中国电信集团集中 MSS 系统
开发规范及注意事项

中国电信集团公司云网运营部

2020 年 10 月

目录

一、引言

1.1. 编写目的

本文档旨在指导中国电信集中 MSS 系统所有项目组的项目开发过程中的系统架构、系统设计、编码开发的过程，遵守系统架构、系统设计原则，统一开发编码规约，提升协作效率，降低沟通成本。培养开发人员的合规编码意识，提高软件代码的可读性和可重用性，杜绝不合规的编码问题，切实提升系统稳定性，保证版本交付质量。

本规范以 WEB 系统开发为出发点，包括了云原生的架构规范、系统设计规约及开发编码规约。开发规范又划分后端 Java 开发、前端 CSS\HTML\JS 开发、数据库开发、安全开发及微服务开发五个维度，再根据内容特征，细分若干二级子目录。

1.2. 适用范围

本规范适用于中国电信集中 MSS 系统中的所有软件源程序。本规范适用人员包括：系统架构师、系统设计师、开发人员、审计人员、项目管理人员。自本标准实施之日起，以后新编写的代码均试应执行本标准。

二、开发规范

2.1. 后端开发规范

2.1.1. 编码规范

2.1.1.1. 命名规范

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__

扫描工具：P3C；规则编号：2

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。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：ai/ h5/ view/ ecs 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

3. 【强制】类名使用 UpperCamelCase 风格，必须遵守驼峰形式，但以下情形例外（领域模型的相关命名）：DO / BO / DTO / VO / AO / PO / UID 等。

   正例：MarcoPolo / UserDO / XmlService / TcpUdpDeal / TaPromotion

   反例：macroPolo / UserDo / XMLService / TCPUDPDeal / TAPromotion

   扫描工具：P3C；规则编号：27

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

   扫描工具：P3C；规则编号：46

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

   扫描工具：P3C；规则编号：47

6. 【推荐】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。除使用工具（mybatis）生产的代码外，其他建议遵守。

   扫描工具：P3C；规则编号：10、14、16

7. 【强制】类型与中括号紧相连来表示数组。即中括号是数组类型的一部分

正例：定义整形数组 int[] arrayDemo;

反例：在 main 参数中，使用 String args[]来定义。

扫描工具：P3C；规则编号：22

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。

   扫描工具：P3C；规则编号：51

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）

   扫描工具：P3C；规则编号：5

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 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，建议将设计模式体现在名字中，有利于阅读者快速理解

    正例：public class OrderFactory;

    public interface MessageListenner

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。

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。

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();

14. 接口和实现类的命名有两套规则：

✓ 【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。

扫描工具：P3C；规则编号：36

✓ 【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：

✓ 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。

✓ 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。

2.1.1.2. 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 = tradeId + “123”；

   cache.put(key, value)；

   扫描工具：P3C；规则编号：8

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。

   反例：long count = 2l ；

   正例：long count = 2L；

   扫描工具：P3C；规则编号：45

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。

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，两位工程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。

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;
    }
}
```

2.1.1.3. 格式规范

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。

   反例：if (空格 a == b 空格)

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

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。

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");
    }
}
```

6. 【强制】注释的双斜线与注释内容之间有且有一个空格。

   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();

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)
```

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);

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。

扫描工具：P3C；规则编号：37

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。

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。

2.1.1.4. 定义对象约定

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 getObject()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。

   扫描工具：sonar；规则编号：2
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<User> listUsers(String type, Long... ids) {...}
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）

扫描工具：P3C；规则编号：50

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。

扫描工具：P3C；规则编号：44

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。

扫描工具：P3C；规则编号：11

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。

扫描工具：P3C；规则编号：35

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。

    扫描工具：P3C；规则编号：18

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。

    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);
```

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。

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。

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;
  }
}
```

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";
}
```

扫描工具：P3C；规则编号：6

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。

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。

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。

2.1.1.5. 集合处理规范

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。

扫描工具：P3C；规则编号：39

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。

扫描工具：P3C；规则编号：30

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<String> list = new ArrayList<String>(2);
list.add("guan");
 list.add("bao");
String[] array = new String[list.size()];
array = list.toArray(array);
```

反例：直接使用 toArray 无参方法存在问题，此方法返回值只能是

Object[]类，若强转其它类型数组将出现 ClassCastException 错误。

扫描工具：P3C；规则编号：15

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。

   扫描工具：P3C；规则编号：53

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>。

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<String> list = new ArrayList<>();
   list.add("1");
   list.add("2");
   Iterator<String> 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？

扫描工具：P3C；规则编号：17

8. 【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：

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

9. 【推荐】集合泛型定义时，在 JDK7 及以上，使用 diamond 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);
```

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 loader factor）默认为 0.75，如果暂时无法确定初始值大小，请设置为 16（即默认值）。

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

    扫描工具：P3C；规则编号：13

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 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。

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（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。

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。

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。

2.1.1.6. 并发处理规范

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：

```
public class TimerTaskThread extends Thread {
    public TimerTaskThread() {
        super.setName("TimerTaskThread");
        ...
    }
}
```

