编程规约

# 命名规约

1. 所有编程相关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 / $Object / name\_ / name$ / Object$

1. 所有编程相关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，即使纯拼音命名方式也要避免采用。

反例： DaZhePromotion [打折] / getPingfenByName() [评分] / int 变量 = 3;

正例： ali / alibaba / taobao / cainiao / aliyun / youku / hangzhou 等国际通用的名称，可视为英文。

1. 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 / DTO / VO / DAO 等。

正例：MarcoPolo / UserDO / XmlService / TcpUdpDeal / TaPromotion

反例：macroPolo / UserDo / XMLService / TCPUDPDeal / TAPromotion

1. 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. 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

反例： MAX\_COUNT

1. 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. 中括号是数组类型的一部分，数组定义如下：String[]

反例：请勿使用 String args[]的方式来定义

1. POJO 类中的任何布尔类型的变量，都不要加 is，否则部分框架解析会引起序列化错误。

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

1. 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.mpp.util、类名为 MessageUtils（此规则参考 spring的框架结构）

1. 杜绝完全不规范的缩写，避免望文不知义。

反例：<某业务代码>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. 如果使用到了设计模式，建议在类名中体现出具体模式。

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思想。

正例：

public class OrderFactory;

public class LoginProxy;

public class ResourceObserver;

1. 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 f();

接口基础常量表示：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。

1. 接口和实现类的命名有两套规则：

1）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。

2） 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。

1. 枚举类名建议带上 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常量类，且构造方法被默认强制是私有。

正例：枚举名字：DealStatusEnum；成员名称：SUCCESS / UNKOWN\_REASON。

1. **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。

3） 获取统计值的方法用 count 做前缀。

4） 插入的方法用 save（推荐）或 insert 做前缀。

5） 删除的方法用 remove（推荐）或 delete 做前缀。

6） 修改的方法用 update 做前缀。

B) 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。

# 常量定义

1. 不允许出现任何魔法值（即未经定义的常量）直接出现在代码中。

反例： String key="Id#taobao\_"+tradeId；

cache.put(key, value);

1. 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。

说明：Long a = 2l; 写的是数字的 21，还是 Long 型的 2?

1. **不要使用一个常量类维护所有常量，应该按常量功能进行归类，分开维护。如：缓存相关的常量放在类：CacheConsts 下；系统配置相关的常量放在类：ConfigConsts 下。**

说明：大而全的常量类，非得 ctrl+f 才定位到修改的常量，不利于理解，也不利于维护。

1. 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 const 目录下。

2） 应用内共享常量：放置在一方库的 modules 中的 const 目录下。

反例：易懂变量也要统一定义成应用内共享常量，两位攻城师在两个类中分别定义了表示“是”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，导致产生线上问题。

3） 子工程内部共享常量：即在当前子工程的 const 目录下。

4） 包内共享常量：即在当前包下单独的 const 目录下。

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

1. 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，下面正例中的数字就是延伸信息，表示星期几。

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

# 格式规约

1. **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. 方法体内的执行语句组、变量的定义语句组、不同的业务逻辑之间或者不同的语义之间插入一个空行。相同业务逻辑和语义之间不需要插入空行。

说明：没有必要插入多行空格进行隔开。

# OOP 规约

1. **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. 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。

1. 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）

1. **所有的相同类型的包装类对象之间值的比较，全部使用 equals 方法比较。**

说明：对于 Integer var=?在-128 至 127 之间的赋值，Integer 对象是在 IntegerCache.cache产生，会复用已有对象，这个区间内的 Integer 值可以直接使用==进行判断，但是这个区间之外的所有数据，都会在堆上产生，并不会复用已有对象，这是一个大坑，推荐使用 equals 方法进行判断。

1. **关于基本数据类型与包装数据类型的使用标准如下：**

**1） 所有的 POJO 类属性必须使用包装数据类型。**

**2） RPC 方法的返回值和参数必须使用包装数据类型。**

**3） 所有的局部变量推荐使用基本数据类型。**

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：某业务的交易报表上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：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。

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

反例：某业务的 DO 的 gmtCreate 默认值为 new Date();但是这个属性在数据提取时并没有置入具体值，在更新其它字段时又附带更新了此字段，导致创建时间被修改成当前时间。

1. 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。

1. **类内方法定义顺序依次是：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；因为方法信息价值较低，所有 Service 和 DAO 的 getter/setter 方法放在类体最后。

1. **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";

}

说明：反编译出的字节码文件显示每次循环都会 new 出一个 StringBuilder 对象，然后进行append 操作，最后通过 toString 方法返回 String 对象，造成内存资源浪费。

