

　　　　　　　研发中心编码规范



|  |  |  |  |
| --- | --- | --- | --- |
| **文档类型：** | 项目文档 | **文档标识：** |  |
| **文档状态：** | [√] 草稿  [ ] 正式发布  [ ] 正在修改 | **模板标识：** |  |
| **当前版本：** |  |
| **作者：** | 安世亚太科技股份有限公司 |
| **完成日期：** | 2012-12-22 |

# 开发编码规范

为了便于项目的维护,因此制定开发规范,请开发人员务必保持一致的开发规范.

## 开发环境

开发中使用以下环境开发:

Eclipse 3.2以上 + JDK 1.6.31 + tomcat 6.0.35 + SVN + jquery1.7.2

+ SiteMesh

## 命名规范

### 方法命名

数据库操作方法命名方式:

查询:以 “query”开头; 如: queryXXX()

插入:以 “add”开头; 如: addXXX()

删除:以 “delete”开头; 如: deleteXXX()

更新:以 “update”开头; 如: updateXXX()

分页查询: 以“queryPage”开头； 如： queryPageXXX();

### 页面名称命名

页面的名称应全部小写,多单词之间用下划线“\_”分隔.

并且已功能模块名称开头.

如:模块: property

对应页面: property\_add.jsp, property\_delete.jsp

### 跳转名称定义

跳转名称应定义为当前类的全局常量,常量名应全部大写, 多个单词之间以下划线‘\_’分隔。

跳转名称以 “模块名/页面名称” 的格式定义。

如: public final static String EXAMPLE\_DEMOS = “example/demos”;

跳转到demos.jsp

### 配置文件命名

配置文件名称以 功能模块来划分为多个文件 .

如 spring 配置文件: spring\_project.xml, spring\_mass.xml;

Mass 和 project 为模块名称.

### 实体类命名

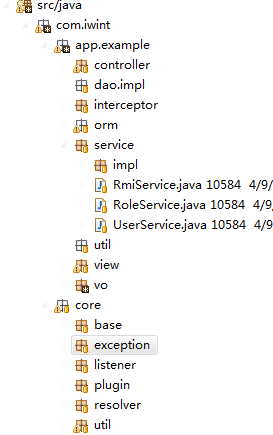
实体类名称为数据库表名去下划线后,首写字母大写.

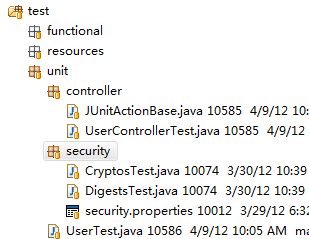
如: DMS\_CUS\_INFO-> DmsCusInfo

DMS\_OBJ -> DmsObj

## 目录结构

### 类的目录结构





说明:

Controller : spring MVC的控制器类包;

Interceptor: 拦截器类包;

Service : 业务逻辑接口包;

Service.impl : 业务逻辑实现类包;

Util : 公用类;

Orm.entiry : 放实体类包;

Orm.mapper: myibatis的映射文件和类包;

Vo: VO类包

Core.exception: 异常类包;

Core.listener : 监听器类包;

Core.plugin : 插件类包;

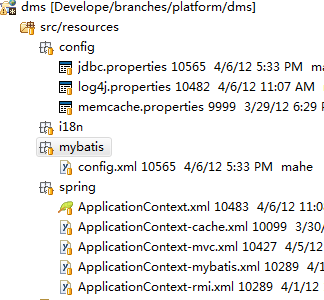
Core.resolver : 存放收集器包;

Core.util : 存放公用类包;

Unit.controller: 控制器测试类;

Unit.security: 安全测试类包;

### 资源文件的目录结构



说明

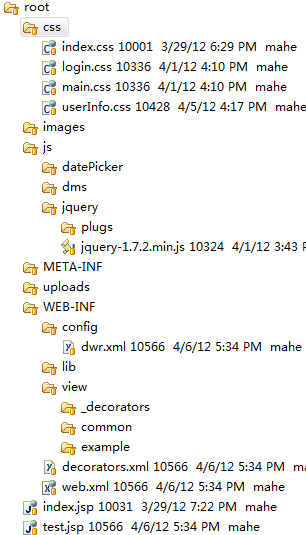
spring : Spring配置文件;

mybatis : myibatis配置文件;

i18n: 国际化资源文件

config: 其他相关配置文件;

### ROOT目录结构



说明:

Css : 存放样式文件;

Images : 存放图片;

Js : 存放javascript文件;

Uploads : 上传文件;

WEB-INF/Config : 配置文件;

WEB-INF/lib : jar包;

WEB-INF/view : 页面文件;

WEB-INF/view/common : 公共页面文件;

WEB-INF/view/\_decorators : sitemesh模板文件

## 类规范

### 实体类规范

1. 属性必须写上注释说明;
2. 必须有“get”, “set” 方法;
3. 必须继承类 “com.iwint.app.example.orm.entity.BaseEntity”;

如:

**public** **class** Role **extends** BaseEntity {

/\*\*

\* 角色名称

\*/

**private** String roleName;

/\*\*

\* 角色资源

\*/

**private** String rights;

**public** String getRoleName() {

**return** roleName;

}

**public** **void** setRoleName(String roleName) {

**this**.roleName = roleName;

}

**public** String getRights() {

**return** rights;

}

**public** **void** setRights(String rights) {

**this**.rights = rights;

}

}

### VO规范

1.属性必须写上注释说明;

2.必须继承对应的实体类;

如:

**public** **class** ViewRole **extends** BaseEntity {

/\*\*

\* 角色名称

\*/

**private** String roleName;

/\*\*

\* 角色资源

\*/

**private** String rights;

**public** String getRoleName() {

**return** roleName;

}

**public** **void** setRoleName(String roleName) {

**this**.roleName = roleName;

}

**public** String getRights() {

**return** rights;

}

**public** **void** setRights(String rights) {

**this**.rights = rights;

}

}

## 方法规范

用注释详细说明每个参数的作用、取值范围及参数

间的关系

* 建议方法的规模限制在500行以内
* 建议一个方法仅完成一件功能
* 建议为简单功能编写方法
* 建议方法的返回值要清楚、明了，让使用者不容易忽视
* 错误情况
* 建议减少方法本身或方法内的递归调用

## 页面规范

### Javascript规范

1. 每个页面的JS脚本应写在一个与页面同名的JS文件中(除特殊情况外),并且为了避免冲突, 请在javascript文件中的脚本加上命名空间.

