

Mybatis

[了解]认识框架

什么是框架

- 框架说白了就是一个加载，比如表演节目，舞台已经搭建，表演节目看你需求
- 框架是一个半成品
- 对于 java 语言来说，框架就是别人代码的封装，拿来主义，在框架的基础上我们进一步开发。复用性/重用性

框架要解决的问题

框架要解决的是技术整合问题。现在软件开发环境和软件规模非常大，我们不可能所有系统代码都从 0 开始敲，需要优秀的框架把基础技术都整合完了，我们在它的基础上进一步开发。

提高性能、安全性、易维护、易扩展。最终提高团队开发效率

什么时候使用框架

企业级大型项目开发（避免大炮打蚊子）

怎么用框架

Java 框架共性：

- 一堆 jar 包，引入工程
- 对框架运行细节进行定制，往往通过 xml 配置文件（有格式规范）
- 在 java 程序中调用框架 API

[了解]软件分层及常用框架

什么是分层

分层就是把不同功能代码放到不同文件当中。本质是代码的拆分

- Model1 模式：jsp+javaBean（jsp 处理请求）

- Model2 模式 (MVC): jsp+serlvet+javabean (servlet 处理请求)
 - M: 模型 (service+dao)
 - V: 视图
 - C: controller
- 经典三层
 - web 表现层 (视图+controller)、service 层、dao 层

分层的好处

代码清晰、解耦、便于维护, 出现问题容易定位

分层开发下的常见框架

dao 层: hibernate、mybatis

service 层: spring 框架 (对象容器)

web 表现层: springmvc 框架

[了解]原生 JDBC 操作数据库分析

- 创建数据库
 - utf8_general_ci: ci 大小写不敏感
 - select * from user where username='admin' 可以同时查出大写 ADMIN 的那条记录
 - 初始化 user 表测试数据
 - 需求: 查询 user 表所有数据, 以 List 集合形式返回
 - 编写 POJO 类 User.java
- domain、entity、pojo, 本质都一样 (一堆的属性和 getter/setter)
- 原生 JDBC 操作数据库存在的问题

```
package com.mybatis.dao;

import com.mybatis.pojo.User;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.ArrayList;
import java.util.List;

public class UserDaoImpl implements UserDao {

    private String driver = "com.mysql.jdbc.Driver";
```

```

private String url =
"jdbc:mysql://localhost:3306/mybatis?characterEncoding=utf-8";
private String username = "root";
private String password = "root";

public List<User> queryUserList() throws Exception {
    List<User> userList = new ArrayList<User>();

    Class.forName(driver);
    /**
     * 问题一：频繁获取/释放数据库连接，影响数据库和应用性能
     * 解决：数据库连接池技术，C3P0,DRUID（阿里巴巴荣誉出品，号称前无
     古人后无来者世界最强没有之一）
     */
    Connection connection = DriverManager.getConnection(url, username,
password);
    /**
     * 问题二：sql 语句硬编码，后期难以维护
     * 解决：若 sql 语句和 java 代码分离，比如 sql 写在配置文件中。Mybatis 就
     是这么干的
     */
    String sql = "select * from user";
    /**
     * 问题三：sql 语句 where 条件和占位符一一对应，后期难以维护
     */
    // String sql1 = "select * from user where username=? and id=?";
    PreparedStatement preparedStatement = connection.prepareStatement(sql);
    // preparedStatement.setInt(1,2);
    ResultSet resultSet = preparedStatement.executeQuery();
    User user = null;
    /**
     * 问题四：结果集解析麻烦，查询列硬编码
     * 期望：如果单条记录直接返回实体对象，如果多条记录返回实体的集合
     */
    while(resultSet.next()) {
        user = new User();
        user.setId(resultSet.getInt("id"));
        user.setUsername(resultSet.getString("username"));
        user.setSex(resultSet.getString("sex"));
        user.setBirthday(resultSet.getDate("birthday"));
        user.setAddress(resultSet.getString("address"));

        userList.add(user);
    }
}

```

```

    }
    resultSet.close();
    preparedStatement.close();
    connection.close();
    return userList;
}
}

```

[了解]Mybatis 框架概述

Mybatis 原本是 Apache 软件基金会的一个开源项目叫做 iBatis，2010 年这个项目由 Apache 迁移到了 google code 管理才改名为 Mybatis，2013 年又迁移到了 GitHub。

注意：招聘信息和面试时候可能问你会 ibatis 吗

Mybatis 是一个优秀的持久层框架（Dao 层框架），它是对 JDBC 的封装，使得开发者只需要关注 Sql 语句（业务）本身即可，无需开发者处理加载驱动、获取连接、创建 Statement 等繁琐的过程。

Mybatis **最大的特点**是把 **Sql 语句**写在 **XML 配置文件**当中。而且 Mybatis 执行完 Sql 语句之后可以以对象形式返回（POJO/POJO 集合等）

Mybatis 是一个实现了 **ORM 思想**的持久层框架

ORM: Object/Relation Mapping 对象/关系映射

ORM 思想：将数据库中的关系数据表映射为 JAVA 中的对象，把对数据表的操作转换为对对象的操作，实现面向对象编程。因此 ORM 的目的是使得开发人员以面向对象的思想来操作数据库

public class User {	Pojo的属性和关系数	
private Integer id;	据表的字段是对应的	<input type="checkbox"/> id
private String username;		<input type="checkbox"/> 张三
private String sex;		<input type="checkbox"/> 李四
private Date birthday;		<input type="checkbox"/> 王五
private String address;		<input type="checkbox"/> 王六
		<input type="checkbox"/> (Auto) (NULL) (NULL) (NULL) (NULL)

比如：原来 insert 使用的是 insert into...，如果使用实现了 ORM 思想的持久层框架，就可以在 Java 程序中直接调用 api，比如 insert(User)，达到操作对象即操作数据库的效果

Hibernate 框架是一个**全自动的 ORM**持久层框架，只需要编写 POJO，在 xml 中定义好 Pojo 属性和数据表字段的映射/对应关系，就可以在 java 中实现类似 insert(User)的操作。Sql 语句都不用写。但是因为性能等问题，市场占有率越来越低

Mybatis 框架是一个**半自动的 ORM**持久层框架，也可以在 Java 中实现类似 insert(User)的操作最终操作数据库，但是需要我们自己写 Sql 语句。Mybatis 是目前比较流行的 Dao 层框架。

[理解]自定义 Mybatis 框架

自定义 Mybatis 框架开发前的共识:

- 所有的 Dao 层框架都是以接口的形式给我们提供增删改查的 API
- 我们今天自定义 Mybatis 框架只完成一个 API 接口: selectList
- 源码开发 → 打 jar 包并安装到本地 maven 仓库 → 其他项目引用自定义框架

开发自定义 Mybatis 框架

(1) 步骤 1: 创建 maven 工程, packing 为 jar

(2) 步骤 2: 定义框架对外 API 接口类, 接口类中只定义一个 selectList 方法

```
package frame.core;

import java.util.List;

/**
 * 框架对外接口类
 */
public interface SqlSession {

    <T> List<T> selectList() throws Exception;

}
```

```
package frame.core;

import java.util.List;

/**
 * 框架对外接口的实现类
 */
public class SqlSessionImpl implements SqlSession {

    @Override
    public <T> List<T> selectList() throws Exception {
        return null;
    }

}
```

(3) 步骤 3: 最后真正使用框架的时候需要 new 这个接口的实现类 SqlSessionImpl, 但是直接 new 接口的实现类不便于后期维护 (如果实现类名字发生变化或者变更实现类, 那么系统中众多调用的地方都需要改), 因此使用工厂设计模式去帮我们返回接口的实现类对象, 这样如果实现类发生变化, 我们只需要修改工厂类中一个地方代码即可, 便于维

护。

```
package frame.factory;

import frame.core.SqlSession;
import frame.core.SqlSessionImpl;

/**
 * 工厂类，用于返回 SqlSession 的实现类对象
 */
public class SqlSessionFactory {

    public SqlSession openSession() {
        SqlSessionImpl sqlSession = new SqlSessionImpl();
        return sqlSession;
    }

}
```

(4) 步骤 4: Dao 层框架终究是要操作数据库的，底层调用的就是 Jdbc，那么 Jdbc 要做的事情(注册驱动、获取连接等)在框架里一件也都不少，这些事情，我们需要在 SqlSessionImpl 这个接口实现类中完成(注意：主线还是按照 JDBC 来的，只是做了一般化处理，为了适应更多的业务场景，不仅是 User 表)

```
package frame.core;

import java.lang.reflect.Method;
import java.sql.*;
import java.util.ArrayList;
import java.util.List;

/**
 * 框架对外接口的实现类
 */
public class SqlSessionImpl implements SqlSession {

    // TODO 待办事项
    private String driver;
    private String url;
    private String username;
    private String password;

    @Override
    public <T> List<T> selectList() throws Exception {
        List<T> list = new ArrayList<>();

        Class.forName(driver);
```

```
Connection connection = DriverManager.getConnection(url,username,password);
```

```
// TODO 待办事项
```

```
String sql = "";
```

```
PreparedStatement preparedStatement = connection.prepareStatement(sql);
```

```
ResultSet resultSet = preparedStatement.executeQuery();
```

```
ResultSetMetaData metaData = resultSet.getMetaData();
```

```
int columnCount = metaData.getColumnCount();
```

```
List<String> columnLabels = new ArrayList<>();
```

```
for(int i = 1; i <= columnCount; i++) {
```

```
    columnLabels.add(metaData.getColumnLabel(i));
```

```
}
```

// 需要知道结果集实体 Pojo 的全限定类名，比如:com.mybatis.pojo.User，然后使用反射技术处理

```
// TODO 待办事项
```

```
String resultType = "";
```

```
Class aClass = Class.forName(resultType);
```

```
Method[] methods = aClass.getMethods();
```

```
// 取代 User user = null;
```

```
Object obj = null;
```

```
while(resultSet.next()) {
```

```
    // 取代 user = new User();
```

```
    obj = aClass.newInstance();
```

```
    // 取代 user.setUsername(resultSet.getString("username"));
```

```
    for(String columnLabel : columnLabels) {
```

```
        for(Method method : methods) {
```

```
            if(method.getName().equalsIgnoreCase("set" + columnLabel)) {
```

```
                method.invoke(obj,resultSet.getObject(columnLabel));
```

```
            }
```

```
        }
```

```
        list.add((T) obj);
```

```
    }
```

```
}
```

```
resultSet.close();
```

```
preparedStatement.close();
```

```
connection.close();
```

```
return list;
```

```

    }
}

```

(5) 步骤 5: 解决第一个 TODO 待办事项 (数据库信息的获取)

- 定义 xml, 约定其格式 (规范)

```

<?xml version="1.0" encoding="UTF-8" ?>
<configuration>
  <environments default="development">
    <environment id="development">
      <transactionManager type="JDBC" />
      <dataSource type="POOLED">
        <property name="driver" value="com.mysql.jdbc.Driver" />
        <property name="url" value="jdbc:mysql://127.0.0.1:3306/mybatis?characterEncoding=utf8" />
        <property name="username" value="root" />
        <property name="password" value="root" />
      </dataSource>
    </environment>
  </environments>
</configuration>

```

- 定义 xml 对应的 pojo 类 Configuration, 并修改 SqlSessionImpl 中的部分代码

```

/**
 * 框架对外接口的实现类
 */
public class SqlSessionImpl implements SqlSession {

    // TODO 待办事项
    private Configuration configuration;

    public void setConfiguration(Configuration configuration) {
        this.configuration = configuration;
    }

    @Override
    public <T> List<T> selectList() throws Exception {
        List<T> list = new ArrayList<>();

        Class.forName(configuration.getDriver());
        Connection connection = DriverManager.getConnection(configuration.getUrl(),
            configuration.getUsername(), configuration.getPassword());
    }
}

