

NSP 开发规范标准书 1.0

编写： _____

校对： _____

审核： _____ 年 月 日

标审： _____ 年 月 日

批准： _____ 年 月 日

郑州新开普电子股份有限公司


修改历史

日期	修改内容	作者	审核

目录

1.	开发环境规范	5
2.	JS 规范	6
2.1	JS 编码规范	6
2.1.1	JS 整体结构	6
3.	JAVA 编码规范	8
3.1	全局性规范	8
3.1.1	命名规范	8
3.1.2	Java 类书写样式	8
3.1.3	代码编写格式	11
3.1.4	其他注意事项	13
3.1.5	代码目录	14
3.1.6	配置文件目录	16
3.1.7	服务单元代码包结构	16
3.1.8	异常处理机制	17
3.1.9	服务端验证	20
3.1.10	开发日志	20
3.1.11	单元测试	21
3.2	开发架构分层（DAO、Biz、REST）规范	22
3.2.1	层次划分方式	22
3.2.2	数据访问（DAO）层开发规范	23
3.2.3	业务（BIZ）层开发规范	25
3.2.4	服务（REST）层开发规范	26
4.	数据库设计规范	30
4.1	数据对象（表、视图、索引等）命名规范	30
4.1.1	通用规范：	30
4.1.2	各自规范：	30
4.2.	设计原则	32
4.2.1	表的设计	32
4.2.2	索引的设计	33
4.2.3	视图的设计	34
4.2.4	存储过程、函数、触发器的设计	35
4.3.	SQL 的设计和使用	36
4.3.1	Sql 书写规范：	36
4.3.2	SQL 性能优化建议：	37
5.	参考资料	39
	Java	
	☐Coding Style Guide	39
	ISO639-1 语言代码	39
	ISO 3166-1-alpha-2 国家/地区代码	39

目录

1.	开发环境规范	5
2.	JS 规范	6
2.1	JS 编码规范	6
2.1.1	JS 整体结构	6
3.	JAVA 编码规范	8
3.1	全局性规范	8
3.1.1	命名规范	8
3.1.2	Java 类书写样式	8
3.1.3	代码编写格式	11
3.1.4	其他注意事项	13
3.1.5	代码目录	14
3.1.6	配置文件目录	16
3.1.7	服务单元代码包结构	16
3.1.8	异常处理机制	17
3.1.9	服务端验证	20
3.1.10	开发日志	20
3.1.11	单元测试	21
3.2	开发架构 (DAO、Biz、REST、MODEL) 规范	22
3.2.1	层次划分方式	22
3.2.2	数据访问 (DAO) 层开发规范	23
3.2.3	业务 (BIZ) 层开发规范	25
3.2.4	服务 (REST) 层开发规范	26
3.2.5	业务实体 (MODEL) 开发规范	28
4.	数据库设计规范	30
4.1	数据对象 (表、视图、索引等) 命名规范	30
4.1.1	通用规范:	30
4.1.2	各自规范:	30
4.2.	设计原则	32
4.2.1	表的设计	32
4.2.2	索引的设计	33
4.2.3	视图的设计	34
4.2.4	存储过程、函数、触发器的设计	35
4.3.	SQL 的设计和使用	36
4.3.1	Sql 书写规范:	36
4.3.2	SQL 性能优化建议:	37
5.	参考资料	39
	Java	
	 Coding Style Guide	39
	ISO639-1 语言代码	39
	ISO 3166-1-alpha-2 国家/地区代码	39

1. 开发环境规范

开发工具:eclipse-jee-galileo-SR1-win32

应用服务器：jetty

JDK：jdk6.0

数据库：oracle

代码检查工具：FindBugs

代码格式检查工具：CheckStyle

JS 开发插件：Spket

JS 调试工具：fireBug、IE WebDeveloper

JS 合并和压缩工具：YUI Compressor

属性文件编辑插件：Properties Editor

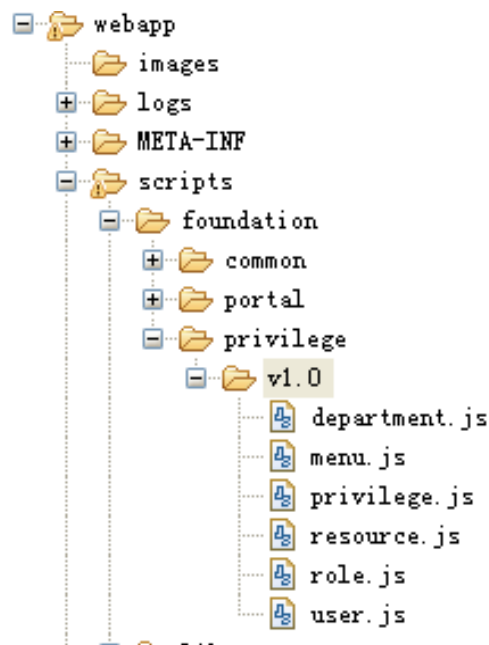
注：开发环境搭建详见开发指南

2. JS 规范

2.1 JS 编码规范

2.1.1 JS 整体结构

1、JS 代码目录结构：



2、自定义命名空间：将所有变量和方法都划分业务到一个全局对象下管理，这样的好处是避免了变量名冲突和由不同函数干扰了全局变量的值。

3、Ajax 调用规范：

代码示例：

```
/**
 * 清理页面缓存 主要这对IE浏览器
 *
 * 原生js方法：
 * 1、在服务端加 header("Cache-Control: no-cache, must-revalidate"); (如php中)
 * 2、在ajax发送请求前加上 anyAjaxObj.setRequestHeader("If-Modified-Since", "0");
 * 3、在ajax发送请求前加上 anyAjaxObj.setRequestHeader("Cache-Control", "no-cache");
 * 4、在 Ajax 的 URL 参数后加上 "?fresh=" + Math.random(); //当然这里参数 fresh 可以任意取了
 * 5、第五种方法和第四种类似，在 URL 参数后加上 "?timestamp=" + new Date().getTime();
 * 6、用POST替代GET：不推荐
 *
```

```
*
*
*
* **/
jQuery.ajaxSetup ({cache:false});

// 添加角色UI
function addRoleUI() {
    $.post('/restful/privilegeProxyService/role/addRoleUI', null,
        function(data) {
            $.dialog( {
                id : 'addroleid',
                title : '添加角色',
                content : data,
                lock : true
            });
        });
}
```

