**Java 项目开发规范**

## 一、编程规范

#### (一) 命名风格

**1.** 【强制】代码中的命名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】代码中的命名中不允许出现中文。

**3.** 【强制】类名使用UpperCamelCase（大骆驼拼写法，又称帕斯卡拼写法，即每一个单词的首字母都大写），但以下情形例外：DAO / DO / BO / DTO / VO / AO / PO /等。

正例：AdminDAO / UserDO / XmlService / TcpUdpDeal / TaPromotion

反例：adminDAO / UserDo / XMLService / TCPUDPDeal / TAPromotion

说明：POJO 在本规范中指实体类，一般只有 setter /getter / toString 的简单类

**4.** 【强制】方法名、参数名、变量都统一使用 lowerCamelCase （小骆驼拼写法，第一个词的首字母小写，后面每个词的首字母大写）。

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，不必担心名字过长。

正例：MAX\_STOCK\_COUNT

反例：MAX\_COUNT

**6.** 【强制】包名统一使用小写形式，点分隔符之间有且仅有一个自然语义的单词。包名统一使用单数形式，类名可以使用复数形式。

正例：应用工具类包名为 com.yuyi.tool，类名为 MessageTools 。

**7.** 【强制】杜绝完全不规范的缩写，避免引起歧义和阅读困难。

反例：AbstractClass“缩写”命名成 AbsClass；condition“缩写”命名成 condi，随意缩写严重降低了代码的可阅读性。

**8**. 【强制】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，一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： void f();

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，应该是对所有实现类都有价值的默认实现。

**9.** 【强制】各层包名命名规范：

1） 一般包的命名格式为：com.yuyi.{开发人员名字首字母}.业务线

说明：添加{开发人员首字母}级是为了便于开发人员快速找到自己编写的代码

正例：张三在项目中的包：

com.yuyi.zs.controller

com.yuyi.zs.dao

2） POJO 所在的包命名为 com.yuyi.model

3） 工具类所在的包命名为 com.yuyi.tool

**10.** 【强制】领域模型层命名规范

POJO命名时类名、属性名严格按照 CamelCase 命名，尤其不能出现下划线，选择 DO、DTO、BO其中一个作为后缀。

DO：表对象，xxxDO，xxx 为数据库表名。（对象属性与表中字段一一对应）

DTO：数据传输对象，xxxDTO，xxx 数据表名（考虑信息的安全，一些敏感信息不能出现在数据传输对象（DTO）中，程序中一般的业务处理使用该类对象）。

BO：业务对象，xxxBO，xxx 一般为业务名称。（如自定义的分页对象，命名为PageBO）。

**11.** 【强制】DAO层类名以 POJO 的类名开始，以 DAO 为后缀，如果出现多人编辑同一个 POJO 的DAO，各自创建自己的DAO，命名方式为：{ POJO 类名}+{开发人员大写的名字拼音首字母}+DAO。

正例： ShipInfoDAO | ShipInfoZSDAO / ShipInfoLMDAO

说明：对于DAO，应提前将多个功能模块都会使用到的 DAO写好，方法尽量丰富化，常用的如 UserDAO、AdminDAO。

**12.** 【强制】 Controller层类名要能大概体现出处理的业务，并以Controller为后缀。

**13.** 【强制】 Service层的接口类名与对应的Controller保存一致，以Service为后缀，实现类类名与对应接口名保持一致，并以ServiceImpl为后缀。

正例：ShipInfoService / ShipInfoServiceImpl

**14.** 【强制】工具类包名为 com.yuyi.tool ,类名均以 Tool 或 Tools 为后缀；抽象类命名使用 Abstract开头；异常类命名使用 Exception 结尾；测试类命名以它要测试的类名或业务名开始，以 Test 结尾。

**15.** 【强制】DAO 层方法命名规范

1） 获取单个对象的方法用 get 作前缀。

2） 获取多个对象的方法用 list 作前缀。

3） 获取统计值的方法用 count 作前缀。

4） 插入的方法用 insert 作前缀。

5） 删除的方法用 delete 作前缀。

6） 修改的方法用 update 作前缀。

7） 方法参数一般是 POJO 的属性，参数名要与 POJO 的属性名一致。

**16.** 【强制】枚举类名带上 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：枚举名字为 ProcessStatusEnum 的成员名称：SUCCESS / UNKNOWN\_REASON。

**17.** 【强制】POJO 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。

说明：为保证不出现以 is 开头的属性，应在数据库设计字段时就避免以 is 开头。

#### (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：魔法数字的例子：

int priceTable[] = new int[16]; //这个16究竟有何含义呢?

