《唯品会Java开发手册》1.0.2版

1. 概述

《阿里巴巴Java开发手册》，是首个对外公布的企业级Java开发手册，对整个业界都有重要的意义。

我们结合唯品会的内部经验，参考《Clean Code》、《Effective Java》等重磅资料，增补了一些条目，也做了些精简。

感谢阿里授权我们定制和再发布。

2. 规范正文

注意： 如需全文pdf版，请下载源码，在docs/standard/目录运行merge.sh生成，阅读时的章节跳转使用pdf阅读器的左侧书签。

3. 规范落地

规则落地主要依靠代码格式化模版与Sonar代码规则检查。

其中Sonar规则不尽如人意的地方，我们进行了定制。

- Eclipse/Intellij 格式模板
- Sonar 规则修改示例

4. 参考资料

- 《Clean Code》
- 《Effective Java 2nd》
- 《SEI CERT Oracle Coding Standard for Java》(在线版)
- Sonar代码检查规则

5. 定制记录

- 《唯品会Java开发手册》 - 与阿里手册的比较文学I

(一) 命名规约

Rule 1. 【强制】禁止拼音缩写，避免阅读者费劲猜测；尽量不用拼音，除非中国式业务词汇没有通用易懂的英文对应。

```
禁止： DZ[打折] / getPFByName() [评分]

尽量避免：Dazhe / DaZhePrice
```

Rule 2. 【强制】禁止使用非标准的英文缩写

```
反例： AbstractClass 缩写成 AbsClass；condition 缩写成 condi。
```

Rule 3. 【强制】禁用其他编程语言风格的前缀和后缀

在其它编程语言中使用的特殊前缀或后缀，如`_name`，`name_`，`mName`，`i_name`，在Java中都不建议使用。

Rule 4. 【推荐】命名的好坏，在于其"模糊度"

1）如果上下文很清晰，局部变量可以使用`list`这种简略命名， 否则应该使用`userList`这种更清晰的命名。

2）禁止`a1, a2, a3`这种带编号的没诚意的命名方式。

3）方法的参数名叫`bookList` ，方法里的局部变量名叫`theBookList`也是很没诚意。

4）如果一个应用里同时存在`Account、AccountInfo、AccountData`类，或者一个类里同时有`getAccountInfo()、getAccountData()`，`save()、 store()`的函数，阅读者将非常困惑。

5）`callerId`与`calleeId`， `mydearfriendswitha`与`mydearfriendswithb`这种拼写极度接近，考验阅读者眼力的。

Rule 5. 【推荐】包名全部小写。点分隔符之间尽量只有一个英语单词，即使有多个单词也不使用下划线或大小写分隔

```
正例： com.vip.javatool

反例： com.vip.java_tool, com.vip.javaTool
```

- Sonar-120:Package names should comply with a naming convention

---

Rule 6. 【强制】类名与接口名使用UpperCamelCase风格，遵从驼峰形式

Tcp, Xml等缩写也遵循驼峰形式，可约定例外如：DTO/ VO等。

```
正例：UserId / XmlService / TcpUdpDeal / UserVO

反例：UserID / XMLService / TCPUDPDeal / UserVo
```

- Sonar-101:Class names should comply with a naming convention
- Sonar-114:Interface names should comply with a naming convention

---

Rule 7. 【强制】方法名、参数名、成员变量、局部变量使用lowerCamelCase风格，遵从驼峰形式

```
正例： localValue / getHttpMessage();
```

- Sonar-100:Method names should comply with a naming convention
- Sonar-116:Field names should comply with a naming convention
- Sonar-117:Local variable and method parameter names should comply with a naming convention

---

Rule 8. 【强制】常量命名全大写，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
```

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。

```
private static final Logger logger = Logger.getLogger(MyClass.class);
```

例外：枚举常量推荐全大写，但如果历史原因未遵循也是允许的，所以我们修改了Sonar的规则。

- Sonar-115:Constant names should comply with a naming convention
- Sonar-308:Static non-final field names should comply with a naming convention

---

Rule 9. 【推荐】如果使用到了通用的设计模式，在类名中体现，有利于阅读者快速理解设计思想

```
正例：OrderFactory， LoginProxy ，ResourceObserver
```

Rule 10. 【推荐】枚举类名以Enum结尾; 抽象类使用Abstract或Base开头；异常类使用Exception结尾；测试类以它要测试的类名开始，以Test结尾

```
正例：DealStatusEnum， AbstractView，BaseView， TimeoutException，UserServiceTest
```

- Sonar-2166:Classes named like "Exception" should extend "Exception" or a subclass
- Sonar-3577:Test classes should comply with a naming convention

---

Rule 11. 【推荐】实现类尽量用Impl的后缀与接口关联，除了形容能力的接口

> 正例：CacheServiceImpl 实现 CacheService接口。
>
> 正例: Foo 实现 Translatable接口。

Rule 12. 【强制】POJO类中布尔类型的变量名，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的成员变量，它的GET方法也是isSuccess()，部分框架在反射解析的时候，“以为”对应的成员变量名称是success，导致出错。

Rule 13. 【强制】避免成员变量，方法参数，局部变量的重名复写，引起混淆

- 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)

下面错误的地方，Java在编译时很坑人的都是合法的，但给阅读者带来极大的障碍。

```
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;
  }
}
```

- Sonar-2387: Child class fields should not shadow parent class fields
- Sonar: Local variables should not shadow class fields

(二) 格式规约

Rule 1. 【强制】使用项目组统一的代码格式模板，基于IDE自动的格式化

1）IDE的默认代码格式模板，能简化绝大部分关于格式规范(如空格，括号)的描述。

2）统一的模板，并在接手旧项目先进行一次全面格式化，可以避免， 不同开发者之间，因为格式不统一产生代码合并冲突，或者代码变更日志中因为格式不同引起的变更，掩盖了真正的逻辑变更。

3）设定项目组统一的行宽，建议120。

4）设定项目组统一的缩进方式(Tab或二空格，四空格均可)，基于IDE自动转换。

- VIP代码格式化模板

Rule 2. 【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

Rule 3. 【推荐】 用小括号来限定运算优先级

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。

```
if ((a == b) && (c == d))
```

- Sonar-1068:Limited dependence should be placed on operator precedence rules in expressions，我们修改了三目运算符 `foo!=null:foo:""` 不需要加括号。

Rule 4. 【推荐】类内方法定义的顺序，不要"总是在类的最后添加新方法"
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。

1）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。

2）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。

```
public Foo(int a) {...}
public Foo(int a, String b) {...}

public void foo(int a) {...}
public void foo(int a, String b) {...}
```

3）作为调用者的方法，尽量放在被调用的方法前面。

```
public void foo() {
    bar();
}

public void bar() {...}
```

Rule 5. 【推荐】通过空行进行逻辑分段

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。

```
int width;
int height;

String name;
```

Rule 6.【推荐】避免IDE格式化

对于一些特殊场景（如使用大量的字符串拼接成一段文字，或者想把大量的枚举值排成一列），为了避免IDE自动格式化，土办法是把注释符号//加在每一行的末尾，但这有视觉的干扰，可以使用@formatter:off和@formatter:on来包装这段代码，让IDE跳过它。

```
// @formatter:off
...
// @formatter:on
```

(三) 注释规约

Rule 1.【推荐】基本的注释要求

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。

代码将被大量后续维护，注释如果对阅读者有帮助，不要吝啬在注释上花费的时间。(但也综合参见规则2，3)

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。

除了特别清晰的类，都尽量编写类级别注释，说明类的目的和使用方法。

除了特别清晰的方法，对外提供的公有方法，抽象类的方法，同样尽量清晰的描述：期待的输入，对应的输出，错误的处理和返回码，以及可能抛出的异常。

Rule 2. 【推荐】通过更清晰的代码来避免注释

在编写注释前，考虑是否可以通过更好的命名，更清晰的代码结构，更好的函数和变量的抽取，让代码不言自明，此时不需要额外的注释。

Rule 3. 【推荐】删除空注释，无意义注释

《Clean Code》建议，如果没有想说的，不要留着IDE自动生成的，空的@param，@return，@throws 标记，让代码更简洁。

反例：方法名为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，不需要任何额外的注释。

