Jarvis开发规范

Java代码规范

|  |  |  |
| --- | --- | --- |
| 文件状态：  [√] 初稿  [ ] 正式发布  [ ]正在修改 | 文件标识： | JARVIS-KFGF-CPKFGF.QZ.001 |
| 当前版本： | V0.1 |
| 作 者： | JARVIS |
| 完成日期： | 2023-01-01 |

**版 本 历 史**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 版本/状态 | 作 者 | 参与者 | 起止日期 | 备注 |
| V0.1 | 王涛 |  | 2023.01.01 | 初稿 |
|  |  |  |  |  |
|  |  |  |  |  |

目录

[1. 名词解释 1](#_Toc61020915)

[2. 编程规约 1](#_Toc61020916)

[2.1. 命名风格 1](#_Toc61020917)

[2.2. 常量定义 2](#_Toc61020918)

[2.3. 代码格式规约 3](#_Toc61020919)

[2.4. 编码规约 3](#_Toc61020920)

[2.5. 集合规约 4](#_Toc61020921)

[2.6. 并发处理 5](#_Toc61020922)

[2.7. 注释规约 7](#_Toc61020923)

[3. 异常日志 7](#_Toc61020924)

[3.1. 异常处理 7](#_Toc61020925)

[3.2. 日志规约 9](#_Toc61020926)

# 名词解释

1. 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。
2. GAV（GroupId、ArtifactctId、Version）: Maven 坐标，是用来唯一标识 jar 包。
3. OOP（Object Oriented Programming）: 本手册泛指类、对象的编程处理方式。
4. 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。
5. NPE（java.lang.NullPointerException）: 空指针异常。
6. 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。
7. 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，主要指 IntelliJ IDEA 和eclipse。
8. 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。
9. 一方库：本工程内部子项目模块依赖的库（jar 包）。
10. 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）。
11. 三方库：公司之外的开源库（jar 包）。

# 编程规约

## 命名风格

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：int a 的随意命名方式。

1. 【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

1. 【强制】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";

接口默认方法 default void defaultMethod(){ …… }

反例：接口方法定义 public abstract void f();

1. 接口和实现类的命名有两套规则：
   1. 【推荐】 如果是形容能力的接口名称，取对应的形容词做接口名（通常是–able的形式）。

正例：AbstractTranslator实现 Translatable。

## 常量定义

1. 【推荐】如果变量值仅在一个范围内变化用Enum类。如果还带有名称之外的延伸属性，必须使用Enum类。

正例：

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

1. 【推荐】常量都定义在类里面，如果常量值为String字符串，尽可能使常量名称与值相同或仅风格不同。

正例：

public static final String YES = "yes";

public static final String MAX\_COUNT = "maxCount";

## 代码格式规约

1. 【推荐】对于Java8以上的Lamda表达式，应当每个表达式一行显示，以便于查看和调试。
2. 【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。
3. 【推荐】推荐尽量少用else， if-else的方式可以改写成卫语句，或者状态模式来实现。

正例：

if (param == 1) {

return param;

}

if (param == 2) {

return param – 1;

}

return param + 1;

反例：

if (param == 1) {

//…

} else if (parma == 2) {

//…

} else {

//…

}

1. 【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。
2. 【推荐】避免采用取反逻辑运算符。

## 编码规约

1. 【推荐】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。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. 【推荐】不允许使用JDK中的sun包。

反例：

sun包：sun.misc.BASE64Decoder

1. 关于基本数据类型与包装数据类型的使用标准如下：
   1. 【推荐】所有的局部变量使用基本数据类型。
2. 【推荐】尽可能不要使用Unicode转义字符，例如：\u107a。
3. 【推荐】定义 DO/DTO/VO 等 POJO 类时，不要设定任何属性默认值。
4. 【推荐】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。
5. 【推荐】构造方法里面不要加入任何业务逻辑，如果有初始化逻辑，请放在init方法中。
6. 【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。
7. 【推荐】循环体内，字符串的联接方式，使用StringBuilder的append方法进行扩展。

反例：

String str = "start";

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

str = str + "hello";

}

1. 【推荐】当一个类有多个构造方法时应该按顺序放置在一起，便于阅读。
2. 【参考】当一个类有或者多个同名方法（重载），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。
3. 【参考】 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。

