MyBatis

2021-03-08 10:35:56

mybatis 就是通过配置xml，然后在有个数据库类（user），写个dao接口，和mybatis里面方法对应，最后就在test类中直接创建各种东西后，，开始调用接口方法就好啦，，CURD，，

2021-03-02 09:47:36

1.1 原始jdbc操作（插入数据）

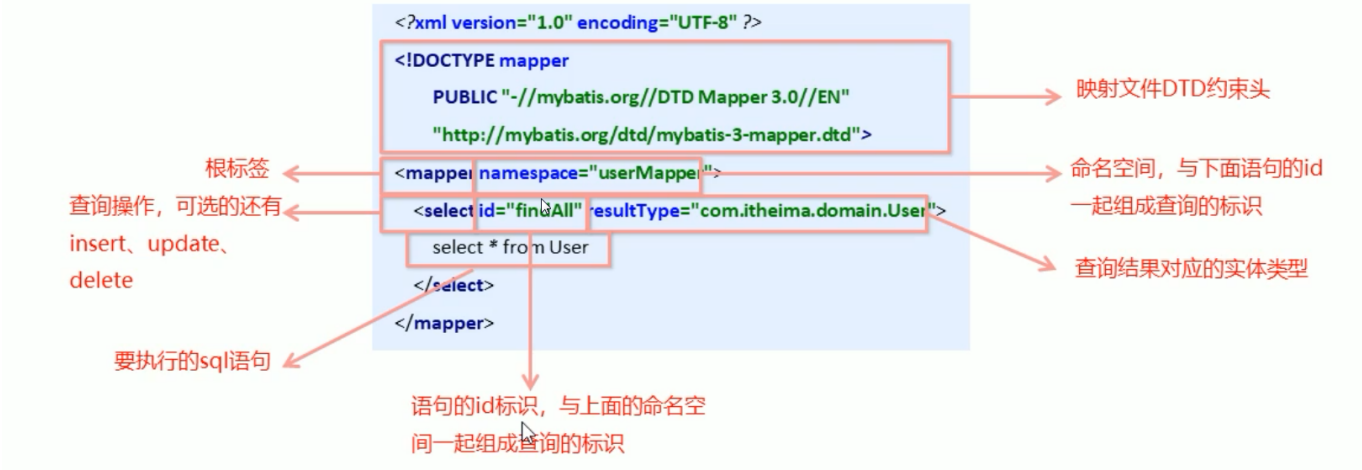
```
//模拟实体对象
User user = new User();
user.setId(2);
user.setUsername("tom");
user.setPassword("lucy");

//注册驱动
Class.forName("com.mysql.jdbc.Driver");
//获得连接
Connection connection = DriverManager.getConnection( url: "jdbc:mysql:///test", user: "root", password: "root");
//获得statement
PreparedStatement statement = connection.prepareStatement( sql: "insert into user(id,username,password) values(?,?,?)");
//设置占位符参数
statement.setInt( parameterIndex: 1,user.getId());
statement.setString( parameterIndex: 2,user.getUsername());
statement.setString( parameterIndex: 3,user.getPassword());
//执行更新操作
statement.executeUpdate();
//释放资源
statement.close();
connection.close();
```

UserMapper.xml × sqlMapConfig.xml ×

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN" "http://mybatis.org/dtd/mybatis-
<configuration>

    <!--数据源环境-->
    <environments default="developement">
        <environment id="developement">
            <transactionManager type="JDBC"></transactionManager>
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.jdbc.Driver"/>
                <property name="url" value="jdbc:mysql://localhost:3306/test"/>
                <property name="username" value="root"/>
                <property name="password" value="root"/>
            </dataSource>
        </environment>
    </environments>
```

2.3 编写测试代码

```java
//加载核心配置文件
InputStream resourceAsStream = Resources.getResourceAsStream("SqlMapConfig.xml");
//获得sqlSession工厂对象
SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(resourceAsStream);
//获得sqlSession对象
SqlSession sqlSession = sqlSessionFactory.openSession();
//执行sql语句
List<User> userList = sqlSession.selectList("userMapper.findAll");
//打印结果
System.out.println(userList);
//释放资源
sqlSession.close();
```



```java
//模拟user对象
User user = new User();
user.setUsername("tom");
user.setPassword("abc");

