**2. 命名规范**

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。

反例： \_name/\_\_name/$Object/name\_/name$/Object$

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UpperCamelCase风格，必须遵从驼峰形式，但以下情形例外：（领

域模型的相关命名）EO / BO / DTO / VO等。

正例：MarcoPolo / UserEO / XmlService / TcpUdpDeal / TaPromotion

反例：macroPolo / UseEo / XMLService / TCPUDPDeal / TAPromotion

4. 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. 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. 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. 中括号是数组类型的一部分，数组定义如下：String[] args;

反例：使用String args[]的方式来定义。

8. 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。

反例：定义为基本数据类型Boolean isSuccess；的属性，它的方法也是isSuccess()，

RPC框架在反向解析的时候，“以为”对应的属性名称是success，导致属性获取不到，进而抛出异常。

9. 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.adc.open.util、类名为MessageUtils（此规则参考

spring的框架结构）

10. 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. 接口和实现类的命名有两套规则：

1）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。

12.【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。

3） 获取统计值的方法用count做前缀。

4） 插入的方法用save（推荐）或insert做前缀。

5） 删除的方法用remove（推荐）或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：xxxEO，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EO/DTO/BO/VO的统称，禁止命名成xxxPOJO。

**3. Java 文件的格式**

所有的 Java(\*.java)文件都必须遵守如下的样式规则。

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。

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。

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。

说明：对子类的实现要求，或者调用注意事项，请一并说明。

3. 所有的类都必须添加创建者和创建日期。

配合 SVN，Eclipse 的使用，所有 Java 文件的头部固定为：

/\*

\*Lastmodifiedby$Author$on$Date$(UTC)

\*Revision:$Revision$

\*/

Author，Date 处写明作者和日期。

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. Imports

避免import整个package而造成的浪费，而必须import具体的类。如：

import java.util.\*;//犯规！

import java.util.Observable;//可以用。

7. ClassFields

所有的静态变量和成员变量出现在类定义之后。一般按照常量、静态变量、

成员变量的顺序定义。

5main方法：如果有main(String[])方法,那么它应该写在类的底部。

8. 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。

**4. 编码风格**

使用 Eclipse 的程序员建议使用 format 功能优化源代码的格式。

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：

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. 左小括号和右边相邻字符之间不出现空格；同样，右小括号和左边相邻字符

之间也不出现空格。详见第 5 条下方正例提示。

3. if/for/while/switch/do 等保留字与小括号之间都必须加空格。

4. 任何运算符左右必须加一个空格。 说明：运算符包括赋值运算符=、逻辑运

算符&&、加减乘除符号、三目运算符等。

5. 缩进采用 4 个空格，禁止使用 tab 字符。

说明：如果使用 tab 缩进，必须设置缩进，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。

6. 单行字符数限不超过 120 个，超出需要换行时遵循如下原则：

1） 第二行相对一缩进 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） 在多个参数超长，逗号后进行换行。

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，并且方法前的点符号一起换行

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

7.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);

7. 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");

8. 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。

**5. 编码约定**

**5.1 exit()**

exit 除了在 main 中可以被调用外，其他的地方不应该调用。因为这样做不给

任何代码代码机会来截获退出。一个类似后台服务地程序不应该因为某一个库模

块决定了要退出就退出。

**5.2 对类成员的访问**

一般成员变量应该定义为 private。子类也应该通过方法来访问。

**5.3 方法的复杂性**

应避免单一方法过于复杂。如果方法实现主要由简单的赋值语句和方法调用

构成，长度不应超过 50 行。如果由分支和循环，建议不超过 30 行。

**5.4 常量定义**

1. 尽量不要出现任何魔法值（即未经定义的常量）直接出现在代码中。若需直

接使用常量则必须加上注释说明含义。

反例： String key = "Id#adc\_"+tradeId； cache.put(key, value);

2. long或者Long初始赋值时，必须使用大写的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?

**5.5 OOP 规约**

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。

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

4. 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. 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. 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. 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之间的赋值，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） 所有的POJO类属性必须使用包装数据类型。

2） RPC方法的返回值和参数必须使用包装数据类型。

9. 定义EO/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. 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. 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. 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。

**5.6 其它**

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。

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

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

3. 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。

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1-10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(); 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。

**6. 注释率要求**

**6.1 名词解释**

代码注释率 = 注释行数/(有效代码行数+注释行数)

有效代码行数是指不包含注释行和空白行的行数