一、编程规约

（一）命名风格

1. 【强制】代码中的命名均不能以下划线或美元符号开始，也不能以下划线或美元符号结束。
   反例：`_name / __name / $name / name_ / name$ / name__`

2. 【强制】代码中的命名严禁使用拼音与英文混合的方式，更不允许直接使用中文的方式。
   说明: 正确的英文拼写和语法可以让阅读者易于理解，避免歧义。注意，纯拼音命名方式更要避免采用。
   正例: renminbi / 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

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: JavaServerlessPlatform / UserDO / XmlService / TcpUdpDeal / TaPromotion
   反例: javaserverlessplatform / UserDo / XMLService / TCPUDPDeal / TAPromotion

4. 【强制】方法名、参数名、成员变量、局部变量都统一使用 lowerCamelCase 风格，必须遵从驼峰形式。
   正例: localValue / getHttpMessage() / inputUserId

5. 【强制】常量命名全部大写，单词间用下划线隔开，力求语义表达完整清楚，不要嫌名字长。
   正例: MAX_STOCK_COUNT / CACHE_EXPIRED_TIME
   反例: MAX_COUNT / EXPIRED_TIME

6. 【强制】抽象类命名使用 Abstract 或 Base 开头;异常类命名使用 Exception 结尾;测试类命名以它要测试的类的名称开始，以 Test 结尾。

7. 【强制】类型与中括号紧挨相连来表示数组。
   正例: 定义整形数组 int[] arrayDemo;
   反例: 在 main 参数中，使用 String args[] 来定义。

8. 【强制】POJO 类中布尔类型变量都不要加 is 前缀，否则部分框架解析会引起序列化错误。
   说明: 在本文 Mysql 规约中的建表规约第一条，表达是否的值采用 is_xxx 的命名方式，所以，需要在 `<resultMap>` 设置从 is_xxx 到 xxx 的映射关系
   反例: 定义为基本数据类型 Boolean isDeleted 的属性，它的方法也是 isDeleted()，RPC 框架在反向解析的时候，“误以为”对应的属性名称是 deleted，导致属性获取不到，进而抛出异常。

9. 【强制】包名统一使用小写，点分隔符之间有且仅有一个自然语义的英语单词。包名统一使 用单数形式，但是类名如果有复数含义，类名可以使用复数形式。
   正例: 应用工具类包名为 com.alibaba.ai.util、类名为 MessageUtils(此规则参考 spring 的框架结构)

10. 【强制】避免在子父类的成员变量之间、或者不同代码块的局部变量之间采用完全相同的命名，使可读性降低。
    说明: 子类、父类成员变量名相同，即使是 public 类型的变量也是能够通过编译，而局部变量在同一方法内的不同

代码块中同名也是合法的，但是要避免使用。对于非 setter/getter 的参数名称也要避免与成员变量名称相同。
反例:

```java
public class ConfusingName {
    public int age;
    // 非 setter/getter 的参数名称，不允许与本类成员变量同名
    public void getData(String alibaba) {
        if(condition) {
            final int money = 531;
            // ...
        }
        for (int i = 0; i < 10; i++) {
            // 在同一方法体中，不允许与其它代码块中的 money 命名相同
            final int money = 615;
            // ...
        }
    }
}
class Son extends ConfusingName {
    // 不允许与父类的成员变量名称相同
    public int age;
}
```

11. 【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。

12. 【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: int a 的随意命名方式。

13. 【推荐】在常量与变量的命名时，表示类型的名词放在词尾，以提升辨识度。
正例: startTime / workQueue / nameList / TERMINATED_THREAD_COUNT
反例: startedAt / QueueOfWork / listName / COUNT_TERMINATED_THREAD

14. 【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:

```java
public class OrderFactory;
public class LoginProxy;
```

```
public class ResourceObserver;
```

15. 【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: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。

16. 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 的形容词)。
    正例: AbstractTranslator 实现 Translatable 接口。

