# 数据治理平台开发规范

版本：1.0

日期：2021-12-03

# 目录


1、 编程规约 ..... 4

(一)命名风格 ..... 4

(二)常量定义 ..... 9

(三)代码格式 ..... 12

(四)OOP 规约 ..... 19

(五)集合处理 ..... 27

(六)控制语句 ..... 37

2、 异常日志 ..... 41

(一)异常处理 ..... 41

(二)日志规约 ..... 51

3、 单元测试 ..... 53

4、 安全规约 ..... 56

5、 MySQL 数据库 ..... 57

(一)建表规约 ..... 57

(二)SQL 语句 ..... 59

(三)ORM 映射 ..... 59

# 1、编程规约

## （一） 命名风格

### 1.代码中的命名均不能以下划线或美元符号开始，也不能以下划线或美元符号结束

说明：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.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
```

### 3.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

说明：在部门内部应该规划好包名的范围，防止产生冲突。部门内部产品使用部门的名称加上模块名称。产品线的产品使用产品的名称加上模块的名称。即使有多个单词也不使用下划线或大小写分隔

正例：

```
应用工具类包名为 com.hginfo.ai.util 、类名为 MessageUtils(此规则参考 spring 的框架结构 )

com.hginfo.产品名.模块名称

com.hginfo.部门名称.项目名称
```

反例：

```
com.vip.java_tool, com.vip.javaTool
```

## 4.类名使用 UpperCamelCase 风格，遵从驼峰形式说

说明：

- 类名通常是名词或名词短语，接口名称有时可能是形容词或形容词短语。
- 现在还没有特定的规则或行之有效的约定来命名注解类型。
- 测试类的命名以它要测试的类的名称开始，以 Test 结束。
- Tcp, Xml 等缩写也遵循驼峰形式，但以下情形例外：DO / BO / DTO / VO / AO /PO / UID 等

正例：

```
MarcoPolo / UserDO / XmlService / TcpUdpDeal / TaPromotion

UserId / XmlService / TcpUdpDeal / UserVO

HashTest 或 HashIntegrationTest
```

反例：

```
macroPolo / UserDo / XMLService / TCPUDPDeal / TAPromotion

UserID / XMLService / TCPUDPDeal / UserVo
```

## 5.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

说明：

正例：

```
localValue / getHttpMessage() / inputUserIdprivate void calculateRate();public void
addNewOrder();
```

## 6.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

说明：例外：当一个 static final 字段不是一个真正常量，比如不是基本类型时，不需要使用大写命名

正例：

```
MAX _ STOCK _ COUNTpublic final static int MAX_VALUE = 1000;public final static String
DEFAULT_START_DATE = "2001-12-08";private static final Logger logger =
Logger.getLogger(MyClass.class);
```

反例：

```
MAX _ COUNTpublic final static int max_value = 1000;
```

## 7.枚举类名以 Enum 结尾；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

说明：

正例：

```
DealStatusEnum，AbstractView，BaseView，TimeoutException，UserServiceTest
```

## 8.类型与中括号紧挨相连来表示数组

说明：

正例：

```
定义整形数组 int[] arrayDemo;public int[] getIndex(){    ....}
```

反例：

```
在 main 参数中，使用 String args[]来定义 public int getIndex()[] {
    ....}
```

## 9.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

说明：

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
```

## 10.杜绝完全不规范的缩写，避免望文不知义

说明：

反例：

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

## 11.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

说明：

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
```

## 12.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.接口类中的方法和属性不要加任何修饰符号 （public 也不要加 ） ，保持代码的简洁性，并加上有效的 Javadoc 注释。尽量不要在接口里定义变量，如果一定要定义变量，肯定是与接口方法相关，并且是整个应用的基础常量

说明：JDK 8 中接口允许有默认实现，那么这个 default 方法，是对所有实现类都有价值的默认实现

正例：

```
接口方法签名 void commit();

接口基础常量 String COMPANY = " alibaba " ;
```

反例：

```
接口方法定义 public abstract void f();
```

## 14.接口和实现类的命名有两套规则

说明：

- 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。
- 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：

