**智能餐饮平台JAVA基础服务框架开发手册（2015版）**

1. **Tuan系统介绍**
   1. Tuan系统介绍

业务领域：

智能餐饮

定位：

互联网应用

智能家居，物联网，云服务（大数据）

区别一般企业内部管理系统，如S2SH+Oracle+Tomcat的OA系统、人事管理系统

目标：

大型网站系统：业务不断膨胀，需求不断变化，适应互联网快速发展的特点

大量用户

高并发高性能

高可用：7\*24小时不间断服务

可扩展：稳定的集群运行环境

1. **整体部署结构**
   1. 整体部署结构



服务器组成

|  |  |  |
| --- | --- | --- |
| lvs | 负载均衡调度器 |  |
| nginx | Web服务器 |  |
| tomcat | 应用服务器 |  |
| mongodb | Nosql服务器 |  |
| pgsql | 数据库服务器 |  |
| elasticsearch | 全文检索服务器 |  |
| hadoop | 文件服务器 |  |
| memcached | 缓存服务器 |  |
| pushserver | 推送服务器 |  |
| activemq | 消息服务器 |  |
| ntp | 时间服务器 |  |
| email | 邮件服务器 |  |

1. **整体软件系统架构**
   1. 初始阶段，小型网站

小型网站最开始没有太多人访问，只需要一台服务器。

应用程序、数据库、文件等所有的资源都在一台服务器上。



* 1. 应用服务和数据服务分离

随着网站业务的发展，一台服务逐渐不能满足要求：

越来越多的用户访问导致性能越来越差

越来越多的数据导致存储空间不足

应用和数据分离后整个网站使用三台服务器，它们对硬件资源的要求各不相同：

应用服务器：需要处理大量的业务逻辑，需要更快更强大的CPU

文件服务器：需要快速磁盘检索和数据缓存，需要更快的磁盘和更大的内存

数据库服务器：需要存储大量用户上传的文件，需要更大的硬盘

应用和数据分离后，

不同特征的服务器承担不同的服务角色

网站的并发处理能力和数据存储空间得到了很大改善，支持网站业务进一步发展。



随着用户逐渐增多，数据库压力太大导致访问延迟，影响整个网站的性能，用户体验是受到影响。

* 1. 使用缓存改善网站性能

二八定律：80%的业务访问集中在20%的数据上。

既然大部分的业务访问集中在一小部分数据上，如果把这一小部分数据缓存在内存中，就可以减少数据库的访问压力，提高整个网站的数据访问的速度，改善数据库的写入性能。

分两种缓存

本地缓存：缓存在应用服务器上

远程缓存：缓存在专门的分布式缓存服务器上

本地缓存访问速度更快一些，受应用服务器内存限制，其缓存数据量有限，会出现和应用程序争用内存的情况。

远程分布式缓存，可以使用集群的方式，部署大内存的服务器，可以在理论上做到不受内存容量限制的缓存服务



使用缓存后，数据库访问压力得到有效缓解

单一应用服务器能够处理的请求连接有限，在网站访问高峰期，应用服务器成为整个网站的瓶颈

* 1. 应用服务器集群改善网站的并发处理能力

对大型网站而言，不管多么强大的服务器，都满足不了网站持续增长的业务需求。

通过增加服务器的方式分担原有服务器的访问和存储压力，减轻负载，改善系统性能，实现系统的可伸缩性



通过负载均衡调度服务器，将来自用户浏览器的访问请求分发到应用服务器集群中的任何一台访问器上，应用服务器的负载压力不再成为整个网站的瓶颈。

* 1. 数据库读写分离

在网站的用户达到一定规模后，数据库因为负载压力过高而成为网站的瓶颈。

大部分的主流数据库都提供主从热备功能

应用服务器在写数据的时候，访问主数据库，主数据库通过主从复制机制将数据更新同步到从数据库，当应用服务器读数据的时候，可以通过从数据库获得数据。

通常在应用服务器端使用专门的数据访问模块，使数据库读写分离对应用透明。



* 1. 反向代理和CDN

随着网站业务不断发展，用户规模越来越大，由于中国复杂的网络环境，不同地区的用户访问网站时，速度差别也极大

网站访问延迟和用户流失率正相关，

为了提供更好的用户体验，留住用户，网站需要加速网站访问速度。

主要手段：

CDN和反向代理的基本原理都是缓存。

目的都是尽早返回数据给用户，一方面加快用户访问速度，另一方面也减轻后端服务器的负载压力

CDN：部署在网络提供商（ISP）的机房，用户在请求网站服务时，可以从距离自己最近的网络提供商机房获取数据

反向代理：部署在网站的中心机房，当用户请求到达中心机房后，首先访问的服务器是反向代理服务器，如果反向代理服务器中缓存着用户请求的资源，就将其直接返回给用户。



* 1. 分布式文件系统，数据库拆分

数据库经过读写分离后，随着网站业务的发展依然不能满足需求，需要数据库拆分，将不同业务的数据库部署在不同的物理机器上。

分布式文件系统：

对一些图片、大文本的存储，不适合使用数据库

解决了单机文件存储容量及安全性的问题，可以满足业务系统对文件存储的要求。



* 1. NoSQL和全文检索

关系型数据库问题：

扩展困难：由于存在类似Join这样多表查询机制，使得数据库在扩展方面很艰难;

读写慢：这种情况主要发生在数据量达到一定规模时由于关系型数据库的系统逻辑非常复杂，使得其非常容易发生死锁等并发问题，所以导致其读写速度下滑非常严重;

成本高：企业级数据库的License价格很惊人，并且随着系统的规模，而不断上升;

有限的支撑容量：现有关系型解决方案还无法支撑Google这样海量的数据存储;

NoSQL：

简单的扩展：能通过轻松地添加新的节点来扩展这个集群

快速的读写

低廉的成本

全文检索：

可以管理结构化和非结构化数据

按词或词的组合搜索

搜索速度快：在上亿数据量情况下，检索速度可毫秒级响应。

易扩展：其分布式产品具备横向扩展能力，可通过增加服务器提高系统性能



* 1. 业务拆分

为了应对日益复杂的业务场景，使用分而治之的手段将整个网站业务分成不同的产品线，分归不同的业务团队负责。根据产品线划分，将一个网站拆分成许多不同的应用，通过访问同一个数据存储系统来构成一个关联的完整系统。

* 1. 异步

使用异步消息队列，多个服务器集群通过分布式消息队列实现异步，业务之间的消息传递不是同步调用，而是将一个操作分成多个阶段，每个阶段通过共享数据的方式异步执行进行协作。

好处：

系统解耦

提供系统可用性：消费者故障时，生产者继续发送消息，消息堆积，消费者恢复后继续处理消息

加快网站响应速度：异步执行，生成者处理请求后发送消息至消息队列，无须等待消费者处理。

消除并发访问高峰。

推送服务器和消息队列服务器的功能类似，实时性更高，可靠性不要求非常高，允许消息丢失，重复消息。



1. **项目工程结构**
   1. 特点

(1) 前后端分离

前后端交互：http+json/xml

(2) 后端服务接口

统一参数、统一返回类型

(3) 后端服务访问入口

按应用划分，每个应用一个访问入口

* 1. 应用分层结构

|  |  |
| --- | --- |
| 控制层 | Servlet |
| 数据访问层 | DAO |
| 服务接口层 | Service |

* 1. 应用划分

|  |  |
| --- | --- |
| 电子商务 | ecommerce |
| 社交圈 | sns |
| 系统管理 | system |
| 智能菜谱服务 | ifood |
| 智能设备服务 | deviceinfo |

* 1. 功能划分

|  |  |
| --- | --- |
| tuna\_audit | 审核流程管理 |
| tuna\_biz\_deviceinfo | 智能设备服务接口 |
| tuna\_biz\_ecommerce | 电子商务服务接口 |
| tuna\_biz\_ifood | 智能菜谱服务接口 |
| tuna\_biz\_sns | 社交圈服务接口 |
| tuna\_system | 系统管理服务接口 |
| tuna\_quartz | 定时任务调度 |
| tuna\_cache | 缓存操作类库 |
| tuna\_dal | 数据库操作类库 |
| tuna\_common | 公共类库，常用工具类 |
| tuna\_facade | web servlet类库 |
| tuna\_resource | 文件资源管理 |
| tuna\_permission | 系统权限管理 |
| tuna\_session | 会话管理 |
| tuna\_config | 系统全局静态配置 |
| tuna\_rpc | 远程调用接口 |
| tuna\_elasticsearch | 全文检索操作类库 |
| tuna\_log | 系统接口调用日志输出 |
| tuna\_nosql | MongoDB操作类库 |
| tuna\_push | 推送服务客户端 |
| tuna\_wordfilter | 敏感词过滤 |
| tuna\_lib | 第三方jar包目录 |
| tuna\_test | 测试工程 |

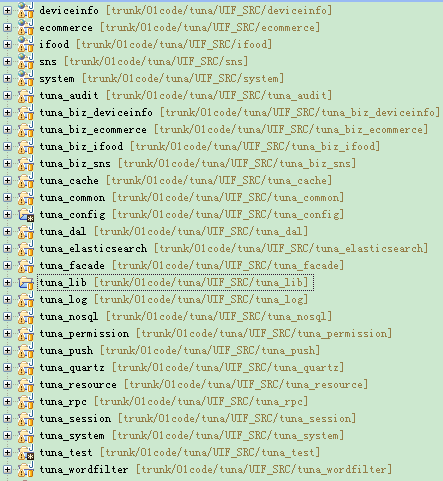
* 1. 项目间依赖

|  |
| --- |
| <projectDescription>  <name>tuna\_audit</name>  <comment></comment>  <projects>  <project>tuna\_cache</project>  <project>tuna\_common</project>  <project>tuna\_dal</project>  <project>tuna\_permission</project>  <project>tuna\_rpc</project>  </projects>  </projectDescription>    <projectDescription>  <name>tuna\_biz\_deviceinfo</name>  <comment></comment>  <projects>  <project>tuna\_cache</project>  <project>tuna\_common</project>  <project>tuna\_dal</project>  <project>tuna\_resource</project>  <project>tuna\_rpc</project>  <project>tuna\_push</project>  </projects>  </projectDescription>    <projectDescription>  <name>tuna\_biz\_ecommerce</name>  <comment></comment>  <projects>  <project>tuna\_cache</project>  <project>tuna\_common</project>  <project>tuna\_dal</project>  <project>tuna\_resource</project>  <project>tuna\_rpc</project>  <project>tuna\_elasticsearch</project>  </projects>  </projectDescription>    <projectDescription>  <name>tuna\_biz\_ifood</name>  <comment></comment>  <projects>  <project>tuna\_cache</project>  <project>tuna\_common</project>  <project>tuna\_dal</project>  <project>tuna\_resource</project>  <project>tuna\_rpc</project>  <project>tuna\_push</project>  </projects>  </projectDescription>    <projectDescription>  <name>tuna\_biz\_sns</name>  <comment></comment>  <projects>  <project>tuna\_cache</project>  <project>tuna\_common</project>  <project>tuna\_dal</project>  <project>tuna\_resource</project>  <project>tuna\_rpc</project>  <project>tuna\_push</project>  <project>tuna\_nosql</project>  </projects>  </projectDescription>    <projectDescription>  <name>tuna\_cache</name>  <comment></comment>  <projects>  <project>tuna\_common</project>  </projects>  </projectDescription>    <projectDescription>  <name>tuna\_common</name>  <comment></comment>  <projects>  </projects>  </projectDescription>    <projectDescription>  <name>tuna\_config</name>  <comment></comment>  <projects>  </projects>  </projectDescription>    <projectDescription>  <name>tuna\_dal</name>  <comment></comment>  <projects>  <project>tuna\_common</project>  <project>tuna\_cache</project>  </projects>  </projectDescription>    <projectDescription>  <name>tuna\_elasticsearch</name>  <comment></comment>  <projects>  <project>tuna\_common</project>  <project>tuna\_cache</project>  <project>tuna\_dal</project>  </projects>  </projectDescription>    <projectDescription>  <name>tuna\_facade</name>  <comment></comment>  <projects>  <project>tuna\_common</project>  <project>tuna\_cache</project>  <project>tuna\_log</project>  <project>tuna\_resource</project>  <project>tuna\_rpc</project>  </projects>  </projectDescription>    <projectDescription>  <name>tuna\_lib</name>  <comment></comment>  <projects>  </projects>  </projectDescription>    <projectDescription>  <name>tuna\_log</name>  <comment></comment>  <projects>  <project>tuna\_common</project>  <project>tuna\_nosql</project>  </projects>  </projectDescription>    <projectDescription>  <name>tuna\_permission</name>  <comment></comment>  <projects>  <project>tuna\_common</project>  <project>tuna\_cache</project>  <project>tuna\_rpc</project>  <project>tuna\_dal</project>  <project>tuna\_resource</project>  </projects>  </projectDescription>    <projectDescription>  <name>tuna\_push</name>  <comment></comment>  <projects>  <project>tuna\_common</project>  <project>tuna\_cache</project>  <project>tuna\_dal</project>  </projects>  </projectDescription>    <projectDescription>  <name>tuna\_quartz</name>  <comment></comment>  <projects>  <project>tuna\_common</project>  <project>tuna\_cache</project>  <project>tuna\_dal</project>  </projects>  </projectDescription>    <projectDescription>  <name>tuna\_resource</name>  <comment></comment>  <projects>  <project>tuna\_common</project>  <project>tuna\_cache</project>  <project>tuna\_dal</project>  <project>tuna\_nosql</project>  </projects>  </projectDescription>    <projectDescription>  <name>tuna\_rpc</name>  <comment></comment>  <projects>  <project>tuna\_common</project>  <project>tuna\_dal</project>  </projects>  </projectDescription>    <projectDescription>  <name>tuna\_session</name>  <comment></comment>  <projects>  <project>tuna\_common</project>  <project>tuna\_cache</project>  <project>tuna\_dal</project>  <project>tuna\_nosql</project>  <project>tuna\_rpc</project>  <project>tuna\_push</project>  </projects>  </projectDescription>    <projectDescription>  <name>tuna\_system</name>  <comment></comment>  <projects>  <project>tuna\_common</project>  <project>tuna\_cache</project>  <project>tuna\_dal</project>  <project>tuna\_resource</project>  <project>tuna\_session</project>  <project>tuna\_permission</project>  <project>tuna\_rpc</project>  <project>tuna\_push</project>  </projects>  </projectDescription>    <projectDescription>  <name>tuna\_wordfilter</name>  <comment></comment>  <projects>  <project>tuna\_common</project>  <project>tuna\_cache</project>  <project>tuna\_dal</project>  </projects>  </projectDescription> |

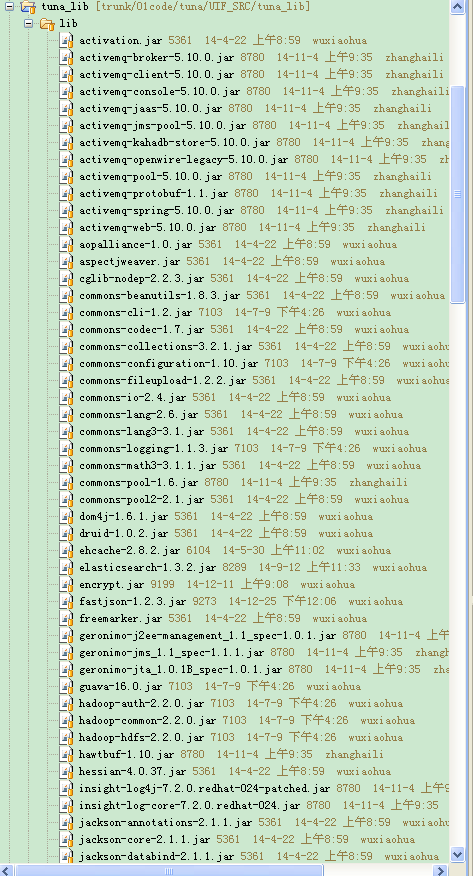
* 1. 应用与项目依赖

|  |
| --- |
| <projectDescription>  <name>deviceinfo</name>  <comment></comment>  <projects>  <project>tuna\_cache</project>  <project>tuna\_dal</project>  <project>tuna\_common</project>  <project>tuna\_facade</project>  <project>tuna\_resource</project>  <project>tuna\_biz\_deviceinfo</project>  <project>tuna\_config</project>  <project>tuna\_rpc</project>  <project>tuna\_log</project>  <project>tuna\_nosql</project>  <project>tuna\_push</project>  </projects>  </projectDescription>  <projectDescription>  <name>ecommerce</name>  <comment></comment>  <projects>  <project>tuna\_cache</project>  <project>tuna\_config</project>  <project>tuna\_dal</project>  <project>tuna\_common</project>  <project>tuna\_facade</project>  <project>tuna\_resource</project>  <project>tuna\_rpc</project>  <project>tuna\_log</project>  <project>tuna\_nosql</project>  <project>tuna\_biz\_ecommerce</project>  <project>tuna\_elasticsearch</project>  <project>tuna\_quartz</project>  </projects>  </projectDescription>  <projectDescription>  <name>ifood</name>  <comment></comment>  <projects>  <project>tuna\_cache</project>  <project>tuna\_config</project>  <project>tuna\_dal</project>  <project>tuna\_common</project>  <project>tuna\_biz\_ifood</project>  <project>tuna\_rpc</project>  <project>tuna\_facade</project>  <project>tuna\_resource</project>  <project>tuna\_nosql</project>  <project>tuna\_log</project>  <project>tuna\_push</project>  </projects>  </projectDescription>  <projectDescription>  <name>sns</name>  <comment></comment>  <projects>  <project>tuna\_cache</project>  <project>tuna\_config</project>  <project>tuna\_dal</project>  <project>tuna\_common</project>  <project>tuna\_rpc</project>  <project>tuna\_facade</project>  <project>tuna\_resource</project>  <project>tuna\_log</project>  <project>tuna\_nosql</project>  <project>tuna\_biz\_sns</project>  <project>tuna\_push</project>  <project>tuna\_elasticsearch</project>  </projects>  </projectDescription>  <projectDescription>  <name>system</name>  <comment></comment>  <projects>  <project>tuna\_cache</project>  <project>tuna\_audit</project>  <project>tuna\_system</project>  <project>tuna\_permission</project>  <project>tuna\_config</project>  <project>tuna\_push</project>  <project>tuna\_dal</project>  <project>tuna\_common</project>  <project>tuna\_facade</project>  <project>tuna\_resource</project>  <project>tuna\_session</project>  <project>tuna\_rpc</project>  <project>tuna\_log</project>  <project>tuna\_nosql</project>  </projects>  </projectDescription>  </deployment> |

