

Java代码开发规范指南

大连海心信息工程有限公司

二〇二二年一月

声 明

1. **本规范是大连海心信息工程有限公司内部Java代码开发规范指南。**
2. **本规范知识产权归大连海心信息工程有限公司所有。**
3. **本规范仅供内部使用，禁止任何形式对此规范向第三方传播。**
4. **本规范最终解释权归大连海心信息工程有限公司所有。**

**目 录**

[1. 编程规约 4](#_Toc93929738)

[1.1 命名风格 4](#_Toc93929739)

[1.2 常量定义 7](#_Toc93929740)

[1.3 代码格式 8](#_Toc93929741)

[1.4 OOP 规约 11](#_Toc93929742)

[1.5 集合处理 15](#_Toc93929743)

[1.6 并发处理 18](#_Toc93929744)

[1.7 控制语句 21](#_Toc93929745)

[1.8 注释规约 23](#_Toc93929746)

[1.9 其它 25](#_Toc93929747)

[2. 异常日志 26](#_Toc93929748)

[2.1 异常处理 26](#_Toc93929749)

[2.2 日志规约 28](#_Toc93929750)

[3. 单元测试 30](#_Toc93929751)

[4. 安全规约 32](#_Toc93929752)

[5. 设计规约 33](#_Toc93929753)

[6. 注释&注解规范 36](#_Toc93929754)

[6.1 代码注释 36](#_Toc93929755)

[6.2 文档注解 39](#_Toc93929756)

[7. 开发示例 41](#_Toc93929757)

[8. 专有名词解释 48](#_Toc93929758)

1. 编程规约
   1. 命名风格√
2. 【强制】代码中的命名均不能以下划线或美元符号开始，也不能以下划线或美元符号结束。

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

1. 【强制】代码中的命名严禁使用拼音与英文混合的方式，更不允许直接使用中文的方式。

说明：正确的英文拼写和语法可以让阅读者易于理解，避免歧义。注意，即使纯拼音命名方式也要避免采用。

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

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

1. 【强制】类名使用UpperCamelCase风格，但以下情形例外：DO/BO/DTO/VO/AO/PO/UID 等。

正例：MarcoPolo / UserDO / XmlService / TcpUdpDeal / TaPromotion。

反例：macroPolo / UserDo / XMLService / TCPUDPDeal / TAPromotion。

1. 【强制】方法名、参数名、成员变量、局部变量都统一使用lowerCamelCase风格，必须遵从驼峰形式。

正例： localValue / getHttpMessage() / inputUserId。

1. 【强制】常量命名全部大写，单词间用下划线隔开，力求语义表达完整清楚，不要嫌名字长。

正例：MAX\_STOCK\_COUNT。

反例：MAX\_COUNT。

1. 【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。
2. 【强制】类型与中括号紧挨相连来表示数组。

正例：定义整形数组 int[] arrayDemo;

反例：在 main 参数中，使用 String args[]来定义。

1. 【强制】POJO 类中布尔类型的变量，都不要加 is 前缀，否则部分框架解析会引起序列化错误。

反例：定义为基本数据类型 Boolean isDeleted 的属性，它的方法也是 isDeleted()，RPC框架在反向解析的时候，“误以为”对应的属性名称是 deleted，导致属性获取不到，进而抛出异常。

1. 【强制】包名统一使用小写，点分隔符之间有且仅有一个自然语义的英语单词。包名统一使用单数形式，但是类名如果有复数含义，类名可以使用复数形式。

正例：应用工具类包名为 com.alibaba.ai.util、类名为 MessageUtils（此规则参考 spring

的框架结构）。

1. 【强制】杜绝完全不规范的缩写，避免望文不知义。

反例：AbstractClass“缩写”命名成 AbsClass；condition“缩写”命名成 condi，此类随意缩写严重降低了代码的可阅读性。

1. 【推荐】为了达到代码自解释的目标，任何自定义编程元素在命名时，使用尽量完整的单词组合来表达其意。

正例：在 JDK 中，表达原子更新的类名为：AtomicReferenceFieldUpdater。

反例：变量 int a 的随意命名方式。

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

说明：将设计模式体现在名字中，有利于阅读者快速理解架构设计理念。

正例：public class OrderFactory;

public class LoginProxy;

public class ResourceObserver;

1. 【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。

1. 接口和实现类的命名有两套规则：
2. 【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。

1. 【推荐】如果是形容能力的接口名称，取对应的形容词为接口名（通常是–able的形式）。

正例：AbstractTranslator 实现 Translatable 接口。

1. 【参考】枚举类名建议带上 Enum 后缀，枚举成员名称需要全大写，单词间用下划线隔开。

说明：枚举其实就是特殊的类，域成员均为常量，且构造方法被默认强制是私有。

正例：枚举名字为 ProcessStatusEnum的成员名称：SUCCESS/UNKNOWN\_REASON。

1. 【参考】各层命名规约：
2. Service/DAO 层方法命名规约：
3. 获取单个对象的方法用get做前缀。
4. 获取多个对象的方法用list做前缀，复数形式结尾如：listObjects。
5. 获取统计值的方法用count做前缀。
6. 插入的方法用save/insert做前缀。
7. 删除的方法用remove/delete做前缀。
8. 修改的方法用update做前缀。
9. 领域模型命名规约：
10. 数据对象：xxxDO，xxx 即为数据表名。
11. 数据传输对象：xxxDTO，xxx 为业务领域相关的名称。
12. 展示对象：xxxVO，xxx 一般为网页名称。
13. POJO 是 DO/DTO/BO/VO 的统称，禁止命名成xxxPOJO。
    1. 常量定义√
14. 【强制】不允许任何魔法值（即未经预先定义的常量）直接出现在代码中。