   扫描工具：P3C；规则编号：21

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。

扫描工具：P3C；规则编号：49

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。

   扫描工具：P3C；规则编号：38

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<DateFormat> df = new
 ThreadLocal<DateFormat>() { @Override
 protected DateFormat initialValue() {
     return new SimpleDateFormat("yyyy-MM-dd");
 }
};
```

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

   扫描工具：P3C；规则编号：4

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。

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。

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。

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。

扫描工具：P3C；规则编号：52

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。

扫描工具：P3C；规则编号：29

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 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。

扫描工具：P3C；规则编号：12

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 JDK5 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...
}
```

扫描工具：sonar；规则编号：50

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

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。

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。

2.1.1.7. 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。

   扫描工具：P3C；规则编号：43

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;

   扫描工具：P3C；规则编号：1

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("stay at home to learn Alibaba Java Coding
    Guidelines.");
    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) && (...) || (...)) {
    ...
}
```

   扫描工具：P3C；规则编号：26

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。

扫描工具：P3C；规则编号：24

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。

2.1.1.8. 注释规范

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 方式和/*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。

   扫描工具：P3C；规则编号：32

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。方法注释结构可参考一下结构：

```
/**
* 功能/function：xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
* xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
*
* 流程描述/steps：
* 步骤/steps：注意步骤要与method体保持一致
*       1、do step
        2、do step
        3、XXXX
* @param 参数 1  参数 1 类型参数 1 说明
* @param 参数 2  参数 2 类型参数 2 说明
* @return 返回值
* @exception 异常 1,异常 2...
*
* 修改记录/revision：
日期：        修改人      描述：
xxxxxxxx      xxxxxx          xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

xxxxxxxx xxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

*/

扫描工具：P3C；规则编号：33

3. 【强制】所有的类都必须添加创建者和创建日期，如：

/**

类描述/Class Description：用详细的语句描述该 class 实现的功能实现中用到算法

具体的使用环境及其他需要说明的地方。

可以使用<pre></pre><p><I>等 Java Doc 定义的类 html 的文档结构化语句

*

负责人/principal：xxxxxxx

修改记录/revision：

日期： 修改人： 修改说明：

xxxxxxxx xxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

xxxxxxxx xxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

*/

扫描工具：P3C；规则编号：28

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。单行注释之前应该有一个空行。如果一行不能完成，使用两行，每个”//”后留一空格。

扫描工具：P3C；规则编号：3

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。

扫描工具：P3C；规则编号：31

6. 【推荐】统一采用中文 UTF-8 编码进行注释说明，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。

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. 【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）。

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。

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。

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。

12. 【推荐】文件头注释结构可参考如下：

```
/*
名称/name：xxxxxxx.java 创建日期/createdate： 2003/05/01
*
描述/description：xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
*
公司版权信息/copyright：程序的版权均属于公司所以应该定义公司名称
*
*/
```

2.1.1.9. 其他

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”);

   扫描工具：P3C；规则编号：41

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。

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。

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。

   扫描工具：P3C；规则编号：40

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 JDK8 中，针对统计时间等场景，推荐使用 Instant 类。

   扫描工具：P3C；规则编号：34

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。

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。

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。

   扫描工具：P3C；规则编号：23

2.1.2. 异常处理

1. 【强制】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：NullPointerException，IndexOutOfBoundsException 等等。

   说明：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 catch

   NumberFormatException 来实现。

   正例：if (obj != null) {...}

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

2. 【强制】异常不要用来做流程控制，条件控制。

   说明：异常设计的初衷是解决程序运行中的各种意外情况，且异常的处理效率比条件判断方式要低很多

3. 【强制】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。

   说明：对大段代码进行 try-catch，使程序无法根据不同的异常做出正确的应激反应，也不利于定位问题，这是一种不负责任的表现。

   正例：用户注册的场景中，如果用户输入非法字符，或用户名称已存在，或用户输入密码过于简单，在程序上作出分门别类的判断，并提示给用户。

4. 【强制】捕获异常是为了处理它，不要捕获了却什么都不处理而抛弃之，如果不想处理它，请将该异常抛给它的调用者。最外层的业务使用者，必须处理异常，将其转化为用户可以理解的内容。

5. 【强制】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。

   扫描工具：P3C；规则编号：19

6. 【强制】finally 块必须对资源对象、流对象进行关闭，有异常也要做 try-catch。

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

7. 【强制】不要在 finally 块中使用 return。

   说明：finally 块中的 return 返回后方法结束执行，不会再执行 try 块中的 return 语句。

   扫描工具：P3C；规则编号：20

8. 【强制】捕获异常与抛异常，必须是完全匹配，或者捕获异常是抛异常的父类。

   说明：如果预期对方抛的是绣球，实际接到的是铅球，就会产生意外情况。

9. 【推荐】方法的返回值可以为 null，不强制返回空集合，或者空对象

等，必须添加注释充分说明什么情况下会返回 null 值。

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

10. 【推荐】防止 NPE，是程序员的基本修养，注意 NPE 产生的场景：

    1）返回类型为基本数据类型，return 包装数据类型的对象时，自动拆箱有可能产生 NPE。

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

    扫描工具：P3C；规则编号：7

    2） 数据库的查询结果可能为 null。

    3） 集合里的元素即使 isNotEmpty，取出的数据元素也可能为 null。

    4） 远程调用返回对象时，一律要求进行空指针判断，防止 NPE。

    5） 对于 Session 中获取的数据，建议 NPE 检查，避免空指针。

    6） 级联调用 obj.getA().getB().getC()；一连串调用，易产生 NPE。

    正例：使用 JDK8 的 Optional 类来防止 NPE 问题。

11. 【推荐】定义时区分 unchecked / checked 异常，避免直接抛出 new RuntimeException()，更不允许抛出 Exception 或者 Throwable，应使用有业务含义的自定义异常。推荐业界已定义过的自定义异常，如：DAOException / ServiceException 等。