正例：static final int PRICE\_TABLE\_MAX = 16; //带名字

int price Table[] = new int [PRICE\_TABLE\_MAX]; //名字的含义很清楚。

**2.** 【强制】long 或者 Long 初始赋值时，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.** 【参考】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); int seq;

SeasonEnum(int seq){

this.seq = seq;

}

}

#### (三) 代码格式

**1.** 【强制】大括号的使用。如果大括号内为空，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。

正例：

public class Test {

public static void main(String[] args) {

if (1 == 1) {

...

} else {

...

}

}

}

**3.** 【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） 方法调用时，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("sd").append("jn")...

.append("ha")...

.append("ha")...

.append("ha");

反例

StringBuffer sb = **new** StringBuffer();

//超过 120 个字符的情况下，不要再括号前换行

sb.append("sd").append("jn")...append

("ha");

// 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);

**5.** 【参考】不同逻辑、不同语义、不同业务的代码之间插入空行分隔开来以提升可读性。

#### (四) 编程规范

**1.** 【强制】严禁私自修改他人的代码，这也是一个程序员基本的修养。

**2.** 【强制】访问静态变量或静态方法时直接用类名来访问，否则会增加编译器解析成本。

**3.** 【强制】所有重写的方法，必须加 @Override 注解。

说明：编码过程中，字母的 O 与数字的0、字母的 l 与数字的1，一般难以区分，加 @Override 可以准确判断覆盖是否成功。另外，如果在抽象类中对方法签名进行修改，其实现类会马上报错提示。

**4.** 【强制】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】使用 java.util.Objects的equals方法代替java.lang.Object的equals方法。（JDK7）

说明：Object 的 equals 方法容易抛空指针异常，需要使用常量或确定有值的对象来调用equals。JDK8 中直接使用Objects的equals即可

**7.** 【强制】所有的相同类型的封装类对象之间值的比较，全部使用 equals 方法比较，禁止使用“==”进行比较。

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，所以使用 equals 方法进行判断。

**8.** 关于基本数据类型与封装数据类型的使用标准如下：

1） 【强制】所有的 POJO 属性必须使用封装数据类型。

2） 【强制】方法的返回值和参数必须使用封装数据类型。

3） 【强制】方法在调用时必须使用封装数据类型。

4） 【参考】局部变量可以使用基本数据类型。

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。（NPE（NullPointerException ）空指针异常，下同）

反例： 显示涨跌情况时，数据查询为 null 时，显示结果应为“-”，当用基本类型时，返回的是0，显示结果为 0% 。

**9.** 【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。

**10.** 【强制】POJO 类必须有 toString 方法（使用快捷键生成，不要手写）。如果继承了另一个 POJO 类，注意在前面加上 super.toString()。

**11.** 【强制】工具类中的方法全部声明为 static 的，并提供 private 的构造方法。

说明：工具类是作为一个全局共享的资源，并且调用者不涉及工具类中的任何属性和变量，所以没有必要进行实例化而增加编译器解析成本。

**12.** 【强制】时间、日期的处理：

使用 Instant 代替 Date，

使用 LocalDateTime 代替 Calendar，

使用 DateTimeFormatter 代替 SimpleDateFormat，

获取当前毫秒数 System.currentTimeMillis() 而不是 new Date().getTime()。

说明：官方给出的解释：simple beautiful strong immutable thread-safe。

**13.** 【强制】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。

**14.** 【参考】循环体内字符串的连接，使用 StringBuilder 的 append 方法进行扩展。

说明：反编译出的字节码文件显示使用String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";