```

- 读取 xml 然后封装 pojo 对象供项目使用, 考虑放在 SqlSessionImpl 和 SqlSessionFactory 中都不合适因为会被读取多次, 所以我们再抽象一层 SqlSessionFactoryBuilder 类去构建工厂, 构建工厂的同时, 准备工厂内部所需要的材料: 读取一次 xml 文件, 然后一直使用

```

<!-- 读取xml所需jar-->
<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>

package frame.factory;

import frame.pojo.Configuration;
import org.dom4j.Document;
import org.dom4j.DocumentException;

```



```

import org.dom4j.Element;
import org.dom4j.io.SAXReader;

import java.io.InputStream;
import java.util.List;

/**
 * 构建工厂，并在这里读取一次 xml，避免 xml 重复读取
 */
public class SqlSessionFactoryBuilder {

    /**
     * 构建工厂的入口，让用户传入配置文件的文件流
     * @param inputStream
     * @return
     */
    public SqlSessionFactory build(InputStream inputStream){
        SqlSessionFactory sqlSessionFactory = new SqlSessionFactory();
        // 读取数据源信息.xml 为 Configuration 对象
        Configuration configuration = loadXmlConfig(inputStream);
        sqlSessionFactory.setConfiguration(configuration);
        return sqlSessionFactory;
    }

    /**
     * 读取 xml 信息为 Configuration 对象
     * @param inputStream
     * @return
     */
    private Configuration loadXmlConfig(InputStream inputStream){
        Configuration configuration = new Configuration();
        SAXReader saxReader = new SAXReader();
        try {
            Document document = saxReader.read(inputStream);
            Element rootElement = document.getRootElement();
            List<Element> selectNodes = rootElement.selectNodes("//property");

            if(selectNodes != null && selectNodes.size() > 0) {
                for (int i = 0; i < selectNodes.size(); i++) {
                    Element element = selectNodes.get(i);
                    String name = element.attributeValue("name");
                    String value = element.attributeValue("value");

                    if("driver".equalsIgnoreCase(name)) {

```

```

        configuration.setDriver(value);
    }
    if("url".equalsIgnoreCase(name)) {
        configuration.setUrl(value);
    }
    if("username".equalsIgnoreCase(name)) {
        configuration.setUsername(value);
    }
    if("password".equalsIgnoreCase(name)) {
        configuration.setPassword(value);
    }
}
}
} catch (DocumentException e) {
    e.printStackTrace();
}

return configuration;
}
}

```

(6) 步骤 6: 处理第二个和第三个待办事项 (sql 和 resultType 的获取)

- 定义 xml, 约定其格式 (规范)

```

<?xml version="1.0" encoding="utf-8" ?>
<mapper namespace="test">
    <select id="queryUserList" resultType="com.mybatis.pojo.User">
        SELECT * FROM USER;
    </select>
</mapper>

```

namespace: 分类管理 sql 的作用, 类似于 java 中的包名

id: 标识 sql 语句

resultType: 封装结果的全限定类名

- 定义对应的 pojo 类 Mapper, 并且修改 Configuration 类

```

package frame.pojo;

public class Mapper {

    private String sql;
    private String resultType;

    public String getSql() {
        return sql;
    }
}

```

```

public void setSql(String sql) {
    this.sql = sql;
}

public String getResultType() {
    return resultType;
}

public void setResultType(String resultType) {
    this.resultType = resultType;
}

@Override
public String toString() {
    return "Mapper{" +
        "sql='" + sql + '\'' +
        ", resultType='" + resultType + '\'' +
        '}';
}
}

```

```

import java.util.Map;

public class Configuration {

    private String driver;
    private String url;
    private String username;
    private String password;

    // 所有的sql信息也一次性初始化到Configuration对象中
    private Map<String, Mapper> mapperMap = new HashMap<>();

    public Map<String, Mapper> getMapperMap() {
        return mapperMap;
    }

    public void setMapperMap(Map<String, Mapper> mapperMap) {
        this.mapperMap = mapperMap;
    }
}

```

- 读取配置 sql 语句的 xml 文件，在数据源 xml 中配置<mappers>关联 sql 配置文件的路径，这样的话通过一个文件流将所有 xml 信息读入到应用当中（我们不可能提供很多个文件流入口）

```

<?xml version="1.0" encoding="UTF-8" ?>
<configuration>
  <environments default="development">
    <environment id="development">
      <transactionManager type="JDBC" />
      <dataSource type="POOLED">
        <property name="driver" value="com.mysql.jdbc.Driver" />
        <property name="url" value="jdbc:mysql://127.0.0.1:3306/mybatis?characterEncoding=utf8" />
        <property name="username" value="root" />
        <property name="password" value="root" />
      </dataSource>
    </environment>
  </environments>

  <mappers>
    <mapper resource="UserMapper.xml"/>
  </mappers>
</configuration>

```

- 修改 SqlSessionFactoryBuilder 类中读取 xml 文件的部分

```

package frame.factory;

import frame.pojo.Configuration;
import frame.pojo.Mapper;
import org.dom4j.Document;
import org.dom4j.DocumentException;
import org.dom4j.Element;
import org.dom4j.io.SAXReader;

import java.io.InputStream;
import java.util.List;

/**
 * 构建工厂，并在这里读取一次 xml，避免 xml 重复读取
 */

public class SqlSessionFactoryBuilder {

    /**
     * 构建工厂的入口，让用户传入配置文件的文件流
     * @param inputStream
     * @return
     */
    public SqlSessionFactory build(InputStream inputStream){
        SqlSessionFactory sqlSessionFactory = new SqlSessionFactory();
        // 读取数据源信息 xml 为 Configuration 对象
        Configuration configuration = loadXmlConfig(inputStream);
        sqlSessionFactory.setConfiguration(configuration);
        return sqlSessionFactory;
    }

    /**
     * 读取 xml 信息为 Configuration 对象
     * @param inputStream
     * @return
     */
}

```

```

*/
private Configuration loadXmlConfig(InputStream inputStream){
    Configuration configuration = new Configuration();
    SAXReader saxReader = new SAXReader();
    try {
        Document document = saxReader.read(inputStream);
        Element rootElement = document.getRootElement();
        List<Element> selectNodes = rootElement.selectNodes("//property");

        if(selectNodes != null && selectNodes.size() > 0) {
            for (int i = 0; i < selectNodes.size(); i++) {
                Element element = selectNodes.get(i);
                String name = element.attributeValue("name");
                String value = element.attributeValue("value");

                if("driver".equalsIgnoreCase(name)) {
                    configuration.setDriver(value);
                }
                if("url".equalsIgnoreCase(name)) {
                    configuration.setUrl(value);
                }
                if("username".equalsIgnoreCase(name)) {
                    configuration.setUsername(value);
                }
                if("password".equalsIgnoreCase(name)) {
                    configuration.setPassword(value);
                }
            }
        }

        List<Element> mappers = rootElement.selectNodes("//mapper");

        if(mappers != null && mappers.size() > 0) {
            for (int i = 0; i < mappers.size(); i++) {
                Element element = mappers.get(i);
                String mapperPath = element.attributeValue("resource");
                loadSqlXmlConfig(configuration,mapperPath);
            }
        }
    } catch (DocumentException e) {
        e.printStackTrace();
    }
}

```

```

        return configuration;
    }

    /**
     * 读取存放 sql 的 xml 文件
     * @param mapperPath
     */
    private void loadSqlXmlConfig(Configuration configuration,String mapperPath) {
        InputStream resourceAsStream =
this.getClass().getClassLoader().getResourceAsStream(mapperPath);
        SAXReader saxReader = new SAXReader();
        try {
            Document document = saxReader.read(resourceAsStream);
            Element rootElement = document.getRootElement();
            String namespace = rootElement.attributeValue("namespace");
            List<Element> selectNodes = rootElement.selectNodes("//select");
            if(selectNodes != null && selectNodes.size() > 0) {
                for (int i = 0; i < selectNodes.size(); i++) {
                    Element element = selectNodes.get(i);
                    String id = element.attributeValue("id");
                    String resultType = element.attributeValue("resultType");
                    String sql = element.getText();
                    Mapper mapper = new Mapper();
                    mapper.setSql(sql);
                    mapper.setResultType(resultType);
                    configuration.getMapperMap().put(namespace + "." +
id,mapper);
                }
            }
        } catch (DocumentException e) {
            e.printStackTrace();
        }
    }
}

```

(7) 步骤 7: 修改 SqlSessionImpl 中第二和第三个待办事项

```

package frame.core;

import java.util.List;

/**
 * 框架对外接口类
 */
public interface SqlSession {

```

```

    <T> List<T> selectList(String sqlId) throws Exception;
}

package frame.core;

import frame.pojo.Configuration;

import java.lang.reflect.Method;
import java.sql.*;
import java.util.ArrayList;
import java.util.List;

/**
 * 框架对外接口的实现类
 */
public class SqlSessionImpl implements SqlSession {

    private Configuration configuration;
    public void setConfiguration(Configuration configuration) {
        this.configuration = configuration;
    }

    @Override
    public <T> List<T> selectList(String sqlId) throws Exception {
        List<T> list = new ArrayList<>();

        Class.forName(configuration.getDriver());
        Connection connection = DriverManager.getConnection(configuration.getUrl(),
            configuration.getUsername(),configuration.getPassword());

        String sql = configuration.getMapperMap().get(sqlId).getSql();
        PreparedStatement preparedStatement = connection.prepareStatement(sql);
        ResultSet resultSet = preparedStatement.executeQuery();

        ResultSetMetaData metaData = resultSet.getMetaData();
        int columnCount = metaData.getColumnCount();

        List<String> columnLabels = new ArrayList<>();
        for(int i = 1; i <= columnCount; i++) {
            columnLabels.add(metaData.getColumnLabel(i));
        }

        // 需要知道结果集实体 Pojo 的全限定类名, 比如:com.mybatis.pojo.User, 然后

```

使用反射技术处理

```
String resultType = configuration.getMapperMap().get(sqlId).getResultType();
Class aClass = Class.forName(resultType);

Method[] methods = aClass.getMethods();
// 取代 User user = null;
Object obj = null;
while(resultSet.next()) {
    // 取代 user = new User();
    obj = aClass.newInstance();
    // 取代 user.setUsername(resultSet.getString("username"));
    for(String columnLabel : columnLabels) {
        for(Method method : methods) {
            if(method.getName().equalsIgnoreCase("set" + columnLabel)) {
                method.invoke(obj,resultSet.getObject(columnLabel));
            }
        }

        list.add((T) obj);
    }
}

resultSet.close();
preparedStatement.close();
connection.close();

return list;
}
}
```

步骤 8：将 SqlSessionFactoryBuilder 中读取 xml 构建 Configuration 对象的代码抽取为 XmlConfigBuilder 类，便于代码重用（不同功能的代码放到不同的 java 类中，各司其职）。

构建 Configuration 复杂对象的过程也是设计模式之构建者模式的一种体现。（构建者模式参见扩展知识部分）

- XmlConfigBuilder.java 类

```
package frame.utils;

import frame.pojo.Configuration;
import frame.pojo.Mapper;
import org.dom4j.Document;
import org.dom4j.DocumentException;
import org.dom4j.Element;
import org.dom4j.io.SAXReader;
```



```

import java.io.InputStream;
import java.util.List;

public class XmlConfigBuilder {

    /**
     * 读取 xml 信息为 Configuration 对象
     * @param inputStream
     * @return
     */
    public Configuration loadXmlConfig(InputStream inputStream){
        Configuration configuration = new Configuration();
        SAXReader saxReader = new SAXReader();
        try {
            Document document = saxReader.read(inputStream);
            Element rootElement = document.getRootElement();
            List<Element> selectNodes = rootElement.selectNodes("//property");

            if(selectNodes != null && selectNodes.size() > 0) {
                for (int i = 0; i < selectNodes.size(); i++) {
                    Element element = selectNodes.get(i);
                    String name = element.attributeValue("name");
                    String value = element.attributeValue("value");

                    if("driver".equalsIgnoreCase(name)) {
                        configuration.setDriver(value);
                    }
                    if("url".equalsIgnoreCase(name)) {
                        configuration.setUrl(value);
                    }
                    if("username".equalsIgnoreCase(name)) {
                        configuration.setUsername(value);
                    }
                    if("password".equalsIgnoreCase(name)) {
                        configuration.setPassword(value);
                    }
                }
            }
            List<Element> mappers = rootElement.selectNodes("//mapper");

            if(mappers != null && mappers.size() > 0) {
                for (int i = 0; i < mappers.size(); i++) {
                    Element element = mappers.get(i);

