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： | 王涛 |
| 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](#_Toc61021039)

[2 编程规约 1](#_Toc61021040)

[2.1. 命名风格 1](#_Toc61021043)

[2.2. 常量定义 4](#_Toc61021044)

[2.3. 代码格式规约 5](#_Toc61021045)

[2.4. 编码规约 8](#_Toc61021046)

[2.5. 集合规约 11](#_Toc61021047)

[2.6. 并发处理 13](#_Toc61021048)

[2.7. 注释规约 17](#_Toc61021049)

[3 异常日志 18](#_Toc61021050)

[3.1. 异常处理 18](#_Toc61021052)

[3.2. 日志规约 19](#_Toc61021053)

# 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. 【强制】 代码中的命名均不能以下划线或美元符等特殊字符开始，也不能以下划线或美元符等特殊字符结束。

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

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。

反例：zhiChengCode【职称代码】/getPinyingByName()

1. 【强制】类名使用UpperCamelCase风格，必须遵从驼峰形式，但以下情形例外：（领域模型的相关命名）DO / BO / DTO / VO等以及特殊的缩写单词，例如CRUD等。

正例：JavaServerlessPlatform / UserDO / XmlService / TcpUdpDeal / TaPromotion

反例：javaserverlessplatform / 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 / CACHE\_EXPIRED\_TIME

反例：MAX\_COUNT / EXPIRED\_TIME

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：boolean isSuccess;

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，也可以是拼音。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】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。

反例：

**public** **class** ConfusingName {

**public** **int** age;

// 非 setter/getter 的参数名称，不允许与本类成员变量同名

**public** **void** getData(String alibaba) {

**if**(condition) {

**final** **int** money = 531;

// ...

}

**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 = 615;

// ...

}

}

}

**class** Son **extends** ConfusingName {

// 不允许与父类的成员变量名称相同

**public** **int** age;

}

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

正例：DateEnum{SUMDAY,MONDAY,……}

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. 【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. 【强制】各层命名规约：
2. DAO层方法命名规约

由于DAO多是ORM框架生成，因此采用一般的ORM框架单实体规则（JPA、Mybaits）。

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 query/find做前缀，一般查询使用“queryBy/ findBy + 属性名”方式。
3. 获取统计值的方法用 count 做前缀。
4. 插入的方法用 save/insert，批量加All。
5. 删除的方法用 remove/delete，批量加All。
6. 修改的方法用 save/update，批量加All。
7. Service层方法命名规约

参考DAO操作加类型的命名方式：例如saveUser，getUsers等为前缀：

1. 获取单个对象的方法用 get XXX做前缀。
2. 获取多个对象的方法用 query XXX/findXXX做前缀。
3. 获取统计值的方法用 countXXX 做前缀。
4. 插入的方法用 saveXXX /insertXXX做前缀。
5. 删除的方法用 removeXXX/deleteXXX 做前缀。
6. 修改的方法用 updateXXX 做前缀。

## 常量定义

1. 【强制】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 a = 2L;

反例：long a = 2l;

## 代码格式规约

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：
   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；表示终止右大括号后必须换行。

正例：

try {

//……

} catch (Exception ex) {}

//……

反例：

try

{

//……

}

catch (Exception ex)

{

}

//……

1. 【强制】 左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。

正例：request.setAttribute(“name”, “abc”);

反例：request.setAttribute( “name”, “abc” );

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。
3. 【强制】缩进采用4个空格，禁止使用tab字符。使用Eclipse或IDEA时，请设置1个tab为4个空格。
4. 【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，避免使用下面的形式：if (condition) statements;
5. 【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。

正例：

switch (value) {

case 1:

//......

break;

case 2:

//......

break;

default:

break;

}

1. 【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。
2. 【强制】单行字符数限不超过 120 个，超出需要换行时，遵循如下原则：
   1. 第二行相对一缩进 4个空格，从第三行开始不再继续缩进。
   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. 在多个参数超长，逗号后进行换行。
   5. 在括号前不要换行。

正例：

StringBuffer sb = new StringBuffer();

//超过120个字符的情况下，换行缩进4个空格，并且方法前的点符号一起换行

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. 【强制】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。
3. 【强制】不允许空代码行，即独立的“;”，也不允许一个多个子句，即一行中有多个“;”。

反例：int len = 1; int max = 10; int min = 0;

1. 【强制】注释的双斜线与注释内容之间有且仅有一个空格。
2. 【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。
3. 【强制】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）。

## 编码规约

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。

public class Status {

public static final String MAX\_VALUE = “maxValue”;

public static void getStatus(){

//……

}

}

正例：Status.MAX\_VALUE

反例：new Status().getStatus();

1. 【强制】所有的覆写方法，必须加@Override注解。
2. 【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。

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

1. 【强制】不能使用过时的类或方法。

反例：

过时：java.net.URLDecoder.decode(String encodeStr)

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。推荐使用java.util.Objects#equals （JDK7引入的工具类）

正例： "test".equals(object);

反例： object.equals("test");

1. 【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 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. 【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，具体原理参考《码出高效》。

反例：

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");

}

1. 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");

}

1. 【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。

1. 【强制】为了防止精度损失，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);

1. 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。
2. 【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。
3. 【强制】一个java文件只能定义一个顶级类（内部类不算顶级类）。
4. 【强制】不允许在import中使用“\*”通配符，也不允许import未使用的类。
5. 【强制】声明前缀必须按照以下顺序：public、protected、private、abstract、static、final、transient、volatile、synchronized、native、strictfp。