17. 【参考】枚举类名带上 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。

18. 【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。
    1) 获取多个对象的方法用 list 做前缀，复数形式结尾如: listObjects。
    2) 获取统计值的方法用 count 做前缀。
    3) 插入的方法用 save/insert 做前缀。
    4) 删除的方法用 remove/delete 做前缀。
    5) 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。

（二）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:

```java
String key = "Id#taobao_" + tradeId;
cache.put(key, value);
// 缓存 get 时，由于在代码复制时，漏掉下划线，导致缓存击穿而出现问题
```

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 = 2l; 写的是数字的 21，还是 Long 型的 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。

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“YES”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:

```java
public enum SeasonEnum {
    SPRING(1), SUMMER(2), AUTUMN(3), WINTER(4);
    private int seq;
    SeasonEnum(int seq) {
        this.seq = seq;
    }
    public int getSeq() {
        return seq;
    }
}
```

（三）代码格式

1. 【强制】如果是大括号内为空，则简洁地写成 {} 即可，大括号中间无需换行和空格; 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)

java

```
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
    }
}
```

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:

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

7. 【强制】在进行类型强制转换时，右括号与强制转换值之间不需要任何空格隔开。

正例:

```java
long first = 1000000000000L;
int second = (int)first + 2;
```

8. 【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:

```java
StringBuilder sb = new StringBuilder();
// 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("Jack").append("Ma")...
    .append("alibaba")...
    .append("alibaba")...
    .append("alibaba");
```

反例:

```java
StringBuilder sb = new StringBuilder();
// 超过 120 个字符的情况下，不要在括号前换行
sb.append("Jack").append("Ma")...append
("alibaba");
// 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);
```

9. 【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。

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

10. 【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。

11. 【推荐】单个方法的总行数不超过 80 行。
说明: 除注释之外的方法签名、左右大括号、方法内代码、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。

12. 【推荐】没有必要增加若干空格来使变量的赋值等号与上一行对应位置的等号对齐。
正例:

```java
int one = 1;
long two = 2L;
float three = 3F;
StringBuilder sb = new StringBuilder();
```

说明: 增加 sb 这个变量，如果需要对齐，则给 one、two、three 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。

13. 【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。

（四）OOP规约

1. 【强制】避免通过一个类的对象引用访问此类的静态变量或静态方法，无谓增加编译器解析成本，直接用类名来访问即可。

2. 【强制】所有的覆写方法，必须加 @Override 注解。
说明: getObject() 与 get0bject()的问题。一个是字母的 O，一个是数字的 0，加 @Override 可以准确判 断是否覆盖成功。另外，如果在抽象类中对方法签名进行修改，其实现类会马上编译报错。

3. 【强制】相同参数类型，相同业务含义，才可以使用 Java 的可变参数，避免使用 Object。
说明: 可变参数必须放置在参数列表的最后。(提倡同学们尽量不用可变参数编程)
正例: public List 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)。

7. 【强制】所有整型包装类对象之间值的比较，全部使用 equals 方法比较。
   说明: 对于 Integer var = ? 在-128 至 127 范围内的赋值，Integer 对象是在 IntegerCache.cache 产生，会复用已有对象，这个区间内的 Integer 值可以直接使用 == 进行判断，但是这个区间之外的所有数据，都会在堆上产生，并不会复用已有对象，这是一个大坑，推荐使用 equals 方法进行判断。

8. 【强制】浮点数之间的等值判断，基本数据类型不能用 == 来比较，包装数据类型不能用 equals 来判断。
   说明: 浮点数采用“尾数+阶码”的编码方式，类似于科学计数法的“有效数字+指数”的表示方式。二进制无法精确表示大部分的十进制小数，具体原理参考《码出高效》。
   反例:

```java
float a = 1.0f - 0.9f;
float b = 0.9f - 0.8f;
if (a == b) {
    // 预期进入此代码快，执行其它业务逻辑 // 但事实上 a==b 的结果为 false
}
Float x = Float.valueOf(a);
Float y = Float.valueOf(b);
if (x.equals(y)) {
    // 预期进入此代码快，执行其它业务逻辑 // 但事实上 equals 的结果为 false
}
```

