远大信息

JAVA 开发规范手册

版本：1.0.0

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】所有编程相关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：ali / alibaba / taobao / cainiao/ aliyun/ 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：ForceCode / UserDO / HtmlDTO / XmlService / TcpUdpDeal / TaPromotion
   反例：forcecode / UserDo / HTMLDto / 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。
   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】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.sinoyd.frame.util、类名为 MessageUtils（此规则参考 spring 的框架结构）

8. 【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：

```
public class ConfusingName
    { public int stock;

    // 非 setter/getter 的参数名称，不允许与本类成员变量同名
    public void get(String alibaba)
        { if (condition) {
            final int money = 666;
            // ...
        }
```

```
            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 = 15978;
                // ...
            }
        }
    }

    class Son extends ConfusingName {
        // 不允许与父类的成员变量名称相同
        public int stock;
    }
```

9. 【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。

10. 【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。

    正例:在 JDK 中，对某个对象引用的 volatile 字段进行原子更新的类名为：AtomicReferenceFieldUpdater。反例：常见的方法内变量为 int a;的定义方式。

11. 【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

12. 【推荐】如果模块、接口、类、方法使用了设计模式，在命名时需体现出具体模式。

    说明：将设计模式体现在名字中，有利于阅读者快速理解架构设计理念。

    正例： public class OrderFactory;

    public class LoginProxy;

    public class ResourceObserver;

13. 【推荐】接口类中的方法和属性不要加任何修饰符号（public 也不要加），保持代码的简洁性，并加上有效的 Javadoc 注释。尽量不要在接口里定义变量，如果一定要定义变量，确定与接口方法相关，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。

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。

15. 【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常量类，且构造方法被默认强制是私有。

    正例：枚举名字为 ProcessStatusEnum 的成员名称：SUCCESS / UNKNOWN_REASON。

16.【参考】各层命名规约：

A) Service/DAO 层方法命名规约

1） 获取单个对象的方法用 get 做前缀。

2） 获取多个对象的方法用 list 做前缀，复数结尾，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。

(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：

```
//本例中同学 A 定义了缓存的 key，然后缓存提取的同学 B 使用了 Id#taobao 来提取，少了下划线，导致故障。
String key = "Id#taobao_" + tradeId;
cache.put(key, value);
```

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，也不利于维护。

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：

```
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】左小括号和右边相邻字符之间不出现空格；右小括号和左边相邻字符之间也不出现空格；而左大括号前需要加空格。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：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");
            // 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：

```
// 这是示例注释，请注意在双斜线之后有一个空格
String commentString = 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：

```
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：

```
StringBuilder sb = new StringBuilder();
// 超过 120 个字符的情况下，换行缩进 4 个空格，并且方法前的点号一起换行
sb.append("zi").append("xin")...
            .append("huang")...
            .append("huang")...
            .append("huang");
```

反例：

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

// 参数很多的方法调用可能超过 120 个字符，逗号后才是换行处
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。

```
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：

```
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<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）。

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 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. 【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。

```
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。

```
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。

```
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】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。

```
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：某业务的交易报表上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，并不能确定哪个方法一定是被优先调用到，神坑之一。

17. 【推荐】任何货币金额，均以最小货币单位且整型类型来进行存储。

18. 【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);
```

19. 【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。

20. 【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。

21. 【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;
    }
}
```

22. 【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";
}
```

23. 【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，如：POJO 类的域变量。

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。

24. 【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。

25. 【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) 日期时间

1. 【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。

正例：表示日期和时间的格式如下所示：

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

2. 【强制】在日期格式中分清楚大写的 M 和小写的 m，大写的 H 和小写的 h 分别指代的意义。

说明：日期格式中的这两对字母表意如下：

1） 表示月份是大写的 M；

2） 表示分钟则是小写的 m；

3） 24 小时制的是大写的H；

4） 12 小时制的则是小写的 h。

3. 【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。

4. 【强制】不允许在程序任何地方中使用：1）java.sql.Date 2）java.sql.Time 3） java.sql.Timestamp。

说明：第 1 个不记录时间，getHours()抛出异常；第 2 个不记录日期，getYear()抛出异常；第 3 个在构造方法 super((time/1000)*1000)，fastTime 和 nanos 分开存储秒和纳秒信息。

