蘑菇租房Java代码开发手册(5.3.1)

2018-07-30

**开发手册学习引导**

本文是蘑菇租房本身的技术栈和开发实践的总结。在成文过程中，也广泛参考了了网络和各大厂的经验教训，特别是代码风格/编写规则部分参考了《*阿里巴巴Java开发手册*》。在前述基础上，我们对于更多的细节进行了补充。15SONAR代码规范解读也详细对本规范的内容进行了解读。同样地，我们“根据约束力强弱及故障敏感性，规约依次分为强制、推荐、参考三大类。对于规约条目的延伸信息中，‘说明’对内容做了适当扩展和解释;‘正例’ 倡什么样的编码和实现方式;‘反例’说明需要防的雷区，以及真实的错误案例。”

请开发同学将其作为一本学习开发技术的指导，抱着学习的态度去看这本手册，这也是将原来的《规范》修改为《手册》的原因。作为手册来讲更能反应出本文是要开发同学用来学习的，因为这些编程的原则，就像是设计模式、TDD（测试驱动开发）、DDD（领域驱动设计）等技术一样，经过了血于火的洗礼。本手册中的内容，需要各位开发同学在学习的过程中，将每一点都学透，而非机械的去搬，一定要知道每一条规定背后的技术点，完全掌握知识，这对于每个人来讲都大有裨益，如果完全吃透了这本手册，相信你的技术水平会更上一层楼，在此，架构部真诚的希望每位开发同学精读本手册。

技术会不断进步，本手册亦会不断更新。

请各位开发同学参照执行。

架构组首页：[http://](http://sdk.mgzf.tech)jiagou.mogo.com

**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 |
| 3.1 | 2016-10-18 | 冯银鹏 | 第二章内容调整，新增编码规范及附件，并将注释规范合并入第二章，文档排版微调 |
| 3.2 | 2016-10-26 | 冯银鹏 | 文档排版调整 |
| 4.0 | 2016-12-15 | 宋伟 | 新增日志与分布式锁规范 |
| 4.1 | 2016-12-17 | 宋伟 | 修改日志与分布式锁规范 |
| 4.2 | 2016-12-19 | 宋伟 | 修改日志规范 |
| 4.3 | 2017-01-13 | 宋伟 | 修改日志规范添加错误编码与[RESTful](http://www.baidu.com/link?url=tku2_MZtWc7PQoZW5rpgUCKarjaYVsvlXfmYTykuCyspZ-LIgmXyXaN6j0s8WUqiQRlFkYB8fpulWw5scJHgJq" \t "_blank)规范 |
| 4.4 | 2017-03-08 | 宋伟 | 添加表单重复提交，数据库版本校验规范 |
| 4.5 | 2017-04-08 | 宋伟 | 添加代码view统计的部分限制用法 |
| 4.7 | 2017-06-01 | 宋伟 | 添加配置文件规范 |
| 4.8 | 2017-06-08 | 宋伟 | 细化服务化分层相关规范 |
| 4.9 | 2017-09-02 | 刘晓辉 | 修改各层命名规范，facade层的param，vo合并为DTO |
| 5.0 | 2017/10/9 | 刘晓辉 | 合入《阿里巴巴Java开发手册》 |
| 5.1 | 2017-12-28 | 沈铖 | 全文梳理 |
| 5.2 | 2018-1-3 | 了结/凿空 | 整合阿里代码规范hook标志等 |
| 5.3 | 2018-4-23 | 朱拱平/沈铖 | 新增1.3系统解耦及降级规则  2.7并发处理节新增16-19点  3章配置管理  5.1节日志概述  新增4.6-4.8等事务规范  11章异常处理  12章单元测试新增17点及“单元测试管理方式”  新增17章Code Review流程及规范 |
| 5.3.1 | 2018-7-30 | 朱拱平/沈铖 | 明确一些规范为【强制】级别 |

目录

[蘑菇租房Java代码开发手册(5.3.1) 1](#_Toc830335548)

[目录 4](#_Toc1823820260)

[1 系统包结构及命名规范 7](#_Toc2031377370)

[1.1 总体代码原则 7](#_Toc1804053721)

[1.2 应用系统分层结构 8](#_Toc949357052)

[1.2.1 包结构说明 9](#_Toc152902810)

[1.2.2 controller层 11](#_Toc1599462307)

[1.2.3 facade层 11](#_Toc605572032)

[1.2.4 业务服务层Service 13](#_Toc1684240013)

[1.2.5 Dao数据层 13](#_Toc487179267)

[1.3 系统解耦及降级规则 14](#_Toc1247218736)

[2 代码编写规范 14](#_Toc2023759294)

[2.1 命名风格 14](#_Toc544106248)

[2.2 分层相关命名规范 17](#_Toc1643025802)

[2.2.1 各层类、方法名命名 17](#_Toc81160590)

[2.3 常量与枚举定义 18](#_Toc1219869542)

[2.4 代码格式 19](#_Toc1441095421)

[2.5 OOP规约 21](#_Toc2015055633)

[2.6 集合处理 24](#_Toc1722265861)

[2.7 并发处理 27](#_Toc1844479219)

[2.8 控制语句 30](#_Toc1462992411)

[2.9 注释规范 32](#_Toc834139414)

[2.9.1 通用规范 32](#_Toc364930406)

[2.9.2 class注释 33](#_Toc846862535)

[2.9.3 方法的具体注释 33](#_Toc640230021)

[2.9.4 sql mapper 注释 34](#_Toc199173890)

[2.9.5 git提交注释 34](#_Toc2094791565)

[2.9.6 Mapper.xml中自增主键的获取 35](#_Toc57547438)

[2.10 二方库依赖 35](#_Toc1301936503)

[2.11 其它 36](#_Toc640181957)

[3 配置管理 38](#_Toc1196123153)

[4 事务规范 39](#_Toc2132272051)

[4.1 【强制】不要对类（的所有方法）添加事务。 39](#_Toc316518569)

[4.2 【推荐】不要总是去新启动一个事务（除非你真的需要） 39](#_Toc1080016875)

[4.3 【强制】不得使用事务超时时间 39](#_Toc1788842125)

[4.4 【强制】任何时候不要去改隔离级别 39](#_Toc1265875622)

[4.5 【强制】只读接口不允许使用事务 40](#_Toc1232919685)

[4.6 【强制】严禁在事务中访问第三方 40](#_Toc1240820784)

[4.7 【推荐】事务下沉 40](#_Toc1871447654)

[4.8 【推荐】事务粒度最小化 40](#_Toc769676050)

[5 日志规范(1.2) 41](#_Toc1728000051)

[5.1 背景、概述及目标 41](#_Toc971182742)

[5.2 基本原则 41](#_Toc645951696)

[5.3 规范方案 42](#_Toc124622652)

[5.3.1 定义日志常量 42](#_Toc466724897)

[5.3.2 日志分级 42](#_Toc727112286)

[5.3.3 日志类型 44](#_Toc1344492194)

[5.3.4 统一日志输出格式及分离错误日志 44](#_Toc1907820318)

[5.3.5 日志编码规范 45](#_Toc594684271)

[5.4 业务日志 46](#_Toc919274407)

[5.4.1 关键词定义 46](#_Toc1604815889)

[5.4.2 业务日志格式 46](#_Toc2057676682)

[5.4.3 示例代码 47](#_Toc1753413821)

[5.5 监控日志 47](#_Toc1969746296)

[5.5.1 监控指标 48](#_Toc757055569)

[5.5.2 监控日志格式 48](#_Toc246160194)

[5.5.3 监控日志实现说明 48](#_Toc21436538)

[5.6 访问日志 49](#_Toc704363486)

[5.6.1 访问日志格式 49](#_Toc303707632)

[5.7 日志收集 50](#_Toc1323373041)

[5.8 日志规约 50](#_Toc1344545444)

[6 分页使用规范 52](#_Toc1499830786)

[6.1 分页查询SQL改写原则为 52](#_Toc1308161444)

[6.2 分页插件场景 52](#_Toc1661064013)

[6.3 不推荐使用分页插件的地方 52](#_Toc432364013)

[6.4 不推荐显示分页总数的场景 52](#_Toc949519921)

[6.5 分页插件使用 53](#_Toc779455987)

[7 业务编码规范 55](#_Toc1665283699)

[8 分布式锁的使用 56](#_Toc42857057)

[9 数据重复提交推荐方案 57](#_Toc503419994)

[10 读写分离 59](#_Toc287476101)

[11 异常规范 60](#_Toc1770857109)

[11.1 异常处理 60](#_Toc1474602736)

[11.2 异常类说明 62](#_Toc933427798)

[11.3 错误编码规则 62](#_Toc1895479761)

[12 单元测试 63](#_Toc1941327633)

[13 安全规约 65](#_Toc1660540084)

[14 服务器 66](#_Toc1092488307)

[15 SONAR代码规范解读 67](#_Toc1701664303)

[15.1 流资源打开后必须关闭。防止内存泄露 67](#_Toc107740708)

[15.2 枚举属性值必须是私有的，且不可变更的 68](#_Toc2011762714)

[15.3 死循环，不允许出现 68](#_Toc1158996545)

[15.4 调用replaceAll方法 参数需要注意 69](#_Toc17933742)

[15.5 线程遇到异常，必须抛出异常或者终止线程 70](#_Toc1617692887)

[15.6 不能使用MD5 和 SHA-1 ,因为已经变的不安全了 71](#_Toc981259193)

[15.7 代码里面创建的对象，都应该被使用 71](#_Toc774989312)

[15.8 序列化接口的子类和非序列化父类 71](#_Toc1863853082)

[15.9 Session中的数据必须验证 72](#_Toc1002695731)

[15.10 不应该存在无意义的逻辑表达式 72](#_Toc1479352798)

[15.11 布尔判断时使用短路逻辑 73](#_Toc20077066)

[15.12 数组无意义的包含判断 73](#_Toc178585124)

[15.13 单精度浮点型运算规则 74](#_Toc676414594)

[15.14 List indexOf 方法注意要点 74](#_Toc1519907852)

[15.15 Readline方法的正确使用方式 75](#_Toc1486746568)

[15.16 读流在读文件时，需要增加判断 76](#_Toc189994960)

[15.17 "Calendars" and "DateFormats” 不能是静态的 76](#_Toc1952271866)

[15.18 Junit 断言不应该出现在多线程的方法中 77](#_Toc288782842)

[15.19 重写的Equals方法必须包含一定的判断 77](#_Toc969450947)

[15.20 Cookies 写入时 安全设置必须为true 77](#_Toc1470071917)

[15.21 循环的判断条件至少要运行一次 78](#_Toc331639899)

[15.22 线程加入同步锁时，使用wait代替sleep 78](#_Toc1472870941)

[15.23 对象写入文件时必须实现序列化接口 78](#_Toc1757548018)

[15.24 时间的参数范围 79](#_Toc2102497008)

[15.25 方法："BigDecimal(double)" 不应该这样调用。 80](#_Toc799990030)

[15.26 StringFromat的使用规范 80](#_Toc543492168)

[15.27 "static final" 修饰的数组应该是 "private" 81](#_Toc1850493121)

[15.28 实现"Cloneable" 应该重写clone方法 82](#_Toc593834015)

[15.29 后加加 前加加的使用规范 83](#_Toc56548605)

[15.30 属性必须加修饰符 83](#_Toc795497780)

[15.31 If判断规则 84](#_Toc148014671)

[15.32 错误的逻辑判断，不应该出现 84](#_Toc164289313)

[15.33 代码中打印到控制台的异常信息应该使用log代替 85](#_Toc659776846)

[15.34 对象判断NULL条件的规则 85](#_Toc1307011216)

[15.35 IF/IF ELSE 条件判断不应该出现重复判断 86](#_Toc182223055)

[15.36 Finally 模块不能抛出异常 86](#_Toc129986086)

[15.37 赋值时，不应该自己给自己赋值 87](#_Toc140786761)

[15.38 Public static 后面应该加入final 87](#_Toc957212367)

[15.39 Finally模块中不能出现return 87](#_Toc1993839168)

[15.40 compareTo 方法 和equals方法 必须一起重写 88](#_Toc1143482492)

[15.41 使用StringBuffer代替加号 89](#_Toc289081518)

[15.42 无限循环递归将加入代码检测 89](#_Toc2013916234)

[15.43 父类强转子类加入代码检测 89](#_Toc1322067616)

[15.44 空引用加入代码检测 89](#_Toc965496112)

[15.45 奇数的正确判断方式 90](#_Toc1386340439)

[15.46 整型计算转DOUBLE 90](#_Toc661330536)

[15.47 equals()在调用前检查是否为NULL 90](#_Toc1155491072)

[15.48 集合清空方式 90](#_Toc1191128657)

[15.49 重写Eqlues方法 必须同时重写hashCode方法 90](#_Toc950113378)

[15.50 serialVersionUID 必须是final的 90](#_Toc2124942020)

[15.51 String的substring方法 参数传0的时候 会返回原值 90](#_Toc513716926)

[15.52 securerandom 代替Random方法 91](#_Toc1281753278)

[15.53 String判断使用equals 91](#_Toc1450329313)

[16 Restful前后端分离开发规范（1.2） 92](#_Toc123781296)

[16.1 背景、概述及目标 93](#_Toc1236766638)

[16.2 基本原则 93](#_Toc102835695)

[16.3 规范方案 93](#_Toc667273465)

[16.3.1 总体架构 93](#_Toc939776112)

[16.3.2 开发流程 94](#_Toc696669711)

[16.3.3 开发规范 94](#_Toc723822070)

[16.3.4 资源地址规范 94](#_Toc1735273892)

[16.3.5 响应JSON数据格式 94](#_Toc844684382)

[16.4 工具选型 95](#_Toc888111383)

[16.4.1 Swagger API定义规范 96](#_Toc247567091)

[16.4.2 Mock测试 97](#_Toc4211950)

[16.5 附录A-DEMO 97](#_Toc1070334438)

[17 Code Review流程及规范 99](#_Toc377553177)

# 系统包结构及命名规范

## 总体代码原则

1. 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。
2. 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。在代码中，要加入必要的注释。最上层的代码要说明此段代码的业务逻辑，描述清楚操作步骤，修改代码时，如果业务逻辑发生变化，也要同步修改注释。
3. 业务代码尽量遵循开闭原则、里氏代换原则、依赖倒转原则、接口隔离原则、单一职责原则等相关规范。
4. 不要重复造轮子，能够调用通用代码库实现的类请调用通用代码库实现。
5. 不要使用规格不同的轮子。对于第三方包，应统一提供方和版本。比如JSON处理，推荐使用gson。
6. 换行符统一为linux回车。需要配置：

*git config --global core.autocrlf false*

*git config --global core.safecrlf true*

*git config --global core.eol lf*

## 应用系统分层结构

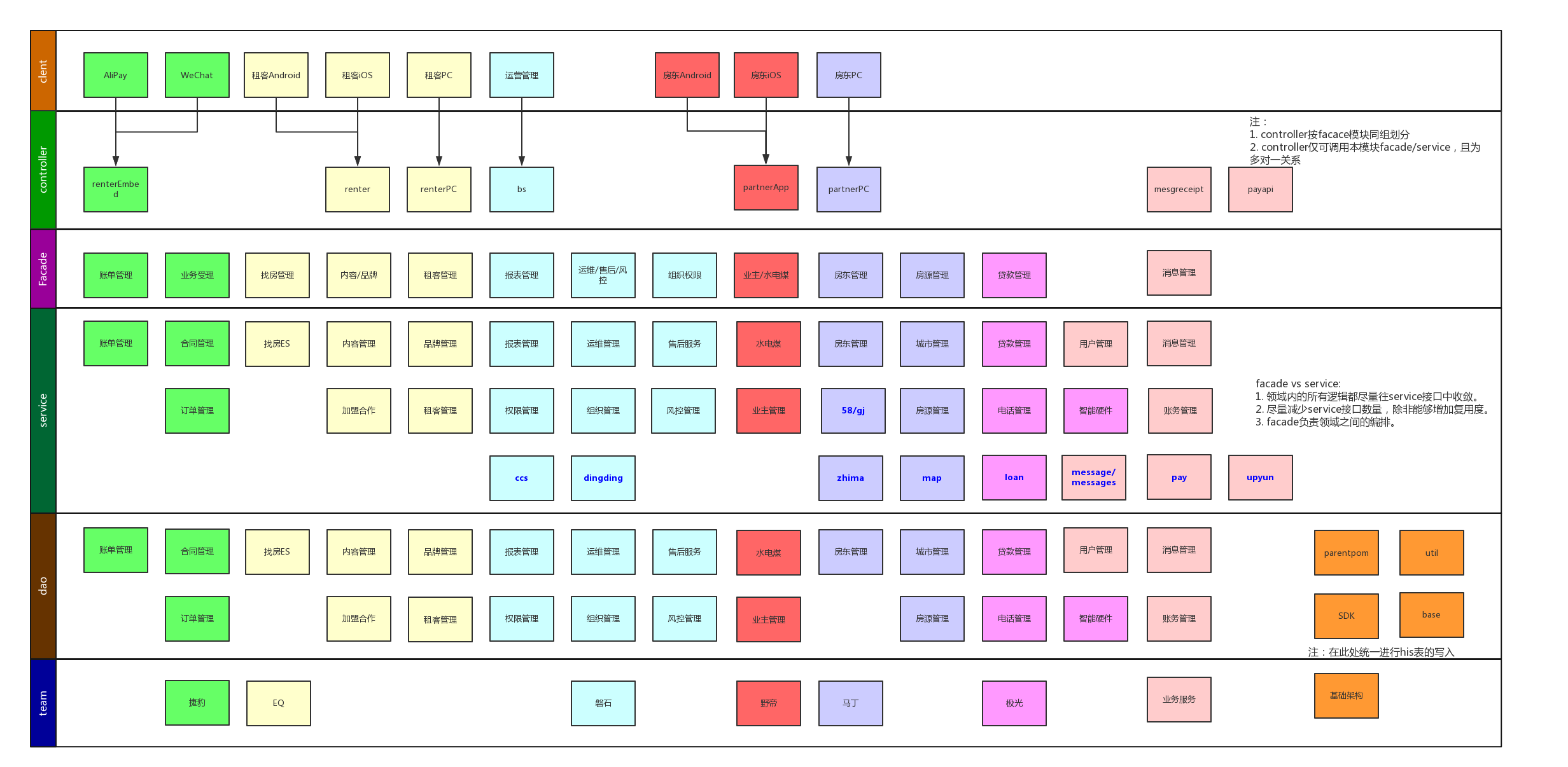


图 1‑1应用分层结构图

一共分四层：controller接口层、facade业务逻辑编排层、servcie业务服务层、dao数据存取层。

1. controller接口层

作为前后端分离的后台api接口层，或者没有前后端分离的web控制层，主要为web发布的对外提供接口，controller作为顶层项目发布部署在web容器tomcat中。controller按facace模块同组划分。controller仅可调用本模块facade/service，且为多对一关系。

1. facade业务逻辑编排层

封装多个业务调用多个业务服务模块，封装业务内部逻辑提供上层controller调用，如果controller业务不跨模块调用，可以不编写facade业务层，直接controller调用servcie，facade为单独的项目以jar或者服务发布的方式提供给controller调用。

facade及service按照模块化的领域边界，严格遵守业务边界。

facade作为跨业务服务集成最大力度的封装业务实现提供给controller调用，facade之间禁止横向调用。

1. service业务逻辑层

**最大粒度**的封装业务逻辑提供上层调用，领域内的所有逻辑都尽量往service接口中收敛。对外隔离内部业务实现。

禁止service跨模块调用。

尽量减少service接口数量，除非能够增加复用度。

一切从服务层出发，从系统需要提供的功能进行分析，确定Service接口中的方法。而不是从数据库的表出发，创建DAO，再创Domain，然后Service，这实际上是对系统分层的误解。

一个领域活动会聚合对应一个或一组DAO，来完成一个领域活动。而一个服务可能需要操作一组领域活动，而每个活动需要操作很多表（调用多个DAO）。 事务的控制可以放到Service层。