1. final 可提高程序响应效率，声明成 final 的情况：

1） 不需要重新赋值的变量，包括类属性、局部变量。

2） 对象参数前加 final，表示不允许修改引用的指向。

3） 类方法确定不允许被重写。

1. 慎用 Object 的 clone 方法来拷贝对象。

说明：对象的 clone 方法默认是浅拷贝，若想实现深拷贝需要重写 clone 方法实现属性对象的拷贝。

# 集合处理

1. **Map/Set 的 key 为自定义对象时，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. ArrayList 的 subList 结果不可强转成 ArrayList，否则会抛出 ClassCastException异常：java.util.RandomAccessSubList cannot be cast to java.util.ArrayList ;

说明：subList 返回的是 ArrayList 的内部类 SubList，并不是 ArrayList ，而是 ArrayList的一个视图，对于 SubList 子列表的所有操作最终会反映到原列表上。

1. subList 场景中，在高度注意对原集合元素个数的修改，会导致子列表的遍历、增加、删除均产生 ConcurrentModificationException 异常。
2. 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);

说明：使用 toArray 带参方法，入参分配的数组空间不够大时，toArray 方法内部将重新分配内存空间，并返回新数组地址；如果数组元素大于实际所需，下标为[list.size()]的数组元素将被置为 null，其它数组元素保持原值，因此最好将方法入参数组大小定义与集合元素个数一致。

1. 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[] { "a", "b" };

List list = Arrays.asList(str);

第一种情况：list.add("c"); 运行时异常。

第二种情况：str[0]= "gujin"; 那么 list.get(0)也会随之修改。

1. **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);

}

}

说明：这个例子的执行结果会出乎大家的意料，那么试一下把“1”换成“2”，会是同样的结果吗？

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. 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;

}

}

1. 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。

1. 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 | 线程局部安全 |
| 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：很多同学认为 ConcurrentHashMap 是可以置入 null 值。在批量翻译场景中，子线程分发时，出现置入 null 值的情况，但主线程没有捕获到此异常，导致排查困难。

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

1. 利用 Set 元素唯一的特性，可以快速对另一个集合进行去重操作，避免使用 List 的contains 方法进行遍历去重操作。

# 并发处理

1. **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：使用线程池的好处是减少在创建和销毁线程上所花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. 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 Simpledateformatter，官方给出的解释：simple beautiful strong immutable thread-safe。

1. 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。
2. 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. 并发修改同一记录时，避免更新丢失，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 3 次。

1. 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，使用 ScheduledExecutorService 则没有这个问题。
2. 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 各个方法的弊端：

1）newFixedThreadPool 和 newSingleThreadExecutor:

主要问题是堆积的请求处理队列可能会耗费非常大的内存，甚至 OOM。

2）newCachedThreadPool 和 newScheduledThreadPool:

主要问题是线程数最大数是 Integer.MAX\_VALUE，可能会创建数量非常多的线程，甚至 OOM。

1. 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：

public class TimerTaskThread extends Thread {

public TimerTaskThread(){

super.setName("TimerTaskThread"); …

}

1. 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 方法可以执行，避免主线程无法执行至countDown 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. 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()实例。

正例：在 JDK7 之后，可以直接使用 API ThreadLocalRandom，在 JDK7 之前，可以做到每个线程一个实例。

1. 通过双重检查锁（double-checked locking）（在并发场景）实现延迟初始化的优化问题隐患(可参考 The "Double-Checked Locking is Broken" Declaration),推荐问题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（比如反例中修改 helper 的属性声明为 private volatile Helper helper = null;）；

反例：

class Foo {

private Helper helper = null;

public Helper getHelper() {

if (helper == null)

synchronized(this) {

if (helper == null)

helper = new Helper();

}

return helper; }

// other functions and members...

