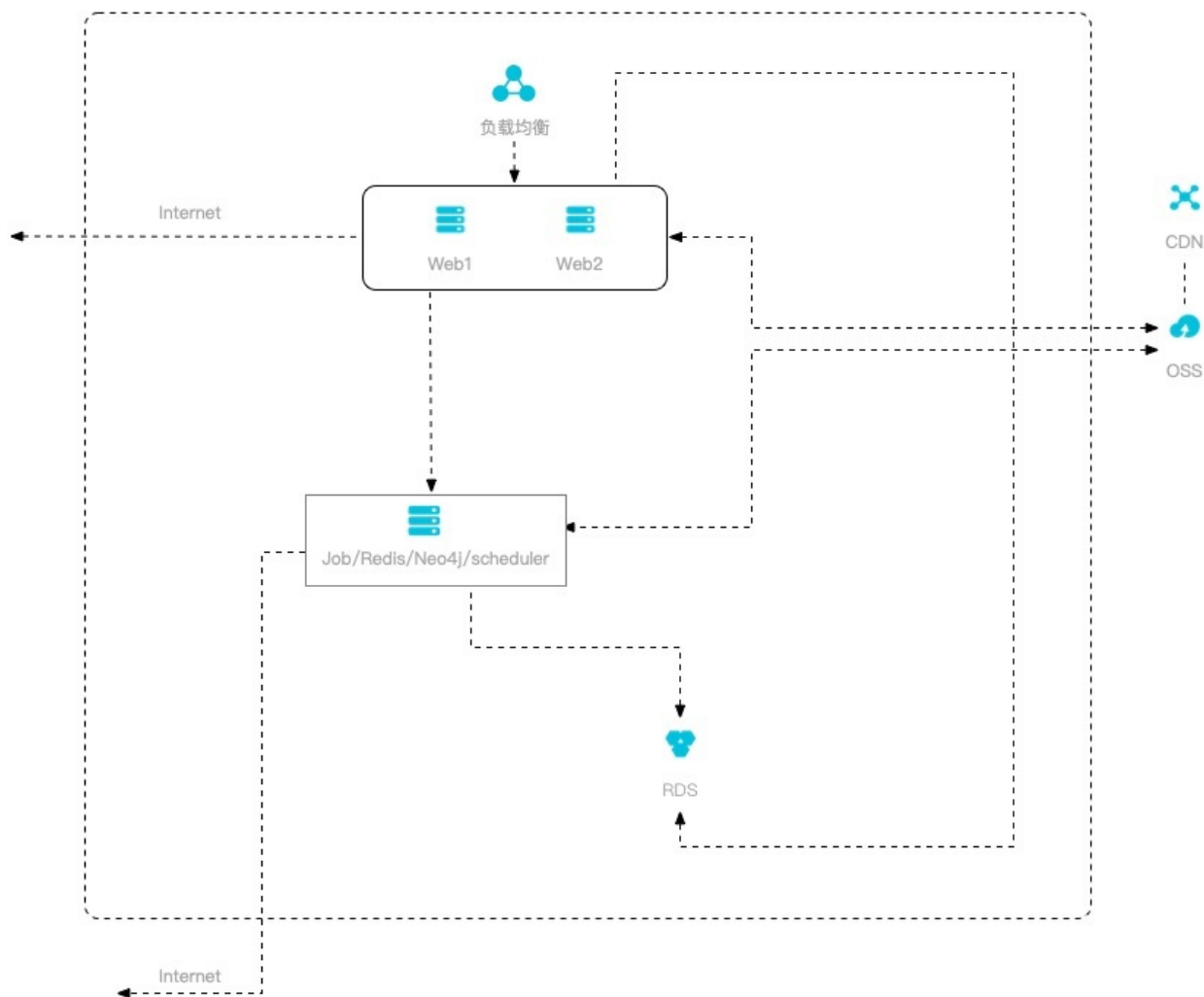
入门级方案

- ECS1台 4核8g 100g高速硬盘
部署 redis/neo4j/ecshopX(web/Job/scheduler)
- RDS 2核4g
- 七牛云 cdn 及 图片

方案简描述:

入门级方案, 不考虑高可用, 仅作为业务刚刚开始, 初始量级不大的情况下.

标准方案



- 负载均衡
- ECS3台 4核8g 100g高速硬盘
 - Web服务: ecs*2
 - 定时任务/job/redis/neo4j ecs*1
- RDS: 2核4g

方案简描述:

优点

Web端提供两台服务器做负载均衡，保证高可用。队列任务分开部署，将可异步处理的业务从前端业务中剥离，后端的任务处理不影响API业务正常的相应。一旦业务发生瓶颈，可以方便增加Web机的数量，提高RDS的配置。以快速相应业务。

缺点

任务机上部署了过多的服务，当任务服务器负载过重时，快速扩展不易。

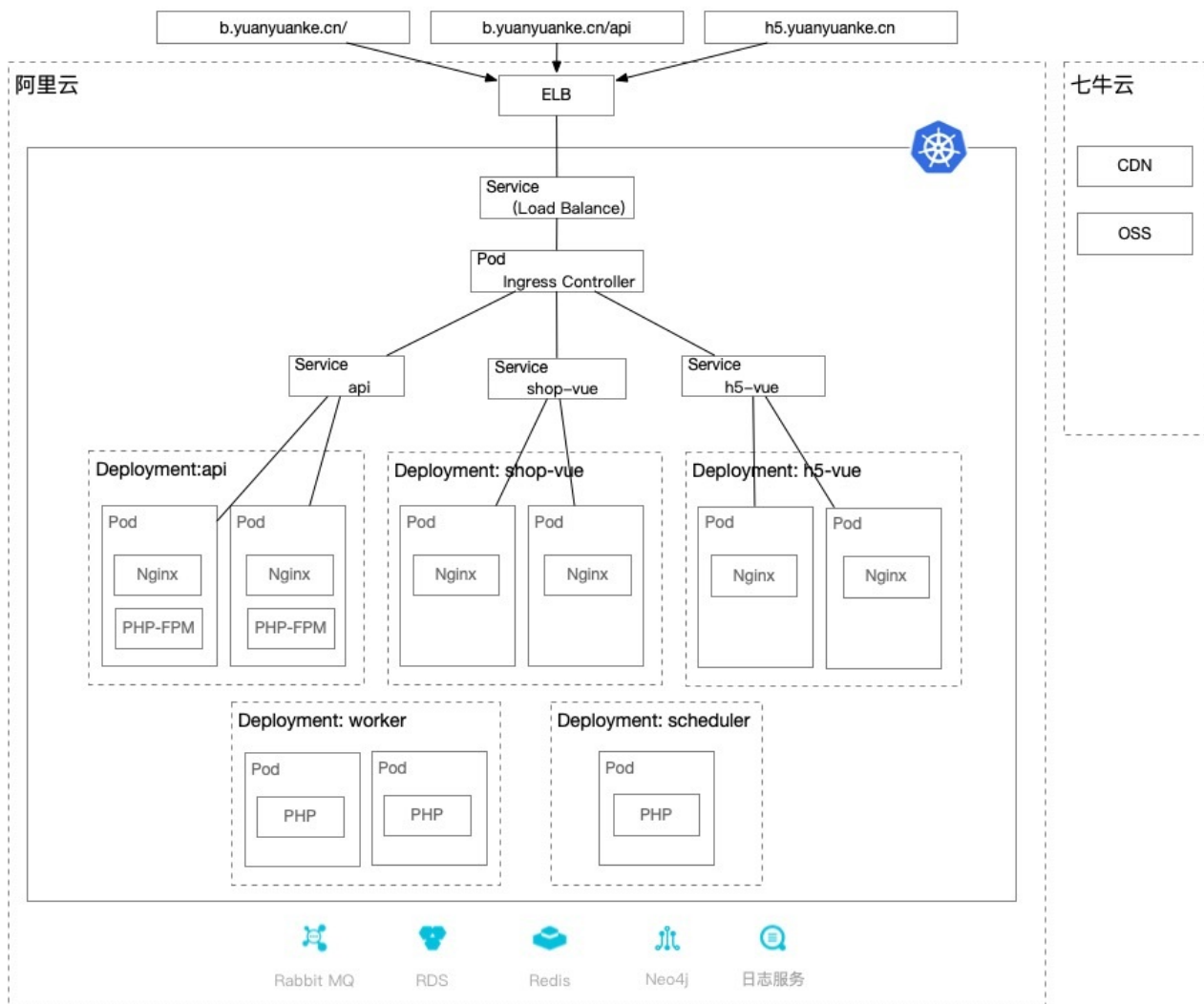
进阶方案(根据业务侧重点调整方案配置)

- 负载均衡

- ecs 4核8g 40g ssd硬盘 * 6
 - Web: ecs * 3
 - Job: ecs * 2
 - Scheduler/redis/neo4j: ecs * 1
- RDS 2核4g 100g * 1 集群版 便于扩展主从
- 七牛云 cdn 及 图片
- 日志服务

Kubernetes方案

KUBERNETES 部署方案



- kubernetes 主节点: ecs 2核4g * 3
- kubernetes node 工作节点: ecs 4核8g * 3
- RDS 2核4g 100g * 1 集群版 便于扩展主从

私有云方案(kubernetes)

服务器托管在商派集群. 使用kubernetes集群方案.

入门方案(默认)

- 共享负载均衡
- 5核5g 容器资源
- RDS 2核4g 100g * 1
- 七牛云 cdn 及 图片

标准方案

- 独享负载均衡
- ECS 4核8g 100g高速硬盘 * 2 (独享)
- RDS 2核4g
- 七牛云 cdn 及 图片

进阶方案(根据业务侧重点调整方案配置)

- 独享负载均衡
- ECS 4核8g 100g高速硬盘 * 6 (独享)
- RDS 2核4g 100g * 1 集群版 便于扩展主从
- 七牛云 cdn 及 图片

腾讯位置服务

接入步骤

小程序

本文档引自EcshopX操作手册. 前半部分为数据初始化及上线前准备工作, 因此罗列在此处.

需确认好小程序的管理员. 后续很多其它操作员的操作都需要管理员微信扫码.

- 申请小程序
- 小程序认证

EcshopX初始化

本文档引自EcshopX操作手册. 前半部分为数据初始化及上线前准备工作, 因此罗列在此处.

目前本文档仅有Word版, 如需要可向售后人员索要.

开通EcshopX

申请服务号、微信支付

申请服务号(请注意不能是订阅号)

服务号认证

申请微信支付(JSAPI支付)

如何查看微信支付签约进度/结果

微信商户平台

申请小程序、小程序微信支付

EcshopX授权绑定服务号

上架小程序须知

微信支付配置

基础设置

新增运费模版

商品管理

新增商品品牌

新增商品规格

商品参数

商品主目录

商品类目关联规格、参数

关联规格

关联参数

新增商品分类

商品录入

商品会员价

批量处理

会员卡

普通等级

付费等级

付费等级会员卡延期

会员导出

优惠券

折扣券

代金券

兑换券

会员营销

会员标签

群发优惠券

群发短信

新客营销

赠送优惠券

赠送付费会员卡

赠送积分

商城小程序模版装修

客服设置

单机部署方案

约定

- 操作系统: Centos 7.x x64
- 域名: 文档内域名规划参见 [系统部署/准备/域名](#)
- 操作权限: root

服务清单

- supervisorctl > 3.0
- neo4j 3.5.x
- mysql 5.7
- nginx + fpm
- npm > 6.0
- redis ~ 4.x
- php 7.2

基础服务部署

调整时间

```
timedatectl set-timezone Asia/Shanghai
```

安装sudo

```
yum install sudo
```

安装PHP

```
yum -y install epel-release
yum install http://rpms.remirepo.net/enterprise/remi-release-7.rpm
yum install yum-utils
yum-config-manager --enable remi-php72

yum install php72 \
    php72-php-gd \
    php72-php-json \
    php72-php-mbstring \
    php72-php-mysqlnd \
    php72-php-xml \
    php72-php-opcache \
    php72-php-bcmath \
    php72-php-pear \
    php72-php-pear-mongodb \
    php72-php-pear-zip \
    php72-php-pear-redis4 \
    php72-php-pear-ncurses \
    php72-php-pear-apcu

ln /opt/remi/php72/root/usr/bin/php /usr/bin/php
```

验证

```
php --version
php --modules
```

安装composer

```
curl -sS https://getcomposer.org/installer | php -- --install-dir=/usr/bin --
filename=composer
/usr/bin/composer config -g repo.packagist composer https://mirrors.aliyun.co
m/composer/
```

安装supervisor

版本对于3.0

安装supervisor


```
yum install -y epel-release  
yum install -y supervisor
```

开启supervisorctl

```
systemctl enable supervisord.service
```

创建用户和组

```
groupadd www  
useradd --shell /sbin/nologin -g www www
```

安装php-fpm

安装php-fpm

```
yum install php72-php-fpm
```

开启php-fpm

```
systemctl enable php72-php-fpm.service
```

修改用户权限, 将用户和组从apache 改为www

```
/etc/opt/remi/php72/php-fpm.d/www.conf
```

```
user = www  
group = www
```

安装npm

1. 添加Node.js Yum仓库

```
curl -sL https://rpm.nodesource.com/setup_12.x | bash -
```

2. 安装Node.js

```
yum install -y nodejs
```

3. 检查**Node.js**和**NPM**版本 `` node -v

v12.6.0 Also, check the version of npm.

npm -v

6.9.0

参考: [How To Install Latest Nodejs on CentOS/RHEL 7/6](<https://tecadmin.net/install-latest-nodejs-and-npm-on-centos/>)

安装nginx

安装nginx

``bash

yum install nginx

开启nginx

```
systemctl enable nginx.service
```

安装crontabs

```
yum install crontabs
```

```
systemctl enable crond.service
```

```
systemctl start crond.service
```

安装redis

安装redis

```
yum install -y redis
```

开启redis

```
yum enable redis
```

修改配置文件/etc/redis.conf

```
# requirepass foobared
requirepass      #指定密码为 redis1234
```

启动redis

```
systemctl start redis
```

测试

```
redis-cli -h 127.0.0.1 -a redis1234
keys *
```

安装neo4j

1. 导入签名并且添加neo4j的源到本地centos源中

```
rpm --import https://debian.neo4j.org/neotechnology.gpg.key
cat <<EOF> /etc/yum.repos.d/neo4j.repo
[neo4j]
name=Neo4j RPM Repository
baseurl=https://yum.neo4j.org/stable
enabled=1
gpgcheck=1
EOF
```

2. 安装neo4j

- 用root权限安装neo4j社区版

```
yum install neo4j-3.5.7
```

1. 运行以下命令会返回已安装的Neo4j的版本:

```
rpm -qa | grep neo4j
```

1. 修改默认密码

生产环境使用时需要修改初始密码, 否则会安装使用时报错.

```
# neo4j-admin set-initial-password <password>
neo4j-admin set-initial-password neo4j1234
```

安装mysql

如果使用云方提供的RDS, 请忽略安装

添加mysql7的源到本地centos源中

```
yum install -y https://dev.mysql.com/get/mysql57-community-release-el7-9.noar
ch.rpm
```

安装mysql

```
yum install -y mysql-server
```

修改配置文件**/etc/my.conf**(配置位置取决于安装方式)

**

如果使用阿里云的RDS可以忽略此步骤

```
sql_mode='STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DIVISION
_BY_ZERO,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION'
```

开启mysql

```
systemctl enable mysqld
```

启动mysql

```
systemctl start mysqld
```

在安装过程中, 为MySQLroot用户生成一个临时密码

```
grep 'temporary password' /var/log/mysqld.log
```

输出:

```
2016-12-01T00:22:31.416107Z 1 [Note] A temporary password is generated for root@localhost: mqRfBU_3Xk>r
```

请记录下密码, 在此例里中为**mqRfBU_3Xk>r**

```
mysql_secure_installation
```

会提示你输入默认的**root**密码, 输入好后, 会被要求改掉密码.

输出

```
The existing password for the user account root has expired. Please set a new password.
```

```
New password:
```

输入一个新的12个字符的密码, 该密码至少包含一个大写字母、一个小写字母、一个数字和一个特殊字符。在提示时重新输入.

在此例中设置为, **Wka25ijklmng0ada-x**

您将收到关于新密码强度的反馈, 然后您将立即被提示再次更改密码.

输出

```
Estimated strength of the password: 100
Change the password for root ? (Press y|Y for Yes, any other key for No) :
```

我们将按下**Y**, 然后进入所有后续问题, 以便删除匿名用户、禁止远程根登录、删除测试数据库并访问它, 并重新加载特权表.

命令行连接**mysql**

```
mysql -uroot -p"Wka25ijklmng0ada-x"
```

创建应用数据库

```
CREATE DATABASE espier_bloated CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

参考: [How To Install MySQL on CentOS 7](#)

系统应用部署

部署几部分

商家端(VUE) + API端

配置授权

安装swoole-loader

确认PHP扩展目录

```
# php -r 'phpinfo();' | grep -i '^extension_dir'

extension_dir => /opt/remi/php72/root/usr/lib64/php/modules => /opt/remi/php72/root/usr/lib64/php/modules
```

本例中为**/opt/remi/php72/root/usr/lib64/php/modules**

```
curl https://www.swoole-cloud.com/static/loader2.0.0/swoole_loader72.so > /opt/remi/php72/root/usr/lib64/php/modules/swoole_loader72.so
```

php扩展配置文件位置

```
# php -r 'phpinfo();' | grep -i 'Scan this dir for additional'

Scan this dir for additional .ini files => /etc/opt/remi/php72/php.d
```

设置授权

注意配置license的正确位置, 默认license.zl在代码包(espier-bloated)根目录下

```
{ \
    echo "extension=swoole_loader72.so"; \
    echo "swoole_license_files=/var/www/html/license.zl"; \
} > /etc/opt/remi/php72/php.d/60-swoole_loader.ini
```

重启fpm

```
systemctl enable php72-php-fpm.service
```

部署API服务(PHP)

配置

后端代码

```
mkdir -p /var/www/espier-bloated  
chown -R www:www /var/www/espier-bloated
```

放置代码到**/var/www/espier-bloated**目录中

修改配置**.env**, 拷贝**.env.production**文件做为模版.

1. 配置数据库

```
DB_HOST=127.0.0.1  
DB_PORT=3306  
DB_DATABASE=espier_bloated  
DB_USERNAME=root  
DB_PASSWORD=wka25ijklmng0ada-x
```

2. 配置REDIS

```
# REDIS 相关配置  
REDIS_CLIENT=predis  
REDIS_HOST=redis-ali-espier-master  
REDIS_PASSWORD=redis1234  
REDIS_MEMBERS_PORT=6379
```

3. 配置队列

```
QUEUE_DRIVER=redis
```

4. 配置neo4j

```
NEO4J_DEFAULT_PROTOCOL=bolt
NEO4J_DEFAULT_HOST=127.0.0.1
NEO4J_DEFAULT_PORT=7687
NEO4J_DEFAULT_USERNAME=neo4j
NEO4J_DEFAULT_PASSWORD=neo4j1234
```

5. 配置缓存

```
CACHE_DRIVER=redis
```

6. 配置七牛

如何获取[Access Key](#)和[Secret Key](#)

[icon.png](#)

```
# 图片CDN域名
QINIU_IMAGE_DOMAIN=b-img-cdn.yuanyuanke.cn
# 图片bucket
QINIU_IMAGE_NAME=espier-images
# 导入导出文件CDN域名
QINIU_FILE_DOMAIN=https://b-import-cdn.yuanyuanke.cn
导入导出bucket
QINIU_FILE_NAME=espier-import-files
# 七牛Access Key
QINIU_ACCESS_KEY=
# 七牛Secret Key
QINIU_SECRET_KEY=
```

7. 配置微信开放平台第三方平台相关配置

需要根据实际

```
# 微信开放平台对应第三方平台APPID
WECHAT_APPID=
# 微信开放平台对应第三方平台APPSECRET
WECHAT_SECRET=
# 微信开放平台对应第三方平台 消息校验Token
WECHAT_TOKEN=
# 微信开放平台对应第三方平台 消息加解密Key
WECHAT_AES_KEY=
WECHAT_DEBUG=true
```

8. 配置商城小程序模版

模版ID, 需要添加到小程序模版库, 获得模版ID, 参考:[开发者平台小程序模板](#)

```
# 商城小程序模版相关配置
# 小程序模版ID号
YYKWEISHOP_TEMPLATE_ID=
# 小程序模版版本
YYKWEISHOP_VERSION=
# request合法域名
YYKWEISHOP_REQUESTDOMAIN=https://b.yuanyuanke.cn
# socket合法域名
YYKWEISHOP_WSREQUESTDOMAIN=wss://b-websocket.yuanyuanke.cn
# uploadFile(上传文件)合法域名
YYKWEISHOP_UPLOADDOMAIN=https://b.yuanyuanke.cn
# downloadFile(下载文件)合法域名
YYKWEISHOP_DOWNLOADDOMAIN1=https://mmbiz.qpic.cn
YYKWEISHOP_DOWNLOADDOMAIN2=https://b.yuanyuanke.cn
YYKWEISHOP_DOWNLOADDOMAIN3=https://wx.qlogo.cn
```

1. 配置腾讯位置服务

腾讯位置服务需要申请账号配置KEY, KEY对接口的访问是有配额的, 配额内是免费的.

在[腾讯位置服务Key管理](#), 可以查询到Key, 并且可以查询到配额.

为了开发环境方便, 提供只为开发环境公用的免费KEY. 因为是公共的, 无法保证一定可用.

```
QQMAP_KEY=PSPBZ-KQ5CW-CSGRF-ON2S4-K2HQJ-XEBQG
```

1. 配置支付通知接口

```
WECHAT_PAYMENT_NOTIFY=https://b.yuanyuanke.cn/wechatAuth/wxpay/notify
```

完整的例子

```
# 标识应用环境, 本地环境设置为local
APP_ENV=local
# 调试模式, 正式环境可以关闭
APP_DEBUG=true
# 时区不要修改
APP_TIMEZONE=PRC

# 数据库相关配置
```

```
# DB_CONNECTION默认不需要改
DB_CONNECTION=default
# 数据库主机
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=espier_bloated
DB_USERNAME=root
DB_PASSWORD=wka25ijklmng0ada-x

# 缓存驱动, 默认使用redis
CACHE_DRIVER=redis
# 队列驱动, 默认使用redis
QUEUE_DRIVER=redis

# 七牛相关配置
QINIU_IMAGE_DOMAIN=b-img-cdn.yuanyuanke.cn
QINIU_IMAGE_NAME=espier-image
QINIU_FILE_DOMAIN=https://b-import-cdn.yuanyuanke.cn
QINIU_FILE_NAME=espier-file
QINIU_ACCESS_KEY=
QINIU_SECRET_KEY=

# 短信会用到, 如果需要商派短信, 误删
PRISM_URL=http://openapi.shopex.cn/api
PRISM_KEY=7zfgrchh
PRISM_SECRET=ykiy27tqhlnbn6ndaddr

# JWT 相关配置, 可根据需要调整或不调整
JWT_SECRET=ker4H1sp4TsdDvcVJMa72SMW8Zsh3drv
JWT_TTL=300
JWT_REFRESH_TTL=20160

# 微信第三方平台设置. 本地开发环境不配置, 会影响到调用微信的相关页面, 整体不影响后台开发.
WECHAT_APPID=
WECHAT_SECRET=
WECHAT_TOKEN=
WECHAT_AES_KEY=
WECHAT_DEBUG=true

# 支付通知接口
WECHAT_PAYMENT_NOTIFY=https://b.yuanyuanke.cn/wechatAuth/wxpay/notify

# dingo标准配置, 根据实际场景进行配置
API_PREFIX=api
API_NAME="Espier API"
```

```

API_STANDARDS_TREE=vnd
API_STRICT=false
API_DEBUG=false
API_VERSION=v1
API_SUBTYPE=espier
API_CONDITIONAL_REQUEST=false
API_TOKEN=0s6Bass1oT5vig2Yod0yiT8dU0as5cIn

RPCCALL_DRIVER=dingo

# 腾讯地图接口KEY, 建议根据项目进行申请
QQMAP_KEY=PSPBZ-KQ5CW-CSGRF-0N2S4-K2HQJ-XEBQG

# REDIS 相关配置
REDIS_CLIENT=predis
REDIS_HOST=redis-ali-espier-master
REDIS_PASSWORD=redis1234
REDIS_MEMBERS_PORT=6379

LICENSE_PRODUCTION_URL=https://service.ec-os.net/api/yyk/register
LICENSE_PRODUCT=yyk

# 商城小程序相关配置
# 小程序模版ID号
YYKWEISHOP_TEMPLATE_ID=
YYKWEISHOP_REQUESTDOMAIN=https://b.yuanyuanke.cn
YYKWEISHOP_WSREQUESTDOMAIN=wss://b-websocket.yuanyuanke.cn
# 小程序版本
YYKWEISHOP_VERSION=
YYKWEISHOP_DOWNLOADDOMAIN1=https://mmbiz.qpic.cn
YYKWEISHOP_DOWNLOADDOMAIN2=https://b.yuanyuanke.cn
YYKWEISHOP_DOWNLOADDOMAIN3=https://wx.qlogo.cn

WEBSOCKET_SERVER_PORT=9051
WEBSOCKET_SERVER_HOST=b-websocket.yuanyuanke.cn

SENTRY_LARAVEL_DSN=https://eb0daff7a4244e2e8e8131e178952c55:d13add823922418f859c081feb671d73@report.shopex.cn/41

SUPER_ADMIN_LOGIN_NAME=yykpms
SUPER_ADMIN_LOGIN_PASSWORD=c0f50922ef5c2d72adac424fca752aa97523fa33

NEO4J_DEFAULT_PROTOCOL=http
NEO4J_DEFAULT_HOST=distribution-neo4j-neo4j-core-0.distribution-neo4j-neo4j.e

```

```
spier.svc.cluster.local
NEO4J_DEFAULT_PORT=7474
NEO4J_DEFAULT_USERNAME=neo4j
NEO4J_DEFAULT_PASSWORD=oPMfxPwMwY

