**MyBetis3.2框架技术**

# 1.MyBetis介绍及基本环境搭建

## 1.1 MyBatis介绍

基于SQL语句的ORM(对象关系映射-Object Relational Mapping)框架，Apache--iBatis。代码库迁致Google Code--MyBatis.

Eclipse中xml自动提示:

|  |
| --- |
| 按ctrl键点击http://mybatis.org/dtd/mybatis-3-mapper.dtd，就会让你下载这个dtd到本机上  下载完成后，打开window–>Preferences–>XML–>XML catalog。  然后点击add，Location地方找到你下载好的dtd文件，key type用public-Key，key复制xml头上的-//mybatis.org//DTD Config 3.0//EN这一部分，然后确定保存 |

## 1.2 与Hibernate比较

a. 学习成本：简单易学，较接近JDBC

b. 程序灵活性：直接用SQL，灵活性高

c. 程序执行效率高

d. 可移植性：hibernate较好（与数据库关联在配置中完成，HQL语句与数据库无关）

## 1.3 适用场合

以下场合不建议：

1. 需要支持多种数据库或数据库有移植要求
2. 完全动态SQL，例如：字段都要动态生成
3. 使用的不是关系型数据库

## 1.4 开发步骤（推荐）

|  |
| --- |
| **新建JAVA项目或WEB项目**  **部署jar包（包括数据库驱动包）：下载地址http://code.google.com/p/mybatis**  **编写主配置文件**  **创建数据库及表（如已创建，可省略）**  **创建实体类及SQL映射文件**  **编写数据库接口及实现**  **编写测试类及测试** |

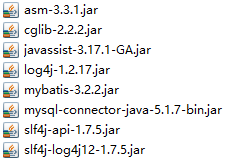
## 1.5 开发示例

**1.5.1 新建项目**

新建java项目或web 项目。

**1.5.2 导入JAR包**

导入mybatis和数据库驱动包、日志包（配置日志配置文件）。



**1.5.3 创建myBatis-config.xml**

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8" ?>  <!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN" "http://mybatis.org/dtd/mybatis-3-config.dtd">  <!--可设置多个运行环境，满足不同需要，如 开发、测试 -->  <configuration>  <!--environments指定数据源环境，default指的是使用哪个数据源 -->  <environments default="development">    <!--environment定义数据源的信息 -->  <environment id="development">    <!-- type="JDBC"表示事务由jdbc连接管理，  type="MANAGED"表示事务由容器来管理 -->  <transactionManager type="JDBC" />    <!-- type="POOLED"表示使用连接池，  type="UNPOOLED"表示不使用连接池 -->  <dataSource type="POOLED">  <property name="driver" value="com.mysql.jdbc.Driver" />  <property name="url" value="jdbc:mysql://localhost:3306/mybatis?useUnicode=true&amp;characterEncoding=UTF-8" />  <property name="username" value="root" />  <property name="password" value="root" />  </dataSource>    </environment>  </environments>    <!-- 加入映射文件信息 -->  <mappers>  <mapper resource="cn/it/entity/DeptMapper.xml" />  </mappers>  </configuration> |

**1.5.4 创建数据库及表(在navicat中操作)**

drop database if exists mybatis;

create database mybatis CHARACTER SET UTF8;

use mybatis;

create table dept(

dept\_id int primary key auto\_increment,

dept\_name varchar(50),

dept\_address varchar(50)

);

insert into dept(dept\_name, dept\_address) values('研发部一部','广州');

insert into dept(dept\_name, dept\_address) values('研发部二部','广州');

insert into dept(dept\_name, dept\_address) values('研发部三部','深圳');

select \* from dept;

**1.5.5 创建实体类：Dept.java**

|  |
| --- |
| package cn.it.entity;  import java.io.Serializable;  public class Dept implements Serializable {  private static final long serialVersionUID = -2525513725816258556L;  private Integer deptId;//部门编号  private String deptName;//部门名称  private String deptAddress;//部门地址  public Integer getDeptId() {  return deptId;  }  public void setDeptId(Integer deptId) {  this.deptId = deptId;  }  public String getDeptName() {  return deptName;  }  public void setDeptName(String deptName) {  this.deptName = deptName;  }  public String getDeptAddress() {  return deptAddress;  }  public void setDeptAddress(String deptAddress) {  this.deptAddress = deptAddress;  }  @Override  public String toString() {  return "Dept [deptId=" + deptId + ", deptName=" + deptName  + ", deptAddress=" + deptAddress + "]";  }  } |

**1.5.6 创建SQL映射文件及修改主配置文件**

SQL映射文件：DeptMapper.xml（可以参考用户手册）

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8" ?>  <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  <!-- 命名空间可任选命名，但最好要定义一定规则，便于后继的使用 -->  <mapper namespace="cn.it.entity.DeptMapper">  <!-- type指定的是对应的实体类 -->  <!-- 一般在查询时用-->  <resultMap type="cn.it.entity.Dept" id="deptResultMap">    <!-- id用来配置表的主键与类的属性的映射关系 ,  column指定表字段名; property指定类属性名-->  <id column="dept\_id" property="deptId"/>    <!-- result用来配置 普通字段与类属性的映射关系 ,column指定的是表的字段名; property指定的是类的属性名-->  <result column="dept\_name" property="deptName"/>  <result column="dept\_address" property="deptAddress"/>    </resultMap>    <!-- 添加一第记录 ; 定义插入的sql语句，通过命名空间+id方式被定位 -->  <insert id="insertDept" parameterType="cn.it.entity.Dept">  <!-- #{} 用来获取传过来的参数 -->  insert into dept(dept\_name,dept\_address) values(#{deptName},#{deptAddress})  </insert>  </mapper> |

**1.5.7 编写数据库操作**

包括操作接口及实现，接口略，实现类为：DeptDaoImpl.jav

package cn.it.dao.impl;

import java.io.IOException;

import java.io.Reader;

import org.apache.ibatis.io.Resources;

import org.apache.ibatis.session.SqlSession;

import org.apache.ibatis.session.SqlSessionFactory;

import org.apache.ibatis.session.SqlSessionFactoryBuilder;

import cn.it.entity.Dept;

public class DeptDaoImpl {

/\*

\* 1. 读取配置文件信息

\* 2. 构建session工厂

\* 3. 创建session

\* 4. 开启事务（可选）(默认开启)

\* 5.处理数据

\* 6.提交、回滚事务

\* 7.关闭session

\*/

public int save(Dept dept){

int i =0;

String path="myBatis-config.xml";

SqlSession session = null;

Reader reader = null;

try {

//1.获取配置文件的信息

reader = Resources.getResourceAsReader(path);

//2.构建sessionfactory

SqlSessionFactory sessionFactory = new SqlSessionFactoryBuilder().build(reader);

//3.创建session

session = sessionFactory.openSession();

//4.启用事务，默认已启用

//5.执行数据处理,第一个参数用“命名空间+sql id"来定位sql,

//第二个参数用来给sql传参数

i = session.insert("cn.it.entity.DeptMapper.insertDept", dept);

//6.提交事务

session.commit();

} catch (Exception e) {

e.printStackTrace();

session.rollback();

}finally{

if(reader!=null){

try {

reader.close();

} catch (IOException e) {

e.printStackTrace();

}

}

if(session!=null){

session.close();

}

}

return i;

}

}

**1.5.8 编写测试类**

需要导入junit包

|  |
| --- |
| package cn.it.test;  import static org.junit.Assert.\*;  import org.junit.AfterClass;  import org.junit.BeforeClass;  import org.junit.Test;  import cn.it.dao.impl.DeptDaoImpl;  import cn.it.entity.Dept;  public class TestDeptDaoImpl {  private static DeptDaoImpl deptDaoImpl;  @BeforeClass  public static void setUpBeforeClass() throws Exception {  deptDaoImpl = new DeptDaoImpl();  }  @AfterClass  public static void tearDownAfterClass() throws Exception {  deptDaoImpl = null;  }  @Test  public void test() {  Dept dept = new Dept();  dept.setDeptName("综合部");  dept.setDeptAddress("广州天河");  System.out.println("受影响 的行数："+deptDaoImpl.save(dept));  }  } |

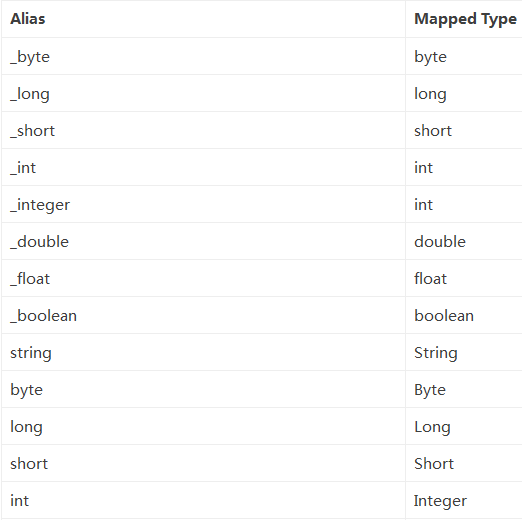
# 2.基本的CRUD操作

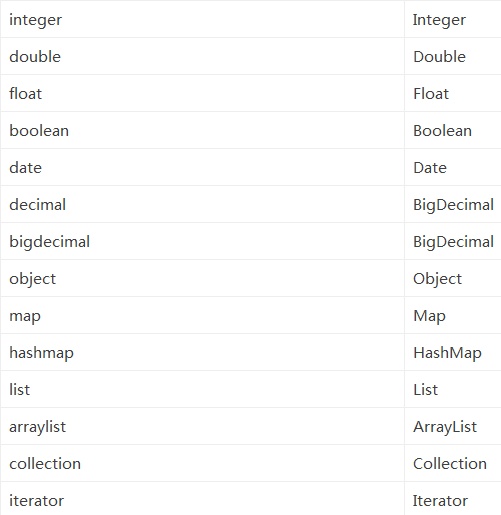
## 2.1 准备工作（继续使用前面的库表和代码）

## 2.2 别名与自定义别名

**2.2.1 内置别名**

对常用的 java 类型，已经内置了一些别名支持。不区分大小写.





**2.2.2 自定义别名**

在myBatis的主配置文件给cn.itcast.entity.Dept类创建别名Dept,

后继的DeptMapper.xml配置文件中可使用别名

* mybatis-config.xml

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-config.dtd">  <configuration>  <!-- 通过别名简化对类的使用 -->  <typeAliases>  <typeAlias type=*"cn.it.entity.Dept"* alias=*"Dept"*/>  </typeAliases>    <!--environments指定数据源环境，default指的是使用哪个数据源 -->  <environments default=*"development"*>  <!--environment定义数据源的信息 -->  <environment id=*"development"*>  <!-- type="JDBC"表示事务由jdbc连接管理，  type="MANAGED"表示事务由容器来管理 -->  <transactionManager type=*"JDBC"*></transactionManager>    <!-- type="POOLED"表使用连接池，type="UNPOOLED"表示不使用连接池-->  <dataSource type=*"POOLED"*>  <property name=*"driver"* value=*"com.mysql.jdbc.Driver"* />  <property name=*"url"* value=*"jdbc:mysql://localhost:3306/mybatis?useUnicode=true&amp;*  *characterEncoding=UTF-8"* />  <property name=*"username"* value=*"root"* />  <property name=*"password"* value=*"123"* />  </dataSource>  </environment>  </environments>    <!-- 加入映射文件信息 -->  <mappers>  <mapper resource=*"cn/it/entity/DeptMapper.xml"*/>  </mappers>  </configuration> |

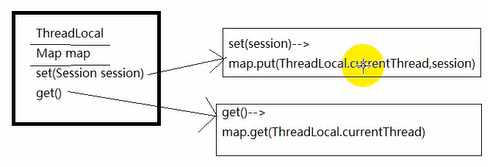
* DeptMapper.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  <mapper namespace="cn.it.entity.DeptMapper">  <!--resultMap用于查询，可把查询后字段值封装到对应类的属性， type指定的是对应的实体类 -->  <resultMap type="Dept" id="deptResultMap">  <!-- id用来配置表的主键与类的属性的映射关系 ,column指定的是表的字段名；  property指定的是类的属性名-->  <id column="dept\_id" property="deptId"/>    <result column="dept\_name" property="deptName"/>  <result column="dept\_address" property="deptAddress"/>  </resultMap>    <!-- 添加一第记录 ; 定义插入的sql语句，通过命名空间+id方式被定位 -->  <insert id="insertDept" parameterType="Dept">  <!-- #{} 用来获取传过来的参数 -->  insert into dept(dept\_name,dept\_address) values(#{deptName},#{deptAddress})  </insert>    <!-- 根据部门编号修改部门信息 -->  <update id="updateDept" parameterType="Dept">  update dept set dept\_name=#{deptName}, dept\_address=#{deptAddress}  where dept\_id = #{deptId}  </update>    <!--根据部门编号删除部门 -->  <delete id="deleteDept" parameterType="integer">  delete from dept where dept\_id = #{deptId}  </delete>    <!-- 查询单个部门信息 -->  <select id="selectDept" parameterType="integer" resultMap="deptResultMap">  select dept\_id,dept\_name,dept\_address from dept where dept\_id = #{deptId}  </select>    <!-- 根据部门地址查询多个部门信息 -->  <!-- 如果返回的是list，resultMap指定的值是list集合里元素的类型-->  <select id="selectList" parameterType="string" resultMap="deptResultMap">  select dept\_id,dept\_name,dept\_address from dept where dept\_address like #{deptAddress}  </select>  </mapper> |

