（九）其它

1. 【强制】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”);

2. 【强制】velocity 调用 POJO 类的属性时，建议直接使用属性名取值即可，模板引擎会自动按规范调用 POJO 的 getXxx()，如果是 boolean 基本数据类型变量（boolean 命名不需要加 is 前缀），会自动调用 isXxx()方法。
   说明：注意如果是 Boolean 包装类对象，优先调用 getXxx()的方法。

3. 【强制】后台输送给页面的变量必须加$!{var}——中间的感叹号。
   说明：如果 var 等于 null 或者不存在，那么${var}会直接显示在页面上。

4. 【强制】注意 Math.random() 这个方法返回是 double 类型，注意取值的范围 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。

5. 【强制】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，使用 System.nanoTime()的方式。在 JDK8 中，针对统计时间等场景，推荐使用 Instant 类。

6. 【推荐】不要在视图模板中加入任何复杂的逻辑。
   说明：根据 MVC 理论，视图的职责是展示，不要抢模型和控制器的活。

7. 【推荐】任何数据结构的构造或初始化，都应指定大小，避免数据结构无限增长吃光内存。

8. 【推荐】及时清理不再使用的代码段或配置信息。
   说明：对于垃圾代码或过时配置，坚决清理干净，避免程序过度臃肿，代码冗余。
   正例：对于暂时被注释掉，后续可能恢复使用的代码片断，在注释代码上方，统一规定使用三个斜杠(///)来说明注释掉代码的理由。

二、异常日志

(一)异常处理

1. 【强制】Java 类库中定义的可以通过预检查方式规避的 RuntimeException 异常不应该通过 catch 的方式来处理，比如：NullPointerException，IndexOutOfBoundsException 等等。
   说明：无法通过预检查的异常除外，比如，在解析字符串形式的数字时，不得不通过 catch NumberFormatException 来实现。
   正例：if (obj != null) {...}
   反例：try { obj.method(); } catch (NullPointerException e) {...}

2. 【强制】异常不要用来做流程控制，条件控制。
   说明：异常设计的初衷是解决程序运行中的各种意外情况，且异常的处理效率比条件判断方式要低很多。

3. 【强制】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。
   说明：对大段代码进行 try-catch，使程序无法根据不同的异常做出正确的应激反应，也不利于定位问题，这是一种不负责任的表现。
   正例：用户注册的场景中，如果用户输入非法字符，或用户名称已存在，或用户输入密码过于简单，在程序上作出分门别类的判断，并提示给用户。

4. 【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。

5. 【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。

6. 【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。

7. 【强制】不要在 finally 块中使用 return。
   说明：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。

8. 【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。
   说明：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。

9. 【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。
   说明：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。

10. 【推荐】防止 NPE，是程序员的基本修养，注意 NPE 产生的场景：

    1）返回类型为基本数据类型，return 包装数据类型的对象时，自动拆箱有可能产生 NPE。

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

    2） 数据库的查询结果可能为 null。

    3） 集合里的元素即使 isNotEmpty，取出的数据元素也可能为 null。

    4） 远程调用返回对象时，一律要求进行空指针判断，防止 NPE。

    5） 对于 Session 中获取的数据，建议 NPE 检查，避免空指针。

    6） 级联调用 obj.getA().getB().getC()；一连串调用，易产生 NPE。

    正例：使用 JDK8 的 Optional 类来防止 NPE 问题。

11. 【推荐】定义时区分 unchecked / checked 异常，避免直接抛出 new RuntimeException()，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。

12. 【参考】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()方法、“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：

    1）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。

    2）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。

13. 【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，甚至是组件化。

    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) {...}
```

（二）日志规约

1. 【强制】应用中不可直接使用日志系统（Log4j、Logback）中的 API，而应依赖使用日志框架 SLF4J 中的 API，使用门面模式的日志框架，有利于维护和各个类的日志处理方式统一。

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

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

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

3．【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，如 stats/monitor/access 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，如将错误日志和业务日志分开存放，便于开发人员查看，也便于通过日志对系统进行及时监控。

4．【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 + " and 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）建设采用如下方式

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

正例：（占位符）

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

5．【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">

6．【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);

7．【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，不利于系统性能提升，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？

8. 【推荐】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。如非必要，请不要在此场景打出 error 级别，避免频繁报警。
说明：注意日志输出的级别，error 级别只记录系统逻辑出错、异常或者重要的错误信息。

9. 【推荐】尽量用英文来描述日志错误信息，如果日志中的错误信息用英文描述不清楚的话使用中文描述即可，否则容易产生歧义。国际化团队或海外部署的服务器由于字符集问题，【强制】使用全英文来注释和描述日志错误信息。

三、单元测试

1. 【强制】好的单元测试必须遵守 AIR 原则。
   说明：单元测试在线上运行时，感觉像空气（AIR）一样并不存在，但在测试质量的保障上，却是非常关键的。好的单元测试宏观上来说，具有自动化、独立性、可重复执行的特点。
   - A：Automatic（自动化）
   - I：Independent（独立性）
   - R：Repeatable（可重复）
2. 【强制】单元测试应该是全自动执行的，并且非交互式的。测试用例通常是被定期执行的，执行过程必须完全自动化才有意义。输出结果需要人工检查的测试不是一个好的单元测试。单元测试中不准使用 System.out 来进行人肉验证，必须使用 assert 来验证。
3. 【强制】保持单元测试的独立性。为了保证单元测试稳定可靠且便于维护，单元测试用例之间决不能互相调用，也不能依赖执行的先后次序。
   反例：method2 需要依赖 method1 的执行，将执行结果作为 method2 的输入。
4. 【强制】单元测试是可以重复执行的，不能受到外界环境的影响。
   说明：单元测试通常会被放到持续集成中，每次有代码 check in 时单元测试都会被执行。如果单测对外部环境（网络、服务、中间件等）有依赖，容易导致持续集成机制的不可用。
   正例：为了不受外界环境影响，要求设计代码时就把 SUT 的依赖改成注入，在测试时用 spring 这样的 DI 框架注入一个本地（内存）实现或者 Mock 实现。
5. 【强制】对于单元测试，要保证测试粒度足够小，有助于精确定位问题。单测粒度至多是类级别，一般是方法级别。
   说明：只有测试粒度小才能在出错时尽快定位到出错位置。单测不负责检查跨类或者跨系统的交互逻辑，那是集成测试的领域。
6. 【强制】核心业务、核心应用、核心模块的增量代码确保单元测试通过。
   说明：新增代码及时补充单元测试，如果新增代码影响了原有单元测试，请及时修正。
7. 【强制】单元测试代码必须写在如下工程目录：src/test/java，不允许写在业务代码目录下。
   说明：源码构建时会跳过此目录，而单元测试框架默认是扫描此目录。
8. 【推荐】单元测试的基本目标：语句覆盖率达到 70%；核心模块的语句覆盖率和分支覆盖率都要达到 100%
   说明：在工程规约的应用分层中提到的 DAO 层，Manager 层，可重用度高的 Service，都应该进行单元测试。