**创新事业部JAVA开发手册**

2019年03月

**修改记录**

|  |  |  |  |
| --- | --- | --- | --- |
| 版本 | 修订日期 | 修订人员 | 修订纪要 |
| 1.0 | 2019-03-23 | 徐强辉 | 初始版本 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**文档目录**

[1. 一级Title1](#_Toc17809)

[1.1. 二级Title1](#_Toc1549)

[1.2. 二级Title2](#_Toc25507)

[1.2.1. 三级Title1](#_Toc27634)

[1.2.2. 三级Title2](#_Toc13629)

[1.2.2.1. 四级Title](#_Toc8803)

[2. 一级Title2](#_Toc28876)

[2.1. 二级Title](#_Toc12598)

[2.2. 二级Title](#_Toc25982)

[2.2.1. 三级Title](#_Toc16068)

[2.2.2. 三级Title](#_Toc3207)

[2.2.2.1. 四级Title](#_Toc8877)

[2.2.2.2. 四级Title](#_Toc32092)

# 编程规约

## 命名风格

1. 【强制】 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。  
   反例： \_name / \_\_name / $name / name\_ / name$ / name\_\_
2. 【强制】 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。  
   说明： 正确的英文拼写和语法可以让阅读者易于理解，避免歧义。注意，即使纯拼音命名方式也要避免采用。  
   反例： DaZhePromotion [打折] / getPingfenByName() [评分] / int 某变量 = 3
3. 【强制】类名使用 UpperCamelCase 风格，但以下情形例外： DO / BO / DTO / VO / AO /PO / UID 等。  
   正例： MarcoPolo / UserDO / XmlService / TcpUdpDeal / TaPromotion  
   反例： macroPolo / UserDo / XMLService / TCPUDPDeal / TAPromotion
4. 【强制】方法名、参数名、成员变量、局部变量都统一使用 lowerCamelCase 风格，必须遵从驼峰形式。  
   正例： localValue / getHttpMessage() / inputUserId
5. 【强制】常量命名全部大写，单词间用下划线隔开，力求语义表达完整清楚，不要嫌名字长。  
   正例： MAX\_STOCK\_COUNT  
   反例： MAX\_COUNT
6. 【强制】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。
7. 【强制】类型与中括号紧挨相连来表示数组。  
   正例： 定义整形数组 int[] arrayDemo;  
   反例： 在 main 参数中，使用 String args[]来定义。
8. 【强制】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。  
   反例： 定义为基本数据类型 Boolean isDeleted 的属性，它的方法也是 isDeleted()， RPC框架在反向解析的时候， “误以为” 对应的属性名称是 deleted，导致属性获取不到，进而抛出异常。
9. 【强制】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.unisound.ai.util、类名为 MessageUtils（此规则参考 spring的框架结构）
10. 【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。
11. 【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：在JDK中， 表达原子更新的类名为： AtomicReferenceFieldUpdater。  
    反例： 变量 int a 的随意命名方式。
12. 【推荐】如果模块、 接口、类、方法使用了设计模式，在命名时需体现出具体模式。  
    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;
13. 【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，肯定是与接口方法相关，并且是整个应用的基础常量。  
    正例： 接口方法签名 void commit();  
    接口基础常量 String COMPANY = "unisound";  
    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。
14. 接口和实现类的命名有两套规则：  
    1） 【强制】对于 Service 和 DAO 类，基于 SOA 的理念，暴露出来的服务一定是接口，内部的实现类用 Impl 的后缀与接口区别。  
    正例： CacheServiceImpl 实现 CacheService 接口。  
    2）【推荐】 如果是形容能力的接口名称，取对应的形容词为接口名（通常是–able 的形式）。  
    正例： AbstractTranslator 实现 Translatable 接口。
15. 【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类， 域成员均为常量， 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 / UNKNOWN\_REASON。
16. 【参考】各层命名规约：  
    A) Service/DAO 层方法命名规约  
     1） 获取单个对象的方法用 get 做前缀。  
     2） 获取多个对象的方法用 list 做前缀，复数形式结尾如： listObjects。  
     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。  
    B) 领域模型命名规约  
     1） 数据对象： xxxDO， xxx 即为数据表名。  
     2） 数据传输对象： xxxDTO， xxx 为业务领域相关的名称。  
     3） 展示对象： xxxVO， xxx 一般为网页名称。  
     4） POJO 是 DO/DTO/BO/VO 的统称，禁止命名成 xxxPOJO。

