|  |  |
| --- | --- |
| 产品名称 | 密级 |
| SIS | 内部公开 |
| 产品版本 | 共 页 |
| ALL |

SIS编码及评审规范

(仅供内部使用)

|  |  |  |  |
| --- | --- | --- | --- |
| 拟制: |  | 日期： | 2016-04-16 |
| 审核: |  | 日期： |  |
| 批准: |  | 日期： |  |



华为技术有限公司

版权所有 侵权必究

修订记录

|  |  |  |  |
| --- | --- | --- | --- |
| 日期 | 修订版本 | 修改描述 | 作者 |
| 2016-4-16 | 1.0 | 初稿完成 | 于凯 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

目 录

[1. 规范设计原则 5](#_Toc454872403)

[2. 规范适用范围 5](#_Toc454872404)

[3. 规范说明 5](#_Toc454872405)

[3.1. 避免使用字符串拼结方式生成数据库操作语句，以防止SQL注入 5](#_Toc454872406)

[3.2. 打印错误日志时，应包含异常堆栈信息 6](#_Toc454872407)

[3.3. 对于资源（非资源池方式）的操作，在finally中释放对资源的占用 7](#_Toc454872408)

[3.4. 将字符串转换为数字时必须处理NumberFormatException异常 8](#_Toc454872409)

[3.5. 不采用防御式编程，默认由函数调用、使用者保证参数及返回值的合法、有效性 8](#_Toc454872410)

[3.6. 访问数组或集合对象时，需要判断数组和集合的大小，防止数组越界 9](#_Toc454872411)

[3.7. 存在多线程并发修改的变量不要定义为类成员变量，尽量使用参数进行传递 10](#_Toc454872412)

[3.8. 放在Set或者Map容器key中的对象确保覆写equals()和hashCode()方法 11](#_Toc454872413)

[3.9. 使用集合类时注意是否是线程安全的 13](#_Toc454872414)

[3.10. 命名规范性 14](#_Toc454872415)

[3.11. 注释规范性 16](#_Toc454872416)

[3.12. 日志规范性 17](#_Toc454872417)

[3.13. 魔鬼数字 19](#_Toc454872418)

[3.14. 空指针异常 20](#_Toc454872419)

[3.15. 循环体性能 21](#_Toc454872420)

[3.16. 尽量避免使用递归算法 22](#_Toc454872421)

[3.17. 数据类重载toString()方法 23](#_Toc454872422)

[3.18. 重复代码的处理 24](#_Toc454872423)

[3.19. 其它注意事项 24](#_Toc454872424)

[3.20. 安全编程相关规范 24](#_Toc454872425)

[3.21. 基线定制划分（ABC分层）相关规范 25](#_Toc454872426)

[3.22. 升级脚本（SQL） 25](#_Toc454872427)

[3.23. 升级脚本（Shell） 27](#_Toc454872428)

[3.24. 数据库查询相关代码 27](#_Toc454872429)

[3.25. 新开局局点配置 27](#_Toc454872430)

[3.26. 升级包中解密数据库密码 30](#_Toc454872431)

[3.27. 基线和定制配置项统一使用sysdb中的SIS\_T\_CONFIG表 30](#_Toc454872432)

[3.28. 非SIS维护的表的处理规范 30](#_Toc454872433)

[4. 常见问题及典型案例 31](#_Toc454872434)

[4.1. 事务 31](#_Toc454872435)

[4.2. 多线程 32](#_Toc454872436)

[4.3. 异常处理 35](#_Toc454872437)

[4.4. 单例模式 35](#_Toc454872438)

SIS编码及评审规范

# 规范设计原则

规范不是checklist，不可能关注到所有细节，本规范只针对非技术能力原因造成的低级错误。规范是需要牢牢记住的，规范所制定的条例是编程中铁的纪律。

# 规范适用范围

SIS开发人员。

# 规范说明

## 避免使用字符串拼结方式生成数据库操作语句，以防止SQL注入

【说明】当需要操作数据库时，不要直接使用字符串拼结的方式生成操作语句传给数据库。因为操作语句一般需要拼结外部传入的参数，而参数是不被信任的，恶意构造的参数可能对系统的功能和安全性造成破坏。对于操作数据库较少的系统，建议使用PreparedStatement对象操作数据库，对于操作数据库较多的系统，建议使用iBatis或是Hibernate等数据对象映射框架。