```
CacheServiceImpl 实现 CacheService 接口

AbstractTranslator 实现 Translatable 接口
```

## 15.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 _ REASONpublic enum SeasonEnum {

    SPRING,SUMMER,FALL,WINTER;}
```

## 16. 避免成员变量，方法参数，局部变量的重名复写，引起混淆

说明：类的私有成员变量名，不与父类的成员变量重名，方法的参数名/局部变量名，不与类的成员变量重名(getter/setter 例外)

反例：

```
public class A {
  int foo;}
public class B extends A {
  int foo; //WRONG
  int bar;

  public void hello(int bar) { //WRONG
    int foo = 0; //WRONG
  }

  public void setBar(int bar) { //OK
```

```
    this.bar = bar;
  }}
```

## 17.各层命名规约

说明：Service / DAO 层方法命名规约

- 获取单个对象的方法用 get 做前缀。
- 获取多个对象的方法用 list 做前缀，复数形式结尾如：listObjects。
- 获取统计值的方法用 count 做前缀。
- 插入的方法用 save/insert 做前缀。
- 删除的方法用 remove/delete 做前缀。
- 修改的方法用 update 做前缀。

领域模型命名规约

- 数据对象： xxx ， xxx 即为数据表名。
- 数据传输对象： xxxDTO ， xxx 为业务领域相关的名称。
- 展示对象： xxxVO ， xxx 一般为网页名称。
- POJO 是 DO / DTO / BO / VO 的统称，禁止命名成 xxxPOJO 。

### （二） 常量定义

## 1.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

说明：

- 使用常量，将项目的常量都抽取到一个 class 文件中，
- 使用 format 方法
- 使用枚举类型

正例：

```
public class ConstantFile{

    public static final String name = "常量";//方法 1

    public void myMethod(){

        String.format("常量,%s,%s",args1,args2);//方法 2
```

```
    }}
public enum  ConstantFile {
  USA("美国",1),CHINA("中国",2);
  private String name;
  private int index;
  private ConstantFile(String name,int index){
    this.name=name;
    this.index=index;
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public int getIndex() {
    return index;
  }

  public void setIndex(int index) {
    this.index = index;
  }}
```

反例：

```
String key = " Id # taobao _" + tradeId;
cache . put(key , value);
String value = new String("常量");
```

## 2.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 = 2 l; 写的是数字的 21，还是 Long 型的 2

正例：

```
Long a = 2 L;
```

反例：

```
Long a = 2 l;
```

# 3.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.常量的复用层次

说明：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

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

此规约中提到的“一方库”、“二方库”可能会引发一些疑惑。

- 一方库是指同一工程内的模块打包成的库.
- 二方库是指公司内的公共库

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。
```

# 5.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;
    }}
```

## （一） 代码格式

# 1.程序块要采用缩进风格编写，缩进的空格数为 4 个，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.
- 对齐只使用空格键，不使用 TAB 键，以免用不同的编辑器阅读程序时，因 TAB 键所设置的空格数目不同而造成程序布局不整齐。

正例：

```
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
    }}
```

## 2.大括号的使用约定

说明：如果是大括号内为空，则简洁地写成{}即可，不需要换行。而左大括号前需要空格。

如果是非空代码块则：

- 左大括号前不换行。
- 左大括号后换行。
- 右大括号前换行。
- 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。

正例：

```
for (...) {
    ... // program code}
for (...) {
    ... // program code}
if (...) {
    ... // program code}
void example_fun( void ) {
    ... // program code}
```

反例：

```
if (...)
    {
    ... // program code
    }
void example_fun( void )
    {
    ... // program code
    }for (...) {
    ... // program code}
```

## 3.较长的语句、表达式或参数（\>120 字符）要分成多行书写

说明：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：

- 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。
- 运算符与下文一起换行。
- 方法调用的点符号与下文一起换行。
- 方法调用中的多个参数需要换行时，在逗号后进行。
- 在括号前不要换行，见反例。

