目录

[一、编程规约 2](#_Toc536172523)

[(一)命名风格 2](#_Toc536172524)

[(二)常量定义 4](#_Toc536172525)

[(三)代码格式 5](#_Toc536172526)

[(四)OOP规约 8](#_Toc536172527)

[(五)控制语句 10](#_Toc536172528)

[(六)注释规约 13](#_Toc536172529)

[(七)异常处理 15](#_Toc536172530)

[二**、**微服务与WEB后端API规范 17](#_Toc536172531)

[(一)命名规范 17](#_Toc536172532)

[(二)参数与返回值 18](#_Toc536172533)

[三**、**项目规范 20](#_Toc536172534)

[(一) 命名规则 20](#_Toc536172535)

[(二) 依赖管理 22](#_Toc536172536)

[附1：版本历史 23](#_Toc536172537)

[附2：阿里代码规范插件获取 23](#_Toc536172538)

[附3：代码注释模板(Ecplise) 23](#_Toc536172539)

[附4：专有名词解释 23](#_Toc536172540)

# 一、编程规约

## (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\_\_

1. 【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

1. 【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

1. 【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

1. 【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

1. 【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。
2. 【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。
3. 【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。
4. 【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）

1. 【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。

1. 【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。

1. 【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 rderFactory;

public class LoginProxy;

public class ResourceObserver;

1. 【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。

1. 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。

* 1. 【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。

1. 【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。

## (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);

1. 【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?

1. 【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。

1. 【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：放置在内部库中，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。

* 1. 子工程内部共享常量：即在当前子工程的constant目录下。
  2. 包内共享常量：即在当前包下单独的constant目录下。
  3. 类内共享常量：直接在类内部private static final定义。

## (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。

|  |  |  |
| --- | --- | --- |
| 空格 | a == b | 空格 |

反例：if ()

1. 【强制】if/for/while/switch/do等保留字与括号之间都必须加空格。
2. 【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。

1. 【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。

1. 运算符与下文一起换行。
2. 方法调用的点符号与下文一起换行。
3. 方法调用中的多个参数需要换行时，在逗号后进行。
4. 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);

1. 【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);

1. 【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。
2. 【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。

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。

1. 【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) {...}

1. 【强制】外部正在调用或者内部库依赖的接口，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。
2. 【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。

1. 【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）

1. 【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。

1. 【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。
2. 【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。

1. 【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)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;

}

1. 【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）。

1. 【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。

1. 【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。
2. 【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。
3. 【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。

## (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。

1. 【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。
2. 【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。

1. 【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。
2. 【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。

1. 【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。

1. 【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）。

1. 【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。
2. 【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）。

1. 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。

## (七)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) {…}

1. 【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。

1. 【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。

1. 【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。
2. 【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. 【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。

1. 【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。

1. 【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。

1. 【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。

* 1. 数据库的查询结果可能为null。
  2. 集合里的元素即使isNotEmpty，取出的数据元素也可能为null。
  3. 远程调用返回对象时，一律要求进行空指针判断，防止NPE。
  4. 对于Session中获取的数据，建议NPE检查，避免空指针。
  5. 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。

1. 【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。
2. 【参考】避免出现重复的代码（Don’t Repeat Yourself），即DRY原则。

说明：随意复制和粘贴代码，必然会导致代码的重复，在以后需要修改时，需要修改所有的副本，容易遗漏。必要时抽取共性方法，或者抽象公共类，甚至是组件化。

正例：一个类中有多个public方法，都需要进行数行相同的参数校验操作，这个时候请抽取：

private boolean checkParam(DTO dto) {...}

## 二**、**微服务与WEB后端API规范

## (一)命名规范

1. 【强制】接口命名采用restful风格，每一级path尽量简短，并统一采用小写。含有多个单词的path首先考虑是否可以进行拆分，如：userservice/msp /userprivilege/xxx 拆分为userservice/msp/user/privilege/xxx。path或参数需要多个单词命名，单词间用下划线分割，如:

unitservice/msp/unit\_convert？unit\_type=xxx。

1. 【强制】围绕对象操作为主的接口(涉及对象状态变更或持久化)，应通过request method对操作进行描述。

说明 ： {服务路径/对象名复数/id} 或 {服务路径/对象名复数?参数}

查询

userservice/msp/users/xxx GET

userservice/msp/users?name=xxx&age=xxx GET

新增

userservice/msp/users POST

更新

userservice/msp/users/xxx PUT

删除

userservice/msp/users/xxx DELETE

1. 【推荐】以独立操作为主的接口(无状态变更与持久化)，path可以采用动词或动词加名词进行命名，如：functionservice/msp/calculate，unitservice/msp/unit\_convert