**错误：**

strSql = "select \* from user where username='"+ username + "' and password='" + password + "'";

result = statement.executeQuery(strSql);

//username传入"1=1’ or ’1=1"，password为任何值，则sql变成了”select \* from user where username=’1=1’ or ’1=1’ and password=’xxx’”，这条语句能查到数据，也就绕过了用户名密码验证

**正确：**

string strSql = "select \* from user where username=? and password=?";

PreparedStatement preparedStatement = connection.prepareStatement(strsql);

preparedStatement.setstring(1, username);

preparedStatement.setstring(2, password);

ResultSet resultSet = preparedStatement.executeQuery();

//PreparedStatement会自动转换参数中的特殊字符，如错误示例中的单引号

[【返回首页】](#_top)

## 打印错误日志时，应包含异常堆栈信息

【说明】业务代码中捕获到异常时，通常需要打印错误日志，在错误日志中必须将原始异常的堆栈信息打印出来。如果是捕获后直接向外抛出异常，则需要将原始异常对象包装后再抛出。因为异常对象中保存了函数调用堆栈，堆栈中包含了出错时函数的调用顺序以及调用的行数，该信息对于问题分析至关重要，特别是对于无法debug代码的已上线局点。

注：如果是包装成BMEException，如果日志打印这个异常对象，是看不到原始的异常堆栈的，可以在转换前把原始异常打出来，也可以之后用getCause()获取到原始异常打印出来。注意如果涉及敏感信息，比如可能会暴露代码路径结果的**FileNotFoundException等，应打印其它简洁模糊的信息，部件自己能够识别就可以。**

注：特殊场景，认为通过ex.toString()就能把异常描述清楚了，而且不会占用太多的日志，没必要每次异常都需要打印异常堆栈，也可以。

**错误：**

try

{

……

}

catch (IOException e)

{

log.error("Some error message.");

}

catch (IndexOutOfBoundsException e)

{

throw new MyException("Some error message.");

}

**正确：**

try

{

……

}

catch (IOException e)

{

log.error("Some error message.", e);

}

catch (IndexOutOfBoundsException e)

{

throw new MyException("Some error message.", e);

}

[【返回首页】](#_软件开发军规细则)

## 对于资源（非资源池方式）的操作，在finally中释放对资源的占用

【说明】系统资源（IO、数据库、FTP……）是有限的，如果线程都占用着资源不释放，资源总会达到枯竭。Java虽然有自动回收机制，但IO等资源在底层可能是非Java的native函数实现资源申请和分配，这些资源实际是非托管资源，自动回收机制无法回收非托管资源，所以需要编码手动释放。建议对于流的关闭，各产品使用各自的工具方法操作。

错误：

BufferedOutputStream out = null;

try

{

FileOutputStream fos = new FileOutputStream("dest2");

out = new BufferedOutputStream(fos);

......

out.close(); //在关闭操作前代码抛出异常的情况下，关闭操作不被执行

}

catch (FileNotFoundException e)

{

// TODO

}

catch (IOException e)

{

// TODO

}

正确：

BufferedOutputStream out = null;

try

{

FileOutputStream fos = new FileOutputStream("dest2");

out = new BufferedOutputStream(fos);

......

