目錄

JAVA开发规范约定

————无规矩不成方圆

标记说明：

- 【强制】—— 必须无条件遵循
- 【推荐】—— 最好按照遵循
- 【参考】—— 不强制要求，可参考

参考资料：

- 《阿里巴巴JAVA开发规范》
- Spring 官方文档
- Rest API 设计规范

通用规约

参考阿里巴巴集团JAVA开发规范整理。

一、命名约定

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

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。)

```
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。
```

3. 【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 / DTO / VM/VO等。

```
正例: UserService / UserDTO / Product / TcpUdpDeal / TaPromotion

反例: Users / macroPolo / UserDto / XMLService / TCPUDPDeal / TAPromotion
```

4. 【强制】类名尽量要以英文名词单数形式命名，类名不要加Info,Table,Detail等后缀；但是类名如果有复数含义，类名可以使用复数形式。

```
正例： Order / OrderServcie / Item / Address / StringUtils

反例： OrderInfo / Users / UserTable
```

5. 【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，并且方法名要以英文动 词开头，变量为英文名词形式并区分好单、复数。

```
正例: localValue / getHttpMessage() / inputUserId ／update
```

6. 【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
```

7. 【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。

8. 【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。

9. 【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。

10. 【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。

11. 【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。
```

12. 【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;
```

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 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。
```

14. 【强制】接口和实现类的命名，基于面向SOA的理念，暴露出来的服务一定是接口，接口要以Service为后缀，内部的实现类用 Impl 的后缀与接口区别。

```
正例:CacheServiceImpl 实现 CacheService 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常量类，且构造方法被默认强制是私有。

```
正例:枚举名字:OrderStatusEnum，成员名称:ORDERED / SHIPPED / PENING / PAID
```

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。

3) 获取统计值的方法用count做前缀。

4) 插入的方法用save(推荐)或insert做前缀。

5) 删除的方法用remove(推荐)或delete做前缀。

6) 修改的方法用update做前缀。
```

B、领域模型命名规约

```
1) 数据对象:xxxDO，xxx即为数据表名。

2) 数据传输对象:xxxDTO，xxx为业务领域相关的名称。

3) 展示对象:xxxVO 或 xxxVM，xxx一般为网页名称。

4) POJO是DO/DTO/BO/VO的统称，禁止命名成xxxPOJO。
```

二、常量定义

1. 【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);
```

2. 【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?
```

3. 【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。
```

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。

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);
      }
```

三、格式规约

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;表示终止右大括号后必须换行。

2. 【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。

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

4. 【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、三目运行符等。

5. 【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。

6. 【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) 在多个参数超长，逗号后进行换行。

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，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, ar
gs3, ...
, argsX);
```

7. 【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");

8. 【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

9. 【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。

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。
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 User getUsers(String type, Integer... 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。
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。
8. 【强制】关于基本数据类型与包装数据类型的使用标准如下:

```
    1) 所有的POJO类属性必须使用包装数据类型。

    2) RPC方法的返回值和参数必须使用包装数据类型。

    3) 所有的局部变量【推荐】使用基本数据类型。
```

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 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。

13. 【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\(
    ary.length\);
    ```

14. 【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。

15. 【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。

16. 【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; }
    }
    ```

17. 【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&lt;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。

18. 【推荐】下列情况，声明成 final 会更有提示性:

1) 不需要重新赋值的变量，包括类属性、局部变量。

2) 对象参数前加final，表示不允许修改引用的指向。

3) 类方法确定不允许被重写。

19. 【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。

20. 【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，必须考虑是否为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。

五、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，即使它什么代码也没有。
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，避免使用 下面的形式:if (condition) statements;
3. 【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。
```

4. 【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。

正例:

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

反例:

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

5. 【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)。

6. 【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。

7. 【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。
```

8. 【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。

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。

六、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信息。

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】注释掉的代码尽量要配合说明，而不是简单的注释掉。

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。

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。

MYSQL规约

一、建表规约

1. 【强制】表名、字段名必须使用小写字母或数字，多单词采用蛇形命名法（下划线分割）;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。

   ```
   正例:getter_admin，task_config，level3_name

   反例:GetterAdmin，taskConfig，level_3_name
   ```

2. 【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。

3. 【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)。

4. 【强制】任何字段如果为非负数，必须是 unsigned。

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

6. 【强制】唯一索引名为 uk_字段名;普通索引名则为 idx_字段名。

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

7. 【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 超出decimal 的范围，建议将数据拆成整数和小数分开存储。
   ```

8. 【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。

9. 【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。

10. 【强制】表必备三字段:id, created_date, *last_modified*_date。 说明:其中id必为主键，类型为unsigned bigint、单表时自增、步长为1。created_date, *last_modified*_date 的类型均为 date_time 类型。