## (二)参数与返回值

1. 【强制】接口接收的body数据与接口返回数据统一定义为JSON格式字符串,接口在request header与response header需对content-type加以限制。
2. 【强制】接口中定义的基本类型参数应通过url进行传递，避免将参数放入body中，对象、大数据量数据、敏感参数应放于body中。

正例：userservice/msp/users?name=xxx PUT

{

"userName":"xxx",

"age":"xxx",

"tel":"xxx"

}

反例：userservice/msp/users GET

{

"userName":"xxx",

"age":"xxx",

}

1. 【强制】接口返回的数据用统一结构进行封装，发现异常时应通过预定义的code来反应调用结果，并返回相应的数据体。

接口数据返回标准结构

HTTP/1.1 200

Content-Type: application/json

{

"code ": "",(请求结果码，请求正常返回0，异常情况以5位数字编码定义，首位数字为异常类别，后续会统一定义)

"data": {} (请求结果数据，统一为JSON对象，请求正常返回相应数据，程序内部异常情况返回统一JSON对象)

}

正常数据返回返回

HTTP/1.1 200

Content-Type: application/json

{

"code ": "0",

"data": {

"userName": "xxx",

"age": "xxx",

"tel": "xxx"

}

}

空指针导致的服务内部错误

HTTP/1.1 200

Content-Type: application/json

{

"code ": "10001",

"data": {

"message": "java.lang.NullPointException",(异常描述)

"detail": "xxx"(异常详细信息)

}

}

对象重复插入(义务类错误code可以表意，data无需做具体描述)

HTTP/1.1 200

Content-Type: application/json

{

"code ": "20001",

"data": {}

}

## 三**、**项目规范

## (一) 命名规则

1. 【强制】项目命名遵从以下规则：
   1. 项目名称统一小写，表意简介明了，可以适当使用简写，单词间有层级关系用短横线分隔，层级最多不超过3级。

说明： {模块-子模块1-子模块2} ，如：

monitorservice

workflow-templateservice

workflow-nodetaskservice

msfunction-dispatcherservice

msfunction-fieldassist-algorithm

msfunction-fieldassist-model

* 1. 服务类项目名称统一用service结尾。

1. 【强制】定义GAV遵从以下规则：
   1. GroupID格式：{com.公司.业务线 .子业务线}，最多4级。
   2. ArtifactID格式：原则上与项目名称相同，可根据特殊需求进行调整。
   3. Version：详细规定参考下方版本命名。

例：

<groupId>com.hal.dfs</groupId>

<artifactId>msfunction-dispatcherservice-client</artifactId>

<version>1.0.0-SNAPSHOT</version>

1. 【强制】内部库版本号命名方式：主版本号.次版本号.修订号
   1. 主版本号：产品方向改变，或者大规模API不兼容，或者架构不兼容升级。
   2. 次版本号：保持相对兼容性，增加主要功能特性，影响范围极小的API不兼容修改。
   3. 修订号：保持完全兼容性，修复BUG、新增次要功能特性等。

说明：注意起始版本号必须为：1.0.0，而不是0.0.1 正式发布的类库必须先去中央仓库进行查证，使版本号有延续性，正式版本号不允许覆盖升级。如当前版本：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

## (二) 依赖管理

1. 【强制】内部库不要自行配置编译JDK版本、上传库、下载库等所有内部库需要统一的配置。

说明：内部库所有项目编译配置原则上要保持的一致，统一从父pom读取，为防止出现编译JDK版本不统一,无法动态切换内部库地址带来的问题。

1. 【强制】正式发布项目不要依赖SNAPSHOT版本（安全包除外）。

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。

1. 【强制】依赖于一个内部库群时，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。

1. 【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。<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。

Java

开发手册

# 附1：版本历史

# 附2：阿里代码规范插件获取

阿里代码规范插件下载地址 <https://github.com/alibaba/p3c>

插件下载与安装方法https://www.cnblogs.com/aflyun/p/7668306.html

# 附3：代码注释模板(Ecplise)



注释模板导入方法https://blog.csdn.net/qq\_36186690/article/details/82505525

# 附4：专有名词解释

1. POJO（Plain Ordinary Java Object）: 在本手册中，POJO专指只有setter / getter / toString的简单类，包括DO/DTO/BO/VO等。
2. GAV（GroupId、ArtifactctId、Version）: Maven坐标，是用来唯一标识jar包。
3. OOP（Object Oriented Programming）: 本手册泛指类、对象的编程处理方式。