**SIS基线编程规范 V1.0版**

**设计篇：**

1、 一体化运作，Story体现开发、测试的设计

2、 接口分析具体到每个字段，充分考虑接口的兼容性、扩展性

**编码篇：**

1、 避免JAVA十大低级错误

2、配置静态检查工具checkStyle和PMD，并设置为自动检查

3、 缩短事务处理时间，事务里尽量只包含增删改操作

4、 公用方法须做参数校验，确保正确性（比如判断空等）

5、 公有功能方法要有正确的javadoc，修改代码要同步修改

6、 打印日志要详细，日志要覆盖关键流程、关键信息

7、 返回集合时不允许返回null，应该返回空集合

8、 静态方法（除工具类）和静态变量（除常量）要慎用

9、 注释描述必须清楚、准确，代码变更时必须对其同步修改

10、在实现Runnable接口的类中的run方法内，必须捕获Throwable异常；为了避免异常抛出，业务代码必须都在try块中

11、避免重复代码，包括文件内和文件之间

12、避免依恋情结（Feature Envy），将属性和行为尽量放在一起

13、注意圈复杂度，新增函数/方法圈复杂度不超过10

14、函数/方法的参数不能超过5个

15、避免循环依赖

# Java十大低级错误

1. 类、方法、变量、常量等命名表达的含义和实际用途不一致。
2. 缺少类、公有方法注释，代码修改后没有同步修改注释。
3. 日志和实际情况不一致；捕获异常后没有在日志中记录异常栈。
4. 魔鬼数字。
5. 空指针异常。
6. 数组下标越界。
7. 将字符串转换为数字时没有捕获NumberFormatException异常。
8. 对文件、IO、数据库等资源进行操作后没有及时、正确进行释放。
9. 循环体编码时不考虑性能，循环体中包含不需要的重复逻辑。
10. 数据类没有重载toString()方法。

# 解读&案例

## 命名规范性

### 解读

在代码中，类、方法、变量、常量等的名称是帮助理解代码最直观的要素。命名良好、逻辑简洁的代码可以达到自注释的目的，通过阅读代码就可以基本理解业务逻辑，减少编写和维护注释的工作量。在编码时必须重视命名的规范性，名称应该简洁、容易理解，能够准确表达所命名对象的含义。临时变量的名称应该符合常用的习惯（如for循环的整形循环子一般使用i、j）。

### 案例

下面就是一个命名规范性不好的例子：

**public** **void** notifyProduct(Integer productId)

{

**if** (productId == **null**)

{

**throw** **new** PMSException(PMSErrorCode.CommonError.*PARAM\_NULL*, "product ID is null");

}

ProductService prodSrv = (ProductService) ServiceLocator.*findService*(ProductService.**class**);

ProductServiceStruct product = prodSrv.queryProduct(productId);

**if** (product == **null**)

{

**throw** **new** PMSException(PMSErrorCode.CommonError.*PARAM\_ERROR*, "product not exist");

}

**if** (PMSConfig.*getInstance*().getConfig1()) // getConfig1没有具体含义

{

NotifyService sss1 = (NotifyService) ServiceLocator.*findService*(NotifyService.**class**);

sss1.notifyProduct(product);

**if** (PMSConfig.*getInstance*().getNotifyWisg()) // getNotifyWisg名称和实际含义不一致

{

MdspProductExtService sss2 = (MdspProductExtService) ServiceLocator.*findService*(MdspProductExtService.**class**);

List<MdspProductExt> eee1 = sss2.queryProductExtInfo(productId);

sss1.notifyProductExtInfo(eee1);

}

// sss1、sss2、eee1没有任何含义，理解困难

}

}

如果采用规范、能表达含义的名称，代码就变得非常容易理解：

**public** **void** notifyProduct(Integer productId)

{

**if** (productId == **null**)

{

**throw** **new** PMSException(PMSErrorCode.CommonError.*PARAM\_NULL*, "product ID is null");

}

ProductService prodSrv = (ProductService) ServiceLocator.*findService*(ProductService.**class**);

ProductServiceStruct product = prodSrv.queryProduct(productId);

**if** (product == **null**)

{

**throw** **new** PMSException(PMSErrorCode.CommonError.*PARAM\_ERROR*, "product not exist");

}

**if** (PMSConfig.*getInstance*().isNeedNotifyProduct())

{

NotifyService notifySrv = (NotifyService) ServiceLocator.*findService*(NotifyService.**class**);

notifySrv.notifyProduct(product);