11. 【推荐】表的命名最好是加上“业务名称_表的作用”。

正例: wms_task / erp_product / mpp_config

12. 【推荐】库名与应用名称尽量一致。

13. 【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。

14. 【推荐】字段允许适当冗余，以提高性能，但是必须考虑数据同步的情况。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。

15. 【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。

16. 【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: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	表示范围
人	150岁之内	unsigned tinyint	无符号值:0 到 255
龟	数百岁	unsigned smallint	无符号值:0 到 65535
恐龙化石	数千万年	unsigned int	无符号值:0 到约 42.9 亿
太阳	约50亿年	unsigned bigint	无符号值:0 到约 10 的 19 次方

二、索引规约

1. 【强制】业务上具有唯一特性的字段，即使是组合字段，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和控制，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，数据类型保持绝对一致;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。

   ```
   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，来避免回表操作。

   ```
   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:主键索引、唯一索引、普通索引，而覆盖索引是一种查询的一种 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=? ，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判断条件时，在建索引时，请把等号条件的列前置。如:where a>? and b=? 那么即使 a 的区分度更高，
也必须把 b 放在索引的最前列。

10. 【参考】创建索引时避免有如下极端误解:

1)误认为一个查询就需要建一个索引。

2)误认为索引会消耗空间、严重拖慢更新和新增速度。

3)误认为唯一索引一律需要在应用层通过“先查后插”方式解决。

三、SQL规约

1. 【强制】不要使用 count(列名)或 count(常量)来替代 count(*)，count(*)就是 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 之外的不重复数量。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。注意:NULL 与任何值的直接比较都为 NULL。 说明:

```
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:
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 studen
t_id，
同时触发成绩表中的 student_id 更新，则为级联更新。 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. 【推荐】对数据量很大的几张表应避免关联查询，应分解成几个单表查询。应在应用层建立关联关系。如下示例：

```
SELECT * FROM tag JOIN tag_post ON tag_post.tag_id = tag.id

                                   JOIN post  ON tag_post.post_id = post.id

                                   WHERE   tag.tag = 'mysql';
```

```
SELECT * FROM tag WHERE tag.tag = 'mysql';

SELECT * FROM tag_post WHERE tag_post.tag_id = 1234;

SELECT * FROM post   WHERE post.id IN (123,456,789,901)；
```

8. 【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。
9. 【强制】数据订正时，删除和修改记录时，要先 select，避免出现误删除，确认无误才能执行更新语句。
10. 【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。
11. 【参考】如果有全球化需要，所有的字符存储与表示，均以 utf-8 编码，那么字符计数方法 注意:

```
    说明:

       SELECT LENGTH("轻松工作"); 返回为12

       SELECT CHARACTER_LENGTH("轻松工作"); 返回为4 如果要使用表情，那么使用 utfmb4 来进行存储，
       注意它与 utf-8 编码的区别。
```

12. 【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。
```

异常日志

一、异常处理

1. 【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){...}
   ```

2. 【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。

3. 【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。

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，可以使用 try-with-resources 方式。

7. 【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。

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。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。

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。

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，一律要求进行NPE判断。

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。

11. 【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:

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。

12. 【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。

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){...}
```

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。

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，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。
```

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 + " 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)

    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);
```