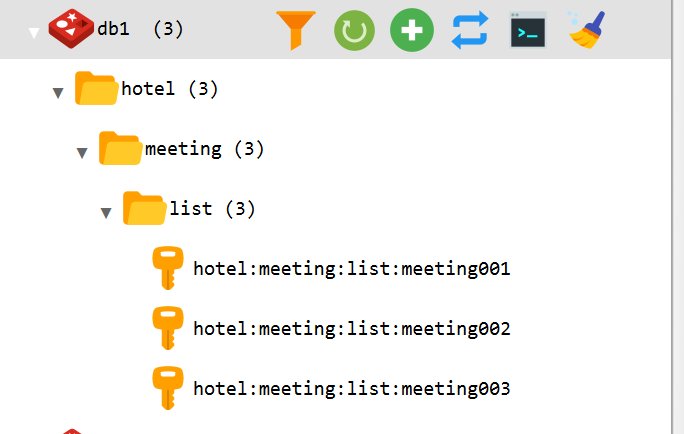
## 常量定义

1. 【强制】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。  
   反例： String key = "Id#unisound\_" + tradeId;  
   cache.put(key, value);
2. 【推荐】缓存redis键命名规范. 能表达大致意思, 并且用”:”冒号连接,不确定参数用 %s替代.

正例：

酒店服务 会议模块 会议记录集合 %s 为对应会议ID.

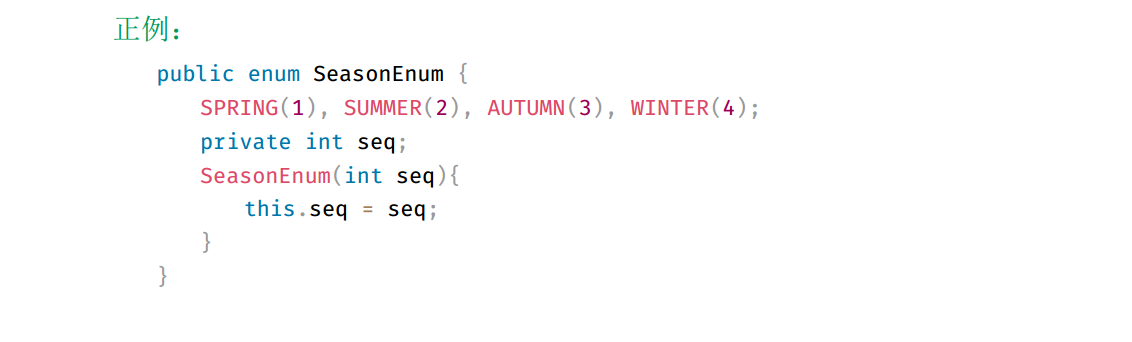
用redis客户端工具查看, 方便 易于管理.



注意: redisKey不宜太长 过长的key会严重影响缓存存取性能.

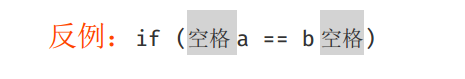
1. 【强制】 在 long 或者 Long 赋值时， 数值后使用大写的 L，不能是小写的 l，小写容易跟数字1 混淆，造成误解。  
   说明： Long a = 2l; 写的是数字的 21，还是 Long 型的 2?
2. 【推荐】不要使用一个常量类维护所有常量， 要按常量功能进行归类，分开维护。  
   说明： 大而全的常量类， 杂乱无章， 使用查找功能才能定位到修改的常量，不利于理解和维护。  
   正例： 缓存相关常量放在类 CacheConsts 下； 系统配置相关常量放在类 ConfigConsts 下。
3. 【推荐】常量的复用层次有五层：跨应用共享常量、应用内共享常量、子工程内共享常量、包内共享常量、类内共享常量。  
   1） 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。  
   2） 应用内共享常量：放置在一方库中， 通常是子模块中的 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。  
3） 子工程内部共享常量：即在当前子工程的 constant 目录下。  
4） 包内共享常量：即在当前包下单独的 constant 目录下。  
5） 类内共享常量：直接在类内部 private static final 定义。

