蘑菇租房系统代码开发规范

**版本控制信息**

| 版本 | 日期 | 拟稿和修改 | 说明 |
| --- | --- | --- | --- |
| 1.0 | 2015-5-18 | 史欣欣 |  |
| 2.0 | 2015-06-04 | 田东兴 | 补充了数据库相关规范 |
| 2.1 | 2016-07-07 | 沙周麟 | 对编码规范以及注释规范进行修改，并添加格式模板 |
| 3.0 | 2016-09-19 | 宋伟 | 服务化拆分 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

目录

[蘑菇租房系统代码开发规范 1](#_Toc462832233)

[一 系统包结构及规范 3](#_Toc462832234)

[包结构 3](#_Toc462832235)

[逻辑定义 4](#_Toc462832236)

[二 代码编写规范 12](#_Toc462832237)

[格式规范 12](#_Toc462832238)

[命名规范 13](#_Toc462832239)

[注释规范 14](#_Toc462832240)

[配置管理 15](#_Toc462832241)

[代码view规范 16](#_Toc462832242)

[缓存规范 16](#_Toc462832243)

[分布式锁的使用 17](#_Toc462832244)

[线程的使用规范 18](#_Toc462832245)

[工具类使用规范 18](#_Toc462832246)

[定时任务使用规范 18](#_Toc462832247)

[事务规范 18](#_Toc462832248)

[异常 19](#_Toc462832249)

[日志 23](#_Toc462832250)

[服务接口文档规范 服务接口上下文定义 24](#_Toc462832251)

[三 findBugs code规则 27](#_Toc462832252)

# 一 系统包结构及规范

### 包结构



如上图，

web接口层为上层renter、bs等系统包。

服务编排层为 facade层分为 接口与实现两个项目。

业务服务层为service分为接口与实现两个项目，其中业务服务层分为servcie与dao两层报包接口，servcie为业务实现，dao为数据实现。

### 逻辑定义

#### web服务层：controller



controller定义：web展示逻辑处理或者接口适配处理，处理与协议接口相关的封装与处理。  
 在web系统中controller只作为前后端交换处理的展示逻辑交换，处理http接口的前后端封装，不负责具体的业务代码，不允许在controller中处理业务逻辑。  
 在app后台或者对外接口中controller只做接口的适配处理逻辑，处理http或者其他协议相关的封装，也不做具体的业务处理。  
样例：controller只处理web界面需要的http协议的数据处理，包括接受参数接收，封装，及返回数据封装，并不做业务处理。

common 包为系统公用的配置等非业务包。

**@RestController**

**public class DemoController {**

**@Autowired**

**private IRenterRegistService renterService;**

**@RequestMapping(value = "/root/demo/{id}")**

**@ResponseBody**

**public String root(HttpServletRequest resquest, @PathVariable("id") String id) throws Exception {**

**RenterRegistQueryRequest rq = new RenterRegistQueryRequest();**

**rq.setStartDate("2016-01-01");**

**rq.setEndDate("2016-09-01");**

**Page<RenterChannel> page = new Page<>();**

**page.setCurrentPage(1);**

**page.setCurrentResult(100);**

**List<RenterChannel> list = renterService.findRenterChannelByDate(rq, page);**

**String data = JsonUtil.getInstance().object2JSON(list);**

**return data;**

**}**

**}**

#### 服务编排：Facade



Facade定义：服务流程编排，封装某个领域或者跨领域业务的流程，隔离流程内部实现，提供上层唯一的入口及返回值，保证某个业务处理的原子性，Facade只提供接口，隔离内部实现。

1.Facade之间禁止相互横向调用。

2.Facade在领域内部可以往下层调用领域内部服务接口，如果涉及跨领域调用，尽量减少调用粒度。

3.Facade要保证一个上层业务逻辑的业务事务完整性及隔离特性，在涉及下层多个领域业务接口时候，要保证最终事务

4. Facade的编写必须有详细的业务接口说明，及业务流程文档状态图。

5. Facade接口要遵循幂等性，及可撤销。

6. Facade 保证下层整体调用链可查询。

Facade方法调用参数使用同一的接口泛化类：ServiceContent，返回值采用统一的接口返回类：ResultContent。

#### 服务编排：facadeImpl



facadeImpl定义：服务流程编排，封装某个领域或者跨领域业务的流程，隔离流程内部实现，提供上层唯一的入口及返回值，保证某个业务处理的原子性，facadeImpl实现IFacade接口，隔离内部实现。