```

```

        String mapperPath = element.attributeValue("resource");
        loadSqlXmlConfig(configuration,mapperPath);
    }
}
} catch (DocumentException e) {
    e.printStackTrace();
}

return configuration;
}

/**
 * 读取存放 sql 的 xml 文件
 * @param mapperPath
 */
private void loadSqlXmlConfig(Configuration configuration,String mapperPath) {
    InputStream resourceAsStream =
this.getClass().getClassLoader().getResourceAsStream(mapperPath);
    SAXReader saxReader = new SAXReader();
    try {
        Document document = saxReader.read(resourceAsStream);
        Element rootElement = document.getRootElement();
        String namespace = rootElement.attributeValue("namespace");
        List<Element> selectNodes = rootElement.selectNodes("//select");
        if(selectNodes != null && selectNodes.size() > 0) {
            for (int i = 0; i < selectNodes.size(); i++) {
                Element element = selectNodes.get(i);
                String id = element.attributeValue("id");
                String resultType = element.attributeValue("resultType");
                String sql = element.getText();
                Mapper mapper = new Mapper();
                mapper.setSql(sql);
                mapper.setResultType(resultType);
                configuration.getMapperMap().put(namespace + "." + id,mapper);
            }
        }
    } catch (DocumentException e) {
        e.printStackTrace();
    }
}
}
}

```

修改 SqlSessionFactoryBuilder 类

```
package frame.factory;
```

```

import frame.pojo.Configuration;
import frame.utils.XmlConfigBuilder;

import java.io.InputStream;

/**
 * 构建工厂，并在这里读取一次 xml，避免 xml 重复读取
 */
public class SqlSessionFactoryBuilder {

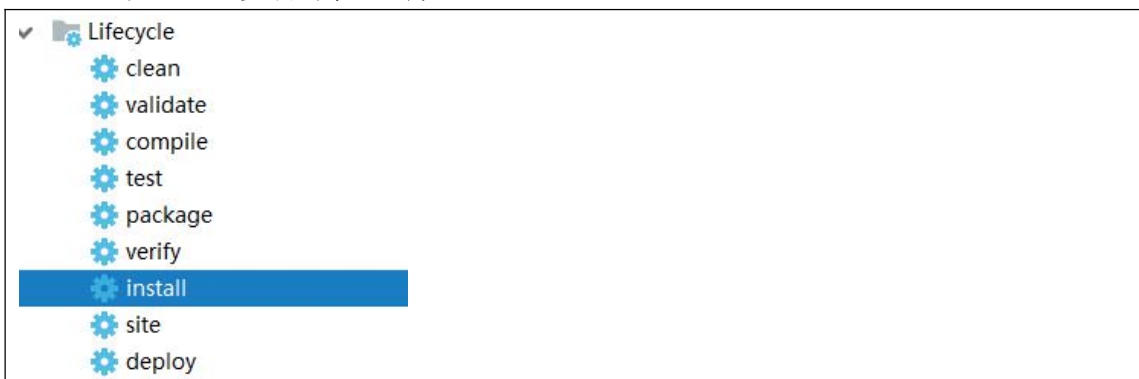
    /**
     * 构建工厂的入口，让用户传入配置文件的文件流
     * @param inputStream
     * @return
     */
    public SqlSessionFactory build(InputStream inputStream){
        SqlSessionFactory sqlSessionFactory = new SqlSessionFactory();
        // 读取数据源信息 xml 为 Configuration 对象
        XmlConfigBuilder xmlConfigBuilder = new XmlConfigBuilder();
        Configuration configuration = xmlConfigBuilder.loadXmlConfig(inputStream);
        sqlSessionFactory.setConfiguration(configuration);
        return sqlSessionFactory;
    }
}

```

打包测试

打包

- 双击 install，安装到本地仓库



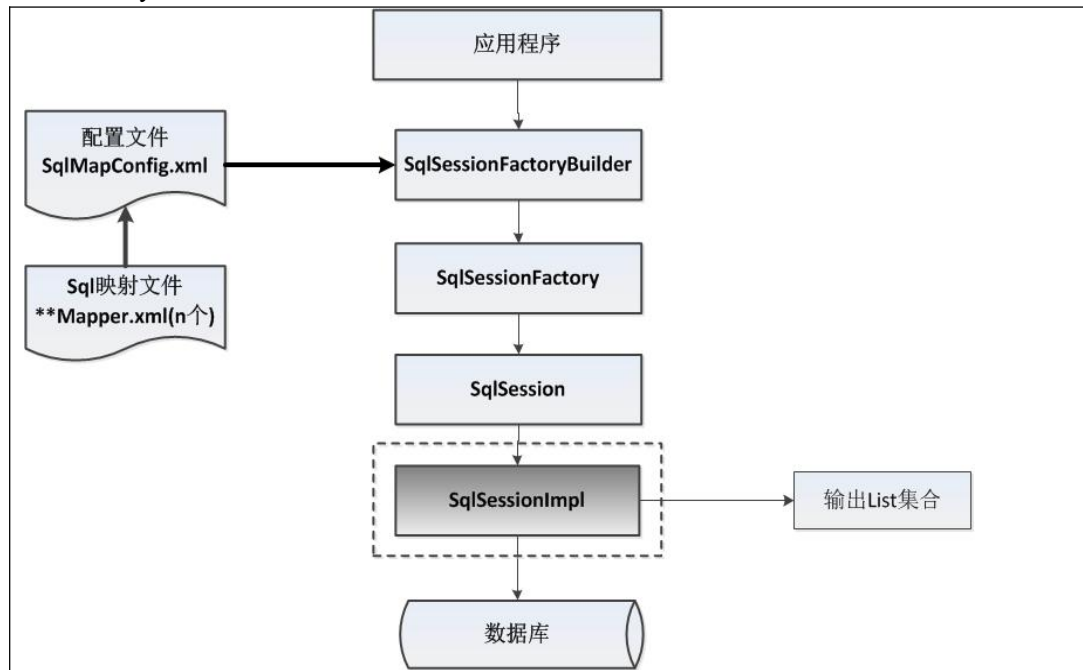
- 本地 Maven 仓库查看是否打包成功

测试

注意：新建一个工程进行测试，否则在同一个工程中的话引用的是自定义 Mybatis 框架的源码工程，而非打好的 jar 包。

自定义 Mybatis 框架小结

- 自定义 Mybatis 结构图



- 框架的开发使用流程
编写源代码—>打 jar 包—>框架使用者引入 jar 包—>按照框架的规范进行 XML 配置—>Java 程序中调用框架 API 实现功能
- 引入框架坐标即可，它所依赖的 dom4j 和 jaxen 都不需要我们引入，这是 Maven 的依赖传递
- 框架是基础知识的整合（本课程中的 jdbc、dom4j、xpath、反射、数据库元数据等）
- 框架往往都会用到设计模式（本课程扩展了工厂设计模式和构建者设计模式）
- 框架展示了代码的拆分思想，各司其职，便于维护

[掌握]Mybatis 框架快速入门

- 下载 Mybatis

[mybatis – MyBatis 3 | Introduction](#)

查看此网页的中文翻译, 请点击 [翻译此页](#)

What is **MyBatis**? **MyBatis** is a first class persistence framework with support for custom SQL, stored procedures and advanced mappings. **MyBatis** eliminates...

[www.mybatis.org/mybati...](#) - 百度快照

入门

安装

要使用 MyBatis, 只需将 `mybatis-x.x.x.jar` 文件置于 classpath 中即可。

如果使用 Maven 来构建项目, 则需将下面的 dependency 代码置于 pom.xml 文件中:

```
<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis</artifactId>
  <version>x.x.x</version>
</dependency>
```

我们本次课程使用 3.4.5 版本

- Mybatis 入门程序
查询 user 表全部记录, 以 list 集合形式返回
- 测试程序
 - 引入 jar

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
  </dependency>

  <dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis</artifactId>
    <version>3.4.5</version>
  </dependency>

  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.46</version>
  </dependency>

  <dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
```

```
</version>1.2.17</version>
</dependency>
</dependencies>
```

■ pojo 类 User.java

```
package com.mybatis.pojo;

import java.util.Date;

public class User {

    private Integer id;
    private String username;
    private String sex;
    private Date birthday;
    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 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;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }

    @Override
    public String toString() {
        return "User{" +
            "id=" + id +
            ", username='" + username + '\'' +
            ", sex='" + sex + '\'' +
            ", birthday=" + birthday +
            ", address='" + address + '\'' +
            '}';
    }
}

```

■ 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>
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC" />
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.jdbc.Driver" />
                <property name="url"
value="jdbc:mysql://127.0.0.1:3306/mybatis?characterEncoding=utf8" />
                <property name="username" value="root" />
                <property name="password" value="root" />
            </dataSource>
        </environment>
    </environments>

```

```

<mappers>
  <mapper resource="UserMapper.xml"></mapper>
</mappers>
</configuration>

```

■ UserMapper.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="test">
  <select id="queryUserList" resultType="com.mybatis.pojo.User">
    select * from user
  </select>
</mapper>

```

■ 测试程序

```

@Test
public void testMybatis() throws IOException {
    InputStream inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");
    SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new SqlSessionFactoryBuilder();
    SqlSessionFactory sqlSessionFactory = sqlSessionFactoryBuilder.build(inputStream);
    SqlSession sqlSession = sqlSessionFactory.openSession();
    List<User> list = sqlSession.selectList("test.queryUserList");
    if(list != null && list.size() > 0) {
        for (int i = 0; i < list.size(); i++) {
            User user = list.get(i);
            System.out.println(user);
        }
    }
    sqlSession.close();
}

```

[了解]扩展知识

设计模式

设计模式是前人代码经验的总结，是别人优秀的代码写法（写代码的套路），适当的场景使用设计模式提高代码的健壮性、可扩展和可维护性

● 工厂设计模式

- 工厂模式是一种比较常用的设计模式，是帮我们实例化对象的（即 new 对象的），所以以后 new 对象的时候你要考虑是否可以使用工厂模式
- 工厂模式下常用的有简单工厂模式和工厂方法模式两种（参考我提供给大家的 demo 工程）

- 构建者（build）模式
构建者模式，又称建造者模式，是将一个复杂对象的构建分为许多小对象的构建，最后在整合在一起的模式。
 - Builder 类：定义组装的对象包括哪些部分（硅谷大表哥）
 - Builder 实现类：具体组装各个小部分的类（具体干活员工）
 - Director 类：指导 Builder 实现类组装的类（员工的师傅，指导干活的人）
- 示例代码参考

>  00_DesignPatternDemo

元数据

数据表是用来存储我们业务数据的，而元数据是用来描述数据表的，比如这个表的表结构，有哪些字段等信息。本节课我们要知道查询结果集中有哪些数据项就可以通过元数据技术获取。

MetaData：元数据的意思

id	username	sex	birthday	address
1	张三	1	2018-07-10	北京
2	李四	1	2018-07-10	上海
3	王五	1	2018-07-10	广州
4	王六	1	2018-07-10	深圳
(Auto)	(NULL)	(NULL)	(NULL)	(NULL)

log4j 日志组件

Log4j 就是一种日志组件，可以把日志打印到控制台，也可以输出到文件

log4j 使用步骤（和框架使用类似）：

- 导入 log4j 的 jar 包


```
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.17</version>
</dependency>
```
- 在 classpath 下创建 log4j.properties 配置文件并进行配置（文件名称是固定为 log4j.properties）
- Java 代码中调用 API 输出日志，例如


```
import org.apache.log4j.Logger;

public class Demo {

    private static Logger logger = Logger.getLogger(Demo.class);

    public void runLog() {
        logger.debug("=====我是debug信息=====");
        logger.info("=====我是info信息=====");
        logger.warn("=====我是warn信息=====");
        logger.error("=====我是error信息=====");
        logger.fatal("=====我是fatal信息=====");
    }
}
```

- 示例代码参考

>  00_log4jDemo

注意：所有 Maven 工程指定为 JDK1.8 编译

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.1</version>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
        <encoding>UTF-8</encoding>
      </configuration>
    </plugin>
  </plugins>
</build>
```