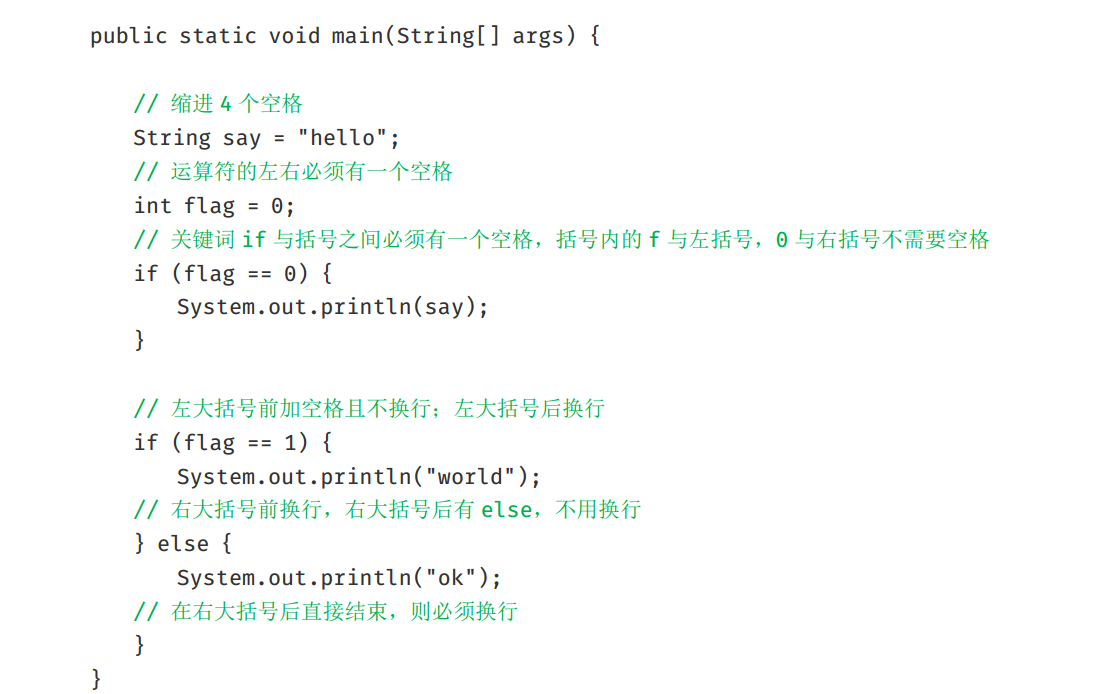
1. 【推荐】 如果变量值仅在一个固定范围内变化用 enum 类型来定义。  
   说明： 如果存在名称之外的延伸属性应使用 enum 类型，下面正例中的数字就是延伸信息，表示一年中的第几个季节.  
   

## 代码格式

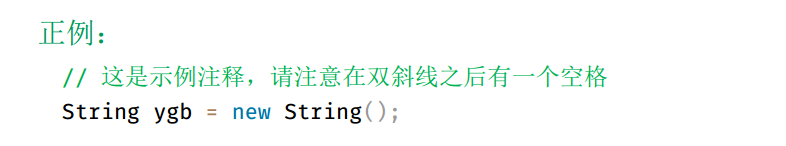
1. 【强制】大括号的使用约定。如果是大括号内为空，则简洁地写成{}即可，不需要换行； 如果  
   是非空代码块则：  
   1） 左大括号前不换行。  
   2） 左大括号后换行。  
   3） 右大括号前换行。  
   4） 右大括号后还有 else 等代码则不换行； 表示终止的右大括号后必须换行。
2. 【强制】左小括号和字符之间不出现空格； 同样，右小括号和字符之间也不出现空格；而左大  
   括号前需要空格。详见第 5 条下方正例提示。