反例：String key = "Id#taobao\_" + tradeId;

cache.put(key, value);

1. 【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?

1. 【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。

1. 【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。
2. 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。
3. 应用内共享常量：放置在一方库中，通常是子模块中的constant目录下。

反例：易懂变量也要统一定义成应用内共享常量，两位攻城师在两个类中分别定义了表示“是”的变量：

类 A 中：public static final String YES = "yes";

类 B 中 ：public static final String YES = "y";

A.YES.equals(B.YES)，预期是 true，但实际返回为 false，导致线上问题。

1. 子工程内部共享常量：即在当前子工程的 constant 目录下。
2. 包内共享常量：即在当前包下单独的 constant 目录下。
3. 类内共享常量：直接在类内部 private static final 定义。
4. 【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;

}

}

* 1. 代码格式√

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：
2. 左大括号前不换行。
3. 左大括号后换行。
4. 右大括号前换行。
5. 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。
6. 【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)

1. 【强制】if/for/while/switch/do 等保留字与括号之间都必须加空格。
2. 【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。

1. 【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

}

}

1. 【强制】注释的双斜线与注释内容之间有且仅有一个空格。

正例：

// 这是示例注释，请注意在双斜线之后有一个空格

String ygb = new String();

1. 【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：
2. 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。
3. 运算符与下文一起换行。
4. 方法调用的点符号与下文一起换行。
5. 方法调用中的多个参数需要换行时，在逗号后进行。
6. 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);

1. 【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);

1. 【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。
2. 【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。

1. 【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。

1. 【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。

* 1. 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。

1. 【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) {...}

1. 【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。
2. 【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。

1. 【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）。

1. 【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。

1. 关于基本数据类型与包装数据类型的使用标准如下：
2. 【强制】所有的 POJO 类属性必须使用包装数据类型。
3. 【强制】RPC 方法的返回值和参数必须使用包装数据类型。
4. 【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。

1. 【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。

1. 【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。

1. 【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。
2. 【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。

1. 【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。

1. 【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);

1. 【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。
2. 【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。

1. 【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;

}

}

1. 【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";

}

1. 【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：
2. 不允许被继承的类，如：String 类。
3. 不允许修改引用的域对象。
4. 不允许被重写的方法，如：POJO 类的setter 方法。
5. 不允许运行过程中重新赋值的局部变量。
6. 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。
7. 【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。

1. 【推荐】类成员与方法访问控制从严：
2. 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。
3. 工具类不允许有 public 或 default 构造方法。
4. 类非 static 成员变量并且与子类共享，必须是 protected。
5. 类非 static 成员变量并且仅在本类使用，必须是 private。
6. 类 static 成员变量如果仅在本类使用，必须是 private。
7. 若是 static 成员变量，考虑是否为 final。
8. 类成员方法只供类内部调用，必须是 private。
9. 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。

* 1. 集合处理

1. 【强制】关于 hashCode 和 equals 的处理，遵循如下规则：
2. 只要重写 equals，就必须重写 hashCode。
3. 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。
4. 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。

1. 【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。

1. 【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。
2. 【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。

1. 【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。

1. 【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>。

1. 【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？

1. 【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;

}

};

1. 【推荐】集合泛型定义时，在 JDK7 及以上，使用 diamond 语法或全省略。

说明：菱形泛型，即 diamond，直接使用<>来指代前边已经指定的类型。

正例：

// <> diamond 方式

HashMap<String, String> userCache = new HashMap<>(16);

// 全省略方式

ArrayList<User> users = new ArrayList(10);

1. 【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。

1. 【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。

1. 【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。

1. 【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。

1. 【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。
   1. 并发处理
2. 【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。

1. 【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");

...

}

}

1. 【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。

1. 【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。

1. 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。

1. 【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。

1. 【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。

1. 【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。

1. 【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。

1. 【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。
2. 【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。

1. 【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。

1. 【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...

}

1. 【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）。
2. 【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。
3. 【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。
   1. 控制语句√
4. 【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。
5. 【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;
6. 【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。

1. 【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;

}

1. 【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) && (...) || (...)) {

...