1. Facade之间禁止相互横向调用。

2. Facade在领域内部可以往下层调用领域内部服务接口，如果涉及跨领域调用，尽量减少调用粒度。

3. Facade要保证一个上层业务逻辑的业务事务完整性及隔离特性，在涉及下层多个领域业务接口时候，要保证最终事务

4. Facade的编写必须有详细的业务接口说明，及业务流程文档状态图。

5. Facade接口要遵循幂等性，及可撤销。

6. Facade 保证下层整体调用链可查询。

Facade方法调用参数使用同一的接口泛化类：ServiceContent，返回值采用统一的接口返回类：ResultContent。

例子1：房源预定，房源预定facade定义房源的预定流程，封装底层实现，暴露最小粒度接口。

代码：

**public ServiceRespond roomRechange(long userId,long roomId) {**

**logger.info("xxxx纪录房源预定日志");**

**User user = userService.getUserInfo(userId);**

**Room room = roomService.getRoomStatus(roomId);**

**if(room.status=1){**

**room = roomService.rechange(userId,roomId);**

**Bill bill = null;**

**Contract cnt = null;**

**if(cnt.status=1){**

**cnt = contractService.createContrant(user,room);**

**}**

**if(cnt.status=1){**

**bill = BillService.createBill(user, cnt);**

**}  
 if(bill.status=1){**

**billAccount bacct = AccountService.createBillAcct(user, bill);**

**}**

**}**

**xxx 其他返回值处理**

**return xxx ;**

**}**

#### 业务服务：Service



Service定义：业务服务对外的服务接口。

1. 封装领域内部一个完整子业务的业务操作,隔离底层实现。

2. 接口保证领域子业务内部事务完整性及隔离性。

3. 接口幂等、可回滚撤销。

Service方法调用参数使用同一的接口泛化类：ServiceContent，返回值采用统一的接口返回类：ResultContent。

例子: RoomService 接口封装了房间状态查询及房间预订两个业务操作。

**blic interface RoomService{**

**public ServiceRespond getRoomStatus(ServiceContent content);**

**public ServiceRespond reserveRoom(ServiceContent content);**

**}**

#### 业务服务：ServiceImpl



ServiceImpl定义：业务服务对外的服务接口。

1. 实现Service业务操作。

2. 接口保证领域子业务内部事务完整性及隔离性。

3. 接口幂等、可回滚撤销。

4. ServiceImpl 禁止直接注入具体的数据实现例如注入数据库的mapper，队列的kafka类。

例子: RoomServiceImpl 接口封装了房间状态查询及房间预订两个业务操作。

**public class RoomService{**

**public ServiceRespond getRoomStatus(ServiceContent content){**

**method1();**

**method2();**

**xxx**

**};**

**public ServiceRespond reserveRoom(ServiceContent content){**

**method1();**

**method2();**

**xxx**

**}**

**}**

#### 业务服务：Dao

Dao定义：dao为数据访问层，隔离底层数据库、缓存、队列等具体数据交互，提供上层业务提供统一的访问接口。

1. Dao接口提供细粒度的接口，同时需要隔离底层实现。

2.缓存等其他数据接口优化方式，Dao内部要有相应的优化实现。

例子:

**public ServiceRespond getRoomStatusByRoomID(ServiceContent content);**

**public ServiceRespond reserveRoom(ServiceContent content);**

#### 业务服务：DaoImpl

DaoImpl定义：dao为数据访问层，隔离底层数据库、缓存、队列等具体数据交互，实现上层访问接口Dao。

1. Dao接口提供细粒度的接口，同时需要隔离底层实现。

2.缓存等其他数据接口优化方式，Dao内部要有相应的优化实现。

例子:

**@autowired**

**private RoomMapper roomMapper;**

**public ServiceRespond getRoomStatusByRoomID(ServiceContent content){**

**int status = roomMapper.getRoomStatusByID(content.getIntegerParams("roomId"));**

**xxx**

**};**

**public ServiceRespond reserveRoom(ServiceContent content){**

**int roomId = content.getIntegerParams("roomId");**

**int status = roomMapper.getRoomStatusByID(roomId);**

**if(status=1){**

**roomMapper. reserveRoom(roomId);**

**}**

**};**

#### 工具包common



每层代码中如果需要自己添加自己的工具类安装以上包结构存放。

onfig 配置目录

util 基础工具类

enums 枚举类

contant 常量

intercepter 拦截器或者aop类

exception 异常类

entity 接口交互需要的bean 或者实体

# 二 代码编写规范

### 格式规范

* 1. 文件必须采用UTF-8编码
  2. import中不能有通配符
  3. 各import中无需空行分隔
  4. 采用4个空格进行缩进
  5. 设置列宽为128
  6. 常量命名应使用CONSTANT\_CASE格式（译注：所有字母大写，单词使用下划线分隔）。注意，所有常量都必须是static final成员，但并不是所有的static final成员都是常量。
  7. 修饰符的顺序应按照Java语言规范推荐的顺序
  8. 在if、else、for、do、while中，即使代码块中没有内容，或者只有一行代码，也必须使用大括号。
  9. 类、枚举、注解或接口中的代码需进行缩进
  10. 任何方法中的代码需进行缩进如：
  11. If、for、while、switch、case(for switch)中的代码需进行缩进
  12. switch语句中，必须包含default语句
  13. 任何类、注解、接口、枚举的声明，其“{”应与所声明对象在同一行，且应与声明对象之间空格
  14. If、while、for、switch、try、catch需与“(”之间空格，且其中的“{”需与“)”、“{”之间空格
  15. 在“=”之前或之后需用空格分隔
  16. 在多个定义、声明、赋值、if、while、for、方法调用，各代码段间需用空行分隔
  17. If代码段必须使用“{”、“}”，即使只有一行代码，另else、else if需与“（”、“{”之间空格
  18. 四则运算符、逻辑操作符、位操作符、三元操作符之前或之后应有空格分隔
  19. 在类、方法或字段上添加注解时，应一个注解占用一行
  20. 在类声明、枚举定义、枚举使用、变量定义、方法定义、方法调用中，如有“，”，则此之后需用空格分隔
  21. 在定义数组并对其初始化数据时，“，”之后需空格分隔，如：
  22. int[] array1 = newint[]{1, 2, 3};
  23. 在使用泛型的场景下，如有多个泛型，则“，”之后需空格分隔，如：
  24. Map<String, Element> map = new HashMap<String, Element>();
  25. class MyGenericType<S, T extends Element & List> {}
  26. 一个class或method 不要过大，如果class或method行数过多请拆分，单个class限制在千行以内，单个方法控制在百行以内。
  27. 写干净整洁的代码，代码以格式规范第一，可读性第一，在保证业务逻辑正确前提必须保证代码可读性，请合理运用空行。空行可以用来隔开相对独立的代码块，有利于阅读和理解。eclipse里面Ctrl+Shilft+F。
  28. 废弃的代码请删除，过时的及不会再调用的业务代码请删除，不要吝惜废弃的老代码，去除没有用到的类引用 eclipse里面Ctrl+Shilft+O
  29. 把所有的类变量放到最前面，如果比较多请按用途分组排列，不要把变量散落在整个类里面。

以下为格式模板文件：

注：由于IDEA与Eclipse在格式设置项上有所区别，故为使两者保持一致，IDEA需安装Eclipse Code Formatter插件后加载Eclipse配置文件，具体如下

[Eclipse](http://www.baidu.com/link?url=l-t8rtDOfPQ_qQcefjmLWVNjAbDn0akegz0O7p2-yGl9VgrgaubDXXiKreON994E)

1.在File->Settings->plugins中点击“Browse repositories”按钮

2.在新窗口中搜索“Eclipse Code Formatter”插件后点击安装，并重启IDEA

3.在File->Settings->Eclipse Code Formatter中，选中“Use the Eclipse Code Formatter”

