标题：软件编码规范—JAVA

编号：HW-GF-05

版本：1.0

页数：21

生效日期：2017-08-31



编制人：唐丽钟

批准人：刘晓玲

目录

1. 目的 3

2. 适用范围 3

3. 定义 3

4. 基本原则 3

5. 布局 4

5.1. 代码格式化 4

5.2. 类模块布局 4

6. 注释 5

6.1. 普通注释 5

6.2. Javadoc注释 6

6.3. Apidoc注释 7

7. 命名规则 8

7.1. 标识符 8

7.2. 包 9

7.3. 类和接口 9

7.4. 方法 10

7.5. 常量和变量 11

8. 声明和调用 12

8.1. 类和接口 12

8.2. 方法 13

8.3. 常量和变量 15

9. 表达式、语句和块 16

10. API使用 18

10.1. Java.lang的核心API 18

10.2. 集合处理 19

11. 版本修订历史记录 21

# 目的

统一编程风格，提高代码的可读性与编码效率，避免团队开发带来的混乱。

# 适用范围

本规范适用于公司所有运用Java作为开发语言的编码活动。

# 定义

规则 : 编程时必须遵守的约定。

建议 : 编程时需要考虑的约定。

# 基本原则

1. 首先是为人编写程序，其次才是计算机

这是软件开发的基本要点，软件的生命周期贯穿产品的开发、测试、生产、用户使用、版本升级和后期维护等长期过程，只有易读、易维护的软件代码才具有生命力。

1. 保持代码的简明清晰，避免过分追求编程技巧

简单是最美。保持代码的简单化是软件工程化的基本要求。不要过分追求技巧，否则会降低程序的可读性。

1. 所有的代码尽量遵循[Sun Java编程语言编码规范](http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html)。

编程时以该规范为准，规范没有规定的内容参考上面的标准。

1. 编程时首先达到正确性，其次考虑效率

编程首先考虑的是满足正确性、健壮性、可维护性、可移植性等质量因素，最后才考虑程序的效率和资源占用。

1. 尽可能重用、修正已经存在的代码

尽量选择可借用的代码，对其修改优化以达到自身要求，但要注意的是，对已经存在的代码进行修改，要做好影响分析，同时做好版本控制。

1. 尽量减少同样的错误出现的次数

事实上，我们无法做到完全消除错误，但通过不懈的努力，可以减少同样的错误出现的次数。

1. 使用工具辅助开发人员完成规约检查

开发团队可以使用同一的工具（IDE代码格式化功能，Eclipse SonarLint、FindBug插件等），实现自动化代码风格格式化和编码规范自动检测。

# 布局

程序布局的目的是显示出程序良好的逻辑结构，提高程序的准确性、连续性、可读性、可维护性。更重要的是，统一的程序布局和编程风格，有助于提高整个项目的开发质量，提高开发效率，降低开发成本。同时，对于普通程序员来说，养成良好的编程习惯有助于提高自己的编程水平，提高编程效率。因此，统一的、良好的程序布局和编程风格不仅仅是个人主观美学上的或是形式上的问题，而且会涉及到产品质量，涉及到个人编程能力的提高，必须要引起重视。

程序布局主要约定了Java源代码文件的缩进、对齐、空行、空格、断行、类模块布局等编码风格。

## 代码格式化

|  |
| --- |
| 【规则】缩进、括号、空格、空行、新行、换行、控制语句块、注释等代码格式的规则，使用IDE提供的代码格式化功能实现。 |

**说明：**项目组需要使用统一的格式化设置，以下是Java开发团队使用的Eclipse Java格式化设置模板。开发者在“Preferences 🡪Java🡪Code Style🡪Formatter”页面，使用“Import”功能导入该配置模板。之后对需要格式化的Java文件“Ctrl+Shift+F”即可进行代码格式化。该功能会对缩进、代码块括号（即{}）、语句/参数括号（即()）、空格、空行、新行、换行、控制语句块、注释等内容进行格式化。