BROKERAGE_URI=
BROKERAGE_URI_ITEM=
ALIPAY_PAYMENT_NOTIFY=
ALIPAY_PAYMENT_RETURN=

H5_BASE_URL=
```

安装

安装composer扩展包

```
cd /var/www/espier-blaoted
composer install
```

数据库迁移

```
cd /var/www/espier-blaoted
./artisan doctrine:migrations:migrate
```

部署商家端(vue)

商家端代码

代码build好后build目录里的文件直接copy至此目录

1. 创建项目目录

```
mkdir -p /var/www/espier-retail-manage
```

2. 放置代码

通常会通过git的方式进行部署及更新

修改**app/config/test.env.js**

```
module.exports = {
  NODE_ENV: '"testing"',
  BASE_API: '"https://b.yuanyuanke.cn/api"',
  //WXIMG_URL: '"https://bbc54.shopex123.com/image/"',
  WXIMG_URL: '""',
  WXAUTHCALL_Url: '"https://b.yuanyuanke.cn/"'
}
```

3. 安装npm宝 `` cd /var/www/espier-retail-manage/app

npm install

4. 代码编译目录

****/var/www/espier-retail-manage/app/config/index.js****

assetsRoot: path.resolve(__dirname, '../test'),

编译后的文件在****/var/www/espier-retail-manage/app/test****下

5. 代码编译

npm run build-test

6. 更改权限

chown -R www:www /var/www/espier-retail-manage

配置nginx

修改/etc/nginx/nginx.conf, 将权限nginx 改为www

use www;

创建项目nginx配置

```
vim /etc/nginx/conf.d/espier.conf
```

```
<br />
```

```
需要配置ssl证书, 在下边配置的所有**配置ssl**处.<br />
```

```
server { listen 443;
```

```
# 配置ssl
```

```
server_name b.yuanyuanke.cn;
```

```
location /api/ {
```

```
    access_log /var/log/nginx/espier-bloated.log;
```

```
    proxy_pass http://localhost:8080;
```

```
}
```

```
location /wechatAuth/ {
```

```
    access_log /var/log/nginx/espier-wechatauth.log;
```

```
    proxy_pass http://localhost:8080;
```

```
}
```

```
location / {
```

```
    proxy_pass http://localhost:8081;
```

```
}
```

```
}
```

配置API服务(espier-bloated)

```
server {
```

```
listen 8080;
listen [::]:8080;

server_name localhost;
root /var/www/espier-bloated/public;
index index.php index.html index.htm;

location / {
    try_files $uri $uri/ /index.php$is_args$args;
}

location ~ /\.php$ {
    add_header Access-Control-Allow-Origin '*' always;
    add_header Access-Control-Allow-Headers "Content-Type, Access-Control-Allow-Headers, Authorization, X-Requested-With";
    add_header Access-Control-Expose-Headers "Authorization";
    add_header Access-Control-Allow-Methods "DELETE, GET, HEAD, POST, PUT, OPTIONS, TRACE, PATCH";

    if ($request_method = OPTIONS ) {
        return 200;
    }

    fastcgi_pass localhost:9000;
    fastcgi_read_timeout 150;
    fastcgi_index index.php;
    fastcgi_buffers 16 16k;
    fastcgi_buffer_size 32k;
    fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
    include fastcgi_params;
}

location ~ /\.ht {
    deny all;
}

}
```

配置商家端(espier-retail-manage)

```
server { listen 8081; listen [::]:8081;
```

```
server_name localhost;

location / {
    root    /var/www/espier-retail-manage/app/test;
    index  index.html index.htm;
    try_files $uri $uri/ /index.html =404;
}

location /wximage/ {
    set $hostx "";
    set $addrs "";
    if ( $uri ~ "^/wximage/http./+([^\./]+)/(.+)$" ) {
        set $hostx $1;
        set $addrs $2;
    }
    resolver 8.8.8.8;
    proxy_pass http://$hostx/$addrs;
    proxy_set_header referer "http://read.html5.qq.com/image";
}
```

}

配置websocket服务 server { listen 443;

```
# 配置ssl

server_name b-websocket.yuanyuanke.cn;

location / {
    proxy_pass http://localhost:9051/;

    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header Host $host;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;

    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "upgrade";
}
```

}


```
<a name="d610204a"></a>
### 配置定时任务
```

crontab -e

- - - - - sudo -E -u www php /var/www/espier-bloated/artisan schedule:run >> /var/log/espier-cront.log 2>&1 ``

配置队列服务

修改队列配置

```
vim /etc/supervisord.d/super-queue.ini
```

```
[program:bloated-queue-default]
command=/var/www/espier-bloated/artisan queue:work --queue=default --delay=3
--memory=128 --timeout=30 --sleep=1 --tries=3
stdout_logfile=/var/www/espier-bloated/storage/logs/supervisor-bloated-queue-
default.log
redirect_stderr=true
process_name=%(program_name)s_%(process_num)02d
autostart=true
autorestart=true
numprocs=1
user=www
startsecs=3
startretries=100000

[program:bloated-queue-slow]
command=/var/www/espier-bloated/artisan queue:work --queue=slow --delay=3 --m
emory=128 --timeout=1800 --sleep=1 --tries=3
stdout_logfile=/var/www/espier-bloated/storage/logs/supervisor-bloated-queue-
slow.log
redirect_stderr=true
process_name=%(program_name)s_%(process_num)02d
redirect_stderr=true
autostart=true
autorestart=true
numprocs=1
user=www
startsecs=3
startretries=100000

[program:bloated-queue-sms]
command=/var/www/espier-bloated/artisan queue:work --queue=sms --delay=3 --me
mory=128 --timeout=1800 --sleep=1 --tries=3
stdout_logfile=/var/www/espier-bloated/storage/logs/supervisor-bloated-queue-
sms.log
redirect_stderr=true
process_name=%(program_name)s_%(process_num)02d
redirect_stderr=true
autostart=true
autorestart=true
numprocs=1
user=www
startsecs=3
startretries=100000
```

配置websocket服务

配置websocket

```
vim /etc/supervisord.d/super-websocket.ini
```

```
[program:websocket]
command=/var/www/espier-bloated/artisan websocket:start
stdout_logfile=/var/www/espier-bloated/storage/logs/supervisor-websocket.log
redirect_stderr=true
autostart=true
autorestart=true
numprocs=1
user=www
startsecs=3
startretries=100000
```

服务启动

启动队列任务, websocket服务

```
systemctl start supervisord.service
```

启动nginx

```
systemctl start nginx.service
```

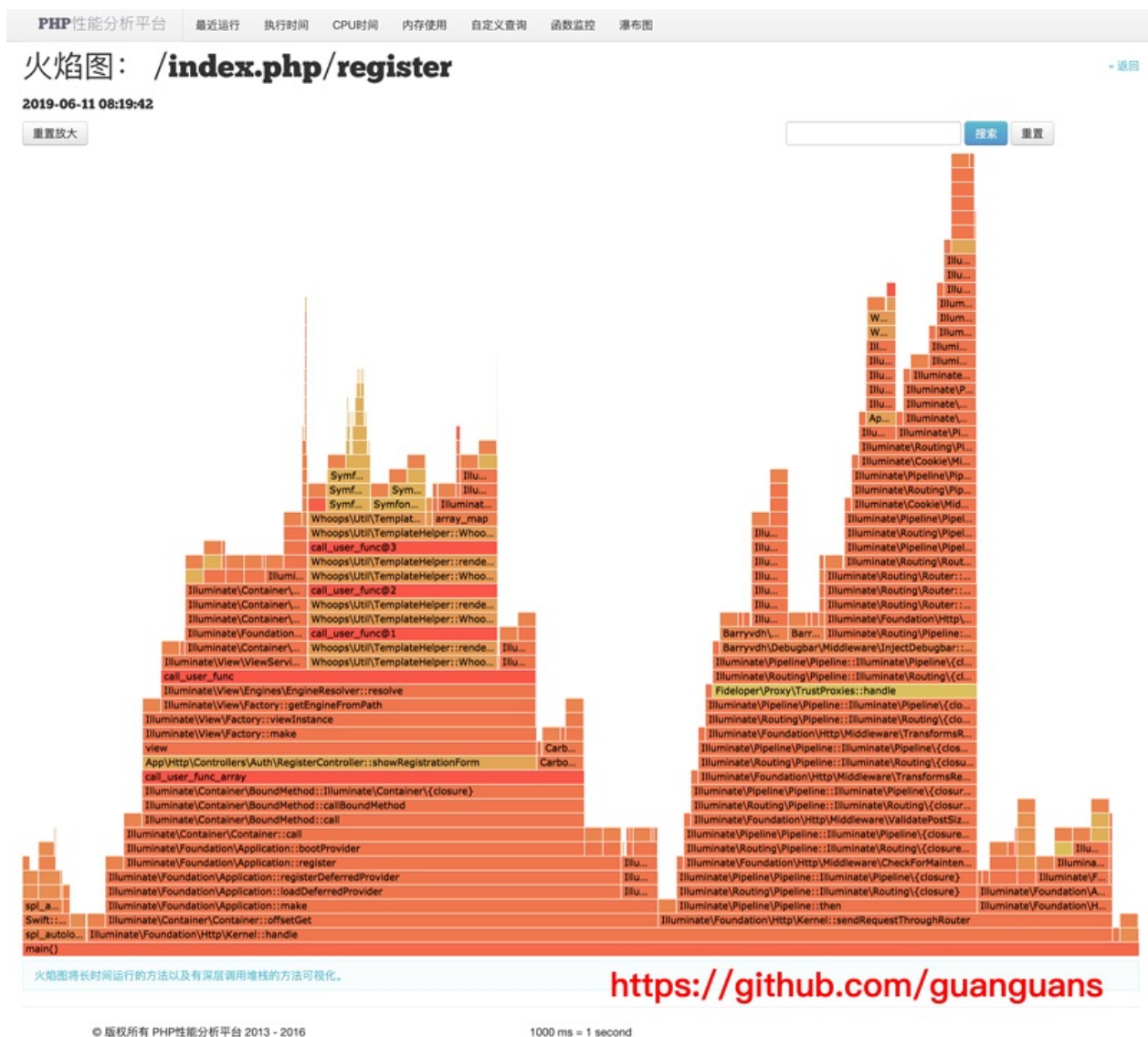
启动php-fpm

```
systemctl start php72-php-fpm.service
```

集群部署方案

本地性能调试

Tideways、xhprof 和 xhgui 打造 PHP 非侵入式监控平台



环境准备

安装之前确保已经正确安装了以下软件

- PHP
- Nginx
- MongoDB

安装 PHP mongodb 扩展

```
$ sudo pecl install mongodb
```

PHP 配置文件中添加

```
[mongodb]
extension=mongodb.so
```

安装 PHP tideaways 扩展

常规编译安装

```
$ git clone https://github.com/tideaways/php-xhprof-extension.git
$ cd /path/php-xhprof-extension
$ phpize
$ ./configure
$ make
$ sudo make install
```

PHP 配置文件中添加

```
[tideaways]
extension=tideaways_xhprof.so
; 不需要自动加载，在程序中控制就行
tideaways.auto_prepend_library=0
; 频率设置为100，在程序调用时可以修改
tideaways.sample_rate=100
```

安装 xhgui-branch (xhgui 的汉化版)

```
$ git clone https://github.com/laynefyc/xhgui-branch.git
$ cd xhgui-branch
$ php install.php
```

修改 xhgui-branch 配置文件

```
<?php
return array(
    ...
    'extension' => 'tideways_xhprof',
    ...
    'save.handler' => 'mongodb',
    'db.host' => 'mongodb://127.0.0.1:27017',
    'db.db' => 'xhprof',
    ...
);
```

启动 mongodb 并设置 xhgui 索引，命令如下：

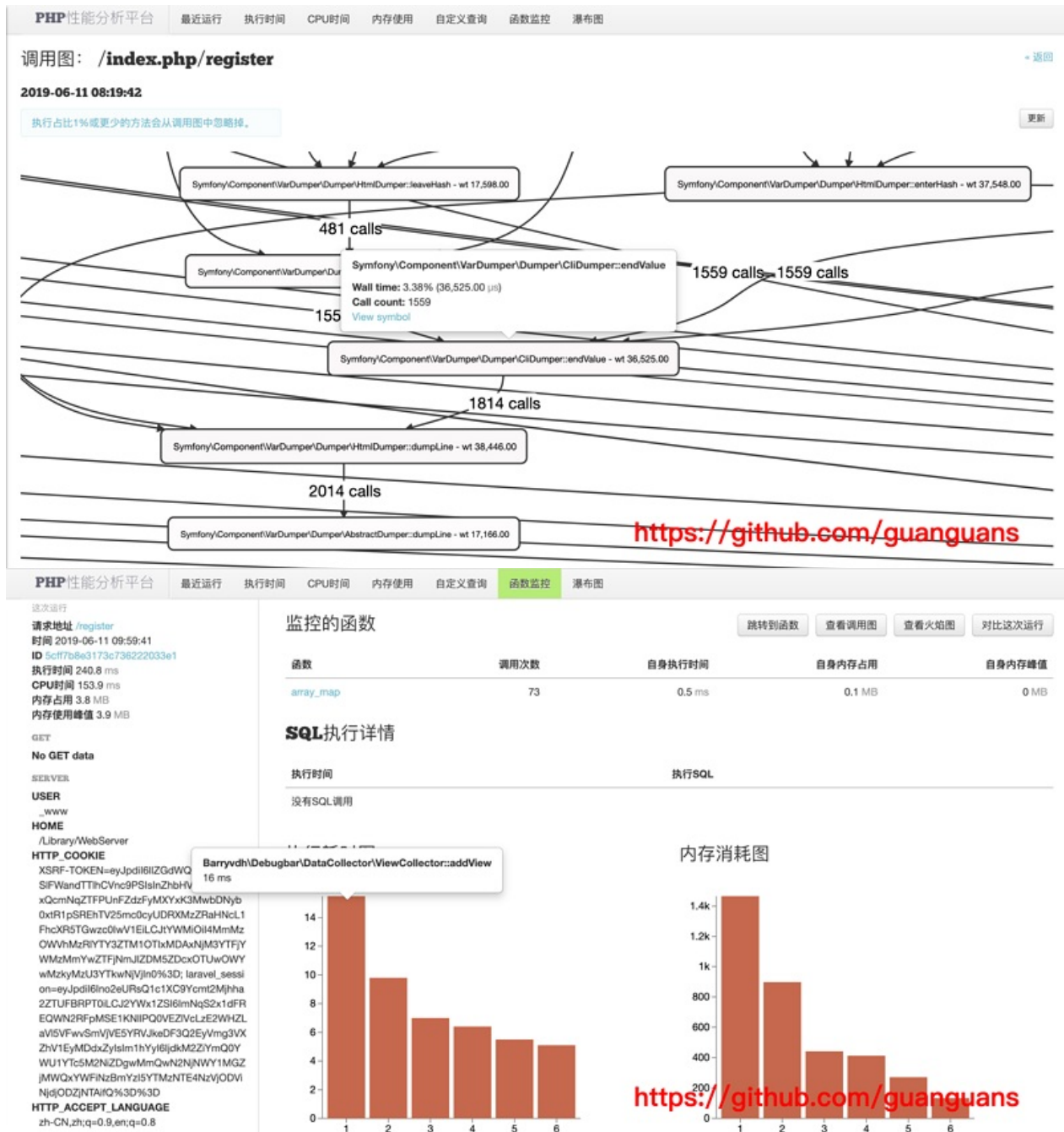
```
$ mongo
> use xhprof
> db.results.ensureIndex( { 'meta.SERVER.REQUEST_TIME' : -1 } )
> db.results.ensureIndex( { 'profile.main().wt' : -1 } )
> db.results.ensureIndex( { 'profile.main().mu' : -1 } )
> db.results.ensureIndex( { 'profile.main().cpu' : -1 } )
> db.results.ensureIndex( { 'meta.url' : 1 } )
```

xhgui 本地虚拟主机配置参考

```
server {
    listen      80;
    server_name xhgui.test;
    root        /path/xhgui-branch/webroot;
    # access_log logs/access.log;
    error_log    logs/error.log;
    location / {
        try_files $uri $uri/ /index.php?$query_string;
        index index.php index.html index.htm;
    }
    location ~ \.php$ {
        fastcgi_pass localhost:9000;
        fastcgi_read_timeout 150;
        fastcgi_index index.php;
        fastcgi_buffers 16 16k;
        fastcgi_buffer_size 32k;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        include fastcgi_params;
    }
}
```

在项目入口的 PHP 文件中 配置，告诉 PHP 程序在执行前要调用的服务

```
require_once('/path/wwwroot/xhgui-branch/external/header.php');
```

文档说明

ecshopX 的 API 层基于 Lumen 框架开发，而 Lumen 的是基于 Laravel 开发，在 ecshopX 中我们根据商派在电商领域的多年经验对其做了一些调整，本文档主要对我们调整部分的说明，及其一些架构思想的阐述。其他文档请参阅 Lumen 和 Laravel 的文档。

附相关文档地址：

- [Lumen 中文文档](#)
- [Laravel 中文文档](#)
- [Dingo 中文文档](#)

docker环境

EcshopX 采用前后端分离的方式开发，代码结构由后端api和多个前端组成。

后端api由lumen开发，部署时包含了web、scheduler、worker和websocket。

前端包含管理端、小程序、PC端和H5端。

本环境目前只包含了web和管理的部署，后续会根据需求将其他环境加入。

详细使用方式[在此查看](#)

composer

Composer 作为依赖包管理工具

使用前请确保您已安装PHP 和 Composer

使用国内源

- aliyun <https://mirrors.aliyun.com/composer/> 推荐
- tencent <https://mirrors.cloud.tencent.com/composer/>
- huaweicloud <https://mirrors.huaweicloud.com/repository/php/>
- cnpkg.org <https://php.cnpkg.org>
- laravel-china <https://packagist.laravel-china.org>

方法1：修改全局配置

打开终端并执行如下命令：

```
composer config -g repo.packagist composer https://mirrors.aliyun.com/composer/  
// OR  
composer config -g repo.packagist composer https://mirrors.huaweicloud.com/repository/php/
```

打开终端并执行如下命令：

```
composer config -g repo.packagist composer https://mirrors.aliyun.com/composer/  
// OR  
composer config -g repo.packagist composer https://mirrors.huaweicloud.com/repository/php/
```

方法2：修改项目配置

打开终端，进入你的项目的根目录（也就是 composer.json 文件所在目录），执行如下命令：

```
composer config repo.packagist composer https://mirrors.aliyun.com/composer/  
// OR  
composer config repo.packagist composer https://mirrors.huaweicloud.com/repos  
itory/php/
```

您也可以手工在composer.json中添加如下内容：

```
"repositories": {  
  "packagist": {  
    "type": "composer",  
    "url": "https://mirrors.aliyun.com/composer/"  
  }  
}
```

相关网址

- PHP官方地址：<http://php.net/>
- Composer官方地址：<https://getcomposer.org/>

目录结构

简介

ecshopX 的 API 层基于 Lumen 框架开发，根据商派在电商领域的多年经验对其做了一些调整，主要调整如下：

- 目录结构调整：除基本配置目录外，lumen自带的app目录基本已经废弃，借鉴商派ecos及symfony思想，ecshopx引入了 Bundle，每个Bundle包含了一个独立的业务。
- ORM调整：Lumen底层基于 Laravel 开发，所以 Luemn 底层ORM采用Laravel 的 Eloquent ORM实现来和数据库进行交互。考虑到更灵活的SQL能力，我们采用了 Doctrine ORM。

根目录

文件/目录	描述
app	在 Lumen 中 app 目录包含了应用的核心代码，在ecshopx中不用关心此目录的内容
bootstrap	在 app.php 中注册 ServiceProvider ，在route.php中注册各个Bundle的路由文件
config	config 目录包含了应用所有的配置文件，建议通读一遍这些配置文件以便熟悉所有配置项
database	包含了数据迁移及填充文件
public	public 目录包含了入口文件 index.php
routes	目录包含了 Bundle 的所有路由定义
src	存放 Bundle 的目录
storage	storage 目录包含了编译过的Blade模板、基于文件的session、文件缓存，以及其它由框架生成的文件，该目录被细分为成app、framework和logs子目录
vendor	vendor目录包含所有Composer依赖

中大型项目架构

Laravel 的初学者分为两种：

- 一种是乖乖的将程序写在 MVC 架构内，这将导致 Controller 和 Model 非常臃肿，日后代码很难维护。
- 一种是不知道将程序写在哪一个 Class 内而犹豫不决。

本文整理出最适合 Laravel 的中大型项目架构，兼具易维护、易扩展、易复用和易测试的特点。

Controller 过于肥大

受RoR的影响，初学者常认为 MVC 架构就是 Model, View, Controller：

- Model 就是数据库。
- Controller 负责与 HTTP 通信，调用 Model 与 View。
- View 就是 HTML。

假如按照此定义，以下需求应该写在哪里呢？

- 1.发送 Email，使用外部 API。
- 2.使用 PHP 写的逻辑。
- 3.依需求将显示格式作转换。
- 4.依需求是否显示某些资料。
- 5.依需求显示不同资料。

其中1, 2 属于商业逻辑，而3, 4, 5 属于显示逻辑，若依照一般人对 MVC 的定义，Model 是数据库，而 View 是HTML，以上这些需求都不能写在Model 与View，只能勉强写在Controller。

因此初学者开始将大量程序写在 Controller，造成 Controller 的肥大难以维护。

Model 过于肥大

既然逻辑写在 Controller 不方便维护，那我将逻辑都写在 Model 就好了？

当你将逻辑从 Controller 搬到 Model 后，虽然 Controller 变瘦了，但却肥了 Model，Model 从原本只处理数据库逻辑，现在变成还要负担商业逻辑与显示逻辑，结果更惨。

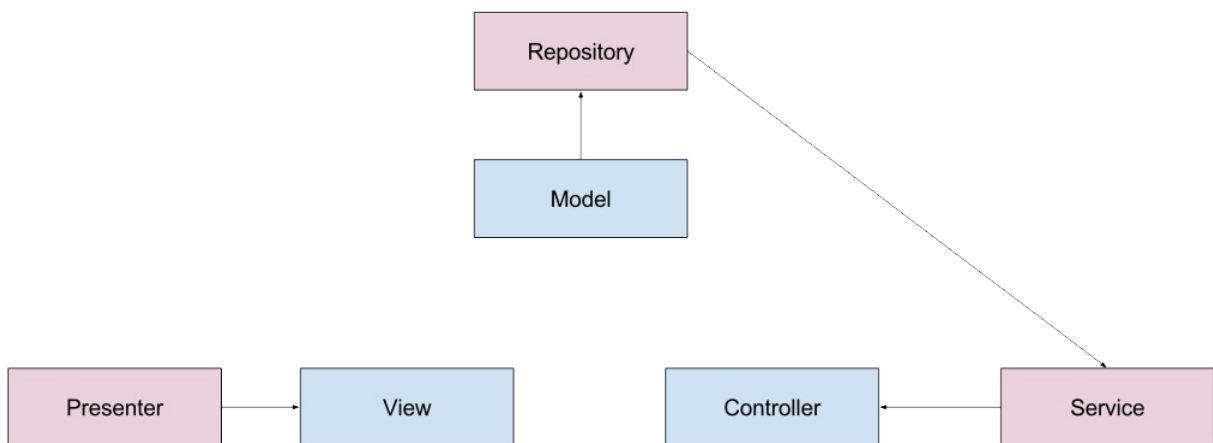
Model 代表数据库吗？把它想成是 Eloquent class 就好，数据库逻辑应该写在 repository 里，这也是为什么 Laravel (Lumen) 已经没有 Models 目录，Eloquent class 仅仅是放在 app 根目录下而已。

中大型项目架构

那我们该怎么写呢？别将我们的思维局限在 MVC 内：

- Model : 仅当成 Eloquent class。
- Repository : 辅助 Model，处理数据库逻辑，然后注入到 service。
- Service : 辅助 Controller，处理商业逻辑，然后注入到 Controller。
- Controller : 接收 HTTP request，调用其他 service。
- Presenter : 处理显示逻辑，然后注入到 View。
- View : 使用 blade 将资料 binding 到 HTML。

其中蓝色为原本的 MVC，而紫色为本文要介绍的重点：Repository 模式，Service 模式与 Presenter 模式。



箭头表示物件依赖注入的方向。

我们可以发现 MVC 架构还在，由于 SOLID 的单一职责原则与依赖反转原则：

- 我们将数据库逻辑从 Model 分离出来，由 repository 辅助 Model，将 Model 依赖注入进 repository。
- 我们将商业逻辑从 Controller 分离出来，由 service 辅助 Controller，将 service 依赖注入进 Controller。
- 我们将显示逻辑从 View 分离出来，由 presenter 辅助 View，将 presenter 依赖注入进 View。

建立目录在 src 的 Bundle 目录下建立 Repositories, Services 与 Presenters 目录。

别害怕在 Laravel 预设目录以外建立的其他目录，根据 SOLID 的单一职责原则，class 功能越多，责任也越多，因此越违反单一职责原则。

所以你应该将你的程序分割成更小的部分，每个部分都有它专属的功能，而不是一个 class 功能包山包海，也就是所谓的 万能类 。

所以整个项目不应该只有 MVC 三个部分，放手根据你的需求建立适当的目录，并将适当的 class 放到该目录下，只要我们的class 有namespace 帮我们分类即可。

Repository

由于篇幅的关系，将 repository 独立成专文讨论，请参考[如何使用 Repository 模式？](#)

Service

由于篇幅的关系，将 service 独立成专文讨论，请参考[如何使用 Service 模式？](#)

单元测试

由于现在Model、View、Controller 的依赖都已经拆开，也都使用依赖注入，因此每个部分都可以单独的做单元测试，如要测试service，就将repository 加以 mock，也可以将其他 service 加以 mock。

结论

本文谈到的架构只是开始，你可以依照实际需求增加更多的目录与 class，当你发现你的 MVC 违反 SOLID原则时，就大胆的将 class 从 MVC 拆开重构，然后依照以下手法：

- 建立新的 class 或 interface。
- 将依赖类依赖注入到 class。
- 在 class 内处理他的职责。
- 将 class 或 interface 注入到 Controller 或 View。

最后搭配单元测试，测试重构后的架构是否与原来的需求结果相同。

Repository

使用 Repository 辅助 Model

若将数据库逻辑都写在 model，会造成 model 的肥大而难以维护，基于 SOLID 原则，我们应该使用 Repository 模式辅助 model，将相关的数据库逻辑封装在不同的 repository，方便中大型项目的维护。

数据库逻辑

在 CRUD 中，CUD 比较稳定，但 R 的部分则千变万化，大部分的数据库逻辑都在描述 R 的部分，若将数据库逻辑写在 controller 或 model 都不适当，会造成 controller 与 model 肥大，造成日后难以维护。

Model

如果使用 Eloquent ORM 来实现 Repository 模式，需要修改 model 不要包含数据库逻辑，仅保留以下部分：

- Property: 如 \$table, \$fillable... 等。
- Mutator: 包括 mutator 與 accessor。
- Method: relation 類的 method，如使用 hasMany() 與 belongsTo()。

简化后的 Eloquent 的 model 类我们可以称之为 Eloquent Class，由于继承了 Illuminate\Database\Eloquent\Model 类，所以 Eloquent Class 仍然有很多方法可以操作数据库。

Eloquent Class 代码结构如下：