```
/**
 * put elephant into fridge.
 *
 * @param elephant
 * @param fridge
 * @return
 */
public void put(Elephant elephant, Fridge fridge);
```

Rule 4.【推荐】避免创建人，创建日期，及更新日志的注释

代码后续还会有多人多次维护，而创建人可能会离职，让我们相信源码版本控制系统对更新记录能做得更好。

Rule 5.【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

Rule 6.【强制】类、类的公有成员、方法的注释必须使用Javadoc规范，使用/ xxx */格式，不得使用`//xxx`方式**

正确的JavaDoc格式可以在IDE中，查看调用方法时，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。

Rule 7.【推荐】JavaDoc中不要为了HTML格式化而大量使用HTML标签和转义字符

如果为了Html版JavaDoc的显示，大量使用`<p\>` `<pre\>`这样的html标签，以及`&lt` `&quot`这样的html转义字符，严重影响了直接阅读代码时的直观性，而直接阅读代码的几率其实比看Html版的JavaDoc大得多。
另外IDE对JavaDoc的格式化也要求`<p>`之类的标签来换行，可以配置让IDE不对JavaDoc的格式化。

Rule 8.【推荐】注释不要为了英文而英文

如果没有国际化要求，中文能表达得更清晰时还是用中文。

Rule 9.【推荐】TODO标记，清晰说明代办事项和处理人

清晰描述待修改的事项，保证过几个月后仍然能够清楚要做什么修改。

如果近期会处理的事项，写明处理人。如果远期的，写明提出人。

通过IDE和Sonar的标记扫描，经常清理此类标记，线上故障经常来源于这些标记但未处理的代码。

正例：

```
//TODO:calvin use xxx to replace yyy.
```

反例：

```
//TODO: refactor it
```

- Sonar: Track uses of "TODO" tags
- Sonar: Track uses of "FIXME" tags

Rule 10. 【推荐】合理处理注释掉的代码

如果后续会恢复此段代码，在目标代码上方用`///`说明注释动机，而不是简单的注释掉代码。
如果很大概率不再使用，则直接删除（版本管理工具保存了历史代码）。

- Sonar: Sections of code should not be "commented out"

(四) 方法设计

Rule 1. 【推荐】方法的长度度量
方法尽量不要超过100行，或其他团队共同商定的行数。

另外，方法长度超过8000个字节码时，将不会被JIT编译成二进制码。

- Sonar-107: Methods should not have too many lines，默认值改为100
- Facebook-Contrib:Performance - This method is too long to be compiled by the JIT

Rule 2. 【推荐】方法的语句在同一个抽象层级上
反例：一个方法里，前20行代码在进行很复杂的基本价格计算，然后调用一个折扣计算函数，再调用一个赠品计算函数。

此时可将前20行也封装成一个价格计算函数，使整个方法在同一抽象层级上。

Rule 3. 【推荐】为了帮助阅读及方法内联，将小概率发生的异常处理及其他极小概率进入的代码路径，封装成独立的方法

```
if(seldomHappenCase) {
  hanldMethod();
}

try {
  ...
} catch(SeldomHappenException e) {
  handleException();
}
```

Rule 4. 【推荐】尽量减少重复的代码，抽取方法

超过5行以上重复的代码，都可以考虑抽取公用的方法。

Rule 5. 【推荐】方法参数最好不超过3个，最多不超过7个

1）如果多个参数同属于一个对象，直接传递对象。

例外: 你不希望依赖整个对象，传播了类之间的依赖性。

2）将多个参数合并为一个新创建的逻辑对象。

例外: 多个参数之间毫无逻辑关联。

3）将函数拆分成多个函数，让每个函数所需的参数减少。

- Sonar-107: Methods should not have too many parameters

Rule 6.【推荐】下列情形，需要进行参数校验

1）调用频次低的方法。

2）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，代价更大。

3）需要极高稳定性和可用性的方法。

4）对外提供的开放接口，不管是RPC/HTTP/公共类库的API接口。

如果使用Apache Validate 或 Guava Precondition进行校验，并附加错误提示信息时，注意不要每次校验都做一次字符串拼接。

```
//WRONG
Validate.isTrue(length > 2, "length is "+keys.length+", less than 2", length);
//RIGHT
Validate.isTrue(length > 2, "length is %d, less than 2", length);
```

Rule 7.【推荐】下列情形，不需要进行参数校验

1）极有可能被循环调用的方法。

2）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。

比如，一般DAO层与Service层都在同一个应用中，所以DAO层的参数校验，可以省略。

3）被声明成private，或其他只会被自己代码所调用的方法，如果能够确定在调用方已经做过检查，或者肯定不会有问题则可省略。

即使忽略检查，也尽量在方法说明里注明参数的要求，比如vjkit中的@NotNull，@Nullable标识。

Rule 8.【推荐】禁用assert做参数校验

assert断言仅用于测试环境调试，无需在生产环境时进行的校验。因为它需要增加-ea启动参数才会被执行。而且校验失败会抛出一个AssertionError(属于Error，需要捕获Throwable）

因此在生产环境进行的校验，需要使用Apache Commons Lang的Validate或Guava的Precondition。

Rule 9.【推荐】返回值可以为Null，可以考虑使用JDK8的Optional类

不强制返回空集合，或者空对象。但需要添加注释充分说明什么情况下会返回null值。

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。

JDK8的Optional类的使用这里不展开。

Rule 10.【推荐】返回值可以为内部数组和集合

如果觉得被外部修改的可能性不大，或没有影响时，不强制在返回前包裹成Immutable集合，或进行数组克隆。

Rule 11.【推荐】不能使用有继承关系的参数类型来重载方法

因为方法重载的参数类型是根据编译时表面类型匹配的，不根据运行时的实际类型匹配。

```
class A {
  void hello(List list);
  void hello(ArrayList arrayList);
}

List arrayList = new ArrayList();

// 下句调用的是hello(List list)，因为arrayList的定义类型是List
a.hello(arrayList);
```

Rule 12.【强制】正被外部调用的接口，不允许修改方法签名，避免对接口的调用方产生影响
只能新增新接口，并对已过时接口加@Deprecated注解，并清晰地说明新接口是什么。

Rule 13.【推荐】不使用`@Deprecated`的类或方法

接口提供方既然明确是过时接口并提供新接口，那么作为调用方来说，有义务去考证过时方法的新实现是什么。

比如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)。

Rule 14.【推荐】不使用不稳定方法，如com.sun.*包下的类，底层类库中internal包下的类
`com.sun.*`，`sun.*`包下的类，或者底层类库中名称为internal的包下的类，都是不对外暴露的，可随时被改变的不稳定类。

- Sonar-1191: Classes from "sun.*" packages should not be used

---

(五) 类设计

Rule 1.【推荐】类成员与方法的可见性最小化

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方法，或者一个public的成员变量，删除一下，不得手心冒点汗吗？

例外：为了单元测试，有时也可能将访问范围扩大，此时需要加上JavaDoc说明或vjkit中的`@VisibleForTesting`注解。

Rule 2.【推荐】 减少类之间的依赖

比如如果A类只依赖B类的某个属性，在构造函数和方法参数中，只传入该属性。让阅读者知道，A类只依赖了B类的这个属性，而不依赖其他属性，也不会调用B类的任何方法。

```
a.foo(b);     //WRONG

a.foo(b.bar); //RIGHT
```

Rule 3.【推荐】 定义变量与方法参数时，尽量使用接口而不是具体类

使用接口可以保持一定的灵活性，也能向读者更清晰的表达你的需求：变量和参数只是要求有一个Map，而不是特定要求一个HashMap。
例外：如果变量和参数要求某种特殊类型的特性，则需要清晰定义该参数类型，同样是为了向读者表达你的需求。

Rule 4. 【推荐】类的长度度量
类尽量不要超过300行，或其他团队共同商定的行数。

对过大的类进行分拆时，可考虑其内聚性，即类的属性与类的方法的关联程度，如果有些属性没有被大部分的方法使用，其内聚性是低的。

Rule 5.【推荐】 构造函数如果有很多参数，且有多种参数组合时，建议使用Builder模式

```
Executor executor = new ThreadPoolBuilder().coreThread(10).queueLenth(100).build();
```

即使仍然使用构造函数，也建议使用chain constructor模式，逐层加入默认值传递调用，仅在参数最多的构造函数里实现构造逻辑。

```
public A(){
  A(DEFAULT_TIMEOUT);
}

public A(int timeout) {
  ...
}
```

Rule 6.【推荐】构造函数要简单，尤其是存在继承关系的时候

可以将复杂逻辑，尤其是业务逻辑，抽取到独立函数，如init()，start()，让使用者显式调用。

```
Foo foo = new Foo();
foo.init();
```

Rule 7.【强制】所有的子类覆写方法，必须加`@Override`注解

比如有时候子类的覆写方法的拼写有误，或方法签名有误，导致没能真正覆写，加`@Override`可以准确判断是否覆写成功。

而且，如果在父类中对方法签名进行了修改，子类会马上编译报错。

另外，也能提醒阅读者这是个覆写方法。
最后，建议在IDE的Save Action中配置自动添加`@Override`注解，如果无意间错误同名覆写了父类方法也能被发现。

- Sonar-1161: "@Override" should be used on overriding and implementing methods

---

Rule 8.【强制】静态方法不能被子类覆写。

因为它只会根据表面类型来决定调用的方法。

```
Base base = new Children();

// 下句实际调用的是父类的静态方法，虽然对象实例是子类的。
base.staticMethod();
```

Rule 9.静态方法访问的原则

9.1【推荐】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，直接用类名来访问即可

目的是向读者更清晰传达调用的是静态方法。可在IDE的Save Action中配置自动转换。