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，那么往上 抛。

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

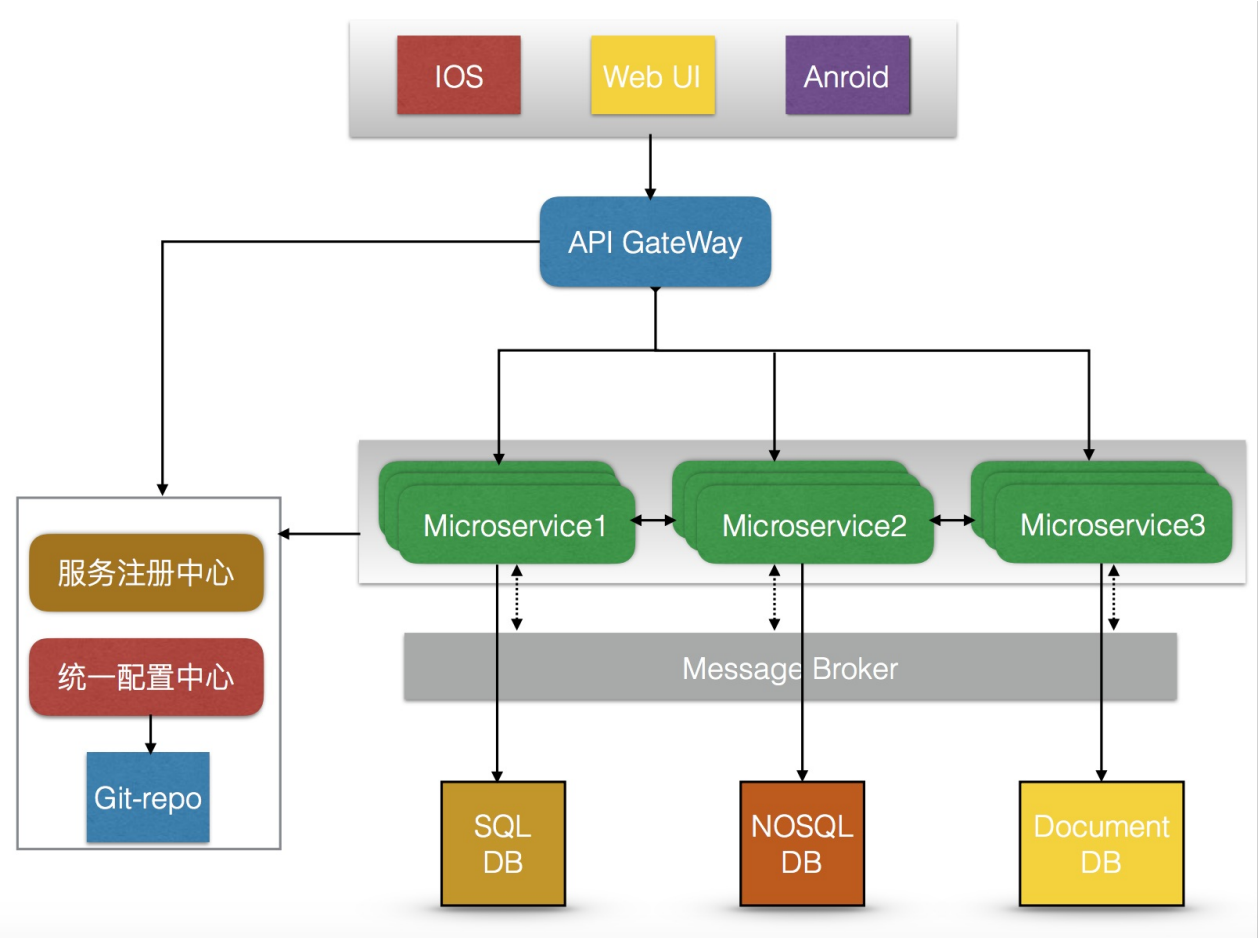
7. 【推荐】可以使用 warn 日志级别来记录用户输入参数错误的情况，避免用户投诉时，无所适 从。注意日志输出的级别，error 级别只记录系统逻辑出错、异常等重要的错误信息。如非必 要，请不要在此场景打出 error 级别。

8. 【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?

工程规约

一、微服务架构

1. 【推荐】依图中所示，微服务API统一通过GateWay路由代理对方开放给外部各种客户端使用，微服务之间内部可以互相调用。



1. 【推荐】各微服务之间调用在满足业务需要前提下，最好以事件驱动的消息模式来互相通信。

2. 【强制】微服务之间通过请求响应模式调用时，在消费端必须加入故障容错处理，在请求失败时作出合理响应，不允许重试。

正例：

```
@HystrixCommand(fallbackMethod = "findEmptyMovies")
public List<Movie> findMoviesByCustomer(@PathVariable Long customerId) {
        /**远程调用用户服务*/
        Customer customer = getCustomer(customerId);
        if (customer==null) return new ArrayList<>();
            return customer.getAge()> 18 ? movieRepository.findAll() : movi
eRepository.findByLevel("P");
        }
  public List<Movie> findEmptyMovies(Long customerId){
        logger.info("customer services was shut down !");
        return new ArrayList<>();
    }
```