4.在“Eclipse Java Formatte config file”中将以上配置文件导入

5.在进行格式化后，如果看到’xxx formatted sucessfully by Eclipse code formatter’则表示格式化成功！



### 命名规范

1. 总体命名规则：
   1. 必须使用驼峰标示,类的名字必须由大写字母开头而单词中的其他字母均为小写
   2. 如果类名称由多个单词组成，则每个单词的首字母均应为大写例如TestPage
   3. 如果类名称中包含单词缩写，则这个缩写词的每个字母均应大写，如：XMLUser
2. 类方法名命名

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **层级** | **类名** | **方法名** | **类型** | **例子** |
| web | xxxController | get\* | 查询 | UserController.getUser(int userId) |
| add\* | 修改 | UserController.addUser(int userId,User u) |
| delete\* | 删除 | UserController.deleteUser(int userId) |
| update\* | 更新 | UserController.updateUser(User u) |
| façade-api | xxxFacade | get\* |  | UserFacade.getUser(int userId) |
| add\* |  | UserFacade.addUser(int userId,User u) |
| delete\* |  | UserFacade.deleteUser(int userId) |
| update\* |  | UserFacade.updateUser(User u) |
| façade | xxxFacadeImpl | get\* |  | UserFacadeImpl.getUser(int userId) |
| add\* |  | UserFacadeImpl.addUser(int userId,User u) |
| delete\* |  | UserFacadeImpl.deleteUser(int userId) |
| update\* |  | UserFacadeImpl.updateUser(User u) |
| service-api | xxxServcie | get\* |  | UserServcie.getUser(int userId) |
| add\* |  | UserServcie.addUser(int userId,User u) |
| delete\* |  | UserServcie.deleteUser(int userId) |
| update\* |  | UserServcie.updateUser(User u) |
| service | xxxServcieImpl | get\* |  | UserServcieImpl.getUser(int userId) |
| add\* |  | UserServcieImpl.addUser(int userId,User u) |
| delete\* |  | UserServcieImpl.deleteUser(int userId) |
| update\* |  | UserServcieImpl.updateUser(User u) |
| dao | xxxDao | select\* |  | UserDao.selectUser(); |
| delete\* |  | UserDao.deleteUser(); |
| update\* |  | UserDao.updateUser(); |
| insert\* |  | UserDao.insertUser(); |
| daoImpl | xxxDaoImpl | select\* |  | UserDaoImpl.selectUser(); |
| delete\* |  | UserDaoImpl.deleteUser(); |
| update\* |  | UserDaoImpl.updateUser(); |
| insert\* |  | UserDaoImpl.insertUser(); |
|  |  |  |  |  |

4. Domain 实体类

* + - 1. 命名单词名称除特殊情况需与数据库表后缀名一样，如：数据库中存在表cntr\_salepreferential，则实体类命名为SalePreferential
      2. 实体类需实现Serializable接口，防止以后会使用序列化,实体类必须实现equals()、toString()、hashCode()方法。
      3. 实体类字段命名，方法的名字的第一个单词应以小写字母作为开头，后面的单词则用大写字母开头，例如：userName
      4. 每个字段属性后必须写明注释，以标注此字段作用，同数据库表字段标注相同
      5. 凡是存在String型的字段属性，请尽量在set方法中加明去前后空格代码。
      6. 凡是与数据库表字段无直接对应关系的属性，需标注为(逻辑字段)。

### 注释规范

1. class注释：

1. @brief 概要描述
2. @details详细描述
3. @date 类创建时间
4. @author 类创建作者(git对应账户名)

2. 方法的具体注释：

1. @brief 概要描述
2. @details 详细描述
3. @param 参数(如有多个则填写多个)
4. @retval 返回值
5. @exception 异常信息
6. @see 相关联类
7. @author 作者名称
8. @date 日期
9. @note 变更信息