```
int i = objectA.staticMethod(); // WRONG

int i = ClassA.staticMethod(); // RIGHT
```

- Sonar-2209: "static" members should be accessed statically
- Sonar-2440: Classes with only "static" methods should not be instantiated

9.2【推荐】除测试用例，不要static import 静态方法

静态导入后忽略掉的类名，给阅读者造成障碍。

例外：测试环境中的assert语句，大家都太熟悉了。

- Sonar-3030: Classes should not have too many "static" imports 但IDEA经常自动转换static import，所以暂不作为规则。

9.3【推荐】尽量避免在非静态方法中修改静态成员变量的值

```
// WRONG
public void foo() {
  ClassA.staticFiled = 1;
}
```

- Sonar-2696: Instance methods should not write to "static" fields
- Sonar-3010: Static fields should not be updated in constructors

---

Rule 10.【推荐】 内部类的定义原则

当一个类与另一个类关联非常紧密，处于从属的关系，特别是只有该类会访问它时，可定义成私有内部类以提高封装性。

另外，内部类也常用作回调函数类，在JDK8下建议写成Lambda。
内部类分匿名内部类，内部类，静态内部类三种。
1) 匿名内部类 与 内部类，按需使用：

在性能上没有区别；当内部类会被多个地方调用，或匿名内部类的长度太长，已影响对调用它的方法的阅读时，定义有名字的内部类。

2) 静态内部类 与 内部类，优先使用静态内部类：

1. 非静态内部类持有外部类的引用，能访问外类的实例方法与属性。构造时多传入一个引用对性能没有太大影响，更关键的是向阅读者传递自己的意图，内部类会否访问外部类。
2. 非静态内部类里不能定义static的属性与方法。

- Sonar-2694: Inner classes which do not reference their owning classes should be "static"
- Sonar-1604: Anonymous inner classes containing only one method should become lambdas

---

Rule 11.【推荐】使用getter/setter方法，还是直接public成员变量的原则。

除非因为特殊原因方法内联失败，否则使用getter方法与直接访问成员变量的性能是一样的。

使用getter/setter，好处是可以进一步的处理：

1. 通过隐藏setter方法使得成员变量只读
2. 增加简单的校验逻辑
3. 增加简单的值处理，值类型转换等

建议通过IDE生成getter/setter。
但getter/seter中不应有复杂的业务处理，建议另外封装函数，并且不要以getXX/setXX命名。

如果是内部类，以及无逻辑的POJO/VO类，使用getter/setter除了让一些纯OO论者感觉舒服，没有任何的好处，建议直接使用public成员变量。

例外：有些序列化框架只能从getter/setter反射，不能直接反射public成员变量。

Rule 12.【强制】POJO类必须覆写toString方法。

便于记录日志，排查问题时调用POJO的toString方法打印其属性值。否则默认的Object.toString()只打印`类名@数字`的无效信息。

Rule 13. hashCode和equals方法的处理，遵循如下规则:

13.1【强制】只要重写equals，就必须重写hashCode。 而且选取相同的属性进行运算。

13.2【推荐】只选取真正能决定对象是否一致的属性，而不是所有属性，可以改善性能。

13.3【推荐】对不可变对象，可以缓存hashCode值改善性能（比如String就是例子）。

13.4【强制】类的属性增加时，及时重新生成toString，hashCode和equals方法。

- Sonar-1206: "equals(Object obj)" and "hashCode()" should be overridden in pairs

---

Rule 14.【强制】使用IDE生成toString，hashCode和equals方法。

使用IDE生成而不是手写，能保证toString有统一的格式，equals和hashCode则避免不正确的Null值处理。

子类生成toString() 时，还需要勾选父类的属性。

Rule 15.【强制】Object的equals方法容易抛空指针异常，应使用常量或确定非空的对象来调用equals

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

```
"test".equals(object);  //RIGHT

Objects.equals(object, "test"); //RIGHT
```

- Sonar-1132: Strings literals should be placed on the left side when checking for equality

---

Rule 16.【强制】除了保持兼容性的情况，总是移除无用属性、方法与参数

特别是private的属性、方法、内部类，private方法上的参数，一旦无用立刻移除。信任代码版本管理系统。

- Sonar-3985: Unused "private" classes should be removed
- Sonar-1068: Unused "private" fields should be removed
- Sonar: Unused "private" methods should be removed
- Sonar-1481: Unused local variables should be removed
- Sonar-1172: Unused method parameters should be removed Sonar-VJ版只对private方法的无用参数告警。

Rule 17.【推荐】final关键字与性能无关，仅用于下列不可修改的场景

1）定义类及方法时，类不可继承，方法不可覆写；

2）定义基本类型的函数参数和变量，不可重新赋值；

3）定义对象型的函数参数和变量，仅表示变量所指向的对象不可修改，而对象自身的属性是可以修改的。

Rule 18.【推荐】得墨忒耳法则，不要和陌生人说话

以下调用，一是导致了对A对象的内部结构(B,C)的紧耦合，二是连串的调用很容易产生NPE，因此链式调用尽量不要过长。

```
obj.getA().getB().getC().hello();
```

(六) 控制语句

Rule 1. 【强制】if, else, for, do, while语句必须使用大括号，即使只有单条语句

曾经试过合并代码时，因为没加括号，单条语句合并成两条语句后，仍然认为只有单条语句，另一条语句在循环外执行。

其他增加调试语句等情况也经常引起同样错误。
可在IDE的Save Action中配置自动添加。

```
if (a == b) {
  ...
}
```

例外：一般由IDE生成的equals()函数

- Sonar-121: Control structures should use curly braces Sonar-VJ版豁免了equals()函数

Rule 2.【推荐】少用if-else方式，多用哨兵语句式以减少嵌套层次

```
if (condition) {
  ...
  return obj;
}

// 接着写else的业务逻辑代码;
```

- Facebook-Contrib: Style - Method buries logic to the right (indented) more than it needs to be

---

Rule 3.【推荐】限定方法的嵌套层次

所有if/else/for/while/try的嵌套，当层次过多时，将引起巨大的阅读障碍，因此一般推荐嵌套层次不超过4。

通过抽取方法，或哨兵语句（见Rule 2）来减少嵌套。

```
public void applyDriverLicense() {
  if (isTooYoung()) {
    System.out.println("You are too young to apply driver license.");
    return;
  }

  if (isTooOld()) {
    System.out.println("You are too old to apply driver license.");
    return;
```

```
  }

  System.out.println("You've applied the driver license successfully.");
  return;
}
```

- Sonar-134: Control flow statements "if", "for", "while", "switch" and "try" should not be nested too deeply，增大为4

Rule 4.【推荐】布尔表达式中的布尔运算符(&&,||)的个数不超过4个，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

```
//WRONG
if ((file.open(fileName, "w") != null) && (...) || (...)|| (...)) {
  ...
}

//RIGHT
boolean existed = (file.open(fileName, "w") != null) && (...) || (...);
if (existed || (...)) {
  ...
}
```

- Sonar-1067: Expressions should not be too complex，增大为4

Rule 5.【推荐】简单逻辑，善用三元运算符，减少if-else语句的编写

```
s != null ? s : "";
```

Rule 6.【推荐】减少使用取反的逻辑

不使用取反的逻辑，有利于快速理解。且大部分情况，取反逻辑存在对应的正向逻辑写法。

```
//WRONG
if (!(x >= 268) { ... }

//RIGHT
if (x < 268) { ... }
```

- Sonar-1940: Boolean checks should not be inverted

Rule 7.【推荐】表达式中，能造成短路概率较大的逻辑尽量放前面，使得后面的判断可以免于执行

```
if (maybeTrue() || maybeFalse()) { ... }

if (maybeFalse() && maybeTrue()) { ... }
```

Rule 8.【强制】switch的规则

1）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；
2）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。

```
String animal = "tomcat";

switch (animal) {
case "cat":
  System.out.println("It's a cat.");
  break;
case "lion": // 执行到tiger
case "tiger":
  System.out.println("It's a beast.");
  break;
default:
  // 什么都不做，也要有default
  break;
}
```

- Sonar: "switch" statements should end with "default" clauses

---

Rule 9.【推荐】循环体中的语句要考量性能，操作尽量移至循环体外处理

1）不必要的耗时较大的对象构造；

2）不必要的try-catch（除非出错时需要循环下去）。

Rule 10.【推荐】能用while循环实现的代码，就不用do-while循环

while语句能在循环开始的时候就看到循环条件，便于帮助理解循环内的代码；

do-while语句要在循环最后才看到循环条件，不利于代码维护，代码逻辑容易出错。

(七) 基本类型与字符串

Rule 1. 原子数据类型(int等)与包装类型(Integer等)的使用原则

1.1 【推荐】需要序列化的POJO类属性使用包装数据类型

1.2 【推荐】RPC方法的返回值和参数使用包装数据类型

1.3 【推荐】局部变量尽量使用基本数据类型

包装类型的坏处:

1）Integer 24字节，而原子类型 int 4字节。

2）包装类型每次赋予还需要额外创建对象，除非在缓存区(见Integer.IntegerCache与Long.LongCache)，Integer var = ?在缓存区间的赋值，会复用h缓存对象。默认缓存区间为-127到128，受启动参数的影响，如-XX:AutoBoxCacheMax=20000。

3）包装类型还有==比较的陷阱（见规则3）

包装类型的好处:

1）包装类型能表达Null的语义。
比如数据库的查询结果可能是null，如果用基本数据类型有NPE风险。又比如显示成交总额涨跌情况，如果调用的RPC服务不成功时，应该返回null，显示成-%，而不是0%。

2）集合需要包装类型，除非使用数组，或者特殊的原子类型集合。
3）泛型需要包装类型，如`Result<Integer>`。

Rule 2.原子数据类型与包装类型的转换原则

2.1【推荐】自动转换(AutoBoxing)有一定成本，调用者与被调用函数间尽量使用同一类型，减少默认转换