如: person\_info.jsp 则javascript文件为: userinfo.js

页面中导入:

<head>

<title>添加</title>

<link type="text/css" rel="stylesheet" href="${ctx}/css/main.css"/>

<link type="text/css" rel="stylesheet" href="${ctx}/css/userInfo.css"/>

<script type="text/javascript" src="${ctx}/js/dms/person/person\_info.js"></script>

</head>

命名空间:

var person\_info = {

data : {},

getSaveDate : function () {

var person = {};

person.personName = $("#personName").val();

person.id = $("#id").val();

person.age = $("#age").val();

return person;

},

save : function () {

$("#saveBtn").disabled;

$.ajax({

url : "save.dms",

type : "POST",

data : person\_info.getSaveDate(),

dataType : "json",

success : function(msg) {

if(msg.success) {

alert("保存成功");

window.close();

}else{

person\_info.failed();

}

},

error : function() {

person\_info.failed();

}

});

},

failed : function () {

alert("系统发生故障,请联系管理员!");

}

}

注: “person\_info” 即命名空间.

2. 页面的javascript常量在dms/root/js/common/Const.js文件中,该文件由java类”com.iwint.core.util.Consts”自动生成,请使用该文件中的常量.

### JSPs.

* 仅仅应该使用显示bean来显示数据内容。
* JSPs应该仅仅包含渲染逻辑。

## 业务层规范

* 业务逻辑实现的地方.
* 接收传输对象或者VO.
* 返回传输对象或者VO.
* 业务逻辑实现，不包含渲染逻辑实现.
* 服务类与DAO的接口.

## 配置文件规范

1. 配置的内容需写上注释.

如 :

<!-- 导入属性配置文件 -->

<context:property-placeholder

location="classpath:config/jdbc.properties"

ignore-unresolvable="true" />

1. 配置内容应写在对应的模块配置文件中.

## 单元测试规范

SpringMVC控制器的测试类必须继承“unit.controller. UserControllerTest”；

## 其他规范

### 编码

　统一使用UTF-8编码.

### 扩展属性

所有的扩展属性都必须写在VO中.

如:

实体类:

**public** **class** Role **extends** BaseEntity {

/\*\*

\* 角色名称

\*/

**private** String roleName;

/\*\*

\* 角色资源

\*/

**private** String rights;

}

VO:

**public** **class** ViewRole **extends** Role {

/\*\*

\* 性别

\*/

**private** String sex;

/\*\*

\* 生日

\*/

**private** String birth;

}

“sex “ 与 “brith “为扩展属性.

### 常量

**常量命名规则**

类常量全部用大写字母，单词间用下划线隔

示例：

static final int MIN\_WIDTH = 2;

static final int MAX\_WIDTH = 99;

1. 常量应写在常量类中,即: com.iwint.core.util.Const.
2. 如果是集合类型的常量,应放在集合类内.

如: **public** **static** List<Integer> *paramPuxiTypeMap* = *getParamPuxiTypeMap*();

**public** **static** List<Integer> getParamPuxiTypeMap(){

**if**(*paramPuxiTypeMap* == **null**){

*paramPuxiTypeMap* = **new** ArrayList<Integer>();

*paramPuxiTypeMap*.add(Constants.*VALUETYPE\_DOUBLETYPE*);

*paramPuxiTypeMap*.add(Constants.*VALUETYPE\_INTEGERTYPE*);

*paramPuxiTypeMap*.add(Constants.*VALUETYPE\_DOUBLEARRAYTYPE*);

*paramPuxiTypeMap*.add(Constants.*VALUETYPE\_INTEGERARRAYTYPE*);

}

**return** *paramPuxiTypeMap*;

}

**public** **static** LinkedHashMap<Integer, String> *paramTypeMap* = *getParamTypeMap*();

**public** **static** LinkedHashMap<Integer, String> getParamTypeMap(){

**if**(*paramTypeMap*==**null**){

*paramTypeMap*=**new** LinkedHashMap<Integer, String>();

*paramTypeMap*.put(Constants.*VALUETYPE\_DOUBLETYPE*, "浮点型");

*paramTypeMap*.put(Constants.*VALUETYPE\_FILETYPE*, "文件型");

*paramTypeMap*.put(Constants.*VALUETYPE\_INTEGERTYPE*, "整数型");

*paramTypeMap*.put(Constants.*VALUETYPE\_STRINGTYPE*, "字符型");

*paramTypeMap*.put(Constants.*VALUETYPE\_BOOLEANTYPE*,"布尔型");

*paramTypeMap*.put(Constants.*VALUETYPE\_DOUBLEARRAYTYPE*,"浮点型数组");

*paramTypeMap*.put(Constants.*VALUETYPE\_INTEGERARRAYTYPE*,"整数型数组");

*paramTypeMap*.put(Constants.*VALUETYPE\_STRINGARRAYTYPE*,"字符型数组");

*paramTypeMap*.put(Constants.*VALUETYPE\_BOOLEANARRAYTYPE*,"布尔型数组");

*paramTypeMap*.put(Constants.*VALUETYPE\_INTEGERARRAYTYPE*,"整数型数组");

*paramTypeMap*.put(Constants.*VALUETYPE\_INTEGERTYPE*,"整数型");

}

**return** *paramTypeMap*;

}

### JSON对象

后台与前台的交互数据为JSON格式,应该把JSON数据转化为统一的对象

即: com.iwint.core.util .AjaxResult;

### 版本管理工具

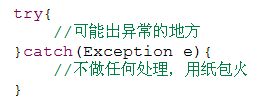
1. 提交代码之前,先进行版本同步,对于有冲突标识的文件,先进行文件对比,确保不会覆盖对方代码后,再进行更新和提交;

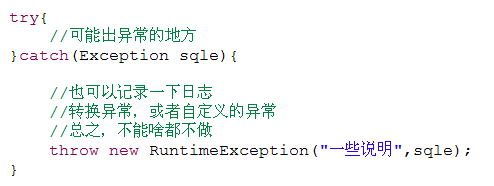
2. 提交代码时,应写上注释,注明所做的工作

如 : 1.添加了XXX 功能. 2.修改了XXX页面.

### 异常处理

* 不能处理的异常，往外层抛
* 即便异常可能影响界面友好
* 异常有助于定位出错位置





* **只针对异常的情况才使用异常**
* **对可恢复的情况使用受检异常，**
* **对编程错误使用运行时异常**

java有三种可抛出的结构（throwable），受检的异常（checked exception）、运行时异常（run-time exception）和错误（error）。

如果期望调用者能够适当地恢复，对于这种情况就应该使用**受检的异常**。或者在catch中处理，或者传播出去，如sql，io Exception。这样的异常可以提供辅助方法，使得调用者得到一些有助于恢复的信息。

用运行时异常来表明编程错误。大多数运行时异常都是前提违例（precondition violation）。就是没有遵守API规范建立的约定。如考虑资源枯竭的情形，可能是因为程序的错误引起的，如分配了不合理的大数组。如果资源是因为临时的短缺或者临时需求太大所造成的，这种情况可能就是可恢复的。API设计者需要判断这样的资源是否允许恢复。如果你相信一种情况可能允许恢复，那么使用受检异常，否则使用运行时异常。不清楚，最好使用未受检的异常。

### 类型转换

项目中所有的类型转化操作必须使用公用类中的类型转化方法.