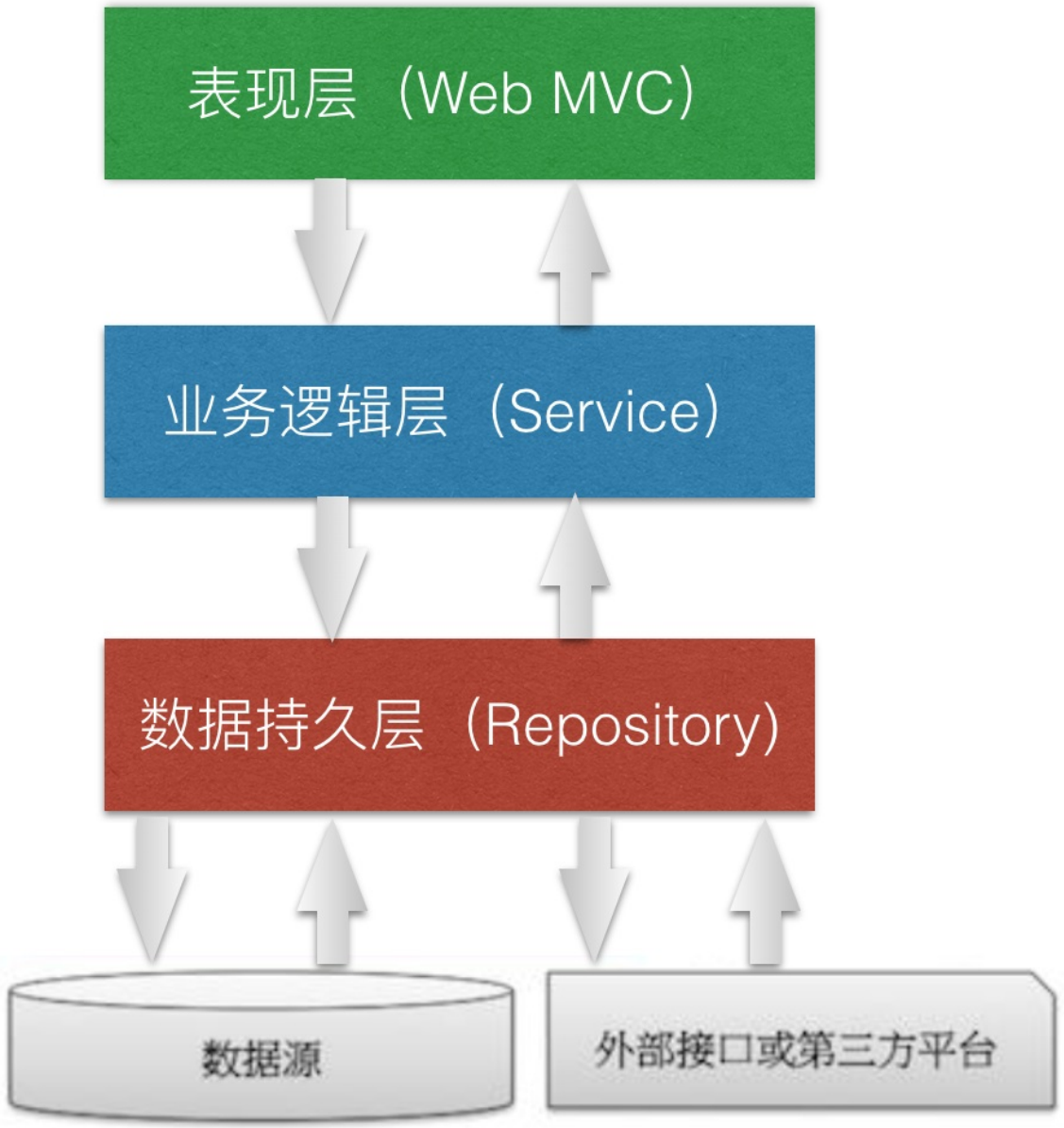
3. 【推荐】微服务API之间调用，HTTP客户端推荐使用Feign，其次RestTemplate，不要自己封装或第三方HTTP CLIENT。

正例：

```
@FeignClient("customer-services")
interface CustomerService{
    @RequestMapping(method = RequestMethod.GET, value = "／customers/{customerId}"
,
            consumes = "application/json")
   Customer getCustomer(@PathVariable("customerId") Long customerId);
}
```

二、应用分层

1. 【强制】每个微服务自身就是一个完整的Web应用，可以独立开发部署。采用常见的三层体系结构Web层、Service层、数据持久层。依下图箭头所示，自上而下产生依赖关系，下层不允许依赖上层。



- 【强制】Web 层:主要是对访问控制进行转发，各类基本参数校验，或者不复用的业务简单处理等。
- 【参考】Service 层:相对具体的业务逻辑服务层。
- 【参考】Manager 层:通用业务处理层，它有如下特征:

```
1) 对第三方平台封装的层，预处理返回结果及转化异常信息;

2) 对Service层通用能力的下沉，如缓存方案、中间件通用处理;

3) 与DAO层交互，对DAO的业务通用能力的封装。
```

2.【强制】Repostory 层:数据访问层，与底层 MySQL、Oracle、Hbase 进行数据交互。

3.【参考】(分层异常处理规约)在 DAO 层，产生的异常类型有很多，无法用细粒度异常进行 catch，使用 catch(Exception e)方式，并 throw new DAOException(e)，不需要打印日志，因为日志在 Manager/Service 层一定需要捕获并打到日志文件中去，如果同台服务器再打日志，浪费性能和存储。在 Service 层出现异常时，必须记录日志信息到磁盘，尽可能带上参数信息，相当于保护案发现场。如果 Manager 层与 Service 同机部署，日志方式与 DAO 层处理 一致，如果是单独部署，则采用与 Service 一致的处理方式。Web 层绝不应该继续往上抛异常，因为已经处于顶层，无继续处理异常的方式，如果意识到这个异常将导致页面无法正常渲染，那么就应该直接跳转到友好错误页面，尽量加上友好的错误提示信息。开放接口层要将异常处 理成错误码和错误信息方式返回。7