```
//WRONG, sum 类型为Long， i类型为long，每次相加都需要AutoBoxing。
Long sum=0L;

for( long i = 0; i < 10000; i++) {
  sum+=i;
}

//RIGHT, 准确使用API返回正确的类型
Integer i = Integer.valueOf(str);
int i = Integer.parseInt(str);
```

- Sonar-2153: Boxing and unboxing should not be immediately reversed

2.2 【推荐】自动拆箱有可能产生NPE，要注意处理

```
//如果intObject为null，产生NPE
int i = intObject;
```

Rule 3. 数值equals比较的原则

3.1【强制】 所有包装类对象之间值的比较，全部使用equals方法比较
\==判断对象是否同一个。Integer var = ?在缓存区间的赋值（见规则1），会复用已有对象，因此这个区间内的Integer使用==进行判断可通过，但是区间之外的所有数据，则会在堆上新产生，不会通过。因此如果用\== 来比较数值，很可能在小的测试数据中通过，而到了生产环境才出问题。

3.2【强制】 BigDecimal需要使用compareTo()

因为BigDecimal的equals()还会比对精度，2.0与2.00不一致。

- Facebook-Contrib: Correctness - Method calls BigDecimal.equals()

3.3【强制】 Atomic* 系列，不能使用equals方法

因为 Atomic* 系列没有覆写equals方法。

```
//RIGHT
if (counter1.get() == counter2.get()){...}
```

- Sonar-2204: ".equals()" should not be used to test the values of "Atomic" classes

3.4【强制】 double及float的比较，要特殊处理

因为精度问题，浮点数间的equals非常不可靠，在vjkit的NumberUtil中有对应的封装函数。

```
float f1 = 0.15f;
float f2 = 0.45f/3; //实际等于0.14999999

//WRONG
if (f1 == f2) {...}
if (Double.compare(f1,f2)==0)

//RIGHT
static final float EPSILON = 0.00001f;
if (Math.abs(f1-f2)<EPSILON) {...}
```

- Sonar-1244: Floating point numbers should not be tested for equality

---

Rule 4. 数字类型的计算原则

4.1【强制】数字运算表达式，因为先进行等式右边的运算，再赋值给等式左边的变量，所以等式两边的类型要一致

例子1: int与int相除后，哪怕被赋值给float或double，结果仍然是四舍五入取整的int。

需要强制将除数或被除数转换为float或double。

```
double d = 24/7;  //结果是3.0
double d =  (double)24/7; //结果是正确的3.42857
```

例子2： int与int相乘，哪怕被赋值给long，仍然会溢出。

需要强制将乘数的一方转换为long。

```
long l = Integer.MAX_VALUE * 2; // 结果是溢出的 - 2
long l = Integer.MAX_VALUE * 2L; //结果是正确的4294967294
```

另外，int的最大值约21亿，留意可能溢出的情况。

- Sonar-2184: Math operands should be cast before assignment

4.2【强制】数字取模的结果不一定是正数，负数取模的结果仍然负数

取模做数组下标时，如果不处理负数的情况，很容易ArrayIndexOutOfBoundException。

另外，Integer.MIN_VALUE取绝对值也仍然是负数。因此，vjkit的MathUtil对上述情况做了安全的封装。

```
-4 % 3  = -1;
Math.abs(Integer.MIN_VALUE) = -2147483648;
```

- Findbugs: Style - Remainder of hashCode could be negative

4.3【推荐】 double 或 float 计算时有不可避免的精度问题

```
float f = 0.45f/3;    //结果是0.14999999

double d1 = 0.45d/3;  //结果是正确的0.15

double d2 = 1.03d - 0.42d; //结果是0.6100000000000001
```

尽量用double而不用float，但如果是金融货币的计算，则必须使用如下选择：

选项1， 使用性能较差的BigDecimal。BigDecimal还能精确控制四舍五入或是其他取舍的方式。

选项2， 在预知小数精度的情况下，将浮点运算放大为整数计数，比如货币以"分"而不是以"元"计算。

- Sonar-2164: Math should not be performed on floats

Rule 5. 【推荐】如果变量值仅有有限的可选值，用枚举类来定义常量

尤其是变量还希望带有名称之外的延伸属性时，如下例：

```
//WRONG
public String MONDAY = "MONDAY";
public int MONDAY_SEQ = 1;

//RIGHT
public enum SeasonEnum {
    SPRING(1), SUMMER(2), AUTUMN(3), WINTER(4);
    int seq;
    SeasonEnum(int seq) { this.seq = seq; }
}
```

业务代码中不要依赖ordinary()函数进行业务运算，而是自定义数字属性，以免枚举值的增减调序造成影响。 例外：永远不会有变化的枚举，比如上例的一年四季。

Rule 6. 字符串拼接的原则

6.1 【推荐】 当字符串拼接不在一个命令行内写完，而是存在多次拼接时(比如循环)，使用StringBuilder的append()

```
String s  = "hello" + str1 +  str2;  //Almost OK，除非初始长度有问题，见第3点.

String s  = "hello";  //WRONG
if (condition) {
  s += str1;
}

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

反编译出的字节码文件显示，其实每条用`+`进行字符拼接的语句，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。所以上面两个错误例子，会重复构造StringBuilder，重复toString()造成资源浪费。

- Sonar-1643: Strings should not be concatenated using '+' in a loop

6.2 【强制】 字符串拼接对象时，不要显式调用对象的toString()

如上，`+`实际是StringBuilder，本身会调用对象的toString()，且能很好的处理null的情况。

```
//WRONG
str = "result:" + myObject.toString();  // myObject为Null时，抛NPE

//RIGHT
str = "result:" + myObject;  // myObject为Null时，输出 result:null
```

6.3【强制】使用StringBuilder，而不是有所有方法都有同步修饰符的StringBuffer

因为内联不成功，逃逸分析并不能抹除StringBuffer上的同步修饰符

- Sonar-1149: Synchronized classes Vector, Hashtable, Stack and StringBuffer should not be used

6.4 【推荐】当拼接后字符串的长度远大于16时，指定StringBuilder的大概长度，避免容量不足时的成倍扩展

6.5 【推荐】如果字符串长度很大且频繁拼接，可考虑ThreadLocal重用StringBuilder对象

参考BigDecimal的toString()实现，及vjkit中的StringBuilderHolder。

Rule 7. 【推荐】字符操作时，优先使用字符参数，而不是字符串，能提升性能

```
//WRONG
str.indexOf("e");

//RIGHT
stringBuilder.append('a');
str.indexOf('e');
str.replace('m','z');
```

其他包括split等方法，在JDK String中未提供针对字符参数的方法，可考虑使用Apache Commons StringUtils 或Guava的Splitter。

- Sonar-3027: String function use should be optimized for single characters

---

Rule 8. 【推荐】利用好正则表达式的预编译功能，可以有效加快正则匹配速度
反例：

```
//直接使用String的matches()方法
result = "abc".matches("[a-zA-z]");

//每次重新构造Pattern
Pattern pattern = Pattern.compile("[a-zA-z]");
result = pattern.matcher("abc").matches();
```

正例：

```
//在某个地方预先编译Pattern，比如类的静态变量
private static Pattern pattern = Pattern.compile("[a-zA-z]");
...
//真正使用Pattern的地方
result = pattern.matcher("abc").matches();
```

(八) 集合处理

Rule 1. 【推荐】底层数据结构是数组的集合，指定集合初始大小

底层数据结构为数组的集合包括 ArrayList，HashMap，HashSet，ArrayDequeue等。

数组有大小限制，当超过容量时，需要进行复制式扩容，新申请一个是原来容量150% or 200%的数组，将原来的内容复制过去，同时浪费了内存与性能。HashMap/HashSet的扩容，还需要所有键值对重新落位，消耗更大。

默认构造函数使用默认的数组大小，比如ArrayList默认大小为10，HashMap为16。因此建议使用ArrayList(int initialCapacity)等构造函数，明确初始化大小。