Android开发人员请使用Intllij专用的格式化设置模板。



## 类模块布局

|  |
| --- |
| 【规则】Java类模块布局顺序是：package、import、类定义、常量、静态变量、属性、构造函数、公共方法、私有方法。 |

**说明：**可见范围越广的变量、方法尽可能放在类的前半部分，便于其他开发者阅读。相比类的私有内容，开发者更关心类的公开的内容。此规则亦适用于接口、内部类（排除接口、内部类不存在的模块）。

**正例：**

package com.huaweisoft.demo;

import java.util.Set;

public class Demo {

public static final int BAR = 0;

private Set<String> foos;

public Demo() {

super();

}

public void foo() {

// do sth;

}

private void bar() {

// do sth

}

}

# 注释

注释有助于理解代码，有效的注释是指在代码的功能、意图层次上进行注释，提供有用、额外的信息，而不是代码表面意义的简单重复。

Javadoc注释、Apidoc注释主要用于文档自动化生成，开发者也必须在开发过程中遵循本节相关内容，编写Javadoc、Apidoc注释。

## 普通注释

对于普通注释的格式，开发者可以自行根据注释的内容选择使用单行或多行注释。

|  |
| --- |
| 【规则】对于复杂的逻辑代码块、算法，一定要通过注释解释清楚原理、意图，便于维护。 |

**正例：**

// 音频aac的试验性功能支持，否则没有声音

sb.append(" -strict -2");

// 输出地址，覆盖已有视频

sb.append(" -y \"").append(outputFilePath).append("\"");

|  |
| --- |
| 【规则】对于局部变量等不需要javadoc注释的部分，优先使用名称自解释，无法做到自解释才使用注释说明。注释应该写在被注释代码的上一行，若注释内容较短，也可以写在被注释代码的行尾 |

**正例：**

private void foo() {

String userName;

// 身份证号（注释写在被注释行的上一行）

String id; // 身份证号（也可以写行尾）

}

## Javadoc注释

Javadoc是Sun公司提供的一项技术，它从程序[源代码](https://baike.baidu.com/item/%E6%BA%90%E4%BB%A3%E7%A0%81/3969)中抽取类、方法、成员等注释形成一个和源代码配套的API帮助文档。主流IDE也能解析Javadoc注释，直接为开发者提供API帮助。

|  |
| --- |
| 【规则】类、公共方法、公共变量必须写Javadoc注释，Javadoc的格式遵守IDE自动生成的模板。 |

**正例：**

/\*\*

\* 根据用户ID查询实时信息

\*

\* @param userId 用户id

\* @return 头盔实时状态信息

\*/

public HelmetStatusDto find(Integer userId);

**反例：**

/\*\*

\* @param userId

\* @return

\*/

public HelmetStatusDto find(Integer userId);

**说明：**Javadoc主要是帮助API使用者了解他们主要关注的公共类、方法、变量的使用。开发者在Eclipse环境下可以选中需要生成Javadoc的类、方法或变量，使用“Alt+Shift+J”快捷键生成Javadoc注释模板块。

|  |
| --- |
| 【建议】类、接口的Javadoc包含@Since注解，用于标记该类、接口首次加入的版本号。 |

**正例：**

/\*\*

\* 头盔实时状态业务层

\*

\* @author tanglz

\* @since 0.1

\*

\*/

public interface HelmetStatusService {

## Apidoc注释

Apidoc是用于生成restful接口的文档的第三方工具。跟Javadoc类似，Apidoc通过提取restful接口类中的相关注释标签而生成html文档。

|  |
| --- |
| 【规则】服务端restful接口层类的接口都必须添加apidoc注释，便于使用apidoc工具生成restful接口文档。 |

**正例：**



**说明：**Javadoc和Apidoc可以现在同一个注释块中，但是必须将Javadoc定义在Apidoc之前，Apidoc的具体语法可以参考[Apidoc官方文档](http://apidocjs.com/)。

|  |
| --- |
| 【规则】Apidoc可以使用apiUse和apiDefine对接口的复杂对象参数和返回值进行预定义和复用。 |

**正例：**



# 命名规则

## 标识符

Java语言中，标识符是用来给类、对象、方法、变量、接口和自定义数据类型命名的符号串，如无特别说明，本小节中的规则适用于类、对象、方法、变量、接口和自定义数据类型的命名。

|  |
| --- |
| 【规则】在遵循Java标准命名规则基础上，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。 |

