# 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

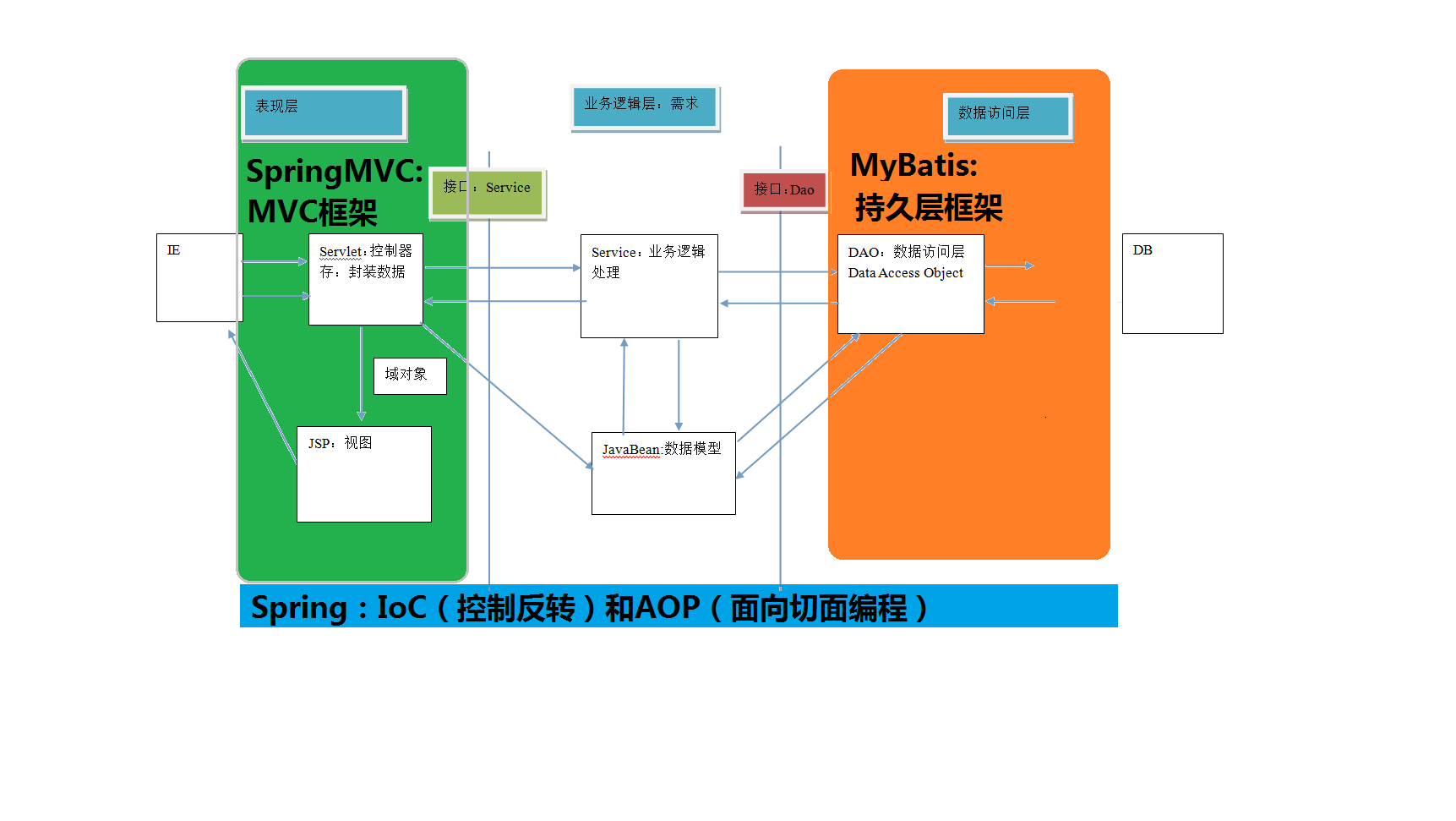
框架（Framework）是整个或部分系统的可重用设计，表现为一组抽象构件及构件实例间交互的方法;另一种定义认为，框架是可被应用开发者定制的应用骨架。前者是从应用方面而后者是从目的方面给出的定义简而言之，框架其实就是某种应用的半成品，就是一组组件，供你选用完成你自己的系统。简单说就是使用别人搭好的舞台，你来做表演。而且，框架一般是成熟的，不断升级的软件。