HashMap/HashSet的初始值还要考虑加载因子:

为了降低哈希冲突的概率(Key的哈希值按数组大小取模后，如果落在同一个数组下标上，将组成一条需要遍历的Entry链)，默认当HashMap中的键值对达到数组大小的75%时，即会触发扩容。因此，如果预估容量是100，即需要设定`100/0.75=134`的数组大小。vjkit的MapUtil的Map创建函数封装了该计算。

如果希望加快Key查找的时间，还可以进一步降低加载因子，加大初始大小，以降低哈希冲突的概率。

Rule 2. 【推荐】尽量使用新式的foreach语法遍历Collection与数组

foreach是语法糖，遍历集合的实际字节码等价于基于Iterator的循环。
foreach代码一来代码简洁，二来有效避免了有多个循环或嵌套循环时，因为不小心的复制粘贴，用错了iterator或循环计数器(i,j)的情况。

Rule 3. 【强制】不要在foreach循环里进行元素的remove/add操作，remove元素可使用Iterator方式

```
//WRONG
for (String str : list) {
  if (condition) {
    list.remove(str);
  }
}

//RIGHT
Iterator<String> it = list.iterator();
while (it.hasNext()) {
  String str = it.next();
  if (condition) {
    it.remove();
  }
}
```

- Facebook-Contrib: Correctness - Method modifies collection element while iterating
- Facebook-Contrib: Correctness - Method deletes collection element while iterating

---

Rule 4. 【强制】使用entrySet遍历Map类集合Key/Value，而不是keySet 方式进行遍历

keySet遍历的方式，增加了N次用key获取value的查询。

- Sonar-2864:"entrySet()" should be iterated when both the key and value are needed

---

Rule 5. 【强制】当对象用于集合时，下列情况需要重新实现hashCode()和 equals()

1 ）以对象做为Map的KEY时；

2 ）将对象存入Set时。

上述两种情况，都需要使用hashCode和equals比较对象，默认的实现会比较是否同一个对象（对象的引用相等）。

另外，对象放入集合后，会影响hashCode()，equals()结果的属性，将不允许修改。

- Sonar-2141:Classes that don't define "hashCode()" should not be used in hashes

---

Rule 6. 【强制】高度注意各种Map类集合Key/Value能不能存储null值的情况

Map	Key	Value
HashMap	Nullable	Nullable
ConcurrentHashMap	NotNull	NotNull
TreeMap	NotNull	Nullable

由于HashMap的干扰，很多人认为ConcurrentHashMap是可以置入null值。同理，Set中的value实际是Map中的key。

Rule 7. 【强制】长生命周期的集合，里面内容需要及时清理，避免内存泄漏

长生命周期集合包括下面情况，都要小心处理。

1）静态属性定义；

2）长生命周期对象的属性；

3）保存在ThreadLocal中的集合。

如无法保证集合的大小是有限的，使用合适的缓存方案代替直接使用HashMap。

另外，如果使用WeakHashMap保存对象，当对象本身失效时，就不会因为它在集合中存在引用而阻止回收。但JDK的WeakHashMap并不支持并发版本，如果需要并发可使用Guava Cache的实现。

Rule 8. 【强制】集合如果存在并发修改的场景，需要使用线程安全的版本

1) 著名的反例，HashMap扩容时，遇到并发修改可能造成100%CPU占用。

推荐使用`java.util.concurrent(JUC)`工具包中的并发版集合，如ConcurrentHashMap等，优于使用Collections.synchronizedXXX()系列函数进行同步化封装(等价于在每个方法都加上synchronized关键字)。

例外：ArrayList所对应的CopyOnWriteArrayList，每次更新时都会复制整个数组，只适合于读多写很少的场景。如果频繁写入，可能退化为使用Collections.synchronizedList(list)。

2) 即使线程安全类仍然要注意函数的正确使用。

例如：即使用了ConcurrentHashMap，但直接是用get/put方法，仍然可能会多线程间互相覆盖。

```
//WRONG
E e = map.get(key);
if (e == null) {
  e = new E();
  map.put(key, e); //仍然能两条线程并发执行put，互相覆盖
}
return e;

//RIGHT
E e = map.get(key);
if (e == null) {
  e = new E();
  E previous = map.putIfAbsent(key, e);
  if(previous != null) {
```

```
    return previous;
  }
}
return e;
```

Rule 9. 【推荐】正确使用集合泛型的通配符

`List<String>`并不是`List<Object>`的子类，如果希望泛型的集合能向上向下兼容转型，而不仅仅适配唯一类，则需定义通配符，可以按需要extends 和 super的字面意义，也可以遵循`PECS(Producer Extends Consumer Super)`原则:

1) 如果集合要被读取，定义成`<? extends T>`

```
Class Stack<E>{
  public void pushAll(Iterable<? extends E> src){
    for (E e: src)
      push(e);
  }
}

Stack<Number> stack = new Stack<Number>();
Iterable<Integer> integers = ...;
stack.pushAll(integers);
```

2) 如果集合要被写入，定义成`<? super T>`

```
Class Stack<E>{
  public void popAll(Collection<? super E> dist){
     while(!isEmpty())
       dist.add(pop);
  }
}

Stack<Number> stack = new Stack<Number>();
Collection<Object> objects = ...;
stack.popAll(objects);
```

Rule 10. 【推荐】`List`，`List<?>`与`List<Object>`的选择

定义成`List`，会被IDE提示需要定义泛型。 如果实在无法确定泛型，就仓促定义成`List<?>`来蒙混过关的话，该list只能读，不能增改。定义成`List<Object>`呢，如规则10所述，`List<String>`并不是`List<Object>`的子类，除非函数定义使用了通配符。

因此实在无法明确其泛型时，使用`List`也是可以的。

Rule 11. 【推荐】如果Key只有有限的可选值，先将Key封装成Enum，并使用EnumMap

EnumMap，以Enum为Key的Map，内部存储结构为`Object[enum.size]`，访问时以`value = Object[enum.ordinal()]`获取值，同时具备HashMap的清晰结构与数组的性能。

```
public enum COLOR {
  RED, GREEN, BLUE, ORANGE;
}

EnumMap<COLOR, String> moodMap = new EnumMap<COLOR, String> (COLOR.class);
```

- Sonar-1640: Maps with keys that are enum values should be replaced with EnumMap

---

Rule 12. 【推荐】Array 与 List互转的正确写法

```
// list -> array，构造数组时不需要设定大小
String[] array = (String[])list.toArray(); //WRONG;
String[] array = list.toArray(new String[0]); //RIGHT
String[] array = list.toArray(new String[list.size()]); //RIGHT，但list.size()可用0代替。


// array -> list
//非原始类型数组，且List不能再扩展
List list = Arrays.asList(array);

//非原始类型数组， 但希望List能再扩展
List list = new ArrayList(array.length);
Collections.addAll(list, array);

//原始类型数组，JDK8
List myList = Arrays.stream(intArray).boxed().collect(Collectors.toList());

//原始类型数组，JDK7则要自己写个循环来加入了
```

Arrays.asList(array)，如果array是原始类型数组如int[]，会把整个array当作List的一个元素，String[] 或 Foo[]则无此问题。 Collections.addAll()实际是循环加入元素，性能相对较低，同样会把int[]认作一个元素。

- Facebook-Contrib: Correctness - Impossible downcast of toArray() result
- Facebook-Contrib: Correctness - Method calls Array.asList on an array of primitive values

---

(九) 并发处理

Rule 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

1）创建单条线程时直接指定线程名称

```
Thread t = new Thread();
t.setName("cleanup-thread");
```

2）线程池则使用guava或自行封装的ThreadFactory，指定命名规则。

```
//guava 或自行封装的ThreadFactory
ThreadFactory threadFactory = new ThreadFactoryBuilder().setNameFormat(threadNamePrefix + "-
%d").build();

ThreadPoolExecutor executor = new ThreadPoolExecutor(..., threadFactory, ...);
```

Rule 2. 【推荐】尽量使用线程池来创建线程

除特殊情况，尽量不要自行创建线程，更好的保护线程资源。

```
//WRONG
Thread thread = new Thread(...);
thread.start();
```

同理，定时器也不要使用Timer，而应该使用ScheduledExecutorService。

因为Timer只有单线程，不能并发的执行多个在其中定义的任务，而且如果其中一个任务抛出异常，整个Timer也会挂掉，而ScheduledExecutorService只有那个没捕获到异常的任务不再定时执行，其他任务不受影响。

Rule 3. 【强制】线程池不允许使用 Executors去创建，避资源耗尽风险

Executors返回的线程池对象的弊端 ：

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 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。

应通过 new ThreadPoolExecutor(xxx,xxx,xxx,xxx)这样的方式，更加明确线程池的运行规则，合理设置Queue及线程池的core size和max size，建议使用vjkit封装的ThreadPoolBuilder。