名字尽量和 rest 名字一致

3. JAVA 编码规范

3.1 全局性规范

3.1.1 命名规范

遵守 SUN Java ☐ Coding Style Guide

3.1.2 Java 类书写样式

3.1.2.1 类的书写顺序

类的结构规定按以下顺序排列：

1.final 常量，2.静态变量，3.实例变量，4.方法（业务方法、属性方法）。

注：访问控制符由 **public** 到 **private** 的顺序进行排列。

3.1.2.2 Package/Import 书写

import 中标准的包名要在本地的包名之前，而且按照字母顺序排列。

Java 标准包放在最前，其它包放在后面。不同意义的 import 进行分段。

3.1.2.3 Class 的注释

接下来的是类的注释，一般是用来解释类的。

```
/**
 * This class is the conntroller in the MVC structure. 类功能描述
 * @author name
 * @date ${date} ${time}
 * 类功能变更列表
```



```
*/
```

3.1.2.1 Class Fields 的注释

接下来是类的成员变量：

```
/**  
 * 属性描述  
 */  
public int[] packets_;
```

public 的成员变量可以加上文档 (JavaDoc) 。

3.1.2.2 类方法的注释

对类方法注释要紧靠在每个方法的上面，如下

```
/**  
 * 类方法注释  
 */
```

方法的用途，这是类注释必须有的。

下面是 javaDoc 必须的注释：

@param 该标记在当前方法的 “parameter” 一节中增添一个条目，具体的描述可以跨越多行，可使用 HTML 标记，要求所有的 @param 要连在一起。

@return 该标记在当前方法的一个 “Returns” 小节。具体的描述可以跨越多行，可使用 HTML 标记。

@throws 该标记在当前方法的 “Throws” 一节中增添一个条目，JAVADOC 会自动建立一个超链接让用户理解所抛出的异常，具体的描述可以跨越多行，可使用 HTML 标记，要求所有的 @throws 要连在一起。

@see 该标记在 “See also” 小节中增加一个超链接使其可以转到 JAVADOC 文档的其他部分或转到一个外部文档。具体后面可以采用以下几种模式之一即

可：

- (1) package.class#method XXXXXX //此形式最常用，省略包名表示在同一包
- (2) XXXXX
- (3) “文字”

```
/**
 * <b> Set the packet counters</b> (such as when RESToring from a
 database)
 *
 * @param r1
 *         input vector store message 1
 * @param r2
 *         input vector store message 2
 * @param r3
 *         input vector store message 3
 * @param r4
 *         input vector store message 4
 * @return null
 * @throws IllegalArgumentException if parameter1 is error throw IllegalArgumentException
 * @see cn.ceopen.wes.sfa.admin.AdminMenuPage#print()
 */
protected final void setArray(int[] r1, int[] r2, int[] r3, int[] r4)
    throws IllegalArgumentException {
    // Ensure the arrays are of equal size
    if (r1.length != r2.length || r1.length != r3.length
        || r1.length != r4.length)
        throw new IllegalArgumentException(
            "Arrays must be of the same size");
    System.arraycopy(r1, 0, r3, 0, r1.length);
    System.arraycopy(r2, 0, r4, 0, r1.length);
}
```

注：推荐做法：写代码前写注释，这样给你一个机会在写代码前去想清楚代码怎样书写，并能确保代码注释的书写。

3.1.2.3 （推荐）方法内注释

对涉及到流程分支较多或算法比较复杂的程序段，建议在此之前对流程或算法进行适当的解释。

对某些关键性的程序行，建议在其前一行或行尾加以注释。

我们书写代码最好的结果是用代码来说话，也就是如果代码写的好，并不需要太多的注释。但是有这样的方法，它们非常复杂不是每个程序员都有能力来看懂、来重写，这样就必须书写详细的注释。

```
/*Check if cookieDepart is available;
 *If the administration login is needed, we should set the
 adminstration group code;
 */
boolean bForceAdmin = false;
if(session != null) {
    Boolean Bool = (Boolean)session.getValue("force_admin");
    if(Bool != null )
        bForceAdmin = Bool.booleanValue();
}
```

3.1.2.4 文档化

必须用 javaDoc 来为类生成文档。不仅因为它是标准，这也是被各种 java 编译器都认可的方法。

3.1.3 代码编写格式

3.1.3.1 缩进

缩进应该是每行 4 个空格。

3.1.3.2 页宽

页宽应该设置为 80 字符。源代码一般不超过这个宽度，超长的语句应该在一个逗号或者一个操作符后折行。一条语句折行后，应该比原来的语句再缩进 4 个字符。

3.1.3.3 {} 对

要求{}对排列整齐，最好上下一致，原则上要求便于阅读和配对，简单的可写成一行，例：`if (i < 0) { i ++ }`

一般情况可用以下格式

```
if (i < 0) {    //{与条件语句在同一行
    i ++
}
```

3.1.3.4 空白行

空白行应该在以下情况中出现：

- ◆ 类之间
- ◆ 类方法之间
- ◆ 最后一个类变量声明和第一个类方法之间
- ◆ 在第一行注释之前
- ◆ 语句中完成相似的功能模块之后

一般情况下，应该只空一行。

3.1.3.5 括号

左括号和后一个字符之间不应该出现空格，同样，右括号和前一个字符之间也不应该出现空格。下面的例子说明括号和空格的错误及正确使用：

`CallProc(AParameter);` //错误

`CallProc(AParameter);` //正确

不要在语句中使用无意义的括号。括号只应该为达到某种目的而出现在源代

码中。下面的例子说明错误和正确的用法：

```
if ((I) = 42) {                // 错误 - 括号毫无意义
```

```
if (I == 42) or (J == 42) then // 正确 - 的确需要括号
```

