# Mybatis自定义框架及使用

## 文档结构

### 1.1目录视图

1. Mybatis入门

·mybatis的概述

·mybatis的环境搭建

·mybatis入门案例

1. mybatis基本使用

·mybatis的单表curd操作

·mybatis的参数和返回值

·mybatis的dao编写

·mybatis的配置细节

几个标签的使用

1. mybatis的深入和多表

·mybatis的连接池

·mybatis的事务控制及设计的方法

·mybatis的多表查询

一对多（多对一）

多对多

1. mybatis的缓存和注解开发

·mybatis中的加载时机（查询的时机）

·mybatis的一级缓存和二级缓存

·mybatis的注解开发

单表CURD

多表查询

## 第2节 框架概述

### 2.1什么是框架

（1）什么是框架

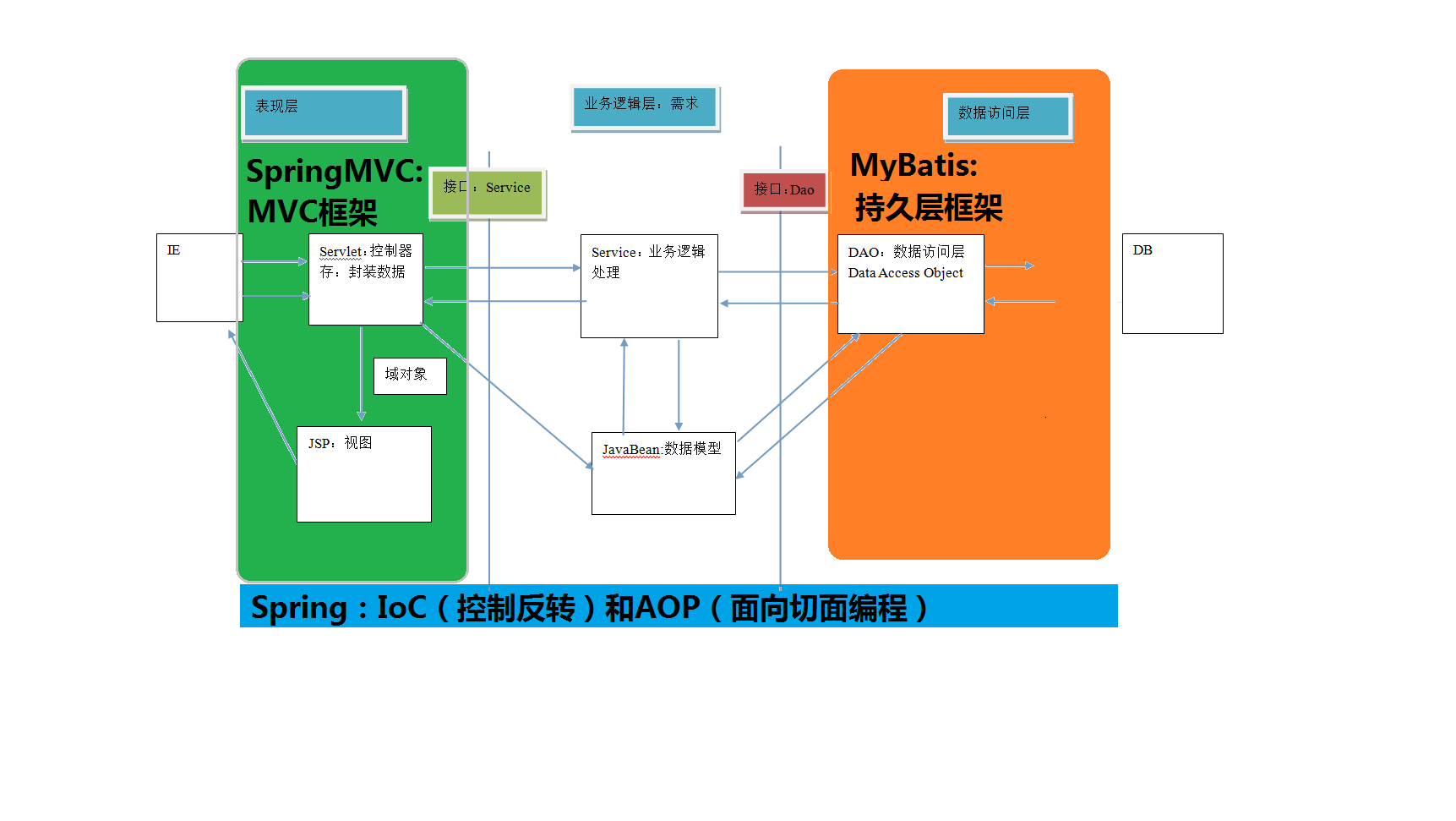
框架（Framework）是整个或部分系统的可重用设计，表现为一组抽象构件及构件实例间交互的方法;另一种定义认为，框架是可被应用开发者定制的应用骨架。前者是从应用方面而后者是从目的方面给出的定义简而言之，框架其实就是某种应用的半成品，就是一组组件，供你选用完成你自己的系统。简单说就是使用别人搭好的舞台，你来做表演。而且，框架一般是成熟的，不断升级的软件。

他是我们软件开发中的一套解决方案，不同框架解决不同的问题。

使用框架的好处：

框架封装了很多细节，使开发者能够提高开发效率，使用极简的方式实现功能。大大提高开发效率

1. 框架要解决的问题



·应用自身的设计与具体实现解耦；

·框架一般处在底层应用平台（如J2EE）和高层业务逻辑之间的中间层

（3）软件开发分层

·表现层：

用于展示数据

·业务层：

处理业务需求

·持久层

和数据库交互

（4）持久层技术解决方案

JDBC技术：

Connection

PreparedStatement

ResultStatement

String的JdbcTemplate：

String对JDBC的简单封装

Apache的DBUtils：

他和Spring的JdbcTemplate很像，也是对Jdbc的简单封装

以上都不是框架

JDBC是规范

Spring的JdbcTemplate和Apache的DBUtils都只是工具类

1. MyBatis概述

基于Java的持久层框架，内部封装了jdbc，使开发者需要关注sql语句半数，而不需要花费精力去处理加载驱动、创建连接、创建statement等繁杂的过程。

注解

XML

ORM：Object Relational Mapping对象关系映射

简单的说：

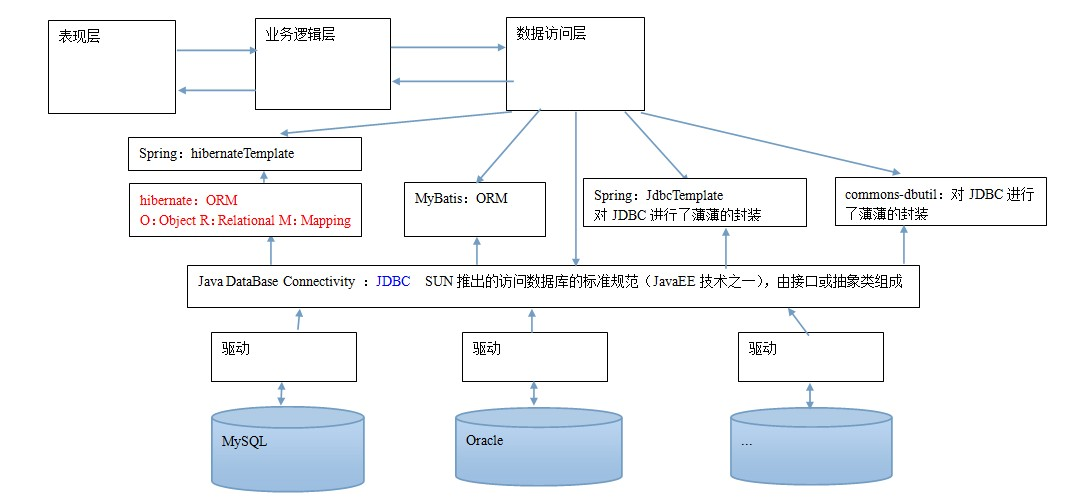
就是把数据库表和实体类及实体类的属性对应起来

让我们可以操作实体类就实现数据库操作

Uase\_table User

id userId

User\_name userName



### 2.1JMybatis的入门

1. Mybatis的环境搭建

第一步：创建maven工程并导入坐标

第二步：创建实体类和dao的接口

第三步：创建mybatis的主配置文件

SqlMapConfig.xml

第四步：创建映射配置文件

IUserDao.xml

环境搭建的注意事项：

第一个：创建IUserDao.xml和IUserDao.java时，名称是为了和我们以前的知识保持一致。在Mybatis中它把持久层的操作接口名称和映射文件也叫作Mapper

所以：IUserDao和IUserMapper是一样的

第二个：在idea中创建目录的时候，它和包是不一样的

包在创建时：com.byf.dao他是三级结构

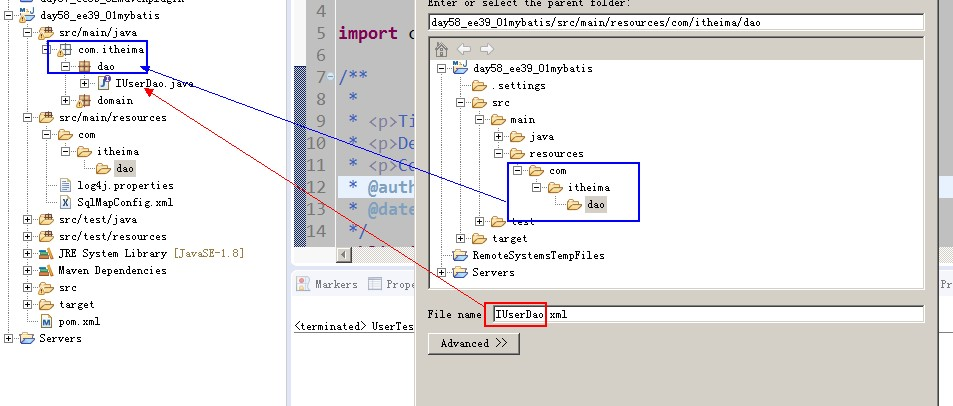
目录在创建时 com/byf/dao是三级目录，但com.byf.dao是一级目录

第三个：mybatis的映射配置文件必须和dao接口包结构相同

第四个：映射配置文件的mapper标签namespace属性的取值必须是dao接口的全限定类名

第五个：映射配置文件的操作配置（select），id属性的取值必须是dao接口的方法名。

当我们遵从了三四五点之后，我们在开发中就无需再写dao的实现类



### 2.3环境搭建

|  |
| --- |
| @ToString **public class** User **implements** Serializable {  **private** Integer **id**;  **private** String **username**;  **private** Date **birthday**;  **private** String **sex**;  **private** String **address**;   **public** Integer getId() {  **return id**;  }   **public void** setId(Integer id) {  **this**.**id** = id;  }   **public** String getUsername() {  **return username**;  }   **public void** setUsername(String username) {  **this**.**username** = username;  }   **public** Date getBirthday() {  **return** (Date) **birthday**.clone();  }   **public void** setBirthday(Date birthday) {  **this**.**birthday** = birthday;  }   **public** String getSex() {  **return sex**;  }   **public void** setSex(String sex) {  **this**.**sex** = sex;  }   **public** String getAddress() {  **return address**;  }   **public void** setAddress(String address) {  **this**.**address** = address;  } } |
| **public interface** IUserDao {  */\*\*  \* 查询所有操作  \** ***@return*** *\*/* List<User> findAll(); } |

G:\Mybatis\mybatis01\src\main\resources\SqlMapConfig.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**>  *<!-- 配置环境 -->* <**environments default="mysql"**>  *<!-- 配置mysql环境 -->* <**environment id="mysql"**>  *<!-- 配置事务类型 -->* <**transactionManager type="JDBC"**></**transactionManager**>  *<!-- 配置数据源（连接池） -->* <**dataSource type="POOLED"**>  <**property name="driver" value="com.mysql.jdbc.Driver"**/>  <**property name="url" value="jdbc:mysql://localhost:3306/eesy\_mybatis"**/>  <**property name="username" value="root"**/>  <**property name="password" value="1234"**/>  </**dataSource**>  </**environment**>  </**environments**>   *<!--指定映射配置文件的位置，映射配置文件指的是每个dao独立的配置文件 -->* <**mappers**>  <**mapper resource="com/byf/dao/IUserDao.xml"**></**mapper**>  </**mappers**> </**configuration**> |

G:\Mybatis\mybatis01\src\main\resources\com\byf\dao\IUserDao.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="com.byf.dao.IUserDao"**>  *<!-- 配置查询所有 -->* <**select id="findAll"**>  select *\** from user  </**select**>  </**mapper**> |

G:\Mybatis\mybatis01\src\main\resources\log4j.properties

|  |
| --- |
| *# Set root category priority to INFO and its only appender to CONSOLE. #log4j.rootCategory=INFO, CONSOLE debug info warn error fatal* **log4j.rootCategory**=**debug, CONSOLE, LOGFILE** *# Set the enterprise logger category to FATAL and its only appender to CONSOLE.* **log4j.logger.org.apache.axis.enterprise**=**FATAL, CONSOLE** *# CONSOLE is set to be a ConsoleAppender using a PatternLayout.* **log4j.appender.CONSOLE**=**org.apache.log4j.ConsoleAppender log4j.appender.CONSOLE.layout**=**org.apache.log4j.PatternLayout log4j.appender.CONSOLE.layout.ConversionPattern**=**%d{ISO8601} %-6r [%15.15t] %-5p %30.30c %x - %m\n** *# LOGFILE is set to be a File appender using a PatternLayout.* **log4j.appender.LOGFILE**=**org.apache.log4j.FileAppender log4j.appender.LOGFILE.File**=**d:**\a**xis.log log4j.appender.LOGFILE.Append**=**true log4j.appender.LOGFILE.layout**=**org.apache.log4j.PatternLayout log4j.appender.LOGFILE.layout.ConversionPattern**=**%d{ISO8601} %-6r [%15.15t] %-5p %30.30c %x - %m\n** |

|  |
| --- |
| **public class** MybatisTest {  @Test  **public void** testConfig() **throws** IOException {  *// 1.读取配置文件* InputStream in = Resources.*getResourceAsStream*(**"SqlMapConfig.xml"**);  *// 2.创建SqlSessionFactory工厂* SqlSessionFactoryBuilder builder = **new** SqlSessionFactoryBuilder();  SqlSessionFactory factory = builder.build(in);  *//3.使用工厂生产生产SqlSession对象* SqlSession session = factory.openSession();  *//4.使用SqlSession创建Dao接口的代理对象* IUserDao userDao = session.getMapper(IUserDao.**class**);  *//5.使用代理对象执行方法* List<User> users = userDao.findAll();  **for** (User user:users){  System.***out***.println(user);  }  *//6.释放资源* session.close();  } } |
| **[ main] DEBUG ansaction.jdbc.JdbcTransaction - Opening JDBC Connection**  **[ main] DEBUG source.pooled.PooledDataSource - Created connection 422392391.**  **[ main] DEBUG ansaction.jdbc.JdbcTransaction - Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@192d3247]**  **[ main] DEBUG com.byf.dao.IUserDao.findAll - ==> Preparing: select \* from user**  **[ main] DEBUG com.byf.dao.IUserDao.findAll - ==> Parameters:**  **[ main] DEBUG com.byf.dao.IUserDao.findAll - <== Total: 6**  **User(id=41, username=Jack, birthday=Tue Feb 27 17:47:08 CST 2018, sex=男, address=北京)**  **User(id=42, username=Tom, birthday=Fri Mar 02 15:09:37 CST 2018, sex=女, address=北京金燕龙)**  **User(id=43, username=Jerry, birthday=Sun Mar 04 11:34:34 CST 2018, sex=女, address=北京金燕龙)**  **User(id=45, username=Bob, birthday=Sun Mar 04 12:04:06 CST 2018, sex=男, address=北京金燕龙)**  **User(id=46, username=Alice, birthday=Wed Mar 07 17:37:26 CST 2018, sex=男, address=北京)**  **User(id=48, username=John, birthday=Thu Mar 08 11:44:00 CST 2018, sex=女, address=北京修正)**  **[ main] DEBUG ansaction.jdbc.JdbcTransaction - Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@192d3247]**  **[ main] DEBUG ansaction.jdbc.JdbcTransaction - Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@192d3247]**  **[ main] DEBUG source.pooled.PooledDataSource - Returned connection 422392391 to pool.** |

