一、编程规约

(一) 命名风格

1. 【强制】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\_\_

解释：有很多开源代码是下划线开头，比如我们要讲的eureka源码中，我们可以不吸收这种风格。因为python下划线开头是内部变量，此规定是让其他语言转过来的开发者更容易理解。

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。

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，纯拼音命名方式更要避免采用。

正例：renminbi / 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

解释：虽然山东舰交付了，中国在世界上影响力增大了，但还是不建议用拼音或中文，中文有同音字，汉字会有编码问题，Pingfen：评分，平分。

1. 【强制】类名使用 UpperCamelCase 风格，但以下情形例外：DO / BO / DTO / VO / AO / PO / UID 等。

正例：JavaServerlessPlatform / UserDO / XmlService / TcpUdpDeal / TaPromotion

反例：javaserverlessplatform / UserDo / XMLService / TCPUDPDeal / TAPromotion

Ta：谢谢。

解释:DO（ Data Object）：与数据库表结构一一对应，通过DAO层向上传输数据源对象。

DTO（ Data Transfer Object）：数据传输对象，Service或Manager向外传输的对象。

BO（ Business Object）：业务对象。 由Service层输出的封装业务逻辑的对象。

VO（ View Object）：显示层对象，通常是Web向模板渲染引擎层传输的对象。

AO (Application Object) :应用对象,在 Web 层与 Service 层之间抽象的复用对象模型,极为贴近展示层,复用度不高。

POJO 是 DO / DTO / BO / VO 的统称.

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。

解释：便于阅读，一下就知道是数组。国产化标准。像高铁。

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。

说明：在本文 MySQL 规约中的建表约定第一条，表达是与否的值采用 is\_xxx 的命名方式，所以，需要在<resultMap>设置从 is\_xxx 到 xxx 的映射关系。

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。

解释：这是开发中的一个坑，如果不知道，这个点，很难发现问题。debug，com.p3c.controller.ApiTestController，在postman中传入

{

"num":1,

"isSuccess":true

}

但是controller收到的却是false;

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。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）

10. 【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。

说明：子类、父类成员变量名相同，即使是 public 类型的变量也是能够通过编译，而局部变量在同一方法内的不同代码块中同名也是合法的，但是要避免使用。对于非 setter/getter 的参数名称也要避免与成员变量名称相同。

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 {

//允许与父类的成员变量名称相同

public int age;

}

解释：便于阅读。

11. 【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。

解释：你懂但是别人不懂。

12. 【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。

13. 【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

解释：没有原因，正常些思考。

14. 【推荐】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;

15. 【推荐】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";

反例：接口方法定义 public abstract void f();

说明：JDK8 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。

解释：接口变量：public static final，接口方法：public abstract，大家都知道的，就没必要累赘。

16. 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。

例：CacheServiceImpl 实现 CacheService 接口。

2） 【推荐】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 的形容词）。

正例：AbstractTranslator 实现 Translatable 接口。

解释：SOA：Service-Oriented Architecture，面向服务架构。服务间只通过接口进行沟通。

17. 【参考】枚举类名带上 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\_REASON。

18. 【参考】各层命名规约：

A) Service/DAO 层方法命名规约

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 list 做前缀，复数形式结尾如：listObjects。

3） 获取统计值的方法用 count 做前缀。

4） 插入的方法用 save/insert 做前缀。

5） 删除的方法用 remove/delete 做前缀。

6） 修改的方法用 update 做前缀。

1. 领域模型命名规约

1） 数据对象：xxxDO，xxx 即为数据表名。

2） 数据传输对象：xxxDTO，xxx 为业务领域相关的名称。

3） 展示对象：xxxVO，xxx 一般为网页名称。

4） POJO 是 DO/DTO/BO/VO 的统称，禁止命名成 xxxPOJO。

解释：领域模型是一个分析模型，帮助系统分析人员、用户认识现实业务的工具，描述的是业务中涉及到的实体及其相互之间的关系，它是需求分析的产物，与问题域相关。

DO（ Data Object）：与数据库表结构一一对应，通过DAO层向上传输数据源对象。

DTO（ Data Transfer Object）：数据传输对象，Service或Manager向外传输的对象。

BO（ Business Object）：业务对象。 由Service层输出的封装业务逻辑的对象。

VO（ View Object）：显示层对象，通常是Web向模板渲染引擎层传输的对象。

(二) 常量定义

1. 【强制】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。

反例：String key = "Id#taobao\_" + tradeId;

cache.put(key, value);

// 缓存 get 时，由于在代码复制时，漏掉下划线，导致缓存击穿而出现问题

1. 【强制】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 = 2l; 写的是数字的 21，还是 Long 型的 2。