   正例:
   (1) 指定一个误差范围，两个浮点数的差值在此范围之内，则认为是相等的。

```java
float a = 1.0f - 0.9f;
float b = 0.9f - 0.8f;
float diff = 1e-6f;
if (Math.abs(a - b) < diff) {
```

```
        System.out.println("true");
    }
```

(2) 使用 BigDecimal 来定义值，再进行浮点数的运算操作。

```java
    BigDecimal a = new BigDecimal("1.0");
    BigDecimal b = new BigDecimal("0.9");
    BigDecimal c = new BigDecimal("0.8");
    BigDecimal x = a.subtract(b);
    BigDecimal y = b.subtract(c);
    if (x.equals(y)) {
        System.out.println("true");
    }
```

9. 【强制】定义数据对象 DO 类时，属性类型要与数据库字段类型相匹配。
   正例: 数据库字段的 bigint 必须与类属性的 Long 类型相对应。
   反例: 某个案例的数据库表 id 字段定义类型 bigint unsigned，实际类对象属性为 Integer，随着 id 越来越大，超过 Integer 的表示范围而溢出成为负数。

10. 【强制】为了防止精度损失，禁止使用构造方法 BigDecimal(double) 的方式把 double 值转化为 BigDecimal 对象。
    说明: BigDecimal(double) 存在精度损失风险，在精确计算或值比较的场景中可能会导致业务逻辑异常。 如: BigDecimal g = new BigDecimal(0.1f); 实际的存储值为: 0.10000000149
    正例: 优先推荐入参为 String 的构造方法，或使用 BigDecimal 的 valueOf 方法，此方法内部其实执行了 Double 的 toString，而 Double 的 toString 按 double 的实际能表达的精度对尾数进行了截断。

```java
    BigDecimal recommend1 = new BigDecimal("0.1");
    BigDecimal recommend2 = BigDecimal.valueOf(0.1);
```

11. 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。

12. 【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 createTime 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。

13. 【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。

14. 【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。

15. 【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。

16. 【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，并不能去定哪个方法一定时被优先调用到。

17. 【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:

java

```
String str = "a,b,c,,";
String[] ary = str.split(",");
// 预期大于 3，结果是 3
System.out.println(ary.length);
```

18. 【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，此条规则优先于下一条。

19. 【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。

20. 【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:

java

```
public Integer getData() {
    if (condition) {
        return this.data + 100;
```

```
        } else {
            return this.data - 100;
        }
    }
```

21. 【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:

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

22. 【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) 不允许被覆写的方法，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 可以强制重新定义一个变量，方便更好地进行重构。

23. 【推荐】慎用 Object 的 clone 方法来拷贝对象。
    说明: 对象 clone 方法默认是浅拷贝，若想实现深拷贝需覆写 clone 方法实现域对象的深度遍历式拷贝。

24. 【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。

（五）集合处理

1. 【强制】关于 hashCode 和 equals 的处理，遵循如下规则:
   1) 只要覆写 equals，就必须覆写 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 存储的对象必须覆写这两个方法。
   3) 如果自定义对象作为 Map 的键，那么必须覆写 hashCode 和 equals。
   说明: String 已覆写 hashCode 和 equals 方法，所以我们可以愉快地使用 String 对象作为 key 来使用。

2. 【强制】ArrayList 的 subList 结果不可强转成 ArrayList，否则会抛出 ClassCastException 异常，即 java.util.RandomAccessSubList cannot be cast to java.util.ArrayList。
   说明: subList 返回的是 ArrayList 的内部类 SubList，并不是 ArrayList 而是 ArrayList 的一个视图，对于 SubList 子列表的所有操作最终会反映到原列表上。