项目工程包：



开源组件库：





* 1. 代码风格
     1. 源码包前导标识

cn.com.ut（与公司域名一致）

* + 1. 源码包层次划分（抽取DAO层，与Service层独立）

例如，父级包名：cn.com.ut.biz.user

entities：实体

dao：数据访问接口

dao.impl：数据访问对象

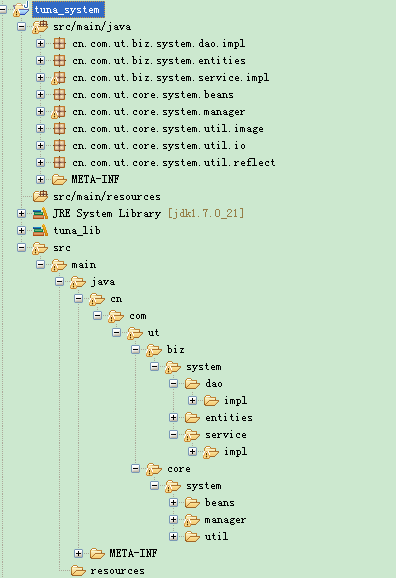
service：业务接口

service.impl：业务实现

beans：非entity类型的java类

manager：对beans的管理

xxx：自定义



* + 1. 源码路径（Maven风格）

Maven风格的约定规则（java开源源码管理事实标准或约定）

1. 主体代码

src/main/java（java程序）:

cn/com/ut/test/Demo.java

src/main/resources（辅助配置，\*.xml，\*.properties）:

cn/com/ut/test/Demo.xml

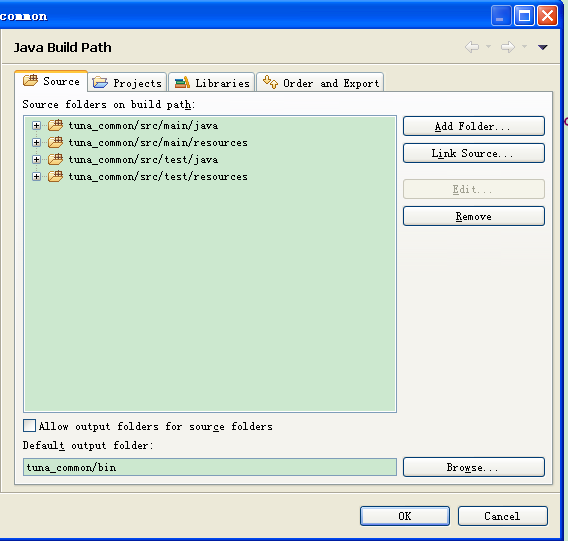
1. 测试代码

src/test/java（java程序）:

cn/com/ut/test/DemoTest.java

src/test/resources（辅助配置，\*.xml，\*.properties）:

cn/com/ut/test/DemoTest.properties



1. **配置模板**

参考《智能餐饮平台JAVA基础服务框架编程指南(2015版)》

1. **系统运行流程**
   1. 请求响应过程



* 1. 统一控制器Servlet（Web层）
     1. 读取请求数据输入流InputStream，转换为原始byte[]
     2. 解析请求头信息、读取Cookie、读取会话封装至User对象
     3. 根据会话对象User对象获取用户接口权限，菜单权限，用户角色
     4. 对原始byte[]进行解析，判断是否加密，如果加密从User对象获取密钥进行解密，返回明文byte[]，如果未加密直接返回
     5. 根据明文byte[]判断请求数据格式，转换为对应的封装对象Data（JSON或XML等）
     6. 解析数据对象Data内容，包括请求的lib、请求的func、请求的参数par，封装至请求参数对象RespResult
     7. 结合User中接口权限验证当前用户是否拥有对lib、func的访问权限，没有权限将直接返回失败
     8. 通过lib、func查找业务对象和业务方法，业务方法参数即RespResult

规则一：由于目前Service对象和方法没有映射配置文件，通过反射调用业务方法须进行约定规则。例如lib为“system/ConfigService”，func为“update”，可以约定ConfigService业务类完全限定名为“cn.com.ut.biz.system.service.impl. ConfigServiceImpl”，业务方法签名为public void update(RespResult respResult).可以发现，红色部分需要由请求而来的lib和func进行替换。

规则二：从Spring Bean容器中查找业务对象，例如“cn.com.ut.biz.system.service.impl.ConfigServiceImpl”业务对象创建时在Spring容器中注册的key值默认为“configServiceImpl”，这个key值可以由请求lib值约定而来，例如lib为“system/ConfigService”，只需要关心lib值中的后半部分“ConfigService”。

规则三：在业务类定义处添加注解和别名，例如



这时Spring会对作此配置的定义类实例化放入容器后，会绑定配置中的别名作为key.

* 1. 业务方法调用（Service层）
     1. 业务方法参数类型RespResult

业务方法参数实例RespResult，由Servlet方法对请求数据封装的RespResult对象直接传递过来。每次请求都会创建一个新的RespResult实例，所以是线程安全的。

* + 1. 从RespResult实例中获取请求数据

1. getDoc()：如果请求数据格式为XML，就获取XML的Document文档对象
2. getJson()：如果请求数据格式为JSON，获取JSON对象
3. getRequest()：获取Servlet方法参数HttpServletRequest实例，由web层传递过来
4. getRequestObjects()：返回Map<String, Object>，获取请求数据中的par部分封装为Map<String, Object>对象
5. getRequestParameters ()：返回Map<String, String>，获取请求数据中的par部分封装为Map<String, String >对象
6. getParameter(Stirng name)：返回Object对象，获取请求数据中的par部分与name名称相同的结点元素文本
7. getVO(Class<T> clazz)：返回T类型的实例对象，封装了请求数据中的par部分
8. getPageBean()：返回PageBean对象，结合RespResult参数实例中绑定的请求数据，解析分页信息、组合查询条件、排序等内容封装至PageBean对象，供DAO层进行分页查询
9. 由于业务方法调用DAO方法时经常需要传递会话数据，例如当前用户ID，所以RespResult实例需要会从web层接收会话数据，根据DAO方法需求选择传递或不传递给DAO方法参数
   * 1. 数据校验

例如，通过getRequestObjects()返回Map<String, Object>对象，其实map中的值初始均为String类型，将Map<String, Object>对象传递给DAO方法参数之前，可能需要进行数据校验和转换。例如，

请求XML原始内容：

|  |
| --- |
| <par>  <func>add</func>  <row>  <username>sunny</username>  <age>20</age>  <row>  <par> |

数据表类型：

|  |
| --- |
| username varchar(50) not null;  age int; |

在插入操作时业务方法需作如下处理：

|  |
| --- |
| Dao dao;  public void add(RespResult respResult) {  Map<String, Object> vo = respResult.getRequestObjects();  String age = (String) vo.get(“age”);  vo.put(“age”, Integer.parseInt(age));  } |

* + 1. 前端有组合查询时，且存在非String类型的查询字段时，例如int类型或日期类型等。

例如，请求原始XML：

|  |
| --- |
| <req>  <call>  <lib>system/DictTypeService</lib>  <func>find</func>  </call>  <par page=*"2"* pagesize=*"2"* total=*"0"*>  <where exp=*"1 # 2 # 3"*>  <field name=*"DICTTYPE\_CODE"* value=*"CorpTrade"* op=*"eq"* id=*"1"* />  <field name=*"IS\_USE"* value=*"Y"* op=*"eq"* id=*"2"* />  <field name=*"SORT\_NUM"* value=*"-1"* op=*"ne"* id=*"3"* />  </where>  <orderby>  <field name=*"SORT\_NUM"* type=*"1"* />  </orderby>  </par>  </req> |

以上解析为sql where子句为 where DICTTYPE\_CODE = ？or IS\_USE = ? or SORT\_NUM = ?

这里假设表结构为：

|  |
| --- |
| DICTTYPE\_CODE varchar(50),  IS\_USE char(1),  SORT\_NUM int2 |

可见SORT\_NUM为整型，由于组合查询sql中where子句和参数值数组是动态的，由系统自动完成，所有在构建where子句之前需要将非String类型的字段名称和实际类型告知给sql构建者。

可以这样处理:

|  |
| --- |
| **public** **void** find(RespResult respResult) {  respResult.setFieldType("*SORT\_NUM*@@int");  List<Map<String, Object>> voList = dao.query(respResult.getPageBean());  respResult.setItemList(voList);  } |

由respResult.setFieldType("*SORT\_NUM*@@int")这句代码完成设置，真正构建sql的时机是在调用respResult.getPageBean()的时候，交由分页类PageBean完成。

|  |
| --- |
| **public** PageBean getPageBean() {  // 解析请求查询参数并设置到分页实例  pageBean.XML2SQL(doc, readFieldType(fieldType));  **return** pageBean;  } |

还有就是，“*SORT\_NUM*@@int,TIME@data”这种形式是沿用上一版的风格，之所以这里需要手动设置，原因很简单，因为这一版去掉了上一版中每个Serveice定义的XML配置文件，上一版可以从配置中自动读取而异。

* + 1. 手动设置数据字典，在查询结果返回时方便进行显示替换。

和上一节的情形一致，由于少了配置文件，所以需要手动配置。

例如：

|  |
| --- |
| **public** **void** find(RespResult respResult) { respResult.getDictionary().put("*IS\_USE*", "IS\_TRUE");  respResult.getDictionary().put("*IS\_DEL*", "IS\_TRUE");  List<Map<String, Object>> voList = dao.query(respResult.getPageBean());  respResult.setItemList(voList);  } |

respResult.getDictionary()返回Map<String,String>对象dictionary，dictionary对象在respResult实例化完成初始化为空集合Map。Map中key表示须要转换的字段，就是在查询列中包含的字段，Map中的value表示转换该字段显示值使用的数据字典名称。

* + 1. 方法调用返回类型

前面已经提及，业务方法调用默认返回void类型，是为了让程序更灵活，简化业务方法实现。数据的具体返回格式由web层入口servlet完成。

例如：

|  |
| --- |
| **public** **void** create(RespResult respResult) {  Map<String, Object> vo = respResult.getRequestObjects();  String sort = (String) vo.get(DictType.*SORT\_NUM*);  **if** (sort != **null**)  vo.put(DictType.*SORT\_NUM*, Integer.*parseInt*(sort));  **boolean** b = dao.checkUnique(**new** Object[] { vo.get(DictType.*DICTTYPE\_CODE*) }, **null**, **null**);  **if** (!b) {  respResult.addErrMsg("字典类型已存在！");  **return**;  }  dao.add(vo);  } |

可见，当验证失败时，直接return就行。

web层入口servlet完成要完成返回数据格式的构建，其实是调用了Result2Dom中的静态方法。

|  |
| --- |
| public static Document toDomContent(RespResult respResult) |

可见，RespResult实例仍然是业务执行结果数据的载体，RespResult中设置数据相关的方法。

设置和获取多个对象，对象类型Map：

|  |
| --- |
| **public** List<Map<String, Object>> getItemList() {  **return** itemList;  }  **public** **void** setItemList(List<Map<String, Object>> itemList) {  **this**.itemList = itemList;  } |

设置和获取单个对象，对象类型Map：

|  |
| --- |
| **public** Map<String, Object> getItem() {  **return** item;  }  **public** **void** setItem(Map<String, Object> item) {  **this**.item = item;  } |

设置和获取单个对象，对象类型Object：

|  |
| --- |
| **public** Object getValueObject() {  **return** valueObject;  }  **public** **void** setValueObject(Object valueObject) {  **this**.valueObject = valueObject;  } |

设置和获取多个对象，对象类型为泛型

|  |
| --- |
| **public** List<?> getVoList() {  **return** voList;  }  **public** **void** setVoList(List<?> voList) {  **this**.voList = voList;  } |

设置数据方法一般是装配DAO方法的返回结果。例如

|  |
| --- |
| **public** **void** find(RespResult respResult) {  List<Map<String, Object>> voList = dao.query(respResult.getPageBean());  respResult.setItemList(voList);  } |

* + 1. 生成树型结构的XML

|  |
| --- |
| /\*\*  \* 根据水平表结构中拥有父子关系的数据生成XML文档中的父子Element,调用{@link RespResult#asXML() asXML}生成  \* {@link org.dom4j.Document Document}  \*  \* **@param** list  \* 水平表中的行  \* **@param** parentIdField  \* 父字段名  \* **@param** idField  \* 子字段名  \* **@param** nodeName  \* XML中Element名称  \* **@param** successMsg  \* 成功提示信息  \* **@param** failureErr  \* 失败提示信息  \*/  **public** **void** createTree(List<Map<String, Object>> list, String parentIdField, String idField,  String nodeName, String successMsg, String failureErr) |

* 1. XML、JSON格式转换

JsonUtil：

|  |
| --- |
| Document doc = **null**;  doc = XmlUtil.*readXml*("test.xml");  Element element = doc.getRootElement();  JSONObject parentObject = **new** JSONObject();  JsonUtil.*xmlToJsonArray*(element, parentObject);  String str = JSON.*toJSONString*(parentObject, **true**);  System.*out*.println(str); |

Xml：

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <contacts idx=*"00"* px=*"11"*>  <contact id=*"1-1-1"*>  <name last=*"wu"*>wuxiaohua</name>  <phone area=*"999"*>123-123</phone>  <address>  <street>123 JFKStreet</street>  <city>123 Any Town</city>  <state>123 Any State</state>  <zipCode>123-123-123</zipCode>  </address>  </contact>  <contact id=*"2-2-2"*>  <name last=*"lee"*>John Doe</name>  <phone>456-456</phone>  <address idx=*"c-c"*>  <street>456 JFKStreet</street>  <city>456 Any Town</city>  <state>456 Any State</state>  <zipCode>456-456-456</zipCode>  </address>  <address idx=*"d-d"*>  <street>789 JFKStreet</street>  <city>789 Any Town</city>  <state>789 Any State</state>  <zipCode>789-789-789</zipCode>  </address>  </contact>  </contacts> |

Json：

|  |
| --- |
| {  "contacts":{  "@attributes":{  "idx":"00",  "px":"11"  },  "contact":[  {  "@attributes":{  "id":"1-1-1"  },  "address":{  "city":"123 Any Town",  "state":"123 Any State",  "street":"123 JFKStreet",  "zipCode":"123-123-123"  },  "name":{  "@attributes":{  "last":"wu"  },  "@text":"wuxiaohua"  },  "phone":{  "@attributes":{  "area":"999"  },  "@text":"123-123"  }  },  {  "@attributes":{  "id":"2-2-2"  },  "address":[  {  "@attributes":{  "idx":"c-c"  },  "city":"456 Any Town",  "state":"456 Any State",  "street":"456 JFKStreet",  "zipCode":"456-456-456"  },  {  "@attributes":{  "idx":"d-d"  },  "city":"789 Any Town",  "state":"789 Any State",  "street":"789 JFKStreet",  "zipCode":"789-789-789"  }  ],  "name":{  "@attributes":{  "last":"lee"  },  "@text":"John Doe"  },  "phone":"456-456"  }  ]  }  } |

说明：Element中包含属性和文本时，json中须借助约定固定key名称完成，"@attributes"对应属性值，"@text"对应文本。例如：

<name last=*"wu"*>wuxiaohua</name>表示为：

|  |
| --- |
| "name":{  "@attributes":{  "last":"wu"  },  "@text":"wuxiaohua"  } |

