白话阿里巴巴Java开发手册高级篇

不久前，阿里巴巴发布了《阿里巴巴Java开发手册》，总结了阿里巴巴内部实际项目开发过程中开发人员应该遵守的研发流程规范，这些流程规范在一定程度上能够保证最终的项目交付质量，通过在时间中总结模式，并推广给广大开发人员，来避免研发人员在实践中容易犯的错误，确保最终在大规模协作的项目中达成既定目标。

无独有偶，笔者去年在公司里负责升级和制定研发流程、设计模板、设计标准、代码标准等规范，并在实际工作中进行了应用和推广，收效颇丰，也总结了适合支付平台的技术规范，由于阿里巴巴Java开发手册本身定位为规约和规范，语言简单、精炼，没有太多的解读和示例，有些条款对于一般开发人员理解起来比较困难，本文借着阿里巴巴发布的Java开发手册，详细解读Java平台下开发规范和标准的制定和实施，强调那些在开发过程中需要重点关注的技术点，特别是解决某类已识别问题的模式和反模式。

异常处理

【强制】Java 类库中定义的一类 RuntimeException 可以通过预先检查进行规避，而不应该通过 catch 来处理，比如: IndexOutOfBoundsException，NullPointerException等等。

说明: 无法通过预检查的异常除外，如在解析一个外部传来的字符串形式数字时，通过 catch NumberFormatException 来实现。

正例: if (obj != null) {…}

反例: try { obj.method() } catch (NullPointerException e) {…}

白话：

判空是一个永恒的话题，只要你不确定变量是否为空，都应该判空，否则后患无穷。

【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。

白话：

禁止使用异常来封装业务逻辑，业务异常应该用错误码来表示，系统异常则使用Java原生异常。

异常处理是通过异常表查询来实现的，肯定没有跳转语句性能高。

【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。

白话：

做事要直切主题，不能一概而论。

不能简单的catch Throwable，然后打印日志，这是不负责任的表现，应该有针对的抓住和处理异常。

【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。

白话：

禁止吃掉异常，吃掉异常就是捕获异常什么都没做，也没抛出，这也是不负责任的表现。

是在不能处理的异常抛出去，但是得让使用方知道这种情况，这是编程提供方契约的一种方式。

【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。

白话：

我们基本采用声明式事务，出现异常需要回滚的情况，建议继续抛出异常让声明式事务自动回滚，不建议代码中手工控制事务。

【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。 说明:如果 JDK7 及以上，可以使用 try-with-resources 方式。

白话：

永恒的资源关闭原则。

【强制】不能在 finally 块中使用 return，finally 块中的 return 返回后方法结束执行，不会再执行 try 块中的 return 语句。

白话：

确实会覆盖try块里面的return语句。

思维不混乱的话，没人会把return语句写在finally语句里。

【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。

如果预期对方抛的是绣球，实际接到的是铅球，就会产生意外情况。

白话：

异常处理后，让异常变得更小，而不是变大，大而化小，小而化了。

【推荐】方法的返回值可以为 null，不强制返回空集合，或者空对象等，必须添加注释充分 说明什么情况下会返回 null 值。调用方需要进行 null 判断防止 NPE 问题。

说明: 本规约明确防止 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，运行时异常等场景返回 null 的情况。

白话：

如前面所说，只要不确定的变量，一定要判空，别自找麻烦。

【推荐】防止 NPE，是程序员的基本修养，注意 NPE 产生的场景:

- 返回类型为包装数据类型，有可能是null，返回int值时注意判空。

  反例: public int f() { return Integer 对象}; 如果为 null，自动解箱抛 NPE。

- 数据库的查询结果可能为null。
- 集合里的元素即使isNotEmpty，取出的数据元素也可能为null。
- 远程调用返回对象，一律要求进行NPE判断。
- 对于Session中获取的数据，建议NPE检查，避免空指针。
- 级联调用obj.getA().getB().getC();一连串调用，易产生NPE。

正例: 可以使用 JDK8 的 Optional 类来防止 NPE 问题。

白话：

判空，判空，缓存的数据，别人的数据，都要判空。

【推荐】在代码中使用“抛异常”还是“返回错误码”，对于公司外的 http/api 开放接口必须使用“错误码”;而应用内部推荐异常抛出;跨应用间 RPC 调用优先考虑使用 Result 方式，封装 isSuccess、“错误码”、“错误简短信息”。