3. 【强制】使用 Map 的方法 keySet()/values()/entrySet() 返回集合对象时，不可以对其进行添加元素操作，否则会抛出 UnsupportedOperationException 异常。

4. 【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。

```java
List<String> list = new ArrayList<>(2);
list.add("guan");
list.add("bao");
String[] array = list.toArray(new String[0]);
```

5. 【强制】Collections 类返回的对象，如: emptyList()/singletonList() 等都是 immutable list，不可对其进行添加或者删除元素的操作。
   反例: 如果查询无结果，返回 Collections.emptyList() 空集合对象，调用方一旦进行了添加元素的操作，就会触发 UnsupportedOperationException 异常。

6. 【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致、长度为 0 的空数组。
   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。
   正例:

```java
List<String> list = new ArrayList<>(2);
list.add("guan");
list.add("bao");
String[] array = list.toArray(new String[0]);
```

说明: 使用 toArray 带参方法，数组空间大小的 length:
1) 等于 0，动态创建与 size 相同的数组，性能最好。
2) 大于 0 但小于 size，重新创建大小等于 size 的数组，增加 GC 负担。
3) 等于 size，在高并发情况下，数组创建完成之后，size 正在变大的情况下，负面影响与上相同。
4) 大于 size，空间浪费，且在 size 处插入 null 值，存在 NPE 隐患。

7. 【强制】在使用 Collection 接口任何实现类的 addAll() 方法时，都要对输入的集合参数进行 NPE 判断。
说明: 在 ArrayList#addAll 方法的第一行代码即 Object[] a = c.toArray(); 其中 c 为输入集合参数，如果为 null，则直接抛出异常。

8. 【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。

```java
String[] str = new String[] { "yang", "hao" };
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] = "changed"; 也会随之修改，反之亦然。

9. 【强制】泛型通配符 <?extendsT> 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>

10. 【强制】在无泛型限制定义的集合赋值给泛型限制的集合时，在使用集合元素时，需要进行 instanceof 判断，避免抛出 ClassCastException 异常。
说明: 毕竟泛型是在 JDK5 后才出现，考虑到向前兼容，编译器是允许非泛型集合与泛型集合互相赋值。 反例:

```java
List<String> generics = null;
List notGenerics = new ArrayList(10);
notGenerics.add(new Object());
notGenerics.add(new Integer(1));
generics = notGenerics;
// 此处抛出 ClassCastException 异常
String string = generics.get(0);
```

11. 【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:

```java
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:

```java
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?

（六）并发处理

1. 【强制】获取单例对象需要保证线程安全，其中的方法也要保证线程安全。
说明: 资源驱动类、工具类、单例工厂类都需要注意。

2. 【强制】创建线程或线程池时请指定有意义的线程名称，方便出错时回溯。
正例: 自定义线程工厂，并且根据外部特征进行分组，比如机房信息。

```java
public class UserThreadFactory implements ThreadFactory {
    private final String namePrefix;
    private final AtomicInteger nextId = new AtomicInteger(1);
    // 定义线程组名称，在 jstack 问题排查时，非常有帮助
    UserThreadFactory(String whatFeaturOfGroup) {
```

```
            namePrefix = "From UserThreadFactory's " + whatFeaturOfGroup + "-Worker-";
        }
        @Override
        public Thread newThread(Runnable task) {
            String name = namePrefix + nextId.getAndIncrement();
            Thread thread = new Thread(null, task, name, 0, false);
            System.out.println(thread.getName());
            return thread;
        }
    }
```

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: 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。

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 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:
   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。

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:

java

```
    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。

6. 【强制】必须回收自定义的 ThreadLocal 变量，尤其在线程池场景下，线程经常会被复用， 如果不清理自定义的 ThreadLocal 变量，可能会影响后续业务逻辑和造成内存泄露等问题。 尽量在代理中使用 try-finally 块进行回收。
   正例:

```java
objectThreadLocal.set(userInfo);
try {
    // ...
} finally {
    objectThreadLocal.remove();
}
```