**正例：** name / object

**反例：** \_name / \_\_name / $Object / name\_ / name$ / Object$

**说明：**没有特殊需要，最好只使用字母。在某些框架的代码中，可能会使用比较特别的命名方式，原因是框架可能会通过名字来处理一些算法，比如：一个context中存放了所有关于请求的参数，其中某些框架特定的参数使用的命名就是\_\_header而不是header，防止开发者使用header覆盖了框架使用的特定参数。

|  |
| --- |
| 【规则】不要使用一些无意义的变量前缀和后缀名。 |

**正例：**userId,userName / TRUE=true

**反例：** mUserId,mUserName / CONSTANT\_TRUE=true

|  |
| --- |
| 【规则】严禁使用拼音与英文混合的方式，更不允许直接使用中文的方式。 |

**正例：**getDiscount / huaweisoft

**反例：**getDazhe

**说明：** 人名、地名等特殊名词等可以使用中文。

|  |
| --- |
| 【规则】使用能够被大家认可的缩写，杜绝完全不规范的缩写，避免望文不知义。标识符的命名应当符合“min-length && max-information”原则。 |

**正例：**HelmetStatusUploadResource / MSG / addr

**反例：**HmtStatUploadResource / MESG / adr

|  |
| --- |
| 【规则】采用应用领域相关的术语来命名。 |

**正例：**ContactBook

**反例：**ContactInformation

**说明：** 在CRM系统中，客户联系信息簿叫contact book

## 包

|  |
| --- |
| 【规则】包名使用小写，点分隔符之间有且仅有一个自然语义的英语单词。包名使用单数形式。 |

**正例：**com.company.project.module.pkgname / org.springframework.beans / java.util

**反例：**com.company\_project.modulePkgname / org.springframeworkbeans

|  |
| --- |
| 【规则】公司项目的包名格式为com.huaweisoft.<项目>.<模块>.<层>。 |

**正例：**com.huaweisoft.ih.media.collector.service

**反例：**huaweisoft.ih.media.collector.service

## 类和接口

|  |
| --- |
| 【规则】类名、接口名使用UpperCamelCase风格，必须遵从驼峰形式。 |

**正例：**MarcoPolo / XmlService / TcpUdpDeal / TaPromotion

**反例：**macroPolo / XMLService / TCPUDPDeal / TAPromotion

|  |
| --- |
| 【规则】类名通常为名词的单数形式，但是工具类用名词的复数形式。 |

**正例：**String / StringUtils

**反例：**Strings / StringHelper

|  |
| --- |
| 【规则】抽象类命名格式以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。 |

**正例：**BizException / MyResourceTest / BaseResource

**反例：**BizExp / TestMyResource

|  |
| --- |
| 【规则】领域模型中的实体（Entity）命名格式为<实体名称>，数据传输/视图对象命名格式为<实体名称>[特殊用途]DTO。 |

**正例：**UserStatus / UserStatusDto / UserStatusEnquiryDto extends PagiableEnquiryDto

**反例：**UserStatusPojo / User\_Status

**说明：** 实体名称一般对应的是数据库的表名，例如UserStatus对应的表名为user\_status。

|  |
| --- |
| 【规则】服务端的业务相关接口命名格式为<业务实体><层的名称>，实现类命名格式为<业务实体><层的名称>Impl。 |

**正例：**UserService / UserServiceImpl

|  |
| --- |
| 【建议】工具类、功能类的接口（相对于服务端的接口来说的），取对应的形容词做接口名（通常是–able的形式）。实现类命名则为具体功能的名词。 |

**正例：**interface Collection<E> extends Iterable<E>

|  |
| --- |
| 【建议】如果使用到了设计模式，建议在类名中体现出具体模式。有利于阅读者快速理解架构设计思想 |

**正例：**public class OrderFactory; / public class LoginProxy;

**反例：**public class OrderHelper; / public class LoginUtils;

## 方法

|  |
| --- |
| 【规则】方法名使用lowerCamelCase风格，必须遵从驼峰形式。 |

**正例：**marcoPolo / xmlService

**反例：**MacroPolo / XmlService

|  |
| --- |
| 【建议】如果方法表示一个操作，因此方法名格式为<表示操作的动词>[操作对象][条件]。操作对象一般可以忽略，因为通常这个方法操作的对象就是方法所在的类。 |