1. 【推荐】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。

解释：大而全其实相当于没有，比如读书，重点划线，如果一本书都划线，和没划线有什么区别呢？

1. 【推荐】常量的复用层次有五层：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。

1） 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。

2） 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。

反例：易懂变量也要统一定义成应用内共享常量，两位工程师在两个类中分别定义了“YES”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。

3） 子工程内部共享常量：即在当前子工程的 constant 目录下。

4） 包内共享常量：即在当前包下单独的 constant 目录下。

5） 类内共享常量：直接在类内部 private static final 定义。

1. 【推荐】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;

}

public int getSeq() {

return seq;

}

}

(三) 代码格式

1. 【强制】如果是大括号内为空，则简洁地写成{}即可，大括号中间无需换行和空格；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；表示终止的右大括号后必须换行。

1. 【强制】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；而左

大括号前需要空格。详见第 5 条下方正例提示。

反例：if (空格 a == b 空格)

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。

说明：运算符包括赋值运算符=、逻辑运算符&&、加减乘除符号等。

解释：小时候第一次看马老师视频，才知道要加运算符两边的空格这个东西。所以花钱学东西很值得，如果花1万，马上知道这个知识点，不花这个钱，有可能2 3年后知道，损失的就不止1万了。

1. 【强制】采用 4 个空格缩进，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。

正例： （涉及 1-5 点）

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 {

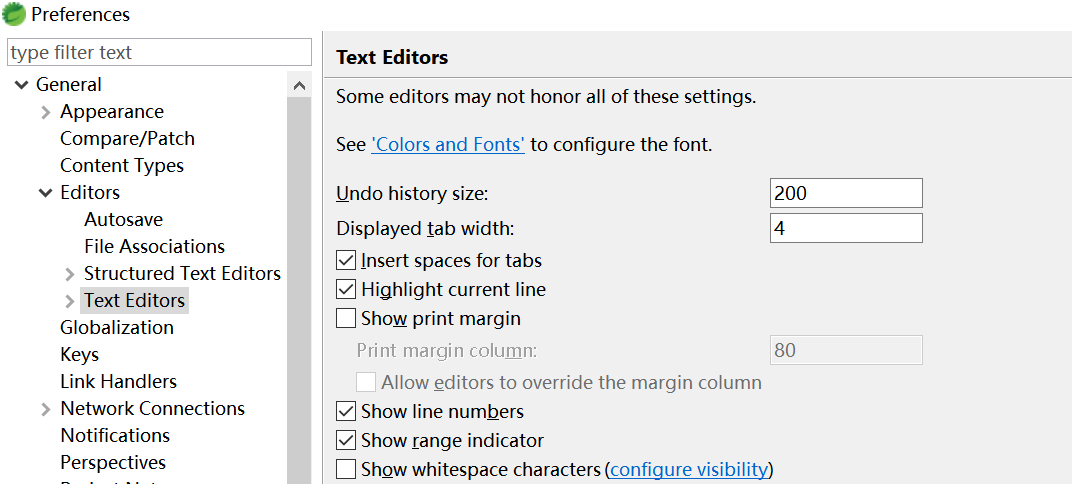
System.out.println("ok");

// 在右大括号后直接结束，则必须换行

}

}

解释：



1. 【强制】注释的双斜线与注释内容之间有且仅有一个空格。

正例：

// 这是示例注释，请注意在双斜线之后有一个空格

String param = new String();

1. 【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。

正例：

long first = 1000000000000L;

int second = (int)first + 2;

1. 【强制】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：

1）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。

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）方法调用中的多个参数需要换行时，在逗号后进行。

5）在括号前不要换行，见反例。

正例：

StringBuilder sb = new StringBuilder();

// 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("Jack").append("Ma")...

.append("alibaba")...

.append("alibaba")...

.append("alibaba");

反例：

StringBuilder sb = new StringBuilder();

// 超过 120 个字符的情况下，不要在括号前换行

sb.append("Jack").append("Ma")...append

("alibaba");

// 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);