正例：

```
if (filename != null
    && new File(logPath + filename).length() < LogConfig.getFileSize()) {
    ... // program code}
public static LogIterator read(String logType, Date startTime, Date endTime,
                               int logLevel, String userName, int bufferNum)


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);
```

## 4.不允许把多个短语句写在一行中，即一行只写一条语句

说明：

正例：

```
LogFilename now = null;
LogFilename that = null;
```

反例：

```
LogFilename now = null;        LogFilename that = null;
```

## 5.if, for, do, while, case, switch, default 等语句自占一行，且 if, for, do, while 等语句的执行语句无论多少都要加括号{}

说明：

正例：

```
if(writeToFile) {
    writeFileThread.interrupt();}
```

反例：

```
if(writeToFile) {
    writeFileThread.interrupt();}
```

## 6. 通过空行进行逻辑分段,但没有必要插入多个空行进行隔开

说明：

- 一段代码也是一段文章，需要合理的分段而不是一口气读到尾。
- 不同组的变量之间，不同业务逻辑的代码行之间，插入一个空行，起逻辑分段的作用。
- 而联系紧密的变量之间、语句之间，则尽量不要插入空行。
- 任何情形，没有必要插入多个空行进行隔开。

正例：

```
if(log.getLevel() < LogConfig.getRecordLevel()) {return;}


LogWriter writer;int index;
```

反例：

```
if(log.getLevel() < LogConfig.getRecordLevel()) {
    return;}
LogWriter writer;
```

## 7.在两个以上的关键字、变量、常量进行对等操作时，它们之间的操作符之前、之后或者前后要加空格；进行非对等操作时，如果是关系密切的立即操作符（如.），后不应加空格

说明：

- 采用这种松散方式编写代码的目的是使代码更加清晰。
- 由于留空格所产生的清晰性是相对的，所以，在已经非常清晰的语句中没有必要再留空格，如果语句已足够清晰则括号内侧(即左括号后面和右括号前面)不需要加空格，多重括号间不必加空格，因为在 Java 语言中括号已经是最清晰的标志了。
- 在长语句中，如果需要加的空格非常多，那么应该保持整体清晰，而在局部不加空格。
- 给操作符留空格时不要连续留两个以上空格。

正例：

```
1.逗号、分号只在后面加空格 int a, b, c;

2.比较操作符, 赋值操作符"="、 "+=", 算术操作符"+"、"%", 逻辑操作符"&&"、"&", 位域操作符
"<<"、"^"等双目操作符的前后加空格 if (current_time >= MAX_TIME_VALUE) { ... }

a = b + c;

a *= 2;

a = b ^ 2;

3."!"、"~"、"++"、"--"、"&"（地址运算符）等单目操作符前后不加空格

flag = !isEmpty; // 非操作"!"与内容之间

i++;              // "++","--"与内容之间

4."."前后不加空格

p.id = pid;     // "."前后不加空格

5.if、for、while、switch 等与后面的括号间应加空格，使 if 等关键字更为突出、明显 if (a >= b &&
c > d) { ....

 }
```

## 8.类属性和类方法不要交叉放置，不同存取范围的属性或者方法也尽量不要交叉放置，类内方法定义的顺序，不要"总是在类的最后添加新方法"

说明：一个类就是一篇文章，想象一个阅读者的存在，合理安排方法的布局。

- 顺序依次是：构造函数 \> (公有方法\>保护方法\>私有方法) \> getter/setter 方法。
- 如果公有方法可以分成几组，私有方法也紧跟公有方法的分组。
- 当一个类有多个构造方法，或者多个同名的重载方法，这些方法应该放置在一起。其中参数较多的方法在后面。
- 作为调用者的方法，尽量放在被调用的方法前面。

正例：

```
格式：
类定义{
类的公有属性定义
类的保护属性定义
类的私有属性定义
构造函数
类的公有方法定义
类的保护方法定义
类的私有方法定义
 getter/setter 方法}public Foo(int a) {...}public Foo(int a, String b) {...}public void foo(int a) {...}public void foo(int a, String b) {...}public void foo() {
    bar();}public void bar() {...}
```