### 2.4Mybatis中的设计模式

为什么在拿到代理对象执行方法前，要增加中间的创建类：

为了能够利用重载实现更灵活的配置加载

|  |
| --- |
| **public class** MybatisTest {  @Test  **public void** testConfig() **throws** IOException {  */\*\*  \* 1.读取配置文件  \* 1）绝对路径：d:/xxx/xxx.xml 不可用  \* 2）相对路径：src/java/main/xxx.xml 不可用  \* 3）使用类加载器，他只能兑取类路径的配置文件  \* 4）使用ServletContext对象的getRealPath()  \*/* InputStream in = Resources.*getResourceAsStream*(**"SqlMapConfig.xml"**);   */\*\*  \* 2.创建SqlSessionFactory工厂  \* 1）创建工厂mybatis使用了构建者模式  \* 优势：把对象创建的细节。因此，使用者直接调用对象  \*/* SqlSessionFactoryBuilder builder = **new** SqlSessionFactoryBuilder();  SqlSessionFactory factory = builder.build(in);   */\*\*  \* 3.使用工厂生产生产SqlSession对象  \* 1）生成SqlSession使用了工厂模式  \* 优势：解耦（降低类之间的依赖关系）  \*/* SqlSession session = factory.openSession();   */\*\*  \* 4.使用SqlSession创建Dao接口的代理对象  \* 1）创建Dao接口实现类使用了代理模式  \* 优势：不修改源码的基础上，增强方法  \*/* IUserDao userDao = session.getMapper(IUserDao.**class**);  *//5.使用代理对象执行方法* List<User> users = userDao.findAll();  **for** (User user:users){  System.***out***.println(user);  }  *//6.释放资源* session.close();  } } |

### 2.5mybatis注解开发和编写dao实现类

1. 删除resources下的dao的mapper文件
2. 在IUserDao接口方法上添加注解
3. 修改SqlMapConfig，映射到IUserDao接口类

|  |
| --- |
| **public interface** IUserDao {  */\*\*  \* 查询所有操作  \** ***@return*** *\*/* @Select(**"select \* from user"**)  List<User> findAll(); } |
| *<!--指定映射配置文件的位置，映射配置文件指的是每个dao独立的配置文件  如果是用注解来配置的话，此处应该使用class属性指定被注解的dao全类名 -->* <**mappers**>  *<!--<mapper resource="com/byf/dao/IUserDao.xml"></mapper>-->* <**mapper class="com.byf.dao.IUserDao"**></**mapper**> </**mappers**> |

Mybatis入门案例：

1. 读取配置文件；
2. 创建SqlSessionFactory工厂；
3. 创建SqlSession对象；
4. 创建Dao接口的代理对象实现类；
5. 执行Dao的接口方法；
6. 关闭/释放SqlSession资源。

注意事项：

不要忘记在映射配置中告诉mybatis要封装到哪个实体类中

配置的方式：指定实体类的权限的类名；

明确：

实际开发中一般不去下dao的实现类；

不管是用XML还是注解

但Mybatis它是支持写dao实现类的。

|  |
| --- |
| **public class** UserDaoImpl **implements** IUserDao {  **private** SqlSessionFactory **factory**;   **public** UserDaoImpl(SqlSessionFactory factory){  **this**.**factory** = factory;  }  **public** List<User> findAll() {  *// 1.使用工程创建SqlSession对象* SqlSession session = **factory**.openSession();  *// 2.使用session执行查询* List<User> users = session.selectList(**"com.byf.dao.IUserDao.findAll"**);  *// 3.返回查询结果* session.close();  **return** users;  } } |

|  |
| --- |
| **public class** MybatisTest {  @Test  **public void** testConfig() **throws** IOException {  */\*\*  \* 1.读取配置文件  \* 1）绝对路径：d:/xxx/xxx.xml 不可用  \* 2）相对路径：src/java/main/xxx.xml 不可用  \* 3）使用类加载器，他只能兑取类路径的配置文件  \* 4）使用ServletContext对象的getRealPath()  \*/* InputStream in = Resources.*getResourceAsStream*(**"SqlMapConfig.xml"**);   */\*\*  \* 2.创建SqlSessionFactory工厂  \* 1）创建工厂mybatis使用了构建者模式  \* 优势：把对象创建的细节。因此，使用者直接调用对象  \*/* SqlSessionFactoryBuilder builder = **new** SqlSessionFactoryBuilder();  SqlSessionFactory factory = builder.build(in);   */\*\*  \* 4.使用SqlSession创建Dao接口的代理对象  \* 1）创建Dao接口实现类使用了代理模式  \* 优势：不修改源码的基础上，增强方法  \*/* IUserDao userDao = **new** UserDaoImpl(factory);   *//5.使用代理对象执行方法* List<User> users = userDao.findAll();  **for** (User user:users){  System.***out***.println(user);  }  *//6.释放资源* in.close();  } } |
| **19:37,079 [ main] DEBUG source.pooled.PooledDataSource - PooledDataSource forcefully closed/removed all connections.**  **19:37,349 [ main] DEBUG ansaction.jdbc.JdbcTransaction - Opening JDBC Connection**  **19:37,749 [ main] DEBUG source.pooled.PooledDataSource - Created connection 249155636.**  **19:37,749 [ main] DEBUG ansaction.jdbc.JdbcTransaction - Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@ed9d034]**  **19:37,759 [ main] DEBUG com.byf.dao.IUserDao.findAll - ==> Preparing: select \* from user**  **19:37,799 [ main] DEBUG com.byf.dao.IUserDao.findAll - ==> Parameters:**  **19:37,889 [ main] DEBUG com.byf.dao.IUserDao.findAll - <== Total: 6**  **19:37,889 [ main] DEBUG ansaction.jdbc.JdbcTransaction - Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@ed9d034]**  **19:37,889 [ main] DEBUG ansaction.jdbc.JdbcTransaction - Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@ed9d034]**  **19:37,889 [ main] DEBUG source.pooled.PooledDataSource - Returned connection 249155636 to pool.**  **User(id=41, username=Jack, birthday=Tue Feb 27 17:47:08 CST 2018, sex=男, address=北京)** |

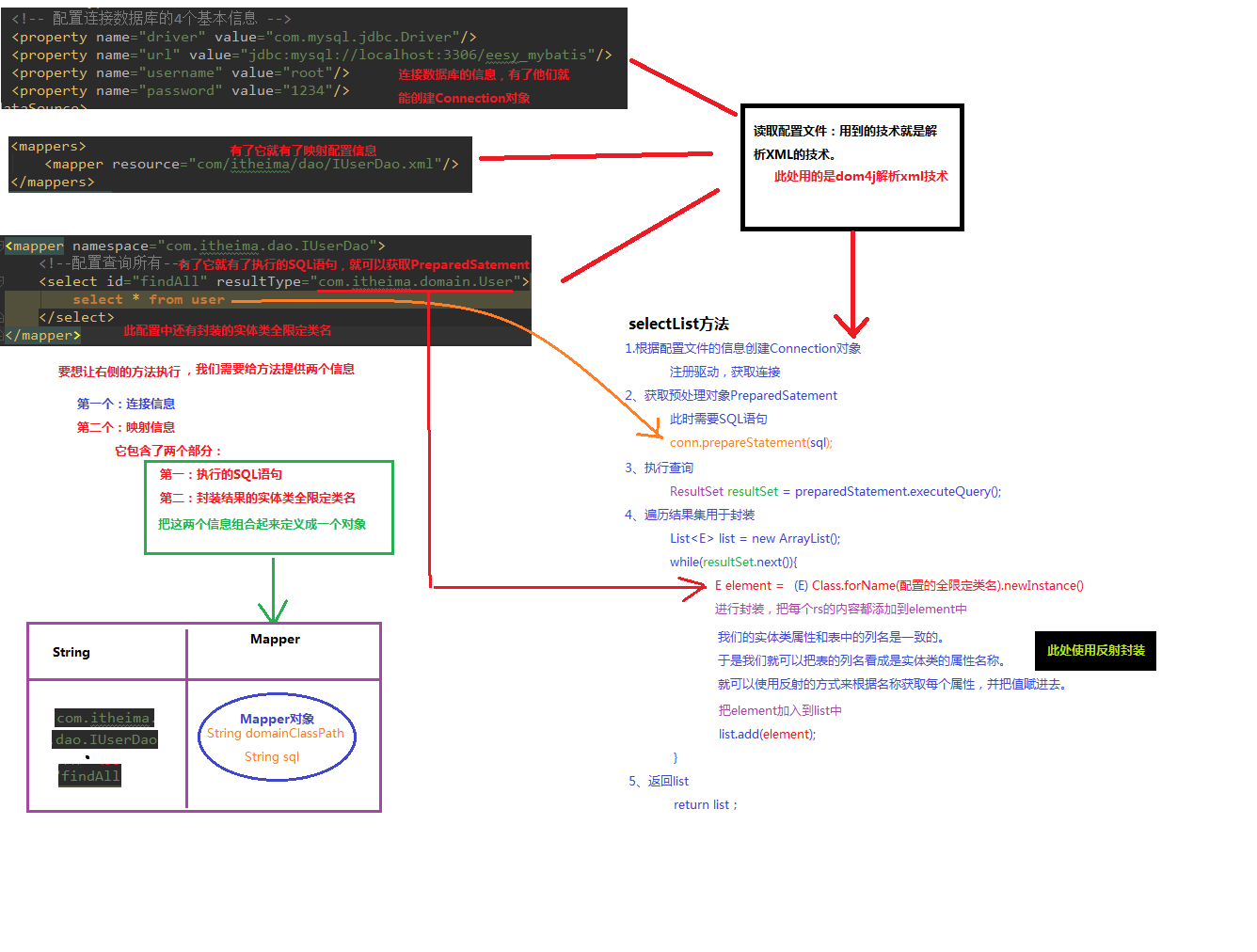
## 第3节自定义Mybatis框架.

### 3.1mybatis的分析

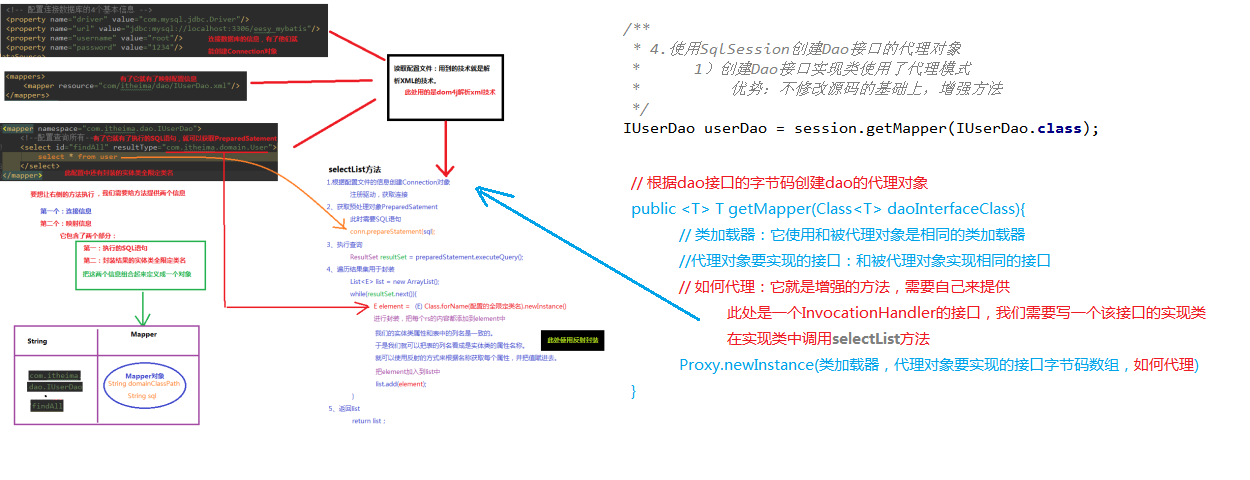
Mybatis使用代理dao方式实现CURD时做了两件事：

1. 创建代理对象
2. 在代理对象中调用selectList方法

**查询所有的分析：**



**创建代理对的分析：**



### 3.2自定义Mybats

**import** org.apache.ibatis.io.Resources;  
**import** org.apache.ibatis.session.SqlSession;  
**import** org.apache.ibatis.session.SqlSessionFactory;  
**import** org.apache.ibatis.session.SqlSessionFactoryBuilder;

**class** Resources

**class** SqlSessionFactoryBuilder

**interface** SqlSessionFactory

**interface** SqlSession

### 3.3根据测试类中缺少的创建接口和类

（1）删除Maven依赖

|  |
| --- |
| *<!--<dependency>  <groupId>org.mybatis</groupId>  <artifactId>mybatis</artifactId>  <version>3.4.6</version> </dependency>-->* |

（2）创建缺少的接口和类

|  |
| --- |
| **public class** Resources {  **public static** InputStream getResourceAsStream(String filePath){  **return** Resources.**class**.getClassLoader().getResourceAsStream(filePath);  } } |

|  |
| --- |
| **public class** SqlSessionFactoryBuilder {  */\*\*  \* 根据入参解析xml  \** ***@param config*** *\** ***@return*** *\*/* **public** SqlSessionFactory build(InputStream config){  **return null**;  } } |

|  |
| --- |
| **public interface** SqlSessionFactory {  SqlSession openSession(); } |

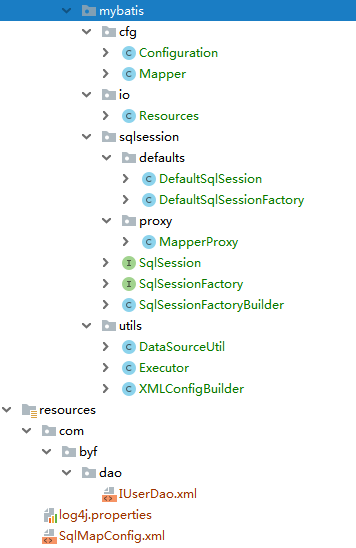
|  |
| --- |
| **public interface** SqlSession {  */\*\*  \*  \** ***@param daoInterfaceClass*** *: dao接口的字节码  \** ***@param <T>*** *\** ***@return*** *\*/* <T> T getMapper(Class<T> daoInterfaceClass);   */\*\*  \* 释放资源  \*/* **void** close(); } |

### 3.4创建两个默认的实现类

|  |
| --- |
| */\*\*  \* SqlSession接口的实现类  \*/* **public class** DefaultSqlSession **implements** SqlSession {   **private** Configuration **cfg**;   **public** DefaultSqlSession(Configuration cfg) {  **this**.**cfg** = cfg;  }   */\*\*  \* 创建代理对象  \** ***@param daoInterfaceClass*** *: dao接口的字节码  \** ***@param <T>*** *\** ***@return*** *\*/* **public** <T> T getMapper(Class<T> daoInterfaceClass) {  **return null**;  }   */\*\*  \* 释放资源  \*/* **public void** close() {   } } |

|  |
| --- |
| **public class** DefaultSqlSessionFactory **implements** SqlSessionFactory {  **private** Configuration **cfg**;   **public** DefaultSqlSessionFactory(Configuration cfg) {  **this**.**cfg** = cfg;  }   */\*\*  \* 创建一个新的操作数据库对象  \** ***@return*** *\*/* **public** SqlSession openSession() {  **return new** DefaultSqlSession(**cfg**);  } } |