12. 【参考】对于公司外的 http/api 开放接口必须使用“错误码”；而应用内部推荐异常抛出；跨应用间 RPC 调用优先考虑使用 Result 方式，封装 isSuccess()方法、“错误码”、“错误简短信息”。

    说明：关于 RPC 方法返回方式使用 Result 方式的理由：

    1）使用抛异常返回方式，调用方如果没有捕获到就会产生运行时错误。

    2）如果不加栈信息，只是 new 自定义异常，加入自己的理解的 error message，对于调用端解决问题的帮助不会太多。如果加了栈信息，在频繁调用出错的情况下，数据序列化和传输的性能损耗也是问题。

13. 【参考】避免出现重复的代码（Don’t Repeat Yourself），即 DRY 原

则。

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

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

2.1.3. 日志规约

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

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
private static final Logger logger = LoggerFactory.getLogger(Abc.class);
```

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 log4j.xml 中设置 additivity=false。

   正例：<logger name="com.taobao.dubbo.config" additivity="false">

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。

2.1.4. 单元测试

1. 【强制】好的单元测试必须遵守 AIR 原则。

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

   A：Automatic（自动化）

   I：Independent（独立性）

   R：Repeatable（可重复）

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

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

4. 【强制】单元测试是可以重复执行的，不能受到外界环境的影响。

   说明：单元测试通常会被放到持续集成中，每次有代码 check in 时单元测试都会被执行。如果单测对外部环境（网络、服务、中间件等）有依赖，容易导致持续集成机制的不可用。

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

5. 【强制】对于单元测试，要保证测试粒度足够小，有助于精确定位问题。单测粒度至多是类级别，一般是方法级别。

   说明：只有测试粒度小才能在出错时尽快定位到出错位置。单测不负责检

查跨类或者跨系统的交互逻辑，那是集成测试的领域。

6. 【强制】核心业务、核心应用、核心模块的增量代码确保单元测试通过。

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

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

   说明：源码构建时会跳过此目录，而单元测试框架默认是扫描此目录。

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

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

9. 【推荐】编写单元测试代码遵守 BCDE 原则，以保证被测试模块的交付质量。

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

   C：Correct，正确的输入，并得到预期的结果。

   D：Design，与设计文档相结合，来编写单元测试。

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

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

    反例：删除某一行数据的单元测试，在数据库中，先直接手动增加一行作为删除目标，但是这一行新增数据并不符合业务插入规则，导致测试结果异常。

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

    正例：在 RDC 内部单元测试中，使用 RDC_UNIT_TEST_的前缀标识数据。

12. 【推荐】对于不可测的代码建议做必要的重构，使代码变得可测，避免为

了达到测试要求而书写不规范测试代码。

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