说明: 关于 RPC 方法返回方式使用 Result 方式的理由:

使用抛异常返回方式，调用方如果没有捕获到就会产生运行时错误。

如果不加栈信息，只是new自定义异常，加入自己的理解的error message，对于调用端解决问题的帮助不会太多。如果加了栈信息，在频繁调用出错的情况下，数据序列化和传输的性能损耗也是问题。

白话：

业务异常使用Result模式，系统异常用Java原生异常。

RPC建议使用Result模式，不想让一个异常在系统间跳来跳去的，异常是包含调用栈的。

【推荐】定义时区分unchecked/checked 异常，避免直接使用RuntimeException抛出，更不允许抛出 Exception 或者 Throwable，应使用有业务含义的自定义异常。推荐业界已定义 过的自定义异常，如:DAOException / ServiceException 等。

白话：

不要所有异常都集成自Runtime异常，希望调用方处理的异常一定用checked异常。

【参考】避免出现重复的代码(Don’t Repeat Yourself)，即DRY原则。

说明: 随意复制和粘贴代码，必然会导致代码的重复，在以后需要修改时，需要修改所有的副 本，容易遗漏。必要时抽取共性方法，或者抽象公共类，甚至是共用模块。

正例: 一个类中有多个 public 方法，都需要进行数行相同的参数校验操作，这个时候请抽取:

```
private boolean checkParam(DTO dto) {...}
```

白话：

如果不知道DRY原则，但是回顾你以前写的代码都是这样写的，那么恭喜你，你是个好程序员，也为你发愁…

并发处理

【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。

白话：

如果延迟加载实现的单例需要并发控制；如果初始化的时候new单例对象，本身是线程安全的，取得实例方法不需要同步。

【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:

```
public class TimerTaskThread extends Thread {
    public TimerTaskThread() {
        super.setName("TimerTaskThread");
        ...
    }
}
```

白话：

写代码的时候就要想到查bug的时候要用到什么信息，然后决定如何命名、打印日志等。

【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: 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。

白话：

一个是使用线程池缓存线程可以提高效率，另外线程池帮我们做了管理线程的事情，提供了优雅关机、interrupt等待IO的线程，饱和策略等功能。

【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:

- 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。
- 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。

白话：

线程池如果没有限制最大数量，线程池撑开的时候，由于内存不够或者系统配置的最大线程数超出，都会产生oom: unalbe to create native thread。

一个组件的核心参数最好要显式的传入，不要默认，就像你交给属下一个任务，任务的目标、原则、时间点、边界都要明确，不能模糊处理一样，免得扯皮。

【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 Simpledateformatter，官方给出的解释: simple beautiful strong

immutable thread-safe。

白话：

记住，打死你，我也不会把SimpleDateFormat共享到类中。

【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。

白话：

优先无锁，不用锁能解决的一定不要用锁，即使用锁也要控制粒度，越细越好。

【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。

白话：

解决死锁的方法：按顺序锁资源、超时、优先级、死锁检测等。

可参考哲学家进餐问题学习更深入的并发机制。

【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 3 次。

白话：

状态流转、维护可用余额等最好直接利用数据库的行级锁，不需要显式的加锁。

【强制】多线程并行处理定时任务时，Timer 运行多个 TimerTask 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。

白话：

线程执行体、任务最上层等一定要抓住Throwable并进行相应的处理，否则会使线程终止。

【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 方法可以执行，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。

白话：

请在try…finally语句里执行countDown方法，与关闭资源类似。

【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()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。

白话：

可以把Random放在ThreadLocal里，只在本线程中使用。

【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。

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...
}
```


白话：

网上对双检锁有N多讨论，这里很负责任的告诉大家，只要不是特别老的JDK版本(1.4以下)，双检锁是没问题的。

【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。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)。

白话：

volatile只有内存可见性语义，synchronized有互斥语义，一写多读使用volatile就可以，多写就必须使用synchronized，fetch-mod-get也必须使用synchronized。

【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，在开发过程中注意规避此风险。

白话：

开发程序的时候要预估使用量，根据使用量来设置初始值。

resize需要重建hash表，严重影响性能，会让程序产生长尾的响应时间。

【参考】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。

白话：

ThreadLocal实际上是一个从线程ID到变量的Map，每次取得ThreadLocal变量，实际上是先取得当前线程ID，再用当前线程ID取得关联的变量。

ThreadLocal使用了WeakHashMap，在key被回收的时候，value也被回收了，不用担心内存泄露。

日志规约

【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。