[理解]关于三个对象

- SqlSessionFactoryBuilder: 创建工厂，生命周期短暂即可，类似于一个工具类
- SqlSessionFactory: 应该只有一个工厂对象即可。
- SqlSession: 和数据库通信的会话对象，底层对应 connection 连接。例如：假如多个线程公用一个 sqlSession 对象（connection），线程 A 操作了数据库尚未提交事务，线程 B 操作了数据库把这个 connection 的事务给提交了，会造成数据安全问题。所以一个线程应该有自己的一个 sqlSession 对象。
换言之，SqlSession 是线程不安全的

[掌握]Mybatis 入门级 CRUD 操作

功能需求：

基于已有数据表 user，使用 MyBatis 实现以下功能：

- 根据用户 id 查询一个用户
- 根据用户名称模糊查询用户列表
- 添加一个用户
- 根据用户 id 修改用户名
- 根据用户 id 删除用户

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
  </dependency>

  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.46</version>
  </dependency>
  <!--mybatis 的 jar 坐标-->
  <dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis</artifactId>
    <version>3.4.5</version>
  </dependency>
  <!--log4j 日志组件，要同时把兄弟 log4j.properties 引入-->
  <dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.1</version>
      <configuration>
```

```

        <source>1.8</source>
        <target>1.8</target>
        <encoding>utf-8</encoding>
    </configuration>
</plugin>
</plugins>
</build>

```

组件:

POJO

SqlMapConfig.xml

新增业务, 修改: **UserMapper.xml**

关联 SqlMapConfig 和 UserMapper

新增业务, 修改: **Java 程序调用** (测试程序)

根据用户 id 查询用户

● UserMapper.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 根标签, 映射的意思
    namespace: 命名空间, 分类管理的 sql 的作用
-->
<mapper namespace="test">

    <!--
        查询使用 select 标签
        parameterType: 传入的参数类型
        resultType: 封装的结果类型
    -->
    <select id="queryUserById" parameterType="Integer"
resultType="com.mybatis.pojo.User">
        <!--
            #{} 固定取参语法
            #{} 取参名称: 当 parameterType 传入简单数据类型的时候, 取参名称任意, 但是建议见名思意, 比如是 id 就#{id}
        -->
        select * from user where id=#{id}
    </select>
</mapper>

```

- 测试程序

```
/**
 * 测试用例(Use_case): 根据 id 查询用户
 */
@Test
public void testQueryUserById() throws IOException {
    InputStream inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");
    SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new SqlSessionFactoryBuilder();
    SqlSessionFactory sqlSessionFactory = sqlSessionFactoryBuilder.build(inputStream);
    SqlSession sqlSession = sqlSessionFactory.openSession();
    /**
     * 调用会话对象的 api 完成 crud
     * param1: 锁定 sql 的唯一标识符
     * param2: 传入的参数
     */
    User user = sqlSession.selectOne("test.queryUserById", 1);
    System.out.println(user);
    sqlSession.close();
    inputStream.close();
}
```

根据用户名模糊查询用户列表

- #{} 方式取参

- UserMapper.xml

```
<!--根据用户名模糊查询用户列表-->
<!--注意: 返回多条记录和返回单条记录, resultType 配置是一样的-->
<select id="queryUserByUsername" parameterType="String"
resultType="com.mybatis.pojo.User">
    <!--
    #{}底层原理: 预编译语句(防止 sql 注入, 性能提高)
    #{}对字符串参数自动添加单引号对
    -->
    select * from user where username like #{username}
</select>
```

- 测试程序

```
/**
 * 测试用例(Use_case): 根据用户名模糊查询用户列表
 */
```

```

@Test
public void testQueryUserByUsername() throws IOException {
    InputStream inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");
    SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new SqlSessionFactoryBuilder();
    SqlSessionFactory sqlSessionFactory = sqlSessionFactoryBuilder.build(inputStream);
    SqlSession sqlSession = sqlSessionFactory.openSession();
    // TooManyResultsException: Expected one result (or null) to be returned by
    // selectOne(), but found: 2
    List<User> userList = sqlSession.selectList("test.queryUserByUsername", "%
    王%");
    if (userList != null && userList.size() > 0) {
        for (int i = 0; i < userList.size(); i++) {
            User user = userList.get(i);
            System.out.println(user);
        }
    }
    sqlSession.close();
    inputStream.close();
}

```

- `${}` 方式取参

- UserMapper.xml

```

<!--
    ${}: 另外一种取参方式
    ${} 取参名称: 当 parameterType 传入简单数据类型的时候, 取参名称必须为
    value 字符串
    ${} 原理: 简单的字符串, 传什么参数拼接什么
    不会对字符串参数添加单引号对
-->

<!--
    #和$取参使用原则
    能用#不用$(从底层原理角度出发说的, #是预编译语句), 特殊情况使用$: 当你传
    入的参数是数据库对象 (比如表名动态传入), 或者是
    order by 的那个字段, 都不希望自动添加单引号对, 所以使用$
-->
<select id="queryUserByUsername" parameterType="String"
    resultType="com.mybatis.pojo.User">
    select * from user where username like ${value}
</select>

```

- 测试程序

```

/**
 * 测试用例(Use_case): 根据用户名模糊查询用户列表$取参形式
 */
@Test
public void testQueryUserByUsername$() throws IOException {
    InputStream inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");
    SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new SqlSessionFactoryBuilder();
    SqlSessionFactory sqlSessionFactory = sqlSessionFactoryBuilder.build(inputStream);
    SqlSession sqlSession = sqlSessionFactory.openSession();
    // TooManyResultsException: Expected one result (or null) to be returned by
    // selectOne(), but found: 2
    List<User> userList = sqlSession.selectList("test.queryUserByUsername$", "%
    王%");
    if (userList != null && userList.size() > 0) {
        for (int i = 0; i < userList.size(); i++) {
            User user = userList.get(i);
            System.out.println(user);
        }
    }
    sqlSession.close();
    inputStream.close();
}

```

小结

查询部分小结:

- selectOne, 当结果集有多条记录的时候, 抛出异常;
- #{}和\${}使用原则
能使用#不用\$, 特殊情况下使用\$, 特殊情况比如传入的参数是数据库对象(表名)或者 order by 的字段名时, 不希望自动添加单引号对;
- 返回结果集列表的时候, resultMap 和返回单个对象配置都一样;

实现添加用户(及 Mysql 自增主键返回)

```

<!--
    #{}和${}取参时, 当 parameterType 为 pojo 的时候, 取参名称为 pojo 的属性名
-->
<insert id="saveUser" parameterType="com.mybatis.pojo.User">

    <!--
        selectKey 用于查询主键数据
    -->

```

order: selectKey 中的 sql 在 insert 之前还是之后执行 AFTER/BEFORE
resultType: 查询出的主键的类型
keyProperty: 查询出的主键数据回显到 pojo 的哪个属性

```
-->
<selectKey order="AFTER" resultType="Integer" keyProperty="id">
    select last_insert_id()
</selectKey>
insert into user(username,sex,birthday,address)
    values(#{username},#{sex},#{birthday},#{address})
</insert>
```

自增主键返回：其实就是 mybatis 在执行完 insert 之后立马查询 last_insert_id()

根据用户 id 修改用户名

```
<update id="updateUsernameById" parameterType="com.mybatis.pojo.User">
    update user set username=#{username} where id=#{id}
</update>
```

根据用户 id 删除用户

```
<delete id="deleteUserById" parameterType="Integer">
    delete from user where id=#{id}
</delete>
```

Mybatis 解决原生 JDBC 操作数据库存在的问题

1、频繁创建、释放数据库连接造成系统资源浪费，影响系统性能。使用数据库连接池技术可以解决此问题。

解决：在 SqlMapConfig.xml 中配置数据连接池，使用连接池管理数据库连接。

2、Sql 语句写在代码中造成代码不易维护，实际应用中 Sql 变化的可能较大，Sql 变动需要改变 java 代码。

解决：将 Sql 语句配置在 XXXXmapper.xml 文件中与 Java 代码分离。

3、向 Sql 语句传参数麻烦，因为 Sql 语句的 where 条件不一定，可能多也可能少，占位符需要和参数一一对应（硬编码）。

解决：Mybatis 自动将 Java 对象映射至 Sql 语句，通过 statement 中的 parameterType 定义输入参数的类型。

4、对结果集解析麻烦（查询列硬编码），Sql 变化会导致解析代码变化，且解析前需要遍历，如果能将数据库记录封装成 Pojo 对象解析比较方便。

解决：Mybatis 自动将 Sql 执行结果映射至 Java 对象，通过 statement 中的 resultType 定义输出结果的类型。

[掌握]Mybatis 的两种 Dao 开发方式

使用 MyBatis 开发 DAO 的方式实现以下的功能：

- 根据用户 id 查询一个用户信息

原始 Dao 开发方式

定义接口（UserDao）

写实现类（UserDaoImpl）：在实现类中写 crud 那套程序

- 定义接口 UserDao

```
package com.mybatis.dao;

import com.mybatis.pojo.User;

/**
 * 原始 Dao 开发方式接口定义
 */
public interface UserDao {

    User queryUserById(Integer id) throws Exception;

}
```

- 写实现类

```
package com.mybatis.dao;

import com.mybatis.pojo.User;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;

public class UserDaoImpl implements UserDao {

    /**
     * 声明一个 SqlSessionFactory 变量，用于接收 service 层调用 dao 层时传入的工厂对象
     * 因为工厂对象全局唯一，对于全局唯一的对象，在分层的架构中，dao 层使用时候应该由
     * 外部传入
     */
}
```

```

    */
    private SqlSessionFactory sqlSessionFactory;
    public UserDaoImpl(SqlSessionFactory sqlSessionFactory) {
        this.sqlSessionFactory = sqlSessionFactory;
    }

    @Override
    public User queryUserById(Integer id) throws Exception {
        SqlSession sqlSession = sqlSessionFactory.openSession();
        User user = sqlSession.selectOne("test.queryUserById", id);
        sqlSession.close();
        return user;
    }
}

```

注意：分层其实是对原有代码的重新组织

- 思考

一个项目当中会有很多表，都对应 dao 接口和实现类，实现类就是做增删改查这些东西能否不写实现类只定义接口，让框架帮我们完成实现类的逻辑，那么就是 Mapper 动态代理的开发方式

Mapper 动态代理开发方式

- 定义一个 Mapper 接口，这个接口其实和我们 UserDao 接口是一样的

Mapper 接口中的接口方法被调用时候，最终要找到这个接口方法所使用的 sql 语句？那如何找到呢？

遵从下面 4 个规范即可

- 从 Mybatis 框架中拿到一个代理对象（代理的是这个 Mapper 接口），通过代理对象调用接口当中的方法完成业务
 - 传统 dao 开发方式中的实现类其实起了一个连接、承上启下的作用，连接了接口和 xml 映射文件，效果就是调用接口方法时能够找到 xml 映射文件
 - Mapper 动态代理开发遵从的规范
 - ◆ sql 映射文件的 namespace 必须和 mapper 接口的全限定类名保持一致
 - ◆ mapper 接口的接口方法名必须和 xml 中的 sql 语句 id 保持一致
 - ◆ mapper 接口的接口方法形参类型必须和 sql 语句的输入参数类型保持一致
 - ◆ mapper 接口的接口方法返回类型必须和 sql 语句的 resultType 保持一致