}

1. 【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）。
2. 【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。

1. 【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。
2. 【参考】下列情形，需要进行参数校验：
3. 调用频次低的方法。
4. 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。
5. 需要极高稳定性和可用性的方法。
6. 对外提供的开放接口，不管是 RPC/API/HTTP 接口。
7. 敏感权限入口。
8. 【参考】下列情形，不需要进行参数校验：
9. 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。
10. 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。
11. 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。
    1. 注释规约√
12. 【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。

1. 【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。

1. 【强制】所有的类都必须添加创建者和创建日期。
2. 【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。
3. 【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。
4. 【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。

1. 【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。

1. 【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）。

1. 【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。
2. 【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。

1. 【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。
2. 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）。

1. 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。

* 1. 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。

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

1. 【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。

1. 【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。

1. 【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。
2. 【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。
3. 【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。

1. 【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。
2. 【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。

1. 异常日志
   1. 异常处理
2. 【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) {…}

1. 【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。

1. 【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。

1. 【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。
2. 【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。
3. 【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。

1. 【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。

1. 【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。

1. 【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。

1. 【推荐】防止 NPE，是程序员的基本修养，注意 NPE 产生的场景：
2. 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。
3. 数据库的查询结果可能为 null。
4. 集合里的元素即使 isNotEmpty，取出的数据元素也可能为 null。
5. 远程调用返回对象时，一律要求进行空指针判断，防止 NPE。
6. 对于 Session 中获取的数据，建议 NPE 检查，避免空指针。
7. 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。
8. 【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。
9. 【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。
3. 【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) {...}

* 1. 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);

1. 【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。
2. 【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。

1. 【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);

1. 【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">
2. 【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);

1. 【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？

1. 【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。

1. 【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. 单元测试
3. 【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）。

1. 【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。
2. 【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。

1. 【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。

1. 【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。

1. 【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。

1. 【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。

1. 【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。

1. 【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。

1. 【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。

1. 【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。

1. 【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。
2. 【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。
3. 【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。
4. 【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。

1. 【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。

1. 安全规约√
2. 【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。

1. 【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:158\*\*\*\*9119，隐藏中间 4 位，防止隐私泄露。

1. 【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。
2. 【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。

1. 【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。
2. 【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。

1. 【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。

1. 【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。
2. 设计规约
3. 【强制】存储方案和底层数据结构的设计获得评审一致通过，并沉淀成为文档。

说明：有缺陷的底层数据结构容易导致系统风险上升，可扩展性下降，重构成本也会因历史数据迁移和系统平滑过渡而陡然增加，所以，存储方案和数据结构需要认真地进行设计和评审，生产环境提交执行后，需要进行double check。

正例：评审内容包括存储介质选型、表结构设计能否满足技术方案、存取性能和存储空间能否满足业务发展、表或字段之间的辩证关系、字段名称、字段类型、索引等；数据结构变更（如在原有表中新增字段）也需要进行评审通过后上线。

1. 【强制】在需求分析阶段，如果与系统交互的 User 超过一类并且相关的 User Case 超过 5个， 使用用例图来表达更加清晰的结构化需求。
2. 【强制】如果某个业务对象的状态超过3个，使用状态图来表达并且明确状态变化的各个触发条件。