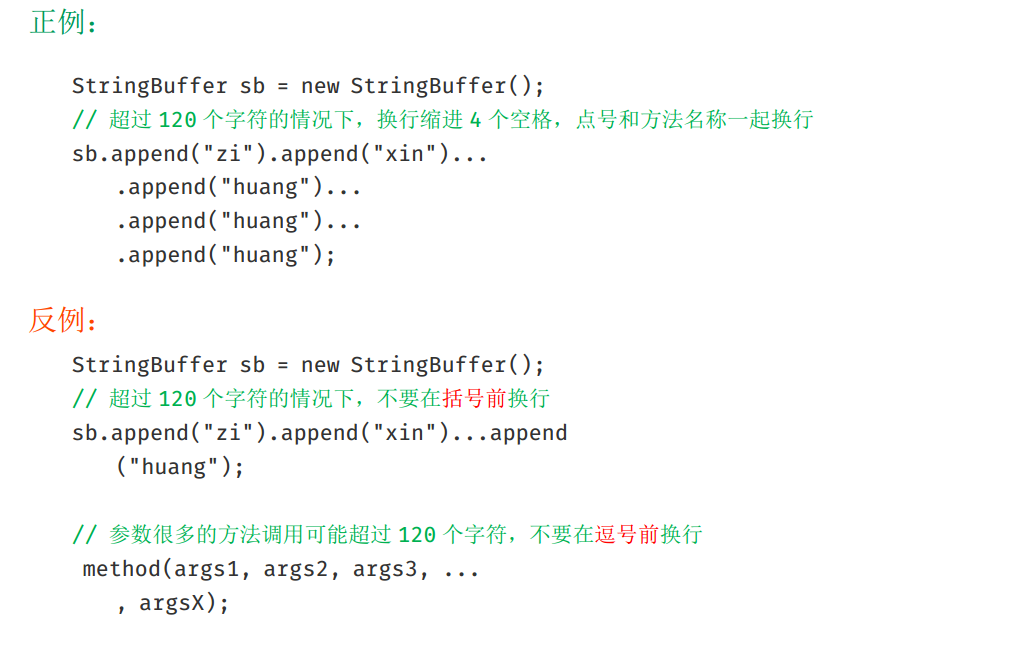
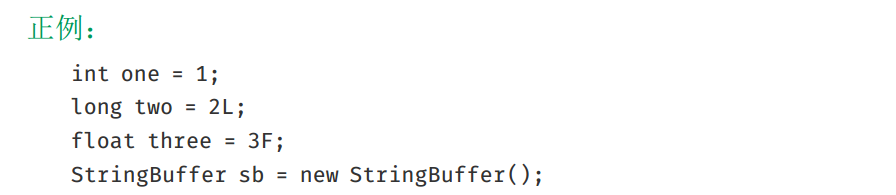
1. 【强制】 if/for/while/switch/do 等保留字与括号之间都必须加空格。
2. 【强制】任何二目、 三目运算符的左右两边都需要加一个空格。  
   说明： 运算符包括赋值运算符=、逻辑运算符&&、加减乘除符号等。
3. 【强制】 采用 4 个空格缩进，禁止使用 tab 字符。  
   说明： 如果使用 tab 缩进，必须设置 1 个 tab 为 4 个空格。 IDEA 设置 tab 为 4 个空格时，  
   请勿勾选 Use tab character；而在 eclipse 中，必须勾选 insert spaces for tabs。  
   正例： （涉及 1-5 点）



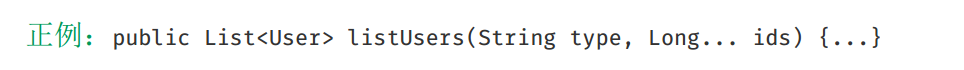
1. 【强制】注释的双斜线与注释内容之间有且仅有一个空格。

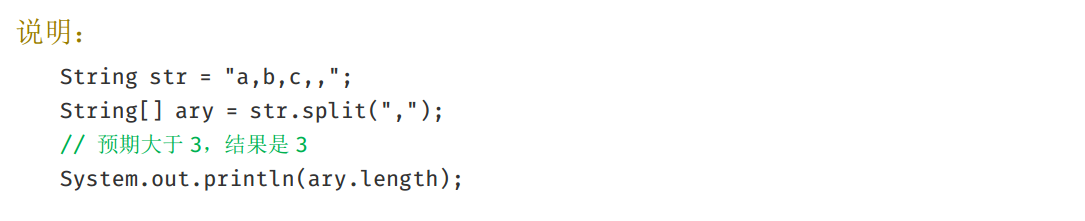


1. 【强制】单行字符数限制不超过 120 个，超出需要换行，换行时遵循如下原则：  
   1） 第二行相对第一行缩进 4 个空格，从第三行开始，不再继续缩进，参考示例。  
   2） 运算符与下文一起换行。  
   3） 方法调用的点符号与下文一起换行。  
   4） 方法调用中的多个参数需要换行时， 在逗号后进行。  
   5） 在括号前不要换行，见反例。