* + 1. 大小写的问题

可能有时候需要显示大写或小写XML

|  |
| --- |
| <cont>  <result>success<result>  <cont> |

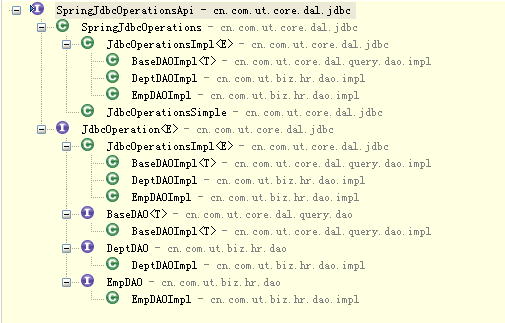
|  |
| --- |
| <cont>  <RESULT>success< RESULT >  <cont> |

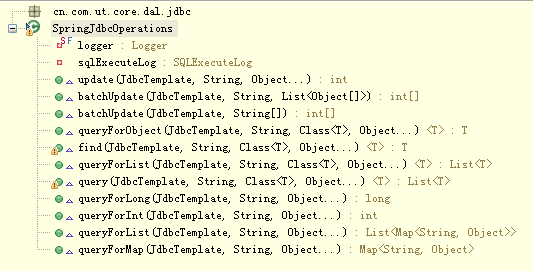
只需要设置RespResult实例的一个属性

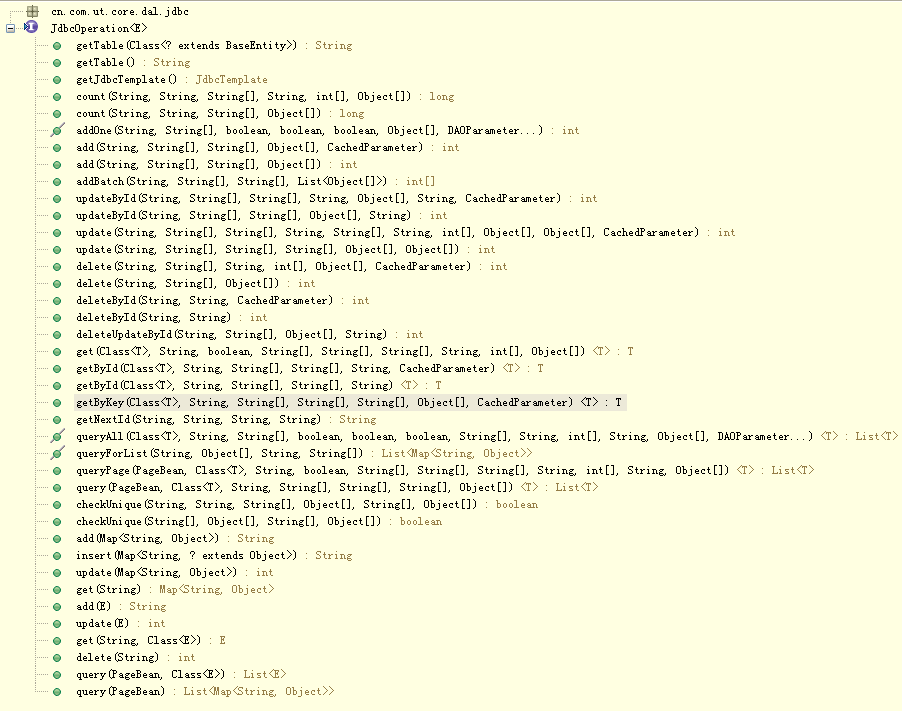
|  |
| --- |
| **public** **void** setUpperLowerCase(EnumConstant.UpperLowerCase upperLowerCase) {  **this**.upperLowerCase = upperLowerCase;  } |

EnumConstant.UpperLowerCase为枚举类型，值为大写小写或维持不变。

* 1. DAO层次结构







* 1. DAO层主要API

1. 更新操作

|  |
| --- |
| /\*\*  \* 更新操作:update table set... where...  \*  \* **@param** table  \* 表名称  \* **@param** updateColumnArray  \* 更新字段(数组形式)  \* **@param** updateColumnJoin  \* 更新字段(文本形式)  \* **@param** useTimestamp  \* 是否更新UPDATE\_TIME字段  \* **@param** useUpdateId  \* 是否更新UPDATE\_ID字段  \* **@param** whereColumnArray  \* where条件(数组形式),如new String["field1",  \* "field2"]将转成sql中where子句的一部分,field1=? and field2=?  \* **@param** whereColumnJoin  \* where条件(文本形式),如field1=? and (field2=1 or field3=?). 这种方式比较灵活  \* **@param** replaceArray  \* 将sql语句中出现的若干{IN}依次替换成replaceArray[x]个数的IN(?,?,...)形式  \* **@param** updateParameterArray  \* 与sql语句中?出现次数匹配的对象数组即参数,这里为更新字段对应的参数数组  \* **@param** whereParameterArray  \* 与sql语句中?出现次数匹配的对象数组即参数，这里为where子句中限定字段对应的参数数组  \* **@param** cachedParameter  \* 缓存参数  \* **@return**  \*/  **public** **int** update(String table, String[] updateColumnArray, String[] appendColumns,  String updateColumnJoin, String[] whereColumnArray, String whereColumnJoin,  **int**[] replaceArray, Object[] updateParameterArray, Object[] whereParameterArray,  CachedParameter cachedParameter) |

1. 根据Id更新一条记录

|  |
| --- |
| /\*\*  \* 按照主键id更新一条记录:update table set... where id=?  \*  \* **@see** JdbcOperationsImpl#update(String, String[], String[], String,  \* String[], String, int[], Object[], Object[], CachedParameter)  \*  \*/  **public** **int** updateById(String table, String[] updateColumnArray, String[] appendColumns,  String updateColumnJoin, Object[] updateParameterArray, String id,  CachedParameter cachedParameter) |

1. 插入一条记录

|  |
| --- |
| /\*\*  \* 插入一条记录  \*  \* **@param** table  \* **@param** insertColumnArray  \* 添加字段  \* **@param** useId  \* 是否添加id  \* **@param** useTimestamp  \* 是否添加创建、更新时间字段  \* **@param** useCreator  \* 是否添加创建者、更新者  \* **@param** parameterArray 参数数组  \* **@param** cachedParameter 缓存参数  \* **@return**  \*/  **public** **int** add(String table, String[] insertColumnArray, String[] appendColumns,  Object[] parameterArray, CachedParameter cachedParameter) {  } |

1. 插入或更新时唯一性约束的验证

|  |
| --- |
| /\*\*  \* 插入或更新时唯一性约束的验证  \*  \* **@param** table  \* 表名称  \* **@param** countField  \* count字段名  \* **@param** uniqueFieldNames  \* 联合唯一字段名数组  \* **@param** uniqueFieldValues  \* 联合唯一字段值数组  \* **@param** pkFieldNames  \* 联合主键名数组，只在更新时用到  \* **@param** pkFieldValues  \* 联合主键值数组，只在更新时用到  \* **@return**  \*/  **protected** **boolean** checkUnique(String table, String countField, String[] uniqueFieldNames,  Object[] uniqueFieldValues, String[] pkFieldNames, Object[] pkFieldValues) |

1. 计数

|  |
| --- |
| /\*\*  \* 计数:select count(\*) from table where...  \*  \* **@param** table  \* 表名称  \* **@param** countColumn  \* 列名称  \* **@param** whereColumnArray  \* where条件(数组形式),如new String["field1",  \* "field2"]将转成sql中where子句的一部分,field1=? and field2=?  \* **@param** whereColumnJoin  \* where条件(文本形式),如field1=? and (field2=1 or field3=?). 这种方式比较灵活  \* **@param** replaceArray  \* 将sql语句中出现的若干{IN}依次替换成replaceArray[x]个数的IN(?,?,...)形式  \* **@param** parameterArray  \* 与sql语句中?出现次数匹配的对象数组即参数  \* **@return**  \*/  **public** **long** count(String table, String countColumn, String[] whereColumnArray,  String whereColumnJoin, **int**[] replaceArray, Object[] parameterArray) |

1. 删除操作

|  |
| --- |
| /\*\*  \* 删除操作  \*  \* **@param** table  \* **@param** whereColumnArray  \* **@param** whereColumnJoin  \* **@param** replaceArray  \* **@param** parameterArray  \* **@return**  \*/  **public** **int** delete(String table, String[] whereColumnArray, String whereColumnJoin,  **int**[] replaceArray, Object[] parameterArray, CachedParameter cachedParameter) |

1. 根据ID删除一条记录

|  |
| --- |
| /\*\*  \* 按照主键id删除一条记录  \*  \* **@see** JdbcOperationsImpl#delete(String, String[], String, int[], Object[])  \* **@param** id  \* 主键id值  \*/  **public** **int** deleteById(String table, String id, CachedParameter cachedParameter) |

1. 查询单条实例

|  |
| --- |
| /\*\*  \* 查询单条实例  \*  \* **@param** clazz  \* 实例类型  \* **@param** table  \* **@param** selectColumnArray  \* 查询字段字符数组  \* **@param** useId  \* 是否查询id字段  \* **@param** useTimestamp  \* 是否查询create\_time,update\_time系统字段  \* **@param** useCreator  \* 是否查询create\_id,update\_id系统字段  \* **@param** whereColumnArray  \* **@param** whereColumnJoin  \* **@param** replaceArray  \* **@param** parameterArray  \* **@return**  \*/  **public** <T> T get(Class<T> clazz, String table, String[] selectColumnArray,  String[] appendColumns, String[] whereColumnArray, String whereColumnJoin,  **int**[] replaceArray, Object[] parameterArray) |

1. 根据ID查询单条实例

|  |
| --- |
| /\*\*  \* 按照主键id查询单条实例  \*  \* **@see** JdbcOperationsImpl#get(Class, String, String[], String[], String[],  \* String, int[], Object[])  \*  \*/  **public** <T> T getById(Class<T> clazz, String table, String[] selectColumnArray,  String[] appendColumn, String id, CachedParameter cachedParameter) |

1. 查询操作，可以分页，可接收动态查询条件

|  |
| --- |
| /\*\*  \* 查询操作，可以分页，可接收动态查询条件  \*  \* **@param** sqlBuilder  \* 分页和动态查询条件的封装类  \* **@param** clazz  \* **@param** table  \* **@param** selectColumnArray  \* **@param** useId  \* **@param** useTimestamp  \* **@param** useCreator  \* **@param** whereColumnArray  \* **@param** whereColumnJoin  \* **@param** replaceArray  \* **@param** orderBy  \* 排序字段  \* **@param** parameterArray  \* **@return**  \*/  **public** <T> List<T> queryPage(PageBean pageBean, Class<T> clazz, String table,  String[] selectColumnArray, String[] appendColumns, String[] whereColumnArray,  String whereColumnJoin, **int**[] replaceArray, String orderBy, Object[] parameterArray) |

1. 查找树型结构当前父节点下子节点id的最大值，生成下一个子结点id的值

|  |
| --- |
| /\*\*  \* 查找树型结构当前父节点下子节点id的最大值，生成下一个子结点id的值  \*  \* **@param** parentId  \* 父节点的值  \* **@param** table  \* 表名称  \* **@param** parentIdColumn  \* 父节点字段名  \* **@param** idColumn  \* 子节点字段名  \* **@return**  \*/  **protected** **synchronized** String getNextId(String parentId, String table, String parentIdColumn,  String idColumn) |

* 1. 元数据自动生成Java类定义

每张数据表对应一个实体类定义，这里提供一个工具类MetaDataToPojo，作用是根据表结构的元数据生成实体类文件。

例如：

表t\_sys\_config

|  |
| --- |
| CREATE TABLE "comm\_global"."t\_sys\_config" (  "id" varchar(32) NOT NULL,  "config\_code" varchar(32) NOT NULL,  "config\_type" varchar(32) NOT NULL,  "config\_name" varchar(32) NOT NULL,  "config\_value" varchar(256) NOT NULL,  "config\_des" varchar(512),  "create\_id" varchar(32),  "create\_time" timestamp(6),  "update\_id" varchar(32),  "update\_time" timestamp(6),  "remark" varchar(128),  "remark1" varchar(128),  "remark2" varchar(128),  "remark3" varchar(128),  CONSTRAINT "pk\_t\_sys\_config" PRIMARY KEY ("id"),  CONSTRAINT "ak\_uk\_ts\_cfg\_ccode\_t\_sys\_co" UNIQUE ("config\_code")  ) |

实体类Config（生成方式1），根据表字段名称和类型生成实体类的成员属性、getter和setter方法。

|  |
| --- |
| **package** cn.com.ut.biz.system.entities;  **import** cn.com.ut.core.dal.jdbc.BaseEntity;  **public** **class** Config **extends** BaseEntity {  **private** **static** **final** **long** *serialVersionUID* = 5441771982121209080L;  **private** String configCode;  **private** String configType;  **private** String configName;  **private** String configValue;  **private** String configDes;  // getter和setter方法省略 |

实体类Config（生成方式2），根据表字段名称生成实体类的静态成员常量。

|  |
| --- |
| **package** cn.com.ut.biz.system.entities;  **import** cn.com.ut.core.dal.jdbc.BaseEntity;  **public** **class** Config **extends** BaseEntity {  **private** **static** **final** **long** *serialVersionUID* = 5441771982121209080L;  **public** **static** **final** String *CONFIG\_CODE* = "CONFIG\_CODE";  **public** **static** **final** String *CONFIG\_TYPE* = "CONFIG\_TYPE";  **public** **static** **final** String *CONFIG\_NAME* = "CONFIG\_NAME";  **public** **static** **final** String *CONFIG\_VALUE* = "CONFIG\_VALUE";  **public** **static** **final** String *CONFIG\_DES* = "CONFIG\_DES";  } |

以上两种生成实体类的方式的适用场景将在“参考示例”章节详细介绍。

* 1. 动态SQL构建器

实现DAO方法时，难免会碰到一些稍微复杂的SQL语句，例如多表join连接、分组、where条件需根据一定条件动态生成的时候。面对这些情况。最简单直接的办法就是拼接字符串来构建一条sql语句，不过这样可维护性就不高，不够灵活。借鉴MyBatis中动态构建sql语句的方法，可以这样做。

目标SQL：

|  |
| --- |
| SELECT P.ID, P.USERNAME, P.PASSWORD, P.FULL\_NAME, P.LAST\_NAME, P.CREATED\_ON, P.UPDATED\_ON  FROM PERSON P, ACCOUNT A  INNER JOIN DEPARTMENT D on D.ID = P.DEPARTMENT\_ID  INNER JOIN COMPANY C on D.COMPANY\_ID = C.ID  WHERE (P.ID = A.ID AND P.FIRST\_NAME like ?)  OR (P.LAST\_NAME like ?)  GROUP BY P.ID  HAVING (P.LAST\_NAME like ?)  OR (P.FIRST\_NAME like ?)  ORDER BY P.ID, P.FULL\_NAME |

使用SQL帮助类：

|  |
| --- |
| **private** String selectSqlTest() {  **return** **new** SQL() {  {  SELECT("P.ID, P.USERNAME, P.PASSWORD, P.FULL\_NAME");  SELECT("P.LAST\_NAME, P.CREATED\_ON, P.UPDATED\_ON");  FROM("PERSON P");  FROM("ACCOUNT A");  INNER\_JOIN("DEPARTMENT D on D.ID = P.DEPARTMENT\_ID");  INNER\_JOIN("COMPANY C on D.COMPANY\_ID = C.ID");  WHERE("P.ID = A.ID");  WHERE("P.FIRST\_NAME like ?");  OR();  WHERE("P.LAST\_NAME like ?");  GROUP\_BY("P.ID");  HAVING("P.LAST\_NAME like ?");  OR();  HAVING("P.FIRST\_NAME like ?");  ORDER\_BY("P.ID");  ORDER\_BY("P.FULL\_NAME");  }  }.toString();  } |

* 1. DAO方法签名标识读或写

读操作：DAO方法名称开头可以是

|  |
| --- |
| add、get、count、list、count、query |

写操作：DAO方法名称开头可以是

|  |
| --- |
| update、delete |

第三种情况，非以上命名的方法上如果使用了cn.com.ut.core.dal.annotation.DAOMethod注解。注解中的isWriteable()标识读或写方法。

|  |
| --- |
| **package** cn.com.ut.core.dal.annotation;  **import** java.lang.annotation.ElementType;  **import** java.lang.annotation.Retention;  **import** java.lang.annotation.RetentionPolicy;  **import** java.lang.annotation.Target;  @Retention(RetentionPolicy.*RUNTIME*)  @Target(value = { ElementType.*TYPE*, ElementType.*METHOD* })  **public** **@interface** DAOMethod {  String group() **default** "";    **boolean** isWriteable() **default** **false**;  } |

