爱旅行项目iTrip—开发规范

爱旅行项目架构从两个方向进行设计：前端架构和后端架构，将前端和后端完全分离。整个项目开发过程由项目组的前端团队与后端团队共同协作完成，需要遵循开发规范如下：

## 1.1 后端- iTripBackend开发规范

### 1.1.1 命名规范

【强制】类名、方法名、参数名、成员变量、局部变量、数据库表的字段名都统一使用 lowerCamelCase 风格，必须遵从驼峰形式。

正例： localValue / getHttpMessage() / inputUserId

下面将逐一详细说明介绍：

#### 1.1.1.1 包名

1. 【强制】统一使用小写字母
2. 以cn.itrip开头，代码结构层次清晰
   1. cn.itrip.controller
   2. cn.itrip.service
   3. cn.itrip.dao
   4. cn.itrip.beans
      1. cn.itrip.beans.vo
      2. cn.itrip.beans.pojo
      3. cn.itrip.beans.dto
   5. cn.itrip.common
3. 配置文件统一放置在/resources目录下

#### 1.1.1.2 类名

1. Controller

以Controller结尾，比如：HotelController.java

1. 业务类

根据具体业务进行分包放置，比如：cn.itrip.service.itriphotel

* 1. 接口

以Service结尾，比如：ItripHotelService.java

* 1. 接口实现类

以Impl结尾，比如：ItripHotelServiceImpl.java

1. Dao

根据具体业务进行分包放置，比如：cn.itrip.dao.itriphotel

* 1. mapper接口

以Mapper结尾，比如：ItripHotelMapper.java

* 1. mapper文件

与mapper接口同名，比如：ItripHotelMapper.xml

1. 实体类

统一放置在cn.itrip.beans包下，具体分为：

* 1. pojo

与数据库表一一对应，并与表名同名，比如：ItripHotel.java，属性名与数据库字段名一一对应，均采用驼峰规则定义。

* 1. vo（view object）

根据业务功能创建各自VO，并根据业务分包放置，命名规则：业务功能名+VO.java，（注：类名中的VO须大写）

* 1. dto

数据传输对象，包括输出给前端的Dto以及接收前端传入的InputDto

#### 1.1.1.3 变量/属性命名

必须统一遵从驼峰形式。前后端变量名、以及数据库字段名都使用同一套命名规则，可有效减少出错率，提高工作效率。

【强制】常量命名全部大写，单词间用下划线隔开，力求语义表达完整清楚，不要嫌名字长。

正例： MAX\_STOCK\_COUNT

反例： MAX\_COUNT

#### 1.1.1.4 方法命名

【参考】service/dao 层方法命名规约：

1. 获取单个对象的方法用 get 做前缀。
2. 获取多个对象的方法用 list 做前缀。
3. 获取统计值的方法用 count 做前缀。
4. 插入的方法用 save或 insert或add（推荐）做前缀。
5. 删除的方法用 remove或 delete（推荐） 做前缀。
6. 修改的方法用 update 做前缀。

### 1.1.3 注解的使用

【强制】该项目的service层和controller层均统一使用@Resource注解，不允许使用@Autowired注解。

### 1.1.4 模块化开发

iTripBackend分为以下四个Web模块：

1. 主业务模块（itripbiz）：包括酒店业务、旅游业务、机票业务、攻略业务
2. 搜索模块（itripsearch）：爱旅行项目中所有搜索功能（solr）
3. 认证模块（itripauth）：负责用户身份验证，生成并维护Token
4. 支付模块（itrptrade）：包括支付宝支付、微信支付

对于上述Web模块，都存在一些公用的部分（比如：bean、dao、utils），为提高代码复用性及可维护性，故采用Maven进行多Module管理，一共7个Module(itripbiz、itripsearch、itripauth、itriptrade、itripdao、itriputils、itripbeans)，具体Module结构及atifactId如图1、2所示：

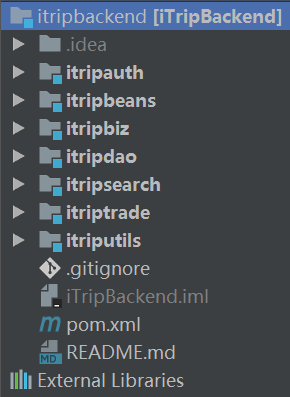
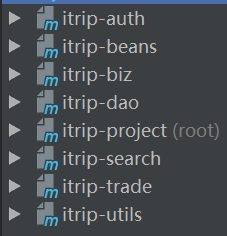
 

