目的

本规范的目的是使本组织能以标准的、规范的方式设计和编码。通过建立编码规范，使每个开发人员养成良好的编码风格和习惯；并以此形成开发小组编码规定，提高程序的可靠性、可读性、可维护性和一致性等，增进团队间的交流，并保证软件产品的质量。

插件安装

1. Lombok
2. alibaba代码规约工具
3. Sonar扫描

日志规范

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，有利于维护和个各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);

2. 【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。
3. 【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，如 stats / monitor / access 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。

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。

4. 【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：（条件）建设采用如下方式

if (logger.isDebugEnabled()) {

logger.debug("Processing trade with id: " + id + " and symbol: " + symbol);

}

正例：（占位符）

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

5. 【强制】避免重复打印日志，浪费磁盘空间，务必在 log4 j.xml 中设置 additivity = false。

正例：

6. 【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(各类参数或者对象toString() + "_" + e.getMessage(), e);

7. 【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，不利于系统性能提升，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？

8. 【推荐】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。如非必要，请不要在此场景打出 error 级别，避免频繁报警。

说明：注意日志输出的级别，error级别只记录系统逻辑出错、异常或者重要的错误信息

9. 【推荐】尽量用英文来描述日志错误信息，如果日志中的错误信息用英文描述不清楚的话使用中文描述即可，否则容易产生歧义。国际化团队或海外部署的服务器由于字符集问题，【强制】使用全英文来注释和描述日志错误信息。

10.【介绍】日志级别

（1） ERROR：系统发生了错误事件，但仍然不影响系统的继续运行。系统需要将错误或异常细节记录ERROR日志中，方便后续人工回溯解决。

（2） WARN：系统在业务处理时触发了异常流程，但系统可恢复到正常态，下一次业务可以正常执行。如程序调用了一个旧版本的接口，可选参数不合法，非业务预期的状态但仍可继续处理等。

（3） INFO：记录系统关键信息，旨在保留系统正常工作期间关键运行指标，开发人员可以将初始化系统配置、业务状态变化信息，或者用户业务流程中的核心处理记录到INFO日志中，方便日常运维工作以及错误回溯时上下文场景复现。

（4） DEBUG：可以将各类详细信息记录到DEBUG里，起到调试的作用，包括参数信息，调试细节信息，返回值信息等等。

（5）TRACE：更详细的跟踪信息。

注意：上述日志级别从高到低排列，是开发中最常用的五种。生产系统一般只打印INFO 级别以上的日志，对于 DEBUG 级别的日志，只在测试环境中打印。打印错误日志时，需要区分是业务异常(如:用户名不能为空)还是系统异常(如:调用会员核心异常)，业务异常使用 warn 级别记录，系统异常使用 error 记录。

编程规范

1. 命名 命名风格 风格

1.1【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__

1.2【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。

正例： alibaba / taobao / youku / hangzhou 等国际通用的名称，可视同英文。

反例： DaZhePromotion [打折] / getPingfenByName() [评分] / int 某变量 = 3

1.3【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

1.4【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

1.5【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

1.6【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。

1.7 【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。

1.8【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。

1.9 【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 . alibaba . ai . util 、类名为 MessageUtils （此规则参考 spring的框架结构）

1.10 【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。

1.11【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。

1.12【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;

1.13【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 = " alibaba " ;

反例：接口方法定义 public abstract void f();

说明： JDK 8 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。

1.14 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 。

1.15【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 。

1.16【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 。

2 常量定义

2.1【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 # taobao _" + tradeId;

cache . put(key , value);

2.2【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 = 2 l; 写的是数字的 21，还是 Long 型的 2?