* 1. 读写分离和缓存演示
     1. 对象关系映射ormapper.xml：

表t\_user所在的逻辑组test\_group\_1

|  |
| --- |
| <schema name=*"test\_group\_1"* shards=*"test1,test2"*>  <ormapper name=*"cn.com.ut.biz.user.entities.UserInfo"* table=*"t\_user"*/>  <ormapper name=*"cn.com.ut.biz.user.entities.Goods"* table=*"t\_goods"*/>  </schema> |

* + 1. DataSourceGroup配置：

逻辑组test\_group\_1有test1,test2两DataSourceShard

|  |
| --- |
| <bean id=*"testGroup1"* class=*"cn.com.ut.core.dal.datasource.DataSourceGroup"*>  <property name=*"groupName"* value=*"test\_group\_1"* />  <property name=*"shards"*>  <util:map map-class=*"java.util.LinkedHashMap"*>  <entry>  <key><value>test1</value></key>  <ref bean=*"test1"*/>  </entry>  <entry>  <key><value>test2</value></key>  <ref bean=*"test2"*/>  </entry>  </util:map>  </property>  </bean> |

* + 1. DataSourceShard配置：

名为test1的DataSourceShard包含一Master数据源t1m和两Slave数据源t1s1、t1s2。

名为test2的DataSourceShard包含一Master数据源t2m和两Slave数据源t2s1、t2s2。

|  |
| --- |
| <bean id=*"test1"* class=*"cn.com.ut.core.dal.datasource.DataSourceShard"*>  <property name=*"shardName"* value=*"test1"* />  <property name=*"master"*>  <bean class=*"cn.com.ut.core.dal.datasource.DataSourceWrapper"*>  <property name=*"dataSourceName"* value=*"t1m"* />  <property name=*"jdbcTemplate"* ref=*"t1mJt"* />  </bean>  </property>  <property name=*"slaves"*>  <util:list list-class=*"java.util.ArrayList"*>  <bean class=*"cn.com.ut.core.dal.datasource.DataSourceWrapper"*>  <property name=*"dataSourceName"* value=*"t1s1"* />  <property name=*"jdbcTemplate"* ref=*"t1s1Jt"* />  <property name=*"weight"* value=*"2"* />  </bean>  <bean class=*"cn.com.ut.core.dal.datasource.DataSourceWrapper"*>  <property name=*"dataSourceName"* value=*"t1s2"* />  <property name=*"jdbcTemplate"* ref=*"t1s2Jt"* />  <property name=*"weight"* value=*"1"* />  </bean>  </util:list>  </property>  </bean>  <bean id=*"test2"* class=*"cn.com.ut.core.dal.datasource.DataSourceShard"*>  <property name=*"shardName"* value=*"test2"* />  <property name=*"master"*>  <bean class=*"cn.com.ut.core.dal.datasource.DataSourceWrapper"*>  <property name=*"dataSourceName"* value=*"t2m"* />  <property name=*"jdbcTemplate"* ref=*"t2mJt"* />  </bean>  </property>  <property name=*"slaves"*>  <util:list list-class=*"java.util.ArrayList"*>  <bean class=*"cn.com.ut.core.dal.datasource.DataSourceWrapper"*>  <property name=*"dataSourceName"* value=*"t2s1"* />  <property name=*"jdbcTemplate"* ref=*"t2s1Jt"* />  <property name=*"weight"* value=*"1"* />  </bean>  <bean class=*"cn.com.ut.core.dal.datasource.DataSourceWrapper"*>  <property name=*"dataSourceName"* value=*"t2s2"* />  <property name=*"jdbcTemplate"* ref=*"t2s2Jt"* />  <property name=*"weight"* value=*"2"* />  </bean>  </util:list>  </property>  </bean> |

* + 1. 测试程序

|  |
| --- |
| @Test  **public** **void** testService() {  UserInfoDAOImpl dao = *ctx*.getBean(UserInfoDAOImpl.**class**);  UserInfo user = **new** UserInfo();  user.setId("007");  user.setUsername("n007");  user.setPassword("p007");  user.setEmail("e007@ut.com");  String opUserId = "wuxiaohua";  Timestamp time = DateTimeUtil.*currentDateTime*();  *logger*.debug("未开启缓存");  *logger*.debug("插入开始前，查询一条记录id=001，第1次（开始）");  dao.getById(UserInfo.**class**, **null**, **null**, **null**, "001");  *logger*.debug("插入开始前，查询一条记录id=001，第1次（结束）");    *logger*.debug("插入开始前，查询一条记录id=001，第2次（开始）");  dao.getById(UserInfo.**class**, **null**, **null**, **null**, "001");  *logger*.debug("插入开始前，查询一条记录id=001，第2次（结束）");  *logger*.debug("插入一条记录id = 007（开始）");  dao.add(**null**,  **new** String[] { "id", "username", "password", "email" },  **new** String[] { "create\_time", "create\_id" },  **new** Object[] { user.getId(), user.getUsername(), user.getPassword(),  user.getEmail(), time, opUserId });  *logger*.debug("插入一条记录id = 007（结束）");  *logger*.debug("插入结束后，查询一条记录id=007，第1次（开始）");  dao.getById(UserInfo.**class**, **null**, **null**, **null**, user.getId());  *logger*.debug("插入结束后，查询一条记录id=007，第1次（结束）");  *logger*.debug("插入结束后，查询一条记录id=007，第2次（开始）");  dao.getById(UserInfo.**class**, **null**, **null**, **null**, user.getId());  *logger*.debug("插入结束后，查询一条记录id=007，第2次（结束）");  *logger*.debug("更新一条记录id = 007（开始）");  dao.updateById(**null**, **new** String[] { "email" }, **new** String[] { "update\_time", "update\_id" },  **new** Object[] {"e007-1@ut.com",DateTimeUtil.*currentDateTime*(),"wu2"}, user.getId());  *logger*.debug("更新一条记录id = 007（结束）");  *logger*.debug("更新结束后，查询一条记录id=007，第1次（开始）");  dao.getById(UserInfo.**class**, **null**, **null**, **null**, user.getId());  *logger*.debug("更新结束后，查询一条记录id=007，第1次（结束）");  *logger*.debug("更新结束后，查询一条记录id=007，第2次（开始）");  dao.getById(UserInfo.**class**, **null**, **null**, **null**, user.getId());  *logger*.debug("更新结束后，查询一条记录id=007，第2次（结束）");  *logger*.debug("删除一条记录id=007（开始）");  dao.deleteById(**null**, user.getId());  *logger*.debug("删除一条记录id=007（结束）");  } |

* + 1. 运行结果（未开启缓存）

|  |
| --- |
| 未开启缓存  插入开始前，查询一条记录id=001，第1次（开始）  DataSource==t1s1  插入开始前，查询一条记录id=001，第1次（结束）  插入开始前，查询一条记录id=001，第2次（开始）  DataSource==t1s2  插入开始前，查询一条记录id=001，第2次（结束）  插入一条记录id = 007（开始）  DataSource==t1m  插入一条记录id = 007（结束）  插入结束后，查询一条记录id=007，第1次（开始）  DataSource==t1m  插入结束后，查询一条记录id=007，第1次（结束）  插入结束后，查询一条记录id=007，第2次（开始）  DataSource==t1m  插入结束后，查询一条记录id=007，第2次（结束）  更新一条记录id = 007（开始）  DataSource==t1m  更新一条记录id = 007（结束）  更新结束后，查询一条记录id=007，第1次（开始）  DataSource==t1m  更新结束后，查询一条记录id=007，第1次（结束）  更新结束后，查询一条记录id=007，第2次（开始）  DataSource==t1m  更新结束后，查询一条记录id=007，第2次（结束）  删除一条记录id=007（开始）  DataSource==t1m  删除一条记录id=007（结束） |

* + 1. 运行结果（未开启缓存）

|  |
| --- |
| 开启缓存  插入开始前，查询一条记录id=001，第1次（开始）  get data from cache by key = t\_user/001  DataSource==t1s1  set data to cache by key = t\_user/001  插入开始前，查询一条记录id=001，第1次（结束）  插入开始前，查询一条记录id=001，第2次（开始）  get data from cache by key = t\_user/001  插入开始前，查询一条记录id=001，第2次（结束）  插入一条记录id = 007（开始）  DataSource==t1m  插入一条记录id = 007（结束）  插入结束后，查询一条记录id=007，第1次（开始）  get data from cache by key = t\_user/007  DataSource==t1m  set data to cache by key = t\_user/007  插入结束后，查询一条记录id=007，第1次（结束）  插入结束后，查询一条记录id=007，第2次（开始）  get data from cache by key = t\_user/007  插入结束后，查询一条记录id=007，第2次（结束）  更新一条记录id = 007（开始）  DataSource==t1m  delete data from cache by key = t\_user/007  更新一条记录id = 007（结束）  更新结束后，查询一条记录id=007，第1次（开始）  get data from cache by key = t\_user/007  DataSource==t1m  set data to cache by key = t\_user/007  更新结束后，查询一条记录id=007，第1次（结束）  更新结束后，查询一条记录id=007，第2次（开始）  get data from cache by key = t\_user/007  更新结束后，查询一条记录id=007，第2次（结束）  删除一条记录id=007（开始）  DataSource==t1m  delete data from cache by key = t\_user/007  删除一条记录id=007（结束） |

* 1. 简单分片演示

继续上一小节的内容，逻辑组test\_group\_1有test1,test2两DataSourceShard，名为test1的DataSourceShard包含一Master数据源t1m和两Slave数据源t1s1、t1s2。

名为test2的DataSourceShard包含一Master数据源t2m和两Slave数据源t2s1、t2s2。

* + 1. 数据切分规则

数据表t\_bas\_user以id字段作为片键，先对id值作hashCode()操作得到一整数，该整数若为偶数则使用名为test2的DataSourceShard，若为奇数则使用名为test1的DataSourceShard。

简单起见，分片规则逻辑放在与数据表对应的实体类中定义，例如：

|  |
| --- |
| **public** ShardParameter getShardParameter(String action) {  **if** ("one".equals(action)) {  **if** (getId() == **null**)  **return** **null**;  **int** result = getId().hashCode() % 2;  String shardName = result == 0 ? "test2" : "test1";  **return** **new** ShardParameter(**new** String[] { shardName });  }  **if** ("many".equals(action)) {  **return** **new** ShardParameter(**new** String[] { "test1", "test2" });  }  **return** **null**;  } |

* + 1. 插入操作（更新、删除操作类似）

数据准备：

|  |
| --- |
| UserInfo user = **new** UserInfo();  user.setId("0025");  user.setUsername("n0025");  user.setPassword("p0025");  user.setEmail("e0025@ut.com");  UserInfo user2 = **new** UserInfo();  user2.setId("0026");  user2.setUsername("n0026");  user2.setPassword("p0026");  user2.setEmail("e0026@ut.com"); |

测试程序

|  |
| --- |
| @Test  **public** **void** testShard() {  UserInfoDAOImpl dao = *ctx*.getBean(UserInfoDAOImpl.**class**);  UserInfo user = **new** UserInfo();  user.setId("0025");  user.setUsername("n0025");  user.setPassword("p0025");  user.setEmail("e0025@ut.com");  UserInfo user2 = **new** UserInfo();  user2.setId("0026");  user2.setUsername("n0026");  user2.setPassword("p0026");  user2.setEmail("e0026@ut.com");  ShardParameter sp = user.getShardParameter("one");  ShardParameter sp2 = user2.getShardParameter("one");  *logger*.debug("插入记录id=0025（开始）");  dao.addOne(  **null**,  **new** String[] { "id", "username", "password", "email" },  **false**,  **false**,  **false**,  **new** Object[] { user.getId(), user.getUsername(), user.getPassword(),  user.getEmail() }, sp);  *logger*.debug("插入记录id=0025（结束）");    *logger*.debug("插入记录id=0026（开始）");  dao.addOne(  **null**,  **new** String[] { "id", "username", "password", "email" },  **false**,  **false**,  **false**,  **new** Object[] { user2.getId(), user2.getUsername(), user2.getPassword(),  user2.getEmail() }, sp2);  *logger*.debug("插入记录id=0026（结束）");  } |

运行结果

|  |
| --- |
| 插入记录id=0025（开始）  groupName===test\_group\_1  ShardName==test1  DataSource==t1m  插入记录id=0025（结束）  插入记录id=0026（开始）  groupName===test\_group\_1  ShardName==test2  DataSource==t2m  插入记录id=0026（结束） |

* + 1. 查询操作

查询表t\_user的数据，需要分别查询test1和test2两个DataSourceShard中的数据，然后进行合并、排序，多个分片查询可以并发进行，对每个分片的查询开启一个子线程，通过线程回调操作获取每个线程执行返回的结果。

测试程序

|  |
| --- |
| @Test  **public** **void** testShards() {  UserInfoDAOImpl dao = *ctx*.getBean(UserInfoDAOImpl.**class**);  ShardParameter sp = **new** UserInfo().getShardParameter("many");  List<Map<String, Object>> list = dao.queryAll(**null**, **null**, **new** String[] { "username", "id",  "email" }, **false**, **false**, **false**, **null**, **null**, **null**, "username, id desc", **null**, sp);  *logger*.debug("total size==" + list.size());  *logger*.debug(list.toString());  } |

运行结果

|  |
| --- |
| // 数据源组  groupName===test\_group\_1  // 数据源组中的分片数  ShardName==test1  ShardName==test2  // 查询时使用分片test1中的t1s1从库，使用分片test2中的t2s1从库  DataSource==t1s1  DataSource==t2s1  // 分片test1中的查询结果  test1 size ==7  test1 rows ==[  {username=aaa, id=115, email=aaa},  {username=aaa, id=113, email=aaa},  {username=aaa, id=111, email=aaa},  {username=n001, id=001, email=e001@ut.com},  {username=u09, id=116, email=aaa},  {username=u12, id=112, email=aaa},  {username=u30, id=114, email=aaa}]  分片test2中的查询结果  test2 size ==2  test2 rows ==[  {username=u20, id=20, email=e20},  {username=u21, id=21, email=e21}]  // 排序  orderBys=={username=1, id=-1}  // 查询结果汇总  total size==9  rows ==[  {username=aaa, id=115, email=aaa},  {username=aaa, id=113, email=aaa},  {username=aaa, id=111, email=aaa},  {username=n001, id=001, email=e001@ut.com},  {username=u09, id=116, email=aaa},  {username=u12, id=112, email=aaa},  {username=u20, id=20, email=e20},  {username=u21, id=21, email=e21},  {username=u30, id=114, email=aaa}] |

多个分片结果集的排序合并，通过DAO方法中参数orderBy = "username, id desc"构建一个Map或JavaBean类型数据之间排序的比较器Comparator。有了Comparator实例就可以对集合中的元素进行排序。

1. **加密解密库**
   1. 报文协议格式

报文的二进制数据块的格式如下：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 报文  类型  版本 | 加密  报文  长度 | 会话  SID | 加密  方法 | 加密后的内容 | | | |
| 压缩报  文长度 | 压缩后的内容 | | 压缩内容的MD5 |
| 报文长度 | XML数据 |
|  | 1B（0x71） | 4B | 32B | 1B | 4B | 4B | nB | 16B |
| 0x71 通讯控制  0x73 数据报文 | | Base64处理,固定8B | UUID | 原文 | Base64处理 | | | |

报文数据项说明：

1. 报文版本：

|  |  |
| --- | --- |
| 报文版本 | 取值 |
| 通讯控制 | 0x71 |
| 数据报文 | 0x73 |

1. 加密方法：

|  |  |
| --- | --- |
| 加密方法 | 取值 |
| MODE\_MD5 | 1 |
| MODE\_AES256 | 2 |
| MODE\_RSA1024 | 3 |
| MODE\_RSA2048 | 4 |
| MODE\_BASE64 | 5 |

* 1. 报文交互过程

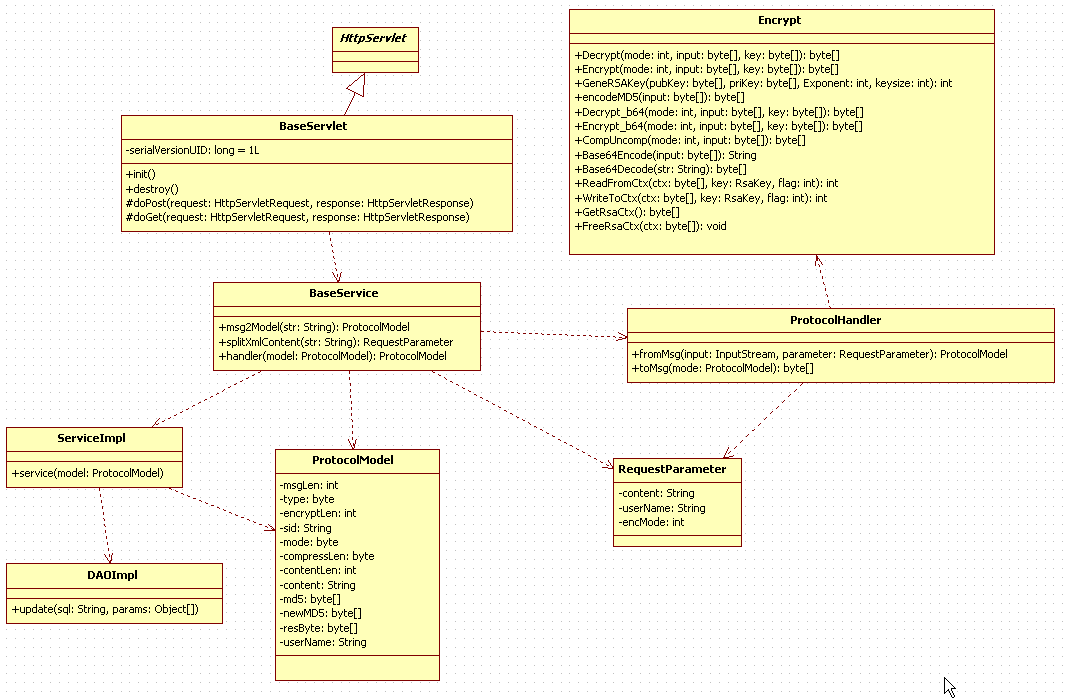




* 1. 设计实现

为提高数据传输的安全性和高效性，结合使用RSA、AES、BASE64、MD5、解压缩等算法对请求和响应的XML格式数据进行编码后，以二进制流的形式进行网络间传输。