# 9.IDE 的 text file encoding 设置为 UTF-8; IDE 中文件的换行符使用 Unix 格式，不要使用 Windows 格式

# 10. 用小括号来限定运算优先级

说明：我们没有理由假设读者能记住整个 Java 运算符优先级表。除非作者和 Reviewer 都认为去掉小括号也不会使代码被误解，甚至更易于阅读。

正例：

```
if ((a == b) && (c == d))
三目运算符 foo != null ? foo : "" 不需要加括号
```

# 11.小括号的使用约定

说明：

- 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；
- if / for / while / switch / do 等保留字与括号之间都必须加空格

反例：

```
if (空格 a == b 空格)
```

## 12.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。

正例：

```
i --;

i = a + b;if (a && b)
```

反例：

```
i=a+b;
```

## 13.注释的双斜线与注释内容之间有且仅有一个空格。

说明：

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();
```

## 14.方法参数在定义和传入时，多个参数逗号后边必须加空格

说明：

正例：

```
下例中实参的 args1，后边必须要有一个空格。method(args1, args2, args3);
```

## 15.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
```

## 16.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

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

正例：

```
int one = 1;long two = 2L;float three = 3F;

StringBuffer sb = new StringBuffer();
```

反例：

```
Int          one    = 1;long         two    = 2L;float        three = 3F;

StringBuffer sb     = new StringBuffer();
```

## （一） OOP 规约

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

说明：直观的好处是少一些代码,少创建一些对象。

正例：

```
StringUtils.isEmpty(str);
```

反例：

```
StringUtils stringUtils = new StringUtils();

stringUtils.isEmpty(str);
```

## 2. 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

正例：

```
@Override 注解本身其实不起任何作用,但使用它可以做到两件很有用的事:

告诉代码的阅读者,这个方法是在覆盖父类的方法

编写程序时,如果方法名、参数列表、异常等定义错误导致不能正确覆盖父类方法时,编译器会提示错误
```

## 3. 相同参数类型，相同业务含义，才可以使用 Java 的可变参数，避免使用 Object

说明：可变参数必须放置在参数列表的最后。（ 提倡同学们尽量不用可变参数编程 ） 可变参数是 J2SE 1.5 的新特性,用起来很方便,但在工程中使用不当会严重影响代码的可读性和可维护性。 所以对于庞大的、不好管控的开发团队来说,直接限制使用或许也是一种方法

正例：

```
public List<User> listUsers(String type, Long... ids) {...}
```

## 4. 外部正在调用或者二方库依赖的接口，不允许修改方法签名，避免对接口调用方产生影响。接口过时必须加@Deprecated 注解，并清晰地说明采用的新接口或者新服务是什么

说明：避免修改方法签名是因为你不清楚都有谁在什么场景下调用此方法,如果贸然修改方法签名,可能会引发问题

## 5. 不能使用过时的类或方法

说明：java.net.URLDecoder 中的方法 decode(String encodeStr)这个方法已经过时， 应该使用双参数 decode(String source, String encode) 。 接口提供方既然明确是过时接口，那么有义务同时提供新的接口； 作为调用方来说，有义务去考证过时方法的新实现是什么。

## 6. Object 的 equals 方法容易抛空指针异常，应使用常量或确定有值的对象来调用 equals

说明：推荐使用 java . util . Objects # equals（JDK 7 引入的工具类 ）， 通过常量来调用 equals 方法这一条大家应该都清楚。 然而如果要比较的两个都是对象的话,似乎把哪一个放在前面都可能发生空指针，对此,使用 J2SE 1.7 中引入的 Objects 类是最佳的: if(Objects.equals(a, b)) {...} 这样的话,无论如何都不会产生空指针异常,需要注意的是，如果比较的两个对象都为 null 的话,Objects.equals 方法会返回 true

正例：

```
" test " .equals(object)
```

反例：

```
object.equals( " test " )
```

## 7.所有的相同类型的包装类对象之间值的比较，全部使用 equals 方法比较

说明：对于 Integer var = ? 在-128 至 127 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

正例：

包装类是指 Byte/Boolean/Short/Integer/Long/Float/Double/Character 等对 Java 基本类型进行封装的在使用包装类时,很容易把使用基本类型时的习惯带入,引用原文的说法,这是一个大坑。

