

一、变成规范

(一) 命名风格

1. 【强制】代码中的命名均不能以下划线或美元符号开始，也不能以下划线或美元符号结束。
反例：`_name` / `__name` / `$name` / `name_` / `name$` / `name__`
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。

批注 [张1]: 待确定

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。

批注 [张2]: 待定

(二) 常量定义

1. 【强制】不允许任何魔法值（即未经预先定义的常量）直接出现在代码中。

反例：

```
String key = "Id#taobao_" + tradeId;
cache.put(key, value);
```

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;
    }
}
```

(三) 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。

(四) OOP 规约

1. 【强制】避免通过一个类的对象引用访问此类的静态变量或静态方法，无谓增加编译器解析成本，直接用类名来访问即可。

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 方法容易抛出指针异常，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，避免反序列化失败；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 出一个 StringBuilder 对象，然后进行 append 操作，最后通过 toString 方法返回 String 对象，造成内存资源浪费。反例：

```
String str = "start";
for (int i = 0; i < 100; i++)
{
    str = str + "hello";
}
```

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，方便更好地进行重构。

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。

(五) 集合处理

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

1) 只要重写 `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，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[] { "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。

(六) 并发处理

1. 【强制】获取单例对象需要保证线程安全，其中的方法也要保证线程安全。

说明： 资源驱动类、工具类、单例工厂类都需要注意。

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。

(七) 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。

(八) 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。

(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 $0 \leq x < 1$ （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、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】** 发帖、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。

三、MySQL 数据库

(一) 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】**超过四个表禁止 `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。

(二) SQL 语句

1. 【强制】不要使用 `count(列名)` 或 `count(常量)` 来替代 `count(*)`，`count(*)` 是 SQL92 定义的标准统计计数的语法，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。

(三) 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】iBATIC 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。