## 2.3 MyBatisUtil工具类

* MybatisSessionFactory.java

|  |
| --- |
| package cn.it.dao.utils;  import java.io.IOException;  import java.io.Reader;  import org.apache.ibatis.io.Resources;  import org.apache.ibatis.session.SqlSession;  import org.apache.ibatis.session.SqlSessionFactory;  import org.apache.ibatis.session.SqlSessionFactoryBuilder;  public class MybatisSessionFactory {  private static final ThreadLocal<SqlSession> threadLocal = new ThreadLocal<SqlSession>();    private static SqlSessionFactory sqlSessionFactory;    private static final String CONFIG\_FILE\_LOCATION = "mybatis-config.xml";    private MybatisSessionFactory(){  }    static {  buildSessionFactory();  }    public static SqlSessionFactory getSessionFactory() {  return sqlSessionFactory;  }    public static void buildSessionFactory() {  Reader reader = null;    try {  reader = Resources.getResourceAsReader(CONFIG\_FILE\_LOCATION);  sqlSessionFactory = new SqlSessionFactoryBuilder().build(reader);  } catch (IOException e) {  System.err.println("%%%% Error Creating SessionFactory %%%%");  e.printStackTrace();  } finally {  try {  reader.close();  } catch (IOException e) {  e.printStackTrace();  }  }  }    public static SqlSession getSession() throws Exception {  SqlSession sqlSession = (SqlSession)threadLocal.get();    if(sqlSession == null) {  if(sqlSessionFactory == null) {  buildSessionFactory();  }    sqlSession = (sqlSessionFactory != null) ? sqlSessionFactory.openSession() : null;  threadLocal.set(sqlSession);  }  return sqlSession;  }    public static void closeSession() {  SqlSession sqlSession = (SqlSession)threadLocal.get();  threadLocal.set(null);    if(sqlSession != null) {  sqlSession.close();  }  }  } |



## 2.4 CRUD操作

**2.4.1 DeptDaoImpl.java**

|  |
| --- |
| package cn.it.dao.impl;  import java.util.List;  import org.apache.ibatis.session.SqlSession;  import cn.it.dao.utils.MybatisSessionFactory;  import cn.it.entity.Dept;  public class DeptDaoImpl {    SqlSession sqlSession;    /\*  \* 1.读取配置文件信息  \* 2.构建session工厂  \* 3.创建session  \* 4.开启事务（可选）  \* 5.处理数据  \* 6.提交、回滚事务  \* 7.关闭session  \*/  //增  public int save(Dept dept) {  int i = 0;  try {  sqlSession = MybatisSessionFactory.getSession();    i = sqlSession.insert("cn.it.entity.DeptMapper.insertDept", dept);    //提交事务  sqlSession.commit();    } catch (Exception e) {  e.printStackTrace();  sqlSession.rollback();  } finally {  MybatisSessionFactory.closeSession();  }  return i;  }    //修改  public int update(Dept dept) {  int i = 0;    try {  sqlSession = MybatisSessionFactory.getSession();  i = sqlSession.update("cn.it.entity.DeptMapper.updateDept",dept);  sqlSession.commit();  } catch (Exception e) {  e.printStackTrace();  sqlSession.rollback();  } finally {  MybatisSessionFactory.closeSession();  }  return i;  }    //删除  public int delete(Integer id) {  int i = 0;  try {  sqlSession = MybatisSessionFactory.getSession();  i = sqlSession.update("cn.it.entity.DeptMapper.deleteDept", id);  sqlSession.commit();  } catch (Exception e) {  e.printStackTrace();  sqlSession.rollback();  } finally {  MybatisSessionFactory.closeSession();  }    return i;  }    //根据部门编号查询单个部门  public Dept selectOne(Integer id){  Dept dept =null;  try {  sqlSession=MybatisSessionFactory.getSession();  dept = sqlSession.selectOne("cn.it.entity.DeptMapper.selectDept",id);  } catch (Exception e) {  e.printStackTrace();  }finally{  try {  MybatisSessionFactory.closeSession();  } catch (Exception e) {  e.printStackTrace();  }  }    return dept;  }    //根据部门地址查询多个部门  public List<Dept> selectList(String deptAddress){  List<Dept> depts =null;  try {  sqlSession=MybatisSessionFactory.getSession();  depts = sqlSession.selectList("cn.it.entity.DeptMapper.selectList",deptAddress);  } catch (Exception e) {  e.printStackTrace();  }finally{  try {  MybatisSessionFactory.closeSession();  } catch (Exception e) {  e.printStackTrace();  }  }    return depts;  }  } |

**2.4.2 TestDeptDaoImpl.java(junit测试)**

|  |
| --- |
| package cn.it.test;  import java.util.List;  import org.junit.AfterClass;  import org.junit.BeforeClass;  import org.junit.Test;  import cn.it.dao.impl.DeptDaoImpl;  import cn.it.entity.Dept;  public class TestDeptDaoImpl {  private static DeptDaoImpl deptDaoImpl;  @BeforeClass  public static void setUpBeforeClass() throws Exception {  deptDaoImpl = new DeptDaoImpl();  }  @AfterClass  public static void tearDownAfterClass() throws Exception {  deptDaoImpl = null;  }  @Test  public void testSave() {  Dept dept = new Dept();  dept.setDeptName("财务部2");  dept.setDeptAddress("广州天河");  System.out.println("save受影响 的行数："+deptDaoImpl.save(dept));  }    @Test  public void testUpdate() {  Dept dept = new Dept();  dept.setDeptId(6);  dept.setDeptName("秘书部3");  dept.setDeptAddress("深圳");  System.out.println("update受影响 的行数："+deptDaoImpl.update(dept));  }    @Test  public void testDelete() {  /\*Dept dept = new Dept();  dept.setDeptId(6);  dept.setDeptName("秘书部");  dept.setDeptAddress("深圳");\*/  System.out.println("delete受影响 的行数："+deptDaoImpl.delete(8));  }    @Test  public void testSelectOne() {  System.out.println("id=6的部门信息："+deptDaoImpl.selectOne(6));  }    @Test  public void testSelectList() {  List<Dept> depts = deptDaoImpl.selectList("%广%");  for (Dept dept : depts) {  System.out.println("全部部门信息："+dept);  }  }  } |

**2.4.3[补充]查询操作（返回单条记录）**

配置deptMapper.xml文件的resultMap元素及SQL查询语句

<!-- 表字段和实体属性命名一致时, 属性可不配置 -->

<resultMap id=*"deptResultMap"* type=*"Dept"*>

<id property=*"deptId"* column=*"dept\_id"*/>

<result property=*"deptName"* column=*"dept\_name"*/>

<result property=*"deptAddress"* column=*"dept\_address"*/>

</resultMap>

<!—省略其它配置信息 -->

<!—返回单条记录，表字段和对应实体属性命名一致时，可不用resultMap属性配置，直接使用resultType="返回的全类名或别名"，建议使用前者;查询结果为所有字段时，也可以用\*表示 -->

<select id=*"selectOne"* parameterType=*"int"* resultMap=*"deptResultMap"* >

select dept\_id, dept\_name from dept where dept\_id=#{deptId}

</select>

**2.4.4[补充]查询操作（返回多条记录）**

修改配置文件deptMapper.xml，添加

<!-- 返回多条记录，返回结果配置的不是集合类型，而是集合元素的类型；

参数也可通过Map等方式封装 -->

<select id=*"selectList"* parameterType=*"Map"* resultMap=*"deptResultMap"*>

select \* from dept where dept\_name like #{deptName}

</select>

修改DeptDaoImpl.java，添加selectList方法：

**public** List<Dept> selectList(Map map){

List<Dept> depts=**null**;

**try** {

session=MyBatisUtil.*getSession*();

depts=session.selectList("cn.itcast.entity.DeptMapper.selectList",map);

} **catch** (Exception e) {

e.printStackTrace();

}**finally**{

MyBatisUtil.*closeSession*();

}

**return** depts;

}

测试类代码：

@Test

**public** **void** testSelectList() {

Map map=**new** HashMap();

map.put("deptName", "%研%");

List<Dept> depts=*deptDaoImpl*.selectList(map);

**for**(Dept dept:depts){

System.*out*.println("dept:"+dept);

}

}

# 3.动态SQL操作

## 3.1 准备工作

创建表及库，实体类，配置文件（参考上章节内容），以下为建表和库的SQL：

drop database if exists mybatis;

create database mybatis CHARACTER SET UTF8;

use mybatis;

create table dept(

dept\_id int primary key auto\_increment,

dept\_name varchar(50),

dept\_address varchar(50)

);

insert into dept(dept\_name,dept\_address) values('研发部一部','广州');

insert into dept(dept\_name,dept\_address) values('研发部二部','广州');

insert into dept(dept\_name,dept\_address) values('研发部三部','深圳');

select \* from dept;

## 3.2 IF语句

修改配置文件deptMapper.xml，添加

<!-- 动态IF条件 -->

<select id=*"selectListUseIf"* parameterType=*"Dept"* resultMap=*"deptResultMap"*>

select \* from dept where 1=1

<if test=*"deptId!=null"*>

and dept\_id=#{deptId}

</if>

<if test=*"deptName!=null"*>

and dept\_name=#{deptName}

</if>

<if test=*"deptAddress!=null"*>

and dept\_address=#{deptAddress}

</if>

</select>

修改DeptDaoImpl.java，添加selectListUseIf方法：

//根据参数使用配置文件的IF语句自动填充查询的过滤条件

**public** List<Dept> selectListUseIf(Dept dept){

List<Dept> depts=**null**;

**try** {

session=MyBatisUtil.*getSession*();

depts=session.selectList("cn.itcast.entity.DeptMapper.selectListUseIf",dept);

} **catch** (Exception e) {

e.printStackTrace();

}**finally**{

MyBatisUtil.*closeSession*();

}

**return** depts;

}

## 3.3 WHERE语句

修改配置文件deptMapper.xml，添加

<!--动态Where条件 ,一般也要与if结合使用,与纯if比较,省略了where 1=1-->

<select id=*"selectListUseWhere"* parameterType=*"Dept"*

resultMap=*"deptResultMap"*>

select \* from dept

<where>

<if test=*"deptId!=null"*>

and dept\_id=#{deptId}

</if>

<if test=*"deptName!=null"*>

and dept\_name=#{deptName}

</if>

<if test=*"deptAddress!=null"*>

and dept\_address=#{deptAddress}

</if>

</where>

</select>

|  |  |
| --- | --- |
| <where>相当于:   |  | | --- | | <trim prefix="WHERE" prefixOverrides="AND |OR ">  ...  </trim> |   意思是: 当WHERE后紧随AND或OR时，就去除AND或者OR。 |

## 3.4 choose(when,otherwise)语句

修改配置文件deptMapper.xml，添加

<select id=*"selectListUseChoose"* parameterType=*"Dept"*

resultMap=*"deptResultMap"*>

select \* from dept where 1=1

<choose>

<when test=*"deptId!=null"*>and dept\_id=#{deptId}</when>

<when test=*"deptName!=null"*>and dept\_name=#{deptName}</when>

<when test=*"deptAddress!=null"*>and dept\_address=#{deptAddress}</when>

<otherwise>and !1 = 1</otherwise>

</choose>

</select>

* 例子:

|  |
| --- |
| <select id="findActiveBlogLike"  resultType="Blog">  SELECT \* FROM BLOG WHERE state = ‘ACTIVE’  <choose>  <when test="title != null">  AND title like #{title}  </when>  <when test="author != null and author.name != null">  AND author\_name like #{author.name}  </when>  <otherwise>  AND featured = 1  </otherwise>  </choose>  </select> |

|  |
| --- |
| 分析：  当title和author都不为null时, 选择二选一（前者优先）  如都为null, 就选择 otherwise中的  如tilte和author只有一个不为null, 就选择不为null的那个。 |

## 3.5 SET语句

修改配置文件deptMapper.xml，添加

<!--动态set语句可以用来更新数据 -->

<update id=*"updateUseSet"* parameterType=*"Dept"*>

update dept

<set>

<if test=*"deptName!=null"*>dept\_name=#{deptName},</if>

<if test=*"deptAddress!=null"*>dept\_address=#{deptAddress},</if>

</set>

where dept\_id=#{deptId}

</update>

* 例子:

|  |
| --- |
| <update id="updateUser" parameterType="com.dy.entity.User">  update user set  <if test="name != null">  name = #{name},  </if>  <if test="password != null">  password = #{password},  </if>  <if test="age != null">  age = #{age}  </if>  <where>  <if test="id != null">  id = #{id}  </if>  and deleteFlag = 0;  </where>  </update> |

|  |  |  |
| --- | --- | --- |
| 问题又来了：“如只有name不为null, SQL就成了 update set name = #{name}**,** where ........ ? name后面那逗号会导致出错啊！”  这时，就可用set 标签。下面是set标签改造后：   |  | | --- | | <update id="updateUser" parameterType="com.dy.entity.User">  update user  <set>  <if test="name != null">name = #{name},</if>  <if test="password != null">password = #{password},</if>  <if test="age != null">age = #{age},</if>  </set>  <where>  <if test="id != null">  id = #{id}  </if>  and deleteFlag = 0;  </where>  </update> |   <set>相当于   |  | | --- | | <trim prefix="SET" suffixOverrides=",">  ...  </trim> |   注: WHERE是使用的 prefixOverrides（前缀）， SET是使用的 suffixOverrides （后缀） |

## 3.6 ForEach语句

修改配置文件deptMapper.xml，添加

<!-- 定义根据多个部门ID查询部门相关部门信息的SQL语句 ,resultMap的值是指集合里元素的类型,parameterType不用指定 -->

<select id=*"selectListUseForeach"* resultMap=*"deptResultMap"*>

select \* from dept where dept\_id in

<!-- collection＝"array或list",array用来对应参数为数组，list对应集合 -->

<foreach collection=*"array"* item=*"deptId"* open=*"("* separator=*","* close=*")"*>

#{deptId}

</foreach>

</select>

* 例子:

java中有for循环， mybatis中有foreach, 可通过它实现循环，循环对象是java容器和数组。

foreach元素属性: item，index，collection，open，separator，close。

分别代表：

|  |
| --- |
| item表集合中每个元素进行迭代时的别名，  index表在迭代过程中，每次迭代到的位置，  collection: list array map  open表该语句以什么开始，  separator表在每次进行迭代间以什么符号作为分隔符，  close表示以什么结束 |

1. 传入参数为list时

|  |
| --- |
| <select id="selectPostIn" resultType="domain.blog.Post">  SELECT \*  FROM POST P  WHERE ID in  <foreach item="item" index="index" collection="list"  open="(" separator="," close=")">  #{item}  </foreach>  </select> |

将一个 List 实例或数组作为参数对象传给 MyBatis，MyBatis 会自动将它包装在一个 Map 中并以名称为键。

List 实例将会以“list”作为键，而数组实例的键将是“array”。(固定值)

