移联科技

技术部开发手册

V1.1

文件编号：TD-JS-0001			
生效日期：2019-10-10			
保密级别：⭘公开文件 ☼秘密文件 ⭘机密文件 ⭘绝密文件			
修订记录			
版本号	日期	修订内容	修订者
V1.0	2019-10-10	初始稿	袁妙
V1.1	2020-05-01	修订分支管理	袁妙

目 录

1.概述

1.1 目的

现代软件的高速发展对开发者的综合素质要求越来越高，不仅仅编程的知识点，其他维度的因素（比如方法、习惯、风格、经验等）也会影响软件的交付质量。在整个软件的生命周期中，大部分的时间都将花费在维护上，而且现代大型软件都是整个团队多人协同完成的，无规矩不成方圆，无规范难以协同，因此，为了增强团队成员之间的编码理解，减少阅读代码的障碍，提高代码质量，尽可能少踩坑，杜绝踩重复的坑，少走弯路，切实提升系统稳定性，码出高效码出质量，我们制定了本手册作为技术开发过程中的参考准则。

本手册的整理参考了阿里开发手册，并结合了本公司软件开发的特点和实际情况，是智慧的结晶、经验的总结、教训的吸取，同时，本手册将根据实际情况不断的完善与修订。

2. 开发规约

2.1 编码规约

2.1.1 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__

2. 【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。

正例：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

3. 【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：MarcoPolo / UserDO / XmlService / TcpUdpDeal / TaPromotion

反例：macroPolo / UserDo / XMLService / TCPUDPDeal / TAPromotion

4. 【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

5. 【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

6. 【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。

7. 【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。

8. 【强制】POJO 类中布尔类型的变量，都不要加 is 前缀，否则部分框架解析会引起序列化错误。

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。

9. 【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】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。

11. 【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：变量 int a 的随意命名方式。

12. 【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;
```

13. 【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。

14. 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。

正例：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 的形式）。

正例：AbstractTranslator 实现 Translatable 接口。

15. 【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。

16. 【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。

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。

2.1.2 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);

2. 【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?

3. 【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。

4. 【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。

5. 【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;
    }
}
```

2.1.3 代码格式

1. 【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：

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。

2. 【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)

3. 【强制】if/for/while/switch/do 等保留字与括号之间都必须加空格。

4. 【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。

5. 【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
    }
}
```

6. 【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 ygb = new String();
```

7. 【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：

```
StringBuffer sb = new StringBuffer();
// 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("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);
```

8. 【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);
```

9. 【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。

10. 【推荐】单个方法的总行数不超过 80 行。

说明：包括方法签名、结束右大括号、方法内代码、注释、空行、回车及任何不可见字符的总行数不超过 80 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。

11. 【推荐】没有必要增加若干空格来使某一行的字符与上一行对应位置的字符对齐。

正例：

```
int one = 1;
long two = 2L;
float three = 3F;
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，是非常累赘的事情。

12. 【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。

2.1.4 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。

2. 【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。

3. 【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) {...}

4. 【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。

5. 【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。

6. 【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）

7. 【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 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。

8. 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。

9. 【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。

10. 【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。

11. 【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。

12. 【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。

13. 【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。

14. 【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);
```

15. 【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，此条规则优先于第 16 条规则。

16. 【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。

17. 【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;
    }
```

```
}
```

18. 【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 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";
}
```

19. 【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） 不允许被重写的方法，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。

5） 避免上下文重复使用一个变量，使用 final 描述可以强制重新定义一个变量，方便更地进行重构。

20. 【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 方法实现域对象的深度遍历式拷贝。

21. 【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。

2.1.5 集合处理

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

1） 只要重写 equals，就必须重写 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 存储的对象必须重写这两个方法。

3） 如果自定义对象作为 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。

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。

3. 【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。

4. 【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；如果数组元素个数大于实际所需，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：

```
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。

5. 【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，add/remove/clear 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[] { "you", "wu" };
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] = "gujin"; 那么 list.get(0)也会随之修改。

6. 【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>。

7. 【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？

8. 【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 会报 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，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;
    }
};
```

9. 【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
HashMap<String, String> userCache = new HashMap<>(16);
// 全省略方式
ArrayList<User> users = new ArrayList(10);
```

10. 【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。

11. 【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。

12. 【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。

13. 【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。

14. 【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。

2.1.6 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。

2. 【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");
    ...
    }
}
```

3. 【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。

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 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。

5. 【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。

6. 【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。

7. 【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。

8. 【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。

9. 【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，使用 ScheduledExecutorService 则没有这个问题。

10. 【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。

11. 【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。

12. 【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：

```
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...
}
```

13. 【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）。

14. 【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。

15. 【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。

2.1.7 控制语句

1. 【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，即使空代码。

2. 【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，避免采用 单行的编码方式：if (condition) statements;

3. 【强制】在高并发场景中，避免使用”等于”判断作为中断或退出的条件。

说明：如果并发控制没有处理好，容易产生等值判断被“击穿”的情况，使用大于或小于的区间判断条件来代替。

反例：判断剩余奖品数量等于 0 时，终止发放奖品，但因为并发处理错误导致奖品数量瞬间变成了负数，这样的话，活动无法终止。

4. 【推荐】表达异常的分支时，少用 if-else 方式，这种方式可以改写成：

