消费者BG JAVA代码规范与最佳实践

V1.0.0

版本历史

|  |  |  |  |
| --- | --- | --- | --- |
| 日期 | 版本 | 描述 | 作者 |
| 2017-08-04 | 0.9.0 | 初稿 | 张文凤 |
| 2017-08-28 | 0.9.1 | 根据建议对文档进行重新整理 | 张文凤 |
| 2017-09-21 | 1.0.0 | 根据其他专家建议进行修改 | 张文凤 |
|  |  |  |  |

目录

[一、 介绍 3](#_Toc493766633)

[1 目的 3](#_Toc493766634)

[2 适用范围 3](#_Toc493766635)

[3 术语 3](#_Toc493766636)

[4 总体原则 3](#_Toc493766637)

[二、 命名规范 3](#_Toc493766638)

[1 基本原则 3](#_Toc493766639)

[2 通用命名规范 4](#_Toc493766640)

[3 类命名 5](#_Toc493766641)

[4 方法命名 6](#_Toc493766642)

[三、 排版规范 6](#_Toc493766643)

[1 源文件结构 6](#_Toc493766644)

[2 代码格式 7](#_Toc493766645)

[四、 注释 9](#_Toc493766646)

[1 基本原则 9](#_Toc493766647)

[2 Javadoc说明 10](#_Toc493766648)

[3 Javadoc规范 12](#_Toc493766649)

[五、 编码规范 13](#_Toc493766650)

[1 声明、赋值等语句 13](#_Toc493766651)

[2 控制语句、程序结构 14](#_Toc493766652)

[3 常量 15](#_Toc493766653)

[4 并发、线程、性能 17](#_Toc493766654)

[5 日志 18](#_Toc493766655)

[6 异常 20](#_Toc493766656)

[7 其他 20](#_Toc493766657)

[六、 系统工程 23](#_Toc493766658)

[1 工程化 23](#_Toc493766659)

[2 包管理、版本管理 26](#_Toc493766660)

[3 安全性 27](#_Toc493766661)

[4 系统、服务器 28](#_Toc493766662)

[七、MySql数据库 28](#_Toc493766663)

[1 建表 28](#_Toc493766664)

[2 索引 30](#_Toc493766665)

[3 SQL语句 32](#_Toc493766666)

[4其他 32](#_Toc493766667)

# 介绍

## 1 目的

编码规范对于程序员而言尤为首要，有以下几个原因：

* 一个软件的生命周期中，80％的花费在于维护。
* 几乎没有任何一个软件，在其全部生命周期中，均由最初的开辟人员来维护。
* 编码规范可以改良软件的可读性，可以让程序员尽快而彻底地懂得新的代码。

最佳实践主要是包括编码习惯、禁忌等，能有效避免常见问题，提高代码质量。

## 2 适用范围

本规范适用于消费者BG所有使用Java语言编程的产品和项目。

## 3 术语

**强制**：编程时强制必须遵守的原则。

**推荐**：编程时必须加以考虑的习惯、写法等。

**参考：**编程时无法用代码量化的原则或描述。

**说明**：对此规范或建议进行必要的解释。

**正例、反例**：对此规范或建议从正、反两个方面给出例子。

## 4 总体原则

简单、易读、易实施执行

# 命名规范

## 1 基本原则

**【参考】**命名力求做到统一、达意、简洁。

统一即对于同一个概念，在程序中用同一种表示方法，比如对于供应商，既可以用supplier，也可以用provider，但是我们只能选定一个使用，至少在一个Java项目中保持统一。

说明:如果对同一概念有不同的表示方法，会使代码混乱难以理解。即使不能取得好的名称，但是只要统一，阅读起来也不会太困难，因为阅读者只要理解一次。

**达意**是指，标识符能准确的表达出它所代表的意义，比如：newSupplier,OrderPaymentGatewayService等；而supplier1,service2，idtts等则不是好的命名方式。

说明:准确有两成含义，一是正确，二是丰富。如果给一个代表供应商的变量起名是order，显然没有正确表达。同样的，supplier1远没有targetSupplier意义丰富。

**简洁**是指，在统一和达意的前提下，用尽量少的标识符。如果不能达意，宁愿不要简洁。比如：theOrderNameOfTheTargetSupplierWhichIsTransfered太长， transferedTargetSupplierOrderName则较好，但是transTgtSplOrdNm就不好了。注意：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。