当循环的对象为map的时候，index其实就是map的key。

2. 传入参数为Array时

|  |
| --- |
| 对应的Dao中的Mapper文件是：  public List<User> selectByIds(int[] ids);  xml文件代码片段：  <select id="selectByIds" resultType="com.wuuushao.pojo.User">  select \* from user where id in  <foreach collection="array" index="index" item="item" open="(" separator="," close=")">  #{item}  </foreach>  </select> |

3. 传入参数为Map

|  |  |  |
| --- | --- | --- |
| 对应的Dao中的Mapper文件是：   |  | | --- | | public List<User> selectByIds(Map<String, Object> params); |   xml文件代码片段：   |  | | --- | | <select id="selectByIds" resultType="com.wuuushao.pojo.User">  select \* from user where id in  <foreach collection="ids" index="index" item="item" open="(" separator="," close=")">  #{item}  </foreach>  </select> |   map时需注意：collection的值“ids”是存储在map中的key（比如：map.put("ids",ids)）; |

## 3.7 include语句

修改配置文件deptMapper.xml，添加

<!-- 使用include语句动态插入表的字段及对应的值 -->

<sql id=*"key"*>

<!--suffixOverrides="," 可以忽略最后“，”号 -->

<trim suffixOverrides=*","*>

<if test=*"deptName!=null"*>

dept\_name,

</if>

<if test=*"deptAddress!=null"*>

dept\_address,

</if>

</trim>

</sql>

<sql id=*"value"*>

<trim suffixOverrides=*","* >

<if test=*"deptName!=null"*>

#{deptName},

</if>

<if test=*"deptAddress!=null"*>

#{deptAddress},

</if>

</trim>

</sql>

<insert id=*"insertUseInclude"* parameterType=*"Dept"*>

insert into dept(<include refid=*"key"* />)

values(<include refid=*"value"*/>

)

</insert>

* 例子:

|  |  |
| --- | --- |
| <sql>封装SQL语句, <include>调用  如:   |  | | --- | | <sql id="columns">  id, name, address  </sql>  <select ....>  select <include refid="columns"/> from user  </select> | |

## 练习

1.完成环境的配置及部门信息的添加

2.解决sqlsession线程安全问题（工具类封装）

3.完成单个部门查询和多个部门查询

4.根据指定部门编号（列表）修改部门信息地址为“武汉”

# 4．关联查询

## 4.1 准备工作

**4.1.1 创建项目并导入包**

略

**4.1.2 配置myBatis-config.xml**

与前章节一样，这里略

**4.1.3 创建数据库及表**

drop database if exists mybatis;

create database mybatis CHARACTER SET UTF8;

use mybatis;

create table dept(

dept\_id int primary key auto\_increment,

dept\_name varchar(50),

dept\_address varchar(50)

);

create table emp(

emp\_id varchar(18) primary key,

emp\_name varchar(50),

emp\_sex char(1),

dept\_id int

);

insert into dept(dept\_name,dept\_address) values('研发部一部','广州');

insert into dept(dept\_name,dept\_address) values('研发部二部','广州');

insert into dept(dept\_name,dept\_address) values('研发部三部','深圳');

insert into emp(emp\_id,emp\_name,emp\_sex,dept\_id) values('44152199507052110','张大',"男","1");

insert into emp(emp\_id,emp\_name,emp\_sex,dept\_id) values('44152199507052111','张一',"女","1");

insert into emp(emp\_id,emp\_name,emp\_sex,dept\_id) values('44152199507052112','张二',"男","1");

select \* from dept;

select \* from emp;

## 4.2 [\*\*]基于association查询（用于多对一或一对一）

**4.2.1 创建实体类：Dept.java/Emp.java**

部门实体类：Dept.java

**public** **class** Dept **implements** Serializable{

**private** String deptAddress;

**private** String deptName;

**private** Integer deptId;

省略set和get方法

员工实体类：Emp.java

**public** **class** Emp **implements** Serializable{

**private** String empId;

**private** String empName;

**private** String empSex;

**private** Dept dept;

省略set和get方法

**4.2.2 配置DeptMapper.xml/EmpMapper.xml**

（重点加入级联的查询语句），并映射文件信息到mybatis-config.xml中：

DeptMapper.xml

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace=*"cn.itcast.entity.DeptMapper"*>

<!-- 表字段和对应实体属性命名一致时可以不配置 -->

<resultMap id=*"deptResultMap"* type=*"Dept"*>

<id property=*"deptId"* column=*"dept\_id"* />

<result property=*"deptName"* column=*"dept\_name"* />

<result property=*"deptAddress"* column=*"dept\_address"* />

</resultMap>

</mapper>

EmpMapper.xml,配置多对一的关联

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace=*"cn.itcast.entity.EmpMapper"*>

<!-- 表字段和对应实体属性命名一致时可不配置 -->

<resultMap id=*"empResultMap"* type=*"Emp"*>

<id property=*"empId"* column=*"emp\_id"* />

<result property=*"empName"* column=*"emp\_name"* />

<result property=*"empSex"* column=*"emp\_sex"* />

<!-- association配置多对一关联 -->

<association property=*"dept"* column=*"dept\_id"* javaType=*"Dept"*

resultMap=*"cn.itcast.entity.DeptMapper.deptResultMap"* />

</resultMap>

<!--根据部门名称查询员工（包括员工所在部门）信息 -->

<select id=*"selectEmpDeptList"* parameterType=*"Emp"* resultMap=*"empResultMap"*>

<!-- 访问emp.dept.deptName, 前面不用写emp, 直接写 dept.deptName-->

select e.\*,d.\* from emp e inner join dept d on

e.dept\_id=d.dept\_id where d.dept\_name like #{dept.deptName}

</select>

</mapper>

**4.2.3 配置文件myBatis-config.xml**

<!-- 通过别名简化对类的使用 -->

<typeAliases>

<typeAlias type=*"cn.itcast.entity.Dept"* alias=*"Dept"* />

<typeAlias type=*"cn.itcast.entity.Emp"* alias=*"Emp"* />

</typeAliases>

…….

<!--导入SQL映射文件 -->

<mappers>

<mapper resource=*"cn/itcast/entity/DeptMapper.xml"* />

<mapper resource=*"cn/itcast/entity/EmpMapper.xml"* />

</mappers>

**4.2.4 编写EmpDaoImpl.java实现查询**

**public** **class** EmpDaoImpl {

SqlSession session;

**public** List<Emp> selectEmpDeptList(Emp emp){

List<Emp> emps=**null**;

**try**{

session=MyBatisUtil.*getSession*();

emps=session.selectList("cn.itcast.entity.EmpMapper.selectEmpDeptList",emp);

//session.commit();

}**catch** (Exception e) {

// **TODO**: handle exception

e.printStackTrace();

//session.rollback();

}**finally**{

MyBatisUtil.*closeSession*();

}

**return** emps;

}

}

**4.2.5 编写测试类**

EmplTest.java

@Test

**public** **void** selectEmpDeptList() {

Emp emp=**new** Emp();

Dept dept=**new** Dept();

dept.setDeptName("%研%");

emp.setDept(dept);

List<Emp> emps=*empDaoImpl*.selectEmpDeptList(emp);

**for**(Emp emp1:emps){

System.*out*.println("emp="+emp1);

//System.*out*.println("dept＝"+emp1.getDept());

}

}

## 4.3 [\*\*] 基于collection查询（用于一对多或多对多）

**4.3.1 编写 Dept.java/Emp.java实体类**

Dept.java

**public** **class** Dept **implements** Serializable{

**private** String deptAddress;

**private** String deptName;

**private** Integer deptId;

**private** List<Emp> emps;

//省略set和get方法

Emp.java

**public** **class** Emp **implements** Serializable{

**private** String empId;

**private** String empName;

**private** String empSex;

//省略set和get方法

**4.3.2 配置DeptMapper.xml**

DeptMapper.xml文件，配置resultMap（重点是collection配置）和查询SQL语句：

…………

<mapper namespace=*"cn.itcast.entity.DeptMapper"*>

<!-- 表字段和对应实体属性命名一致时可以不配置 -->

<resultMap id=*"deptResultMap"* type=*"Dept"*>

<id property=*"deptId"* column=*"dept\_id"* />

<result property=*"deptName"* column=*"dept\_name"* />

<result property=*"deptAddress"* column=*"dept\_address"* />

<!-- collection中resultMap引用的是其它文件的map 需要命名空间+id,例

如：cn.itcast.entity.EmpMapper.empResultMap -->

<collection property=*"emps"* ofType=*"Emp"*

resultMap=*"cn.itcast.entity.EmpMapper.empResultMap"*/>

</resultMap>

<select id=*"selectDeptEmpList"* parameterType=*"Dept"*

resultMap=*"deptResultMap"*>

select d.\*, e.\* from dept d inner join emp e on d.dept\_id=e.dept\_id where

d.dept\_name like #{deptName}

</select>

</mapper>

**4.3.3 配置EmpMapper.xml文件**

不用配置对一关联

……

<mapper namespace=*"cn.itcast.entity.EmpMapper"*>

<!-- 表字段和对应实体属性命名一致时可以不配置 -->

<resultMap id=*"empResultMap"* type=*"Emp"*>

<id property=*"empId"* column=*"emp\_id"* />

<result property=*"empName"* column=*"emp\_name"* />

<result property=*"empSex"* column=*"emp\_sex"* />

</resultMap>

</mapper>

**4.3.4 编写数据库操作类DeptDaoImpl.java**

**public** **class** DeptDaoImpl {

//同时查询部门及部门员工信息

**public** List<Dept> selectDeptEmpList(Dept dept){

SqlSession session=**null**;

List<Dept> deps=**null**;

**try**{

session=MyBatisUtil.*getSession*();

deps=session.selectList("cn.itcast.entity.DeptMapper.selectDeptEmpList",dept);

session.commit();

}**catch** (Exception e) {

// **TODO**: handle exception

e.printStackTrace();

session.rollback();

}**finally**{

MyBatisUtil.*closeSession*();

}

**return** deps;

}

}

**4.3.5 编写测试类，**

@Test

**public** **void** selectDeptEmpList() {

Dept paramDept = **new** Dept();

paramDept.setDeptName("%研%");

List<Dept> depts = *deptDaoImpl*.selectDeptEmpList(paramDept);

**for** (Dept dept : depts) {

System.*out*.println("dept:" + dept);

}

}

## 4.4 [\*\*] 一对多双向关联查询示例【实质:4.2 4.3的组合】

**4.4.1 编写实体类:Dept.java/Emp.java**

Dept.java

**public** **class** Dept **implements** Serializable{

**private** String deptAddress;

**private** String deptName;

**private** Integer deptId;

**private** List<Emp> emps;

//省略set和get方法

Emp.java

**public** **class** Emp **implements** Serializable{

**private** String empId;

**private** String empName;

**private** String empSex;

**private** Dept dept;

//省略set/get方法

**4.4.2 编写DeptMapper.xml/EmpMapper.xml文件**

DeptMapper.xml

【实质与4.2一样，只是使用了extends，于是重新定义了一个resultMap】

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace=*"cn.itcast.entity.DeptMapper"*>

<!-- 表字段和对应实体属性命名一致时可以不配置 -->

<resultMap id=*"deptResultMap"* type=*"Dept"*>

<id property=*"deptId"* column=*"dept\_id"* />

<result property=*"deptName"* column=*"dept\_name"* />

<result property=*"deptAddress"* column=*"dept\_address"* />

</resultMap>

<!-- 一对多时，“多”的关联属性可独立配置resultMap，采用extends继承基本

属性的resultMap -->

<resultMap id=*"deptExtResultMap"* type=*"Dept"* extends=*"deptResultMap"*>

<!-- collection中resultMap引用的是其它文件的map 需要命名空间+id,例

如：cn.itcast.entity.EmpMapper.empResultMap -->

<collection property=*"emps"* ofType=*"Emp"*

resultMap=*"cn.itcast.entity.EmpMapper.empResultMap"* />

</resultMap>

<!--用于部门和员工关联查询，返回部门信息（包含部门员工信息）列表，

采用extends继承基本属性的resultMap -->

<select id=*"selectDeptEmpList"* parameterType=*"Dept"*

resultMap=*"deptExtResultMap"*>

select d.\*, e.\* from dept d inner join emp e on d.dept\_id=e.dept\_id

where d.dept\_name like #{deptName}

</select>

</mapper>

EmpMapper.xml[同4.2节]

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace=*"cn.itcast.entity.EmpMapper"*>

<!-- 表字段和对应实体属性命名一致时可以不配置 -->

<resultMap id=*"empResultMap"* type=*"Emp"*>

<id property=*"empId"* column=*"emp\_id"* />

<result property=*"empName"* column=*"emp\_name"* />

<result property=*"empSex"* column=*"emp\_sex"* />

<!--注意association元素的resultMap的值为没有配置“多”的属性映射的

deptResultMap，如下 -->

<association property=*"dept"* column=*"dept\_id"*

resultMap=*"cn.itcast.entity.DeptMapper.deptResultMap"*/>

</resultMap>

<!-- 用于员工和部门关联查询，返回员工信息（包含部门信息）列表 -->

<select id=*"selectEmpDeptList"* parameterType=*"Emp"*

resultMap=*"empResultMap"*>

select e.\*,d.\* from emp e inner join dept d on

e.dept\_id=d.dept\_id where d.dept\_name like #{dept.deptName}

</select>

</mapper>

**4.4.3 编写数据操作类：DeptDaoImpl.java/EmpDaoImpl.java**

DeptDaoImpl.java，查询部门员工信息，返回类型为List<Dept>，关键代码：

**public** List<Dept> selectDeptEmpList(Dept dept){

SqlSession session=**null**;

List<Dept> deps=**null**;

**try**{

session=MyBatisUtil.*getSession*();

deps=session.selectList("cn.itcast.entity.DeptMapper.selectDeptEmpList",dept);

session.commit();

}**catch** (Exception e) {

// **TODO**: handle exception

e.printStackTrace();

session.rollback();

}**finally**{

MyBatisUtil.*closeSession*();

}

**return** deps;

}

EmpDaoImpl.java

查询员工及其所在部门信息，返回类型为List< Emp >，关键代码

**public** List<Emp> selectEmpDeptList(Emp emp){

SqlSession session=**null**;

List<Emp> emps=**null**;