```
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】避免后续代码维 护困难，请勿超过 3 层。

正例：超过 3 层的 if-else 的逻辑判断代码可以使用卫语句、策略模式、状态模式等来实现，其中卫语句示例如下：

```
public void today() {
    if (isBusy()) {
        System.out.println(“change time.”);
        return;
    }
    if (isFree()) {
        System.out.println(“go to travel.”);
        return;
    }
    System.out.println(“stay at home to learn Alibaba Java Coding Guidelines.”);
    return;
}
```

5. 【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
final boolean existed = (file.open(fileName, "w") != null) && (...) || (...);
if (existed) {
    ...
}
```

反例：

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

6. 【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）。

7. 【推荐】避免采用取反逻辑运算符。

说明：取反逻辑不利于快速理解，并且取反逻辑写法必然存在对应的正向逻辑写法。

正例：使用 if (x < 628) 来表达 x 小于 628。

反例：使用 if (!(x >= 628)) 来表达 x 小于 628。

8. 【推荐】接口入参保护，这种场景常见的是用作批量操作的接口。

9. 【参考】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，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。

10. 【参考】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。但在方法说明里必须注明外部参数检查要求。

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。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。

2.1.8 注释规约

1. 【强制】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。

2. 【强制】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：对子类的实现要求，或者调用注意事项，请一并说明。

3. 【强制】所有的类都必须添加创建者和创建日期。

4. 【强制】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。

5. 【强制】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。

6. 【推荐】与其“半吊子”英文来注释，不如用中文注释把问题说清楚。专有名词与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。

7. 【推荐】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。

8. 【参考】谨慎注释掉代码。在上方详细说明，而不是简单地注释掉。如果无用，则删除。

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）。

9. 【参考】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。

10. 【参考】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。

11. 【参考】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。

2.1.9 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。

2.2 异常日志

2.2.1 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) {...}

2.2.2 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。

2.3 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。

9. 【推荐】编写单元测试代码遵守 BCDE 原则，以保证被测试模块的交付质量。

B：Border，边界值测试，包括循环边界、特殊取值、特殊时间点、数据顺序等。

C：Correct，正确的输入，并得到预期的结果。

D：Design，与设计文档相结合，来编写单元测试。

E：Error，强制错误信息输入（如：非法数据、异常流程、非业务允许输入等），并得到预期的结果。

10. 【推荐】对于数据库相关的查询，更新，删除等操作，不能假设数据库里的数据是存在的，或者直接操作数据库把数据插入进去，请使用程序插入或者导入数据的方式来准备数据。

反例：删除某一行数据的单元测试，在数据库中，先直接手动增加一行作为删除目标，但是这一行新增数据并不符合业务插入规则，导致测试结果异常。

11. 【推荐】和数据库相关的单元测试，可以设定自动回滚机制，不给数据库造成脏数据。或者对单元测试产生的数据有明确的前后缀标识。

正例：在 RDC 内部单元测试中，使用 RDC_UNIT_TEST_的前缀标识数据。

12. 【推荐】对于不可测的代码建议做必要的重构，使代码变得可测，避免为了达到测试要求而书写不规范测试代码。

13.【推荐】在设计评审阶段，开发人员需要和测试人员一起确定单元测试范围，单元测试最好覆盖所有测试用例。

14. 【推荐】单元测试作为一种质量保障手段，不建议项目发布后补充单元测试用例，建议在项目提测前完成单元测试。

15. 【参考】为了更方便地进行单元测试，业务代码应避免以下情况：

构造方法中做的事情过多。

存在过多的全局变量和静态方法。

存在过多的外部依赖。

存在过多的条件语句。

说明：多层条件语句建议使用卫语句、策略模式、状态模式等方式重构。

16. 【参考】不要对单元测试存在如下误解：

那是测试同学干的事情。本文是开发手册，凡是本文内容都是与开发同学强相关的。

单元测试代码是多余的。系统的整体功能与各单元部件的测试正常与否是强相关的。

单元测试代码不需要维护。一年半载后，那么单元测试几乎处于废弃状态。

单元测试与线上故障没有辩证关系。好的单元测试能够最大限度地规避线上故障。

2.4 安全规约

1. 【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。

2. 【强制】用户敏感数据禁止直接展示，必须对展示数据进行脱敏。

说明：中国大陆个人手机号码显示为:158****9119，隐藏中间 4 位，防止隐私泄露。

3. 【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。

4. 【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。

5. 【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。

6. 【强制】表单、AJAX 提交必须执行 CSRF 安全验证。

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的情况下对数据库中用户参数进行相应修改。

7. 【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的机制，如数量限制、疲劳度控制、验证码校验，避免被滥刷而导致资损。

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。

8. 【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。

2.5 数据库规约（MySQL）

2.5.1 建表规约

1. 【强制】表达是与否概念的字段，必须使用 is_xxx 的方式命名，数据类型是 unsigned tinyint

（1 表示是，0 表示否）。

说明：任何字段如果为非负数，必须是 unsigned。