### 3.5实现基于XML查询所有操作



|  |
| --- |
| @Data **public class** Configuration {  **private** String **driver**;  **private** String **url**;  **private** String **username**;  **private** String **password**;  **private** Map<String, Mapper> **mappers** = **new** HashMap<String,Mapper>();;   **public** Map<String, Mapper> getMappers() {  **return mappers**;  }   **public void** setMappers(Map<String, Mapper> mappers) {  **this**.**mappers**.putAll(mappers);  } } |

|  |
| --- |
| @Data **public class** Mapper {  **private** String **queryString**;  **private** String **resultType**; } |

|  |
| --- |
| **public class** Resources {  **public static** InputStream getResourceAsStream(String filePath){  **return** Resources.**class**.getClassLoader().getResourceAsStream(filePath);  } } |

|  |
| --- |
| */\*\*  \* SqlSession接口的实现类  \*/* **public class** DefaultSqlSession **implements** SqlSession {   **private** Configuration **cfg**;  **private** Connection **conn**;    **public** DefaultSqlSession(Configuration cfg) {  **this**.**cfg** = cfg;  **this**.**conn** = DataSourceUtil.*getConnection*(cfg);  }   */\*\*  \* 创建代理对象  \** ***@param daoInterfaceClass*** *: dao接口的字节码  \** ***@param <T>*** *\** ***@return*** *\*/* **public** <T> T getMapper(Class<T> daoInterfaceClass) {  **return** (T) Proxy.*newProxyInstance*(daoInterfaceClass.getClassLoader(),  **new** Class[]{daoInterfaceClass},**new** MapperProxy(**cfg**.getMappers(),**conn**));  *//return null;* }   */\*\*  \* 释放资源  \*/* **public void** close() {  **if** (**conn** != **null**){  **try** {  **conn**.close();  } **catch** (SQLException e) {  e.printStackTrace();  }  }  } } |

|  |
| --- |
| **public class** DefaultSqlSessionFactory **implements** SqlSessionFactory {  **private** Configuration **cfg**;   **public** DefaultSqlSessionFactory(Configuration cfg) {  **this**.**cfg** = cfg;  }   */\*\*  \* 创建一个新的操作数据库对象  \** ***@return*** *\*/* **public** SqlSession openSession() {  **return new** DefaultSqlSession(**cfg**);  } } |

|  |
| --- |
| **public class** MapperProxy **implements** InvocationHandler {  **private** Map<String, Mapper> **mappers**;  **private** Connection **conn**;   **public** MapperProxy(Map<String, Mapper> mappers, Connection conn) {  **this**.**mappers** = mappers;  **this**.**conn** = conn;  }   */\*\*  \* 用于增强方法，增强的就是调用selectList方法  \** ***@param proxy*** *\** ***@param method*** *\** ***@param args*** *\** ***@return*** *\** ***@throws*** *Throwable  \*/* **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {  *// 1.获取方法名* String methodName = method.getName();  *// 2.获取方法所在的类名* String className = method.getDeclaringClass().getName();  *// 3.组合key* String key = className+**"."**+methodName;  *// 4.获取mappers中的Mapper对象* Mapper mapper = **mappers**.get(key);  *// 5.判断是否有mapper* **if** (mapper == **null**){  **throw new** IllegalArgumentException(**""**);  }  *// 6.调用工具类执行查询所有* **return new** Executor().selectList(mapper,**conn**);  } } |

|  |
| --- |
| **public interface** SqlSession {  */\*\*  \*  \** ***@param daoInterfaceClass*** *: dao接口的字节码  \** ***@param <T>*** *\** ***@return*** *\*/* <T> T getMapper(Class<T> daoInterfaceClass);   */\*\*  \* 释放资源  \*/* **void** close(); } |

|  |
| --- |
| **public interface** SqlSessionFactory {  SqlSession openSession(); } |

|  |
| --- |
| **public class** SqlSessionFactoryBuilder {  */\*\*  \* 根据入参解析xml  \** ***@param config*** *\** ***@return*** *\*/* **public** SqlSessionFactory build(InputStream config){  Configuration cfg = XMLConfigBuilder.*loadConfiguration*(config);  **return new** DefaultSqlSessionFactory(cfg);  } } |

|  |
| --- |
| *<!--<dependency>  <groupId>org.mybatis</groupId>  <artifactId>mybatis</artifactId>  <version>3.4.6</version> </dependency>-->* <**dependency**>  <**groupId**>dom4j</**groupId**>  <**artifactId**>dom4j</**artifactId**>  <**version**>1.6.1</**version**> </**dependency**> <**dependency**>  <**groupId**>jaxen</**groupId**>  <**artifactId**>jaxen</**artifactId**>  <**version**>1.1.6</**version**> </**dependency**> |

|  |
| --- |
| **public class** DataSourceUtil {  **public static** Connection getConnection(Configuration cfg){  **try** {  Class.*forName*(cfg.getDriver());  **return** DriverManager.*getConnection*(cfg.getUrl(),cfg.getUsername(),cfg.getPassword());  } **catch** (Exception e) {  **throw new** RuntimeException(e);  }  } } |
| */\*\*  \* 负责执行SQL语句，并且封装结果集  \*/* **public class** Executor {   **public** <E> List<E> selectList(Mapper mapper, Connection conn) {  PreparedStatement pstm = **null**;  ResultSet rs = **null**;  **try** {  *//1.取出mapper中的数据* String queryString = mapper.getQueryString();*//select \* from user* String resultType = mapper.getResultType();*//com.itheima.domain.User* Class domainClass = Class.*forName*(resultType);  *//2.获取PreparedStatement对象* pstm = conn.prepareStatement(queryString);  *//3.执行SQL语句，获取结果集* rs = pstm.executeQuery();  *//4.封装结果集* List<E> list = **new** ArrayList<E>();*//定义返回值* **while**(rs.next()) {  *//实例化要封装的实体类对象* E obj = (E)domainClass.newInstance();   *//取出结果集的元信息：ResultSetMetaData* ResultSetMetaData rsmd = rs.getMetaData();  *//取出总列数* **int** columnCount = rsmd.getColumnCount();  *//遍历总列数* **for** (**int** i = 1; i <= columnCount; i++) {  *//获取每列的名称，列名的序号是从1开始的* String columnName = rsmd.getColumnName(i);  *//根据得到列名，获取每列的值* Object columnValue = rs.getObject(columnName);  *//给obj赋值：使用Java内省机制（借助PropertyDescriptor实现属性的封装）* PropertyDescriptor pd = **new** PropertyDescriptor(columnName,domainClass);*//要求：实体类的属性和数据库表的列名保持一种  //获取它的写入方法* Method writeMethod = pd.getWriteMethod();  *//把获取的列的值，给对象赋值* writeMethod.invoke(obj,columnValue);  }  *//把赋好值的对象加入到集合中* list.add(obj);  }  **return** list;  } **catch** (Exception e) {  **throw new** RuntimeException(e);  } **finally** {  release(pstm,rs);  }  }    **private void** release(PreparedStatement pstm,ResultSet rs){  **if**(rs != **null**){  **try** {  rs.close();  }**catch**(Exception e){  e.printStackTrace();  }  }   **if**(pstm != **null**){  **try** {  pstm.close();  }**catch**(Exception e){  e.printStackTrace();  }  }  } } |
| */\*\*  \* 用于解析配置文件  \*/* **public class** XMLConfigBuilder {     */\*\*  \* 解析主配置文件，把里面的内容填充到DefaultSqlSession所需要的地方  \* 使用的技术：  \* dom4j+xpath  \*/* **public static** Configuration loadConfiguration(InputStream config){  **try**{  *//定义封装连接信息的配置对象（mybatis的配置对象）* Configuration cfg = **new** Configuration();   *//1.获取SAXReader对象* SAXReader reader = **new** SAXReader();  *//2.根据字节输入流获取Document对象* Document document = reader.read(config);  *//3.获取根节点* Element root = document.getRootElement();  *//4.使用xpath中选择指定节点的方式，获取所有property节点* List<Element> propertyElements = root.selectNodes(**"//property"**);  *//5.遍历节点* **for**(Element propertyElement : propertyElements){  *//判断节点是连接数据库的哪部分信息  //取出name属性的值* String name = propertyElement.attributeValue(**"name"**);  **if**(**"driver"**.equals(name)){  *//表示驱动  //获取property标签value属性的值* String driver = propertyElement.attributeValue(**"value"**);  cfg.setDriver(driver);  }  **if**(**"url"**.equals(name)){  *//表示连接字符串  //获取property标签value属性的值* String url = propertyElement.attributeValue(**"value"**);  cfg.setUrl(url);  }  **if**(**"username"**.equals(name)){  *//表示用户名  //获取property标签value属性的值* String username = propertyElement.attributeValue(**"value"**);  cfg.setUsername(username);  }  **if**(**"password"**.equals(name)){  *//表示密码  //获取property标签value属性的值* String password = propertyElement.attributeValue(**"value"**);  cfg.setPassword(password);  }  }  *//取出mappers中的所有mapper标签，判断他们使用了resource还是class属性* List<Element> mapperElements = root.selectNodes(**"//mappers/mapper"**);  *//遍历集合* **for**(Element mapperElement : mapperElements){  *//判断mapperElement使用的是哪个属性* Attribute attribute = mapperElement.attribute(**"resource"**);  **if**(attribute != **null**){  System.***out***.println(**"使用的是XML"**);  *//表示有resource属性，用的是XML  //取出属性的值* String mapperPath = attribute.getValue();*//获取属性的值"com/itheima/dao/IUserDao.xml"  //把映射配置文件的内容获取出来，封装成一个map* Map<String,Mapper> mappers = *loadMapperConfiguration*(mapperPath);  *//给configuration中的mappers赋值* cfg.setMappers(mappers);  }**else**{  */\*System.out.println("使用的是注解");  //表示没有resource属性，用的是注解  //获取class属性的值  String daoClassPath = mapperElement.attributeValue("class");  //根据daoClassPath获取封装的必要信息  Map<String,Mapper> mappers = loadMapperAnnotation(daoClassPath);  //给configuration中的mappers赋值  cfg.setMappers(mappers);\*/* }  }  *//返回Configuration* **return** cfg;  }**catch**(Exception e){  **throw new** RuntimeException(e);  }**finally**{  **try** {  config.close();  }**catch**(Exception e){  e.printStackTrace();  }  }   }   */\*\*  \* 根据传入的参数，解析XML，并且封装到Map中  \** ***@param mapperPath*** *映射配置文件的位置  \** ***@return*** *map中包含了获取的唯一标识（key是由dao的全限定类名和方法名组成）  \* 以及执行所需的必要信息（value是一个Mapper对象，里面存放的是执行的SQL语句和要封装的实体类全限定类名）  \*/* **private static** Map<String,Mapper> loadMapperConfiguration(String mapperPath)**throws** IOException {  InputStream in = **null**;  **try**{  *//定义返回值对象* Map<String,Mapper> mappers = **new** HashMap<String,Mapper>();  *//1.根据路径获取字节输入流* in = Resources.*getResourceAsStream*(mapperPath);  *//2.根据字节输入流获取Document对象* SAXReader reader = **new** SAXReader();  Document document = reader.read(in);  *//3.获取根节点* Element root = document.getRootElement();  *//4.获取根节点的namespace属性取值* String namespace = root.attributeValue(**"namespace"**);*//是组成map中key的部分  //5.获取所有的select节点* List<Element> selectElements = root.selectNodes(**"//select"**);  *//6.遍历select节点集合* **for**(Element selectElement : selectElements){  *//取出id属性的值 组成map中key的部分* String id = selectElement.attributeValue(**"id"**);  *//取出resultType属性的值 组成map中value的部分* String resultType = selectElement.attributeValue(**"resultType"**);  *//取出文本内容 组成map中value的部分* String queryString = selectElement.getText();  *//创建Key* String key = namespace+**"."**+id;  *//创建Value* Mapper mapper = **new** Mapper();  mapper.setQueryString(queryString);  mapper.setResultType(resultType);  *//把key和value存入mappers中* mappers.put(key,mapper);  }  **return** mappers;  }**catch**(Exception e){  **throw new** RuntimeException(e);  }**finally**{  in.close();  }  }   */\*\*  \* 根据传入的参数，得到dao中所有被select注解标注的方法。  \* 根据方法名称和类名，以及方法上注解value属性的值，组成Mapper的必要信息  \** ***@param daoClassPath*** *\** ***@return*** *\*/  /\*private static Map<String,Mapper> loadMapperAnnotation(String daoClassPath)throws Exception{  //定义返回值对象  Map<String,Mapper> mappers = new HashMap<String, Mapper>();   //1.得到dao接口的字节码对象  Class daoClass = Class.forName(daoClassPath);  //2.得到dao接口中的方法数组  Method[] methods = daoClass.getMethods();  //3.遍历Method数组  for(Method method : methods){  //取出每一个方法，判断是否有select注解  boolean isAnnotated = method.isAnnotationPresent(Select.class);  if(isAnnotated){  //创建Mapper对象  Mapper mapper = new Mapper();  //取出注解的value属性值  Select selectAnno = method.getAnnotation(Select.class);  String queryString = selectAnno.value();  mapper.setQueryString(queryString);  //获取当前方法的返回值，还要求必须带有泛型信息  Type type = method.getGenericReturnType();//List<User>  //判断type是不是参数化的类型  if(type instanceof ParameterizedType){  //强转  ParameterizedType ptype = (ParameterizedType)type;  //得到参数化类型中的实际类型参数  Type[] types = ptype.getActualTypeArguments();  //取出第一个  Class domainClass = (Class)types[0];  //获取domainClass的类名  String resultType = domainClass.getName();  //给Mapper赋值  mapper.setResultType(resultType);  }  //组装key的信息  //获取方法的名称  String methodName = method.getName();  String className = method.getDeclaringClass().getName();  String key = className+"."+methodName;  //给map赋值  mappers.put(key,mapper);  }  }  return mappers;  }\*/* } |
| *使用的是XML*  *User(id=41, username=Jack, birthday=2018-02-27 17:47:08.0, sex=男, address=北京)* |

### 3.6实现基于注解查询所有操作

|  |
| --- |
| @Retention(RetentionPolicy.***RUNTIME***) @Target(ElementType.***METHOD***) **public** @**interface** Select {  */\*\*  \* 配置sql语句  \** ***@return*** *\*/* String value(); } |

|  |
| --- |
| */\*\*  \* 用户的持久层接口  \*/* **public interface** IUserDao {  */\*\*  \* 查询所有操作  \** ***@return*** *\*/* @Select(**"select \* from user"**)  List<User> findAll(); } |