Rule 4. 【强制】正确停止线程

Thread.stop()不推荐使用，强行的退出太不安全，会导致逻辑不完整，操作不原子，已被定义成Deprecate方法。

停止单条线程，执行Thread.interrupt()。

停止线程池：

- ExecutorService.shutdown(): 不允许提交新任务，等待当前任务及队列中的任务全部执行完毕后退出；
- ExecutorService.shutdownNow(): 通过Thread.interrupt()试图停止所有正在执行的线程，并不再处理还在队列中等待的任务。

最优雅的退出方式是先执行shutdown()，再执行shutdownNow()，vjkit的`ThreadPoolUtil`进行了封装。

注意，Thread.interrupt()并不保证能中断正在运行的线程，需编写可中断退出的Runnable，见规则5。

Rule 5. 【强制】编写可停止的Runnable

执行Thread.interrupt()时，如果线程处于sleep(), wait(), join(), lock.lockInterruptibly()等blocking状态，会抛出InterruptedException，如果线程未处于上述状态，则将线程状态设为interrupted。

因此，如下的代码无法中断线程:

```
public void run() {

  while (true) { //WRONG，无判断线程状态。
    sleep();
  }

  public void sleep() {
    try {
      Thread.sleep(1000);
    } catch (InterruptedException e) {
      logger.warn("Interrupted!", e); //WRONG，吃掉了异常，interrupt状态未再传递
    }
  }
}
```

5.1 正确处理InterruptException

因为InterruptException异常是个必须处理的Checked Exception，所以run()所调用的子函数很容易吃掉异常并简单的处理成打印日志，但这等于停止了中断的传递，外层函数将收不到中断请求，继续原有循环或进入下一个堵塞。

正确处理是调用`Thread.currentThread().interrupt();`将中断往外传递。

```
//RIGHT
public void myMethod() {
  try {
    ...
  } catch (InterruptedException e) {
    Thread.currentThread().interrupt();
  }
}
```

- Sonar-2142: "InterruptedException" should not be ignored

5.2 主循环及进入阻塞状态前要判断线程状态

```
//RIGHT
public void run() {
  try {
    while (!Thread.isInterrupted()) {
      // do stuff
    }
  } catch (InterruptedException e) {
    logger.warn("Interrupted!", e);
  }
}
```

其他如Thread.sleep()的代码，在正式sleep前也会判断线程状态。

Rule 6. 【强制】Runnable中必须捕获一切异常

如果Runnable中没有捕获RuntimeException而向外抛出，会发生下列情况：

1) ScheduledExecutorService执行定时任务，任务会被中断，该任务将不再定时调度，但线程池里的线程还能用于其他任务。

2) ExecutorService执行任务，当前线程会中断，线程池需要创建新的线程来响应后续任务。

3) 如果没有在ThreadFactory设置自定义的UncaughtExceptionHanlder，则异常最终只打印在System.err，而不会打印在项目的日志中。

因此建议自写的Runnable都要保证捕获异常; 如果是第三方的Runnable，可以将其再包裹一层vjkit中的SafeRunnable。

```
executor.execute(ThreadPoolUtil.safeRunner(runner));
```

Rule 7. 【强制】全局的非线程安全的对象可考虑使用ThreadLocal存放

全局变量包括单例对象，static成员变量。

著名的非线程安全类包括SimpleDateFormat，MD5/SHA1的Digest。

对这些类，需要每次使用时创建。

但如果创建有一定成本，可以使用ThreadLocal存放并重用。

ThreadLocal变量需要定义成static，并在每次使用前重置。

```
private static final ThreadLocal<MessageDigest> SHA1_DIGEST = new ThreadLocal<MessageDigest>() {
  @Override
  protected MessageDigest initialValue() {
    try {
      return MessageDigest.getInstance("SHA");
    } catch (NoSuchAlgorithmException e) {
      throw new RuntimeException("...", e);
    }
  }
```

```
};

public void digest(byte[] input) {
  MessageDigest digest = SHA1_DIGEST.get();
  digest.reset();
  return digest.digest(input);
}
```

- Sonar-2885: Non-thread-safe fields should not be static
- Facebook-Contrib: Correctness - Field is an instance based ThreadLocal variable

---

Rule 8. 【推荐】缩短锁

1）能锁区块，就不要锁整个方法体；

```
//锁整个方法，等价于整个方法体内synchronized(this)
public synchronized boolean foo(){};

//锁区块方法，仅对需要保护的原子操作的连续代码块进行加锁。
public boolean foo() {
    synchronized(this) {
        ...
        ...
    }
    //other stuff
}
```

2）能用对象锁，就不要用类锁。

```
//对象锁，只影响使用同一个对象加锁的线程
synchronized(this) {
    ...
}

//类锁，使用类对象作为锁对象，影响所有线程。
synchronized(A.class) {
    ...
}
```

Rule 10. 【推荐】选择分离锁，分散锁甚至无锁的数据结构

- 分离锁：

1）读写分离锁ReentrantReadWriteLock，读读之间不加锁，仅在写读和写写之间加锁；

2）Array Base的queue一般是全局一把锁，而Linked Base的queue一般是队头队尾两把锁。

- 分散锁（又称分段锁）：

1）如JDK7的ConcurrentHashMap，分散成16把锁；

2）对于经常写，少量读的计数器，推荐使用JDK8或vjkit封装的LongAdder对象性能更好（内部分散成多个counter，减少乐观锁的使用，取值时再相加所有counter）

- 无锁的数据结构：

1）完全无锁无等待的结构，如JDK8的ConcurrentHashMap；

2）基于CAS的无锁有等待的数据结构，如AtomicXXX系列。

Rule 11.【推荐】基于ThreadLocal来避免锁

比如Random实例虽然是线程安全的，但其实它的seed的访问是有锁保护的。因此建议使用JDK7的ThreadLocalRandom，通过在每个线程里放一个seed来避免了加锁。

Rule 12.【推荐】规避死锁风险

对多个资源多个对象的加锁顺序要一致。

如果无法确定完全避免死锁，可以使用带超时控制的tryLock语句加锁。

Rule 13.【推荐】volatile修饰符，AtomicXX系列的正确使用

多线程共享的对象，在单一线程内的修改并不保证对所有线程可见。使用volatile定义变量可以解决（解决了可见性）。

但是如果多条线程并发进行基于当前值的修改，如并发的counter++，volatile则无能为力（解决不了原子性）。

此时可使用Atomic*系列:

```
AtomicInteger count = new AtomicInteger();
count.addAndGet(2);
```

但如果需要原子地同时对多个AtomicXXX的Counter进行操作，则仍然需要使用synchronized将改动代码块加锁。

Rule 14.【推荐】延时初始化的正确写法

通过双重检查锁（double-checked locking）实现延迟初始化存在隐患，需要将目标属性声明为volatile型，为了更高的性能，还要把volatile属性赋予给临时变量，写法复杂。

所以如果只是想简单的延迟初始化，可用下面的静态类的做法，利用JDK本身的class加载机制保证唯一初始化。

```
private static class LazyObjectHolder {
  static final LazyObject instance = new LazyObject();
}

public void myMethod() {
  LazyObjectHolder.instance.doSomething();
}
```

- Sonar-2168: Double-checked locking should not be used

(十) 异常处理

Rule 1. 【强制】创建异常的消耗大，只用在真正异常的场景

构造异常时，需要获得整个调用栈，有一定消耗。
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判断低。

发生概率较高的条件，应该先进行检查规避，比如：IndexOutOfBoundsException，NullPointerException等，所以如果代码里捕获这些异常通常是个坏味道。

```
//WRONG
try {
  return obj.method();
} catch (NullPointerException e) {
  return false;
}

//RIGHT
if (obj == null) {
  return false;
}
```

- Sonar-1696: "NullPointerException" should not be caught

Rule 2. 【推荐】在特定场景，避免每次构造异常

如上，异常的构造函数需要获得整个调用栈。

如果异常频繁发生，且不需要打印完整的调用栈时，可以考虑绕过异常的构造函数。

1）如果异常的message不变，将异常定义为静态成员变量;

下例定义静态异常，并简单定义一层的StackTrace。`ExceptionUtil`见vjkit。

```
private static RuntimeException TIMEOUT_EXCEPTION = ExceptionUtil.setStackTrace(new
RuntimeException("Timeout"),
MyClass.class, "mymethod");

...

throw TIMEOUT_EXCEPTION;
```

2）如果异常的message会变化，则对静态的异常实例进行clone()再修改message。

Exception默认不是Cloneable的，`CloneableException`见vjkit。

```
private static CloneableException TIMEOUT_EXCEPTION = new CloneableException("Timeout")
.setStackTrace(My.class,
 "hello");

...

throw TIMEOUT_EXCEPTION.clone("Timeout for 40ms");
```