注意：POJO 类中的任何布尔类型的变量，都不要加 is 前缀，所以，需要在<resultMap>设置从 is_xxx 到 Xxx 的映射关系。数据库表示是与否的值，使用 tinyint 类型，坚持 is_xxx 的命名方式是为了明确其取值含义与取值范围。

正例：表达逻辑删除的字段名 is_deleted，1 表示删除，0 表示未删除。

2. 【强制】表名、字段名必须使用小写字母或数字，禁止出现数字开头，禁止两个下划线中间只出现数字。数据库字段名的修改代价很大，因为无法进行预发布，所以字段名称需要慎重考虑。

说明：MySQL 在 Windows 下不区分大小写，但在 Linux 下默认是区分大小写。因此，数据库名、表名、字段名，都不允许出现任何大写字母，避免节外生枝。

正例：aliyun_admin，rdc_config，level3_name

反例：AliyunAdmin，rdcConfig，level_3_name

3. 【强制】表名不使用复数名词。

说明：表名应该仅仅表示表里面的实体内容，不应该表示实体数量，对应于 DO 类名也是单数形式，符合表达习惯。

4. 【强制】禁用保留字，如 desc、range、match、delayed 等，请参考 MySQL 官方保留字。

5. 【强制】主键索引名为 pk_字段名；唯一索引名为 uk_字段名；普通索引名则为 idx_字段名。

说明：pk_ 即 primary key；uk_ 即 unique key；idx_ 即 index 的简称。

6. 【强制】小数类型为 decimal，禁止使用 float 和 double。

说明：float 和 double 在存储的时候，存在精度损失的问题，很可能在值的比较时，得到不正确的结果。如果存储的数据范围超过 decimal 的范围，建议将数据拆成整数和小数分开存储。

7. 【强制】如果存储的字符串长度几乎相等，使用 char 定长字符串类型。

8.【强制】varchar 是可变长字符串，不预先分配存储空间，长度不要超过 5000，如果存储长度大于此值，定义字段类型为 text，独立出来一张表，用主键来对应，避免影响其它字段索引效率。

9. 【强制】表必备三字段：id, gmt_create, gmt_modified。

说明：其中 id 必为主键，类型为 bigint unsigned、单表时自增、步长为 1。gmt_create, gmt_modified 的类型均为 datetime 类型，前者现在时表示主动创建，后者过去分词表示被动更新。

10. 【推荐】表的命名最好是加上“业务名称_表的作用”。

正例：alipay_task / force_project / trade_config

11. 【推荐】库名与应用名称尽量一致。

12. 【推荐】如果修改字段含义或对字段表示的状态追加时，需要及时更新字段注释。

13. 【推荐】字段允许适当冗余，以提高查询性能，但必须考虑数据一致。冗余字段应遵循：

1）不是频繁修改的字段。

2）不是 varchar 超长字段，更不能是 text 字段。

正例：商品类目名称使用频率高，字段长度短，名称基本一成不变，可在相关联的表中冗余存储类目名称，避免关联查询。

14. 【推荐】单表行数超过 500 万行或者单表容量超过 2GB，才推荐进行分库分表。

说明：如果预计三年后的数据量根本达不到这个级别，请不要在创建表时就分库分表。

15. 【参考】合适的字符存储长度，不但节约数据库表空间、节约索引存储，更重要的是提升检索速度。

正例：如下表，其中无符号值可以避免误存负数，且扩大了表示范围。

对象	年龄区间	类型	字节	表示范围
人	150 岁之内	tinyint unsigned	1	无符号值：0 到 255
龟	数百岁	smallint unsigned	2	无符号值：0 到 65535
恐龙化石	数千万年	int unsigned	4	无符号值：0 到约 42.9 亿
太阳	50 亿年	bigint unsigned	8	无符号值：0 到约 10 的 19 次方

2.5.2 索引规约

1. 【强制】业务上具有唯一特性的字段，即使是多个字段的组合，也必须建成唯一索引。

说明：不要以为唯一索引影响了 insert 速度，这个速度损耗可以忽略，但提高查找速度是明显的；另外，即使在应用层做了非常完善的校验控制，只要没有唯一索引，根据墨菲定律，必然有脏数据产生。

2. 【强制】超过三个表禁止 join。需要 join 的字段，数据类型必须绝对一致；多表关联查询时，保证被关联的字段需要有索引。

说明：即使双表 join 也要注意表索引、SQL 性能。

3. 【强制】在 varchar 字段上建立索引时，必须指定索引长度，没必要对全字段建立索引，根据 实际文本区分度决定索引长度即可。

说明：索引的长度与区分度是一对矛盾体，一般对字符串类型数据，长度为 20 的索引，区分度会高达 90%以上，可以使用 count(distinct left(列名, 索引长度))/count(*)的区分度来确定。

4. 【强制】页面搜索严禁左模糊或者全模糊，如果需要请走搜索引擎来解决。

说明：索引文件具有 B-Tree 的最左前缀匹配特性，如果左边的值未确定，那么无法使用此索引。

5. 【推荐】如果有 order by 的场景，请注意利用索引的有序性。order by 最后的字段是组合索引的一部分，并且放在索引组合顺序的最后，避免出现 file_sort 的情况，影响查询性能。

正例：where a=? and b=? order by c; 索引：a_b_c

反例：索引中有范围查找，那么索引有序性无法利用，如：

WHERE a>10 ORDER BY b; 索引 a_b 无法排序。

6. 【推荐】利用覆盖索引来进行查询操作，避免回表。

说明：如果一本书需要知道第 11 章是什么标题，会翻开第 11 章对应的那一页吗？目录浏览一下就好，这个目录就是起到覆盖索引的作用。

正例：能够建立索引的种类分为主键索引、唯一索引、普通索引三种，而覆盖索引只是一种查询的一种效果，用 explain 的结果，extra 列会出现：using index。

7. 【推荐】利用延迟关联或者子查询优化超多分页场景。

说明：MySQL 并不是跳过 offset 行，而是取 offset+N 行，然后返回放弃前 offset 行，返回 N 行，那当 offset 特别大的时候，效率就非常的低下，要么控制返回的总页数，要么对超过特定阈值的页数进行 SQL 改写。

正例：先快速定位需要获取的 id 段，然后再关联：

SELECT a.* FROM 表 1 a, (select id from 表 1 where 条件 LIMIT 100000,20) b where a.id=b.id

8. 【推荐】SQL 性能优化的目标：至少要达到 range 级别，要求是 ref 级别，如果可以是 consts 最好。

说明：

1）consts 单表中最多只有一个匹配行（主键或者唯一索引），在优化阶段即可读取到数据。

2）ref 指的是使用普通的索引（normal index）。

3）range 对索引进行范围检索。

反例：explain 表的结果，type=index，索引物理文件全扫描，速度非常慢，这个 index 级别比较 range 还低，与全表扫描是小巫见大巫。

9. 【推荐】建组合索引的时候，区分度最高的在最左边。

正例：如果 where a=? and b=? ，如果 a 列的几乎接近于唯一值，那么只需要单建 idx_a 索引即可。

说明：存在非等号和等号混合时，在建索引时，请把等号条件的列前置。如：where c>? and d=? 那么即使 c 的区分度更高，也必须把 d 放在索引的最前列，即索引 idx_d_c。

10. 【推荐】防止因字段类型不同造成的隐式转换，导致索引失效。

11. 【参考】创建索引时避免有如下极端误解：

1）宁滥勿缺。认为一个查询就需要建一个索引。

2）宁缺勿滥。认为索引会消耗空间、严重拖慢更新和新增速度。

3）抵制惟一索引。认为业务的惟一性一律需要在应用层通过“先查后插”方式解决。

2.6 SQL 语句

1. 【强制】不要使用 count(列名)或 count(常量)来替代 count(*)，count(*)是 SQL92 定义的标准统计行数的语法，跟数据库无关，跟 NULL 和非 NULL 无关。

说明：count(*)会统计值为 NULL 的行，而 count(列名)不会统计此列为 NULL 值的行。

2. 【强制】count(distinct col) 计算该列除 NULL 之外的不重复行数，注意 count(distinct col1, col2) 如果其中一列全为 NULL，那么即使另一列有不同的值，也返回为 0。

3.【强制】当某一列的值全是 NULL 时，count(col)的返回结果为 0，但 sum(col)的返回结果为 NULL，因此使用 sum()时需注意 NPE 问题。

正例：可以使用如下方式来避免 sum 的 NPE 问题：