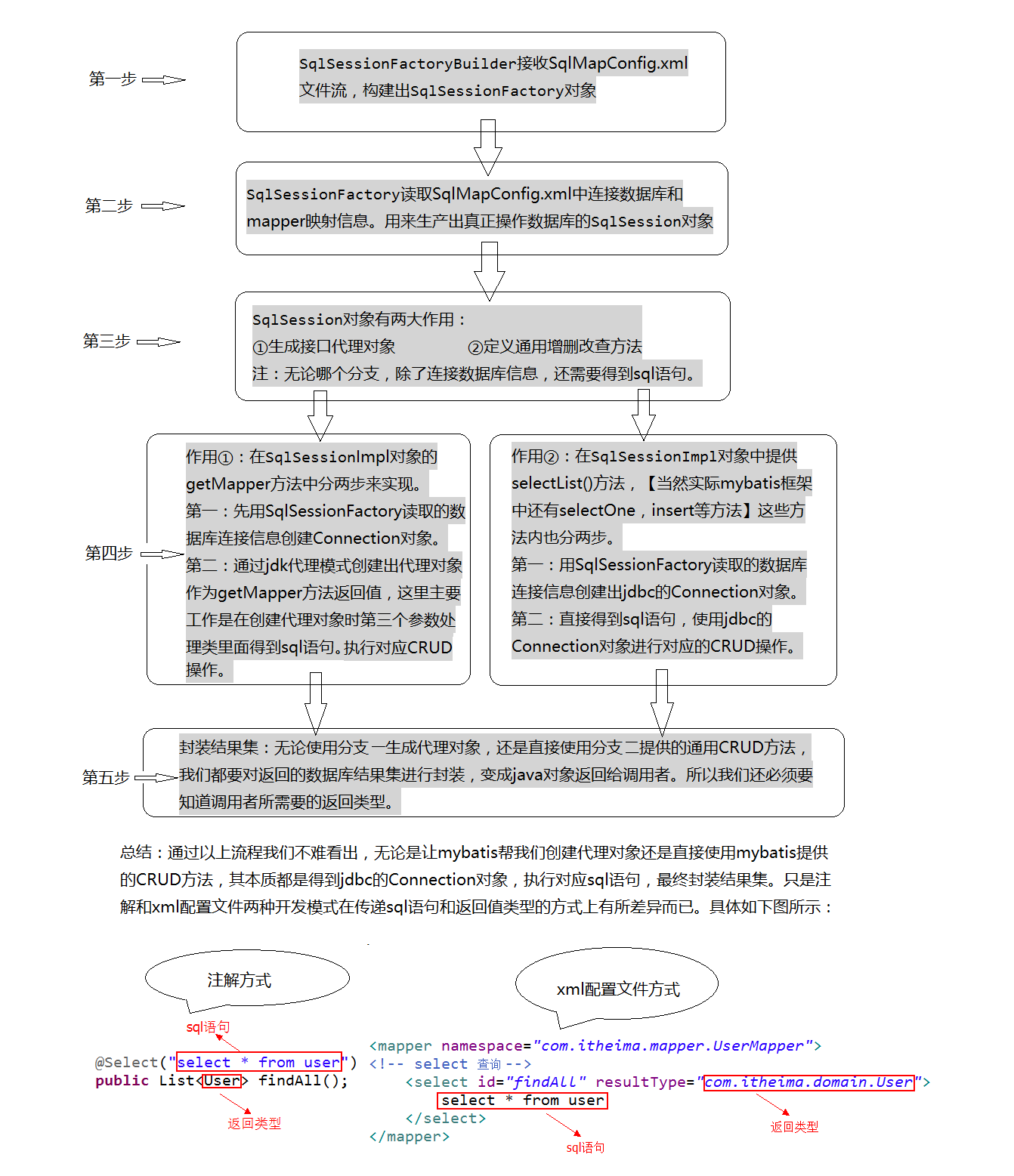
|  |
| --- |
| *<!--指定映射配置文件的位置，映射配置文件指的是每个dao独立的配置文件 -->* <**mappers**>  *<!--<mapper resource="com/byf/dao/IUserDao.xml"></mapper>-->* <**mapper class="com.byf.dao.IUserDao"**></**mapper**> </**mappers**> |

|  |
| --- |
| **public class** XMLConfigBuilder {     */\*\*  \* 解析主配置文件，把里面的内容填充到DefaultSqlSession所需要的地方  \* 使用的技术：  \* dom4j+xpath  \*/* **public static** Configuration loadConfiguration(InputStream config){  **try**{  *//定义封装连接信息的配置对象（mybatis的配置对象）* Configuration cfg = **new** Configuration();   *//1.获取SAXReader对象* SAXReader reader = **new** SAXReader();  *//2.根据字节输入流获取Document对象* Document document = reader.read(config);  *//3.获取根节点* Element root = document.getRootElement();  *//4.使用xpath中选择指定节点的方式，获取所有property节点* List<Element> propertyElements = root.selectNodes(**"//property"**);  *//5.遍历节点* **for**(Element propertyElement : propertyElements){  *//判断节点是连接数据库的哪部分信息  //取出name属性的值* String name = propertyElement.attributeValue(**"name"**);  **if**(**"driver"**.equals(name)){  *//表示驱动  //获取property标签value属性的值* String driver = propertyElement.attributeValue(**"value"**);  cfg.setDriver(driver);  }  **if**(**"url"**.equals(name)){  *//表示连接字符串  //获取property标签value属性的值* String url = propertyElement.attributeValue(**"value"**);  cfg.setUrl(url);  }  **if**(**"username"**.equals(name)){  *//表示用户名  //获取property标签value属性的值* String username = propertyElement.attributeValue(**"value"**);  cfg.setUsername(username);  }  **if**(**"password"**.equals(name)){  *//表示密码  //获取property标签value属性的值* String password = propertyElement.attributeValue(**"value"**);  cfg.setPassword(password);  }  }  *//取出mappers中的所有mapper标签，判断他们使用了resource还是class属性* List<Element> mapperElements = root.selectNodes(**"//mappers/mapper"**);  *//遍历集合* **for**(Element mapperElement : mapperElements){  *//判断mapperElement使用的是哪个属性* Attribute attribute = mapperElement.attribute(**"resource"**);  **if**(attribute != **null**){  System.***out***.println(**"使用的是XML"**);  *//表示有resource属性，用的是XML  //取出属性的值* String mapperPath = attribute.getValue();*//获取属性的值"com/itheima/dao/IUserDao.xml"  //把映射配置文件的内容获取出来，封装成一个map* Map<String,Mapper> mappers = *loadMapperConfiguration*(mapperPath);  *//给configuration中的mappers赋值* cfg.setMappers(mappers);  }**else**{  System.***out***.println(**"使用的是注解"**);  *//表示没有resource属性，用的是注解  //获取class属性的值* String daoClassPath = mapperElement.attributeValue(**"class"**);  *//根据daoClassPath获取封装的必要信息* Map<String,Mapper> mappers = *loadMapperAnnotation*(daoClassPath);  *//给configuration中的mappers赋值* cfg.setMappers(mappers);  }  }  *//返回Configuration* **return** cfg;  }**catch**(Exception e){  **throw new** RuntimeException(e);  }**finally**{  **try** {  config.close();  }**catch**(Exception e){  e.printStackTrace();  }  }   }   */\*\*  \* 根据传入的参数，解析XML，并且封装到Map中  \** ***@param mapperPath*** *映射配置文件的位置  \** ***@return*** *map中包含了获取的唯一标识（key是由dao的全限定类名和方法名组成）  \* 以及执行所需的必要信息（value是一个Mapper对象，里面存放的是执行的SQL语句和要封装的实体类全限定类名）  \*/* **private static** Map<String,Mapper> loadMapperConfiguration(String mapperPath)**throws** IOException {  InputStream in = **null**;  **try**{  *//定义返回值对象* Map<String,Mapper> mappers = **new** HashMap<String,Mapper>();  *//1.根据路径获取字节输入流* in = Resources.*getResourceAsStream*(mapperPath);  *//2.根据字节输入流获取Document对象* SAXReader reader = **new** SAXReader();  Document document = reader.read(in);  *//3.获取根节点* Element root = document.getRootElement();  *//4.获取根节点的namespace属性取值* String namespace = root.attributeValue(**"namespace"**);*//是组成map中key的部分  //5.获取所有的select节点* List<Element> selectElements = root.selectNodes(**"//select"**);  *//6.遍历select节点集合* **for**(Element selectElement : selectElements){  *//取出id属性的值 组成map中key的部分* String id = selectElement.attributeValue(**"id"**);  *//取出resultType属性的值 组成map中value的部分* String resultType = selectElement.attributeValue(**"resultType"**);  *//取出文本内容 组成map中value的部分* String queryString = selectElement.getText();  *//创建Key* String key = namespace+**"."**+id;  *//创建Value* Mapper mapper = **new** Mapper();  mapper.setQueryString(queryString);  mapper.setResultType(resultType);  *//把key和value存入mappers中* mappers.put(key,mapper);  }  **return** mappers;  }**catch**(Exception e){  **throw new** RuntimeException(e);  }**finally**{  in.close();  }  }   */\*\*  \* 根据传入的参数，得到dao中所有被select注解标注的方法。  \* 根据方法名称和类名，以及方法上注解value属性的值，组成Mapper的必要信息  \** ***@param daoClassPath*** *\** ***@return*** *\*/* **private static** Map<String,Mapper> loadMapperAnnotation(String daoClassPath)**throws** Exception{  *//定义返回值对象* Map<String,Mapper> mappers = **new** HashMap<String, Mapper>();   *//1.得到dao接口的字节码对象* Class daoClass = Class.*forName*(daoClassPath);  *//2.得到dao接口中的方法数组* Method[] methods = daoClass.getMethods();  *//3.遍历Method数组* **for**(Method method : methods){  *//取出每一个方法，判断是否有select注解* **boolean** isAnnotated = method.isAnnotationPresent(Select.**class**);  **if**(isAnnotated){  *//创建Mapper对象* Mapper mapper = **new** Mapper();  *//取出注解的value属性值* Select selectAnno = method.getAnnotation(Select.**class**);  String queryString = selectAnno.value();  mapper.setQueryString(queryString);  *//获取当前方法的返回值，还要求必须带有泛型信息* Type type = method.getGenericReturnType();*//List<User>  //判断type是不是参数化的类型* **if**(type **instanceof** ParameterizedType){  *//强转* ParameterizedType ptype = (ParameterizedType)type;  *//得到参数化类型中的实际类型参数* Type[] types = ptype.getActualTypeArguments();  *//取出第一个* Class domainClass = (Class)types[0];  *//获取domainClass的类名* String resultType = domainClass.getName();  *//给Mapper赋值* mapper.setResultType(resultType);  }  *//组装key的信息  //获取方法的名称* String methodName = method.getName();  String className = method.getDeclaringClass().getName();  String key = className+**"."**+methodName;  *//给map赋值* mappers.put(key,mapper);  }  }  **return** mappers;  } } |
|  |

## 第4节Mybatis框架回顾

1. Mybatis的自定义再分析和环境搭建+完善基于注解的mybatis
2. Mybatis的CURD（基于代理Dao的方式）
3. Mybatis中的参数深入及结果集的深入
4. Mybatis中基于传统dao的方式（编写dao的实现类）--了解
5. Mybatis中的配置（主配置文件：SqlMapConfig.xml）
   1. Properties标签
   2. typeAliases标签
   3. Mappers标签

### 4.1自定义mybatis再分析



### 4.2Mybatis的CRUD

（1）Maven依赖

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**project xmlns="http://maven.apache.org/POM/4.0.0"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>  <**modelVersion**>4.0.0</**modelVersion**>   <**groupId**>com.byf</**groupId**>  <**artifactId**>mybatis01</**artifactId**>  <**version**>1.0-SNAPSHOT</**version**>  <**packaging**>jar</**packaging**>  <**dependencies**>  <**dependency**>  <**groupId**>org.mybatis</**groupId**>  <**artifactId**>mybatis</**artifactId**>  <**version**>3.4.6</**version**>  </**dependency**>  <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  <**version**>5.1.6</**version**>  </**dependency**>  <**dependency**>  <**groupId**>log4j</**groupId**>  <**artifactId**>log4j</**artifactId**>  <**version**>1.2.17</**version**>  </**dependency**>  <**dependency**>  <**groupId**>junit</**groupId**>  <**artifactId**>junit</**artifactId**>  <**version**>4.12</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.projectlombok</**groupId**>  <**artifactId**>lombok</**artifactId**>  <**version**>1.16.20</**version**>  </**dependency**>  </**dependencies**>   </**project**> |

（2）G:\Mybatis\mybatis\_coding\mybatis\mybatis02\src\main\resources\log4j.properties

|  |
| --- |
| # Set root category priority to INFO and its only appender to CONSOLE.  #log4j.rootCategory=INFO, CONSOLE debug info warn error fatal  log4j.rootCategory=debug, CONSOLE, LOGFILE  # Set the enterprise logger category to FATAL and its only appender to CONSOLE.  log4j.logger.org.apache.axis.enterprise=FATAL, CONSOLE  # CONSOLE is set to be a ConsoleAppender using a PatternLayout.  log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender  log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout  log4j.appender.CONSOLE.layout.ConversionPattern=%d{mm:ss,SSS} [%5.15t] %-5p %30.30c %x - %m\n  # LOGFILE is set to be a File appender using a PatternLayout.  log4j.appender.LOGFILE=org.apache.log4j.FileAppender  log4j.appender.LOGFILE.File=d:\axis.log  log4j.appender.LOGFILE.Append=true  log4j.appender.LOGFILE.layout=org.apache.log4j.PatternLayout  log4j.appender.LOGFILE.layout.ConversionPattern=%d{ISO8601} %-6r [%15.15t] %-5p %30.30c %x - %m\n |

（3）G:\Mybatis\mybatis\_coding\mybatis\mybatis02\src\main\resources\SqlMapConfig.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**>  *<!-- 配置环境 -->* <**environments default="mysql"**>  *<!-- 配置mysql环境 -->* <**environment id="mysql"**>  *<!-- 配置事务类型 -->* <**transactionManager type="JDBC"**></**transactionManager**>  *<!-- 配置数据源（连接池） -->* <**dataSource type="POOLED"**>  <**property name="driver" value="com.mysql.jdbc.Driver"**/>  <**property name="url" value="jdbc:mysql://localhost:3306/eesy\_mybatis"**/>  <**property name="username" value="root"**/>  <**property name="password" value="1234"**/>  </**dataSource**>  </**environment**>  </**environments**>   *<!--指定映射配置文件的位置，映射配置文件指的是每个dao独立的配置文件 -->* <**mappers**>  <**mapper resource="com/byf/dao/IUserDao.xml"**></**mapper**>  </**mappers**> </**configuration**> |

（4）com.byf.dao.IUserDao

|  |
| --- |
| */\*\*  \* 用户的持久层接口  \*/* **public interface** IUserDao {  */\*\*  \* 查询所有操作  \** ***@return*** *\*/* List<User> findAll(); } |

com.byf.domain.User

|  |
| --- |
| @ToString **public class** User **implements** Serializable {  **private** Integer **id**;  **private** String **username**;  **private** Date **birthday**;  **private** String **sex**;  **private** String **address**;   **public** Integer getId() {  **return id**;  }   **public void** setId(Integer id) {  **this**.**id** = id;  }   **public** String getUsername() {  **return username**;  }   **public void** setUsername(String username) {  **this**.**username** = username;  }   **public** Date getBirthday() {  **return** (Date) **birthday**.clone();  }   **public void** setBirthday(Date birthday) {  **this**.**birthday** = birthday;  }   **public** String getSex() {  **return sex**;  }   **public void** setSex(String sex) {  **this**.**sex** = sex;  }   **public** String getAddress() {  **return address**;  }   **public void** setAddress(String address) {  **this**.**address** = address;  } } |

（5）G:\Mybatis\mybatis\_coding\mybatis\mybatis02\src\main\resources\com\byf\dao\IUserDao.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="com.byf.dao.IUserDao"**>  *<!-- 配置查询所有 -->* <**select id="findAll" resultType="com.byf.domain.User"**>  select *\** from user  </**select**>  </**mapper**> |

（6）com.byf.test.MybatisTest

|  |
| --- |
| **public class** MybatisTest {  @Test  **public void** testConfig() **throws** IOException {  */\*\*  \* 1.读取配置文件  \* 1）绝对路径：d:/xxx/xxx.xml 不可用  \* 2）相对路径：src/java/main/xxx.xml 不可用  \* 3）使用类加载器，他只能兑取类路径的配置文件  \* 4）使用ServletContext对象的getRealPath()  \*/* InputStream in = Resources.*getResourceAsStream*(**"SqlMapConfig.xml"**);   */\*\*  \* 2.创建SqlSessionFactory工厂  \* 1）创建工厂mybatis使用了构建者模式  \* 优势：把对象创建的细节。因此，使用者直接调用对象  \*/* SqlSessionFactoryBuilder builder = **new** SqlSessionFactoryBuilder();  SqlSessionFactory factory = builder.build(in);   */\*\*  \* 3.使用工厂生产生产SqlSession对象  \* 1）生成SqlSession使用了工厂模式  \* 优势：解耦（降低类之间的依赖关系）  \*/* SqlSession session = factory.openSession();   */\*\*  \* 4.使用SqlSession创建Dao接口的代理对象  \* 1）创建Dao接口实现类使用了代理模式  \* 优势：不修改源码的基础上，增强方法  \*/* IUserDao userDao = session.getMapper(IUserDao.**class**);  *//5.使用代理对象执行方法* List<User> users = userDao.findAll();  **for** (User user:users){  System.***out***.println(user);  }  *//6.释放资源* session.close();  } } |
| **28:33,702 [ main] DEBUG source.pooled.PooledDataSource - PooledDataSource forcefully closed/removed all connections.**  **28:33,872 [ main] DEBUG ansaction.jdbc.JdbcTransaction - Opening JDBC Connection**  **28:34,272 [ main] DEBUG source.pooled.PooledDataSource - Created connection 1629911510.**  **28:34,272 [ main] DEBUG ansaction.jdbc.JdbcTransaction - Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@612679d6]**  **28:34,275 [ main] DEBUG com.byf.dao.IUserDao.findAll - ==> Preparing: select \* from user**  **28:34,335 [ main] DEBUG com.byf.dao.IUserDao.findAll - ==> Parameters:**  **28:34,416 [ main] DEBUG com.byf.dao.IUserDao.findAll - <== Total: 6**  **User(id=41, username=Jack, birthday=Tue Feb 27 17:47:08 CST 2018, sex=男, address=北京)**  **......**  **28:34,417 [ main] DEBUG ansaction.jdbc.JdbcTransaction - Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@612679d6]**  **28:34,418 [ main] DEBUG ansaction.jdbc.JdbcTransaction - Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@612679d6]**  **28:34,418 [ main] DEBUG source.pooled.PooledDataSource - Returned connection 1629911510 to pool.** |