小结

原始 Dao 和 Mapper 动态代理开发在企业中都很常见，推荐使用 Mapper 动态代理开发模式

[掌握]全局配置文件 SqlMapConfig.xml 的使用说明

注意：子标签是有顺序要求的，因为使用了 dtd 校验

properties（属性）

引入外部属性文件

```
<!--  
    properties: 引入外部资源文件  
    resource:指向 classpath 下的资源文件路径  
    注意事项：优先级问题，外部引入的资源文件的配置项的优先级 高于 子标签  
    property 配置信息  
-->  
<properties resource="db.properties">  
    <property name="jdbc.username" value="root1"/>  
</properties>
```

typeAliases（类型别名）

Mybatis 默认支持的别名

别名	映射的类型
_byte	byte
_long	long
_short	short
_int	int
_integer	int
_double	double
_float	float
_boolean	boolean
string	String
byte	Byte
long	Long
short	Short
int	Integer
integer	Integer

double	Double
float	Float
boolean	Boolean
date	Date
decimal	BigDecimal
bigdecimal	BigDecimal
map	Map

注意：更多默认别名参见 mybatis 源码中的 **TypeAliasRegistry** 类

自定义别名

```

<!--
    typeAliases 定义别名，主要给 pojo 定义别名
-->
<typeAliases>
    <!--
        type: 给哪个类定义别名
        alias: 别名名称
    -->
    <!--<typeAlias type="com.mybatis.pojo.User" alias="user"/>-->

    <!--批量定义别名：package 指定要扫描的包路径
        注意：package 会扫描包及其子包下的所有类，不要类名相重复的情况，否则异常
    -->
    <package name="com.mybatis.pojo"/>
</typeAliases>

```

mappers（映射器）

引入 sql 语句的 xml 配置文件，把 sql 语句加载到内存当中

- `<mapper resource=""/>` 万能型选手，基础 crud、原始 dao 开发、mapper 动态代理开发都可以使用这种方式
- 针对 mapper 动态代理开发 dao 层，我们进行一个增加
mapper class 属性：执行 mapper 接口类的全限定类名
 前提要求：
 - 1、编译之后 mapper.xml 和 mapper.class 在同一个目录文件夹中
 - 2、两个文件名称相同

```

<!--
    规范：编译之后，UserMapper 接口类的 class 要和 UserMapper.xml 在同一个目录中，而且文件名相同
-->

```

```
<mapper class="com.mybatis.mapper.UserMapper"/>
```

<!--批量扫描 mapper 接口类，其实是扫描到包下面的一个个类，然后再按照<mapper class 的形式去加载

所以，package 扫描的要求（规范）和直接使用 class 属性是一样的

-->

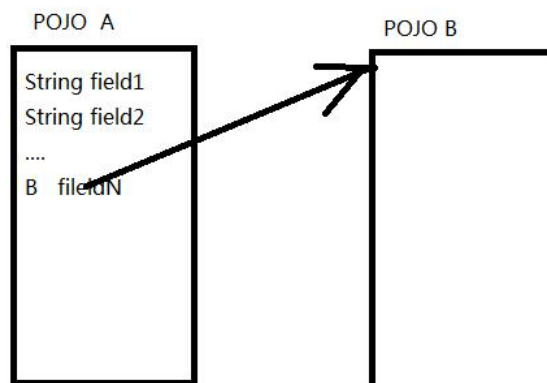
```
<package name="com.mybatis.mapper"/>
```

[掌握]Mybatis 的输入类型和结果类型

parameterType（输入类型）

- 传递简单类型
参考上面
- 传递 Pojo 对象
参考上面
- 传递 Pojo 包装对象
 - 需求：使用 POJO 包装对象，根据用户名模糊查询用户列表
 - 什么是 Pojo 包装对象

A中有属性的类型是另外一个POJO，那么A就叫做POJO包装类型，A的对象就叫做POJO包装对象



- 应用在什么场景

专利类型	<input type="checkbox"/> 发明公布 <input type="checkbox"/> 发明授权 <input type="checkbox"/> 实用新型 <input type="checkbox"/> 外观设计				使用
排序方式	公布公告日 ▼ 申请日 ▼				
公布公告	公布 (公告) 号	<input type="text"/>			
	公布 (公告) 日	<input type="text"/>	至	<input type="text"/>	
	专利文献出版日	<input type="text"/>	至	<input type="text"/>	
申请信息	申请号	<input type="text"/>			
	申请日	<input type="text"/>	至	<input type="text"/>	
	申请 (专利权) 人	<input type="text"/>	发明 (设计) 人	<input type="text"/>	
	地址	<input type="text"/>			
分类	分类号	<input type="text"/>			
文本	名称	<input type="text"/>			
	摘要/简要说明	<input type="text"/>			
专利代理	专利代理机构	<input type="text"/>	代理人	<input type="text"/>	
优先权、分案、生物保藏	优先权	<input type="text"/>	本国优先权	<input type="text"/>	
	分案原申请	<input type="text"/>	生物保藏	<input type="text"/>	
	其他信息	<input type="text"/>			

地址: <http://epub.sipo.gov.cn/gjcx.jsp>

resultType (输出类型)

- 输出简单类型
查询 user 表总记录数
- 输出 Pojo 对象
参考上面
- 输出 Pojo 列表
参考上面

resultMap (手动映射)

- 订单表 POJO 类

```
package com.mybatis.pojo;

import java.util.Date;

public class Orders {
    // 订单 id
    private int id;
    // 用户 id
    private Integer userId;
    // 订单号
    private String number;
    // 订单创建时间
    private Date createtime;
    // 备注
```

```
private String note;

public int getId() {
    return id;
}

public void setId(int id) {
    this.id = id;
}

public Integer getUserId() {
    return userId;
}

public void setUserId(Integer userId) {
    this.userId = userId;
}

public String getNumber() {
    return number;
}

public void setNumber(String number) {
    this.number = number;
}

public Date getCreatetime() {
    return createtime;
}

public void setCreatetime(Date createtime) {
    this.createtime = createtime;
}

public String getNote() {
    return note;
}

public void setNote(String note) {
    this.note = note;
}

@Override
public String toString() {
```

```

        return "Orders{" +
            "id=" + id +
            ", userId=" + userId +
            ", number=" + number + "\" +
            ", createtime=" + createtime +
            ", note=" + note + "\" +
            "}'";
    }
}

```

■ OrdersMapper.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.mybatis.mapper.OrdersMapper">

    <!--
        resultType: 自动映射，要求查询列名和 pojo 属性名保持一致
            1 最终封装的对象
            2 查询列名和 pojo 属性名一致
        当不一致的时候，要使用手动映射
        resultMap: 配置手动映射
    -->
    <select id="queryOrdersAllList" resultMap="ordersResultMap">
        select * from orders
    </select>

    <resultMap id="ordersResultMap" type="orders">
        <!-- 手动配置映射关系-->
        <!--
            主键字段不一致的时候使用<id> 标签配置
            普通字段不一致的时候使用<result> 标签配置
            property 严格区分大小写，要和 pojo 属性名称一致
        -->
        <id column="id" property="id"/>
        <result column="user_id" property="userId"/>
        <result column="number" property="number"/>
        <result column="createtime" property="createtime"/>
        <result column="note" property="note"/>
    </resultMap>

</mapper>

```

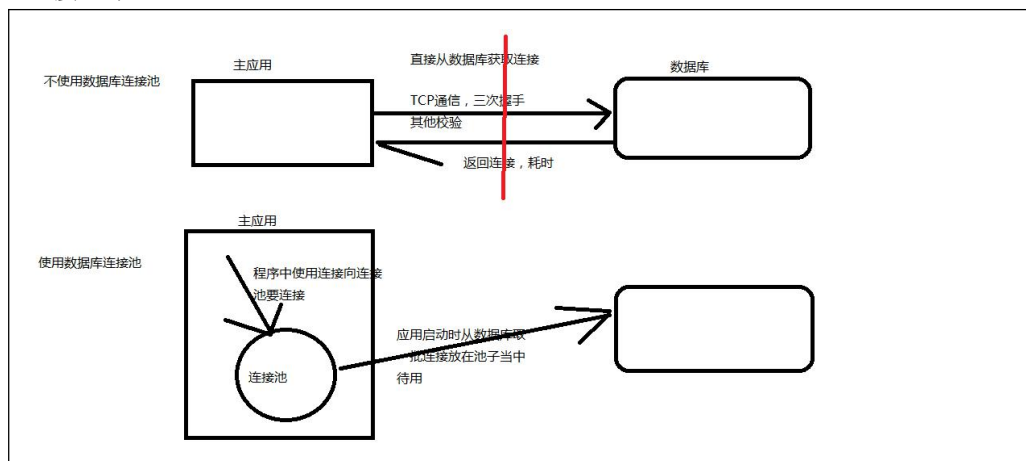

工程准备

新建一个 Maven 类型的 module：实现 Mybatis 第二天的保存用户功能，便于基于该功能的代码程序说明 Mybatis 数据库连接池和 Mybatis 事务控制。

[理解]Mybatis 连接池和事务控制

Mybatis 连接池

- POOLED 使用连接池
 - 连接池示意



- Mybatis 连接池初始化时机

```
private void environmentsElement(XNode context) throws Exception { context: "<env
    if (context != null) {
        if (environment == null) {
            environment = context.getStringAttribute(name: "default");
        }
        for (XNode child : context.getChildren()) { child: "<environment id="developme
            String id = child.getStringAttribute(name: "id"); id: "development"
            if (isSpecifiedEnvironment(id)) 判断使用哪个environment
                TransactionFactory txFactory = transactionManagerElement(child.evalNode("tr
                DataSourceFactory dsFactory = dataSourceElement(child.evalNode("dataSource
                DataSource dataSource = dsFactory.getDataSource(); 创建PooledDataSource数据源对象
                Environment.Builder environmentBuilder = new Environment.Builder(id) id:
                    .transactionFactory(txFactory) txFactory: JdbcTransactionFactory@1535
                    .dataSource(dataSource); dataSource: PooledDataSource@1632
                configuration.setEnvironment(environmentBuilder.build());
                把数据源对象放入configuration对象当中
            }
        }
    }
}
```

在构建工厂的时候创建 Mybatis 的数据库连接池

```
sqlSessionFactory = sqlSessionFactoryBuilder.build(inputStream);
```

■ 什么时候从连接池获取连接

getMapper 的时候还是真正调用 api 操作数据库的时候?

从连接池获取连接的时机: 真正操作数据库调用 api 的时候, 不是 getMapper 的时候

```
▼ this = {UserMapperTest@1665}
  ▼ sqlSessionFactory = {DefaultSqlSessionFactory@1670}
    ▼ configuration = {Configuration@1680}
      ▼ environment = {Environment@1681}
        > id = "development"
        > transactionFactory = {JdbcTransactionFactory@1715}
        ▼ dataSource = {PooledDataSource@1716}
          ▼ state = {PoolState@1717} \n====CONFIGURATION=====
            > dataSource = {PooledDataSource@1716}
            > idleConnections = {ArrayList@1721} size = 1
            > activeConnections = {ArrayList@1722} size = 0
```

● 其他

■ UNPOOLED: 不使用数据库连接池 (一般不使用)

■ JNDI:(前提你的 Mybatis 环境必须是 Web 应用) (了解)

◆ 什么是 JNDI

JNDI:java naming directory interface(java 命名目录接口), 它是一种该服务发布技术, 可以通过 JNDI 技术把数据源发布成一个服务, 那么客户端可以调用这个服务

◆ 为什么必须是 web 应用

因为支持 JNDI 技术的往往是 tomcat/weblogic/websphere 这些中间件才支持 JNDI 技术

◆ 如果在 Mybatis 当中用, 怎么用

参考附录

Mybatis 事务控制

type="JDBC": 使用 Jdbc 的事务管理机制, connection 的事务

Mybatis 默认把 Jdbc 的事务自动提交给关闭了

```
[DEBUG][2018/10/28 15:55:3114 ][org.apache.ibatis.datasource.pooled.PooledDataSource]
Created connection 428910174.
[DEBUG][2018/10/28 15:55:3114 ][org.apache.ibatis.transaction.jdbc.JdbcTransaction]
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@428910174]
[DEBUG][2018/10/28 15:55:3124 ][org.apache.ibatis.logging.jdbc.BaseJdbcLogger]
==> Preparing: insert into user(username,sex,birthday,address) values(?,?,?)
[DEBUG][2018/10/28 15:55:31134][org.apache.ibatis.logging.jdbc.BaseJdbcLogger]
```

注意: 考虑提交的数据量较小的时候使用自动提交, 如果数据量大影响了操作数据库的性能, 那么考虑分批次手动控制事务的提交

type="MANAGED": 什么都不做, 交给其他框架管理事务

[掌握]Mybatis 映射文件的 Sql 深入(即动态 Sql)

更优雅更方便的帮助我们拼接 sql 语句。
根据用户性别和用户名称查询用户列表

if 标签/where 标签