## 集合规约

1. 【推荐】集合构造的时候需要定义初始化大小。

正例： List<Xxx> list = new ArrayList<Xxx>(8);

1. 【推荐】使用entrySet遍历Map类集合KV，而不是keySet方式进行遍历。

正例：

for (Iterator<Map.Entry<String, User>> iterator = map.entrySet().iterator(); iterator.hasNext();) {

Map.Entry<String, User> entry = iterator.next();

String key = entry.getKey();

User value = entry.getValue();

}

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 |

## 并发处理

1. 【推荐】资金相关的金融敏感信息，使用悲观锁策略。

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

1. 【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：

public class LazyInitDemo {

private Helper helper = null;

public Helper getHelper() {

if (helper == null) synchronized(this) {

if (helper == null)

helper = new Helper();

}

return helper;

}

// other methods and fields...

}

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。

说明：如果是 count++操作，使用如下类实现：AtomicInteger count = new AtomicInteger(); count.addAndGet(1); 如果是 JDK8，推荐使用 LongAdder 对象，比 AtomicLong 性能更好（减少乐观锁的重试次数）。

1. 【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。
2. 【参考】ThreadLocal 对象使用 static 修饰，ThreadLocal 无法解决共享对象的更新问题。

说明：这个变量是针对一个线程内所有操作共享的，所以设置为静态变量，所有此类实例共享此静态变量，也就是说在类第一次被使用时装载，只分配一块存储空间，所有此类的对象(只要是这个线程内定义的)都可以操控这个变量。

## 注释规约

1. 【推荐】对于已有代码的修改不要采用注释的方式，直接删除即可。对于部分却有必要保留的代码，在注释时必须同时添加说明原因。
2. 【推荐】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。
3. 【推荐】所有的public属性都要添加Javadoc注释，说明该属性的作用。
4. 【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。
5. 【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。

# 异常日志

## 异常处理

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。调用方需要进行null判断防止NPE问题。
2. 【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。

* 1. 数据库的查询结果可能为 null。
  2. 集合里的元素即使 isNotEmpty，取出的数据元素也可能为 null。
  3. 远程调用返回对象时，一律要求进行空指针判断，防止 NPE。
  4. 对于 Session 中获取的数据，建议进行 NPE 检查，避免空指针。
  5. 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。

1. 【推荐】定义时区分unchecked / checked 异常，避免直接使用RuntimeException抛出，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。
2. 【参考】在代码中使用“抛异常”还是“返回错误码”，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、“错误码”、“错误简短信息”。
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，甚至是共用模块。
4. 【参考】常见异常分类处理方式：
   1. 数据库异常：对于采用Spring等容器的事务管理器的情况，此类异常不允许捕捉处理，由事务管理统一进行捕捉后回滚，也可以捕捉后记录日志后继续抛出。对于非容器的事务管理的情况，需要捕捉异常后手动回滚。
   2. 登录异常：认证框架中的此类异常，不进行处理。如有必要，可以自定义同类异常以便于前台显示。
   3. 前端异常：Service层抛出的异常或Controller层产生的异常，统一通过Controller层框架提供的异常处理方式进行展现，不允许前台页面直接显示未经处理的异常信息。
   4. Ajax的异常处理：Ajax方法发生的异常可以分为2种：连接异常和业务异常。连接异常是指页面连接后台过程中产生的异常（连接超时、服务器连接不上等）。此类异常在Ajax的error方法中进行统一捕获处理。业务异常是指连接服务正常，但在后台业务处理过程中产生的异常。此类异常建议仍然在success方法中处理，通过标识或处理结果进行异常展示。

## 日志规约

1. 【推荐】对trace/debug/info级别的日志输出，必须使用条件输出形式（如：isDebugEnabled()）或者使用占位符的方式。

正例：（条件）

if (logger.isDebugEnabled()) {

logger.debug("Processing trade with id: " + id + " symbol: " + symbol);

}

正例：（占位符）

logger.debug("Processing trade with id: {} symbol : {} ", id, symbol);

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。注意日志输出的级别，error级别只记录系统逻辑出错、异常等重要的错误信息。如非必要，请不要在此场景打出error级别。
2. 【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。