说明：状态图的核心是对象状态，首先明确对象有多少种状态，然后明确两两状态之间是否存在直接转换关系，再明确触发状态转换的条件是什么。

正例：淘宝订单状态有已下单、待付款、已付款、待发货、已发货、已收货等。比如已下单与已收货这两种状态之间是不可能有直接转换关系的。

1. 【强制】如果系统中某个功能的调用链路上的涉及对象超过3个，使用时序图来表达并且明确各调用环节的输入与输出。

说明：时序图反映了一系列对象间的交互与协作关系，清晰立体地反映系统的调用纵深链路。

1. 【强制】如果系统中模型类超过5个，并且存在复杂的依赖关系，使用类图来表达并且明确类之间的关系。

说明：类图像建筑领域的施工图，如果搭平房，可能不需要，但如果建造蚂蚁 Z 空间大楼，肯定需要详细的施工图。

1. 【强制】如果系统中超过2个对象之间存在协作关系，并且需要表示复杂的处理流程，使用活动图来表示。

说明：活动图是流程图的扩展，增加了能够体现协作关系的对象泳道，支持表示并发等。

1. 【推荐】需求分析与系统设计在考虑主干功能的同时，需要充分评估异常流程与业务边界。

反例：用户在淘宝付款过程中，银行扣款成功，发送给用户扣款成功短信，但是支付宝入款时由于断网演练产生异常，淘宝订单页面依然显示未付款，导致用户投诉。

1. 【推荐】类在设计与实现时要符合单一原则。

说明：单一原则最易理解却是最难实现的一条规则，随着系统演进，很多时候，忘记了类设计的初衷。

1. 【推荐】谨慎使用继承的方式来进行扩展，优先使用聚合/组合的方式来实现。

说明：不得已使用继承的话，必须符合里氏代换原则，此原则说父类能够出现的地方子类一定能够出现，比如，“把钱交出来”，钱的子类美元、欧元、人民币等都可以出现。

1. 【推荐】系统设计时，根据依赖倒置原则，尽量依赖抽象类与接口，有利于扩展与维护。

说明：低层次模块依赖于高层次模块的抽象，方便系统间的解耦。

1. 【推荐】系统设计时，注意对扩展开放，对修改闭合。

说明：极端情况下，交付的代码都是不可修改的，同一业务域内的需求变化，通过模块或类的扩展来实现。

1. 【推荐】系统设计阶段，共性业务或公共行为抽取出来公共模块、公共配置、公共类、公共方法等，避免出现重复代码或重复配置的情况。

说明：随着代码的重复次数不断增加，维护成本指数级上升。

1. 【推荐】避免如下误解：敏捷开发 = 讲故事 + 编码 + 发布。

说明：敏捷开发是快速交付迭代可用的系统，省略多余的设计方案，摒弃传统的审批流程，但核心关键点上的必要设计和文档沉淀是需要的。

反例：某团队为了业务快速发展，敏捷成了产品经理催进度的借口，系统中均是勉强能运行但像面条一样的代码，可维护性和可扩展性极差，一年之后，不得不进行大规模重构，得不偿失。

1. 【参考】系统设计主要目的是明确需求、理顺逻辑、后期维护，次要目的用于指导编码。

说明：避免为了设计而设计，系统设计文档有助于后期的系统维护，所以设计结果需要进行分类归档保存。

1. 【参考】设计的本质就是识别和表达系统难点，找到系统的变化点，并隔离变化点。

说明：世间众多设计模式目的是相同的，即隔离系统变化点。

1. 【参考】系统架构设计的目的：

* 确定系统边界。确定系统在技术层面上的做与不做。
* 确定系统内模块之间的关系。确定模块之间的依赖关系及模块的宏观输入与输出。
* 确定指导后续设计与演化的原则。使后续的子系统或模块设计在规定的框架内继续演化。
* 确定非功能性需求。非功能性需求是指安全性、可用性、可扩展性等。

1. 注释&注解规范
   1. 代码注释
2. 【强制】对象头添加上要添加必要的注释信息。

正例：

/\*\*

\* 机构控制器

\*

\* @author 作者

\* @date 时间 - 2021/3/22 16:04

\* @description 控制器描述

\*/

public class SysDeptController {

// ....

}

1. 【强制】服务接口的方法上必须添加注释信息，其中@author作者，如果是创建类的作者添加的，可以不需要添加，如果是其他开发人员添加，则需要添加作者信息。

正例：

/\*\*

\* 机构服务

\*

\* @author 作者

\* @date 时间-2021/3/22 9:39

\* @description 类控制器描述

\*/

public interface SysDeptService {

/\*\*

\* 添加系统机构

\*

\* @param sysDeptCreateInputDto 对象字段描述

\* @return Result 返回结果说明

\* @author 添加的作者

\* @throws Exception 异常信息描述

\*/

Result addSysDept(SysDeptCreateInputDto sysDeptCreateInputDto) throws Exception ;

// ....

}

1. 【强制】如果对象中创建的私有的方法，必须在其方法上添加注释信息。如果是创建对象的作者创建，则该私有方法上面不需要添加创建作者，如果是其他小伙伴创建的，需要添加创建的作者。

正例：

/\*\*

\* 功能服务实现

\*

\* @author 作者

\* @date 2021/3/22 16:04

\* @description 菜单管理服务

\*/

public class SysFuncServiceImpl {

/\*\*

\* 向菜单树节点添加菜单信息

\*

\* @param tree 菜单树节点

\* @param sysFuncExtra 菜单信息

\* @return Result 树节点

\* @description 向菜单树节点添加菜单信息

\* @author 创建的作者

\*/

private Tree<String> putExtra(Tree<String> tree,SysFuncExtra sysFuncExtra){

tree.putExtra("routerUrl", sysFuncExtra.getRouterUrl());

tree.putExtra("componentUrl", sysFuncExtra.getComponentUrl());

// ....

return tree;

}

}

1. 【强制】业务处理过程中，设置一些变量，控制业务处理逻辑，需要添加一些说明。

正例：

/\*\*

\* 用于控制输入框的编辑功能（可录入或者不可修改）

\* 值 : true 代表可录入，false 代表不可录入，默认为 true

\* 这个字段在每次请求调用时都需要进行设置

\* add by author and 创建时间

\* 新增原因：用户控制编辑器的修改操作

\*/

private boolean hxsoftModifyFlag = true;

1. 【强制】创建数据表对应的持久对象的时候，要添加具体的注释，标明每一字段对应的含义，必须使用/\*\*内容\*/格式，不得使用// xxx方式。

正例：

/\*\*

\* 对象描述

\*

\* @author 作者

\* @date 时间-2021-03-26 9:39

\*/

@Data

public class SysDicDetail extends BaseEntity implements Serializable {

/\*\*

\* 字典名称

\*/

private String dicName;

/\*\*

\* 字典数据名称

\*/

private String dicDataName;

/\*\*

\* 字典数据中文名称

\*/

private String dicDataNameCn;

/\*\*

\* 字典数据类型（1：系统，2：用户）

\*/

private Integer dicDataType;