Java端程序通过JNI接口转调以上C++算法的具体实现，以保证Java服务端程序和C++客户端程序使用统一的算法实现，降低异构语言之间的差异。



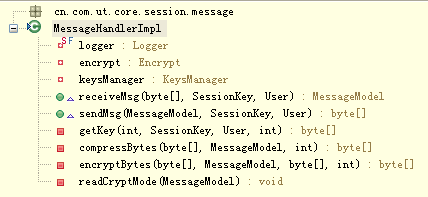
* 1. 主要API

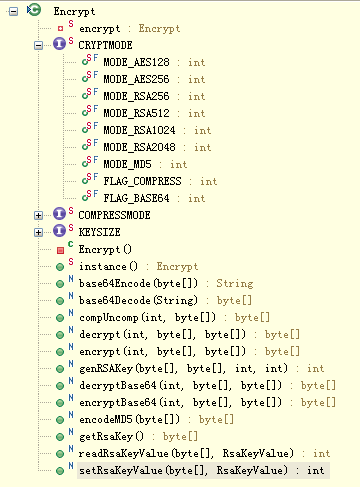
工程：tuna\_common

关键类：

cn.com.ut.core.session.message.MessageHandlerImpl

cn.com.ut.core.common.encrypt.Encrypt





1. **数据访问层（DAL）**
   1. 介绍

为降低数据库节点变化对应用层数据访问方式的影响，充分利用服务器资源，平衡服务器之间的负载，增强服务器运行期间的稳定性，提高服务器的并发数和吞吐量，引入数据访问层中间件（DAL）的概念，DAL大致包括以下功能。

|  |
| --- |
| 数据访问层DAL |
| 负载均衡调度器（Load Balance） |
| 读写分离（Read/Write Splitting） |
| 数据切分（Sharding） |
| 数据源连接池（DataSource Connection Pool） |
| 事务管理器（Transaction Manager） |

* + 1. 负载均衡调度器

主要是数据库多Slave节点之间SQL读操作的负载均衡。负载均衡算法根据一定规则进行任务调度，将请求分发至某Slave节点。如果该Slave失效，再将请求分发给下一个可用的Slave节点为止。

负载均衡器需要和数据库连接池进行联合，负载均衡器需要获取连接池中各数据源的状态，包括是否可用、活动连接数、空闲连接数、池大小等，负载均衡算法需要结合这些状态进行计算，最后将计算所得的数据源对象返回给数据访问对象调用。

连接池可以主动探测池中数据源是否可用，通过定时发送简单的SQL语句给绑定某数据源的connection进行执行，返回成功或失败状态。如果重复一定次数检测到某数据源已经不可用，须通知负载均衡器从池中移除失效的数据源对象，尽可能的保证每次负载均衡计算时池中的可分配的数据源都是有效的。

另外，连接池中每个数据源对应每个slave节点，如果数据库主从节点配置发生变化，主要是从节点数改变，添加或者删除slave节点，salve节点发生故障，连接池需要定时获取数据库配置的最新信息，以便对池中的数据源对象进行有效管理。

* + 1. 读写分离

数据库写操作请求分发给Master节点；读操作由负载均衡器根据一定规则将请求分发至某Slave节点。

由于数据库写操作只能提交给Master节点，如果Master发生故障，这个故障将经由连接池通知给负载均衡器，所有的写操作请求经过负载均衡器将直接返回失败，直到Master节点从故障中恢复，连接池检查到Master节点可用，并通知给负载均衡器更新该Master节点对应数据源对象的可用状态。

* + 1. 连接池

Java连接池技术已经非常成熟，倍受推荐的开源连接池主要有c3p0、DBCP、BoneCP等。

* + 1. 事务管理器

实现思路：

MultipleDataSourcesTransactionManage接受一组数据源作为事务的目标资源，当事务开始的时候，开启所有数据源相关的本地事务, 事务提交或者回滚的时候，则同时提交或者回滚所有的本地事务。同时开始所有数据源上的本地事务，默认情况下，自然会占用每一个数据源的一个物理连接，进而可能造成性能问题，可以对依赖的数据源进行拦截, 加入了一层LazyConnectionDataSourceProxy，该LazyConnectionDataSourceProxy将保证只有存在确切的数据访问操作的时候，才会真正的从目标数据源中获取真正的物理连接。

实际上, 为了避免不必要的物理连接的占用, 也可以在事务定义中加入相应的标志数据来表明要在哪些资源之上开启本地事务, 这可以通过在Service层次标注Annotation或者附带外部配置文件的形式实现。

* 1. 设计实现



* 1. 基本配置，Spring管理

DataSourceShard：

|  |
| --- |
| <bean id=*"test1"* class=*"cn.com.ut.core.dal.datasource.DataSourceShard"*>  <property name=*"shardName"* value=*"test1"* />  <property name=*"master"*>  <bean class=*"cn.com.ut.core.dal.datasource.DataSourceWrapper"*>  <property name=*"dataSourceName"* value=*"t1m"* />  <property name=*"jdbcTemplate"* ref=*"t1mJt"* />  </bean>  </property>  <property name=*"slaves"*>  <util:list list-class=*"java.util.ArrayList"*>  <bean class=*"cn.com.ut.core.dal.datasource.DataSourceWrapper"*>  <property name=*"dataSourceName"* value=*"t1s1"* />  <property name=*"jdbcTemplate"* ref=*"t1s1Jt"* />  <property name=*"weight"* value=*"2"* />  </bean>  <bean class=*"cn.com.ut.core.dal.datasource.DataSourceWrapper"*>  <property name=*"dataSourceName"* value=*"t1s2"* />  <property name=*"jdbcTemplate"* ref=*"t1s2Jt"* />  <property name=*"weight"* value=*"1"* />  </bean>  </util:list>  </property>  </bean> |

DataSourceGroup:

|  |
| --- |
| <bean id=*"testGroup1"* class=*"cn.com.ut.core.dal.datasource.DataSourceGroup"*>  <property name=*"groupName"* value=*"test\_group\_1"* />  <property name=*"shards"*>  <util:map map-class=*"java.util.LinkedHashMap"*>  <entry>  <key><value>test1</value></key>  <ref bean=*"test1"*/>  </entry>  <entry>  <key><value>test2</value></key>  <ref bean=*"test2"*/>  </entry>  </util:map>  </property>  </bean> |

DataSourceRouter：

|  |
| --- |
| <bean id=*"dataSourceRouter"* class=*"cn.com.ut.core.dal.datasource.DataSourceRouter"* init-method=*"init"* destroy-method=*"destroy"*>  <property name=*"groups"*>  <util:map map-class=*"java.util.LinkedHashMap"*>  <!-- <entry>  <key><value>test\_group\_1</value></key>  <ref bean="testGroup1"/>  </entry> -->  <entry>  <key><value>baseGroup</value></key>  <ref bean=*"baseGroup"*/>  </entry>  </util:map>  </property>  </bean> |

* 1. 连接池选择

从性能考虑，推荐druid、tomcat-jdbc（tomcat自带实现）、BoneCP，apache common dbcp、c3p0在高并发生产环境下表现不佳。

例如，本机的一组测试数据，分别测试每种类型的连接池，前提是各种连接池的初始配置基本一致，每组测试开启200或500个线程，每个线程执行1000次查询操作，耗时情况如下：

200线程数的情况：

|  |
| --- |
| thread 200 druid millis : 7656  thread 200 tomcat-jdbc millis : 7968  thread 200 boneCP millis : 7796  thread 200 c3p0 millis : 10953  thread 200 dbcp millis : 10531 |

500线程数的情况：

|  |
| --- |
| thread 500 druid millis : 19342  thread 500 tomcat-jdbc millis : 20749  thread 500 boneCP millis : 20093  thread 500 c3p0 millis : 31779  thread 500 dbcp millis : 26015 |

* 1. 运行流程
     1. 完整流程



* + 1. 通过DAO定义的泛型类型找到关联的实体类型，例如

|  |
| --- |
| public class ConfigDAOImpl extends JdbcOperationsImpl<Config> |

那么，ConfigDAOImpl关联的实体类型为Config

* + 1. 查找DAO默认操作的数据表（ormapper.xml配置）

例如，ormapper.xml配置：

|  |
| --- |
| <schema name=*"baseGroup"* shards=*"base1"*>  <ormapper name=*"cn.com.ut.biz.system.entities.Config"* table=*"t\_sys\_config"*/>  </schema> |

可见，Config实体关联的数据表为*"t\_sys\_config"*，即为ConfigDAOImpl默认操作的数据表。

* + 1. 拦截器（动态代理）



1. ThreadLocal<LinkedList<ThreadDataSource>> threadDataSources

每次请求，都会开启一个新的线程，一个线程的生命周期其实可以看作是一次业务方法调用的过程，一次业务方法调用可能会调用多个DAO方法进行多次数据库访问。使用ThreadLocal类型变量的目的是单独存放每个线程（或每次业务方法调用）的中间数据。这里ThreadLocal变量threadDataSources中存放的数据为LinkedList<ThreadDataSource>类型的集合，目的是对一次业务方法调用过程中的每次DAO方法调用创建一个ThreadDataSource类型实例，ThreadDataSource的定义：



ThreadDataSource实例的groupName属性标识当前DAO方法要访问的逻辑数据源组DataSourceGroup，writeable标识当前DAO方法是读操作还是写操作。读和写操作根据DAO的方法签名规则决定。例如，add、update、delete开头的方法为写操作，read、query、select开头的方法为读操作。一次业务调用过程中的多次DAO方法调用是顺序执行的，当前DAO方法调用创建ThreadDataSource实例时，须判断队列中是否存在和当前DAO方法调用groupName相同的ThreadDataSource实例，如果不存在，当前DAO方法的ThreadDataSource实例的writeable属性值由自己决定。如果存在，并且上一个ThreadDataSource实例的writeable的属性值如果为写操作，那么当前ThreadDataSource实例也随之为写操作，如果上次为读操作，那么当前ThreadDataSource实例的writeable属性值由自己决定。

例如:

|  |
| --- |
| serviceMethodTest() {  /\*groupName为group1，insert操作，队列中不存在组名为group1的ThreadDatasource实例，创建ThreadDatasource(group1,true)\*/  dao.insertGroup1();  /\*groupName为group2，select操作，队列中不存在组名为group2的ThreadDatasource实例，创建ThreadDatasource(group2,false)\*/  dao.selectGroup2();  /\*groupName为group1，select操作，队列中存在组名为group1的ThreadDatasource实例，且上次为写操作，故此次也为写操作，创建ThreadDatasource(group1,true)\*/  dao.selectGroup1();  /\*groupName为group2，select操作，队列中存在组名为group2的ThreadDatasource实例，且上次为读操作，，故本次操作还是读操作，创建ThreadDatasource(group2,false)\*/  dao.queryGroup2();  } |

1. daoMethodBeforeAdvice(JoinPoint)

在每个DAO方法调用之前进行拦截，目的是对上一步描述的ThreadLocal<LinkedList<ThreadDataSource>> threadDataSources设置值。

* + 1. 多组数据源管理者DataSourceRouter



1. List<DataSouceWrapper>

应用依赖的所有数据源

1. init()

DataSourceRoute初始化时，会开启一个线程池，线程池中线程数量和List<DataSouceWrapper>的size一致，目的是对List<DataSouceWrapper>中的每一个DataSouceWrapper开启一个线程，进行数据源连接可用性的检测，并且隔一段时间检测一次，检测结果将设置到DataSouceWrapper实例的available属性，当检测到某DataSouceWrapper实例不可用时，如果该实例为从库，那么从库之间负载均衡的时候将跳过无效的DataSouceWrapper.

1. destroy()

与init()对应，取消线程池中的任务。

1. isWriteable()

从ThreadLocal<LinkedList<ThreadDataSource>> threadDataSources中获取当前ThreadDataSource的读写标识。

1. getDataSourceShardByGroupAndShardName(String shardName)

从ThreadLocal<LinkedList<ThreadDataSource>> threadDataSources中获取当前ThreadDataSource的groupName属性，进而获取DataSourceGroup实例，再根据

shardName获取DataSourceGroup中的一个DataSourceShard.

1. getDataSourceWrapperFromShard(DataSourceShard dataSourceShard, boolean writeable)

根据writeable的读写标识从dataSourceShard中获取用于写的Master对应的DataSourceWrapper，或者根据路由规则从一组Slave数据源中获取一个可用的用于读操作的DataSourceWrapper实例。DataSourceWrapper定义：



* + 1. 查找逻辑数据源组DataSourceGroup（ormapper.xml配置）



例如，ormapper.xml配置：

|  |
| --- |
| <schema name=*"baseGroup"* shards=*"base1"*>  <ormapper name=*"cn.com.ut.biz.system.entities.Config"* table=*"t\_sys\_config"*/>  </schema> |

表示Config实体和table数据表之间对象关系映射的元素ormapper结点位于<schema name=*"baseGroup"* shards=*"base1"*>下面，逻辑数据源组名称即为*"baseGroup".*

从DataSourceRoute的Map<String, DataSourceGroup> groups集合中通过组名称获取DataSourceGroup实例。

* + 1. 查找DataSourceShard（默认或自定义规则）



一个逻辑数据源组DataSourceGroup默认只有一个DataSourceShard，只有用到分片的时候才会有1个以上DataSourceShard，没有分片的情况就使用默认唯一的那个DataSourceShard。

使用DataSourceGroup#rdByName(String shardName)获取DataSourceShard.