**try**{

session=MyBatisUtil.*getSession*();

emps=session.selectList("cn.itcast.entity.EmpMapper.selectEmpDeptList",emp);

session.commit();

}**catch** (Exception e) {

// **TODO**: handle exception

e.printStackTrace();

session.rollback();

}**finally**{

MyBatisUtil.*closeSession*();

}

**return** emps;

}

**4.4.4 编写测试类:DeptTest.java/EmpTest.java**

DeptTest.java关键代码

//测试部门和员工的关联查询，并遍历装载部门类型的结果集

@Test

**public** **void** selectDeptEmpList() {

Dept paramDept=**new** Dept();

paramDept.setDeptName("%研%");

List<Dept> depts=*deptDaoImpl*.selectDeptEmpList(paramDept);

**for**(Dept dept:depts){

System.*out*.println("dept:"+dept);

}

}

EmpTest.java关键代码

//测试员工和部门的关联查询，并遍历装载员工类型的结果集

@Test

**public** **void** selectEmpDeptList() {

Emp emp=**new** Emp();

Dept dept=**new** Dept();

dept.setDeptName("%研%");

emp.setDept(dept);

List<Emp> emps=*empDaoImpl*.selectEmpDeptList(emp);

**for**(Emp emp1:emps){

System.*out*.println("emp="+emp1);

System.*out*.println("dept＝"+emp1.getDept());

}

}

# 5. 缓存

Mybatis和hibernate一样，也使用缓存；

分为一级和二级缓存：

一级缓存指在SqlSession内；

二级缓存能被所有的SqlSession共享。

## 5.1 准备工作

数据库及表信息

drop database if exists mybatis;

create database mybatis CHARACTER SET UTF8;

use mybatis;

create table dept(

dept\_id int primary key auto\_increment,

dept\_name varchar(50),

dept\_address varchar(50)

);

insert into dept(dept\_name,dept\_address) values('研发部一部','广州');

insert into dept(dept\_name,dept\_address) values('研发部二部','广州');

insert into dept(dept\_name,dept\_address) values('研发部三部','深圳');

select \* from dept;

实体类：

**public** **class** Dept **implements** Serializable {

**private** Integer deptId;

**private** String deptName;

**private** String deptAddress;

//省略set/get方法

工具类，数据库操作类，测试类（略）

## 5.2一级缓存测试

关健方法代码：

//用来测试一级缓存

**public** **void** selectDept(Integer deptId){

SqlSession session=**null**;

**try** {

session =MyBatisUtil.*getSession*();

//命名空间+id,如cn.itcast.entity.DeptMapper.selectDept

Dept dept1=session.selectOne("cn.itcast.entity.DeptMapper.selectDept", deptId);

System.*out*.println("dept1:"+dept1);

Dept dept2=session.selectOne("cn.itcast.entity.DeptMapper.selectDept", deptId);

System.*out*.println("dept1:"+dept1);

} **catch** (Exception e) {

e.printStackTrace();

}**finally**{

**try** {

MyBatisUtil.*closeSession*();

} **catch** (Exception e) {

e.printStackTrace();

}

}

}

测试用例

@Test //测试一级缓存

**public** **void** testCache() {

*deptDaoImpl*.selectDept(1);

}

测试结果

DEBUG 2014-11-08 16:32:45,484 org.apache.ibatis.datasource.pooled.PooledDataSource: PooledDataSource forcefully closed/removed all connections.

DEBUG 2014-11-08 16:32:45,578 org.apache.ibatis.transaction.jdbc.JdbcTransaction:

Opening JDBC Connection

DEBUG 2014-11-08 16:32:45,859 org.apache.ibatis.datasource.pooled.PooledDataSource: Created connection 25374911.

DEBUG 2014-11-08 16:32:45,859 org.apache.ibatis.logging.jdbc.BaseJdbcLogger: ooo Using Connection [com.mysql.jdbc.JDBC4Connection@18330bf]

DEBUG 2014-11-08 16:32:45,859 org.apache.ibatis.logging.jdbc.BaseJdbcLogger: ==> Preparing: select \* from dept where dept\_id=?

DEBUG 2014-11-08 16:32:45,906 org.apache.ibatis.logging.jdbc.BaseJdbcLogger: ==> Parameters: 1(Integer)

dept1:Dept [deptId=1, deptName=研发部一部, deptAddress=广州]

dept1:Dept [deptId=1, deptName=研发部一部, deptAddress=广州]

DEBUG 2014-11-08 16:32:45,937 org.apache.ibatis.transaction.jdbc.JdbcTransaction: Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@18330bf]

DEBUG 2014-11-08 16:32:45,937 org.apache.ibatis.transaction.jdbc.JdbcTransaction:

Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@18330bf]

DEBUG 2014-11-08 16:32:45,937 org.apache.ibatis.datasource.pooled.PooledDataSource: Returned connection 25374911 to pool.

结果分析：

*用同一个SqlSession，依次执行同样的sql语句，*

发现只执行一次SQL语句，第二次查询时直接从缓存中获得，默认开启一级缓存。

## 5.3 (未使用)二级缓存测试

操作类关键方法代码

//用来测试二级缓存

**public** Dept testCache2(Integer deptId){

SqlSession session=**null**;

Dept dept=**null**;

**try** {

session =MyBatisUtil.*getSession*();

//命名空间+id,如cn.itcast.entity.DeptMapper.selectDept

dept=session.selectOne("cn.itcast.entity.DeptMapper.selectDept", deptId);

} **catch** (Exception e) {

e.printStackTrace();

}**finally**{

**try** {

MyBatisUtil.*closeSession*();

} **catch** (Exception e) {

e.printStackTrace();

}

}

**return** dept;

}

测试用例代码

@Test //用来测试二级缓存

**public** **void** testCache2() {

Dept dept = *deptDaoImpl*.testCache2(1);

System.*out*.println("dept:" + dept);

Dept dept1 = *deptDaoImpl*.testCache2(1);

System.*out*.println("dept1"+dept1);

}

测试结果

DEBUG 2014-11-08 16:40:52,717 org.apache.ibatis.logging.jdbc.BaseJdbcLogger:

ooo Using Connection [com.mysql.jdbc.JDBC4Connection@18330bf]

DEBUG 2014-11-08 16:40:52,717 org.apache.ibatis.logging.jdbc.BaseJdbcLogger: ==> Preparing: select \* from dept where dept\_id=?

DEBUG 2014-11-08 16:40:52,764 org.apache.ibatis.logging.jdbc.BaseJdbcLogger: ==> Parameters: 1(Integer)

DEBUG 2014-11-08 16:40:52,811 org.apache.ibatis.transaction.jdbc.JdbcTransaction:

Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@18330bf]

DEBUG 2014-11-08 16:40:52,811 org.apache.ibatis.transaction.jdbc.JdbcTransaction:

Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@18330bf]

DEBUG 2014-11-08 16:40:52,811 org.apache.ibatis.datasource.pooled.PooledDataSource: Returned connection 25374911 to pool.

dept:Dept [deptId=1, deptName=研发部一部, deptAddress=广州]

DEBUG 2014-11-08 16:40:52,811 org.apache.ibatis.transaction.jdbc.JdbcTransaction:

Opening JDBC Connection

DEBUG 2014-11-08 16:40:52,811 org.apache.ibatis.datasource.pooled.PooledDataSource: Checked out connection 25374911 from pool.

DEBUG 2014-11-08 16:40:52,811 org.apache.ibatis.transaction.jdbc.JdbcTransaction:

Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@18330bf]

DEBUG 2014-11-08 16:40:52,811 org.apache.ibatis.logging.jdbc.BaseJdbcLogger:

ooo Using Connection [com.mysql.jdbc.JDBC4Connection@18330bf]

DEBUG 2014-11-08 16:40:52,811 org.apache.ibatis.logging.jdbc.BaseJdbcLogger: ==> Preparing: select \* from dept where dept\_id=?

DEBUG 2014-11-08 16:40:52,811 org.apache.ibatis.logging.jdbc.BaseJdbcLogger: ==> Parameters: 1(Integer)

DEBUG 2014-11-08 16:40:52,811 org.apache.ibatis.transaction.jdbc.JdbcTransaction:

Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@18330bf]

DEBUG 2014-11-08 16:40:52,811 org.apache.ibatis.transaction.jdbc.JdbcTransaction:

Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@18330bf]

DEBUG 2014-11-08 16:40:52,811 org.apache.ibatis.datasource.pooled.PooledDataSource: Returned connection 25374911 to pool.

dept1Dept [deptId=1, deptName=研发部一部, deptAddress=广州]

测试结果分析

*使用了不同的SqlSession，依次执行同样的sql语句，*

发现执行了两次sql语句（下划线部分），默认不启用二级缓存。

## 5.4二级缓存配置

使用二级缓存机制：

|  |
| --- |
| 需开启 全局缓存，Mapper文件级缓存 ，SQL语句级缓存，才能使用。  默认文件级缓存没开启。 |

**5.4.1 全局缓存配置**

在mybatis的主配置文件中进行配置：

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-config.dtd">

<configuration>

<settings>

<!-- 默认启用全局缓存，禁用可把value设为false,如这里设为

false，Mapper.xml或SQL语句级的缓存配置不再起作用 -->

<setting name=*"cacheEnabled"* value=*"true"*/>

</settings>

<!—省略其它配置信息 -->

</configuration>

**5.4.2 Mapper文件级缓存配置**

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace=*"cn.itcast.entity.DeptMapper"*>

<cache/>

<!—省略其它配置信息-->

</mapper>

**5.4.3 SQL语句级缓存配置**

在相关的Mapper.xml文件中配置SQL查询，关键代码如下（示例）：

<!-- useCache默认值为true,设为false时缓存不起作用 -->

<select id=*"selectOne"* parameterType=*"int"* resultMap=*"deptResultMap"* useCache=*"true"* >

select \* from dept where dept\_id=#{id}

</select>

**5.4.4缓存分析**

修改DeptMapper.xml文件，增加缓存设置，如下

|  |  |  |
| --- | --- | --- |
| <!--  <cache eviction="FIFO" flushInterval="60000" size="512" readOnly="false"/>  创建一个 FIFO 缓存,每隔 60 秒刷新,存储结果对象或列表的 512 个引用,且返回的对象被认为是只读的,因此在不同线程中的调用者间修改它们会导致冲突。  可用收回策略:   |  | | --- | | 1. LRU(默认) – 最近最少使用的:移除最长时间不被使用的对象。(Least Recently Used)  2. FIFO – 先进先出:按对象进入缓存的顺序来移除它们。  3. SOFT – 软引用:移除基于垃圾回收器状态和软引用规则的对象。  4. WEAK – 弱引用:更积极地移除基于垃圾收集器状态和弱引用规则的对象。 |   flushInterval：刷新间隔时间，可被设置为任意的正整数,单位毫秒。  默认不设置,就是没有刷新间隔,缓存仅调用语句时刷新。  size：内存资源数目，可被设置为任意正整数。默认值是1024。  readOnly(只读)属性可被设置为 true 或 false。   |  | | --- | | * 只读的缓存会给所有调用者返回缓存对象的相同实例。因此这些对象不能被修改。提供了很重要的性能优势。 * 可读写的缓存返回缓存对象的拷贝(通过序列化) .慢一些,但安全,默认false |   -->  <cache eviction=*"LRU"* size=*"2"* readOnly=*"false"* /> |

测试代码

@Test

**public** **void** testSelectDept() {

*deptDaoImpl*.selectDept(1);

*deptDaoImpl*.selectDept(2);

*deptDaoImpl*.selectDept(2);

*deptDaoImpl*.selectDept(1);

*deptDaoImpl*.selectDept(3);

*deptDaoImpl*.selectDept(1);

*deptDaoImpl*.selectDept(2);

}

# 6.XML 中的特殊字符处理

如 MyBatis 用 XML 配置，会遇到一些对 XML 来说是特殊的字符。如小于号 “<”，因此要转义。主要有两个方式：

## 6.1 使用转义实体

下面是五个在 XML 文档中预定义好的转义实体:

|  |
| --- |
| &lt; < 小于号 (**less than**)  &gt; > 大于号 (greater than)  小于等于“<=”，其转义为：&lt;=  大小等于“>=”，转义为：&gt;=  &amp; &  &apos; ' 单引号  &quot; " 双引号 |

## 6.2 使用 CDATA 部件

|  |
| --- |
| CDATA:字符数据（character data）。  CDATA 是不会被解析器解析的文本。其中的标签不会被当作标记对待，实体不会被展开。 |

以"<![CDATA[" 标记开始，以"]]>"标记结束。

"<![CDATA["和"]]>"间的特殊字符的意义都不起作用，转变为普通字符串内容。

在 MyBatis的 XML 映射语句配置文件中,如 SQL 语句有特殊字符,用CDTA 部件括起来，如：

<select id= "selectBlog\_use\_collection" resultMap= "blogResult" >

<![CDATA[ SELECT id , title, author\_id as authored FROM BLOG

WHERE ID > 0 and ID < 10 ]]>

</select>

# 7常用批量操作(实质:见3.6节)

## 7.1准备数据

drop database if exists mybatis;

create database mybatis CHARACTER SET UTF8;

use mybatis;

create table dept(

dept\_id int primary key auto\_increment,

dept\_name varchar(50),

dept\_address varchar(50)

);

insert into dept(dept\_name,dept\_address) values('研发部一部','广州');

insert into dept(dept\_name,dept\_address) values('研发部二部','广州');

insert into dept(dept\_name,dept\_address) values('研发部三部','深圳');

--insert into dept( dept\_name, dept\_address ) values ('研发部4部','广州'),('研发部5部','广州'),('研发部6部','广州')

select \* from dept;

## 7.2批量新增部门

**7.2.1映射文件定义SQL**

<sql id=*"key"*>

<!--suffixOverrides="," 可以忽略最后“，”号 -->

<trim suffixOverrides=*","*>

dept\_name,

dept\_address,

</trim>

</sql>

<insert id=*"insertDeptList"*>

insert into dept(<include refid=*"key"* />)

values

<foreach collection=*"list"* item=*"item"* separator=*","*>

(#{item.deptName},#{item.deptAddress})