3.1.4 其他注意事项

- io 流应该手动关闭，

举例：fileInputStream.close();

- 近可能使类的成员属性和方法的可访问能力最小化；
- 使用 StringBuffer，

对象在处理 String 的时候要尽量使用 StringBuffer 类，StringBuffer 类是构成 String 类的基础。当我们在构造字符串的时候，我们应该用 StringBuffer 来实现大部分的工作，当工作完成后将 StringBuffer 对象再转换为需要的 String 对象。比如：如果有一个字符串必须不断地在其后添加许多字符来完成构造，那么我们应该使用 StringBuffer 对象的 append() 方法。如果我们用 String 对象代替 StringBuffer 对象的话，会花费许多不必要的创建和释放对象的 CPU 时间。

- synchronized 关键字，适用场景用于保证单 JVM 环境中线程同步；
- 除了定义常量值外，不使用 static 定义变量；
- if、for 等操作的注意事项，

if, for 语句的逻辑不要太深。这样可读性会非常低，因此请尽量控制在 3~4 重嵌套左右。 还需要更深时，另外作为一个 private method 分开处理。 在 loop 中不要变更 loop 计数值。

- 开发结束提交代码时，一定要将未使用了的方法和变量删除
- 在 int 型中达到 10 位以上时则无法处理。使用 long 型。
- session 管理，

1.禁止将 Session 从 REST 向下层传递

2.只能使用框架提供 Session 读写属性的 API

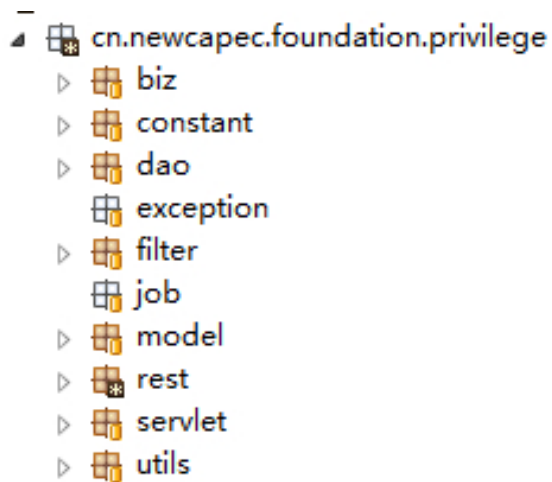
内容规范：

Session 内容：目前可放置登陆账号信息标识、角色标识的信息。

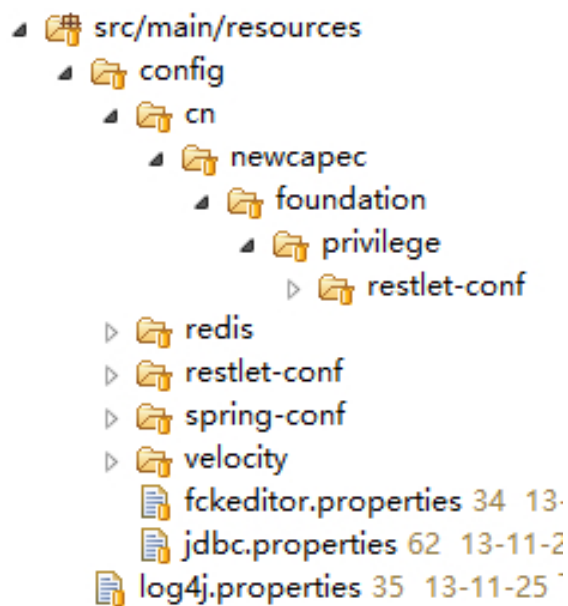
3.1.5 代码目录

开发目录结构：

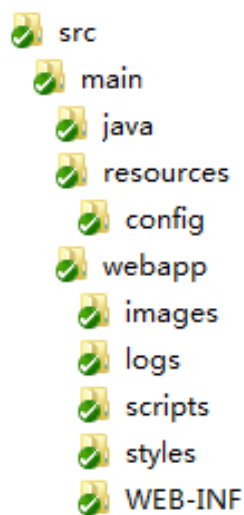
产品开发下开发目录结构：



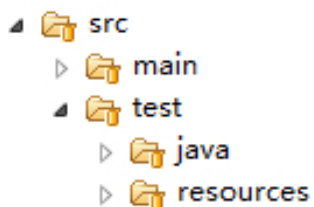
Resource 目录为配置文件目录：



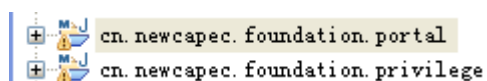
Main 为代码目录。



Test 为测试用例目录。



模块层



在以上目录结构中，每个应用程序包包含两个源代码树和一个资源包，两个源代码树分别是 Java 应用程序的源代码和单元测试源代码，资源包用来管理本模块的配置文件。

3.1.6 配置文件目录

配置文件的存放必须遵循下图所示的目录结构：

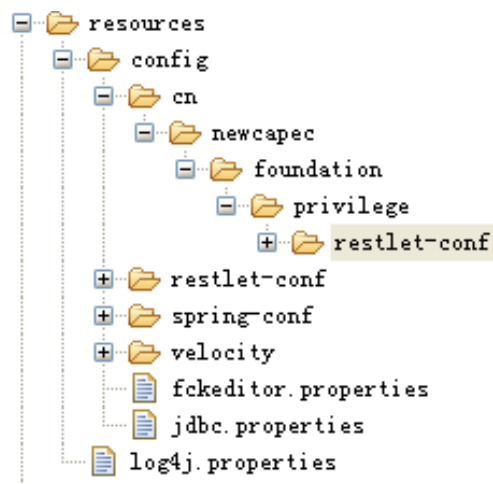
全局配置目录：

restlet-conf 存放 RESTlet 的总配置文件

spring-conf 存放 spring 的配置文件

cn.newcapec.foundation.privilege.restlet-conf 模块的配置文件

velocity 存放 velocity 配置文件



3.1.7 服务单元代码包结构

biz 中存放业务层逻辑代码

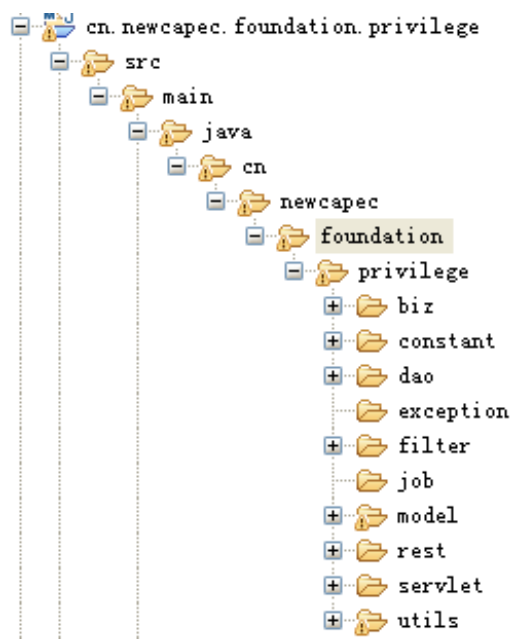
dao 中存放数据访问层代码

model 中存放数据实体模型代码

filter 中存放过滤器代码

job 中存放定时任务代码

rest 中存放 restlet 代码



3.1.8 异常处理机制

3.1.8.1 异常体系说明

- 1、异常体系的实现基于 RuntimeException 建立，这样可以不污染系统中的接口定义，并简化开发人员的代码。
- 2、异常类型定义简单、明确，对开发人员有指导意义。
- 3、系统为异常处理提供统一的执行点。

通过表示层(RESTlet)的过滤器统一捕获 BaseException,然后进行处理,最后返回到前台(JS)展现,即通过重写 JS 中 Ajax 请求失败的方法,对表示层返