图1 iTripBackend-模块结构 图2 artifactId

注意：各模块之间存在依赖，切不可存在相互依赖。依赖关系如下：

1. itripdao模块depedency：itrip-beans、itrip-utils
2. itriputils模块depedency：itrip-beans
3. itripbiz模块depedency：itrip-dao
4. itripsearch模块depedency：itrip-utils
5. itriptrade模块depedency：itrip-dao
6. itripauth模块depedency：itrip-dao

各模块具体说明如下：

1. itripbeans：包括dto、pojo、vo

注：该模块需要打成jar包，供其他Web模块使用

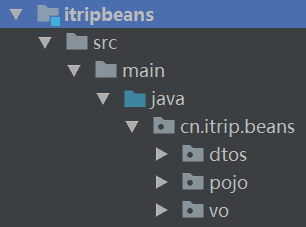


图3 itripbeans

1. itripdao：包括dao层所有代码

注：该模块需要打成jar包，供其他Web模块使用

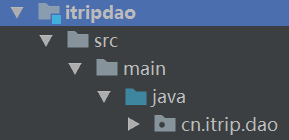


图4 itripdao

1. itriputils：包括common包下所有工具类

注：该模块需要打成jar包，供其他Web模块使用

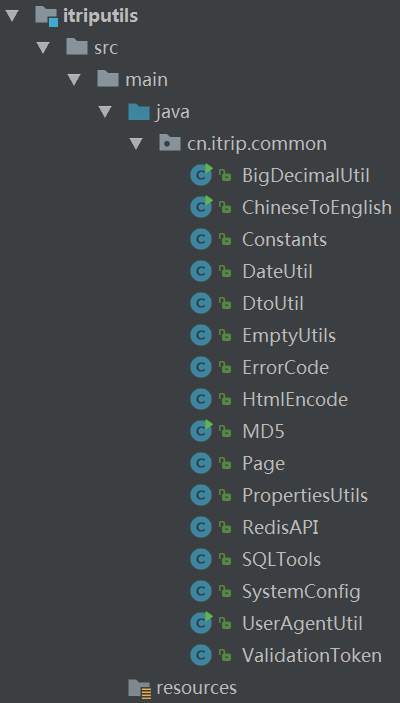


图5 itriputils

1. itripbiz：包括controller、service层以及resources、webapp

注：该web模块需要打成war包，部署路径：/biz

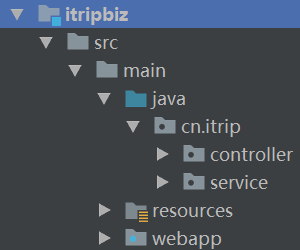


图6 itripbiz

1. itripsearch：包括controller、service、dao层以及resources、webapp

注：该web模块需要打成war包，部署路径：/ search

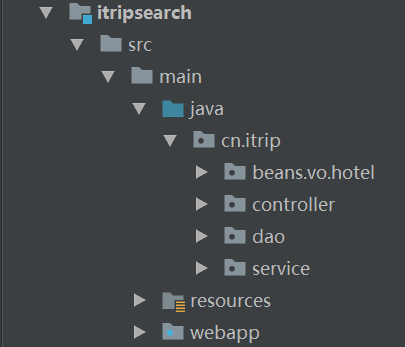


图7 itripsearch

1. itripauth：包括controller、service层以及resources、webapp

注：该web模块需要打成war包，部署路径：/ auth

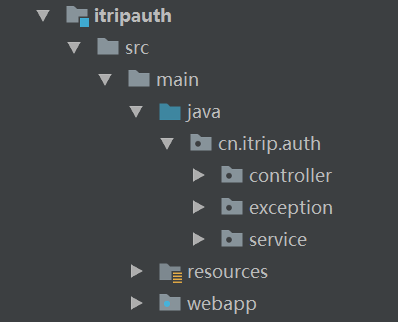


图8 itripauth

1. itriptrade：包括controller、service层以及resources、webapp