2.3【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。

2.4【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。

2.5【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。

正例：

public enum SeasonEnum {

SPRING(1), SUMMER(2), AUTUMN(3), WINTER(4);

private int seq;

SeasonEnum(int seq){

this.seq = seq;

}

}

3 代码 代码格式 格式

3.1【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。

3.2【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。

反例： if (空格 a == b 空格)

3.3【强制】 if / for / while / switch / do 等保留字与括号之间都必须加空格。

3.4【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。

3.5【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
}
}
```

3.6【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：

```
// 这是示例注释，请注意在双斜线之后有一个空格
String ygb = new String();
```

3.7【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。

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);
```

3.8【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。

```
  method(args1, args2, args3);
```

3.9【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。

3.10【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。

3.11【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。

正例：

```
int one = 1;

long two = 2L;

float three = 3F;

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，是非常累赘的事情。

3.12【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。

4 OOP 规约

4.1【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。

4.2【强制】所有的覆写方法，必须加@ Override 注解。

说明： getObject() 与 get 0 bject() 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。

4.3【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）

正例： public List listUsers(String type, Long... ids) {...}

4.4【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。

4.5【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。

4.6【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（JDK 7 引入的工具类）

4.7【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。

4.8 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 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。

4.9【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。

4.10【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。

4.11【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。

4.12【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。

4.13【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。

4.14 【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。

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);

4.15【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。

4.16【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。

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。

4.17【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 (condition) {

return this.data + 100;

} else {

return this.data - 100;

}

}

4.18【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。

反例：

String str = "start";

```
for (int i = 0; i < 100; i++) {
str = str + "hello";
}
```

4.19【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。

4.20【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。

4.21【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。

5 集合处理

5.1【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。

5.2【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。

5.3【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。

5.4【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。

正例：

List list = new ArrayList(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。

5.5【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。

5.6【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，作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：第一、频繁往外读取内容的，适合用<?extends T >。第二、经常往里插入的，适合用 <? super T> 。

5.7【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。

正例：

List list = new ArrayList<>();

list.add("1");

list.add("2");

Iterator iterator = list.iterator();

while (iterator.hasNext()) {

String item = iterator.next();

```
if (删除元素的条件) {
iterator.remove();
}
}
```

反例：

```
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？

5.8【强制】 在 JDK 7 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：

```
new Comparator() {
@Override
public int compare(Student o1, Student o2) {
return o1.getId() > o2.getId() ? 1 : -1;
}
};
```

5.9【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 users = new ArrayList(10);
```

5.10【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。

5.11【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 JDK 8，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。

5.12【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
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。

5.13【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 。

5.14【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。

6 并发处理

6.1【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。

6.2【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：

```
public class TimerTaskThread extends Thread {

public TimerTaskThread() {

super.setName("TimerTaskThread");

...

}

}
```

6.3【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。

6.4【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 。

6.5【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：

private static final ThreadLocal df = new ThreadLocal() {

@ Override

protected DateFormat initialValue() {

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

}

};

说明：如果是 JDK 8 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 。

6.6【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。

6.7【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。

6.8【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。

6.9【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。

6.10【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。

6.11【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 JDK 7 之后，可以直接使用 API ThreadLocalRandom ，而在 JDK 7 之前，需要编码保证每个线程持有一个实例。

6.12【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 JDK 5 及以上版本 ） ，将目标属性声明为 volatile 型 。

反例：

class LazyInitDemo {

private Helper helper = null;

public Helper getHelper() {

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

if (helper == null)

helper = new Helper();

}

return helper;

}

// other methods and fields...

}

6.13【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 JDK 8，推荐使用 LongAdder 对象，比 AtomicLong 性能更好 （ 减少乐观锁的重试次数 ） 。

6.14【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。

6.15【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。

7 控制语句

7.1【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。

7.2【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;

7.3【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。

7.4【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(“stay at home to learn Alibaba Java Coding Guidelines.”);

return;

}
```