// 省略其他代码片段

}

1. 【推荐】业务代码的实现过程中，对一些功能代码要添加注释，标明每一块代码的实现功能。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。

正例：

/\*\*

\* 菜单管理服务

\*

\* @author zhangpishui

\* @date 2021/3/22 16:04

\*/

@Service

public class SysFuncServiceImpl implements SysFuncService {

/\*\*

\* 查询菜单树信息

\*

\* @param funcId 菜单id

\* @return Result 查询机构树结果

\*/

@Override

public Result<List<Tree<String>>> getFuncTree(Long funcId) {

// 从缓存中获取数据

List<SysFuncExtra> sysFuncList = sysFuncRedis.getFuncSet();

// 如果从缓存获取不到

if (sysFuncList == null) {

log.info("--------------------从数据库中读取菜单信息--------------------");

// 从数据库获取所有菜单及按钮

sysFuncList = sysFuncDao.getFuncList();

}

/\*

自定义tree配置

将4个关键字段（id,parentId,weight,name）字段重命

\*/

TreeNodeConfig treeNodeConfig = new TreeNodeConfig();

treeNodeConfig.setWeightKey("funcLevelSeqNo");

treeNodeConfig.setParentIdKey("parentFuncId");

treeNodeConfig.setIdKey("funcId");

treeNodeConfig.setNameKey("funcName");

// 构建tree结构

List<Tree<String>> treeNodes = TreeUtil.build(sysFuncList, String.valueOf(funcId), treeNodeConfig,

(treeNode, tree) -> {

tree.setId(String.valueOf(treeNode.getFuncId()));

tree.setParentId(String.valueOf(treeNode.getParentFuncId()));

tree.setWeight(treeNode.getFuncLevelSeqNo());

tree.setName(treeNode.getFuncName());

// 设置自定义字段

this.putExtra(tree,treeNode);

});

if(!String.valueOf(funcId).equals("0")){

// 封装父级节点

Tree<String> tree = new Tree();

// 获取父级节点

SysFuncExtra parentFunc = sysFuncList.stream().filter(func -> func.getFuncId().equals(funcId) ).collect(Collectors.toList()).get(0);

tree.putExtra("funcId",parentFunc.getFuncId());

tree.putExtra("funcName",parentFunc.getFuncName());

tree.putExtra("parentFuncId",parentFunc.getParentFuncId());

tree.putExtra("funcLevelSeqNo",parentFunc.getFuncLevel());

// 设置自定义字段

this.putExtra(tree,parentFunc);

// 封装子节点

tree.setChildren(treeNodes);

List<Tree<String>> listResult = new ArrayList<>();

listResult.add(tree);

return Result.success(listResult);

}

return Result.success(treeNodes);

}

* 1. 文档注解

1. 【强制】控制层的对象头添加上要添加必要的Swagger注解信息，让其自动生成可视化的文档页面。其中@Api(tags="控制层描述")控制层描述，如果接收的参数不是对象，通常使用@ApiImplicitParams({@ApiImplicitParam()}) 添加每一个输入字段的描述。

正例：

/\*\*

\* 数据字典表控制器

\*

\* @author 作者

\* @date 2021-03-26

\* /

@Slf4j

@RestController

@RequestMapping("/dic")

@Api(tags="字典表管理")

public class SysDicDetailController {

@ApiOperation("校验数据字典名唯一性")

@GetMapping("checkDicNameUnique")

@ApiImplicitParams({

@ApiImplicitParam(name = "dicName", value = "字典名称" , paramType = "query", required = true,example = "DEPT\_TYPE\_CLASS") ,

})

Result checkDicNameUnique(@RequestParam String dicName){

return sysDicDetailService.checkDicNameUnique(dicName);

}

@ApiOperation("修改数据字典数据")

@PostMapping("updateDicDetail")

Result updateDicDetail(@RequestBody SysDicDetailInputDto sysDicDetailInputDto){

return sysDicDetailService.updateDicDetail(sysDicDetailInputDto);

}

// ....

}

1. 【强制】如果输入的参数是一个对象或者返回的也是一个对象，则该对象中要添加Swagger注解属性。首先是@ApiModel()添加对象描述，@ApiModelProperty()添加对象属性描述。

正例：

/\*\*

\* 字典详情表入参

\*

\* @author 作者

\* @date 2021/4/6 8:55

\*/

@Data

@ApiModel(description = "字典详情表入参")

public class SysDicDetailInputDto implements Serializable {

/\*\*

\* 字典名称

\*/

@ApiModelProperty(value = "字典名称",name = "dicName", required = true , example = "DEPT\_TYPE\_CLASS")

private String dicName;

/\*\*

\* 字典数据名称

\*/

@ApiModelProperty(value = "字典数据名称",name = "dicDataName", required = true , example = "1")

private String dicDataName;

/\*\*

\* 字典数据中文名称

\*/

@ApiModelProperty(value = "字典数据中文名称",name = "dicDataNameCn", required = true , example = "集团")

private String dicDataNameCn;

/\*\*

\* 字典数据序号

\*/

@ApiModelProperty(value = "字典数据序号",name = "dicDataSeqNo", required = true , example = "1")

private Integer dicDataSeqNo;