### 4.3Mybatis的保存操作

|  |
| --- |
| */\*\*  \* 用户的持久层接口  \*/* **public interface** IUserDao {  */\*\*  \* 查询所有操作  \** ***@return*** *\*/* List<User> findAll();   */\*\*  \* 保存用户  \** ***@param user*** *\*/* **void** saveUser(User user); } |
| *<?***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="com.byf.dao.IUserDao"**>  *<!-- 配置查询所有 -->* <**select id="findAll" resultType="com.byf.domain.User"**>  select *\** from user  </**select**>  <**insert id="saveUser" parameterType="com.byf.domain.User"**>  insert into user(username,address,sex,birthday)  values(#{username},#{address},#{sex},#{birthday});  </**insert**> </**mapper**> |
| **public class** MybatisTest {   **private** InputStream **in**;  **private** SqlSession **sqlSession**;  **private** IUserDao **userDao**;    @Before  **public void** init() **throws** IOException {  */\*\*  \* 1.读取配置文件  \* 1）绝对路径：d:/xxx/xxx.xml 不可用  \* 2）相对路径：src/java/main/xxx.xml 不可用  \* 3）使用类加载器，他只能兑取类路径的配置文件  \* 4）使用ServletContext对象的getRealPath()  \*/* **in** = Resources.*getResourceAsStream*(**"SqlMapConfig.xml"**);  */\*\*  \* 2.创建SqlSessionFactory工厂  \* 1）创建工厂mybatis使用了构建者模式  \* 优势：把对象创建的细节。因此，使用者直接调用对象  \*/* SqlSessionFactory factory = **new** SqlSessionFactoryBuilder().build(**in**);  */\*\*  \* 3.使用工厂生产生产SqlSession对象  \* 1）生成SqlSession使用了工厂模式  \* 优势：解耦（降低类之间的依赖关系）  \*/* **sqlSession** = factory.openSession();  */\*\*  \* 4.使用SqlSession创建Dao接口的代理对象  \* 1）创建Dao接口实现类使用了代理模式  \* 优势：不修改源码的基础上，增强方法  \*/* **userDao** = **sqlSession**.getMapper(IUserDao.**class**);  }   */\*\*  \* 测试保存用户  \*/* @Test  **public void** testSave(){  User user = **new** User();  user.setUsername(**"Jack"**);  user.setAddress(**"Nanjing"**);  user.setSex(**"男"**);  user.setBirthday(**new** Date());  *//5.使用代理对象执行方法* **userDao**.saveUser(user);  *//6.提交保存操作* **sqlSession**.commit();  }   @After  **public void** destroy() **throws** IOException {  **sqlSession**.close();  **in**.close();  } } |

### 4.4Mybatis的修改和删除操作

|  |
| --- |
| */\*\*  \* 更新用户  \** ***@param user*** *\*/* **void** updateUser(User user);  */\*\*  \* 删除用户  \** ***@param userId*** *\*/* **void** deleteUser(Integer userId); |
| <**update id="updateUser" parameterType="com.byf.domain.User"**>  update user  set username=#{username},address=#{address},sex=#{sex},birthday=#{birthday}  where id=#{id} </**update**> <**delete id="deleteUser" parameterType="java.lang.Integer"**>  delete from user  where id=#{uid} </**delete**> |
| */\*\*  \* 测试保存用户  \*/* @Test **public void** testUpdate(){  User user = **new** User();  user.setId(50);  user.setUsername(**"Jack"**);  user.setAddress(**"Shanghai"**);  user.setSex(**"男"**);  user.setBirthday(**new** Date());  *//5.使用代理对象执行方法* **userDao**.updateUser(user);  *//6.提交保存操作* **sqlSession**.commit(); }  */\*\*  \* 测试删除用户  \*/* @Test **public void** testDelete(){  *//5.使用代理对象执行方法* **userDao**.deleteUser(50);  *//6.提交保存操作* **sqlSession**.commit(); } |

### 4.5Mybatis查询一个和模糊查询

|  |
| --- |
| User findById(Integer userId);  List<User> findByName(String userName); |
| <**select id="findById" parameterType="INT" resultType="com.byf.domain.User"**>  select *\** from user  where id=#{uid} </**select**> <**select id="findByName" parameterType="string" resultType="com.byf.domain.User"**>  select *\** from user  where username like #{userName} </**select**> |
| */\*\*  \* 测试查询一个用户  \*/* @Test **public void** testFindById(){  *//5.使用代理对象执行方法* User user = **userDao**.findById(48);  System.***out***.println(user); }  */\*\*  \* 测试模糊查询  \*/* @Test **public void** testFindByName(){  *//5.使用代理对象执行方法* List<User> users = **userDao**.findByName(**"J%"**);  **for** (User user:users){  System.***out***.println(user);  } } |

### 4.6查询返回一行一列和占位符分析

回顾模糊查询

|  |
| --- |
| <**select id="findByName" parameterType="string" resultType="com.byf.domain.User"**>  *<!--select \* from user  where username like #{userName}  Preparing: select \* from user where username like '%J%'  Statement 拼接  -->* select *\** from user  where username like '%${value}%'  *<!-- Preparing: select \* from user where username like ?  PrepareStatement 占位符实现 -->* </**select**> |

|  |
| --- |
| **int** findTotal(); |
| <**select id="findTotal" resultType="int"**>  select count(\*) from user </**select**> |
| */\*\*  \* 测试总记录条数  \*/* @Test **public void** testFindTotal(){  *//5.使用代理对象执行方法* **int** couont = **userDao**.findTotal();  System.***out***.println(couont); } |

### 4.7保存操作的细节

|  |
| --- |
| insert into user(username,address,sex,birthday)  VALUES('testid','Beijing','男',NOW());  select LAST\_INSERT\_ID(); |
| 51 |

|  |
| --- |
| <**insert id="saveUser" parameterType="com.byf.domain.User"**> *<!-- 配置插入操作后，获取插入数据的id -->* <**selectKey keyProperty="id" keyColumn="id" resultType="int" order="AFTER"**>  select LAST\_INSERT\_ID();  </**selectKey**>  insert into user(username,address,sex,birthday)  values(#{username},#{address},#{sex},#{birthday}); </**insert**> |
| */\*\*  \* 测试保存用户  \*/* @Test **public void** testSave1(){  User user = **new** User();  user.setUsername(**"Jack"**);  user.setAddress(**"Nanjing"**);  user.setSex(**"男"**);  user.setBirthday(**new** Date());  System.***out***.println(**"Before"**);  System.***out***.println(user);  *//5.使用代理对象执行方法* **userDao**.saveUser(user);  System.***out***.println(**"After"**);  System.***out***.println(user);  *//6.提交保存操作  //sqlSession.commit();* } |
| *Before*  *User(id=null, username=Jack, birthday=Mon Aug 12 22:19:32 CST 2019, sex=男, address=Nanjing)*  *.....*  *After*  *User(id=53, username=Jack, birthday=Mon Aug 12 22:19:32 CST 2019, sex=男, address=Nanjing)* |

### 4.8Mybatis的参数深入

·简单类型

·传递pojo对象

OGNL表达式：Object Graphic Navigation Language

对象 图 导航 语言

它是通过对象的取值方法来获取数据。在写法上把get给省略了。

比如：我们获取用户的名称

类中的写法：user.getUserName();

OGNL表达式写法：user.username

Mybatis中为什么直接写username，而不用user.呢：

因为在parameterType中已经提供了属性所属的类，所以此时不需要写类名

·传递pojo包装对象

Mapped Statements collection already contains value for com.byf.dao.IUserDao.findByName

|  |
| --- |
| <**select id="findByQueryVo" parameterType="com.byf.domain.QueryVO" resultType="com.byf.domain.User"**>  *<!--select \* from user  where username like #{userName}  Preparing: select \* from user where username like '%J%'  Statement 拼接  -->* select *\** from user  where username like #{user.username}  *<!-- Preparing: select \* from user where username like ?  PrepareStatement 占位符实现 -->* </**select**> |

java.lang.NullPointerException

at com.byf.domain.User.getBirthday(User.java:33)

at com.byf.domain.User.toString(User.java:8)

at java.lang.String.valueOf(String.java:2994)

at java.io.PrintStream.println(PrintStream.java:821)

at com.byf.test.MybatisTest.testFindByQueryVo(MybatisTest.java:155)

at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)

at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)

at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)

at java.lang.reflect.Method.invoke(Method.java:498)

at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:50)

at org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.java:12)

at org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod.java:47)

at org.junit.internal.runners.statements.InvokeMethod.evaluate(InvokeMethod.java:17)

at org.junit.internal.runners.statements.RunBefores.evaluate(RunBefores.java:26)

at org.junit.internal.runners.statements.RunAfters.evaluate(RunAfters.java:27)

at org.junit.runners.ParentRunner.runLeaf(ParentRunner.java:325)

at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:78)

at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:57)

at org.junit.runners.ParentRunner$3.run(ParentRunner.java:290)

at org.junit.runners.ParentRunner$1.schedule(ParentRunner.java:71)

at org.junit.runners.ParentRunner.runChildren(ParentRunner.java:288)

at org.junit.runners.ParentRunner.access$000(ParentRunner.java:58)

at org.junit.runners.ParentRunner$2.evaluate(ParentRunner.java:268)

at org.junit.runners.ParentRunner.run(ParentRunner.java:363)

at org.junit.runners.Suite.runChild(Suite.java:128)

at org.junit.runners.Suite.runChild(Suite.java:27)

at org.junit.runners.ParentRunner$3.run(ParentRunner.java:290)

at org.junit.runners.ParentRunner$1.schedule(ParentRunner.java:71)

at org.junit.runners.ParentRunner.runChildren(ParentRunner.java:288)

at org.junit.runners.ParentRunner.access$000(ParentRunner.java:58)

at org.junit.runners.ParentRunner$2.evaluate(ParentRunner.java:268)

at org.junit.runners.ParentRunner.run(ParentRunner.java:363)

at org.junit.runner.JUnitCore.run(JUnitCore.java:137)

at com.intellij.junit4.JUnit4IdeaTestRunner.startRunnerWithArgs(JUnit4IdeaTestRunner.java:68)

at com.intellij.rt.execution.junit.IdeaTestRunner$Repeater.startRunnerWithArgs(IdeaTestRunner.java:47)

at com.intellij.rt.execution.junit.JUnitStarter.prepareStreamsAndStart(JUnitStarter.java:242)

at com.intellij.rt.execution.junit.JUnitStarter.main(JUnitStarter.java:70)

### 4.9Mybatis的返回值深入

|  |
| --- |
| **public class** User **implements** Serializable {   **private** Integer **userId**;  **private** String **userName**;  **private** String **userAddress**;  **private** String **userSex**;  **private** Date **userBirthday**;   **public** Integer getUserId() {  **return userId**;  }   **public void** setUserId(Integer userId) {  **this**.**userId** = userId;  }   **public** String getUserName() {  **return userName**;  }   **public void** setUserName(String userName) {  **this**.**userName** = userName;  }   **public** String getUserAddress() {  **return userAddress**;  }   **public void** setUserAddress(String userAddress) {  **this**.**userAddress** = userAddress;  }   **public** String getUserSex() {  **return userSex**;  }   **public void** setUserSex(String userSex) {  **this**.**userSex** = userSex;  }   **public** Date getUserBirthday() {  **return userBirthday**;  }   **public void** setUserBirthday(Date userBirthday) {  **this**.**userBirthday** = userBirthday;  }   @Override  **public** String toString() {  **return "User{"** +  **"userId="** + **userId** +  **", userName='"** + **userName** + **'\''** +  **", userAddress='"** + **userAddress** + **'\''** +  **", userSex='"** + **userSex** + **'\''** +  **", userBirthday="** + **userBirthday** +  **'}'**;  } } |
| *<?***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="com.byf.dao.IUserDao"**>  *<!-- 配置查询结果的列名和实体类的属性名对象 -->* <**resultMap id="userMap" type="com.byf.domain.User"**>  *<!-- 主键字段的对应 -->* <**id property="userId" column="id"**></**id**>  *<!-- 其他列的对应 -->* <**result property="userName" column="username"**></**result**>  <**result property="userAddress" column="address"**></**result**>  <**result property="userSex" column="sex"**></**result**>  <**result property="userBirthday" column="birthday"**></**result**>   </**resultMap**>   *<!-- 配置查询所有 -->* <**select id="findAll" resultMap="userMap"**>  *<!-- select id as userId,username as userName, sex as userSex, birthday as userBirthday,address as userAddress from user -->* select *\** from user  </**select**>  <**insert id="saveUser" parameterType="com.byf.domain.User"**>  *<!-- 配置插入操作后，获取插入数据的id -->* <**selectKey keyProperty="userId" keyColumn="id" resultType="int" order="AFTER"**>  select LAST\_INSERT\_ID();  </**selectKey**>  insert into user(username,address,sex,birthday)  values(#{userName},#{userAddress},#{userSex},#{userBirthday});  </**insert**>  <**update id="updateUser" parameterType="com.byf.domain.User"**>  update user  set username=#{userName},address=#{userAddress},sex=#{userSex},birthday=#{userBirthday}  where id=#{userId}  </**update**>  <**delete id="deleteUser" parameterType="java.lang.Integer"**>  delete from user  where id=#{uid}  </**delete**>   <**select id="findById" parameterType="INT" resultMap="userMap"**>  select *\** from user  where id=#{uid}  </**select**>   <**select id="findByName" parameterType="string" resultMap="userMap"**>  *<!--select \* from user  where username like #{userName}  Preparing: select \* from user where username like '%J%'  Statement 拼接  -->* select *\** from user  where username like '%${value}%'  *<!-- Preparing: select \* from user where username like ?  PrepareStatement 占位符实现 -->* </**select**>  <**select id="findTotal" resultType="int"**>  select *count*(*\**) from user  </**select**>   <**select id="findByQueryVo" parameterType="com.byf.domain.QueryVO" resultMap="userMap"**>  *<!--select \* from user  where username like #{userName}  Preparing: select \* from user where username like '%J%'  Statement 拼接  -->* select *\** from user  where username like #{user.username}  *<!-- Preparing: select \* from user where username like ?  PrepareStatement 占位符实现 -->* </**select**> </**mapper**> |
| **public class** MybatisTest {   **private** InputStream **in**;  **private** SqlSession **sqlSession**;  **private** IUserDao **userDao**;    @Before  **public void** init() **throws** IOException {  */\*\*  \* 1.读取配置文件  \* 1）绝对路径：d:/xxx/xxx.xml 不可用  \* 2）相对路径：src/java/main/xxx.xml 不可用  \* 3）使用类加载器，他只能兑取类路径的配置文件  \* 4）使用ServletContext对象的getRealPath()  \*/* **in** = Resources.*getResourceAsStream*(**"SqlMapConfig.xml"**);  */\*\*  \* 2.创建SqlSessionFactory工厂  \* 1）创建工厂mybatis使用了构建者模式  \* 优势：把对象创建的细节。因此，使用者直接调用对象  \*/* SqlSessionFactory factory = **new** SqlSessionFactoryBuilder().build(**in**);  */\*\*  \* 3.使用工厂生产生产SqlSession对象  \* 1）生成SqlSession使用了工厂模式  \* 优势：解耦（降低类之间的依赖关系）  \*/* **sqlSession** = factory.openSession();  */\*\*  \* 4.使用SqlSession创建Dao接口的代理对象  \* 1）创建Dao接口实现类使用了代理模式  \* 优势：不修改源码的基础上，增强方法  \*/* **userDao** = **sqlSession**.getMapper(IUserDao.**class**);  }  @Test  **public void** testConfig() **throws** IOException {  **sqlSession**.getMapper(IUserDao.**class**);  *//5.使用代理对象执行方法* List<User> users = **userDao**.findAll();  **for** (User user:users){  System.***out***.println(user);  }  }   */\*\*  \* 测试保存用户  \*/* @Test  **public void** testSave(){  User user = **new** User();  user.setUserName(**"Jack1"**);  user.setUserAddress(**"Nanjing1"**);  user.setUserSex(**"男"**);  user.setUserBirthday(**new** Date());  *//5.使用代理对象执行方法* **userDao**.saveUser(user);  *//6.提交保存操作* **sqlSession**.commit();  }   */\*\*  \* 测试保存用户  \*/* @Test  **public void** testUpdate(){  User user = **new** User();  user.setUserId(55);  user.setUserName(**"Jack"**);  user.setUserAddress(**"Shanghai"**);  user.setUserSex(**"男"**);  user.setUserBirthday(**new** Date());  *//5.使用代理对象执行方法* **userDao**.updateUser(user);  *//6.提交保存操作* **sqlSession**.commit();  }   */\*\*  \* 测试删除用户  \*/* @Test  **public void** testDelete(){  *//5.使用代理对象执行方法* **userDao**.deleteUser(50);  *//6.提交保存操作* **sqlSession**.commit();  }   */\*\*  \* 测试查询一个用户  \*/* @Test  **public void** testFindById(){  *//5.使用代理对象执行方法* User user = **userDao**.findById(48);  System.***out***.println(user);  }   */\*\*  \* 测试模糊查询  \*/* @Test  **public void** testFindByName(){  *//5.使用代理对象执行方法* List<User> users = **userDao**.findByName(**"J"**);  **for** (User user:users){  System.***out***.println(user);  }  }   */\*\*  \* 测试总记录条数  \*/* @Test  **public void** testFindTotal(){  *//5.使用代理对象执行方法* **int** couont = **userDao**.findTotal();  System.***out***.println(couont);  }    */\*\*  \* 测试模糊查询  \*/* @Test  **public void** testFindByQueryVo(){  QueryVO queryVO = **new** QueryVO();  User user = **new** User();  user.setUserName(**"J%"**);  queryVO.setUser(user);  *//5.使用代理对象执行方法* List<User> users = **userDao**.findByQueryVo(queryVO);  **for** (User user1:users){  System.***out***.println(user);  }  }    @After  **public void** destroy() **throws** IOException {  **sqlSession**.close();  **in**.close();  }   */\*\*  \* 测试保存用户  \*/* @Test  **public void** testSave1(){  User user = **new** User();  user.setUserName(**"Jack"**);  user.setUserAddress(**"Nanjing"**);  user.setUserSex(**"男"**);  user.setUserBirthday(**new** Date());  System.***out***.println(**"Before"**);  System.***out***.println(user);  *//5.使用代理对象执行方法* **userDao**.saveUser(user);  System.***out***.println(**"After"**);  System.***out***.println(user);  *//6.提交保存操作  //sqlSession.commit();* } } |