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

15. 【参考】为了更方便地进行单元测试，业务代码应避免以下情况：

    构造方法中做的事情过多。

    存在过多的全局变量和静态方法。

    存在过多的外部依赖。

    存在过多的条件语句。

    说明：多层条件语句建议使用卫语句、策略模式、状态模式等方式重构。

16. 【参考】不要对单元测试存在如下误解：

    那是测试同学干的事情。本文是开发手册，凡是本文内容都是与开发同学强相关的。

    单元测试代码是多余的。系统的整体功能与各单元部件的测试正常与否是强相关的。

    单元测试代码不需要维护。一年半载后，那么单元测试几乎处于废弃状态。

    单元测试与线上故障没有辩证关系。好的单元测试能够最大限度地规避线上故障。

2.2. 前端开发规范

2.2.1. CSS 规范

1. 【推荐】以组件为单位组织代码段。

   说明：组件化，便于统一维护。

2. 【推荐】组件块和子组件块以及声明块之间使用一空行分隔，子组件块之间三空行分隔。

说明：统一规范，增强可读性。

3. 【推荐】Class 和 ID 使用语义化、通用的命名方式。

   说明：统一规范，增强可读性。

4. 【推荐】类名使用小写字母，以中划线分隔。id 采用驼峰式命名。scss 中的变量、函数、混合、placeholder 采用驼峰式命名。

   说明：统一规范，增强可读性。

5. 【推荐】避免选择器嵌套层级过多，尽量少于 3 级。

   说明：增强可读性，减少代码的复杂度。

6. 【推荐】避免选择器和 Class、ID 叠加使用。

   说明：增强可读性，减少代码的复杂度。

7. 【推荐】相关属性应为一组，推荐的样式编写顺序 Positioning>Box model>Typographic>Visual。

   说明：统一规范，增强可读性。

8. 【推荐】url() 、属性选择符、属性值使用双引号。

   说明：统一规范，增强可读性。

9. 【推荐】媒体查询（Media query）的位置将媒体查询放在尽可能相关规则的附近。不要将他们打包放在一个单一样式文件中或者放在文档底部。如果你把他们分开了，将来只会被大家遗忘。

   说明：统一规范，增强可读性。

10. 【推荐】对于通用元素使用 class ，这样利于渲染性能的优化。

    说明：性能考滤。

11. 【推荐】对于经常出现的组件，避免使用属性选择器（例如，[class^="..."]）。浏览器的性能会受到这些因素的影响。

    说明：性能考滤。

12. 【推荐】选择器要尽可能短，并且尽量限制组成选择器的元素个数，建议不要超过 3。

    说明：增强可读性，减少代码的复杂度。

13. 【推荐】在需要显示地设置所有值的情况下，应当尽量限制使用

简写形式的属性声明。常见的滥用简写属性声明的情况如：
padding、margin、font、background、border、border-radius。
说明：统一规范，增强可读性。

14. 【推荐】Less 和 Sass 中的操作符:为了提高可读性，在圆括号中的数学计算表达式的数值、变量和操作符之间均添加一个空格。
说明：统一规范，增强可读性。
15. 【推荐】Less 和 Sass 中的嵌套:
避免不必要的嵌套。这是因为虽然你可以使用嵌套，但是并不意味着应该使用嵌套。只有在必须将样式限制在父元素内（也就是后代选择器），并且存在多个需要嵌套的元素时才使用嵌套。
说明：增强可读性，减少代码的复杂度。
16. 【推荐】注释统一用'/* */'（scss 中也不要用'//'），具体参照右边的写法； 缩进与下一行代码保持一致； 可位于一个代码行的末尾，与代码间隔一个空格。
说明：统一规范，增强可读性。
扫描工具：sonar；规则编号：21
17. 【推荐】链接的样式顺序 a:link -> a:visited -> a:hover -> a:active（LoVeHAte）。
说明：统一规范，增强可读性。

2.2.2. HTML 规范

1. 【推荐】尽量减少标签数量。
说明：增强可读性，减少代码的复杂度。
2. 【推荐】class 应以功能或内容命名，不以表现形式命名。
说明：统一规范，增强可读性。
3. 【推荐】class 与 id 单词字母小写，多个单词组成时，采用中划线-分隔。
说明：统一规范，增强可读性。
4. 【推荐】使用唯一的 id 作为 Javascript hook，同时避免创建无样式

信息的 class。

说明：避免错误。

5. 【推荐】HTML 属性应该按照特定的顺序出现以保证易读性。

Id>class>name>data->src\for\type\href>title\alt>role\aria-。

说明：HTML 属性应该按照特定的顺序出现以保证易读性。。

6. 【推荐】属性的定义，统一使用双引号。

说明：统一规范，增强可读性。

7. 【推荐】注意 HTML 的嵌套约束。如：a 不允许嵌套 div 这种约束属于语义嵌套约束，与之区别的约束还有严格嵌套约束，比如 a 不允许嵌套 a。参考 WEB 标准系列-HTML 元素嵌套。