7.5【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) && (...) || (...)) {
```

…

}

7.6【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） 。

7.7【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。

7.8【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。

7.9【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。

7.10【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。

8 注释规约

8.1【强制】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 方式会提示相关注释，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提示方法、参数、返回值的意义，提高阅读效率。

8.2【强制】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。

8.3【强制】所有的类都必须添加创建者和创建日期。

8.4【强制】方法内部单行注释，在被注释语句上方另起一行，使用//注释。方法内部多行注释使用/* */注释，注意与代码对齐。

8.5【强制】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。

8.6【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。

8.7【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。

8.8【参考】谨慎注释掉代码。在上方详细说明，而不是简单地注释掉。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 ） 。

8.9 【参考】对于注释的要求：第一、能够准确反应设计思想和代码逻辑； 第二、能够描述业务含义，使别的程序员能够迅速了解到代码背后的信息。完全没有注释的大段代码对于阅读者形同天书，注释是给自己看的，即使隔很长时间，也能清晰理解当时的思路； 注释也是给继任者看

的，使其能够快速接替自己的工作。

8.10【参考】好的命名、代码结构是自解释的，注释力求精简准确、表达到位。避免出现注释的一个极端：过多过滥的注释，代码的逻辑一旦修改，修改注释是相当大的负担。

反例：

// put elephant into fridge

put(elephant, fridge);

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

8.11【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，通过标记扫描， 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。

1） 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 ） 。

2） 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。

9 其它

9.1【强制】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 ”);

9.2【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。

9.3【强制】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 等于 null 或者不存在，那么 ${var} 会直接显示在页面上。

9.4【强制】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。

9.5【强制】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 JDK 8 中，针对统计时间等场景，推荐使用 Instant 类。

9.6【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。

9.7【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。

9.8【推荐】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

Redis规范

\1. 键值设计

1.1 key名设计

(1）可读性和可管理性**

以业务名(或数据库名)为前缀(防止key冲突)，用冒号分隔，比如 业务名:表名:id

ugc:video:1

(2）简洁性**

保证语义的前提下，控制key的长度，当key较多时，内存占用也不容忽视，例如：

user:{uid}:friends:messages:{mid}简化为u:{uid}frm:{mid}。

(3）不要包含特殊字符**

反例：包含空格、换行、单双引号以及其他转义字符

1.2 value设计

(1）拒绝bigkey**

防止网卡流量、慢查询，string类型控制在10KB以内，hash、list、set、zset元素个数不要超过5000。

反例：一个包含200万个元素的list。

非字符串的bigkey，不要使用del删除，使用hscan、sscan、zscan方式渐进式删除，同时要注意防止bigkey过期时间自动删除问题(例如一个200万的zset设置1小时过期，会触发del操作，造成阻塞，而且该操作不会不出现在慢查询中(latency可查))，查找方法和删除方法

(2）选择适合的数据类型**

例如：实体类型(要合理控制和使用数据结构内存编码优化配置,例如ziplist，但也要注意节省内存和性能之间的平衡)。了解下，Redis 为什么这么快?

反例：

setuser:1:name tom

setuser:1:age 19

setuser:1:favor football

正例：hmset user:1 name tom age19 favor football

(3）控制key的生命周期**

redis不是垃圾桶，建议使用expire设置过期时间(条件允许可以打散过期时间，防止集中过期)，不过期的数据重点关注idletime。

\2. 命令使用

2.1 O(N)命令关注N的数量

例如hgetall、lrange、smembers、zrange、sinter等并非不能使用，但是需要明确N的值。有遍历的需求可以使用hscan、sscan、zscan代替。

2.2 禁用命令

禁止线上使用keys、flushall、flushdb等，通过redis的rename机制禁掉命令，或者使用scan的方式渐进式处理。

2.3 合理使用select

redis的多数据库较弱，使用数字进行区分，很多客户端支持较差，同时多业务用多数据库实际还是单线程处理，会有干扰。

2.4 使用批量操作提高效率

（1）原生命令：例如mget、mset。

（2）非原生命令：可以使用pipeline提高效率。

但要注意控制一次批量操作的元素个数(例如500以内，实际也和元素字节数有关)。

注意两者不同：

原生是原子操作，pipeline是非原子操作

pipeline可以打包不同的命令，原生做不到

pipeline需要客户端和服务端同时支持。

2.5 不建议过多使用Redis事务功能

Redis的事务功能较弱(不支持回滚)，而且集群版本(自研和官方)要求一次事务操作的key必须在一个slot上(可以使用hashtag功能解决)。分布式事务不理解？一次给你讲清楚！

2.6 Redis集群版本在使用Lua上有特殊要求

（1）所有key都应该由 KEYS 数组来传递，redis.call/pcall 里面调用的redis命令，key的位置，必须是KEYS array, 否则直接返回error，"-ERR bad lua script for redis cluster, all the keys that the script uses should be passed using the KEYS arrayrn"

（2） 所有key，必须在1个slot上，否则直接返回error, "-ERR eval/evalsha command keys must in same slotrn"

2.7 monitor命令

必要情况下使用monitor命令时，要注意不要长时间使用。

\3. 客户端使用

3.1 避免多个应用使用一个Redis实例

不相干的业务拆分，公共数据做服务化。

3.2 使用连接池

可以有效控制连接，同时提高效率，标准使用方式：

Jedis jedis = null;