```
java
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
private static final Logger logger =
LoggerFactory.getLogger(Abc.class);
```

白话：

使用slf4jAPI更爽更清新。

通过占位符的方式不但代码清晰，对象的toString等方法也是根据日志等级来调用的。

【强制】日志文件推荐至少保存 15 天，因为有些异常具备以“周”为频次发生的特点。

白话：

其实需要更长，有的线上事故复盘周期更长，需要更长的日志保存。

这其实不是开发的职责，应该构建大数据日志系统，比如：ELK等。

【强制】应用中的扩展日志(如打点、临时监控、访问日志等)命名方式:
`appName_logType_logName.log` 。

logType: 日志类型，推荐分类有stats/desc/monitor/visit 等;

logName: 日志描述。

这种命名的好处: 通过文件名就可知道日志文件属于什么应用，什么类型，什么目的，也有利于归类查找。

正例：mppserver 应用中单独监控时区转换异常，如：`mppserver_monitor_timeZoneConvert.log`

说明: 推荐对日志进行分类，错误日志和业务日志尽量分开存放，便于开发人员查看，也便于 通过日志对系统进行及时监控。

白话：

逻辑上分开更利于日志管理。

性能上，机械硬盘如果是单文件写可以一定程度利用磁盘顺序写提高性能。

【强制】对 trace/debug/info 级别的日志输出，必须使用条件输出形式或者使用占位符的方 式。

说明: logger.debug(“Processing trade with id: ” + id + ” symbol: ” + symbol); 如果日志级别是 warn，上述日志不会打印，但是会执行字符串拼接操作，如果 symbol 是对象， 会执行 toString()方法，浪费了系统资源，执行了上述操作，最终日志却没有打印。

正例: (条件)

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

正例: (占位符)

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

白话：

一定不要用字符串相加，一定要用占位符。

【强制】避免重复打印日志，浪费磁盘空间，务必在 log4j.xml 中设置additivity=false。

正例: `<logger name="com.taobao.dubbo.config" additivity="false">`

白话：

日志需要CPU处理，缓存的时候需要占用内存，打印过程中要占用IO带宽，存储到磁盘又需要存储空间，要绿色环保。

【强制】异常信息应该包括两类信息: 案发现场信息和异常堆栈信息。如果不处理，那么通过 关键字 throws 往上抛出。

正例: `logger.error(各类参数或者对象 toString + "_" + e.getMessage(), e);`

白话：

打印日志一定要包含环境，否则找bug的时候日志对不上，勤奋爱干活的人一下就听懂我在说啥了。

【推荐】谨慎地记录日志。生产环境禁止输出 debug 日志;有选择地输出 info 日志;如果使 用 warn 来记录刚上线时的业务行为信息，一定要注意日志输出量的问题，避免把服务器磁盘撑爆，并记得及时删除这些观察日志。

说明: 大量地输出无效日志，不利于系统性能升，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?

白话：

我常常和小伙伴们说，写代码的时候就要想到查bug的时候怎么查，需要哪些日志，打印日志只需要打印核心内容，不要随便就把对象json序列化打印出来，如果是列表会很大。

【参考】可以使用 warn 日志级别来记录用户输入参数错误的情况，避免用户投诉时，无所适 从。注意日志输出的级别，error 级别只记录系统逻辑出错、异常等重要的错误信息。如非必要，请不要在此场景打出 error 级别。

白话：

合理利用warn级别日志，error级别日志最重要，理想情况下生产上产生的error和warn日志开发要定期的梳理。

安全规约

【强制】隶属于用户个人的页面或者功能必须进行权限控制校验。

说明: 防止没有做水平权限校验就可随意访问、修改、删除别人的数据，比如查看他人的私信内容、修改他人的订单。

白话：

面向用户的所有服务都要有权限校验。

后端服务没有权限校验，也要有服务化平台下的调用权限管理。

【强制】用户敏感数据禁止直接展示，必须对展示数据脱敏。

说明: 查看个人手机号码会显示成:158****9119，隐藏中间 4 位，防止隐私泄露。

白话：

除了手机号，在金融领域会有更多的敏感信息。

防泄露必须加密，防篡改必须签名，防抵赖必须非对称签名。

【强制】用户输入的 SQL 参数严格使用参数绑定或者 METADATA 字段值限定，防止 SQL 注入， 禁止字符串拼接 SQL 访问数据库。

白话：

这条一般用代码检查工具都会检查出来。

开发人员千万不要做字符串连接SQL。

【强制】用户请求传入的任何参数必须做有效性验证。

说明:

忽略参数校验可能导致:

- page size过大导致内存溢出。
- 恶意order by导致数据库慢查询。
- 任意重定向。
- SQL注入。
- 反序列化注入。