## 第5节Mybatis编写dao实现的使用方式

### 5.1Dao实现类的执行过程分析

PrepareStatement对象它的执行方法

·execute：他能执行CURD的任何一种语句，他的返回值是一个boolean类型，表示是否有结果集。有：true，无：false

·executeUpdate：他只能执行CRD语句，查询语句无法执行。他的返回值影响数据库记录的行数；

·executeQuery：他只能执行SELECT语句，无法执行增删改，执行结果分装的结果集ResultSet对象

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |



### 5.2Mybatis中使用代理Dao的执行过程分析



## 第6节Mybatis的连接池及事务

1. Mybatis中的连接池以及事务

·mybatis中的连接池使用及分析

·mybatis事务控制的分析

1. Mybatis基于XML配置的动态SQL语句使用

·mappers配置文件中的几个标签

<if>

<where>

<foreach>

<sql>

1. Mybatis中的多表操作

一对多

一对一

多对一

### 6.1连接池

减少获取连接所消耗的时间

Mybatis中的连接池

·mybatis连接池提供了3中方式的配置：

配置的位置：主配置文件SqlMapConfig.xml中的dataSource标签，type属性就是表示采用何种连接池方式

Type属性的取值

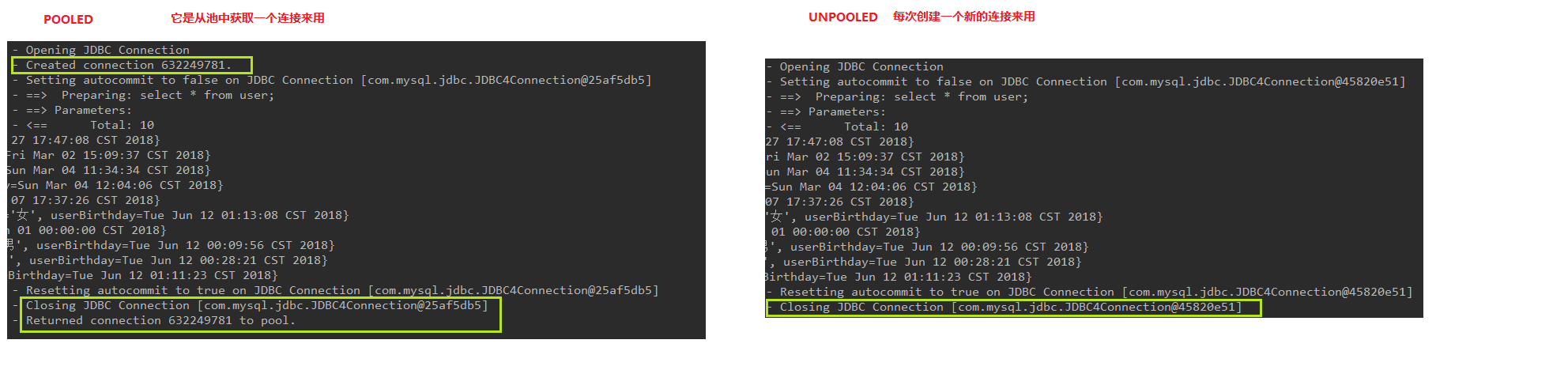
POOLED 采用传统的javax.sql.DataSource规范中的连接池，mybatis中有针对规范的实现

UNPOOLED 采用传统的获取连接的方式，虽然也实现java.sql.DataSource接口，但并没有使用连接池

JNDI 采用服务器提供的JNDI技术实现，来获取DataSource对象，不同的服务器所能拿到

注意：如果不是web或者maven的war工厂，是不能使用的。

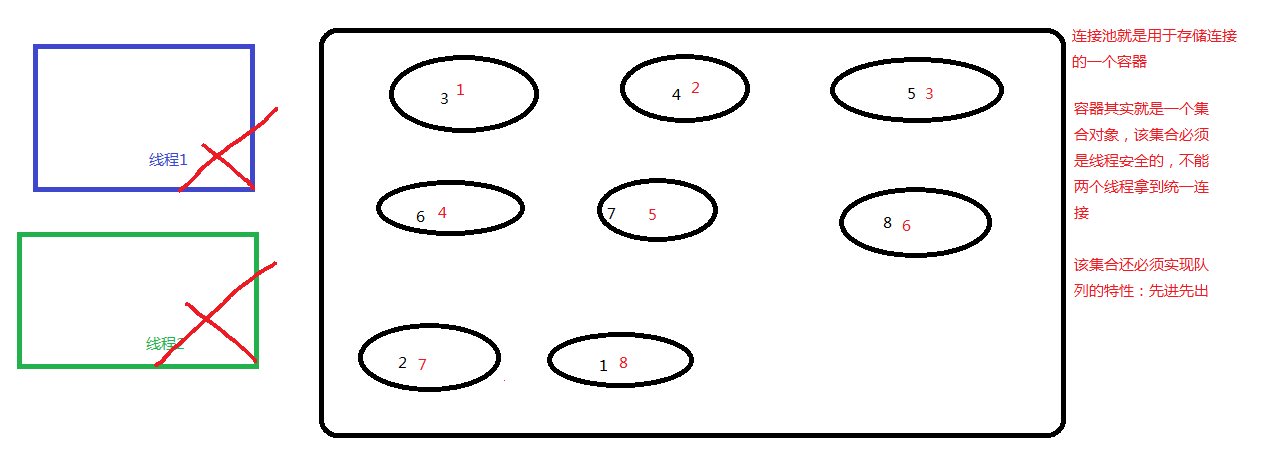
我们课程中使用的是tomcat服务器，采用连接池就是dbcp连接池

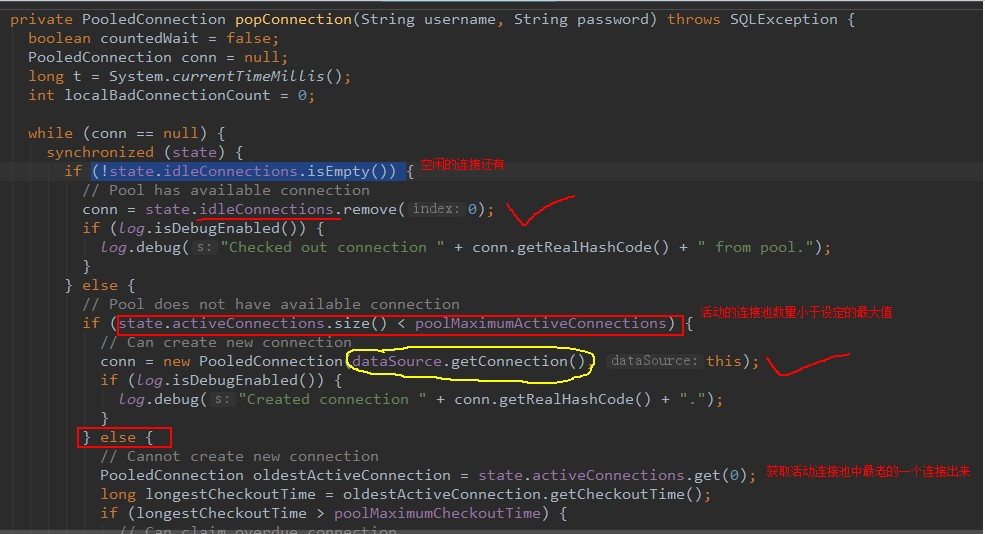


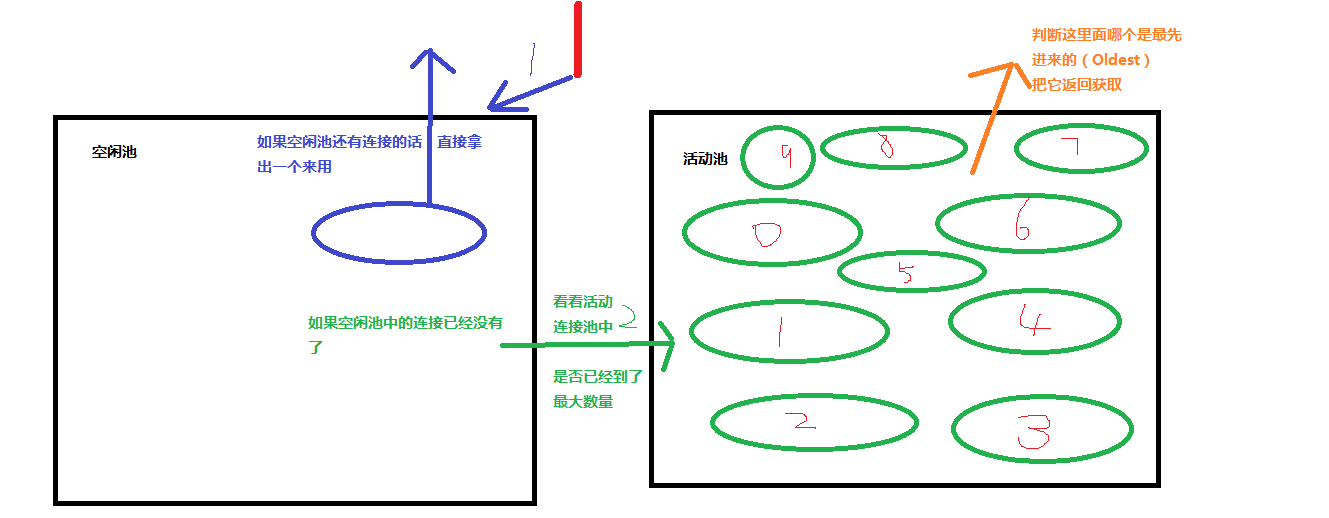
UnPooledDataSource

|  |
| --- |
| **private** Connection doGetConnection(Properties properties) **throws** SQLException {  **this**.initializeDriver();  Connection connection = DriverManager.getConnection(**this**.url, properties);  **this**.configureConnection(connection);  **return** connection; } |

PooledDataSource







### 6.2事务原理和自动提交设置

|  |
| --- |
| **public void** saveUser(User user) {  *// 1.根据factory获取SqlSession对象* SqlSession session = **factory**.openSession(**true**);  *// 2.调用SqlSession中的方法，实现查询列表* session.insert(**"com.byf.dao.IUserDao.saveUser"**, user);  *// session.commit();* session.close(); } |

## 第7节动态SQL

|  |
| --- |
| *<!-- 根据queryVO中的id集合查询用户列表 -->* <**select id="findUserInIds" resultMap="userMap" parameterType="com.byf.domain.QueryVO"**>  *<!-- select id as userId,username as userName, sex as userSex, birthday as userBirthday,address as userAddress from user -->* <**include refid="defaultUser"**></**include**>  <**where**>  *<!-- 入参中的属性要使用Bean的属性，保持大小写一致 -->* <**if test="ids != null and ids.size() > 0"**>  <**foreach collection="ids" open="and id in (" close=")" item="uid" separator=","**>  #{uid}  </**foreach**>  </**if**>  </**where**> </**select**> |
| *<!-- 抽取重复的sql语句 -->* <**sql id="defaultUser"**>  select *\** from user </**sql**> |