}

catch (FileNotFoundException e)

{

// TODO

}

catch (IOException e)

{

// TODO

}

finally

{

// 关闭流操作

}

[【返回首页】](#_top)

## 将字符串转换为数字时必须处理NumberFormatException异常

【说明】在字符串转换成数字时可能抛异常，对于转换时不能保证该字符串一定为数字时，必须对该异常进行捕获并根据业务需要进行相应的处理。BFM统一使用封装好的工具类NumberParser，工具类内部已经完成上述处理，不允许额外另行实现。

错误：

try

{

int value = Integer.valueOf(number);

//TODO

}

catch (NumberFormatException e)

{

//TODO

}

正确：

int value = NumberParser.string2Integer(number, 0);

[【返回首页】](#_top)

## 不采用防御式编程，默认由函数调用、使用者保证参数及返回值的合法、有效性

【说明】对于方法调用的返回值，任何返回都有可能，在使用返回值前需要进行合法性校验。

对于方法调用的输入，任何输入都有可能，在使用输入参数前需要进行合法性校验。

错误：

UserInfo userInfo = UserManager.getInstance.getUser(userName);

userInfo.getPassword(); // 没有判断获取的UserInfo对象是否为null

正确：

UserInfo userInfo = UserManager.getInstance.getUser(userName);

if (null != userInfo)

{

userInfo.getPassword();

}

下面的场景：

public String getUserPassword(UserInfo userInfo)

{

return userInfo.getPassword(); //没有判断传入的UserInfo对象是否为null

}

不强制要求做这样的判断：

public String getUserPassword(UserInfo userInfo)

{  
 if (null != userInfo)

{

return userInfo.getPassword();

}

else

{

// TODO

}

}

[【返回首页】](#_top)

## 访问数组或集合对象时，需要判断数组和集合的大小，防止数组越界

【说明】如果再没有判断数据或集合的大小下直接访问，可能因为数据并没有期望的大小而引发数据越界异常。

错误：

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();

}

}

正确：

public void checkArray(String name)

{

// 获取一个数组对象

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

if(null != cIds && cIds.length > 0)

{

String cid = cIds[0];

cid.toCharArray();

}

}

【[返回首页](#_top)】

## 存在多线程并发修改的变量不要定义为类成员变量，尽量使用参数进行传递

【说明】多线程场景下，如果两个线程操作同一个实例，都对类中的成员进行操作，则实例中该成员是不可预期的。

错误：

public class UserManager

{

private static UserInfo user; // 定义为静态成员变量，可能引起多线程并发问题

......

public static int insertUser()

{

return new UserDao.insert(user);

}

......

private static setUserInfo(UserInfo user)

{

UserManager.user = user;

}

}

正确：

public class UserManager

{

......

public static int insertUser(UserInfo user) // 通过接口参数传入

{

return new UserDao.insert(user);

}

......

}

[【返回首页】](#_军规细则)

## 放在Set或者Map容器key中的对象确保覆写equals()和hashCode()方法

【说明】Set或者Map容器key中的对象都是通过hashCode和equals方法进行存取的，如果不进行覆写，则不能正确的进行读取操作。

错误：

public class UserInfo()

{

private String name;

public UserInfo(String name)

{

this.setName(name);

}

public void setName(String name)

{

this.name = name;

}

public String getName()

{

return this.name;

}

}

......

Set<UserInfo> set = new HashSet<UserInfo>();

UserInfo user = new UserInfo("13712345678");

UserInfo user2 = new UserInfo("13712345678");

set.add(user);

set.add(user2); // set中存在两个13712345678用户

......

正确：

public class UserInfo()

{

private String name;

public UserInfo(String name)

{

this.setName(name);

}

public void setName(String name)

{

this.name = name;

}

public String getName()

{

return this.name;

}

public int hashCode()

{

return this.name \* 31;

}

public boolean equals(Object obj)

{

if (null == obj)

{

return false;

}

if (!(obj instanceof UserInfo))

{

return false;

}

return this.name.equals(((UserInfo)obj).name);

}

}

......

Set<UserInfo> set = new HashSet<UserInfo>();

UserInfo user = new UserInfo("13712345678");

UserInfo user2 = new UserInfo("13712345678");

set.add(user);

set.add(user2); // set中只会存在一个13712345678的用户

......