### 注释

为了统一代码注释,请导入项目的doc目录下的” codetemplates.xml” 代码模板文件.

导入方法:

进入设置注释模板的入口： Window->Preference->Java->Code Style->Code Template , 点击”import” 按钮, 然后选择模板文件, 导入成功后, 选择 “apply”按钮.

删除标记

删除数据时,并不是从数据库中删除,而是把数据表中的”delete\_flag” 字段更新为 “1”;未删除状态为: “0”.

注释必须正确且准确的传递有用的信息。

有用的注释也可不需要遵循某个正式风格。对于一个例程定义，我们需要注释传达的信息有：其完成什么功能、参数是什么、返回值是什么、调用前的先决条件是什么，以及调用后满足的后置条件。

代码一般来讲是自解释的。

代码如果发生改变，SVN是个记录变化内容的好地方。

SVN也包含JIRA IDs，不需要把他们放到代码里面。

### 开发日志

**开发日志配置：**

所有开发日志必须使用此日志接口输出。

目前使用log4j实现开发日志，log4j配置文件所在目录：resource/log4j.properties。配置方式详见log4j文档。

开发日志使用：

在所有需要记录开发日志的类中，加入以下代码：

**private** **static** **final** Log *log* = Log.*getLog*(类名.**class**.getName());

Log提供org.apache.commons.logging.Log的所有标准API，其中isXXXEnabled系列方法，用来判断某日志记录级别是否是当前设置的级别，当构造日志记录的具体内容是一个消耗较大的过程时，需要通过这些判断方法先行判断是否需要记录，避免无谓的消耗，从而提高性能。

# SVN使用规范

1．修改文件之前首先update。这意味着修改时的版本尽可能新，一旦发生冲突，解决它的工作量会比较小；

2．及时commit。本地代码与代码库中的代码差异越小，别人合并的难度也就越小(他们有比较大的概率能够拿到新的版本) ；

3．将不同的功能单元修改分开commit。一方面，这样做能够尽早地commit，减少别人合并的难度；另一方面，由于svn提供了回退到先前版本的能力，一旦由于某项功能修改造成问题，也很容易将那次修改的内容，而不是整个修改回退到正常的代码；

4．同一功能涉及的所有代码一次commit。不希望将涉及同一功能修改的代码分开commit，因为这会给日后的追踪带来麻烦；

5．先调试后提交。这将减少别人不会因为同步了中间结果引发问题，甚至发生提交冲突的可能；

6．写清commit log(提交日志)。svn中允许保存commit log，在这里可以写为什么进行代码的修改，以及进行了什么样的修改，清楚的commit log能够帮助其他开发者在不仔细阅读代码的情况下了解修改的内容，从而极大地提高开发效率；另一方面，这些日志对于开发者自己，以及整个开发团队，都是非常宝贵的财富。

7.开发人员在创建新目录或加入新jar包后，当提交目录或jar包时，需要提交.classpath文件。

8．开发人员在提交代码前需要对代码进行格式化处理，

即必须依次使用Ctrl+A Ctrl+Shift+I,Ctrl+Shift+O, Ctrl+Shift+F快捷键进行格式化

后再提交。

9．代码提交时间

代码提交时间：?下午三点

# 数据库设计规范

## 5.1 数据对象（表、视图、索引等）命名规范

### 5.1.1 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，最好一律用大写；

### 5.1.2 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）

## 5.2. 设计原则

### 5.2.1 表的设计

#### 5.2.1.1 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. 一个表的某列与另一表有关联关系的时候，如果加得上的话，请加上外键约束。外键是很重要的，所以特别强调几点：

* 适量建外键。为了保证外键的一致性，数据库会增加一些开销，如果有确凿的并且是对性能影响到无法满足用户需求的证据，最好考虑不建外键。
* 不同组件之间表的关联，不允许建外键。
* 不要以数据操作不方便为理由而不建外键。是的，加上外键以后，一些数据操作变得有些麻烦，但是这正是对数据一致性的保护。这种保护很有效，所以最好不要拒绝它；
* 外键在保护数据一致方面非常有效。如果不建外键，数据库中容易出现垃圾数据，并且无人知晓。当数据量很大的时候，查找这些垃圾数据也是相当困难的。而应用程序在设计时，往往没有考虑或者也无法照顾到垃圾数据。因此垃圾数据很可能造成应用程序工作不正常，并且表现出来的现象会很奇怪，让人摸不着头脑。

#### 5.2.1.2 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；

### 5.2.2 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，对性能会有一定影响，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；索引对性能会有一定影响，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。

### 5.2.3 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；

### 5.2.4 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；

## 5.3. SQL的设计和使用

### 5.3.1 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；

### 5.3.2 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；

SELECT　＊

FROM　service\_promotion

WHERE TO\_CHAR(gmt\_modified,’yyyy-mm-dd’)

= ‘20001-09-01’;

　而应使用：

SELECT \*

FROM service\_promotion

WHERE gmt\_modified

>= TO\_DATE(‘2001-9-01’,’yyyy-mm-dd’)

AND gmt\_modified

< TO\_DATE(‘2001-9-02’,’yyyy-mm-dd’);

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；

## 5.4. 其他与性能有关的设计原则

上文中已经有不少条目，是与提高数据库访问的性能相关的，这里就不再重复了。本段落中提出一些与数据库性能极为相关，但上文又未涉及的条目；

1. 大数据量的开发环境

开发过程中，开发人员往往使用一个非常简单的、只有很少数据的数据库环境，便于程序的调试。但是，最好还提供一个数据量比较大的环境，数据量和应用的设计目标差不多，供开发人员使用。这样开发人员能立刻发现那些非常慢的数据库操作，比如写了一个非常不合理的SQL，因而能够在此时就排除，而不必等到测试或者甚至上线时才发觉。因为很多开发人员，由于对数据库的认识不够，通常只以完成功能为首要目的，容易写出效率非常低的SQL；

2. 模拟测试

这里说的测试主要是指模拟实际应用对数据库的操作，进行测试，并发及访问量要模拟应用的设计需求。主要目的是了解在系统上线后，数据库的大概状况，避免一上线就崩溃或者慢得不能忍受的情况。提前发现瓶颈，做相应的优化；