}

1. 开发示例
2. 收费7.0业务代码中没有定义统一的注释规范，导致开发人员随意的去定义。

反例：版本在作者的上面，但是使用version表示日期有点不合适。

/\*\*

\* 该类是所有异常处理类的基类

\*

\* @version 2013-5-16

\* @TODO 该类是所有异常处理类的基类

\* @author leibf

\*/

public class BaseException extends RuntimeException{

}

反例：版本在作者的下面。

/\*\*

\* 面积名称定义表 Controller类

\*

\* @author lyp

\* @version 2018-04-19

\*/

@Controller

@RequestMapping("/sfszAreaName")

public class SfszAreaNameController extends BaseController {

}

反例：没有添加任何头注释信息。

@Controller

@RequestMapping("/sfszStandardNew")

public class SfszStandardNewController extends BaseController {

}

正例：在每一个对象的头部添加相应的注释，标明该对象的具体含义，这样可以大大提高对象的可读性。

/\*\*

\* 对象描述

\*

\* @author:作者

\* @date: 2019/5/13 09:40 - 时间

\*/

public class SfChargeCommonDao extends BaseDao {

}

1. 收费7.0业务代码，很多接口类，仅仅对方法进行描述，并没有添加对参数、创建人、返回结果和异常的表述。

反例：该接口方法中，仅仅添加接口的描述，其他信息都没有添加。

/\*\*

\* 房屋新增

\*/

Integer addHouse(SfChargeCommonModel sfChargeCommonModel) throws Exception;

正例：在每一接口类中的每一个接口方法上，要添加尽量详细的接口描述，方便后续的维护工作，如果创建的接口不是你创建的，要在方法的注释上标注具体的作者姓名。

/\*\*

\* 房屋新增 - 接口描述

\*

\* @param sfChargeCommonModel 对象描述

\* @return 返回值描述

\* @author:作者

\* @throws Exception 异常描述

\*/

Integer addHouse(SfChargeCommonModel sfChargeCommonModel) throws Exception;

1. 收费7.0业务代码，很多对象属性都没有添加注释，导致阅读起来特别浪费时间。

反例：该示例代码中没有添加属性的注释，所有的属性都需要开发人员去猜测，大大的增加的代码的开发和维护成本。

/\*\*

\* 对象描述

\*

\* @author:作者

\* @date: 2019/5/13 09:40 - 时间

\*/

public class ExecuteAuditModel {

private List<String> detailIds;

private String type = "select";

private String planId;

private String auditGroup;

private String inspector;

private String auditResult;

// 省略其他代码片段

}

正例：在每一个属性的头部添加相应的注释，标明该属性的具体含义，这样可以大大提高代码的可读性，也增强代码对象的复用率。

/\*\*

\* 对象描述

\*

\* @author:作者

\* @date: 2019/5/13 09:40 - 时间

\*/

public class ExecuteAuditModel {

/\*\* 详情Ids \*/

private List<String> detailIds;

/\*\* 类型 \*/

private String type = "select";

/\*\* 计划Id \*/

private String planId;

/\*\* 审计组\*/

private String auditGroup;

/\*\* 稽查人 \*/

private String inspector;

/\*\* 稽查结果 \*/

private String auditResult;

// 省略其他代码片段

}

1. 收费业务的处理过程中，很多业务代码没有添加注释，就会导致遇到问题进行梳理的过程中特别的痛苦。

反例：收费7.0业务代码中的文件上传功能，其中一段代码是编写文件上传具体功能，其中没有添加任何注释。

if ("0".equals(tem)) {

byte[] fileBytes = item.get();

String userCode = SessionUtil.getUserCode(request);

FileMetas fileMetas = new FileMetas(fileName, fileSuffix, userCode, fileSize);

fastdfs\_file\_path = fileStorageOperations.upload(fileBytes, fileMetas);

fileMetas.getMetaList()));

} else {

ByteArrayInputStream in = new ByteArrayInputStream(item.get());

BufferedImage image = ImageIO.read(in);

image = FileUtil.rotateImage(image,Integer.valueOf(tem));

ByteArrayOutputStream out = new ByteArrayOutputStream();

ImageIO.write(image,fileSuffix,out);

byte[] fileBytes = out.toByteArray();

String userCode = SessionUtil.getUserCode(request);

FileMetas fileMetas = new FileMetas(fileName, fileSuffix, userCode, fileSize);

fastdfs\_file\_path = fileStorageOperations.upload(fileBytes, fileMetas);

}

正例：在业务代码中添加必要的代码注释，既能体现你的专业性，也能方便其他小伙伴的阅读和修正。

// 执行Update， 更新应收台账（SF\_CHARGE）表

sfChargeCommonDao.updateSfChargeT(sfChargeCommonModel, chargeModel);

// 执行Update， 更新应收台账房屋（SF\_HOUSE）表

sfChargeCommonDao.updateSfHouseT(sfChargeCommonModel, chargeModel);