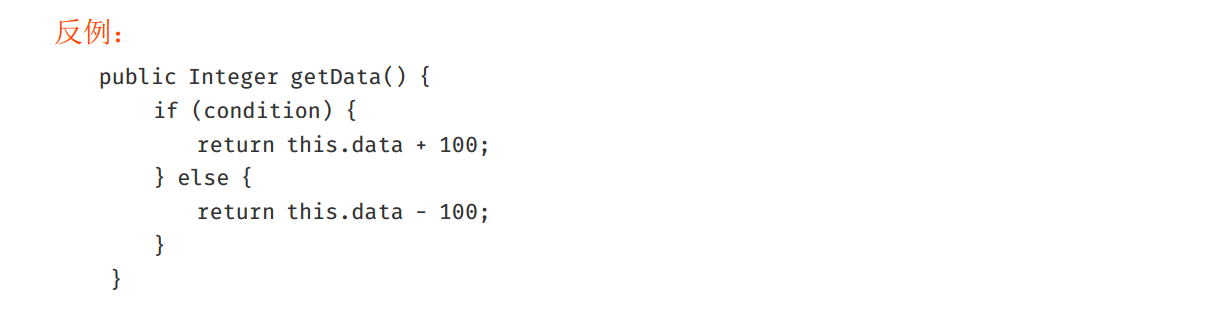
  
8. 【强制】方法参数在定义和传入时，多个参数逗号后边必须加空格。  
正例： 下例中实参的 args1， 后边必须要有一个空格。  
  
9. 【强制】 IDE 的 text file encoding 设置为 UTF-8; IDE 中文件的换行符使用 Unix 格式，不要使用 Windows 格式。  
10. 【推荐】单个方法的总行数不超过 80 行。  
说明： 包括方法签名、结束右大括号、方法内代码、注释、空行、回车及任何不可见字符的总行数不超过 80 行。  
正例： 代码逻辑分清红花和绿叶， 个性和共性，绿叶逻辑单独出来成为额外方法，使主干代码更加清晰； 共性逻辑抽取成为共性方法，便于复用和维护。  
11. 【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。  
  
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，是非常累赘的事情。  
12.【推荐】 不同逻辑、不同语义、不同业务的代码之间插入一个空行分隔开来以提升可读性。  
说明： 任何情形， 没有必要插入多个空行进行隔开。

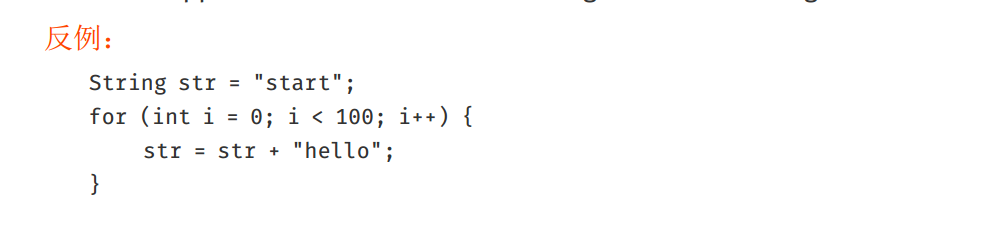
## OOP规约

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。
3. 【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。 （提倡同学们尽量不用可变参数编程）  
   
4. 【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。
5. 【强制】不能使用过时的类或方法。  
   说明： java.net.URLDecoder 中的方法 decode(String encodeStr) 这个方法已经过时，应该使用双参数 decode(String source, String encode)。接口提供方既然明确是过时接口，那么有义务同时提供新的接口； 作为调用方来说，有义务去考证过时方法的新实现是什么。
6. 【强制】 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）
7. 【强制】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 Integer var = ? 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。
8. 关于基本数据类型与包装数据类型的使用标准如下：  
   1） 【强制】 所有的 POJO 类属性必须使用包装数据类型。  
   2） 【强制】 RPC 方法的返回值和参数必须使用包装数据类型。  
   3） 【推荐】 所有的局部变量使用基本数据类型。  
   说明： POJO 类属性没有初值是提醒使用者在需要使用时，必须自己显式地进行赋值，任何NPE(空指针) 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。
9. 【强制】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提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。
10. 【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。
11. 【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。
12. 【强制】 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。  
    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。
13. 【强制】禁止在 POJO 类中，同时存在对应属性 xxx 的 isXxx()和 getXxx()方法。  
    说明： 框架在调用属性 xxx 的提取方法时，并不能确定哪个方法一定是被优先调用到。
14. 【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