4.【参考】分层领域模型规约:

```
Entity(Domain Object):与数据库表结构一一对应，通过 Repository 层向上传输数据源对象。

DTO(Data Transfer Object):数据传输对象，Service 和 Manager 向外传输的对象。

VM(View Model):显示层对象，通常是 Web 向前端传输的对象。
```

三、命名规约

1. 【强制】各服务模块工程名统一 以服务名—service 形式命名。如 blog-service ，user-service，order-service 等。
2. 【强制】config/ 包下存放系统配置类，自定义配置类可以重载Spring-Boot的各项默认配置。工程里的初始配置类原则上不允许擅自修改，可以添加配置类。配置类名前要加**@Configuration** 注解，配置类必须以 **Configuration** 作后缀。
3. 【推荐】工程里默认生成配置属性类 ApplicationProperties 位于 config/，所有属性项在这个类里添加，嵌套属性要在该内部类增加静态类进行属性获取。建议配置文件采用 YAML 格式文件，语义、可读性更友好。

```
foo:
  enabled : true
  servers : www.host1.com,ww.host2.com
  list:
    - name: my name
      description: my description
```

正例：

```
@ConfigurationProperties("foo")
public class ApplicationProperties {
  private boolean enabled;
  private List<String>servers;
  private final List<MyPojo> list = new ArrayList<>();
  public static class MyPojo{
        private String name;
        private String description;
        /***省略 get/set 方法***/
  }
  public List<MyPojo> getList() {
        return this.list;
   }
   /***省略 get/set 方法***/
}
```

5. 【强制】实体类必须放置在 domain/ 包下，类名前添加注解**@Entity** 或 **@Document** 根据数据源类型来选择，SQL选用@Entity，MongoDB选用@Document。属性前不加@Column 的话，默认生成的数据列名时属性的全小写单词，要求多单词属性添加**@Column**定义列名为下划线分割，或属性与列名不符时也要通过**@Column**自定义列名，跟数据库实际列名对应一致。
6. 【强制】service类必须放置在 service/ 包下，类名前必须通过**@Service** 注解，原则上必须先定义接口再定义实现类，业务接口以 **Service** 为后缀，业务实现类以 **ServiceImpl** 为后缀。DTO相关类要放置在 dto/ 包下，类名以 DTO 为后缀。
7. 【强制】数据库操作类放在 repository/ 目录下，名称后面以 Repository 为后缀。
8. 【强制】Rest API 统一放置在 web/rest 包下，统一以 Resource 为后缀名，若有 VM 对象，应放在vm/ 包下面，以VM为后缀。

四、ORM规约

1. 【推荐】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) 返回不必要的多余字段，增加额外网络传输数据。

2. 【强制】POJO 类的 Boolean 属性不能加 is，而数据库字段必须加 is_ 。

3. 【强制】禁止使用本地SQL查询，可以用JPA 命名查询 @Query 代替，本地SQL查询与数据库类型绑定，不利于扩展迁移。

   正例：

   ```
   public interface BlogRepository extends JpaRepository<Blog,Long>{

                   @Query("select blog from Blog blog where blog.user.login = ?#
   {principal.username}")

                   List<Blog> findByUserIsCurrentUser();

   }
   ```

   反例：

   ```
   public interface BlogRepository extends JpaRepository<Blog,Long> {

                   @Query(nativeQuery=true,value="select * from BlogTable blog b
   log , UserTable user

                                   where blog.user_id = user.id and user.id = ? ")

                   List<Blog> findByUserIsCurrentUser( Long userId);

   }
   ```

4. 【推荐】在实体类之间有一对多，多对多关系，尽量避免双向关联，尽量在多的一侧关联，默认 为懒加载，根据需要在业务实现时，在业务服务类例根据需要决定是否加载，禁止 FetchType.EAGER。

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

6. 【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，尽量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。

7. 【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，在业务服务类例更新多个实体时，必须显性的在方法添加 @Transactional。

8. 【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。

五、MAVEN规约

1. 【强制】定义 GAV 遵从以下规则：

```
1）GroupID格式:com.{公司/BU }.业务线.[子业务线]，最多4级。

正例:com.joymef.platform 或 com.joymef.social.blog

2）ArtifactID格式:产品线名-模块名。语义不重复不遗漏，先到仓库中心去查证一下。

正例:user-service / user-client / blog-service ) Version:详细规定参考下方。
```

2. 【强制】MAVEN库版本号命名方式:主版本号.次版本号.修订号

```
主版本号:当做了不兼容的API 修改，或者底层依赖框架变更，或者增加了能改变产品方向的新功能。

次版本号:当做了向下兼容的功能性新增(新增类、接口等)。

修订号:修复 bug，没有修改方法签名的功能加强，保持 API 兼容性。

说明:起始版本号必须为:1.0.0，而不是 0.0.1
```