看下面的例子:

```
int a = 1000;

int b = 1000;

if(a == b) //true

Integer b1 = 1000;

if(a == b1) //true

Integer a1 = 1000;

if(a1 == b1) //false

Integer a2 = 10;

Integer b2 = 10;

if(a2 == b2) //true
```

对于不了解包装类机制的程序员来说 ,上面这段代码会造成不小的困惑。

简单解释一下:

当包装类对象和基本类型变量进行==判断时,Java 会把包装类对象转换成基本类型变量后进行比较,这和两个基本类型变量进行==操作是等价的当两个包装类对象进行==判断时,

Java 比较的是两个对象的内存地址是否一致,即判断这两个变量引用的对象是否是同一个,

所以上例中的 a1 == b1 结果是 false

那么为什么 a2 == b2 结果是 true 呢?

因为包装类 Integer 维护了一个对象缓冲池,池中有 256 个 Integer 对象,体现的是-128~127 之间的数值。

所以说 a2 和 b2 实际上都指向了缓冲池中的同一个对象,a2 == b2 自然就是 true 了

包装类 Byte/Short/Integer/Long/Character 中都有这样的缓冲池机制,

所以当我们对这些包装类进行比较时,使用 equals 方法是最稳妥的

# 8.关于基本数据类型与包装数据类型的使用

说明：

所有的 POJO 类属性必须使用包装数据类型

RPC 方法的返回值和参数必须使用包装数据类型

所有的局部变量使用基本数据类型

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。

总结：基本数据类型变量是不能赋值为 null 的,但包装类对象可以

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。

## 9.定义 DO / DTO / VO 等 POJO 类时，不要设定任何属性默认值

说明：

反例：

`POJO` 类的 `gmtCreate` 默认值为 `new Date()`， 但是这个属性在数据提取时并没有置入具体值，在更新其它字段时又附带更新了此字段，导致创建时间被修改成当前时间

## 10.序列化类新增属性时，请不要修改 serialVersionUID 字段，避免反序列失败；如果完全不兼容升级，避免反序列化混乱，那么请修改 serialVersionUID 值

说明：注意 serialVersionUID 不一致会抛出序列化运行时异常

正例：

在进行反序列化时,`JVM` 会判断待反序列化的字节流中的 `serialVersionUID` 和对应类的 `serialVersionUID` 是否一致,如果不一致,会产生 `InvalidCastException`。

## 11.构造方法里面禁止加入任何业务逻辑,如果有初始化逻辑,请放在 init 方法中

说明： 在你向构造方法里加入业务逻辑时,你是不可能确保这段业务逻辑能够覆盖今后所有的需求的,

万一哪一天出现"某场景下实例化此类时不可以执行这段业务逻辑"的情况,那就悲催了,

只能重载一个构造方法。

## 12.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。

## 13.禁止在 POJO 类中，同时存在对应属性 xxx 的 isXxx() 和 getXxx() 方法

说明：框架在调用属性 xxx 的提取方法时，并不能确定哪个方法一定是被优先调用到

## 14.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：

反例：

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

## 15.类内方法定义的顺序依次是：构造方法>公有方法或保护方法 > 私有方法 > getter / setter 方法

说明：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。

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。

## 16.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

说明：在 getter / setter 方法中，不要增加业务逻辑，会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;
    }}
```

## 17.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：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。

使用"+";拼接字符串的代码,在编译成字节码时会被编译器自动转换为使用 StringBuilder 的

append 方法拼接,所以说单行代码中使用"+"拼接字符串是没有问题的,

但如果这样的拼接频繁出现(不仅限于循环中),那么每次拼接时都会 new 一个新的 StringBuffer 对象,

对性能产生很大影响。

正例：