```
try {
  jedis = jedisPool.getResource();
  //具体的命令
  jedis.executeCommand()
} catch (Exception e) {
  logger.error("op key {} error: " + e.getMessage(), key, e);
} finally {
  //注意这里不是关闭连接，在JedisPool模式下，Jedis会被归还给资源池。
  if(jedis != null)
    jedis.close();
}
```

3.3 熔断功能

高并发下建议客户端添加熔断功能(例如netflix hystrix)

3.4 合理的加密

设置合理的密码，如有必要可以使用SSL加密访问（阿里云Redis支持）

3.5 淘汰策略

根据自身业务类型，选好maxmemory-policy(最大内存淘汰策略)，设置好过期时间。默认策略是volatile-lru，即超过最大内存后，在过期键中使用lru算法进行key的剔除，保证不过期数据不被删除，但是可能会出现OOM问题。

其他策略如下：

allkeys-lru：根据LRU算法删除键，不管数据有没有设置超时属性，直到腾出足够空间为止。

allkeys-random：随机删除所有键，直到腾出足够空间为止。

volatile-random:随机删除过期键，直到腾出足够空间为止。

volatile-ttl：根据键值对象的ttl属性，删除最近将要过期数据。如果没有，回退到noeviction策略。

noeviction：不会剔除任何数据，拒绝所有写入操作并返回客户端错误信息"(error) OOM command not allowed when used memory"，此时Redis只响应读操作。

\4. 相关工具

4.1 数据同步

redis间数据同步可以使用：redis-port

4.2 big key搜索

redis大key搜索工具

4.3 热点key寻找

内部实现使用monitor，所以建议短时间使用facebook的redis-faina 阿里云Redis已经在内核层面解决热点key问题

\5. 删除bigkey

5.1 下面操作可以使用pipeline加速。

5.2 redis 4.0已经支持key的异步删除，欢迎使用。

（1）Hash删除: hscan + hdel

```
publicvoid delBigHash(String host, int port, String password, String bigHashKey) {
  Jedis jedis = new Jedis(host, port);
  if(password != null && !"".equals(password)) {
    jedis.auth(password);
  }
  ScanParams scanParams = new ScanParams().count(100);
  String cursor = "0";
  do {
    ScanResult<Entry<String, String>> scanResult = jedis.hscan(bigHashKey, cursor, scanParams);
    List<Entry<String, String>> entryList = scanResult.getResult();
    if(entryList != null && !entryList.isEmpty()) {
     for (Entry<String, String> entry : entryList) {
        jedis.hdel(bigHashKey, entry.getKey());
      }
    }
    cursor = scanResult.getStringCursor();
  } while(!"0".equals(cursor));
  //删除bigkey
  jedis.del(bigHashKey);
}
```

（2）List删除: ltrim

```
publicvoid delBigList(String host, int port, String password, String bigListKey) {
  Jedis jedis = new Jedis(host, port);
  if(password != null && !"".equals(password)) {
    jedis.auth(password);
  }
  longllen = jedis.llen(bigListKey);
  intcounter = 0;
  intleft = 100;
  while(counter < llen) {
    //每次从左侧截掉100个
    jedis.ltrim(bigListKey, left, llen);
```

```
    counter += left;
  }
  //最终删除key
  jedis.del(bigListKey);
}
```

（3）Set删除: sscan + srem

```
publicvoid delBigSet(String host, int port, String password, String bigSetKey) {
  Jedis jedis = new Jedis(host, port);
  if(password != null && !"".equals(password)) {
    jedis.auth(password);
  }
  ScanParams scanParams = new ScanParams().count(100);
  String cursor = "0";
  do {
    ScanResult scanResult = jedis.sscan(bigSetKey, cursor, scanParams);
    List memberList = scanResult.getResult();
    if(memberList != null && !memberList.isEmpty()) {
     for (String member : memberList) {
        jedis.srem(bigSetKey, member);
      }
    }
    cursor = scanResult.getStringCursor();
  } while(!"0".equals(cursor));
  //删除bigkey
  jedis.del(bigSetKey);
}
```

（4）SortedSet删除: zscan + zrem