3. 【强制】开发阶段版本号定义为SNAPSHOT,发布后版本改为RELEASE。

4. 【强制】线上应用不要依赖 SNAPSHOT 版本(安全包除外);正式发布的类库必须先去中央仓 库进行查证，使 RELEASE 版本号有延续性，版本号不允许覆盖升级。

```
说明:不依赖 SNAPSHOT 版本是保证应用发布的幂等性。另外，也可以加快编译时的打包构建。

当前版本:1.3.3，那么下一个合 理的版本号:1.3.4 或 1.4.0 或 2.0.0
```

5. 【强制】依赖于一个第三方库群时，必须定义一个统一的版本变量，避免版本号不一致。

```
说明:依赖 springframework-core,-context,-beans，它们都是同一个版本，

可以定义一 个变量来保存版本:${spring.version}，定义依赖的时候，引用该版本。
```

6. 【强制】禁止在子项目的 pom 依赖中出现相同的 GroupId，相同的 ArtifactId，但是不同的 Version。

7. 【推荐】所有 pom 文件中的依赖声明放在<dependencies>语句块中，所有版本仲裁放在<dependencyManagement>语句块中。

说明:<dependencyManagement>里只是声明版本，并不实现引入，因此子项目需要显式的声 明依赖，

version 和 scope 都读取自父 pom。而<dependencies>所有声明在主 pom 的 <dependencies>里的依赖都会自动引入，

并默认被所有的子项目继承。

8. 【参考】为避免应用二方库的依赖冲突问题，二方库发布者应当遵循以下原则:

1)精简可控原则。移除一切不必要的 API 和依赖，只包含 Service API、必要的领域模型对 象、Utils 类、常量、枚举等。

如果依赖其它二方库，尽量是 provided 引入，让二方库使用 者去依赖具体版本号;无 log 具体实现，只依赖日志框架。

2)稳定可追溯原则。每个版本的变化应该被记录，二方库由谁维护，源码在哪里，都需要能 方便查到。

除非用户主动升级版本，否则公共二方库的行为不应该发生变化。

六、其他

1. 【推荐】工具包，系统默认引入了Apache commons 提供的工具包 commons-lang3、commons-beanutils、commons-io等等非常好用的utils，需要可以自己查找，不要重复造轮子。
2. 【推荐】另外spring framework 框架里也提供了许多的utils，里面也封装了许多常用的工具类，字符串、编码、加密解密、日期处理、数学计算、IO 等常用功能。

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、操作别人的数据，比如查看、修改别人的订单。
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。
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。

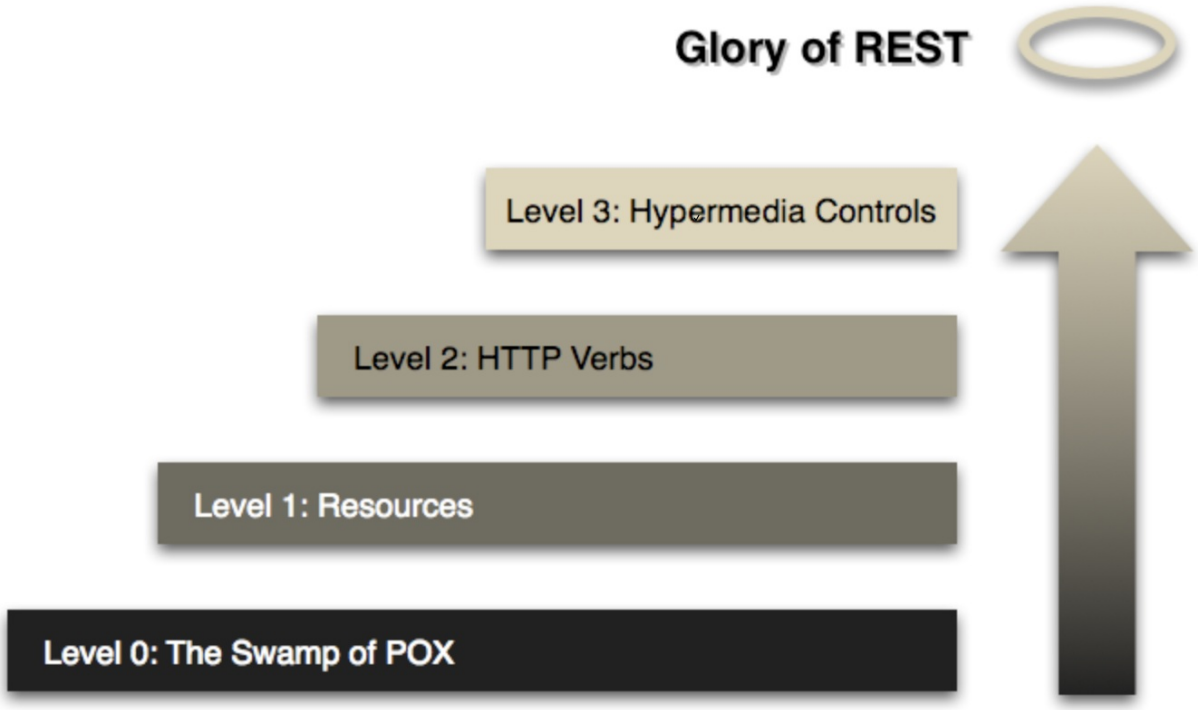
```
    说明:

    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。
```