3）自定义异常，也可以考虑重载fillStackTrace()为空函数，但相对没那么灵活，比如无法按场景指定一层的StackTrace。

Rule 3. 【推荐】自定义异常，建议继承`RuntimeException`

详见《Clean Code》，争论已经结束，不再推荐原本初衷很好的CheckedException。

因为CheckedException需要在抛出异常的地方，与捕获处理异常的地方之间，层层定义throws XXX来传递Exception，如果底层代码改动，将影响所有上层函数的签名，导致编译出错，对封装的破坏严重。对CheckedException的处理也给上层程序员带来了额外的负担。因此其他语言都没有CheckedException的设计。

Rule 4. 【推荐】异常日志应包含排查问题的足够信息

异常信息应包含排查问题时足够的上下文信息。

捕获异常并记录异常日志的地方，同样需要记录没有包含在异常信息中，而排查问题需要的信息，比如捕获处的上下文信息。

```
//WRONG
new TimeoutException("timeout");
logger.error(e.getMessage(), e);


//RIGHT
new TimeoutException("timeout:" + eclapsedTime + ", configuration:" + configTime);
logger.error("user[" + userId + "] expired:" + e.getMessage(), e);
```

- Facebook-Contrib: Style - Method throws exception with static message string

---

Rule 5. 异常抛出的原则

5.1 【推荐】尽量使用JDK标准异常，项目标准异常

尽量使用JDK标准的Runtime异常如`IllegalArgumentException`，`IllegalStateException`，`UnsupportedOperationException`，项目定义的Exception如`ServiceException`。

5.2 【推荐】根据调用者的需要来定义异常类，直接使用`RuntimeException`是允许的

是否定义独立的异常类，关键是调用者会如何处理这个异常，如果没有需要特别的处理，直接抛出RuntimeException也是允许的。

Rule 6. 异常捕获的原则

6.1 【推荐】按需要捕获异常，捕获`Exception`或`Throwable`是允许的

如果无特殊处理逻辑，统一捕获Exception统一处理是允许的。

捕获Throwable是为了捕获Error类异常，包括其实无法处理的`OOM` `StackOverflow` `ThreadDeath`，以及类加载，反射时可能抛出的`NoSuchMethodError` `NoClassDefFoundError`等。

6.2【推荐】多个异常的处理逻辑一致时，使用JDK7的语法避免重复代码

```
try {
  ...
} catch (AException | BException | CException ex) {
  handleException(ex);
}
```

- Sonar-2147: Catches should be combined

Rule 7.异常处理的原则

7.1 【强制】捕获异常一定要处理；如果故意捕获并忽略异常，须要注释写明原因

方便后面的阅读者知道，此处不是漏了处理。

```
//WRONG
try {
} catch(Exception e) {
}

//RIGHT
try {
} catch(Exception ignoredExcetpion) {
    //continue the loop
}
```

7.2 【强制】异常处理不能吞掉原异常，要么在日志打印，要么在重新抛出的异常里包含原异常

```
 //WRONG
throw new MyException("message");

//RIGHT 记录日志后抛出新异常，向上次调用者屏蔽底层异常
logger.error("message", ex);
throw new MyException("message");

//RIGHT 传递底层异常
throw new MyException("message", ex);
```

- Sonar-1166: Exception handlers should preserve the original exceptions，其中默认包含了InterruptedException, NumberFormatException，NoSuchMethodException等若干例外

7.3 【强制】如果不想处理异常，可以不进行捕获。但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

Rule 8. finally块的处理原则

8.1 【强制】必须对资源对象、流对象进行关闭，或使用语法try-with-resource

关闭动作必需放在finally块，不能放在try块 或 catch块，这是经典的错误。

更加推荐直接使用JDK7的try-with-resource语法自动关闭Closeable的资源，无需在finally块处理，避免潜在问题。

```
try (Writer writer = ...) {
  writer.append(content);
}
```

8.2 【强制】如果处理过程中有抛出异常的可能，也要做try-catch，否则finally块中抛出的异常，将代替try块中抛出的异常

```
//WRONG
try {
  ...
  throw new TimeoutException();
} finally {
  file.close();//如果file.close()抛出IOException，将代替TimeoutException
}

//RIGHT，在finally块中try - catch
try {
  ...
  throw new TimeoutException();
} finally {
  IOUtil.closeQuietly(file); //该方法中对所有异常进行了捕获
}
```

- Sonar-1163: Exceptions should not be thrown in finally blocks

8.3 【强制】不能在finally块中使用return，finally块中的return将代替try块中的return及throw Exception

```
//WRONG
try {
  ...
  return 1;
} finally {
  return 2; //实际return 2 而不是1
}

try {
  ...
  throw TimeoutException();
} finally {
  return 2; //实际return 2 而不是TimeoutException
}
```

- Sonar-1143: Jump statements should not occur in "finally" blocks

---

(十一) 日志规约

Rule 1. 【强制】应用中不可直接使用日志库（Log4j、Logback）中的API，而应使用日志框架SLF4J中的API

使用门面模式的日志框架，有利于维护各个类的日志处理方式统一。

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

private static Logger logger = LoggerFactory.getLogger(Foo.class);
```

Rule 2. 【推荐】对不确定会否输出的日志，采用占位符或条件判断

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

如果日志级别是info，上述日志不会打印，但是会执行1)字符串拼接操作，2)如果symbol是对象，还会执行toString()方法，浪费了系统资源，最终日志却没有打印。

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

但如果symbol.getMessage()本身是个消耗较大的动作，占位符在此时并没有帮助，须要改为条件判断方式来完全避免它的执行。

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

//RIGHT
if (logger.isDebugEnabled()) {
  logger.debug("Processing trade with id: " + id + " symbol: " + symbol.getMessage());
}
```

Rule 3. 【推荐】对确定输出，而且频繁输出的日志，采用直接拼装字符串的方式

如果这是一条WARN，ERROR级别的日志，或者确定输出的INFO级别的业务日志，直接字符串拼接，比使用占位符替换，更加高效。

Slf4j的占位符并没有魔术，每次输出日志都要进行占位符的查找，字符串的切割与重新拼接。

```
//RIGHT
logger.info("I am a business log with id: " + id + " symbol: " + symbol);

//RIGHT
logger.warn("Processing trade with id: " + id + " symbol: " + symbol);
```

Rule 4. 【推荐】尽量使用异步日志

低延时的应用，使用异步输出的形式(以AsyncAppender串接真正的Appender)，可减少IO造成的停顿。

需要正确配置异步队列长度及队列满的行为，是丢弃还是等待可用，业务上允许丢弃的尽量选丢弃。

Rule 5. 【强制】禁止使用性能很低的System.out()打印日志信息

同理也禁止e.printStackTrace();

例外: 应用启动和关闭时，担心日志框架还未初始化或已关闭。

- Sonar-106: Standard outputs should not be used directly to log anything
- Sonar-1148: Throwable.printStackTrace(...) should not be called

---

Rule 6. 【强制】禁止配置日志框架输出日志打印处的类名，方法名及行号的信息

日志框架在每次打印时，通过主动获得当前线程的StackTrace来获取上述信息的消耗非常大，尽量通过Logger名本身给出足够信息。

Rule 7. 【推荐】谨慎地记录日志，避免大量输出无效日志，信息不全的日志

大量地输出无效日志，不利于系统性能，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？

Rule 8. 【推荐】使用warn级别而不是error级别，记录外部输入参数错误的情况

如非必要，请不在此场景打印error级别日志，避免频繁报警。

error级别只记录系统逻辑出错、异常或重要的错误信息。

(十二) 其他规约

Rule 1. 【参考】尽量不要让魔法值（即未经定义的数字或字符串常量）直接出现在代码中

```
//WRONG
String key = "Id#taobao_"+tradeId;
cache.put(key, value);
```

例外：-1,0,1,2,3 不认为是魔法数

- Sonar-109: Magic numbers should not be used 但现实中所谓魔法数还是太多，该规则不能被真正执行。

Rule 2. 【推荐】时间获取的原则

1）获取当前毫秒数System.currentTimeMillis() 而不是new Date().getTime()，后者的消耗要大得多。

2）如果要获得更精确的，且不受NTP时间调整影响的流逝时间，使用System.nanoTime()获得机器从启动到现在流逝的纳秒数。

3）如果希望在测试用例中控制当前时间的值，则使用vjkit的Clock类封装，在测试和生产环境中使用不同的实现。

Rule 3. 【推荐】变量声明尽量靠近使用的分支

不要在一个代码块的开头把局部变量一次性都声明了(这是c语言的做法)，而是在第一次需要使用它时才声明。
否则如果方法已经退出或进入其他分支，就白白初始化了变量。

```
//WRONG
Foo foo = new Foo();

if(ok){
    return;
}

foo.bar();
```

Rule 4. 【推荐】不要像C那样一行里做多件事情

```
 //WRONG
fooBar.fChar = barFoo.lchar = 'c';
argv++; argc--;
int level, size;
```