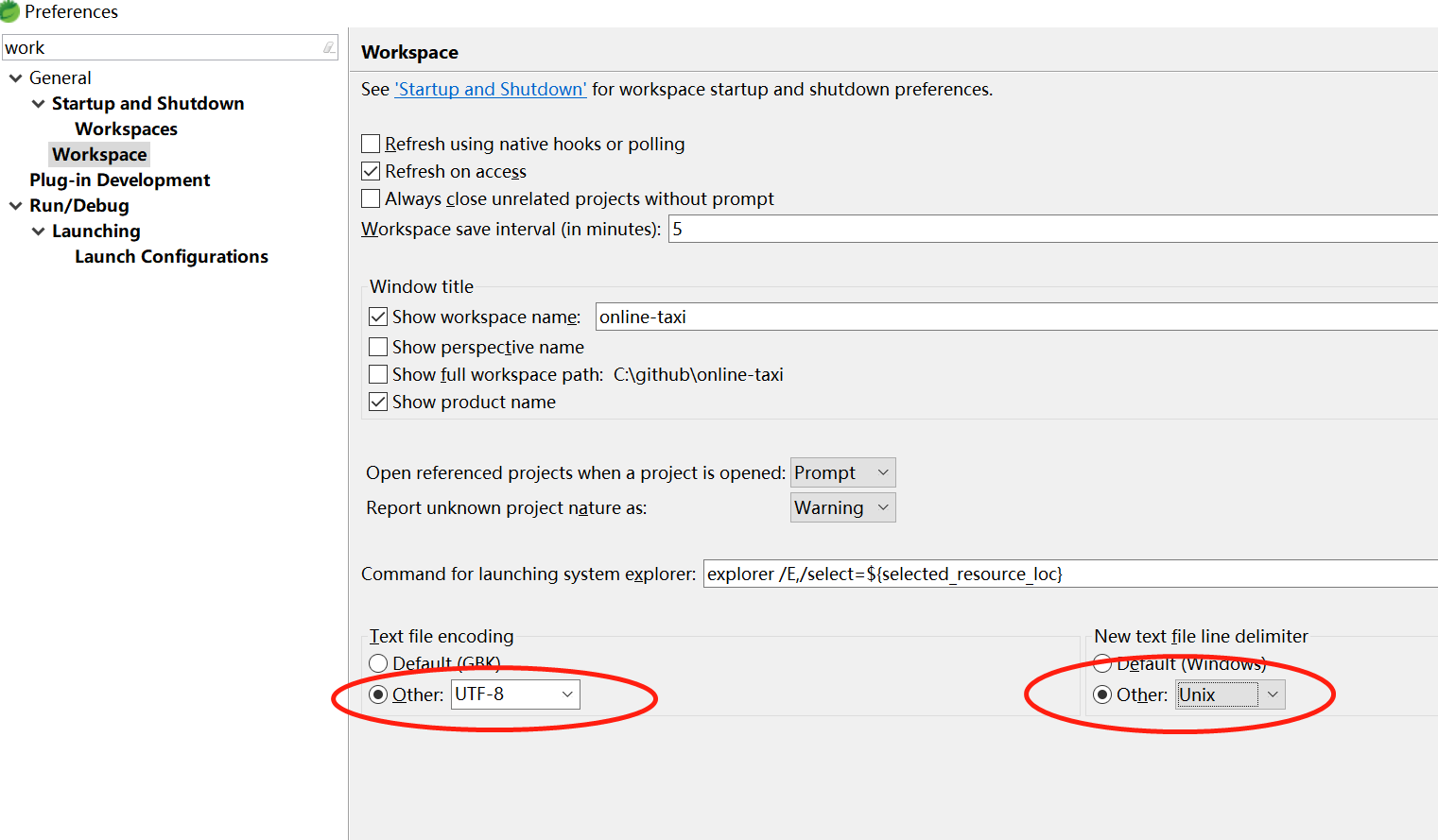
1. 【强制】方法参数在定义和传入时，多个参数逗号后边必须加空格。

正例：下例中实参的 args1，后边必须要有一个空格。

method(args1, args2, args3);

10. 【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。

解释：很多windows格式会有问题，比如记事本开头添加0xefbbbf（十六进制）的字符。会导致文本问题。



11. 【推荐】单个方法的总行数不超过 80 行。

说明：除注释之外的方法签名、左右大括号、方法内代码、空行、回车及任何不可见字符的总行数不超过

1. 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。

解释：不仅仅写代码，生活中方方面面也一样，绿叶就当好绿叶，否则惹人烦。比如重要会议高层讲话，一个低层员工总插嘴，那就很可能会被单独抽取出来。放到公司外面。

12. 【推荐】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。

正例：

int one = 1;

long two = 2L;

float three = 3F;

StringBuilder sb = new StringBuilder();

说明：增加 sb 这个变量，如果需要对齐，则给 one、two、three 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。

13. 【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。

说明：任何情形，没有必要插入多个空行进行隔开。

解释：点到为止，不要过分。否则会激发逆反情绪，小时候玩CS，人已经被打死了，就不要一直突突了，小时候在网吧有一次因为游戏中的鞭尸发生肢体冲突。

(四) OOP 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。

解释：不是说语法错，而是增加jvm成本，每增加一个对象变量，new一个对象需要分配内存空间，增加无谓的内存消耗。调用时会去堆中检索对象，用类名访问只需要去方法区中找，减少了堆中检索的成本。