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。

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。

REST API 规约



1. 【参考】RESTfull架构定义：

   （1）每一个URI代表一种资源；

   （2）客户端和服务器之间，传递这种资源的某种表现层；

   （3）客户端通过四个HTTP动词，对服务器端资源进行操作，实现"表现层状态转化"。

2. 【推荐】URI 正确设计应不包含动词，因为"资源"表示一种实体，所以应该是名词，URI不应该有动词，动词应放在HTTP协议中。

   反例： `GET /posts/show/1`

   正例： `POST /accounts/1/transfer/500/to/2` `PUT /posts/1 { ....}`

3. 【推荐】URI 资源名词应是复数形式，针对增删改查使用正确的HTTP 动作

正例：

GET /zoos：列出所有动物园

POST /zoos：新建一个动物园

GET /zoos/${ID}：获取某个指定动物园的信息

PUT /zoos/${id}：更新某个指定动物园的信息（提供该动物园的全部信息）

PATCH /zoos/${ID}：更新某个指定动物园的信息（提供该动物园的部分信息）

DELETE /zoos/${ID}：删除某个动物园

GET /zoos/${ID}/animals：列出某个指定动物园的所有动物

DELETE /zoos/${ID}/animals/ID：删除某个指定动物园的指定动物

4. 【推荐】过滤信息，如果记录数量很多，服务器不可能都将它们返回给用户。API应该提供参数，过滤返回结果。

正例：

?limit=10：指定返回记录的数量

?offset=10：指定返回记录的开始位置。

?page=2&per_page=100：指定第几页，以及每页的记录数。

?sortby=name&order=asc：指定返回结果按照哪个属性排序，以及排序顺序。

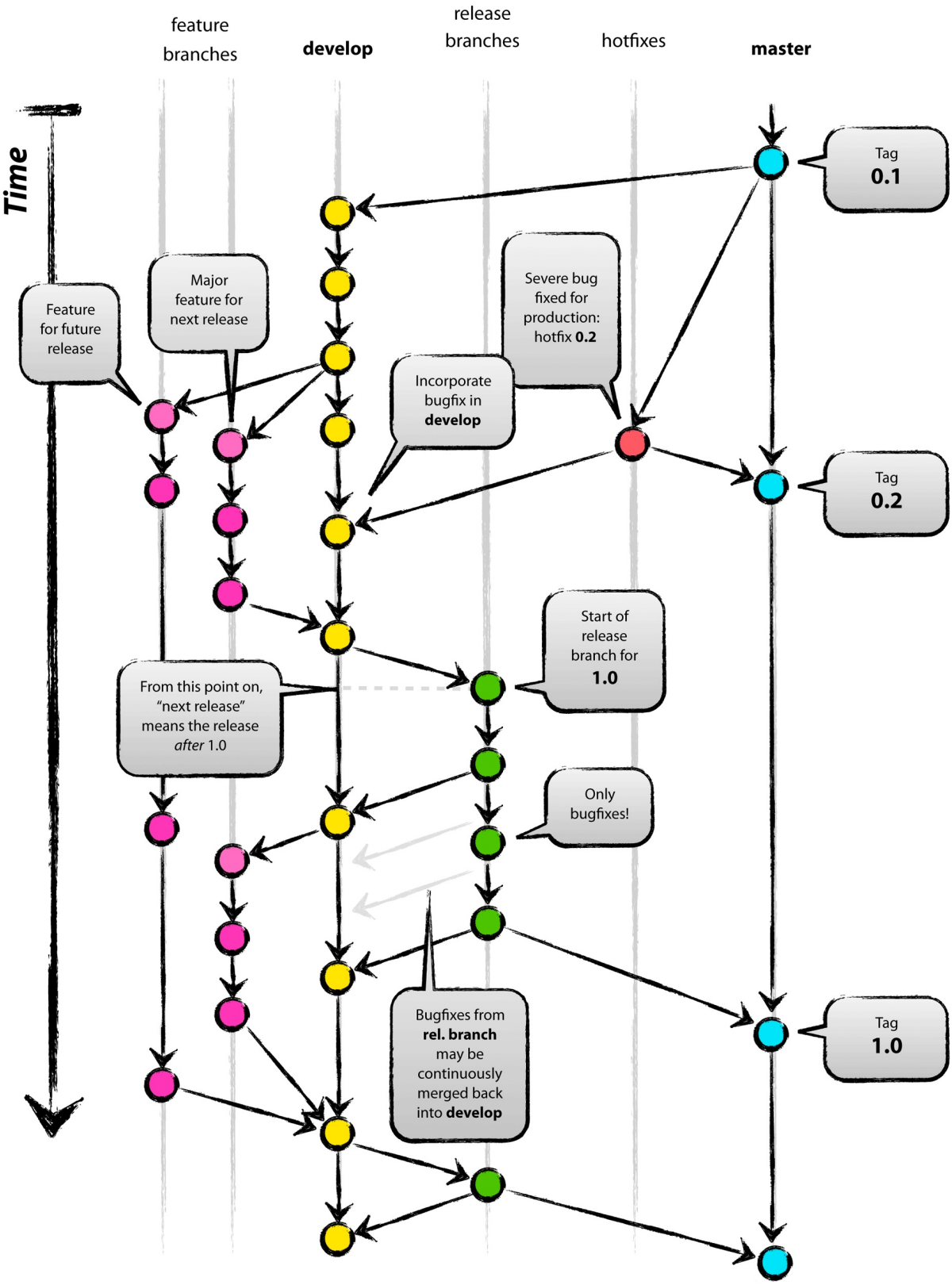
?animal_type_id=1：指定筛选条件

5. 【推荐】服务器应向用户返回的状态码和提示信息，2xx 代表成功； 3xx 重定向 ; 4xx 请求错误 ； 5xx 服务端错误

```
    200 OK - [GET]：服务器成功返回用户请求的数据，该操作是幂等的（Idempotent）。

    201 CREATED - [POST/PUT/PATCH]：用户新建或修改数据成功。

    202 Accepted - [*]：表示一个请求已经进入后台排队（异步任务）

    204 NO CONTENT - [DELETE]：用户删除数据成功。

    400 INVALID REQUEST - [POST/PUT/PATCH]：用户发出的请求有错误，服务器没有进行新建或修改数据的操作

    401 Unauthorized - [*]：表示用户没有权限（令牌、用户名、密码错误）。

    403 Forbidden - [*] 表示用户得到授权（与401错误相对），但是访问是被禁止的。

    404 NOT FOUND - [*]：用户发出的请求针对的是不存在的记录，服务器没有进行操作，该操作是幂等的。