```
publicvoid delBigZset(String host, int port, String password, String bigZsetKey) {
  Jedis jedis = new Jedis(host, port);
  if(password != null && !"".equals(password)) {
    jedis.auth(password);
  }
  ScanParams scanParams = new ScanParams().count(100);
  String cursor = "0";
```

```
  do {
    ScanResult scanResult = jedis.zscan(bigZsetKey, cursor, scanParams);
    List tupleList = scanResult.getResult();
    if(tupleList != null && !tupleList.isEmpty()) {
     for (Tuple tuple : tupleList) {
        jedis.zrem(bigZsetKey, tuple.getElement());
      }
    }
    cursor = scanResult.getStringCursor();
  } while(!"0".equals(cursor));
  //删除bigkey
  jedis.del(bigZsetKey);
}
```

补充:

1. 建议全部大写
2. key不能太长也不能太短,键名越长越占资源，太短可读性太差
3. key 单词与单词之间以 ： 分开

mysql规范

1.命名 命名风格 风格

1.1【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__

1.2【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。

正例： alibaba / taobao / youku / hangzhou 等国际通用的名称，可视同英文。

反例： DaZhePromotion [打折] / getPingfenByName() [评分] / int 某变量 = 3

1.3【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

1.4【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

1.5【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

1.6【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。

1.7 【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。

1.8【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。

1.9 【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 . alibaba . ai . util 、类名为 MessageUtils （此规则参考 spring的框架结构）

1.10 【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。

1.11【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。

1.12【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;

1.13【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 = " alibaba " ;

反例：接口方法定义 public abstract void f();

说明： JDK 8 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。

1.14 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 。

1.15【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 。

1.16【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 。

2 常量定义

2.1【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 # taobao _" + tradeId;

cache . put(key , value);

2.2【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 = 2 l; 写的是数字的 21，还是 Long 型的 2?

2.3【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。

2.4【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。

2.5【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。

正例：

public enum SeasonEnum {

SPRING(1), SUMMER(2), AUTUMN(3), WINTER(4);

private int seq;

SeasonEnum(int seq){

this.seq = seq;

}

}

3 代码 代码格式 格式

3.1【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。

3.2【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。

反例： if (空格 a == b 空格)

3.3【强制】 if / for / while / switch / do 等保留字与括号之间都必须加空格。

3.4【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。

3.5【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
}
}
```

3.6【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：

```
// 这是示例注释，请注意在双斜线之后有一个空格
String ygb = new String();
```

3.7【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。

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);

3.8【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。

method(args1, args2, args3);

3.9【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。

3.10【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。

3.11【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。

正例：

int one = 1;

long two = 2L;

float three = 3F;

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，是非常累赘的事情。

3.12【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。

4 OOP 规约

4.1【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。

4.2【强制】所有的覆写方法，必须加@ Override 注解。

说明： getObject() 与 get 0 bject() 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。

4.3【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）

正例： public List listUsers(String type, Long... ids) {...}

4.4【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。

4.5【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。

4.6【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（JDK 7 引入的工具类 ）

4.7【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。

4.8 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 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。

4.9【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。

4.10【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。

4.11【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。

4.12【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。

4.13【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。

4.14 【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。

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);

4.15【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。

4.16【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。

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。

4.17【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 (condition) {

return this.data + 100;

} else {

return this.data - 100;

}

}

4.18【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。

反例：

String str = "start";

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

str = str + "hello";

}

4.19【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。

4.20【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。

4.21【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。

5 集合处理

5.1【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。

5.2【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。

5.3【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。

5.4【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。

正例：

List list = new ArrayList(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。

5.5【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。

5.6【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，作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：第一、频繁往外读取内容的，适合用<?extends T >。第二、经常往里插入的，适合用 <? super T> 。

5.7【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。

正例：

List list = new ArrayList<>();

list.add("1");

list.add("2");

Iterator iterator = list.iterator();

while (iterator.hasNext()) {

String item = iterator.next();

if (删除元素的条件) {

iterator.remove();