```
namespace MyBlog;

use Illuminate\Database\Eloquent\Model;
/**
 * MyBlog\User
 *
 * */
class User extends Model {

    /**
     * The database table used by the model.
     *
     * @var string
     */
    protected $table = 'users';

    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = ['name', 'email', 'password'];

    /**
     * The attributes excluded from the model's JSON form.
     *
     * @var array
     */
    protected $hidden = ['password', 'remember_token'];
}
```

而在 ecshopx 中，由于我们采用了 Doctrine ORM，其自带 Repository 模式。在 Doctrine ORM 中也有简化的 model 类：Entity Class。

相比 Eloquent Class，Entity Class 更简单，其只包含 protected 或 private 的属性和 getter 和 setter 方法（在 JAVA 中，这种对象成为值对象）。

Entity Class 代码结构如下：

```
<?php
namespace MyBlog;

use Doctrine\ORM\Mapping as ORM;
/**
 * @ORM\Entity
 * @ORM\Table(name="users")
 */
class User
{
    /**
     * @ORM\Id
     * @ORM\Column(type="integer")
     * @ORM\GeneratedValue
     */
    protected $id;
    /**
     * @ORM\Column(type="string")
     */
    protected $name;
    /**
     * @ORM\Column(type="string")
     */
    protected $email;
    /**
     * @ORM\Column(type="string")
     */
    protected $password;
    protected $remember_token;
    // .. (other code getter setter )
}
```

Repository

初学者常会在 controller 直接调用 model 写数据库逻辑：

```
public function index()
{
    $users = User::where('age', '>', 20)
        ->orderBy('age')
        ->get();

    return view('users.index', compact('users'));
}
```

数据库逻辑是 获取 20 岁以上的用户 。

在中大型项目中，会有几个问题：

- 将数据库逻辑写在 controller，造成 controller 的肥大难以维护。
- 违反 SOLID 的单一职责原则：数据库逻辑不应该写在 controller。
- controller 直接相依赖于model，使得我们无法对 controller 做单元测试。

比较好的方式是使用 repository：

- 将 model 依赖注入到 repository。
- 将数据库逻辑写在 repository。
- 将 repository 依赖注入到 service。

以此原则，我们看下，采用 Eloquent ORM 如何使用 Repository 模式。

UserRepository.php

```
namespace MyBlog\Repositories;

use Doctrine\Common\Collections\Collection;
use MyBlog\User;

class UserRepository
{
    /** @var User 注入的User model */
    protected $user;

    /**
     * UserRepository constructor.
     * @param User $user
     */
    public function __construct(User $user)
    {
        $this->user = $user;
    }

    /**
     * 返回年龄大于$sage的用户
     * @param integer $sage
     * @return Collection
     */
    public function getAgeLargerThan($sage)
    {
        return $this->user
            ->where('age', '>', $sage)
            ->orderBy('age')
            ->get();
    }
}
```

第 8 行

```
/** @var User 注入的User model */  
protected $user;  
  
/**  
 * UserRepository constructor.  
 * @param User $user  
 */  
public function __construct(User $user)  
{  
    $this->user = $user;  
}
```

将依赖的 User model 依赖注入到 UserRepository。

UserController.php

```
<?php
namespace App\Http\Controllers;

use App\Http\Requests;
use MyBlog\Repositories\UserRepository;

class UserController extends Controller
{
    /** @var UserRepository 注入的UserRepository */
    protected $userRepository;

    /**
     * UserController constructor.
     *
     * @param UserRepository $userRepository
     */
    public function __construct(UserRepository $userRepository)
    {
        $this->userRepository = $userRepository;
    }

    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        $users = $this->userRepository
            ->getAgeLargerThan(20);

        return view('users.index', compact('users'));
    }
}
```

将依赖的 UserRepository 依赖注入到 UserController。

26行


```
/**
 * Display a listing of the resource.
 *
 * @return \Illuminate\Http\Response
 */
public function index()
{
    $users = $this->userRepository
        ->getAgeLargerThan(20);

    return view('users.index', compact('users'));
}
```

从原本直接依赖 User model，改成依赖注入的 UserRepository。

改用这种写法，有几个优点：

- 将数据库逻辑写在存储库中，解决控制器肥大问题。
- 符合SOLID的单一职责原则：数据库逻辑写在repository，没写在controller。
- 符合SOLID的依赖转换原则：controller 并非直接相依赖于存储库，而是将存储库依赖注入进controller。

实际上建议repository仅依赖注入于service，而不要直接注入在controller，本范例因为还没介绍到service模式，以便简化起见，所以直接注入于controller。

是否该建立存储库接口？

理论上使用依赖注入时，应该使用接口，不过接口目的在于通过抽象化，方便在后期更换接口的具体实现，以便让程序达到开放封闭的要求，但是实际上要更换存储库的机会不高，除非你有更换资料库的需求，如从MySQL抽换到MongoDB，此时就该建立存储库接口。

不过由于我们使用了依赖注入，将来要从类改成interface也很方便，只要在构造方法的类型提示改成interface即可，维护成本很低，所以在此大可使用的存储库类，用接口反而会造成设计过度，等真正需求来时再重构成接口即可。

采用 Doctrine ORM 如何使用 Repository 模式。

由于 Doctrine ORM 自带 Repository 模式，所以直接使用即可。

UserRepository.php

```
<?php

namespace App\Repositories;

use Doctrine\ORM\EntityRepository;

class UserRepository extends EntityRepository {

    /**
     * 返回年龄大于$sage的用户
     * @param integer $sage
     * @return Collection
     */
    public function getAgeLargerThan($sage)
    {
        return $this->user
            ->where('age', '>', $sage)
            ->orderBy('age')
            ->get();
    }
}
```

UserController.php

```
namespace App\Http\Controllers;

use App\Http\Requests;
use MyBlog\Repositories\UserRepository;

class UserController extends Controller
{
    /** @var UserRepository 注入的UserRepository */
    protected $userRepository;

    /**
     * UserController constructor.
     *
     * @param UserRepository $userRepository
     */
    public function __construct(UserRepository $userRepository)
    {
        $this->userRepository = $userRepository;
    }

    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        $users = $this->userRepository
            ->getAgeLargerThan(20);

        return view('users.index', compact('users'));
    }
}
```

Service

使用 Service 辅助 Controller

若将商业逻辑都写在 controller，会造成 controller 肥大而难以维护，基于SOLID原则，我们应该使用 Service 模式辅助 controller，将相关的商业逻辑封装在不同的 service，方便中大型项目的维护。

商业逻辑

商业逻辑中，常见的如：

- 牵涉到外部行为：如发送Email，使用外部API...
- 使用PHP写的逻辑：如促销规则计算、订单创建等。

若将商业逻辑写在 controller，会造成 controller 肥大，日后难以维护。

Service

牵涉到外部行为

如发 送Email，初学者常会在 controller 直接调用 `Mail::queue()`：

```
public function store(Request $request)
{
    Mail::queue('email.index', $request->all(), function (Message $message) {
        $message->sender(env('MAIL_USERNAME'));
        $message->subject(env('MAIL_SUBJECT'));
        $message->to(env('MAIL_TO_ADDR'));
    });
}
```

`Mail::queue()`只有一行可能无感，但很多外部服务需要一连串 API，甚至还要有 try/catch 处理。

在中大型项目，会有几个问题：

- 将牵涉到外部行为的商业逻辑写在 controller，造成 controller 的肥大难以维护。
- 违反 SOLID 的单一职责原则：外部行为不应该写在 controller。
- controller 直接相依赖于外部行为，使得我们无法对 controller 做单元测试。

比较好的方式是使用 service：

- 将外部行为注入到 service。
- 在 service 使用外部行为。
- 将 service 注入到 controller。

EmailService.php

```
namespace App\Services;

use Illuminate\Mail\Mailer;
use Illuminate\Mail\Message;

class EmailService
{
    /** @var Mailer */
    private $mail;

    /**
     * EmailService constructor.
     * 将依赖的 Mailer 注入到 EmailService。
     * @param Mailer $mail
     */
    public function __construct(Mailer $mail)
    {
        $this->mail = $mail;
    }

    /**
     * 发送邮件
     * 将发送 Email 的商业逻辑写在 send()。
     * 不是使用 Mail facade，而是使用注入的 $this->mail。
     * @param array $request
     */
    public function send(array $request)
    {
        $this->mail->queue('email.index', $request, function (Message $message) {
            $message->sender(env('MAIL_USERNAME'));
            $message->subject(env('MAIL_SUBJECT'));
            $message->to(env('MAIL_TO_ADDR'));
        });
    }
}
```

UserController.php

```
namespace App\Http\Controllers;

use App\Http\Requests;
use Illuminate\Http\Request;
use MyBlog\Services\EmailService;

class UserController extends Controller
{
    /** @var EmailService */
    protected $emailService;

    /**
     * UserController constructor.
     * @param EmailService $emailService
     */
    public function __construct(EmailService $emailService)
    {
        //将依赖的 EmailService 注入到 UserController。
        $this->emailService = $emailService;
    }

    /**
     * Store a newly created resource in storage.
     *
     * @param \Illuminate\Http\Request $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        //从原本直接依赖于 Mail facade，改成相依于注入的 EmailService。
        $this->emailService->send($request->all());
    }
}
```

改用这种写法，有几个优点：

- 将外部行为写在 service，解决 controller 肥大问题。
- 符合 SOLID 的单一职责原则：外部行为写在 service，没写在 controller。
- 符合 SOLID 的依赖反转原则：controller 并非直接相依于 service，而是将 service 依赖注入进 controller。

使用 PHP 写的逻辑：

如根据购买的件数，有不同的折扣，初学者常会在 controller 直接写 if...else 逻辑。

```
public function store(Request $request)
{
    $qty = $request->input('qty');

    $price = 500;

    if ($qty == 1) {
        $discount = 1.0;
    } elseif ($qty == 2) {
        $discount = 0.9;
    } elseif ($qty == 3) {
        $discount = 0.8;
    } else {
        $discount = 0.7;
    }

    $total = $price * $qty * $discount;

    echo($total);
}
```

在中大型项目，会有几个问题：

- 将 PHP 写的商业逻辑直接写在 controller，造成 controller 的肥大难以维护。
- 违反 SOLID 的单一职责原则：商业逻辑不应该写在 controller。
- 违反 SOLID 的单一职责原则：若未来想要改变折扣与加总的算法，都需要改到此 method，也就是说，此 method 同时包含了计算折扣与计算加总的职责，因此违反 SOLID 的单一职责原则。
- 直接写在 controller 的逻辑无法被其他 controller 使用。

比较好的方式是使用 service。

- 将依赖的类注入到 service。
- 在 service 写 PHP 逻辑使用相依物件。
- 将 service 注入到 controller。

OrderService.php

```
namespace App\Services;

class OrderService
{
    /**
     * 计算折扣
     * @param int $qty
     * @return float
     */
    //为了符合 SOLID 的单一职责原则，将计算折扣独立成 getDiscount()，将PHP写的判断逻辑写在里面。
    public function getDiscount($qty)
    {
        if ($qty == 1) {
            return 1.0;
        } elseif ($qty == 2) {
            return 0.9;
        } elseif ($qty == 3) {
            return 0.8;
        } else {
            return 0.7;
        }
    }

    /**
     * 计算最后价格
     * @param integer $qty
     * @param float $discount
     * @return float
     */
    //为了符合 SOLID 的单一职责原则，将计算加总独立成 getTotal()，将PHP写的计算逻辑写在里面。
    public function getTotal($qty, $discount)
    {
        return 500 * $qty * $discount;
    }
}
```

OrderController.php


```

namespace App\Http\Controllers;

use App\Http\Requests;
use App\MyBlog\Services\OrderService;
use Illuminate\Http\Request;

class OrderController extends Controller
{
    /** @var OrderService */
    protected $orderService;

    /**
     * OrderController constructor.
     * @param OrderService $orderService
     */
    public function __construct(OrderService $orderService)
    {
        //将依赖的 OrderService 注入到 UserController。
        $this->orderService = $orderService;
    }

    /**
     * Store a newly created resource in storage.
     * @param \Illuminate\Http\Request $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        $qty = $request->input('qty');
        //将原本的 if...else 逻辑改成调用 OrderService,
        //controller 变得非常干净, 也达到了controller 接收 HTTP request, 调用其他
class 的责任。
        $discount = $this->orderService->getDiscount($qty);
        $total = $this->orderService->getTotal($qty, $discount);

        echo($total);
    }
}

```

改用这种写法，有几个优点：

- 将PHP写的商业逻辑写在 service，解决 controller 肥大问题。
- 符合 SOLID 的单一职责原则：商业逻辑写在 service，没写在 controller。

- 符合 SOLID 的单一职责原则：计算折扣与计算加总价分开在不同 method，且归属于 OrderService，而非 OrderController。
- 符合 SOLID 的依赖反转原则：controller 并非直接相依于 service，而是将 service 依赖注入进 controller。
- 其他 controller 也可以重复使用此段商业逻辑。

Controller

涉及到外部行为

```
public function store(Request $request)
{
    $this->emailService->send($request->all());
}
```

使用 PHP 写的逻辑

```
public function store(Request $request)
{
    $qty = $request->input('qty');

    $discount = $this->orderService->getDiscount($qty);
    $total = $this->orderService->getTotal($qty, $discount);

    echo($total);
}
```

若使用了 service 辅助 controller，再搭配依赖注入与 service container，则 controller 就非常干净，能专心处理 接收HTTP request，调用其他class 的职责了。

Conclusion

- 实际上会有很多 service，须自行依照 SOLID 原则去判断是否该建立 service。
- Service 使得商业逻辑从 controller 中解放，不仅更容易维护、更容易扩展、更容易重复使用，且更容易测试。

Bundle

Laravel (Lumen) 官方建议将所有代码都放到 `app` 目录下面：

- `app` 目录包含应用程序的核心代码。你应用中几乎所有的类都应该放在这里。
- `app` 目录包含额外的各种目录，比如：`Console`, `Http`, 和 `Providers`

按照官方的建议将所有的应用程序代码都放到 `app` 目录里面，随着项目的增大，代码增多，需求逐渐复杂，后期将会导致代码过于复杂，维护成本急剧增高。

借鉴商派 ECOS 框架 APP 机制和 symfony 的 Bundle 系统，我们在 ecshopx 中也实现了一种模块化的机制，我们也称为 Bundle。

一个 Bundle 就是一个模块，其包含模块所有的资源。所有的 Bundle 都位于 `src` 目录下，Bundle 的命名规则为 `业务名称+Bundle`，它的目录结构如下：

文件/目录	描述
Entities	包含本 Bundle 的所有实体类，一个实体类对应一张数据表。可以通过修改实体类来维护表结构
Events	包含事件类
Http	包含控制器，按照Api、FrontApi、ShopApi等分组，每个控制器都有所属版本号
Interfaces	包含本 Bundle 所有接口抽象
Jobs	包含队列任务类
Listeners	目录包含事件的处理类
Providers	包含本 Bundle 的服务提供者
Repositories	处理数据库逻辑，一般和entity表文件一对一。
Services	所有业务处理的逻辑放在此目录，供控制器调用
Traits	本 Bundle 可被复用的逻辑可以抽象为trait放在此目录
README.md	本 Bundle 的说明文件，介绍该 Bundle 的基本功能，主要类和一些设计思想，供其他开发人员参阅

路由

ecshopx 采用 Dingo API 来构建和管理 API。

为了避免与项目路由冲突，dingo/api 将会使用其专属的路由实例。

在 bootstrap/route.php 中定义了API路由的实例：

```
$api = app('Dingo\Api\Routing\Router');
```

路由定义

ecshopx 前端包含了 admin管理端、shop 管理端、小程序、PC 和 H5 等终端，所以在 routes 文件下，我们对路由做了分组：

- admin
- api
- frontapi
- shopapi
- super
- thirdparty

每个 Bundle 的路由根据需求定义在 routes 文件夹下的分组中，路由文件名称为业务名称，路由定义是必须定义版本号，默认为 v1，定义了一条路由就定义了一个接口。

路由定义的语法请参考 Dingo API，以 AftersalesBundle 为例：

定义路由

```
$api->version('v1', function($api) {  
    // 售后相关api  
    $api->group(['namespace' => 'AftersalesBundle\Http\Api\V1\Action', 'middleware' => ['api.auth', 'activated', 'shoplog'], 'providers' => 'jwt'], function($api) {  
        $api->get('/aftersales', ['name' => '获取售后列表', 'as' => 'aftersales.list', 'uses' => 'Aftersales@getAftersalesList']);  
    });  
});
```

其中

```
'uses' => 'Aftersales@getAftersalesList'
```

`Aftersales` 为对用控制器类名，`getAftersalesList` 为方法名。

`AftersalesBundle` 的路由会存在于以下几个分组：

- `routes/api/aftersales.php`
- `routes/frontapi/aftersales.php`
- `routes/shopapi/aftersales.php`

路由注册

在定义路由之后，必须手工注册到 `bootstrap/route.php` 中，才可以使用。

比如 `AftersalesBundle` 新增了一组 `admin` 的路由，我们定义文件路径为：

```
routes/admin/aftersales.php
```

我们需要将此文件加入到 `bootstrap/route.php` 中：

```
<?PHP

...
require __DIR__.'/../routes/admin/aftersales.php';
...
```

控制器

控制器都放在 Http 目录，与 [路由](#) 相对应，控制器也按照终端进行了分组：

- Admin
- Api
- Frontapi
- Shopapi
- Super
- Thirdparty

每个分组下的目录结构如下：

- Api
 - V1
 - Action 控制器实际所在目录
 - Swagger Swagger定义目录

定义控制器

下面是一个基础控制器类的例子。需要注意的是，该控制器继承了 Laravel 的基类控制器。该基类控制器提供了一些便利的方法，比如 middleware 方法，该方法可以为控制器行为添加中间件：

```
namespace AftersalesBundle\Http\Api\V1\Action;

use EspierBundle\Jobs\ExportFileJob;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller as Controller;

use AftersalesBundle\Services\AftersalesService;

use Dingo\Api\Exception\ResourceException;

use EspierBundle\Traits\GetExportServiceTraits;

class Aftersales extends Controller
{
    use GetExportServiceTraits;

    public function getAftersalesDetail($aftersales_bn)
    {
        $companyId = app('auth')->user()->get('company_id');
        $aftersalesService = new AftersalesService();
        $result = $aftersalesService->getAftersalesInfo($companyId, $aftersal
es_bn);

        return $this->response->array($result);
    }
}
```

定义 API 文档

在 EcshopX 中每一个控制器方法都是一个 api 都是一个接口，接口的出参和入参都是通过 注释来定义的，采用 Swagger 格式来定义。

控制器分组下的每一个版本都是一组 API，定义一组 api 需要两步，以 AftersalesBundle 举例如下：

1、在 Swagger 目录下定义分组信息

定义 API 基本信息

```
//AftersalesBundle/Http/Api/V1/Swagger/Info.php
<?php
/**
 * @SWG\Swagger(
 *     schemes={"http"},
 *     host="localhost",
 *     basePath="/api",
 *     @SWG\Info(
 *         version="1.0",
 *         title="售后单商家端调用接口",
 *         description="售后单",
 *     ),
 *     produces={"application/json"},
 * )
 */
```

定义 API Tag

```
//AftersalesBundle/Http/Api/V1/Swagger/Tags.php
<?php
/**
 * @SWG\Tag(
 *     name="aftersales",
 *     description="售后单",
 * )
 */
```

根据需求定义接口所需数据对象：


```
//AftersalesBundle/Http/Api/V1/Swagger/AftersalesErrorResponses.php
```

```
<?php
namespace AftersalesBundle\Http\Api\V1\Swagger;

/**
 * @SWG\Definition(type="object")
 */
class AftersalesErrorResponses
{
    /**
     * @SWG\Property
     * @var string
     */
    public $message;

    /**
     * @SWG\Property
     * @var string
     */
    public $errors;

    /**
     * @SWG\Property(format="int32")
     * @var int
     */
    public $code;

    /**
     * @SWG\Property(format="int32")
     * @var int
     */
    public $status_code;

    /**
     * @SWG\Property
     * @var string
     */
    public $debug;
}
```

2、在 Action 目录中的控制器的方法定义接口

```

/**
 * @SWG\Get(
 *   path="/aftersales/{$aftersales_bn}",
 *   summary="获取售后单详情",
 *   tags={"aftersales"},
 *   description="获取售后单详情",
 *   operationId="getAftersalesDetail",
 *   @SWG\Parameter(
 *     name="Authorization",
 *     in="header",
 *     description="JWT验证token",
 *     required=true,
 *     type="string",
 *   ),
 *   @SWG\Parameter(
 *     name="aftersales_bn",
 *     in="path",
 *     description="售后单号",
 *     required=true,
 *     type="integer",
 *   ),
 *   @SWG\Response(
 *     response=200,
 *     description="成功返回结构",
 *   ),
 *   @SWG\Response( response="default", description="错误返回结构", @SWG\
Schema( type="array", @SWG\Items(ref="#/definitions/AftersalesErrorResponses")
) )
 * )
 */
public function getAftersalesDetail($aftersales_bn)
{
    $companyId = app('auth')->user()->get('company_id');
    $aftersalesService = new AftersalesService();
    $result = $aftersalesService->getAftersalesInfo($companyId, $aftersal
es_bn);

    return $this->response->array($result);
}

```

Swagger 定义参数详见 [OpenAPI Specification](#)

接口文档生成和接口测试

在通过 Swagger 注释的方式定义好接口文档后，可以通过本地开发环境查看和测试接口。

一、将 Swagger UI的资源拷贝到public目录中

执行以下命令：

```
php artisan api:swagger --setup
```

二、生成指定目录的Swagger API Josn

以 AftersalesBundle 举例如下：

```
php artisan api:swagger --output=src/AftersalesBundle/Http/Api/V1
```

执行完命令后，默认会在 `storage/app/apidocs` 目录下生成类似 `售后单商家端调用接口 [1.0].json` 文件。

可通过在 `.env` 中新增配置改变存储目录名称

```
SWAGGER_STORAGE_DIR=apidocs
```

三、启动服务查看和测试接口

可以通过 PHP 内置 server 快速启动

```
php -S 0.0.0.0:8005 -t public/
```

访问地址为：<http://127.0.0.1:8085/api-doc>

本机开发时，还需要将测试接口的地址修改为本地环境：

```
SWAGGER_API_HOST=127.0.0.1:8080  
SWAGGER_API_BASE_PATH=/api
```

使用技巧：

在开发过程中可以，每次修改接口文档都需要重新生成接口文档，可以这样使用命令：

```
php artisan api:swagger --output=src/AftersalesBundle/Http/Api/V1 && php -S 0.0.0.0:8005 -t public/
```

快速入门

对于任何应用程序来说，一个最常见和最具挑战的任务，就是从数据库中读取和持久化数据信息。在 Ecshopx 中我们采用 Doctrine ORM 替代 Laravel 自带的 Eloquent ORM，前者更灵活，且自带 [Repository 模式](#)。本节将为大家介绍如何使用 Doctrine ORM。

几个概念

什么是 Doctrine ORM？

Doctrine ORM 是 PHP 7.1+ 的对象关系映射器（ORM），可以将PHP对象持久化到数据库中。它以Data Mapper模式为核心，旨在将您的业务逻辑与数据库中的持久化完全分离。

什么是 Entities？

实体是简单的PHP对象，其含可持久化的属性，可持久属性是实体的一个实例变量，通过 Doctrine的数据映射功能可以将其保存到数据库中或从数据库中检索出来。实体类不需要扩展任何抽象基类或接口。

实体类不能为final，尽管它可以包含final方法。

什么是 DQL？

DQL 是 Doctrine Query Language 的缩写，代表 Doctrine 查询语言，并且是对象查询语言（Object Query Language）的派生类，它与 Hibernate 查询语言（HQL）或Java持久化查询语言（JPQL）非常相似。

对于初学者来说，常见的错误是将DQL误认为只是某种形式的SQL，因此尝试在查询中使用表名和列名或将任意表连接在一起。您需要考虑DQL作为对象模型而不是关系模式的查询语言。

定义数据库 Schema

在 ECOS 中可以在 dbschema 文件夹下，使用数组定义数据库 Schema，laravel 中可以书写数据库迁移（database migration）文件，而在 ecshopx 中，则需要定义 Doctrine ORM 的实体类来完成数据库 Schema定义。

我们从最简单的实体 `Product` 开始。创建一个

`src/DemoBundle/Entities/Product.php` 包含 `Product` 实体定义的文件：

```
<?php
// src/DemoBundle/Entities/Product.php
class Product
{
    /**
     * @var int
     */
    private $id;
    /**
     * @var string
     */
    private $name;
}
```

创建实体类时，所有字段都应为 `private`。

这个类仅仅是一个普通的 `php` 类，还不能称之为 `Entity` 类。我们需要通过为这个类添加注解使之成为 `Entity` 类。

```
<?php
// src/DemoBundle/Entities/Product.php
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="products")
 */
class Product
{
    /**
     * @ORM\Id
     * @ORM\Column(type="integer")
     * @ORM\GeneratedValue
     */
    protected $id;
    /**
     * @ORM\Column(type="string")
     */
    protected $name;
}
```

在添加完注解之后，可以通过以下命令，生成 `getter` 和 `setter` 方法：

```
php artisan doctrine:generate:entities --filter=DemoBundle
```

生成之后，一个实体类就完整了，完整的实体类如下：