```
SELECT IF(ISNULL(SUM(g)),0,SUM(g)) FROM table;
```

4. 【强制】使用 ISNULL()来判断是否为 NULL 值。

说明：NULL 与任何值的直接比较都为 NULL。

1） NULL<>NULL 的返回结果是 NULL，而不是 false。

2） NULL=NULL 的返回结果是 NULL，而不是 true。

3） NULL<>1 的返回结果是 NULL，而不是 true。

5. 【强制】在代码中写分页查询逻辑时，若 count 为 0 应直接返回，避免执行后面的分页语句。

6. 【强制】不得使用外键与级联，一切外键概念必须在应用层解决。

说明：以学生和成绩的关系为例，学生表中的 student_id 是主键，那么成绩表中的 student_id 则为外键。如果更新学生表中的 student_id，同时触发成绩表中的 student_id 更新，即为级联更新。外键与级联更新适用于单机低并发，不适合分布式、高并发集群；级联更新是强阻塞，存在数据库更新风暴的风险；外键影响数据库的插入速度。

7. 【强制】禁止使用存储过程，存储过程难以调试和扩展，更没有移植性。

8. 【强制】数据订正（特别是删除、修改记录操作）时，要先 select，避免出现误删除，确认无误才能执行更新语句。

9. 【推荐】in 操作能避免则避免，若实在避免不了，需要仔细评估 in 后边的集合元素数量，控制在 1000 个之内。

10. 【参考】如果有国际化需要，所有的字符存储与表示，均以 utf-8 编码，注意字符统计函数的区别。

说明：

SELECT LENGTH("轻松工作"); 返回为 12

SELECT CHARACTER_LENGTH("轻松工作"); 返回为 4

如果需要存储表情，那么选择 utf8mb4 来进行存储，注意它与 utf-8 编码的区别。

11. 【参考】TRUNCATE TABLE 比 DELETE 速度快，且使用的系统和事务日志资源少，但 TRUNCATE 无事务且不触发 trigger，有可能造成事故，故不建议在开发代码中使用此语句。

说明：TRUNCATE TABLE 在功能上与不带 WHERE 子句的 DELETE 语句相同。

2.7 ORM 映射

1. 【强制】在表查询中，一律不要使用 * 作为查询的字段列表，需要哪些字段必须明确写明。

说明：

1）增加查询分析器解析成本。

2）增减字段容易与 resultMap 配置不一致。

3）无用字段增加网络消耗，尤其是 text 类型的字段。

2. 【强制】POJO 类的布尔属性不能加 is，而数据库字段必须加 is_，要求在 resultMap 中进行字段与属性之间的映射。

说明：参见定义 POJO 类以及数据库字段定义规定，在<resultMap>中增加映射，是必须的。 在 MyBatis Generator 生成的代码中，需要进行对应的修改。

3. 【强制】不要用 resultClass 当返回参数，即使所有类属性名与数据库字段一一对应，也需要定义；反过来，每一个表也必然有一个 POJO 类与之对应。

说明：配置映射关系，使字段与 DO 类解耦，方便维护。

4. 【强制】sql.xml 配置参数使用：#{}，#param# 不要使用${} 此种方式容易出现 SQL 注入。

5. 【强制】iBATIS 自带的 queryForList(String statementName,int start,int size) 不推荐使用。

说明：其实现方式是在数据库取到 statementName 对应的 SQL 语句的所有记录，再通过 subList 取 start,size 的子集合。 正例：