- Sonar-1659: Multiple variables should not be declared on the same line

---

Rule 5. 【推荐】不要为了性能而使用JNI本地方法

Java在JIT后并不比C代码慢，JNI方法因为要反复跨越JNI与Java的边界反而有额外的性能损耗。
因此JNI方法仅建议用于调用"JDK所没有包括的, 对特定操作系统的系统调用"

Rule 6. 【推荐】正确使用反射，减少性能损耗

获取Method/Field对象的性能消耗较大, 而如果对Method与Field对象进行缓存再反复调用，则并不会比直接调用类的方法与成员变量慢（前15次使用NativeAccessor，第15次后会生成GeneratedAccessorXXX，bytecode为直接调用实际方法）

```
//用于对同一个方法多次调用
private Method method = ....

public void foo(){
  method.invoke(obj, args);
}

//用于仅会对同一个方法单次调用
ReflectionUtils.invoke(obj, methodName, args);
```

Rule 7.【推荐】可降低优先级的常见代码检查规则

1. 接口内容的定义中，去除所有modifier，如public等。(多个public也没啥，反正大家都看惯了)
2. 工具类，定义private构造函数使其不能被实例化。（命名清晰的工具类，也没人会去实例化它，对静态方法通过类来访问也能避免实例化）

《阿里Java开发手册》定制纪录

只记录较大的改动，对更多条目内容的重新组织与扩写，则未一一冗述。

- 《唯品会Java开发手册》 - 与阿里手册的比较文学I

(一) 命名规约

对应 阿里规范《命名风格》一章

VIP 规范	阿里规范	修改
13. 变量、参数重名覆盖		新增规则
1. 禁止拼音缩写	2. 严禁使用拼音与英文混合的方式	改写规则
3. 禁用其他编程语言风格的前缀和后缀	1. 代码中的命名均不能以下划线或美元符号开始	扩写规则，把其他语言的啰嗦都禁止掉
4. 命名的好坏，在于其“模糊度”	11. 为了达到代码自解释的目标	扩写规则，参考《Clean Code》的更多例子
6. 常量命名全部大写	5.常量名大写	扩写规则
	7. 类型与中括号紧挨相连来定义数组	删除规则，非命名风格，也不重要
	13. 接口类中的方法和成员变量不要加任何修饰符号	移动规则，非命名风格，移到类设计
	16. 各层命名规约	删除规则，各公司有自己的习惯

(二) 格式规约

对应 阿里规范《代码格式》一章

VIP 规范	阿里规范	修改
1. 项目组统一的代码格式模板	规则1-8	用IDE模版代替逐条描述 同时对Tab/空格不做硬性规定
3. 用小括号来限定运算优先级		新增规则
4. 类内方法定义的顺序		新增规则
5. 通过空行进行逻辑分段	11. 不同逻辑、不同语义	改写规则
6. 避免IDE格式化		新增规则
	10.单个方法行数不超过80行	删除规则，非格式规约，移动方法设计
	11.没有必要增加若干空格来对齐	删除规则，现在很少人这么做

(三) 注释规约

对应 阿里规范《注释规约》一章

VIP 规范	阿里规范	修改
2. 删除空注释，无意义注释		增加规则
7. JavaDoc中不要大量使用HTML标签和转义字符		增加规则
1. 注释的基本要求	9. 对于注释的要求	扩写规则
4.避免创建人的注释	3.所有的类都必须添加创建者	冲突规则
	2.所有的抽象方法必须用Javadoc注释	删除规则，因为规则2不强制，并入规则1
	4.方法内部单行注释，使用//注释	删除规则，区别不大不强求
	5. 所有的枚举类型字段必须要有注释	删除规则，因为规则2不强制

(四) 方法设计

- 规则 6，7，12，13 从阿里规范《控制语句》一章 移入
- 规则 9 从阿里规范《异常处理》一章 移入
- 规则1，2，3，4，5，8，10，11，14为新建规则

(五) 类设计

对应 阿里规范《OOP规范》一章

VIP 规范	阿里规范	修改
2.减少类之间的依赖		增加规则
3.定义变量与方法参数时，尽量使用接口		增加规则
4.类的长度度量		增加规则
5.Builder模式		增加规则
8.静态方法不能被覆写		增加规则
9.静态方法的访问原则		扩写规则
10.内部类原则		增加规则
12-14.hashCode，equals，toString的规则		增加规则
16.总是移除无用属性、方法与参数		增加规则
18.【推荐】得墨忒耳法则		增加规则
	3. 提倡同学们尽量不用可变参数编程	删除规则
	9. 定义DO/DTO/VO等POJO类时，不要设定任何属性默认值	删除规则
	10. 序列化类新增属性时，请不要修改serialVersionUID字段	删除规则
	13. 使用索引访问用String的split方法时	删除规则
	19. 慎用Object的clone方法来拷贝对象	删除规则
	规则4，5	移到《方法规约》
	规则6	移到《通用设计》
	规则7，8，17	移到《基础类型》
	规则14，15	移到《格式规约》

(六) 控制语句

对应 阿里规范《控制语句》一章

VIP 规范	阿里规范	修改
4.布尔表达式中的运算符个数不超过4个		扩写规则
5.善用三元运算符		增加规则
6.能造成短路概率较大的逻辑放前面		增加规则
10.能用while循环实现的代码，就不用do-while循环		增加规则
	3. 在高并发场景中，避免使用 ”等于”作为条件	删除规则
	8. 接口入参保护	删除规则
	9. 下列情形，需要进行参数校验	移到《方法规约》
	10. 下列情形，不需要进行参数校	移到《方法规约》

(七) 基本类型与字符串

- 规则1，3 从 阿里规范《OOP规范》移入，并对2进行扩写
- 规则5 从 阿里规范《常量定义》移入并扩写
- 规则8 从 阿里规范《其他》移入
- 规则4，6，7为新建规则

(八) 集合处理

对应 阿里规范《集合处理》一章

VIP 规范	阿里规范	修改
2. foreach语法遍历		增加规则
7. 长生命周期的集合		增加规则
8. 并发集合		增加规则
9. 泛型的通配符		增加规则
10. `List, List<?>` 与 `List<Object>` 的选择		增加规则
11. EnumMap		增加规则
	2. ArrayList的subList结果	删除规则
	6. 泛型通配符	删除规则
	12.合理利用好集合的有序性	删除规则
	13.利用Set元素唯一的特性	删除规则
12.Array 与 List互转的	使用集合转数组的方法，必须使用集合的toArray	某位老大的测试，new String[0]也不错

(九) 并发处理： 并发与多线程

对应 阿里规范《并发处理》一章

VIP 规范	阿里规范	修改
1. 指定线程名		扩写规则
4. 正确停止线程		扩写规则
5. 编写可中断的Runnable		增加规则
6. Runnable中必须捕获一切异常	9.多线程并行处理定时任务	扩写规则
7. 全局变量的线程安全		扩写规则
10. 选择分离锁, 分散锁甚至无锁的数据结构		增加规则
13. volatile修饰符，AtomicXX系列的正确使用		扩写规则
	8.并发修改同一记录时，需要加锁	删除规则
	10.使用CountDownLatch进行异步转同步操作	删除规则
	14.HashMap在容量不够进行resize	移到《集合规约》一章
14. 延时初始化的正确写法	12.双重检查锁	冲突规则

(十) 异常处理

对应 阿里规范《异常处理》一章

VIP 规范	阿里规范	修改
2.在特定场合，避免每次构造异常		增加规则
5.异常抛出的原则		增加规则
6.异常捕获的原则		增加规则
7.异常处理的原则		增加规则
	8.捕获异常与抛异常，必须是完全匹配	删除规则
	12.对于公司外的开放接口必须使用“错误码”	删除规则
	13.DRY原则	删除规则，为什么会出现在这章，太著名了
	9.返回值可以为null	移到《方法设计》一章
	10. 【推荐】防止NPE，是程序员的基本修养	拆开到各章
	11.避免直接抛出RuntimeException	规则冲突

(十一) 日志规约

对应 阿里规范《日志规约》一章

VIP 规范	阿里规范	修改
4.尽量使用异步日志		增加规则
5.禁止使用System.out()		增加规则
6.禁止配置日志框架打印日志打印时的类名，行号等信息		增加规则
	2.日志文件推荐至少保存15天	删除规则
	3.应用中的扩展日志命名方式	删除规则
	6.异常信息应该包括两类信息	移到《异常处理》
2.合理使用使用占位符	4.对trace/debug/info级别的日志使用占位符	还是要判断日志是否必然输出， 并强调条件判断与占位符之间的差别

(十二) 其他规约

保留 阿里规范《常量定义》一章的规则1

VIP 规范	阿里规范	修改
	规则2 - 4	删除规则
	规则5. 如果变量值仅在一个固定范围内变化用enum类型来定义	移到《基本类型》

保留 阿里规范《其他》一章的规则7

VIP 规范	阿里规范	修改
	规则2 - 4，6 - 8	删除规则
	规则1. 在使用正则表达式时，利用好其预编译功	移到《基本类型》

- 规则3，4，5，6，7均为新增规则