[【返回首页】](#_top)

## 使用集合类时注意是否是线程安全的

【说明】Java的集合类中，有线程安全的，也有线程不安全的，前者有HashTable、Vector等，后者有HashMap、HashSet、ArrayList、LinkedList等。在编码时，多线程下应使用线程安全的类，或者使用synchronized加锁，或使用java.util.concurrent包中的对应类，或使用java.util.Collections的synchronizedList()、synchronizedMap()对集合进行包装，而单线程下可使用线程不安全类以提升性能。

错误：

public class SomeJob extends Thread

{

private static Map<String, Integer> counter = new HashMap<String, Integer>();

public void run()

{

for (String key : counter.keySet())

{

counter.put(key, counter.get(key) + 1);

}

}

}

正确：

public class SomeJob extends Thread

{

private Map<String, Integer> counter = new HashMap<String, Integer>();

public void run()

{

synchronized (counter)

{

for (String key : counter.keySet())

{

counter.put(key, counter.get(key) + 1);

}

}

}

}

[【返回首页】](#_top)

## 命名规范性

【说明】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）。

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);

}

}

}

[【返回首页】](#_top)

## 注释规范性

【说明】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。

/\*\*

\* <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** ...;

}

[【返回首页】](#_top)

## 日志规范性

【说明】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WARN或以上级别记录异常栈**（不会引起程序中断的使用WARN，会引起程序中断，需要往上层抛出的异常使用ERROR）**，否则会导致异常栈丢失**（需要注意不要包含敏感信息，敏感异常要封装后再抛出）**，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。

捕获异常的地方一定要记录日志。谁捕获谁记日志，一旦捕获必须打日志

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);

}

}

[【返回首页】](#_top)

## 魔鬼数字

【说明】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。

**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;

}

[【返回首页】](#_top)

## 空指针异常

【说明】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。

**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)

{

...

}

}

[【返回首页】](#_top)

## 循环体性能

【说明】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。

**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)

{

...

}

}

}

[【返回首页】](#_top)

## 尽量避免使用递归算法

【说明】递归算法存在资源消耗过大，可能陷入死循环，造成栈溢出或堆冲突，耗光数据库连接等问题。在实际的代码编写中应尽量避免使用递归算法。或者设置合理的递归深度进行控制。

可能存在问题的递归算法，当其中的后缀系统级配置项accessCodeRandomRangeBegin和accessCodeRandomRangeEnd的间隔比较短时，可能会造成无限递归：

private String generatAccesscode(Carrier carrier, UserID userID)

{

log.info("enter generatAccesscode", "carrier", carrier);

//前缀子网级可以配置

String accessCodeForSubProd = carrier.getExt().get("accessCodeForSubProd");

if (null == accessCodeForSubProd)

{

accessCodeForSubProd = carrier.getExt().get("ACCESSCODEFORSUBPROD");

}

//后缀系统级配置

int accessCodeRandomRangeBegin = ConfigManage.getIntValue("accessCodeRandomRangeBegin", 10000);

int accessCodeRandomRangeEnd = ConfigManage.getIntValue("accessCodeRandomRangeEnd", 19000);

Random random = new Random();

int accessCodeRandom =

random.nextInt(accessCodeRandomRangeEnd) % (accessCodeRandomRangeEnd - accessCodeRandomRangeBegin + 1)

+ accessCodeRandomRangeBegin;

String accessCode = accessCodeForSubProd + accessCodeRandom;

//TokenExt tokenExt = mtnTokenService.queryTokenExtByToken(null, accessCode);

boolean isContainAccessCode = isContainAccessCode(carrier, userID, accessCode);

if (isContainAccessCode)

{

//自身调用

log.info("enter cycle generatAccesscode");

accessCode = generatAccesscode(carrier, userID);

}

log.info("out generatAccesscode", "generatAccesscode", accessCode);

return accessCode;

}

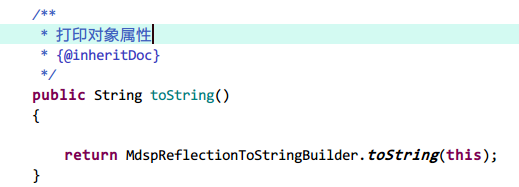
[【返回首页】](#_top)