1. DAO数据存取层

隔离底层数据实现，提供给serviceCRUD接口调用，dao内部集成DB、cache、ES、MQ等逻辑实现，dao层决定发布的接口是否添加cache、查询ES等，service及dao封装在同一个业务模块项目中，以jar或者服务发布的方式提供给facade或service调用。

备注：特殊场景的业务级别的cache可以添加在facade及service，但是最好的实现是最小粒度的封装在dao层。

### 包结构说明

对于每个应用都分成如下图所示的结构：

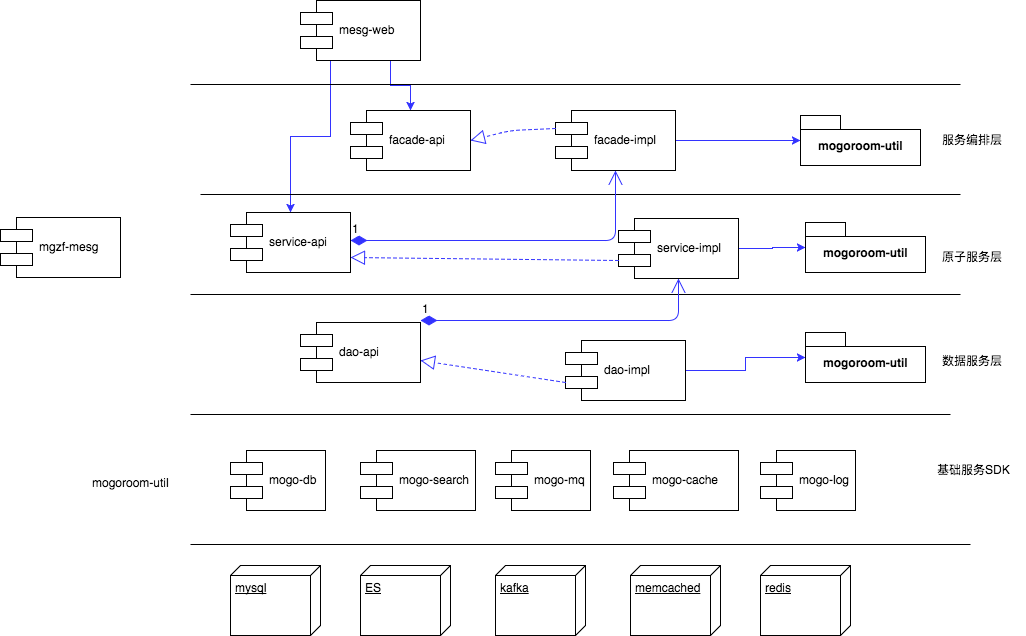


图 1‑2 模块代码结构图

例如mgzf-mesg模块的结构如上图所示。在最上层的mgzf-web代表controller层，以下依次是服务编排层（facade）、原子服务层（service）、数据服务层（DAO）。

代码分层严格遵循接口隔离的规范，层与层直接必须通过接口隔离，层内部之间实现无需接口隔离。比如facade层api，不允许依赖于service层的api。

Mogo-web(即controller层)调用façade-api module接口，façade-impl module实现façade-api中定义的接口，service层、DAO层与façade层的结构类似。

service作为业务服务层，初期作为业务模块jar提供上层调用，后台作为服务发布给facade及controller调用。

注：模块内的Service层公共业务逻辑，以Helper类的方式在 service-impl层横向调用，这样不会影响上层的业务逻辑。

基础服务SDK分三个部分：工具类（utils）、第三方封装（sdk）和分层框架（base）。

### controller层

#### 包结构

域名.系统

com.mogoroom.renterController

|\_\_ user 用户管理

|\_\_ flat 房源管理

|\_\_ …… 其他其他模块

具体要求参见WIKI：

http://192.168.60.125:8080/dokuwiki/doku.php?id=技术规范:app接口开发规范

返回值代码：参见mogo-framework-sdk中关于web模块的response的代码说明及UT。

### facade层

#### 描述

[门面模式](http://blog.csdn.net/hguisu/article/details/7533759)，我们通过外观的包装，使应用程序只能看到外观对象，而不会看到具体的细节对象，这样无疑会降低应用程序的复杂度，并且提高了程序的可维护性，在蘑菇系统中实际是服务流程编排。

【强制】facade层不可以跨模块相互调用。

Facade层用来聚合跨模块的业务逻辑，用来组合跨模块的service以组成更大粒度的业务功能，即我们常说的服务编排。服务就是系统对外提供的一个业务功能。

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 **String** 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 ;

}

#### 使用原则

a. facade 的添加必须按功能模块隔离，并且最大粒度封装。

b. facade 不做底层业务处理，只做作service的编排，处理跨模块的service调用，组合业务逻辑,参见4.1节的例子。

c. facade 的编写必须有详细的业务流程文档状态图。

d. facade 添加后需要经过代码审核，如果编写不规范需要重构。

e. facade层，如果不调用外部服务，可以跳过facade直接调用service层。

#### 包结构

包结构，按com.mgzf.功能.分层命名。如com.mgzf.flat.facade

com.mgzf.flat.facade

|\_\_ api(模块) 服务流程接口层，单独一个项目，对上层renter/bs等暴露原子性的接口。

|\_\_ enums 枚举。

|\_\_ dto 接口入参及出参。

|\_\_ exception 本层的异常，命名以FlatFacadeException命名，继承于MogoException。

|\_\_ MogoException.java 这个类独立复制过来就可以

com.mgzf.flat.facade

|\_\_ impl **服务流程编排层**，处理跨维度、跨服务调用的业务流程，规范业务流程，必须分包。

|\_\_ community 小区管理

|\_\_ room 房源管理

|\_\_ …… 其他其他模块

|\_\_util 本模块内部使用的工具类

### 业务服务层Service

#### 包结构

com.mgzf.user.service用户管理

|\_\_ api 服务流程接口层，单独一个项目，对上层renter/bs等暴露原子性的接口。

|\_\_ enums 枚举。

|\_\_ dto 接口入参及出参。

|\_\_ exception 本层的异常，命名方式UserServiceException

com.mgzf.user.service用户管理

|\_\_ impl **业务逻辑层**，提供上层调用的基础接口，添加事物，保证接口幂等性，回滚等发布原则。

|\_\_ renter 租客管理

|\_\_ landlord 房东管理

|\_\_ …… 其他其他模块

|\_\_util 本模块内部使用的工具类

先按业务分包，然后再按接口实现。

枚举统一放在自己service.api包中。

【强制】service层不可以跨模块相互调用（服务化之后可以开放，但是JVM级别单向不循环）。

### Dao数据层

先按业务分包，然后再按接口实现。

DAO层只能执行单表操作，多表的业务逻辑放到Service层作为业务功能实现。

Dao层实现类，实现BaseDao接口。

com.mgzf.user.service.dao **数据逻辑层**提供上层业务需要的基础CURD操作，添加缓存提高性能，禁止批量事务提交，

|\_\_api 服务流程接口层，单独一个项目，对上层renter/bs等暴露原子性的接口。

|\_\_enums 枚举。

|\_\_exception 本层的异常，命名方式UserServiceException

|\_\_impl basedao基础Dao，baseModel,基础bean。数据库的entity禁止暴漏给service接口外部。

|\_\_UserDaoImpl

|\_\_ selectById() 基础curd,隔离底层实现，对外不要暴露是数据库还是缓存或者队列。

|\_\_ long deleteById() 基础curd

|\_\_ long updateById() 基础curd

|\_\_ long insert(T t) 基础curd

|\_\_ selectById() 基础curd,bean/map

|\_\_ long delete () 基础curd, bean/map

|\_\_ long update () 基础curd, bean/map

|\_\_ mapper XML 映射，只处理单表操作

|\_\_ entity 表实体 只在DAO层使用

|\_\_ dto 表示DAO层输出的实体（包括入参、出参、JOIN的结果）

|\_\_util 本模块内部使用的工具类

实现层可以增加一个本层的UtilModule,这样将来做替换的时候可以方便的进行各层依赖，要求，不依赖其它模块。

## 系统解耦及降级规则

1. 重要系统独立部署（eg：支付）。
2. 读写分离：减少主库压力。
3. 动静分离：能静态化优先静态化。
4. 事务一致性：采用最终一致原则，具体可以用异步通知等方式。
5. 在合适的地方采用合适的技术，比如能够缓存的加缓存，启用全文检索（elasticsearch等）。
6. 与第三方系统对接的所有接口统一部署到oapi中，对外接口要有限流机制，接受到请求之后直接存储到消息队列中，业务异步处理；不允许和我们的主业务系统部署在一起
7. 所有调用第三方服务的业务，都必须实现服务降级。
8. 重要的第三方服务，需实现备选服务商的接入

# 代码编写规范

## 命名风格

1. 【强制】【hook】代码中的命名均不能以下划线或美元符号开始，也不能以下划线或美元符号结束。
2. 【强制】【hook】代码中的命名严禁使用拼音与英文混合的方式，更不允许直接使用中文的方式。
3. 【强制】【hook】类名使用 UpperCamelCase 风格，必须遵从驼峰形式，但以下情形例外:DO/BO/DTO/VO/AO

正例:MarcoPolo / UserDO / XmlService / TcpUdpDeal / TaPromotion

反例:macroPolo / UserDo / XMLService / TCPUDPDeal / TAPromotion

1. 【强制】【hook】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. 【强制】【hook】抽象类命名使用 Abstract 或 Base 开头;异常类命名使用 Exception 结尾;测试类 命名以它要测试的类的名称开始，以 Test 结尾。
2. 【强制】 【hook】中括号是数组类型的一部分，数组定义如下:String[] args;
3. 【强制】【hook】POJO 类中布尔类型的变量，都不要加 is，否则部分框架解析会引起序列化错误。
4. 【强制】【hook】包名统一使用小写，点分隔符之间有且仅有一个自然语义的英语单词。包名统一使用 单数形式，但是类名如果有复数含义，类名可以使用复数形式。

正例: 应用工具类包名为com.alibaba.open.util、类名为MessageUtils(此规则参考 spring 的框架结构)

1. 【强制】杜绝完全不规范的缩写，避免望文不知义。

反例:AbstractClass“缩写”命名成 AbsClass;condition“缩写”命名成 condi，此类随 意缩写严重降低了代码的可阅读性。

1. 【推荐】为了达到代码自解释的目标，任何自定义编程元素在命名时，使用尽量完整的单词 组合来表达其意。

正例:从远程仓库拉取代码的类命名为 PullCodeFromRemoteRepository。

反例:变量 int a; 的随意命名方式。

1. 【推荐】如果模块、接口、类、方法使用了设计模式，在命名时体现出具体模式。

说明:将设计模式体现在名字中，有利于阅读者快速理解架构设计理念。

正例:

public class OrderFactory;

public class LoginProxy;

public class ResourceObserver;

1. 【推荐】接口类中的方法和属性不要加任何修饰符号(public 也不要加)，保持代码的简洁 性，并加上有效的 Javadoc 注释。尽量不要在接口里定义变量，如果一定要定义变量，如果一定要使用变量，那么应该是常量，在相应的接口的平级包enum中定义。

正例:接口方法签名:void f();

接口基础常量表示:String COMPANY = "alibaba";

反例:接口方法定义:public abstract void f();

说明:JDK8 中接口允许有默认实现，那么这个default ­­­方法，是对所有实现类都有价值的默认实现。

1. 【推荐】 【hook】如果是形容能力的接口名称，取对应的形容词做接口名(通常是–able 的形式)。 正例:AbstractTranslator 实现 Translatable。
2. 【参考】枚举类名建议带上 Enum 后缀，枚举成员名称需要全大写，单词间用下划线隔开。

说明:枚举其实就是特殊的常量类，且构造方法被默认强制是私有。

正例:枚举名字为ProcessStatusEnum的成员名称:

SUCCESS /UNKOWN\_REASON。

1. 【强制】【hook】方法名称，参数名称，成员变量名称和局部变量名称应该写在lowerCamelCase中。正面的例子：localValue / getHttpMessage（）/ inputUserId

## 分层相关命名规范

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/insert做前缀。
5. 删除的方法用remove/delete做前缀。
6. 修改的方法用update做前缀。
7. 分页查找：listXXXPage

facade、service的接口层的出入参数统一为DTO。

限制规则：所有的入参与出参必须实现 序列化与clone接口，并生成：’serialVersionUID’。

方法是：在idea中创建类时启用’serialVersionUID’，设置如下：Setting->Inspections->java->Serialization issues->Serializable class without ’serialVersionUID’

### 各层类、方法名命名

**Service/facade/controller层**

**Service的类**

**接口命名： 模块IxxxService，模块Ixxxfacade**

**实现命名： 模块xxxServiceImpl，模块xxxFacadeImpl**

**Dao层**

**类命名：模块xxxDaoImp，例如 UserRenterDaoImpl**

具体的Dao层\*命名规则参考spring-data 语法，例如

|  |  |  |
| --- | --- | --- |
| 无条件 | getOne()/listAll() | 查找所有 |
| And | listByLastnameAndFirstname | … where x.lastname = ?1 and x.firstname = ?2 |
| Or | listByLastnameOrFirstname | … where x.lastname = ?1 or x.firstname = ?2 |
| Between | listByStartDateBetween | … where x.startDate between 1? and ?2 |
| LessThan | listByAgeLessThan | … where x.age < ?1 |
| GreaterThan | listByAgeGreaterThan | … where x.age > ?1 |
| IsNull | listByAgeIsNull | … where x.age is null |
| IsNotNull,NotNull | listByAge(Is)NotNull | … where x.age not null |
| Like | listByFirstnameLike | … where x.firstname like ?1 |
| NotLike | listByFirstnameNotLike | … where x.firstname not like ?1 |
| OrderBy | listByAgeOrderByLastnameDesc | … where x.age = ?1 order by x.lastname desc |
| Not | listByLastnameNot | … where x.lastname <> ?1 |
| In | listByAgeIn(Collection<Age> ages) | … where x.age in ?1 |
| NotIn | listByAgeNotIn(Collection<Age> age) | … where x.age not in ?1 |

**Mapper层(数据库连接层)**

* + - 1. 实体命名均采用实体类名称 + Entity，如：

com.mgzf.service.模块.dao.entity.UserEntity

* + - 1. 每个Mapper需继承BaseMapper，此类采用的泛型，如下图：

public interface UserMapper extends BaseMapper<UserEntity>;

* + - 1. xml统一存放于与mapper相同的包结构目录：

com.mgzf.service.模块.dao.mapper.userMapper.xml

* + - 1. mapper命名均采用实体类名称(第一个字母小写) + Mapper，如：

com.mgzf.service.模块.dao.mapper.userMapper.java;

* + - 1. 每个Xml在生成的时候会存在一个“BaseResultMap”，此ResultMap为基础Map，禁止在其加入其它和数据库表字段不对应的属性，如图：

<**resultMap id="BaseResultMap" type="map"**>  
 <**result column="income" property="income" jdbcType="DECIMAL"** />  
 <**result column="shouldIncome" property="shouldIncome" jdbcType="DECIMAL"** />  
 <**result column="owedIncome" property="owedIncome" jdbcType="DECIMAL"** />  
 <**result column="outcome" property="outcome" jdbcType="DECIMAL"** />  
 <**result column="shouldOutcome" property="shouldOutcome" jdbcType="DECIMAL"** />  
 <**result column="owedOutcome" property="owedOutcome" jdbcType="DECIMAL"** />  
 <**result column="date" property="date" jdbcType="TIMESTAMP"** />  
 <**result column="totalAmount" property="totalAmount" jdbcType="DECIMAL"** />  
 <**result column="totalPartPayAmount" property="totalPartPayAmount" jdbcType="DECIMAL"** />  
 <**result column="partPayBillCount" property="partPayBillCount" jdbcType="INTEGER" javaType="int"**/>  
</**resultMap**>

* + - 1. **mapper**所有**更新接口必须返回long类型**
      2. **dao 类统一采用 模块+daoImpl 命名放在impl包下：**

com.mgzf.模块.dao.impl.UserDaoImpl.java;

## 常量与枚举定义

1. 【强制】【hook】不允许任何魔法值(即未经定义的常量)直接出现在代码中。

反例:String key = "Id#taobao\_" + tradeId;

cache.put(key, value);

1. 【强制】【hook】long 或者 Long 初始赋值时，使用大写的 L，不能是小写的 l，小写容易跟数字 1 混 淆，造成误解。

说明:Long a = 2l; 写的是数字的 21，还是 Long 型的 2?

1. 【强制】不要使用一个常量类维护所有常量，按常量功能进行归类，分开维护。

说明:大而全的常量类，非得使用查找功能才能定位到修改的常量，不利于理解和维护。

正例:缓存相关常量放在类 CacheConsts 下;系统配置相关常量放在类 ConfigConsts 下。

1. 【推荐】常量的复用层次有五层:跨应用共享常量、应用内共享常量、子工程内共享常量、包 内共享常量、类内共享常量。
2. 跨应用共享常量:放置在二方库中，通常是client.jar中的constant目录下。
3. 应用内共享常量:放置在一方库中，通常是modules中的constant目录下。

反例:易懂变量也要统一定义成应用内共享常量，两位攻城师在两个类中分别定义了表示 “是”的变量:

类 A 中:public static final String YES = "yes";

类 B 中:public static final String YES = "y"; A.YES.equals(B.YES)，预期是 true，但实际返回为 false，导致线上问题。

1. 子工程内部共享常量:即在当前子工程的constant目录下。
2. 包内共享常量:即在当前包下单独的constant目录下。
3. 类内共享常量:直接在类内部private static final定义。
4. 【推荐】如果变量值仅在一个范围内变化，且带有名称之外的延伸属性，定义为枚举类。下面 正例中的数字就是延伸信息，表示星期几。

正例:public Enum { MONDAY(1), TUESDAY(2), WEDNESDAY(3), THURSDAY(4), FRIDAY(5), SATURDAY(6), SUNDAY(7);}

1. 【强制】模块内部使用，或者非公共的枚举放到自己模块内部。

## 代码格式

1. 【强制】大括号的使用约定。如果是大括号内为空，则简洁地写成{}即可，不需要换行;如果 是非空代码块则:
2. 左大括号前不换行。
3. 左大括号后换行。
4. 右大括号前换行。
5. 右大括号后还有else等代码则不换行;表示终止的右大括号后必须换行。
6. 【强制】 左小括号和字符之间不出现空格;同样，右小括号和字符之间也不出现空格。详见 第 5 条下方正例 示。

反例:if (空格a == b空格)

1. 【强制】if/for/while/switch/do 等保留字与括号之间都必须加空格。
2. 【强制】任何二目、三目运算符的左右两边都需要加一个空格。

说明:运算符包括赋值运算符=、逻辑运算符&&、加减乘除符号等。

1. 【强制】采用 4 个空格缩进，禁止使用 tab 字符。

说明:如果使用 tab 缩进，必须设置 1 个 tab 为 4 个空格。IDEA 设置 tab 为 4 个空格时， 请勿勾选 Use tab character；而在eclipse中，必须勾选insert spaces for tabs。

正例: (涉及1-5点)

public static void main(String[] args) {

// 缩进 4 个空格

String say = "hello";

// 运算符的左右必须有一个空格 int flag = 0;

// 关键词 if 与括号之间必须有一个空格，括号内的 f 与左括号，0 与右括号不需要空格

if(flag==0){

System.out.println(say);

}

// 左大括号前加空格且不换行;左大括号后换行

if(flag==1){

System.out.println("world");

// 右大括号前换行，右大括号后有 else，不用换行

}else{

System.out.println("ok");

// 在右大括号后直接结束，则必须换行

}

}

1. 【强制】注释的双斜线与注释内容之间有且仅有一个空格。

正例:// 注释内容，注意在//和注释内容之间有一个空格

1. 【强制】单行字符数限制不超过 120 个，超出需要换行，换行时遵循如下原则:
2. 第二行相对第一行缩进 4 个空格，从第三行开始，不再继续缩进，参考示例。
3. 运算符与下文一起换行。
4. 方法调用的点符号与下文一起换行。
5. 方法调用时，多个参数，需要换行时，在逗号后进行。
6. 在括号前不要换行，见反例。

正例:

StringBuffer sb = new StringBuffer();

// 超过 120 个字符的情况下，换行缩进 4 个空格，点号和方法名称一起换行

sb.append("zi").append("xin")...