    406 Not Acceptable - [GET]：用户请求的格式不可得（比如用户请求JSON格式，但是只有XML格式）。

    410 Gone -[GET]：用户请求的资源被永久删除，且不会再得到的。

    422 Unprocesable entity - [POST/PUT/PATCH] 当创建一个对象时，发生一个验证错误。

    500 INTERNAL SERVER ERROR - [*]：服务器发生错误，用户将无法判断发出的请求是否成功。
```

6. 【推荐】服务器应采用无状态机制，认证授权应考虑使用JWT、OAUTH2
7. 【推荐】服务器返回的数据格式，应该尽量使用JSON，避免使用XML。

1. 【参考】git-flow 是一个 git 扩展集，提供非常合理的分支管理模型，模型图如下：



2. 【推荐】严禁本地直接在master 、developer分支下直接进行开发，要建立特性分支；工作发布流程如下：

```
一、初始化 ： git flow init      你必须回答几个关于分支的命名约定的问题。建议使用默认值
```

二、增加新特性：git flow feature start MYFEATURE 这个操作创建了一个基于'develop'的特性分支，并切换到这个分支之下进行开发。

三、完成新特性：git flow feature finish MYFEATURE 这个动作执行下面的操作.

1）合并 MYFEATURE 分支到 'develop'

2）删除这个新特性分支

3）切换回 'develop' 分支

四、发布新特性：git flow feature publish MYFEATURE 。

你是否合作开发一项新特性？发布新特性分支到远程服务器，所以，其它用户也可以使用这分支。

五、取得发布的新特性：git flow feature pull origin MYFEATURE

六、准备release版本： git flow release start RELEASE \[BASE\] 它从 'develop' 分支开始创建一个 release 分支。

七、发布release版本： git flow release publish RELEASE

创建 release 分支之后立即发布允许其它用户向这个 release 分支提交内容是个明智的做法

八、完成 release 版本 ： git flow release finish RELEASE 完成 release 版本是一个大 git 分支操作

1、归并 release 分支到 'master' 分支

2、用 release 分支名打 Tag

3、归并 release 分支到 'develop'

4、移除 release 分支

3. 【推荐】紧急修复来自这样的需求：生产环境的版本处于一个不预期状态，需要立即修正；有可能是需要修正 master 分支上某个 TAG 标记的生产版本。流程如下：

一、开始 git flow 紧急修复 ： git flow hotfix start VERSION VERSION 参数标记着修正版本

二、完成紧急修复： git flow hotfix finish VERSION

当完成紧急修复分支，代码归并回 develop 和 master 分支。相应地，master 分支打上修正版本的 TAG 。