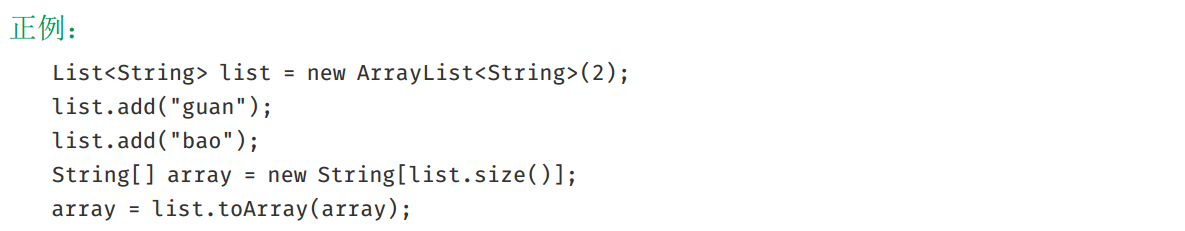
1. 【推荐】当一个类有多个构造方法，或者多个同名方法，这些方法应该按顺序放置在一起，便于阅读，此条规则优先于第 16 条规则。
2. 【推荐】 类内方法定义的顺序依次是：公有方法或保护方法 > 私有方法 > getter/setter方法(推荐使用lombok 不用写getter/setter方法 编译之后自动生成)。  
   说明： 公有方法是类的调用者和维护者最关心的方法，首屏展示最好； 保护方法虽然只是子类关心，也可能是“模板设计模式”下的核心方法； 而私有方法外部一般不需要特别关心，是一个黑盒实现； 因为承载的信息价值较低，所有 Service 和 DAO 的 getter/setter 方法放在类体最后。
3. 【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。



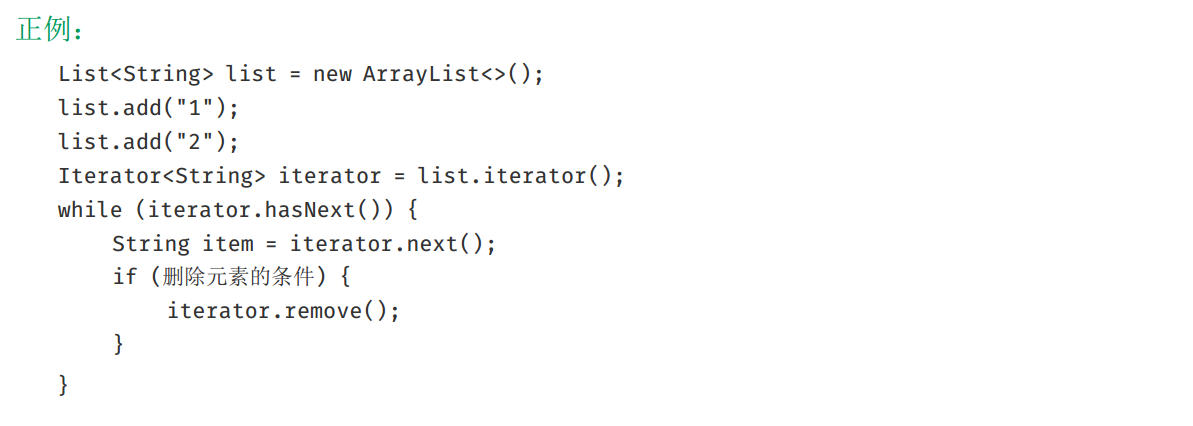
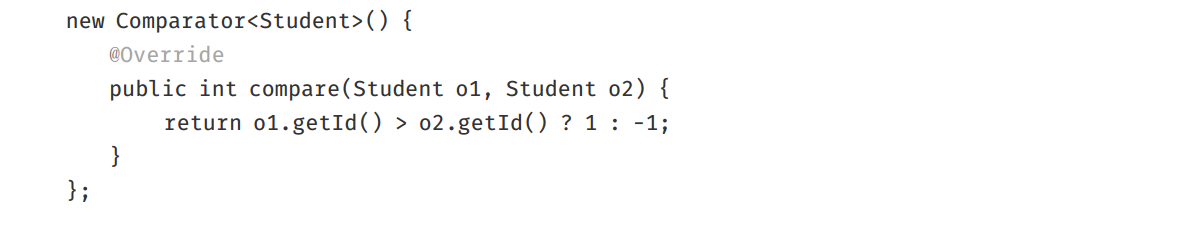
1. 【推荐】循环体内，字符串的连接方式，使用 StringBuilder 的 append 方法进行扩展。  
   说明： 下例中， 反编译出的字节码文件显示每次循环都会 new 出一个 StringBuilder 对象，然后进行 append 操作，最后通过 toString 方法返回 String 对象，造成内存资源浪费。  
   