</foreach>

</insert>

**7.2.2编写批量添加部门方法**

数据操作类定义批量添加部门的方法

**public** **int** saveDeptList(List<Dept> depts) {

**int** i = 0;

SqlSession session = **null**;

**try** {

session = MyBatisUtil.*getSession*();

i = session.insert("cn.itcast.entity.DeptMapper.insertDeptList", depts);

session.commit();

} **catch** (Exception e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

session.rollback();

} **finally** {

MyBatisUtil.*closeSession*();

}

**return** i;

}

**7.2.3编写测试代码**

@Test

**public** **void** testInsertDeptList() {

List<Dept> depts = **new** ArrayList<Dept>();

**for**(**int** i=0;i<5;i++){

Dept dept = **new** Dept();

dept.setDeptName("deptname"+i);

dept.setDeptAddress("address"+i);

depts.add(dept);

}

**int** i=*deptDao*.saveDeptList(depts);

System.*out*.println("受影响的行数:"+i);

}

## 7.3批量删除部门

**7.3.1映射文件定义SQL**

<delete id=*"deleteDeptList"*>

delete from dept where dept\_id in

<foreach collection=*"list"* item=*"item"* open=*"("* close=*")"* separator=*","*>

#{item}

</foreach>

</delete>

补充:

|  |
| --- |
| SELECT column\_name(s) FROM table\_name WHERE column\_name IN (value1,value2,...) |

**7.3.2编写批量删除部门的方法**

**public** **int** deleteDeptList(List<Integer> deptIds) {

**int** i = 0;

SqlSession session = **null**;

**try** {

session = MyBatisUtil.*getSession*();

i = session.delete("cn.itcast.entity.DeptMapper.deleteDeptList", deptIds);

session.commit();

} **catch** (Exception e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

session.rollback();

} **finally** {

MyBatisUtil.*closeSession*();

}

**return** i;

}

**7.3.3编写测试代码**

@Test

**public** **void** testDeleteDeptList() {

List<Integer> deptIds = **new** ArrayList<Integer>();

**for**(**int** i=4;i<7;i++){

deptIds.add(i);

}

**int** i=*deptDao*.deleteDeptList(deptIds);

System.*out*.println("受影响的行数:"+i);

}

## 7.4批量修改员工信息

**7.4.1修改mybatis-config.xml文件**

支持上有点麻烦，需修改mybatis-config.xml文件相关数据库连接的信息（主要红色部分），以支持批量更新

|  |
| --- |
| <property name=*"url"* value=*"jdbc:mysql://localhost:3306/mybatis?useUnicode=true&amp;*  *characterEncoding=UTF-8&amp;allowMultiQueries=true"* /> |

**7.4.2配置批量更新的sql**

<update id=*"updateDeptList"*>

<!-- 多个update语句，用;号分隔开，如是oracle数据库，一般需要加“begin”

前缀，后缀“end;" -->

<foreach collection=*"list"* item=*"dept"* separator=*";"*>

update dept

<set>

<if test=*"dept.deptName!=null"*>dept\_name=#{dept.deptName},</if>

<if test=*"dept.deptAddress!=null"*>dept\_address=#{dept.deptAddress},</if>

</set>

where dept\_id=#{dept.deptId}

</foreach>

</update>

**7.4.3编写批量更新部门的方法**

**public** **int** updateDeptList(List<Dept> depts) {

**int** i = 0;

SqlSession session = **null**;

**try** {

session = MyBatisUtil.*getSession*();

i = session.update("cn.itcast.entity.DeptMapper.updateDeptList", depts);

session.commit();

} **catch** (Exception e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

session.rollback();

} **finally** {

MyBatisUtil.*closeSession*();

}

**return** i;

}

**7.4.4编写测试代码**

@Test

**public** **void** testUpdateDeptList() {

List<Dept> depts = **new** ArrayList<Dept>();

**for**(**int** i=1;i<4;i++){

Dept dept = **new** Dept();

dept.setDeptId(i);

dept.setDeptName("deptName"+i);

dept.setDeptAddress("deptAddress" + i);

depts.add(dept);

}

**int** i=*deptDao*.updateDeptList(depts);

System.*out*.println("受影响的行数:"+i);

}

# 8. Spring+myBatis整合

## 8.1 准备工作

新建项目并导入jar包

配置mybatis-config.xml

创建库及表

创建实体

编写映射文件，修改mybatis-config.xml内容

进行简单测试（除了导入spring相关jar包外，以上内容可能直接使用上一章节内容）

## 8.2 配置applicationContext.xml

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans

xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns:tx=*"http://www.springframework.org/schema/tx"*

xmlns:aop=*"http://www.springframework.org/schema/aop"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:p=*"http://www.springframework.org/schema/p"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-3.0.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-3.0.xsd*

*http://www.springframework.org/schema/tx*

*http://www.springframework.org/schema/tx/spring-tx-3.0.xsd*

[*http://www.springframework.org/schema/aop*](http://www.springframework.org/schema/aop)

*http://www.springframework.org/schema/aop/spring-aop-3.0.xsd"*>

<!-- 配置数据源，记得去掉myBatis-config.xml的数据源相关配置 -->

<bean id=*"dataSource"* class=*"com.mchange.v2.c3p0.ComboPooledDataSource"*>

<property name=*"driverClass"* value=*"com.mysql.jdbc.Driver"* />

<property name=*"jdbcUrl"* value=*"jdbc:mysql://localhost:3306/mybatis?*

*useUnicode=true&amp;characterEncoding=UTF-8"* />

<property name=*"user"* value=*"root"* />

<property name=*"password"* value=*"root"* />

</bean>

<!-- 配置session工厂 -->

<bean id=*"sqlSessionFactory"*

class=*"org.mybatis.spring.SqlSessionFactoryBean"*>

<property name=*"dataSource"* ref=*"dataSource"* />

<property name=*"configLocation"* value=*"classpath:myBatis-config.xml"* />

</bean>

<!-- 配置事务管理器,管理数据源事务处理-->

<bean id=*"transactionManager"*

class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>

<property name=*"dataSource"* ref=*"dataSource"* />

</bean>

<!-- 配置SessionTemplate，已封装了繁琐的数据操作-->

<bean id=*"sqlSessionTemplate"* class=*"org.mybatis.spring.SqlSessionTemplate"*>

<constructor-arg name=*"sqlSessionFactory"* ref=*"sqlSessionFactory"*/>

</bean>

</beans>

|  |
| --- |
| 理解:  代码中只需要通过sqlSessionTemplate调用mapper文件中定义的操作数据库的方法即可。 |

## 8.3 修改myBatis-config.xml

去掉数据源配置

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-config.dtd">

<configuration>

<typeAliases>

<typeAlias type=*"cn.itcast.entity.Dept"* alias=*"Dept"* />

</typeAliases>

<mappers>

<mapper resource=*"cn/itcast/entity/DeptMapper.xml"* />

</mappers>

</configuration>

## 8.4 编写dao层接口及实现

DeptDao.java

**public** **interface** DeptDao {

//根据部门ID查询部门信息

**public** Dept selectOne(**int** deptId);

}

修改接口实现类：DeptDaoImpl.java

**public** **class** DeptDaoImpl **implements** DeptDao{

**private** SqlSessionTemplate sqlSessionTemplate;

**public** SqlSessionTemplate getSqlSessionTemplate() {

**return** sqlSessionTemplate;

}

**public** **void** setSqlSessionTemplate(SqlSessionTemplate sqlSessionTemplate) {

**this**.sqlSessionTemplate = sqlSessionTemplate;

}

//根据部门ID查询部门信息

**public** Dept selectOne(**int** deptId){

System.*out*.println("dao :"+deptId);

//SqlSession session=null;

Dept dept=**null**;

**try** {

dept=sqlSessionTemplate.selectOne("cn.itcast.entity.DeptMapper.

selectOne",deptId);

System.*out*.println("dao.dept:"+dept);

} **catch** (Exception e) {

e.printStackTrace();

}

**return** dept;

}

}

## 8.5 编写业务层代码

业务层接口略，这里只写业务层实现类：DeptServiceImpl.java

**public** **class** DeptServiceImpl {

**private** DeptDao deptDao;

**public** Dept selectOne(**int** deptId){

Dept dept=deptDao.selectOne(deptId);

**return** dept;

}

**public** DeptDao getDeptDao() {

**return** deptDao;

}

**public** **void** setDeptDao(DeptDao deptDao) {

**this**.deptDao = deptDao;

}

}

## 8.6 配置bean信息到sping配置文件

<!-- DAO层部门信息表的数据操作对象 -->

<bean id=*"deptDao"* class=*"cn.itcast.dao.impl.DeptDaoImpl"* >

<property name=*"sqlSessionTemplate"* ref=*"sqlSessionTemplate"*/>

</bean>

<!-- 业务层部门信息业务处理对象 -->

<bean id=*"deptService"* class=*"cn.itcast.service.impl.DeptServiceImpl"*>

<property name=*"deptDao"* ref=*"deptDao"*/>

</bean>

## 8.7 编写测试类

@Test

**public** **void** selectOne() {

ApplicationContext context=**new**

ClassPathXmlApplicationContext("applicationContext.xml");

DeptServiceImpl deptService=

(DeptServiceImpl)context.getBean("deptService");

Dept dept = deptService.selectOne(1);

System.*out*.println("dept:" + dept);

}

## 8.8 简化配置

**8.8.1 扫描式加载SQL映射文件**

修改myBatis-config.xml文件，去掉<mappers>配置

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-config.dtd">

<configuration>

<typeAliases>

<typeAlias type=*"cn.itcast.entity.Dept"* alias=*"Dept"* />

</typeAliases>

<!-- 采用扫描式加载映射文件，以下将不用配置,可以减少映射文件过多时

维护的麻烦 -->

<!—

<mappers>

<mapper resource="cn/itcast/entity/DeptMapper.xml" />

</mappers>

-->

</configuration>

修改applicationContext.xml,为SqlSessionFactoryBean设置mapperLocations属性

<!-- 配置session工厂 -->

<bean id=*"sqlSessionFactory"*

class=*"org.mybatis.spring.SqlSessionFactoryBean"*>

<property name=*"dataSource"* ref=*"dataSource"* />

<property name=*"configLocation"* value=*"classpath:myBatis-config.xml"* />

<!-- 配置扫描式加载SQL映射文件 -->

<property name=*"mapperLocations"*

value=*"classpath:cn/itcast/entity/\*.xml"*/>

</bean>

**8.8.2【没看懂】MapperScannerConfigurer简化配置**

**1）**spring配置文件中添加*MapperScannerConfigurer* 配置

并去掉所有的Dao接口实现类配置

<!-- 配置转换器，对于在basePackage设置的包（包括子包）下的接口类的全类名和在Mapper.xml文件中定义过的命名空间一致，spring将会生成对应的代理对象（在调用 的地方通过@Autowired方式将可以注入接口实例）-->

<bean class=*"org.mybatis.spring.mapper.MapperScannerConfigurer"*>

<property name=*"sqlSessionFactory"* ref=*"sqlSessionFactory"*/>

<property name=*"basePackage"* value=*"cn.itcast.dao"*/>

</bean>

<!-- DAO层部门信息表的数据操作对象，上面如果配置

MapperScannerConfigurer转换器，DAO接口将不再使用实现类 -->

<!--

<bean id="deptDao" class="cn.itcast.dao.impl.DeptDaoImpl" >

<property name="sqlSessionTemplate" ref="sqlSessionTemplate"/>

</bean>

-->

<!-- 业务层部门信息业务处理对象 -->

<bean id=*"deptService"* class=*"cn.itcast.service.impl.DeptServiceImpl"*>

<!-- 上面如果配置MapperScannerConfigurer转换器，DAO接口将不再用实

现类注入 -->

<!-- <property name="deptDao" ref="deptDao"/> -->

</bean>

**2）** 检查或修改DeptMapper.xml文件：

注意：命名空间+id和接口+方法名 一致

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<!-- 这时的命名空间就需要和dao接口类全类名一致了 -->

<mapper namespace=*"cn.itcast.dao.DeptDao"*>

<!-- 表字段和对应实体属性命名一致时可以不配置 -->

<resultMap id=*"deptResultMap"* type=*"Dept"*>

<id property=*"deptId"* column=*"dept\_id"* />

<result property=*"deptName"* column=*"dept\_name"* />

<result property=*"deptAddress"* column=*"dept\_address"* />

</resultMap>

<!-- 这时的id就需要和dao接口的方法一致了 -->

<select id=*"selectOne"* parameterType=*"int"* resultMap=*"deptResultMap"*>

select \* from dept where dept\_id=#{id}

</select>

</mapper>

**3）业务类中，使用**@Autowired**为DAO接口注入对象**

**public** **class** DeptServiceImpl {

@Autowired

**private** DeptDao deptDao;

//省略其它代码

4）删除Dao实现类（存在也没有意义）

# 9. MyBatis的优缺点

一：优点：

|  |
| --- |
| 1. 易于上手和掌握。  2. sql写在xml里，便于统一管理和优化。  3. 解除sql与程序代码的耦合。  4. 提供映射标签，支持对象与数据库的orm字段关系映射  5. 提供对象关系映射标签，支持对象关系组建维护  6. 提供xml标签，支持编写动态sql。 |

二：缺点：

|  |
| --- |
| 1. sql工作量很大，尤其是字段多、关联表多时，更是如此。  2. sql依赖于数据库，导致数据库移植性差。  3. 由于xml里标签id必须唯一，导致DAO中方法不支持方法重载。  4. 字段映射标签和对象关系映射标签仅是对映射关系的描述，具体实现仍然依赖于sql。  注意：如果多配置了一对Collection标签，sql里没有join子表或查询子表的话，查询后返回的对象是不具备对象关系的，即Collection的对象为null  5. DAO层过于简单，对象组装的工作量较大。  6. 不支持级联更新、级联删除。  7. 编写动态sql时,不方便调试，尤其逻辑复杂时。  8 提供的写动态sql的xml标签功能简单，编写动态sql仍然受限，且可读性低。  9. 若不查询主键字段，容易造成查询出的对象有“覆盖”现象。  10. 参数的数据类型支持不完善。  注意：如参数为Date类型时，容易报没有get、set方法，需在参数上加@param  11. 多参数时，使用不方便，功能不够强大。  12. 缓存使用不当，容易产生脏数据 |