.append("huang")...

.append("huang")...

.append("huang");

反例:

StringBuffer sb = new StringBuffer();

// 超过 120 个字符的情况下，不要在括号前换行 sb.append("zi").append("xin")...append

("huang");

// 参数很多的方法调用可能超过 120 个字符，不要在逗号前换行

method(args1, args2, args3, ...

, argsX);

1. 【强制】方法参数在定义和传入时，多个参数逗号后边必须加空格。

正例:下例中实参的"a",后边必须要有一个空格。

method("a", "b", "c");

1. 【强制】IDE的text file encoding设置为UTF-8; IDE中文件的换行符使用Unix格式，不要使用 Windows 格式。
2. 【推荐】没有必要增加若干空格来使某一行的字符与上一行对应位置的字符对齐。

正例:

inta=3;

longb=4L;

float c = 5F;

StringBuffer sb = new StringBuffer();

说明:增加 sb 这个变量，如果需要对齐，则给 a、b、c 都要增加几个空格，在变量比较多的 情况下，是一种累赘的事情。

1. 【推荐】方法体内的执行语句组、变量的定义语句组、不同的业务逻辑之间或者不同的语义之间插入一个空行。相同业务逻辑和语义之间不需要插入空行。

说明:没有必要插入多个空行进行隔开。

1. 【强制】import中不能有通配符
2. 【强制】代码行数不得超过1000行
3. 【强制】文件回车检查-必须是Linux的LF，不能是Windows的CRLF，也不能是Mac的CR
4. 【强制】代码字符编码检查UTF-8
5. 【强制】git提交邮箱必须是mgzf.com|mogoroom.com

## OOP规约

1. 【强制】避免通过一个类的对象引用访问此类的静态变量或静态方法，无谓增加编译器解析成本，直接用类名来访问即可。
2. 【强制】所有的覆写方法，必须加@Override 注解。

说明:getObject()与 get0bject()的问题。一个是字母的 O，一个是数字的 0，加@Override 可以准确判断是否覆盖成功。另外，如果在抽象类中对方法签名进行修改，其实现类会马上编译报错。

1. 【强制】相同参数类型，相同业务含义，才可以使用 Java 的可变参数，避免使用 Object。

说明:可变参数必须放置在参数列表的最后。( 倡同学们尽量不用可变参数编程)

正例:public User getUsers(String type, Integer... ids) {...}

1. 【强制】外部正在调用或者二方库依赖的接口，不允许修改方法签名，避免对接口调用方产生影响。接口过时必须加@Deprecated 注解，并清晰地说明采用的新接口或者新服务是什么。
2. 【强制】【hook】不能使用过时的类或方法。

说明:java.net.URLDecoder 中的方法decode(StringencodeStr) 这个方法已经过时，应 该使用双参数 decode(String source, String encode)。接口供方既然明确是过时接口，那么有义务同时 供新的接口;作为调用方来说，有义务去考证过时方法的新实现是什么。

1. 【强制】【hook】Object的equals 方法容易抛空指针异常，应使用常量或确定有值的对象来调用equals。

正例:"test".equals(object);

反例:object.equals("test");

说明:推荐使用 java.util.Objects#equals(JDK7 引入的工具类)

1. 【强制】【hook】所有的相同类型的包装类对象之间值的比较，全部使用 equals 方法比较。说明:对于Integervar=? 在-128至127范围内的赋值，Integer对象是在IntegerCache.cache产生，会复用已有对象，这个区间内的Integer值可以直接使用==进行判断，但是这个区间之外的所有数据，都会在堆上产生，并不会复用已有对象，这是一个大坑，推荐使用 equals方法进行判断。
2. 关于基本数据类型与包装数据类型的使用标准如下:
3. 【强制】【hook】所有的POJO类属性必须使用包装数据类型。
4. 【强制】【hook】RPC方法的返回值和参数必须使用包装数据类型。
5. 【推荐】所有的局部变量使用基本数据类型。

说明:POJO 类属性没有初值是 醒使用者在需要使用时，必须自己显式地进行赋值，任何NPE 问题，或者入库检查，都由使用者来保证。

正例:数据库的查询结果可能是 null，因为自动拆箱，用基本数据类型接收有 NPE 风险。

反例:比如显示成交总额涨跌情况，即正负 x%，x 为基本数据类型，调用的 RPC 服务，调用不成功时，返回的是默认值，页面显示为 0%，这是不合理的，应该显示成中划线。所以包装 数据类型的 null 值，能够表示额外的信息，如:远程调用失败，异常退出。

1. 【强制】【hook】定义 DO/DTO/VO 等 POJO 类时，不要设定任何属性默认值。 反例:POJO类的gmtCreate默认值为new Date();但是这个属性在数据 取时并没有置入具体值，在更新其它字段时又附带更新了此字段，导致创建时间被修改成当前时间。
2. 【强制】序列化类新增属性时，请不要修改 serialVersionUID 字段，避免反序列失败;如 果完全不兼容升级，避免反序列化混乱，那么请修改 serialVersionUID 值。

说明:注意 serialVersionUID 不一致会抛出序列化运行时异常。

1. 【强制】构造方法里面禁止加入任何业务逻辑，如果有初始化逻辑，请放在 init 方法中。
2. 【强制】【hook】POJO 类必须写 toString 方法。使用 IDE 的中工具:source> generate toString 时，如果继承了另一个 POJO 类，注意在前面加一下 super.toString。 说明:在方法执行抛出异常时，可以直接调用 POJO 的 toString()方法打印其属性值，便于排查问题。
3. 【推荐】使用索引访问用 String 的 split 方法得到的数组时，需做最后一个分隔符后有无内容的检查，否则会有抛 IndexOutOfBoundsException 的风险。

说明:

String str = "a,b,c,,"; String[] ary = str.split(",");

// 预期大于 3，结果是 3

System.out.println(ary.length);

1. 【推荐】当一个类有多个构造方法，或者多个同名方法，这些方法应该按顺序放置在一起， 便于阅读，此条规则优先于第 15 条规则。
2. 【推荐】 类内方法定义顺序依次是:公有方法或保护方法 > 私有方法 > getter/setter 方法。

说明:公有方法是类的调用者和维护者最关心的方法，首屏展示最好;保护方法虽然只是子类关心，也可能是“模板设计模式”下的核心方法;而私有方法外部一般不需要特别关心，是一个 黑盒实现;因为承载的信息价值较低，所有 Service 和 DAO 的 getter/setter 方法放在类体最后。

1. 【推荐】setter 方法中，参数名称与类成员变量名称一致，this.成员名 = 参数名。在getter/setter 方法中，不要增加业务逻辑，增加排查问题的难度。

反例:

public Integer getData() { if (true) {

return this.data + 100; }else{

return this.data - 100; }

}

1. 【强制】【hook】循环体内，字符串的连接方式，使用 StringBuilder 的 append 方法进行扩展。 说明:反编译出的字节码文件显示每次循环都会 new 出一个 StringBuilder 对象，然后进行 append 操作，最后通过 toString 方法返回 String 对象，造成内存资源浪费。

反例:

String str = "start";

for(inti=0;i<100;i++){

str = str + "hello";

}

1. 【推荐】final 可以声明类、成员变量、方法、以及本地变量，下列情况使用 final 关键字:
2. 不允许被继承的类，如:String 类。
3. 不允许修改引用的域对象，如:POJO类的域变量。
4. 不允许被重写的方法，如:POJO类的setter方法。
5. 不允许运行过程中重新赋值的局部变量。
6. 避免上下文重复使用一个变量，使用 final描述可以强制重新定义一个变量，方便更好地进行重构。
7. 【推荐】慎用 Object的clone 方法来拷贝对象。

说明:对象的 clone 方法默认是浅拷贝，若想实现深拷贝需要重写 clone 方法实现属性对象的拷贝。

1. 【推荐】类成员与方法访问控制从严:
2. 如果不允许外部直接通过new来创建对象，那么构造方法必须是private。
3. 工具类不允许有public或default构造方法。
4. 类非static成员变量并且与子类共享，必须是protected。
5. 类非static成员变量并且仅在本类使用，必须是private。
6. 类static成员变量如果仅在本类使用，必须是private。
7. 若是static成员变量，必须考虑是否为final。
8. 类成员方法只供类内部调用，必须是private。
9. 类成员方法只对继承类公开，那么限制为protected。

说明:任何类、方法、参数、变量，严控访问范围。过于宽泛的访问范围，不利于模块解耦。

思考:如果是一个 private 的方法，想删除就删除，可是一个 public 的 service 方法，或者 一个 public 的成员变量，删除一下，不得手心冒点汗吗?**变量像自己的小孩，**尽量在自己的视线内，变量作用域太大，无限制的到处跑，那么你会担心的。

## 集合处理

1. 【强制】关于 hashCode 和 equals 的处理，遵循如下规则:
2. 只要重写equals，就必须重写hashCode。
3. 因为Set存储的是不重复的对象，依据hashCode和equals进行判断，所以Set存储的 对象必须重写这两个方法。
4. 如果自定义对象做为Map的键，那么必须重写hashCode和equals。

说明:String 重写了 hashCode 和 equals 方法，所以我们可以非常愉快地使用 String 对象 作为 key 来使用。

1. 【强制】【hook】ArrayList的subList结果不可强转成ArrayList，否则会抛出ClassCastException 异常，即java.util.RandomAccessSubList cannot be cast to java.util.ArrayList.

说明:subList 返回的是 ArrayList 的内部类 SubList，并不是 ArrayList ，而是 ArrayList 的一个视图，对于SubList子列表的所有操作最终会反映到原列表上。

1. 【强制】【hook】在 subList 场景中，高度注意对原集合元素个数的修改，会导致子列表的遍历、增加、 删除均会产生ConcurrentModificationException 异常。
2. 【强制】 【hook】使用集合转数组的方法，必须使用集合的 toArray(T[] array)，传入的是类型完全 一样的数组，大小就是 list.size()。

说明:使用 toArray 带参方法，入参分配的数组空间不够大时，toArray 方法内部将重新分配 内存空间，并返回新数组地址;如果数组元素大于实际所需，下标为[ list.size() ]的数组 元素将被置为 null，其它数组元素保持原值，因此最好将方法入参数组大小定义与集合元素 个数一致。

正例:

List<String> list = new ArrayList<String>(2);

list.add("guan");

list.add("bao");

String[] array = new String[list.size()];

array = list.toArray(array);

反例:

直接使用 toArray 无参方法存在问题，此方法返回值只能是 Object[]类，若强转其它 类型数组将出现 ClassCastException 错误。

1. 【强制】【hook】使用工具类 Arrays.asList()把数组转换成集合时，不能使用其修改集合相关的方 法，它的 add/remove/clear 方法会抛出 UnsupportedOperationException 异常。

说明:asList 的返回对象是一个 Arrays 内部类，并没有实现集合的修改方法。Arrays.asList 体现的是适配器模式，只是转换接口，后台的数据仍是数组。

String[] str = new String[] { "you", "wu" };

List list = Arrays.asList(str);

第一种情况:list.add("yangguanbao"); 运行时异常。

第二种情况:str[0] = "gujin"; 那么 list.get(0)也会随之修改。

1. 【强制】泛型通配符<? extends T>来接收返回的数据，此写法的泛型集合不能使用 add 方 法，而<? super T>不能使用 get 方法，做为接口调用赋值时易出错。

说明:扩展说一下 PECS(Producer Extends Consumer Super)原则:

1. 频繁往外读取内 容的，适合用<? extends T>。
2. 第二、经常往里插入的，适合用<? super T>。
3. 【强制】 【hook】不要在 foreach 循环里进行元素的 remove/add 操作。remove 元素请使用 Iterator方式，如果并发操作，需要对 Iterator 对象加锁。

正例:

Iterator<String> iterator = list.iterator();

while (iterator.hasNext()) {

String item = iterator.next();

if (删除元素的条件) {

iterator.remove();

}

}

反例:

List<String> a = new ArrayList<String>();

list.add("1");

list.add("2");

for (String item : list) {

if ("1".equals(item)) {

list.remove(item);

}

}

说明:以上代码的执行结果肯定会出乎大家的意料，那么试一下把“1”换成“2”，会是同样的 结果吗?

1. 【强制】 在 JDK7 版本及以上，Comparator 要满足如下三个条件，不然 Arrays.sort， Collections.sort 会报 IllegalArgumentException 异常。

说明:三个条件如下

1. 自反性：x，y的比较结果和y，x的比较结果相反。
2. 传递性：x>y，y>z，则x>z。
3. 对称性：x=y，则x，z比较结果和y，z比较结果相同。

反例:

下例中没有处理相等的情况，实际使用中可能会出现异常:

new Comparator<Student>() {

@Override

public int compare(Student o1, Student o2) {

return o1.getId() > o2.getId() ? 1 : -1; }

};

1. 【推荐】【hook】集合初始化时，指定集合初始值大小。

说明:HashMap使用HashMap(int initialCapacity) 初始化。

正例:initialCapacity = (需要存储的元素个数 / 负载因子) + 1。注意负载因子(即 loader factor)默认为 0.75，如果暂时无法确定初始值大小，请设置为 16(即默认值)。

反例:HashMap 需要放置 1024 个元素，由于没有设置容量初始大小，随着元素不断增加，容 量 7 次被迫扩大，resize 需要重建 hash 表，严重影响性能。

1. 【推荐】使用 entrySet 遍历 Map 类集合 KV，而不是 keySet 方式进行遍历。

说明:keySet 其实是遍历了 2 次，一次是转为 Iterator 对象，另一次是从 hashMap 中取出 key 所对应的 value。而 entrySet 只是遍历了一次就把 key 和 value 都放到entry中，效率更高。如果是 JDK8，使用Map.foreach方法。

正例:values()返回的是V值集合，是一个list集合对象;keySet()返回的是 K 值集合，是 一个 Set 集合对象;entrySet()返回的是 K-V 值组合集合。

1. 【推荐】高度注意 Map 类集合 K/V 能不能存储 null 值的情况，如下表格:



反例: 由于 HashMap 的干扰，很多人认为 ConcurrentHashMap 是可以置入 null 值，而事实上， 存储 null 值时会抛出 NPE 异常。

1. 【参考】合理利用好集合的有序性(sort)和稳定性(order)，避免集合的无序性(unsort)和 不稳定性(unorder)带来的负面影响。

说明:有序性是指遍历的结果是按某种比较规则依次排列的。稳定性指集合每次遍历的元素次 序是一定的。如:ArrayList 是 order/unsort;HashMap 是 unorder/unsort;TreeSet 是order/sort。

1. 【参考】利用 Set 元素唯一的特性，可以快速对一个集合进行去重操作，避免使用 List 的contains 方法进行遍历、对比、去重操作。

## 并发处理

【强制】【hook】获取单例对象需要保证线程安全，其中的方法也要保证线程安全。

说明:资源驱动类、工具类、单例工厂类都需要注意。

1. 【强制】【hook】创建线程或线程池时请指定有意义的线程名称，方便出错时回溯。

正例:

public class TimerTaskThread extends Thread {

public TimerTaskThread() {

super.setName("TimerTaskThread");

...

}

}

1. 【强制】【hook】线程资源必须通过线程池 供，不允许在应用中自行显式创建线程。

说明:使用线程池的好处是减少在创建和销毁线程上所花的时间以及系统资源的开销，解决资 源不足的问题。如果不使用线程池，有可能造成系统创建大量同类线程而导致消耗完内存或者 “过度切换”的问题。

1. 【强制】【hook】线程池不允许使用Executors 去创建，而是通过 ThreadPoolExecutor 的方式，这样 的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险。

说明:Executors 返回的线程池对象的弊端如下:

1. FixedThreadPool 和 SingleThreadPool:

允许的请求队列长度为 Integer.MAX\_VALUE，可能会堆积大量的请求，从而导致 OOM。

1. CachedThreadPool 和 ScheduledThreadPool:

允许的创建线程数量为 Integer.MAX\_VALUE，可能会创建大量的线程，从而导致 OOM。

1. 【强制】【hook】SimpleDateFormat 是线程不安全的类，一般不要定义为static变量，如果定义为static，必须加锁，或者使用 DateUtils 工具类。

正例:注意线程安全，使用 DateUtils。亦推荐如下处理:

private static final ThreadLocal<DateFormat> df = new ThreadLocal<DateFormat>(){

@Override

protected DateFormat initialValue() {

return new SimpleDateFormat("yyyy-MM-dd"); }

};

说明:如果是 JDK8 的应用，可以使用 Instant 代替 Date，LocalDateTime 代替 Calendar，DateTimeFormatter 代替 SimpleDateFormat，官方给出的解释:simple beautiful strong immutable thread-safe。

1. 【强制】高并发时，同步调用应该去考量锁的性能损耗。能用无锁数据结构，就不要用锁;能锁区块，就不要锁整个方法体;能用对象锁，就不要用类锁。

说明:尽可能使加锁的代码块工作量尽可能的小，避免在锁代码块中调用 RPC 方法。

1. 【强制】对多个资源、数据库表、对象同时加锁时，需要保持一致的加锁顺序，否则可能会造 成死锁。

说明:线程一需要对表 A、B、C 依次全部加锁后才可以进行更新操作，那么线程二的加锁顺序 也必须是 A、B、C，否则可能出现死锁。

1. 【强制】并发修改同一记录时，避免更新丢失，需要加锁。要么在应用层加锁，要么在缓存加 锁，要么在数据库层使用乐观锁，使用 version 作为更新依据。

说明:如果每次访问冲突概率小于 20%，推荐使用乐观锁，否则使用悲观锁。乐观锁的重试次 数不得小于 3 次。

1. 【强制】【hook】多线程并行处理定时任务时，Timer 运行多个 TimeTask 时，只要其中之一没有捕获 抛出的异常，其它任务便会自动终止运行，使用 ScheduledExecutorService 则没有这个问题。
2. 【强制】【hook】使用 CountDownLatch 进行异步转同步操作，每个线程退出前必须调用 countDown 方法，线程执行代码注意 catch 异常，确保 countDown 方法被执行到，避免主线程无法执行 至 await 方法，直到超时才返回结果。

说明:注意，子线程抛出异常堆栈，不能在主线程 try-catch 到。

1. 【强制】【hook】避免 Random 实例被多线程使用，虽然共享该实例是线程安全的，但会因竞争同一 seed 导致的性能下降。

说明:Random 实例包括 java.util.Random 的实例或者 Math.random()的方式。

正例:在 JDK7 之后，可以直接使用 API ThreadLocalRandom，而在 JDK7 之前，需要编码保 证每个线程持有一个实例。

1. 【推荐】在并发场景下，通过双重检查锁(double-checked locking)实现延迟初始化的优化问题隐患(可参考 The "Double-Checked Locking is Broken" Declaration)，推荐解 决方案中较为简单一种(适用于 JDK5 及以上版本)，将目标属性声明为 volatile 型。

参考：http://www.infoq.com/cn/articles/double-checked-locking-with-delay-initialization

反例:

class Singleton {

private Helper helper = null;

public Helper getHelper() {

if (helper == null) synchronized(this) {

if (helper == null){

helper = new Helper();

}

return helper;

}

}

// other methods and fields

...