## 2 通用命名规范

**【强制】**代码中所有命名必须以英文开头，名称只能由英文字母、数字、下划线、“-”符号组成。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。如下表：

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尽量避免采用。

|  |  |  |
| --- | --- | --- |
| **类型** | **规范** | **示例** |
| **项目名** | 工程名或模块名全部小写，多个英文单词以“-”连接。应用名与模块名，采用英文单词，不建议使用3个以上的单词，禁止使用特殊字符与数字 | lx-web-vaclient  lx-dubbo-user |
| **文件** | 跟类名保持一致 | PayController.java |
| **包** | 包名只允许使用小写字母和数字，并且单词之间点分隔符连接（不允许使用下划线） | com.iflytek.cbg.lx.  web.user.controller |
| **类**  **接口** | 类名、接口名使用UpperCamelCase风格,必须遵从驼峰形式，类名如果有复数含义,可以使用复数形式。另类名通常使用名词或名词短语 | PayController  UserService |
| **方法名**  **参数名**  **成员变量**  **局部变量** | 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。另方法名通常使用动词、动词短语、形容词  在单测方法中可以含有下划线，通常用来指定特定场景：test<测试方法名>\_<场景>，比如：testPop\_emptyStack。对于测试方法的命名没有强制性要求，也可以采用其他的命名方式 | getScore  userId |
| **常量** | 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 | MAX\_PAY\_TIMEOUT |

## 3 类命名

类名往往用不同的前缀或后缀表达额外的意思。如下表：

|  |  |  |
| --- | --- | --- |
| **前缀/后缀** | **意义** | **要求** |
| \*\*\*Controller | 直接处理页面请求、管理页面逻辑类、Http请求处理类 | 【强制】 |
| \*\*\*Service | 表明这个类是个服务类，里面包含了给其他类提同业务服务的方法 | 【强制】 |
| \*\*\*Dao | 这个类封装了数据访问方法 | 【强制】 |
| \*\*\*Entity  \*\*\*DTO  \*\*\*VO  \*\*\*Req  \*\*\*Resp | 不同用处的POJO类 | 【推荐】 |
| \*\*\*Enum | 枚举类 | 【强制】 |
| I\*\*\* | 这个类是一个接口 | 【强制】 |
| \*\*\*Impl | 这个类是一个实现类，而不是接口 | 【强制】 |
| Abstract\*\*\* | 抽象类 | 【强制】 |
| \*\*\*Base | 其他类的基类 | 【推荐】 |
| \*\*\*Factory  \*\*\*Proxy  \*\*\*Adapter  \*\*\*Observer | 如果使用到了设计模式,建议通过后缀在类名中体现出具体模式 | 【推荐】 |
| \*\*\*Wrapper | 这是一个包装类，为了给某个类提供没有的能力 | 【推荐】 |
| \*\*\*Listener  \*\*\*Event | 响应某种事件相关的类 | 【推荐】 |
| \*\*\*Job | 某种按时间运行的任务 | 【推荐】 |
| \*\*\*Utils | 工具类 | 【推荐】 |
| \*\*\*Exception | 异常类 | 【强制】 |
| \*\*\*Test | 测试类 | 【强制】 |

## 4 方法命名

方法名往往用不同的前缀表达特定的含义，如下表：

|  |  |  |
| --- | --- | --- |
| **前缀/后缀** | **意义** | **要求** |
| get**\*\*\*** | 获取单个对象 | 【推荐】 |
| list**\*\*\*** | 获取多个对象 | 【推荐】 |
| count**\*\*\*** | 计数，获取统计值 | 【推荐】 |
| save**\*\*\*** | 插入 | 【推荐】 |
| remove\*\*\* | 删除 | 【推荐】 |
| update\*\*\* | 更新 | 【推荐】 |
| find\*\*\*  search\*\*\* | 搜索、查询 | 【推荐】 |
| do\*\*\*  execute\*\*\* | 执行某个过程或者流程 | 【推荐】 |
| init\*\*\* | 初始化 | 【推荐】 |
| check\*\*\*  validate\*\*\* | 校验合法性 | 【推荐】 |

# 排版规范

## 1 源文件结构

1.**【强制】**所有源文件必须使用utf-8编码。

2.**【强制】**所有未使用的import语句应该被删除。

3.**【强制】**禁止使用通配符import。

说明：Eclipse中可以通过快捷键将不需要的引入包去掉。快捷键：Ctrl + Shift + O。

3.**【推荐】**同名的构造函数或方法之间禁止插入其他成员，例如重载的方法必须放在一起；不建议把新添加的方法一律放在最后，而是应该插入到合适的地方。

说明:当一个类有多个构造方法,或者多个同名方法，或者类似方法,这些方法应该按顺序放置在一起,便于阅读。

4.**【推荐】**类内定义顺序依次是:常量>公共变量或保护变量>私有变量>构造方法>静态公共方法>静态私有方法>公有方法或保护方法>getter/setter方法>main方法。