- 正则输入源串拒绝服务ReDoS.

说明: Java 代码用正则来验证客户端的输入，有些正则写法验证普通用户输入没有问题， 但是如果攻击人员使用的是特殊构造的字符串来验证，有可能导致死循环的结果。

白话：

一般在框架层都要做特殊字符的过滤，包括：大于号、小于号、单引号等。

任何使用集合的时候，输入参数是集合的时候，返回是集合的时候，一定要有条数的限制，不能无限大。

【强制】禁止向 HTML 页面输出未经安全过滤或未正确转义的用户数据。

白话：

系统的入口要堵住特殊字符，入口可能是web界面，也可能是开发的接口。

系统的出口也要堵住特殊字符，出口一般指的是web界面。

【强制】表单、AJAX 提交必须执行 CSRF 安全过滤。

说明: CSRF(Cross-site request forgery) 跨站请求伪造是一类常见编程漏洞。对于存在 CSRF 漏洞的应用/网站，攻击者可以事先构造好 URL，只要受害者用户一访问，后台便在用户 不知情情况下对数据库中用户参数进行相应修改。

白话：

堵住系统的入口！

【强制】在使用平台资源，譬如短信、邮件、电话、下单、支付，必须实现正确的防重放限制， 如数量限制、疲劳度控制、验证码校验，避免被滥刷、资损。

说明: 如注册时发送验证码到手机，如果没有限制次数和频率，那么可以利用此功能骚扰到其它用户，并造成短信平台资源浪费。

白话：

针对用户输入，一定要做防御式编程。

【推荐】发贴、评论、发送即时消息等用户生成内容的场景必须实现防刷、文本内容违禁词过滤等风控策略。

白话：

这个一般是大数据部门提供决策数据，各个业务方埋点。

OOP 规约

【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。

白话：

也不直观，看调用代码看不出来是静态方法，容易误解。

【强制】所有的覆写方法，必须加@Override 注解。

反例: `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。

白话：

Java和C++不一样，C++是在父类先声明虚拟函数子类才覆写，Java是任何方法都能覆写，也可以不覆写，所以覆写不覆写是没有编译器检查的，除非接口中某一个方法完全没有被实现才会编译报错。

【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 User getUsers(String type, Integer... ids)`

白话：

用处不大，可以用重载方法或者数组参数代替。

一般应用在日志的 API 定义上，用于传不定的日志参数。

【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。

白话：

设计时没有考虑周全，需要改造接口，需要通过增加新接口，迁移后下线老接口的方式实现。

REST接口只能增加参数，不能减少参数，返回值的内容也是只增不减。

【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。

白话：

明确了责任和义务，接口提供方也有义务推动接口使用方尽早迁移，不要积累技术负债。

【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)

白话：

常量比变量，永远都不变的原则。

【强制】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。

白话：

Java世界里相等请用equals方法，==表示对象相等，一般在框架开发中会用到。

关于基本数据类型与包装数据类型的使用标准如下:

- 【强制】所有的POJO类属性必须使用包装数据类型。
- 【强制】RPC方法的返回值和参数必须使用包装数据类型。
- 【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: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。

白话：

其实包装数据类型与基本数据类型相比，增加了一个null的状态，可以携带更多的语义。

【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。

白话：

虽然这里反例不太容易看懂，但是要记得持久领域对象之前由应用层统一赋值gmtCreate和gmtModify字段。

【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。

白话：

不到万不得已不要使用JDK自身的序列化，机制很重，信息冗余有版本。

【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。

白话：

这样做一种是规范，代码清晰，还有就是异常堆栈上更容易识别出错的方法和语句。

【强制】POJO 类必须写 toString 方法。使用 IDE 的中工具 :`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。

白话：

这里还有一个大坑，写toString的时候要保证不会发生NPE，有的时候toString调用实例变量的toString，实例变量由于某些原因为null，导致NPE，代码没有处理好就终止，这个问题坑了好多次。

【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);
```