}

正例：把instance声明为volatile型

public class SafeDoubleCheckedLocking {

private volatile static Instance instance;

public static Instance getInstance() {

if (instance == null) {

synchronized (SafeDoubleCheckedLocking.class) {

if (instance == null){

//instance为volatile，现在没问题了

instance = new Instance();

}

}

}

return instance;

}

}

1. 【参考】volatile 解决多线程内存不可见问题。对于一写多读，是可以解决变量同步问题，但是如果多写，同样无法解决线程安全问题。如果是 count++操作，使用如下类实现:

AtomicInteger count = new AtomicInteger();

count.addAndGet(1);

如果是 JDK8，推 荐使用 LongAdder 对象，比 AtomicLong 性能更好(减少乐观锁的重试次数)。

1. 【参考】 HashMap 在容量不够进行 resize 时由于高并发可能出现死链，导致 CPU 飙升，在 开发过程中可以使用其它数据结构或加锁来规避此风险。
2. 【参考】ThreadLocal 无法解决共享对象的更新问题，ThreadLocal 对象建议使用 static 修饰。这个变量是针对一个线程内所有操作共享的，所以设置为静态变量，所有此类实例共享 此静态变量 ，也就是说在类第一次被使用时装载，只分配一块存储空间，所有此类的对象(只 要是这个线程内定义的)都可以操控这个变量。
3. 【参考】尽量不使用多线程，因为多线程在程序的资源占用、问题查证等方面的透明度都较差。
4. 【强制】web应用内部严禁启用线程池/多线程。如有特别需求请找架构组沟通确认。
5. 【强制】开发人员如果需要在非web应用使用多线程机制，请在应用层面统一创建线程池。
6. 【强制】ThreadLocal只允许用在技术框架级别，不允许在业务场景传参数，业务相关的参数全部采用方法参数的方式传递。

## 控制语句

【强制】【hook】在一个switch块内，每个case要么通过break/return等来终止，要么注释说明程序将继续执行到哪一个case为止;在一个switch 块内， 都必须包含一个 default 语句并且 放在最后，即使它什么代码也没有。

【强制】【hook】在 if/else/for/while/do 语句中必须使用大括号。即使只有一行代码，避免采用 单行的编码方式:if (condition) statements;

【推荐】表达异常的分支时，少用 if-else 方式，这种方式可以改写成:

if (condition) { ...

return obj;

}

// 接着写 else 的业务逻辑代码;

说明:如果非得使用 if()...else if()...else...方式表达逻辑，【强制】避免后续代码维护困难，请勿超过 3 层。

正例:超过 3 层的 if-else 的逻辑判断代码可以使用卫语句、策略模式、状态模式等来实现， 其中卫语句示例如下:

public void today() {

if (isBusy()) {

System.out.println(“change time.”);

return;

}

if (isFree()) {

System.out.println(“go to travel.”);

return;

}

System.out.println(“stay at home to learn Alibaba Java Coding Guidelines.”);

return;

}

【强制】【hook】除常用方法(如 getXxx/isXxx)等外，不要在条件判断中执行其它复杂的语句，将复杂逻辑判断的结果赋值给一个有意义的布尔变量名，以 高可读性。

说明:很多 if 语句内的逻辑相当复杂，阅读者需要分析条件表达式的最终结果，才能明确什么 样的条件执行什么样的语句，那么，如果阅读者分析逻辑表达式错误呢?

正例:

// 伪代码如下

final boolean existed = (file.open(fileName, "w") != null) && (...) || (...);

if (existed) {

...

}

反例:

if ((file.open(fileName, "w") != null) && (...) || (...)) {

...

}

【推荐】循环体中的语句要考量性能，以下操作尽量移至循环体外处理，如定义对象、变量、获取数据库连接，进行不必要的 try-catch 操作(这个 try-catch 是否可以移至循环体外)。

【推荐】接口入参保护，这种场景常见的是用于做批量操作的接口。

【参考】下列情形，需要进行参数校验:

1. 调用频次低的方法。
2. 执行时间开销很大的方法。此情形中，参数校验时间几乎可以忽略不计，但如果因为参数错误导致中间执行回退，或者错误，那得不偿失。
3. 需要极高稳定性和可用性的方法。
4. 对外 供的开放接口，不管是RPC/API/HTTP接口。
5. 敏感权限入口。

【参考】下列情形，不需要进行参数校验:

1. 极有可能被循环调用的方法。但在方法说明里必须注明外部参数检查要求。
2. 底层调用频度比较高的方法。毕竟是像纯净水过滤的最后一道，参数错误不太可能到底层才会暴露问题。一般 DAO 层与 Service 层都在同一个应用中，部署在同一台服务器中，所以DAO的参数校验，可以省略。
3. 被声明成private只会被自己代码所调用的方法，如果能够确定调用方法的代码传入参 数已经做过检查或者肯定不会有问题，此时可以不校验参数。

## 注释规范

### 通用规范

1. 【强制】【hook】类、类属性、类方法的注释必须使用 Javadoc 规范，使用：

/\*\*内容\*/格式，不得使用 // xxx 方式。

说明:在 IDE 编辑窗口中，Javadoc 方式会 示相关注释，生成 Javadoc 可以正确输出相应注 释;在 IDE 中，工程调用方法时，不进入方法即可悬浮示方法、参数、返回值的意义， 高阅读效率。

1. 【强制】【hook】所有的抽象方法(包括接口中的方法)必须要用 Javadoc 注释、除了返回值、参数、异常说明外，还必须指出该方法做什么事情，实现什么功能。

说明:对子类的实现要求，或者调用注意事项，请一并说明。

1. 【强制】【hook】所有的类都必须添加创建者和创建日期。
2. 【强制】【hook】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。
3. 【强制】【hook】所有的枚举类型字段必须要有注释，说明每个数据项的用途。
4. 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】谨慎注释掉代码。在上方详细说明，而不是简单的注释掉。如果无用，则删除。

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 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。

1. 【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 fridgeput(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. 【参考】特殊注释标记，请注明标记人与标记时间。注意及时处理这些标记，通过标记扫 ，经常清理此类标记。线上故障有时候就是来源于这些标记处的代码。
2. 待办事宜(TODO):( 标记人，标记时间，[预计处理时间]) 表示需要实现，但目前还未实现的功能。这实际上是一个 Javadoc 的标签，目前的Javadoc还没有实现，但已经被广泛使用。只能应用于类，接口和方法(因为它是一个 Javadoc 标签)。
3. 错误，不能工作(FIXME):(标记人，标记时间，[预计处理时间])。在注释中用 FIXME 标记某代码是错误的，而且不能工作，需要及时纠正的情况。

### class注释

@brief 概要描述

@details详细描述

@date 类创建时间

@author 类创建作者(git对应账户名)

### 方法的具体注释

**请使用快捷键”/\*\*”,打完后，自动生成的注释就可以了，比如：**

*/\*\*  
 \*   
 \** ***@param*** *context  
 \** ***@return*** *\*/*

* 在最上面的空格部分，请写明方法的主要业务逻辑、关键算法及注意事项，另外如果使用了设计模式，也要说明。
* 注意：在每个自动生成的需要写注释的项目，都要写注释代码，对于代码编写过后不存在的参数请及时删除 。

### sql mapper 注释

@brief 概要描述

@details详细描述

@date 类创建时间

@author 创建作者

通过在mapper文件中添加<!-- -->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

-->

<selectid=*"selectLandlordByMap"*parameterType=*"map"* resultMap=*"BaseResultMap"*>

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

【强制】业务处理模块内只能单表查询，报表可以允许三表join。

【强制】不允许where条件为空，或者where 1=1的语句。

【强制】单次查询不允许返回超过500条数据。如果是数据导出可以分页导出。另外推荐分页时，使用id作为下次查询的条件，否则后续分页会越来越慢。

### git提交注释

请说明本次提交代码是修改、新增信息，主要修改内容。

注释规范：#[new/opt/bug]@版本#env/空:配置说明

              # 为注释

              new/opt为新增或者修改或bug修复

              @后面为版本

              #后面为环境相关，如果需要运维配置的需要加env，无关的不需要填

              :后面为配置说明

              例如：[new@release5.1.0#](mailto:opt@release5.1.0)env：中行支付密钥

符号必须是应该 不要使用中文的：#

### Mapper.xml中自增主键的获取

自增主键的获取用 useGeneratedKeys="true" keyProperty="id" 获取，例如：

     <insert id="insertSelective" useGeneratedKeys="true" keyProperty="id" parameterType="com.youzhu.user.dao.entity.Renter">

## 二方库依赖

1. 【强制】定义 GAV 遵从以下规则:
2. GroupID格式:com.{公司/BU }.业务线.[子业务线]，最多4级。

说明:{公司/BU} 例如:mogoroom/mgzf 等 BU 一级;子业务线可选。

正例:com.mgzf.mesg 或 com.mogoroom.user.register

1. ArtifactID格式:产品线名-模块名。语义不重复不遗漏，先到中央仓库去查证一下。

正例:mgzf-base / mgzf-service-façade-api

1. Version:详细规定参考2。

2. 【强制】二方库版本号命名方式:主版本号.次版本号.修订号

* 1. 主版本号:产品方向改变，或者大规模API不兼容，或者架构不兼容升级。
  2. 次版本号:保持相对兼容性，增加主要功能特性，影响范围极小的API不兼容修改。
  3. 修订号:保持完全兼容性，修复BUG、新增次要功能特性等。

说明:注意起始版本号必须为:1.0.0，而不是 0.0.1 正式发布的类库必须先去中央仓库进 行查证，使版本号有延续性，正式版本号不允许覆盖升级。如当前版本:1.3.3，那么下一个合理的版本号:1.3.4 或 1.4.0 或 2.0.0

3. 【强制】线上应用不要依赖 SNAPSHOT 版本(安全包除外)。

说明:不依赖 SNAPSHOT版本是**保证应用发布的幂等性**。另外，也可以加快编译时的打包构建。

4. 【强制】二方库的新增或升级，保持除功能点之外的其它 jar 包仲裁结果不变。如果有改变，必须明确评估和验证，建议进行 dependency:resolve 前后信息比对，如果仲裁结果完全不一 致，那么通过 dependency:tree 命令，找出差异点，进行<excludes>排除 jar 包。

5. 【强制】二方库里可以定义枚举类型，参数可以使用枚举类型，但是接口返回值不允许使用枚 举类型或者包含枚举类型的 POJO 对象。

6. 【强制】依赖于一个二方库群时，必须定义一个统一的版本变量，避免版本号不一致。

说明:依赖 springframework-core,-context,-beans，它们都是同一个版本，可以定义一 个变量来保存版本:${spring.version}，定义依赖的时候，引用该版本。

7. 【强制】禁止在子项目的 pom 依赖中出现相同的 GroupId，相同的 ArtifactId，但是不同的 Version。

说明:在本地调试时会使用各子项目指定的版本号，但是合并成一个 war，只能有一个版本号 出现在最后的 lib 目录中。可能出现线下调试是正确的，发布到线上却出故障的问题。

【推荐】所有 pom 文件中的依赖声明放在<dependencies>语句块中，所有版本仲裁放在 <dependencyManagement>语句块中。

说明:<dependencyManagement>里只是声明版本，并不实现引入，因此子项目需要显式的声 明依赖，version 和 scope 都读取自父 pom。而<dependencies>所有声明在主 pom 的 <dependencies>里的依赖都会自动引入，并默认被所有的子项目继承。

9. 【推荐】二方库不要有配置项，最低限度不要再增加配置项。

10. 【参考】为避免应用二方库的依赖冲突问题，二方库发布者应当遵循以下原则:

1. 精简可控原则。移除一切不必要的 API 和依赖，只包含 Service API、必要的领域模型对 象、Utils 类、常量、枚举等。如果依赖其它二方库，尽量是 provided 引入，让二方库使用 者去依赖具体版本号;无 log 具体实现，只依赖日志框架。
2. 稳定可追溯原则。每个版本的变化应该被记录，二方库由谁维护，源码在哪里，都需要能 方便查到。除非用户主动升级版本，否则公共二方库的行为不应该发生变化。

## 其它

1. 【强制】【hook】在使用正则表达式时，利用好其预编译功能，可以有效加快正则匹配速度。

说明:不要在方法体内定义:Pattern pattern = Pattern.compile(规则);

1. 【强制】velocity 调用 POJO 类的属性时，建议直接使用属性名取值即可，模板引擎会自动按 规范调用 POJO 的 getXxx()，如果是 boolean 基本数据类型变量(boolean 命名不需要加 is 前缀)，会自动调用 isXxx()方法。

说明:注意如果是 Boolean 包装类对象，优先调用 getXxx()的方法。

1. 【强制】【hook】后台输送给页面的变量必须加$!{var}——中间的感叹号。

说明:如果 var=null 或者不存在，那么${var}会直接显示在页面上。

1. 【强制】【hook】注意 Math.random() 这个方法返回是 double 类型，注意取值的范围 0≤x<1(能够 取到零值，注意除零异常)，如果想获取整数类型的随机数，不要将 x 放大 10 的若干倍然后 取整，直接使用 Random 对象的 nextInt 或者 nextLong 方法。
2. 【强制】【hook】获取当前毫秒数 System.currentTimeMillis(); 而不是 new Date().getTime();

说明:如果想获取更加精确的纳秒级时间值，使用 System.nanoTime()的方式。在 JDK8 中，针对统计时间等场景，推荐使用 Instant 类。

1. 【推荐】不要在视图模板中加入任何复杂的逻辑。

说明:根据 MVC 理论，视图的职责是展示，不要抢模型和控制器的活。

1. 【推荐】任何数据结构的构造或初始化，都应指定大小，避免数据结构无限增长吃光内存。
2. 【强制】【hook】及时清理不再使用的代码段或配置信息。

说明:对于垃圾代码或过时配置，坚决清理干净，避免程序过度臃肿，代码冗余。

正例:对于暂时被注释掉，后续可能恢复使用的代码片断，在注释代码上方，统一规定使用三 个斜杠(///)来说明注释掉代码的理由。

1. 【强制】【hook】避免使用Apache Beanutils来复制属性。

# 配置管理

配置文件存放在单独的Git库，<http://git.mogo.com/pb/mogoroom-config>

配置提交流程为：

1．项目创建时ipublish会以test\_config分支为源，创建项目开发分支dev\_yyyyMMddxxx，开发阶段开发人员维护项目分支dev\_yyyyMMddxxx的内容，测试时直接由ipublish工具把项目分支的内容同步到测试环境.

2．测试通过后，ipublish进行版本合并dev\_yyyyMMddxxx--test\_config

3．生产配置请在iPublish上提交。后续将切换到apollo上。

开发测试分支结构为：



配置文件开发使用：

1. 配置文件使用方式

String newFileName = BaseMogoSystem.*getInstance*().getMybank\_localFileSaveRootPath()+"/"+bizDate+"/"+fileName;

不再使用配置文件中注入的方式，也不需要配置xml(以下方式的方式不再使用)

<!-- 系统配置文件 -->  
<bean id="mogoSystem" class="com.mogoroom.bs.system.MogoSystem">  
 <property name="environment" value="${sysconfig.environment}" />  
 <property name="phoneNum" value="${sysconfig.phoneNum}" />

</bean>

2. 配置文件项目使用与启动

配置文件统一采用外部分离的方式，配置文件使用方式为：

1. 从<http://git.mogo.com/pb/mogoroom-config> 拉取配置文件,放在本地
2. 修改项目中filter/sysconfig.properties对应的配置文件地址，比如：

filter.confFilePath =file:D:/mogoProject/mogoroom-config/renter/\*.properties

配置提交与注释规范

配置文件的提交

业务添加配置需要在[http://testgitlab.mogoroom.com/pb/mogoroom-config](http://testgitlab.mogoroom.com/pb/mogoroom-config中提交)中提交自己的配置到开发分支，如果需要集成测试时候需要提交到测试分支。

dev\_config分支中

--renter

-- systemconfig.properties

#[new@release5.1.0#](mailto:opt@release5.1.0)env：中行支付key

system\_config.pay.boc.key=1

# 事务规范

## **【强制】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

## 【推荐】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

## 【强制】不得使用事务超时时间

@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！

## 【强制】任何时候不要去改隔离级别

最好的事务添加方式，在需要加事物的方法上加 @Transactional(rollbackFor=Exception.class)

一定要加rollbackFor=Exception.class

## 【强制】只读接口不允许使用事务

只读接口不允许使用事务，更不得使用只读事务。

为了确保可重复读而使用只读事务，只能是自欺欺人的行为。其实这完全可以通过缓存来解决。

而且引入不必要的事务，会引起连接无法及时释放。

## 【强制】严禁在事务中访问第三方

严禁在事务中访问第三方。因为只要进入了Transactional，哪怕没有任何SQL语句，都会占用数据库连接。所以，一旦在事务中访问第三方，第三方的故障可能会导致我方的数据库连接被大量占用。会产生灾难性的结果。

## 【推荐】事务下沉

推荐事务下沉到service级别，不在facade层进行控制。

在服务化阶段这将是强制规则。届时，多个service将意味着多个jvm/db，facade层完全无法控制事务。

## 【推荐】事务粒度最小化

为了尽量少占用数据库连接，需要将事务粒度最小化。对于大多数场景/模块而言，可以不使用事务。只有账务等极少数场景是必须使用事务的。

# 日志规范(1.2)

**修订历史**

|  |  |  |  |
| --- | --- | --- | --- |
| 版本号 | 作者 | 内容提要 | 发布日期 |
| 1.0 | 九天 | 拟定 |  |
| 1.1 | 九天 | 增加监控日志规范 |  |
| 1.2 | 宋伟 | 增加错误编码规则 |  |
|  |  |  |  |

## 背景、概述及目标

|  |  |
| --- | --- |
| 背景基本介绍 | 一般来说日志分为三种：业务日志、监控日志和访问日志，使用日志我们希望能达到以下目标：对程序运行情况的记录和监控；在必要时可详细了解程序内部的运行状态；对系统性能的影响尽量小。统一蘑菇租房平台各业务线日志记录方式、格式；保证日志格式清晰，内容可追溯、分析；以便在异常、错误发生时能方便、快速地定位业务及系统异常并及时告警、通知相关人员处理，保证线上业务安全、正确、高效地运行。 |

1. 业务日志：按开发人员调试/查证要求，可随意添加的日志。
2. 监控日志：有架构组统一添加，用于查看系统调用性能、慢方法、慢SQL、空where、空limit、返回超过1000条等情况的日志，一般采用aop的方式实现；
3. 访问日志：用户进行统计分析、漏斗分析的日志。

业务日志和监控日志的前两个字段都为时间和requestId，后者可以参见5.5.3。日志长度：信息以简要为好，不应该输出过多信息【指每个参数的值应该控制在200字符以内】

## 基本原则

1. 日志输出不允许影响系统正常运行；
2. 日志输出不允许产生安全问题；
3. 日志不允许输出机密信息；
4. 日志可供开发人员定位问题的真正原因；
5. 日志可供监控系统自动监控与分析；
6. 日志使用符合安全审计要求。

## 规范方案

### 定义日志常量

日志变量往往不变，最好定义成final static（常量），名称大写。如：

|  |
| --- |
| static final Logger LOGGER = Logger.getLogger(XXX.class); |

蘑菇的日志工具类统一使用蘑菇自己封装的工具类做日志记录，

|  |
| --- |
| **import** com.mogoroom.core.util.log.Logger;  **import** com.mogoroom.core.util.log.LoggerFactory;  **private static final Logger** LOGGER **= LoggerFactory.getLogger(XXX.class)** |

### 日志分级

Java的日志框架一般会提供以下日志级别，缺省打开info级别，也就是debug，trace级别的日志在生产环境不会输出，在开发和测试环境可以通过不同的日志配置文件打开debug级别，上线后应该设置成info级别。

|  |  |  |
| --- | --- | --- |
| 序号 | 级别 | 说明 |
| 1 | fatal | 严重的，造成服务中断的错误； |
| 2 | error | 业务异常和其他运行期错误； |
| 3 | warn | 警告信息，如程序调用了一个即将作废的接口，接口的不当使用，运行状态不是期望的但仍可继续处理等； |
| 4 | info | 有意义的事件信息，如程序启动，关闭事件，收到请求事件等； |
| 5 | debug | 调试信息，可记录详细的业务处理到哪一步了，以及当前的变量状态； |
| 6 | trace | 更详细的跟踪信息； |