```
Map<String, Object> map = new HashMap<>();
map.put("start", start);
map.put("size", size);
```

6. 【强制】不允许直接拿 HashMap 与 Hashtable 作为查询结果集的输出。

说明：resultClass=”Hashtable”，会置入字段名和属性值，但是值的类型不可控。

7. 【强制】更新数据表记录时，必须同时更新记录对应的 gmt_modified 字段值为当前时间。

8. 【推荐】不要写一个大而全的数据更新接口。传入为 POJO 类，不管是不是自己的目标更新字段，都进行 update table set c1=value1,c2=value2,c3=value3; 这是不对的。执行 SQL 时，不要更新无改动的字段，一是易出错；二是效率低；三是增加 binlog 存储。

9. 【参考】@Transactional 事务不要滥用。事务会影响数据库的 QPS，另外使用事务的地方需要考虑各方面的回滚方案，包括缓存回滚、搜索引擎回滚、消息补偿、统计修正等。

10. 【参考】<isEqual>中的 compareValue 是与属性值对比的常量，一般是数字，表示相等时带上此条件；<isNotEmpty>表示不为空且不为 null 时执行；<isNotNull>表示不为 null 值时执行。

2.8 设计规约

1. 【强制】存储方案和底层数据结构的设计获得评审一致通过，并沉淀成为文档。

说明：有缺陷的底层数据结构容易导致系统风险上升，可扩展性下降，重构成本也会因历史数据迁移和系统平滑过渡而陡然增加，所以，存储方案和数据结构需要认真地进行设计和评审，生产环境提交执行后，需要进行 double check。

正例：评审内容包括存储介质选型、表结构设计能否满足技术方案、存取性能和存储空间能否满足业务发展、表或字段之间的辩证关系、字段名称、字段类型、索引等；数据结构变更（如 在原有表中新增字段）也需要进行评审通过后上线。

2. 【强制】在需求分析阶段，如果与系统交互的 User 超过一类并且相关的 User Case 超过 5 个，使用用例图来表达更加清晰的结构化需求。

3. 【强制】如果某个业务对象的状态超过 3 个，使用状态图来表达并且明确状态变化的各个触发条件。

说明：状态图的核心是对象状态，首先明确对象有多少种状态，然后明确两两状态之间是否存在直接转换关系，再明确触发状态转换的条件是什么。

正例：订单状态有已下单、待付款、已付款、待发货、已发货、已收货等。比如已下单与已收货这两种状态之间是不可能有直接转换关系的。

4. 【强制】如果系统中某个功能的调用链路上的涉及对象超过 3 个，使用时序图来表达并且明确各调用环节的输入与输出。

说明：时序图反映了一系列对象间的交互与协作关系，清晰立体地反映系统的调用纵深链路。

5. 【强制】如果系统中模型类超过 5 个，并且存在复杂的依赖关系，使用类图来表达并且明确类之间的关系。

说明：类图像建筑领域的施工图，如果搭平房，可能不需要，但如果建造高楼大厦，肯定需要详细的施工图。

6. 【强制】如果系统中超过 2 个对象之间存在协作关系，并且需要表示复杂的处理流程，使用活动图来表示。

说明：活动图是流程图的扩展，增加了能够体现协作关系的对象泳道，支持表示并发等。

7. 【推荐】需求分析与系统设计在考虑主干功能的同时，需要充分评估异常流程与业务边界。

8. 【推荐】类在设计与实现时要符合单一原则。

说明：单一原则最易理解却是最难实现的一条规则，随着系统演进，很多时候，忘记了类设计的初衷。

9. 【推荐】谨慎使用继承的方式来进行扩展，优先使用聚合/组合的方式来实现。

说明：不得已使用继承的话，必须符合里氏代换原则，此原则说父类能够出现的地方子类一定能够出现，比如，“把钱交出来”，钱的子类美元、欧元、人民币等都可以出现。

10.【推荐】系统设计时，根据依赖倒置原则，尽量依赖抽象类与接口，有利于扩展与维护。

说明：低层次模块依赖于高层次模块的抽象，方便系统间的解耦。

11.【推荐】系统设计时，注意对扩展开放，对修改闭合。

说明：极端情况下，交付的代码都是不可修改的，同一业务域内的需求变化，通过模块或类的扩展来实现。

12.【推荐】系统设计阶段，共性业务或公共行为抽取出来公共模块、公共配置、公共类、公共方法等，避免出现重复代码或重复配置的情况。

说明：随着代码的重复次数不断增加，维护成本指数级上升。

13.【推荐】避免如下误解：敏捷开发 = 讲故事 + 编码 + 发布。

说明：敏捷开发是快速交付迭代可用的系统，省略多余的设计方案，摒弃传统的审批流程，但核心关键点上的必要设计和文档沉淀是需要的。

反例：为了业务快速发展，敏捷成了所有人催进度的借口，系统中均是勉强能运行但像面条一样的代码，可维护性和可扩展性极差，不久之后，不得不进行大规模重构，得不偿失。

14.【参考】系统设计主要目的是明确需求、理顺逻辑、后期维护，次要目的用于指导编码。