1. 【强制】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。

解释：jdk5.0才有这个注解，可以帮助我们做数据检测，上例中的O和0眼睛看不出来。

1. 【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。

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）

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

解释：可变参数本质上就是一个数组（经编译器编译后转化为对应类型的数组）。因为如果遇到此种业务，绝对会定下类型，而不是Object，可变参数会增加jdk编译成本，会将它作为一个数组转换，增加编译成本，运行时也会预留内存空间，额外增加运行空间。

1. 【强制】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。

解释：因为现在微服务比较多，有dubbo，rpc调用，spring cloud rest（特例，bean参数不同）调用，如果方法签名改了，会对调用者造成麻烦。我们可以用重写，重载的方式进行扩展。扩展接口升级url中加上版本号v1,v2等。但是原来使用的接口不要停。

1. 【强制】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。

解释：如果有可替代的，不要用过时的。肯定是有问题，才会声明过时。没人喜欢过期的食物，为什么编程时就喜欢用过期的方法，而就不想着用新方法呢？

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。

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）。

解释：字面值做调用方。避免空指针。看代码：oop包下EqualsTest

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。

解释：面试大坑，Integer intA = 3,intB =3 ,用==判断是不是true？不是基本类型，都用equals。equal比较内容。看代码oop.EqualsTest

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《马士兵Java基础课程》。

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");

}

解释：在10进制下，我们将1分成10份，还能表示0.1，0.2等，但表示1/3，只能无限接近，永远达不到。所以精度不够。完全相等是不可能的。

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。

解释：门当户对，否则匹配不上出问题。天天吵架调bug，太费精力。

10. 【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);

解释：官方注释说了unpredictable，你以为的你以为的不是你以为的。BigDecimal(0.1)这货实际上等于0.1000000000000000055511151231257827021181583404541015625，因为准确的来说0.1本身不能算是一个double（其实0.1不能代表任何一个定长二进制分数）。去EqualsTests看源码 注释

11. 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。

说明：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。

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。

解释：局部变量大多数用于计算，基本类型有优势，不需要拆箱装箱，效率高。属性用包装类型：因为有可能序列化。看代码，当用Integer做参数时，num为空时，接受到的是null，由使用者注意，但 为int时收到的是0。如果使用者由于没有npe忘了，上线会出问题。

12. 【强制】定义 DO/DTO/VO 等 POJO 类时，不要设定任何属性默认值。

反例：POJO 类的 createTime 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。

解释：

DO:data object 和数据库表对应，可以1个do对应多个表，。

Dto：trans 转换 dao输出的对象，do和dto不绝对一样。Dto可以对数据做额外的处理，do整理后需要向上传递时用。

Vo：渲染视图。Vo和do有时候也相同。

当do用于页面渲染用分页时，需要vo，do做一些特殊处理时，用dto。

pojo：属性，set，get，object方法，没有业务含义。

Pojo是纯净的。

设置默认值，会让错误不好排查。

13. 【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。

说明：注意 serialVersionUID 不一致会抛出序列化运行时异常。

解释：用了版本号，如果改了版本号，有地方有序列化的对象，在反序列化时就会报错。如果抛弃原来所有的，全部进行升级，无所谓了。Io 文件，内存，网络。反序列化 会读取流中的 版本号，如果版本号和类中版本号不一致，会报错。也可以让所以类版本号有的都是1L，就像2个人同一个身份证号，正常活没问题。Hashcode方法，返回0也行，但是不建议。看代码：oop.SerializeTest

1. 【强制】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。

解释：为了逻辑清晰，不要让孩子快出生的时候，干些他成为独立的人之后才能干的活。先让他生出来再说，别着急。可以在init上加上，@PostConstruct注解，在构造方法执行结束后调用。在服务发现DiscoveryClient中构造方法执行了业务逻辑，我们不学。国产标准。

15. 【强制】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。

解释：异常处理会用到。

16. 【强制】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。

17. 【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。

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);

解释：看代码：oop.SpiltTest

1. 【推荐】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，此条规则优先于下一条。

解释：构造方法，overload，属性，getter/setter分别放在一起。

19. 【推荐】 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。

说明：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。

解释：方便看代码的人。

20. 【推荐】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，增加排查问题的难度。

反例：

public Integer getData() {

if (condition) {

return this.data + 100;

} else {

return this.data - 100;

}

}

解释：加了业务逻辑，别人会误解。

21. 【推荐】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：下例中，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。

反例：

String str = "start";

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

str = str + "hello";