**正例：**add / addAll / sort / deleteUserById

|  |
| --- |
| 【建议】服务端三层模型中的Resource/Service/DAO基本是对资源的操作，命名规则根据对资源的增删改查操作归纳为以下几种情况： |

|  |  |  |
| --- | --- | --- |
| 操作 | 命名规则 | 正例 |
| 获取单个对象 | 用get/find做前缀 | fineOne / getUserById |
| 获取多个对象 | 1、用list做前缀  2、或者使用get/find前缀并且复数的名词结尾 | listUsers / list  getUsersByName / getUsers |
| 获取统计值 | 用count做前缀 | countByName |
| 插入 | 用save或insert做前缀 | save |
| 删除 | 用remove或delete做前缀 | removeAll / deleteByName |
| 修改 | 用update做前缀 | update |

**说明：**Spring Data JPA的DAO层（Repository）有比上表更为详细的命名规则，请参考Repository的接口定义。

## 常量和变量

|  |
| --- |
| 【规则】变量名使用lowerCamelCase风格，必须遵从驼峰形式。 |

**正例：**marcoPolo / xmlService

**反例：**MacroPolo / XmlService

|  |
| --- |
| 【规则】常量命名全部大写，单词间用下划线隔开。 |

**正例：**MAX\_STOCK\_COUNT

**反例：**MAX\_COUNT

|  |
| --- |
| 【规则】集合数据类型的常量/变量名必须是名词的复数形式，非集合数据类型的常量/变量名则为名词的单数形式。 |

**正例：**int customerNumber; List<Integer> customerNumbers;

**反例：**List<Integer> customerNumberList;

|  |
| --- |
| 【规则】接口中的属性一定是被public static final修饰的（即使没有对属性加public static final限定词），因此命名必须遵循常量的命名规则。 |

**正例：**

public interface Demo {

int P\_1 = 0;

public static final int P\_2 = 0;

}

|  |
| --- |
| 【规则】枚举成员名称需要全大写，单词间用下划线隔开。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：**成员名称：SUCCESS / UNKOWN\_REASON。

|  |
| --- |
| 【规则】实体、DTO类中的任何布尔类型的变量，都不要加is，否则部分框架解析会引起序列化错误。 |

**正例：**boolean success

**反例：**boolean isSuccess

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

# 声明和调用

## 类和接口

|  |
| --- |
| 【规则】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。 |

**正例：**ThisClass.staticMethod() / ThisClass.staticProperty;

**反例：**this.staticMethod() / this.staticProperty

|  |
| --- |
| 【规则】类和接口应该按照功能和分层模型声明package语句，将同类功能的类和接口放在各自的包中。 |

**说明：**服务端可以按类的功能和分层模型分为以下几个包：Resource、Service、Dao、DTO、Model。开发者可以根据实际需要进行调整。

|  |
| --- |
| 【建议】从严控制类成员与方法访问控制。 |

|  |  |
| --- | --- |
| 常见应用场景 | 访问控制实现方式 |
| 不允许外部直接通过new来创建对象 | private构造函数 |
| 工具类 | 不允许有public或default构造方法 |
| 类的非static成员变量并且与子类共享 | protected |
| 类的非static成员变量并且仅在本类使用 | private |
| 类的static成员变量如果仅在本类使用 | private |
| 类的成员方法且只供类内部调用 | private |
| 类的成员方法且只对继承类公开 | protected |

**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。对越宽泛的访问对象进行修改/删除，需要做更多的影响分析，引起错误的风险也更大。

## 方法

|  |
| --- |
| 【规则】所有的覆写方法，必须加@Override注解。 |

**正例：**

@Override

public boolean equals(Object obj) {

return false;

}

**反例：**

public boolean equals(Object obj) {

return false;

}

**说明：**加Override注解可以起到判断方法是否正确覆盖的作用，也便于代码阅读。

|  |
| --- |
| 【规则】对过时的方法必须加@Deprecated注解，并且在javadoc中表明过时原因和替代方法，过时方法不能直接删除。调用方也不能在新写的代码中使用已经过时的方法。 |