## 第8节Mybatis多表操作

### 8.1Mybatis表之间关系分析

表之间的关系有四种：

·一对多

·多对一

·一对一

·多对多

举例：

·用户和订单就是一对多

·订单和用户就是多对一

一个用户可以下多个订单

多个订单属于同一个用户

·人和身份证号就是一对一

一个人只能有一个身份证号

一个身份证号只能属于一个人

·老师和学生之间就是多对多

一个学生可以被多个老师教过

一个老师可以教多个学生

特例：

如果拿出每一个订单，与用户对应一对一

### 8.2一对多查询操作

|  |
| --- |
| DROP TABLE IF EXISTS `account`;  CREATE TABLE `account` (  `ID` int(11) NOT NULL COMMENT '编号',  `UID` int(11) default NULL COMMENT '用户编号',  `MONEY` double default NULL COMMENT '金额',  PRIMARY KEY (`ID`),  KEY `FK\_Reference\_8` (`UID`),  CONSTRAINT `FK\_Reference\_8` FOREIGN KEY (`UID`) REFERENCES `user` (`id`)  ) ENGINE=InnoDB DEFAULT CHARSET=utf8;  insert into `account`(`ID`,`UID`,`MONEY`) values (1,46,1000),(2,45,1000),(3,46,2000);  select a.\*,u.username,u.address from user u,account a  where u.id=a.UID |

|  |
| --- |
| *<?***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="com.byf.dao.IAccountDao"**>   *<!-- 定义封装account和user的resultMap -->* <**resultMap id="accountUserMap" type="account"**>  <**id property="id" column="aid"**></**id**>  <**result property="uid" column="uid"**></**result**>  <**result property="money" column="money"**></**result**>  *<!-- 一对一的关系映射：配置封装user的内容-->* <**association property="user" column="uid" javaType="user"**>  <**id property="id" column="id"**></**id**>  <**result column="username" property="username"**></**result**>  <**result column="address" property="address"**></**result**>  <**result column="sex" property="sex"**></**result**>  <**result column="birthday" property="birthday"**></**result**>  </**association**>  </**resultMap**>   *<!-- 查询所有 -->* <**select id="findAll" resultMap="accountUserMap"**>  select u.*\**,a.id as aid,a.uid,a.money from account a , user u where u.id = a.uid;  </**select**>   *<!--查询所有账户同时包含用户名和地址信息-->* <**select id="findAllAccount" resultType="accountuser"**>  select a.*\**,u.username,u.address from account a , user u where u.id = a.uid;  </**select**>  </**mapper**> |

|  |
| --- |
| **public class** Account **implements** Serializable {   **private** Integer **id**;  **private** Integer **uid**;  **private** Double **money**;   *//从表实体应该包含一个主表实体的对象引用* **private** User **user**;   **public** User getUser() {  **return user**;  }   **public void** setUser(User user) {  **this**.**user** = user;  }   **public** Integer getId() {  **return id**;  }   **public void** setId(Integer id) {  **this**.**id** = id;  }   **public** Integer getUid() {  **return uid**;  }   **public void** setUid(Integer uid) {  **this**.**uid** = uid;  }   **public** Double getMoney() {  **return money**;  }   **public void** setMoney(Double money) {  **this**.**money** = money;  }   @Override  **public** String toString() {  **return "Account{"** +  **"id="** + **id** +  **", uid="** + **uid** +  **", money="** + **money** +  **'}'**;  } } |
| **public class** AccountUser **extends** Account {   **private** String **username**;  **private** String **address**;   **public** String getUsername() {  **return username**;  }   **public void** setUsername(String username) {  **this**.**username** = username;  }   **public** String getAddress() {  **return address**;  }   **public void** setAddress(String address) {  **this**.**address** = address;  }   @Override  **public** String toString() {  **return super**.toString()+**" AccountUser{"** +  **"username='"** + **username** + **'\''** +  **", address='"** + **address** + **'\''** +  **'}'**;  } } |
| **public interface** IAccountDao {   */\*\*  \* 查询所有账户，同时还要获取到当前账户的所属用户信息  \** ***@return*** *\*/* List<Account> findAll();   */\*\*  \* 查询所有账户，并且带有用户名称和地址信息  \** ***@return*** *\*/* List<AccountUser> findAllAccount(); } |
| **public class** AccountTest {   **private** InputStream **in**;  **private** SqlSession **sqlSession**;  **private** IAccountDao **accountDao**;   @Before*//用于在测试方法执行之前执行* **public void** init()**throws** Exception{  *//1.读取配置文件，生成字节输入流* **in** = Resources.*getResourceAsStream*(**"SqlMapConfig.xml"**);  *//2.获取SqlSessionFactory* SqlSessionFactory factory = **new** SqlSessionFactoryBuilder().build(**in**);  *//3.获取SqlSession对象* **sqlSession** = factory.openSession(**true**);  *//4.获取dao的代理对象* **accountDao** = **sqlSession**.getMapper(IAccountDao.**class**);  }   @After*//用于在测试方法执行之后执行* **public void** destroy()**throws** Exception{  *//提交事务  // sqlSession.commit();  //6.释放资源* **sqlSession**.close();  **in**.close();  }   */\*\*  \* 测试查询所有  \*/* @Test  **public void** testFindAll(){  List<Account> accounts = **accountDao**.findAll();  **for**(Account account : accounts){  System.***out***.println(**"--------每个account的信息------------"**);  System.***out***.println(account);  System.***out***.println(account.getUser());  }  }    */\*\*  \* 测试查询所有账户，同时包含用户名称和地址  \*/* @Test  **public void** testFindAllAccountUser(){  List<AccountUser> aus = **accountDao**.findAllAccount();  **for**(AccountUser au : aus){  System.***out***.println(au);  }  } } |

### 8.3多对一查询操作

|  |
| --- |
| *<?***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="com.byf.dao.IUserDao"**>   *<!-- 定义User的resultMap-->* <**resultMap id="userAccountMap" type="user"**>  <**id property="id" column="id"**></**id**>  <**result property="username" column="username"**></**result**>  <**result property="address" column="address"**></**result**>  <**result property="sex" column="sex"**></**result**>  <**result property="birthday" column="birthday"**></**result**>  *<!-- 配置user对象中accounts集合的映射 -->* <**collection property="accounts" ofType="account"**>  <**id column="aid" property="id"**></**id**>  <**result column="uid" property="uid"**></**result**>  <**result column="money" property="money"**></**result**>  </**collection**>  </**resultMap**>   *<!-- 查询所有 -->* <**select id="findAll" resultMap="userAccountMap"**>  select *\** from user u left outer join account a on u.id = a.uid  </**select**>   *<!-- 根据id查询用户 -->* <**select id="findById" parameterType="INT" resultType="user"**>  select *\** from user where id = #{uid}  </**select**>  </**mapper**> |

|  |
| --- |
| **public class** User **implements** Serializable {   **private** Integer **id**;  **private** String **username**;  **private** String **address**;  **private** String **sex**;  **private** Date **birthday**;   *//一对多关系映射：主表实体应该包含从表实体的集合引用* **private** List<Account> **accounts**;   **public** List<Account> getAccounts() {  **return accounts**;  }   **public void** setAccounts(List<Account> accounts) {  **this**.**accounts** = accounts;  }   **public** Integer getId() {  **return id**;  }   **public void** setId(Integer id) {  **this**.**id** = id;  }   **public** String getUsername() {  **return username**;  }   **public void** setUsername(String username) {  **this**.**username** = username;  }   **public** String getAddress() {  **return address**;  }   **public void** setAddress(String address) {  **this**.**address** = address;  }   **public** String getSex() {  **return sex**;  }   **public void** setSex(String sex) {  **this**.**sex** = sex;  }   **public** Date getBirthday() {  **return birthday**;  }   **public void** setBirthday(Date birthday) {  **this**.**birthday** = birthday;  }   @Override  **public** String toString() {  **return "User{"** +  **"id="** + **id** +  **", username='"** + **username** + **'\''** +  **", address='"** + **address** + **'\''** +  **", sex='"** + **sex** + **'\''** +  **", birthday="** + **birthday** +  **'}'**;  } } |

|  |
| --- |
| **public class** UserTest {   **private** InputStream **in**;  **private** SqlSession **sqlSession**;  **private** IUserDao **userDao**;   @Before*//用于在测试方法执行之前执行* **public void** init()**throws** Exception{  *//1.读取配置文件，生成字节输入流* **in** = Resources.*getResourceAsStream*(**"SqlMapConfig.xml"**);  *//2.获取SqlSessionFactory* SqlSessionFactory factory = **new** SqlSessionFactoryBuilder().build(**in**);  *//3.获取SqlSession对象* **sqlSession** = factory.openSession(**true**);  *//4.获取dao的代理对象* **userDao** = **sqlSession**.getMapper(IUserDao.**class**);  }   @After*//用于在测试方法执行之后执行* **public void** destroy()**throws** Exception{  *//提交事务  // sqlSession.commit();  //6.释放资源* **sqlSession**.close();  **in**.close();  }   */\*\*  \* 测试查询所有  \*/* @Test  **public void** testFindAll(){  List<User> users = **userDao**.findAll();  **for**(User user : users){  System.***out***.println(**"-----每个用户的信息------"**);  System.***out***.println(user);  System.***out***.println(user.getAccounts());  }  } } |

### 8.4多对多查询操作

|  |
| --- |
| DROP TABLE IF EXISTS `role`;  CREATE TABLE `role` (  `ID` int(11) NOT NULL COMMENT '编号',  `ROLE\_NAME` varchar(30) default NULL COMMENT '角色名称',  `ROLE\_DESC` varchar(60) default NULL COMMENT '角色描述',  PRIMARY KEY (`ID`)  ) ENGINE=InnoDB DEFAULT CHARSET=utf8;  insert into `role`(`ID`,`ROLE\_NAME`,`ROLE\_DESC`) values (1,'院长','管理整个学院'),(2,'总裁','管理整个公司'),(3,'校长','管理整个学校');  DROP TABLE IF EXISTS `user\_role`;  CREATE TABLE `user\_role` (  `UID` int(11) NOT NULL COMMENT '用户编号',  `RID` int(11) NOT NULL COMMENT '角色编号',  PRIMARY KEY (`UID`,`RID`),  KEY `FK\_Reference\_10` (`RID`),  CONSTRAINT `FK\_Reference\_10` FOREIGN KEY (`RID`) REFERENCES `role` (`ID`),  CONSTRAINT `FK\_Reference\_9` FOREIGN KEY (`UID`) REFERENCES `user` (`id`)  ) ENGINE=InnoDB DEFAULT CHARSET=utf8;  insert into `user\_role`(`UID`,`RID`) values (41,1),(45,1),(41,2); |
| *<?***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**>  *<!-- 配置properties-->* <**properties resource="jdbcConfig.properties"**></**properties**>   *<!--使用typeAliases配置别名，它只能配置domain中类的别名 -->* <**typeAliases**>  <**package name="com.itheima.domain"**></**package**>  </**typeAliases**>   *<!--配置环境-->* <**environments default="mysql"**>  *<!-- 配置mysql的环境-->* <**environment id="mysql"**>  *<!-- 配置事务 -->* <**transactionManager type="JDBC"**></**transactionManager**>   *<!--配置连接池-->* <**dataSource type="POOLED"**>  <**property name="driver" value="${jdbc.driver}"**></**property**>  <**property name="url" value="${jdbc.url}"**></**property**>  <**property name="username" value="${jdbc.username}"**></**property**>  <**property name="password" value="${jdbc.password}"**></**property**>  </**dataSource**>  </**environment**>  </**environments**>  *<!-- 配置映射文件的位置 -->* <**mappers**>  <**package name="com.itheima.dao"**></**package**>  </**mappers**> </**configuration**> |
| *# Set root category priority to INFO and its only appender to CONSOLE. #log4j.rootCategory=INFO, CONSOLE debug info warn error fatal* **log4j.rootCategory**=**debug, CONSOLE, LOGFILE** *# Set the enterprise logger category to FATAL and its only appender to CONSOLE.* **log4j.logger.org.apache.axis.enterprise**=**FATAL, CONSOLE** *# CONSOLE is set to be a ConsoleAppender using a PatternLayout.* **log4j.appender.CONSOLE**=**org.apache.log4j.ConsoleAppender log4j.appender.CONSOLE.layout**=**org.apache.log4j.PatternLayout log4j.appender.CONSOLE.layout.ConversionPattern**=**%d{ISO8601} %-6r [%15.15t] %-5p %30.30c %x - %m\n** *# LOGFILE is set to be a File appender using a PatternLayout.* **log4j.appender.LOGFILE**=**org.apache.log4j.FileAppender log4j.appender.LOGFILE.File**=**d:**\a**xis.log log4j.appender.LOGFILE.Append**=**true log4j.appender.LOGFILE.layout**=**org.apache.log4j.PatternLayout log4j.appender.LOGFILE.layout.ConversionPattern**=**%d{ISO8601} %-6r [%15.15t] %-5p %30.30c %x - %m\n** |
| **jdbc.driver**=**com.mysql.jdbc.Driver jdbc.url**=**jdbc:mysql://localhost:3306/eesy\_mybatis jdbc.username**=**root jdbc.password**=**1234** |
| *<?***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="com.itheima.dao.IUserDao"**>   *<!-- 定义User的resultMap-->* <**resultMap id="userMap" type="user"**>  <**id property="id" column="id"**></**id**>  <**result property="username" column="username"**></**result**>  <**result property="address" column="address"**></**result**>  <**result property="sex" column="sex"**></**result**>  <**result property="birthday" column="birthday"**></**result**>  *<!-- 配置角色集合的映射 -->* <**collection property="roles" ofType="role"**>  <**id property="roleId" column="rid"**></**id**>  <**result property="roleName" column="role\_name"**></**result**>  <**result property="roleDesc" column="role\_desc"**></**result**>  </**collection**>  </**resultMap**>   *<!-- 查询所有 -->* <**select id="findAll" resultMap="userMap"**>  select u.*\**,r.id as rid,r.role\_name,r.role\_desc from user u  left outer join user\_role ur on u.id = ur.uid  left outer join role r on r.id = ur.rid  </**select**>   *<!-- 根据id查询用户 -->* <**select id="findById" parameterType="INT" resultType="user"**>  select *\** from user where id = #{uid}  </**select**>  </**mapper**> |
| *<?***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="com.itheima.dao.IRoleDao"**>   *<!--定义role表的ResultMap-->* <**resultMap id="roleMap" type="role"**>  <**id property="roleId" column="rid"**></**id**>  <**result property="roleName" column="role\_name"**></**result**>  <**result property="roleDesc" column="role\_desc"**></**result**>  <**collection property="users" ofType="user"**>  <**id column="id" property="id"**></**id**>  <**result column="username" property="username"**></**result**>  <**result column="address" property="address"**></**result**>  <**result column="sex" property="sex"**></**result**>  <**result column="birthday" property="birthday"**></**result**>  </**collection**>  </**resultMap**>   *<!--查询所有-->* <**select id="findAll" resultMap="roleMap"**>  select u.*\**,r.id as rid,r.role\_name,r.role\_desc from role r  left outer join user\_role ur on r.id = ur.rid  left outer join user u on u.id = ur.uid  </**select**> </**mapper**> |
| **public class** User **implements** Serializable {   **private** Integer **id**;  **private** String **username**;  **private** String **address**;  **private** String **sex**;  **private** Date **birthday**;   *//多对多的关系映射：一个用户可以具备多个角色* **private** List<Role> **roles**;   **public** List<Role> getRoles() {  **return roles**;  }   **public void** setRoles(List<Role> roles) {  **this**.**roles** = roles;  }   **public** Integer getId() {  **return id**;  }   **public void** setId(Integer id) {  **this**.**id** = id;  }   **public** String getUsername() {  **return username**;  }   **public void** setUsername(String username) {  **this**.**username** = username;  }   **public** String getAddress() {  **return address**;  }   **public void** setAddress(String address) {  **this**.**address** = address;  }   **public** String getSex() {  **return sex**;  }   **public void** setSex(String sex) {  **this**.**sex** = sex;  }   **public** Date getBirthday() {  **return birthday**;  }   **public void** setBirthday(Date birthday) {  **this**.**birthday** = birthday;  }   @Override  **public** String toString() {  **return "User{"** +  **"id="** + **id** +  **", username='"** + **username** + **'\''** +  **", address='"** + **address** + **'\''** +  **", sex='"** + **sex** + **'\''** +  **", birthday="** + **birthday** +  **'}'**;  } } |
| **public class** Role **implements** Serializable {   **private** Integer **roleId**;  **private** String **roleName**;  **private** String **roleDesc**;   *//多对多的关系映射：一个角色可以赋予多个用户* **private** List<User> **users**;   **public** List<User> getUsers() {  **return users**;  }   **public void** setUsers(List<User> users) {  **this**.**users** = users;  }   **public** Integer getRoleId() {  **return roleId**;  }   **public void** setRoleId(Integer roleId) {  **this**.**roleId** = roleId;  }   **public** String getRoleName() {  **return roleName**;  }   **public void** setRoleName(String roleName) {  **this**.**roleName** = roleName;  }   **public** String getRoleDesc() {  **return roleDesc**;  }   **public void** setRoleDesc(String roleDesc) {  **this**.**roleDesc** = roleDesc;  }   @Override  **public** String toString() {  **return "Role{"** +  **"roleId="** + **roleId** +  **", roleName='"** + **roleName** + **'\''** +  **", roleDesc='"** + **roleDesc** + **'\''** +  **'}'**;  } } |
| **public class** UserTest {   **private** InputStream **in**;  **private** SqlSession **sqlSession**;  **private** IUserDao **userDao**;   @Before*//用于在测试方法执行之前执行* **public void** init()**throws** Exception{  *//1.读取配置文件，生成字节输入流* **in** = Resources.*getResourceAsStream*(**"SqlMapConfig.xml"**);  *//2.获取SqlSessionFactory* SqlSessionFactory factory = **new** SqlSessionFactoryBuilder().build(**in**);  *//3.获取SqlSession对象* **sqlSession** = factory.openSession(**true**);  *//4.获取dao的代理对象* **userDao** = **sqlSession**.getMapper(IUserDao.**class**);  }   @After*//用于在测试方法执行之后执行* **public void** destroy()**throws** Exception{  *//提交事务  // sqlSession.commit();  //6.释放资源* **sqlSession**.close();  **in**.close();  }   */\*\*  \* 测试查询所有  \*/* @Test  **public void** testFindAll(){  List<User> users = **userDao**.findAll();  **for**(User user : users){  System.***out***.println(**"-----每个用户的信息------"**);  System.***out***.println(user);  System.***out***.println(user.getRoles());  }  }    } |
| **public class** RoleTest {   **private** InputStream **in**;  **private** SqlSession **sqlSession**;  **private** IRoleDao **roleDao**;   @Before*//用于在测试方法执行之前执行* **public void** init()**throws** Exception{  *//1.读取配置文件，生成字节输入流* **in** = Resources.*getResourceAsStream*(**"SqlMapConfig.xml"**);  *//2.获取SqlSessionFactory* SqlSessionFactory factory = **new** SqlSessionFactoryBuilder().build(**in**);  *//3.获取SqlSession对象* **sqlSession** = factory.openSession(**true**);  *//4.获取dao的代理对象* **roleDao** = **sqlSession**.getMapper(IRoleDao.**class**);  }   @After*//用于在测试方法执行之后执行* **public void** destroy()**throws** Exception{  *//提交事务  // sqlSession.commit();  //6.释放资源* **sqlSession**.close();  **in**.close();  }   */\*\*  \* 测试查询所有  \*/* @Test  **public void** testFindAll(){  List<Role> roles = **roleDao**.findAll();  **for**(Role role : roles){  System.***out***.println(**"---每个角色的信息----"**);  System.***out***.println(role);  System.***out***.println(role.getUsers());  }  }    } |