白话：

编程要留心眼，任何不确定的地方都要判断、处理，否则掉到坑里了自己爬出来很费劲。

Java编程判空的思想要实施萦绕在每个开发人员的脑海里。

【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。

白话：

这规范说的咋就和我的习惯一模一样呢！

【推荐】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。

白话：

我推荐把一套逻辑的共有方法、保护方法、私有方法放在一起，所有getter/setter放在最后，这样感觉更有逻辑！

【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，尽量不要增加业务逻辑，增加排查问题的难度。

反例:

```
public Integer getData() {
    if (true) {
        return data + 100;
    } else {
        return data - 100; }
    }
```

白话：

双手赞成。

【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。

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。

白话：

一定使用StringBuilder，不要使用StringBuffer，StringBuffer是线程安全的，太重。

我就一直想不明白Java编译器为什么不做个优化呢？

【推荐】下列情况，声明成 final 会更有提示性:

- 不需要重新赋值的变量，包括类属性、局部变量。
- 对象参数前加final，表示不允许修改引用的指向。
- 类方法确定不允许被重写。

白话：

尽量多使用final关键字，保证编译器的校验机制起作用，也体现了“契约式编程”的思想。

【推荐】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。

白话：

最好是使用构造函数来重新构造对象，使用clone浅拷贝的时候，对象引用关系可能很复杂，不直观，不好理解。

【推荐】类成员与方法访问控制从严:

- 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.
- 工具类不允许有public或default构造方法。
- 类非static成员变量并且与子类共享，必须是protected.
- 类非static成员变量并且仅在本类使用，必须是private.
- 类static成员变量如果仅在本类使用，必须是private.
- 若是static成员变量，必须考虑是否为final.
- 类成员方法只供类内部调用，必须是private.

- 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?变量像自己的小孩，尽量在自己的视 线内，变量作用域太大，如果无限制的到处跑，那么你会担心的。

白话：

没什么好说的，两个词，高内聚，低耦合，功能模块闭包，哦，是三个词。

集合处理

【强制】关于 hashCode 和 equals 的处理，遵循如下规则:

- 只要重写equals，就必须重写hashCode.
- 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。
- 如果自定义对象做为Map的键，那么必须重写hashCode和equals.

说明: 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。

白话：

Hash是个永恒的话题，大家可以看下times33和Murmurhash算法。

【强制】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。

白话：

这种问题本来测试可以测试到，但是开发永远都不要有依赖测试的想法，一切靠自己，当然我们的测试人员都是很靠谱的。

【强制】 在 subList 场景中，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。

白话：

如果一定要更改子列表，重新构造新的ArrayList，使用 `public ArrayList(Collection<? extends E> c)`。

【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()。

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。

正例:

```
 java
    List<String> list = new ArrayList<String>(2);
list.add("guan");
    list.add("bao");
    String[] array = new String[list.size()];
    array = list.toArray(array);
```

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。

白话：

搞不懂Java编译器为什么不做优化，人用逻辑能推导的，程序一定可以自动实现。

【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[] { "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。

白话：

如果需要对asList返回的List做更改，可以构造新的ArrayList，使用 `public ArrayList(Collection<? extends E> c)` 构造器。

【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，做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: 1)频繁往外读取内容的，适合用上界 Extends。 2)经常往里插入的，适合用下界 Super。

白话： `<extends T>` ，必须是T或T的子类。

集合写(add)： 因为不能确定集合实例化时用的是T或T的子类，所以没有办法写。例如： `List<? extends Number> foo = new ArrayList<Number/Integer/Double>()` ，你不能add Number，因为也可能是Integer或Double的List， 同理也不能add Integer或Double，即，extends T， 不能集合add。集合读(get)： 只能读出T类型的数据。< super T>， 必须是T或T的父类。集合写(add)： 可以add T或T的子类。集合读(get)： 不能确定从集合里读出的是哪个类型（可能是T也可能是T的父类，或者Object），所以没有办法使用get。例如：`List<? super Integer> foo3 = new ArrayList<Integer/Number/Object>();` 只能保证get出来是Object。