```
<select id="queryUserByWhere" parameterType="user" resultType="user">
  select * from user
  <!--
    where 1=1

    if(sex != null) {
      sql += " and sex='"
    }
  -->
  <!--where 标签会自动添加where关键字，并且去掉紧跟其后的第一个and或者or-->
  <where>
    <!--if 标签帮我们做判断，test 判断表达式真假-->
    <if test="sex != null and sex != ''">
      and sex=#{sex}
    </if>
    <if test="username != null and username != ''">
      and username like #{username}
    </if>
  </where>
```

sql 片段

```
<!--sql 标签，抽取公共的sql片段，便于sql语句共享-->
<sql id="commonSql">
  select id,username,sex,birthday,address
</sql>

<include refid="com.mybatis.mapper.OrdersMapper.commonSql"/> from user
```

引用 sql 片段使用 include 标签，refid 指向 sql 片段的 id，如果共享其他 mapper 文件当中的 sql 片段，只需要 refid 前面加上另外一个 mapper 映射文件的 namespace 即可

foreach 标签

```
delete from user where id in(1,2,3)
String sql = "delete from user where id in("
for(String s: list) {
  sql += s + ","
}
```

```
}
```

```
substring
```

```
sql += ")"
```

需求：根据多个 id 来查询用户列表

```
select * from user where id in(1,2,3)
```

- 形式一：直接传入 list 集合给 mybatis

```
<select id="queryUserByIdsList" parameterType="list" resultType="user">
  select id,username,sex,birthday,address from user
<!--
    foreach: 遍历拼接sql语句
    collection: 遍历的集合, 注意当直接传List集合, 除了知道是List集合外没有别的名字, 所以collection固定配置为
    open: 遍历拼接前干点啥 (拼接点什么字符串)
    close: 遍历结束后干点啥 (拼接点什么字符串)
    separator: 遍历拼接时候使用的分隔符
    item: 遍历到的具体元素值, 等同于下面for循环中的s
    for(String s : List) {
        s
    }
-->
<foreach collection="list" open=" where id in(" close=")" separator="," item="item">
  #{item}
</foreach>
</select>
```

```
@Test
public void testQueryUserByIdsList() {
    SqlSession sqlSession = sqlSessionFactory.openSession();
    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
    List<Integer> ids = new ArrayList<>();
    ids.add(1);
    ids.add(2);
    ids.add(3);
    List<User> userList = userMapper.queryUserByIdsList(ids);
    if(userList != null && userList.size() > 0) {
        for (int i = 0; i < userList.size(); i++) {
            User user = userList.get(i);
            System.out.println(user);
        }
    }
    sqlSession.close();
}
```

- 形式二：直接传入数组给 Mybatis

Mybatis 底层别名定义类

```
Decompiled .class file, bytecode version: 50.0 (Java 6)

28 public TypeAliasRegistry() {
29     this.registerAlias( alias: "string", String.class);
30     this.registerAlias( alias: "byte", Byte.class);
31     this.registerAlias( alias: "long", Long.class);
32     this.registerAlias( alias: "short", Short.class);
33     this.registerAlias( alias: "int", Integer.class);
34     this.registerAlias( alias: "integer", Integer.class);
35     this.registerAlias( alias: "double", Double.class);
36     this.registerAlias( alias: "float", Float.class);
37     this.registerAlias( alias: "boolean", Boolean.class);
38     this.registerAlias( alias: "byte[]", Byte[].class);
39     this.registerAlias( alias: "long[]", Long[].class);
40     this.registerAlias( alias: "short[]", Short[].class);
41     this.registerAlias( alias: "int[]", Integer[].class);
42     this.registerAlias( alias: "integer[]", Integer[].class);
43     this.registerAlias( alias: "double[]", Double[].class);
44     this.registerAlias( alias: "float[]", Float[].class);
45     this.registerAlias( alias: "boolean[]", Boolean[].class);
46     this.registerAlias( alias: "_byte", Byte.TYPE);
47     this.registerAlias( alias: "_long", Long.TYPE);
48 }
49 }
410 }
411 }
412 }
413 }
414 }
415 }
416 }
417 }
418 }
419 }
420 }
421 }
422 }
423 }
424 }
425 }
426 }
427 }
428 }
429 }
430 }
431 }
432 }
433 }
434 }
435 }
436 }
437 }
438 }
439 }
440 }
441 }
442 }
443 }
444 }
445 }
446 }
447 }
448 }
449 }
450 }
451 }
452 }
453 }
454 }
455 }
456 }
457 }
458 }
459 }
460 }
461 }
462 }
463 }
464 }
465 }
466 }
467 }
468 }
469 }
470 }
471 }
472 }
473 }
474 }
475 }
476 }
477 }
478 }
479 }
480 }
481 }
482 }
483 }
484 }
485 }
486 }
487 }
488 }
489 }
490 }
491 }
492 }
493 }
494 }
495 }
496 }
497 }
498 }
499 }
500 }
501 }
502 }
503 }
504 }
505 }
506 }
507 }
508 }
509 }
510 }
511 }
512 }
513 }
514 }
515 }
516 }
517 }
518 }
519 }
520 }
521 }
522 }
523 }
524 }
525 }
526 }
527 }
528 }
529 }
530 }
531 }
532 }
533 }
534 }
535 }
536 }
537 }
538 }
539 }
540 }
541 }
542 }
543 }
544 }
545 }
546 }
547 }
548 }
549 }
550 }
551 }
552 }
553 }
554 }
555 }
556 }
557 }
558 }
559 }
560 }
561 }
562 }
563 }
564 }
565 }
566 }
567 }
568 }
569 }
570 }
571 }
572 }
573 }
574 }
575 }
576 }
577 }
578 }
579 }
580 }
581 }
582 }
583 }
584 }
585 }
586 }
587 }
588 }
589 }
590 }
591 }
592 }
593 }
594 }
595 }
596 }
597 }
598 }
599 }
600 }
601 }
602 }
603 }
604 }
605 }
606 }
607 }
608 }
609 }
610 }
611 }
612 }
613 }
614 }
615 }
616 }
617 }
618 }
619 }
620 }
621 }
622 }
623 }
624 }
625 }
626 }
627 }
628 }
629 }
630 }
631 }
632 }
633 }
634 }
635 }
636 }
637 }
638 }
639 }
640 }
641 }
642 }
643 }
644 }
645 }
646 }
647 }
648 }
649 }
650 }
651 }
652 }
653 }
654 }
655 }
656 }
657 }
658 }
659 }
660 }
661 }
662 }
663 }
664 }
665 }
666 }
667 }
668 }
669 }
670 }
671 }
672 }
673 }
674 }
675 }
676 }
677 }
678 }
679 }
680 }
681 }
682 }
683 }
684 }
685 }
686 }
687 }
688 }
689 }
690 }
691 }
692 }
693 }
694 }
695 }
696 }
697 }
698 }
699 }
700 }
701 }
702 }
703 }
704 }
705 }
706 }
707 }
708 }
709 }
710 }
711 }
712 }
713 }
714 }
715 }
716 }
717 }
718 }
719 }
720 }
721 }
722 }
723 }
724 }
725 }
726 }
727 }
728 }
729 }
730 }
731 }
732 }
733 }
734 }
735 }
736 }
737 }
738 }
739 }
740 }
741 }
742 }
743 }
744 }
745 }
746 }
747 }
748 }
749 }
750 }
751 }
752 }
753 }
754 }
755 }
756 }
757 }
758 }
759 }
760 }
761 }
762 }
763 }
764 }
765 }
766 }
767 }
768 }
769 }
770 }
771 }
772 }
773 }
774 }
775 }
776 }
777 }
778 }
779 }
780 }
781 }
782 }
783 }
784 }
785 }
786 }
787 }
788 }
789 }
790 }
791 }
792 }
793 }
794 }
795 }
796 }
797 }
798 }
799 }
800 }
801 }
802 }
803 }
804 }
805 }
806 }
807 }
808 }
809 }
810 }
811 }
812 }
813 }
814 }
815 }
816 }
817 }
818 }
819 }
820 }
821 }
822 }
823 }
824 }
825 }
826 }
827 }
828 }
829 }
830 }
831 }
832 }
833 }
834 }
835 }
836 }
837 }
838 }
839 }
840 }
841 }
842 }
843 }
844 }
845 }
846 }
847 }
848 }
849 }
850 }
851 }
852 }
853 }
854 }
855 }
856 }
857 }
858 }
859 }
860 }
861 }
862 }
863 }
864 }
865 }
866 }
867 }
868 }
869 }
870 }
871 }
872 }
873 }
874 }
875 }
876 }
877 }
878 }
879 }
880 }
881 }
882 }
883 }
884 }
885 }
886 }
887 }
888 }
889 }
890 }
891 }
892 }
893 }
894 }
895 }
896 }
897 }
898 }
899 }
900 }
901 }
902 }
903 }
904 }
905 }
906 }
907 }
908 }
909 }
910 }
911 }
912 }
913 }
914 }
915 }
916 }
917 }
918 }
919 }
920 }
921 }
922 }
923 }
924 }
925 }
926 }
927 }
928 }
929 }
930 }
931 }
932 }
933 }
934 }
935 }
936 }
937 }
938 }
939 }
940 }
941 }
942 }
943 }
944 }
945 }
946 }
947 }
948 }
949 }
950 }
951 }
952 }
953 }
954 }
955 }
956 }
957 }
958 }
959 }
960 }
961 }
962 }
963 }
964 }
965 }
966 }
967 }
968 }
969 }
970 }
971 }
972 }
973 }
974 }
975 }
976 }
977 }
978 }
979 }
980 }
981 }
982 }
983 }
984 }
985 }
986 }
987 }
988 }
989 }
990 }
991 }
992 }
993 }
994 }
995 }
996 }
997 }
998 }
999 }
```

foreach 直接传入 *Array* 数组

```
-->
<select id="queryUserByIdsArray" parameterType="int[]" resultType="user">
    select id,username,sex,birthday,address from user
<!--
    foreach: 遍历拼接 sql 语句
    collection: 遍历的集合, 注意当直接传数组, 除了知道是数组外没有别的名
    字, 所以 collection 固定配置为 array
    open: 遍历拼接前干点啥 (拼接点什么字符串)
    close: 遍历结束后干点啥 (拼接点什么字符串)
    separator: 遍历拼接时候使用的分隔符
    item: 遍历到的具体元素值, 等同于下面 for 循环中的 s
    for(String s : list) {
        s
    }
-->
<foreach collection="array" open=" where id in(" close=")" separator=","
item="item">
    #{item}
</foreach>
</select>
```

- 形式三: 传入一个 pojo, pojo 中有 list 或者数组

```
<!--
    foreach 直接传入 pojo, pojo 中封装 list 或者数组
-->
<select id="queryUserByIdsQueryVo" parameterType="queryvo"
resultType="user">
    select id,username,sex,birthday,address from user
<!--
    foreach: 遍历拼接 sql 语句
    collection: 遍历的集合, 当传入 pojo 的时候, collection 配置为 pojo 当中集
    合的属性名
    如果不指定具体属性名, 那么 mybatis 是不知道要取 pojo 哪个
    属性的
    open: 遍历拼接前干点啥 (拼接点什么字符串)
    close: 遍历结束后干点啥 (拼接点什么字符串)
    separator: 遍历拼接时候使用的分隔符
    item: 遍历到的具体元素值, 等同于下面 for 循环中的 s
    for(String s : list) {
        s
    }
-->
```