## 2 代码格式

**【强制】**代码格式规范主要是缩进、空格、换行、空行的运用。具体规范如下表：

说明:合理的运用缩进、空格、换行、空行，对代码的结构、可读性有非常大影响；另外，可以直接使用IDE自带的格式化快捷键进行全局或局部代码格式化。

|  |  |  |
| --- | --- | --- |
| **分类** | **规范解释** | **其他描述** |
| **行长度** | 单行字符数限制不超过120个,超出需要换行，换行时遵循换行原则； | 设置IDE单行字符数配置 |
| **缩进** | 1. 缩进采用4个空格,禁止使用tab字符； 2. 方法体的开始、类的定义、以及if、for、do、while、switch、case语句中的代码都要采用缩进方式； 3. 由于列宽限制需要缩进的情况遵循换行原则； | 设置IDE Tab键相关配置 |
| **空格** | 1. 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； 2. if/for/while/switch/do等保留字与括号之间都必须加空格； 3. 任何二目、三目运算符的左右两边都需要加一个空格；一元操作符如" !"、" ~"、" ++"、" --"等前后不加空格； 4. 方法参数在定义和传入时,多个参数逗号后边必须加空格； 5. 强制转型后必须跟一个空格; | 在if、else、for、do和while语句中，即使没有语句或者只有一行，也不得省略花括号。 |
| **换行** | 1、每行最多只能写一条语句，每条语句之后都要换行。  2、单行字符数超过120个限制需换行：   * 第二行相对第一行缩进,从第三行开始,不再继续缩进； * 运算符与下文一起换行； * 方法调用的点符号与下文一起换行； * 在多个参数超长,在逗号后换行； * 在括号前不要换行；   3、大括号：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:   * 左大括号前不换行； * 左大括号后换行； * 右大括号前换行； * 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； |  |
| **空行** | 空行可以表达代码在语义逻辑上的分割，注释的作用范围，等等。将类似操作，或一组操作放在一起不用空行隔开，而用空行隔开不同组的代码。例如：  1、在类的不同的成员间增加空行，包括：成员变量、构造函数、方法、内部类、静态初始化块、实例初始化块等；  2、方法体内，按需增加空行，以便从逻辑上对语句进行分组；  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；  3、注释与其上面的代码用空行隔开； | 说明:超过十行的代码如果还不用空行分割，就会增加阅读困难。另外，没有必要插入多个空行进行隔开。 |

正例:



# 注释

## 1 基本原则

* 注释力求精简准确、表达到位。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。好的命名、代码结构是自解释的,往往不需要或者只需要很少注释，就可以让人读懂；相反，不能正确表达代码意义的注释，只会损害代码的可读性。
* 注释宜少而精，不宜多而滥，更不能误导，过多的注释会成为后期维护的负担。防止过多过滥的注释，没必要的重复注释，或对显而易见的代码添加注释。
* 代码混乱，再多的注释都不能弥补，所以，应当先在代码本身下功夫。
* 注释要和代码同步。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。

* 注释不是用来管理代码版本的，如果有代码不要了，直接删除，svn会有记录的，不要注释掉，否则以后没人知道那段注释掉的代码该不该删除。

## 2 Javadoc说明

Java程序有两类注释：实现注释和文档注释。

实现注释是那些在C++中见过的，应用/\*...\*/和//界定的注释。文档注释是Java独有的（即Javadoc），由/\*\*...\*/界定,文档注释可以经由Javadoc工具转换成HTML文件。

Javadoc注释是给类的使用者来看的，主要介绍是什么功能，什么含义，怎么用等信息。凡是类的使用者需要知道，都要用Javadoc来写。Javadoc注释只负责描述类(class)、接口(interface)、方法(method)、构造器(constructor)、成员字段(field)。相应地，文档注释必须写在类、接口、方法、构造器、成员字段前面，而写在其他位置，比如函数内部，是无效的文档注释。

非Javadoc的注释往往是给代码的维护者看的，用以注释代码或者实现细节，着重告述读者功能以及逻辑，特殊或者复杂的逻辑可以注释说明为什么这样写，如何修改，注意什么问题等。

Javadoc说明可参考文档：<http://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html>

<http://blog.csdn.net/garfielder007/article/details/54959597>

正例:

## 3 Javadoc规范

下面规范要求不包括符合“注释-基本原则”中代码能自解释的例外情况。

1.**【强制】**所有的类都必须添加类功能描述、创建者和创建日期。

2.**【推荐】**类公共属性、类公共方法需要增加Javadoc注释,即使用/\*\*内容\*/格式,不得使用//xxx方式。

3.**【强制】** 所有对外暴露的服务(RPC..)中类、方法、输入参数、输出参数必须添加Javadoc注释。

4.**【强制】**项目组内通用类库或者开发的可复用的类库中对使用者暴露的类、方法、属性必须增加Javadoc注释。

5.**【强制】**所有抽象类和接口必须要用Javadoc注释；另外，子类中override的方法和接口的实现类对应的方法实现大部分情况下不需要添加注释。

6.**【强制】**对已经不推荐使用的类和方法需要注明@Deprecated，并说明替代的类或者方法。

7.**【推荐】**枚举类和常量类字段尽量要有注释, 说明每个字段的含义。

说明：字段命名能自解释的除外。

8.**【推荐】**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。

* + 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)。

* 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。

# 编码规范

## 1 声明、赋值等语句

1.**【强制】**一行只声明一个变量，不允许一行声明多个变量；在需要时声明变量，声明后尽快初始化，即初始化和被使用尽量放在一起。

正例:

int level = 0;

int size = 0;

反例:

int level, size;

2.**【强制】**不允许把多个短语句写在一行中，即一行只写一条语句。

3.**【强制】**避免在一个语句中执行多种操作，例如：给多个变量赋雷同的值，或赋值运算符与相等关系运算符混合使用，它很难读懂。

反例:

fooBar.fChar = barFoo.lchar = "c";

if （c++ = d++） {

//TODO

}

d = （a = b + c） + r;

4.**【强制】**必须用String[] args的方式来声明数组。

反例:使用String args[]的方式来定义。

5.**【强制】**长整型数字必须使用大写字母L结尾，不能使用小写字母l，以便和数字1进行区分。

6.**【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。

7.**【推荐】**含有集合意义的属性命名，尽量包含其复数的意义。

正例: List<Customer> customers = new ArrayList<Customer>();

8.**【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。另外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。

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。

正例: "test".equals(object);

反例: object.equals("test");

## 2 控制语句、程序结构

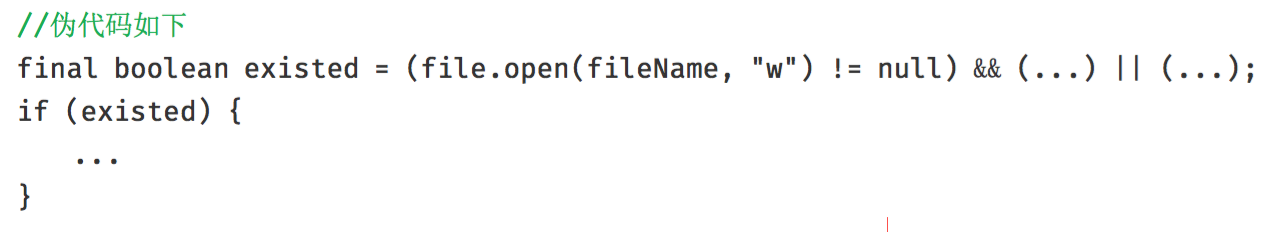
1. **【强制】**在一个switch块内,每个case要么通过break/return等来终止,要么注释说明程序将继续执行到哪一个case为止;在一个switch块内,都必须包含一个default语句并且放在最后,即使它什么代码也没有。
2. **【强制】**在if/else/for/while/do语句中必须使用大括号，即使只有一行代码,避免使用单行的形式:if (condition) statements。

3.**【强制】**不要在if语句中使用等号=进行赋值操作。

1. **【强制】**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,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:



反例:

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

...

}

5.**【推荐】**代码嵌套层次不要超过3层。

说明:代码嵌套层次达3层以上时，一般人理解起来都会困难。减少嵌套的方法有很多：合并条件、利用return以省略后面的else、利用子方法(将嵌套的程序提取出来放到另外的方法里)等。

6.**【推荐】**类超过2000行的程序难以阅读，应该尽量避免出现超过2000行的程序；

7.**【推荐】**注意运算符的优先级，避免使用默认优先级。一般而言，在含有多种运算符的表达式中应用圆括号来明确表达式的操作顺序，是个好办法。

说明：防止阅读程序时产生误解，防止因默认的优先级与设计思想不符而导致程序出错。

8.**【推荐】**可以通过return语句减少嵌套层次:

 if (condition) {

...  return obj;