# 10.原理分析之一：从JDBC到Mybatis

重点讲解了为什么要将JDBC封装成Mybaits这样一个持久层框架。

**10.1 JDBC实现查询分析**

七个步骤：

|  |
| --- |
| (1)加载JDBC驱动  (2)建立并获取数据库连接  (3)创建 JDBC Statements 对象  (4)设置SQL语句的传入参数  (5)执行SQL语句并获得查询结果  (6)对查询结果进行转换处理并将处理结果返回  (7)释放相关资源（关闭Connection，关闭Statement，关闭ResultSet） |

具体的实现代码：

|  |
| --- |
| public static List<Map<String,Object>> queryForList(){  Connection connection = null;  ResultSet rs = null;  PreparedStatement stmt = null;  List<Map<String,Object>> resultList = new ArrayList<Map<String,Object>>();    try {  //加载JDBC驱动  Class.forName("oracle.jdbc.driver.OracleDriver").newInstance();  String url = "jdbc:oracle:thin:@localhost:1521:ORACLEDB";    String user = "trainer";  String password = "trainer";    //获取数据库连接  connection = DriverManager.getConnection(url,user,password);// **10.2.1**    String sql = "select \* from userinfo where user\_id = ? ";  //创建Statement对象（每一个Statement为一次数据库执行请求）  stmt = connection.prepareStatement(sql);    //设置传入参数  stmt.setString(1, "zhangsan");    //执行SQL语句  rs = stmt.executeQuery();    //处理查询结果（将查询结果转换成List<Map>格式）  ResultSetMetaData rsmd = rs.getMetaData();  int num = rsmd.getColumnCount();    while(rs.next()){  Map map = new HashMap();  for(int i = 0;i < num;i++){  String columnName = rsmd.getColumnName(i+1);  map.put(columnName,rs.getString(columnName));  }  resultList.add(map);  }    } catch (Exception e) {  e.printStackTrace();  } finally {  try {  //关闭结果集  if (rs != null) {  rs.close();  rs = null;  }  //关闭执行  if (stmt != null) {  stmt.close();  stmt = null;  }  if (connection != null) {  connection.close(); //**10.2.1**  connection = null;  }  } catch (SQLException e) {  e.printStackTrace();  }  }    return resultList;  } |

**10.2 JDBC演变到Mybatis过程**

**第一步优化：连接获取和释放**

* 问题描述：数据库连接频繁的开启和关闭本身就造成资源浪费，影响系统性能。
* 解决问题：

可使用数据库连接池解决资源浪费问题。

可反复利用已建立的连接去访问数据库。减少连接的开启和关闭的时间。

* 问题描述：

但现在连接池多种多样，如DBCP连接池，容器本身的JNDI数据库连接池。

* 解决问题：

可通过DataSource隔离解耦，统一从DataSource里获取数据库连接，DataSource具体由DBCP还是由容器的JNDI实现都可，将DataSource的具体实现通过让用户配置来应对变化。

**第二步优化：SQL统一存取**

* 问题描述：

用JDBC时，SQL语句基本都在JAVA类中，不足：

第一，可读性很差，不利于维护以及做性能调优。

第二，改动Java代码需重新编译、打包部署。

第三，不利于取出SQL在数据库客户端执行（取出后还得删掉中间的Java代码，编写好的SQL语句写好后还得通过＋号在Java进行拼凑）。

* 解决问题：

可将SQL语句集中放到配置文件或数据库里（key-value的格式）。

**第三步优化：传入参数映射和动态SQL**

* 问题描述：

很多情况下，都可通过在SQL语句中设置占位符来达到使用传入参数的目的，有一定局限性，是按照一定顺序传入参数的，要与占位符一一匹配。

但如果传入的参数不确定（如列表查询，根据用户填写的查询条件不同，传入查询的参数也是不同的，有时是一个参数、有时可能是三个参数），就得在后台代码中自己根据请求的传入参数去拼凑相应的SQL语句，还是避免不了在Java代码里面写SQL语句的。

* 解决问题：

第一，动态问题~~，按照我们正常的程序员思维是，通过if和else这类的判断来进行是最直观的，这个时候我们想到了JSTL中的<if test=””></if>这样的标签，那么，能不能将这类的标签引入到SQL语句中呢？假设可以，那么我们这里就需要一个专门的SQL解析器来解析这样的SQL语句，但是，if判断的变量来自于哪里呢？传入的值本身是可变的，那么我们得为这个值定义一个不变的变量名称，而且这个变量名称必须和对应的值要有对应关系，可以通过这个变量名称找到对应的值，这个时候我们想到了key-value的Map。解析的时候根据变量名的具体值来判断。~~

假如前面可判断没有问题，那么假如判断的结果是true，那么就需要输出的标签里面的SQL片段，但是怎么解决在标签里面使用变量名称的问题呢？这里我们需要使用一种有别于SQL的语法来嵌入变量（比如使用＃变量名＃）。这样，SQL语句经过解析后就可以动态的生成符合上下文的SQL语句。

还有，怎么区分开占位符变量和非占位变量？有时候我们单单使用占位符是满足不了的，占位符只能为查询条件占位，SQL语句其他地方使用不了。这里我们可以使用#变量名#表示占位符变量，使用$变量名$表示非占位符变量。

**第四步优化：结果映射和结果缓存**

* 问题描述：

执行SQL语句后，返回的ResultSet结果集，需将ResultSet对象的数据取出，不然等到释放资源时就取不到这些结果信息了。

从前面优化来看，以及将获取连接、设置传入参数、执行SQL语句、释放资源这些都封装起来了，只剩下结果处理这块还没有进行封装

* 解决问题：

有可能将结果不做任何处理就直接返回，也可能将结果转换成一个JavaBean对象、Map、List返回等，结果处理可能多种多样。

必须告诉SQL处理器两点：

1. 需要返回什么类型的对象；
2. 需要返回的对象的数据结构怎么跟执行的结果映射，才能将具体的值copy到对应的数据结构上。

可进而考虑对SQL执行结果的缓存来提升性能。

缓存数据都是key-value的格式。即使同一条SQL语句几次访问的过程中由于传入参数的不同，得到的执行SQL语句也不同。那么缓存起来的时候是多对。

SQL语句和传入参数两部分合起来可以作为数据缓存的key值。

**第五步优化：解决重复SQL语句问题**

* 问题描述：

将所有SQL语句都放到配置文件中，会遇到SQL重复的问题，几个功能的SQL语句其实都差不多，有些可能是SELECT后面那段不同、有些可能是WHERE语句不同。有时候表结构改了，那么我们就需要改多个地方，不利于维护。

* 解决问题：

通过将SQL片段模块化，将重复的SQL片段独立成一个SQL块，然后在各个SQL语句引用重复的SQL块，这样需修改时只需要修改一处即可。

**4. 优化总结：**

总结上面对JDBC的优化和封装：

（1） 使用数据库连接池对连接进行管理

（2） SQL语句统一存放到配置文件

（3） SQL语句变量和传入参数的映射以及动态SQL

（4） 动态SQL语句的处理

（5） 对数据库操作结果的映射和结果缓存

（6） SQL语句的重复

**5. Mybaits有待改进之处**

* 问题描述：

SQL语句太多

* 改进方法：

单表操作可不可以不写SQL语句，通过JavaBean的默认映射器生成对应的SQL语句。

# 11.《深入理解mybatis原理》Mybatis数据源与连接池

数据源的组织直接影响框架性能。

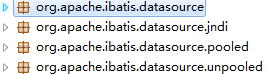
首先会讲述MyBatis的数据源分类，然后介绍数据源加载和使用。

紧接着将分类介绍UNPOOLED、POOLED和JNDI类型的数据源组织；

会重点讲解POOLED类型的数据源和其实现的连接池原理。

**一、MyBatis数据源DataSource分类**

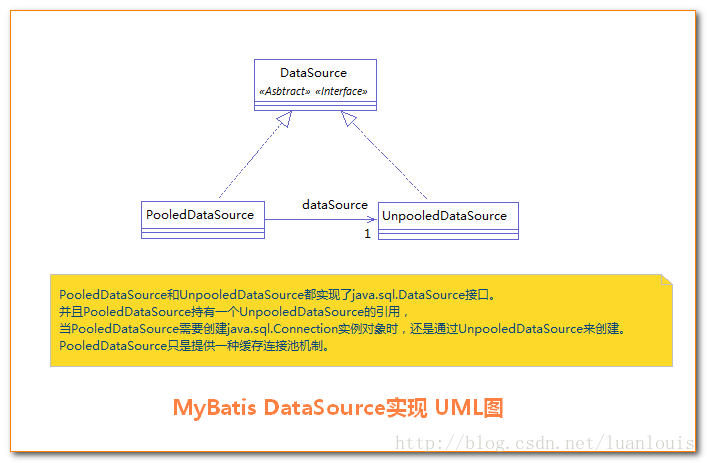
MyBatis数据源实现：



数据源DataSource分为三种：

|  |
| --- |
| UNPOOLED 不使用连接池的数据源  POOLED 使用连接池的数据源  JNDI 使用JNDI实现的数据源 |

* MyBatis内部分别定义了实现了java.sql.DataSource接口的UnpooledDataSource，PooledDataSource类来表示UNPOOLED、POOLED类型的数据源。 如下图：



对于JNDI类型的数据源DataSource，通过JNDI上下文中取值。

**二、数据源DataSource的创建过程**

DataSource对象的创建发生在MyBatis初始化的过程中。

在mybatis的XML配置文件中，使用<dataSource>元素来配置数据源



1. MyBatis初始化时，解析此文件，根据<dataSource>的type属性来创建相应类型的的数据源DataSource，即：

type=”POOLED” ：MyBatis会创建PooledDataSource实例

type=”UNPOOLED” ：MyBatis会创建UnpooledDataSource实例

type=”JNDI” ：MyBatis会从JNDI服务上查找DataSource实例，然后返回使用

2. MyBatis通过工厂模式创建数据源DataSource对象，MyBatis定义了抽象的工厂接口:org.apache.ibatis.datasource.DataSourceFactory,通过其getDataSource()方法返回数据源DataSource：

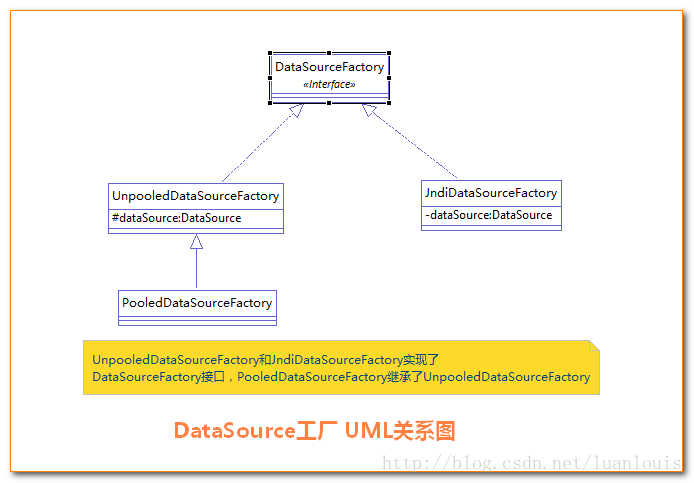
定义如下：

|  |
| --- |
| public interface DataSourceFactory {  void setProperties(Properties props);  //生产DataSource  DataSource getDataSource();  } |

* 上述三种不同类型的type，有对应的以下dataSource工厂：

|  |
| --- |
| POOLED PooledDataSourceFactory  UNPOOLED UnpooledDataSourceFactory  JNDI JndiDataSourceFactory |

* 其类图如下所示：



3. 创建DataSource实例后，将其放到Configuration对象内的Environment对象中， 供以后使用。

**三、 DataSource什么时候创建Connection对象**

当需创建SqlSession对象并执行SQL语句时，MyBatis才会调用dataSource对象来创建java.sql.Connection对象。即java.sql.Connection对象的创建一直延迟到执行SQL语句时。

比如，有如下方法执行一简单SQL语句：

|  |
| --- |
| String resource = "mybatis-config.xml";  InputStream inputStream = Resources.getResourceAsStream(resource);  SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);  SqlSession sqlSession = sqlSessionFactory.openSession();  sqlSession.selectList("SELECT \* FROM STUDENTS");// |

触发MyBatis在底层执行下面这个方法来创建java.sql.Connection对象：

|  |
| --- |
| protected void openConnection() throws SQLException {  if (log.isDebugEnabled()) {  log.debug("Opening JDBC Connection");  }  connection = dataSource.getConnection();  if (level != null) {  connection.setTransactionIsolation(level.getLevel());  }  setDesiredAutoCommit(autoCommmit);  } |

**四、不使用连接池的UnpooledDataSource**

当 <dataSource>的type=”UNPOOLED”，MyBatis首先会实例化一个UnpooledDataSourceFactory工厂实例，然后通过.getDataSource()方法返回一个UnpooledDataSource实例对象引用，假定为dataSource。

用UnpooledDataSource的getConnection(),每调用一次就会产生一个新的Connection实例对象。

|  |
| --- |
| /\*  UnpooledDataSource的getConnection()实现  \*/  public Connection getConnection() throws SQLException  {  return doGetConnection(username, password);  }    private Connection doGetConnection(String username, String password) throws SQLException  {  //封装username和password成properties  Properties props = new Properties();  if (driverProperties != null)  {  props.putAll(driverProperties);  }  if (username != null)  {  props.setProperty("user", username);  }  if (password != null)  {  props.setProperty("password", password);  }  return doGetConnection(props);  }    /\*  \* 获取数据连接  \*/  private Connection doGetConnection(Properties properties) throws SQLException  {  //1.初始化驱动  initializeDriver();  //2.从DriverManager中获取连接，获取新的Connection对象  Connection connection = DriverManager.getConnection(url, properties);  //3.配置connection属性  configureConnection(connection);  return connection;  } |