}

1. 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。如果想取回 count++数据，使用如下类实现：AtomicInteger count = new AtomicInteger(); count.addAndGet(1); count++操作如果是JDK8，推荐使用 LongAdder 对象，比 AtomicLong 性能更好（减少乐观锁的重试次数）。
2. 注意 HashMap 的扩容死链，导致 CPU 飙升的问题。
3. ThreadLocal 无法解决共享对象的更新问题，ThreadLocal 对象建议使用 static 修饰。这个变量是针对一个线程内所有操作共有的，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。

# 控制语句

1. **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;**
2. **推荐尽量少用 else， if-else 的方式可以改写成：**

if(condition){

…

return obj;

}

// 接着写 else 的业务逻辑代码;

说明：如果使用要 if-else if-else 方式表达逻辑，请勿超过 3 层，超过请使用状态设计模式。

1. 除常用方法（如 getXxx/isXxx）等外，不要在条件判断中执行复杂的语句，以提高可读性。

正例：

//伪代码如下

InputStream stream = file.open(fileName, "w");

if (stream != null) {

…

}

反例：

if (file.open(fileName, "w") != null)) {

…

}

1. 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）。
2. 接口入参保护，这种场景常见的是用于做批量操作的接口。
3. **方法中需要进行参数校验的场景：**

1） 调用频次低的方法。

2） 执行时间开销很大的方法，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，那得不偿失。

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/API/HTTP 接口。

1. **方法中不需要参数校验的场景：**

1） 极有可能被循环调用的方法，不建议对参数进行校验。但在方法说明里必须注明外部参数检查。

2） 底层的方法调用频度都比较高，一般不校验。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。一般 DAO 层与 Service 层都在同一个应用中，部署在同一台服务器中，所以 DAO 的参数校验，可以省略。

3） 被声明成 private 只会被自己代码所调用的方法，如果能够确定调用方法的代码传入参数已经做过检查或者肯定不会有问题，此时可以不校验参数。

# 注释规约

1. 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。

说明：在 IDE 编辑窗口中，javadoc 方式会提示相关注释，生成 javadoc 可以正确输出相应注释；在 IDE 中，工程调用方法时，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。

1. 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。

说明：如有实现和调用注意事项，请一并说明。

1. 所有的类都必须添加创建者信息。
2. 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. 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。
4. 与其“半吊子”英文来注释，不如用中文注释把问题说清楚。专有名词、关键字，保持英文原文即可。

反例：“TCP 连接超时”解释成“传输控制协议连接超时”，理解反而费脑筋。

1. 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。

说明：代码与注释更新不同步，就像路网与导航软件更新不同步一样，如果导航软件严重滞后，就失去了导航的意义。

1. 注释掉的代码尽量要配合说明，而不是简单的注释掉。

说明：代码被注释掉有两种可能性：1）后续会恢复此段代码逻辑。2）永久不用。前者如果没有备注信息，难以知晓注释动机。后者建议直接删掉（代码仓库保存了历史代码）。

1. 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。
2. 好的命名、代码结构是自解释的，注释力求精简准确、表达到位。避免出现注释的一个极端：过多过滥的注释，代码的逻辑一旦修改，修改注释是相当大的负担。

反例：

// put elephant into fridge

put(elephant, fridge);

方法名 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，

语义清晰的代码不需要额外的注释。

1. 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。

1） 待办事宜（TODO）:（ 标记人，标记时间，[预计处理时间]）

表示需要实现，但目前还未实现的功能。这实际上是一个 javadoc 的标签，目前的javadoc 还没有实现，但已经被广泛使用。只能应用于类，接口和方法（因为它是一个 javadoc标签）。

2） 错误，不能工作（FIXME）:（标记人，标记时间，[预计处理时间]）

在注释中用 FIXME 标记某代码是错误的，而且不能工作，需要及时纠正的情况。

# 其它

1. **在使用正则表达式时，利用好其预编译功能，可以有效加快正则匹配速度。**

说明：不要在方法体内定义：Pattern pattern = Pattern.compile(规则);

1. **避免用 Apache Beanutils 进行属性的 copy。**

说明：Apache BeanUtils 性能较差，可以使用其他方案比如 Spring BeanUtils, CglibBeanCopier。

1. 注意 Math.random() 这个方法返回是 double 类型，注意取值范围 0≤x<1（能够取到零值，注意除零异常），如果想获取整数类型的随机数，不要将 x 放大 10 的若干倍然后取整，直接使用 Random 对象的 nextInt 或者 nextLong 方法。
2. 获取当前毫秒数：System.currentTimeMillis(); 而不是 new Date().getTime();

说明：如果想获取更加精确的纳秒级时间值，用 System.nanoTime。在 JDK8 中，针对统计时间等场景，推荐使用 Instant 类。

1. 任何数据结构的使用都应限制大小。

说明：这点很难完全做到，但很多次的故障都是因为数据结构自增长，结果造成内存被吃光。

1. 对于“明确停止使用的代码和配置”，如方法、变量、类、配置文件、动态配置属性等要坚决从程序中清理出去，避免造成过多垃圾。清理这类垃圾代码是技术气场，不要有这样的观念：“不做不错，多做多错”。