* + 1. 获取JdbcTemplate或DataSource

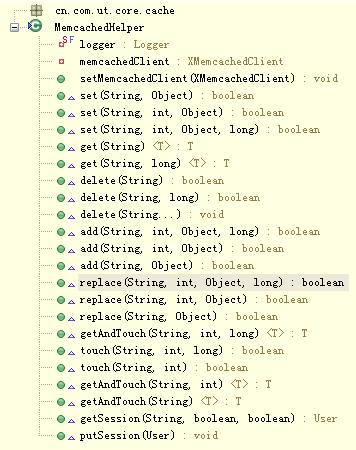
DataSourceWrapper实例其实是一个标准DataSource的包装器，可以直接从DataSourceWrapper实例获取JdbcTemplate成员实例进行JDBC操作，JdbcTemplate实例本身绑定了一个DataSource实例，有了DataSource实例就能获取数据库物理Connection的代理，有了Connection就可以开始数据库的CRUD操作。

1. **分布式缓存**
   1. 设计实现

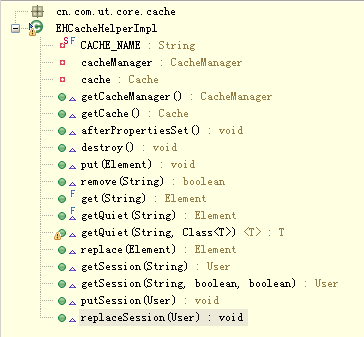


* 1. 缓存层次结构









* 1. 基本使用

Set操作

|  |
| --- |
| /\*\*  \* 保存键值对至缓存中  \*  \* **@param** key  \* 键  \* **@param** exp  \* 过期时间  \* **@param** value  \* 值  \* **@param** timeout  \* 超时时间  \* **@return**  \*/  **boolean** set(String key, **int** exp, Object value, **long** timeout); |

Get操作

|  |
| --- |
| /\*\*  \* 按照key从缓存中查找数据  \*  \* **@param** key  \* **@param** timeout  \* **@return**  \*/  <T> T get(String key, **long** timeout); |

Delete操作

|  |
| --- |
| /\*\*  \* 按照key清除缓存  \*  \* **@param** key  \* **@param** timeout  \* **@return**  \*/  **boolean** delete(String key, **long** timeout);  /\*\*  \* 清除一组key  \*  \* **@param** keys  \*/  **void** delete(String... keys); |

* 1. DAO层的缓存

DAO父类JdbcOperationsImpl方法集成缓存操作

* + 1. add

当插入一条数据成功后，如果缓存参数CachedParameter不为空，CachedParameter#getgetEvict()返回要清除的关联缓存的一组key

|  |
| --- |
| **public** **int** add(String table, String[] insertColumnArray, String[] appendColumns,  Object[] parameterArray, CachedParameter cachedParameter) |

* + 1. updateById

当根据ID成功更新一条记录后，清除以“table/id”为key的缓存数据

|  |
| --- |
| **public** **int** updateById(String table, String[] updateColumnArray, String[] appendColumns,  String updateColumnJoin, Object[] updateParameterArray, String id,  CachedParameter cachedParameter) |

* + 1. update

更新单表的若干条记录，该表以ID字段作主键。这里Where子句不确定，再更新操作之前，先解析Where子句，以为子句为条件查询查询待更新的记录的一组Id，当更新操作成功后，从缓存系统中清除以这一组中的每个“table/id”为key的缓存数据

|  |
| --- |
| **public** **int** update(String table, String[] updateColumnArray, String[] appendColumns,  String updateColumnJoin, String[] whereColumnArray, String whereColumnJoin,  **int**[] replaceArray, Object[] updateParameterArray, Object[] whereParameterArray,  CachedParameter cachedParameter) |

* + 1. deleteById

处理方式与updateById类似

|  |
| --- |
| public int deleteById(String table, String id, CachedParameter cachedParameter) |

* + 1. delete

处理方式与update类似

|  |
| --- |
| **public** **int** delete(String table, String[] whereColumnArray, String whereColumnJoin,  **int**[] replaceArray, Object[] parameterArray, CachedParameter cachedParameter) |

* + 1. getById

以Id为主键获取单表的一条记录，查询成功返回时，以“table/id”为key添加数据至缓存系统中。

|  |
| --- |
| **public** <T> T getById(Class<T> clazz, String table, String[] selectColumnArray,  String[] appendColumn, String id, CachedParameter cachedParameter) |

* 1. 基于配置的自动缓存
     1. 使用注解

使用@Cached标识的方法将自动应用缓存，由拦截器负责处理。

例如：

|  |
| --- |
| @Cached(key = ConfigDAOImpl.*CONFIGS\_KEY*, expire = CacheHelper.*LONGTIME\_EXPIRE*)  **public** Map<String, String> findConfig() {  } |

* + 1. Key值

@Cached标识的key值即为缓存的key，如果被缓存的方法参数也有@Cached标识，那么key将由@Cached标识的key加“/”加参数值组成

例如：假设方法参数dictType=“a”,这里的key值为“DictType/a”

|  |
| --- |
| @Cached(key = "DictType", expire = CacheHelper.*LONGTIME\_EXPIRE*)  **public** Map<String, Object> findDictDataMap(@Cached String dictType) {  } |

* + 1. 拦截器

CacheMethodInterceptor方法

|  |
| --- |
| public Object aroundServiceMethodAdvice(ProceedingJoinPoint pjp) throws Throwable |

该方法完对使用@Cached标识的方法设置缓存，并动态构建缓存的key

* + 1. 开启或关闭缓存功能

CacheMethodInterceptor方法

|  |
| --- |
| public Object aroundCacheMethodAdvice(ProceedingJoinPoint pjp) throws Throwable |

该方法根据缓存配置决定是否开启缓存，如果不开启，在调用CacheHelper中的方法进行缓存操作时，将直接返回，无实际操作，不连接缓存服务器。

* + 1. 缓存带来的问题

缓存的清除时机，例如，

List<UserInfo> queryUsers(String groupId)方法查询一个用户组的全部用户，然后进行缓存，缓存的key值为groupId，如果该用户组下，加入了新的用户，更新了某个或某些用户，删除了某个或某些用户，都需要清除以该groupId为key的缓存数据。缓存的合理清除无疑带来工作量的增加。

还有，开发测试的时候，我们习惯直接修改数据表数据进行测试，但是，如果正好被修改的行记录进行了缓存，测试时拿到的数据来自缓存，而非刚刚被修改过的数据表中的记录，带来调试的不便。所以，建议在开发调试的时候，关闭全局缓存配置。

1. **缓存会话管理**
   1. 具体的问题

当一个带有会话表示的Http请求到Web服务器后，需求在请求中的处理过程中找到session数据。而问题就在于，session是保存在单机上的。 假设我们有应用A和应用B，现在一位用户第一次访问网站，session数据保存在应用A中。如果我们不做处理，怎么保障接下来的请求每次都请求到应用A呢? 如请求到了应用B中，就会发现没有这位用户的session数据，这绝对是不能容忍的。

* 1. 解决方案

解决方案有Session Stick，Session复制，Session集中管理，基于Cookie管理，下面一一说明。

* + 1. Session Stick

在单机情况，session保存在单机上，请求也是到这台单机上，不会有问题。变成多台后，如果能保障每次请求都到同一台服务，那就和单机一样了。 这需要在负载均衡设备上修改。这就是Session Stick，这种方式也会有问题：

如果某一台服务器宕机或重启，那么这台服务器上的session数据就丢失了。如果session数据中还有登录状态信息，那么用户需要重现登录。

负载均衡要处理具体的session到服务器的映射。

* + 1. Session复制

Session复制顾名思义，就是每台应用服务，都保存会话session数据，一般的应用容器都支持。与Session Stick相比，sessioon复制对负载均衡 没有太多的要求。不过这个方案还是有缺点：

同步session数据带来都网络开销。只要session数据变化，就需要同步到所有机器上，机器越多，网络开销越大。

由于每台服务器都保存session数据，如果集群的session数据很多，比如90万人在访问网站，每台机器用于保存session数据的内容占用很严重。

这就是Session复制，这个方案是靠应用容器来完成，并不依赖应用，如果应用服务数量并不是很多，可以考虑。

* + 1. Session集中管理

这个也很好理解，再加一台服务，专门来管理session数据，每台应用服务都从专门的session管理服务中取会话session数据。可以使用数据库，NOSQL数据库等。 和Session复制相比，减少了每台应用服务的内存使用，同步session带来的网络开销问题。但还是有缺点：

读写session引入了网络操作，相对于本机读写session，带来了延时和不稳定性。

如Session集中服务有问题，会影响应用。

* + 1. 基于Cookie管理

最后一个是基于Cookie管理，我们把session数据存放在cookie中，然后请求过来后，从cookie中获取session数据。与集中管理相比，这个方案并不依赖外部 的存储系统，读写session数据带来的网络操作延时和不稳定性。但依然有缺点：

Cookie有长度限制，这会影响session数据的长度。

安全性。session数据本来存储在服务端的，而这个方案是让session数据转到外部网络或客户端中，所以会有安全性问题。不过可以对写入Cookie的session 数据做加密。

带宽消耗。由于加了session数据，带宽当然也会增加一点。

性能消耗。每次Http请求和响应都带有Session数据，对于Web服务器来说，在同样的处理情况下，响应的结果输出越少，支持的并发请求越多。

总结

这4种方案都是可用的方案，我比较倾向于使用Session集中管理，不过这4种方案都各有优劣，需要根据具体的实际场景做出合适的选择。

* 1. 设计实现



* + 1. 使用域名

统一级域名，\*.ut.com，确保cookie数据从浏览器传递给集群服务器

安全证书

* + 1. 会话格式

键值对

Key：cookie值

Value：User对象序列化（包括用户信息、权限、店铺、企业信息等）

* + 1. 全局会话

应用服务器远程访问，创建时设置过期时间t1（分钟），每次访问修改过期时间

* + 1. 本地会话

应用服务器本地访问，创建时设置过期时间t2，每次访问不修改这个过期时间，且t2<t1，尽量使t2隔一段时间就过期，过期后去访问远程缓存，重新修改时间t1.

* + 1. 会话更新

如果在应用服务A中进行会话(key=”xxx”)更新，如权限范围变大。

更新远程缓存中会话数据，服务器可以生成新的cookie值作为key。

从远程缓存获取此会话的原值，从值中获取保存过此会话缓存的其他应用服务器列表，服务器A以远程调用方式通知其他服务器删除本地会话(key=”xxx”)的缓存，以便下次从远程缓存重新加载一次最新会话数据。

* + 1. 问题

本地会话与远程会话可能不能完全同步，比如远程缓存故障丢失，本地还有此缓存数据。

用户在线列表，按用户查找会话困难，缓存服务器需要进行全遍历全部key.虽然有的缓存服务器如redis支持key的模糊查询，但是还是会影响性能急剧下降，CPU负载过高，官方不推荐使用。

如果要实现方面检索的会话，如按店铺、企业、用户、主机来检索会话，只得用关系数据库或NoSQL数据库来实现。

1. **定时任务调度**
   1. 介绍

场景：

非前端发起的业务请求，后端按时间点触发

处理有时效限制的业务，如定时促销，订单支付超时

对时间点有严格控制的业务，比如在凌晨服务器访问量低的时候进行数据汇总

容许被重复执行的业务，不会造成错误

可以对时间点进行灵活的配置管理

分布式：数据库排他锁，存在单点故障问题，每一次时间点的任务调度只分配给一台机器

负载均衡：每一次时间点的任务调度可分配给不同的机器来完成，跳过失效的服务器

要求：应用服务器都跟随相同的时间服务器

* 1. 设计实现

使用开源组件quartz，结合Spring进行配置管理

* 1. 举例

Spring配置

|  |
| --- |
| <bean name=*"partIndexDocUpdateJobDetail"* class=*"org.springframework.scheduling.quartz.JobDetailBean"*>  <property name=*"jobClass"*>  <value>cn.com.ut.quartz.service.MyDetailQuartzJobBean</value>  </property>  <property name=*"jobDataAsMap"*>  <map>  <entry key=*"targetObject"* value=*"autoUpdateIndex"* />  <entry key=*"targetMethod"* value=*"partIndexDocUpdate"* />  </map>  </property>  <property name=*"group"*>  <value>${quartz.group}</value>  </property>  </bean>  <bean id=*"partIndexDocUpdateCronTrigger"* class=*"org.springframework.scheduling.quartz.CronTriggerBean"*>  <property name=*"jobDetail"*>  <ref bean=*"partIndexDocUpdateJobDetail"* />  </property>  <property name=*"cronExpression"*>  <value>0 0/30 \* \* \* ?</value>  </property>  <property name=*"group"*>  <value>${quartz.group}</value>  </property>  </bean>  <!--总管理类 如果将lazy-init='false'那么容器启动就会执行调度程序 如定义了多个任务，则在<list>节点下增加多个<ref  local="触发"> -->  <bean name=*"quartzScheduler"*  class=*"org.springframework.scheduling.quartz.SchedulerFactoryBean"*>  <property name=*"dataSource"*>  <ref bean=*"quartzM"* /> <!-- 数据源引用指向，包含集群所需的所有表 -->  </property>  <property name=*"configLocation"*  value=*"classpath:cn/com/ut/config/properties/quartz.properties"* /><!--  quartz.properties的位置 -->  <property name=*"triggers"*>  <list>  <ref local=*"partIndexDocUpdateCronTrigger"* />  </list>  </property>  <property name=*"applicationContextSchedulerContextKey"* value=*"applicationContext"* />  </bean> |

1. **远程调用**
   1. 介绍

本质：http+序列化（hessian）

业务拆分：

适应业务量的迅速增长，业务划分后职责分工更清晰

适应业务数据量的快速增长，配合数据库的拆分

业务模块重用，降低耦合，避免一个超大的应用依赖所有的数据库、缓存等服务器的连接资源

部署、维护方便，只影响部分应用服务器

负载均衡：

根据服务接口和接口方法可以将请求转发给有此服务的任意一台远程应用服务器，且保证此服务器的可用性

其他参数传递：

本地请求设置http请求头信息，远程主机解析请求头信息

优化：

如果分页查询列表中，每条数据中的部分字段来自远程接口，尽量避免1次本地访问依赖多次的远程调用，可靠性、性能大大降低

可以将本地接口分页查询结果中与远程接口有关的参数字段集合一次传递给远程接口，让远程接口在本地完成一次或多次数据库查询，然后将结果一次返回给主调服务器。缺点就是有些复杂，请求时需要拼凑远程接口的方法参数集合，从远程接口返回结果后，需要将其和主调接口的查询数据进行拼接Join，工作量复杂度增加。



参数和返回类型：结构简单、轻量，尽量避免使用RespResult这样的参数，直接从会话中获取的user对象作为参数

会话：主调接口验证会话，远程接口不验证，接口依赖的会话由主调方法通过参数传递

* 1. 设计实现

使用开源组件Hessian，结合Spring进行配置管理

* 1. 举例

服务发布

|  |
| --- |
| <bean id=*"/ConfigServiceHessian"* class=*"org.springframework.remoting.caucho.HessianServiceExporter"*>  <property name=*"service"* ref=*"configServiceImpl"* />  <property name=*"serviceInterface"* value=*"cn.com.ut.biz.system.service.ConfigService"* />  </bean> |

服务订阅

|  |
| --- |
| <bean id=*"configService"* class=*"cn.com.ut.core.rpc.hessian.ext.MyHessianProxyFactoryBean"*>  <property name=*"serviceUrl"* value=*"${rpc.server.system}/ConfigServiceHessian"*/>  <property name=*"serviceInterface"* value=*"cn.com.ut.biz.system.service.ConfigService"*/>  <property name=*"aoper"* ref=*"aoper"* />  </bean> |

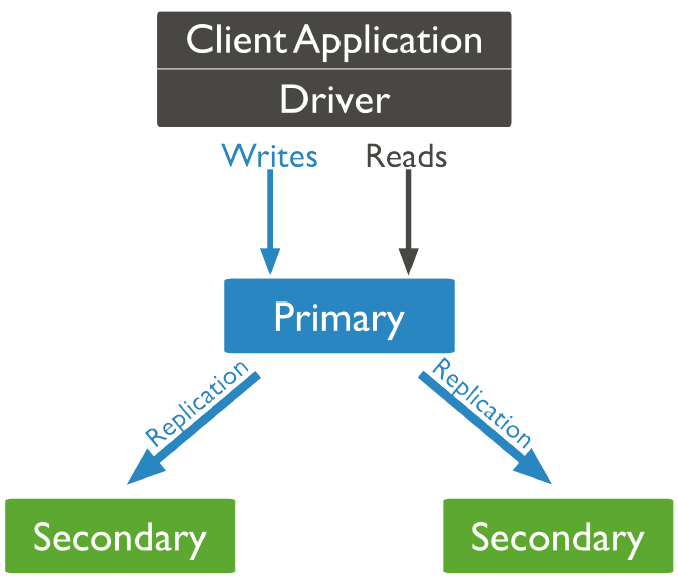
1. **数据库集群**
   1. 服务器集群部署



* 1. 应用客户端

参考章节“数据访问层”

1. **NoSQL数据库**
   1. 服务器集群部署



* 1. 应用客户端
     1. 官网客户端

<https://github.com/mongodb/mongo-java-driver>

* + 1. 应用客户端封装类库

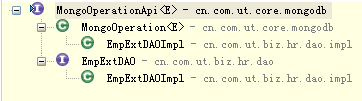
项目：tuna\_nosql

关键类：

cn.com.ut.core.mongodb. MongoOperation

cn.com.ut.core.mongodb.manager. MongoClientManager

* + 1. Nosql DAO层次结构





* + 1. 配置

服务器配置

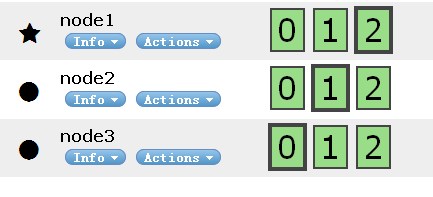
|  |
| --- |
| <servers>  <server name=*"s1"*>  <host name=*"h1"* addr=*"MON5"* port=*"27010"*></host>  <host name=*"h2"* addr=*"MON6"* port=*"27010"*></host>  <host name=*"h3"* addr=*"MON4"* port=*"27010"*></host>  </server>    <!-- <server name="s1">  <host name="h1" addr="192.168.2.41" port="27017"></host>  </server> -->    </servers> |

数据库、集合、实体类配置

|  |
| --- |
| <dbs>  <server name=*"s1"*>  <db name=*"DB\_PORTAL\_SOF"*>  <coll name=*"C\_Bas\_Friend"* entity=*"cn.com.ut.sns.friends.entities.FriendsRelation"*></coll>  </db>  </server>  </dbs> |