```
// src/DemoBundle/Entities/Product.php
<?php
namespace DemoBundle\Entities;

use Doctrine\ORM\Mapping as ORM;
/**
 * @ORM\Entity
 * @ORM\Table(name="products")
 */
class Product
{
    /**
     * @ORM\Id
     * @ORM\Column(type="integer")
     * @ORM\GeneratedValue
     */
    protected $id;
    /**
     * @ORM\Column(type="string")
     */
    protected $name;

    /**
     * Get id.
     *
     * @return int
     */
    public function getId()
    {
        return $this->id;
    }

    /**
     * Set name.
     *
     * @param string $name
     *
     * @return Product
     */
    public function setName($name)
```

```
{
    $this->name = $name;

    return $this;
}

/**
 * Get name.
 *
 * @return string
 */
public function getName()
{
    return $this->name;
}
}
```

生成数据表

一个实体类定义了一张数据表，如何将数据表同步到数据库呢？
首先需要先运行：

```
php artisan doctrine:migrations:diff
```

运行以上命令成功后，会生成数据库迁移文件：

```
Generated new migration class to "Version20191225161708" from schema differences.
```

文件的路径在 `database/migrations/Version20191225161708.php`

生成之后，即可运行一下命令，将数据表写入到数据库：

```
php artisan doctrine:migrations:migrate
```

持久化对象到数据库

现在我们有Product实体和与之映射的product数据库表。接下来我们演示如何把数据持久化到数据库里。

为了方便演示，我们采用测试用例的方式来演示此功能：

```
//src/DemoBundle/Tests/ProductTest.php
<?php
namespace DemoBundle\Tests;

use EspierBundle\Services\TestBaseService;
use DemoBundle\Entities\Product;

class RepositoryFactoryTest extends TestBaseService
{
    /**
     * A basic test example.
     *
     * @return void
     */
    public function testCreateProduct()
    {
        $product = new Product();
        $product->setName("EcshopX");

        //这一行取出了Doctrine的 entity manager 对象，它负责处理数据库的持久化（写入
        //）和取出对象的过程。
        $em = app('registry')->getManager('default');

        // 告诉Doctrine你希望（最终）存储Product对象（还没有语句执行）
        $em->persist($product);

        // 真正执行语句（如，INSERT 查询）
        $em->flush();

        $this->assertNotNull($product->getId());
    }
}
```

当 `flush()` 方法被调用时，Doctrine会遍历它管理的所有对象以确定是否需要被持久化到数据库。本例中，`$product` 对象的数据在库中并不存在，因此entity manager要执行 `INSERT` 请求，在 `product` 表中创建一个新行。

从数据库中获取对象

从数据库中取回对象就更简单了：


```
//src/DemoBundle/Tests/ProductTest.php
public function testGetProduct()
{
    $productId = 1;
    //这一行取出了Doctrine的 entity manager 对象，它负责处理数据库的持久化（写入）和
    取出对象的过程。
    $em = app('registry')->getManager('default');

    $productRepository = $em->getRepository(Product::class);

    $product = $productRepository->find($productId);

    $this->assertEquals($productId, $product->getId());
}
```

当你要查询某个特定类型的对象时，你总是要使用它的“respository”。你可以认为Respository是一个PHP类，它的唯一工作就是帮助你从那个特定的类中取出entity。对于一个entity类，要访问其宝库，通过：

```
app('registry')->getManager('default')->getRepository(Product::class);
```

一旦有了Repository对象，你就可以访问它的全部有用的方法了。

```
$repository = app('registry')->getManager('default')->getRepository(Product::
class);
// 通过主键（通常是id）查询一件产品
$product = $repository->find($productId);

// 动态方法名称，基于字段的值来找到一件产品
$product = $repository->findOneById($productId);
$product = $repository->findOneByName('Keyboard');

// 动态方法名称，基于字段值来找出一组产品
$products = $repository->findByPrice(19.99);

// find *all* products / 查出 *全部* 产品
$products = $repository->findAll();
```

你也可以有效利用 `findBy` 和 `findOneBy` 方法，基于多个条件来轻松获取对象：

```
$repository = app('registry')->getManager('default')->getRepository(Product::class);

// 查询一件产品，要匹配给定的名称和价格
$product = $repository->findOneBy(
    array('name' => 'Keyboard', 'price' => 19.99)
);

// 查询多件产品，要匹配给定的名称和价格
$products = $repository->findBy(
    array('name' => 'Keyboard'),
    array('price' => 'ASC')
);
```

对象更新

一旦从Doctrine中获取了一个对象，更新它就很容易了：

```
public function testUpdateProduct()
{
    $productId = 1;
    //这一行取出了Doctrine的 entity manager 对象，它负责处理数据库的持久化（写入）和
    取出对象的过程。
    $em = app('registry')->getManager('default');
    $productRepository = $em->getRepository(Product::class);
    $product = $productRepository->find($productId);

    $newName = "Ecshopx 2.0";

    $product->setName($newName);
    $em->flush();

    //判断数据是否更新成功
    $newProduct = $productRepository->find($productId);
    $this->assertEquals($newName, $newProduct->getName());
}
```

更新一个对象包括三步：

- 1、从Doctrine中取出对象；
- 2、修改对象；
- 3、调用entity manager的 flush() 方法。

注意调用 `$em->persist($product)` 是不必要的。回想一下，这个方法只是告诉Doctrine去管理或者“观察” `$product` 对象。此处，因为你已经取到了 `$product` 对象了，它已经被管理了。

删除对象

删除一个对象十分类似，但需要从entity manager调用 `remove()` 方法：

```
$em->remove($product);  
$em->flush();
```

你可能已经预期，`remove()` 方法通知Doctrine你想从数据库中删除指定的entity。真正的DELETE 查询不会被真正执行，直到 `flush()` 方法被调用。

对象查询

你已经看到 repository 对象是如何让你执行一些基本查询而毋须做任何工作了：

```
$repository = app('registry')->getManager('default')->getRepository(Product::class);  
  
$product = $repository->find($productId);  
$product = $repository->findOneByName('Keyboard');
```

当然，Doctrine 也允许你使用Doctrine Query Language(DQL)来写一些复杂的查询，DQL类似于SQL，只是它用于查询一个或者多个entity类的对象(如 `product`)，而SQL则是查询一个数据表中的行（如 `product`）。

在Doctrine中查询时，你有两个主要选择：

- 编写纯正的Doctrine查询（DQL）
- 使用Doctrine的Query Builder

使用DQL进行对象查询

假设你要查询价格高于 19.99 的产品，并且按价格从低到高排列。你可以使用DQL，Doctrine 中类似原生SQL的语法，来构造一个用于此场景的查询：

```
$em = $this->getDoctrine()->getManager();
$query = $em->createQuery(
    'SELECT p
    FROM DemoBundle\Entities\Product p
    WHERE p.price > :price
    ORDER BY p.price ASC'
)->setParameter('price', 19.99);

$products = $query->getResult();
```

如果你习惯了写SQL，那么对于DQL也会非常自然。它们之间最大的不同就是你需要就“select PHP对象”来进行思考，而不是数据表的行。正因为如此，你要从 Product 这个 entity 来 select，然后给entity一个 p 的别名。

注意 `setParameter()` 方法。当使用 Doctrine 时，通过“占位符”来设置任意的外部值（上面例子的 :price），是一个好办法，因为它可以防止SQL注入攻击。

getResult() 方法返回一个结果数组。要得到一个结果，可以使用getSingleResult()（这个方法在没有结果时会抛出一个异常）或者 getOneOrNullResult()：

```
$product = $query->setMaxResults(1)->getOneOrNullResult();
```

DQL语法强大到令人难以置信，允许轻松地在entity之间进行join（稍后会覆盖relations）和group等。参考 [Doctrine Query Language](#) 文档以了解更多。

使用Doctrine's Query Builder进行对象查询

除了去写DQL，你还可以使用一个非常有用的QueryBuilder对象，来构建查询 SQL。当你的查询取决于动态条件时，这很有用，因为随着你的连接字符串不断增加，DQL代码会越来越难以阅读：

```
$em = app('registry')->getManager('default');
$productRepository = $em->getRepository(Product::class);

// createQueryBuilder() 自动从 AppBundle:Product 进行 select 并赋予 p 假名
$query = $productRepository->createQueryBuilder('p')
    ->where('p.price > :price')
    ->setParameter('price', '19.99')
    ->orderBy('p.price', 'ASC')
    ->getQuery();

$products = $query->getResult();
// 要得到一个结果:
$product = $query->setMaxResults(1)->getOneOrNullResult();
```

QueryBuilder对象包含了创建查询时的所有必要方法。通过调用getQuery()方法，query builder将返回一个标准的Query对象，可用于取得请求的结果集。

Query Builder更多信息，参考 Doctrine 的 [Query Builder](#)文档。

Repository

在上一章，所有的查询是直接写在你的测试用例的方法中的。但对于程序的组织来说，为了隔离，复用和测试这些查询 Doctrine 提供了一个专门的 repository 类，它允许你保存所有查询逻辑到一个中心位置。

所有的 repository 类，都应该放到 Bundle 的 Repositories 文件夹中。

要定义 repository 类，有两种方法。

一、在实体类中添加 repository 类的声明：

```
// src/DemoBundle/Entities/Product.php
<?php
namespace DemoBundle\Entities;

use Doctrine\ORM\Mapping as ORM;
/**
 * @ORM\Entity
 * @ORM\Table(name="products")
 * @ORM\Entity(repositoryClass="DemoBundle\Repositories\ProductRepository")
 */
class Product
{
    //...
}
```

新建 Repositories 文件夹，在 Repositories 下新建 `ProductRepository` 类：

```
//src/DemoBundle/Repositories/ProductRepository.php
<?php
namespace DemoBundle\Repositories;

use Doctrine\ORM\EntityRepository;

class ProductRepository extends EntityRepository
{
    public function findAllOrderedByName()
    {
        return $this->getEntityManager()
            ->createQuery(
                'SELECT p FROM DemoBundle\Entities\Product p ORDER BY p.name
ASC'
            )
            ->getResult();
    }
}
```

在 Repository 类中可以通过 `$this->getEntityManager()` 方法类获取entity管理。

你就可以像使用默认的方法一样使用这个新定义的方法了：

```
//src/DemoBundle/Tests/ProductTest.php
/**
 * A basic test example.
 *
 * @return void
 */
public function testProductRepository()
{
    //这一行取出了Doctrine的 entity manager 对象，它负责处理数据库的持久化（写入）和
    取出对象的过程。
    $em = app('registry')->getManager('default');

    $productRepository = $em->getRepository(Product::class);

    $products = $productRepository->findAllOrderedByName();

    $this->assertEquals(count($productRepository->findAll()), count($products
));
}
```

当使用一个自定义的 repository 类时，你依然可以访问原有的默认查找方法，比如find()和findAll()等。

使用注解的方式 Doctrine 会在调用 `$em->getRepository(Product::class);` 时，通过实体的注解找到实体对应的 repository 类。这种方式是 Doctrine 默认提供的，其有一个确定就是：这样定义一个实体只能有一个 repository，将程序所有自定义查询都放到一个类里面，随着项目的发展，势必会导致 repository 类的肥大。这显然违反 SOLID 的单一职责原则。

二、通过在 repository 类中声明关联的实体

与第一种方法相反，这种方式是通过在 repository 类中声明所关联的实体，来实现 repository 类的声明。由于 repository 类初始化时，需要传入 Entity Manager 和实体类的 ClassMate，为了方便初始化 repository 类，需要引入一个 Traits。

在 Repositories 文件夹下新建一个 ProductPriceRepository 类，用来处理价格相关的查询：

```
//src/DemoBundle/Repositories/ProductPriceRepository.php
<?php
namespace DemoBundle\Repositories;

use Doctrine\ORM\EntityRepository;
use EspierBundle\Traits\RepositoryFactory;
use DemoBundle\Entities\Product;

class ProductPriceRepository extends EntityRepository
{
    public static $entityClass = Product::class;

    use RepositoryFactory;

    public function findAllOrderedByName()
    {
        return $this->getEntityManager()
            ->createQuery(
                'SELECT p FROM DemoBundle\Entities\Product p ORDER BY p.name
ASC'
            )
            ->getResult();
    }
}
```

使用方式如下：


```
//src/DemoBundle/Tests/ProductTest.php

use DemoBundle\Repositories\ProductPriceRepository;

//...

public function testProductPriceRepository()
{
    $productRepository = ProductPriceRepository::instance();

    $products = $productRepository->findAllOrderedByName();

    $this->assertEquals(count($productRepository->findAll()), count($products));
}
```

CRUD Repository

为提高开发速度，降低代码重复率，新增 2 个 CURD Trait： `DBALCrudRepository` 和 `ORMCrudRepository`。

在 Doctrine ORM 中 ORM 依赖于 DBAL 层。ORM 调用 DBAL 并将 DBAL 返回的数组结构转化为对象。由于多了这一层转化，开发人员在使用时，可以根据使用场景和项目规模决定使用哪个 Trait。

两者都是基于 Doctrine 提供的 `QueryBuilder` 实现，不同的是：

- `DBALCrudRepository` 基于 SQL Query Builder（`Doctrine\DBAL\Query\QueryBuilder`）
- `ORMCrudRepository` 基于 DQL Query Builder（`Doctrine\ORM\Query\QueryBuilder`）

DBALCrudRepository

`DBALCrudRepository` 基于 SQL Query Builder（`Doctrine\DBAL\Query\QueryBuilder`），其结果返回为数组，性能比 `ORMCrudRepository` 略高。

```
trait DBALCrudRepository{
    public function save(array &$amp;data)
    public function create(array &$amp;data) {
    public function batchUpdate(array $filter, array $data)
    public function batchDelete(array $filter)
    public function count(array $filter=[])
    public function getList($cols='*', array $filter=[], $page = 1, $pageSize
    = 100, $orderBy = [])
}
```

使用方法如下：

```
<?php
namespace ItemBundle\Repositories;

use ItemBundle\Entities\ItemType;
use Doctrine\ORM\EntityRepository;
use EspierBundle\Traits\DBALCrudRepository;

class ItemTypeRepository extends EntityRepository
{
    use DBALCrudRepository;
}
```

ORMCrudRepository

ORMCrudRepository 基于 DQL Query Builder (Doctrine\ORM\QueryBuilder)

```
public function save($entity)
public function delete($entity)
public function create(array $data) {
public function batchUpdate(array $filter, array $data)
public function batchDelete(array $filter)
public function count(array $filter=[])
public function getList(array $filter=[], $page = 1, $pageSize = 100, $orderBy = [])
```

ORMCrudRepository 与 DBALCrudRepository 不同是 save 和 delete 方法，其输入为实体类，而不是数组。

使用方法如下：

```
<?php
namespace ItemBundle\Repositories;

use ItemBundle\Entities\ItemType;
use Doctrine\ORM\EntityRepository;
use EspierBundle\Traits\ORMCrudRepository;
class ItemTypeRepository extends EntityRepository
{
    use ORMCrudRepository;
}
```

DoctrineArrayFilter

为了兼容现有代码和大家在 ECOS 中的使用习惯，在batchUpdate、batchDelete、count和getList中的 filter数组统一由 DoctrineArrayFilter 来解析。除支持 字段|操作符 外，还支持多层嵌套的 OR 和 AND 结构。

支持结构

and

```
$filter = [  
    'name|eq'=>'100',  
    'name|neq'=>'100',  
    'name|lt'=>'100',  
    'name|lte'=>'100',  
    'name|gt'=>'100',  
    'name|gte'=>'100',  
    'name|isNull'=>',  
    'name|isNotNull'=>',  
    'name|like'=>'%bar%',  
    'name|notLike'=>'%foo',  
    'name|in'=>[100,1000],  
    'name|notIn'=>[1100,200],  
];
```

生成where 条件如下：

```
(name = '100') AND (name <> '100') AND (name < '100') AND (name <= '100') AND  
(name > '100') AND (name >= '100') AND (name IS NULL) AND (name IS NOT NULL)  
AND (name LIKE '%bar%') AND (name NOT LIKE '%foo') AND (name IN ('100', '100  
0')) AND (name NOT IN ('1100', '200'))
```

OR

```
$filter = [  
    'OR'=>[  
        'name_or_or'=>1000,  
        'name_or_or2'=>1000,  
    ],  
];
```

生成where 条件如下:

```
(name_or_or = '1000') OR (name_or_or2 = '1000')
```

多层嵌套

```
$filter1 = [  
  'AND'=>[  
    'name_and1|like'=>'zhang',  
    'name_and2|isNull'=>',  
    'OR'=>[  
      'name_or1|in'=>[100, 2020],  
      'name_or2|notIn'=>[100, 2020],  
      'AND'=>[  
        'name_or_or'=>1000,  
        'name_or_or2'=>1000,  
      ],  
    ],  
  ],  
  'OR'=>[  
    'name_or1|in'=>[100, 2020],  
    'name_or2|notIn'=>[100, 2020],  
    'AND'=>[  
      'name_or_or'=>1000,  
      'name_or_or2'=>1000,  
    ],  
  ],  
];
```

生成where 条件如下:

```
((name_and1 LIKE 'zhang') AND (name_and2 IS NULL) AND ((name_or1 IN ('100',  
'2020')) OR (name_or2 NOT IN ('100', '2020')) OR ((name_or_or = '1000') AND (  
name_or_or2 = '1000')))) AND ((name_or1 IN ('100', '2020')) OR (name_or2 NOT  
IN ('100', '2020')) OR ((name_or_or = '1000') AND (name_or_or2 = '1000')))
```

支持操作符

操作符	数据库含义
eq	=
neq	<>
lt	<
lte	<=
gt	>
gte	>=
in	in
notIn	not in
isNull	name IS NULL
isNotNull	name IS NOT NULL
like	LIKE
notLike	NOT LIKE

事件系统

事件系统介绍

Laravel 为事件提供了一个简单的观察者实现，允许你在应用中订阅和监听各种发生的事件。在 Laravel 中事件类通常放在 `app/Events` 目录下，这些事件类的监听器则放在 `app/Listeners` 目录下。而在 Ecshopx 中事件和监听器则放在 Bundle 对应的目录中。

事件系统为应用各个方面的解耦提供了非常棒的方法，因为单个事件可以拥有多个互不依赖的监听器。

举个例子，在电商系统中，订单完成时，可能希望向用户发送短信通知或者微信通知，同时还需要处理一些订单相关的逻辑。你可以简单地发起一个可以被监听器接收并转化为短信通知的 `TradeFinishEvent` 事件，而不是将订单处理代码和发送短信通知代码耦合在一起。

定义事件

事件类是一个保存与事件相关信息的容器。它保存在 Bundle 的 `Events` 目录中。例如，假设我们定义一个 `TradeFinishEvent` 事件，其接收一个 `Trade Entity` 对象：

```
<?php

namespace OrdersBundle\Events;

use App\Events\Event;

class TradeFinishEvent extends Event
{
    public $entities;

    /**
     * Create a new event instance.
     *
     * @return void
     */
    public function __construct($eventData)
    {
        $this->entities = $eventData;
    }
}
```

如你所见，这个事件类中没有包含其它逻辑。它只是一个 Trade 的实例的容器。

定义监听器

事件监听器是一个特殊的 PHP 类，其包含了一个接受事件的 handle 方法，handle 方法中加入事件的类型提示。它保存在 Bundle 的 Listeners 目录中。例如，在订单完成时，我们会给用户发送一个微信的模板消息。

```
<?php

namespace OrdersBundle\Listeners;

use OrdersBundle\Events\TradeFinishEvent;

class TradeFinishWxaTemplateMsg {

    /**
     * Handle the event.
     *
     * @param TradeFinishEvent $event
     * @return void
     */
    public function handle(TradeFinishEvent $event)
    {
        //...
    }
}
```

注册事件和监听器

现在我们定义了一个订单完成的事件 TradeFinishEvent 和一个订单完成时的监听器 TradeFinishWxaTemplateMsg，现在我们需要注册事件和监听器。

在 Laravel 中事件和监听器注册是在 EventServiceProvider 类中完成的，在 ecshopx 中，我们可以把它们放在每个 Bundle 的 Provider 中。Provider 放在 Providers 目录下。


```
<?php

namespace OrdersBundle\Providers;

use Laravel\Lumen\Providers\EventServiceProvider as ServiceProvider;

class TradeFinishServiceProvider extends ServiceProvider
{
    /**
     * The event listener mappings for the application.
     *
     * @var array
     */
    protected $listen = [
        'OrdersBundle\Events\TradeFinishEvent' => [
            'OrdersBundle\Listeners\TradeFinishWxaTemplateMsg',
        ],
    ];
}
```

其中，`listen` 属性包含了所有事件 (键) 以及事件对应的监听器 (值) 的数组。当然，你可以根据应用的需要，添加多个事件到 `listen` 属性包含的数组中。

在注册完成之后，还需要将 `TradeFinishServiceProvider` 在 `bootstrap/app.php` 中：

```
$app->register(OrdersBundle\Providers\TradeFinishServiceProvider::class);
```

分发事件

如果要分发事件，你可以将事件实例传递给辅助函数 `event`。该辅助函数将会把事件分发到所有该事件相应的已经注册了的监听器上。`event` 辅助函数可以全局使用，你可以在应用中的任何位置进行调用：

```
event(new TradeFinishEvent($eventsParams));
```

事件监听器队列（异步事件）

如果你的监听器中要执行诸如发送电子邮件或发出 HTTP 请求之类的耗时任务，你可以将任务丢给队列处理。在开始使用队列监听器之前，请确保在你的服务器或者本地开发环境中能够配置队列并启动一个队列监听器。

要指定监听器启动队列，你可以在监听器类中实现 `ShouldQueue` 接口。同时需要继承 `EspierBundle\Listeners\BaseListeners` 类：

```
<?php

namespace SystemLinkBundle\Listeners;

// use OrdersBundle\Events\TradeFinishEvent;
use Illuminate\Contracts\Queue\ShouldQueue;
use EspierBundle\Listeners\BaseListeners;

class TradeAftersaleCancelSendOme extends BaseListeners implements ShouldQueue
{

    protected $queue = 'default';

    /**
     * Handle the event.
     *
     * @param TradeFinishEvent $event
     * @return void
     */
    public function handle(TradeAftersalesCancelEvent $event)
    {

        return true;
    }
}
```

现在，当这个监听器被事件调用时，事件调度器会自动使用 Laravel 的队列系统自动排队。如果在队列中执行监听器时没有抛出异常，任务会在执行完成后自动从队列中删除。

自定义队列连接 & 队列名称

如果你想要自定义事件监听器所使用的队列的连接和名称，你可以在监听器类中定义 `$connection`，`$queue` 或 `$delay` 属性：

```
<?php

namespace App\Listeners;

use App\Events\OrderShipped;
use Illuminate\Contracts\Queue\ShouldQueue;

class SendShipmentNotification implements ShouldQueue
{
    /**
     * 任务连接名称。
     *
     * @var string|null
     */
    public $connection = 'sqs';

    /**
     * 任务发送到的队列的名称。
     *
     * @var string|null
     */
    public $queue = 'listeners';

    /**
     * 处理任务的延迟时间。
     *
     * @var int
     */
    public $delay = 60;
}
```

处理失败任务

有时事件监听器的队列任务可能会失败。如果监听器的队列任务超过了队列中定义的最大尝试次数，则会在监听器上调用 `failed` 方法。`failed` 方法接收事件实例和导致失败的异常作为参数：

```
<?php

namespace App\Listeners;

use App\Events\OrderShipped;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;

class SendShipmentNotification implements ShouldQueue
{
    use InteractsWithQueue;

    /**
     * 处理事件.
     *
     * @param \App\Events\OrderShipped $event
     * @return void
     */
    public function handle(OrderShipped $event)
    {
        //
    }

    /**
     * 处理失败任务
     *
     * @param \App\Events\OrderShipped $event
     * @param \Exception $exception
     * @return void
     */
    public function failed(OrderShipped $event, $exception)
    {
        //
    }
}
```

消息队列

什么是消息队列？

消息队列，一般我们会简称它为MQ(Message Queue)，队列是一种先进先出的数据结构。可以简单理解为：把要传输的数据放在队列中。

目前使用较多的消息队列中间件有ActiveMQ，RabbitMQ，ZeroMQ，Kafka，MetaMQ，RocketMQ。

主要解决应用解耦，异步消息，流量削锋等问题。

消息队列中有四个重要的角色：

- 消息：可被其他业务使用的数据
- 队列：用于存储消息
- 生产者：生产消息发送到消息队列中
- 消费者：从消息队列中取消息

这些角色在不同的消息中间件中，有不同的实现方式，使用方式也不同，好在 Laravel 队列为不同的后台队列服务提供统一的 API，例如 RabbitMQ，Amazon SQS，Redis，甚至其他基于关系型数据库的队列。

配置

与队列相关的配置都放在 `config/queue.php` 中。

`connections` 这个选项给 RabbitMQ，Beanstalk，或者 Redis 这样的后端服务定义了一个特有的 连接。每个 连接 可以多多个 队列。

需要注意的是 `config/queue.php` 中，每个 连接 都包含了一个 `queue` 属性。队列任务被发给指定连接的时候会被分发到 `queue` 属性和指定连接相同的队列中。换句话说，如果你分发任务的时候没有定义分配到哪个队列，那么它就会被放到连接配置中 `queue` 属性所定义的默认队列中

`default` 选项定义了默认的是哪个连接。

创建任务类

所有的任务类，都是放在每个 Bundle 的 Jobs 目录下。任务类需要继承

`Illuminate\Contracts\Queue\ShouldQueue` 接口，这意味着这个任务将会被推送到队列中，而不是同步执行。

任务类包含两个方法和一些属性，下面看下产品中的一个例子：

```
<?php
namespace MembersBundle\Jobs;

use CompanysBundle\Services\CompanysService;
use PromotionsBundle\Services\SmsDriver\ShopexSmsClient;
use PromotionsBundle\Services\SmsService;
use EspierBundle\Jobs\Job;

class GroupSendSms extends Job
{
    public $smsData;
    public function __construct($smsData)
    {
        $this->smsData = $smsData;
    }

    public function handle()
    {
        $smsData = $this->smsData;
        try {
            $companyId = $smsData['company_id'];
            $mobiles = $smsData['send_to_phones'];
            $content = $smsData['sms_content'];

            app('log')->debug('短信群发1: fan-out =>'.$companyId);
            $companysService = new CompanysService();
            $shopexUid = $companysService->getPassportUidByCompanyId($companyId);

            app('log')->debug('短信群发2: fan-out =>'.$shopexUid);
            $smsService = new SmsService(new ShopexSmsClient($companyId, $shopexUid));