**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接口是什么。

|  |
| --- |
| 【规则】对外暴露的接口签名，原则上不允许修改方法签名，避免对接口调用方产生影响。 |

**说明：**遵守面向对象设计原则的开闭原则。

|  |
| --- |
| 【规则】POJO类必须改写toString方法。可以使用IDE提供的toString代码生成功能。 |

**说明：**使用工具类source> generate toString时，如果继承了另一个POJO类，注意在前面加一下super.toString。 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。

|  |
| --- |
| 【规则】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，在构造函数调用init方法。 |

|  |
| --- |
| 【建议】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。 |

|  |
| --- |
| 【建议】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，增加排查问题难度。 |

**正例：**

public void setName(String name) {

this.name = name;

}

**反例：**

public void setName(String firstName) {

this.name = firstName+this.lastName;

}

|  |
| --- |
| 【建议】不使用可变参数编程, 使用数组代替。 |

**正例：**public User getUsers(Integer[] ids);

**反例：**public User getUsers(Integer... ids);

**说明：**当同时存在public User getUsers(Integer... ids); public User getUsers(Object ids); public User getUsers(Integer ids);这样的重载方法时。如果使用者对参数类型绑定理解不透彻，很容易理解错误，从而导致使用了非预期的方法。

## 常量和变量

|  |
| --- |
| 【规则】一行只声明一个变量/常量。 |

**正例：**

int size = 0;

int length = 0;

**反例：**int size=0, length=0;

|  |
| --- |
| 【规则】long或者Long初始赋值时，必须使用大写的L，不能是小写的l，小写容易跟数字1混淆，造成误解。 |

**正例：**long size = 1l;

**反例：**long size = 1L;

|  |
| --- |
| 【规则】中括号数组类型的一部分，数组定义如下：String[] args; |

**正例：**String[] args

**反例：**String args[] / String [] args / String []args

|  |
| --- |
| 【规则】定义Entity/DTO/VO等POJO类时，不要设定任何属性默认值。也不要使用基本数据类型的属性（基本数据类型有默认值，例如boolean默认为false，而Boolean默认值为null） |

**反例：**某业务的DO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

|  |
| --- |
| 【建议】不需要重新赋值的变量，包括类属性、局部变量。使用final关键字修饰。 |

**说明：**final可以提高程序执行效率。

|  |
| --- |
| 【建议】魔法值（即未经定义的常量）不要直接出现在代码中。 |

**正例：**String key = KEY\_PREFIX + tradeId； cache.put(key, value);

**反例：**String key = "id\_" + tradeId； cache.put(key, value);

|  |
| --- |
| 【建议】不要使用一个常量类维护所有常量，应该按常量功能进行归类，分开维护。 |

**说明：**例如，缓存相关的常量放在类：CacheConsts下；系统配置相关的常量放在类：ConfigConsts下。 如果全部都放在Constants类中，不利于理解，也不利于维护。

|  |
| --- |
| 【建议】如果变量值仅在一个范围内变化用Enum类。 |

**正例：**public Enum{ MONDAY(1), TUESDAY(2), WEDNESDAY(3), THURSDAY(4), FRIDAY(5), SATURDAY(6), SUNDAY(7);}

**说明：**如果还带有名称之外的延伸属性，必须使用Enum类，下面正例中的数字就是延伸信息，表示星期几。

# 表达式、语句和块

|  |
| --- |
| 【规则】 在表达式中使用括号，使表达式的运算顺序更清晰。 |

**正例：**if (((year % 4 == 0) && (year % 100 != 0)) || (year % 400 == 0))

**反例：**if (year % 4 == 0 && year % 100 != 0 || year % 400 == 0)

**说明：**由于将运算符的优先级与结合律熟记是比较困难的，为了防止产生歧义并提高可读性，即使不加括号时运算顺序不会改变，也应当用括号确定表达式的操作顺序。

|  |
| --- |
| 【规则】在switch语句中，每一个case分支必须使用break结尾，最后一个分支必须是default分支。 |

**正例：**

switch (count) {

case 0:

[处理语句]

break;

case 1:

[处理语句]

break;

default:

[处理语句]

break;

}

|  |
| --- |
| 【规则】如果if…else if...else判断次数过多，使用switch case代替。 |

**说明：**多个if else的效率低于switch case。

|  |
| --- |
| 【规则】如果if…else、for等语句块嵌套层次过多，将部分逻辑独立写成一个方法。 |

|  |
| --- |
| 【规则】不要随意使用try…catch代码块捕获的异常并且不做任何处理。如果当前方法不能处理该异常，就往外层抛出异常。 |

**说明：**捕获了异常而又不做处理会导致外层程序执行出错，而且不能获得正确的错误跟踪信息。

|  |
| --- |
| 【规则】一定要在finally块中进行资源回收，例如关闭数据库连接、关闭文件流等。JDK1.7之后，使用try-with-resources实现自动回收资源。 |

**正例：**

File file =new File("");

try( FileInputStream fis = new FileInputStream(file)){

fis.read();

}

# API使用

实际开发中通常大量使用由JDK或者第三方提供的API实现项目的业务需求。本章节选取了部分在Java编码中使用的频率很高但是容易被不正确使用的API，定义了使用规范，旨在提高代码的正确性、健壮性。

## Java.lang的核心API

|  |
| --- |
| 【规则】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）。

|  |
| --- |
| 【规则】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。所以使用equals方法进行判断。

|  |
| --- |
| 【规则】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);

|  |
| --- |
| 【规则】循环体内，字符串的联接方式，使用StringBuilder的append方法进行扩展。 |

**正例：**

StringBuilder str = new StringBuilder("start");

for(int i=0; i<100; i++){

str.append("hello");

}

**反例：**

String str = "start";

for(int i=0; i<100; i++){

str = str + "hello";

}

**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。

|  |
| --- |
| 【规则】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方法实现属性对象的拷贝。

## 集合处理

|  |
| --- |
| 【规则】Map/Set的key为自定义对象时，必须重写hashCode和equals 。 |

**说明：**Map/Set等类的操作是基于hashCode、equals被正确实现的基础上。

|  |
| --- |
| 【规则】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。

|  |
| --- |
| 【规则】在subList场景中，高度注意对原集合元素个数的修改，会导致子列表的遍历、增加、删除均产生ConcurrentModificationException 异常 。 |

|  |
| --- |
| 【规则】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() 。 |

**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。

**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。

|  |
| --- |
| 【规则】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，底层的数据仍是数组。 String[] str = new String[] { "a", "b" }; List list = Arrays.asList(str); 第一种情况：list.add("c"); 运行时异常。 第二种情况：str[0]= "gujin"; 那么list.get(0)也会随之修改。

|  |
| --- |
| 【规则】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：**

Iterator<String> it = a.iterator();

while(it.hasNext()){

String temp = it.next();

if(删除元素的条件){

it.remove();

}

}

**反例：**

List<String> a = new ArrayList<String>();

a.add("1");

a.add("2");

for (String temp : a) {

if("1".equals(temp)){

a.remove(temp);

}

}

|  |
| --- |
| 【规则】使用entrySet遍历Map类集合KV，而不是keySet方式进行遍历。 |

**正例：**

Set<Map.Entry<Integer,String>> entrys = map.entrySet();

for(Map.Entry<Integer,String> entry : entrys){

Integer key = entry.getKey();

String val = entry.getValue();

}

**反例：**

Set<Integer> keys = map.keySet();

for(Integer key : keys){

String val = map.get(key);

}

**说明：**keySet其实是遍历了2次，一次是map.keySet()将entrySet转为Iterator对象，另一次是map.get(key)从hashMap中取出key所对应的value（只需要遍历key的hashcode相同的节点）。而map.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。

# 版本修订历史记录

|  |  |  |  |
| --- | --- | --- | --- |
| 日期 | 作者 | 内容 | 版本 |
| 2017-8-20 | 唐丽钟 | 初稿 | 0.1 |
| 2017-8-21 | 唐丽钟 | 根据会议评审内容修改文件的逻辑 | 0.2 |
| 2017-8-23 | 唐丽钟 | 修改细节后，通过评审 | 1.0 |