3. 应用测试

每个应用的测试阶段，特别是性能方面的测试，也要关注数据库的表现。如果出现瓶颈，根据具体情况，此时仍然有可能修改部分数据库设计并修改相应程序；

## 5.5. 数据库相关工具

为需要写SQL的开发人员提供方便的、图形化的客户端软件，可以用来执行SQL，看到表结构、索引等等；

# 程序效率规范

程序优化的第一条原则 ：不要优化。

从实现的角度，当你考虑一个备选方案时，一定要使用测试数据对它们进行评估；在编写代码时，要尽早并经常的测量它的性能关键部分，以避免之后会有意想不到的事情发生。

1：编程时要经常注意代码的效率。

说明：代码效率分为**全局效率、局部效率、时间效率及空间效率**。

全局效率是站在整个系统的角度上的系统效率；

局部效率是站在模块或函数角度上的效率；

时间效率是程序处理输入任务所需的时间长短；

空间效率是程序所需内存空间，如机器代码空间大小、数据空间大小、栈空间大小等。

2：在保证软件系统的正确性、稳定性、可读性及可测性的前提下，提高代码效率。

说明：不能一味地追求代码效率，而对软件的正确性、稳定性、可读性及可测性造成影响。

3：局部效率应为全局效率服务，不能因为提高局部效率而对全局效率造成影响。

4：通过对系统数据结构的划分与组织的改进，以及对程序算法的优化来提高空间效率。

说明：这种方式是解决软件空间效率的根本办法。

示例：如下记录学生学习成绩的结构不合理。

public StudentScoreStru{

string name;

string age;

string sex;

string class;

string subject;

float score;

}

因为每位学生都有多科学习成绩，故如上结构将占用较大空间。应如下改进（分为两个结构），总的存贮空间将变小，操作也变得更方便。

public StudentStru

{

string name;

string age;

string sex;

string class;

}

public StudentScoreStru

{

Inteter studentIndex;

string subject;

float score;

}

**5：循环体内工作量最小化。**

说明：应仔细考虑循环体内的语句是否可以放在循环体之外，使循环体内工作量最小，从而提高程序的时间效率。

示例：如下代码效率不高。

for (ind = 0; ind < MAX\_ADD\_NUMBER; ind++)

{

sum += ind;

back\_sum = sum; /\* backup sum \*/

}

语句“back\_sum = sum;”完全可以放在for语句之后，如下。

for (ind = 0; ind < MAX\_ADD\_NUMBER; ind++)

{

sum += ind;

}

back\_sum = sum; /\* backup sum \*/

6：仔细分析有关算法，并进行优化。

7：仔细考查、分析系统及模块处理输入（如事务、消息等）的方式，并加以改进。

8：对模块中函数的划分及组织方式进行分析、优化，改进模块中函数的组织结构，提高程序效率。

说明：软件系统的效率主要与算法、处理任务方式、系统功能及函数结构有很大关系，仅在代码上下功夫一般不能解决根本问题。

9：编程时，要随时留心代码效率；优化代码时，要考虑周全。

10：不应花过多的时间拼命地提高调用不很频繁的函数代码效率。

说明：对代码优化可提高效率，但若考虑不周很有可能引起严重后果。

11：要仔细地构造或直接用汇编编写调用频繁或性能要求极高的函数。

说明：只有对编译系统产生机器码的方式以及硬件系统较为熟悉时，才可使用汇编嵌入方式。嵌入汇编可提高时间及空间效率，但也存在一定风险。

12：在保证程序质量的前提下，通过压缩代码量、去掉不必要代码以及减少不必要的局部和全局变量，来提高空间效率。

说明：这种方式对提高空间效率可起到一定作用，但往往不能解决根本问题。

13：在多重循环中，应将最忙的循环放在最内层。

说明：减少CPU切入循环层的次数。

示例：如下代码效率不高。

for (row = 0; row < 100; row++)

{

for (col = 0; col < 5; col++)

{

sum += a[row][col];

}

}

可以改为如下方式，以提高效率。

for (col = 0; col < 5; col++)

{

for (row = 0; row < 100; row++)

{

sum += a[row][col];

}

}

14：尽量减少循环嵌套层次。

15：避免循环体内含判断语句，应将循环语句置于判断语句的代码块之中。

说明：目的是减少判断次数。循环体中的判断语句是否可以移到循环体外，要视程序的具体情况而言，一般情况，与循环变量无关的判断语句可以移到循环体外，而有关的则不可以。

示例：如下代码效率稍低。

for (ind = 0; ind < MAX\_RECT\_NUMBER; ind++)

{

if (data\_type == RECT\_AREA)

{

area\_sum += rect\_area[ind];

}

else

{

rect\_length\_sum += rect[ind].length;

rect\_width\_sum += rect[ind].width;

}

}

因为判断语句与循环变量无关，故可如下改进，以减少判断次数。

if (data\_type == RECT\_AREA)

{

for (ind = 0; ind < MAX\_RECT\_NUMBER; ind++)

{

area\_sum += rect\_area[ind];

}

}

else

{

for (ind = 0; ind < MAX\_RECT\_NUMBER; ind++)

{

rect\_length\_sum += rect[ind].length;

rect\_width\_sum += rect[ind].width;

}

}

16：尽量用乘法或其它方法代替除法，特别是浮点运算中的除法。

说明：浮点运算除法要占用较多CPU资源。

17：不要一味追求紧凑的代码。

说明：因为紧凑的代码并不代表高效的机器码。

18：显式地把已经不再被引用的对象赋为null。

说明：因为紧凑的代码并不代表高效的机器码。

19：不要频繁初始化对象。

* + 除非必要，否则不要在循环内初始化对象。

示例：

**for (int i = 0; i <rows; i++) {**

**CommonWrkSht commonWrkSht = new CommonWrkSht();**

**…**

**}**

**应写成：**

**CommonWrkSht commonWrkSht = new CommonWrkSht();**

**for (int i = 0; i <rows; i++) {**

**…**

**}**

20：打开的数据库连接、文件在使用后必须关闭。

* + 在finally中关闭数据库连接、文件。

21：合理设计程序，避免占用太大内存。

22：不要在循环内执行重复操作。

* 在循环外调用一次的，就避免在循环内进行不必要的反复调用。

for (int i = 0; i < dw.getRowCount(); i++) {

…

}

应写成：

int rows = dw.getRowCount();

for (int i = 0; i <rows; i++) {

…

}

23：在对字符串有附加操作时，使用StringBuffer而非String。

# 程序设计优化

* **对于系统的关键组件和频繁使用的对象，使用单例模式便可以有效地改善系统的性能**。

ppublic abstract class Singleton {

private Singleton(){

//创建单例的过程可能会比较慢

}

private static Singleton instance = new Singleton(){};

public static Singleton getInstance() {

return instance;

}

}

public class LazySingleton {

private LazySingleton(){

//创建单例的过程可能会比较慢

System.out.println("LazySingleton is create");

}

private static LazySingleton instance = null;

public static **synchronized** LazySingleton getInstance() {

if (instance==null)

instance=new LazySingleton();

return instance;

}

}

上面延迟加载的单例类，虽然实现了延迟加载的功能，但和第一种方法相比，它引入了同步关键字，因此在多线程环境中，它的耗时要远远大于第一种单例模式。

为解决这个问题，还需要对其进行改进：

public class StaticSingleton {

private StaticSingleton(){

System.out.println("StaticSingleton is create");