回的异常信息进行统一展现。这样程序员只需要负责请求成功的处理，请求失败的处理都由异常体系来完成。

3.1.8.2 异常处理规范

- 1、开发人员如果没有特殊需要就直接使用以上定义好的异常基类来包装异常，如果由于业务需求需要创建自定义的异常类则必须继承以上的异常基类。
- 2、对于保存和修改过程中底层抛出的异常开发人员不需要做任何处理,由异常体系系统直接捕获、处理。
- 3、异常信息不能直接用 `e.printStackTrace()`或 `System.out.println()`输出，而是需要记录到日志里。例如：

```
try {
    String url = "/foundation/privilege/role/pagelet/v1.0/role_list.html"
    getNewcapectViewContext().put("menuName", "角色管理");
    roleListGrid(request, response);

    response.toHtml(url, getNewcapectViewContext());
} catch (Exception e) {
    log.error(ExceptionUtils.getFullStackTrace(e));
    if (e instanceof BaseException) {
        throw (BaseException) e;
    } else {
        throw new BaseException("系统出错！", e);
    }
}
```

- 4、程序员只需要处理请求成功的情况，对于请求失败的情况统一由异常体系来处理，程序员只需要在相应的位置抛出异常即可。
- 5、对于字段的验证由于性能问题不允许采用异常机制。
- 6、保护封装性

不要让你要抛出的 checked exception 升级到较高的层次。例如，不要让 `SQLException` 延伸到业务层。业务层并不需要（不关心？）`SQLException`。

你通过以下方法来解决这种问题：

转变 SQLException 为一个 unchecked exception ,

看看下边的代码:

```
public VOid dataAccessCode(){
    try{
        ..some code that throws SQLException
    }catch(SQLException ex){
        throw new BaseException (ex);
    }
}
```

上边的做法是把 SQLException 转换为 RuntimeException , 一旦 SQLException 被抛出 ,那么程序将抛出 RuntimeException,此时程序被挂起并返回客户端异常信息。

7、 不要使用异常来控制流程

8、 不要忽略异常

当有异常被抛出的时候 , 如果你不想恢复它 , 那么你要将其转换为 unchecked exception , 而不是用一个空的 catch 块或者什么也不做来忽略它 , 以至于从表面来看象是什么也没有发生一样,有 try 必须有 catch。

9、 不要捕获顶层的 Exception

unchecked exception 都是 RuntimeException 的子类 , RuntimeException 又继承 Exception,因此 , 如果单纯的捕获 Exception,那么你同样也捕获了 RuntimeException,如下代码 :

```
try{
    ..
}catch(Exception ex){
}
```

3.1.8.3 异常编码

系统异常唯一异常类为：

cn.newcapec.framework.core.exception.BaseException

使用其构造方法如下：

A: public BaseException(String code,String msg)

B: public BaseException(String code,String msg, Throwable ex)