3. sql mapper 注释

a) @brief 概要描述

b) @details详细描述

c) @date 类创建时间

d) @author 创建作者

通过在mapper文件中添加<!-- --> 禁止添加在sql 注释中。

例子：

<!--

a) @brief 概要描述

b) @details详细描述

c) @date 类创建时间

d) @author 创建作者

-->

<select id=*"selectLandlordByMap"* parameterType=*"map"* resultMap=*"BaseResultMap"*>

select name,id from user\_user;   
 </select>

### 配置管理

* 配置文件存储目录：
  + 配置文件格式支持
    - xml
    - properties
  + 支持动态加载机制［上面1，2项配置支持动态加载］

相关配置文件：从框架级别进行解耦

* 配置定义

properties配置文件

jdbc.properties 数据库相关配置

log4j.properties 日志配置

systemconfig.properties 业务配置

threadpool.properties线程池配置

context spring等配置文件

mybatis.xml mybatis数据库配置

spring-beans.xml spring依赖注入相关配置

spring-cache.xml缓存相关

spring-intercepter.xml 拦截器

spring-kafkaProducer.xml kafka配置

spring-quartz.xml 定时任务配置

spring-servlet.xml spring web对应配置

META-INF cat等配置

app.properties cat监控配置

* 格式规范

当前业务配置：

1. 在配置文件--resources/conf/ systemconfig.properties中添加需要的配置。

格式key=${filter.key}。

2. filters配置

如果配的的变量需要根据环境打包成不同的配置，则需要配置filters， sysconfig-xxx.properties

格式filter.key = value

配置demo

resources/conf/ systemconfig.properties  
 内容：sysconfig.sendSMS = ${filter.sysconfig.sendSMS}

filters/sysconfig-dev.properties

内容：filter.sysconfig.sendSMS=123

b. 统一后业务配置：

续配置统一采用配置中心，实现配置与代码分离，为配合今后服务化的实施，其配置信息需统一配置、管理，可进行自动配置，以保障服务可快速发布、部署。

为此本文重点讲解在开发、测试、生产环境中，其配置信息的配置、管理方式。

其主要使用Disconf进行配置信息的统一管理，开发与测试公用同一套配置环境，生产单独使用一套配置环境。

具体查看附表



### 代码view规范

1. 业务开发代码，每个组技术经理负责view本组内代码，每个业务必须经过人工view才能上线，如果后期上线发现严重的可以view发现的使用不当问题，要追究技术经理责任。

2. 为了规范基础语法使用，会使用gerrit 定期扫描代码，发现不合格的代码要限时修改掉。

3. 架构组每隔一个月发布非业务代码缺陷明细，统计出来发给每隔组。

### 缓存规范

缓存使用方式

统一的代码控制方式，缓存统一工具类com.mogoroom.service.base.impl. CacheServiceImpl

缓存归类与使用原则

1. 缓存过期时间尽量设置较短。

2. 缓存的只作为辅助使用，进制把缓存当做存储，如果因为缓存失效导致业务无法使用，业务自己负责。

3. 缓存序列化统一使用json序列化。

api接口：

|  |  |  |
| --- | --- | --- |
| **方法** | **类型** | **备注** |
| public boolean setRedisKey(String channel, String type, String key, String value, int exp) | Redis | 添加缓存 |
| public String getRedisKey(String channel, String type, String key) | Redis | 获取缓存 |
| public boolean setKey(String channel, String type, String key, Object value, int exp) | memcache | 添加缓存， |
| public boolean setKey(String channel, String type, String key, Object value, int exp, long timeout) | memcache | 添加缓存， |
| public Object getKey(String channel, String type, String key) | memcache | 获取缓存 |
| public Object getKey(String channel, String type, String key, long timeout) | memcache | 获取缓存 |

### 分布式锁的使用

分布式锁，为重量级锁，只有在单jvm无法控制并发的时候才可以使用，禁止在单jvm内部使用。

使用方式主要分为两类：

手工方式：

调用的类：

**public class DistributedLock{**

**public synchronized static void init(String host, String password)**

**public static boolean acquire(String key)**

**public static boolean acquire(String key, long timeout)**

**public static void release(String key)**

**}**

锁key规则：

Strign key = channel + "\_" + busitype + "\_" + id;