## 数据类重载toString()方法

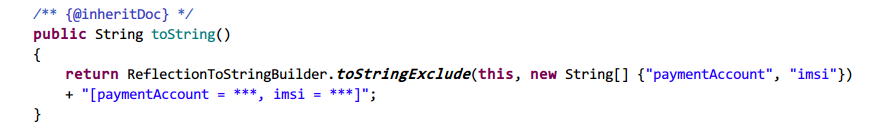
【说明】数据类如果没有重载toString()方法，在记录日志的时候会无法记录数据对象的属性值，给定位问题带来困难。

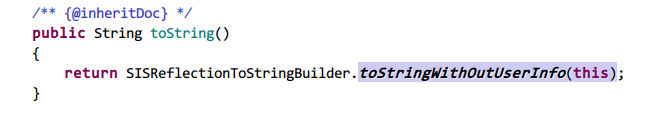
按照目前SIS的实现分成两种情况：

1. 不包含敏感字段的数据类。



1. 包含敏感字段的数据类

**或者**



[【返回首页】](#_top)

## 重复代码的处理

【说明】代码重复率也是代码质量的一个评定标准，如果存在6行以上的带有业务逻辑的代码重复，建议抽取公共的方法。常量如果在2个类以上有使用，建议抽取到公共的常量类，不包含上下文中的key。

定制Flow中的step时，要遵守步骤原子化原则。基线流程（flow）被定制时，应该不涉及到步骤(step)的修改，只涉及步骤(step)的新增和删除。反之定制流程(flow)被收编时，也须满足同样原则。如果有实在不能满足的可以拿出来讨论。

[【返回首页】](#_top)

## 其它注意事项

【说明】可见《Java语言编程规范 上卷 综合篇V2.0.docx》。

重点关注以下几条原则：

2.原则4：谨慎使用静态成员变量

6：多线程并发

8：性能与资源管理

[【返回首页】](#_top)

## 安全编程相关规范

【说明】可见《Java语言编程规范\_下卷\_安全篇\_V2.0.docx》。

[【返回首页】](#_top)

## 升级脚本（SQL）

【说明】

1. 所有SQL，存储过程等都必须支持可重复执行。
2. 所有SQL、存储过程等都不能携带数据库用户名，不同db用户下执行的脚本写在各自的文件夹中。（sysdb，userdb）
3. 升级脚本不允许跨db用户访问。
4. 如果表新增字段有默认值，但是没有非空属性，速度非常慢，请在升级时关注此类操作，避免超时。具体见下图：



1. 数据库回滚脚本规则

1）加表 回滚

2）加字段 不允许非空字段 不回滚

3）改字段 不允许改类型，只允许增加长度 不回滚

4）改视图 回滚

5）加索引 回滚

6）加配置项 必须支持可重复升级 不回滚

7）修改配置项 回滚

8）加定时任务 回滚

9）修改定时任务 回滚

10）预置业务数据 回滚

11）大表清理的存储过程修改 回滚

12）修改序列 不回滚

13）加存储过程 回滚

14）修改存储过程 回滚

15）加job 回滚

1. 除了以上，其它请参考“数据库”目录下其它相关规范文档。

[【返回首页】](#_top)

## 升级脚本（Shell）

【说明】

1. 所有脚本必须支持可重复执行。
2. 涉及配置文件（应用、web容器、操作系统等）的修改，需要事先收集现网、测试床、解决方案环境中的文件拿回来进行比对，做到兼容修改。
3. 脚本执行错误的情况下，一定要记录ERROR日志，并且return 1（标识脚本执行错误）。
4. 请大家补充。

[【返回首页】](#_top)

## 异常场景日志打印规范

【说明】

1、打印异常除了打印异常堆栈还要把：入参和出参 等关键信息用ERROR级别打印出来，同时也要注意安全规范，不要打印敏感信息。

[【返回首页】](#_top)

## 查询场景数据大于1024时注意事项

【说明】

1. 由于bme的查询最大记录默认是1024条记录，当table中超过1024条时，去查询全量数据的时候，会始终只能获取前面的1024 条，导致查询记录不全，引入功能性问题。
2. 约束：查询场景尽量带条件或者分页查询，并且评估查询是否合理。

[【返回首页】](#_top)