7. 【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。

8. 【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。

9. 【强制】在使用阻塞等待获取锁的方式中，必须在try代码块之外，并且在加锁方法与try代码块之间没有任何可能抛出异常的方法调用，避免加锁成功后，在 finally 中无法解锁。
   说明一: 如果在 lock 方法与 try 代码块之间的方法调用抛出异常，那么无法解锁，造成其它线程无法成功获取锁。
   说明二: 如果 lock 方法在 try 代码块之内，可能由于其它方法抛出异常，导致在 finally 代码块中， unlock 对未加锁的对象解锁，它会调用 AQS 的 tryRelease 方法(取决于具体实现类)，抛出 IllegalMonitorStateException 异常。
   说明三: 在 Lock 对象的 lock 方法实现中可能抛出 unchecked 异常，产生的后果与说明二相同。
   正例:

```java
Lock lock = new XxxLock();
// ...
lock.lock();
try {
    doSomething();
    doOthers();
} finally {
    lock.unlock();
}
```

   反例:

```java
// ...
try {
    // 如果此处抛出异常，则直接执行 finally 代码块
    doSomething();
```

```
        // 无论加锁是否成功，finally 代码块都会执行
        lock.lock();
        doOthers();
    } finally {
        lock.unlock();
    }
```

10. 【强制】在使用尝试机制来获取锁的方式中，进入业务代码块之前，必须先判断当前线程是否持有锁。锁的释放规则与锁的阻塞等待方式相同。
    说明: Lock 对象的 unlock 方法在执行时，它会调用 AQS 的 tryRelease 方法(取决于具体实现类)，如果当前线程不持有锁，则抛出 IllegalMonitorStateException 异常。
    正例:

```java
Lock lock = new XxxLock();
// ...
boolean isLocked = lock.tryLock();
if (isLocked) {
    try {
        doSomething();
        Lock lock = new XxxLock();
        doOthers();
    } finally {
        lock.unlock();
    }
}
```

11. 【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。

12. 【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，如果在处理定时任务时使用 ScheduledExecutorService 则没有这个问题。

13. 【推荐】资金相关的金融敏感信息，使用悲观锁策略。
    说明: 乐观锁在获得锁的同时已经完成了更新操作，校验逻辑容易出现漏洞，另外，乐观锁对冲突的解决策略有较复杂的要求，处理不当容易造成系统压力或数据异常，所以资金相关的金融敏感信息不建议使用乐观锁更新。

14. 【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。

15. 【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。

16. 【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:

```java
public 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...
}
```

17. 【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。
说明: 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)。

18. 【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。

19. 【参考】ThreadLocal 对象使用 static 修饰，ThreadLocal 无法解决共享对象的更新问题。
说明: 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。

（七）控制语句

1. 【强制】在一个 switch 块内，每个 case 要么通过 continue/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，即使它什么代码也没有。

说明: 注意 break 是退出 switch 语句块，而 return 是退出方法体。

2. 【强制】当 switch 括号内的变量类型为 String 并且此变量为外部参数时，必须先进行 null 判断。
反例: 猜猜下面的代码输出是什么?

java

```
public class SwitchString {
    public static void main(String[] args) {
        method(null);
    }
    public static void method(String param) {
        switch (param) {
            // 肯定不是进入这里
            case "sth":
                System.out.println("it's sth");
                break;
            // 也不是进入这里
            case "null":
                System.out.println("it's null");
                break;
            // 也不是进入这里
            default:
                System.out.println("default")
        }
    }
}
```

3. 【强制】在 if/else/for/while/do 语句中必须使用大括号。
说明: 即使只有一行代码，避免采用单行的编码方式: `if (condition) statements;`