说明：html 兼容性。

扫描工具：sonar；规则编号：1\2\3\4\5\6\7\8

8. 【推荐】HTML5 规范中 disabled、checked、selected 等属性不用设置值。

9. 【推荐】html 标签语义化，尽量每个标签都是有语义，即使没有 CSS 也能通过 HTML 的标签也能看懂 HTML 的结构。

说明：此外语义化的 HTML 结构，有助于机器（搜索引擎）理解，另一方面多人协作时，能迅速了解开发者意图。

10. 【推荐】统一使用该 HTML 开头

例子：<!DOCTYPE html> <html lang="zh-Hans"> <head><meta charset="utf-8">... 非简体中文网页，其 lang 属性的取值应该遵循 BCP 47 - Tags for Identifying Languages。 繁体中文<html lang="zh-cmn-Hant"> 英文<html lang="en">。

说明：确保在每个浏览器中拥有一致的表现。

2.2.3. JS 规范

1. 【推荐】As short as possible（如无必要，勿增注释）：尽量提高代码本身的清晰性、可读性。

说明：统一规范，增强可读性。

2. 【强制】As long as necessary（如有必要，尽量详尽）：合理的注释、空行排版等，可以让代码更易阅读、更具美感。

   说明：统一规范，增强可读性。

3. 【推荐】函数/方法注释必须包含函数说明，有参数和返回值时必须使用注释标识。 参数和返回值注释必须包含类型信息和说明； 当函数是内部函数，外部不可访问时，可以使用 @inner 标识。

   说明：统一规范，增强可读性。

4. 【推荐】文件注释用于告诉不熟悉这段代码的读者这个文件中包含哪些东西。 应该提供文件的大体内容，它的作者，依赖关系和兼容性信息。

   说明：统一规范，增强可读性。

5. 【强制】变量，使用 Camel (驼峰)命名法。

   说明：统一规范，增强可读性。

6. 【推荐】私有属性、变量和方法以下划线 _ 开头。

   说明：统一规范，增强可读性。

7. 【推荐】常量，使用全部字母大写，单词间下划线分隔的命名方式。

   说明：统一规范，增强可读性。

8. 【推荐】类型检测优先使用 typeof。对象类型检测使用 instanceof。

   说明：性能考滤。

9. 【强制】不要在 Array 上使用 for-in 循环，for-in 循环只用于 object/map/hash 的遍历，对 Array 用 for-in 循环有时会出错. 因为它并不是从 0 到 length - 1 进行遍历，而是所有出现在对象及其原型链的键值。

   说明：for-in 循环只用于 object/map/hash 的遍历，对 Array 用 for-in 循环有时会出错. 因为它并不是从 0 到 length - 1 进行遍历，而是所有出现在对象及其原型链的键值。

10. 【推荐】初始化变量，加一个对应类型的初始值。

11. 【推荐】类，使用 Pascal 命名法，使用名词作为类名。 类的 方法 / 属性，使用 Camel 命名法，使用动宾短语做为方法名，尽量可读性强，见名晓义。

说明：增强可读性。

12. 【强制】单行注释必须独占一行。// 后跟一个空格，缩进与下一行被注释说明的代码一致。

    说明：增强可读性。

13. 【推荐】避免 JS 生成标签。

    说明：通过 JavaScript 生成的标签让内容变得不易查找、编辑，并且降低性能。能避免时尽量避免。

14. 【推荐】用'===', '!=='代替'==', '!='。

    说明：严格相等，避免逻辑错误。

    扫描工具：sonar；规则编号：162

15. 【强制】不要声明了变量却不使用。

    说明：增强可读性。

16. 【强制】不要直接使用 undefined 进行变量判断；使用 typeof 和字符串'undefined'对变量进行判断。

    说明：变量没有声名时，会报错。

    扫描工具：sonar；规则编号：124

17. 【推荐】不要用 null 来判断函数调用时有无传参。

    说明：可能会导致 BUG。

18. 【推荐】一个函数作用域中所有的变量声明尽量提到函数首部，用一个 var 声明，不要出现两个连续的 var 声明。

    说明：增强可读性。

2.2.4. 统一规范

1. 【强制】CSS,html,js 使用不带 BOM 的 UTF-8 编码。

   说明：统一字符集，解决乱码问题，无需使用 @charset 指定样式表的编码，它默认为 UTF-8 。

2. 【强制】HTML 和 CSS 一律使用小写字母。

   说明：统一规范，增强可读性。

3. 【推荐】尽量不用拼音命名。

   说明：重音太多，很难见名晓义。

4. 【推荐】尽量不要手工操作 DOM

说明:团队计划使用 VUE 框架,所以在项目开发中尽量使用 VUE 的特性去满足我们的需求，尽量（不到万不得已）不要手动操作 DOM，包括：增删改 DOM 元素、以及更改样式、添加事件等。

5. 【推荐】删除弃用代码