反例： java.util.Date.after(Date)进行时间比较时，当入参是 java.sql.Timestamp 时，会触发 JDK BUG(JDK9 已修复)，可能导致比较时的意外结果。

5. 【强制】不要在程序中写死一年为 365 天，避免在公历闰年时出现日期转换错误或程序逻辑错误。

正例：

```
// 获取今年的天数
int daysOfThisYear = LocalDate.now().lengthOfYear();

// 获取指定某年的天数
LocalDate.of(2011, 1, 1).lengthOfYear();
```

反例：

```
// 第一种情况：在闰年 366 天时，出现数组越界异常
int[] dayArray = new int[365];
// 第二种情况：一年有效期的会员制，今年 1 月 26 日注册，硬编码 365 返回的却是 1 月 25 日
Calendar calendar = Calendar.getInstance();
calendar.set(2020, 1, 26);
calendar.add(Calendar.DATE, 365);
```

6. 【强制】数据查询时，查询数据区间，应该为>=beginDate && <endDate+1天（特殊情况除外）。

7. 【推荐】避免公历闰年 2 月问题。闰年的 2 月份有 29 天，一年后的那一天不可能是 2 月 29 日。

8. 【推荐】使用枚举值来指代月份。如果使用数字，注意 Date，Calendar 等日期相关类的月份

month 取值在 0-11 之间。

说明：参考 JDK 原生注释，Month value is 0-based. e.g., 0 for January.

正例： Calendar.JANUARY，Calendar.FEBRUARY，Calendar.MARCH 等来指代相应月份来进行传参或比较。

(六) 集合处理

1. 【强制】关于 hashCode 和 equals 的处理，遵循如下规则：

   1）只要重写 equals，就必须重写 hashCode。

   2）因为 Set 存储的是不重复的对象，依据 hashCode 和 equals 进行判断，所以 Set 存储的对象必须重写这两个方法。

   3）如果自定义对象作为 Map 的键，那么必须覆写hashCode 和 equals。

   说明 :String 因为重写了 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】判断所有集合内部的元素是否为空，使用 isEmpty()方法，而不是 size()==0 的方式。

   说明：前者的时间复杂度为 O(1)，而且可读性更好。

   正例：

```
Map<String, Object> map = new HashMap<>();
if(map.isEmpty()) {
        System.out.println("no element in this map.");
}
```

3. 【强制】在使用 java.util.stream.Collectors 类的 toMap()方法转为 Map 集合时，一定要使用含有参数类型为 BinaryOperator，参数名为 mergeFunction 的方法，否则当出现相同 key 值时会抛出 IllegalStateException 异常。

   说明：参数 mergeFunction 的作用是当出现 key 重复时，自定义对 value 的处理策略。

   正例：

```
List<Pair<String, Double>> pairArrayList = new ArrayList<>(3);
pairArrayList.add(new Pair<>("version", 6.19));
pairArrayList.add(new Pair<>("version", 10.24));
pairArrayList.add(new Pair<>("version", 13.14));
Map<String, Double> map = pairArrayList.stream().collect(
// 生成的map集合中只有一个键值对：
{version=13.14} Collectors.toMap(Pair::getKey, Pair::getValue,
(v1, v2) -> v2));
```

   反例：

```
String[] departments = new String[] {"iERP", "iERP", "EIBU"};
// 抛出 IllegalStateException 异常
Map<Integer, String> map = Arrays.stream(departments)
    .collect(Collectors.toMap(String::hashCode, str -> str));
```

4. 【强制】在使用 java.util.stream.Collectors 类的 toMap()方法转为 Map 集合时，一定要注意当 value 为 null 时会抛 NPE 异常。

   说明：在 java.util.HashMap 的 merge 方法里会进行如下的判断：

```
if (value == null || remappingFunction == null)
```

```
throw new NullPointerException();
```