注：该web模块需要打成war包，部署路径：/ itrade

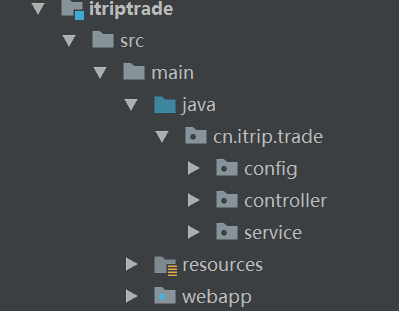


图9 itriptrade

基于以上多模块开发，需要遵守以下的代码约定：

#### 1.1.4.1 itripbeans模块

1. pojo：代码自动生成
2. dto：数据传输对象，包括输出前端Dto以及接收前端传入的InputDto，数据结构如下：

1> Dto .java

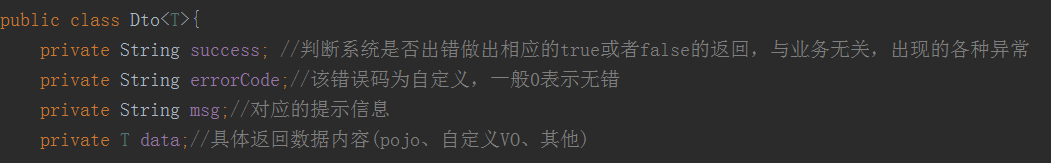


图10 dto

根据service返回的结果，Controller层的API定义往前端输出的数据内容（json格式），e.g：

图11正确返回 图12 错误返回

注意：错误码errorCode根据业务定义

|  |  |
| --- | --- |
| **业务模块** | **模块规则** |
| itripbiz | 1开头（100000） |
| itripsearch | 2开头（200000） |
| itripauth | 3开头（300000） |
| itrptrade | 4开头（400000） |

2> InputDto.java

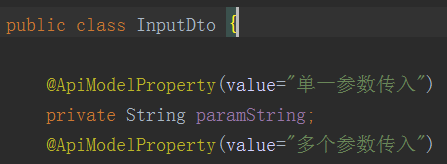


图11 Inputdto

1. vo：根据业务功能创建各自VO，并根据业务分包放置，如图12所示：

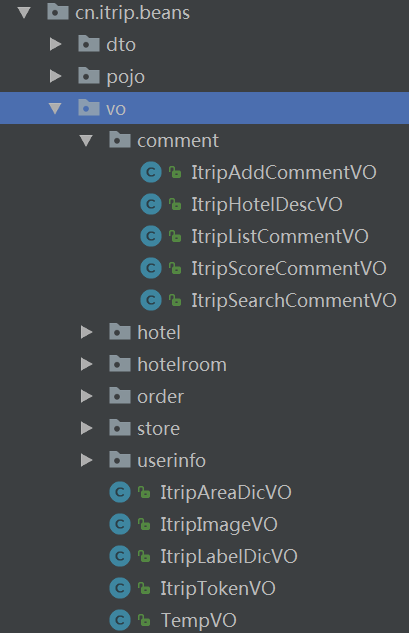


图12 vo

注意：VO是前后端参数传递的接收对象，需要加入Swagger的@ApiModel，进行属性的详细说明及输入要求。

#### 1.1.4.2 itripdao模块

该模块的初始化代码（普通的增、删、改、查）均是自动生成，若因业务需求，则需新增接口方法。

#### 1.1.4.3 itriputils模块

所有的工具类都放置在此模块内，注意：不能使用注解，以降低耦合度。

#### 1.1.4.4 itripbiz模块

1. service层

1> 均需throws Exception，以便事务控制。

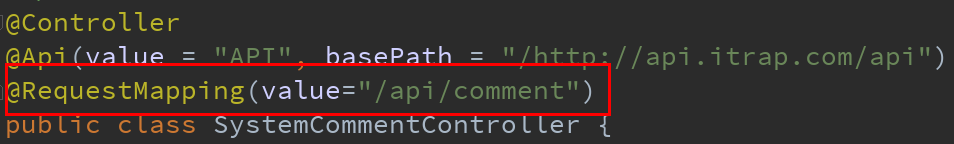
2> 若业务需进行事务处理，则需在接口方法名加上前缀itriptx，比如：

public Integer itriptxModifyItripHotelOrder(ItripHotelOrder itripHotelOrder)throws Exception;