3.1.9 服务端验证

在 REST 层提供服务的处理方法中进行请求参数的合法性功能。

3.1.10 开发日志

所有开发日志必须使用此日志接口输出。

日志类： cn.newcapec.framework.core.logs.LogEnabled

```
/**
 * 日志接口
 */
public interface LogEnabled {
    /**
     * 存储到数据库
     */
    public NspLogger nspLog=NewcapecLoggerFactory.getNspLogger(LogEnabled.class);
    /**
     * 日志打印
     */
    public Logger log = Logger.getRootLogger();
}
```

Resource、Biz、Dao 三层的框架基类已经都实现该接口，所以开发人员集成这些基类实现自己的业务逻辑类时，可以直接调用 nspLog 类，进行日志输出。

3.1.11 单元测试

3.1.11.1 代码位置

单元测试用例代码放置于对应服务单元的 test 目录下，包名与被测试类的包名保持一致。

3.1.11.2 类命名

单元测试用例类名：被测试实现类名+Tests。

3.1.11.3 测试方法命名

方法名：test+被测试方法名。当被测试方法含有复杂关键分支，为此分支单独编写的测试方法，方法名：test+被测试方法名+分支大概含义描述。测试方法必须添加标注@Test。

3.1.11.4 配置

单元测试用例的配置，使用所在服务单元的配置（包括全局配置和所在服务单元依赖的 jar）。

3.1.11.5 测试类初始化

整个测试类的所有方法执行前需要执行的一些操作（例如初始化 spring 上下文），写在测试类初始化方法中，必须修饰为 public static，必须添加标注@BeforeClass。

3.1.11.6 测试类清理

整个测试类的所有方法执行后需要执行的一些操作(例如全面清理数据库), 写在测试类清理方法中, 必须修饰为 `public static`, 必须添加标注 `@AfterClass`。

3.1.11.7 测试方法初始化

每个测试方法执行前需要执行的一些操作(例如准备 mock 数据), 写在测试方法初始化方法中, 必须修饰为 `public`, 必须添加标注 `@Before`。

3.1.11.8 测试方法清理

每个测试方法执行后需要执行的一些操作(例如清理 mock 数据), 写在测试方法清理方法中, 必须修饰为 `public`, 必须添加标注 `@After`。

3.1.11.9 初始化 spring 上下文

参考开发指南“单元测试用例”一节中的“测试类初始化方法”

3.1.11.10 单元测试数据

单元测试要保证测试数据隔离的原则, 单元测试用例要在独立的数据环境中。

3.2 开发架构(DAO、Biz、REST、MODEL) 规范

3.2.1 层次划分方式

1. 数据访问层(DAO): 提供对数据访问的接口和实现类。

2 . 业务层(Biz) : 处理业务逻辑的接口和实现类。

3 . 服务层(REST) : 为客户端提供服务的资源类实现。

3.2.2 数据访问 (DAO) 层开发规范

2.2.2.1 代码位置

dao 包中放置各数据访问对象(data access object) , 也就是对数据访问的实现定义。

model 包中放置数据模型 DO (DataObject) 对象。

2.2.2.2 类命名

类的命名 :

基础的命名格式 : 数据实体名称 + ' Dao' , 例如 : BasePersonDao

实现类的命名格式 : 数据实体名称 + ' Dao' , 例如 : PersonDao

方法的命名 :

1. 格式 : 功能名 + 数据实体名称

2. 功能名定义 :

c/r/u/d 的完整单词 + 操作名称 + 数据实体名称(c:创建,r:读取,u:更新,d:删除)

注 : CRUD 是指在做计算处理时的

增加(save)、查询(find[返回单体]、query[返回集合])(重新得到数据)、

更新(Update)和删除>Delete)

2.2.2.3 继承关系

DO 类必须继承 AppBaseModel , 并实现 DataObject 接口

DAO 基础必须继承

cn.newcapec.framework.core.dao.hibernate.HibernateEntityDao

2.2.2.7 配置

Hibernate 的懒加载机制

查询时, 不允许使用 Hibernate 的懒加载机制, 对于关联对象在业务层做关联查询。

在 Hibernate 配置文件中不允许使用任何关联关系配置。

2.2.2.8 HQL 或 SQL 语句的编写规则

关键字必须大写, 其他书写按 JAVA 命名规则, 比如:

```
SELECT  user_id,  user_name  FROM  User  WHERE  user_id  =  
'tom' 。
```

由于在开发阶段数据库字段名称和持久化对象的属性名称不段在变化, 为了避免当字段或属性名称发生变化时修改程序中的 SQL 语句或 HQL 语句 , 现提出以下规范:

在拼接 SQL 语句或 HQL 语句时, 字段名称不能直接写成常量字符串, 而是在持久化对象 (即 DO) 中将属性名称和数据库字段名称定义为常量, 然后使用这些常量去拼接 SQL 或 HQL 语句, 这样当名称发生变化时, 只需要修改持久化对象中的常量即可。

3.2.3 业务（BIZ）层开发规范

3.2.3.1 业务实体设计原则

1.面向一个业务服务有且只有一个主业务实体

2.对业务实体操作分为（增/查/改/删）四类，对业务实体的（增/查/改/删）

是分别 是由对数据模型的（C/R/U/D）组合而成的。

3.2.3.2 代码位置

biz 包中放置业务处理对象（business）接口定义

biz.impl 包中放置业务处理接口实现

3.2.3.3 类命名

Biz 层中服务名称与 REST 层的服务名称相互一致对应

业务逻辑层 Biz 接口：服务名 + Service

业务逻辑层 Biz 接口的实现类：服务名+Service+Impl

3.2.3.4 继承关系

Service 类实现 cn.newcapec.framework.core.biz.BaseService 接口

3.2.3.5 方法命名

Biz 的对应功能点的主业务方法命名

例如转账实体的创建操作为主业务方法

增加(save)、

查询(query)(复数结果操作)