反例：

```
List<Pair<String, Double>> pairArrayList = new ArrayList<>(2);
pairArrayList.add(new Pair<>("version1", 4.22));
pairArrayList.add(new Pair<>("version2", null));
Map<String, Double> map = pairArrayList.stream().collect(
// 抛出 NullPointerException 异常
Collectors.toMap(Pair::getKey, Pair::getValue, (v1, v2) -> v2));
```

5. 【强制】ArrayList 的 subList 结果不可强转成 ArrayList ,否则会抛出 ClassCastException 异常：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。

6. 【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。

7. 【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。

8. 【强制】在 subList 场景中，高度注意对父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。

9. 【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：

```
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 2 相同。

   4）大于 size，空间浪费，且在 size 处插入 null 值，存在 NPE 隐患。

10. 【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。

11. 【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[] { "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。

12. 【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，两者在接口调用赋值的场景中容易出错。

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>

13. 【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：

```
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);
```

14. 【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)
        ;
    }
}
```

15. 【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 会抛 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，交换两个对象判断结果并不互反，不符合第一个条件，在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;
    }
};
```

16. 【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，即<>
HashMap<String, String> userCache = new HashMap<>(16);
// 全省略方式
ArrayList<User> users = new ArrayList(10);
```

17. 【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，如果暂时无法确定集合大小，那么指定默认值（16）即可。

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。当放置的集合元素个数达千万级别时，不断扩容会严重影响性能。

18. 【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。

19. 【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。

20. 【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。

21. 【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()进行遍历去重或者判断包含操作。

(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：如下的代码输出是什么？

```
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，禁止不采用大括号的编码方式：if (condition) statements;

4. 【强制】三目运算符 condition? 表达式 1：表达式 2 中，高度注意表达式 1 和 2 在类型对齐时，可能抛出因自动拆箱导致的 NPE 异常。

说明：以下两种场景会触发类型对齐的拆箱操作：

1） 表达式 1 或表达式 2 的值只要有一个是原始类型。

2） 表达式 1 或表达式 2 的值的类型不一致，会强制拆箱升级成表示范围更大的那个类型。反例：

```
Integer a = 1;
Integer b = 2;
Integer c = null;
Boolean flag = false;
// a*b 的结果是 int 类型，那么 c 会强制拆箱成 int 类型，抛出 NPE 异常
Integer result=(flag? a*b : c);
```

5. 【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。

6. 【推荐】当某个方法的代码行数超过 10 行时，return / throw 等中断逻辑的右大括号后加一个空行。

说明：这样做逻辑清晰，有利于代码阅读时重点关注。

7. 【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：如果非使用 if()...else if()...else...方式表达逻辑，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 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");
}
```

8. 【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：

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

```
        ...
    }
```

反例：

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

9. 【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：

```
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();
}
```

10. 【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）。

11. 【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。

12. 【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。

反例：某业务系统，提供一个用户批量查询的接口，API 文档上有说最多查多少个，但接口实现上没做任何保护，导致调用方传了一个 1000 的用户 id 数组过来后，查询信息后，内存爆了。

13. 【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。

14. 【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。

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。

(九) 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。
   说明：在设置模板时，注意 IDEA 的@author 为`${USER}`，而 eclipse 的@author 为`${user}`，大小写有区别，而日期的设置统一为 yyyy/MM/dd 的格式。
   正例：
   ```
   /**
    * @author yangguanbao
    * @date 2016/10/31
    */
   ```

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. 【推荐】在类中删除未使用的任何字段和方法；在方法中删除未使用的任何参数声明与内部变量。

9. 【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。后者建议直接删掉即可，假如需要查阅历史代码，登录代码仓库即可。

10. 【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。

11. 【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。

12. 【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。

(十) 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】避免用 Apache Beanutils 进行属性的 copy。

说明：Apache BeanUtils 性能较差，可以使用其他方案比如 Spring BeanUtils, Cglib BeanCopier，注意均是浅拷贝。

3. 【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。

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】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：

```
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;
    }
```

```
}
```

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。如：

```
public static void hello() {
    /// 业务方通知活动暂停
    // Business business = new Business();
    // business.active();
    System.out.println("it's finished");
}
```

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。

    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。

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”；而应用内部推荐异常抛出。

    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) {...}