}

private static class SingletonHolder {

private static StaticSingleton instance = new StaticSingleton();

}

public static StaticSingleton getInstance() {

return SingletonHolder.instance;

}

public static void createString(){

System.out.println("createString in Singleton");

}

}

单例模式使用内部类来维护单例的实例，当StaticSingleton被加载时，其内部类并不会被初始化，故可以确保当StaticSingleton类被载入JVM时，不会初始化单例类，而当

getInstance()方法被调用时，才会加载SingletonHolder，从而初始化instance。同时，由于实例的建立是在类加载时完成，故天生对多线程友好，getInstance()方法也不需要使用同步关键字。因此，这种实现方式同时具备以上两种实现方式的优点。

* **使用代理模式实现延迟加载，从而提升系统的性能和反映速度**

在系统启动时，将消耗资源最多的方法都使用代理模式分离，就可以加快系统的启动速度，减少用户的等待时间。而在用户真正做查询操作时，再由代理类，单独去加载真实的数据库查询类，完成用户的请求。这个过程就用代理模式实现了延迟加载。

代理模式可以用于多种场合，如用于远程调用的网路代理，考虑安全因素的安全代理等，延迟加载只是代理模式的一种应用场景。

public interface IDBQuery {

String request();

}

DBQuery实现如下，它是一个重量级对象，构造会比较慢：

public class DBQuery implements IDBQuery{

public DBQuery(){

//可能包含数据库连接等耗时操作

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

@Override

public String request() {

return "request string";

}

}

代理类DBQueryProxy是轻量级对象，创建很快，用于替代DBQuery的位置：

public class DBQueryProxy implements IDBQuery {

private DBQuery real=null;

@Override

public String request() {

//在真正需要的时候 ，才创建真实对象，创建过程可能很慢

if(real==null)

real=new DBQuery();

//在多线程环境下，这里返回一个虚假类，类似于Future模式

return real.request();

}

}

最后，主函数如下，它引用IDBQuery接口，并使用代理类工作：

public class Main {

public static void main(String args[]){

IDBQuery q=new DBQueryProxy(); //使用代理

q.request(); //在真正使用时才创建真实对象

}

}

动态代理，CGLIB是高级的字节码生成库，总体性能要比JDK自带的动态代理好。

import java.lang.reflect.Method;

import javatuning.ch2.proxy.IDBQuery;

import net.sf.cglib.proxy.MethodInterceptor;

import net.sf.cglib.proxy.MethodProxy;

public class CglibDbQueryInterceptor implements MethodInterceptor {

IDBQuery real=null;

@Override

public Object intercept(Object arg0, Method arg1, Object[] arg2,

MethodProxy arg3) throws Throwable {

if(real==null)

real=new DBQuery();

return real.request();

}

}

* **使用亨元模式提高系统性能**

亨元模式是为数不多的，只为提升系统性能而生的设计模式，它的主要作用就是复用大对象（重量级的对象），以节省内存空间和对象创建时间。

public interface IReportManager {

public String createReport();

}

public class FinancialReportManager implements IReportManager {

protected String tenantId=null;

public FinancialReportManager(String tenantId){

this.tenantId=tenantId;

}

@Override

public String createReport() {

return "This is a financial report";

}

}

public class EmployeeReportManager implements IReportManager {

protected String tenantId=null;

public EmployeeReportManager(String tenantId){

this.tenantId=tenantId;

}

@Override

public String createReport() {

return "This is a employee report";

}

}

最为核心的亨元工厂类实现如下，它也是亨元模式的精髓。它确保同一个公司使用相同的对象产生报表。这是相当有意义的，否则系统可能为每个员工生成各自的报表对象，导致系统开销激增。

import java.util.HashMap;

import java.util.Map;

public class ReportManagerFactory {

Map<String ,IReportManager>

financialReportManager=new HashMap<String ,IReportManager>();

Map<String ,IReportManager>

employeeReportManager=new HashMap<String ,IReportManager>();

IReportManager getFinancialReportManager(String tenantId){

IReportManager r=financialReportManager.get(tenantId);//通过租户ID获取亨元

if(r==null){

r=new FinancialReportManager(tenantId);

financialReportManager.put(tenantId, r);//维护已创建的亨元对象

}

return r;

}

IReportManager getEmployeeReportReportManager(String tenantId){

IReportManager r=employeeReportManager.get(tenantId);

if(r==null){

r=new EmployeeReportManager(tenantId);

employeeReportManager.put(tenantId, r);

}

return r;

}

}

public class Main {

public static void main(String[] args) {

ReportManagerFactory rmf=new ReportManagerFactory();

IReportManager rm=rmf.getFinancialReportManager("A");

System.out.println(rm.createReport());

}

}

* **使用装饰者模式可以有效分离性能组件和功能组件，从而提高模块的可维护性并增加模块的复用性。**

import java.io.BufferedOutputStream;

import java.io.DataOutputStream;

import java.io.FileNotFoundException;

import java.io.FileOutputStream;

import java.io.IOException;

public class Main {

/\*\*

\* @param args

\* @throws IOException

\*/

public static void main(String[] args) throws IOException {

//生成一个有缓冲功能的流对象

**DataOutputStream dout=new**

**DataOutputStream(new BufferedOutputStream(new**

**FileOutputStream("C:\\a.txt")));**

**//**没有有缓冲功能的流对象

//DataOutputStream dout=new DataOutputStream(new

FileOutputStream("C:\\a.txt"));

long begin=System.currentTimeMillis();

for(int i=0;i<100000;i++)

dout.writeLong(i);

System.out.println("spend:"+(System.currentTimeMillis()-begin));

}

}

以上代码，第一种加入了性能组件**BufferedOutputStream，**第二种则没有。因此，第一种方法产生的OutputStream拥有更好的I/O性能。

装饰者和被装饰者拥有相同的接口。被装饰者通常是系统的核心组件，完成特定的功能目标。而装饰者则可以在被装饰者的方法前后，加上特定的前置处理和后置处理，增强被装饰者的功能。

public interface IPacketCreator {

public String handleContent();

}

public abstract class PacketDecorator implements IPacketCreator{

IPacketCreator componet;

public PacketDecorator(IPacketCreator c){

componet=c;

}

}

public class PacketBodyCreator implements IPacketCreator{

@Override

public String handleContent() {

return "Content of Packet";

}

}

public class PacketHTTPHeaderCreator extends PacketDecorator{

public PacketHTTPHeaderCreator(IPacketCreator c) {

super(c);

}

@Override

public String handleContent() {

StringBuffer sb=new StringBuffer();

sb.append("Cache-Control:no-cache\n");

sb.append("Date:Mon,31Dec201204:25:57GMT\n");

sb.append(componet.handleContent());

return sb.toString();

}

}

public class PacketHTMLHeaderCreator extends PacketDecorator{

public PacketHTMLHeaderCreator(IPacketCreator c) {

super(c);