例子：

项目启动时候初始化：**DistributedLock.init("192.168.30.107:2181","");**

使用锁：

**String key = "/lock";//需要加锁的资源**

**if(DistributedLock.acquire(key, 2)){**

**xxx 业务操作**

**DistributedLock.release(key);**

**}**

注解方式：

在需要的方法上添加注解：

例如：

**@DistributedLockAnnotation(key="user-add")**

**public String add( @LockKey String value) throws Exception{**

**xxx**

**}**

**其中最终key=user-key 比如 key=user-add-1**

### 线程的使用规范

如果业务需要异步开启多线程处理业务，统一使用异步线程池方式处理，禁止采用阻塞模式的new task处理，也就是禁止自己开一个线程单独处理。

使用方式：

**TaskExecutor taskExecutor = (TaskExecutor) MogoSpringContextUtil.getBean("taskExecutor");**

**taskExecutor.execute(new LogTask(operator, operatorType, operationSource, operationTarget, operationType, map));**

线程池的配置在：threadpool.properties中配置

### 工具类使用规范

### 定时任务使用规范

目前配置

定时任务项目 ：mogoroom-tasktracker

### 事务规范

统一事物添加方式：

在方法上添加注解：

**@Transactional(rollbackFor=Exception.class)一定要加rollbackFor=Exception.class**

禁止及错误的用法：

1. 不要对类（的所有方法）添加事务。

例如：

**@Transactional(propagation = Propagation.REQUIRES\_NEW, readOnly = false, rollbackFor = Exception.class) BusinessRecordFacadeImpl**

2. 不要总是去新启动一个事务（除非你真的需要）  
尽量使用 Propagation.REQUIRED 默认代替 Propagation.REQUIRES\_NEW。如果你（真的）需要在自己的事务上下文中执行业务请使用 Propagation.NESTED

3. 移除这种明显错误配置事务的方式

**@Transactional(rollbackFor=Exception.class,propagation=Propagation.REQUIRED,timeout=2) public class FlatsFacadeImpl**

不要指定超时时间，你控制不了！

4. 任何时候不要去改隔离级别

### 异常

整体开发过程

1. 开发人员按照规范、根据业务需要定义错误码。
2. 工具每天从代码里收集最新的错误码一览、生成文档。
3. 审核人员审核错误码、特别是审核最终面向用户的错误消息是否规范。
4. 开发人员在发生异常的地方抛出MogoException或者其子类、需传入相应的ErrorCode。
5. 产品经理或者项目经理可根据业务需要定义面向最终用户的错误message、这些message将由开发人员定义在资源文件中。

窗体顶端

具体规范方案

**错误码格式**

错误码信息统一由ErrorCode类定义、类中各个字段定义如下：

| **字段英文** | **字段名** | **格式** | **例子** | **备注** |
| --- | --- | --- | --- | --- |
| **code** | 错误码 | 模块编码+四位整数 | C0001 | 模块编码参考下面 |
| **msg** | 内部错误消息 | String | “参数非法。{paramName=%s, value=%s}” | 内部错误消息模板、抛出异常时根据本模板定义具体的错误消息。 |
| **userMsg** | 用户错误消息 | String | “参数非法。{paramName=%s, value=%s}” | 用户错误消息模板、抛出异常时根据本模板定义具体的错误消息。若不设置、则和msg一致。错误消息可以定义在资源文件、默认文件名为errorcodes.properties。 |
| **type** | 错误类型 | 枚举 | COMMON | 枚举定义在ErrorType类 |

错误消息资源文件

可将面向用户的错误消息定义在资源文件中、默认在资源文件为classpath下的errorcodes.properties。 可通过系统变量指定资源文件名、系统变量KEY为sysconfig.errorcode.filepath。

例子：指定资源文件: “System.setProperty(”“sysconfig.errorcode.filepath”“, ”“classpath:errorcodes.properties”“);”