4. 【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。

5. 【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:

java

```
if (condition) {
    ...
    return obj;
}
// 接着写 else 的业务逻辑代码;
```

说明: 如果非使用 if()...else if()...else... 方式表达逻辑，避免后续代码维护困难，【强制】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，其中卫语句即代码逻辑先考虑失败、异常、中断、退出等直接返回的情况，以方法多个出口的方式，解决代码中判断分支嵌套的问题，这是逆向思维的体现。
示例如下:

java

```
public void findBoyfriend(Man man) {
    if (man.isUgly()) {
        System.out.println("本姑娘是外貌协会的资深会员");
        return;
    }
    if (man.isPoor()) {
        System.out.println("贫贱夫妻百事哀");
        return;
    }
    if (man.isBadTemper()) {
        System.out.println("银河有多远，你就给我滚多远");
        return;
    }
    System.out.println("可以先交往一段时间看看");
}
```

6. 【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 语句内的逻辑表达式相当复杂，与、或、取反混合运算，甚至各种方法纵深调用，理解成本非常高。如果赋值一个非常好理解的布尔变量名字，则是件令人爽心悦目的事情。
正例:

java

```
// 伪代码如下
final boolean existed = (file.open(fileName, "w") != null) && (...) || (...);
if (existed) {
    ...
}
```

反例:

java

```
public final void acquire(long arg) {
    if (!tryAcquire(arg) && acquireQueued(addWaiter(Node.EXCLUSIVE), arg)) {
        selfInterrupt();
    }
}
```

7. 【推荐】不要在其它表达式(尤其是条件表达式)中，插入赋值语句。
   说明: 赋值点类似于人体的穴位，对于代码的理解至关重要，所以赋值语句需要清晰地单独成为一行。
   反例:

java

```
public Lock getLock(boolean fair) {
    // 算术表达式中出现赋值操作，容易忽略 count 值已经被改变
    threshold = (count = Integer.MAX_VALUE) - 1;
    // 条件表达式中出现赋值操作，容易误认为是 sync==fair
    return (sync = fair) ? new FairSync() : new NonfairSync();
}
```

8. 【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)。

9. 【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读经的正向逻辑写法。
   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。

10. 【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。

11. 【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。

12. 【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。

（八）注释规约

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。

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。

3. 【强制】所有的类都必须添加创建者和创建日期。

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。

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。

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。

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(代码仓库已然保存了历史代码)。

9. 【参考】对于注释的要求:
   第一、能够准确反映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。

（九）其它

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");`

2. 【强制】velocity 调用 POJO 类的属性时，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。

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。

6. 【强制】日期格式化时，传入 pattern 中表示年份统一使用小写的y。
   说明: 日期格式化时，yyyy 表示当天所在的年，而大写的 YYYY 代表是 week in which year (JDK7 之后引入的概念)，意思是当天所在的周属于的年份，一周从周日开始，周六结束， 只要本周跨年，返回的 YYYY 就是下一年。另外需要注意:

   - 表示月份是大写的 M
   - 表示分钟则是小写的 m

- 24 小时制的是大写的 H
- 12 小时制的则是小写的 h

  正例: 表示日期和时间的格式如下所示:

```java
new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
```

7. 【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。

8. 【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. 【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。

二、异常处理

（一）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，可能存在数字格式错误，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。

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: try 块中的 return 语句执行成功后，并不马上返回，而是继续执行 finally 块中的语句，如果此处存在 return 语句，则在此直接返回，无情丢弃掉 try 块中的返回点。
   反例:

```java
private int x = 0;
public int checkReturn() {
    try {
        // x 等于 1，此处不返回
        return ++x;
    } finally {
        // 返回的结果是 2
        return ++x;
    }
}
```

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. 【强制】在调用 RPC、二方包、或动态生成类的相关方法时，捕捉异常必须使用 Throwable 类来进行拦截。

10. 【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。

11. 【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。
    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，建议进行 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。

12. 【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。

13. 【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。

14. 【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:

```java
private boolean checkParam(DTO dto) {...}
```

（二）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。

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

2. 【强制】所有日志文件至少保存15天，因为有些异常具备以“周”为频次发生的特点。网络运行状态、安全相关信息、系统监测、管理后台操作、用户敏感操作需要留存相关的网络日 志不少于 6 个月。