}

@Override

public String handleContent() {

StringBuffer sb=new StringBuffer();

sb.append("<html>");

sb.append("<body>");

sb.append(componet.handleContent());

sb.append("</body>");

sb.append("</html>\n");

return sb.toString();

}

}

public class Main {

public static void main(String[] args) {

IPacketCreator pc=new PacketHTTPHeaderCreator(

new PacketHTMLHeaderCreator(

new PacketBodyCreator()));

System.out.println(pc.handleContent());

}

}

* **使用****Value Object模式，对****传输数据进行有效的封装，有效减少到网络交互次数，可以明显提升远程方法调用的性能**

Value Object模式提倡将一个对象的各个属性进行封装，将封装后的对象

在网络中传递，从而使系统拥有更好的交互模型，并且减少网络通信数

据，从而提高性能。

RIM服务端的接口实现如下，其中**getOrder()**方法取得一个**Value Object，**其他

方法取得Order对象的一部份信息：

import java.rmi.Remote;

import java.rmi.RemoteException;

public interface IOrderManager extends Remote {

**public Order getOrder(int id) throws RemoteException;**

public String getClientName(int id) throws RemoteException;

public String getProdName(int id) throws RemoteException;

public int getNumber(int id) throws RemoteException;

}

一个最简单的IOrderManager的实现，它什么也不做，只是返回数据：

import java.rmi.RemoteException;

import java.rmi.server.UnicastRemoteObject;

public class OrderManager extends UnicastRemoteObject implements IOrderManager {

protected OrderManager() throws RemoteException {

super();

}

private static final long serialVersionUID = -1717013007581295639L;

@Override

public Order getOrder(int id) throws RemoteException {

Order o=new Order();

o.setClientName("billy");

o.setNumber(20);

o.setProdunctName("desk");

return o;

}

@Override

public String getClientName(int id) throws RemoteException {

return "billy";

}

@Override

public String getProdName(int id) throws RemoteException {

return "desk";

}

@Override

public int getNumber(int id) throws RemoteException {

return 20;

}

}

作为**Value Object的Order**对象实现如下：

public class Order implements java.io.Serializable{

private int orderid;

private String clientName;

private int number;

private String produnctName;

public int getOrderid() {

return orderid;

}

public void setOrderid(int orderid) {

this.orderid = orderid;

}

public String getClientName() {

return clientName;

}

public void setClientName(String name) {

this.clientName = name;

}

public int getNumber() {

return number;

}

public void setNumber(int number) {

this.number = number;

}

public String getProdunctName() {

return produnctName;

}

public void setProdunctName(String produnctName) {

this.produnctName = produnctName;

}

}

业务逻辑层注册并开启RMI服务器：

import java.rmi.Naming;

import java.rmi.registry.LocateRegistry;

public class OrderManagerServer {

public static void main(String[] argv)

{

try

{

LocateRegistry.createRegistry(1099);

IOrderManager usermananger = new OrderManager();

Naming.rebind("OrderManager", usermananger);

System.out.println("OrderManager is ready.");

}

catch (Exception e)

{

System.out.println("OrderManager Server failed: " + e);

}

}

}

客户端的测试代码如下，它分别展示了使用Value Object模式封装数据和不使用Value Object模式的性能差异：

import java.rmi.Naming;

public class Client {

public static void main(String[] argv) {

try {

IOrderManager usermanager = (IOrderManager) Naming

.lookup("OrderManager");

long begin = System.currentTimeMillis();

for (int i = 0; i < 1000; i++) {

**usermanager.getOrder(i);　　　　　／／**Value Object模式

}

System.out.println("getOrder spend:"

+ (System.currentTimeMillis() - begin));

begin = System.currentTimeMillis();

for (int i = 0; i < 1000; i++) {

usermanager.getClientName(i);　／／通过多次交互获取数据

usermanager.getNumber(i);

usermanager.getProdName(i);

}

System.out.println("3 Method call spend:"

+ (System.currentTimeMillis() - begin));

System.out.println(usermanager.getOrder(0).getClientName());

} catch (Exception e) {

System.out.println("OrderManager exception: " + e);

}

}

}

结果显示，使用getOrder()方法相对耗时４６９ms,　而使用连续３次离散的远程调用耗时

７６６ms。由此可见，对传输数据进行有效的封装，有效减少到网络交互次数，可以明显提升远程方法调用的性能。

* **业务代理模式**

是将一组由远程方法调用构成的业务流程，封装在一个位于展示层的代理类中。比如，如果用户要修改一个订单，订单修改操作可细分为３个子操作：

* 校验用户
* 获取旧的定单信息
* 更新订单

展示层 业务逻辑层 持久层

RDBMS

checkUser

getOrder

updateOrder

以上结构存在两个问题：

1. 当展示层存在大量的并发线程时，这些线程都会直接进行远程调用，进而会加重网络负担；
2. 由于缺乏对订单修改操作流程的有效封装，如果将来修改流程发生变化，那么展示层组件需要修改。

为解决以上两个问题，可以在展示层中加入业务代理对象，业务代理对象负责和远程服务器对象通信，完成订单修改操作。而业务代理对象本身只暴露简单的updateOrder()订单修改操作供展示层组件使用，修改后的结构如下图 ：

展示层 业务逻辑层 持久层

updateOrder 业务代理

基于远程

调用封装

业务逻辑，

提供缓存

功能

RDBMS

checkUser

getOrder

updateOrder

业务代理模式架构图

这种结构体现了业务代理模式的核心思想。由于该业务代理对象被所有的展示层请求线程和多个客户端共享，故系统将会有较好的可维护性。如果业务流程发生变化，只需要简单地修改业务代理对象暴露的updateOrder()方法即可。除此之外，通过业务代理对象，可以更容易地在多个线程或者客户端请求之间共享数据，从而有效地利用缓存，减少远程调用次数，提高系统性能。

import java.rmi.Remote;

import java.rmi.RemoteException;

import javatuning.ch2.vo.Order;

public interface IOrderManager extends Remote {

public Order getOrder(int id) throws RemoteException;

public boolean checkUser(int userid) throws RemoteException;

public boolean updateOrder(Order order) throws RemoteException;

}

import java.rmi.RemoteException;

import java.rmi.server.UnicastRemoteObject;

import javatuning.ch2.vo.Order;

public class OrderManager extends UnicastRemoteObject implements IOrderManager {

public OrderManager() throws RemoteException {

super();

}

private static final long serialVersionUID = -1717013007581295639L;

@Override

public Order getOrder(int id) throws RemoteException {

Order o=new Order();

o.setClientName("billy");

o.setNumber(20);

o.setProdunctName("desk");

return o;

}

@Override

public boolean checkUser(int userid) throws RemoteException {

return true;

}

@Override

public boolean updateOrder(Order order) throws RemoteException {

return true;