## 涉及IP地址配置

【说明】

1. 需要从现网搜集包含源IP信息的配置文件或者数据库表；
2. 确认该IP地址应该配置为fabric平面的还是basic平面的；
3. 在升级指导书中增加傻瓜说明，比如检查源文件中的IP信息与数据库表中的某个IP是否一致，提示一线升级前检查，确保脚本读取的IP信息正确。

[【返回首页】](#_top)

# 常见问题及典型案例

## 事务

【说明】

SIS涉及的事务配置文件：

1、PMP：bme.common.service.xml，bme.domain.service.xml。

2、SIS：global.datasource.xml，aop.service.xml，custom.aop.service.xml，framework.flow.service.xml

1、对于需要批量提交的场景（一次操作涉及多张表，多个业务场景涉及到对同个表相同字段的增删改的时候防止长时间等待或冲突锁表等），一定要包含在同一个事务中，进行事务配置。

案例：C0x：沙特mobily，埃及mobinil，孟加拉robi

高并发接口涉及用户锁的表的批量删除和新增访问，由于存在操作同一条数据且没有配置事务，导致锁表，不释放数据库连接，导致连接数满，接口报错。

2、SIS的DAO层基本上都是使用的BME的DAS，使用的是Spring的JTA分布式事务管理，使用tx标签配置的拦截器。

3、还有其它的事务管理方式：使用JDBC方式进行事务处理，Java事务API（JTA；Java Transaction API）和它的同胞Java事务服务(JTS；Java Transaction Service)，为J2EE平台提供了分布式事务服务等等。

4、Spring的事务管理还有其它配置方式：给每个BEAN配置一个事务代理；所有BEAN共享一个事务代理基类；普通的拦截器；全注解。

5、SIS建议使用2中的事务管理方式和配置方式，特殊情况下，可以考虑3和4中提到的事务管理方式和配置方式。

[【返回首页】](#_top)

## 多线程

【说明】

1、对公用的全局变量，静态变量或者缓存，如果是固定不会去修改的，在多线程处理中一定不能对其进行修改，如果要修改返回某个对象，需要进行Clone后再处理。

案例：

MTN：com.huawei.mdsp.sis.route.RouteContext.getRoute(String)

错误代码示例：

public Route getRoute(String id)

{

Route route = getRoute(calcInstNum(id));

if (Constants.USERROUTEWAY\_BY\_NOSEG == ConfigManage.getIntValue(ConfigConstants.SysConf.USERROUTEWAY,

Constants.USERROUTEWAY\_BY\_MD5))

{

Route routeByID = getRouteByID(id);

routeByID.setCbpNode(route.getCbpNode());**//这里改变了缓存中对象的属性值。导致其它线程来获取时得到错误的值。**

//如果是伊朗子网，会设置多个用户库，则bmpdbnode需要设置成散列路由的方式

if (NEEDSETBMPNODE.equals(ConfigManage.getValue("setBmpDbNode", routeByID.getOperatorID())))

{

LOG.info("Iran's flow need set bmpnode again", "routeByID", routeByID, "md5idm", calcInstNum(id));

setBmpDbNode(routeByID, calcInstNum(id));

LOG.info("set bmpnode end", "routeByID", routeByID);

}

return routeByID;

}

return route;

}

正确代码示例：

public Route getRoute(String id)

{

Route route = getRoute(calcInstNum(id));

if (Constants.USERROUTEWAY\_BY\_NOSEG == ConfigManage.getIntValue(ConfigConstants.SysConf.USERROUTEWAY,

Constants.USERROUTEWAY\_BY\_MD5))

{

Route routeByID = getRouteByID(id);

**// 这里从缓存对象创建了一个新对象进行处理，不会影响其它线程从缓存中取值**

Route reSetRoute = new Route();

convertRoute(routeByID, reSetRoute);

reSetRoute.setCbpNode(route.getCbpNode());

//如果是伊朗子网，会设置多个用户库，则bmpdbnode需要设置成散列路由的方式

if (NEEDSETBMPNODE.equals(ConfigManage.getValue("setBmpDbNode", reSetRoute.getOperatorID())))