            $smsService->sendContent($companyId, $mobiles, $content, 'fan-out');
        } catch (\Exception $e) {
            app('log')->debug('短信群发失败: fan-out =>'.var_export($e->getMessage(), 1));
        }
    }
}
```

- `__construct` : 构造方法用来初始化任务所需要的数据
- `handle` : 队列执行时所调用的方法

在队列处理任务时，会调用 `handle` 方法，而这里我们也可以通过 `handle` 方法的参数类型提示，让 `Laravel` 的 `服务容器` 自动注入依赖对象。

比如，上面例子中的 `handle` 方法中有这样一段代码 `$companysService = new CompanysService();` 我们可以这样改造：

```
public function handle(CompanysService $companysService)
{
    $smsData = $this->smsData;
    try {
        $companyId = $smsData['company_id'];
        $mobiles = $smsData['send_to_phones'];
        $content = $smsData['sms_content'];

        app('log')->debug('短信群发1: fan-out =>'.$companyId);
        $shopexUid = $companysService->getPassportUidByCompanyId($companyId);

        app('log')->debug('短信群发2: fan-out =>'.$shopexUid);
        $smsService = new SmsService(new ShopexSmsClient($companyId, $shopexUid));

        $smsService->sendContent($companyId, $mobiles, $content, 'fan-out');
    } catch (\Exception $e) {
        app('log')->debug('短信群发失败: fan-out =>'.var_export($e->getMessage(), 1));
    }
}
```

这样写 `handle(CompanysService $companysService)` 在执行任务时 `Laravel` 会自动初始化 `CompanysService` 类，并注入 `handle` 中。

分发任务

分发任务分为两步：先初始化任务类，然后再使用辅助方法`dispatch()`分发任务。

```
public function smsSends(Request $request)
{
    $inputdata = $request->all('mobile', 'sms_content');

    $memberSmsLogService = new MemberSmsLogService();
    $params['company_id'] = app('auth')->user()->get('company_id');
    $params['operator'] = '管理员';
    $params['send_to_phones'] = $inputdata['mobile'];
    $params['sms_content'] = $inputdata['sms_content'];
    $result = $memberSmsLogService->create($params);
    //分发队列
    $job = (new GroupSendSms($params))->onQueue('sms');
    dispatch($job);

    return $this->response->array($result);
}
```

dispatch() 辅助方法的具体实现如下：

```
function dispatch($job)
{
    return app(Illuminate\Contracts\Bus\Dispatcher::class)->dispatch($job);
}
```

同步调度

如果您想立即（同步）执行队列任务，可以使用 `dispatchNow` 方法。使用此方法时，队列任务将不会排队，并立即在当前进程中运行：

```
app(Illuminate\Contracts\Bus\Dispatcher::class)->dispatchNow($job);
```

运行队列处理器

创建完成任务类，dispatch任务后，任务会在 `队列` 中，要想真正执行任务还要执行 `队列处理器`

Laravel 包含了一个队列处理器以将推送到队列中的任务执行。你可以使用 `queue:work` Artisan 命令运行处理器。注意一旦 `queue:work` 命令开始执行，它会一直运行直到它被手动停止或终端被关闭。


```
php artisan queue:work
```

Tip: 要使 queue:work 进程一直在后台运行, 你应该使用进程管理器比如 Supervisor 来确保队列处理器不会停止运行

记住, 队列处理器是一个常驻的进程并且在内存中保存着已经启动的应用状态。因此, 它们并不会在启动后注意到你代码的更改。所以, 在你的重新部署过程中, 请记得 重启你的队列处理器。

思考

上面我们通过创建任务类, 分发任务, 运行队列处理器我们的任务就可以异步执行了。在文章开头我们提到的消息队列中有四个重要的角色: 消息、队列、生产者、消费者, 而在 Laravel 的封装下只有任务类、任务分发 (dispatch)、队列处理器。它们之间的对应关系可以这样理解:

- 消息 是 初始化的任务类。
- 队列 是我们在 config/queue.php 中配置的连接 所对应的 queue, 消息被分发后它们可能会保存在RabbitMQ、Redis或者数据库中的队列中。
- 生产者 是 dispatch
- 消费者 是 队列处理器 + 任务类的 handle 方法

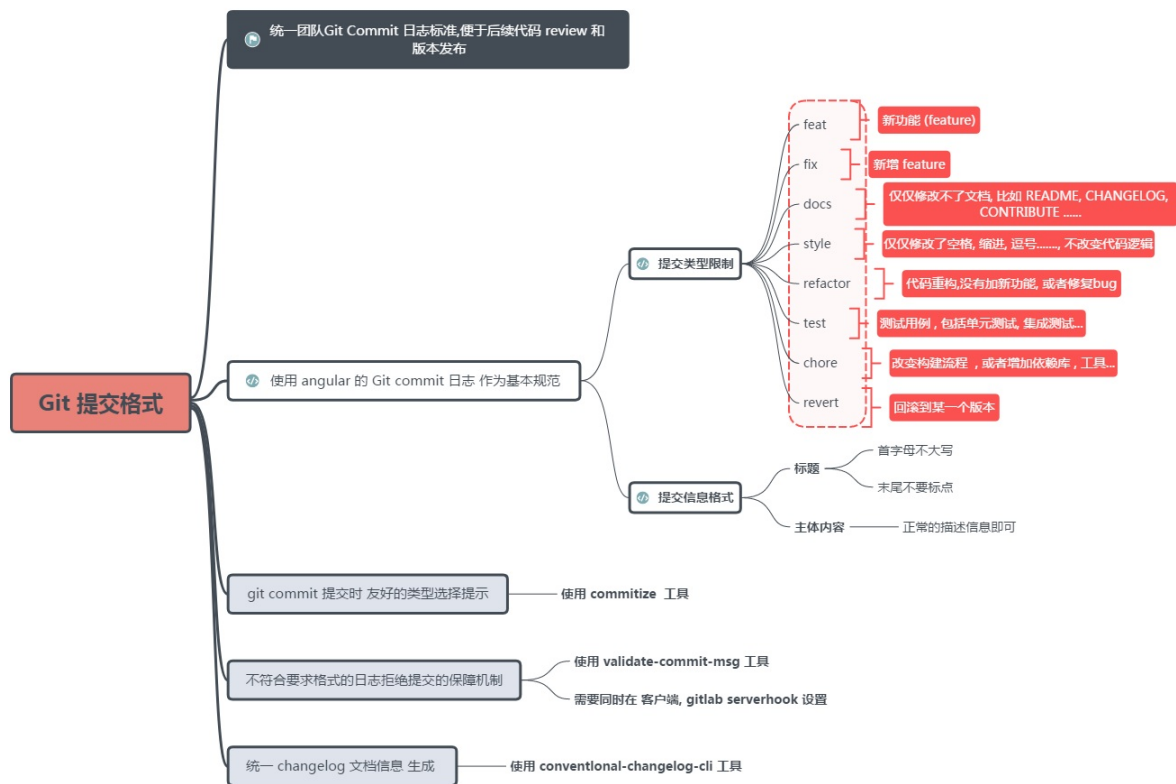
一个消息从产出到消费的整个流程可以分三步理解:

第一步: 通过定义任务类的属性, 定义消息结构体。同时在任务类中定义了消息的处理方法 handle 。

第二步: 我们根据业务初始化任务类然后调用 dispatch 方法。dispatch 方法会根据规则将任务类格式化为字符串格, 然后将字符串放入队列中, 此时 dispatch 完成生产者的职责。

第三步: 消息被投递到队列之后, 队列处理器 就会从队列中获取字符串消息, 根据规则将字符串消息初始化为对应的任务类, 然后再调用任务的 handle 方法, 完成消息的消费。

git提交规范



1. Git commit规范

1.1. 格式

<type>(<scope>): <subject>

示例:

fix(ngRepeat): fix trackBy function being invoked with incorrect scope

1.2. type

主要的提交类型如下:

Type	说明	备注
feat	提交新功能	常用
fix	修复bug	常用
docs	修改文档	
style	修改格式，例如格式化代码，空格，拼写错误等	
refactor	重构代码，没有添加新功能也没有修复bug	
test	添加或修改测试用例	
perf	代码性能调优	
chore	修改构建工具、构建流程、更新依赖库、文档生成逻辑	例如vendor包

1.3. scope

表示此次commit涉及的文件范围，可以使用 `*` 来表示涉及多个范围。

1.4. subject

描述此次commit涉及的修改内容。

- 使用祈使句（动词开头）、动宾短语。
- 第一个字母不要大写。
- 不要以 `.` 句号结尾。

2. Git commit工具

安装 `commitizen` 和 `cz-conventional-changelog` 。

```
npm install -g commitizen cz-conventional-changelog
echo '{ "path": "cz-conventional-changelog" }' > ~/.czrc
```

使用cz-cli

```
$ git cz
cz-cli@4.0.3, cz-conventional-changelog@3.0.1
? Select the type of change that you're committing: (Use arrow keys)
> feat:      A new feature
fix:        A bug fix
docs:       Documentation only changes
style:      Changes that do not affect the meaning of the code (white-space,
            formatting, missing semi-colons, etc)
refactor:   A code change that neither fixes a bug nor adds a feature
perf:       A code change that improves performance
test:       Adding missing tests or correcting existing tests
(Move up and down to reveal more choices)
```

说明

说明

作为中国电子商务全面解决方案的先行者，商派技术团队通过十八年的实践，积累了大量的电子商务前后端的架构模式和中间件设计经验，正是通过这些积累，保证了商派在国内电商业界的领先地位。

从公司ShopEx 4.8系列产品开始，商派一直在探索一种行之有效的方式，既能够降低研发成本和研发工作的复杂性，又能够快速跟进业务的发展变化，在这种方式下，可以省去很多基础性的研发工作，复用八年来获得的经验，同时使研发周期大幅缩短，提高研发效率。

自2009年立项开始，商派投入了30余名工程师，用将近一年的时间打造出了开源的电子商务业务架构平台，并将其命名为ECOS，寓意电子商务操作系统。

在经过 10 年发展后，我们基于 ECOS 做出了大量电商产品，于此同时开源框架也发展迅速，在新零售来临时，我们决定采用开源框架来重构我们的产品，在重构过程中，我们对商派多年的开发经验和开源社区经验进行总结，形成了此开发规范。

目的

规范有以下目的：

- 高效编码 - 避免了过多的选择造成的『决策时间』浪费；
- 风格统一 - 最大程度统一了开发团队成员代码书写风格和思路，代码阅读起来如出一辙；
- 减少错误 - 减小初级工程师的犯错几率。
- 代码可被生成 - 通过统一的规范，可以编写脚手架来生成重复性的代码。

开发哲学

因为篇幅原因本规范无法涉及到项目里每一块代码的编写标准，所以此处重点说明下此规范遵循的『开发哲学』，开发中请把其当做指明灯，来指引你做决策：

- DRY – 「Don't Repeat Yourself」不写重复的逻辑代码；
- 约定俗成 - 「Convention Over Configuration」，优先选择框架提倡的做法，不过度配置；
- KISS - 「Keep it Simple, Stupid」提倡简单易读的代码，不写高深、晦涩难懂的代码，**不过度设计**；
- 主厨精选 - 让有经验的人来为你选择方案，不独创方案；
- 官方提倡 - 优先选择官方推崇的方案。

设计理念

以下是一些优秀的『程序设计理念』：

- MVC - Model, View, Controller ，以 MVC 为核心，新增 [Repository 模式](#) 和 [Service 模式](#)，来控制 Controller 和 Model 代码行数；
- Restful - 利用『资源化概念』和标准的 HTTP 动词来组织你的程序；

在此规范中，我们会将使用这两套理念作为程序设计基础。这些设计理念为我们设计程序提供了依据，遵循这些理念，能让程序变得清晰易读。

关于「能愿动词」的使用

能愿动词

为了避免歧义，文档大量使用了「能愿动词」，对应的解释如下：

- 必须（Must） - 只能这样子做，请无条件遵循，没有别的选项；
- 绝不（Must Not） - 严令禁止，在任何情况下都不能这样做；
- 应该（Should） - 强烈建议这样做，但是不强求；
- 不应该（Should Not） - 强烈建议不这样做，但是不强求；
- 可以（May） - 选择性高一点，在这个文档内，此词语使用较少；

参考：[RFC 2119](#)

关于执行

在这份规范里，有些内容里会解释『这样做的理由』，这样做的目的是为了达成共识。

请不要以此『理由』的准确性来怀疑规范的权威性，规范就是规范，可以讨论改正，但在执行的时候 **必须** 严格遵守。

请把『团队项目开发』想象就是在行军打仗，对于规范要绝对服从。要有大局观，做到团结一致，把个人的喜好放一边，把整个团队的执行效率放在第一位。

基本代码规范

本篇规范制定了代码基本元素的相关标准，以确保共享的 PHP 代码间具有较高程度的技术互通性。

本文件中的 **必须**，**不得**，**需要**，**应**，**不应**，**应该**，**不应该**，**推荐**，**可能** 和 **可选** 等能愿动词按照 [RFC 2119](#) 中的描述进行解释。

1. 概览

- PHP代码文件 **必须** 以 `<?php` 或 `<?=' 标签开始；`
- PHP代码文件 **必须** 以 不带 BOM 的 UTF-8 编码；
- PHP代码中 **应该** 只定义类、函数、常量等声明，或其他会产生 **副作用** 的操作（如：生成文件输出以及修改 .ini 配置文件等），二者只能选其一；
- 命名空间以及类 **必须** 符合 PSR 的自动加载规范：[\[PSR-0（已废弃）或 PSR-4\]](#) 中的一个。
- 类的命名 **必须** 遵循 `StudlyCaps` 大写开头的驼峰命名规范；
- 类中的常量所有字母都 **必须** 大写，单词间用下划线分隔；
- 方法名称 **必须** 符合 `camelCase` 式的小写开头驼峰命名规范。

2. 文件

2.1. PHP 标签

PHP 代码 **必须** 使用 `<?php ?>` 长标签 或 `<?=' ?>` 短输出标签；**一定不可** 使用其它自定义标签。

2.2. 字符集编码

PHP代码 **必须** 且只可使用 不带 BOM 的 UTF-8 编码。

2.3. 副作用

一份 PHP 文件中 **应该** 要不就只定义新的声明，如类、函数或常量等不产生 **副作用** 的操作，要不就只书写会产生 **副作用** 的逻辑操作，但 **不该** 同时具有两者。

「副作用」(side effects)一词的意思是，仅仅通过包含文件，不直接声明类、函数和常量等，而执行的逻辑操作。

「副作用」包含却不仅限于：生成输出，明确使用require或include，连接到外部服务，修改ini设置，发出错误或异常，修改全局或静态变量，读取或写入一个文件，等等。

以下是一个反例，一份包含「函数声明」以及产生「副作用」的代码：

```
<?php
// 「副作用」：修改 ini 配置
ini_set('error_reporting', E_ALL);

// 「副作用」：引入文件
include "file.php";

// 「副作用」：生成输出
echo "<html>\n";

// 声明函数
function foo()
{
    // function body
}
```

下面是一个范例，一份只包含声明不产生「副作用」的代码：

```
<?php
// 声明函数
function foo()
{
    // 函数主体部分
}

// 条件声明 **不** 属于「副作用」
if (! function_exists('bar')) {
    function bar()
    {
        // 函数主体部分
    }
}
```

3. 命名空间和类名

命名空间和类名 **必须** 遵循『自动加载』规范：[[PSR-0](#)，[PSR-4](#)]。

这意味着每个类都独立为一个文件，并且至少在一个层次的命名空间内，那就是：顶级组织名（vendor name）。

类名 **必须** 以类似 `StudlyCaps` 形式的大写开头的驼峰命名方式声明。

PHP 5.3 及更高版本的代码 **必须** 使用正式的命名空间。

举个例子：

```
<?php
// PHP 5.3 及更高版本：
namespace Vendor\Model;

class Foo
{
}
```

PHP 5.2 及更低版本 **应该** 使用伪命名空间，约定俗成，以顶级组织名称 `Vendor_` 为类名前缀：

```
<?php
// PHP 5.2.x 及更低版本：
class Vendor_Model_Foo
{
}
```

4. 类的常量、属性和方法

此处的「类」指代所有的类、接口以及可复用代码块（traits）。

4.1. 常量

类的常量中所有字母都 **必须** 大写，词间以下划线分隔。例如：

```
<?php
namespace Vendor\Model;

class Foo
{
    const VERSION = '1.0';
    const DATE_APPROVED = '2012-06-01';
}
```

4.2. 属性

类的属性命名 **可以** 遵循：

- 大写开头的驼峰式 (`$StudlyCaps`)
- 小写开头的驼峰式 (`$camelCase`)
- 下划线分隔式 (`$under_score`)

本规范不做强制要求，但无论遵循哪种命名方式，都 **应该** 在一定的范围内保持一致。这个范围可以是整个团队、整个包、整个类或整个方法。

4.3. 方法

方法名称 **必须** 符合 `camelCase()` 式的小写开头驼峰命名规范。

本文转自：<https://learnku.com/docs/psr/basic-coding-standard/1605>

编码风格规范

由于 PHP7 的一些语法的变更原因，[PSR-2](#) 在 2019-08-10 已被废弃，由 [PSR-12](#) 来代替。

文章中的关键词 `MUST`，`MUST NOT`，`REQUIRED`，`SHALL`，`SHALL NOT`，`SHOULD`，`SHOULD NOT`，`RECOMMENDED`，`MAY`，和 `OPTIONAL` 都在 [RFC 2119](#) 中进行来解释。

摘要

此规范起到继承，扩展和替换 [\[PSR-2\]](#) 的作用，同时编码风格遵守 [\[PSR-1\]](#) 这个基础编码标准。

和 [\[PSR-2\]](#) 一样，此规范的目的是减少不同人在阅读代码时认知冲突。它通过列举一套如何格式化 PHP 代码的公共的规则和期望来实现这个目标。[PSR](#) 力图提供一套方法，编码风格工具可以利用，项目可以遵守，开发人员可以方便的在不同的项目中使用。当各个的开发人员进行多项目合作的时候，它可以帮助在这些项目中提供一套通用的指导。所以，本指南的价值不是规则本身，而是这些规则的共享。

[\[PSR-2\]](#) 在 2012 年被接受，随后 PHP 经历了很多变化，影响了编码风格。同时 [\[PSR-2\]](#) 是 PHP 编码时候的基础功能，被广泛的采用。因此，[PSR](#) 力图通过一种更加现代的方式说明 [PSR-2](#) 的内容和新功能，并对 [PSR-2](#) 进行更正。

以前的语言版本

在整个文档中，任何说明都可以被忽略，如果它们不存在于你项目所支持的 PHP 版本中。

例如

此示例包含以下一些规则作为快速概述：

```
<?php

declare(strict_types=1);

namespace Vendor\Package;

use Vendor\Package\{ClassA as A, ClassB, ClassC as C};
use Vendor\Package\SomeNamespace\ClassD as D;

use function Vendor\Package\{functionA, functionB, functionC};

use const Vendor\Package\{ConstantA, ConstantB, ConstantC};

class Foo extends Bar implements FooInterface
{
    public function sampleFunction(int $a, int $b = null): array
    {
        if ($a === $b) {
            bar();
        } elseif ($a > $b) {
            $foo->bar($arg1);
        } else {
            BazClass::bar($arg2, $arg3);
        }
    }

    final public static function bar()
    {
        // 方法内容
    }
}
```

2. 总则

2.1 基本编码标准

代码必须遵循[PSR-1]中列出的所有规则。

PSR-1中的术语 'StudlyCaps' 必须解释为 PascalCase（帕斯卡命名法：大驼峰式命名法），其中每个单词的第一个字母大写，包括第一个字母。

2.2 文件

所有 PHP 文件只能使用 Unix LF (换行符) 结尾。

所有的 PHP 文件都必须以非空行结尾，以一个 LF 结尾。

在仅包含 PHP 代码的文件中，必须省略结尾的 `? >` 标记。

2.3 代码行

行长度不得有硬限制。

行长度的软限制必须为120个字符。

行的长度不应超过80个字符；超过该长度的行应拆分为多个后续行，每个行的长度不应超过80个字符。

行尾不能有尾随空格。

可以添加空行以提高可读性并指示相关的代码块，除非明确禁止。

每行不能有多个语句。

2.4 缩进

代码必须为每个缩进级别使用4个空格的缩进，并且不能使用缩进标签。

2.5 关键词和类型

PHP 的所有关键字和类型 [\[1\]\[2\]](#) 都必须使用小写。

PHP未来版本中新加的所有关键字和类型也都**必须**使用小写。

类型关键字**必须**使用缩写。使用 `bool` 而不是 `boolean`，使用 `int` 而不是 `integer` 等等。

3. 声明、命名空间以及导入

一个PHP文件的头部可能会包含多个块。如果包含多个块，则每个块都**必须**用空白行和其他块分隔，并且块内**不能**包含空白行。所有的块都**必须**按照下面的顺序排列，如果不存在该块则忽略。

- PHP文件开始标签： `<?php` 。
- 文件级文档块。
- 一个或多个声明语句。
- 命名空间声明语句。
- 一个或多个基于类的 `use` 声明语句。

- 一个或多个基于方法的 `use` 声明语句。
- 一个或多个基于常量的 `use` 声明语句。
- 其余代码。

当文件包含HTML和PHP的混合代码时，可以使用上面列出的任何部分。如果是这种情况的话，即时代码的其他部分包含有PHP结束符，然后再包含HTML和PHP代码，声明、命名空间和导入语句块也必须放在文件的顶部。

什么时候开始 `<? php` 标签位于文件的第一行，它必须位于自己的行，没有其他语句，除非它是一个包含 PHP 之外的标记的文件打开和关闭标记。

`import`语句不能以前导反斜杠开头，因为它们必须始终完全合格。

以下示例演示了所有块的完整列表：

```
<?php

/**
 * This file contains an example of coding styles.
 */

declare(strict_types=1);

namespace Vendor\Package;

use Vendor\Package\{ClassA as A, ClassB, ClassC as C};
use Vendor\Package\SomeNamespace\ClassD as D;
use Vendor\Package\AnotherNamespace\ClassE as E;

use function Vendor\Package\{functionA, functionB, functionC};
use function Another\Vendor\functionD;

use const Vendor\Package\{CONSTANT_A, CONSTANT_B, CONSTANT_C};
use const Another\Vendor\CONSTANT_D;

/**
 * FooBar is an example class.
 */
class FooBar
{
    // ... 其他php代码 ...
}
```

深度不能超过两层的复合名称空间，因此以下展示了允许的最大复合深度。

```
<?php

use Vendor\Package\SomeNamespace\{
    SubnamespaceOne\ClassA,
    SubnamespaceOne\ClassB,
    SubnamespaceTwo\ClassY,
    ClassZ,
};
```

并且不允许以下内容:

```
<?php

use Vendor\Package\SomeNamespace\{
    SubnamespaceOne\AnotherNamespace\ClassA,
    SubnamespaceOne\ClassB,
    ClassZ,
};
```

当希望在 PHP 外部包含标记的文件中声明严格类型时打开和关闭标签，声明必须写在文件的第一行并且包含在一个开始的 PHP 标签，以及严格的类型声明和结束标签。

例如:

```
<?php declare(strict_types=1) ?>
<html>
<body>
    <?php
        // ... 其他 PHP 代码 ...
    ?>
</body>
</html>
```

声明语句不能包含空格，并且必须完全是 `declare(strict_types=1)` (带有可选的分号终止符)。

允许使用块声明语句，并且必须按照以下的格式设置。注意的位置括号和间距：

```
declare(ticks=1) {
    // 一些代码
}
```


4. 类，属性，和方法

这里的『类』指的是所有类，接口，以及 trait。

任何注释和语句 **不得** 跟在其右花括号后的同一行。

当实例化一个类时，后面的圆括号 **必须** 写出来，即使没有参数传进其构造函数。

```
new Foo();
```

4.1 继承和实现

关键字 `继承` 和 `实现` **必须** 在类名的同一行声明。

类的左花括号 **必须** 另起一行；右花括号 **必须** 跟在类主体的下一行。

类的左花括号 **必须** 独自成行，且 **不得** 在其上一行或下一行存在空行。

右花括号 **必须** 独自成行，且 **不得** 在其上一行存在空行。

```
<?php

namespace Vendor\Package;

use FooClass;
use BarClass as Bar;
use OtherVendor\OtherPackage\BazClass;

class ClassName extends ParentClass implements \ArrayAccess, \Countable
{
    // 常量，属性，方法
}
```

如果有接口，`实现` 接口和 `继承` 父类 **可以** 分为多行，前者每行需缩进一次。当这么做时，第一个接口 **必须** 写在下一行，且每行 **必须** 只能写一个接口。

```
<?php

namespace Vendor\Package;

use FooClass;
use BarClass as Bar;
use OtherVendor\OtherPackage\BazClass;

class ClassName extends ParentClass implements
    \ArrayAccess,
    \Countable,
    \Serializable
{
    // 常量, 属性, 方法
}
```

4.2 使用 trait

在类里面用于实现 trait 的关键字 `use` **必须** 在左花括号的下一行声明。

```
<?php

namespace Vendor\Package;

use Vendor\Package\FirstTrait;

class ClassName
{
    use FirstTrait;
}
```

每个导入类的 trait **必须** 每行一个包含声明，且每个包含声明 **必须** 有其 `use` 导入语句。

```
<?php

namespace Vendor\Package;

use Vendor\Package\FirstTrait;
use Vendor\Package\SecondTrait;
use Vendor\Package\ThirdTrait;

class ClassName
{
    use FirstTrait;
    use SecondTrait;
    use ThirdTrait;
}
```

在类文件中，如果在使用'use Trait'之后没有其他内容了，类名右大括号必须另起一行。