## 第9节JNDI扩展知识



G:\Mybatis\mybatis\_coding\mybatis\myabtis03\_jndi\src\main\webapp\META-INF\context.xml

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**Context**> *<!--  <Resource  name="jdbc/eesy\_mybatis" 数据源的名称 type="javax.sql.DataSource" 数据源类型 auth="Container" 数据源提供者 maxActive="20" 最大活动数 maxWait="10000" 最大等待时间 maxIdle="5" 最大空闲数 username="root" 用户名 password="1234" 密码 driverClassName="com.mysql.jdbc.Driver" 驱动类 url="jdbc:mysql://localhost:3306/eesy\_mybatis" 连接url字符串 />  -->* <**Resource  name="jdbc/eesy\_mybatis" type="javax.sql.DataSource" auth="Container" maxActive="20" maxWait="10000" maxIdle="5" username="root" password="1234" driverClassName="com.mysql.jdbc.Driver" url="jdbc:mysql://localhost:3306/eesy\_mybatis"** /> </**Context**> |
| <%@ **page import**="**java.io.InputStream**" %> <%@ **page import**="**org.apache.ibatis.io.Resources**" %> <%@ **page import**="**org.apache.ibatis.session.SqlSessionFactoryBuilder**" %> <%@ **page import**="**org.apache.ibatis.session.SqlSessionFactory**" %> <%@ **page import**="**org.apache.ibatis.session.SqlSession**" %> <%@ **page import**="**com.byf.dao.IUserDao**" %> <%@ **page import**="**com.byf.domain.User**" %> <%@ **page import**="**java.util.List**" %> <%@ **page language**="**java**" **contentType**="**text/html; charset=UTF-8**" **pageEncoding**="**UTF-8**"%> <**html**> <**body**> <**h2**>Hello World!</**h2**> **<%** *//1.读取配置文件* InputStream in = Resources.*getResourceAsStream*(**"SqlMapConfig.xml"**);  *//2.根据配置文件构建SqlSessionFactory* SqlSessionFactoryBuilder builder = **new** SqlSessionFactoryBuilder();  SqlSessionFactory factory = builder.build(in);  *//3.使用SqlSessionFactory创建SqlSession对象* SqlSession sqlSession = factory.openSession();  *//4.使用SqlSession构建Dao的代理对象* IUserDao userDao = sqlSession.getMapper(IUserDao.**class**);  *//5.执行dao中的findAll方法* List<User> users = userDao.findAll();  **for**(User user : users){  System.***out***.println(user);  }  *//6.释放资源* sqlSession.close();  in.close(); **%>** </**body**> </**html**> |
| *<?***xml version="1.0" encoding="UTF-8"***?> <!-- 导入约束 -->* **<!DOCTYPE configuration  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-config.dtd"*>*** <**configuration**>  <**typeAliases**>  <**package name="com.byf.domain"**></**package**>  </**typeAliases**>  *<!-- 配置mybatis的环境 -->* <**environments default="mysql"**>  *<!-- 配置mysql的环境 -->* <**environment id="mysql"**>  *<!-- 配置事务控制的方式 -->* <**transactionManager type="JDBC"**></**transactionManager**>  *<!-- 配置连接数据库的必备信息 type属性表示是否使用数据源（连接池）-->* <**dataSource type="JNDI"**>  <**property name="data\_source" value="java:comp/env/jdbc/eesy\_mybatis"**/>  </**dataSource**>  </**environment**>  </**environments**>   *<!-- 指定mapper配置文件的位置 -->* <**mappers**>  <**mapper resource="com/byf/dao/IUserDao.xml"**/>  </**mappers**> </**configuration**> |

### 第10节Mybatis的缓存

Mybatis中的延迟加载

·延迟加载

·立即加载

Mybatis中的缓存

·什么是缓存

·为什么使用缓存

·什么样的数据能使用缓存，什么样的数据不能使用

·Mybatis中的一级缓存和二级缓存

Mybatis中的注解开发

·环境搭建

·单表CRUD操作（代理Dao方式）

·多表查询操作

·缓存的配置

### 10.1延迟加载

问题：在一对多中，一个用户有100个账户。

在查询用户的时候，要不要把关联的账户查出来。

在查询账户的时候，要不要把对应的用户查出来。

在查询用户时，用户下的账户信息应该是，什么时候使用，什么时候查询的。

在查询账户时，账户的所属用户信息应该是随着账户查询时，一起查询出来。

什么是延迟加载？

在真正使用数据时才发起查询，不用的时候不需要查询。按需加载（懒加载）

什么是立即加载？

不管用不用，只要一调用方法，马上发起查询。

在对应的四种表关系中：一对多，多对一，一对一，多对多

一对多，多对多：延迟加载

多对一，一对一：立即加载

[http://www.mybatis.org/mybatis-3/zh/configuration.html#settings](http://www.mybatis.org/mybatis-3/zh/configuration.html" \l "settings)

|  |  |  |  |
| --- | --- | --- | --- |
| lazyLoadingEnabled | 延迟加载的全局开关。当开启时，所有关联对象都会延迟加载。 特定关联关系中可通过设置 fetchType 属性来覆盖该项的开关状态。 | true | false | false |
| aggressiveLazyLoading | 当开启时，任何方法的调用都会加载该对象的所有属性。 否则，每个属性会按需加载（参考 lazyLoadTriggerMethods)。 | true | false | false （在 3.4.1 及之前的版本默认值为 true） |
|  |  |  |  |

多对一，一对一延迟加载

|  |
| --- |
| <**configuration**>  *<!-- 配置properties-->* <**properties resource="jdbcConfig.properties"**></**properties**>   *<!-- 配置参数-->* <**settings**>  *<!-- 开启Mybatis支持延迟加载 -->* <**setting name="lazyLoadingEnabled" value="false"**/>  <**setting name="aggressiveLazyLoading" value="false"**/>  </**settings**>  ....  </**configuration**> |
| 14:13,412 [ main] DEBUG ansaction.jdbc.JdbcTransaction - Opening JDBC Connection  14:13,725 [ main] DEBUG source.pooled.PooledDataSource - Created connection 1154002927.  14:13,728 [ main] DEBUG om.byf.dao.IAccountDao.findAll - ==> Preparing: select \* from account;  14:13,773 [ main] DEBUG om.byf.dao.IAccountDao.findAll - ==> Parameters:  14:13,803 [ main] DEBUG com.byf.dao.IUserDao.findById - ====> Preparing: select \* from user where id = ?  14:13,804 [ main] DEBUG com.byf.dao.IUserDao.findById - ====> Parameters: 46(Integer)  14:13,813 [ main] DEBUG com.byf.dao.IUserDao.findById - <==== Total: 1  14:13,814 [ main] DEBUG com.byf.dao.IUserDao.findById - ====> Preparing: select \* from user where id = ?  14:13,815 [ main] DEBUG com.byf.dao.IUserDao.findById - ====> Parameters: 45(Integer)  14:13,825 [ main] DEBUG com.byf.dao.IUserDao.findById - <==== Total: 1  14:13,826 [ main] DEBUG om.byf.dao.IAccountDao.findAll - <== Total: 3  14:13,826 [ main] DEBUG ansaction.jdbc.JdbcTransaction - Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@44c8afef]  14:13,827 [ main] DEBUG source.pooled.PooledDataSource - Returned connection 1154002927 to pool. |

|  |
| --- |
| <**settings**>  *<!-- 开启Mybatis支持延迟加载 -->* <**setting name="lazyLoadingEnabled" value="true"**/>  <**setting name="aggressiveLazyLoading" value="false"**/> </**settings**> |
| 17:08,278 [ main] DEBUG ansaction.jdbc.JdbcTransaction - Opening JDBC Connection  17:08,577 [ main] DEBUG source.pooled.PooledDataSource - Created connection 1154002927.  17:08,580 [ main] DEBUG om.byf.dao.IAccountDao.findAll - ==> Preparing: select \* from account;  17:08,632 [ main] DEBUG om.byf.dao.IAccountDao.findAll - ==> Parameters:  17:08,705 [ main] DEBUG om.byf.dao.IAccountDao.findAll - <== Total: 3  17:08,706 [ main] DEBUG ansaction.jdbc.JdbcTransaction - Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@44c8afef]  17:08,706 [ main] DEBUG source.pooled.PooledDataSource - Returned connection 1154002927 to pool. |

一对多

|  |
| --- |
| <**resultMap id="userAccountMap" type="user"**>  <**id property="id" column="id"**></**id**>  <**result property="username" column="username"**></**result**>  <**result property="address" column="address"**></**result**>  <**result property="sex" column="sex"**></**result**>  <**result property="birthday" column="birthday"**></**result**>  *<!-- 配置user对象中accounts集合的映射 -->  <!--<collection property="accounts" ofType="account">  <id column="aid" property="id"></id>  <result column="uid" property="uid"></result>  <result column="money" property="money"></result>  </collection>-->* <**collection property="accounts" ofType="account" select="com.byf.dao.IAccountDao.findAccountByUid" column="id"**></**collection**>  </**resultMap**> |
| *<!-- 配置参数-->* <**settings**>  *<!-- 开启Mybatis支持延迟加载 -->* <**setting name="lazyLoadingEnabled" value="true"**/>  <**setting name="aggressiveLazyLoading" value="false"**/> </**settings**> |
| 40:41,201 [ main] DEBUG ansaction.jdbc.JdbcTransaction - Opening JDBC Connection  40:41,489 [ main] DEBUG source.pooled.PooledDataSource - Created connection 1154002927.  40:41,492 [ main] DEBUG com.byf.dao.IUserDao.findAll - ==> Preparing: select \* from user  40:41,557 [ main] DEBUG com.byf.dao.IUserDao.findAll - ==> Parameters:  40:41,584 [ main] DEBUG o.IAccountDao.findAccountByUid - ====> Preparing: select \* from account where uid=?  40:41,585 [ main] DEBUG o.IAccountDao.findAccountByUid - ====> Parameters: 41(Integer)  40:41,586 [ main] DEBUG o.IAccountDao.findAccountByUid - <==== Total: 0  40:41,587 [ main] DEBUG o.IAccountDao.findAccountByUid - ====> Preparing: select \* from account where uid=?  40:41,588 [ main] DEBUG o.IAccountDao.findAccountByUid - ====> Parameters: 42(Integer)  40:41,589 [ main] DEBUG o.IAccountDao.findAccountByUid - <==== Total: 0  40:41,594 [ main] DEBUG o.IAccountDao.findAccountByUid - ====> Preparing: select \* from account where uid=?  40:41,594 [ main] DEBUG o.IAccountDao.findAccountByUid - ====> Parameters: 43(Integer)  40:41,595 [ main] DEBUG o.IAccountDao.findAccountByUid - <==== Total: 0  40:41,596 [ main] DEBUG o.IAccountDao.findAccountByUid - ====> Preparing: select \* from account where uid=?  40:41,596 [ main] DEBUG o.IAccountDao.findAccountByUid - ====> Parameters: 45(Integer)  40:41,598 [ main] DEBUG o.IAccountDao.findAccountByUid - <==== Total: 1  40:41,599 [ main] DEBUG o.IAccountDao.findAccountByUid - ====> Preparing: select \* from account where uid=?  40:41,600 [ main] DEBUG o.IAccountDao.findAccountByUid - ====> Parameters: 46(Integer)  40:41,602 [ main] DEBUG o.IAccountDao.findAccountByUid - <==== Total: 2  40:41,603 [ main] DEBUG o.IAccountDao.findAccountByUid - ====> Preparing: select \* from account where uid=?  40:41,603 [ main] DEBUG o.IAccountDao.findAccountByUid - ====> Parameters: 48(Integer)  40:41,604 [ main] DEBUG o.IAccountDao.findAccountByUid - <==== Total: 0  40:41,606 [ main] DEBUG o.IAccountDao.findAccountByUid - ====> Preparing: select \* from account where uid=?  40:41,607 [ main] DEBUG o.IAccountDao.findAccountByUid - ====> Parameters: 59(Integer)  40:41,607 [ main] DEBUG o.IAccountDao.findAccountByUid - <==== Total: 0  40:41,609 [ main] DEBUG com.byf.dao.IUserDao.findAll - <== Total: 7  40:41,609 [ main] DEBUG ansaction.jdbc.JdbcTransaction - Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@44c8afef]  40:41,610 [ main] DEBUG source.pooled.PooledDataSource - Returned connection 1154002927 to pool. |
| *<!-- 根据用户id查询账户列表 -->* <**select id="findAccountByUid" resultType="account"**>  select *\** from account where uid=#{uid} </**select**> |

Ctrl + Shift + z 前进

|  |
| --- |
| *<!-- 配置参数-->* <**settings**>  *<!-- 开启Mybatis支持延迟加载 -->* <**setting name="lazyLoadingEnabled" value="true"**/>  <**setting name="aggressiveLazyLoading" value="false"**/> </**settings**> |
| *<!-- 根据id查询用户 -->* <**select id="findById" parameterType="INT" resultType="user"**>  select *\** from user where id = #{uid} </**select**> |
| 43:59,849 [ main] DEBUG ansaction.jdbc.JdbcTransaction - Opening JDBC Connection  44:00,170 [ main] DEBUG source.pooled.PooledDataSource - Created connection 1154002927.  44:00,174 [ main] DEBUG com.byf.dao.IUserDao.findAll - ==> Preparing: select \* from user  44:00,242 [ main] DEBUG com.byf.dao.IUserDao.findAll - ==> Parameters:  44:00,337 [ main] DEBUG com.byf.dao.IUserDao.findAll - <== Total: 7  44:00,338 [ main] DEBUG ansaction.jdbc.JdbcTransaction - Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@44c8afef]  44:00,338 [ main] DEBUG source.pooled.PooledDataSource - Returned connection 1154002927 to pool. |