后端开发规范

目录

[（一） 编码规范 1](#_Toc113277322)

[1.1 命名风格 1](#_Toc113277323)

[1.2 常量定义 2](#_Toc113277324)

[1.3 代码格式 3](#_Toc113277325)

[1.4 面向对象编程规范 4](#_Toc113277326)

[1.5 集合处理 4](#_Toc113277327)

[1.6 并发处理 6](#_Toc113277328)

[1.7 控制语句 6](#_Toc113277329)

[1.8 注释规约 6](#_Toc113277330)

[1.9 其他 6](#_Toc113277331)

[（二） 异常日志 7](#_Toc113277332)

[2.1 异常处理 7](#_Toc113277333)

[2.2 日志规约 8](#_Toc113277334)

[（三） 单元测试 9](#_Toc113277335)

[（四） 安全规约 9](#_Toc113277336)

[（五） MySQL 数据库 9](#_Toc113277337)

[5.1 建表规约 9](#_Toc113277338)

[5.2 索引规约 10](#_Toc113277339)

[5.3 SQL 语句 11](#_Toc113277340)

[5.4 ORM 映射 12](#_Toc113277341)

[（六） 工程结构 13](#_Toc113277342)

[6.1 应用分层 13](#_Toc113277343)

[6.2 服务器 13](#_Toc113277344)

[6.3 二方库依赖 14](#_Toc113277345)

# 编码规范

## 命名风格

1. 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. 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. 类名使用首字母小写的驼峰式命名法。

例如：userServiceImpl、userController

1. 方法名、参数名、成员变量、局部变量都统一使用首字母小写的驼峰式命名法。

例如：localValue,getUserId(),int userName;

1. 常量命名全部大写，单词间用下划线隔开

例如：MAX\_STOCK\_COUNT

1. POJO 类中布尔类型的变量，都不要加 is 前缀，否则部分框架解析会引起序列化错误。
2. 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。
3. 杜绝完全不规范的缩写，避免望文不知义。
4. 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。
5. 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。

## 常量定义

1. 不允许任何魔法值（即未经预先定义的常量）直接出现在代码中。
2. 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。
3. 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。

## 代码格式

1. 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. 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。
7. if/for/while/switch/do 等保留字与括号之间都必须加空格。
8. 任何二目、三目运算符的左右两边都需要加一个空格。
9. 采用 4 个空格缩进，禁止使用 tab 字符。
10. 注释的双斜线与注释内容之间有且仅有一个空格。
11. 单行字符数限制不超过 120 个（链式编程），超出需要换行，换行时遵循如下原则：
12. 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。
13. 运算符与下文一起换行。
14. 方法调用的点符号与下文一起换行。
15. 方法调用时，多个参数，需要换行时，在逗号后进行。
16. 在括号前不要换行。
17. 方法参数在定义和传入时，多个参数逗号后边必须加空格。
18. 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。

## 面向对象编程规范

1. 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. 所有的覆写方法，必须加@Override 注解。
3. 不能使用过时的类或方法。
4. 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。
5. 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。关于基本数据类型与包装数据类型的使用标准如下：
6. 所有的 POJO 类属性必须使用包装数据类型。
7. RPC 方法的返回值和参数必须使用包装数据类型。
8. 所有的局部变量使用基本数据类型。
9. 定义 DO/DTO/VO 等 POJO 类时，不要设定任何属性默认值。
10. 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。
11. POJO 类必须写 toString 方法。使用 IDE 中的工具：source>generate toString 时，如果继承了另一个 POJO 类，注意在前面加一下 super.toString。
12. 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。
13. 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。
14. 类内方法定义的顺序依次是：公有方法或保护方法 > 私有方法 > getter/setter 方法。
15. 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。
16. 循环体内，字符串的连接方式，使用 StringBuilder 的 append 方法进行扩展。
17. 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：
18. 不允许被继承的类，如：String 类。
19. 不允许修改引用的域对象，如：POJO 类的域变量。
20. 不允许被重写的方法，如：POJO 类的 setter 方法。
21. 不允许运行过程中重新赋值的局部变量。
22. 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。

## 集合处理

1. 关于 hashCode 和 equals 的处理，遵循如下规则：

1） 只要重写 equals，就必须重写 hashCode。

2） 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。

3） 如果自定义对象作为 Map 的键，那么必须重写 hashCode 和equals。

1. 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.
2. 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；如果数组元素个数大于实际所需，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。