说明：很多时候有些代码已经没有用了，但是没有及时去删除，这样导致代码里面包括很多注释的代码块,好的习惯是提交代码前记得删除已经确认弃用的代码，例如：一些调试的 console 语句、无用的弃用代码。

2.3. 数据库规范

数据库开发规范以 PAAS 平台提供的数据库开发规范为准，目前涉及到 TeleDB 和 TelePG 两类数据库的规范，请参考附件 1《TELEDB 用户手册-ch.pdf》、附件 2《TelePG 用户指南.pdf》要求。

2.4. 安全开发规范

安全开发规范以集团下发的应用开发安全指南里的要求为准，请参考附件 3《T-3-8 技术分册-应用开发安全指南_v1.0.doc》要求。

2.5. 微服务开发规范

2.5.1. 认证原则

1. 【强制】外部服务的认证和授权，通常由外部应用发起，通过反向代理或网关向安全边界内的服务发起请求，因此必须执行严格的认证过程。无线端 APP、Web 端、桌面客户端等外部应用下的各类服务，都属于外部服务。

2. 【推荐】通常，内部服务处于安全的内网环境之下，例如鉴权服务、商品服务、订单服务等，在对安全需求不高的情况下，可不执行认证过程，服务与服务之间是相互信任的。内部服务 API 认证和鉴权，可采用基于 Token 的认证方式，对于请求的用户身份的授权以及合法性鉴权。同一注册中心下的服务可视为内部服务。

3. 【推荐】网关统一进行外部服务的认证和授权；也可搭建统一认证服务，由各微服务单独进行认证和授权。

4. 【推荐】JWT 是一种自包含的客户端令牌系统技术规范，可采用 JWT + API 网关进行服务的认证和鉴权。

2.5.2. 通信原则

1. 【强制】专业内使用 Feign 进行服务间通信，跨专业使用 RestTemplate 访问 REST 服务进行服务间通信。

2. 【推荐】异步接口，建议尽量使用消息队列进行服务间解耦。

2.5.3. 配置管理原则

1. 【强制】配置中心统一使用集团云道平台的 Apollo(阿波罗)应用配置中心。配置中心中服务的命名采用 专业 + 微服务 ，例如：财辅无纸化服务：fssc-paperless。

2.5.4. 日志管理原则

1. 【强制】对于业务系统操作日志统一通过接口输出到日志平台进行存储和查询。

2. 【强制】对于业务系统内部日志或接口报文可统一通过接口输出到日志平台进行存储和查询。

2.5.5. 其他原则

1. 【推荐】建议网关使用 gateway，配合使用熔断、限流、监控等组件；建议各专业自建网关（待确认）。

2. 【推荐】对于吞吐量大的接口消息队列组件采用 kafka，对数据一致性高的消息队列组件采用 ctg-mq。

3. 【强制】缓存采用集团分布式缓存组件 CTG-CACHE，使用上遵循 CTG-CACHE 组件开发规约。

4. 【推荐】服务发现组件使用 PAAS 平台提供的 consul 或者 Nacos，由各专业自建。（待确认）。

三、架构设计规范

3.1. 可监控设计规范

上云系统的监控均由智能运维平台云眼提供，应用系统要做 IaaS、PaaS、SaaS 以及安全方面的信息点埋放以及相关日志的输出，以保障系统上云后，能及时把系统纳入智能运维平台 AIOPS 进行纳管。相关的可监控规范以集团下发的云眼使用要求为准，请参考附件 4《系统上云指导意见—智能运维平台分册 0329.docx》要求。

3.2. 灰度发布设计规范

1. 【强制】 有任务调度类业务流程时，业务逻辑必须都放在服务里，任务调度只是负责触发。

   说明：如果将业务逻辑放到任务调度里执行，会导致服务灰度后，无法区分流量

2. 【推荐】 为了支持灰度发布，服务之间需要尽量保持隔离

   说明：避免因为服用数据库、服用队列等场景下，灰度版本和正式版本互相影响

3. 【推荐】 识别具有灰度意义的请求属性，并在制定接口时，支持以 header 的方式传递此类属性。

   说明：灰度控制时 header 处理更方便

4. 【推荐】 尽量避免接口的有状态和粘性

   说明：灰度控制时有状态和粘性会导致控制复杂

3.3. 前后端分离接口规范

3.3.1. 规范原则

1. 【强制】分离明确：后端处理所有的业务逻辑，前端仅接受数据，做页面渲染。

2. 【强制】框架选型：使用 Vue 前端框架来实现前台页面开发。

3.3.2. 参数规范

1. 【强制】包装类参数：参数类型尽量定义成包装类，以免前端传空报错。

3.3.3. 内容规范

1. 【强制】代码精炼：在保证拓展性的情况下，尽量做到代码精简。
   说明：比如判空操作能够 sql 查询判空就尽量用 SQL 查询判空，就不用在代码里面判断空了。使用工具类代替原生方法，比如参数拼接可以使用工具类 HttpUtil.toParams(params)代替。
2. 【强制】内容规整：有数字的地方尽量使用枚举类代替
   说明：因为使用枚举类能够明确表述该数字代表的含义，便于阅读。
   循环时尽量用增强 for 循环，可以减少判空操作。
3. 【强制】必要注释：在关键代码处编写必要的注释
   说明：有利于后期迭代与更新。
4. 【推荐】包装类结果：状态码：0-表示失败，1-表示成功；接口返回信息描述：msg 为结果提示信息，如果 APP 的话就可以返回给用户提示。
   接口返回数据:返回数据对象，通常是实体类对象，VO