资源文件格式 资源文件key为ErrorCode中定义的错误码、value为面向用户的错误消息模板。

例子：

**errorcodes.properties**

**C0001=参数必须输入**

**C0002=参数%s有误**

**错误类型**

错误类型定义在ErrorType类中、根据业务需要定义下面几种

1）COMMON

一般错误类型。该类型的异常会导致DB事务回滚、并由统一异常拦截器处理。

一般会向client端返回错误页面或者错误消息的JSON串（若客户端请求为AJAX请求）。

2）NOROLLBACK

事务不回滚类型。该类型的异常不会导致DB事务回滚（需要配置事务管理器）、由统一异常拦截器处理。

一般会向client端返回错误页面或者错误消息的JSON串（若客户端请求为AJAX请求）。

抛出本类型的异常意味着之前的业务操作有效、但是后续的业务处理将终止。

3）NEW\_FEATURE\_NOT\_READY

新功能未就绪类型。该类型的异常需要在业务侧捕捉、捕捉到之后按照旧功能的业务逻辑处理、

捕捉异常之后不会导致DB事务回滚(若不捕捉则会回滚)。

本异常一般不由统一异常拦截器处理。

**错误码常量收集类**

1. 普通列表项目共通的错误码作为常量统一定义在GeneralErrorCodes类中
2. ErrorCode实例只能通过GeneralErrorCodes的protected的工厂方法实例化。
3. 各个模块的私有的错误码定义在自身模块的常量收集类中。
4. 命名格式： 模块名+ErrorCodes
5. 要求： 需要继承GeneralErrorCodes类、并定义为抽象类。理由参见

例子：

**public abstract class AcctErrorCodes extends GeneralErrorCodes{...}**

**App层(Controller层）由于不应该包含复杂业务逻辑、该层的错误码定义在共通错误码常量类GeneralErrorCodes里**

异常**类MogoException**

1. 代码里所有需要抛出异常的地方必须抛出MogoException及其子类。
2. MogoException实例化时需要传入某个ErrorCode实例、并根据ErrorCode的消息模板定义具体的错误消息。

例子：

**"throw new MogoException(GeneralErrorCodes.PARAM\_INVALID, ""roomId"", -1);"**

**"GeneralErrorCodes.PARAM\_INVALID的消息模板为：参数非法。{paramName=%s, value=%s}"**

**- 可定义MogoException的子类、但是若没必要不推荐这样。**

**例如， 定义一个不会导致DB事务回滚的异常：**

**public class NoRollBackMogoException extends MogoException{...}**

* **各个模块在错误码中的模块编码**

| **模块名** | **模块编码** | **错误码例子** | **错误码常量收集类** |
| --- | --- | --- | --- |
| **mogoroom-util** | C | C0001 | GeneralErrorCodes |
| **mogoroom-facade** | F | F0001 | FacadeErrorCodes |
| **service-acct** | S01 | S010001 | AcctErrorCodes |
| **service-base** | S02 | S020001 | BaseErrorCodes |
| **service-bill** | S03 | S030001 | BillErrorCodes |
| **service-city** | S04 | S040001 | CityErrorCodes |
| **service-cntr** | S05 | S050001 | CntrErrorCodes |
| **service-comm** | S06 | S060001 | CommErrorCodes |
| **service-coms** | S07 | S070001 | ComsErrorCodes |
| **service-domain** | S08 | S080001 | DomainErrorCodes |
| **service-flat** | S09 | S090001 | FlatErrorCodes |
| **service-fund** | S10 | S100001 | FundErrorCodes |
| **service-host** | S11 | S110001 | HostErrorCodes |
| **service-kafka** | S12 | S120001 | KafkaErrorCodes |
| **service-loan** | S13 | S130001 | LoanErrorCodes |
| **service-lock** | S14 | S140001 | LockErrorCodes |
| **service-mesg** | S15 | S150001 | MesgErrorCodes |
| **service-oder** | S16 | S160001 | OderErrorCodes |
| **service-opex** | S17 | S170001 | OpexErrorCodes |
| **service-orga** | S18 | S180001 | OrgaErrorCodes |
| **service-othr** | S19 | S190001 | OthrErrorCodes |
| **service-perm** | S20 | S200001 | PermErrorCodes |
| **service-repo** | S21 | S210001 | RepoErrorCodes |
| **service-supp** | S22 | S220001 | SuppErrorCodes |
| **service-user** | S23 | S230001 | UserErrorCodes |

* **校验工具类ValidateUtil**

目前系统提供了一个通用的输入校验类ValidateUtil、该类定义在service-util模块中。可使用该类完成一些基本的校验功能。

·检查参数是否为空

·检查参数是否匹配正则表达式

·检查参数是否匹配一些常用模式、例如是否为URL、是否全是数字...