说明：避免为了设计而设计，系统设计文档有助于后期的系统维护，所以设计结果需要进行分类归档保存。

15.【参考】设计的本质就是识别和表达系统难点，找到系统的变化点，并隔离变化点。

说明：世间众多设计模式目的是相同的，即隔离系统变化点。

16.【参考】系统架构设计的目的：

确定系统边界。确定系统在技术层面上的做与不做。

确定系统内模块之间的关系。确定模块之间的依赖关系及模块的宏观输入与输出。

确定指导后续设计与演化的原则。使后续的子系统或模块设计在规定的框架内继续演化。

确定非功能性需求。非功能性需求是指安全性、可用性、可扩展性等。

2.9 日常约定

1. 【强制】所有的参数输入都必须遵循不可信的原则，必须做必要的校验，至少包括非空、格式和长度等。应避免只做前端校验，前端校验主要目的是用户体验，减少无效访问占用服务端性能，后端校验才是保证逻辑准确性的必要手段，

且校验与操作必须是在同一个不可拆分的调用中，以防止绕过校验直接调用。

2. 【强制】数据库表中的 ID 不能作为业务参数使用。

3. 【强制】所有的操作型业务入口都必须配置防止重复调用机制。例如，由业务前置下发全局唯一敏感参数，业务操作中对敏感参数做校验，防止多次提交。

4. 【推荐】接口设计中应该避免全静态参数。敏感接口中应该至少有一个动态参数，且对调用次数和调用有效期进行控制。

5. 【推荐】在状态或者分类的设计中，应避免对未知的状态和分类进行定义，应使所有的可能性在逻辑的掌控之下。例如，在类似通过判断状态或者标识来确定业务流向的场景下，应该仅对确定的状态进行判断和分类，而避免使用“其他”来归类。在进行判断时，对每一种确定的分类使用“if”，而少用或者不用“else”，因为状态或者标识在未来可能会增加，新增的状态或者表示可能会是完全不同的另外一种意义，而如果使用了 else，在判断逻辑没有及时且正确的调整的情况下，可能会发生严重的不可掌控的业务错误。

6. 【强制】对外生成唯一性标识时，应该避免使用可观察可推测的规则。例如订单号、商户编号、邀请码等标识在生成时应该避免使用序号自增这样的简单规则，应该至少包含一种复杂且无法逆向推导的因子。

7. 【强制】在逻辑与算法设计时，不能将不确定因素作为影响核心和关键结果的条件。例如，在需要绝对唯一的标识符的场景下，应在严谨的唯一性算法作为核心因子的基础上产生，而不能将时间戳或随机数等参数以重复的可能性低作为理由来当做算法的参考因素。例如，在设计查询业务的实现方案时，不能以当前数据量小作为固定条件，导致在数据量小的前提下业务能正常运转，而数据量发生变化时就会出现异常甚至无法提供服务。

8. 【推荐】任何可能发生变化的参数都应设计为动态配置。

9. 【推荐】任何可组合和变化的能力和入口，都应考虑设计为动态配置，包括开关、组合和包装等。

10. 【强制】任何属性、能力等状态性的标识，不宜以过程性的记录作为判断的

关键条件。例如某个用户是否开通某项业务，不能以流水表是否存在某条数据作为判断条件，因为流水表一般数据量增大后会进行归档，只保留近期的一部分数据，如果以是否存在某条历史数据作为判断依据，明显是不准确的。

11. 【强制】任何具有时空顺序性的逻辑流程在执行时都应在同一个原子性操作中校验当前是否满足时空顺序特征。例如,一笔业务流程或一笔订单，状态是初始化-->处理中-->处理成功/处理失败的顺序，在业务处理完成后将订单状态修改为处理成功时就应该校验当前的状态只能为“处理中”（update 时应 where 条件中必须存在 status=“处理中”），以防止复杂情况下出现业务越权的情况；例如，一笔入账记录在审核通过之后会向用户账户充值一笔资金，在审核操作 update 时 where 条件也必须存在 status=“待审核”，否则并发情况下可能存在同一笔记录审核多次引起多笔入账的严重问题；例如，在进行账户余额扣款操作时，虽然操作前会对账户余额进行查询和比较验证，但是验证与扣款操作并非在同一个原子操作中，所以在扣款操作的 update 的 where 中也应该存在一个条件 balance>=扣款金额，防止并发或复杂情况下出现扣款后余额为负数的严重错误。

12. 【推荐】敏感的接口或方法都应该对调用的合法性做校验。例如，通过对调用方 ip、经纬度等参数校验防止未授权的访问；通过对参数加密防止重要数据泄露；通过对参数进行签名防止数据被篡改；通多对调用次数、有效期和频率的控制防止不合理的访问等等。

13. 【参考】方法或者业务复用时，应同时满足共性和特性，遵循最小化修改的原则。例如，包含多个参数修改的 SQL 脚本如果作为公用方法，方法本身应该根据业务场景作为条件做动态调整，满足在不同场景下调用时只修改自身业务的参数，而非固定脚本修改方法中所有的参数，因为方法中包含非调用场景下的业务参数。

14. 【强制】禁止使用同步代码块、同步方法实现业务的同步性要求。同步的核

心是数据而非过程，同步的代码并不能解决相似业务和集群部署场景下共享数据的同步问题，却会导致严重的性能问题。