**if** (PMSConfig.*getInstance*().isNeedNotifyProductExtInfo())

{

MdspProductExtService prodExtSrv = (MdspProductExtService) ServiceLocator.*findService*(MdspProductExtService.**class**);

List<MdspProductExt> prodExtInfos = prodExtSrv.queryProductExtInfo(productId);

notifySrv.notifyProductExtInfo(prodExtInfos);

}

}

}

## 注释规范性

### 解读

注释是帮助理解代码的重要资料，缺少必要的注释，可能需要浪费大量的时间阅读、调试代码，才能真正理解代码的逻辑，造成巨大的浪费，也会给维护代码带来很多困难。但注释也不是越多约好，在没有必要写注释的地方写冗余注释，或者不规范、甚至和代码逻辑不一致的注释，会给理解代码造成严重的干扰，不如不写。

类注释和方法注释应该按照编程规范的规定编写。方法注释需要说明方法入参含义、取值范围、可否为空、是否会被方法修改，以及方法返回值的含义、能否为空等情况。

对于方法中的注释，建议通过良好的命名规范、代码结构和统一的缩进风格，尽量使代码自注释，只在代码无法完全表达业务逻辑的地方添加必要的注释。以代码为主，注释为辅。修改代码时应同步修改注释，杜绝注释和代码不一致的情况。

### 案例

/\*\*

\* <一句话功能简述>。

\* <功能详细描述>

\* // 没有方法注释，应该将自动生成的注释模板删除，添加有意义的类注释

\*/

**public** **void** sendMsg()

{

...

}

/\*\*

\* 根据内容ID和产品类型查询产品列表。

\*

\* **@param** contentId 内容ID // 能否为null？如果为null如何处理？

\* **@param** productType 产品类型 // 都有哪些类型？能否为null？为null如何处理？

\* **@return** 产品列表 // 如果没有查到，是返回空List还是返回null？

\*/

**public** List<ProductServiceStruct> queryProduct(String contentId, String productType)

{

...

**if** (*PRODUCT\_TYPE\_SUB*.equals(productType))

{

// 查询点播类产品

// 注释和代码不一致

**return** querySubProduct(contentId);

}

...

**return** **null**;

}

/\*\*

\* 根据内容ID查询订购类产品列表。

\*

\* **@param** contentId 内容ID，不能为null

\* **@return** 产品列表，如果没有返回空List

\*/

**public** List<ProductServiceStruct> querySubProduct(String contentId)

{

**return** ...;

}

## 日志规范性

### 解读

日志是定位问题时最重要的依据，业务流程中缺少必要的日志会给定位问题带来很多麻烦，甚至可能造成问题完全无法定位。

异常产生后，必须在日志中以ERROR或以上级别记录异常栈，否则会导致异常栈丢失，无法确认异常产生的位置。并不需要在每次捕获异常时都记录异常日志，这样可能导致异常被多次重复记录，影响问题的定位。但异常发生后其异常栈必须至少被记录一次。

和注释一样，日志也不是越多越好。无用的冗余日志不但不能帮助定位问题，还会干扰问题的定位。而错误的日志更是会误导问题，必须杜绝。

### 案例

下面的例子虽然打印了很多日志，但基本上都是无用的日志，难以帮助定位问题。甚至还有错误的日志会干扰问题的定位：

**public** **void** saveProduct1(ProductServiceStruct product)

{

log.debug("enter method: addProduct()");

log.debug("check product status");

**if** (product.getProduct().getProductStatus() != ProductFieldEnum.ProductStatus.*RELEASE*)

{

**throw** **new** PMSException(PMSErrorCode.Product.*ADD\_ERROR*);

}

log.debug("check tariff");

BooleanResult result = checkTariff(product.getTariffs());

**if** (!result.getResult())

{

**throw** **new** PMSException(PMSErrorCode.Product.*ADD\_ERROR*);

}

log.debug("before add product");

ProductService prodSrv = (ProductService) ServiceLocator.*findService*(ProductService.**class**);

**try**

{

prodSrv.addProduct(product);

}

**catch** (BMEException e)

{

// 未记录异常栈，无法定位问题根源

}

log.debug("after add product");

log.debug("exit method: updateProduct()"); // 错误的日志

}

而下面的例子日志打印的不多，但都是关键信息，可以很好的帮助定位问题：

**public** **void** saveProduct2(ProductServiceStruct product)

{

**if** (product.getProduct().getProductStatus() != ProductFieldEnum.ProductStatus.*RELEASE*)

{

log.error(

"product status "

+ product.getProduct().getProductStatus()