}

// 接着写 else 的业务逻辑代码;

## 3 常量

1.**【推荐】**魔法值(未经定义的常量)尽量不要直接出现在代码中，代之以有名字的static final静态常量或者enum值。

说明：魔法值使代码的可读性大大下降，而且，如果同样的数值多次出现时，到底这些数值是不是带有同样的含义呢，谁也说不清楚。另一方面，如果本来应该使用相同数值的地方，一旦用错了，也很难发现，修改维护也不方便。因此，需要注意以下几点，极力避免使用魔法数值。

反例:

String key = "Id#taobao\_" + tradeId;

cache.put(key, value);

2.**【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。

3.**【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。

4.**【推荐】**如果变量值仅在一个范围内变化,且带有名称之外的延伸属性,定义为枚举类。下面正例中的数字就是延伸信息,表示星期几。

正例:

public enum WeekEnum{

MONDAY(1), TUESDAY(2), WEDNESDAY(3), THURSDAY(4), FRIDAY(5), SATURDAY(6), SUNDAY(7);

}

5.**【推荐】**避免使用不易理解的数字，用有意义的标识来替代。涉及物理状态或者含有物理意义的常量，不应直接使用数字，用有意义的静态变量来代替更好。

反例：如下的程序可读性差。

if (state == 0) {

state = 1;

// program code

}

正例:

private final static int TRUNK\_IDLE = 0;

private final static int TRUNK\_BUSY = 1;

private final static int TRUNK\_UNKNOWN = -1;

if (state == TRUNK\_IDLE) {

state = TRUNK\_BUSY;

// program code

}

## 4 并发、线程、性能

1.**【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JodaTime。

2.**【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);

3.**【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。

4.**【推荐】**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法。

5.**【推荐】**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");

...

}

}

6.**【推荐】**循环体中的语句要考量性能,循环体是最容易发送性能问题的地方。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.**【强制】**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: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";

}

8.**【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。

9.**【推荐】**[在明确的场景下,为集合指定初始容量](http://www.cnblogs.com/DreamDrive/p/5422175.html)。

说明:集合有默认大小，首先每次新增时都会判断是否达到临界点，且当长度不足时动态扩展严重影响性能。

10.**【强制】**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。

## 5 日志

1.**【强制】**不允许使用System.out.print，也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和更换日志系统。

说明：日志框架可以设定级别，可以控制输出到哪里，容易区分是在代码的什么地方打印的，而System.out.print则不行。而且System.out.print的速度很慢。所以，除非是有意的，否则都要用日志框架。至少在提交到版本库之前把System.out.print换成日志框架。

正例：

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class Demo {

private static final Logger logger = LoggerFactory.getLogger(Abc.class);

}

2.**【强制】**避免重复打印日志,浪费磁盘空间,务必在log4j.xml或logback.xml中设置 additivity=false。

正例:

<logger name="com.iflytek.lx.config" additivity="false">

3.**【强制】**生产环境禁止输出debug、trace和其他调试日志；相应的，生产环境一定要记录下错误日志和异常异常，利于现网故障或用户投诉时排查问题。

说明：谨慎地根据需要配置日志级别。例如测试环境可以打开info日志，性能测试需要关闭debug和其他调试日志，保持与生存环境一致。

日志的级别由高到低分别为：ERROR >> WARN >> INFO >> DEBUG >> TRACE。

* Error：用于记录影响系统正常运行的一切信息，包括：Throwable、Error、Exception、与其他系统通讯错误，例如：SQL执行异常、其他系统返回的错误信息；
* Warn：用于记录不影响系统正常运行，但需要注意需要及时解决，例如：某个信息；
* Info：通常生产环境的最低级别，用于记录这种重要的提示信息，例如：各种通讯信息，关键的提示信息，以及这种需要在生产环境下提示的信息；
* Debug： 记录一些详细的数据，比如我们想知道一个方法的多个入参值，那么在生产环境中，一般不会打印出来，在开发阶段和调试阶段，建议使用debug进行记录查看。在生产环境中将不会出现此类信息（当然可能在某些特定情况下，通过特殊手段某一时间段打开用于生产环境调试）。
* Trace：用于代替System.out.println()和System.err.println()，供开发阶段调试用。

4.**【推荐】**在生产环境中配置日志框架自动加载刷新，即不重启应用就可以调整日志的配置。例如在logback.xml的配置中开源加上：

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<configuration scan=*"true"* scanPeriod=*"60 seconds"*>