正例： public static final String ABC = "ABC";

反例： abstract public synchronized String doSomething();

1. 【强制】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。

## 集合规约

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. 如果自定义对象做为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。

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。
2. 【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。
3. 【强制】Collections 类返回的对象，如：emptyList()/singletonList()等都是 immutablelist，不可对其进行添加或者删除元素的操作。
4. 【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。
5. 【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()。

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<String>(2);

list.add("guan");

list.add("bao");

String[] array = new String[list.size()];

array = list.toArray(array);

1. 【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。

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。

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】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。
2. 【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。

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);

}

}

正例：

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

while(it.hasNext()){

String temp = it.next();

if(删除元素的条件){

it.remove();

}

}

1. 【强制】 在JDK7版本以上，Comparator要满足自反性，传递性，对称性，不然Arrays.sort，Collections.sort会报IllegalArgumentException异常。

说明：

* 1. 自反性：x，y的比较结果和y，x的比较结果相反。
  2. 传递性：x>y,y>z,则x>z。
  3. 对称性：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;

}

}

## 并发处理

1. 【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。
2. 【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：自定义线程工厂，并且根据外部特征进行分组，比如机房信息。

public class UserThreadFactory implements ThreadFactory {

private final String namePrefix;

private final AtomicInteger nextId = new AtomicInteger(1);

* 定义线程组名称，在 jstack 问题排查时，非常有帮助

UserThreadFactory(String whatFeaturOfGroup) {

namePrefix = "From UserThreadFactory's " + whatFeaturOfGroup + "-Worker-";

}

@Override

public Thread newThread(Runnable task) {

String name = namePrefix + nextId.getAndIncrement(); Thread thread = new Thread(null, task, name, 0, false);

System.out.println(thread.getName());

return thread;

}

}

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：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。

2） CachedThreadPool：

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】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】在使用阻塞等待获取锁的方式中，必须在 try 代码块之外，并且在加锁方法与 try 代码块之间没有任何可能抛出异常的方法调用，避免加锁成功后，在 finally 中无法解锁。

说明一：如果在 lock 方法与 try 代码块之间的方法调用抛出异常，那么无法解锁，造成其它线程无法成功获取锁。

说明二：如果 lock 方法在 try 代码块之内，可能由于其它方法抛出异常，导致在 finally 代码块中， unlock 对未加锁的对象解锁，它会调用 AQS 的 tryRelease 方法（取决于具体实现类），抛出 IllegalMonitorStateException 异常。

说明三：在 Lock 对象的 lock 方法实现中可能抛出 unchecked 异常，产生的后果与说明二相同。

正例：

Lock lock = new XxxLock();

// ...

lock.lock();

try {

doSomething();

doOthers();

} finally {

lock.unlock();

}

反例：

Lock lock = new XxxLock();

// ...

try {

// 如果此处抛出异常，则直接执行 finally 代码块

doSomething();

// 无论加锁是否成功，finally 代码块都会执行

lock.lock();

doOthers();

} finally {

lock.unlock();

}

1. 【强制】在使用尝试机制来获取锁的方式中，进入业务代码块之前，必须先判断当前线程是否持有锁。锁的释放规则与锁的阻塞等待方式相同。

说明：Lock 对象的 unlock 方法在执行时，它会调用 AQS 的 tryRelease 方法（取决于具体实现类），如果当前线程不持有锁，则抛出 IllegalMonitorStateException 异常。

正例：

Lock lock = new XxxLock();

// ...

boolean isLocked = lock.tryLock();

if (isLocked) {

try {

doSomething();

doOthers();

} finally {

lock.unlock();

}

}

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，如果在处理定时任务时使用 ScheduledExecutorService 则没有这个问题。

## 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。（除了带有@Override的继承方法）

正例：

/\*\*

\* 用户管理

\* @author huz

\*/

@Service("demoUserService")

public class UserService {

……

}

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。

正例：

/\*\*

\* 登录信息认证

\* @param token 验证信息字符串

\* @return 认证信息对象

\* @throws AuthenticationException

\* @see org.apache.shiro.realm.AuthenticatingRealm#doGetAuthenticationInfo(org.apache.shiro.authc.AuthenticationToken)

\*/

@Override

protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token) throws AuthenticationException{

……

}

1. 【强制】所有的类都必须添加创建者信息和创建日期。
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】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。

# 异常日志



## 异常处理

1. 【强制】不要捕获Java类库中定义的继承自RuntimeException的运行时异常类，如：IndexOutOfBoundsException / NullPointerException，这类异常由程序员预检查来规避，保证程序健壮性。

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，因为异常的处理效率比条件分支低。
2. 【强制】对大段代码进行try-catch，这是不负责任的表现。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。
3. 【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。
4. 【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。
5. 【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。
6. 【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。
7. 【强制】在调用 RPC、二方包、或动态生成类的相关方法时，捕捉异常必须使用 Throwable类来进行拦截。

## 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。
2. 【强制】所有日志文件至少保存 15 天，最多尽可能不要超过3个月，单个日志文件大小不超过10M，以便于查看。对于部分监控类日志最少不要低于3个月。
3. 【强制】在日志输出时，字符串变量之间的拼接使用占位符的方式。

说明：因为 String 字符串的拼接会使用 StringBuilder 的 append()方式，有一定的性能损耗。使用占位符仅是替换动作，可以有效提升性能。

正例：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，务必在对应配置文件（logback.xml或log4j.xml）中设置additivity=false。

正例：<logger name="com.cesgroup.demo.config" additivity="false">