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：

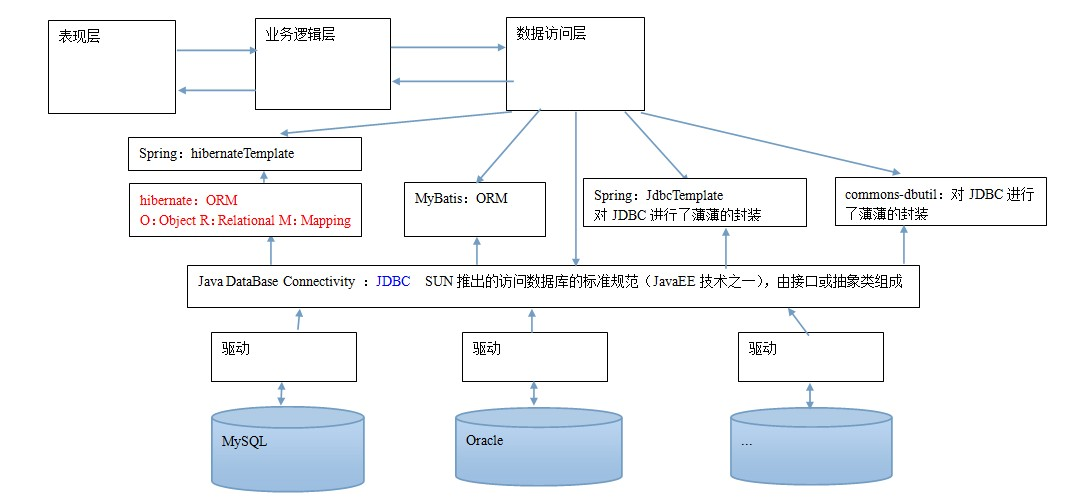
就是把数据库表和实体类及实体类的属性对应起来

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

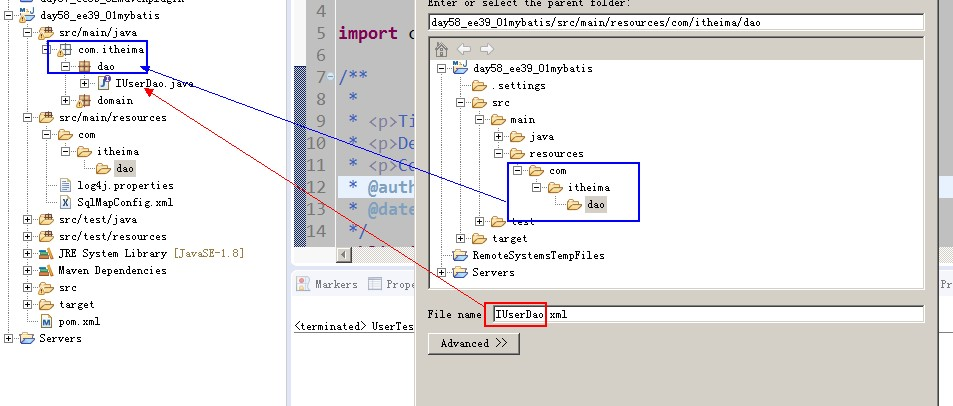
目录在创建时 com/byf/dao是三级目录，但com.byf.dao是一级目录

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)** |

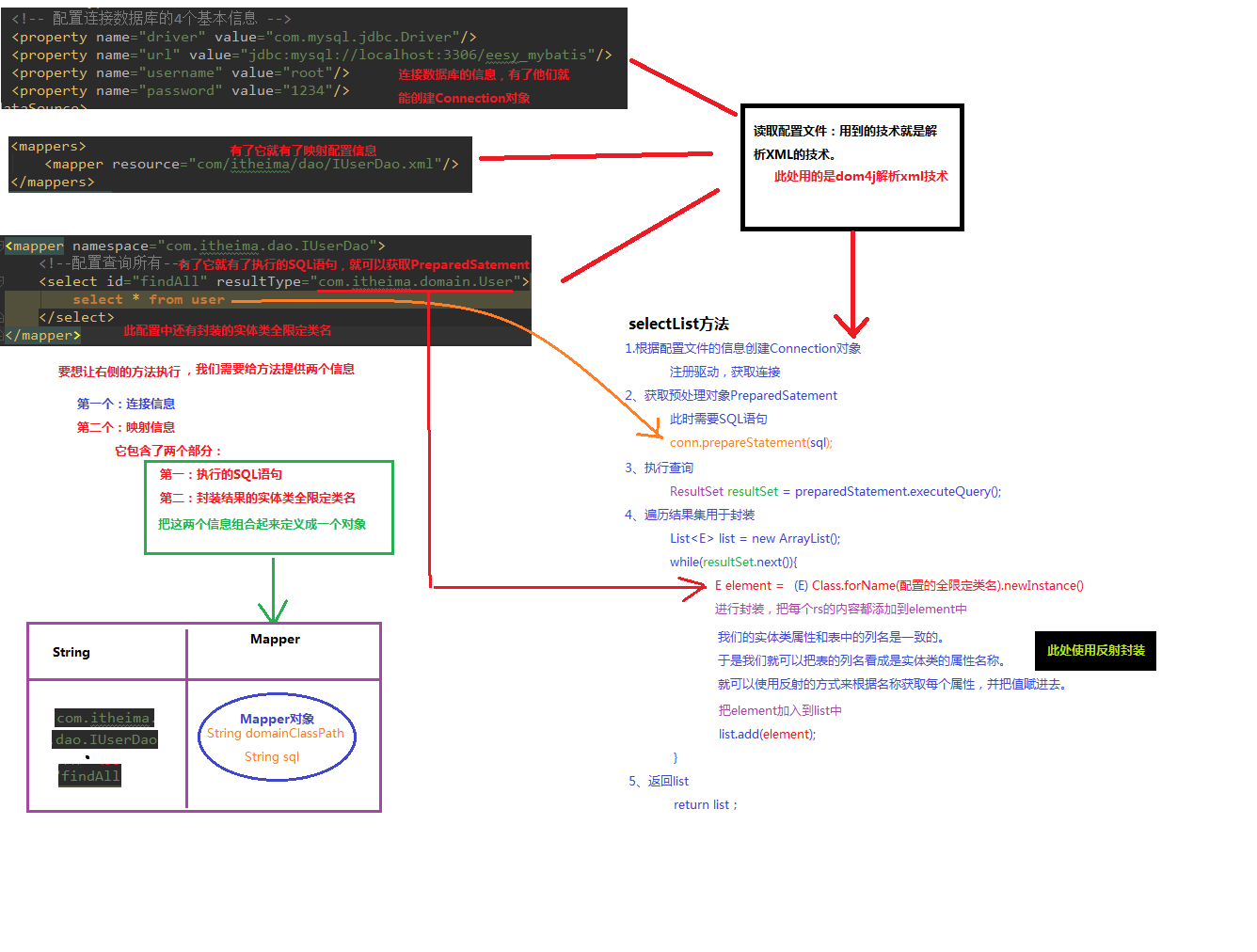
## 第3节自定义Mybatis框架.