}

}

反例：

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？

5.8【强制】 在 JDK 7 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：

```
new Comparator() {

@Override

public int compare(Student o1, Student o2) {

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

}

};
```

5.9【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 users = new ArrayList(10);
```

5.10【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。

5.11【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 JDK 8，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。

5.12【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
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。

5.13【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 。

5.14【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。

6 并发处理

6.1【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。

6.2【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：

public class TimerTaskThread extends Thread {

public TimerTaskThread() {

super.setName("TimerTaskThread");

…

}

}

6.3【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。

6.4【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 。

6.5【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：

private static final ThreadLocal df = new ThreadLocal() {

@ Override

protected DateFormat initialValue() {

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

}

};

说明：如果是 JDK 8 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 。

6.6【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。

6.7【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。

6.8【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。

6.9【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。

6.10【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。

6.11【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 JDK 7 之后，可以直接使用 API ThreadLocalRandom ， 而在 JDK 7 之前，需要编码保证每个线程持有一个实例。

6.12【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 JDK 5 及以上版本 ） ，将目标属性声明为 volatile 型 。

反例：

```
class LazyInitDemo {

private Helper helper = null;

public Helper getHelper() {

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

if (helper == null)

helper = new Helper();

}

return helper;

}

// other methods and fields...

}
```

6.13【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 JDK 8，推荐使用 LongAdder 对象，比 AtomicLong 性能更好 （减少乐观锁的重试次数） 。

6.14【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。

6.15【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。

7 控制语句

7.1【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。

7.2【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;

7.3【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。

7.4【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(“stay at home to learn Alibaba Java Coding Guidelines.”);

return;

}

7.5【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) && (...) || (...)) {

...

}

7.6【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） 。

7.7【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。

7.8【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。

7.9【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。

7.10【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。

8 注释规约

8.1【强制】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 方式会提示相关注释，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提示方法、参数、返回值的意义，提高阅读效率。

8.2【强制】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。

8.3【强制】所有的类都必须添加创建者和创建日期。

8.4【强制】方法内部单行注释，在被注释语句上方另起一行，使用//注释。方法内部多行注释使用/* */注释，注意与代码对齐。

8.5【强制】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。

8.6【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。

8.7【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。

8.8【参考】谨慎注释掉代码。在上方详细说明，而不是简单地注释掉。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） 。

8.9【参考】对于注释的要求：第一、能够准确反应设计思想和代码逻辑； 第二、能够描述业务含义，使别的程序员能够迅速了解到代码背后的信息。完全没有注释的大段代码对于阅读者形同天书，注释是给自己看的，即使隔很长时间，也能清晰理解当时的思路； 注释也是给继任者看

的，使其能够快速接替自己的工作。

8.10【参考】好的命名、代码结构是自解释的，注释力求精简准确、表达到位。避免出现注释的一个极端：过多过滥的注释，代码的逻辑一旦修改，修改注释是相当大的负担。

反例：

// put elephant into fridge

put(elephant, fridge);

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

8.11【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，通过标记扫描， 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。

1 ） 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 ） 。

2 ） 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。

9 其它

9.1【强制】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 ”);

9.2【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。

9.3【强制】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 等于 null 或者不存在，那么 ${var} 会直接显示在页面上。

9.4【强制】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。

9.5【强制】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 JDK 8 中，针对统计时间等场景，推荐使用 Instant 类。

9.6【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。

9.7【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。

9.8【推荐】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

git message 提交规范

1. 提交内容格式

主要分为三部分: 提交头Header、提交内容体Body、提交内容尾说明Footer

注意: messgage中的冒号是 英文半角符号的冒号(:)，而不是 中文冒号(：)，冒号后多一个空格

英文格式:

```
<type>(<scope>): <subject>
<BLANK LINE>
<body>
<BLANK LINE>
<footer>
```

中文格式:

```
[提交类型](影响范围): 提交内容的简要描述
// 空一行
详细内容，可以多行
// 空一行
尾部内容，可以多行
```

2. 提交内容类型

> Angular规范中限制的类型

1. feat：新功能（feature）
2. fix：修补bug
3. docs：文档（documentation）
4. style： 格式（不影响代码运行的变动）
5. refactor：重构（即不是新增功能，也不是修改bug的代码变动）
6. test：增加测试
7. chore：构建过程或辅助工具的变动

> 推荐中文自定义类型

1. [新增功能]
2. 1. 新增需求、功能代码
3. [修改功能]
4. 1. 修改需求、功能代码
5. [删除功能]
6. 1. 撤销需求、删除功能代码
7. [重构代码]
8. 1. 优化代码、重构项目结构、代码结构，不影响功能
9. [修改配置]
10. 1. 调整配置文件、添加、删除、修改服务配置
11. [修改依赖]
12. 1. 更新maven依赖、其他环境依赖
13. [修复缺陷]
14. 1. 修复BUG
15. [回滚提交]
16. 1. 回滚代码，回滚到指定版本
17. [新增测试]
18. 1. 新增测试用例代码
19. [新增文档]
20. 1. 新增相关说明文档
21. [其他修改]
22. 1. 非以上修改内容

3. 提交内容说明

Header

Header部分只有一行，包括三个字段：type（必需）、scope（可选）和subject（必需）

- 类型 type
- - 推荐使用以上提交内容类型，统一规范
- 影响范围 scope
- - 可选。默认不填写即可
- 提交内容概要 subject
- - 不管是哪一个部分，任何一行都不得超过72个字符（或100个字符）
  - subject： 不超过50个字符
  - 以动词开头，使用第一人称现在时，比如change，而不是changed或changes
  - 第一个字母小写
  - 结尾不加句号（.）

Body

有两个注意点。

（1）使用第一人称现在时，比如使用change而不是changed或changes，中文也需要注意

（2）应该说明代码变动的动机，以及与以前行为的对比

Footer

> **不兼容变动**

**
**

**
**

> **关闭 Issue**

如果当前 commit 针对某个issue，那么可以在 Footer 部分关闭这个 issue

```
Closes #123, #245, #992
```

提交范例

> 新增功能范例

```
[新增功能]: 商品录入业务开发

详细内容:
1. 商品录入接口
2. 商品分类维护
```

> 修复缺陷范例

```
[修改缺陷]: 修复商品发布中的线上问题

详细内容:
1. BUG内容描述
2. 解决方案简单描述

关闭BUG #234,#235
```

> 回滚提交范例

```
[回滚提交]: 版本号-667ecc1654a317a13331b17617d973392f415f02.
```

参考资料

1. http://www.ruanyifeng.com/blog/2016/01/commit_message_change_log.html

Maven规范

- Maven目录结构说明

src/main/java	**源代码目录**
src/main/resources	所需资源目录
src/main/filters	资源过滤文件目录
src/main/assembly	Assembly descriptors
src/main/config	配置文件根目录
src/main/webapps	各种静态资源或者web相关资源
src/test/java	测试代码目录
src/test/resources	测试所需资源目录
src/test/filters	测试资源过滤文件目录
src/site	与site相关的资源目录
Target/	输出目录根
Target/classes	项目主体输出目录
Test-classes	项目测试输出目录
LICENSE.txt	Project's license
README.txt	Project's readme

- Maven聚合工程项目结构规范

如下为maven聚合工程的一个项目结构

```
maven-project(Maven Project)
  |- maven-sub1(Maven Module)
  |- maven-sub2
  |- maven-sub3
|- maven-sub4
```

maven-project就是个建一个普通的Maven Project，这里省略。唯一注意的一点是，Packaging必须选择pom。

1、要在pom工程中定义公共变量

```
<!-- 定义公共变量 -->

<properties>
       <maven.compiler.source>1.8</maven.compiler.source>
       <maven.compiler.target>1.8</maven.compiler.target>
       <maven.compiler.compilerVersion>1.8</maven.compiler.compilerVersion>
       <junit.version>4.12</junit.version>
</properties>
```

2、pom工程中定义聚合模块

```
<!-- 待聚合模块 -->

<modules>
     <module>study-common</module>
     <module>study-pojo</module>
     <module>study-dao</module>
     <module>study-service</module>
     <module>study-elasticsearch</module>
     <module>study-zookeeper</module>
     <module>study-web</module>
</modules>
```

3、pom工程中配置远程部署仓库地址