1. 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。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。
2. 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Unsupported- OperationException 异常。
3. 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。
4. 集合初始化时，指定集合初始值大小
5. 使用 entrySet 遍历 Map 类集合 KV，而不是 keySet 方式进行遍历。
6. Map 类集合只有TreeMap的Value和HashMap的Key和Value 才允许为null, Hashtable、ConcurrentHashMap的Key和Value 不允许为null。
7. 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。
8. 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

1. 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。
2. 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。
3. 线程资源必须通过线程池提供，不允许在应用中自行显式创建线程。
4. 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。
5. 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。
6. 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。
7. 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。

## 控制语句

1. 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，即使空代码。
2. 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;
3. 在高并发场景中，避免使用”等于”判断作为中断或退出的条件。
4. 表达异常的分支时，少用 if-else 方式
5. 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，以提高可读性。
6. 避免采用取反逻辑运算符。

## 注释规约

1. 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。
2. 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
3. 所有的类都必须添加创建者和创建日期。
4. 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。
5. 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。
6. 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。
7. 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。

## 其他

* 1. 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。
  2. 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。
  3. 后台输送给页面的变量必须加$!{var}——中间的感叹号。
  4. 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。
  5. 获取当前毫秒数 System.currentTimeMillis(); 而不是 new Date().getTime();
  6. 不要在视图模板中加入任何复杂的逻辑。
  7. 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。
  8. 及时清理不再使用的代码段或配置信息。

# 异常日志

## 2.1异常处理

1. Java 类库中定义的可以通过预检查方式规避的 RuntimeException 异常不应该通过catch 的方式来处理，比如：NullPointerExceptionIndexOutOf-BoundsException 等等。无法通过预检查的异常除外，比如，在解析字符串形式的数字时，不得不通过 catchNumberFormatException 来实现。
2. 异常不要用来做流程控制，条件控制。
3. 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。
4. 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。
5. 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。
6. finally 块必须对资源对象、流对象进行关闭，有异常也要做 try-catch。
7. 不要在 finally 块中使用 return。
8. 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。
9. 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。
10. 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”。

## 2.2 日志规约

* 1. 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。
  2. 日志文件推荐至少保存 15 天，因为有些异常具备以“周”为频次发生的特点。
  3. 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/monitor/visit 等； logName:日志描述。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。
  4. 避免重复打印日志，浪费磁盘空间，务必在 log4j.xml 中设置 additivity=false。
  5. 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。

# 单元测试

1. 单元测试必须遵守 AIR 原则。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。
2. 单元测试应该是全自动执行的，并且非交互式的。测试用例通常是被定期执行的，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。
3. 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。
4. 单元测试是可以重复执行的，不能受到外界环境的影响。
5. 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，如果新增代码影响了原有单元测试，应该及时修正。
6. 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，
7. 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。
8. 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。
9. 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。

# 安全规约

1. 隶属于用户个人的页面或者功能必须进行权限控制校验。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。
2. 用户敏感数据禁止直接展示，必须对展示数据进行脱敏，防止隐私泄露。
3. 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. 用户请求传入的任何参数必须做有效性验证。
5. 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。
6. 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，避免被滥刷导致资损。
7. 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。

# MySQL 数据库

## 建表规约

1. 表达是与否概念的字段，必须使用 is\_xxx 的方式命名，数据类型是 unsigned tinyint（ 1 表示是，0 表示否）。
2. 表名、字段名必须使用小写字母或数字，禁止出现数字开头，禁止两个下划线中间只出现数字。数据库字段名的修改代价很大，因为无法进行预发布，所以字段名称需要慎重考虑。
3. 表名之中不能使用复数名词。
4. 主键索引名为 pk\_字段名，pk\_ 即 primary key；唯一索引名为 uk\_字段名，uk\_ 即 unique key；普通索引名则为 idx\_字段名，idx\_ 即 index。
5. 由于float 和 double 在存储的时候，存在精度损失的问题，很可能在值的比较时，得到不正确的结果。小数类型为 decimal，禁止使用 float 和 double。如果存储的数据范围超过 decimal 的范围，建议将数据拆成整数和小数分开存储。
6. 如果存储的字符串长度几乎相等，使用 char 定长字符串类型。
7. varchar 是可变长字符串，不预先分配存储空间，长度不要超过 5000，如果存储长度大于此值，定义字段类型为 text，独立出来一张表，用主键来对应，避免影响其它字段索引效率。
8. 如果修改字段含义或对字段表示的状态追加时，需要及时更新字段注释。
9. 字段允许适当冗余，以提高查询性能，但必须考虑数据一致。所谓冗余字段应当既不满足频繁修改的字段，也不满足 varchar 超长字段，更不能是 text 字段。
10. 使用合适的字符存储长度，不但可以节约数据库表空间、节约索引存储，还可以提升检索速度。