下面是示例，test1和test2在编译时都有错误提示。

```
package com.robert.javaspec;

import java.util.LinkedList;
import java.util.List;

/**
 * Created by WangMeng on 2017-04-13.
 * FIX ME
 */
public class Main {
    public static void main(String[] args) {

    }

    public void test1(){
        List<? extends A> childofa=new LinkedList<>();
        B b=new B();
        A a=new A();
        childofa.add(a);
        childofa.add(b);
        A ta= childofa.get(0);
    }

    public void test2(){
        List<? super B> superOfb = new LinkedList<>();
        B b = new B();
        A a = new A();
        superOfb.add(a);
        superOfb.add(b);
        A ta = superOfb.get(0);
```

```
            B tb = superOfb.get(0);
        }
    }

    class A {
        @Override
        public String toString() {
            return "A";
        }
    }

    class B extends A {

        @Override
        public String toString() {
            return "B";
        }
    }
```

【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);
        }
    }
```

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?

正例:

```
    Iterator<String> it = a.iterator();
    while (it.hasNext()) {
        String temp = it.next();
        if (删除元素的条件) {
            it.remove();
        }
    }
```

白话：

修改一定要使用Iterator。

反例中改成2，抛出ConcurrentModificationException，因为2是数组的结束边界。

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

说明:

x，y的比较结果和y，x的比较结果相反。

x>y，y>z，则x>z。

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;
    }
}
```


白话：

除非逻辑混乱，否则这些条件都能满足。

【推荐】集合初始化时，尽量指定集合初始值大小。

说明: ArrayList尽量使用ArrayList(int initialCapacity) 初始化。

白话：

预估数组大小，能够提高程序效率，写代码的时候脑袋里面要有运行的思想。

想了解性能和容量评估，请参考互联网性能与容量评估的方法论和典型案例。

【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。

白话：

写代码其实就是在程序员脑袋里执行代码的过程，直觉就是两次肯定不如一次做完事更快。

【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	分段锁技术
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，注意存储 null 值时会抛出 NPE 异常。

白话：

存储null值场景不多，在防止缓存穿透的情况下，有的时候会缓存null key

【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。

白话：

对于HashMap理论上是无序的，我做了个试验，每次输出都是稳定的。

数值：

```
HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();

map.put(3, 3);
map.put(1, 1);
map.put(2, 2);
map.put(4, 4);

for (Entry<Integer, Integer> entry : map.entrySet()) {
    System.out.println(entry.getKey());
}
```

事实证明，每次输出也是1、2、3、4，有序并且稳定的。

字符串值：

```
HashMap<String, String> map = new HashMap<String, String>();

map.put("3000", "3");
map.put("1000", "1");
map.put("2000", "2");
map.put("4000", "4");

for (Entry<Integer, Integer> entry : map.entrySet()) {
    System.out.println(entry.getKey());
}
```

事实证明，每次输出也是4000、1000、2000、3000，无序但是稳定的。

与阿里专家咨询，这里HashMap不稳定性是指rehash后输出顺序则会变化。

【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。

白话：

如果不需要精确去重，参考布隆过滤器（Bloom Filter）。

控制语句

【强制】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;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。

白话：

最好每个case都用break结束，不要组合几个分支到一个逻辑，太不直观。

【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;

白话：

这条有歧义，个人认为有的时候就一行语句不加也可以。

【推荐】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: 如果非得使用if()…else if()…else…方式表达逻辑，【强制】请勿超过3层， 超过请使用状态设计模式。

正例: 逻辑上超过 3 层的 if-else 代码可以使用卫语句，或者状态模式来实现。

白话：

朋友说超过三层考虑状态设计模式也不完全正确，大概可以理解为多层的逻辑嵌套不是好的代码风格，需要使用对应的重构方法做出优化，而每种坏味都有对应的优化方法和步骤，以及优缺点限制条件。

写程序一定要遵守红花绿叶原则，主逻辑放在主方法中，这是红花，子逻辑封装成小方法调用，这是绿叶，不要把不同层次的逻辑写在一个大方法体里，很难理解，就像绿叶把红花挡住了，谁还能看到。举例说明：

```
public void handleProcess() {
    // 骨架逻辑
    validate();
    doProcess();
    declareResource();
}
```