在程序内应该合理使用日志分级:



error/fatal级别的日志，是以需要向技术经理发送告警短信为标准的。我们暂不设置具体的数量要求，以技术经理的手机性能和能支付的短信费用为准。下面是几点说明：

1. 对于业务过程中的正常分支（产品经理能够画出来的），不允许打印error/fatal级别的日志。

比如房东买回的时候：

l 如果余额足够，则支付买回账单。

l 如果余额为0，则将房东设置为黑名单，不允许提现。

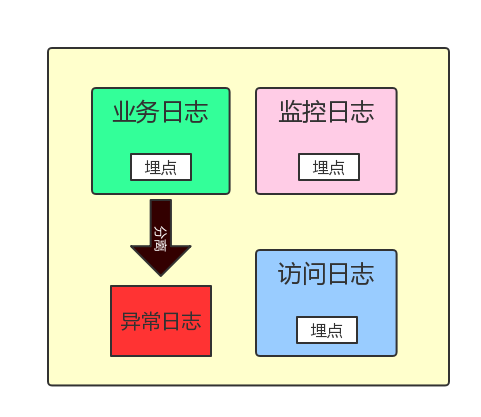
l 如果余额为负数，则告警（产品经理肯定不会画这个，因为这个是技术的严重bug）。

在上述过程中，余额>0或者=0都是正常的业务流转过程，不允许打印error/fatal级别的日志。而如果余额为负数，这是不可能出现的情况，如果开发人员想加强数据跟踪，可以打印error/fatal。

此外，关于到第三方/中间件的重连，一般可以在N次重连都失败之后，打印error/fatal。

2. error/fatal级别日志和异常没有必然关系。即使某些业务流转是通过抓取异常判断的（我们禁止这样写，不过老代码暂时可保留），也可以catch之后打印info日志。

### 日志类型



### 统一日志输出格式及分离错误日志

|  |
| --- |
| log4j. rootLogger=INFO, file, wFile  log4j.appender.file=org.apache.log4j.DailyRollingFileAppender  log4j.appender.file.DatePattern='\_'yyyy-MM-dd-a'.log'  log4j.appender.file.layout=org.apache.log4j.PatternLayout  log4j.appender.file.layout.ConversionPattern=%p %d{yyyy-MM-dd HH\:mm\:ss} %C.%M(%L) | %m %n |

分离出错误及告警日志：

|  |  |
| --- | --- |
| log4j.appender. wFile.Threshold=WARN  log4j.appender. wFile =org.apache.log4j.DailyRollingFileAppender  log4j.appender. wFile.File=/data/mogologs/mogoroom-error.log  log4j.appender. wFile.DatePattern='\_'yyyy-MM-dd-a'.log'  log4j.appender. wFile.layout=org.apache.log4j.PatternLayout  log4j.appender. wFile.layout.ConversionPattern=%p %d{yyyy-MM-dd HH\:mm\:ss} %C.%M(%L) | %m %n |  |

### 日志编码规范

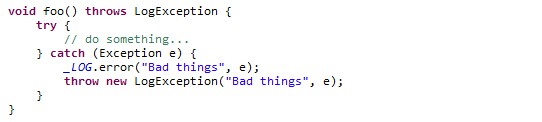
1. 在一个对象中通常只使用一个Logger对象，Logger应该是static final的，只有在少数需要在构造函数中传递logger的情况下才使用private final。

http://img.blog.csdn.net/20150314163709819

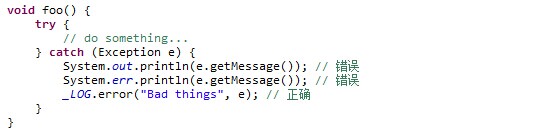
2.输出Exceptions的全部Throwable信息，因为logger.error(msg)和logger.error(msg,e.getMessage())这样的日志输出方法会丢失掉最重要的StackTrace信息。



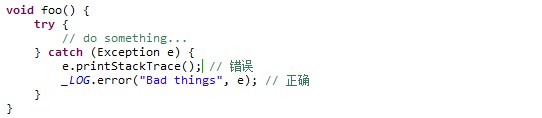
3.不允许记录日志后又抛出异常，因为这样会多次记录日志，只允许记录一次日志。



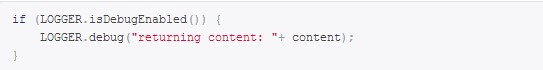
4.不允许出现System print(包括System.out.println和System.error.println)语句。



5.不允许出现printStackTrace。



6.日志性能的考虑，如果代码为核心代码，执行频率非常高，则输出日志建议增加判断，尤其是低级别的输出<debug、info、warn>。debug日志太多后可能会影响性能，有一种改进方法是：



但更好的方法是Slf4j提供的最佳实践:

http://img.blog.csdn.net/20150314163831580

一方面可以减少参数构造的开销，另一方面也不用多写两行代码。

7.有意义的日志

通常情况下在程序日志里记录一些比较有意义的状态数据：程序启动，退出的时间点；程序运行消耗时间；耗时程序的执行进度；重要变量的状态变化。初次之外，在公共的日志里规避打印程序的调试或者提示信息。

## 业务日志

### 关键词定义

|  |  |  |
| --- | --- | --- |
| 名称 | 定义 | 说明 |
| action | 行为[Warn, Error, Monitor, Get,Post,Delete] | 日志头部关键词，便于数据分类、定位、解析 |
| type | 类型 | 类似key，区分不同类型的指标数据 |
| time | 时间，单位：毫秒 | 日志输出的时间 |
| module | 业务模块 | 微服务后对应的业务模块 |
| bizType | 业务类型 | 参考CommBusiTypeEnum类 |
| params | 方法参数，输入参数 |  |
| msg | 日志内容 |  |
| errCode | 异常编码 |  |
| errMsg | 异常信息说明 |  |

### 业务日志格式

在调用Façade、以及各业务的服务接口时，在其所调用的方法中，需在方法调用前、调用返回、执行出错时，记录日志信息， 其具体日志信息需包括:

1. 业务模块(module)
2. 业务编码(bizType)
3. 参数信息(params)
4. 异常编码，异常信息
5. 告警信息，其它附加信息等

【强制】业务错误日志级别统一使用waring 级别，对业务产生阻塞的使用error级别，方便在日志系统中做监控

日志格式为：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| action | module | bizType | time | params | msg | [errCode-errMsg] |

信息记录：

|  |
| --- |
| |Info|${module}|${bizType}|${time}|${params}|${msg}|[${errCode-errMsg}] |

运行告警：

|  |
| --- |
| |Warn|${module}|${bizType}|${time}|${params}|${msg}|[${errCode-errMsg}] |

运行错误：

|  |
| --- |
| |Error|${module}|${bizType}|${time}|${params}|${msg}|${errCode-errMsg} |

### 示例代码

|  |
| --- |
| public ResultContext confirmBill(ServiceContext serviceContext) {  ResultContext resultContext = null;  try {  resultContext = billServiceImpl.confirmBill(serviceContext);  } catch (MogoException e){  // 记录业务异常信息  logger.error(”|Error|${module}|${bizType}|${time}|${params}|${e.getMessage()}|${errCode-errMsg}”);  }  } |

注：(1) 具体日志记录方式请参考以上代码标红处。

(2) 另模块编码与业务编码请参考BusiTypeEnum类。

(3) 如参数中存在敏感信息，则不可在日志中记录。

## 监控日志

全局统一名称：com.mogoroom.util.log.MonitorLogger

|  |
| --- |
| # Custom monitor logs  log4j.logger.com.mogoroom.util.log.MonitorLogger=info,monitor  log4j.appender.monitor=org.apache.log4j.DailyRollingFileAppender  log4j.appender.monitor.DatePattern='\_'yyyy-MM-dd-a'.log'  log4j.appender.monitor.layout=org.apache.log4j.PatternLayout  log4j.appender.monitor.File=/data/mogologs/mogoroom-monitor.log  log4j.appender.monitor.layout.ConversionPattern=%p %d{yyyy-MM-dd HH\:mm\:ss} %C.%M(%L) | %m %n |

|  |  |  |
| --- | --- | --- |
| 指标 | 说明 | 收集途径 |
| PV | Page View | 日志埋点，独立页URL访问日志 |
| UV | Unique Visitor | 日志埋点，Token数量 |
| TPS | Transactions Per Second | 日志埋点，每秒URL访问量 |
| App在线租客 | App在线活跃租客数量 | 日志埋点，type：statAppOnlineRenter |
| Web在线租客 | Web在线活跃租客数量 | 日志埋点，type：statWebOnlineRenter |
| App在线房东 | App在线活跃房东数量 | 日志埋点，type：statAppOnlineLandlord |
| Web在线房东 | Web在线活跃房东数量 | 日志埋点，type：statWebOnlineLandlord |
| 发房量 | 新发布房源数量 | 日志埋点，type：statIssueRoom |
| 慢方法 | 执行时长超过3000毫秒的方法 | 日志埋点，type：slowMethod |
| 慢SQL | 执行时长超过3000毫秒的方法 | 日志埋点，type：slowSql |
| 异常数 | 检查型异常和非检查型异常出现的总次数 | 日志埋点，统计 |

### 监控指标

### 监控日志格式

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| action | type | module | channel | time | value | msg |

|Monitor|${type}|${module}|${channel}|${time}|${value}|${msg}

### 监控日志实现说明

1.慢方法

mogoroom-util/com.mogoroom.util.minitor.InvokeAspect拦截了从controller、facade、service、dao所有层与层之间的方法调用。

2.慢SQL

mogoroom-util/com.mogoroom.page.PagePlugin 拦截了所有的sql执行记录sql的执行时间。 监控字段说明：

monitor|${type}|${module}|${channel}|${time}|${value}|${msg}

monitor|监控类型|模块|渠道|时间戳|监控值|监控描述|

RequestId使用mogoroom-appcommon/com.mogoroom.appcommon.intercepter/

从nginx获取requestid然后绑定到当前系统ThreadLocal，后续直接获取，

3.慢索引监控

慢索引通过explain 分析sql执行计划记录到运维系统分析，埋点：mogoroom-util/com.mogoroom.util.minitor.IndexMonitor

日志记录地址：sysconfig.logMinuteIndexUrl=埋点服务器地址

4.【强制】监控日志中，必须输出每个方法的耗时。

## 访问日志

### 访问日志格式

当用户发起请求时，需记录相关请求信息，其包括：

1. 访问URL
2. 客户端IP地址
3. userId
4. 用户类型
5. 版本信息

访问日志格式为：

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| action | time | IP | module | channel | userId | userType | URL | duration | appVersion |

Get请求：

|  |
| --- |
| |Get|${time}|${IP}|${module}|${channel}|${userId}|${userType}|${URL}|${duration}|${appVersion} |

Post请求：

|  |
| --- |
| |Post|${time}|${IP}|${module}|${channel}|${userId}|${userType}|${URL}|${duration}|${appVersion} |

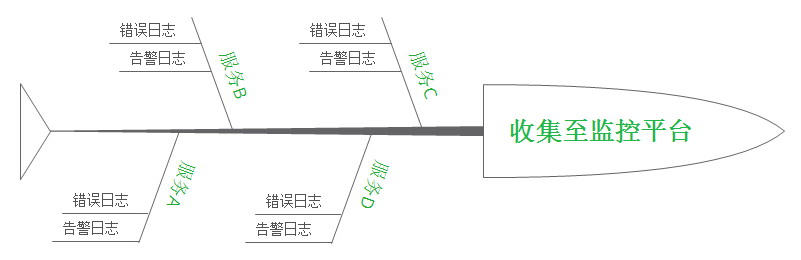
Delete请求：

|  |
| --- |
| |Delete|${time}|${IP}|${module}|${channel}|${userId}|${userType}|${URL}|${duration}|${appVersion} |

Export请求：

|  |
| --- |
| |Export|${time}|${IP}|${module}|${channel}|${userId}|${userType}|${URL}|${duration}|${appVersion} |

## 日志收集



## 日志规约

1. 【强制】应用中不可直接使用日志系统(Log4j、Logback)中的 API，而应依赖使用日志框架SLF4J 中的 API，使用门面模式的日志框架，有利于维护和各个类的日志处理方式统一。

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

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

1. 【强制】日志文件至少保存 15 天，因为有些异常具备以“周”为频次发生的特点。
2. 【强制】应用中的扩展日志(如打点、临时监控、访问日志等)命名方式: appName\_logType\_logName.log。logType:日志类型，推荐分类有 stats/desc/monitor/visit 等;logName:日志 述。这种命名的好处:通过文件名就可知 道日志文件属于什么应用，什么类型，什么目的，也有利于归类查找。

正例:mppserver 应用中单独监控时区转换异常，如:

mppserver\_monitor\_timeZoneConvert.log

说明:推荐对日志进行分类，如将错误日志和业务日志分开存放，便于开发人员查看，也便于 通过日志对系统进行及时监控。

1. 【强制】对 trace/debug/info 级别的日志输出，必须使用条件输出形式或者使用占位符的方式。

说明:logger.debug("Processing trade with id: " + id + " and symbol: " + symbol); 如果日志级别是 warn，上述日志不会打印，但是会执行字符串拼接操作，如果 symbol 是对象， 会执行 toString()方法，浪费了系统资源，执行了上述操作，最终日志却没有打印。

正例:(占位符，需要依赖slf4j)

logger.debug("Processing trade with id: {} and symbol : {} ", id, symbol);

1. 【强制】避免重复打印日志，浪费磁盘空间，务必在 log4j.xml，log4j.properties中设置 additivity=false。生产禁止开启debug级别日志。

正例:**<logger name="com.mgzf.bs.config" additivity="false">**

1. 【强制】异常信息应该包括两类信息:案发现场信息和异常堆栈信息。如果不处理，那么通过 关键字 throws 往上抛出。

正例:**logger.error(“消息模板”，各类参数或者对象 toString ， e.getMessage(), e);**

1. 【推荐】谨慎地记录日志。

生产环境禁止输出 debug 日志;

有选择地输出 info 日志;

如果使用 warn 来记录刚上线时的业务行为信息，一定要注意日志输出量的问题，避免把服务器磁盘 撑爆，并记得及时删除这些观察日志。

说明:大量地输出无效日志，不利于系统性能 升，也不利于快速定位错误点。记录日志时请

思考:这些日志真的有人看吗?看到这条日志你能做什么?能不能给问题排查带来好处?

1. 【参考】可以使用 warn 日志级别来记录用户输入参数错误的情况，避免用户投诉时，无所适从。注意日志输出的级别，error 级别只记录系统逻辑出错、异常等重要的错误信息。如非必要，请不要在此场景打出 error 级别。

# 分页使用规范

代码框架提供了默认的分页支持，通过分页插件来实现自动分页，分页插件是为了简单查询提高开发效率采用的方式，复杂查询、慢查询，需要自己优化count 查询，不要用统一的分页插件。【需要更新分页插件的使用方式】

## 分页查询SQL改写原则为

1. 普通查询（无group by聚合）原始sql自动改写为count。

*例如：*

*Select id,name,address from room where type=1;*

*改写为：*

*数据sql：Select id,name,address from room where type=1 limit start,limit;*

*总数sql：select count(\*) from Select id,name,address from room where type=1;*

1. 复杂查询（2次聚合）删除内部类枚举，然后外包select count（\*）

例如：

*Select type,count(id) from room where type in (1,2,3) group by type;*

*改写为：*

*数据sql：Select type,count(id) from room where type in (1,2,3) group by type start,limit;*

*总数sql：select count(\*) from （*

*Select 1 from room where type in (1,2,3) group by type;*

*）*

## 分页插件场景

1. 简单查询，使用分页插件提高开发效率。

## 不推荐使用分页插件的地方

1. 二次聚合。
2. 报表统计类查询。

## 不推荐显示分页总数的场景

1. 二次聚合性质的报表统计类查询。

例如：

Select count(a.id),b.type,c.name city

From room a,room\_detail b,city c

Where a.id=b.room\_id

And a.city\_id=c.id

Group by c.name,b.type

这类查询不建议显示总数，建议只简单显示“上一页”、“下一页”即可。

## 分页插件使用

public class Page<T> implements Serializable {

private int showCount = 20; // 每页显示记录数

private int totalPage; // 总页数

private int totalResult; // 总记录数

private int currentPage; // 当前页, 从1开始

private int currentResult; // 当前记录起始索引, 从0开始

private boolean needTotal = true;//分页时候需要总页数

private String pageCountSql = "";//分页查询总数sql，外部传入，如果不传，自动生成count sql

private boolean selfCountSql = false;//分页查询总数sql，外部传入，如果不传，自动生成count sql

}

1. 如果采用默认的框架自动分页，mybatise mapper 名字需要采用 \*selectListPage\* 关键字，采用此关键字命名的会自动走分页规则。
2. 如果不想自动生成count语句，需要外部自己count语句，selfCountSql 设置为true，

同时需要在mapper的xml文件中有一个与数据mapperid匹配的语句，count 语句规则为业务mapperstateid+“\_count”，插件会自动找到自定义count执行。

例如：

*<select id="selectListPageRegRenterDetailByParams\_count" resultType="java.lang.Integer" >*

*<select id=" selectListPageRegRenterDetailByParams " resultType="com.mogoroom.service.repo.vo.RenterChannelDetail" >*

1. 如果不想每次都计算count，需要设置*needTotal 为false；*

# 业务编码规范

1.业务记录的记录范围为，一切操作（读、写、修改、删除）\*\*业务需要做记录\*\*的都要做日志记录。这个表主要服务于业务，服务于最终用户。如果用户需要查看，此处就需要记录。反之则不必，可记录于各his表中。极少数敏感信息的查询，如果需要查询，也可记录于此。

2.记录的业务编码为统一分配，如果新增业务需要申请添加业务编码。

3.业务编码与业务自己的历史记录表需要收集，尽可能的收集完全，形成业务编码与业务日志应收关系表，后续业务新增业务历史表重新分配业务编码与历史表的关系映射。

4.BS查询时候基础服务组提供记录查询接口：

1.默认显示业务变更主记录，查询comm\_business\_record 业务变更记录，可以根据业务编码及房东，租客公寓等其他业务条件的查询。

2.当选择某一个业务变更记录的时候可以查询具体的业务变更明细。

1. 本表适用于蘑菇租房的主体业务群（租房/金融/子账号等）。新的业务群如VIC系统，需单独记录自己的业务记录表。

具体使用方法参见：

<http://192.168.60.125:8080/dokuwiki/doku.php?id=业务非读操作记录规范>

# 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。

具体使用方法参见：

<http://192.168.60.125:8080/dokuwiki/doku.php?id=分布式锁>}

# 数据重复提交推荐方案

1. 表单重复提交

设计：

在每次controller或者RESTfull接口访问时候，在当前session或者token中，生成一个随机的时间戳数，然后前端访问当前读对应的更新或者写接口的时候带回后端生成的时间戳，调用完毕后重新刷新界面或者接口获取时间戳；后端判断同一个接口的的重复性，如果同一个接口多次调用时间戳不一致，认为表单或接口重复提交。

后端使用：

@TokenGen  
**public** String tokenInit(HttpServletRequest request) **throws** Exception{  
  
 request.setAttribute(**"time"**,**commonFacadeImpl**.now());  
  
 **return "/pages/tool/time"**;  
}  
  
@RequestMapping(value=**"/token/check"**)  
@TokenValid  
**public** String tokenCheck(HttpServletRequest request) **throws** Exception{  
   
 request.setAttribute(**"time"**,**commonFacadeImpl**.now());

//如果前端想减少获取token的交互次数，需要自己设置token

String formUuid = request.getAttribute(**"formUuid"**).toString();

request.setAttribute(**"formUuid** **"**, formUuid);  
   