## 索引规约

1. 业务上具有唯一特性的字段，即使是多个字段的组合，也必须建成唯一索引。
2. 超过三个表禁止join。需要 join 的字段，数据类型必须绝对一致；多表关联查询时，保证被关联的字段需要有索引。
3. 在 varchar 字段上建立索引时，必须指定索引长度，没必要对全字段建立索引，根据实际文本区分度决定索引长度即可。
4. 建组合索引的时候，区分度最高的在最左边。
5. 防止因字段类型不同造成的隐式转换，导致索引失效。
6. 如果有 order by 的场景，请注意利用索引的有序性。

## SQL 语句

1. 不要使用 count(列名)或 count(常量)来替代 count(\*)，count(\*)是 SQL92 定义的标准统计行数的语法，跟数据库无关，跟 NULL 和非 NULL 无关。
2. count(\*)会统计值为 NULL 的行，而 count(列名)不会统计此列为 NULL 值的行。
3. count(distinct col) 计算该列除 NULL 之外的不重复行数，注意 count(distinct

col1, col2) 如果其中一列全为 NULL，那么即使另一列有不同的值，也返回为 0。

1. 当某一列的值全是 NULL 时，count(col)的返回结果为 0，但 sum(col)的返回结果为NULL，因此使用 sum()时需注意 NPE 问题。
2. 使用 ISNULL()来判断是否为 NULL 值。
3. 在代码中写分页查询逻辑时，若 count 为 0 应直接返回，避免执行后面的分页语句。
4. 不得使用外键与级联，一切外键概念必须在应用层解决。
5. 数据订正（特别是删除、修改记录操作）时，要先 select，避免出现误删除，确认无误才能执行更新语句。

## ORM 映射

1. 在表查询中，一律不要使用 \* 作为查询的字段列表，需要哪些字段必须明确写明。
2. 不要用 resultClass 当返回参数，即使所有类属性名与数据库字段一一对应，也需要定义；反过来，每一个表也必然有一个与之对应。
3. 不允许直接拿 HashMap 与 Hashtable 作为查询结果集的输出。resultClass=”Hashtable”，会置入字段名和属性值，但是值的类型不可控。
4. 更新数据表记录时，必须同时更新记录对应的 gmt\_modified 字段值为当前时间。
5. 不要写一个大而全的数据更新接口。
6. 执行 SQL时，不要更新无改动的字段，不但易出错，效率低，还会增加 binlog 存储。
7. sql.xml 配置参数使用：#{}，#param# 不要使用${} 此种方式容易出现 SQL 注入。
8. POJO 类的布尔属性不能加 is，而数据库字段必须加 is\_，要求在 resultMap 中进行字段与属性之间的映射。

# 工程结构

## 应用分层

1. 开放接口层：可直接封装 Service 方法暴露成 RPC 接口；通过 Web 封装成 http 接口；进行网关安全控制、流量控制等。
2. 终端显示层：各个端的模板渲染并执行显示的层。当前主要是 velocity 渲染，JS 渲染，JSP 渲染，移动端展示等。
3. Web 层：主要是对访问控制进行转发，各类基本参数校验，或者不复用的业务简单处理等。
4. Service 层：相对具体的业务逻辑服务层。
5. Manager 层：通用业务处理层，它有如下特征：
6. 对第三方平台封装的层，预处理返回结果及转化异常信息；
7. 对 Service 层通用能力的下沉，如缓存方案、中间件通用处理；
8. 与 DAO 层交互，对多个 DAO 的组合复用。
9. DAO 层：数据访问层，与底层 MySQL、Oracle等进行数据交互。

## 服务器

1. 高并发服务器建议调小 TCP 协议的 time\_wait 超时时间。
2. 给 JVM 设置-XX:+HeapDumpOnOutOfMemoryError 参数，让 JVM 碰到 OOM 场景时输出dump 信息。
3. 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。
4. 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

## 二方库依赖

1. 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。
3. 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。
4. 二方库版本号命名方式：主版本号.次版本号.修订号
5. 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。
6. 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。
7. 修订号：保持完全兼容性，修复 BUG、新增次要功能特性等。
8. 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。
9. 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。
10. 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。
11. 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。
12. 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。<dependencies>里的依赖都会自动引入，并默认被所有的子项目继承。
13. 二方库不要有配置项，最低限度不要再增加配置项。
14. 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：
15. 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。
16. 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。