15. 【强制】接口或方法，应遵循定义单一原则，输入输出参数的设计应只符合明确的业务定义，禁止将数据容器（数据库表、数据文件、数据对象等）中的所有数据不做设计和约束的直接映射到输出中。

16. 【强制】数据结构设计时，数据类型的选择必须将数据量的预期作为参考值，且相同的数据字段定义必须使用相同的数据类型，包括表与表，表对对象。

17. 【推荐】应使用稳定的实体数据对象代替数据集作为数据传递的载体。例如，使用 POJO 实体类代替 Map，将数据关系暴露在编译期而非运行时，利于数据流向追查和问题追溯。

18. 【强制】数据库表中必须存在 id 自增列、创建时间和修改时间字段，且修改时间必须为自动更新，大部分情况下可提前预留适量的字段作为扩展备用。

19. 【参考】请求入口设计（action/controller）时，应遵循分层实现的基本原则，并将层与层解耦，强关联类型数据结构不作为层与层之间的交互参数。例如，action 只做请求的基本校验，不做核心业务，业务逻辑由 service 层实现，且 action 不宜将 request、session、response 等专有数据结构作为与 service 的业务交互参数，单元测试时影响明显。

20. 【强制】接口中存在金额、日期、时间等数字类型参数时，必须约定并显式说明数据的格式和单位，时间格式模板中应该特别注意字母大小写代表的不同意义。相同业务体系中，应该尽量保持统一的标准。例如，A 系统的 1 号接口中金额是以分为单位的整数类型，2 号接口中的创建时间为 yyyyMMddHHmmss 格式，则 A 系统所有接口中的金额和时间参数都应尽量保持这两种格式。

21. 【强制】禁止一切锁表操作。

3 代码管理

3.1 代码权限

(1) 新项目统一由自有 Git（GitLab）仓库托管代码；
(2) Git 服务账号由专有管理员统一分配和管理；
(3) 代码禁止互联网托管与共享；

3.2 分支管理

(1).源代码版本库（每个工程）固定建立【开发分支 develop】；

(2).项目启动开发后，开发经理从 develop 分支建立项目分支【feature_时间+项目名】，开发人员拉取项目分支到本地开发；

(3).开发完成后，开发人员提交各自代码到项目分支，解决可能的文件冲突，并提交更新文件目录清单；

(4).开发经理根据开发人员提交的更新文件目录清单，检查实际提交代码内容与清单是否一致（不仅检查文件名，还需对文件更新内容做检查，是否存在与本项目不相关的内容）。

(5).检查通过后，开发经理汇总所有开发人员的提交内容和文件清单，形成本项目最终提交文件清单.

(6).开发负责人，根据项目经理和测试经理提供的上线顺序和版本规划，确定上线版本，以项目分支为基础合包分支【feature_时间+项目 1+项目 2】，并在此分支上打包提测给测试部门；

(7).测试阶段，在开发版本分支【feature_时间+项目名】上修复问题，自测完成后打包时合到合包分支【feature_时间+项目 1+项目 2】进行打包，测试结束后，还需解决并行项目上线顺序变动可能引起的代码问题；

(8).上线后，开发经理再次确认更新文件清单与版本库代码记录，将合包分支【feature_时间+项目 1+项目 2】合并到【develop 分支】上；

(9).开发分支 feature_项目名，合包分支 feature_时间+项目 1+项目 2，作为临时分支，在

上线完成后可以保留一段时间，待线上稳定后评估一下，没有必要保留的临时分支可以直接删除。

3.3 代码合并

本部分可作为上一部分【分支管理】的补充和扩展，作为具体项目实施时操作的指导准则。

(1) 同一个项目不同系统在建立分支时，应使用相同的分支名称，且分支名称可以一定程度反映和区别项目内容。合并后的分支名称应该体现所合并的分支的内容。A 分支和 B 分支合并后的分支名称建议为 A+B，从分支名称上就可以快速分辨这是个合并分支，且合并了哪些分支。

(2) 项目上线后，从项目经理处获得上线内容是否正常且稳定，稳定之后由开发经理显式确认所有代码均已提交，由开发经理或者开发经理指定人员合并到线上分支。注意，在提交代时，应再次更新代码，确保当前提交的代码为最新版本。

(3) 上线前需要重新确认上线的版本和分支内容。

a. 测试人员或者项目经理根据上线安排确认之前的版本计划是否有调整和变化，提测的包是否需要重新打包。

b. 开发人员根据当前上线顺序确定是否需要合并或者回退分支，确定最终上线版本需要包含的内容。

c. 开发人员根据最终版上线内容，对比打包的版本分支与线上分支差异，确保没有遗漏或多余的误提交文件，保证计划于实际相符。如果最终上线版本包含多个项目，则需要各自项目的开发人员来合作共同完成对比的工作。

3.4 代码评审

开发完成后，为了尽可能尽早发现代码本身的问题，避免减少代码引起的关联性问题，需要对代码做评审，以保证代码的质量和稳定性，同时保证相关联开发人员获得足够的信息和认知度。