 **return "/pages/tool/time"**;  
}

如上面例子，在一个接口初始化的时候，添加校验token的注解，然后在需要做重复校验的接口添加校验注释，如果框架判断重复会报错，请求无法处理。

前端使用：

* + - 1. 前端在一个界面打开的时候，设置一个隐藏域保存token，例如：

<input type="hidden" name=" **formUuid** " value="${ **formUuid** }">

* + - 1. 在请求的时候再把参数带回：例如

Xxx?formUuid=e3efa635-2af4-487c-9c51-3d35b3e7675b

1. 高并发排队

如果是存在线程不安全，需要做并发控制的方法，需要加java的锁同步synchronized，如果涉及分布式调用的需要加分布式锁zookeeper锁，并且由于数据库隔离级别默认为可重复读，需要将锁加载事务外部，禁止在事务内部加锁。

实现：框架提供统一zookeeper锁机制,具体查询锁的使用。

1. 数据版本号

 涉及金额或者重要数据的表，需要对记录加业务版本号，添加方式为，在业务表比如余额表添加版本号字段version，默认值为1，如果当期记录做了修改，需要更新version版本字段为累加，即：version=version+1；如果不同的业务端，或者由于重复提交获取到相同的version，去updae数据，这时候，程序判断版本一致，认为非法操作。

   例如：

      select xxx,version from account where id=1;

      结果为：id=1,version=1,balance=100;

     执行：

         a. 事务外部（由于数据库可重复读，查询必须在事务外部）

          select xxx,version from account where id=1 and version=1;

          如果没有查询到数据认为数据版本不一致被修改。

        b. update account set balance=100+1 where id=100 and version=1 and balance=100;

         如果jdbc 返回影响的行数为0，认为version改变，重复操作或者数据被第修改。

或者：

update account set balance=100+1 where id=100 and version=1 and balance=100 and version=1; ;

         如果jdbc 返回影响的行数为0，认为version改变，重复操作或者数据被第修改。

实现：业务表添加版本字段，自己控制

# 读写分离

手工方式：

在需要读写分离的上层，一般是dao层（禁止加在controller及façade切换），比如dao的入口：

import com.mogoroom.facade.system.DataSourceContextHolder

DataSourceContextHolder.setDbType(balance.balance());

业务操作xxx

DataSourceContextHolder.setDbType. clearDbType();

注释方式：

@ReadBalance(balance=**"ds\_se"**,desc=**"从库"**)  
@Transactional  
**public** String getUserInfo(HttpServletRequest request) **throws** Exception{  
 业务操作xxx  
 **return "xxx"**;  
}

# 异常规范

## 异常处理

1. 【强制】Java 类库中定义的一类 RuntimeException 可以通过预先检查进行规避，而不应该 通过catch 来处理，比如:IndexOutOfBoundsException，NullPointerException等等。 说明:无法通过预检查的异常除外，如在解析一个外部传来的字符串形式数字时，通过 catch NumberFormatException 来实现。

正例:if (obj != null) {...}

反例:try { obj.method() } catch (NullPointerException e) {...}

1. 【强制】异常不要用来做流程控制，条件控制，因为异常的处理效率比条件分支低。
2. 【强制】对大段代码进行 try-catch，这是不负责任的表现。catch 时请分清稳定代码和非稳 定代码，稳定代码指的是无论如何不会出错的代码。对于非稳定代码的catch尽可能进行区分 异常类型，再做对应的异常处理。
3. 【强制】捕获异常是为了处理它，不要捕获了却什么都不处理而抛弃之，如果不想处理它，请将该异常抛给它的调用者。最外层的业务使用者，必须处理异常，将其转化为用户可以理解的内容。
4. 【强制】【hook】有 try 块放到了事务代码中，catch 异常后，如果需要回滚事务，一定要注意手动回滚事务。
5. 【强制】finally 块必须对资源对象、流对象进行关闭，有异常也要做 try-catch。

说明:如果 JDK7 及以上，可以使用 try-with-resources 方式。

1. 【强制】【hook】不能在 finally 块中使用 return，finally 块中的 return 返回后方法结束执行，不 会再执行 try 块中的 return 语句。
2. 【强制】捕获异常与抛异常，必须是完全匹配，或者捕获异常是抛异常的父类。

说明:如果预期对方抛的是绣球，实际接到的是铅球，就会产生意外情况。

1. 【推荐】方法的返回值可以为 null，不强制返回空集合，或者空对象等，必须添加注释充分 说明什么情况下会返回 null 值。调用方需要进行 null 判断防止 NPE 问题。

说明:本手册明确防止 NPE 是调用者的责任。即使被调用方法返回空集合或者空对象，对调用者来说，也并非高枕无忧，必须考虑到远程调用失败、序列化失败、运行时异常等场景返回 null 的情况。

1. 【推荐】【hook】防止 NPE，是程序员的基本修养，注意 NPE 产生的场景:
2. 返回类型为基本数据类型，return 包装数据类型的对象时，自动拆箱有可能产生 NPE。

反例:public int f() { return Integer对象}， 如果为null，自动解箱抛NPE。

1. 数据库的查询结果可能为null。
2. 集合里的元素即使isNotEmpty，取出的数据元素也可能为null。
3. 远程调用返回对象时，一律要求进行空指针判断，防止NPE。
4. 对于Session中获取的数据，建议NPE检查，避免空指针。
5. 级联调用obj.getA().getB().getC();一连串调用，易产生NPE。

正例:使用 JDK8 的 Optional 类来防止 NPE 问题。

1. 【推荐】定义时区分unchecked/checked 异常，避免直接抛出new RuntimeException()， 更不允许抛出 Exception 或者 Throwable，应使用有业务含义的自定义异常。推荐业界已定义过的自定义异常，如:DAOException / ServiceException 等。
2. 【参考】在代码中使用“抛异常”还是“返回错误码”，对于公司外的 http/api 开放接口必须 使用“错误码”;而应用内部推荐异常抛出;跨应用间 RPC 调用优先考虑使用 Result 方式，封 装 isSuccess()方法、“错误码”、“错误简短信息”。

说明:关于 RPC 方法返回方式使用 Result 方式的理由:

1. 使用抛异常返回方式，调用方如果没有捕获到就会产生运行时错误。
2. 如果不加栈信息，只是new自定义异常，加入自己的理解的error message，对于调用 端解决问题的帮助不会太多。如果加了栈信息，在频繁调用出错的情况下，数据序列化和传输 的性能损耗也是问题。
3. 【参考】避免出现重复的代码(Don’t Repeat Yourself)，即DRY原则。

说明:随意复制和粘贴代码，必然会导致代码的重复，在以后需要修改时，需要修改所有的副本，容易遗漏。必要时抽取共性方法，或者抽象公共类，甚至是组件化。

正例:一个类中有多个public方法，都需要进行数行相同的参数校验操作，这个时候请抽取: private boolean checkParam(DTO dto) {...}

1. 【参考】 分层异常处理规约

在 DAO 层，产生的异常类型有很多，无法用细粒度的异常进行catch，使用catch(Exception e)方式，并throw new DAOException(e)，不需要打印日志，因为日志在 Facade/Service 层一定需要捕获并打到日志文件中去，如果同台服务器 再打日志，浪费性能和存储。

在 Service 层出现异常时，必须记录出错日志到磁盘，尽可能带 上参数信息，相当于保护案发现场。

Web 层绝不应该继续往上抛异常，因为已经处于顶层，如果意识到这个异常将导致页面无法正常渲染，那么就应该直接跳转到友好错误页面，加上用户容易理解的错误示信息。开放接口层要将异常处理成错误码 和错误信息方式返回。

1. 【强制】对于被调用方，应该尽量少返回异常，确保业务流程正常运行。比如获取缓存失败，只需要返回null即可。如果缓存宕机，可以采用探针方式发现。
2. 【强制】明确需要调用方处理的，必须声明为checked Exception。比如无法打开配置文件，并且默认配置不可用。Checked Exception应当只在异常情况对于API以及API的使用者都无法避免的情况下被使用。一个Checked Exception应该有明确的意义。我们应该最近化地对Checked Exception进行处理。这是因为随着Checked Exception沿着函数调用的轨迹向上传递的过程中，这些被抛出的Checked Exception的意义将逐渐模糊。
3. 【参考】尽量少抛异常，如是手机号码不存在，返回null，在返回值中进行说明，然后客户端处理这个逻辑，而不需要返回异常。能用return 返回值处理的，尽量用返回值，返回值复杂或者碰到业务处理复杂的可以抛出异常。
4. 【强制】方法接口禁止throw exception。
5. 【推荐】大部分情况下不建议在循环中进行异常处理，应该在循环外对异常进行捕获处理。
6. 【参考】为你的异常生成足够的文档说明，至少是 JavaDoc。为每个异常消息定义一个数值，这对好的文档来说是非常重要的，常见的做法就是使用枚举类来管理错误码。

## 异常类说明

异常的抛出统一抛出继承自MogoException的子类，禁止使用断言及直接抛出Exception, Assert.notNull(criteria);

异常基础类：MogoException

使用方式，业务组在某个模块或者功能编写时，自己编写自己的业务异常类，继承以上异常，同时异常需要返回异常的编码。

异常统一框架会在Web端对异常进行默认的拦截处理。

实例变量及方法参数验证统一采用JSR 303（bean validation）标准进行校验处理，aop统一处理throw MogoException 子类 ParamException

方法内部业务逻辑异常统一throw MogoException 子类 （Checked Exception）

## 错误编码规则

业务编码 + 错误编码，说明：

业务编码 ，参考业务编码规则。

http://192.168.60.125:8080/dokuwiki/doku.php?id=异常错误编码规范

错误编码：3位数字

# 单元测试

【强制】好的单元测试必须遵守 AIR 原则。 说明:单元测试在线上运行时，感觉像空气(AIR)一样并不存在，但在测试质量的保障上，却是非常关键的。好的单元测试宏观上来说，具有自动化、独立性、可重复执行的特点。

A:Automatic(自动化)

I:Independent(独立性)

R:Repeatable(可重复)

【强制】单元测试应该是全自动执行的，并且非交互式的。测试框架通常是定期执行的，执行过程必须完全自动化才有意义。输出结果需要人工检查的测试不是一个好的单元测试。单元测 试中不准使用 System.out 来进行人肉验证，必须使用 assert 来验证。

【强制】保持单元测试的独立性。为了保证单元测试稳定可靠且便于维护，单元测试用例之间 决不能互相调用，也不能依赖执行的先后次序。

反例:method2 需要依赖 method1 的执行，将执行结果做为 method2 的输入。

【强制】单元测试是可以重复执行的，不能受到外界环境的影响。 说明:单元测试通常会被放到持续集成中，每次有代码 check in 时单元测试都会被执行。如 果单测对外部环境(网络、服务、中间件等)有依赖，容易导致持续集成机制的不可用。

正例:为了不受外界环境影响，要求设计代码时就把 SUT 的依赖改成注入，在测试时用 spring 这样的 DI 框架注入一个本地(内存)实现或者 Mock 实现。

【强制】对于单元测试，要保证测试粒度足够小，有助于精确定位问题。单测粒度至多是类级 别，一般是方法级别。 说明:只有测试粒度小才能在出错时尽快定位到出错位置。单测不负责检查跨类或者跨系统的 交互逻辑，那是集成测试的领域。

【强制】核心业务、核心应用、核心模块的增量代码确保单元测试通过。

说明:新增代码及时补充单元测试，如果新增代码影响了原有单元测试，请及时修正。

【强制】单元测试代码必须写在如下工程目录:src/test/java，不允许写在业务代码目录下。

说明:源码构建时会跳过此目录，而单元测试框架默认是扫 此目录。

【推荐】单元测试的基本目标:语句覆盖率达到 70%;核心模块的语句覆盖率和分支覆盖率都 要达到 100%

说明:在工程规约的应用分层中 到的 DAO 层，Manager 层，可重用度高的 Service，都应该 进行单元测试。

【推荐】编写单元测试代码遵守 BCDE 原则，以保证被测试模块的交付质量。

B:Border，边界值测试，包括循环边界、特殊取值、特殊时间点、数据顺序等。

C:Correct，正确的输入，并得到预期的结果。

D:Design，与设计文档相结合，来编写单元测试。

E:Error，强制错误信息输入(如:非法数据、异常流程、非业务允许输入等)，并得 到预期的结果。

【推荐】对于数据库相关的查询，更新，删除等操作，不能假设数据库里的数据是存在的， 或者直接操作数据库把数据插入进去，请使用程序插入或者导入数据的方式来准备数据。

反例:删除某一行数据的单元测试，在数据库中，先直接手动增加一行作为删除目标，但是这 一行新增数据并不符合业务插入规则，导致测试结果异常。

【推荐】和数据库相关的单元测试，可以设定自动回滚机制，不给数据库造成脏数据。或者 对单元测试产生的数据有明确的前后缀标识。

正例:在 RDC 内部单元测试中，使用 RDC\_UNIT\_TEST\_的前缀标识数据。

【推荐】对于不可测的代码建议做必要的重构，使代码变得可测，避免为了达到测试要求而 书写不规范测试代码。

【推荐】在设计评审阶段，开发人员需要和测试人员一起确定单元测试范围，单元测试最好 覆盖所有测试用例(UC)。

【推荐】单元测试作为一种质量保障手段，不建议项目发布后补充单元测试用例，建议在项 目 测前完成单元测试。

【参考】为了更方便地进行单元测试，业务代码应避免以下情况:

构造方法中做的事情过多。

存在过多的全局变量和静态方法。

存在过多的外部依赖。

存在过多的条件语句。

说明:多层条件语句建议使用卫语句、策略模式、状态模式等方式重构。

【参考】不要对单元测试存在如下误解:

那是测试同学干的事情。本文是开发手册，凡是本文内容都是与开发同学强相关的。

单元测试代码是多余的。汽车的整体功能与各单元部件的测试正常与否是强相关的。

单元测试代码不需要维护。一年半载后，那么单元测试几乎处于废弃状态。

单元测试与线上故障没有辩证关系。好的单元测试能够最大限度地规避线上故障。

【推荐】facade&service内的代码全部采用mock机制，dao层采用junit机制

单元测试管理方式

* 范围：新代码库（后端代码）
* UT覆盖率：facade&service，第一阶段2018-04-01起效40%
* 在提测阶段检查，不达标不能申请测试环境
* 技术经理抽查UT代码/每周必须一次
* UT代码规则：http://git.mogo.com/tech/doc/blob/master/spec/java/UT如何写.md

# 安全规约

1. 【强制】隶属于用户个人的页面或者功能必须进行权限控制校验。 说明:防止没有做水平权限校验就可随意访问、修改、删除别人的数据，比如查看他人的私信 内容、修改他人的订单。
2. 【强制】用户敏感数据禁止直接展示，必须对展示数据进行脱敏。 说明:查看个人手机号码会显示成:158\*\*\*\*9119，隐藏中间 4 位，防止隐私泄露。
3. 【强制】用户输入的 SQL 参数严格使用参数绑定或者 METADATA 字段值限定，防止 SQL 注入， 禁止字符串拼接 SQL 访问数据库。
4. 【强制】用户请求传入的任何参数必须做有效性验证。 说明:忽略参数校验可能导致:
   1. page size过大导致内存溢出
   2. 恶意order by导致数据库慢查询
   3. 任意重定向
   4. SQL注入
   5. 反序列化注入
   6. 正则输入源串拒绝服务ReDoS

说明:Java 代码用正则来验证客户端的输入，有些正则写法验证普通用户输入没有问题， 但是如果攻击人员使用的是特殊构造的字符串来验证，有可能导致死循环的结果。

1. 【强制】禁止向 HTML 页面输出未经安全过滤或未正确转义的用户数据。
2. 【强制】表单、AJAX 交必须执行 CSRF 安全过滤。

说明:CSRF(Cross-site request forgery)跨站请求伪造是一类常见编程漏洞。对于存在 CSRF 漏洞的应用/网站，攻击者可以事先构造好 URL，只要受害者用户一访问，后台便在用户 不知情情况下对数据库中用户参数进行相应修改。

1. 【强制】在使用平台资源，譬如短信、邮件、电话、下单、支付，必须实现正确的防重放限制， 如数量限制、疲劳度控制、验证码校验，避免被滥刷、资损。 说明:如注册时发送验证码到手机，如果没有限制次数和频率，那么可以利用此功能骚扰到其 它用户，并造成短信平台资源浪费。
2. 【推荐】发贴、评论、发送即时消息等用户生成内容的场景必须实现防刷、文本内容违禁词过 滤等风控策略

# 服务器

* + - 1. 【推荐】高并发服务器建议调小 TCP 协议的 time\_wait 超时时间。

说明:操作系统默认 240 秒后，才会关闭处于 time\_wait 状态的连接，在高并发访问下，服 务器端会因为处于 time\_wait 的连接数太多，可能无法建立新的连接，所以需要在服务器上 调小此等待值。

正例:在 linux 服务器上请通过变更/etc/sysctl.conf 文件去修改该缺省值(秒):

net.ipv4.tcp\_fin\_timeout = 30

* + - 1. 【推荐】调大服务器所支持的最大文件句柄数(File Descriptor，简写为fd)。

说明:主流操作系统的设计是将 TCP/UDP 连接采用与文件一样的方式去管理，即一个连接对 应于一个 fd。主流的 linux 服务器默认所支持最大 fd 数量为 1024，当并发连接数很大时很 容易因为 fd 不足而出现“open too many files”错误，导致新的连接无法建立。 建议将 linux 服务器所支持的最大句柄数调高数倍(与服务器的内存数量相关)。

* + - 1. 【推荐】给 JVM 设置-XX:+HeapDumpOnOutOfMemoryError 参数，让 JVM 碰到 OOM 场景时输出 dump 信息。

说明:OOM 的发生是有概率的，甚至有规律地相隔数月才出现一例，出现时的现场信息对查错 非常有价值。

* + - 1. 【推荐】在线上生产环境，JVM的Xms和Xmx设置一样大小的内存容量，避免在GC 后调整堆 大小带来的压力。
      2. 【参考】服务器内部重定向使用 forward;外部重定向地址使用 URL 拼装工具类来生成，否则 会带来 URL 维护不一致的问题和潜在的安全风险

# SONAR代码规范解读

为了减少阻断和严重类型问题代码所产生的BUG，提高研发人员上传代码的质量。规范将同步加入SONAR平台检测，在git提交时会进行代码扫描。

规范列举了常见的容易忽略的错误，编码中尚未意识到的错误,未来存在的隐患。部分给出了示例代码和原因，详细解释可根据编号在SONAR平台查找。

规范中条目不仅限于当前数量，会随时添加和补充。请在提交代码前熟读规范，务必严格遵循。

SONAR的代码规范的建议一般都有**Noncompliant、Compliant，也就是反例、正例的意思。**

## 流资源打开后必须关闭。防止内存泄露

编号：S2095

循环中打开多个流，但是最终只关闭了一个。

**Noncompliant Code Example**

OutputStream stream = null;

try{

for (String property : propertyList) {

stream = new FileOutputStream("myfile.txt"); // Noncompliant

// ...

}

}catch(Exception e){

// ...

}finally{

stream.close(); // Multiple streams were opened. Only the last is closed.

}

**Compliant Solution**

OutputStream stream = null;

try{

stream = new FileOutputStream("myfile.txt");

for (String property : propertyList) {

// ...

}

}catch(Exception e){

// ...

}finally{

stream.close();

}

## 枚举属性值必须是私有的，且不可变更的

编号：S3066

**Noncompliant Code Example**

public enum Continent {

NORTH\_AMERICA (23, 24709000),

// ...

EUROPE (50, 39310000);

public int countryCount; // Noncompliant

private int landMass;

Continent(int countryCount, int landMass) {

// ...

}

public void setLandMass(int landMass) { // Noncompliant

this.landMass = landMass;

}

**Compliant Solution**

public enum Continent {

NORTH\_AMERICA (23, 24709000),

// ...

EUROPE (50, 39310000);

private int countryCount;

private int landMass;

Continent(int countryCount, int landMass) {

// ...

}

## 死循环，不允许出现

编号：S2189

**Noncompliant Code Example**

for (;;) { // Noncompliant; end condition omitted

// ...