```
{
    "code": "1",
    "msg": "查询成功",
    "data": {
    "id": 1,
    "webServicePhone": "028-80000000",
   "webServiceQq": "123456789",
    "webServiceMail": "yoli_tin@163.com",
    "companyName": "华盛集团",
    "companyAddress": "天府二街",
    },
    "success": true
```

}

5. 【推荐】捕获异常：写的接口基本都要捕获异常。
6. 【推荐】抛出运行时异常：有的异常信息不需要系统处理或者直接需要反馈给用户的信息,就需要抛出。通常使用 ApplicationException extends RuntimeException 来进行处理 。
7. 【推荐】VO：View Object，视图对象。当从数据库返回的查询结果不能被某一个实体直接接受的话,我们就需要创建 VO 来接受。
8. 【推荐】接口开发包括提供给管理端接口，前端 APP 接口，H5 页面接口，第三方接口。

3.3.4. 安全规范

1. 【推荐】接口访问权限：通常除了系统内的 API 接口供给 H5 使用不需要访问权限，其他接口均有访问权限。通过在拦截器里面拦截用户请求，对用户进行 Token 和签名校验，通过才能访问该接口，也可以防接口被攻击。
2. 【推荐】接口跨域考虑：在 nginx 配置跨域配置。
   说明：但第三方接口的我们系统接口的话，需要设置接口白名单。
3. 【强制】数据校验：对接口传输的数据需做校验处理，达到数据防篡改、防重放的目的，例如：MD5。

3.3.5. 接口访问速度，并发量

1. 【推荐】索引：通过增加索引来提高数据库的查询速度,进而提高接口的响应速度。
2. 【推荐】缓存：使用 redis 等缓存工具提高响应速度。

3.4. 容错设计规范

3.4.1. 服务隔离规范

将系统按照一定原则划分为若干个服务模块，各个模块之间相互独立，无强依赖。当故障发生时，能将问题和影响隔离在某个模块内部，而不影响

其他模块，提升整体的稳定性。

1. 【强制】 仔细甄别关键业务和关键路径，将涉及服务做为隔离依据
2. 【强制】 仔细甄别关键服务对象，将其做为隔离依据。
3. 【推荐】 隔离服务涉及的基础资源和数据分区
4. 【推荐】 根据需求设计隔离原则，基于用户、地域等属性隔离服务

3.4.2. 降级规范

当整个微服务架构整体的负载超出了预设的上限阈值或即将到来的流量预计将会超过预设的阈值时，为了保证重要或基本的服务能正常运行，我们可以将一些不重要或不紧急的服务或任务进行服务的延迟使用或暂停使用。

1. 【推荐】 需要支持的降级方式
   1) 开关降级 - 通过人工设置开关来触发降级
   2) 参数降级 - 通过调用参数来触发降级
   3) 超时降级 - 主要配置好超时时间和超时重试次数和机制，并使用异步机制探测恢复情况
   4) 失败次数降级 - 主要是一些不稳定的 API，当失败调用次数达到一定阀值自动降级，同样要使用异步机制探测回复情况
   5) 限流降级 - 当触发了限流超额时，可以使用暂时屏蔽的方式来进行短暂的屏蔽
2. 【推荐】 需要支持分级降级机制
   1）根据服务的对比而进行选择式舍弃(即丢车保帅的原则)，从而进一步保障核心的服务的正常运作

3.4.3. 限流规范

1. 【推荐】 常用的限流算法：1）固定时间窗口；2）滑动时间窗口；3）令牌桶算法；4）漏桶算法；
2. 【推荐】 针对不同的业务场景，采用不同的限流策略
   1) 异步限流：适用于后端逻辑处理业务，无需及时向客户端返回处理结果，允许处理请求暂时积压，延迟处理(重要数据要求必须被处理)
   2) 丢弃式限流：适用于需及时返回的前端业务，比如一个状态查询，前

端页面要求及时返回查询结果

3.4.4. 重试幂等规范

1. 【强制】小心定义服务重试条件，避免产生大量重试阻止系统恢复
2. 【强制】应该采用指数退避算法来持续增加重试的间隔时间
3. 【强制】应该考虑目标服务的幂等支持，如不支持需要协商一致性方案
4. 【强制】幂等的唯一 ID 设计应该放在服务外存储，服务应该保持无状态
5. 【推荐】利用 HTTP 方法来辅助服务幂等性设计

3.5. 组件使用规范

1. 【强制】使用云翼平台提供的 paas 组件清单里的组件，若有新增组件请按云翼平台要求申请。
2. 【强制】相同功能必须使用已有组件完成，不重新制作轮子。
3. 【推荐】在应用使用过程中，仅添加本应用需要的版本，禁止添加同一组件多个版本。
4. 【推荐】在应用使用过程中，原则上只能添加 Release 版本，不建议添加 Snapshot 版本。
5. 【推荐】在应用使用过程中，需要注意组件支持的环境版本，如：JDK8、DB2 9.7 等；
6. 【推荐】在应用使用过程中，需要注意是否有必要使用某组件，如：进行数据库迁移时，是否有必要使用配置类组件。

3.6. 持续集成规范

1. 【推荐】所有 java 项目必须使用 spring+mybatis 框架开发；
2. 【强制】所有 JAVA 项目必须使用 maven 进行集成开发；
3. 【强制】L1 上云项目需采用 GIT 进行源码的版本控制，采用云道平台进行源码的编译和构建，需要满足云道源码安全审计的要求。
4. 【强制】L2 上云项目需将源码通过云道进行托管，采用云道用云道平台进行源码的编译和构建，需要满足云道源码安全审计的要求，并且使用云道部署

上线能力，进行应用的一键部署与发布。

5. 【强制】L3 上云项目需使用的云道的需求管理、开发管理、CICD 能力，满足云道安全审计的要求。

3.7. 兼容性规范

1. 【推荐】除特殊浏览器要求，必须兼容安全浏览器、谷歌、IE 三种，其中 IE 需支持 ie10 及以上版本。
2. 【强制】软件存在不同版本时，功能上要考虑向前兼容和向后兼容
3. 【强制】软件存在不同版本时，要考虑数据兼容，确保确保迁移和转换后的数据，用户能正常读取