+ " error, expect " + ProductFieldEnum.ProductStatus.*RELEASE*);

**throw** **new** PMSException(PMSErrorCode.Product.*ADD\_ERROR*);

}

BooleanResult result = checkTariff(product.getTariffs());

**if** (!result.getResult())

{

log.error(

"check product tariff error "

+ result.getResultCode()

+ ": "

+ result.getResultDesc());

**throw** **new** PMSException(PMSErrorCode.Product.*ADD\_ERROR*);

}

ProductService prodSrv = (ProductService) ServiceLocator.*findService*(ProductService.**class**);

**try**

{

prodSrv.addProduct(product);

}

**catch** (BMEException e)

{

log.error("add product error", e);

**throw** **new** PMSException(PMSErrorCode.Product.*ADD\_ERROR*, e);

}

}

## 魔鬼数字

### 解读

在代码中使用魔鬼数字（没有具体含义的数字、字符串等）将会导致代码难以理解，应该将数字定义为名称有意义的常量。

将数字定义为常量的最终目的是为了使代码更容易理解，所以并不是只要将数字定义为常量就不是魔鬼数字了。如果常量的名称没有意义，无法帮助理解代码，同样是一种魔鬼数字。

在个别情况下，将数字定义为常量反而会导致代码更难以理解，此时就不应该强求将数字定义为常量。

### 案例

**public** **void** addProduct(ProductServiceStruct product)

{

// 魔鬼数字，无法理解3具体代表产品的什么状态

**if** (product.getProduct().getProductStatus() != 3)

{

**throw** **new** PMSException(PMSErrorCode.Product.*ADD\_ERROR*);

}

BooleanResult result = checkTariff(product.getTariffs());

**if** (!result.getResult())

{

**throw** **new** PMSException(PMSErrorCode.Product.*ADD\_ERROR*);

}

}

**public** **void** addProduct2(ProductServiceStruct product)

{

// 仍然是魔鬼数字，无法理解NUM\_THREE具体代表产品的什么状态

**if** (product.getProduct().getProductStatus() != *NUM\_THREE*)

{

**throw** **new** PMSException(PMSErrorCode.Product.*ADD\_ERROR*);

}

BooleanResult result = checkTariff(product.getTariffs());

**if** (!result.getResult())

{

**throw** **new** PMSException(PMSErrorCode.Product.*ADD\_ERROR*);

}

}

下面的例子中虽然将数字定义为了常量，但代码却并不容易理解：

/\*\*

\* 获取将子窗口绘制在父窗口中间时，子窗口的坐标。

\*

\* **@param** parentWindow 父窗口的位置

\* **@param** clientWindow 子窗口的位置

\* **@return** 子窗口在父窗口中间时的坐标

\*/

**public** Point getDrawCenter1(Rect parentWindow, Rect clientWindow)

{

Point drawCenter = **new** Point();

drawCenter.x = parentWindow.x + (parentWindow.width - clientWindow.width) / *HALF\_SIZE\_DIV*;

drawCenter.y = parentWindow.y + (parentWindow.height - clientWindow.height) / *HALF\_SIZE\_DIV*;

**return** drawCenter;

}

直接使用数字，代码反而更容易理解：

/\*\*

\* 获取将子窗口绘制在父窗口中间时，子窗口的坐标。

\*

\* **@param** parentWindow 父窗口的位置

\* **@param** clientWindow 子窗口的位置

\* **@return** 子窗口在父窗口中间时的坐标

\*/

**public** Point getDrawCenter2(Rect parentWindow, Rect clientWindow)

{

Point drawCenter = **new** Point();

drawCenter.x = parentWindow.x + (parentWindow.width - clientWindow.width) / 2;

drawCenter.y = parentWindow.y + (parentWindow.height - clientWindow.height) / 2;

**return** drawCenter;

}

## 空指针异常

### 解读

空指针异常是编码过程中最常见的异常，在使用一个对象的时候，如果对象可能为空，需要先判断对象是否为空，再使用这个对象。

在进行常量和变量的相等判断时，建议将常量定义为Java对象封装类型（如将int类型的常量定义为Integer类型），这样在比较时可以将常量放在左边，调用equals方法进行比较，可以省去不必要的判空。

### 案例

**public** **class** NullPointer

{

**static** **final** Integer *RESULT\_CODE\_OK* = 0;

**static** **final** Result *RESULT\_OK* = **new** Result();

**public** **void** printResult(Integer resultCode)

{

Result result = getResult(resultCode);

// result可能为null，造成空指针异常

**if** (result.isValid())

{

print(result);

}

}

**public** Result getResult(Integer resultCode)

{

// 即使resultCode为null，仍然可以正确执行，减少额外的判空语句

**if** (*RESULT\_CODE\_OK*.equals(resultCode))

{

**return** *RESULT\_OK*;

}

**return** **null**;

}

**public** **void** print(Result result)

{

...