2. 【推荐】 final 可以声明类、成员变量、方法、以及本地变量，下列情况使用 final 关键字：  
   1） 不允许被继承的类，如： String 类。  
   2） 不允许修改引用的域对象。  
   3） 不允许被重写的方法，如： POJO 类的 setter 方法。  
   4） 不允许运行过程中重新赋值的局部变量。  
   5） 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。
3. 【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 方法实现域对象的深度遍历式拷贝。
4. 【推荐】类成员与方法访问控制从严：  
   1） 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。  
   2） 工具类不允许有 public 或 default 构造方法。  
   3） 类非 static 成员变量并且与子类共享，必须是 protected。  
   4） 类非 static 成员变量并且仅在本类使用，必须是 private。  
   5） 类 static 成员变量如果仅在本类使用，必须是 private。  
   6） 若是 static 成员变量， 考虑是否为 final。  
   7） 类成员方法只供类内部调用，必须是 private。  
   8） 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 成员方法或成员变量，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：  
   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。  
   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。
2. 【强制】 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。
3. 【强制】在 subList 场景中， 高度注意对原集合元素的增加或删除， 均会导致子列表的遍历、增加、删除产生 ConcurrentModificationException 异常。
4. 【强制】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。



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】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。
2. 【强制】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。  
     
   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？
3. 【强制】 在 JDK7 版本及以上， Comparator 实现类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） x， y 的比较结果和 y， x 的比较结果相反。  
   2） x>y， y>z， 则 x>z。  
   3） 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：  
   
4. 【推荐】 集合泛型定义时， 在 JDK7 及以上，使用 diamond 语法或全省略。  
   说明： 菱形泛型，即 diamond， 直接使用<>来指代前边已经指定的类型。  
   正例：  
   // <> diamond 方式  
   HashMap<String, String> userCache = new HashMap<>(16);  
   // 全省略方式  
   ArrayList<User> users = new ArrayList(10);
5. 【推荐】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 loaderFactor） 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。
6. 【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 都放到了 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
7. 【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：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 集合类 | Key | Value | super | 说明 |
| Hashtable | 不允许为 null | 不允许为 null | Dictionary | 线程安全 |
| ConcurrentHashMap | 不允许为 null | 不允许为 null | AbstractMap | 锁分段技术（JDK8:CAS） |
| TreeMap | 不允许为 null | 允许为 null | AbstractMap | 线程不安全 |
| HashMap | 允许为 null | 允许为 null | AbstractMap | 线程不安全 |

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。
2. 【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。
3. 【参考】lambda 对集合去重.

List<TestDemo> resList = list.stream().collect(Collectors.collectingAndThen(

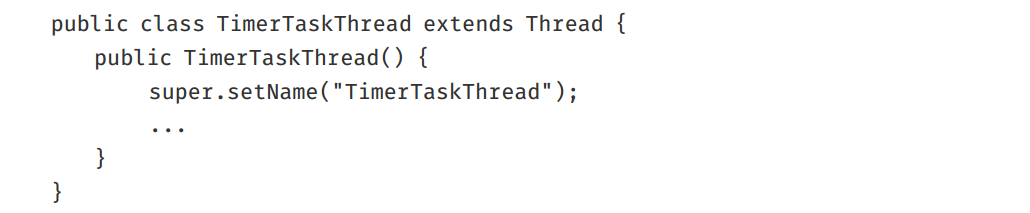
Collectors.toCollection(()-> new TreeSet<>(Comparator.comparing(c-> c.getId()))), ArrayList::new));

1. 【参考】lambda 对集合进行分组.