查询(find)(单数结果操作)为：用过 ID 查询返回单个实体。

更新(update)

删除(delete)

3.2.3.6 事务方法前缀

增加(save)、读写事务

查询(query)(复数结果操作) 只读事务

查询(find)(单数结果操作)为：用过 ID 查询返回单个实体。只读事务

更新(update) 读写事务

删除(delete) 读写事务

批量操作为：

批量增加(batchAdd)、读写事务

批量更新(batchModify) 读写事务

批量删除(batchRemove) 读写事务

3.2.4 服务（REST）层开发规范

3.2.4.1 设计原则

Biz 层输入参数

增加操作：业务实体参数必须具备完整的属性信息。（除非必填外）

更新操作：业务实体参数必须具备完整的属性信息。（除非必填外）

删除操作：业务实体参数包含 UUID 属性信息。

3.2.4.2 代码位置

rest 包中放置业务层以 REST 方式为客户端暴露服务

3.2.4.3 URI 命名

<http://域名:端口/restful/服务名称/操作名称>

3.2.4.4 类命名

与之 URI 服务部分对应功能点实现类命名：服务名+Resource

与功能点对应的方法命名规则：

功能 URI 中的操作名称即为方法名称

入口方法的参数类型

3.2.4.5 继承关系

REST 层服务类必须继承

`cn.newcapec.framework.core.rest.BaseResource`

3.2.4.6 方法命名

与之 URI 操作部分对应功与功能点方法命名：

功能操作名称即为方法名称

方法的访问控制符必须是 public。

示例：例如：新增订单 URI：

`order/create` 类名称：`OrderResource`

方法名称：add

3.2.4.7 HTTP 请求

目前基础框架支持 get、post 两种提交方式。

原则上规定只能使用 post 方式提交。

REST 服务层，对外部数据交互格式为 JSON 格式。

客户端与 REST 服务层数据交互格式为 JSON 格式。

3.2.4.8 配置

配置文件目录为参照全局规范目录章节

REST 配置文件命名为服务名称+Context.xml

服务的操作方法（即 REST 二级路由）要配置到此文件中

REST Bean 要配置在此文件中

REST Bean 名称与服务名称相同

REST Bean 配置为多例形式 注：Spring 多例配置 scope="prototype"

3.2.5 业务实体（MODEL）开发规范

3.2.5.1 设计原则

针对系统中的业务实体进行建模，这样 model 描述了包含了一个业务实体的所有属性，以及针对这些属性的数据操作。

3.2.5.2 代码位置

存在放在 model 目录中，为整个业务实现服务。

3.2.5.3 类命名

MODEL 类以业务实体的名称进行命名。

3.2.5.4 继承关系

存在两个基类，MODEL 类可以根据需要，选择其中的一个：

CommonModel：常规继承的基类。

AppBaseModel：当需要数据版本控制时，继承这个基类。

3.2.5.5 方法命名

动作名称 (set/get...) + 实体名 (User)

首字母消息，遵循驼峰方式。

3.2.5.6 注解说明

类注解：

```
@Entity
@Table(name = "表名")
```

get 方法注解：

```
@Column(name = "字段名", 其它字段约束)
```

对于 Java 类型与数据库类型的部分映射需要特殊注解说明，例如对于数据库 Timestamp 类型的读、写，需要在 get 方法中加上如下注解：

```
@Temporal(TemporalType.TIMESTAMP)
```

请针对规范，调整相应的注解。

4. 数据库设计规范

4.1 数据对象（表、视图、索引等）命名规范

4.1.1 通用规范：

1. 使用英文：要用简单明了的英文单词，不要用拼音，特别是拼音缩写。

主要目的很明确，让人容易明白这个对象是做什么用的；

2. 一律大写，特别是表名：有些数据库，表的命名乃至其他数据对象的命名是大小写敏感的，为了避免不必要的麻烦，并且尊重通常的习惯，一律使用大写命名；

4.1.2 各自规范：

1. 表的命名

表名的前缀：为表的名称增加一个或者多个前缀，前缀名不要太长，可以用缩写，最好用下划线与后面的单词分开；其目的有这样几个：

- 为了不与其他项目或者其他系统、子系统的表重名；
- 表示某种从属关系，比如表明是属于某个子系统、某个模块或者某个项目等等。表示这种从属关系的一个主要目的是，从表名能够大概知道如何去找相关的人员。比如以子系统为前缀的，当看到这个表的时候，就知道有问题可以去找该子系统的开发和使用人员；

2. 视图命名：V_相关表名（或者根据需要另取名字）

3. 存储过程命名：PRO_存储过程名（用英文表达存储过程意义）

4. 函数命名：FUN_函数名称（用英文表达函数作用）

5. 触发器命名：TRI_触发器名称（用英文表达触发器作用）

6. 索引命名：I_表名_字段名(如果存在多字段索引，取每字段前三个字符加下划线组合，如：在 HYID,HYNAME,HYMOBILE 上建立联合索引，命名：I_表名_HYI_HYN_HYM,如果前三个截取字符相同，就从字段名称中不同的字符开始取三个字符加下划线组合，如：在 ZHYID,ZHYNAME,ZHYMOBILE 上建立联合索引，命名：I_表_ID_NAM_MOB)

7. 唯一索引命名：UI_表名_字段名(如果存在多字段唯一索引，取每字段前三个字符加下划线组合，如：在 HYID,HYNAME,HYMOBILE 上建立唯一索引，命名：UI_表名_HYI_HYN_HYM,如果前三个截取字符相同，就从字段名称中不同的字符开始取三个字符加下划线组合，如：在 ZHYID,ZHYNAME,ZHYMOBILE 上建立唯一索引 命名 :UI_表_ID_NAM_MOB)