</configuration>

## 6 异常

1.**【强制】**不允许使用printStackTrace打印异常日志。

2.**【强制】**记录异常日志时不要只保存exception.getMessage()，记录整个异常堆栈信息。

3．**【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。另外自己抛出的异常必须要填写详细的描述信息，且必须保留原来的异常，便于问题定位，例如：throw new IOException("Writing data error! Data: " + data.toString(), e)。

4.**【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。

5.**【强制】**finally块必须对资源对象、流对象进行关闭，例如数据库操作、IO操作,且有异常也要进行try-catch，因为finally块中如果发生异常,不会再执行try块中的return语句。

6.**【推荐】**防止NullPointerException,IndexOutOfBoundsException是程序员的基本修养,代码中可以通过预先检查进行规避,而不应该通过catch来处理。

正例:

if (obj != null) {

...

}

反例:

try {

obj.method();

} catch (NullPointerException e) {

...

}

## 7 其他

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

* + 只要重写equals,就必须重写 hashCode。
  + 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。
  + 如果自定义对象做为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】**所有的覆写方法,必须加@Override注解。

3.**【强制】**不能使用过时的类或方法。

4.**【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。

5.**【推荐】**POJO类重写toString方法。

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。

6.**【强制】**不要在foreach循环里进行集合元素的remove/add操作。

7.**【推荐】**使用索引访问用String的split方法得到的数组时,做边界检查,否则会有抛IndexOutOfBoundsException的风险。

8.**【强制】**禁止在代码中显示调用System.gc()。

9.**【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。

正例：

Foo.aStaticMethod(); // good

反例：

Foo aFoo = ...;

aFoo.aStaticMethod();

somethingThatYieldsAFoo().aStaticMethod();

10.**【推荐】**类成员与方法访问控制从严，不是必须使用public属性的，请使用protected，不是必须使用protected,请使用private。

* + 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。
  + 工具类不允许有public或default构造方法。
  + 类非static成员变量并且与子类共享,必须是protected。
  + 类非static成员变量并且仅在本类使用,必须是private。
  + 类static成员变量如果仅在本类使用,必须是private。
  + 若是 static 成员变量,必须考虑是否为final。
  + 类成员方法只供类内部调用,必须是private。
  + 类成员方法只对继承类公开,那么限制为protected。

说明:良好的程序设计应该尽可能减小类与类之间耦合，所遵循的经验法则是：尽量限制成员函数的可见性。

11**.【强制】**在一个项目组内开发环境、测试环境、生产环境使用统一版本的JDK。

说明:不同版本的JDK编译出来的class不一样，可能存在冲突，不同版本的JDK也会带来莫名奇妙的问题且很难定位，所以一开始就要求必须统一。

12.**【推荐】**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。

13.**【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()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:

* 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。
* 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。

14.**【推荐】**logback比log4j性能要好很多，但是logback和log4j互斥，只能使用其中之一。因此，使用logback需要排除掉log4j依赖所有相关jar包依赖，且为了兼容一些二方库、三方库（可能使用了log4j），需要引入引入log4j-over-slf4j.jar;

15.**【推荐】**对于“明确停止使用的代码和配置”,如方法、变量、类、配置文件、动态配置属性等要坚决从程序中清理出去,避免造成过多垃圾。

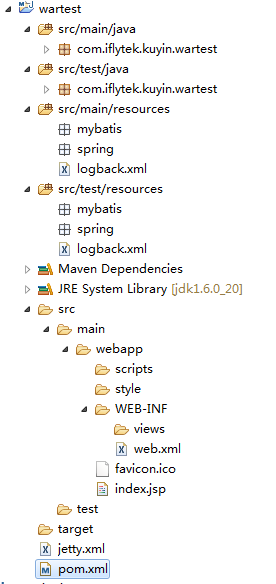
# 系统工程

## 1 工程化

1.**【推荐】**针对业务形态，工程分为以下几种情况：

* Web工程
* web-应用名，以“web-”前缀开头。
* Dubbo工程
* service-应用名，以“service-”前缀开头。
* Interface工程
* interface-应用名，以“interface-”前缀开头。
* 工具或者任务
* tool-应用名，以“tool-”前缀开头，或以“job-”前缀开头。
* Java公共Jar包
* common-模块名，以“common-”前缀开头。