```
publicstaticvoidtestForStr(){
    StringBuilder stringBuilder = new StringBuilder("abc");
    for (int i=0; i<127; i++){
        stringBuilder.append(i);
    } }
```

反例：

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

# 18.final 可以声明类、成员变量、方法、以及本地变量

说明： 1） 不允许被继承的类，如：String 类。

2） 不允许修改引用的域对象。

3） 不允许被重写的方法，如： POJO 类的 setter 方法。

4） 不允许运行过程中重新赋值的局部变量。

5） 避免上下文重复使用一个变量，使用 final 描述可以强制重新定义一个变量，方便更好地进行重构。

final 修饰变量时对于性能是没有任何影响的。final 修饰编译时常量时,会带来些许的性能提升。

```
static int foo() {
    final int a = 2; // 声明常量 a
    final int b = 3; // 声明常量 b
    return a + b; // 常量表达式}
```

编译成字节码后则是:

iconst_5 // 常量折叠了,没有"访问局部变量" ireturn 而 final 修饰的方法,则会在编译期省略掉方法调用,直接把被调用的方法内容编译过去

然而,无论是哪种情况下 final 带来的性能提升都是很小的,而且如果使用的是比较新的 JVM,在经过 JIT 编译后的机器码也会屏蔽掉加不加 final 的性能区别。所以说 final 的主要价值还是在于其语义本身提供的不可变性检查特性,从提升性能的角度来制定 final 规约显得有点怪了。

# 19.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 浅拷贝,即仅拷贝对象中基本数据类型的成员属性,如果成员属性引用了其他对象，则拷贝出的对象副本中的这些成员属性引用的还是同一个对象。 深拷贝则是被拷贝对象的成员属性引用的其他对象也是拷贝出的副本

# 20.类成员与方法访问控制

说明：任何类、方法、参数、变量,严控访问范围。过宽泛的访问范围,不利于模块解耦。

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,必须考虑是否为 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。

正例：

编写代码不要只用 public，要考虑访问控制范围攻击者可以用意想不到的方式操纵 public 或 protected 的数据成员，所以需要将数据成员为 private，对外提供可控的包装方法访问数据成员

# 21.明确类的功能，精确（而非近似）地实现类的设计。一个类仅实现一组相近的功能。

说明：划分类的时候，应该尽量把逻辑处理、数据和显示分离，实现类功能的单一性

正例：

数据类不能包含数据处理的逻辑

通信类不能包含显示处理的逻辑

# 22.敏感类不允许复制

说明：包含私人的，机密或其他敏感数据的类是不允许被复制的

正例：

解决的方法有两种：

1. 类声明为 final

```
final class SensitiveClass {
    // ...}
```

2. Clone 方法抛出 CloneNotSupportedException 异常

```
class SensitiveClass {
    // ...
    public final SensitiveClass clone() throws CloneNotSupportedException {
        throw new CloneNotSupportedException();
    }}
```

# 23.比较类的正确做法

说明：如果由同一个类装载器装载，它们具有相同的完全限定名称，则它们是两个相同的类

正例：

```
// Determine whether object auth has required/expected class name if (auth.getClass()
== com.application.auth.DefaultAuthenticationHandler.class) {
   // ...}
```

反例：

```
// Determine whether object auth has required/expected class object if
(auth.getClass().getName().equals("com.application.auth.DefaultAuthenticationHandler"))
{
   // ...}
```

## 24.应明确规定对接口方法参数的合法性检查应由方法的调用者负责还是由接口方法本身负责，缺省是由方法调用者负责

说明：对于模块间接口方法的参数的合法性检查这一问题，往往有两个极端现象，

即：要么是调用者和被调用者对参数均不作合法性检查，

结果就遗漏了合法性检查这一必要的处理过程，造成问题隐患；

要么就是调用者和被调用者均对参数进行合法性检查，这种情况虽不会造成问题，

但产生了冗余代码，降低了效率

### (五)集合处理

## 1.关于 hashCode 和 equals 的处理规则

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
  - 只要重写 equals ，就必须重写 hashCode 。
  - 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。
  - 如果自定义对象作为 Map 的键，那么必须重写 hashCode 和 equals 。

正例：

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。

以上内容仅为本文档的试下载部分，为可阅读页数的一半内容。如要下载或阅读全文，请访问：

https://d.book118.com/635044013123011131