{

LOG.info("Iran's flow need set bmpnode again", "reSetRoute", reSetRoute, "md5idm", calcInstNum(id));

setBmpDbNode(reSetRoute, calcInstNum(id));

LOG.info("set bmpnode end", "reSetRoute", reSetRoute);

}

return reSetRoute;

}

return route;

}

2、对公用的全局变量，静态变量或者缓存，如果是需要去修改的，需增加同步机制。或者使用线程安全集合在多线程间共享可变数据，应首选java.util.concurrent包中提供的集合如ConcurrentHashMap、ConcurrentSkipListSet等。不仅线程安全而且性能更好。多线程操作同一个字符串相加，应采用StringBuffer，是线程安全的。

案例：

C1X：同一订购关系工单处理并发处理时会出现乱序乱序问题：com.huawei.mdsp.ccengine.workorder.utils.WorkOrderLock

在每次工单处理的入口进行判断是否有相同订购关系时间上更加新的已经处理过了，如果处理过了这条工单就不处理了，其中使用了ConcurrentHashMap保存已经处理过的工单信息。

3、禁止不加控制的创建新线程，Java虚拟机能够管理的线程数量有限，不加控制的创建新线程可能会导致Java虚拟机崩溃。建议使用JAVA自带的ThreadPoolExecutor或者spring的org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor来管理线程资源。

可以自行在工程中找相关的配置和代码，比如：

thread.service.xml

com.huawei.sis.framework.workorder.executor.WorkOrderExecutor

com.huawei.sis.framework.workorder.executor.WorkOrderSpareExecutor

4、创建新线程时需指定线程名。指定线程名可以给问题定位带来很多方便。日志或者dump中会包含线程的名称，而缺省的线程名Thread-n无法区分出是哪个线程，给问题定位带来不便。

使用Thread，在创建的时候加上名称，new Thread(Runnable target, String name)；或者使用setName的方法。

如果是使用线程池执行runnable的线程，可以在线程的run方法入口增加Thread.currentThread().setName("threadName");

5、当对TP又要求的实时接口，如果流程中有使用到线程池，除了要注意以上第三点之外，还要看在线程池中处理的业务逻辑的耗时，如果耗时很长，主线程会因为线程池满而在排队等待加入线程队列，造成接口的响应时长变大，影响TPS。如果耗时很长的话需要进行优化尽量减少线程执行需要的时间。

案例：

印尼XL：com.huawei.mdsp.ccengine.task.notify.SubRelationNotifyTask

同步通知，SP返回跟IB延迟高，需要改成异步通知，通过使用另起一个线程去进行通知，但是通知本身还是同步的，由于高延迟导致线程池满，接口在塞线程池时发生了等待还是会造成接口延迟很高。在线程中改成入库通知消息，异步通知的方式解决。

[【返回首页】](#_top)

## 异常处理

【说明】

[【返回首页】](#_top)

## 单例模式

【说明】具体的危害可见3.7和3.18小节中的说明，以下列了对SIS代码排查的结果。涉及C8x基线和C1x定制。

1、 com.huawei.sis.framework.cache.impl.CachedServiceManagerImpl

其中删除了对FuckingDamnDependence的依赖，我看了下确实应该是用不到了。

其中变量cachedServiceMap是有状态的，但从代码的实现中，实现接口CachedService的实现类在初始化的时候被put进去，之后没有变化，只有当触发clearAll的时候去遍历各个实现类去调用其中的clear方法，各个实现类中clear的对象是线程安全的。对cachedServiceMap本身没有修改。

2、 com.huawei.mdsp.sis.trace.ReportItraceMsg

getInstance()方法加了同步锁，保证获取的对象为当前1个

3、 com.huawei.mdsp.sis.route.RouteContext

单例模式下private List<Route> routes = new ArrayList<Route>();是都有状态的。

把其中对routes的操作方法（addRoute，removeRoute，clear，loadNoSegRoutes），加了同步锁。

4、 com.huawei.mdsp.sis.range.SubscriberKeyRangeContext

单例模式下rangeMap是都有状态的。由线程不安全的HashMap改成了线程安全的ConcurrentHashMap

5、 com.huawei.mdsp.ccengine.common.configuration.Configuration4HttpMsg

单例模式下private static UConfiguration uConfig = null;是有状态的，初始化这个对象的方法getHttpMsgUconfig ()加了同步锁，保证获取的对象为当前1个