2. **【推荐】**J2EE项目工程结构



说明：J2EE项目源码结构如上图。所有代码均在src文件夹下，target存放的是编译后的中间class文件。

要注意src/main下面是正式代码和配置存放路径，编译输出到target/classes下面。src/test下面是单元测试代码存放路径，同时src/test下面编译后输出路径为target/test-classes下面。

src/main/resources文件夹下面存放应用的配置文件包括日志配置文件和spring配置XML文件，采用平铺的方式。Mybatis相关的配置放到src/main/java源文件夹下，并分为读写分离的2个包。

src/main/java下面存放正式的Java源码。

src/test/resources文件夹与src/main/resources功能一样，但这里是针对单元测试使用到的配置，需要针对单元测试进行配置项的值进行修改。

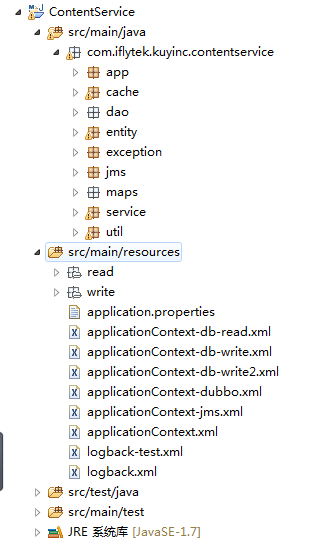
src/test/java下面是单元测试类，原则上单元测试的类的包路径要与原来的类包路径一致。

所有CSS样式及样式图片均存放于src/main/webapp/style下面，可以建文件夹以区分肤色或不同类型的文件。

所有Javascript脚本文件存放于src/main/webapp/scripts下面，可以根据不同的脚本建文件夹分开存放。

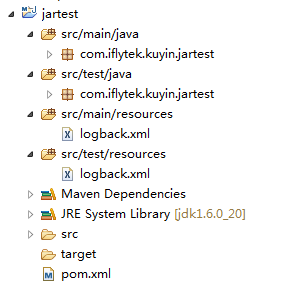
所有Spring MVC中使用到的JSP页面存放于src/main/webapp/WEB-INF/views下面。

3.**【推荐】**Dubbo项目工程结构



说明：除了没有webapp目录，其它的main/test，Resources、Spring、Mybatis配置与J2EE项目一样。

4.**【推荐】**J2SE项目工程结构



说明：J2SE项目有两种情况：

一个是小工具，可以独立运行，处理一些简单的业务，打包为zip包。

另一个是公共开发包，打包为jar包。以API方式提供给其它应用程序，以专业专注的精神进行分包，例如：通信协议包，常用工具包，加密解密包等等。公共开发包可能无法独立运行，所有的服务API都需要进行单元测试，并要保证线程安全。

## 2 包管理、版本管理

1.**【推荐】**Maven定义GAV遵从以下规则:

* GroupID格式:com.{公司}.{部门}.业务线.[子业务线],最多4级。
* 正例:com.iflytek.cbg.lx 或 com.iflytek.cgb.kuyin
* ArtifactID格式:产品线名-模块名。语义不重复不遗漏,先到中央仓库去查证一下。
* 正例:lx-web-user/lx-dubbo-user/lx-common
* Version:详细规定参考下方。

2.**【推荐】**二方库版本号命名方式: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: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

3.**【推荐】**线上应用不要依赖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。

4.**【推荐】**上传代码至svn时，项目相关的配置文件（.settings、.classpath、.project）不要上传本地调试文件，仅上传src文件夹、pom.xml或bin文件夹。

5.**【推荐】**Maven工程需要在POM配置仓库地址，避免在本地maven setting中配置，其他组员check out后无法下载或者需要从远程仓库下载；

## 3 安全性

**【备注】公司已经有大量项目被恶意攻击历史，所以研发必须提高安全意识和相关技术、方法、手段。**

1.**【强制】**对外提供的Http Api，必须进行签名、加密。

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。接口对敏感数据也必须加密，例如：用户密码。

3.**【参考】**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

4.**【参考】**表单、AJAX请求必须执行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。

5.**【参考】**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。

6.**【参考】**活动类需求，尤其是中奖类活动，必须实现防刷策略。

## 4 系统、服务器

1.**【推荐】**高并发服务器建议调小TCP协议的time\_wait超时时间。

说明:操作系统默认240秒后,才会关闭处于time\_wait状态的连接,在高并发访问下,服务器端会因为处于time\_wait的连接数太多,可能无法建立新的连接,所以需要在服务器上调小此等待值。