}

#### (五) 集合处理

**1.** 【强制】不要在 foreach 循环里进行元素的 remove/add 操作。remove 元素请务必使用 Iterator方式，如果并发操作，需要对 Iterator 对象加锁。

正例：

Iterator<String> iterator = list.iterator();

while (iterator.hasNext()) {

String item = iterator.next();

if (删除元素的条件) {

iterator.remove();

}

}

反例：

List<String> list = new ArrayList<String>();

list.add("1");

list.add("2");

for (String item : list) {

if ("1".equals(item)) {

list.remove(item);

}

}

说明：以上代码，把“1”换成“2”，出现不同结果，一种正常运行，一种出现异常：java.util.ConcurrentModificationException 。

**2.** 【强制】Map 集合的遍历避免使用keySet 方式，选择以下方式：

1）使用 entrySet 方法。

2）使用 Map.forEach 方法。（JDK8）

正例：Map.foreach 方法示例

Map<String, Integer> map = new HashMap<String, Integer>(3);

map.put("a", 100);

map.put("b", 200);

map.put("c", 300);

map.forEach((k, v) -> {

System.*out*.println(k+" "+v);

if(Objects.*equals*("a", k)){

System.*out*.println("a的值："+v);

}

});

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。

#### (六) 控制语句

**1.** 【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;

**2.** 【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：判断奖品余量等于0时，活动停止，但因为并发处理错误导致奖品数量瞬间变为负数，导致活动无法终止。

**3.** 【参考】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");

return;

}

**4.** 【参考】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：

final boolean ex = (file.open(name, "w") != null) && (...) || (...);

if (ex) {

...

}

反例：

if (file.open(name, "w") != null) && (...) || (...)) {

...

}

#### (八) 注释规范

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。

**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、创建日期、创建时间。

**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（快捷键：Ctrl+/）。方法内部多行注释使用/\* \*/注释（添加注释：Ctrl+Shift+/ 除去注释：Ctrl+Shift+\）。

说明：不要使用“//”进行多行注释，避免多选或是少选导致的不必要的程序BUG

**5.** 【强制】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，使用三个斜杠(///)来说明注释掉代码的理由。

说明：及时清理不再使用的代码段以及配置信息。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。

**6.** 【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。

**7.** 【强制】代码修改的同时，注释也要进行相应的修改。

说明：代码与注释更新不是自动同步的，需要手动维护。

**8.** 【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；第三、力求精简准确、表达到位，避免冗杂。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。

## 二、异常

**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。

**5.** 【强制】向上抛出的异常必须是自己定义的异常，不可以将系统异常直接抛出。

**6.** 【强制】在事务中，异常需要抛出，否则事务不能回滚。

**7.** 【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。

1） try-catch 方式（手动关闭资源）：

BufferedReader br = new BufferedReader(new FileReader(path));

try {

return br.readLine();

} finally {

if (br != null){

br.close();

}

}

2） try-with-resources 方式（自动关闭资源）：

try (BufferedReader br = new BufferedReader(new FileReader(path))) {

return br.readLine();

}

**8.** 【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。

**9.** 【参考】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 封装数据类型的对象时，自动拆箱有可能产生 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。

## 三、MySQL 数据库

#### (一) 建表规范

**1.** 【强制】表的命名均使用小写字母，并以 \_table 为后缀。

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，我们都使用小写字母，避免节外生枝。

正例：admin\_table / e\_beans\_details\_table

**2.** 【强制】字段名均使用小写字母或数据，禁止出现数字开头，禁止两个下划线中间只出现数字，禁止以 is 开头。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： admin\_name，rdc\_config，level3\_name