}

解释：减少对象的构建过程，builder用的时char[]实现。减少堆中不必要对象的创建过程。

22. 【推荐】final 可以声明类、成员变量、方法、以及本地变量，下列情况使用 final 关键字：

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） 不允许被覆写的方法，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。

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

23. 【推荐】慎用 Object 的 clone 方法来拷贝对象。

说明：对象 clone 方法默认是浅拷贝，若想实现深拷贝需覆写 clone 方法实现域对象的深度遍历式拷贝。

解释：clone是浅拷贝。就当没有clone这个方法。Oop.CloneTest

<https://blog.csdn.net/meism5/article/details/90414050>

24. 【推荐】类成员与方法访问控制从严：

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，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。

说明：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 成员方法或成员变量，删除一下，不得手心冒点汗吗？变量像自己的小孩，尽量在自己的视线内，变量作用域太大，无限制的到处跑，那么你会担心的。

解释：最小知道原则。阿里安全条例，关于信息传递也是。

(五) 集合处理

1. 【强制】关于 hashCode 和 equals 的处理，遵循如下规则：

1） 只要覆写 equals，就必须覆写 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 存储的对象必须覆写这两个方法。

3） 如果自定义对象作为 Map 的键，那么必须覆写 hashCode 和 equals。

说明：String 已覆写 hashCode 和 equals 方法，所以我们可以愉快地使用 String 对象作为 key 来使用。

解释：

比特币:Collision resistance:哈希碰撞，哈希碰撞不可避免，因为输入空间远远大于输出空间。比如256位的哈希值，输出空间只有2的 256次方。根据鸽笼原理。（鸽笼原理(抽屉原理)就是"如果有五个鸽子笼，养鸽人养了6只鸽子，那么当鸽子飞回笼中后，至少有一个笼子中装有2只鸽子。"）

hashCode规范：

在程序的一次运行中，只要没有修改equals()中用于比较的信息，x.hashCode()总是返回一致的hash值

如果x.equals(y)返回true，则x和y具有相等的hash值

如果x和y具有相等的hash值，x.equals(y)并不一定返回true

map key要排除重复。这个问题主要和映射(Map接口)相关. 我们知道Map接口的类会使用到键(Key)的哈希码, 当我们调用put()/get()方法操作Map容器时, 都是根据Key的哈希码来计算存储位置的, 因此如果我们对哈希码的获取没有相关保证, 就可能会得不到预期的结果.

eqaul方法就可以判断对象相等，但是equal方法里面判断的逻辑会很多，很复杂，说白了就是性能不够好。

hashcode方法判断起来，就一行简单的代码，执行效率高，但是hashcode 有一个致命问题就是两个对象hash的时候值可能会一样。所以hashcode有时并不是都靠谱，但是如果hashcode不一样，两个对象肯定不一样。

正确的对比过程应该是这样：先用hash函数对比，如果结果一样，再用equal方法对比判断两个对象是否相等；如果结果不一样，直接判定这两个对象不是同一个。

2.【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。

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。

解释：sublist重置了下标规范。看代码collector.SubListTest。SubList1，

Sub可以影响源集合，但源集合不影响外面。

1. 【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。

解释：怕程序员代码写hign了，忘了。警钟长鸣。镜像不能修改，就像不能做伪证，原来是啥就是啥。Collector.MapTest中

1. 【强制】Collections 类返回的对象，如：emptyList()/singletonList()等都是 immutable

list，不可对其进行添加或者删除元素的操作。

反例：如果查询无结果，返回 Collections.emptyList()空集合对象，调用方一旦进行了添加元素的操作，就会触发 UnsupportedOperationException 异常。

解释：看Collections 的emptyList源码，因为返回的是final类型。

1. 【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。

解释：父集合 1 2 3 4 5 ，sublist 刚开始用 4 5下标分布式3 4，结果父集合删掉了123，45往前挪。Sublis访问会报错。看代码collector.SubListTest.subList2

1. 【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

致、长度为 0 的空数组。

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

* ClassCastException 错误。

正例：List<String> list = new ArrayList<>(2);

list.add("guan");

list.add("bao");

String[] array = list.toArray(new String[0]);

说明：使用 toArray 带参方法，数组空间大小的 length：

1） 等于 0，动态创建与 size 相同的数组，性能最好。

1. 大于 0 但小于 size，重新创建大小等于 size 的数组，增加 GC 负担。

3） 等于 size，在高并发情况下，数组创建完成之后，size 正在变大的情况下，负面影响与上相同。

4）大于 size，空间浪费，且在 size 处插入 null 值，存在 NPE 隐患。

解释：把内存中的数据直接进行数值转换，会比较快。

数组大小设置不当的时候，不仅回降低性能，还会浪费空间，

当不需要获取新数组的时候，尽量避免数组长度小于集合长度的情况发生，同时尽可能使用带参方法。看源码： **public** <T> T[] toArray(T[] a) {