```

<foreach collection="idsList" open=" where id in(" close=")" separator=","
item="item">
    #{item}
</foreach>
</select>

```

[掌握]Mybatis 的多表关联查询

多表：至少 2 张表

多表关系分析技巧：**从一条记录出发**，比如分析 A 表和 B 表的关系，就看 A 表的一条记录可以对应 B 表中几条记录，如果对应一条，从 A 到 B 表就是一对一的关系，如果对应多条就是一对多的关系。

多对多其实是双向的一对多

- 一对一

从订单表出发到用户表，一条订单记录只会对应用户表的一条记录，一对一

- 一对多

从用户表出发到订单表，一个用户记录可以对应订单表的多条记录，一对多

(通过**主外键**，一的一方是主表，多的一方是从表，外键在从表多的那一方)

- 多对多

用户和角色，一个用户可以有多个角色，一个角色也可以对应多个用户

通过中间表表达两个表之间的关系

a 表 b 表

id field.. id field..

中间表（往往两个字段即可）

aid bid

多表关联的 SQL 语句表达分析

- Sql 语句表达

- 笛卡尔积

- ◆ SELECT * FROM USER,orders

笛卡尔积会产生无用数据

user表的id

id	username	sex	age	city	user_id	number	createtime	note
1	张三	1	2018-07-10	北京	3	31000010	2015-02-04 13:22:35	(NULL)
1	张三	1	2018-07-10	北京	4	31000011	2015-02-03 13:22:41	(NULL)
1	张三	1	2018-07-10	北京	5	41000012	2015-02-12 16:13:23	(NULL)
2	李四	1	2018-07-10	上海	3	31000010	2015-02-04 13:22:35	(NULL)
2	李四	1	2018-07-10	上海	4	31000011	2015-02-03 13:22:41	(NULL)
2	李四	1	2018-07-10	上海	5	41000012	2015-02-12 16:13:23	(NULL)
3	王五	1	2018-07-10	广州	3	31000010	2015-02-04 13:22:35	(NULL)
3	王五	1	2018-07-10	广州	4	31000011	2015-02-03 13:22:41	(NULL)
3	王五	1	2018-07-10	广州	5	41000012	2015-02-12 16:13:23	(NULL)
4	王六	1	2018-07-10	深圳	3	31000010	2015-02-04 13:22:35	(NULL)
4	王六	1	2018-07-10	深圳	4	31000011	2015-02-03 13:22:41	(NULL)
4	王六	1	2018-07-10	深圳	5	41000012	2015-02-12 16:13:23	(NULL)
30	张飞	1	2018-10-28	河北涿州	3	31000010	2015-02-04 13:22:35	(NULL)
30	张飞	1	2018-10-28	河北涿州	4	31000011	2015-02-03 13:22:41	(NULL)
30	张飞	1	2018-10-28	河北涿州	5	41000012	2015-02-12 16:13:23	(NULL)

SELECT * FROM USER u,orders o WHERE u.id=o.user_id(笛卡尔积进一步筛选)

■ 关联查询

- ◆ 内关联 inner join on (和笛卡尔积进一步筛选效果一样)

SELECT * FROM USER u INNER JOIN orders o ON u.id=o.user_id

- ◆ 外连接(outer join on) (使用较多)

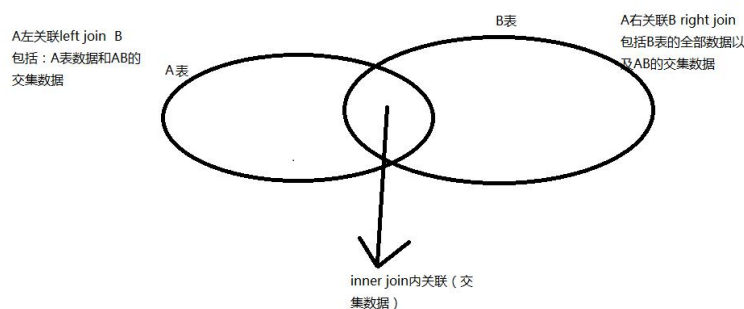
左外连接(left join) left join 左边的表是基础表

SELECT * FROM USER u LEFT JOIN orders o ON u.id=o.user_id

右外连接(right join) right join 右边的表是基础表

SELECT * FROM USER u RIGHT JOIN orders o ON u.id=o.user_id

● 内关联和外关联分析



一对一查询

Mybatis 帮我们做的事情:

- 1、SQL 语句是用户自己写, Mybatis 框架只是帮我们执行而已
- 2、封装结果集

需求: 查询订单表全部数据, 关联查询出订单对应的用户数据 (username address)

● Mapper 映射文件

```
<select id="queryOrdersUser" resultMap="ordersUserResultMap">
    SELECT
        o.`id`,
        o.`user_id`,
        o.`number`,
        o.`createtime`,
        o.`note`,
        u.`username`,
        u.`address`
    FROM
        orders o
        LEFT JOIN USER u
            ON o.`user_id` = u.`id`
</select>
```

```

<resultMap id="ordersUserResultMap" type="orders">
    <!--两部分数据： 订单+用户-->
    <id column="id" property="id"/>
    <result column="user_id" property="userId"/>
    <result column="number" property="number"/>
    <result column="createtime" property="createtime"/>
    <result column="note" property="note"/>

    <!--association： 关联的意思,用于一对一关联封装数据
        property： 一对一关联后封装成的数据所对应的属性名
        javaType： 属性的类型
    -->
    <association property="user" javaType="user">
        <id column="user_id" property="id"/>
        <result column="username" property="username"/>
        <result column="address" property="address"/>
    </association>
</resultMap>

```

- 测试程序

```

@Test
public void testQueryOrdersUser() {
    SqlSession sqlSession = sqlSessionFactory.openSession();
    OrdersMapper ordersMapper = sqlSession.getMapper(OrdersMapper.class);
    List<Orders> ordersList = ordersMapper.queryOrdersUser();
    if(ordersList != null && ordersList.size() > 0) {
        for (int i = 0; i < ordersList.size(); i++) {
            Orders orders = ordersList.get(i);
            System.out.println(orders);
        }
    }
    sqlSession.close();
}

```

一对多查询

需求：查询全部用户数据，关联查询出订单数据

- Mapper 映射文件

```

<!-- 一对多用例 -->
<select id="queryUserOrders" resultMap="userOrdersResultmap">
    SELECT
        u.`id`,

```



```

        u.`username`,
        u.`sex`,
        u.`birthday`,
        u.`address`,
        o.`id` oid,
        o.`user_id`,
        o.`number`,
        o.`createtime`,
        o.`note`
FROM
    USER u
    LEFT JOIN orders o
        ON u.`id` = o.`user_id`
</select>

<resultMap id="userOrdersResultmap" type="user">
    <!--两部分数据： 用户+ 订单-->
    <id column="id" property="id"/>
    <result column="username" property="username"/>
    <result column="sex" property="sex"/>
    <result column="birthday" property="birthday"/>
    <result column="address" property="address"/>

    <!--一对多使用 collection 标签
        property: 封装成对应的属性的属性名
        ofType: 因为此时属性是一个 list 集合， collection 已经表明了集合的意思， 最
        重要的指定 list 里面的泛型类型，
        所以使用 ofType
    -->
    <collection property="ordersList" ofType="orders">
        <id column="oid" property="id"/>
        <result column="user_id" property="userId"/>
        <result column="number" property="number"/>
        <result column="createtime" property="createtime"/>
        <result column="note" property="note"/>
    </collection>
</resultMap>

```

多对多查询

多对多关系分析

多对多是双向的一对多

实现 Role 到 User 的一对多

- Mapper 映射文件

```
<select id="queryRoleUsers" resultMap="roleUsersResultMap">
    SELECT
        r.rid,
        r.rname,
        r.rdesc,
        u.`username`,
        u.`address`
    FROM
        role r
        LEFT JOIN user_role ur
            ON r.`RID` = ur.`RID`
        LEFT JOIN USER u
            ON ur.`UID` = u.`id`
</select>

<!-- 多对多拆分成双向的一对多-->
<resultMap id="roleUsersResultMap" type="role">
    <id column="rid" property="rid"/>
    <result column="rname" property="rname"/>
    <result column="rdesc" property="rdesc"/>

    <collection property="userList" ofType="user">
        <result column="username" property="username"/>
        <result column="address" property="address"/>
    </collection>
</resultMap>
```

实现 User 到 Role 的一对多

下去练习下

[了解]扩展知识

Tomcat 配置 JNDI 数据源（了解）

- 第一步：将数据库驱动程序(jar 包)放到 tomcat 安装目录下的 common\lib 文件夹下



- 第二步：在 Tomcat 的 conf/context.xml 文件中进行 jndi 数据源服务配置

```

the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
-->
<!-- The contents of this file will be loaded for each web application -->
<Context>

    <!-- Default set of monitored resources -->
    <WatchedResource>WEB-INF/web.xml</WatchedResource>

    <!-- Uncomment this to disable session persistence across Tomcat restarts -->
    <!--
    <Manager pathname="" />
    -->

    <!-- Uncomment this to enable Comet connection tacking (provides events
    on session expiration as well as webapp lifecycle) -->
    <!--
    <Valve className="org.apache.catalina.valves.EventValve" />
    -->

    <Resource name="jndi/mybatis"
        auth="Container"
        type="javax.sql.DataSource"
        driverClassName="com.mysql.jdbc.Driver"
        url="jdbc:mysql://127.0.0.1:3306/mybatis?characterEncoding=utf8"
        username="root"
        password="root"
        maxActive="20"
        maxIdle="10"
        maxWait="10000"/>
</Context>

```

在tomcat中配置JNDI数据源服务

name: 在 JNDI 中叫做目录名，等同于服务名，此处的 jndi/mybatis 是自定义的，往往以/连接前后字符串即可。

auth 和 type 是固定的，其他都是数据库连接池的具体配置信息

- 第三步：在自己 web 项目的 web.xml 中引用 Jndi 数据源服务

```

<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd" >

<web-app>
    <display-name>Archetype Created Web Application</display-name>

    <resource-ref>
        <res-ref-name>jndi/mybatis</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
    </resource-ref>
</web-app>

```

- 第四步：在自己 web 项目的 Mybatis 配置文件中使用

```
<environment id="development">
  <transactionManager type="JDBC" />
  <dataSource type="JNDI">
    <property name="data_source" value="java:comp/env/jndi/mybatis"/>
  </dataSource>
</environment>
```

配置 data_source 属性，指向你的数据源引用，java:comp/env/jndi/mybatis 中红色部分是固定的，绿色部分是你自己定义的目录名（服务名）

工程准备

使用 Mapper 动态代理的方式实现：

- 根据用户 id 查询用户
- 查询全部订单数据并关联查询出用户数据（一对一）
- 查询全部用户数据并关联查询出订单数据（一对多）

便于基于代码程序讲解 Mybatis 延迟加载和缓存。

[掌握]Mybatis 延迟加载策略

关联对象：数据库层次用户和订单是关联表，在对象层次订单对象和用户对象就叫做关联对象

什么是延迟加载？

在使用关联对象的过程中

比如使用了 Orders 对象，但是我只是获取 Orders 的订单数据尚未获取 user 的数据，但是依然查询了 user 表（这不是延迟）

我 orders.getNumber()你不要查 user 表的数据，当我 orders.getUser()你再去查 user 表

关联查询的关联对象，在较多的场合中并不需要的时候，只在少部分场合需要关联的那个对象数据，可以考虑使用延迟加载

Mybatis 中怎么实现延迟加载？

步骤 1：原来关联 sql 语句要拆分

- 1) 第一条 sql 查询基础表
- 2) 第二条 sql 根据基础表的条件查询关联表

步骤 2：让两条 sql 自动产生联系，通过<association column="" select=""/> 或者<collection

column="" select =""/>

- 全局延迟加载开关配置（一对一和一对多都需要开启）

```
<!-- 开启延迟加载 -->
<setting name="lazyLoadingEnabled" value="true" />
<!-- 关闭立即加载 -->
<setting name="aggressiveLazyLoading" value="false" />
<!-- 设定 toString 等方法延迟加载 -->
<setting name="lazyLoadTriggerMethods" value="true" />
```

- 一对一关联查询的延迟加载