反例：AdminName，rdcConfig，level\_3\_name，is\_delete

**3.** 【强制】禁用数据库关键字、保留字，如 desc、count、max、delayed 等。

说明：将名字复制到数据库管理工具中，关键字、保留字都会出现变色提示。

**4.** 【强制】尽量避免使用小数类型，通过改用小单位使用整数类型。如果必须使用，设置小数类型为 decimal 或numeric，禁止使用 float 和 double。

说明： float 和 double 在存储数据和传递数据的时候，存在精度损失的问题，在数值比较时，可能得到错误结果。

正例：存储金额时使用 BigInt 类型，以“分”为单位，java中以Long类型对应。

**5.**【强制】时间日期字段类型：

1）禁止使用 varchar、timestamp。

2）使用 datetime 类型，对应java 中的 java.sql.Timestamp（若使用date、time ，java中都使用 Timestamp 对应）。

3）使用 BigInt 类型，存储毫秒数，对应java中的Long类型。

说明：varchar 占用空间过大，timestamp 限制区间为1970-2037

正例：

// 根据指指定格式的字符串日期获取 java.sql.Timestamp

String str = "2017-12-01 00:00:00";

Timestamp timestamp = Timestamp.*valueOf*(str);

//根据毫秒数获取 java.sql.Timestamp

Long currentTime=System.*currentTimeMillis*();

Timestamp timestamp2 =new Timestamp(currentTime);

//处理 Timestamp 日期对象 （转为java.time.LocalDateTime）

LocalDateTime localDateTime = timestamp.toLocalDateTime();

Integer year = localDateTime.getYear(); // 年

Integer moth = localDateTime.getMonthValue(); // 月

Integer day = localDateTime.getDayOfMonth(); // 日

Integer hour = localDateTime.getHour(); // 小时

Integer minute = localDateTime.getMinute(); // 分钟

Integer second = localDateTime.getSecond(); // 秒

Long longTime = timestamp.getTime(); //毫秒数

**6.** 【参考】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。

#### (二) SQL 语句

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

**2.** 【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。

注：根据实际开发情况，考虑到开发效率，在开发过程中，可以使用 \* ，但项目开发完成，数据库结构稳定之后，必须将 \* 全部替换为确定的字段。

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

**4.** 【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。

**5.** 【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。

**6.** 【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;

**7.** 【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。

**8.** 【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。

**9.** 【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。

**10.** 【参考】in 操作尽量避免，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】<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。

## 四、工程结构

#### (一) 应用分层

**1.** 工程按照三层结构模式搭建：

1. 接口层（Controller层）：对外开放程序访问接口，负责接收请求、校验上传数据、响应请求结果，具体的业务处理交给Service层来处理。通常一个Controller单元（即一个Controller类）对应一个Service单元（一个Service接口和一个ServiceImpl实现类）；
2. 业务逻辑层（Service层）：进行具体的业务逻辑处理，一个Service单元通常对应一个或多个DAO单元。事务控制放在Service层。
3. 数据访问层（DAO层）：该层一般只声明接口，负责与数据库的交互。

#### (二) 接口规范

**1.** 【强制】接口请求中如果只作数据查询，并且请求参数不包含数据库表的信息时，使用 get 请求方式，其他一律使用 post 请求方式。

正例：增、删、改数据库信息时使用 post请求方式

查询商品列表（需要上传分页参数）时，使用 get 请求方式

反例：查询商品详情（需要上传商品id）时，使用 get 请求方式

**2.** 【强制】接口中用户请求传入的任何参数必须做有效性验证。如 ID 的正整数验证、字符串的长度验证、数字的范围验证、日期的格式验证等。

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（尽量避免使用正则表达式作校验）

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。

**3.** 【强制】接口响应的外层格式统一为如下格式，并且执行成功时“code”为 0 ，“msg”为“success”。

**{**

**code : 0**

**msg : success**

**data : null**

**}**

说明：“code”：状态码，由开发人员自己定义（为避免重复，提前进行范围划分）；

“msg”：执行信息，此次请求的状态描述；

“data”：请求的数据信息，可以为null。

后台程序无论执行结果如何，响应的结果必须保证该格式，这就需要在程序中进行统一异常处理及自定义异常。

## 四、Eclipse设置规范

1. 【强制】字符编码设置，编码方式均为“UTF-8”