1. **全文检索**
   1. 服务器集群部署

使用开源组件ElasticSearch



* 1. 应用客户端

项目工程：tuna\_elasticsearch

* 1. 配置

服务器配置

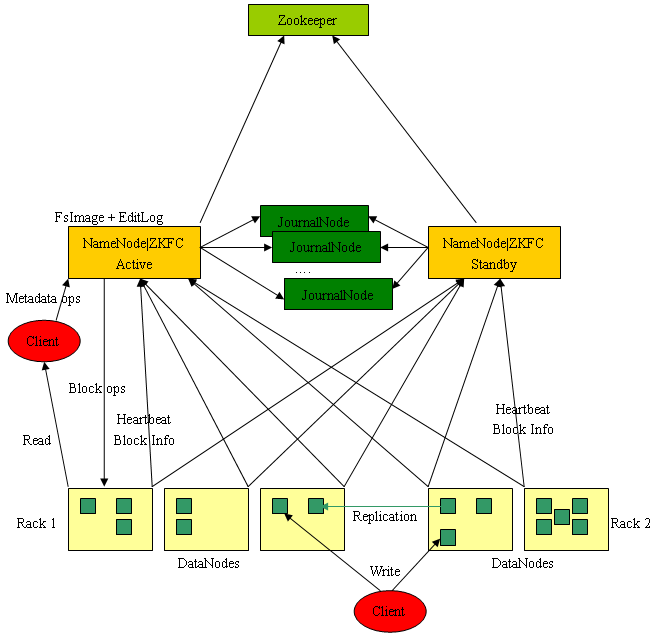
|  |
| --- |
| <servers>  <server name=*"one"* host=*"192.168.5.138"* port=*"8300"* />  <server name=*"two"* host=*"192.168.6.137"* port=*"8300"* />  <server name=*"three"* host=*"192.168.6.138"* port=*"8300"* />  </servers> |

索引配置

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <index>  <config>  <name>goods</name>  <indexName>shopgoods</indexName>  <indexType>goods</indexType>  <arrayField>sgc\_id,attr</arrayField>  <aliasIndexName>shopgoods\_alias</aliasIndexName>  <mappingStr>  {    "goods":{  "\_all":{"enabled":false},  "\_source":{"excludes":["category\_name","brand\_name","goods\_keywords","goods\_brief"]},  "\_parent":{  "type":"shop"  },  "properties":{  "goods\_id":{  "type":"string",  "index":"not\_analyzed"  },  "category\_id":{  "type":"string",  "index":"not\_analyzed"  },"img\_id":{  "type":"string",  "index":"not\_analyzed"  },  "category\_name":{  "type":"string",  "index\_analyzer":"ik\_max\_word",  "index":"analyzed"  },  "brand\_id":{  "type":"string",  "index":"not\_analyzed"  },  "brand\_name":{  "type":"string",  "index\_analyzer":"ik\_max\_word",  "index":"analyzed"  },  "goods\_name":{  "type":"string",  "index\_analyzer":"ik\_max\_word",  "index":"analyzed"  },  "shop\_name":{  "type":"string",  "index\_analyzer":"ik\_max\_word",  "index":"analyzed"  },  "shop\_id":{  "type":"string",  "index":"not\_analyzed"  },  "goods\_keywords":{  "type":"string",  "index\_analyzer":"ik\_max\_word",  "index":"analyzed"  },  "goods\_brief":{  "type":"string",  "index\_analyzer":"ik\_max\_word",  "index":"analyzed"  },  "goodstype\_id":{  "type":"string",  "index":"not\_analyzed"  },  "goodstype\_name":{  "type":"string",  "index\_analyzer":"ik\_max\_word",  "index":"analyzed"  },  "is\_on\_sale":{  "type":"string",  "index":"not\_analyzed"  },"is\_delivery":{  "type":"string",  "index":"not\_analyzed"  },"is\_discount":{  "type":"string",  "index":"not\_analyzed"  },"is\_recommed":{  "type":"string",  "index":"not\_analyzed"  },"shop\_province":{  "type":"string",  "index":"not\_analyzed"  },"shop\_city":{  "type":"string",  "index":"not\_analyzed"  },"province":{  "type":"string",  "index":"not\_analyzed"  },"city":{  "type":"string",  "index":"not\_analyzed"  },  "category\_property":{  "type":"string",  "index":"not\_analyzed"  },  "sales\_volume":{  "type":"long",  "index":"not\_analyzed"  },  "is\_del":{  "type":"string",  "index":"not\_analyzed"  },  "shop\_price":{  "type":"float",  "index":"not\_analyzed"  },  "click\_count":{  "type":"long",  "index":"not\_analyzed"  },  "sgc\_id":{  "type":"string",  "index":"not\_analyzed"  },  "update\_time":{  "type":"date",  "index":"not\_analyzed"  },"last\_update":{  "type":"date",  "index":"not\_analyzed"  },"test":{  "type":"string",  "index\_analyzer":"ik\_max\_word",  "index":"analyzed"  },  "attr":{  "properties":{  "attr\_id":{  "type":"string",  "index":"not\_analyzed"  },  "attr\_value":{  "type":"string",  "index\_analyzer":"ik\_max\_word",  "index":"not\_analyzed"  }  }  }  }  }    }  </mappingStr>  <parentId>shop\_id</parentId>  </config>  </index> |

1. **分布式文件系统**
   1. 服务器集群部署

使用开源分布式文件系统Hadoop



* 1. 配置

模板：

/tuna\_config/src/main/resources/cn/com/ut/config/hadoop/site/core-site.xml

|  |
| --- |
| <property>  <name>fs.defaultFS</name>  <value>hdfs://cluster1</value>  </property> |

/tuna\_config/src/main/resources/cn/com/ut/config/hadoop/site/hdfs-site.xml

|  |
| --- |
| <property>  <name>dfs.nameservices</name>  <value>cluster1</value>  </property>  <property>  <name>dfs.ha.namenodes.cluster1</name>  <value>HFS6,HFS7</value>  </property>  <property>  <name>dfs.namenode.rpc-address.cluster1.HFS6</name>  <value>HFS6:9000</value>  </property>  <property>  <name>dfs.namenode.rpc-address.cluster1.HFS7</name>  <value>HFS7:9000</value>  </property> |

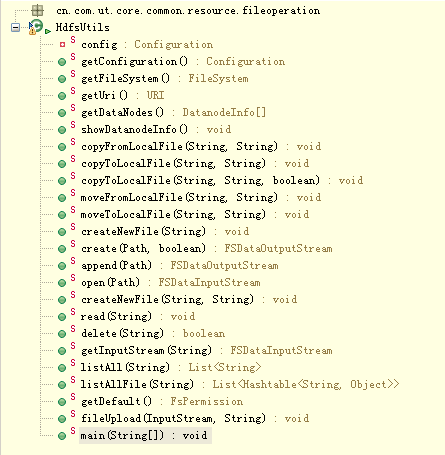
/tuna\_config/src/main/resources/cn/com/ut/config/hadoop/site/mapred-site.xml

* 1. 应用客户端

项目工程：tuna\_common

主要类：

cn.com.ut.core.common.resource.fileoperation.HdfsUtils



1. **参考示例**
   1. 系统全局配置维护
      1. Ormapper.xml配置

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <root>  <schemas>  <schema name=*"test\_group\_1"* shards=*"test1,test2"*>  <ormapper name=*"cn.com.ut.biz.user.entities.UserInfo"* table=*"t\_user"*/>  <ormapper name=*"cn.com.ut.biz.user.entities.Goods"* table=*"t\_goods"*/>  </schema>  <schema name=*"baseGroup"* shards=*"base1"*>  <ormapper name=*"cn.com.ut.biz.system.entities.Config"* table=*"t\_sys\_config"*/>  <ormapper name=*"cn.com.ut.biz.system.entities.DictType"* table=*"t\_sys\_dicttype"*/>  <ormapper name=*"cn.com.ut.biz.system.entities.DictData"* table=*"t\_sys\_dictdata"*/>  </schema>  </schemas>  <resources>  <resource name=*"ormapper-\*.xml"* />  </resources>  </root> |

* + 1. Entity定义

自定的Entity继承自BaseEntity，避免重复定义所有Entity共有的字段。

|  |
| --- |
| **package** cn.com.ut.biz.system.entities;  **import** cn.com.ut.core.dal.jdbc.BaseEntity;  **public** **class** Config **extends** BaseEntity {  **private** **static** **final** **long** *serialVersionUID* = 5441771982121209080L;  **private** String configCode;  **private** String configType;  **private** String configName;  **private** String configValue;  **private** String configDes;  **public** String getConfigCode() {  **return** configCode;  }  **public** **void** setConfigCode(String configCode) {  **this**.configCode = configCode;  }  **public** String getConfigType() {  **return** configType;  }  **public** **void** setConfigType(String configType) {  **this**.configType = configType;  }  **public** String getConfigName() {  **return** configName;  }  **public** **void** setConfigName(String configName) {  **this**.configName = configName;  }  **public** String getConfigValue() {  **return** configValue;  }  **public** **void** setConfigValue(String configValue) {  **this**.configValue = configValue;  }  **public** String getConfigDes() {  **return** configDes;  }  **public** **void** setConfigDes(String configDes) {  **this**.configDes = configDes;  }  } |

* + 1. DAO定义（实现类和接口）

自定义DAO实现类必须继承DAO公共父类JdbcOperationsImpl，复用父类的公开方法，同时标识泛型参数，泛型参数的作用在上面的章节已作详细阐述。

|  |
| --- |
| **package** cn.com.ut.biz.system.dao.impl;  **import** java.sql.Timestamp;  **import** java.util.List;  **import** org.springframework.stereotype.Repository;  **import** cn.com.ut.biz.system.entities.Config;  **import** cn.com.ut.core.cache.CachedParameter;  **import** cn.com.ut.core.common.util.CommonUtil;  **import** cn.com.ut.core.common.util.DateTimeUtil;  **import** cn.com.ut.core.dal.annotation.DAOMethod;  **import** cn.com.ut.core.dal.jdbc.JdbcOperationsImpl;  **import** cn.com.ut.core.dal.jdbc.PageBean;  **import** **static** cn.com.ut.biz.system.entities.Config.\*;  @Repository  **public** **class** ConfigDAOImpl **extends** JdbcOperationsImpl<Config> {  **public** **static** **final** String *CONFIGS\_KEY* = "config";  **protected** CachedParameter cachedParameter = **new** CachedParameter(**null**, 0, **true**,  **new** String[] { *CONFIGS\_KEY* });  /\*\*  \* 插入一条记录  \* **@param** vo  \*/  **public** **void** add(Config vo) {  Timestamp time = DateTimeUtil.*currentDateTime*();  add(**null**,  **new** String[] { *CONFIG\_CODE*, *CONFIG\_DES*, *CONFIG\_NAME*, *CONFIG\_TYPE*, *CONFIG\_VALUE* },  *NAMES*,  **new** Object[] { vo.getConfigCode(), vo.getConfigDes(), vo.getConfigName(),  vo.getConfigType(), vo.getConfigValue(), CommonUtil.*getUUID*(), time, time,  vo.getCreateId(), vo.getCreateId() }, cachedParameter);  }  /\*\*  \* 更新一条记录  \* **@param** vo  \*/  **public** **void** update(Config vo) {  Timestamp time = DateTimeUtil.*currentDateTime*();  updateById(  **null**,  **new** String[] { *CONFIG\_DES*, *CONFIG\_NAME*, *CONFIG\_TYPE*, *CONFIG\_VALUE* },  *NAMES\_UT\_UID*,  **null**,  **new** Object[] { vo.getConfigDes(), vo.getConfigName(), vo.getConfigType(),  vo.getConfigValue(), time, vo.getCreateId() }, vo.getId(), cachedParameter);  }  /\*\*  \* 获取一条记录  \* **@param** id  \* **@return**  \*/  **public** Config get(String id) {  **return** getById(Config.**class**, **null**, **new** String[] { *CONFIG\_CODE*, *CONFIG\_DES*, *CONFIG\_NAME*,  *CONFIG\_TYPE*, *CONFIG\_VALUE* }, *NAMES*, id);  }  /\*\*  \* 删除一条记录  \* **@param** id  \*/  **public** **void** delete(String id) {  deleteById(**null**, id, cachedParameter);  }  /\*\*  \* 分页查询  \* **@param** page  \* **@return**  \*/  **public** List<Config> query(PageBean page) {  **return** query(page, Config.**class**, **null**, **null**, **null**, **null**, **null**);  }  /\*\*  \* 唯一性判断  \* **@param** uniqueFieldValues  \* **@param** pkFieldNames  \* **@param** pkFieldValues  \* **@return**  \*/  @DAOMethod  **public** **boolean** checkUnique(Object[] uniqueFieldValues, String[] pkFieldNames,  Object[] pkFieldValues) {  **return** checkUnique(**null**, *ID*, **new** String[] { *CONFIG\_CODE* }, uniqueFieldValues, pkFieldNames,  pkFieldValues);  }  } |

* + 1. Service定义（实现类和接口）

Service没什么特别之处，有数据库操作时依赖DAO实例。

可见，由于引入了DAO层代码，Service方法的实现与以前相比，更简洁明了。

|  |
| --- |
| **package** cn.com.ut.biz.system.service.impl;  **import** java.util.LinkedHashMap;  **import** java.util.List;  **import** java.util.Map;  **import** javax.annotation.Resource;  **import** org.springframework.stereotype.Service;  **import** cn.com.ut.biz.system.dao.impl.ConfigDAOImpl;  **import** cn.com.ut.biz.system.entities.Config;  **import** cn.com.ut.core.cache.CacheHelper;  **import** cn.com.ut.core.cache.annotation.Cached;  **import** cn.com.ut.core.system.manager.RespResult;  /\*\*  \* 系统配置管理  \*  \* **@author** wuxiaohua  \* **@since** 2014-3-28  \*/  @Service  **public** **class** ConfigServiceImpl {  @Resource  **private** ConfigDAOImpl dao;  /\*\*  \* 新建配置类型  \*  \*/  **public** **void** create(RespResult respResult) {  Config vo = respResult.getVO(Config.**class**);  **boolean** b = dao.checkUnique(**new** Object[] { vo.getConfigCode() }, **null**, **null**);  **if** (!b) {  respResult.addErrMsg("编码重复！");  **return**;  }  dao.add(vo);  respResult.setErr("添加成功！");  }  /\*\*  \* 更新配置类型  \*  \*/  **public** **void** update(RespResult respResult) {  Config vo = respResult.getVO(Config.**class**);  **boolean** b = dao.checkUnique(**new** Object[] {  vo.getConfigCode() },  **new** String[] { "ID" },  **new** Object[] { vo.getId() });  **if** (!b) {  respResult.addErrMsg("编码重复！");  **return**;  }  dao.update(vo);  }  /\*\*  \* 查询配置类型列表  \*  \*/  **public** **void** find(RespResult respResult) {  List<Config> voList = dao.query(respResult.getPageBean());  respResult.setVoList(voList);  }  /\*\*  \* 获取某配置类型详细信息  \*  \*/  **public** **void** getDetail(RespResult respResult) {  Config vo = respResult.getVO(Config.**class**);  vo = dao.get(vo.getId());  respResult.setValueObject(vo);  }  **public** **void** delete(RespResult respResult) {  Config vo = respResult.getVO(Config.**class**);  dao.delete(vo.getId());  }  /\*\*  \* 查询所有配置项目并缓存  \* **@return**  \*/  @Cached(key = ConfigDAOImpl.*CONFIGS\_KEY*, expire = CacheHelper.*LONGTIME\_EXPIRE*)  **public** Map<String, String> findConfig() {  List<Config> voList = dao.query(**null**);  Map<String, String> cache = **null**;  **if** (voList != **null** && !voList.isEmpty()) {  cache = **new** LinkedHashMap<String, String>();  **for** (Config config : voList) {  cache.put(config.getConfigCode(), config.getConfigValue());  }  }  **return** cache;  }  } |

* + 1. 测试用例

部署项目tuna\_web，发布访问地址：

|  |
| --- |
| http://192.168.50.233:8089/tuna\_web/TunaAppSrv |

测试请求报文：与上一版无变化

|  |
| --- |
| <?xml version=*'1.0'* encoding=*'UTF-8'*?>  <!-- <req>  <call>  <lib>system/ConfigService</lib>  <func>create</func>  </call>  <par>  <row>  <CONFIG\_CODE>RECIPEMAX5</CONFIG\_CODE>  <CONFIG\_TYPE>sys</CONFIG\_TYPE>  <CONFIG\_NAME>菜谱总上限5</CONFIG\_NAME>  <CONFIG\_VALUE>105</CONFIG\_VALUE>  <CONFIG\_DES>菜谱总上限值5</CONFIG\_DES>  </row>  </par>  </req> -->  <!-- <req>  <call>  <lib>system/ConfigService</lib>  <func>update</func>  </call>  <par>  <row>  <ID>ec90085b268f499f9460bac4699f15b2</ID>  <CONFIG\_CODE>RECIPEMAX2</CONFIG\_CODE>  <CONFIG\_TYPE>sys</CONFIG\_TYPE>  <CONFIG\_NAME>菜谱总上限1</CONFIG\_NAME>  <CONFIG\_VALUE>101</CONFIG\_VALUE>  <CONFIG\_DES>菜谱总上限值1</CONFIG\_DES>  </row>  </par>  </req> -->  <!-- <req>  <call>  <lib>system/ConfigService</lib>  <func>getDetail</func>  </call>  <par>  <row>  <ID>ec90085b268f499f9460bac4699f15b2</ID>  </row>  </par>  </req> -->  <!-- <req>  <call>  <lib>system/ConfigService</lib>  <func>delete</func>  </call>  <par>  <row>  <ID>ec90085b268f499f9460bac4699f15b2</ID>  </row>  </par>  </req> -->  <req>  <call>  <lib>system/ConfigService</lib>  <func>find</func>  </call>  <par page=*"2"* pagesize=*"2"* total=*"0"*>  <where exp=*"1 # 2"*>  <field name=*"CONFIG\_TYPE"* value=*"sys"* op=*"eq"* id=*"1"* />  <field name=*"CONFIG\_NAME"* value=*"菜谱总上限"* op=*"eq"* id=*"2"* />  </where>  <orderby>  <field name=*"CONFIG\_NAME"* type=*"0"* />  <field name=*"CONFIG\_CODE"* type=*"1"* />  </orderby>  </par>  </req> |

