电信软件与核心网业务部编程军规-java问题-案例

# 【违背军规1】避免在程序中使用魔鬼数字，必须用有意义的常量来标识。

**说明**：是否是魔鬼数字要基于容易阅读和便于全局替换的原则。0、1作为某种专业领域物理量枚举数值时必须定义常量，严禁出现类似NUMBER\_ZERO的“魔鬼常量”。

## 案例一

### **【问题描述】**

根据操作类型分别调用不同的接口，这个时候，操作类型值写成了枚举值，没有定义常量，导致不便于阅读。

### **【错误代码】**

if (1 == operateType)

{

}

else if (2 == operateType)

{

}

else

{

}

### **【正确代码】**

if (ELBOperateVmType.ADD\_VM\_TO\_ELB == operateType)

{

}

else if (ELBOperateVmType.DEL\_VM\_TO\_ELB == operateType)

{

}

else

{

}

# 【违背军规2】明确方法的功能，一个方法仅完成一个功能。

**说明**：方法功能太多，会增加方法的复杂度和依赖关系，不利于程序阅读和将来的持续维护，无论是方法还是类设计都应符合单一职责原则。

## 2.1案例一

### **【问题描述】**

原子功能方法，添加了业务逻辑，在新增的时候，又做了修改，这个应该在业务方法处理。

### **【错误代码】**

public void addElbInfo(ElbInfo elbInfo)

{

// 更新ELB

update(elbInfo);

// 新增ELB

CmdRequest cmdRequest = new CmdRequest();

cmdRequest.setNamingLabel("addElbInfo");

DataResult dataResult = DAS\_CMD.execute(cmdRequest);

}

### **【正确代码】**

public void addElbInfo(ElbInfo elbInfo)

{

// 新增ELB

CmdRequest cmdRequest = new CmdRequest();

cmdRequest.setNamingLabel("addElbInfo");

DataResult dataResult = DAS\_CMD.execute(cmdRequest);

}

# 【违背军规3】方法参数不能超过5个

**说明：**参数太多影响代码阅读和使用，为减少参数，首先要考虑这些参数的合理性，保持方法功能单一、优化方法设计，如果参数确实无法减少，可以将多个参数封装成一个类（对象），同时考虑在新的类（对象）中增加相应的行为，以期更符合OOP。

## 3.1案例一

### **【问题描述】**

方法参数超过5个，影响了可阅读性。

### **【错误代码】**

**public** CreateImageRsp createImage(String accountId, String userId, String imageDesc, String instanceId,

String imageName, **int** imageFormat)

{

}

### **【正确代码】**

**public** CreateImageRsp createImage(CreateImageReq req)

{

}

# 【违背军规4】方法调用尽量不要返回null，取而代之以抛出异常，或是返回特例对象（SPECIAL CASE object，SPECIAL CASE PATTERN）；对于以集合或数组类型作为返回值的方法，取而代之以空集合或0长度数组。

**说明：**返回null会增加不必要的空指针判断，遗漏判断也会导致严重的NullPointerException错误。

## 3.1案例一

### **【问题描述】**

如果方法没有注释说明，且调用者不进行空判断，则会发生严重的空指针异常。

### **【错误代码】**

**public** List<Host> queryElbAndVmRelPageList(Host queryCondition, Page page)

{

List<Host> hostList = null;

try

{

//处理

hostList.add(…);

}

catch (DASException e)

{

}

return hostList;

}

### **【正确代码】**

**public** List<Host> queryElbAndVmRelPageList(Host queryCondition, Page page)

{

List<Host> hostList = new ArrayList<Host>();

try

{

//处理

hostList.add(…);

}

catch (DASException e)

{

}

return hostList;

}

# 【违背军规5】在进行数据库操作或IO操作时，必须确保资源在使用完毕后得到释放，并且必须确保释放操作在finally中进行。

**说明：**数据库操作、IO操作等需要关闭对象必须在try -catch-finally 的finally中close()，如果有多个IO对象需要关闭，需要分别对每个对象的close()方法进行try-catch,防止一个IO对象关闭失败其他IO对象都未关闭。

## 3.1案例一

### **【问题描述】**

如果数据库操作、Io操作不在finally块进行关闭，出现资源不能释放，可能会导致系统卡死。

### **【错误代码】**

**try**

{

conn = SqlHelp.*getConnection*();

}

**catch** (DASException e)

{

SqlHelp.*closeConnection*(rs, ps, **null**);

}

### **【正确代码】**

**try**

{

conn = SqlHelp.*getConnection*();

}

**catch** (DASException e)

{

}

**finally**

{

SqlHelp.*closeConnection*(rs, ps, **null**);