普及一下，如下类似排比句的代码就是卫语句，以前每天都这么写但是还真是刚刚知道这叫卫语句:)


```
public double getPayAmount() {
   if (isDead()) return deadPayAmount();
   if (isSeparated()) return separatedPayAmount();
   if (isRetired()) return retiredPayAmount();
   return normalPayAmount();

}
```

不提倡的写法：

```
public double getPayAmount() {
   if (isDead())
       return deadPayAmount();
   else if (isSeparated())
       return separatedPayAmount();
   else if (isRetired())
       return retiredPayAmount();
   else
       return normalPayAmount();
}
```

【推荐】除常用方法(如 getXxx/isXxx)等外，不要在条件判断中执行其它复杂的语句，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。

说明: 很多 if 语句内的逻辑相当复杂，阅读者需要分析条件表达式的最终结果，才能明确什么 样的条件执行什么样的语句，那么，如果阅读者分析逻辑表达式错误呢?

正例:

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

反例:

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

白话：

这个反例真的经常见到，写这个代码的人自己不觉得这样很难看吗？

【推荐】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)。

白话：

切记，循环体内尽量不要获取资源、不要处理异常。

【推荐】接口入参保护，这种场景常见的是用于做批量操作的接口。

白话：

用白话说，就是控制批量参数的数量，一次不能太多，否则内存溢出。

【参考】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。

5. 敏感权限入口。

白话：

在这个框框内，根据业务适当调整是可以的。

【参考】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
数检查要求。

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。

白话：

在这个框框里，根据业务适当调整是可以的。

命名规约

【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 / $Object / name_ / name$ / Object$`

白话：

这条不够严格，普通的变量、类名、方法名必须使用驼峰式命名，最好不要使用下划线和美元符号，否则看起来像脚本语言似得，常量可以使用下划线，但是也不要放在常量开头和结尾。

【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，即使纯拼音命名方式 也要避免采用。

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

正例: alibaba / taobao / youku / hangzhou 等国际通用的名称，可视同英文。

白话：

中英混合的人种咱不歧视，变量名混合太丑了。

Java编译器支持Unicode（UTF-8)，允许中文命名变量，不过打中文还是没有英文快。

英文！英文起名，洋气、大方、高大上…

【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。

正例:MarcoPolo / UserDO / XmlService / TcpUdpDeal / TaPromotion

反例:macroPolo / UserDo / XMLService / TCPUDPDeal / TAPromotion

白话：

约定俗成的名称或者缩写例外。

【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

白话：

约定俗称的名称或者缩写例外。

ID为简写，Id和ID均可。

【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

反例: MAX_COUNT

白话：

必须全部大写，除了字母数字只可以使用下划线，并且不能用在开头和结尾。

【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。

白话：

家里放一瓶敌敌畏，上面不写标签，万一喝大了、渴了、喝了、就惨了，你懂的。

【强制】中括号是数组类型的一部分，数组定义如下:String[] args;