}

}

import java.rmi.Naming;

import java.rmi.registry.LocateRegistry;

public class OrderManagerServer {

public static void main(String[] argv)

{

try

{

LocateRegistry.createRegistry(1099);

IOrderManager usermananger = new OrderManager();

Naming.rebind("OrderManager", usermananger);

System.out.println("OrderManager is ready.");

}

catch (Exception e)

{

System.out.println("OrderManager Server failed: " + e);

}

}

}

在业务代理对象BusinessDelegate中，可以增加缓存，从而直接减少远程方法调用的次数。

import java.net.MalformedURLException;

import java.rmi.Naming;

import java.rmi.NotBoundException;

import java.rmi.RemoteException;

import javatuning.ch2.vo.Order;

public class BusinessDelegate {

IOrderManager usermanager =null; /**/封装远程方法调用的流程**

public BusinessDelegate(){

try {

usermanager = (IOrderManager) Naming.lookup("OrderManager");

} catch (MalformedURLException e) {

e.printStackTrace();

} catch (RemoteException e) {

e.printStackTrace();

} catch (NotBoundException e) {

e.printStackTrace();

}

}

public boolean checkUserFromCache(int uid){

return true;

}

public boolean checkUser(int uid) throws RemoteException{

**//可以在本地缓存中获取订单，减少远程方法调用次数**

if(!checkUserFromCache(uid)){

return usermanager.checkUser(1);

}

return true;

}

public Order getOrderFromCache(int oid){

return null;

}

public Order getOrder(int oid) throws RemoteException{

Order order=getOrderFromCache(oid);

if(order==null){

return usermanager.getOrder(oid);

}

return order;

}

**//暴露给展示层的方法，封装了业务流程，可能在缓存中执行**

public boolean updateOrder(Order order) throws Exception{

if(checkUser(1)){

Order o=getOrder(1);

o.setNumber(10);

usermanager.updateOrder(o);

}

return true;

}

}

import java.rmi.Naming;

import java.rmi.RemoteException;

import javatuning.ch2.vo.Order;

public class BusinessDelegateClient {

public static void main(String[] argv) throws Exception {

//未使用业务代理模式的展示层实现

IOrderManager usermanager =null;

usermanager = (IOrderManager) Naming.lookup("OrderManager");

if(usermanager.**checkUser**(1)){//所有的远程调用都会被执行，当并发量很大时，严

//重影响性能

Order o = usermanager.**getOrder**(1);

o.setNumber(10);

usermanager.**updateOrder**(o);

}

//使用了业务代理层后，展示层组件可以优化成如下

BusinessDelegate bd=new BusinessDelegate();

Order o=bd.getOrder(11);

o.setNumber(11);

bd.updateOrder(o);

}

* **常用优化组件和方法—缓冲(buffer)**

缓冲区是一块特定的内存区域。开辟缓冲区的目的是通过缓解应用程序上下层之间的性能差异，提高系统的性能。

缓冲最常用的场景是提高I/O的速度。

import java.io.BufferedWriter;

import java.io.File;

import java.io.FileWriter;

import java.io.IOException;

import java.io.Writer;

import org.junit.Test;

public class FileWriterBufferTest {

public static final int CIRCLE=100000;

@Test

public void testFileWriter() throws IOException {

Writer writer = new FileWriter(new File("file.txt"));

long begin=System.currentTimeMillis();

for (int i = 0; i < CIRCLE; i++) {

writer.write(i);

}

writer.close();

System.out.println("testFileWriter spend:"+(System.currentTimeMillis()-begin));

}

@Test

public void testFileWriterBuffer() throws IOException {

//为进行I/O优化，可以为FileWriter加上缓冲

Writer writer = new **BufferedWriter**(new FileWriter(new File("file.txt")));

long begin=System.currentTimeMillis();

for (int i = 0; i < CIRCLE; i++) {

writer.write(i);

}

writer.close();

System.out.println("testFileWriterBuffer

spend:"+(System.currentTimeMillis()-begin));

}

}

除了能够改善I/O性能，缓冲区对于任何一种上下层组件存在性能差异的场合都可以起到很好的效果。

下面的例子使用缓冲区提升动画显示效果。

import java.awt.Color;

import java.awt.Graphics;

import java.awt.Graphics2D;

import java.awt.Image;

import javax.swing.JApplet;

public class NoBufferMovingCircle extends JApplet implements Runnable {

Image screenImage = null;

Thread thread;

int x = 5;

int move = 1;

public void init() {

screenImage = createImage(230, 160);

}

public void start() {

if (thread == null) {

thread = new Thread(this);

thread.start();

}

}

public void run() {

try {

while (true)

{

x += move;

if ((x > 105) || (x < 5))

move \*= -1;

repaint();

Thread.sleep(10);

}

} catch (Exception e) {

}

}

public void drawCircle(Graphics gc) {

Graphics2D g = (Graphics2D) gc;

g.setColor(Color.GREEN);

g.fillRect(0, 0, 200, 100);

g.setColor(Color.red);

g.fillOval(x, 5, 90, 90);

}

public void paint(Graphics g) {

g.setColor(Color.white);

g.fillRect(0, 0, 200, 100);

drawCircle(g);

}

}

以上代码没有main函数，但在Eclipse中，可以通过右键菜单中，Run As 下的Java Applet运行。结果显示，效果较差。

为了能得到更优质的显示效果，可以为它加上缓冲区：

import java.awt.Color;

import java.awt.Graphics;

public class BufferMovingCircle extends NoBufferMovingCircle {

Graphics doubleBuffer = null; //缓冲区

public void init() {

super.init();

doubleBuffer = screenImage.getGraphics();

}

public void paint(Graphics g) {

doubleBuffer.setColor(Color.white); //先在内存中画图

doubleBuffer.fillRect(0, 0, 200, 100);

drawCircle(doubleBuffer);

g.drawImage(screenImage, 0, 0, this); //将buffer一次性显示出来

}

}

* **常用优化组件和方法—缓存****(Cac he)**

缓存(Cache)也是一块为提升系统性能而开辟的内存空间。缓存的主要作用是暂存数据处理结果，并提供下次访问使用。

缓存可以保存一些来之不易的数据或者处理结果。当需要再次使用这些数据时，可以从

缓存中低成本的获取，而不需要再占用宝贵的系统资源。

以　EHCache　为例介绍一下缓存的基本使用方法：

<ehcache>

<diskStore path="data/ehcache" />

<defaultCache maxElementsInMemory="10000" eternal="false"

overflowToDisk="true" timeToIdleSeconds="120" timeToLiveSeconds="120"

diskPersistent="false" diskExpiryThreadIntervalSeconds="120" />

<cache name="cache1" maxElementsInMemory="100" eternal="false"

timeToIdleSeconds="6" timeToLiveSeconds="60" overflowToDisk="true"

diskPersistent="false" />

<cache name="cache2" maxElementsInMemory="100000" eternal="false"