1. controller层

1> 提供API接口，各web工程都需要加入Swagger，以便给前端生成API文档

2> Controller必须加统一前缀：/api/\*\*\*，如图13所示：



注：其他模块（itripsearch、itriptrade、itripauth）与该模块规范一致，此处不再赘述。

### 1.1.5 常用字典对照表

#### 1.1.5.1 常用值

|  |  |  |
| --- | --- | --- |
| **名称** | **值** | **说明** |
| 返回结果 | 0：失败/否/未激活/有待改善  1：成功/是/已激活/值得推荐 | 适用于：  数据库是否的标识值 |
| 性别 | 0：女  1：男 |  |
| 区域级别 | 0: 国家级  1: 省级  2: 市级  3: 县/区 |  |
| 国家 | 1: 国内  2：国外 |  |
| 商品类型 | 0: 旅游产品  1: 酒店产品  2: 机票产品 |  |
| 酒店级别 | 1: 经济酒店  2: 二星级酒店  3: 三星级酒店  4: 四星级酒店  5: 五星级酒店 |  |
| 订单类型 | 0: 旅游产品  1: 酒店产品  2: 机票产品 |  |
| 订单状态 | 0：待支付  1: 已取消  2: 支付成功  3: 已消费 |  |
| 订单支付方式 | 1: 支付宝  2: 微信  3: 到店付 | 订单所选择的支付方式 |
| 发票类型 | 0：个人  1：公司 |  |
| 客户端预定类型 | 0: WEB端  1: 手机端  2: 其他客户端 |  |
| 酒店预定支付方式 | 1: 在线付  2: 到店付  3: 不限 | 预定酒店所规定的支付方式 |
| 图片类型 | 0: 酒店图片  1: 房间图片  2: 评论图片 |  |
| 用户类型 | 0: 自注册用户  1: 微信登录  2: QQ登录  3: 微博登录 |  |
| 证件类型 | 0: 身份证  1: 护照  2: 学生证  3: 军人证  4: 驾驶证  5: 旅行证 |  |

#### 1.1.5.2 常用名称

|  |  |
| --- | --- |
| **名称** | **英文** |
| 商品 | goods |
| 货品 | product |
| 规格 | spec |
| 订单 | order |
| 会员 | member |
| 积分 | point |
| 购物车 | cart |
| 结算 | checkout |
| 订单 | order |
| 品牌 | brand |
| 分类 | cat |
| 优惠劵 | coupon |
| 支付 | payment |
| 团购 | groupbuy |
| 虚拟 | virtual |
| 发票 | receipt |
| 属性 | prop |
| 参数 | param |
| 标签 | tag |
| 地区 | region |

### 1.1.7注释规约

1. 【强制】方法内部单行注释，在被注释语句上方另起一行，使用//注释。方法内部多行注释使用/\* \*/注释，注意与代码对齐。
2. 【推荐】与其“半吊子”英文来注释，不如用中文注释把问题说清楚。专有名词与关键字保持英文原文即可。

反例：“TCP 连接超时”解释成“传输控制协议连接超时”，理解反而费脑筋。

1. 【推荐】代码修改的同时，注释也要进行相应的修改，尤其是参数、返回值、异常、核心逻辑等的修改。

说明：代码与注释更新不同步，就像路网与导航软件更新不同步一样，如果导航软件严重滞后，就失去了导航的意义。

1. 【参考】注释掉的代码尽量要配合说明，而不是简单的注释掉。

说明：代码被注释掉有两种可能性：

1）后续会恢复此段代码逻辑。

2）永久不用。前者如果没有备注信息，难以知晓注释动机。

后者建议直接删掉（代码仓库保存了历史代码）。

1. 【参考】对于注释的要求：第一、能够准确反应设计思想和代码逻辑；第二、能够描述业务含义，使别的程序员能够迅速了解到代码背后的信息。完全没有注释的大段代码对于阅读者形同天书，注释是给自己看的，即使隔很长时间，也能清晰理解当时的思路；注释也是给继任者看的，使其能够快速接替自己的工作。
2. 【参考】好的命名、代码结构是自解释的，注释力求精简准确、表达到位。避免出现注释的一个极端：过多过滥的注释，代码的逻辑一旦修改，修改注释是相当大的负担。

反例：

// put elephant into fridge

put(elephant, fridge);

方法名 put，加上两个有意义的变量名 elephant 和 fridge，已经说明了这是在干什么，语义清晰的代码不需要额外的注释。

## 1.2 前端- iTripFront开发规范

1、【强制】前端的健壮性开发，需要达到：在无API的情况下，前端一样能够完美奔跑

2、【推荐】前端采用mock开发

## 1.3 前后端交互规范

1. 【强制】前后端开发前必须先定义schema
2. 【强制】API不能干涉产品，只提供数据
3. 【强制】前后端边界处理
4. 【强制】开发环境下，后端开发人员必须保证API本地测试通过，统一使用API测试工具：Postman
5. 【强制】开发环境下，后端开发人员须保证使用Swagger 生成API文档对接口描述清晰明确