代码评审的方式有召开代码评审会、开发经理复审和结对编程三种方式，由开发经理根据项目的体量、复杂度和人员构成决定代码评审的方式。

(1) 代码评审会

由开发经理判断和召集参会人员，要做到开发角色横向和纵向关联的所有人员都参与进来。

横向关联是指当前项目中和本模块有直接或者间接关联的开发人员，一般包括当前项目

中包括的大部分甚至全部开发人员，例如本项目涉及两个系统三个开发人员，这个开发人员的代码有直接或者间接的关联关系。

纵向关联是指本项目之外与当前模块有直接或间接关联的开发人员，一般包括当前代码所属系统的其他未参加本项目的开发人员，例如本项目涉及两个系统，1 号开发人员负责 A 系统内容开发，那同样负责 A 系统但没有参与本项目的 3 号开发人员就属于 1 号开发人员的纵向关联。

代码评讲时，由代码作者按逻辑流程讲解代码内容，所有评审人员严格仔细的检查和发现代码是否存在问题。开发经理安排每一位开发人员讲解时需要哪些评审者参与，既能达到评审的目的，又能提高效率节省时间和人力成本。

(2) 开发经理复审

对于内容较少，复杂度低，参与开发人员少的项目，可以由开发经理对代码进行复审，以完成代码评审的目的，具体的操作方式由开发经理与开发人员决定。

(3) 结对编程

对于符合结对编程条件，结对编程对项目有积极作用同时成本与不利因素在可接受范围时，由开发经理安排部署结对编程的人员和任务。

结对开发人员需要共同分担开发任务，而不是两个人做一个人的工作内容，且要与结对的开发人员有足够的认知共性，可以融洽且快速的融入对方的代码内容中去，高效且高质量的完成与对方的代码监督作用，又不会大量浪费人力资源。

4 流程管理

4.1 技术评审

(1) 可行性分析

需求评审阶段，技术人员需要对需求的技术可行性做分析与评估。

a. 原则上技术实现做最大化考虑。除非能明确判定属于现阶段确实无法实现的难点，否则不能以“不能实现”作为理由影响产品设计。

b. 技术实现成本可作为可行性的关键因素。实际中，对于实现成本过高的需求和设计，成本可作为考虑因素来影响需求设计，在成本远大于需求价值时，技术人员需反馈至产品经理和项目经理，综合考虑得出最优结论。

(2) 实现方案设计

在可行性分析通过后，开发团队需要对实现的技术方案做评估与设计。

a. 方案设计前必须完全确认需求内容，避免遗漏与错误理解。

b. 设计由开发经理负责牵头与跟进，开发经理必须至少作为主要人员，设计团队可以有多人，但开发经理必须非常清楚与了解设计内容。

c. 设计方案必须是一个完整的闭环，本次涉及的内容必须在本次实现中有个交代和结果，不能留黑洞。实现成本过高，或实现周期特别紧的情况下，可分期或分批涉及实现，每一期或每一批都是对前面的完善和优化，而不是完全没有或者不处理。方案必须闭环，当时可以不完善，后期优化至完整，但每一步骤必须都具备完整的功能方案和处理能力，而不是缺失。

d. 技术方案设计时，需要进行一定程度的抽象和全局考虑，除了应对本次需求之外，还需具备长期整合的灵活性和适应大量复杂业务的扩展性，降低维护成本。

e. 技术设计时，还必须考虑几个关键因素：存量数据的影响、性能瓶颈、安全风险和用户体验，综合所有要素评估出整体最优方案。

(3) 设计文档输出

对于项目前期的可行性分析评估、技术方案设计、开发过程管控要点等重点内容，都要记录在有效载体上存档积累下来，以便于后续维护时的查阅。

设计文档必须达到或者包括：可行性分析的考虑因素和结果，方案设计的具体内容（包括整体架构、业务流程、逻辑结构、数据模型等等，还可以包括性能预期与应对、抽象与扩展设计、安全风险设计等等）、开发管控或目标关键等。

设计文档提供建议模板，格式不做强制要求，但必须结构清晰数据准确内容明了。

设计文档必须由开发经理作为第一作者负责完成。

设计文档的输出参考项目管理过程条例中关于项目等级与输出标示的说明，特殊情况时相关负责人达成一致的情况下，可灵活处理。

4.2 周期预估

(1) 开发周期

项目启动开发后，基于项目管理更好的管控，需要评估并提供开发工作量与周期。

a. 需求评审与技术评审之后，由开发经理根据需求内容与实现方案，将需求进一步分解为多个任务，并将任务分配给开发人员。

b. 开发人员对每个任务进行工作量评估。

c. 开发经理对每个任务进行工作量评估。

d. 开发经理将自己的评估结果与开发人员评估结果进行对比，两者有差异的情况下，开发经理主动与开发人员进行沟通，寻找差异的原因并讨论达成一致，得出两者皆认可的工作量。

e. 开发经理汇总每个任务的工作量，综合人员分配、人员占用、项目优先级等因素得出工作量分布，再结合实际情况整理出项目开发周期。

(2) 测试周期

项目提测后，开发人员需要在整个测试周期内关注测试情况，并尽快解决测试问题。测试周期一般由测试人员提供。开发人员需及时配合高效处理，保证整个测试周期在测试人员提供的范围内，保证测试结果在在测试的可控范围内。

特殊情况下，由相关人员根据实际讨论灵活处理。