// 修改面积 更新应收台账记录表（Sf\_Change\_record）表(新增信息)

if (sfChargeCommonModel.getLogType().indexOf(area) != -1) {

sfChargeCommonDao.updateSfChange\_recordTAreaAfter(sfChargeCommonModel);

}

// 修改性质 更新应收台账记录表（Sf\_Change\_record）表

else if (sfChargeCommonModel.getLogType().indexOf(type) != -1) {

sfChargeCommonDao.updateSfChange\_recordTTypeAfter(sfChargeCommonModel);

}

1. 收费7.0前端业务代码，很多前端控件没有添加相应的注释说明，维护人员无法知道该组件具体的功能。

反例：前端页面控件开发过程中，添加的控件一定要添加相应的注释信息，描述该控件具体的作用。但是收费7.0业务代码的前端控件中很多都没有添加注释信息。

<div class="searchBtnWrap" style="text-align:right">

<hx:privilege url="/rosterResultCache/save.act" id="btn\_save" btnType="save" type="button" style="disabled" value="保存" onclick="javascript:btn\_save\_click(this.form);" />

<input class="btnClose MyResetBtn" type="button" value="关闭" onclick="closeWindow();"/>

</div>

正例：收费7.0业务代码的前端控件中添加注释信息。

<!-- 表单按钮区 保存和关闭控件-->

<div class="searchBtnWrap" style="text-align:right">

<hx:privilege url="/rosterResultCache/save.act" id="btn\_save" btnType="save" type="button" style="disabled" value="保存" onclick="javascript:btn\_save\_click(this.form);" />

<input class="btnClose MyResetBtn" type="button" value="关闭" onclick="closeWindow();"/>

</div>

1. 收费7.0前端业务代码，很多前端JS函数没有添加相应的注释，导致后期维护人员无法快速的了解该函数具体的功能。

反例：很多前端JS代码仅仅添加注释说明，缺少作者、参数和创建时间的说明。

/\*\*

\* 保存按钮的js方法

\*/

function btn\_save\_click(fm){

if(validForm("rosterQualificationForm")){

disabledBtn("btn\_save");

showProgressBar("操作进行中,请稍候...");

var url = controllerUrl + "/rosterQualification/save.act";

ajaxFormRequest(url,"doCallBack",null,fm);

}

}

正例：将前端JS代码函数添加详细的注释信息，方便前端业务代码的维护。

/\*\*

\* 保存按钮的js方法

\*

\* @param fm 参数描述

\* @date: 2019/5/13 09:40 - 时间

\* @author:作者

\*/

function btn\_save\_click(fm){

if(validForm("rosterQualificationForm")){

disabledBtn("btn\_save");

showProgressBar("操作进行中,请稍候...");

var url = controllerUrl + "/rosterQualification/save.act";

ajaxFormRequest(url,"doCallBack",null,fm);

}

}

1. 收费7.0业务代码，类的名字必须由大写字母开头而单词中的其他字母均为小写；如果类名称由多个单词组成，则每个单词的首字母均应为大写例如TestPage。由于类是设计用来代表对象的，所以在命名类时应尽量选择名词。

反例：收费7.0业务代码中很多没有遵循这条设计原则，其中使用下划线分割（红色字体标注）。

/\*\*

\* 对象描述

\*

\* @author:作者

\* @date: 2019/5/13 09:40 - 时间

\*/

public class Cus\_catalogModel implements BaseModel

{

/\*\*

\* 目录id

\*/

@NotBlank

@Length(max=32,message="长度不能超过32" )

private String catalog\_id;

/\*\*

\* 父目录id

\*/

@NotBlank

@Length(max=32,message="长度不能超过32" )

private String parent\_id;

/\*\*

\* 目录名称

\*/

@NotBlank

@Length(max=100,message="长度不能超过32" )

private String catalog\_name;

// 省略其他代码片段

}

反例：收费7.0业务代码中很多没有遵循这条设计原则，其中全部使用小写字母（红色字体标注）。

/\*\*

\* 对象描述

\*

\* @author:作者

\* @date: 2019/5/13 09:40 - 时间

\*/

public class MDHouseBatchImportModel {

/\*\* 换热站 \*/

private String stationname;

/\*\* 机组 \*/

private String unitname;

/\*\* 小区 \*/

private String regionname;

/\*\* 大楼 \*/

private String buildingname;

/\*\* 行政区 \*/

private String adname;

/\*\* 建筑年度 \*/

private String builtypesrs;

// 省略其他代码片段

}

正例：收费7.0业务代码中遵循这条设计原则，其中使用驼峰结构定义字段。

/\*\*

\* 对象描述

\*

\* @author:作者

\* @date: 2019/5/13 09:40 - 时间

\*/

public class ExecuteAuditModel {

/\*\* 详情Ids \*/

private List<String> detailIds;

/\*\* 类型 \*/

private String type = "select";

/\*\* 计划Id \*/

private String planId;

/\*\* 审计组 \*/

private String auditGroup;

/\*\* 稽查人 \*/

private String inspector;

/\*\* 稽查结果 \*/

private String auditResult;