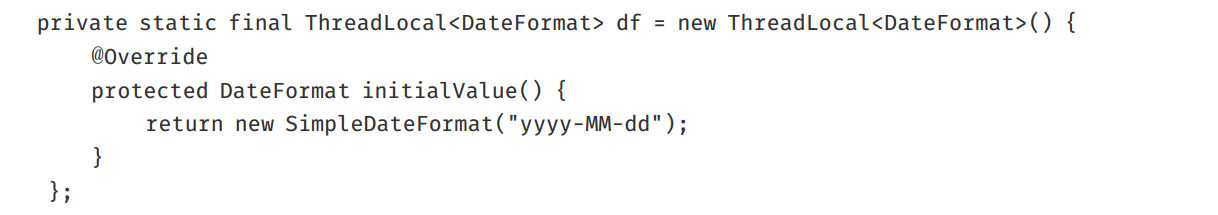
Map<String, List<TestDemo>> map = list.stream().collect(Collectors.groupingBy(TestDemo::getId));

## 并发处理

1. 【强制】 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。
2. 【强制】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：

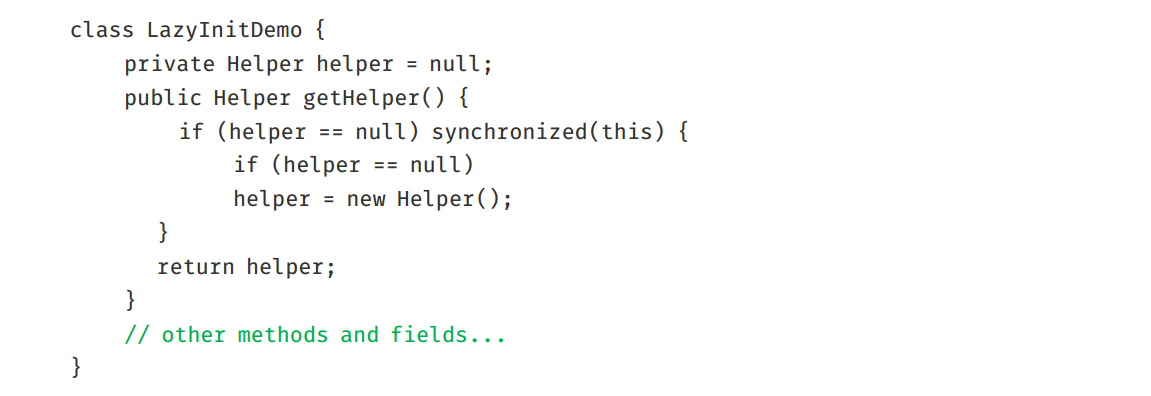


1. 【强制】线程资源必须通过线程池提供，不允许在应用中自行显式创建线程。  
   说明： 使用线程池的好处是减少在创建和销毁线程上所消耗的时间以及系统资源的开销，解决资源不足的问题。如果不使用线程池，有可能造成系统创建大量同类线程而导致消耗完内存或者“过度切换”的问题。
2. 【强制】线程池不允许使用 Executors 去创建，而是通过 ThreadPoolExecutor 的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险。  
   说明： Executors 返回的线程池对象的弊端如下：  
   1） FixedThreadPool 和 SingleThreadPool:  
   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。  
   2） 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。
3. 【强制】 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：



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。
2. 【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。
3. 【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。
4. 【强制】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。
5. 【推荐】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。
6. 【推荐】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。
7. 【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。  
   反例：



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） 。
2. 【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
3. 【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。

## 控制语句

1. 【强制】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. 【强制】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. 【强制】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。  
说明： 如果并发控制没有处理好，容易产生等值判断被“击穿”的情况，使用大于或小于的区间判断条件来代替。  
反例： 判断剩余奖品数量等于 0 时，终止发放奖品，但因为并发处理错误导致奖品数量瞬间变成了负数， 这样的话，活动无法终止。  
4. 【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(“do something…”);  
 return;  
}  
5. 【推荐】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。  
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) && (...) || (...)) {  
 ...  
}  
6. 【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）。  
7. 【推荐】避免采用取反逻辑运算符。  
说明： 取反逻辑不利于快速理解，并且取反逻辑写法必然存在对应的正向逻辑写法。  
正例： 使用 if (x < 628) 来表达 x 小于 628。  
反例： 使用 if (!(x >= 628)) 来表达 x 小于 628。  
8. 【推荐】接口入参保护，这种场景常见的是用作批量操作的接口。  
9. 【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） 对外提供的开放接口，不管是 RPC/API/HTTP 接口。  
5） 敏感权限入口。  
10.【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。

## 注释规约

## 其他