```
<?php

namespace Vendor\Package;

use Vendor\Package\FirstTrait;

class ClassName
{
    use FirstTrait;
}
```

如有其他内容，两者之间需空一行。

```
<?php

namespace Vendor\Package;

use Vendor\Package\FirstTrait;

class ClassName
{
    use FirstTrait;

    private $property;
}
```

当使用 'insteadof' 和 'as' 运算符时，它们必须如图所示使用，注意缩进、间距和另起一行。

```
<?php

class Talker
{
    use A, B, C {
        B::smallTalk insteadof A;
        A::bigTalk insteadof C;
        C::mediumTalk as FooBar;
    }
}
```

4.3 属性和常量

所有属性 **必须** 声明可见性。

如果你的项目 PHP 最小版本支持常量可见性（PHP 7.1 或以上），所有常量 **必须** 声明可见性。

关键字 `var` **不得** 用于声明属性。

每条声明语句 **不得** 声明多于一个属性。

属性名 **不得** 用单个下划线开头表明其受保护的或私有的可见性。也就是说，一个下划线开头显然是没有意义的。

类型声明和属性名之间 **必须** 有一个空格。

一个属性声明看上去如下所示：

```
<?php

namespace Vendor\Package;

class ClassName
{
    public $foo = null;
    public static int $bar = 0;
}
```

4.4 方法和函数

所有的方法 **必须** 事先声明类型。

方法命名 **一定不可** 用单个下划线来区分是 `protected` 或 `private` 类型。也就是说，不要用一个没有意义的下划线开头。

方法和函数名称中，方法命名后面 **一定不可** 使用空格。方法开始的花括号 **必须** 写在方法声明后自成一，结束花括号也 **必须** 写在方法后面自成一。开始左括号后和结束右括号前，都 **一定不可** 有空格符。

一个方法的声明应该如下所示。注意括号，逗号，空格和花括号的位置：

```
<?php

namespace Vendor\Package;

class ClassName
{
    public function fooBarBaz($arg1, &$arg2, $arg3 = [])
    {
        // 方法主体
    }
}
```

一个函数的声明应该如下所示。注意括号，逗号，空格和花括号的位置：

```
<?php

function fooBarBaz($arg1, &$arg2, $arg3 = [])
{
    // 函数主体
}
```

4.5 方法和函数参数

在参数列表中，**不得** 在每个逗号前存在空格，且 **必须** 在每个逗号后有一个空格。

方法和函数中带有默认值的参数 **必须** 放在参数列表的最后。

```
<?php

namespace Vendor\Package;

class ClassName
{
    public function foo(int $arg1, &$arg2, $arg3 = [])
    {
        // 方法主体
    }
}
```

参数列表 **可以** 分为多行，每行参数缩进一次。当这么做时，第一个参数 **必须** 放在下一行，且每行 **必须** 只能有一个参数。

当参数列表分成多行时，右圆括号和左花括号 **必须** 放在同一行且单独成行，两者之间存在一个空格。

```
<?php

namespace Vendor\Package;

class ClassName
{
    public function aVeryLongMethodName(
        ClassTypeHint $arg1,
        &$arg2,
        array $arg3 = []
    ) {
        // 方法主体
    }
}
```

当你定义一个返回值类型声明时，冒号后面的类型声明 **必须** 用空格符隔开。冒号和声明 **必须** 在同一行，且跟参数列表后的结束括号之间没有空格。

```
<?php

declare(strict_types=1);

namespace Vendor\Package;

class ReturnTypeVariations
{
    public function functionName(int $arg1, $arg2): string
    {
        return 'foo';
    }

    public function anotherFunction(
        string $foo,
        string $bar,
        int $baz
    ): string {
        return 'foo';
    }
}
```

在可空类型声明中，问号和类型声明之间**不能**有空格。

```
<?php

declare(strict_types=1);

namespace Vendor\Package;

class ReturnTypeVariations
{
    public function functionName(?string $arg1, ?int &$arg2): ?string
    {
        return 'foo';
    }
}
```

当在参数之前使用引用运算符 `&` 时，引用运算符之后**不能**有空格，例如上面的示例。

可变参数声明的三个点和参数名称之间不能有空格：

```
public function process(string $algorithm, ...$parts)
{
    // 函数体
}
```

当同时使用引用运算符和可变参数运算符时，它们之间**不能**有任何空格：

```
public function process(string $algorithm, &...$parts)
{
    // 函数体
}
```

4.6 abstract , final , and static

如果是 `abstract` and `final` ，那么声明的时候必须是可见性声明。

如果是 `static` ，声明必须位于可见性声明之后。

```
<?php

namespace Vendor\Package;

abstract class ClassName
{
    protected static $foo;

    abstract protected function zim();

    final public static function bar()
    {
        // 请求体
    }
}
```

4.7 方法和函数的调用

当我们在进行方法或者函数调用的时候，方法或函数与左括号之间不能出现空格，在右括号之后也不能出现空格，并且在右括号之前也一定不能有空格。在参数列表中，每个逗号前面一定不能有空格，每个逗号后面也一定不能有空格。


```
<?php

bar();
$foo->bar($arg1);
Foo::bar($arg2, $arg3);
```

参数列表可以分为多行，每行后面缩进一次。这个做之后，列表中的第一行必须位于下一行，并且每一行必须只有一个参数。跨多个行拆分单个参数(就像匿名函数或者数组那样)并不构成拆分参数列表本身。

```
<?php

$foo->bar(
    $longArgument,
    $longerArgument,
    $muchLongerArgument
);
```

```
<?php

somefunction($foo, $bar, [
    // ...
], $baz);

$app->get('/hello/{name}', function ($name) use ($app) {
    return 'Hello ' . $app->escape($name);
});
```

5. 流程控制

如下是主要的流程控制风格规则：

- 流程控制关键词之后 **必须** 要有一个空格
- 左括号后面 **不能** 有空格
- 右括号前面 **不能** 有空格
- 右括号与左大括号之间 **必须** 要有一个空格
- 流程主体 **必须** 要缩进一次
- 流程主体 **必须** 在左大括号之后另起一行
- 右大括号 **必须** 在流程主体之后另起一行

每个流程控制主体 **必须** 以封闭的括号结束。这将标准化流程结构，同时减少由于流程中添加新的内容而引入错误的可能性。

5.1 if , elseif , else

if 结构如下。注意括号，空格，和大括号的位置； else 和 elseif 都在同一行，和右大括号一样在主体的前面。

```
<?php

if ($expr1) {
    // if body
} elseif ($expr2) {
    // elseif body
} else {
    // else body;
}
```

关键词 elseif **应该** 替换 else if ，这样控制关键词看起来像一个词。

括号中的表达式 **可能** 会被分开为多行，每一行至少缩进一次。如果这样做，第一个条件 **必须** 在新的一行。右括号和左大括号 **必须** 在同一行，而且中间有一个空格。条件中间的布尔控制符 **必须** 在每一行的开头或者结尾，而不是混在一起。

```
<?php

if (
    $expr1
    && $expr2
) {
    // if body
} elseif (
    $expr3
    && $expr4
) {
    // elseif body
}
```

5.2 switch , case

`switch` 结构如下。注意括号，空格和大括号的位置。 `case` **必须** 缩进一次从 `switch` 开始， `break` 关键词 (或者其他终止关键词) **必须** 缩进和 `case` 主体保持一致。**必须** 要有一个像 `// 不终止` 这样的注释在不为空且继续的 `case` 主体之中。

```
<?php

switch ($expr) {
    case 0:
        echo 'First case, with a break';
        break;
    case 1:
        echo 'Second case, which falls through';
        // no break
    case 2:
    case 3:
    case 4:
        echo 'Third case, return instead of break';
        return;
    default:
        echo 'Default case';
        break;
}
```

括号中的表达式 **可能** 会被分开多行，每一行至少要缩进一次。如果这样做，第一个条件 **必须** 在新的一行。右括号和左大括号 **必须** 在同一行，而且中间有一个空格。条件中间的布尔控制符 **必须** 在没一行的开头或者结尾，而不是混在一起。

```
<?php

switch (
    $expr1
    && $expr2
) {
    // structure body
}
```

5.3 while , do while

`while` 结构如下。注意括号，空格和大括号的位置。

```
<?php

while ($expr) {
    // structure body
}
```

括号中的表达式 **可能** 会被分开多行，没一行至少要缩进一次。如果这样做，第一个条件 **必须** 在新的一行。右括号和左大括号 **必须** 在同一行，而且中间有一个空格。条件中间的布尔控制符 **必须** 在没一行的开头或者结尾，而不是混在一起。

```
<?php

while (
    $expr1
    && $expr2
) {
    // structure body
}
```

同样的，`do while` 申明如下。注意括号，空格和大括号的位置。

```
<?php

do {
    // structure body;
} while ($expr);
```

括号中的表达式 **可能** 会被分开多行，没一行至少要缩进一次。如果这样做，第一个条件 **必须** 在新的一行。条件中间的布尔控制符 **必须** 在每一行的开头或者结尾，而不是混在一起。

```
<?php

do {
    // structure body;
} while (
    $expr1
    && $expr2
);
```

5.4 for

`for` 申明如下。注意括号，空格和大括号的位置。

```
<?php

for ($i = 0; $i < 10; $i++) {
    // for body
}
```

括号中的表达式 **可能** 会被分开多行，每一行至少要缩进一次。如果这样做，第一个条件 **必须** 在新的一行。右括号和左大括号 **必须** 在同一行，而且中间有一个空格。

```
<?php

for (
    $i = 0;
    $i < 10;
    $i++
) {
    // for body
}
```

5.5 foreach

`foreach` 语句的写法如下所示。请注意它的圆括号、空格和花括号。

```
<?php

foreach ($iterable as $key => $value) {
    // 迭代主体
}
```

5.6 try ， catch ， finally

一个 `try-catch-finally` 模块包含下面这些内容。请注意它的圆括号、空格和花括号。

```
<?php

try {
    // try 主体
} catch (FirstThrowableType $e) {
    // 捕获异常主体
} catch (OtherThrowableType | AnotherThrowableType $e) {
    // 捕获异常主体
} finally {
    // finally 主体
}
```

6. 运算符

运算符的样式规则按元数分组（其接受的操作数个数）。

当运算符周围允许出现空格时，**可以** 出于可读性目的打多个空格。

所有这里没描述的运算符暂不作限定。

6.1. 一元运算符

递增 / 递减运算符和操作数之间 **不得** 有任何空格。

```
$i++;
++$j;
```

类型转换运算符的圆括号内部 **不得** 有任何空格：

```
$intValue = (int) $input;
```

6.2. 二元运算符

所有二进制 [算术](#)，[比较](#)，[赋值](#)，[按位](#)，[逻辑](#)、[字符串](#)和[类型](#)运算符必须在前后跟至少一个空格：

```
if ($a === $b) {
    $foo = $bar ?? $a ?? $b;
} elseif ($a > $b) {
    $foo = $a + $b * $c;
}
```

6.3. 三元运算符

条件运算符，也称为三元运算符，必须在 `?` 和 `:` 这两个字符之间：

```
$variable = $foo ? 'foo' : 'bar';
```

如果省略条件运算符的中间操作数，运算符必须遵循与其他二进制[比较](#)运算符相同的样式规则：

```
$variable = $foo ?: 'bar';
```

7. 闭包（Closures）

闭包声明时必须在 `function` 关键字后留有1个空格，并且在 `use` 关键字前后各留有1个空格。

左花括号必须跟随前文写在同一行，右花括号必须在函数体后换行放置。

不能在参数和变量的左括号后和右括号前放置空格。

不能在参数和变量的逗号前放置空格，但必须在逗号后放置1个空格。

闭包参数如果有默认值，该参数必须放在参数列表末尾。

如果声明了返回类型，它必须遵循普通函数和方法相同的规则；如果使用 `use` 关键字，冒号必须在 `use` 右括号后且冒号前后不能有空格。

闭包的声明方式如下，留意括号，逗号，空格和花括号：

```
<?php

$closureWithArgs = function ($arg1, $arg2) {
    // 函数体
};

$closureWithArgsAndVars = function ($arg1, $arg2) use ($var1, $var2) {
    // 函数体
};

$closureWithArgsVarsAndReturn = function ($arg1, $arg2) use ($var1, $var2): bool {
    // 函数体
};
```

参数和变量可以分多行放置，每个后续行缩进一次。执行此操作时，列表中的第一项必须放在下一行，并且每行只能有一个参数或变量。

结束多行列表（或者参数，变量）的时候，右括号和左大括号 **必须** 要放在一行，而且中间有一个空格。

下面是有和没有多行参数列表与变量列表的闭包示例。

```
<?php

$longArgs_noVars = function (
    $longArgument,
    $longerArgument,
    $muchLongerArgument
) {
    // body
};

$noArgs_longVars = function () use (
    $longVar1,
    $longerVar2,
    $muchLongerVar3
) {
    // body
};

$longArgs_longVars = function (
    $longArgument,
    $longerArgument,
    $muchLongerArgument
) use (
    $longVar1,
    $longerVar2,
    $muchLongerVar3
) {
    // body
};

$longArgs_shortVars = function (
    $longArgument,
    $longerArgument,
    $muchLongerArgument
) use ($var1) {
    // body
};
```



```
$shortArgs_longVars = function ($arg) use (  
    $longVar1,  
    $longerVar2,  
    $muchLongerVar3  
) {  
    // body  
};
```

注意格式化规则也适用一个闭包在一个方法或者操作中作为参数被直接引用。

```
<?php  
  
$foo->bar(  
    $arg1,  
    function ($arg2) use ($var1) {  
        // body  
    },  
    $arg3  
);
```

8. 匿名类

匿名类 **必须** 遵循上面章节中和闭包一样的方针和准则。

```
<?php  
  
$instance = new class {};
```

只要 `implements` 接口列表不换行，左花括号 **可以** 和关键字 `class` 在同一行。如果接口列表换行，花括号 **必须** 放在最后一个接口的下一行。

```
<?php

// 花括号在同一行
$instance = new class extends \Foo implements \HandleableInterface {
    // 类内容
};

// 花括号在下一行
$instance = new class extends \Foo implements
    \ArrayAccess,
    \Countable,
    \Serializable
{
    // 类内容
};
```

本文转自：<https://learnku.com/docs/psr/psr-12-extended-coding-style-guide/5789>

命名规范

命名规范

- Bundle 名 必须 为 业务名称+Bundle 整体命名遵循 大驼峰 规范。
- Repository 类 必须 为 业务名称+Repository
- Service 类 必须 为 业务名称+Service
- Event 类 必须 为 业务名称+Event

Entity 类

所有实体类 必须 放置在 {Name}Bundle/Entities 目录下。

实体类的所有属性 必须 为 private ， 绝不 使用 public

实体类的所有属性 必须 有对应的 getter 方法，除主键字段外，其他属性 必须 设置 setter 方法。

控制器

控制器方法 应该 只包含以下三个职责：

- 验证输入参数有效性
- 组织数据，调用 Service
- 对调用 Service 返回的数据根据需求调整数据格式返回

控制器方法代码行数 应该 不超过 80 行，超过 80 行很可能需要将处理逻辑写到 Service 中。

控制器的方法 应该 只调用 Service ，不能调用 Repository 。

一旦控制器的方法中调用了 Repository 。后续参与的开发人员就会延续之前的思路继续在控制器中写代码，破窗效应 一旦形成，后续的代码质量将无法控制。

调试

- 接口调试可通过POSTMAN来进行.
- 异常处理可依赖, 系统日志.
- 微信调试, 可以依赖微信日志.
- 正式环境强烈建议使用 *SENTRY*, 捕获错误并及时处理

日志调试

系统日志

- 系统日志 DEBUG/INFO/NOTICE/WARNING/ERROR/CRITICAL/EMERGENCY
- 数据库执行日志

```
# 修改 .env
DOCTRINE_LOGGER=LaravelDoctrine\ORM\Loggers\FileLogger
```

系统日常默认放置在 *storage/logs/lumen.log*

可参考: [lumen Errors & Logging](#)

微信调试

- 微信开放平台第三方平台日志
- 微信公众号日志
- 微信小程序日志

日志默认放置在 *storage/logs/wechat.log*

手动异常抛出

有的场景捕获异常后, 依然需要完整的*Exception*抛出, 以便问题排查, 这是可以通过 *app('api.exception')->report(\$e)*进行抛错

例如:

```
<?php
try {
    #.....
} catch (Exception $e) {
    $status = 'fail';

    app('api.exception')->report($e);

    $exceptionMessage = $e->getMessage();

    #...
}
```

SENTRY异常捕获平台

可配置`.env`

```
SENTRY_LARAVEL_DSN=
```

参考:

- [SENTRY官方文档](#)
- [sentry使用实践](#)

记录日志

```
<?php
app('log')->debug('debug');
app('log')->info('info');
app('log')->notice('notice');
app('log')->warning('warning');
app('log')->error('error');
app('log')->crit('critical');
app('log')->alert('alert');
app('log')->emerg('emerg');
```

测试

lumen本身就有单元测试功能,目录是项目根目录下的tests目录

```
./vendor/bin/phpunit ./tests/TestCase.php
```

快速入门

需求

假如我们要开发一个商品模块，我们将它命名为 `ItemBundle`，其包含了商品、sku、商品类型、标签，可以实现商品相关的增删改查。

环境说明

本文档是用的开发环境是官方的 `docker` 环境，[点此查看](#)，其他环境使用命令时请按需修改。

本文档所使用的 `php artisan` 命令，都需要进入容器内执行，进入容器的命令为：

```
docker exec -it ecshopx-dev-docker_espier-bloated-web_1 sh
```

开发说明

在使用 `Doctrine ORM` 开发应用程序时，可以使用以下三种方式来启动开发：

- **Code First**：首先要开发 `Entity` 类，然后将他们映射到数据库中
- **Model First**：需要是用工具对（UML）进行建模，并从改模型生成数据库和 `Entity` 类
- **Database First**：首先要设计数据表，然后根据数据表生成 `Entity` 类

Code First

此方式需要根据需求设计 `Entity` 类，设计时需要熟知 `Doctrine ORM` 注解，只要熟悉数据库设计即可，所以用这种方式效率更高。此方式具体开发流程如下：

- 一、设计 `Entity` 类
- 二、通过命令生成数据库迁移

```
php artisan doctrine:migrations:diff
```

- 三、生成数据表

```
php artisan doctrine:migrations:migrate
```

Ecstore 和 Bbc 和此方式类似

Database First

此方式和 Database First 正好相反，先设计数据表，再生成 Entity 类。由于此方式不需要熟知 Doctrine ORM 注解，只要熟悉数据库设计即可，所以用这种方式效率更高。此方式具体开发流程如下：

- 一、采用数据库设计软件设计数据库 ER 图。
- 二、通过 ER 图生成的 SQL 创建数据表
- 三、通过命令生成实体

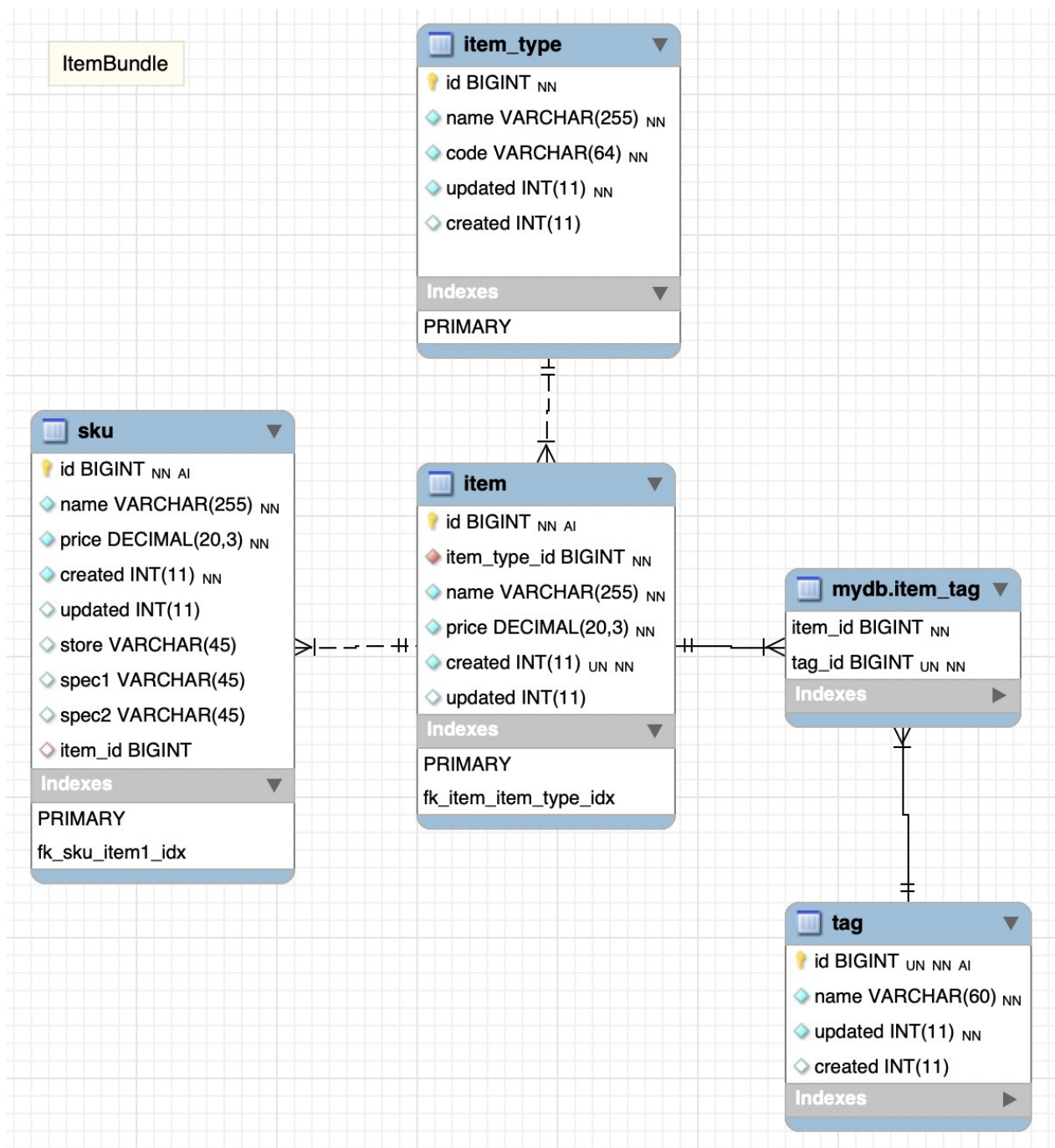
```
php artisan doctrine:mapping:import annotation --filter=DemoUser --namespace=ItemBundle\\Entities\\
```

--filter=DemoUser 中的 DemoUser 是想要生成的实体名，是数据表的大驼峰写法，不写 --filter 将会把数据库中所有的数据表都生成 Entity 放到 src/ItemBundle/Entities 目录中

本教程将采用 Database First 的方式。

设计数据库

根据需求，我们通过 Mysql Workbench 来设计 ER 图：



设计完 ER 图之后，在 `Mysql Workbench` 中可以通过 `Database->Forward Engineer` 直接将 ER 图直接生成数据表。

为了方便讲解，我们将此 ER 图生成的 SQL 放在此处，方便大家直接创建数据库：

```

-----
-- Table `mydb`.`item_type`
-----

DROP TABLE IF EXISTS `mydb`.`item_type` ;

CREATE TABLE IF NOT EXISTS `mydb`.`item_type` (
  `id` BIGINT NOT NULL,
  `name` VARCHAR(255) NOT NULL COMMENT '名称',

```

```
`code` VARCHAR(64) NOT NULL COMMENT '编码',
`updated` INT(11) NOT NULL,
`created` INT(11) NULL,
PRIMARY KEY (`id`))
COMMENT = '类型表';

-----
-- Table `mydb`.`item`
-----

DROP TABLE IF EXISTS `mydb`.`item` ;

CREATE TABLE IF NOT EXISTS `mydb`.`item` (
  `id` BIGINT NOT NULL AUTO_INCREMENT,
  `item_type_id` BIGINT NOT NULL COMMENT '类型',
  `name` VARCHAR(255) NOT NULL COMMENT '名称',
  `price` DECIMAL(20,3) NOT NULL COMMENT '价格',
  `created` INT(11) UNSIGNED NOT NULL COMMENT '创建时间',
  `updated` INT(11) NULL COMMENT '最后更新时间',
  PRIMARY KEY (`id`),
  CONSTRAINT `fk_item_item_type`
    FOREIGN KEY (`item_type_id`)
      REFERENCES `mydb`.`item_type` (`id`)
      ON DELETE NO ACTION
      ON UPDATE NO ACTION)
COMMENT = '商品';

CREATE INDEX `fk_item_item_type_idx` ON `mydb`.`item` (`item_type_id` ASC);

-----
-- Table `mydb`.`sku`
-----

DROP TABLE IF EXISTS `mydb`.`sku` ;

CREATE TABLE IF NOT EXISTS `mydb`.`sku` (
  `id` BIGINT NOT NULL AUTO_INCREMENT,
  `name` VARCHAR(255) NOT NULL COMMENT '名称',
  `price` DECIMAL(20,3) NOT NULL COMMENT '价格',
  `created` INT(11) NOT NULL,
  `updated` INT(11) NULL,
  `store` VARCHAR(45) NULL COMMENT '库存',
  `spec1` VARCHAR(45) NULL COMMENT '规格 1',
  `spec2` VARCHAR(45) NULL COMMENT '规格 2\n',
  `item_id` BIGINT NULL,
  PRIMARY KEY (`id`),
```

```
CONSTRAINT `fk_sku_item1`
  FOREIGN KEY (`item_id`)
  REFERENCES `mydb`.`item` (`id`)
  ON DELETE NO ACTION
  ON UPDATE NO ACTION)
COMMENT = 'SKU';

CREATE INDEX `fk_sku_item1_idx` ON `mydb`.`sku` (`item_id` ASC);

-----
-- Table `mydb`.`tag`
-----

DROP TABLE IF EXISTS `mydb`.`tag` ;

CREATE TABLE IF NOT EXISTS `mydb`.`tag` (
  `id` BIGINT UNSIGNED NOT NULL AUTO_INCREMENT,
  `name` VARCHAR(60) NOT NULL COMMENT '标签名称',
  `updated` INT(11) NOT NULL,
  `created` INT(11) NULL,
  PRIMARY KEY (`id`))
COMMENT = '标签';