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。
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。
正例: force-web 应用中单独监控时区转换异常，如: force_web_timeZoneConvert.log

4. 【强制】在日志输出时，字符串变量之间的拼接使用占位符的方式。
说明: 因为 String 字符串的拼接会使用 StringBuilder 的 append()方式，有一定的性能损耗。使用占位符 仅是替换

动作，可以有效提升性能。
正例: logger.debug("Processing trade with id: {} and symbol: {}", id, symbol);

5. 【强制】对于 trace/debug/info 级别的日志输出，必须进行日志级别的开关判断。
说明: 虽然在 debug(参数) 的方法体内第一行代码 isDisabled(Level.DEBUG_INT) 为真时(Slf4j 的常见实现 Log4j 和 Logback)，就直接 return，但是参数可能会进行字符串拼接运算。此外，如果 debug(getName()) 这种参数内有 getName() 方法调用，无谓浪费方法调用的开销。
正例:

```java
// 如果判断为真，那么可以输出 trace 和 debug 级别的日志
if (logger.isDebugEnabled()) {
    logger.debug("Current ID is: {} and name is: {}", id, getName());
}
```

6. 【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">`

7. 【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);`

8. 【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?

9. 【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。

10. 【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。【强制】国际化团队或海外部署的服务器由于字符集问题，使用全英文来注释和描述日志错误信息。

三、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: 源码编译时会跳过此目录，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、业务允许外等)，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: 在企业智能事业部的内部单元测试中，使用 ENTERPRISE_INTELLIGENCE *UNIT*TEST_的前缀来 标识单元测试相关代码。

12. 【推荐】对于不可测的代码在适当的时机做必要的重构，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，在项目提测前完成单元测试，不建议项目发布后补充单元测试用例。

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。

四、安全规约

1. 【强制】隶属于用户个人的页面或者功能必须进行权限控制校验。
   说明: 防止没有做水平权限校验就可随意访问、修改、删除别人的数据，比如查看他人的私信内容、修改他人的订单。

2. 【强制】用户敏感数据禁止直接展示，必须对展示数据进行脱敏。
   说明: 中国大陆个人手机号码显示为: 137****0969，隐藏中间 4 位，防止隐私泄露。

3. 【强制】用户输入的 SQL 参数严格使用参数绑定或者 METADATA 字段值限定，防止 SQL 注入，禁止字符串拼接 SQL 访问数据库。

4. 【强制】用户请求传入的任何参数必须做有效性验证。
   说明: 忽略参数校验可能导致:

    - page size 过大导致内存溢出
    - 恶意 order by 导致数据库慢查询

- 任意重定向
- SQL 注入
- 反序列化注入
- 正则输入源串拒绝服务 ReDoS
  说明: Java 代码用正则来验证客户端的输入，有些正则写法验证普通用户输入没有问题，但是如果攻击人员使用的是特殊构造的字符串来验证，有可能导致死循环的结果。

5. 【强制】禁止向 HTML 页面输出未经安全过滤或未正确转义的用户数据。

6. 【强制】表单、AJAX 提交必须执行 CSRF 安全验证。
   说明: CSRF(Cross-site request forgery) 跨站请求伪造是一类常见编程漏洞。对于存在 CSRF 漏洞的应用/网站，攻击者可以事先构造好 URL，只要受害者用户一访问，后台便在用户不知情的情况下对数据库中用户参数进行相应修改。

7. 【强制】在使用平台资源，譬如短信、邮件、电话、下单、支付，必须实现正确的防重放的机制，如数量限制、疲劳度控制、验证码校验，避免被滥刷而导致资损。
   说明: 如注册时发送验证码到手机，如果没有限制次数和频率，那么可以利用此功能骚扰到其它用户，并造成短信平台资源浪费。

8. 【推荐】发贴、评论、发送即时消息等用户生成内容的场景必须实现防刷、文本内容违禁词过滤等风控策略。