注：每一步设置之后，记得点击右下方的“应用”按钮，否则设置不生效。

1）窗口 --》 首先项 --》 常规 --》 工作空间

右侧页面中“文本文件编码”选择“其他”，格式设置为“UTF-8”，“新的文本文件行定界符”选择“其他”，格式设置为“Unix”；

2）a.窗口 --》 首先项 --》 Web --》 CSS Files

右侧页面中 “Encodding”选择 “ISO 10646/Unicode(UTF-8)”，

b.窗口 --》 首先项 --》 Web --》 HTML Files

右侧页面中 “Encodding”选择 “ISO 10646/Unicode(UTF-8)”，

c.窗口 --》 首先项 --》 Web --》 JSP Files

右侧页面中 “Encodding”选择 “ISO 10646/Unicode(UTF-8)”；

3) 窗口 --》 首先项 --》 常规 --》 内容类型

右侧页面中，点击每一项，在下方“缺省编码”中写入“UTF-8”，然后点击“更新”。

**2.** 【强制】tab 键设置，设置 1 个 tab 为 4 个空格。

1） 窗口 --》 首选项 --》 常规 --》 编辑器 --》 文本编辑器

右侧页面中勾选“插入空格代替制表符”，“显示的 Tab 键宽度”设置为“4”。

2） 窗口 --》 首选项 --》 Java --》 代码样式 --》 格式化程序

右侧页面中点击“编辑”，在“缩进”选项中找到“常规设置”模块，设置“跳格策略”为“仅空格”，然后点击右下方的“应用”。

如果“应用”按钮为灰色，不能点击，就退出该页面，回到“格式化程序”页面，点击“新建”，自定义名字，之后进入编辑页面，“缩进”选项中“常规设置”模块，“跳格策略”选择“仅空格”，然后点击“应用”，再点击“OK”退出。

**3.** 【强制】注释模板设置，创建类时自动为类添加注释

窗口 --》 首选项 --》 Java --》 代码样式 --》 代码模板

在右侧页面中，点击“注释”展开，选择“类型”，点击右侧“编辑”按钮，在“模式”文本域中填入如下代码：

/\*\*

\*

\* **@author** lyb <br>

\* ${date} ${time}

\*/

点击“OK”退出编辑，在“代码模板”页面中下方勾选“自动为新方法和类型添加注释”，然后点击“应用”，再点击“OK”退出。

#### 附：Eclipse常用快捷键

|  |  |  |
| --- | --- | --- |
| **序号** | **快捷键** | **功能** |
| 1 | Ctrl+1（数字1） | 快速修复 |
| 2 | Ctrl+Shift+O | 整理当前类所引的包（多余的删去，缺少的引入） |
| 3 | Alt+Shift+Z | 为当前选中区域添加try、catch块 |
| 4 | Ctrl+D | 删除当前行 |
| 5 | Ctrl+Shift+/ | 为当前选中区域添加多行注释 |
| 6 | Ctrl+Shift+\ | 为当前选中区域取消多行注释 |
| 7 | Ctrl+Shift+X | 把当前选中的文本全部变为大写 |
| 8 | Ctrl+Shift+Y | 把当前选中的文本全部变为小写 |
| 9 | Alt+Shift+R | 重命名（所有引用该名字的地方，都会自动同步） |
| 10 | Alt+Shift+C | 修改方法结构（所有调用了该方法的地方，都会自动同步） |
| 11 | Ctrl+Alt+↓ | 复制当前行到下一行 |
| 12 | Ctrl+Alt+↑ | 复制当前行到上一行 |
| 13 | Alt+↓ | 当前行下移一行 |
| 14 | Alt+↑ | 当前行上移一行 |
| 15 | F2 | 显示类、方法、变量的详细信息 |
| 16 | F3 | 跳到类、方法、变量声明的地方 |
| 17 | Alt+Shift+S–>R | 为属性生成 setter / getter （Alt+A：选择全部属性） |
| 18 | Alt+Shift+S–>S | 生成 toString 方法 |