}

# 【违背军规6】异常捕获不要直接 catch (Exception ex) ，应该把异常细分处理。

**说明：**catch (Exception ex)的结果会把RuntimeException异常捕获，RuntimeException是运行期异常，是程序本身考虑不周而抛出的异常，是程序的BUG，如无效参数、数组越界、被零除等，程序必须确保不能抛出RuntimeException异常，不允许显示捕获RuntimeException异常就是为了方便测试中容易发现程序问题。

## 3.1案例一

### **【问题描述】**

直接捕获Exception，可能会导致不同的异常分支遗漏或者捕获了运行时异常，二者不能明确区分，导致处理不合理或者不便于问题发现。

### **【错误代码】**

try

{

// 文件操作

// 日期转换

// 数字字符转换  
}

catch (Exception e)

{

}

### **【正确代码】**

try

{

// 文件操作

// 日期转换

// 数字字符转换  
}

catch (FileNotFoundException e)

{

}

catch (IOException e)

{

}

catch (ParseException e)

{

}

catch (NumberFormatException e)

{

}

catch (Exception e)

{

}

# 【违背军规7】对于if else if (后续可能有多个else if …)这种类型的条件判断，最后必须包含一个else分支，避免出现分支遗漏造成错误；每个switch-case语句都必须保证有default，避免出现分支遗漏，造成错误。

**说明：**if () else if()最后追加一个else的原因时避免分支遗漏

## 3.1案例一

### **【问题描述】**

if 分支一般都要有对应的else分支，避免逻辑遗漏，如果是if () else if() 的情况，则必须要在最后添加一个else分支，以表示分支处理完毕。

### **【错误代码】**

if (ELBOperateVmType.ADD\_VM\_TO\_ELB == operateType)

{

}

else if (ELBOperateVmType.DEL\_VM\_TO\_ELB == operateType)

{

}

### **【正确代码】**

if (ELBOperateVmType.ADD\_VM\_TO\_ELB == operateType)

{

}

else if (ELBOperateVmType.DEL\_VM\_TO\_ELB == operateType)

{

}

else

{

}

# 【违背军规8】覆写对象的equals()方法时必须同时覆写hashCode()方法。

**说明：**equals和hashCode方法是对象在hash容器内高效工作的基础，正确的覆写这两个方法才能保证在hash容器内查找对象的正确性，同时一个好的hashCode方法能大幅提升hash容器效率。

## 3.1案例一

### **【问题描述】**

在重写对象的equals方法是，没有重写hashCode方法，可能导致错误处理或者效率低下。

### **【错误代码】**

**public** **class** RelationshipInfor

{

/\*\*

\* 资源信息

\*/

**private** Resource resource;

/\*\*

\* {@inheritDoc}

\*/

**public** **boolean** equals(Object obj)

{

}

}

### **【正确代码】**

**public** **class** RelationshipInfor

{

/\*\*

\* 资源信息

\*/

**private** Resource resource;

/\*\*

\* {@inheritDoc}

\*/

**public** **boolean** equals(Object obj)

{

}

/\*\*

\* {@inheritDoc}

\*/

**public** **int** hashCode()

{

}

}

# 【违背军规9】禁止循环中创建新线程，尽量使用线程池。

**说明：**此类情况是避免资源浪费和线程安全问题

## 3.1案例一

### **【问题描述】**

在循环中循环new线程，导致性能低下。

### **【错误代码】**

try

{

while (true)

{

Thread tt = new Thread();

tt .start()

}

}

catch (InterruptedException e)

{  
}

### **【正确代码】**

try

{

while (true)

{

threadPool. execute(command);

}

}

catch (InterruptedException e)

{  
}

# 【违背军规10】在进行精确计算时(例如:货币计算)避免使用float和double，浮点数计算都是不精确的，必须使用BigDecimal或将浮点数运算转换为整型运算。

**说明：**浮点运算在一个范围很广的值域上提供了很好的近似，但是它不能产生精确的结果。二进制浮点对于精度计算是非常不适合的，因为它不可能将0.1——或者10的其它任何次负幂精确表示为一个长度有限的二进制小数。

## 3.1案例一

### **【问题描述】**

浮点数计算不准确，导致计费、计数错误。

### **【错误代码】**

double dubleFIrst = 0.0000;

double dubleSecond = 0.0001;

double dubleThree = dubleFIrst \* dubleSecond

### **【正确代码】**

try

{

double dubleFIrst = 0.0000;

double dubleSecond = 0.0001;

BigDecimal bigDecimalFirst = new BigDecimal(dubleFIrst);

BigDecimal bigDecimalSecond = new BigDecimal(dubleSecond);

BigDecimal bigDecimalThreee = bigDecimalFirst.multiply(bigDecimalSecond);

bigDecimalThreee = bigDecimalThreee.setScale(4, BigDecimal.ROUND\_HALF\_UP);

double dubleThree = Double.parseDouble(String. valueOf (bigDecimalThreee));

}

catch (ParseException e)

{  
}