4. 【强制】分辨率兼容，网页或产品 UI 在各种分辨率下的显示器或各种分辨率、尺寸屏幕的移动设备上都能正常显示

3.8. 网络部署规范

1. 【推荐】必须明确集群 session 会话机制：
   1） L1：原生中间件，建议超时时间设置 30 分钟；
   2） L2：集中式 session 存储到 redis 或者数据库等，建议超时时间设置 30 分钟；
   3） L3：无状态 token 或者有状态 token 设计，建议超时时间设置 30 分钟；
2. 【推荐】应用服务器不允许有单点
3. 【推荐】不允许直接通过 IP 进行相互调用，但也不能排除 DNS 解析失败的情况
4. 【推荐】不允许直接使用 JVM 的默认配置上线

3.9. 缓存规范

在 redis 使用过程中，要正视网络往返时间，合理利用批量操作命令，减少通讯时延和 redis 访问频次。

1. 【推荐】key 名称规范性：使用业务名作为前缀，用冒号分割。有子系统时，可以使用多个冒号或者下划线。比如：order:time:123456，data_123_456

2. 【推荐】注意 key 的长度，key 过长会导致占用较多的内存空间

3. 【推荐】避免转移字符：不能使用逗号，换行，空格，双引号，单引号，大括号等转义字符。

4. 【推荐】redis 存放过期时间，避免集群内存使用量一直持续上涨

3.10. 文件存储规范

1. 【推荐】禁止本地存储有关用户数据的文件；

2. 【推荐】文件和图片的存储采用 MMT 团队提供的接口进行存储；

3. 【推荐】通过 cache 加速的图片必须提供刷新调用接口；

3.11. 云原生设计规范

1. 【强制】 基准代码：一份基准代码，多份部署

说明：一个应用一份基准代码，可以部署不同环境（测试、准生产、生产），基准代码与应用是一对一关系。

2. 【强制】 依赖：显式声明依赖关系

说明：这里的依赖指所有的依赖，不允许在应用中隐式使用依赖的类库、依赖的某些系统工具，必须显式申明；同时简化环境配置，保持运行环境一致性（比如容器）。

3. 【强制】 配置：在环境中存储配置

说明：在环境中存储配置：代码与配置分离，配置统一存储在云道的应用配置中心。配置指与部署环境有关的配置，例如：

1）数据库、消息代理、缓存系统等后端服务的连接配置和位置信息，如 URL、用户名、密码等。

2）第三方服务的证书。

3）每份部署独有的配置，例如：域名、连接数、与部署目标环境资源规模有关的 JVM 参数等。

4. 【强制】 后端服务：把后端服务当作附加资源

说明：把后端服务当作附加资源：后端资源与应用部署保持松耦合，后端服务（数据库、缓存 redis 等）不能与应用一起部署在本地；后端可以按需要扩缩容资源，对应用保持透明，不需要修改代码。

5. 【强制】 创建、发布、运行：严格分离构建和运行

说明：在云道平台做持续集成、应用配置、持续部署、软件资产规定等功能，严格分离构建和运行。

6. 【强制】 进程：以一个或多个无状态进程运行应用

说明：以一个或多个无状态进程运行应用：应用无状态且应用之间无数据共享，任何需要持久化使用的数据都要存储在数据库、缓存系统等外部服务中。通常做法是将数据存储在集中缓存系统中共享使用，如 Session 数据，粘滞会话是应用实现可用性和扩展性的重要障碍。

7. 【强制】 端口绑定：通过端口绑定提供服务

说明：通过端口绑定提供服务：应用直接与端口绑定，而不依赖服务器，避免一个服务器中运行多个应用。端口绑定由平台自动完成，自动实现内、外部端口映射，端口发生变化，访问 URL 保持不变，实现动态注册。 端口绑定指的是应用直接与端口绑定，而不是通过应用服务器进行端口绑定。

8. 【强制】 并发：通过进程模型进行扩展

说明：通过进程模型进行扩展：通过快速复制、启动多个相同进程，提高系统并发处理能力。对于 Java 应用而言，就是通过复制 JVM 所在容器的数量达到进程横向扩展的能力，而不是通过加大 JVM 上限的纵向扩展的能力。容器的横向扩展由云平台完成。

9. 【强制】 易处理：快速启动和优雅终止可最大化健壮性

说明：快速启动和优雅终止：应用可以迅速启动和停止，时间建议控制在 30s 以内。应用应该避免体积过大或者是引用了太多的库文件，应用的运行步骤应该非常简单。优雅终止对于短任务来说一般意味着拒绝所有新的请求，并将已经接收的请求处理完毕后再终止；对于长任务来说，这一般意味着应用重启后的客户端重连和为任务设置断点并在重启后继续执行。除此之外，对事务的完整性和操作的幂等性需要做出完备的考虑。

10. 【强制】 开发环境与线上环境等价：尽可能的保持开发，准生产，线上环境相同

说明：开发环境与线上环境等价：实现持续部署，需要两个环境差异尽可能小，降低交付失败率。比如缓存开发使用 memcached，而生产使用 redis 类似行为需要严格禁止。同时操作系统及 PaaS 组件的版本也要尽可能的保持一致。

11. 【强制】 日志：把日志当作事件流

说明:日志作为事件流：日志以流的形式输出到统一的日志平台进行分析、监控，避免应用本地自行管理日志文件。 应用程序应该将其产生的事件以每个事件一行的格式按时间顺序输出，应用程序不应自行管理日志文件。

该原则是要求应用程序将日志以事件流的方式输出到标准输出 STDOUT 和标准错误输出 STDERR，然后由运行环境捕获这些事件流，并转发到专门的日志处理服务进行处理。

12. 【强制】 管理进程：后台管理任务当作一次性进程运行

说明：管理和维护应用的任务应该与运行的应用使用相同环境、配置，后台管理代码应该随应用程序一同发布，避免发生不同步问题。即一次性任务也当作应用去运行，走正常的编写，提交，构建，发布流程. 而不是随便写一些脚本扔到线上直接跑。比如配置中心完成配置修改、分布式调度框架完成 job、迁移工具完成数据迁移等一次性的任务。

四、附录

1. 附件 1：《TELEDB 用户手册-ch.pdf》
2. 附件 2：《TelePG 用户指南.pdf》
3. 附件 3：《T-3-8 技术分册-应用开发安全指南_v1.0.doc》
4. 附件 4：《系统上云指导意见—智能运维平台分册 0329.docx》