正例:在 linux服务器上请通过变更/etc/sysctl.conf文件去修改该缺省值(秒):

net.ipv4.tcp\_fin\_timeout = 30

2.**【推荐】**调大服务器所支持的最大文件句柄数(File Descriptor,简写为fd)。

说明:主流操作系统的设计是将TCP/UDP连接采用与文件一样的方式去管理,即一个连接对应于一个fd。主流的linux服务器默认所支持最大fd数量为1024,当并发连接数很大时很容易因为fd不足而出现“open too many files”错误,导致新的连接无法建立。建议将linux服务器所支持的最大句柄数调高数倍(与服务器的内存数量相关)。

3.**【推荐】**给JVM设置-XX:+HeapDumpOnOutOfMemoryError 参数,让JVM碰到OOM场景时输出dump信息。

说明:OOM的发生是有概率的,甚至有规律地相隔数月才出现一例,出现时的现场信息对查错 非常有价值。

4.**【推荐】**当生产环境组件异常、无响应、假死时，务必使用jstack dump线程堆栈，使用jstat –gcutil查看下gc信息，方便后期定位、分析问题。

# 七、MySql数据库

## 1 建表

1.**【推荐】**使用InnoDB存储引擎、UTF-8(utf8mb4)字符集。

2.**【强制】**表要有主键。

3.**【推荐】**若业务上非实时一致性要求，禁止使用外键，由程序保证最终一致性。

说明:外键会导致表与表之间耦合，影响sql性能，高并发下有可能会导致死锁。

4.**【推荐】**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。

正例:getter\_admin,task\_config,level3\_name

反例:GetterAdmin,taskConfig,level\_3\_name

5.**【推荐】**表名最好不使用复数名词。

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。

6.**【推荐】**表的命名最好是加上“模块名称\_表名”。

正例:score\_rule / score\_user /score\_log

7.**【强制】**千万不要使用mysql中关键字命名，例如：desc, order等；

8.**【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。

9.**【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。

10.**【强制】**所有字段定义为Not Null。

说明：

* null列影响索引。
* MySql内部需要对null进行特殊处理,会影响性能
* null需要更多的存储空间
* null查询必须使用is或is not,不能使用=、in等

11.**【强制】**禁止使用ENUM类型。

说明：增加新的枚举值时，需要DDL操作。

12.**【推荐】**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。

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

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。

14.**【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:

* 不是频繁修改的字段。
* 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.**【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。

## 2 索引

说明：务必用explain确认SQL语句是否走了索引。

1.**【参考】**创建索引时避免有如下极端误解:

* 宁滥勿缺。误认为一个查询就需要建一个索引。
* 宁缺勿滥。误认为索引会消耗空间、严重拖慢更新和新增速度。
* 抵制惟一索引。误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.**【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。

3.**【参考】**建立组合索引，必须根据业务上的查询需求使用合理顺序。

说明:联合索引左前缀原则：(a,b,c)只有查询条件为a|(a,b)|(a,b,c)才会走该组合索引，但跟条件顺序无关，例如：ba、bac、bca、cab都会走索引，而bc、ac不会走索引。

4.**【参考】**负向查询(!=、not in、not exists)、前导模糊查询(like ‘%xxx’)、在查询条件属性上进行函数计算、字段类型需强制转换都不会走索引。

反例：

select \* from score where status!=2

select \* from score where name like ‘%XX’

select \* from score where YEAR(create\_time) <= ‘2017’

select \* from user where phone=13800001234

6.**【推荐】**数据区分度不大的字段不宜使用索引，效果不明显。

说明："状态"、"性别"这种区分度不大的属性，建立索引是没有什么意义的，不能有效过滤数据，性能与全表扫描类似。

反例：select \* from user whre sex=1

7.**【推荐】**OR查询最好改为IN查询。

说明：InnoDB引擎中OR查询不能命中索引，MyISAM能命中索引，但会耗费更多的CPU。

8.**【推荐】**超过三个表不要使用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。

9.**【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。

10.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。

## 3 SQL语句

1.**【推荐】**尽量不要使用select \*。

2.**【推荐】**禁止大表使用join，禁止大表使用子查询。

3.**【推荐】**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。

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:

* NULL<>NULL的返回结果是NULL,而不是false。
* NULL=NULL的返回结果是NULL,而不是true。
* NULL<>1的返回结果是NULL,而不是true。

6.**【推荐】**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。

7.**【推荐】**不要使用存储过程,存储过程难以调试和扩展,更没有移植性。

## 4其他

1.**【强制】**不要使用BLOB相关类型存储大文件或者大照片。

说明：文件应使用文件系统存储，数据库存文件的URI

2.**【推荐】**把计算尽量放到业务层(代码层)而不是数据库层，可以节省数据库服务的CPU资源，还可以根据场景在业务层进行缓存等优化，减少数据库压力。