-----
-- Table `mydb`.`item_tag`
-----

DROP TABLE IF EXISTS `mydb`.`item_tag` ;

CREATE TABLE IF NOT EXISTS `mydb`.`item_tag` (
  `item_id` BIGINT NOT NULL,
  `tag_id` BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY (`item_id`, `tag_id`),
  CONSTRAINT `fk_item_tag_item1`
    FOREIGN KEY (`item_id`)
    REFERENCES `mydb`.`item` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT `fk_item_tag_tag1`
    FOREIGN KEY (`tag_id`)
    REFERENCES `mydb`.`tag` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION);

CREATE INDEX `fk_item_tag_tag1_idx` ON `mydb`.`item_tag` (`tag_id` ASC);
```

```
CREATE INDEX `fk_item_tag_item1_idx` ON `mydb`.`item_tag` (`item_id` ASC);
```

生成实体类

我们可以通过命令行生成 Entity 类

```
php artisan doctrine:mapping:import annotation --namespace=ItemBundle\\Entities\\
```

生成 Entity 成功后可以看到以下输出：

```
Importing mapping information from "mydb" entity manager
> writing /app/src/ItemBundle/Entities/Item.php
> writing /app/src/ItemBundle/Entities/ItemType.php
> writing /app/src/ItemBundle/Entities/Sku.php
> writing /app/src/ItemBundle/Entities/Tag.php
```

以 ItemType 实体为例，打开 ItemType.php 可以看到以下内容：

```
<?php

namespace ItemBundle\Entities;

use Doctrine\ORM\Mapping as ORM;

/**
 * ItemType
 *
 * @ORM\Table(name="item_type")
 * @ORM\Entity
 */
class ItemType
{
    /**
     * @var int
     *
     * @ORM\Column(name="id", type="bigint", nullable=false)
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="IDENTITY")
     */
    private $id;
```

```
/**
 * @var string
 *
 * @ORM\Column(name="name", type="string", length=255, nullable=false, options={"comment"="名称"})
 */
private $name;

/**
 * @var string
 *
 * @ORM\Column(name="code", type="string", length=64, nullable=false, options={"comment"="编码"})
 */
private $code;

/**
 * @var int
 *
 * @ORM\Column(name="updated", type="integer", nullable=false)
 */
private $updated;

/**
 * @var int|null
 *
 * @ORM\Column(name="created", type="integer", nullable=true)
 */
private $created;
}
```

生成实体类的 Getter 和 Setter

根据数据表生成的实体类，只有 `private` 的属性，我们还需要为其设置 Getter 和 Setter 方法，可以以下命令生成：

```
php artisan doctrine:generate:entities --filter=ItemType
```

定义接口

一个接口对一个一条路由，一个路由对应一个控制器方法。

定义控制器

我们先来完成 `ItemType` 实体的增删改查操作。

先创建一个带有增删改查空方法的控制器：

```
<?php
//src/ItemBundle/Http/Api/V1/Action/ItemTypesController.php
namespace ItemBundle\Http\Api\V1\Action;

use Illuminate\Http\Request;
use App\Http\Controllers\Controller as BaseController;
use Dingo\Api\Exception\StoreResourceFailedException;

class ItemTypesController extends BaseController
{
    public function index(Request $request) {
        return $this->response->array([]);
    }
    public function save(Request $request) {
        return $this->response->array([]);
    }

    public function show(Request $request, $id) {
        return $this->response->array([$id]);
    }
    public function update(Request $request, $id) {

    }
    public function destroy(Request $request, $id) {

    }
}
```

定义路由

```
<?php
//routes/api/item.php
$api->version('v1', function($api) {
    $api->group(['namespace' => 'ItemBundle\Http\Api\V1\Action', 'middleware'
=> [], 'providers' => 'jwt'], function($api) {
        $api->get('/item_types', ['name' => '获取商品类型列表', 'as' => 'item_t
ypes.list', 'uses'=>'ItemTypesController@index']);
        $api->post('/item_types', ['name' => '创建商品类型', 'as' => 'item_type
s.save', 'uses'=>'ItemTypesController@save']);
        $api->get('/item_types/{id}', ['name' => '获取商品类型详情', 'as' => 'i
tem_types.show', 'uses'=>'ItemTypesController@show']);
        $api->put('/item_types/{id}/', ['name' => '更新商品类型', 'as' => 'item
_types.update', 'uses'=>'ItemTypesController@update']);
        $api->delete('/item_types', ['name' => '删除商品类型', 'as' => 'item_ty
pes.destroy', 'uses'=>'ItemTypesController@destroy']);
    });
});
```

此时，可以通过浏览器访问 http://127.0.0.1:8080/api/item_types 获取商品类型列表接口。

定义接口文档

一、定义 ItemBundle 接口文档分组

基本信息：

```
<?php
//src/ItemBundle/Http/Api/V1/Swagger/Info.php
/**
 * @SWG\Swagger(
 *     swagger="2.0",
 *     schemes={"http"},
 *     host="localhost",
 *     basePath="/espier/public/index.php/api/",
 *     @SWG\Info(
 *         version="1.0",
 *         title="商品相关接口",
 *         description="item api",
 *     ),
 * )
 */
```

接口标签

```
<?php
//src/ItemBundle/Http/Api/V1/Swagger/Tags.php
/**
 * @SWG\Tag(
 *     name="ItemCrud",
 *     description="商品增删改查相关",
 * )
 */

/**
 * @SWG\Tag(
 *     name="ItemTag",
 *     description="商品标签",
 * )
 */
```

接口统一错误响应结构体：


```
<?php
//src/ItemBundle/Http/Api/V1/Swagger/ItemsErrorResponses.php
namespace src\GoodsBundle\Http\Api\V1\Swagger;
/**
 * @SWG\Definition(type="object")
 */
class ItemsErrorResponses
{
    /**
     * @SWG\Property
     * @var string
     */
    public $message;

    /**
     * @SWG\Property
     * @var string
     */
    public $errors;

    /**
     * @SWG\Property(format="int32")
     * @var int
     */
    public $code;

    /**
     * @SWG\Property(format="int32")
     * @var int
     */
    public $status_code;

    /**
     * @SWG\Property
     * @var string
     */
    public $debug;
}
```

二、定义具体接口的输入和输出

```
<?php
//src/ItemBundle/Http/Api/V1/Action/ItemTypesController.php
namespace ItemBundle\Http\Api\V1\Action;
```

```

use Illuminate\Http\Request;
use App\Http\Controllers\Controller as BaseController;
use Dingo\Api\Exception\StoreResourceFailedException;

class ItemTypesController extends BaseController
{
    /**
     * @SWG\Get(
     *     path="/item_types",
     *     summary="获取商品类型列表",
     *     tags={"ItemType"},
     *     description="获取商品类型列表",
     *     operationId="index",
     *     @SWG\Response(
     *         response=200,
     *         description="成功返回结构",
     *         @SWG\Schema(
     *             @SWG\Property(
     *                 property="data",
     *                 type="array",
     *                 @SWG\Items(
     *                     type="object",
     *                     @SWG\Property(property="status", type="string"),
     *                 )
     *             ),
     *         ),
     *     ),
     *     @SWG\Response( response="default", description="错误返回结构", @SWG\
Schema( type="array", @SWG\Items(ref="#/definitions/ItemsErrorResponses") ) )
     * )
     */
    public function index(Request $request) {
        return $this->response->array([]);
    }
    /**
     * @SWG\Post(
     *     path="/item_types",
     *     summary="新增商品类型",
     *     tags={"ItemType"},
     *     description="新增商品类型",
     *     operationId="save",
     *     @SWG\Parameter( name="name", in="formData", description="类型名称",
required=true, type="string"),
     *     @SWG\Parameter( name="code", in="formData", description="类型描述",

```

```

required=true, type="string"),
    *      @SWG\Response(
    *          response=200,
    *          description="成功返回结构",
    *          @SWG\Schema(
    *              @SWG\Property(
    *                  property="data",
    *                  type="array",
    *                  @SWG\Items(
    *                      type="object",
    *                      @SWG\Property(property="id", type="integer"),
    *                      @SWG\Property(property="name", type="string"),
    *                      @SWG\Property(property="code", type="string"),
    *                  )
    *              ),
    *          ),
    *      ),
    *      @SWG\Response( response="default", description="错误返回结构", @SWG\
Schema( type="array", @SWG\Items(ref="#/definitions/ItemsErrorResponses") ) )
    *  )
    */
    public function save(Request $request) {
        return $this->response->array([]);
    }
    /**
    * @SWG\Get(
    *     path="/item_types/{id}",
    *     summary="获取商品详情",
    *     tags={"ItemType"},
    *     description="获取商品类型列表",
    *     operationId="show",
    *     @SWG\Parameter( name="id", in="path", description="类型名称", requi
red=true, type="string"),
    *     @SWG\Response(
    *         response=200,
    *         description="成功返回结构",
    *         @SWG\Schema(
    *             @SWG\Property(
    *                 property="data",
    *                 type="array",
    *                 @SWG\Items(
    *                     type="object",
    *                     @SWG\Property(property="status", type="stirng"),
    *                 )
    *             ),
    *         ),
    *     ),
    * )

```

```

        ),
        @SWG\Response( response="default", description="错误返回结构", @SWG\
Schema( type="array", @SWG\Items(ref="#/definitions/ItemsErrorResponses") ) )
    )
}
*/

public function show(Request $request,$id) {
    return $this->response->array([$id]);
}
/**
 * @SWG\Put(
 *     path="/item_types/{id}",
 *     summary="更新商品类型",
 *     tags={"ItemType"},
 *     description="获取商品类型列表",
 *     operationId="update",
 *     @SWG\Parameter( name="id", in="path", description="类型名称", requi
red=true, type="string"),
 *     @SWG\Parameter( name="name", in="formData", description="类型名称",
required=false, type="string"),
 *     @SWG\Parameter( name="code", in="formData", description="类型描述",
required=false, type="string"),
 *     @SWG\Response(
 *         response=200,
 *         description="成功返回结构",
 *         @SWG\Schema(
 *             @SWG\Property(
 *                 property="data",
 *                 type="array",
 *                 @SWG\Items(
 *                     type="object",
 *                     @SWG\Property(property="status", type="stirng"),
 *                 )
 *             ),
 *         ),
 *     ),
 * ),
 * ),
 * @SWG\Response( response="default", description="错误返回结构", @SWG\
Schema( type="array", @SWG\Items(ref="#/definitions/ItemsErrorResponses") ) )
 * )
 */

public function update(Request $request, $id) {

}
/**
 * @SWG\Delete(
 *     path="/item_types/{id}",
 *     summary="删除商品类型",

```

```

        *      tags={"ItemType"},
        *      description="删除商品类型",
        *      operationId="destroy",
        *      @SWG\Parameter( name="id", in="path", description="类型名称", requi
red=true, type="string"),
        *      @SWG\Response(
        *          response=200,
        *          description="成功返回结构",
        *          @SWG\Schema(
        *              @SWG\Property(
        *                  property="data",
        *                  type="array",
        *                  @SWG\Items(
        *                      type="object",
        *                      @SWG\Property(property="status", type="stirng"),
        *                  )
        *              ),
        *          ),
        *      ),
        *      @SWG\Response( response="default", description="错误返回结构", @SWG\
Schema( type="array", @SWG\Items(ref="#/definitions/ItemsErrorResponses") ) )
        *  )
        */
        public function destroy(Request $request, $id) {

        }

    }
}

```

三、接口文档

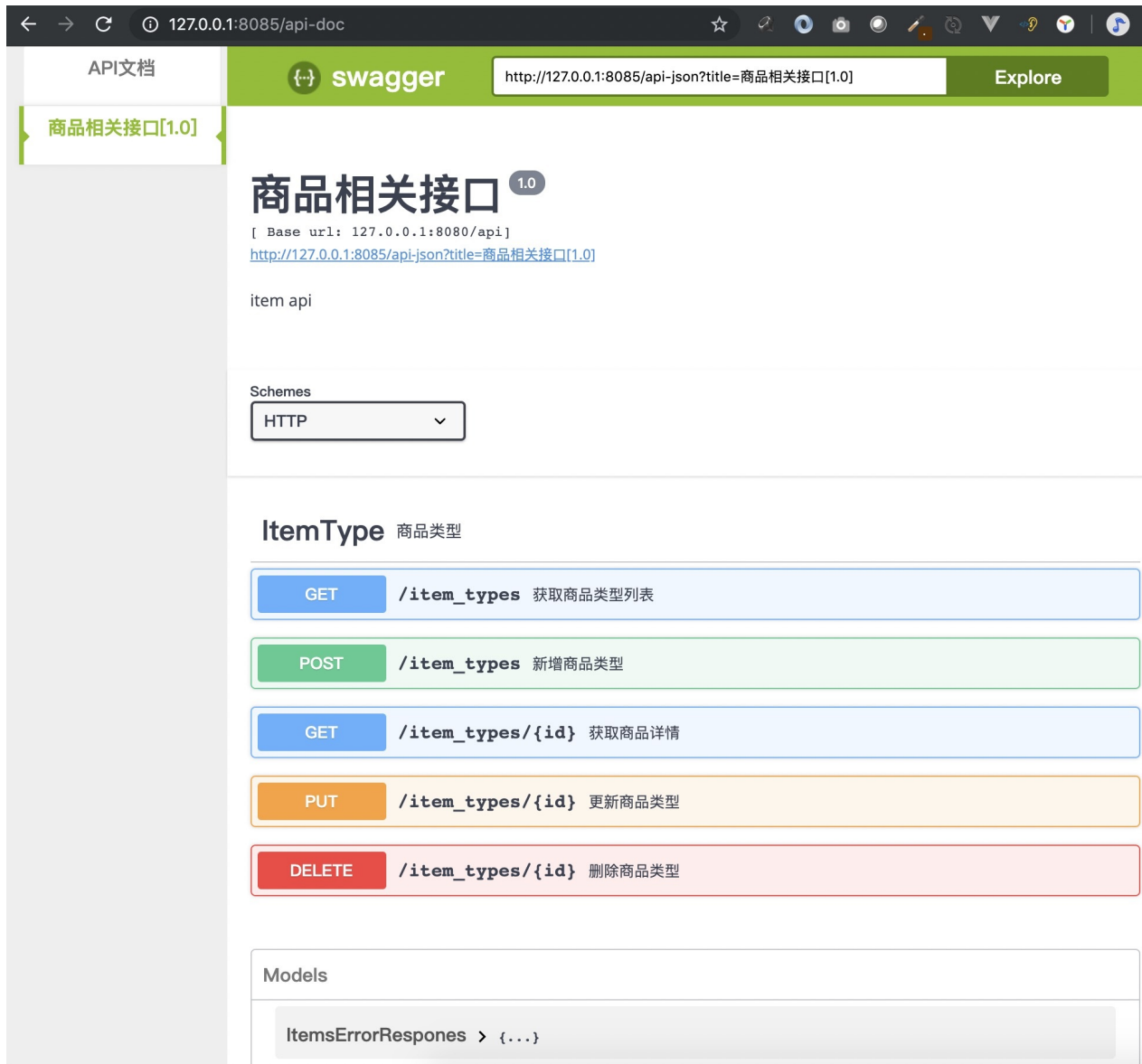
接口文档的生成请参阅 [\[接口文档生成和接口测试\]](#)

```

php artisan api:swagger --output=src/ItemBundle/Http/Api/V1 && php -S 0.0.0.0
:8005 -t public/

```

成功之后，将会看到以下界面：



实现商品类型的 CRUD

我们以 MVC 架构为基础，加入 [Repository 模式](#) 和 [Service 模式](#) 来完成我们的开发。

第一步：新增相关类

新建与 `ItemType` 相关的 `Service` 和 `Repository` 类。

新建 `ItemTypeService` 类：

```
//src/ItemBundle/Services/ItemTypeService.php
<?php
namespace ItemBundle\Services;

class ItemTypeService
{

}
```

新建 `ItemTypeRepository` 类：

```
//src/ItemBundle/Repositories/ItemTypeRepository.php
<?php
namespace ItemBundle\Repositories;
use Doctrine\ORM\EntityRepository;

class ItemTypeRepository extends EntityRepository
{

}
```

同时修改 `ItemType` 类与之相关联：

```
<?php
//src/ItemBundle/Entities/ItemType.php
namespace ItemBundle\Entities;

use Doctrine\ORM\Mapping as ORM;
/**
 * ItemType
 *
 * @ORM\Table(name="item_type")
 * @ORM\Entity(repositoryClass="ItemBundle\Repositories\ItemTypeRepository")
 */
class ItemType
{
    //...
}
```

第二步：确定调用关系

采用 Service 和 Repository 模式，相关类的调用关系为 `Controller->Service->Repository`。

我们利用 Laravel 自带的依赖注入的方式，在 Controller 中新增构造方法来初始化 Service 类

```
<?php
//src/ItemBundle/Http/Api/V1/Action/ItemTypesController.php
namespace ItemBundle\Http\Api\V1\Action;

use Illuminate\Http\Request;
use App\Http\Controllers\Controller as BaseController;
use Dingo\Api\Exception\StoreResourceFailedException;
use ItemBundle\Entities\ItemType;
use Dingo\Api\Exception\ResourceException;
use ItemBundle\Services\ItemTypeService;

class ItemTypesController extends BaseController
{
    protected $itemTypeService;

    public function __construct(ItemTypeService $itemTypeService)
    {
        $this->itemTypeService = $itemTypeService;
    }

    //...
}
```

因为初始化 Repository 类需要特定的方法，所以在 ItemTypeService 类中，我们需要手动初始化 Repository 类：


```
//src/ItemBundle/Services/ItemTypeService.php
<?php
namespace ItemBundle\Services;

use ItemBundle\Entities\ItemType;

class ItemTypeService
{
    protected $itemTypeRepository;

    public function __construct()
    {
        $this->itemTypeRepository = app('registry')->getManager('default')->getRepository(ItemType::class);
    }
}
```

第三步：编写逻辑

ItemTypeRepository 类

```
//src/ItemBundle/Repositories/ItemTypeRepository.php
<?php
namespace ItemBundle\Repositories;

use ItemBundle\Entities\ItemType;
use Doctrine\ORM\EntityRepository;

class ItemTypeRepository extends EntityRepository
{

    public function store(ItemType $itemType)
    {
        $em = $this->getEntityManager();
        $em->persist($itemType);
        $em->flush();
        return $itemType;
    }
    public function delete(ItemType $itemType)
    {
        $em = $this->getEntityManager();
        $em->remove($itemType);
        $em->flush();
        return $itemType;
    }
}
```

ItemTypeService 类:

```
//src/ItemBundle/Services/ItemTypeService.php
<?php
namespace ItemBundle\Services;

use ItemBundle\Entities\ItemType;

class ItemTypeService
{
    protected $itemTypeRepository;

    public function __construct()
    {
        $this->itemTypeRepository = app('registry')->getManager('default')->getRepository(ItemType::class);
    }
}
```

```
public function create($params)
{
    $itemType = new ItemType();
    $itemType->setName($params['name']);
    $itemType->setCode($params['code']);
    $itemType->setCreated(time());

    return $this->itemTypeRepository->store($itemType);
}

public function update($id,$data)
{
    $itemType = $this->itemTypeRepository->find($id);

    if( !$itemType ) {
        throw new ResourceException("未查询到更新数据");
    }
    $itemType->setName($data['name']);
    $itemType->setCode($data['code']);
    $itemType->setUpdated(time());
    return $this->itemTypeRepository->store($itemType);
}

public function deleteById($id)
{
    $itemType = $this->itemTypeRepository->find($id);

    if( !$itemType ) {
        throw new ResourceException("未查询到更新数据");
    }
    return $this->itemTypeRepository->delete($itemType);
}

public function findAll()
{
    return $this->itemTypeRepository->findAll();
}

public function find($id)
{
    return $this->itemTypeRepository->find($id);
}

public function toArray(ItemType $itemType)
{
    return [
        'id'=>$itemType->getId(),
        'name'=>$itemType->getName(),
        'code'=>$itemType->getCode(),
        'created'=>$itemType->getCreated(),
    ]
}
```

```

        'updated'=>$itemType->getUpdated(),
    ];
}
}

```

控制器

```

<?php
//src/ItemBundle/Http/Api/V1/Action/ItemTypesController.php
namespace ItemBundle\Http\Api\V1\Action;

use Illuminate\Http\Request;
use App\Http\Controllers\Controller as BaseController;
use Dingo\Api\Exception\StoreResourceFailedException;
use ItemBundle\Entities\ItemType;
use Dingo\Api\Exception\ResourceException;
use ItemBundle\Services\ItemTypeService;

class ItemTypesController extends BaseController
{
    protected $itemTypeService;

    public function __construct(ItemTypeService $itemTypeService)
    {
        $this->itemTypeService = $itemTypeService;
    }

    public function index(Request $request) {
        $list = $this->itemTypeService->findAll();
        foreach ($list as $itemType) {
            $result[] = $this->itemTypeService->toArray($itemType);
        }
        return $this->response->array($result);
    }

    public function save(Request $request) {
        $rules = [
            'name'    => ['required', '类型名称不能为空'],
            'code'    => ['required', '类型编码不能为空'],
        ];
        $params = $request->all(array_keys($rules));
        $error = validator_params($params, $rules);

        if($error) {
            throw new StoreResourceFailedException($error);
        }
    }
}

```

```
$itemType = $this->itemTypeService->create($params);

return $this->response->array(['id'=>$itemType->getId()]);
}

public function show(Request $request,$id) {
    $itemType = $this->itemTypeService->find($id);
    return $this->response->array($this->itemTypeService->toArray($itemTy
pe));
}

public function update(Request $request, $id) {
    $rules = [
        'name'    => ['required', '类型名称不能为空'],
        'code'    => ['required', '类型编码不能为空'],
    ];
    $params = $request->all(array_keys($rules));
    $error = validator_params($params, $rules);

    if($error) {
        throw new StoreResourceFailedException($error);
    }

    $itemType = $this->itemTypeService->update($id,$params);