}

}

## 下标越界

### 解读

访问数组、List等容器内的元素时，必须首先检查下标是否越界，杜绝下标越界异常的发生。

### 案例

**public** **class** ArrayOver

{

**public** **void** checkArray(String name)

{

// 获取一个数组对象

String[] cIds = ContentService.*queryByName*(name);

**if**(**null** != cIds)

{

// 只是考虑到cids有可能为null的情况，但是cids完全有可能是个0长度的数组，因此cIds[0]有可能数组下标越界

String cid=cIds[0];

cid.toCharArray();

}

}

}

## 字符串转数字

### 解读

调用Java方法将字符串转换为数字时，如果字符串的格式非法，会抛出运行时异常NumberFormatException。

### 案例

错误例子：

**public** Integer getInteger1(String number)

{

// 如果number格式非法，会抛出NumberFormatException

**return** Integer.*valueOf*(number);

}

正确的处理方法如下：

**public** Integer getInteger2(String number)

{

**try**

{

**return** Integer.*valueOf*(number);

}

**catch** (NumberFormatException e)

{

...

**return** **null**;

}

}

## 资源释放

### 解读

在使用文件、IO流、数据库连接等不会自动释放的资源时，应该在使用完毕后马上将其关闭。关闭资源的代码应该在try...catch...finally的finally内执行，否则可能造成资源无法释放。

### 案例

错误案例如下：

**public** **void** writeProduct1(ProductServiceStruct product)

{

**try**

{

FileWriter fileWriter = **new** FileWriter("");

fileWriter.append(product.toString());

// 如果append()抛出异常，close()方法就不会执行，造成IO流长时间无法释放

fileWriter.close();

}

**catch** (IOException e)

{

...

}

}

关闭IO流的正确方法如下：

**public** **void** writeProduct2(ProductServiceStruct product)

{

FileWriter fileWriter = **null**;

**try**

{

fileWriter = **new** FileWriter("");

fileWriter.append(product.toString());

}

**catch** (IOException e)

{

...

}

**finally**

{

// 不管前面是否发生异常，finally中的代码一定会执行

**if** (fileWriter != **null**)

{

**try**

{

fileWriter.close();

}

**catch** (IOException e)

{

...

}

}

}

}

## 循环体性能

### 解读

循环体是软件中最容易造成性能问题的地方，所以在进行循环体编码时务必考虑性能问题。

在循环体内重复使用且不会变化的资源（如变量、文件对象、数据库连接等），应该在循环体开始前构造并初始化，避免在循环体内重复和构造初始化造成CPU资源的浪费。

除非业务场景需要，避免在循环体内构造try...catch块，因为每次进入、退出try...catch块都会消耗一定的CPU资源，将try...catch块放在循环体之外可以节省大量的执行时间。

### 案例

**public** **void** addProducts(List<ProductServiceStruct> prodList)

{

**for** (ProductServiceStruct product : prodList)

{

// prodSrv在每次循环时都会重新获取，造成不必要的资源消耗

ProductService prodSrv = (ProductService) ServiceLocator.*findService*(ProductService.**class**);

// 避免在循环体内try...catch，放在循环体之外可以节省执行时间

**try**

{

prodSrv.addProduct(product);

}

**catch** (BMEException e)

{

...

}

}

}

## 数据类重载toString()方法

### 解读

数据类如果没有重载toString()方法，在记录日志的时候会无法记录数据对象的属性值，给定位问题带来困难。

### 案例

**public** **class** MdspProductExt

{

**private** String key;

**private** String value;

**public** String getKey()

{

**return** key;

}

**public** **void** setKey(String key)

{

**this**.key = key;

}

**public** String getValue()

{

**return** value;

}

**public** **void** setValue(String value)

{

**this**.value = value;

}

}

**class** BusinessProcess

{

**private** DebugLog log = LogFactory.*getDebugLog*(BusinessProcess.**class**);

**public** **void** doBusiness(MdspProductExt prodExt)

{

**try**

{

...

}

**catch** (PMSException e)

{

// MdspProductExt未重载toString()方法，日志中无法记录对象内属性的值，只能记录对象地址

log.error("error while process prodExt " + prodExt);

}

}

}