8. 主主键命名：PK_表名_字段名(如果存在多字段主键，取每字段前三个字符加下划线组合，如：在 HYID,HYNAME,HYMOBILE 上建立主键，命名：PK_表名_HYI_HYN_HYM,如果前三个截取字符相同，就从字段名称中不同的字符开始取三个字符加下划线组合，如：在 ZHYID,ZHYNAME,ZHYMOBILE 上建立主键，命名：PK_表_ID_NAM_MOB)

9. 外键命名：FK_表名_主表名_字段名

10. Sequence 命名：SEQ_表名_列名（或者根据需要另取名字）

4.2. 设计原则

4.2.1 表的设计

4.2.1.1 主、外键

1、每个表，都必须要有主键。主键是每行数据的唯一标识，保证主键不可随意更新修改，在不知道是否需要主键的时候，请加上主键，它会为你的程序以及将来查找数据中的错误等等，提供一定的帮助；

2、禁止使用表外键约束

4.2.1.2 列的设计

1. 字段的宽度要在一定时间内足够用，但也不要过宽，占用过多的存储空间；

2. 字段的类型及宽度在设计以及后面进行开发时，往往要与应用的设计、开发人员商讨，以得到双方认可的类型及宽度；

3. 除非必要，否则尽量不加冗余列。所谓冗余列，是指能通过其他列计算出来的列，或者是与某列表达同一含义的列，或者是从其他表复制过来的列等等。冗余列需要应用程序来维护一致性，相关列的值改变的时候，冗余列也需要随之修改，而这一规则未必所有人都知道，就有可能因此发生不一致的情况。如果是应用的特殊需要，或者是为了优化某些逻辑很复杂的查询等操作，可以加冗余列；

4.添加 version 字段，长度 `VERSION` int(11) default NULL`

5.行长约束，长度范围待确定

4.2.2 索引的设计

索引是从数据库中获取数据的最高效方式之一。95%的数据库性能问题都可以采用索引技术得到解决。但大量的 DML 操作会增加系统对索引的维护成本，对性能会有一些影响，对于插入相当频繁的表要慎重建索引，索引也会占相当的存储空间，所以要根据硬件环境和应用需求在空间和时间上达到最好的平衡点，主要原则：

1. 适当利用索引提高查询速度：当数据量比较大，了解应用程序的会有哪些查询，依据这些查询需求建相应的索引；最好亲自试验一下，模拟一下生产环境的数据量，在此数据量下，比较一下建索引前后的查询速度；索引对性能会有一些影响，对于 DML 频繁列的索引要定期维护（重建）。但是，索引的结构对于索引的更新（比如在插入数据的时候）是有一定优化的，所以不要在没有试验以前过分夸大它对性能的影响。最终还是以试验为准；

2. 不要建实际用不上的索引，与上条相关，如果建的索引并不提高任何一应用中的查询速度，则要把它删除；有些数据库有相关工具可以发现实际未被使用的索引，可以利用一下；

3. 索引类型的选择：要根据数据分布及应用来决定如何建立索引，一般的高基数数据列（高基数数据列是指该列有很多不同的值）时，建立 BTree 索引（一般数据库索引的缺省类型）；当低基数数据列（该列有大量相同的值）时，可以考虑建立位图索引（如果所选数据库支持的话），但位图索引是压缩类型索引，所以 DML（增、删、改）的代价更高，要综合考虑；

4. 索引列的选择：如果检索条件有可能包含多列，创建联合主键或者联合索引，把最常用于检索条件的列放在最前端，其他的列排在后面；不要索引使用

频繁的小型表，假如这些小表有频繁的 DML 就更不要建立索引，维护索引的代价远远高于扫描表的代价；

5. 主键索引在建立的时候一定要明确的指定名称，不能让系统默认建立主键索引（可能有些数据库无法指定主键名，则例外）；

6. 外键必须需建索引。当有一定数据量，并且经常以外键所在列为关联，进行关联查询时，需要建索引（可能有些数据库自动为外键建索引，则例外）；

7. 当有联合主键或者联合索引时，注意不要建重复的索引。举例说明：

1) 表 EMPLOYEES，它的主键是建立在列 DEPARTID 和 EMPLOYEEID 上的联合主键，并且创建主键的语句中 DEPARTID 在前，EMPLOYEEID 在后。在这样一个表里，通常就没有必要再为 DEPARTID 建一个索引了；联合索引的情况也一样；

2) 更复杂的情况，比如表 EMPLOYEES，有一个索引建立在列 CORPID,DEPARTID,EMPLOYEEID 三列上，在创建语句中也依据上述顺序。那么，就没有必要再为 CORPID 建立索引；也没有必要再建立以 CORPID 在前，DEPARTID 在后的联合索引；如果 EMPLOYEEID 需要索引，那么为 EMPLOYEEID 建立一个索引是不与上面的索引重复的；DEPARTID 列也类似；

8.对于不能重名的字段必须建立唯一性约束。

4.2.3 视图的设计

1. 在不太清楚视图用法的情况下，尽量不建。因为一旦建了，就有被滥用的危险；

2. 如果需要建视图，只要是打算长期使用的，请写入数据库设计中。明确

它的用途、目的；

3. 建立视图时要明确写出所有要选择出的列名而不要以 `SELECT *` 来代替，可以使结构清晰可读性增强，也不会增加它对表的所有字段的依赖，而表是很可能修改的，特别是增加字段。就很有可能导致使用该视图的应用程序出错；