**if** (a.length < size)

// Make a new array of a's runtime type, but my contents:

**return** (T[]) Arrays.*copyOf*(elementData, size, a.getClass());

System.*arraycopy*(elementData, 0, a, 0, size);

**if** (a.length > size)

a[size] = **null**;

**return** a;

}

发现只要小于size，则效率一样。

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。

解释：看源码：因为会调用c.add(element);;如果为null就会 npe。Collector.ListToArray

@SafeVarargs

**public** **static** <T> **boolean** addAll(Collection<? **super** T> c, T... elements) {

**boolean** result = **false**;

**for** (T element : elements)

result |= c.add(element);

**return** result;

}

会调用c.add.

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。

String[] str = new String[] { "yang", "hao" };

List list = Arrays.asList(str);

第一种情况：list.add("yangguanbao"); 运行时异常。

第二种情况：str[0] = "changed"; 也会随之修改，反之亦然。

解释：看返回ArrayList的源码，其实并没有实现add等方法。此规定是怕程序员写high了。忘了。ListTest看源码发现set正常，而add异常。因为内部类有set没有add。

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>

解释:5.0出现的泛型时一种主观的认为，而不是实际上真正的东西。List<integer> a = List<String>出错。List get(){return List<String>} a = get();不出错。

Extends 读取撑死了是T，插入不知道插的啥，super插入撑死了T，读取不知道是啥。

PECS,的主角是 它本身。它自己往外输出是P，消费是C。

10. 【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。

说明：毕竟泛型是在 JDK5 后才出现，考虑到向前兼容，编译器是允许非泛型集合与泛型集合互相赋值。

反例：

List<String> generics = null;

List notGenerics = new ArrayList(10);

notGenerics.add(new Object());

notGenerics.add(new Integer(1));

generics = notGenerics;

// 此处抛出 ClassCastException 异常

String string = generics.get(0);

解释：GenericsTest

11. 【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。

正例：

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

list.add("1");

list.add("2");

Iterator<String> iterator = list.iterator();

while (iterator.hasNext()) {

String item = iterator.next();

if (删除元素的条件) {

iterator.remove();

}

}

反例：

for (String item : list) {

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

list.remove(item);

}

}

说明：以上代码的执行结果肯定会出乎大家的意料，那么试一下把“1”换成“2”，会是同样的结果吗？

解释：foreach底层用的 iterator实现，游标可能会错乱。一般foreach只允许读，不要写。

代码解释：modCount时ArrayList从AbstractList继承下来的属性，在add,remove等时，会修改，而expectedModCount时ArrayList的内部类ltr的属性，只有在迭代器操作list时才会修改。报异常是由于取下一个值 next()时才报。参见：ForeachTest

12. 【强制】在 JDK7 版本及以上，Comparator 实现类要满足如下三个条件，不然 Arrays.sort， Collections.sort 会抛 IllegalArgumentException 异常。

说明：三个条件如下

1） x，y 的比较结果和 y，x 的比较结果相反。

2） x>y，y>z，则 x>z。

3） 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，交换两个对象判断结果并不互反，不符合第一个条件，在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;

}

};

解释：自反性，传递性，对称性。

Sort种用到了插入排序，快速排序，归并排序，并不是挨着比较，可能下标1和8比，后来2和8比，然后1和2比，1>8,2<8,如果compare方法有问题，不能确定1和2 的大小。就抛异常。

13. 【推荐】集合泛型定义时，在 JDK7 及以上，使用 diamond 语法或全省略。

说明：菱形泛型，即 diamond，直接使用<>来指代前边已经指定的类型。

正例：

// diamond 方式，即<>

HashMap<String, String> userCache = new HashMap<>(16); // 全省略方式

ArrayList<User> users = new ArrayList(10);

14. 【推荐】集合初始化时，指定集合初始值大小。

说明：HashMap 使用 HashMap(int initialCapacity) 初始化。

正例：initialCapacity = (需要存储的元素个数 / 负载因子) + 1。注意负载因子（即 loader factor）默认为 0.75，如果暂时无法确定初始值大小，请设置为 16（即默认值）。

反例：HashMap 需要放置 1024 个元素，由于没有设置容量初始大小，随着元素不断增加，容量 7 次被迫扩大，resize 需要重建 hash 表，严重影响性能。

解释：不指定大小，程序执行时会扩容，影响效率，所以我们要减少内存的膨胀次数。

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

说明：keySet 其实是遍历了 2 次，一次是转为 Iterator 对象，另一次是从 hashMap 中取出 key 所对应的 value。而 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。