timeToIdleSeconds="300" timeToLiveSeconds="600" overflowToDisk="false"

diskPersistent="false" />

</ehcache>

* maxElementsInMemory：该缓存中允许存放的最大条目数量
* eternal：缓存内容是否永久储存
* overflowToDisk：如果内存中的数据超过maxElementsInMemory，是否使用磁盘存储
* timeToIdleSeconds：如果不是永久储存的缓存，那么在timeToIdleSeconds指定时间内没有访问一个条目，则移除它。
* timeToLiveSeconds：如果不是永久储存的缓存，一个条目可以存在的最长时间。
* diskPersistent：磁盘存储的条目是否永久保存。

import java.io.IOException;

import java.io.Serializable;

import java.util.HashMap;

import net.sf.ehcache.Cache;

import net.sf.ehcache.CacheException;

import net.sf.ehcache.CacheManager;

import net.sf.ehcache.Element;

public class EHCacheUtil {

static CacheManager manager=null;

static String configfile="javatuning/ch2/buffer/ehcache.xml";

static{

try {

manager = CacheManager.create(EHCacheUtil.class.getClassLoader().getResourceAsStream(configfile));

} catch (CacheException e) {

e.printStackTrace();

}

}

public static void put(String cachename,Serializable key,Serializable value){

manager.getCache(cachename).put(new Element(key, value));

}

public static Serializable get(String cachename,Serializable key){

try {

Element e=manager.getCache(cachename).get(key);

if(e==null)return null;

return e.getValue();//取得缓存中的数据

} catch (IllegalStateException e) {

e.printStackTrace();

} catch (CacheException e) {

e.printStackTrace();

}

return null;

}

import static org.junit.Assert.\*;

import org.junit.Test;

public class TestEHCacheUtil {

//@Test

public void test1() throws InterruptedException {

EHCacheUtil.put("cache1", "key1","value1");

System.out.println(EHCacheUtil.get("cache1", "key1"));

for(int i=0;i<7;i++){

//EHCacheUtil.get("cache1", "key1");

Thread.sleep(1000);

}

System.out.println(EHCacheUtil.get("cache1", "key1"));

}

@Test

public void test2() throws InterruptedException {

for(int i=0;i<200;i++){

EHCacheUtil.put("cache1", "key"+i,"value"+i);

System.out.println("i="+i+" key1="+EHCacheUtil.get("cache1", "key1"));

}

}

}

//基于动态代理的缓存解决方案，最大好处是，在业务层，无须关注对缓存的操作，缓存操//作代码被完全独立并隔离。

public class HeavyMethodDemo {

public String heavyMethod(int num) {

int i;

StringBuffer sb = new StringBuffer();

boolean flag = false;

for (i = 2; i <= num; i++) {

if (flag == true) {

sb.append("\*");

flag = false;

}

if (num % i == 0) {

sb.append(i);

flag = true;

num = num / i;

while (num % i == 0) {

num = num / i;

if (flag == true) {

sb.append("\*");

flag = false;

}

sb.append(i);

flag = true;

}

}

}

return sb.toString();

}

public static void main(String[] args) {

HeavyMethodDemo y = new HeavyMethodDemo();

System.out.println(y.heavyMethod(100));

System.out.println(Integer.MAX\_VALUE-1);

}

}

//使用ＣＧＬＩＢ生成动态代理类的方法拦截器的逻辑如下：

import java.io.Serializable;

import java.lang.reflect.Method;

import net.sf.cglib.proxy.Enhancer;

import net.sf.cglib.proxy.MethodInterceptor;

import net.sf.cglib.proxy.MethodProxy;

public class CglibHeavyMethodInterceptor implements MethodInterceptor {

HeavyMethodDemo real=new HeavyMethodDemo();

@Override

public Object intercept(Object arg0, Method arg1, Object[] arg2,

MethodProxy arg3) throws Throwable {

String v=(String)EHCacheUtil.get("cache1", (Serializable)arg2[0]);

if(v==null){

v=real.heavyMethod((Integer)arg2[0]);

EHCacheUtil.put("cache1", (Integer)arg2[0], v);

}

return v;

}

//在这个方法拦截器中，实现了对缓存的操作，它首先查询系统是否已经计算并缓存了所//请求的数字，如果没有，则进行计算，并将结果保存在缓存中；如果有，则直接从缓存//中取得结果。在使用动态代理时，可以通过下面的代码生成动态代理对象，包含上述缓//存逻辑：

public static HeavyMethodDemo newCacheHeavyMethod(){／／生成带有缓存功能的类

Enhancer enhancer = new Enhancer();

enhancer.setSuperclass(HeavyMethodDemo.class);

enhancer.setCallback(new CglibHeavyMethodInterceptor()); ／／设置缓存逻辑

HeavyMethodDemo cglibProxy = (HeavyMethodDemo) enhancer.create();

return cglibProxy;

}

public static HeavyMethodDemo newHeavyMethod(){

return new HeavyMethodDemo();

}

public static void main(String args[]){

HeavyMethodDemo m=**newCacheHeavyMethod();　／／使用缓存**

long begin = System.currentTimeMillis();

for(int i=0;i<100000;i++)

m.heavyMethod(2147483646);

System.out.println("cache method spend:"+(System.currentTimeMillis()-begin));

m=**newHeavyMethod();　　　／／不使用缓存**

begin = System.currentTimeMillis();

for(int i=0;i<100000;i++)　　　／／不使用缓存时每次都要计算

m.heavyMethod(2147483646);

System.out.println("no cache method spend:"+(System.currentTimeMillis()-begin));

}

}

* **对象复用------“池”**

# 数据库调优规范

对绝大部分应用系统而言，数据库是必不可少的一部分。Java程序可以使用JDBC的方式连接数据库。对数据库的调优可以分为3个部分：

* 在应用层对SQL语句进行优化；
* 对数据库进行优化；
* 对数据库软件进行优化。

在应用层优化数据库访问，涉及大量的编程技巧。比如，当使用JDBC进行查询时，对于大量的拥有相同结构的SQL查询，可以使用PreparedStatement代替Statement，以提高数据库的查询效率；在Select语句中，显示指定要查询的列名，避免使用星号“\*”。

在对数据库进行优化时，主要目的是建立一个具有良好表结构的数据库。比如，为了提高多表级联查询效率，可以合理地使用冗余字段；对于大表，可以使用行的水平切割或者类似Oracle分区表的技术；为了提高数据库查询效率，可以建立有效且合理的索引。

对于数据库软件的优化，根据不同的数据库，如Oracle、MySQL或者SQL Server都拥有不同的方式。以Oracle为例，设置合理大小的共享池、缓存缓冲区或者PGA，对Oracle的运行性能都有很大的影响。

具体调优内容请参见：

[面向程序员的数据库访问性能优化法则.doc](面向程序员的数据库访问性能优化法则.docx)