    return $this->response->array(['status'=>'succ']);
}

public function destroy(Request $request, $id) {
    $itemType = $this->itemTypeService->deleteById($id);
    return $this->response->array(['status'=>'succ']);
}
}
```

未完待续

5机集群

测试场景

- 压测时间：共计15分钟
- 压力并发数设置为2000并发
- 首页，访问占比50%
- 自定义分类页，访问占比35%
- 商品详情页，访问占比12%
- 下单操作，访问占比3%，其中下单操作包含如下行为：
 - 会员登陆
 - 加入购物车
 - 进行下单操作

压测工具

tsung

测试环境

主机名	配置	角色
web1	8core,16G mem,100G SSD	web服务
web2	8core,16G mem,100G SSD	web服务
web3	8core,16G mem,100G SSD	web服务
tsung	8core,16G mem	压力测试服务
mysql	8core,16G	数据库服务
Job	8core,16G mem	脚本服务器

服务器架构



Stats Report(统计数据报告)

Main Statistics(主要统计数据)

- connect: 连接
- page: 页面（一个页面表示一组没有被thinktime间隔的请求）
- request: 请求
- session: 会话
- Highest 10sec mean: 最高10秒均值
- Lowest 10sec mean: 最低10秒均值
- Highest rate: 最高速率
- Mean rate: 平均处理速率
- Mean: 平均持续时间
- Count: 总数 | Name | highest 10sec mean | lowest 10sec mean | Highest Rate | Mean Rate
| Mean | Count | | :--- | :--- | :--- | :--- | :--- | :--- | :--- | | connect | 27.52 msec | 2.40 msec | 1731 /
sec | 1584.80 / sec | 4.12 msec | 1438497 | | page | 0.47 sec | 43.19 msec | 1732.2 / sec |

1585.18 / sec | 71.36 msec | 1438077 || request | 0.23 sec | 21.05 msec | 3580.7 / sec |
 3267.45 / sec | 34.35 msec | 2964134 || session | 0.49 sec | 45.95 msec | 1788.8 / sec |
 1585.19 / sec | 74.28 msec | 1438067 |

Transactions Statistics(事务统计)

- tr_index:首页
- tr_item:商品详细页
- tr_order_create:订单提交
- Highest 10sec mean:最长持续时间
- Lowest 10sec mean:最短持续时间
- Highest rate:最高速率
- Mean rate:平均处理速率
- Mean:平均持续时间
- Count:总数 | Name | highest 10sec mean | lowest 10sec mean | Highest Rate | Mean Rate |
 Mean | Count || :--- | :--- | :--- | :--- | :--- | :--- || tr_crate_order | 4.76 sec | 0.99 sec | 60.6 /
 sec | 47.89 / sec | 1.23 sec | 43347 || tr_index | 0.63 sec | 21.37 msec | 869 / sec | 793.37 /
 sec | 52.04 msec | 719575 || tr_itemdetail | 0.15 sec | 6.58 msec | 209.5 / sec | 189.39 /
 sec | 14.27 msec | 171947 || tr_productlist | 0.22 sec | 8.09 msec | 601.5 / sec | 554.52 /
 sec | 20.51 msec | 503208 |

Network Throughput(网络流量)

- size_rcv:响应请求数据量
- size_rcv:发送请求数据量
- Highest rate:每秒最高 响应/发送 请求数据量
- Total:响应/发送 请求总数据量 | Name | Highest Rate | Total || :--- | :--- | :--- || size_rcv |
 433.80 Mbits/sec | 44.93 GB || size_sent | 6.42 Mbits/sec | 660.59 MB |

Counters Statistics(计数统计)

- connected:表示会话开始且尚未结束，并且已建立连接的最大用户数
- finished_users_count:表示已经完成会话的最大用户数
- users:表示会话开始且尚未结束的最大用户数
- users_count:表示Tsong总共生成的用户总数 | Name | Max || :--- | :--- || connected | 493 ||
 finish_users_count | 1439058 || users | 1708 || users_count | 1439058 |

Server monitoring(服务器监控)

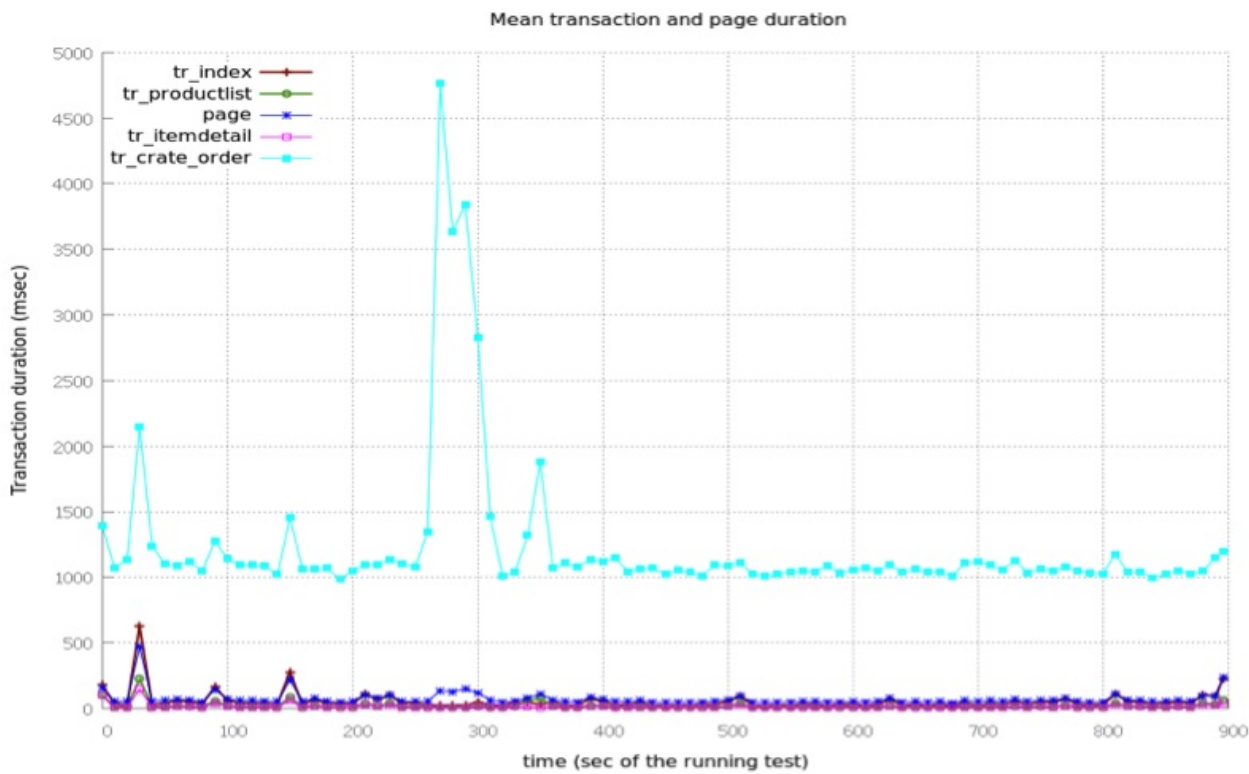
Name	highest 10sec mean	lowest 10sec mean
cpu:os_mon@tsung_controller@uat-xinbai-canal-1	92.69 %	77.16 %
cpu:os_mon@xinbai-job	45.85 %	24.63 %
cpu:os_mon@xinbai-kv	3.38 %	2.01 %
cpu:os_mon@xinbai-mysql-master	58.33 %	23.92 %
cpu:os_mon@xinbai-web1	80.36 %	54.96 %
cpu:os_mon@xinbai-web2	82.65 %	56.88 %
cpu:os_mon@xinbai-web3	89.15 %	56.48 %
freemem:os_mon@tsung_controller@uat-xinbai-canal-1	14851.02 MB	14699.86 MB
freemem:os_mon@xinbai-job	13976.99 MB	13303.06 MB
freemem:os_mon@xinbai-kv	14271.77 MB	14261.32 MB
freemem:os_mon@xinbai-mysql-master	10087.31 MB	9821.55 MB
freemem:os_mon@xinbai-web1	10980.93 MB	10625.84 MB
freemem:os_mon@xinbai-web2	11013.51 MB	10781.66 MB
freemem:os_mon@xinbai-web3	10897.38 MB	10693.00 MB
load:os_mon@tsung_controller@uat-xinbai-canal-1	5.48	1.73
load:os_mon@xinbai-job	4.34	0.33
load:os_mon@xinbai-kv	0.53	0.00
load:os_mon@xinbai-mysql-master	4.58	2.86
load:os_mon@xinbai-web1	49.98	7.79
load:os_mon@xinbai-web2	51.97	8.24
load:os_mon@xinbai-web3	48.93	9.32

HTTP return code(HTTP响应状态码统计)

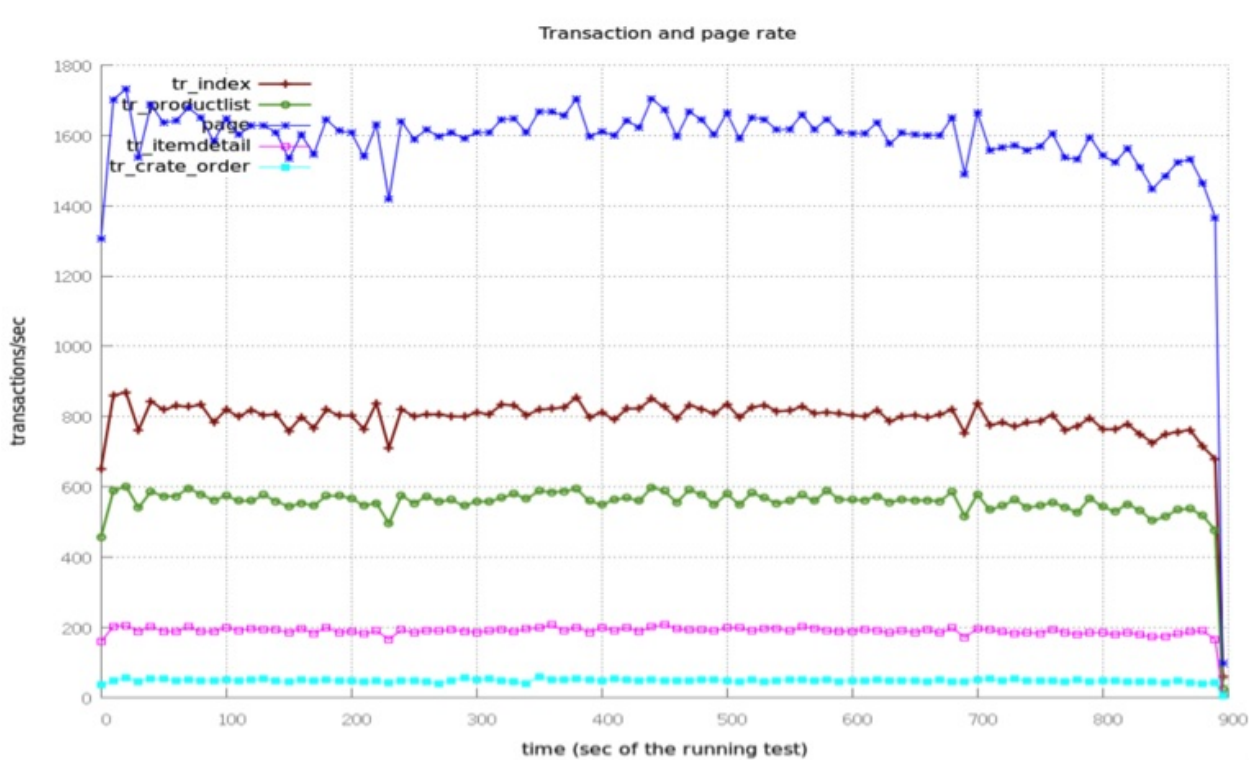
- Code:http状态码(200: 表示客户端请求已成功响应, 302:请求暂时重定向)
- Highest Rate:状态码返回最高速率
- Mean Rate:状态码返回平均速率
- Total number:返回状态码的总个数 | Code | Highest Rate | Mean Rate | Total number | | :--- | :--- | :--- | :--- | | 200 | 3580.7 / sec | 3267.29 / sec | 2966208 |

Graphical Reports(图形化报告及分析)

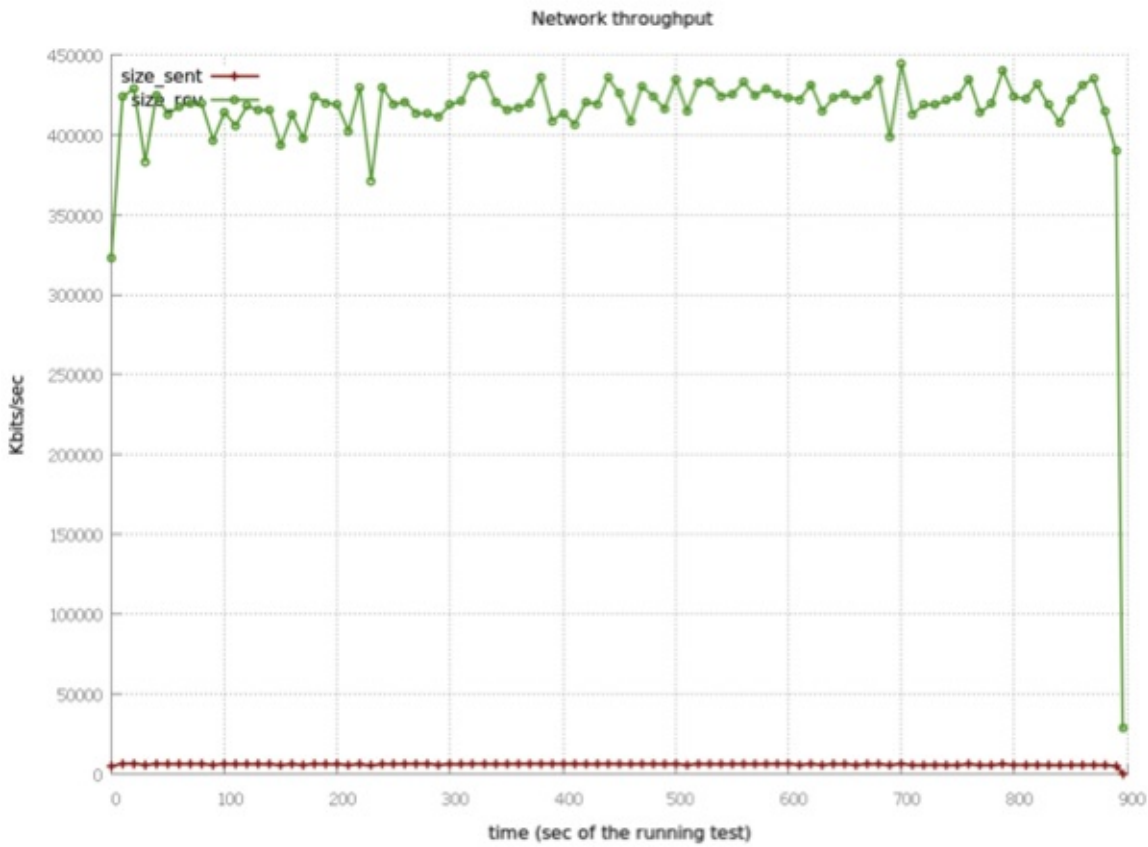
事物响应时间



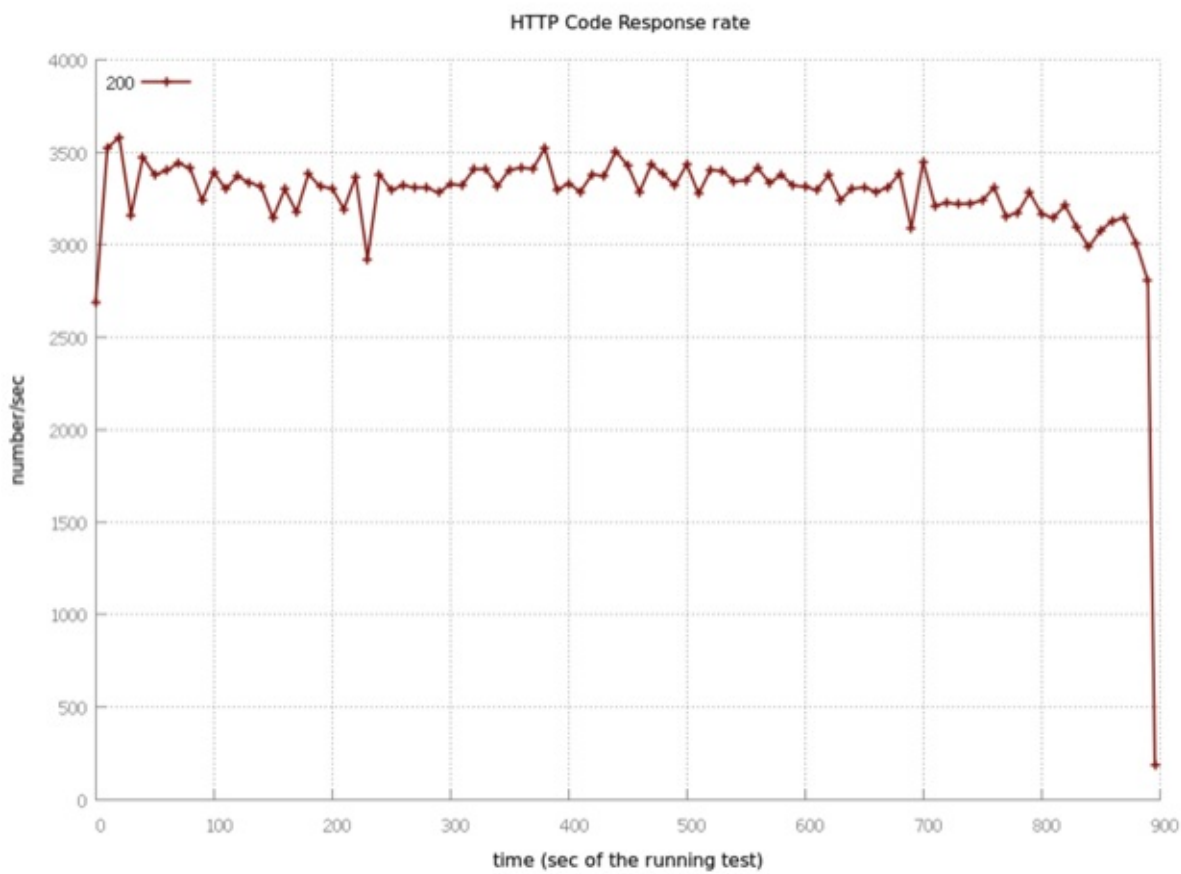
事物及页面传输速度



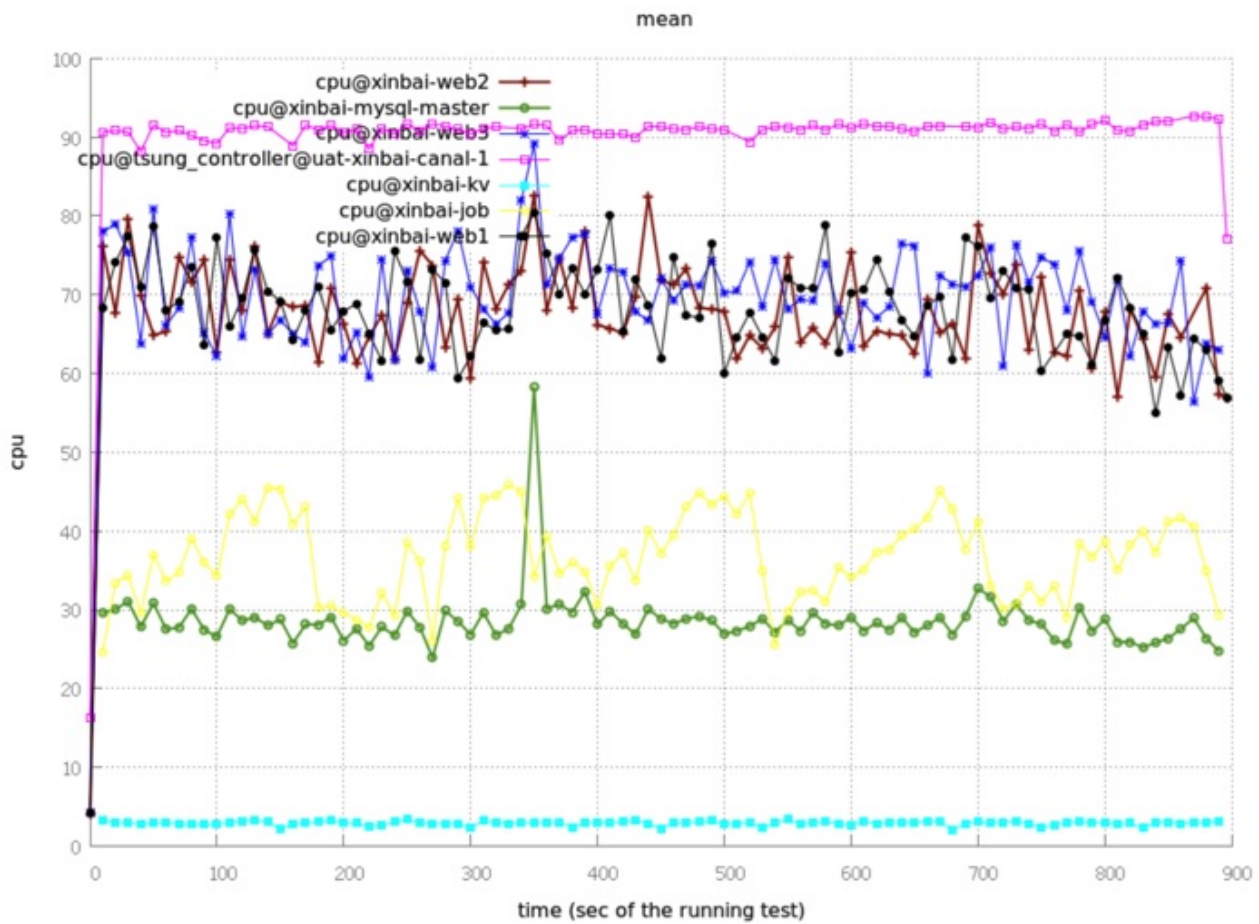
网络传输



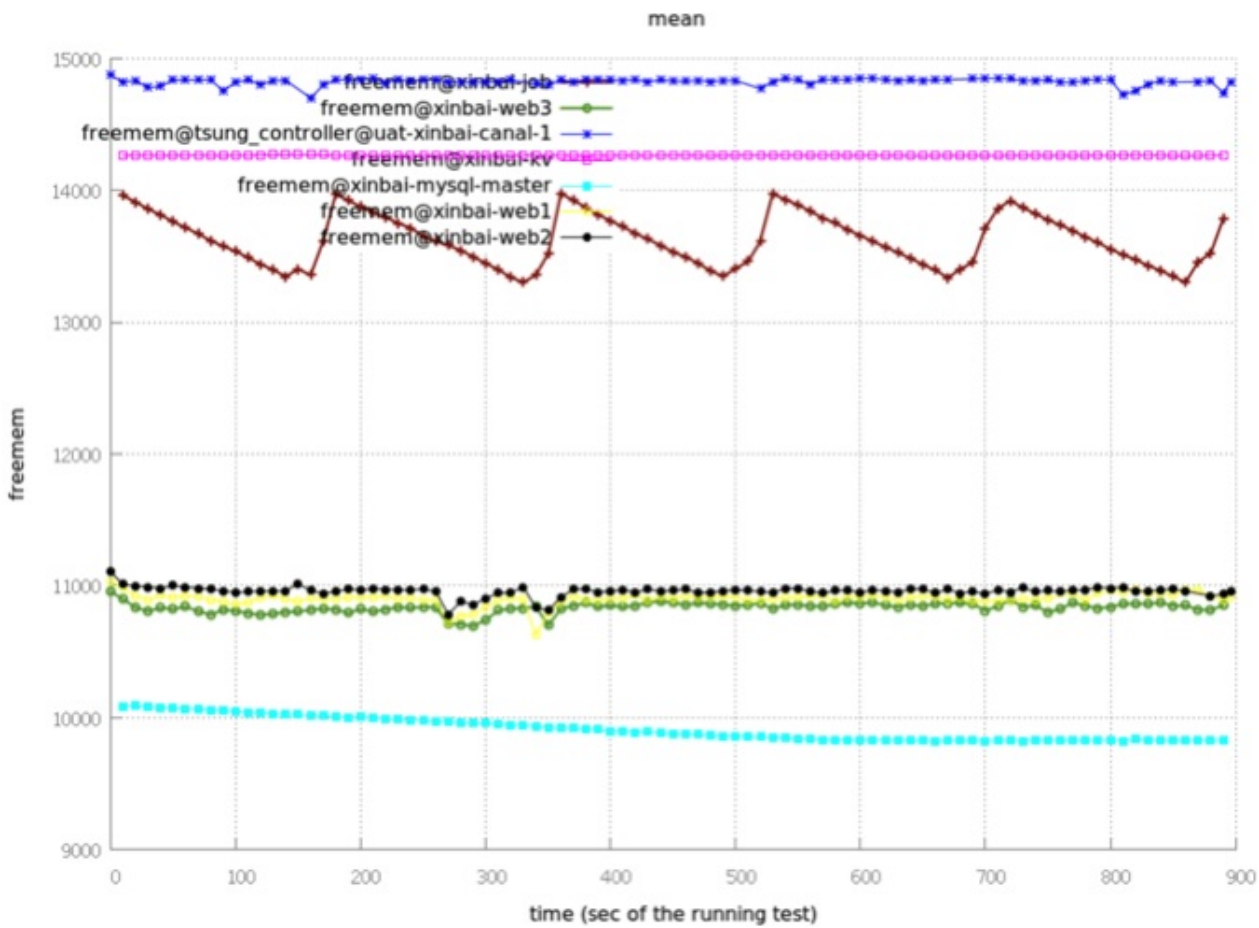
http 响应状态



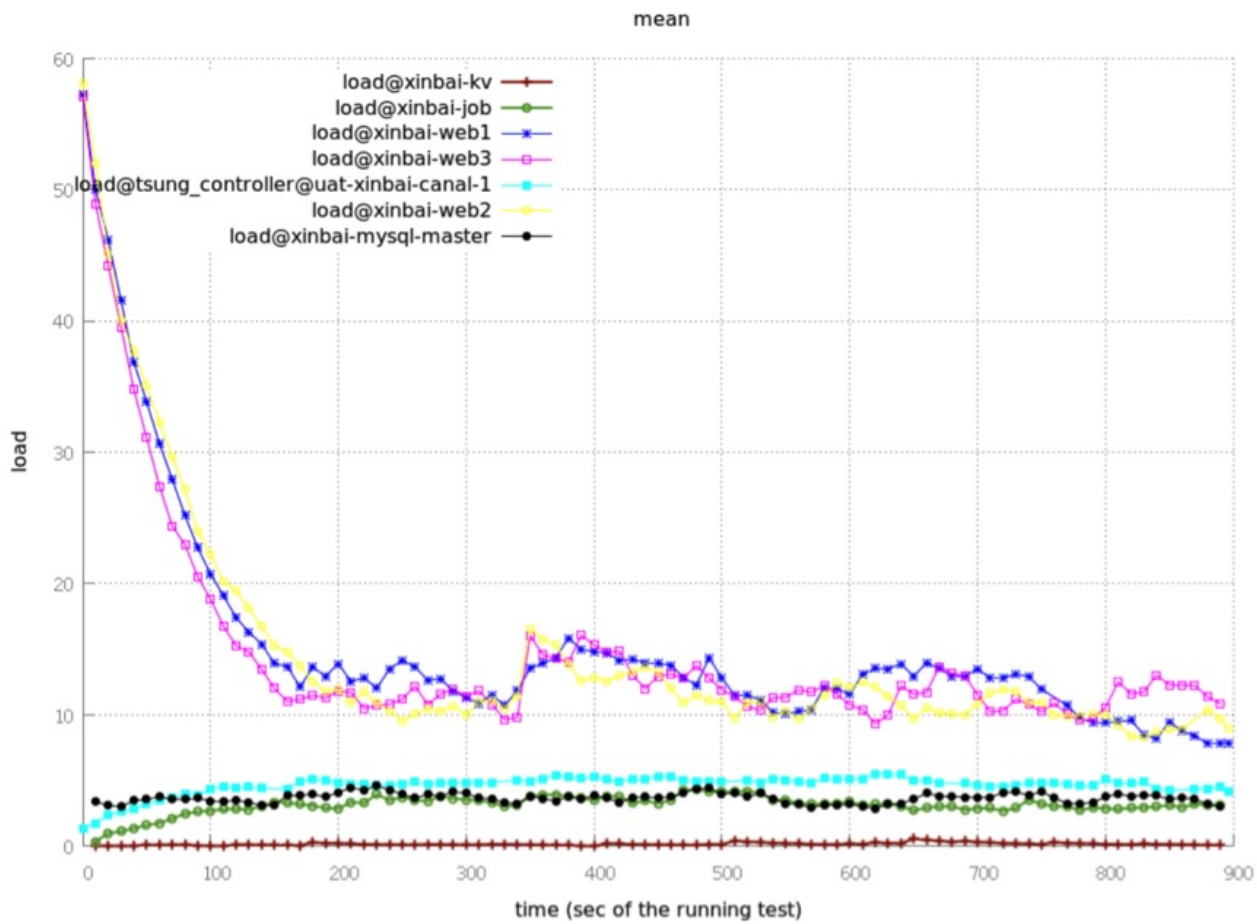
服务器CPU使用率



服务器内存富余量



服务器cpu负载



FAQ

1、用阿里云oss上传文件报错

```
lumen.ERROR: OSS\Core\OssException: : RequestId: 5EC37C223D124D353126DC73  
in /Users/lujunyi/Sites/yuanyuanke/amore/amorestore/qianyi/development/espier-  
bloated/vendor/aliyuncs/oss-sdk-php/src/OSS/Result/Result.php:97 解决： a、一般都是  
配置错了，检查阿里oss配置 b、检查php的时区配置
```

2、本地部署环境的一些问题

其实也很简单，只是关键上的问题都出现了理解上的错误。

1、docker环境需要和匹配的代码版本一致才能顺利部署

2、测试环境 需要导入线上数据库数据 并且线上版本的数据库要和本地搭建的数据库版本一致才能正常匹配数据

3、配置文件需要与线上一致 除了mysql neo4j redis 需要配置成本地的 其他的都需要与线上的
一直。

上面这3个问题确认好了很快就能部署出来本地的开发环境。

4、IS_SAAS,SYSTEM_COMPANY_ID等的配置问题