// 省略其他代码片段

}

1. 任何写代码的人都希望在调试生产代码时获得更多的日志信息。日志的重要性在一般的开发和维护中都很重要。

反例：收费7.0业务代码中很多业务逻辑中没有添加日志逻辑。

public JsonResult insertHeatWork(Md\_heat\_networkModel model) throws Exception {

// 判断热网编号是否有重复的或者热网名称是否有重复

Map<String, Object> condition = new HashMap<String, Object>();

StringBuffer conditions = new StringBuffer();

StringBuffer params = new StringBuffer();

// 2020收费基础版本-基础信息-管网信息-热网管理对应

// 判断[热网名称]是否有重复

conditions.append(" networkname = ?");

params.append(model.getNetworkname());

condition.put("condition", conditions.toString());

condition.put("params", params.toString());

if (heatnetworks.size() > 0) {

return new JsonResult(false, "新增失败！该热网名称已存在。");

}

// 判断[热网编号]是否有重复

if (!StringUtil.isNullString(model.getNetworkcode())) {

conditions.setLength(0);

params.setLength(0);

conditions.append(" networkcode = ?");

params.append(model.getNetworkcode());

condition.clear();

condition.put("condition", conditions.toString());

condition.put("params", params.toString());

heatnetworks = this.mdHeatNetworkCustomDao.selectMdHeatNetworkByCondition(condition);

if (heatnetworks.size() > 0) {

return new JsonResult(false, "新增失败！该热网编号已存在。");

}

}

// 省略的业务代码

正例：收费7.0业务代码中添加的日志逻辑。

// 省略业务逻辑代码

SysLogger.info("停供校验返回信息，stopHeatingPreCallBackInfo="+gson.toJson(stopHeatingPreCallBackInfo));

return stopHeatingPreCallBackInfo;

}catch (Exception e){

e.printStackTrace();

SysLogger.info("停供校验异常，message="+e.getMessage());

ChargeStopHeatingModel.StopHeatingPreCallBackInfo info = new ChargeStopHeatingModel().new StopHeatingPreCallBackInfo();

stopHeatingPreCallBackInfo.setLimit(1);

stopHeatingPreCallBackInfo.setReason("服务器繁忙，请稍后再试");

return stopHeatingPreCallBackInfo;

}

1. 收费7.0业务代码的编写过程中，一定要保持良好的代码规范，如换行、缩进和括号等。好的编码风格不是对自己负责，也是对整个团队负责，不能想怎么写就怎么写，如果后期其他同事看了你的代码，心情也不会太好。

反例：收费7.0业务代码中少量的代码没有遵循优秀的代码格式（红色字体标注）。

// 拼接填充

for (Map<String, String> dataMap : datalist) {

String str = "";

for (Map<String, String> fieldMap : fieldlist) {

if (!StringUtil.isNullString(str)) str += ",";

str += "\"" + String.valueOf(fieldMap.get("param\_name")) + "\":\"" + StringUtil.null2String(String.valueOf(dataMap.get(String.valueOf(fieldMap.get("param\_name"))))) + "\"";

}

vhtml += " mygrid" + list\_no + ".addRow({" + str + "},false);\n";

}

正例：收费7.0业务中的优秀的代码格式，具体可以参考编程规约中的代码格式。

for (int i = 0; i < listDatas.size(); i++) {

JsonObject area = listDatas.get(i).getAsJsonObject();

String areaId = area.get("areaid").getAsString();

areaIdList.add(areaId);

houseIdList.add(area.get("houseid").getAsString());

}

1. 专有名词解释
2. POJO（Plain Ordinary Java Object）: 在本手册中，POJO 专指只有 setter / getter/ toString 的简单类，包括 DO/DTO/BO/VO 等。
3. GAV（GroupId、ArtifactctId、Version）: Maven 坐标，是用来唯一标识 jar 包。
4. OOP（Object Oriented Programming）: 本手册泛指类、对象的编程处理方式。
5. ORM（Object Relation Mapping）: 对象关系映射，对象领域模型与底层数据之间的转换， 本文泛指 iBATIS, mybatis 等框架。
6. NPE（java.lang.NullPointerException）: 空指针异常。
7. SOA（Service-Oriented Architecture）: 面向服务架构，它可以根据需求通过网络对松散耦合的粗粒度应用组件进行分布式部署、组合和使用，有利于提升组件可重用性，可维护性。
8. IDE（Integrated Development Environment）: 用于提供程序开发环境的应用程序，一般包括代码编辑器、编译器、调试器和图形用户界面等工具，本《手册》泛指 IntelliJ IDEA 和 eclipse。
9. OOM（Out Of Memory）: 源于 java.lang.OutOfMemoryError，当 JVM 没有足够的内存来为对象分配空间并且垃圾回收器也无法回收空间时，系统出现的严重状况。
10. 一方库: 本工程内部子项目模块依赖的库（jar 包）。
11. 二方库: 公司内部发布到中央仓库，可供公司内部其它应用依赖的库（jar 包）。
12. 三方库: 公司之外的开源库（jar 包）。