正例：values()返回的是 V 值集合，是一个 list 集合对象；keySet()返回的是 K 值集合，是一个 Set 集合对象；entrySet()返回的是 K-V 值组合集合。

解释：entrySert遍历一次，keySet遍历2次。getValue(key),减少迭代次数。因为最终get(key)时，都需要遍历一遍。

16. 【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 |
|  |  |  |  |  |

反例：由于 HashMap 的干扰，很多人认为 ConcurrentHashMap 是可以置入 null 值，而事实上，存储 null 值时会抛出 NPE 异常。

17. 【参考】合理利用好集合的有序性(sort)和稳定性(order)，避免集合的无序性(unsort)和不稳定性(unorder)带来的负面影响。

说明：有序性是指遍历的结果是按某种比较规则依次排列的。稳定性指集合每次遍历的元素次序是一定的。如：ArrayList 是 order/unsort；HashMap 是 unorder/unsort；TreeSet 是 order/sort。

解释：可以降低迭代压力，提高迭代效率。Sort是值存入1，4，2输出124，稳定性：3，5，1，351

18. 【参考】利用 Set 元素唯一的特性，可以快速对一个集合进行去重操作，避免使用 List 的contains 方法进行遍历、对比、去重操作。

解释：list contains方法是迭代比较去 去重。从0开始比，依次比，集合多，会很慢。而set底层是依据hashCode和equals进行判断，

Set<String> set = **new** HashSet<>(arrayList);

System.***out***.println(set.toString());

(六) 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。

说明：资源驱动类、工具类、单例工厂类都需要注意。

解释：因为不是线程安全，会创建不出真正的单例，有可能一个类有多个对象。

看代码：安全且在多线程情况下能保持高性能。thread.SingletonTest

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。

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。

解释：就像我们使用各种连接池一样，jdbc连接池等。池化很重要。tcp和http一个长连接好处：效率高，减少了创建和销毁链接，http每次都得重新建立连接。

4.【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和*newScheduledThreadPool*

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。

解释：看代码。thread包下ExecutorsTest。点进去：newFixedThreadPool，发现其实Executor都是基于ThreadPoolExecutor创建的，只不过传入不同的参数，

参数含义如下：

corePoolSize：线程池中核心线程数量。

maximumPoolSize：线程池最大线程数量。

keepAliveTime：超出核心线程数量时，多出的空闲线程的存活时间。

unit: keepAliveTime的单位。

workQueue：任务队列，当没有空闲线程时，任务存储在该队列。

threadFactory：线程工厂，被用于创建线程。一般使用Executors.defaultThreadFactory()即可。若想清楚的了解自己任务的线程，可以利用该工厂定制化线程。

handler：拒绝策略，当任务太多，如何拒绝任务。

其他参数容易理解：重点：workQueue和handler。

有界队列：ArrayBlockingQueue

创建时必须指定容量。当有新任务时，如果即将使用线程数量 < corePoolSize,则会创建新的线程执行，如果 > corePoolSize,则会将新的任务加入到任务队列中。当任务队列已满，如果线程数量<maximumPoolSize,则创建新的线程执行任务，否则执行拒绝策略。

无界队列：LinkedBlockingQueue

该队列没有界限，除非是系统资源耗尽，否则它不会出现入队失败情况。当新的任务提交，如果使用线程数量<corePoolSize,线程池会创建新的线程执行任务，当线程数量=corePoolSize时并且没有空闲线程时，任务就会添加到任务队列中。像newFixedThreadPool()、newSingleThreadExecutor()就是使用的此队列。

直接提交队列：SynchronousQueue

该队列没有容量，也就是说它不存储任务。当有新的任务来时，如果没有空闲的线程，则会创建新的线程，如果线程数量已经达到最大值，就会执行拒绝策略。newCachedThreadPool()就是使用该队列，使用该队列时，应该把最大线程数量设置尽量大，否则很容易就会执行拒绝策略，newCachedThreadPool()就设置的为Integer.MAX\_VALUE。

优先任务队列:PriorityBlockingQueue

控制任务的执行顺序，它属于无界队列，但是比较特殊。不论是无界还是有界队列都是按照先进先出的顺序处理任务，而PriorityBlockingQueue则可以根据任务自身的优先级顺序执行，它总能确保高优先级的任务先执行。

拒绝策略:

AbortPolicy:直接抛出异常，阻止应用正常工作。（默认使用该策略）

CallerRunsPolicy:不会真正丢弃任务，当线程池未关闭，该策略直接在调用者线程中运行当前丢弃任务。

DiscardPolicy:丢弃无法处理的任务，不做任何处理。

DiscardOldestPolicy:丢弃最老的任务，也就是即将执行的任务，并尝试提交当前任务。

也可以自定义拒绝策略。

newFixedThreadPool和newSingleThreadExecutor点参数队列进去：

**public** LinkedBlockingQueue() {

**this**(Integer.***MAX\_VALUE***);

}

**public** **static** ExecutorService newCachedThreadPool() {

**return** **new** ThreadPoolExecutor(0, Integer.***MAX\_VALUE***,

60L, TimeUnit.***SECONDS***,

**new** SynchronousQueue<Runnable>());

}

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() {

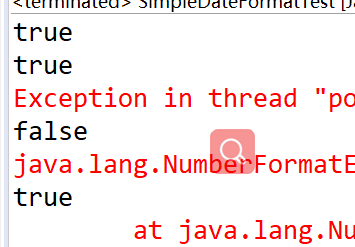
return new SimpleDateFormat("yyyy-MM-dd");

}

};

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。

解释：看代码：SimpleDateFormatTest



代码中有两种方式，不安全和安全。

ThreadLocal 是线程的局部变量， 是每一个线程所单独持有的，其他线程不能对其进行访问。

当使用ThreadLocal维护变量的时候 为每一个使用该变量的线程提供一个独立的变量副本，即每个线程内部都会有一个该变量，这样同时多个线程访问该变量并不会彼此相互影响，因此他们使用的都是自己从内存中拷贝过来的变量的副本， 这样就不存在线程安全问题，也不会影响程序的执行性能。

但是要注意，虽然ThreadLocal能够解决上面说的问题，但是由于在每个线程中都创建了副本，所以要考虑它对资源的消耗，比如内存的占用会比不使用ThreadLocal要大。

1. 【强制】必须回收自定义的 ThreadLocal 变量，尤其在线程池场景下，线程经常会被复用，如果不清理自定义的 ThreadLocal 变量，可能会影响后续业务逻辑和造成内存泄露等问题。

尽量在代理中使用 try-finally 块进行回收。

正例：

objectThreadLocal.set(userInfo);

try {

// ...

} finally {

objectThreadLocal.remove();

}

解释：simpleDateFormatSalfe.remove();

1. 【强制】高并发时，同步调用应该去考量锁的性能损耗。能用无锁数据结构，就不要用锁；能锁区块，就不要锁整个方法体；能用对象锁，就不要用类锁。

说明：尽可能使加锁的代码块工作量尽可能的小，避免在锁代码块中调用 RPC 方法。

解释：调用rpc会时间长，怕导致死锁。

8、【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。

解释：讨厌的锁，能不用就不用。不按照顺序，会造成互相等待死锁。

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();

}

解释：保持锁的完整性。网约车中会讲到这部分，有分布锁讲解。比如锁订单号出现了异常，下个线程进来正常执行完，释放锁，结果把上次的订单锁释放掉。<https://www.cnblogs.com/waterystone/p/4920797.html>

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();

}

}

11. 【强制】并发修改同一记录时，避免更新丢失，需要加锁。要么在应用层加锁，要么在缓存加锁，要么在数据库层使用乐观锁，使用 version 作为更新依据。

说明：如果每次访问冲突概率小于 20%，推荐使用乐观锁，否则使用悲观锁。乐观锁的重试次数不得小于

1. 次。

解释：概率计算，看资源和访问资源的情况。

重试：<https://blog.csdn.net/HXNLYW/article/details/94558449>

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。

解释：在我们讲eureka源码时可以看到这种用法。com.netflix.discovery.DiscoveryClient 1267行

private void initScheduledTasks() {

点scheduler，private final ScheduledExecutorService scheduler;

13. 【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。

解释：<https://blog.csdn.net/qq_34337272/article/details/81072874>

14. 【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。

解释：看代码CountDownLatchTest

15. 【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。

解释：JDK7之后，可以使用ThreadLocalRandom来获取随机数。

**public** Random(**long** seed) {

**if** (getClass() == Random.**class**)

**this**.seed = **new** AtomicLong(*initialScramble*(seed));

**else** {

// subclass might have overriden setSeed

**this**.seed = **new** AtomicLong();

setSeed(seed);

}

}

**private** **static** **long** initialScramble(**long** seed) {

**return** (seed ^ ***multiplier***) & ***mask***;

}

而***multiplier*** & ***mask都是static的，所以共享。***

***seed含义见代码：***RandomTest，seed一样，一组产生的随机数一样。

16. 【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...

}

解释：1.线程的可见性

2.防止指令重排

感兴趣 Java基础课。

17. 【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）。

18. 【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。

19. 【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。