反例: 使用String args[]的方式来定义。

白话：

这种语法编译器也认，但是我们毕竟写Java程序，而不是写C/C++程序。

这怪Java编译器小组，一开始就不应该支持这种语法。

【强制】POJO 类中布尔类型的变量，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。

白话：

一些框架使用getter和setter做序列化，有的根据属性本身取值，带了is前缀就找不到了，变量名不要带be动词，语法不对，英文补考！

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

白话：

包名大写、带下划线等，不专业、难看、不高大上。

【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。

白话：

不要太抠，不是太长的名字直接写上就好，编译器编译优化后变量名将不存在，会编译成相对于方法堆栈bp指针地址的相对地址，长变量名不会占用更多空间。

英文中的缩写有个惯例，去掉元音留下辅音即可，不能乱缩写。

【推荐】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;
```

白话：

让全世界都知道你会设计模式，这是个崇尚显摆的社会。

【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 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。

白话：

脱了裤子放屁始终有点麻烦。

接口和实现类的命名有两套规则:

【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。

【推荐】 如果是形容能力的接口名称，取对应的形容词做接口名(通常是–able 的形式)。

正例: AbstractTranslator 实现 Translatable。

白话：

严重同意！可是想想Observer和Observable，我就不说话了。

【参考】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。

白话：

不要驼峰！记住枚举不要驼峰！总是有好多人枚举用驼峰。

【参考】各层命名规约:

- Service/DAO层方法命名规约：
    - 获取单个对象的方法用get做前缀。
    - 获取多个对象的方法用list做前缀。
    - 获取统计值的方法用count做前缀。
    - 插入的方法用save(推荐)或insert做前缀。
    - 删除的方法用remove(推荐)或delete做前缀。
    - 修改的方法用update做前缀。
- 领域模型命名规约：
    - 数据对象:xxxDO，xxx即为数据表名。
    - 数据传输对象:xxxDTO，xxx为业务领域相关的名称。
    - 展示对象:xxxVO，xxx一般为网页名称。
    - POJO是DO/DTO/BO/VO的统称，禁止命名成xxxPOJO。

白话：

大家都这么认为很重要。

常量定义

【强制】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);`

白话：

这个不用说了，随地吐痰和随地大小便是不应该的，新加坡是要鞭刑的！

【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。

说明: Long a = 2l; 写的是数字的21，还是Long型的2?

白话：

看看区块链中用了base58，而不是base64，秒懂什么是从用户角度考虑产品设计！

【推荐】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，非得使用查找功能才能定位到修改的常量，不利于理解和维护。

白话：

尽量让功能自闭包，标准是一个小模块拷贝出去直接就能用，而不是缺这缺那的，是不是读者很多时候拷贝了一套类，运行时候发现不能用，缺常量，把常量类拷贝过来，发现常量类中有很多不相关的常量，还得清理。

【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。

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。

应用内共享常量: 放置在一方库的modules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。

子工程内部共享常量: 即在当前子工程的constant目录下。

包内共享常量: 即在当前包下单独的constant目录下。

类内共享常量: 直接在类内部private static final定义。

白话：

一方库、二方库、三方库，叫法很专业，放在离自己最近的上面一个层次即可。

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

白话：

枚举值需要定义延伸属性的场景通常是要持久数据库，或者显示在界面上。

格式规约

【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:

- 左大括号前不换行。
- 左大括号后换行。
- 右大括号前换行。
- 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。

白话：

好风格，讨厌那种左大括号前换行的，看不惯。

【强制】 左括号和后一个字符之间不出现空格; 同样，右括号和前一个字符之间也不出现空 格。详见第 5 条下方正例提示。

白话：

程序写完可以用编辑器的格式化功能格式化，Eclipse中快捷键是shift+alt+f，笔者写程序的时候有个习惯，每次谢了一段代码都会按ctrl+alt+o、ctrl+alt+f、ctrl+s，相信会有相同习惯的同行。

【强制】if/for/while/switch/do 等保留字与左右括号之间都必须加空格。

白话：

程序写完可以用编辑器的格式化功能格式化，Eclipse中快捷键是shift+alt+f，笔者写程序的时候有个习惯，每次谢了一段代码都会按ctrl+alt+o、ctrl+alt+f、ctrl+s，相信会有相同习惯的同行。

【强制】任何运算符左右必须加一个空格。

说明: 运算符包括赋值运算符=、逻辑运算符&&、加减乘除符号、三目运算符等。

白话：

程序写完可以用编辑器的格式化功能格式化，Eclipse中快捷键是shift+alt+f，笔者写程序的时候有个习惯，每次谢了一段代码都会按ctrl+alt+o、ctrl+alt+f、ctrl+s，相信会有相同习惯的同行。

【强制】缩进采用 4 个空格，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
    }
}
```

白话：

这样看惯了，怎么看怎么清晰。

【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:

- 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。
- 运算符与下文一起换行。
- 方法调用的点符号与下文一起换行。
- 在多个参数超长，逗号后进行换行。
- 在括号前不要换行，见反例。

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);
```

白话：

一行代码尽量不要写太长，长了拆开不就得了。

【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"a", 后边必须要有一个空格。

```
method("a", "b", "c");
```

白话：

不加空格太挤了，就像人没长开似得。

【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。

白话：

请不要用GB字符集，换了环境总有问题，Java程序多数跑在Linux上，当然要用Unix换行符。

【推荐】没有必要增加若干空格来使某一行的字符与上一行的相应字符对齐。

正例:

```
int a = 3;
long b = 4L;
float c = 5F;
StringBuffer sb = new StringBuffer();
```

说明: 增加 sb 这个变量，如果需要对齐，则给 a、b、c 都要增加几个空格，在变量比较多的 情况下，是一种累赘的事情。

白话：

没必要，没必要，那样反而不好看。

【推荐】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。

白话：

和我的习惯一样一样的，一段逻辑空一行。