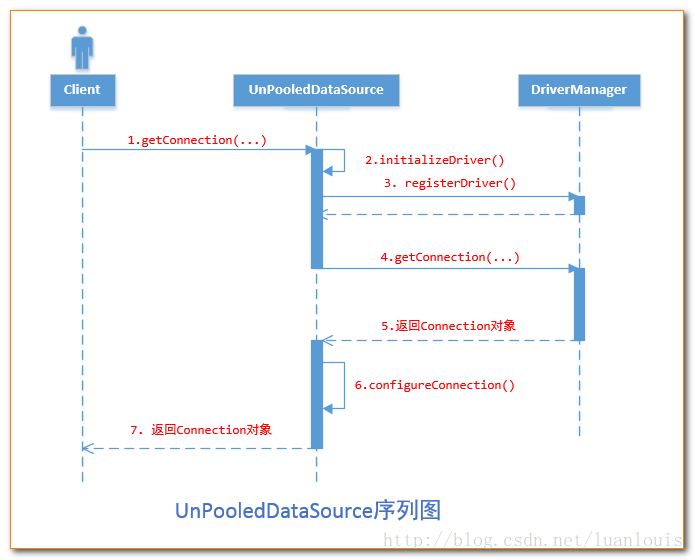
如上代码所示，UnpooledDataSource会做以下事情：

1. 初始化驱动：判断driver驱动是否已加载到内存中，如还没加载，则会动态地加载driver类，并实例化一个Driver对象，用DriverManager.registerDriver()将其注册到内存中，以供后续使用。

2. 创建Connection对象： 使用DriverManager.getConnection()方法创建连接。

3. 配置Connection对象： 设置是否自动提交autoCommit和隔离级别isolationLevel。

4. 返回Connection对象。



总结：从上述代码中可看到，每调用一次getConnection()方法，都会通过DriverManager.getConnection()返回新的java.sql.Connection实例。

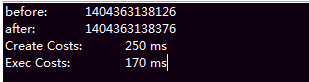
**五、为什么要使用连接池？**

1. 创建一java.sql.Connection实例对象的代价

通过连接Oracle数据库，创建创建Connection对象，来看创建一个Connection对象、执行SQL语句各消耗多长时间。代码如下：

|  |
| --- |
| public static void main(String[] args) throws Exception  {  String sql = "select \* from hr.employees where employee\_id < ? and employee\_id >= ?";  PreparedStatement st = null;  ResultSet rs = null;    long beforeTimeOffset = -1L; //创建Connection对象前时间  long afterTimeOffset = -1L; //创建Connection对象后时间  long executeTimeOffset = -1L; //创建Connection对象后时间    Connection con = null;  Class.forName("oracle.jdbc.driver.OracleDriver");    beforeTimeOffset = new Date().getTime();  System.out.println("before:\t" + beforeTimeOffset);    con = DriverManager.getConnection("jdbc:oracle:thin:@127.0.0.1:1521:xe", "louluan", "123456");    afterTimeOffset = new Date().getTime();  System.out.println("after:\t\t" + afterTimeOffset);  System.out.println("Create Costs:\t\t" + (afterTimeOffset - beforeTimeOffset) + " ms");    st = con.prepareStatement(sql);  //设置参数  st.setInt(1, 101);  st.setInt(2, 0);  //查询，得出结果集  rs = st.executeQuery();  executeTimeOffset = new Date().getTime();  System.out.println("Exec Costs:\t\t" + (executeTimeOffset - afterTimeOffset) + " ms");    } |

上述程序在我笔记本上的执行结果为：



可看出，创建一Connection对象，用了250 毫秒；而执行SQL的时间用了170毫秒。

这仅是一个Connection对象就有这么大的代价，设想一下另一种情况：如在Web应用程序中，为用户的每一个请求就操作一次数据库，当有10000个在线用户并发操作的话，对计算机而言，仅仅创建Connection对象不包括做业务的时间就要损耗10000×250ms= 250 0000 ms = 2500 s = 41.6667 min,竟然要41分钟！！！如果对高用户群体使用这样的系统，简直就是开玩笑！

2. 问题分析：

创建一java.sql.Connection对象的代价如此巨大，因为创建一Connection对象的过程，在底层就相当于和数据库建立的通信连接，在建立通信连接的过程，消耗了这么多的时间，而往往建立连接后（即创建Connection对象后），就执行一个简单的SQL语句，然后就要抛弃掉，这是一个非常大的资源浪费！

3.解决方案:

对于需频繁地跟数据库交互的应用程序，可在创建了Connection对象，并操作完数据库后不释放掉资源，放到内存中，当下次需操作数据库时，可直接从内存中取出Connection对象，不需再创建了，这样就极大地节省了创建Connection对象的资源消耗。

由于内存有限和宝贵，这又对我们对内存中的Connection对象怎么有效地维护提出了很高的要求。

将在内存中存放Connection对象的容器称为连接池（Connection Pool）。

**六、使用了连接池的PooledDataSource**

同样地，也是用PooledDataSource的getConnection()返回Connection对象。基本原理：

PooledDataSource将java.sql.Connection对象包裹成PooledConnection对象放到了PoolState类型的容器中维护。

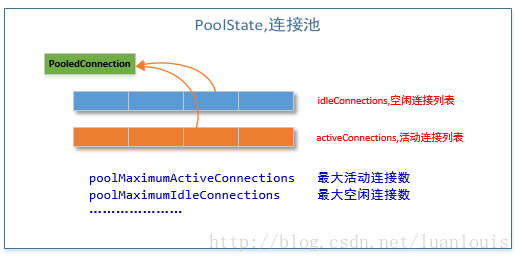
MyBatis将连接池中的PooledConnection分为两种状态：

空闲状态（idle）和活动状态(active)，分别被存储到PoolState容器内的idleConnections和activeConnections两个List集合中：

idleConnections:表当前闲置的没有被使用的PooledConnection集合，调用PooledDataSource的getConnection()方法时，会优先从此集合中取PooledConnection对象。当用完一java.sql.Connection对象时，MyBatis会将其包裹成PooledConnection对象放到此集合中。

activeConnections:当前正被使用的PooledConnection集合，调用PooledDataSource的getConnection()方法时，会优先从idleConnections集合中取PooledConnection对象,如没有，则看此集合是否已满，如未满，PooledDataSource会创建出一个PooledConnection，添加到此集合中，并返回。

PoolState连接池的大致结构如下所示：



6.1 获取java.sql.Connection对象的过程

PooledDataSource 的getConnection()方法获取Connection对象的实现：

|  |
| --- |
| public Connection getConnection() throws SQLException {  return popConnection(dataSource.getUsername(), dataSource.getPassword()).getProxyConnection();  }    public Connection getConnection(String username, String password) throws SQLException {  return popConnection(username, password).getProxyConnection();  } |

popConnection()，会从连接池中返回一个可用的PooledConnection对象，然后再调用getProxyConnection()方法最终返回Conection对象。

现在看一下popConnection()到底做了什么：

1. 先看是否有空闲(idle)状态下的PooledConnection对象，如果有，就直接返回一个可用的PooledConnection对象；否则进行第2步。

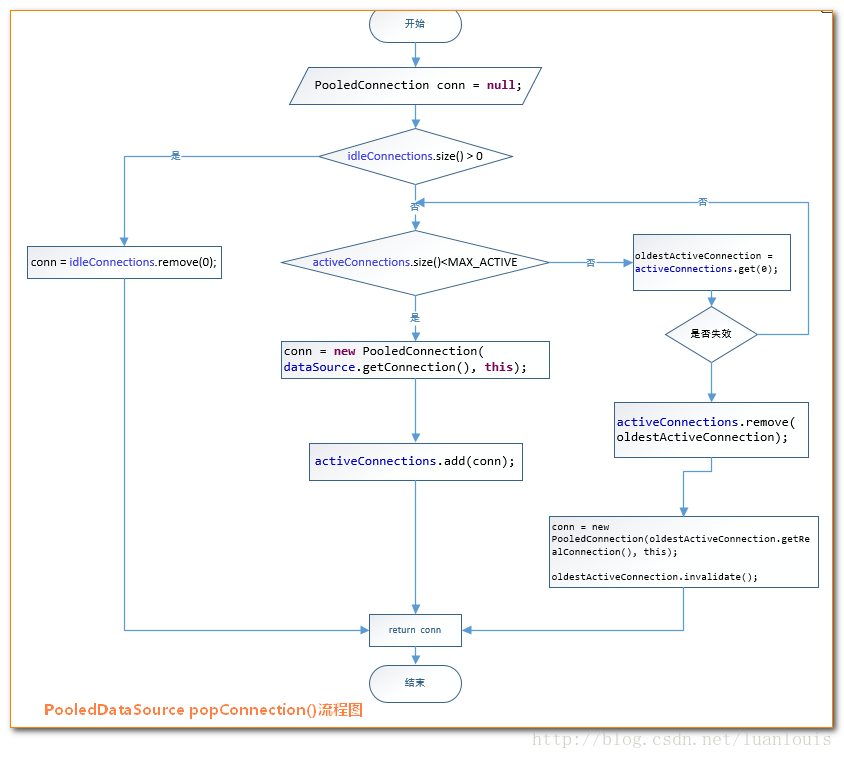
2. 查看活动状态的PooledConnection池activeConnections是否已满；如没满，则创建一个新的PooledConnection对象，放到activeConnections池中，然后返回此PooledConnection对象；否则进行第三步；

3. 看最先进入activeConnections池中的PooledConnection对象是否已经过期：如已过期，从activeConnections池中移除此对象，然后创建一个新的PooledConnection对象，添加到activeConnections中，然后将此对象返回；否则进行第4步。

4. 线程等待，循环2步

|  |
| --- |
| /\*  \* 传递一个用户名和密码，从连接池中返回可用的PooledConnection  \*/  private PooledConnection popConnection(String username, String password) throws SQLException  {  boolean countedWait = false;  PooledConnection conn = null;  long t = System.currentTimeMillis();  int localBadConnectionCount = 0;    while (conn == null)  {  synchronized (state)  {  if (state.idleConnections.size() > 0)  {  // 连接池中有空闲连接，取出第一个  conn = state.idleConnections.remove(0);  if (log.isDebugEnabled())  {  log.debug("Checked out connection " + conn.getRealHashCode() + " from pool.");  }  }  else  {  // 连接池中没有空闲连接，则取当前正在使用的连接数小于最大限定值，  if (state.activeConnections.size() < poolMaximumActiveConnections)  {  // 创建一个新的connection对象  conn = new PooledConnection(dataSource.getConnection(), this);  @SuppressWarnings("unused")  //used in logging, if enabled  Connection realConn = conn.getRealConnection();  if (log.isDebugEnabled())  {  log.debug("Created connection " + conn.getRealHashCode() + ".");  }  }  else  {  // Cannot create new connection 当活动连接池已满，不能创建时，取出活动连接池的第一个，即最先进入连接池的PooledConnection对象  // 计算它的校验时间，如果校验时间大于连接池规定的最大校验时间，则认为它已经过期了，利用这个PoolConnection内部的realConnection重新生成一个PooledConnection  //  PooledConnection oldestActiveConnection = state.activeConnections.get(0);  long longestCheckoutTime = oldestActiveConnection.getCheckoutTime();  if (longestCheckoutTime > poolMaximumCheckoutTime)  {  // Can claim overdue connection  state.claimedOverdueConnectionCount++;  state.accumulatedCheckoutTimeOfOverdueConnections += longestCheckoutTime;  state.accumulatedCheckoutTime += longestCheckoutTime;  state.activeConnections.remove(oldestActiveConnection);  if (!oldestActiveConnection.getRealConnection().getAutoCommit())  {  oldestActiveConnection.getRealConnection().rollback();  }  conn = new PooledConnection(oldestActiveConnection.getRealConnection(), this);  oldestActiveConnection.invalidate();  if (log.isDebugEnabled())  {  log.debug("Claimed overdue connection " + conn.getRealHashCode() + ".");  }  }  else  {    //如果不能释放，则必须等待有  // Must wait  try  {  if (!countedWait)  {  state.hadToWaitCount++;  countedWait = true;  }  if (log.isDebugEnabled())  {  log.debug("Waiting as long as " + poolTimeToWait + " milliseconds for connection.");  }  long wt = System.currentTimeMillis();  state.wait(poolTimeToWait);  state.accumulatedWaitTime += System.currentTimeMillis() - wt;  }  catch (InterruptedException e)  {  break;  }  }  }  }    //如果获取PooledConnection成功，则更新其信息    if (conn != null)  {  if (conn.isValid())  {  if (!conn.getRealConnection().getAutoCommit())  {  conn.getRealConnection().rollback();  }  conn.setConnectionTypeCode(assembleConnectionTypeCode(dataSource.getUrl(), username, password));  conn.setCheckoutTimestamp(System.currentTimeMillis());  conn.setLastUsedTimestamp(System.currentTimeMillis());  state.activeConnections.add(conn);  state.requestCount++;  state.accumulatedRequestTime += System.currentTimeMillis() - t;  }  else  {  if (log.isDebugEnabled())  {  log.debug("A bad connection (" + conn.getRealHashCode() + ") was returned from the pool, getting another connection.");  }  state.badConnectionCount++;  localBadConnectionCount++;  conn = null;  if (localBadConnectionCount > (poolMaximumIdleConnections + 3))  {  if (log.isDebugEnabled())  {  log.debug("PooledDataSource: Could not get a good connection to the database.");  }  throw new SQLException("PooledDataSource: Could not get a good connection to the database.");  }  }  }  }    }    if (conn == null)  {  if (log.isDebugEnabled())  {  log.debug("PooledDataSource: Unknown severe error condition. The connection pool returned a null connection.");  }  throw new SQLException("PooledDataSource: Unknown severe error condition. The connection pool returned a null connection.");  }    return conn;  } |

对应的处理流程图如下所示：



如上所示,对于PooledDataSource的getConnection()方法内，先是调用类PooledDataSource的popConnection()方法返回了一个PooledConnection对象，然后调用了PooledConnection的getProxyConnection()来返回Connection对象。

6.2java.sql.Connection对象的回收

程序使用完Connection对象时，如不用数据库连接池，一般会调用 connection.close()方法，关闭connection连接，释放资源。如下所示：

|  |
| --- |
| private void test() throws ClassNotFoundException, SQLException  {  String sql = "select \* from hr.employees where employee\_id < ? and employee\_id >= ?";  PreparedStatement st = null;  ResultSet rs = null;    Connection con = null;  Class.forName("oracle.jdbc.driver.OracleDriver");  try  {  con = DriverManager.getConnection("jdbc:oracle:thin:@127.0.0.1:1521:xe", "louluan", "123456");  st = con.prepareStatement(sql);  //设置参数  st.setInt(1, 101);  st.setInt(2, 0);  //查询，得出结果集  rs = st.executeQuery();  //取数据，省略  //关闭，释放资源  con.close();  }  catch (SQLException e)  {  con.close();  e.printStackTrace();  }  } |