### 3.1mybatis的分析

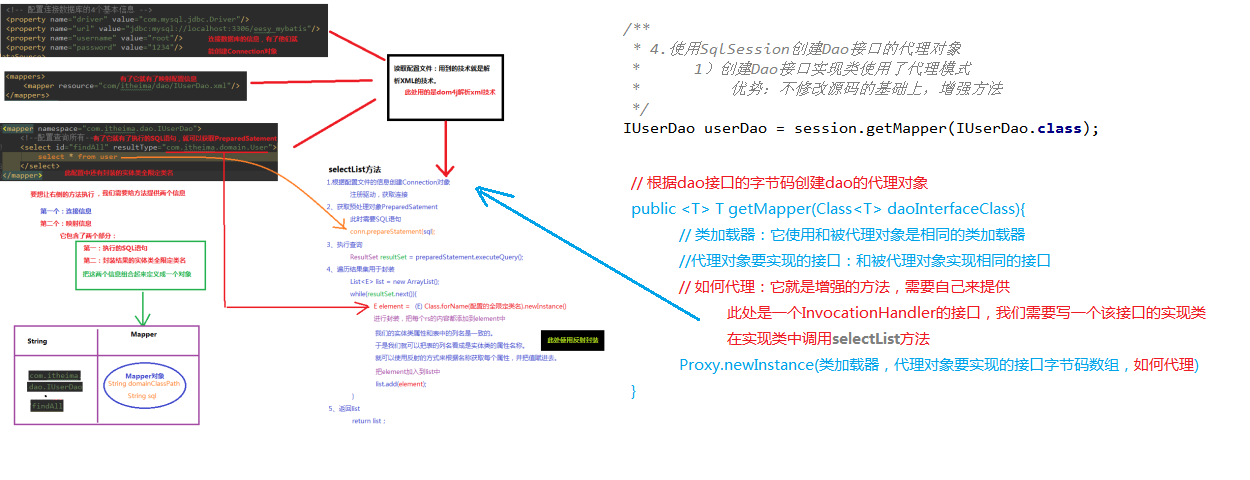
Mybatis使用代理dao方式实现CURD时做了两件事：

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(); } |

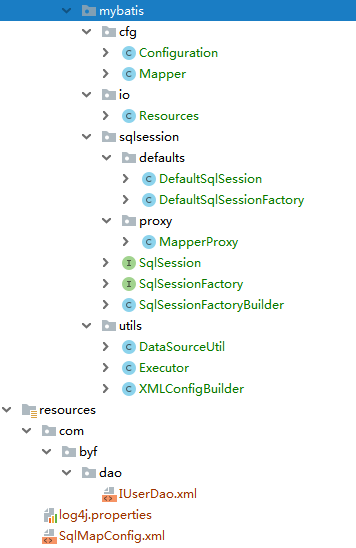
|  |
| --- |
| **public interface** SqlSession {  */\*\*  \*  \** ***@param daoInterfaceClass*** *: dao接口的字节码  \** ***@param <T>*** *\** ***@return*** *\*/* <T> T getMapper(Class<T> daoInterfaceClass);   */\*\*  \* 释放资源  \*/* **void** close(); } |

### 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;  } } |
|  |