4.2.4 存储过程、函数、触发器的设计

1. 触发器的功能通常可以用其他方式实现。在调试程序时触发器可能成为干扰。假如你确实需要采用触发器，一定要经过测试再应用在生产系统中，而且必须集中对它文档化。

2. 请把存储过程、函数、触发器，与应用程序一同加入 CVS 中，进行版本控制。因为，此三者包含了代码，应用程序对他们的依赖程度，比对表、视图的依赖程度更高；

3. 适量使用存储过程、函数、触发器。使用存储过程、函数、触发器的影响：

- 1) 可以减少数据库与客户端的交互，提高性能；
- 2) 有的数据库还对他们进行了某种程度的编译，在执行的时候，不用再对其中的 SQL 等语句进行解析，从而提高速度；
- 3) 如果有多个应用，使用了不同的开发语言，当有某些关键的或者复杂逻辑希望共享，则可以考虑使用存储过程或者函数。因为存储过程等在数据库一级是共享的；
- 4) 增强了应用对数据库的依赖，如果打算将来移植数据库的话，使用得越多，则移植的困难越大；数据库中的业务逻辑越多（存储过程等），应用以及存

储过程等的维护难度也会增大；

5) 通常存储过程等没有面向对象的特性，不容易设计出易于扩展的结构。

当存储过程比较复杂时，或者它们相互间的调用关系比较复杂时，可能难于维护；

4.3. SQL 的设计和使用的

4.3.1 Sql 书写规范：

1. 尽量不要写复杂的 SQL 过于复杂的 SQL 可以用存储过程或函数来代替，效率更高；甚至如果能保证不造成瓶颈的话，把条 SQL 拆成多条也是可以的。这与一般的编码规范很相似的，首先是要易懂。易懂也就意味着容易维护，对较为复杂的 sql 语句加上注释，说明算法、功能注释风格：注释单独成行、放在语句前面。

2. 应对不易理解的分支条件表达式加注释；

3. 对重要的计算应说明其功能；

4. 过长的函数实现，应将其语句按实现的功能分段加以概括性说明；

5. 每条复杂 SQL 语句均应有注释说明(表名、字段名 主要是说明此句 SQL 执行 的作用及所取得结果集的意义)；

6. 常量及变量注释时，应注释被保存值的含义(必须)，合法取值的范围(可选) ；

7. 可采用单行/多行注释。(-- 或 /* */ 方式，不同数据库可能语法不同)；

8. 连接符 or、in、and、以及 =、<=、>=等前后加上一个空格；

9. 不要用 SELECT *：SELECT 语句中写出必要的要选择的全部列名，增强语句可读性，避免不必要的选择；SELECT *增加了对所有字段的依赖，当表增

加了字段后，有可能发生错误；此外还可能增加了数据的流量，查询了一些实际不需要的字段；

10. 尽量避免过长的事务 (Transaction)：长的事务容易造成死锁，尽量避免；

11. 行最长不能超过 80 字符,同一语句不同字句之间逗号以后空格,其他分割符前空格 where 子句书写时，每个条件占一行，语句令起一行时，以保留字或者连接符开始，连接符右对齐；

12. 多表连接时，使用表的别名来引用列；

13 SQL 中对视图的引用：在不太清楚视图用法的情况下，尽量不用。只是因为视图中有自己想要的字段就拿来用，是相当普遍和错误的用法。原因如下：

- 增加了不必要的数据流量，对你的实际需求，那很可能是一个非常复杂的视图，有大量你不需要的字段，并且关联了很多你实际不需要的表，对数据库资源会有过多的消耗；
- 增加了应用程序对视图的依赖，我们知道，不必要的依赖是越少越好的。当有应用程序依赖了某个视图，不久可能其他人因为某种原因会修改此视图，原来的应用有可能会受到不同程度的影响；

4.3.2 SQL 性能优化建议：

1. 系统可能选择基于规则的优化器，所以将结果集返回数据量小的表作为驱动表，即将结果集返回数据量小的表放在 FROM 后边最后一个表；

2. 大量的排序操作影响系统性能，所以尽量减少 order by 和 group by 排序操作；

3. 如必须使用排序操作，排序尽量建立在有索引的列上；

4. 索引的使用

- 尽量避免对索引列进行计算。如对索引列计算较多，请提请系统管理员建立函数索引；
- 尽量注意比较值与索引列数据类型的一致性(number 与 number 比较、char 与 char 比较)，避免使用数据库的类型自动转换功能；

```
SELECT * FROM category
```

```
WHERE id = '123' ; -- id' s type is number
```

- 对于复合索引，SQL 语句必须使用主索引列；
- 索引字段中，尽量避免使用 NULL 值；
- 对于索引的比较，尽量避免使用 NOT= (!=)
- 查询列和排序列与索引列次序保持一致 ；

5. 尽量避免相同语句由于书写格式的不同，而导致多次语法分析(减少数据库的硬分析)

6. 查询的 WHERE 过滤原则，应使过滤记录数最多的条件放在最前面；

7. 在 WHERE 中，数据库函数、计算表达式等等，要尽可能将放在等号右边。否则会使所比较的字段上的索引失效；

8. in、or 子句常会使索引失效，尽可能不使用 in、or ；

9. 尽量避免在循环中使用 SQL 语句；

5. 参考资料

Java ☐ Coding Style Guide

http://developers.sun.com/solaris/reference/docs/sunstudio_whitepapers/java-style.pdf
<http://java.sun.com/docs/codeconv/>

ISO639-1 语言代码

标准参考网址: http://www.loc.gov/standards/iso639-2/php/English_list.php;

ISO 3166-1-alpha-2 国家/地区代码

标准参考网址

http://www.iso.org/iso/country_codes/iso_3166_code_lists/english_country_names_and_code_elements.htm