}

for (int i = 0; i < 10; i--) { // Noncompliant; end condition but unreachedable

//...

}

int j;

while (true) { // Noncompliant; end condition omitted

j++;

}

int k;

boolean b = true;

while (b) { // Noncompliant; b never written to in loop

k++;

}

**Compliant Solution**

for (int i = 0; i < 10; i++) { // end condition now reachable

//...

}

int j;

while (true) { // reachable end condition added

j++;

if (j == Integer.MIN\_VALUE) { // true at Integer.MAX\_VALUE +1

break;

}

}

int k;

boolean b = true;

while (b) {

k++;

b = k < Integer.MAX\_VALUE;

}

## 调用replaceAll方法 参数需要注意

编号：S2639。

**Noncompliant Code Example**

String str = "/File|Name.txt";

String clean = str.replaceAll(".",""); // Noncompliant; probably meant to remove only dot chars, but returns an empty string

String clean2 = str.replaceAll("|","\_"); // Noncompliant; yields \_/\_F\_i\_l\_e\_|\_N\_a\_m\_e\_.\_t\_x\_t\_

String clean3 = str.replaceAll(File.separator,""); // Noncompliant; exception on Windows

## 线程遇到异常，必须抛出异常或者终止线程

编号：S2142

**Noncompliant Code Example**

public void run () {

try {

while (true) {

// do stuff

}

}catch (InterruptedException e) { // Noncompliant; logging is not enough

LOGGER.log(Level.WARN, "Interrupted!", e);

}

}

**Compliant Solution**

public void run () throws InterruptedException{

try {

while (true) {

// do stuff

}

}catch (InterruptedException e) {

LOGGER.log(Level.WARN, "Interrupted!", e);

// clean up state...

throw e;

}

}

**or**

public void run () {

try {

while (true) {

// do stuff

}

}catch (InterruptedException e) {

LOGGER.log(Level.WARN, "Interrupted!", e);

// clean up state...

Thread.currentThread().interrupt();

}

}

## 不能使用MD5 和 SHA-1 ,因为已经变的不安全了

编号：S2070

The MD5 algorithm and its successor, SHA-1, are no longer considered secure, because it is too easy to create hash collisions with them

**Noncompliant Code Example**

MessageDigest md = MessageDigest.getInstance("SHA1"); // Noncompliant

**Compliant Solution**

MessageDigest md = MessageDigest.getInstance("SHA-256");

**or**

Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5PADDING");

## 代码里面创建的对象，都应该被使用

编号：S1848

不要创建无意义且不使用的对象。

**Noncompliant Code Example**

if (x < 0)

new IllegalArgumentException("x must be nonnegative");

**Compliant Solution**

if (x < 0)

throw new IllegalArgumentException("x must be nonnegative");

## 序列化接口的子类和非序列化父类

编号：S2055

一个实现序列化接口的子类的父类(父类没有实现序列化)，父类应该包含一个无参的构造方法。因为子类实现序列化以后，子类内部的父类域将不会被序列化，会造成数据丢失。因此需要父类提供无参的构造方法，和初始化父类属性的构造方法，由子类在序列化时调用时父类构造方法进行初始化。

**Noncompliant Code Example**

public class Fruit {

private Season ripe;

public Fruit (Season ripe) {...}

public void setRipe(Season ripe) {...}

public Season getRipe() {...}

}

public class Raspberry extends Fruit

implements Serializable { // Noncompliant; nonserializable ancestor doesn't have no-arg constructor

private static final long serialVersionUID = 1;

private String variety;

public Raspberry(Season ripe, String variety) { ...}

public void setVariety(String variety) {...}

public String getVarity() {...}

}

**Compliant Solution**

public class Fruit {

private Season ripe;

public Fruit () {...}; // Compliant; no-arg constructor added to ancestor

public Fruit (Season ripe) {...}

public void setRipe(Season ripe) {...}

public Season getRipe() {...}

}

public class Raspberry extends Fruit

implements Serializable {

private static final long serialVersionUID = 1;

private String variety;

public Raspberry(Season ripe, String variety) {...}

public void setVariety(String variety) {...}

public String getVarity() {...}

}

## Session中的数据必须验证

编号：S3318

未经验证的数据不应该被存在Session中，我们要保证保存在Session里面的数据，都是经过验证的，无误的，数据合法的。

**Noncompliant Code Example**

login = request.getParameter("login");

session.setAttribute("login", login); // Noncompliant

## 不应该存在无意义的逻辑表达式

编号：S2583

在逻辑判断中，恒定为True 或者 False的判断，不应该在后续的代码中再次出现该值的重新判断。后续代码将会没有任何意义，且不会被执行。

**Noncompliant Code Example**

//foo can't be both equal and not equal to bar in the same expression

if(foo == bar && something && foo != bar) {...}

private void compute(int foo) {

if (foo == 4) {

doSomething();

// We know foo is equal to 4 at this point, so the next condition is always false

if (foo > 4) {...}

...

}

...

}

private void compute(boolean foo) {

if (foo) {

return;

}

doSomething();

// foo is always false here

if (foo){...}

...

}

## 布尔判断时使用短路逻辑

编号：S2178

**Noncompliant Code Example**

if(getTrue() | getFalse()) { ... } // Noncompliant; both sides evaluated

**Compliant Solution**

if(getTrue() || getFalse()) { ... } // true short-circuit logic

## 数组无意义的包含判断

编号：S2175

泛型是String ,而判断是否包含Integer 是没有任何意思的。代码中不能出现这种无意义判断

List<String> list = new ArrayList<String>();

Integer integer = Integer.valueOf(1);

if (list.contains(integer)) { // Noncompliant. Always false.

list.remove(integer); // Noncompliant. list.add(integer) doesn't compile, so this will always return false

}

## 单精度浮点型运算规则

编号：S2164

应该使用BigDecimal,或者强转Double 运算

**Noncompliant Code Example**

float a = 16777216.0f;

float b = 1.0f;

float c = a + b; // Noncompliant; yields 1.6777216E7 not 1.6777217E7

double d = a + b; // Noncompliant; addition is still between 2 floats

**Compliant Solution**

float a = 16777216.0f;

float b = 1.0f;

BigDecimal c = BigDecimal.valueOf(a).add(BigDecimal.valueOf(b));

double d = (double)a + (double)b;

## List indexOf 方法注意要点

编号：S2692

使用，针对字符串和对象查找时的区别

**Noncompliant Code Example**

String color = "blue";

String name = "ishmael";

List<String> strings = new ArrayList<String>();

strings.add(color);

strings.add(name);

if (strings.indexOf(color) > 0) { // Noncompliant

// ...

}

if (name.indexOf("ish") > 0) { // Noncompliant

// ...

}

if (name.indexOf("hma") > 2) { // Noncompliant

// ...

}

**Compliant Solution**

String color = "blue";

String name = "ishmael";

List<String> strings = new ArrayList<String>();

strings.add(color);

strings.add(name);

if (strings.indexOf(color) > -1) {

// ...

}

if (name.indexOf("ish") >= 0) {

// ...

}

if (name.indexOf("hma") > -1) {

// ...

}

## Readline方法的正确使用方式

编号：S2677

**Noncompliant Code Example**

public void doSomethingWithFile(String fileName) {

BufferedReader buffReader = null;

try {

buffReader = new BufferedReader(new FileReader(fileName));

while (buffReader.readLine() != null) { // Noncompliant

// ...

}

} catch (IOException e) {

// ...

}

}

**Compliant Solution**

public void doSomethingWithFile(String fileName) {

BufferedReader buffReader = null;

try {

buffReader = new BufferedReader(new FileReader(fileName));

String line = null;

while ((line = buffReader.readLine()) != null) {

// ...

}

} catch (IOException e) {

// ...

}

}

## 读流在读文件时，需要增加判断

编号：S2674

**Noncompliant Code Example**

public void doSomething(String fileName) {

try {

InputStream is = new InputStream(file);

byte [] buffer = new byte[1000];

is.read(buffer); // Noncompliant

// ...

} catch (IOException e) { ... }

}

**Compliant Solution**

public void doSomething(String fileName) {

try {

InputStream is = new InputStream(file);

byte [] buffer = new byte[1000];

int count = 0;

while (count = is.read(buffer) > 0) {

// ...

}

} catch (IOException e) { ... }

}

## "Calendars" and "DateFormats” 不能是静态的

编号：S2885

因为线程不安全

**Noncompliant Code Example**

public class MyClass {

static private SimpleDateFormat format = new SimpleDateFormat("HH-mm-ss"); // Noncompliant

static private Calendar calendar = Calendar.getInstance(); // Noncompliant

**Compliant Solution**

public class MyClass {

private SimpleDateFormat format = new SimpleDateFormat("HH-mm-ss");

private Calendar calendar = Calendar.getInstance();

## Junit 断言不应该出现在多线程的方法中

编号：S2186

这样有可以会导致线程抛出异常时，断言仍然成功。

**Noncompliant Code Example**

public void run() {

// ...

Assert.assertEquals(expected, actual); // Noncompliant

}

## 重写的Equals方法必须包含一定的判断

编号：S2097

**Noncompliant Code Example**

public boolean equals(Object obj) {

MyClass mc = (MyClass)obj; // Noncompliant

// ...

}

**Compliant Solution**

public boolean equals(Object obj) {

if (obj == null)

return false;

if (this.getClass() != obj.getClass())

return false;

MyClass mc = (MyClass)obj;

// ...

}

## Cookies 写入时 安全设置必须为true

编号：S2092

**Noncompliant Code Example**

Cookie c = new Cookie(SECRET, secret); // Noncompliant; cookie is not secure

response.addCookie(c);

**Compliant Solution**

Cookie c = new Cookie(SECRET, secret);

c.setSecure(true);

response.addCookie(c);

## 循环的判断条件至少要运行一次

编号：S2252

**Noncompliant Code Example**

for (int i = 10; i < 10; i++) { // Noncompliant

// ...

## 线程加入同步锁时，使用wait代替sleep

编号：S2276

线程加入同步锁时，应该使用wait方法代替sleep方法。Sleep方法会占有锁，wait方法不会。

**Noncompliant Code Example**

public void doSomething(){

synchronized(monitor) {

while(notReady()){

Thread.sleep(200);

}

process();

}

...

}

**Compliant Solution**

public void doSomething(){

synchronized(monitor) {

while(notReady()){

monitor.wait(200);

}

process();

}

...

}

## 对象写入文件时必须实现序列化接口

编号：S2118

**Noncompliant Code Example**

public class Vegetable { // neither implements Serializable nor extends a class that does

//...

}

public class Menu {

public void meal() throws IOException {

Vegetable veg;

//...

FileOutputStream fout = new FileOutputStream(veg.getName());

ObjectOutputStream oos = new ObjectOutputStream(fout);

oos.writeObject(veg); // Noncompliant. Nothing will be written

}

}

**Compliant Solution**

public class Vegetable implements Serializable { // can now be serialized

//...

}

public class Menu {

public void meal() throws IOException {

Vegetable veg;

//...

FileOutputStream fout = new FileOutputStream(veg.getName());

ObjectOutputStream oos = new ObjectOutputStream(fout);

oos.writeObject(veg);

}

}

## 时间的参数范围

编号：S2110

JAVA Date 或 Calendar 时间参数范围

|  |  |
| --- | --- |
| \* @param year the year minus 1900.  \* @param month the month between 0-11.  \* @param date the day of the month between 1-31.  \* @param hrs the hours between 0-23.  \* @param min the minutes between 0-59.  \* @param sec the seconds between 0-59. |  |

**Noncompliant Code Example**

Date d = new Date();

d.setDate(25);

d.setYear(2014);

d.setMonth(12); // Noncompliant; rolls d into the next year

Calendar c = new GregorianCalendar(2014, 12, 25); // Noncompliant

if (c.get(Calendar.MONTH) == 12) { // Noncompliant; invalid comparison

// ...

}

**Compliant Solution**

Date d = new Date();

d.setDate(25);

d.setYear(2014);

d.setMonth(11);

Calendar c = new Gregorian Calendar(2014, 11, 25);

if (c.get(Calendar.MONTH) == 11) {

// ...

}

## 方法："BigDecimal(double)" 不应该这样调用。

编号：S2111

否则会造成生成的BigDecimal值异常.Double类型的小数位会变长

**Noncompliant Code Example**

double d = 1.1;

BigDecimal bd1 = new BigDecimal(d); // Noncompliant; see comment above

BigDecimal bd2 = new BigDecimal(1.1); // Noncompliant; same result

**Compliant Solution**

double d = 1.1;

BigDecimal bd1 = BigDecimal.valueOf(d);

BigDecimal bd2 = BigDecimal.valueOf(1.1);

## StringFromat的使用规范

编号：S2275

**Noncompliant Code Example**

String.format("The value of my integer is %d", "Hello World"); // Noncompliant; an 'int' is expected rather than a String

String.format("First {0} and then {1}", "foo", "bar"); //Noncompliant. Looks like there is a confusion with the use of {{java.text.MessageFormat}}, parameters "foo" and "bar" will be simply ignored here

String.format("Duke's Birthday year is %tX", c); //Noncompliant; X is not a supported time conversion character

String.format("Display %3$d and then %d", 1, 2, 3); //Noncompliant; the second argument '2' is unused

String.format("Display %0$d and then %d", 1); //Noncompliant; arguments are numbered starting from 1

String.format("Too many arguments %d and %d", 1, 2, 3); //Noncompliant; the third argument '3' is unused

String.format("Not enough arguments %d and %d", 1); //Noncompliant; the second argument is missing

String.format("First Line\n"); //Noncompliant; %n should be used in place of \n to produce the platform-specific line separator

String.format("%< is equals to %d", 2); //Noncompliant; the argument index '<' refers to the previous format specifier but there isn't one

String.format("Is myObject null ? %b", myObject); //Noncompliant; when a non-boolean argument is formatted with %b, it prints true for any nonnull value, and false for null. Even if intended, this is misleading. It's better to directly inject the boolean value (myObject == null in this case)

String.format("value is " + value); // Noncompliant

String s = String.format("string without arguments"); // Noncompliant

**Compliant Solution**

String.format("The value of my integer is %d", 3);

String.format("First %s and then %s", "foo", "bar");

String.format("Duke's Birthday year is %tY", c);

String.format("Display %2$d and then %d", 1, 3);

String.format("Display %1$d and then %d", 1);

String.format("Too many arguments %d %d", 1, 2);

String.format("Not enough arguments %d and %d", 1, 2);

String.format("First Line%n");

String.format("%d is equals to %<", 2);

String.format("Is myObject null ? %b", myObject == null);

String.format("value is %d", value);

String s = "string without arguments";

## "static final" 修饰的数组应该是 "private"

编号：S1873

"static final" 修饰的数组应该是 "private"，因为这样不会造成数据被修改。

**Noncompliant Code Example**

public class Estate {

// Noncompliant; array contents can be modified

public static final String [] HEIRS = new String [] {

"Betty", "Suzy" };

}

public class Malicious {

public void changeWill() {

Estate.HEIRS[0] = "Biff";

if (Estate.HEIRS.length > 1) {

for (int i = 1; i < Estate.HEIRS.length; i++) {

Estate.HEIRS[i] = "";

}

}

}

**Compliant Solution**

public class Estate {

private static final String [] HEIRS = new String [] {

"Betty", "Suzy" };

public String [] getHeirs() {

// return copy of HEIRS

}

}

## 实现"Cloneable" 应该重写clone方法

编号：S2157

**Noncompliant Code Example**

class Team implements Cloneable { // Noncompliant

private Person coach;

private List<Person> players;

public void addPlayer(Person p) {...}

public Person getCoach() {...}

}

**Compliant Solution**

class Team implements Cloneable {

private Person coach;

private List<Person> players;

public void addPlayer(Person p) { ... }

public Person getCoach() { ... }

@Override

public Object clone() {

Team clone = (Team) super.clone();

//...

}

}

## 后加加 前加加的使用规范

编号：S2123

**Noncompliant Code Example**

public int pickNumber() {

int i = 0;

int j = 0;

i = i++; // Noncompliant; i is still zero

return j++; // Noncompliant; 0 returned

}

**Compliant Solution**

public int pickNumber() {

int i = 0;

int j = 0;

i++;

return ++j;

}

## 属性必须加修饰符

编号：S2039

**Noncompliant Code Sample**

class Ball {

String color="red"; // Noncompliant

}

enum A {

B;

int a;

}

**Compliant Solution**

class Ball {

private String color="red"; // Compliant

}

enum A {

B;

private int a;

}

## If判断规则

编号： S1764

**Noncompliant Code Example**

if ( a == a ) { // always true

doZ();

}

if ( a != a ) { // always false

doY();

}

if ( a == b && a == b ) { // if the first one is true, the second one is too

doX();

}

if ( a == b || a == b ) { // if the first one is true, the second one is too

doW();

}

int j = 5 / 5; //always 1

int k = 5 - 5; //always 0

**Compliant Solution**

doZ();

if ( a == b ) {

doX();

}

if ( a == b ) {

doW();

}

int j = 1;

int k = 0;

## 错误的逻辑判断，不应该出现

编号：S1697

**Noncompliant Code Example**

if (str == null && str.length() == 0) {

System.out.println("String is empty");

}

if (str != null || str.length() > 0) {

System.out.println("String is not empty");

}

**Compliant Solution**

if (str == null || str.length() == 0) {

System.out.println("String is empty");

}

if (str != null && str.length() > 0) {

System.out.println("String is not empty");

}

## 代码中打印到控制台的异常信息应该使用log代替

编号：S1148

**The following code:**

try {

/\* ... \*/

} catch(Exception e) {

e.printStackTrace(); // Noncompliant

}

**should be refactored into:**

try {

/\* ... \*/

} catch(Exception e) {

LOGGER.log("context", e); // Compliant

}

## 对象判断NULL条件的规则

编号：S1318

**Noncompliant Code Example**

if (variable.equals(null)) { /\* ... \*/ } // Noncompliant - "variable" is really null, a NullPointerException is thrown

**Compliant Solution**

if (variable == null) { /\* ... \*/ } // Compliant

## IF/IF ELSE 条件判断不应该出现重复判断

编号：S1862

**Noncompliant Code Example**

if (param == 1)

openWindow();

else if (param == 2)

closeWindow();

else if (param == 1) // Noncompliant

moveWindowToTheBackground();

}

**Compliant Solution**

if (param == 1)

openWindow();

else if (param == 2)

closeWindow();

else if (param == 3)

moveWindowToTheBackground();

}

## Finally 模块不能抛出异常

编号：S1163

**The following code:**

try {

/\* some work which end up throwing an exception \*/

throw new IllegalArgumentException();

} finally {

/\* clean up \*/

throw new RuntimeException(); // Noncompliant - will mask the IllegalArgumentException

}

**should be refactored into:**

try {

/\* some work which end up throwing an exception \*/

throw new IllegalArgumentException();

} finally {

/\* clean up \*/ // Compliant

}

## 赋值时，不应该自己给自己赋值

编号：S1656

**Noncompliant Code Example**

public void setName(String name) {

name = name;

}

**Compliant Solution**

public void setName(String name) {

this.name = name;

}

## Public static 后面应该加入final

编号：S1444

**Noncompliant Code Example**

public class Greeter {

public static Foo foo = new Foo();

...

}

**Compliant Solution**

public class Greeter {

public static final Foo FOO = new Foo();

...