为了使用上的方便、ValidateUtil校验类设计为：若校验不通过则抛出异常。

抛出的异常均为MogoException类及其子类、可通过成员方法getErrorCode获取详细错误码。

### 日志

异常日志统一引入类,不允许引入第三方包。

**import com.mogoroom.core.util.log.Logger;**

**import com.mogoroom.core.util.log.LoggerFactory;**

**private static final Logger logger = LoggerFactory.getLogger(RenterPortraintServiceImpl.class);**

其他规则以日志规范为准



### 服务接口文档规范 服务接口上下文定义

传参：

**/\*\***

**\* 远程服务调用上下文**

**\***

**\* @author songwie**

**\***

**\*/**

**public class ServiceContext implements Serializable, Cloneable {**

**private static final long serialVersionUID = 766707733493948524L;**

**/\*\***

**\* 业务顺序，选传**

**\*/**

**protected String order;**

**/\*\***

**\* 用户ID，选传**

**\*/**

**protected Integer userId;**

**/\*\***

**\* 字母账号ID，选传**

**\*/**

**protected Integer userInfoId;**

**/\*\***

**\* 用户类型，选传**

**\*/**

**protected Integer userType;**

**/\*\***

**\* 渠道，必传**

**\*/**

**protected Integer channel;**

**/\*\***

**\* 业务模块，必传**

**\*/**

**protected String module;**

**/\*\***

**\* 业务类型，必传**

**\*/**

**protected Integer bizType;**

**/\*\***

**\* 服务版本号，选传**

**\*/**

**protected String version;**

**/\*\***

**\* 核心业务模块，选传**

**\*/**

**protected String coreModule;**

**/\*\***

**\* 核心业务类型，选传**

**\*/**

**protected Integer coreBizType;**

**/\*\***

**\* 核心服务版本号，选传**

**\*/**

**protected String coreVersion;**

**/\*\***

**\* 排序，选传**

**\*/**

**protected String orderBy;**

**/\*\***

**\* 分页开始页数**

**\*/**

**protected int start = -1;**

**/\*\***

**\* 分页偏移量，每页条数**

**\*/**

**protected int limit = -1;**

**/\*\***

**\* 全局性业务调用链事务标识，必传**

**\*/**

**protected String transactionId;**

**/\*\***

**\* 是否开启事务**

**\*/**

**protected boolean startTransaction = false;**

**/\*\***

**\* 参数属性**

**\*/**

**protected Map<String, Object> params = new HashMap<String, Object>();**

**protected String[] securityParam;**

**}**

返回值：

**/\*\***

**\* 远程服务返回结果**

**\***

**\* @author songwie**

**\***

**\*/**

**public class ResultContext implements Serializable, Cloneable {**

**private static final long serialVersionUID = 766707733493948524L;**

**/\*\***

**\* 返回数据 单条记录**

**\*/**

**protected Map<String, Object> data = new LinkedHashMap<>();**

**/\*\***

**\* 返回数据多条记录**

**\*/**

**protected List<Map<String, Object>> rows = new ArrayList<>();**

**/\*\***

**\* 返回数据json数据**

**\*/**

**protected String jsonData;**

**/\*\***

**\* 返回服务调用上下文**

**\*/**

**protected ServiceContext serviceContext;**

**/\*\***

**\* 服务返回消息**

**\*/**

**protected String msg;**

**/\*\***

**\* 服务返回code**

**\*/**

**protected String code;**

**/\*\***

**\* 服务状态**

**\*/**

**protected boolean success = true;**

**protected int showCount = -1; // 每页显示记录数**

**protected int totalPage; // 总页数**

**protected int totalResult; // 总记录数**

**protected int currentPage; // 当前页, 从1开始**

**}**

# 三 findBugs code规则