6、 com.huawei.mdsp.ccengine.common.configuration.Configuration4NotifyDevice

单例模式下private static UConfiguration uConfig = null;是有状态的，初始化这个对象的方法getUconfig()加了同步锁，保证获取的对象为当前1个

7、 com.huawei.mdsp.ccengine.common.performance.SystemResource

MEMLIST和CPULIST改成private的。在对他们的add方法getMemAndCpuParam()和get方法加了同步锁，保证MEMLIST和CPULIST不会出现访问冲突。

sis\_dfx工程

8、 com.huawei.mdsp.ccengine.common.configuration.Configuration4Ftp

init和save方法加同步锁。

9、 com.huawei.mdsp.ccengine.common.license.cache.IptvLicenseCache

init方法加同步锁。

10、 com.huawei.mdsp.ccengine.common.license.License

init方法加同步锁。

11、 com.huawei.mdsp.sis.gdr.GdrConfig

init方法加同步锁。

sis\_business工程

12、 com.huawei.mdsp.ccengine.notify.service.notifyflow.ChangeMSISDNNotify

notifyTimesMap、priceInfoMap、serviceIDListMap字段是有状态的，线程不安全。改成方法中的局部变量。

13、 com.huawei.mdsp.ccengine.workorder.WorkOrderField

reloadCDRTagMap方法线程不安全。此方法没有地方调用，先删除。

14、 com.huawei.sis.business.service.impl.SubscribeRelationBusinessImpl

productAttrDict字段是有状态的，初始化方法中增加同步锁。

15、 com.huawei.sis.business.flow.authuser.CheckUserStateStep

errorCodeMap字段是有状态的，可能第一个线程进来还没初始化完，第二个线程就进来了，可能错误码没有正确取到。改成在实例化的时候就把errorCodeMap初始化好。

16、 com.huawei.sis.core.iptvloyalty.service.impl.IptvTimeServiceImpl

areaTimeZone和areaTimeOffset字段是有状态的，再处理它们的方法中增加同步锁。

17、 com.huawei.sis.core.user.builder.ProfileInfoToSubscriberBuilder

subscriber字段是有状态的，改成多例模式。

18、 com.huawei.sis.core.user.builder.ProfileInfoToUserBuilder

user字段是有状态的，改成多例模式。

19、 com.huawei.sis.core.user.builder.SyncUserIdentityInfoBuilder

多例改成单例。

20、 com.huawei.mdsp.ccengine.atomic.product.impl.PromotionDealASImpl

promotionTypeMap字段是有状态的，可能第一个线程进来还没初始化完，第二个线程就进来了，可能没有正确取到。改成在实例化的时候就把promotionTypeMap初始化好。

21、 com.huawei.mdsp.ccengine.interfacecdr.InterfaceCDRUtil

存在多个有状态的字段，相关方法加同步锁。

22、 com.huawei.mdsp.ccengine.service.present.impl.AssignPresentToUsersServiceImpl

存在多个有状态的字段，其中rspResultCode是测试态用的，不会影响正常的业务逻辑。其它改成局部变量使用。

23、 com.huawei.mdsp.ccengine.service.smapmsg.cmanager.custom.impl.SMSNotifyServiceImpl

subscriberID字段是有状态的，改成局部变量。

24、 com.huawei.mdsp.ccengine.service.subscription.util.impl.CommonUtilASImpl

beginTime字段是有状态的，改成用ThreadLocal线程变量的方式去实现。

25、 com.huawei.sis.telkomsel.business.user.step.CheckUserState4FirstAuthStep

errorCodeMap字段是有状态的，可能第一个线程进来还没初始化完，第二个线程就进来了，可能错误码没有正确取到。改成在实例化的时候就把errorCodeMap初始化好。

[【返回首页】](#_top)