调用过close()方法的Connection对象所持有的资源会被全部释放掉，Connection对象也就不能再使用。

那么，如用了连接池在用完了Connection对象时，需将它放在连接池中，该怎样做呢？

可能大家第一个在脑海里闪现出来的想法就是：我在应该调用con.close()方法的时候，不调用close()f方法，将其换成将Connection对象放到连接池容器中的代码！

好，我们将上述的想法实现，首先定义一个简易连接池Pool，然后将上面的代码改写：

|  |
| --- |
| package com.foo.jdbc;  import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.SQLException;  import java.util.Vector;    /\*\*  \*  \* 一个线程安全的简易连接池实现，此连接池是单例的  \* putConnection()将Connection添加到连接池中  \* getConnection()返回一个Connection对象  \*/  public class Pool {  private static Vector<Connection> pool = new Vector<Connection>();  private static int MAX\_CONNECTION =100;  private static String DRIVER="oracle.jdbc.driver.OracleDriver";  private static String URL = "jdbc:oracle:thin:@127.0.0.1:1521:xe";  private static String USERNAME = "louluan";  private static String PASSWROD = "123456";    static {  try {  Class.forName(DRIVER);  } catch (ClassNotFoundException e) {  e.printStackTrace();  }  }    /\*\*  \* 将一个Connection对象放置到连接池中  \*/  public static void putConnection(Connection connection){  synchronized(pool)  {  if(pool.size()<MAX\_CONNECTION)  {  pool.add(connection);  }  }  }    /\*\*  \* 返回一个Connection对象，如连接池内有元素，则pop出第一个元素；  \* 如果连接池Pool中没有元素，则创建一connection对象，然后添加到pool中  \* @return Connection  \*/  public static Connection getConnection(){  Connection connection = null;  synchronized(pool)  {  if(pool.size()>0)  {  connection = pool.get(0);  pool.remove(0);  }  else  {  connection = createConnection();  pool.add(connection);  }  }  return connection;  }    /\*\*  \* 创建一个新的Connection对象  \*/  private static Connection createConnection()  {  Connection connection = null;  try {  connection = DriverManager.getConnection(URL, USERNAME,PASSWROD);  } catch (SQLException e) {  e.printStackTrace();  }  return connection;  }  } |

|  |
| --- |
| package com.foo.jdbc;  import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.PreparedStatement;  import java.sql.ResultSet;  import java.sql.SQLException;  import java.util.Vector;  public class PoolTest  {  private void test() throws ClassNotFoundException, SQLException  {  String sql = "select \* from hr.employees where employee\_id < ? and employee\_id >= ?";  PreparedStatement st = null;  ResultSet rs = null;  Connection con = null;  Class.forName("oracle.jdbc.driver.OracleDriver");  try  {  con = DriverManager.getConnection("jdbc:oracle:thin:@127.0.0.1:1521:xe", "louluan", "123456");  st = con.prepareStatement(sql);  //设置参数  st.setInt(1, 101);  st.setInt(2, 0);  //查询，得出结果集  rs = st.executeQuery();  //取数据，省略  //将不再使用的Connection对象放到连接池中，供以后使用  Pool.putConnection(con);  }  catch (SQLException e)  {  e.printStackTrace();  }  }  } |

将使用过的Connection对象放到Pool连接池中，需Connection对象的话，只需用Pool.getConnection()方法从里面取即可。

是的,上述的代码完全可实现此能力，不过有一个很不优雅的实现：就是需手动地将Connection对象放到Pool连接池中，这是一个很傻的实现方式。这也和一般使用Connection对象的方式不一样：一般使用Connection的方式是使用完后，然后调用.close()方法释放资源。

为了和一般的使用Conneciton对象的方式保持一致，希望当Connection使用完后，调用.close()方法，而实际上Connection资源并没有被释放，而实际上被添加到了连接池中。怎样实现：

用代理模式，为真正的Connection对象创建一个代理对象，代理对象所有的方法都是调用相应的真正Connection对象的方法实现。当代理对象执行close()方法时，要特殊处理，不调用真正Connection对象的close()方法，而是将Connection对象添加到连接池中。

MyBatis的PooledDataSource的PoolState内部维护的对象是PooledConnection类型的对象，而PooledConnection则是对真正的数据库连接java.sql.Connection实例对象的包裹器。

PooledConnection对象内持有一个真正的数据库连接java.sql.Connection实例对象和一个java.sql.Connection的代理：

其部分定义如下：

|  |
| --- |
| class PooledConnection implements InvocationHandler {    //......  //所创建它的datasource引用  private PooledDataSource dataSource;  //真正的Connection对象  private Connection realConnection;  //代理自己的代理Connection  private Connection proxyConnection;    //......  } |

调用PooledDataSource的getConnection()方法返回的就是这个proxyConnection对象。

调用此proxyConnection对象上任何方法时，都会调用PooledConnection对象内invoke()方法。

让我们看一下PooledConnection类中的invoke()方法定义：

|  |
| --- |
| public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  String methodName = method.getName();  //当调用关闭时，回收此Connection到PooledDataSource中  if (CLOSE.hashCode() == methodName.hashCode() && CLOSE.equals(methodName)) {  dataSource.pushConnection(this);  return null;  } else {  try {  if (!Object.class.equals(method.getDeclaringClass())) {  checkConnection();  }  return method.invoke(realConnection, args);  } catch (Throwable t) {  throw ExceptionUtil.unwrapThrowable(t);  }  }  } |

可看到，当使用了pooledDataSource.getConnection()返回的Connection对象的close()方法时，不会调用真正Connection的close()方法，而是将此Connection对象放到连接池中。

七、JNDI类型的数据源DataSource

对JNDI类型的数据源的获取比较简单，MyBatis定义了一个JndiDataSourceFactory工厂来创建通过JNDI形式生成的DataSource。

下面看一下JndiDataSourceFactory的关键代码：

|  |
| --- |
| if (properties.containsKey(INITIAL\_CONTEXT)  && properties.containsKey(DATA\_SOURCE))  {  //从JNDI上下文中找到DataSource并返回  Context ctx = (Context) initCtx.lookup(properties.getProperty(INITIAL\_CONTEXT));  dataSource = (DataSource) ctx.lookup(properties.getProperty(DATA\_SOURCE));  }  else if (properties.containsKey(DATA\_SOURCE))  {  // //从JNDI上下文中找到DataSource并返回  dataSource = (DataSource) initCtx.lookup(properties.getProperty(DATA\_SOURCE));  } |

# 12.[W] Mybatis原理

<http://blog.csdn.net/d12345678a/article/details/53956485>

|  |
| --- |
| 支持定制化SQL、存储过程及高级映射的持久层框架。  避免了几乎所有的 JDBC 代码和手工设置参数及抽取结果集。  用简单的XML或注解来配置和映射基本体，将接口和Java的POJOs(Plain Old Java Objects, 普通的Java对象)映射成数据库中记录。  **原理详解**：  根据XML配置文件创建SqlSessionFactory，SqlSessionFactory在根据配置，配置来源于两个地方，一处是配置文件，一处是Java代码的注解，获取一个SqlSession。SqlSession包含了执行sql所需要的所有方法，可通过SqlSession实例直接运行映射的sql语句，完成对数据的增删改查和事务提交等，用完之后关闭SqlSession。  通过factory方法获取sqlsession----通过MapperProxy代理到dao--执行底层数据库操作  优点：  1、简单易学  mybatis本身就很小且简单。没第三方依赖，只要两jar文件+配置几个sql映射文件易于学习，使用，通过文档和源代码，可较完全的掌握设计思路和实现。  2、灵活  不会对应用程序或数据库的现有设计强加任何影响。  sql写在xml里，便于统一管理和优化。  通过sql基本上可实现不使用数据访问框架可实现的所有功能，或许更多。  3、解除sql与程序代码的耦合  通过提供DAL层，将业务逻辑和数据访问逻辑分离，使系统的设计更清晰，更易维护，更易单元测试。sql和代码的分离，提高了可维护性。  4、提供映射标签，支持对象与数据库的orm字段关系映射  5、提供对象关系映射标签，支持对象关系组建维护  6、提供xml标签，支持编写动态sql。  **缺点：**  1、编写SQL语句时工作量大，尤其是字段多、关联表多时。  2、SQL语句依赖于数据库，导致数据库移植性差，不能更换数据库。  3、框架还比较简陋，功能有缺失，虽简化了数据绑定代码，但整个底层数据库查询实际还是要自己写，工作量也比较大，且不太容易适应快速数据库修改。  4、二级缓存机制不佳   * **MyBatis的框架设计**     1.接口层---和数据库交互的方式  1.1.使用传统的MyBatis提供的API  传递Statement Id 和查询参数给SqlSession对象，用SqlSession对象完成和数据库的交互；MyBatis提供了方便和简单的API，供用户实现对数据库的增删改查，及对数据库连接信息和MyBatis自身配置信息的维护。    创建一个和数据库打交道的SqlSession对象，根据Statement Id 和参数操作数据库，简单实用，但不符合面向对象语言的概念和面向接口编程的习惯。  由于面向接口的编程是面向对象的大趋势，MyBatis为了适应这一趋势，增加了第二种使用MyBatis支持接口（Interface）调用方式。  1.2. 使用Mapper接口  MyBatis将配置文件中的每一个<mapper>节点抽象为一个 Mapper 接口，而这个接口中声明的方法和跟<mapper>节点中的<select|update|delete|insert>节点项对应，即<select|update|delete|insert>节点的id值为Mapper接口中的方法名称，parameterType值表Mapper对应方法的入参类型，而resultMap值则对应了Mapper接口表示的返回值类型或返回结果集的元素类型。    根据MyBatis的配置规范配置好后，通过SqlSession.getMapper(XXXMapper.class) 方法，MyBatis会根据相应的接口声明的方法，通过动态代理机制生成一个Mapper实例，使用Mapper接口的某一个方法时，MyBatis会根据这个方法的方法名和参数类型，确定Statement Id，底层还是通过SqlSession.select("statementId",parameterObject);或SqlSession.update("statementId",parameterObject);等来实现对数据库的操作  MyBatis引用Mapper接口这种调用方式，纯粹是为了满足面向接口编程的需要。（其实还有一个原因是在于，面向接口的编程，使得用户在接口上可使用注解来配置SQL语句，这样就可以脱离XML配置文件，实现“0配置”）。  2.**数据处理层**  可以说是MyBatis的核心，要完成三个功能：  a. 通过传入参数构建动态SQL语句；  b.可使用拦截器重组SQL语句；（补充）  c.SQL语句的执行及封装查询结果集成List<E>  2.1.参数映射和动态SQL语句生成  MyBatis通过传入的参数值，用Ognl 动态构造SQL语句，有很强的灵活性和扩展性。  参数映射:Java数据类型和jdbc数据类型间的转换：两个过程  查询阶段，要将java类型数据，转换成jdbc类型数据，通过preparedStatement.setXXX()来设值；另一个就是对resultset查询结果集的jdbcType数据转换成java数据类型。  2.2. SQL语句的执行及封装查询结果集成List<E>  动态SQL语句生成后，MyBatis将执行SQL语句，并将可能返回的结果集转换成List<E>列表。  MyBatis在对结果集的处理中，支持结果集关系一对多和多对一的转换，且有两种支持方式，一种为嵌套查询语句的查询，还有一种是嵌套结果集的查询。  3. **框架支撑层**  3.1. **事务管理机制**  事务管理机制的质量也是考量一ORM框架是否优秀的一标准  http://blog.csdn.net/luanlouis/article/details/37992171  3.2. **连接池管理机制**  由于创建一数据库连接所占用的资源较大，对于数据吞吐量大和访问量大的应用，连接池的设计就显得非常重要  http://blog.csdn.net/luanlouis/article/details/37671851  3.3.[F] **缓存机制**  为提高数据利用率和减小服务器和数据库压力，MyBatis会对于一些查询提供会话级别的数据缓存，会将对某一次查询，放置到SqlSession中，在允许的时间间隔内，对于完全相同的查询，MyBatis会直接将缓存结果返回给用户，而不用再到数据库中查找。（至于具体的MyBatis缓存机制，我将准备专门一篇文章来讨论，敬请关注~）  3. 4. **SQL语句的配置方式**  传统方式是用XML文件，但不能很好地支持面向接口编程的理念  为支持面向接口的编程，MyBatis引入了Mapper接口的概，对使用注解来配置SQL语句成为可能，用户只需在接口上添加必要的注解即可，不用再配置XML文件了，但目前的MyBatis只是对注解配置SQL语句提供了有限的支持，某些高级功能还是要依赖XML配置文件配置SQL语句。  4 **引导层**  是配置和启动MyBatis配置信息的方式。  MyBatis提供两种方式来引导MyBatis：  基于XML配置文件的方式和基于JavaAPI的方式  http://blog.csdn.net/luanlouis/article/details/37671851  二、MyBatis的主要构件及其相互关系  从代码实现角度，主要的核心部件：  SqlSession MyBatis工作的顶层API，表示和数据库交互的会话，完成数据库增删改查  Executor MyBatis执行器，MyBatis调度的核心，负责SQL语句的生成和查询缓存的维护  StatementHandler 封装了JDBC Statement操作，负责对JDBC statement 的操作，如设置参数、将Statement结果集转换成List集合。  ParameterHandler 负责对用户传递的参数转换成JDBC Statement 所需的参数，  ResultSetHandler 负责将JDBC返回的ResultSet结果集对象转换成List类型的集合；  TypeHandler 负责java数据类型和jdbc数据类型间的映射和转换  MappedStatement MappedStatement维护了一条<select|update|delete|insert>节点的封装，  SqlSource 负责根据用户传递的parameterObject，动态生成SQL语句，将信息封装  到BoundSql对象中，并返回  BoundSql 表动态生成的SQL语句及相应的参数信息  Configuration MyBatis所有的配置信息都维持在Configuration对象中。  关系如下图所示： |