```
<select id="queryOrdersWithUser" resultMap="ordersUserResultMap">
    SELECT
        o.`id`,
        o.`user_id`,
        o.`number`,
        o.`createtime`,
        o.`note`
    FROM
        orders o
</select>
```

<!--

延迟加载分析

1、原来关联查询的 sql 语句必然要拆分，如果不拆分，那肯定要执行关联查询，两个表用不用都要查

2、sql=sql1+sql2 拆分之后，要让这两个 sql 自动产生联系

sql1:select id,user_id,number,createtime,note from orders

sql2:select * from user u where id=user_id

-->

```
<resultMap id="ordersUserResultMap" type="orders">
    <id column="id" property="id"/>
    <result column="user_id" property="userId"/>
    <result column="number" property="number"/>
    <result column="createtime" property="createtime"/>
    <result column="note" property="note"/>
    <association property="user" javaType="user" column="user_id"
select="com.mybatis.mapper.UserMapper.queryUserById">
    </association>
</resultMap>
```

```
<select id="queryUserById" parameterType="int" resultType="user">
    select id,username,sex,birthday,address from user where id=#{user_id}
```

```
</select>
```

- 一对多关联查询的延迟加载

```
<select id="queryUserWithOrders" resultMap="userOrdersResultMap">
```

```
    SELECT
        u.`id`,
        u.`username`,
        u.`sex`,
        u.`birthday`,
        u.`address`
```

```
    FROM
        USER u
```

```
</select>
```

```
<resultMap id="userOrdersResultMap" type="user">
```

```
    <id column="id" property="id"/>
```

```
    <result column="username" property="username"/>
```

```
    <result column="sex" property="sex"/>
```

```
    <result column="birthday" property="birthday"/>
```

```
    <result column="address" property="address"/>
```

```
    <collection property="ordersList" ofType="orders" column="id"
        select="com.mybatis.mapper.OrdersMapper.queryOrdersByUserId">
    </collection>
```

```
</resultMap>
```

<!-- 拆分的第二条 sql 语句-->

```
<select id="queryOrdersByUserId" parameterType="int" resultType="orders">
```

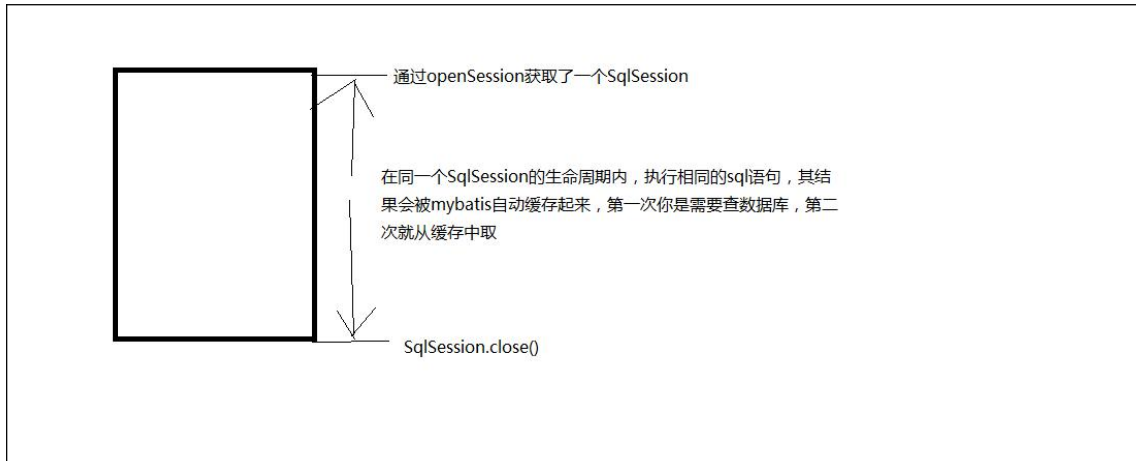
```
    select  o.id,o.user_id,o.number,o.createtime,o.note      from  orders  o      where
    o.user_id=#{id}
```

```
</select>
```

[了解]Mybatis 缓存

Mybatis 的一级缓存

Mybatis 的一级缓存是 SqlSession 级别的缓存，是默认开启的

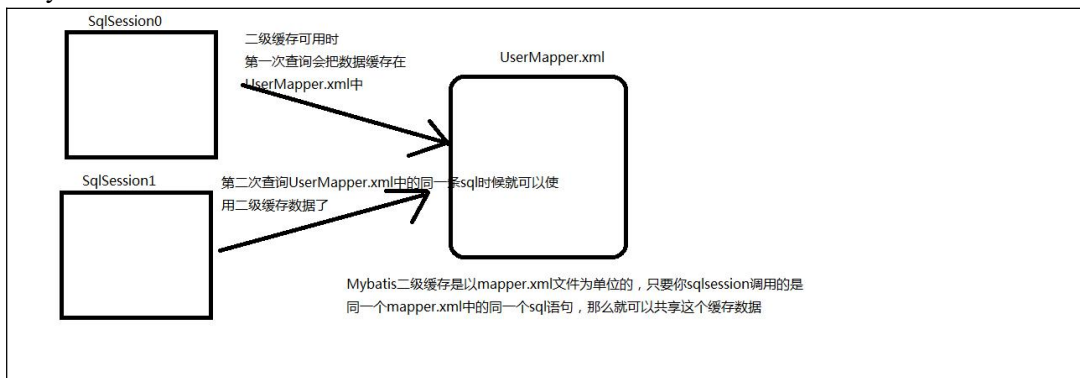


注意：以下情形缓存会失效（被清理掉）：

- sqlSession 关闭
- sqlSession 提交事务：意味着可能是一个增删改的动作，数据表数据产生变化，那么这个时候 Mybatis 就会把这个 sqlSession 已有的一级缓存给清理掉

Mybatis 的二级缓存

● Mybatis 二级缓存机制



● 怎么用 Mybatis 的二级缓存

- 开启二级缓存开关
 - 第一个开关：需要在 SqlMapConfig.xml 中开启二级缓存 **总开关**
 - 第二个开关：需要在使用二级缓存的 mapper.xml 中开启（因为二级缓存是 mapper 级别的）
- Pojo 实现序列化


```
public class User implements Serializable{

    private Integer id;
    private String username;
    private String sex;
    private Date birthday;
    private String address;
```

● 结果

```
[DEBUG][2018/09/23 10:16:28822][org.apache.ibatis.datasource.pooled.PooledDataSource.pushConne
Returned connection 838411509 to pool.
[DEBUG][2018/09/23 10:16:2966 ][org.apache.ibatis.cache.decorators.LoggingCache.getObject(Logg
Cache Hit Ratio [com.itheima.mybatis.mapper.UserMapper]: 0.5
user1===:User{id=1, username='张三', sex='1', birthday=Tue Jul 10 00:00:00 CST 2018, address='
```

ratio: 命中率的意思，两次都去查询了二级缓存，但是第一次的时候缓存中没有数据，所以没有命中，第二次查询的时候缓存中已经有数据了，所以命中率是 0.5

- Mybatis 二级缓存的清理时机：当这个 mapper.xml 中执行了非 select 语句的时候，整个 Mapper 的缓存全部清除掉。

注意：避免使用二级缓存，因为二级缓存带来的好处远远比不上他所隐藏的危害

- 如果你针对 user 表的 sql 操作并不是全部都在 UserMapper.xml 中，使用缓存的结果可能会不正确。

- 多表操作一定不能使用缓存

为什么不能？

首先不管多表操作写到那个 Mapper.xml 下，都会存在某个表不在这个 Mapper 下的情况。

例如两个表：role 和 user_role，如果我想查询出某个用户的全部角色 role，就一定会涉及到多表的操作

```
<select id="selectUserRoles" resultType="UserRoleVO">
    select * from user_role a,role b where a.roleid = b.roleid and a.userid = #{userid}
</select>
```

像上面这个查询，你会写到那个 xml 中呢？？

不管是写到 RoleMapper.xml 还是 UserRoleMapper.xml，或者是一个独立的 XxxMapper.xml 中。如果使用了二级缓存，都会导致上面这个查询结果可能不正确。

如果你正好修改了这个用户的角色，上面这个查询使用缓存的时候结果就是错的。

[掌握]Mybatis 的注解开发

效果：没有写 sql 语句的 xml 了

- 基础的 crud（基础 crud 怎么使用注解）
- 关联查询延迟加载（怎么使用注解完成）
- 动态 sql 的注解方式

原则：把原来 xml 里面的内容都对应的放到注解上即可

注解开发不需要引入额外的 jar 包，和原来一致即可

注解添加在接口类的方法上

使用注解实现基本的 CRUD

```
@Insert("insert      into      user(username,sex,birthday,address)      values
({username},{sex},{birthday},{address})")

/*回显 mysql 自增主键 id
  等同于原来我们 xml 中使用<selectKey/>, 一一对应过来即可
*/
@SelectKey(statement = "select last_insert_id()",before = false,keyProperty = "id",resultType
= Integer.class)
void saveUser(User user);

@Update("update user set username=#{username} where id=#{id}")
void updateUserNameById(User user);

@Delete("delete from user where id=#{id}")
void deleteUserById(Integer id);

@Select("select id,username,sex,birthday,address from user where id=#{id}")
User queryUserById(Integer id);

@Select("select id,username,sex,birthday,address from user")
List<User> queryUserList();

@Select("<script>select id,username,sex,birthday,address from user" +
    "<where>" +
    "<if test=\"sex != null and sex != \">" +
    " and sex=#{sex}" +
    "</if>" +
    "<if test=\"username != null and username != \">" +
    " and username like #{username}" +
    "</if>" +
    "</where></script>")
List<User> queryUserByWhere(User user);

@Select("<script>select id,username,sex,birthday,address from user" +
    "<foreach collection=\"list\" open=\" where id in(\" close=\\\"\\\" separator=\\\",\\\"
item=\\\"item\\\">" +
    "#{item}" +
```

```
"</foreach></script>")
List<User> queryUserByIdsList(List<Integer> ids);
```

使用注解实现复杂关系映射开发

使用注解实现一对一查询

```
/*
 * 对应关系
 * @Results 对应 xml 中的 resultMap
 * @Result 对应 xml 中的 id/result
 * id: true/false, 是否是主键, 默认 false
 * column: 查询列名
 * property: pojo 属性名
 * association: 在注解中使用@Result+one=@One 的方式去表达
 * fetchType: 延迟加载类型, 默认是延迟加载, eager: 积极加载
 */

@Results({
    @Result(id = true,column = "id",property = "id"),
    @Result(column = "user_id",property = "userId"),
    @Result(column = "number",property = "number"),
    @Result(column = "createtime",property = "createtime"),
    @Result(column = "note",property = "note"),
    @Result(property = "user",column = "user_id",javaType =
User.class,one=@One(select
"com.mybatis.mapper.UserMapper.queryUserByUserid",fetchType = FetchType.LAZY))
})
List<Orders> queryOrdersUser();

@Select("select id,username,sex,birthday,address from user where id=#{user_id}")
User queryUserByUserid(Integer id);
```

使用注解实现一对多查询

- mapper 接口

```
/*
 * 对应关系
 * @Results 对应 xml 中的 resultMap
```

- * @Result 对应 xml 中的 id/result
- * id: true/false, 是否是主键, 默认 false
- * column: 查询列名
- * property: pojo 属性名
- * collection: 在注解中使用@Result+many=@Many 的方式去表达
- * xml 中的 ofType 在注解中使用 javaType 表达=List.class
- * fetchType: 延迟加载类型, 默认是延迟加载, eager: 积极加载
- * */

```

@Select("select id,username,sex,birthday,address from user")
@Results({
    @Result(id = true,column = "id",property = "id"),
    @Result(column = "username",property = "username"),
    @Result(column="sex",property="sex"),
    @Result(column="birthday",property="birthday"),
    @Result(column="address",property="address"),
    @Result( property="ordersList",javaType = List.class,column="id",
        many
        =
@Many(select="com.mybatis.mapper.OrdersMapper.queryOrdersByUserid",fetchType
= FetchType.LAZY))
})
List<User> queryUserOrders();

```

```

@Select("select id,number from orders where user_id=#{userId}")
List<Orders> queryOrdersByUserid(Integer userId);

```