```
<!-- 配置部署的远程仓库 -->
<distributionManagement>
            <repository>
                  <id>spdbccc</id>
                  <name>Nexus deppen</name>
                  <url>

http://192.168.17.183:8081/nexus/content/repositories/releases
                  </url>
            </repository>
            <snapshotRepository>
                  <id>snapshot</id>
                  <name>Repository for pamirs</name>
                  <url>

http://192.168.17.183:8081/nexus/content/repositories/snapshots
                  </url>
            </snapshotRepository>
<dependencyManagement/>
</distributionManagement>
```

4、<denpendencyManagement>元素中引入所有要进入的jar包

a、这里配置了，这样子项目就不需要重复配置了

b、通过，对一些插件进行了公共的配置，这里主要是为了消除构建时的告警

- pom工程规范举例

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
           <modelVersion>4.0.0</modelVersion>
           <groupId>com.study</groupId>
           <artifactId>study</artifactId>
           <version>0.0.1-SNAPSHOT</version>
           <packaging>pom</packaging>
           <name>study</name>
           <url>http://www.example.com</url>
           <properties>
                      <jdk.version>1.8</jdk.version>
                      <maven.compiler.source>1.8</maven.compiler.source>
                      <maven.compiler.target>1.8</maven.compiler.target>
                      <knife4j.version>2.0.3</knife4j.version>
                      <lombok.version>1.18.10</lombok.version>

           </properties>
           <modules>
                      <module>study-common</module>
                      <module>study-pojo</module>
                      <module>study-dao</module>
                      <module>study-service</module>
                      <module>study-elasticsearch</module>
                      <module>study-zookeeper</module>
                      <module>study-web</module>
           </modules>
           <dependencyManagement>
                      <dependencies>
                          <dependency>
                            <groupId>com.github.xiaoymin</groupId>
                            <artifactId>knife4j-dependencies</artifactId>
                            <version>${knife4j.version}</version>
                            <type>pom</type>
                            <scope>import</scope>
                          </dependency>
                          <dependency>
                            <groupId>org.projectlombok</groupId>
                            <artifactId>lombok</artifactId>
                            <version>${lombok.version}</version>
                          </dependency>
                      </dependencies>
           </dependencyManagement>
           <!-- 使用aliyun镜像 -->
```

```
            <repositories>
                      <repository>
                         <id>aliyun</id>
                         <name>aliyun</name>
                         <url>阿里云镜像地址</url>
                      </repository>
            </repositories>
</project>
```

- Maven常用插件总结

```
<build>
             <finalName>study-common</finalName>
             <plugins>
                   <plugin>
                        <groupId>org.apache.maven.plugins</groupId>
                        <artifactId>maven-compiler-plugin</artifactId>
                        <version>2.3.2</version>
                        <configuration>
                              <source>${jdk.version}</source>
                              <target>${jdk.version}</target>
                              <encoding>${project.build.sourceEncoding}
</encoding>
                        </configuration>
                  </plugin>
                  <plugin>
                        <groupId>org.apache.maven.plugins</groupId>
                        <artifactId>maven-surefire-plugin</artifactId>
                        <version>2.12.4</version>
                        <configuration>
                              <excludes>
                                    <exclude>**/*Test.java</exclude>
                              </excludes>
                        </configuration>
                  </plugin>
                  <plugin>
                        <groupId>org.apache.maven.plugins</groupId>
                        <artifactId>maven-jar-plugin</artifactId>
                        <version>3.0.2</version>
                  </plugin>

                <!-- 自动在运行的时候进行打成war包 -->
                  <plugin>
                        <groupId>org.apache.maven.plugins</groupId>
                        <artifactId>maven-war-plugin</artifactId>
                        <version>2.1.1</version>
                        <configuration>
                              <!-- 重点是这个配置 -->
                              <warName>cubc</warName>
                              <warSourceDirectory>
                                 ${project.basedir}/${pro.outerDirectory}
</warSourceDirectory>
                              <packagingExcludes>WEB-
INF/web.xml</packagingExcludes>

<failOnMissingWebXml>false</failOnMissingWebXml>
                              <warSourceExcludes>
```

```
                                js/**/*.js,libs/**/*.js,
                                css/**/*.css,libs/**/*.css,WEBINF/**/*.html
                            </warSourceExcludes>
                        </configuration>
                    </plugin>
                </plugins>
            </build>
```