}

## Finally模块中不能出现return

编号：S1143

**Noncompliant Code Example**

public static void main(String[] args) {

try {

doSomethingWhichThrowsException();

System.out.println("OK"); // incorrect "OK" message is printed

} catch (RuntimeException e) {

System.out.println("ERROR"); // this message is not shown

}

}

public static void doSomethingWhichThrowsException() {

try {

throw new RuntimeException();

} finally {

for (int i = 0; i < 10; i ++) {

//...

if (q == i) {

break; // ignored

}

}

/\* ... \*/

return; // Noncompliant - prevents the RuntimeException from being propagated

}

}

**Compliant Solution**

public static void main(String[] args) {

try {

doSomethingWhichThrowsException();

System.out.println("OK");

} catch (RuntimeException e) {

System.out.println("ERROR"); // "ERROR" is printed as expected

}

}

public static void doSomethingWhichThrowsException() {

try {

throw new RuntimeException();

} finally {

for (int i = 0; i < 10; i ++) {

//...

if (q == i) {

break; // ignored

}

}

/\* ... \*/

}

}

## compareTo 方法 和equals方法 必须一起重写

编号：S1210

否则在集合中判断是同一个对象。

**Noncompliant Code Example**

public class Foo implements Comparable<Foo> {

@Override

public int compareTo(Foo foo) { /\* ... \*/ } // Noncompliant as the equals(Object obj) method is not overridden

}

**Compliant Solution**

public class Foo implements Comparable<Foo> {

@Override

public int compareTo(Foo foo) { /\* ... \*/ } // Compliant

@Override

public boolean equals(Object obj) { /\* ... \*/ }

}

## 使用StringBuffer代替加号

编号： SBSC\_USE\_STRINGBUFFER\_CONCATENATION

**For example:**

// This is bad

String s = "";

for (int i = 0; i < field.length; ++i) {

s = s + field[i];

}

// This is better

StringBuffer buf = new StringBuffer();

for (int i = 0; i < field.length; ++i) {

buf.append(field[i]);

}

String s = buf.toString();

## 无限循环递归将加入代码检测

编号：IL\_INFINITE\_RECURSIVE\_LOOP

## 父类强转子类加入代码检测

编号：BC\_IMPOSSIBLE\_DOWNCAST

## 空引用加入代码检测

编号：Possible null pointer dereference

## 奇数的正确判断方式

编号： IM\_BAD\_CHECK\_FOR\_ODD

应该用x % 2 != 0 不能使用  x % 2 == 1

## 整型计算转DOUBLE

编号：ICAST\_IDIV\_CAST\_TO\_DOUBLE

整型计算不要在计算后的结果强转Double。在计算前就转成Double计算，防止精度丢失。

int x = 2;

int y = 5;

// Wrong: yields result 0.0

double value1 = x / y;

// Right: yields result 0.4

double value2 = x / (double) y;

## equals()在调用前检查是否为NULL

编号：NULLNP\_EQUALS\_SHOULD\_HANDLE\_NULL\_ARGUMENT

## 集合清空方式

编号：DMI\_USING\_REMOVEALL\_TO\_CLEAR\_COLLECTION

不要使用removeALL去清空集合，应该使用clear方法

## 重写Eqlues方法 必须同时重写hashCode方法

编号：HE\_INHERITS\_EQUALS\_USE\_HASHCODE

## serialVersionUID 必须是final的

编号：SE\_NONFINAL\_SERIALVERSIONID

## String的substring方法 参数传0的时候 会返回原值

编号：DMI\_USELESS\_SUBSTRING

## securerandom 代替Random方法

编号：DMI\_RANDOM\_USED\_ONLY\_ONCE

代码中尽量不要使用java.util.Random方法 因为他产生的随机数 是可以预测的。且多线程下线程不安全，且效率低下。 如果真要使用建议， 生成一次随机数以后重新生成对象。

推荐使用java.security.securerandom

## String判断使用equals

编号：StringEqualityComparisonCheck

**Noncompliant Code Example**

if (variable == "foo") { /\* ... \*/ }

if (variable != "foo") { /\* ... \*/ }

**Compliant Solution**

if ("foo".equals(variable)) { /\* ... \*/ } // Compliant

if (!"foo".equals(variable)) { /\* ... \*/ } // Compliant

# Restful前后端分离开发规范（1.2）

**修订历史**

|  |  |  |  |
| --- | --- | --- | --- |
| 版本号 | 作者 | 内容提要 | 发布日期 |
| 1.0 | 九天 | 拟定 |  |
| 1.1 | 九天 | 增加选型及DEMO部分 |  |
| 1.2 | 九天 | 修订响应JSON数据格式 |  |
|  |  |  |  |
|  |  |  |  |

## 背景、概述及目标

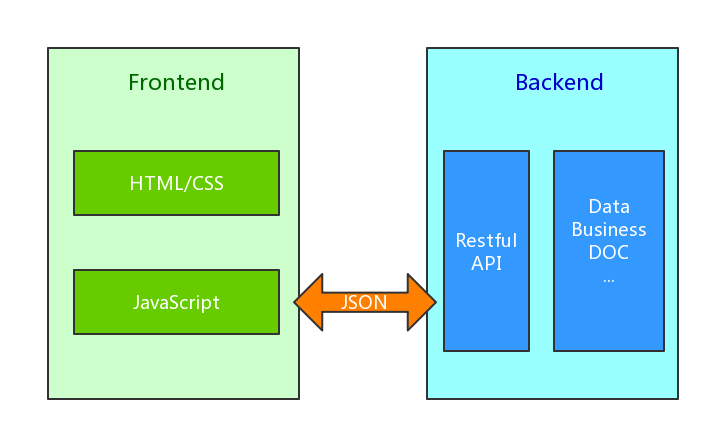
|  |  |
| --- | --- |
| 背景基本介绍 | 目前前后端开发主要存在以下问题：  1. 前后端开发步调不一致，中间联调状况频出；  2. 接口文档和接口真实提供接口不一致；  3. 前端联调没有对应的真实数据；  4. Server端接口开发完成，认为事情便是做完了，便投入到其他项目中；  5. 跨组沟通成本高，响应不能实时； |

## 基本原则

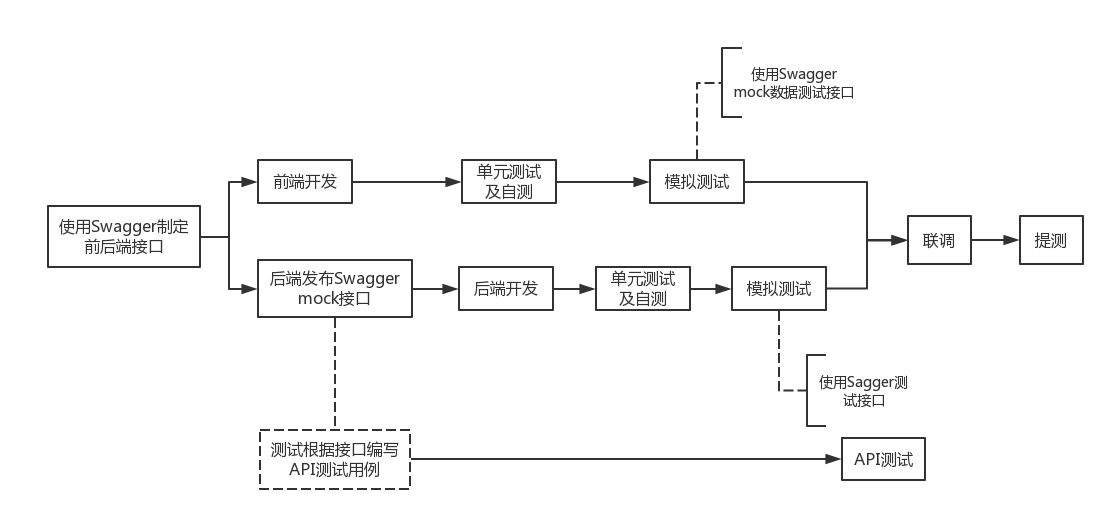
1. 接口和接口文档同步变动。
2. 接口文档语义明确、简洁清晰、便于查阅。
3. 能Mock服务提供给到客户端测试。
4. 能方便地从Mock环境切换到正式环境。

## 规范方案

### 总体架构



### 开发流程



### 开发规范

1. 数据交互格式统一使用JSON。
2. 参数统一使用驼峰命名。
3. 需明确标注参数的类型、格式及区分可选和必选参数。
4. 在POST，PUT，PATCH上使用JSON作为输入参数（参考第1条）。
5. RESTFul API是无状态的，也就是说用户请求的鉴权和Cookie以及Session无关，每一次请求都应该包含鉴权证明。
6. GET，DELETE方法应该是幂等的。

### 资源地址规范

请参考APP接口规范。

### 响应JSON数据格式

响应JSON数据包括头部和业务数据两部分：

|  |
| --- |
| {  "mogoCode" : 10000,  "msg" : "ok",  "showMsg" : "操作成功",  "sign":"",  "sign\_type ":"RSA",  "token":"",  "version":"1.0.1",  "page":{  "currentPage ":5,  " totalPage ":10,  " totalResult ":195  "pageSize":20  }  "data" : {  ... ...  }  } |

## 工具选型

目前流行的工具主要有Swagger，RAP，Postman。

Swagger是为了描述一套标准的而且是和语言无关的REST API的规范。对于外部调用者来说，只需通过Swagger文档即可清楚Server端提供的服务，而不需去阅读源码或接口文档说明，引入Swagger可以保证接口和接口文档的同步变动。使用Swagger或Spring 的标准注解之后，Swagger UI即可自动生成API文档，并可以直接模拟测试。

RAP通过GUI工具帮助WEB工程师更高效的管理接口文档，同时通过分析接口结构自动生成Mock数据、校验接口的正确性，使接口文档成为开发流程中的强依赖。在开发时前端将请求转至RAP，由RAP提供模拟数据；而后端使用RAP测试接口的正确性。这样RAP就成为了开发过程中的强依赖，进而确保接口文档的实时正确性。RAP采用JSON-Schema规范描述HTTP请求及其JSON响应。除了异步的HTTP接口，输出给模板系统的任何数据结构都将被RAP管理起来。

Postman是被大家所熟知的网页调试Chrome插件，我们常常用它来进行临时的http请求调试。幸运的是，Postman可以将调试过的请求保存到Collection中。形成的Collection就可以作为一份简单有效且支持在线测试的接口文档，使用同一账号登录就可以做到分享和同步。对QA来说，使用Postman进行接口测试和接口文档维护是同一件事情，测试即文档，维护成本也很低。所以Postman是一个测试导向的API小工具，并没有具体的API文档，是先有接口实现，再维护一份“测试记录”，适合小的测试团队自己使用并维护，这里先排除掉，就不再讨论。

Swagger丰富且独立的各个功能使得它可以被应用在各种需求下，不论是开发还是测试都可以使用这个工具，来优化自己的开发过程，进行接口文档维护、接口测试等； RAP的应用范围非常明确，是一个面向开发人员自测和联调的工具性平台，它更适合以开发为核心对接口进行维护，供测试人员参考。选型标准还是基于之前的4个基本原则：

|  |  |  |
| --- | --- | --- |
| 比较项 | RAP | Swagger |
| 文档编辑方式 | GUI手动编辑 | 注解描述，自动生成 |
| 接口文档同步 | 接口变动后手动编辑 | 接口、注解重构后自动同步更新 |
| 接口文档准确性及可阅读性 | 语义、类型明确，查阅方便 | 语义、类型明确，查阅方便 |
| Mock数据生成 | 定义接口时指定Mock数据生成规则 | 定义接口时按约定返回默认实现 |
| Mock服务 | 即时提供 | 即时提供 |
| Mock环境切换到正式环境 | 方便、瞬时切换 | 方便、瞬时切换 |
| 稳定及健壮性 | 一般，还在迭代开发中 | 比较成熟稳定 |

综上对比：Swagger生成和同步API文档更为方便，系统也更成熟、稳定；RAP生成Mock数据更方便，更有质量，但稳定性稍差。

建议优先使用Swagger。

### Swagger API定义规范

#### API设计的基本要求

1. API应该对程序员友好，并且在浏览器地址栏容易输入。
2. API应该简单，直观，容易使用同时优雅。
3. API应该具有足够的灵活性来支持上层UI。
4. API应该明确定义可能出现error code及error message。

#### API开发规范

ApiInfo必填字段：

|  |  |  |
| --- | --- | --- |
| 属性 | 是否必填 | 说明 |
| title | Y | 模块API名称，应简短、精确。 |
| description | Y | 详细描述，包括的接口及实现的功能、方式等。 |
| contact | Y | 开发者的联系方式：公司邮箱地址。 |

API应该首先描述其访问路径及访问方法，通常使用注解描述：

@Path必填字段：

|  |  |  |
| --- | --- | --- |
| 属性 | 是否必填 | 说明 |
| value | Y | URL路径。 |

@RequestMapping必填字段：

|  |  |  |
| --- | --- | --- |
| 属性 | 是否必填 | 说明 |
| value | Y | URL路径。 |
| method | Y | [GET ,POST,PUT,DELETE,PATCH,HEAD,OPTIONS] |
| produces | Y | 请求返回内容体媒体类型 |

@ApiOperation必填字段：

|  |  |  |
| --- | --- | --- |
| 属性 | 是否必填 | 说明 |
| value | Y | API功能说明，应简短、精确。 |
| httpMethod | Y | [GET ,POST,PUT,DELETE,PATCH,HEAD,OPTIONS] |
| response | Y | 返回值类型。 |
| notes | Y | API详细描述，应包括调用（参数）说明及返回值详细说明。 |

@ApiParam必填字段：

|  |  |  |
| --- | --- | --- |
| 属性 | 是否必填 | 说明 |
| required | Y | 是否必传参数。 |
| name | Y | 参数名称。 |
| value | Y | 参数简短说明。 |

### Mock测试

使用上述注解定义接口并返回正确的JSON对象后，通过Swagger就可以查看API文档,并且可以访问、测试Mock接口了。

#### Mock API质量

1. Mock服务应集中部署，统一访问入口及文档，坚决避免在开发者机子上部署、访问Mock服务。
2. 前端和后端任何一方均不能单方面变更API，变更API需双方协商一致后方可进行。
3. Server端变更API定义后应及时发布代码到Mock服务器上并通知前端及测试人员。

#### Mock数据质量

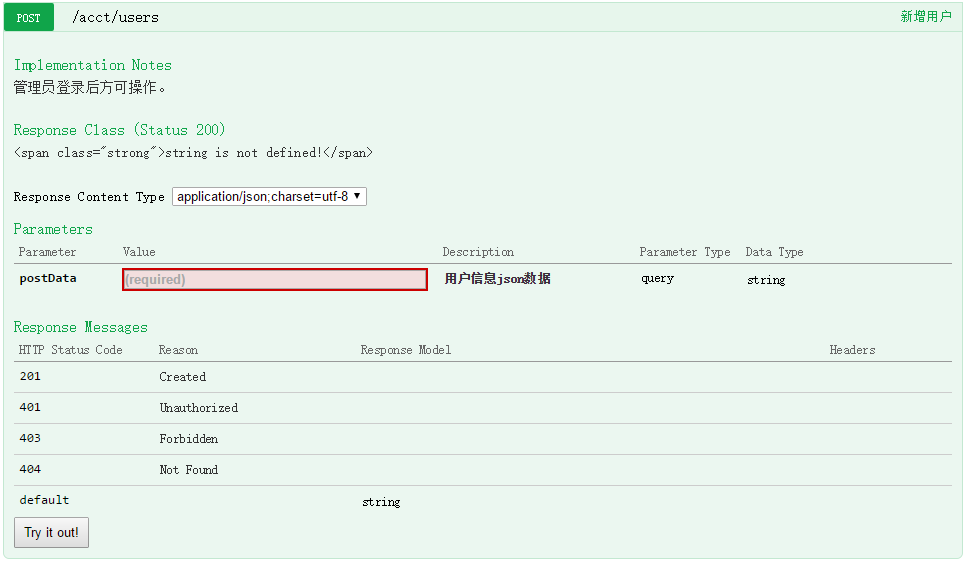
1. Mock数据应该由API Server端定义者生成。
2. 生成的Mock数据除需保证一致性、完备性外，还应覆盖API定义中的各种例外情况。
3. Mock数据应尽量跟线上真实环境的数据保持类似，样本数据的覆盖率应达到6成以上。

## 附录A-DEMO

#### API POST请求

|  |
| --- |
| @RequestMapping(value = "/users", method = RequestMethod.***POST***, produces = "application/json; charset=utf-8")  @ApiOperation(value = "新增用户", notes = "管理员登录后方可操作。")  **public** String addUser(  @ApiParam(required = **true**, name = "postData", value = "用户信息json数据") @RequestParam(value = "postData") String postData,  HttpServletRequest request) **throws** JSONException {  JSONObject json = **new** JSONObject();  json.put("postData", postData);  **return** json.toString();  } |

API 文档：



#### API GET请求

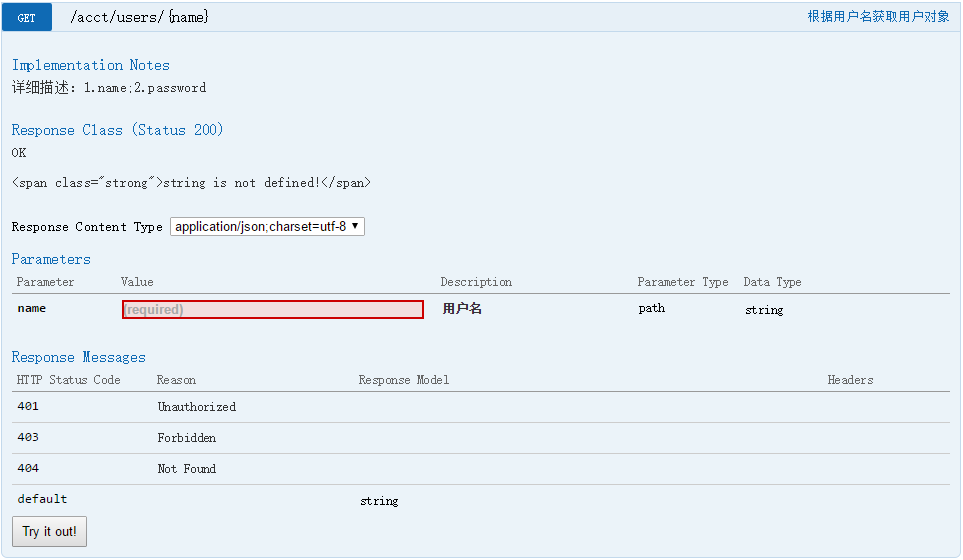
|  |
| --- |
| @RequestMapping(value = "/users/{name}", method = RequestMethod.***GET***, produces = "application/json; charset=utf-8")  @ApiOperation(value = "根据用户名获取用户对象", httpMethod = "GET", response = String.**class**, notes = "详细描述：1.name;2.password")  **public** String getUserByName(@ApiParam(required = **true**, name = "name", value = "用户名") @PathVariable String name)  **throws** Exception {  **return** "Your name is:" + name;  } |

API 文档：

#### API DELETE请求

|  |
| --- |
| @RequestMapping(value = "/users/{id}", method = RequestMethod.***DELETE***, produces = "application/json; charset=utf-8")  @ApiOperation(value = "删除指定ID的用户", httpMethod = "DELETE", notes = "删除指定ID用户")  **public** String delUserById(@ApiParam(required = **true**, name = "id", value = "用户ID") @PathVariable String id)  **throws** Exception {  **return** "Your input id:" + id;  } |

API 文档：



# Code Review流程及规范

1. 统一采用git merge request机制，可参考“[操作步骤](https://www.jianshu.com/p/6bcd082101c1)”。简而言之，有以下几个步骤：
   1. Step 1：创建项目（这步可以省略）
   2. Step 2：源仓库的构建（大家只要看最后的fork就行了）
   3. Step 3：在自己的项目上进行开发
   4. Step 4：向管理员提交pull request
   5. Step 5：Code Review
2. 在代码开发提交时进行（review 时必须描述出功能及缺陷列表）
3. 系统自动把CR的结果邮件周知到项目团队
4. 生产bug责任追踪到code review相关人，建议编写者与review者四六开。
5. 业务组集中review 每周或者每月一次
6. 开发在提测时，分析当前项目分支的CR日志，如果发现没有CR日志记录，则不允许提测
7. 周报上体现（业务组别、项目名、review人次、缺陷数量、生产bug数量、项目是否冒烟打回）
8. 好的review后的好建议补充道代码规范中