测试工具类：

上一版已经有用HttpLogonClient工具类，基于Netty实现，使用HttpLogonClient设置请求头信息，读取XML内容为请求数据，连接服务器地址发起http请求，并接受服务器的响应报文。

其他选择，比较好用的Socket组件还有Apache HttpClient开源组件，Android系统默认就集成了该组件。

* 1. 数据字典项维护
     1. Entity定义（简洁）

这里DictType实例类的定义与之前Config的差别在于，Config定义了与数据表字段、字段类型对应的成员属性，以及成员属性的setter和getter方法。而这里是通过定义静态常量成员，常量的值对应数据表的字段名。实体类的这些静态常量将在DAO方法中用到。

|  |
| --- |
| **package** cn.com.ut.biz.system.entities;  **import** cn.com.ut.core.dal.jdbc.BaseEntity;  **public** **class** DictType **extends** BaseEntity {  **public** **static** **final** String *DICTTYPE\_CODE* = "DICTTYPE\_CODE";  **public** **static** **final** String *DICTTYPE\_TEXT* = "DICTTYPE\_TEXT";  **public** **static** **final** String *DICTTYPE\_DES* = "DICTTYPE\_DES";  **public** **static** **final** String *PARENT\_CODE* = "PARENT\_CODE";  **public** **static** **final** String *GROUP\_CODE* = "GROUP\_CODE";  **public** **static** **final** String *SORT\_NUM* = "SORT\_NUM";  **public** **static** **final** String *IS\_USE* = "IS\_USE";  } |

* + 1. DAO定义

与ConfigDAOImpl不同的是，DAO方法参数（或者称为值对象VO）不再是实体对象，而是一个Map<String, Object>类型的集合。

注意这样一行代码：作用是静态导入DictType的全部静态成员，便于在DAO方法中引用。

阅读该DAO的方法实现代码，可以看到，DictType中的静态常量成员用来构建sql语句中的要查询、要插入、或者要更新的字段，还有就是，这些静态常量还对应值对象VO集合中的key键，如此一来，我们在构建sql语句中字段和从map中去取值作为sql语句参数的时候，不需要自己去写字段名称或map的key键，而是从直接引用实体类的静态成员。

这样做的几个好处：

1. 可以偷懒，不必定义一个像Config实体那样完整的实体类；
2. 可以规范化，DictType实体类中静态常量成员与数据表一一对应，一旦数据表变化，比如字段名称变化，我们只需要修改DictType实体类中静态常量成员的值为新的字段名称即可，因为DAO方法中是引用DictType实体类中静态常量，而非硬编码的的方式，所以不需要修改代码，如果DAO中是硬编码的的方式，一旦某个字段变化，DAO中所有操作该数据字段的方法都要修改，很麻烦，在修改时还可能漏掉一些地方；
3. DictType实体类中静态常量成员与请求数据中的参数名对应，这样在将请求参数转为map集合后，map.get(“name”)的写法显示比map.get(DictType.NAME)麻烦，因为map.get(“name”)这种硬编码的方式容易写错，还有和上一条一样的问题，可维护性差，一旦请求参数名称发生变化，要修改的地方可能到处分散。

使用形如Config定义那样完整的实体类的几个问题：

1. 请求参数数据稍微复杂的结构，例如层次较深的XML格式数据，转换为标准形式的Java对象的属性值比较困难；
2. 查询操作时，返回的每行记录的多列字段可以转换为一个Map实例或一个标准Java对象，当多表连接查询时，如果查询字段包含多个表的字段，无法直接转换成与某个数据表对应的实体类型的实例，会漏掉其他表的字段值；
3. 标准Java对象的属性值转换成XMl或Json时，无法确定哪些属性值将显示为XMl或Json的元素，可能Java对象的某些属性值从来不会用到；
4. 关于对象序列化的问题，使用分布式缓存和远程方法调用时，传递的对象必须可序列化和反序列化，简单测试对比了一下标准Java对象、Map对象的序列化耗时。

Java标准对象定义

|  |
| --- |
| **import** java.io.Serializable;  **import** java.math.BigDecimal;  **import** java.util.Date;  **import** cn.com.ut.core.common.util.FileUtil;  **public** **class** SBean **implements** Serializable {  **private** **static** **final** **long** *serialVersionUID* = -2970387524689036304L;  **private** Integer iVar;  **private** String sVar;  **private** Date dVar;  **private** BigDecimal bdVar;  **public** Integer getiVar() {  **return** iVar;  }  **public** **void** setiVar(Integer iVar) {  **this**.iVar = iVar;  }  **public** String getsVar() {  **return** sVar;  }  **public** **void** setsVar(String sVar) {  **this**.sVar = sVar;  }  **public** Date getdVar() {  **return** dVar;  }  **public** **void** setdVar(Date dVar) {  **this**.dVar = dVar;  }  **public** BigDecimal getBdVar() {  **return** bdVar;  }  **public** **void** setBdVar(BigDecimal bdVar) {  **this**.bdVar = bdVar;  } |

各自序列化、反序列化5000次

|  |
| --- |
| @Test  **public** **void** testSerial() {  SBean sBean = **new** SBean();    sBean.setiVar(1);  sBean.setsVar("a");  sBean.setdVar(**new** Date());  sBean.setBdVar(**new** BigDecimal(1));  Map<String, Object> map = **new** HashMap<String, Object>();  map.put("iVar", 1);  map.put("sVar", "a");  map.put("dVar", **new** Date());  map.put("bdVar", **new** BigDecimal(1));  **long** s1 = System.*currentTimeMillis*();  **for** (**int** i = 0; i < 5000; i++) {  **byte**[] bs = FileUtil.*object2Byte*(sBean);  Object o = FileUtil.*byte2Object*(bs);  }  System.*out*.println("time1==" + (System.*currentTimeMillis*() - s1));    **long** s2 = System.*currentTimeMillis*();  **for** (**int** i = 0; i < 5000; i++) {  **byte**[] bs = FileUtil.*object2Byte*(map);  Object o = FileUtil.*byte2Object*(bs);  }  System.*out*.println("time2==" + (System.*currentTimeMillis*() - s2));  } |

耗时：

|  |
| --- |
| time1==500ms  time2==328ms |

|  |
| --- |
| **import** **static** cn.com.ut.biz.system.entities.DictType.\*; |

|  |
| --- |
| **package** cn.com.ut.biz.system.dao.impl;  **import** java.sql.Timestamp;  **import** java.util.List;  **import** java.util.Map;  **import** org.springframework.stereotype.Repository;  **import** cn.com.ut.biz.system.entities.DictType;  **import** cn.com.ut.core.common.util.CommonUtil;  **import** cn.com.ut.core.common.util.DateTimeUtil;  **import** cn.com.ut.core.dal.annotation.DAOMethod;  **import** cn.com.ut.core.dal.jdbc.JdbcOperationsImpl;  **import** cn.com.ut.core.dal.jdbc.PageBean;  **import** **static** cn.com.ut.biz.system.entities.DictType.\*;  @Repository  **public** **class** DictTypeDAOImpl **extends** JdbcOperationsImpl<DictType> {  **private** **static** **final** String[] *COLUMNS* = { *DICTTYPE\_CODE*, *DICTTYPE\_TEXT*, *GROUP\_CODE*, *IS\_USE*,  *SORT\_NUM*, *DICTTYPE\_DES* };  /\*\*  \* 插入一条记录  \*  \* **@param** vo  \*/  **public** **void** add(Map<String, Object> vo) {  Timestamp time = DateTimeUtil.*currentDateTime*();  add(**null**,  *COLUMNS*,  *NAMES*,  **new** Object[] { vo.get(*DICTTYPE\_CODE*), vo.get(*DICTTYPE\_TEXT*), vo.get(*GROUP\_CODE*),  vo.get(*IS\_USE*), vo.get(*SORT\_NUM*), vo.get(*DICTTYPE\_DES*),  CommonUtil.*getUUID*(), time, time, vo.get(*CREATE\_ID*), vo.get(*CREATE\_ID*) },  **null**);  }  /\*\*  \* 更新一条记录  \*  \* **@param** vo  \*/  **public** **void** update(Map<String, Object> vo) {  Timestamp time = DateTimeUtil.*currentDateTime*();  updateById(  **null**,  *COLUMNS*,  *NAMES\_UT\_UID*,  **new** Object[] { vo.get(*DICTTYPE\_CODE*), vo.get(*DICTTYPE\_TEXT*), vo.get(*GROUP\_CODE*),  vo.get(*IS\_USE*), vo.get(*SORT\_NUM*), vo.get(*DICTTYPE\_DES*), time,  vo.get(*CREATE\_ID*) }, (String) vo.get(*ID*));  }  /\*\*  \* 获取一条记录  \*  \* **@param** id  \* **@return**  \*/  **public** Map<String, Object> get(String id) {  **return** getById(**null**, **null**, *COLUMNS*, *NAMES*, id);  }  /\*\*  \* 删除一条记录  \*  \* **@param** id  \*/  **public** **void** delete(String id) {  deleteById(**null**, id);  }  /\*\*  \* 分页查询  \*  \* **@param** page  \* **@return**  \*/  **public** List<Map<String, Object>> query(PageBean page) {  **return** query(page, **null**, **null**, **null**, **null**, **null**, **null**);  }  /\*\*  \* 唯一性验证  \*  \* **@param** uniqueFieldValues  \* **@param** pkFieldNames  \* **@param** pkFieldValues  \* **@return**  \*/  @DAOMethod  **public** **boolean** checkUnique(Object[] uniqueFieldValues, String[] pkFieldNames,  Object[] pkFieldValues) {  **return** checkUnique(**null**, *ID*, **new** String[] { *DICTTYPE\_CODE* }, uniqueFieldValues,  pkFieldNames, pkFieldValues);  }  } |

* + 1. Service定义

同样的道理，在Service方法中，当对map集合进行put设值和get取值的时候，我们也尽量使用实体类的静态常量成员作为map的key键，可维护性好。

|  |
| --- |
| **package** cn.com.ut.biz.system.service.impl;  **import** java.util.List;  **import** java.util.Map;  **import** javax.annotation.Resource;  **import** org.springframework.stereotype.Service;  **import** cn.com.ut.biz.system.dao.impl.DictDataDAOImpl;  **import** cn.com.ut.biz.system.dao.impl.DictTypeDAOImpl;  **import** cn.com.ut.biz.system.entities.DictType;  **import** cn.com.ut.core.system.manager.RespResult;  /\*\*  \* 字典类型管理  \*  \* **@author** wuxiaohua  \* **@since** 2014-3-28  \*/  @Service  **public** **class** DictTypeServiceImpl {  @Resource  **private** DictTypeDAOImpl dao;  @Resource  **private** DictDataDAOImpl dictDataDAO;  /\*\*  \* 新建字典类型  \*  \*/  **public** **void** create(RespResult respResult) {  Map<String, Object> vo = respResult.getRequestObjects();  String sort = (String) vo.get(DictType.*SORT\_NUM*);  **if** (sort != **null**)  vo.put(DictType.*SORT\_NUM*, Integer.*parseInt*(sort));  **boolean** b = dao.checkUnique(**new** Object[] { vo.get(DictType.*DICTTYPE\_CODE*) }, **null**, **null**);  **if** (!b) {  respResult.addErrMsg("字典类型已存在！");  **return**;  }  dao.add(vo);  }  /\*\*  \* 更新字典类型  \*  \*/  **public** **void** update(RespResult respResult) {  Map<String, Object> vo = respResult.getRequestObjects();  String sort = (String) vo.get(DictType.*SORT\_NUM*);  **if** (sort != **null**)  vo.put(DictType.*SORT\_NUM*, Integer.*parseInt*(sort));  String id = (String) vo.get(DictType.*ID*);  String code = (String) vo.get(DictType.*DICTTYPE\_CODE*);  Map<String, Object> oldVO = dao.get(id);  String oldCode = (String) oldVO.get(DictType.*DICTTYPE\_CODE*);  **boolean** b = dao.checkUnique(**new** Object[] { code }, **new** String[] { DictType.*ID* },  **new** Object[] { id });  **if** (!b) {  respResult.addErrMsg("字典类型已存在！");  **return**;  }  **long** count = dictDataDAO.countByType(oldCode);  **if** (count > 0 && !oldCode.equals(code)) {  respResult.addErrMsg("该字典类型下存在字典项，不允许修改编码！");  **return**;  }  dao.update(vo);  }  /\*\*  \* 查询字典类型列表  \*  \*/  **public** **void** find(RespResult respResult) {  respResult.setFieldType(DictType.*SORT\_NUM* + "@@int");  respResult.getDictionary().put(DictType.*IS\_USE*, "IS\_TRUE");  respResult.getDictionary().put(DictType.*IS\_DEL*, "IS\_TRUE");  List<Map<String, Object>> voList = dao.query(respResult.getPageBean());  respResult.setItemList(voList);  }  /\*\*  \* 获取某字典类型详细信息  \*  \*/  **public** **void** getDetail(RespResult respResult) {  Map<String, Object> vo = dao.get(respResult.getParameter(DictType.*ID*));  respResult.setItem(vo);  }  **public** **void** delete(RespResult respResult) {  String id = respResult.getParameter(DictType.*ID*);  Map<String, Object> vo = dao.get(id);  **long** count = dictDataDAO.countByType((String) vo.get(DictType.*DICTTYPE\_CODE*));  **if** (count > 0) {  respResult.addErrMsg("该字典类型下存在字典项，不允许删除！");  **return**;  }  dao.delete(id);  }  } |

* + 1. 测试用例

|  |
| --- |
| <?xml version=*'1.0'* encoding=*'UTF-8'*?>  <!-- <req>  <call>  <lib>system/DictTypeService</lib>  <func>create</func>  </call>  <par>  <row>  <DICTTYPE\_CODE>CorpTrade</DICTTYPE\_CODE>  <DICTTYPE\_TEXT>公司行业</DICTTYPE\_TEXT>  <IS\_USE>Y</IS\_USE>  <SORT\_NUM>7</SORT\_NUM>  <DICTTYPE\_DES>公司行业领域</DICTTYPE\_DES>  <GROUP\_CODE>SERVER\_GROUP</GROUP\_CODE>  </row>  </par>  </req> -->  <!-- <req>  <call>  <lib>system/DictTypeService</lib>  <func>update</func>  </call>  <par>  <row>  <ID>268ca9ef1c364f79836f37d81b000156</ID>  <DICTTYPE\_CODE>CorpTrade</DICTTYPE\_CODE>  <DICTTYPE\_TEXT>公司行业</DICTTYPE\_TEXT>  <IS\_USE>Y</IS\_USE>  <SORT\_NUM>7</SORT\_NUM>  <DICTTYPE\_DES>公司行业领域3</DICTTYPE\_DES>  <GROUP\_CODE>SERVER\_GROUP</GROUP\_CODE>  </row>  </par>  </req> -->  <!-- <req>  <call>  <lib>system/DictTypeService</lib>  <func>getDetail</func>  </call>  <par>  <row>  <ID>268ca9ef1c364f79836f37d81b000156</ID>  </row>  </par>  </req> -->  <!-- <req>  <call>  <lib>system/DictTypeService</lib>  <func>delete</func>  </call>  <par>  <row>  <ID>268ca9ef1c364f79836f37d81b000156</ID>  </row>  </par>  </req> -->  <req>  <call>  <lib>system/DictTypeService</lib>  <func>find</func>  </call>  <par page=*"2"* pagesize=*"2"* total=*"0"*>  <where exp=*"1 # 2 # 3"*>  <field name=*"DICTTYPE\_CODE"* value=*"CorpTrade"* op=*"eq"* id=*"1"* />  <field name=*"IS\_USE"* value=*"Y"* op=*"eq"* id=*"2"* />  <field name=*"SORT\_NUM"* value=*"-1"* op=*"ne"* id=*"3"* />  </where>  <orderby>  <field name=*"SORT\_NUM"* type=*"1"* />  </orderby>  </par>  </req> |

1. **编程指南**

参考《智能餐饮平台JAVA基础服务框架编程指南(2015版)》

1. **API参考**

tuna\_api/index.html