//获得核心配置文件
InputStream resourceAsStream = Resources.getResourceAsStream("sqlMapConfig.xml");
//获得session工厂对象
SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(resourceAsStream);
//获得session回话对象
SqlSession sqlSession = sqlSessionFactory.openSession();
//执行操作  参数：namespace+id
sqlSession.insert( s: "userMapper.save",user);

//mybatis执行更新操作  提交事务
sqlSession.commit();
```

4.1 MyBatis的插入数据操作

3. 插入操作注意问题

- 插入语句使用insert标签
- 在映射文件中使用parameterType属性指定要插入的数据类型
- Sql语句中使用#{实体属性名}方式引用实体中的属性值
- 插入操作使用的API是sqlSession.insert(“命名空间.id”,实体对象);
- 插入操作涉及数据库数据变化，所以要使用sqlSession对象显示的提交事务，即sqlSession.commit()

```
<!--删除操作-->
<delete id="delete" parameterType="java.lang.Integer">
    delete from user where id=#{id}
</delete>

<!--修改操作-->
<update id="update" parameterType="com.itheima.domain.User">
    update user set username=#{username},password=#{password} where id=#{id}
</update>

<!--插入操作-->
<insert id="save" parameterType="com.itheima.domain.User">
    insert into user values(#{id},#{username},#{password})
</insert>
```

4.4 知识小结

增删改查映射配置与API：

查询数据：`List<User> userList = sqlSession.selectList("userMapper.findAll");`

```
<select id="findAll" resultType="com.itheima.domain.User">
    select * from User
</select>
```

添加数据：`sqlSession.insert("userMapper.add", user);`

```
<insert id="add" parameterType="com.itheima.domain.User">
    insert into user values(#{id},#{username},#{password})
</insert>
```

修改数据：`sqlSession.update("userMapper.update", user);`

```
<update id="update" parameterType="com.itheima.domain.User">
    update user set username=#{username},password=#{password} where id=#{id}
</update>
```

删除数据：`sqlSession.delete("userMapper.delete",3);`

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

1. environments标签

数据库环境的配置，支持多环境配置

```
<environments default="development">
    <environment id="development">
        <transactionManager type="JDBC"/>
        <dataSource type="POOLED">
            <property name="driver" value="${jdbc.driver}"/>
            <property name="url" value="${jdbc.url}"/>
            <property name="username" value="${jdbc.username}"/>
            <property name="password" value="${jdbc.password}"/>
        </dataSource>
    </environment>
</environments>
```

指定默认的环境名称

指定当前环境的名称

指定事务管理类型是JDBC

指定当前数据源类型是连接池

数据源配置的基本参数

5.2 MyBatis常用配置解析

2. mapper标签

924x34

该标签的作用是加载映射的，加载方式有如下几种：

- 使用相对于类路径的资源引用，例如：<mapper resource="org/mybatis/builder/AuthorMapper.xml"/>
- 使用完全限定资源定位符（URL），例如：<mapper url="file:///var/mappers/AuthorMapper.xml"/>
- 使用映射器接口实现类的完全限定类名，例如：<mapper class="org.mybatis.builder.AuthorMapper"/>
- 将包内的映射器接口实现全部注册为映射器，例如：<package name="org.mybatis.builder"/>

5.3 知识小结

核心配置文件常用配置：

1、properties标签：该标签可以加载外部的properties文件

```
<properties resource="jdbc.properties"></properties>
```

2、typeAliases标签：设置类型别名

```
<typeAlias type="com.itheima.domain.User" alias="user"></typeAlias>
```

3、mappers标签：加载映射配置

```
<mapper resource="com/itheima/mapper/UserMapper.xml"></mapper>
```

6.1 SqlSession工厂构建器SqlSessionFactoryBuilder

常用API：SqlSessionFactory build(InputStream inputStream)
通过加载mybatis的核心文件的输入流的形式构建一个SqlSessionFactory对象

```
String resource = "org/mybatis/builder/mybatis-config.xml";
InputStream inputStream = Resources.getResourceAsStream(resource);
SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
SqlSessionFactory factory = builder.build(inputStream);
```

其中， Resources 工具类，这个类在 org.apache.ibatis.io 包中。Resources 类帮助你从类路径下、文件系统或一个 web URL 中加载资源文件。

1.2 代理开发方式

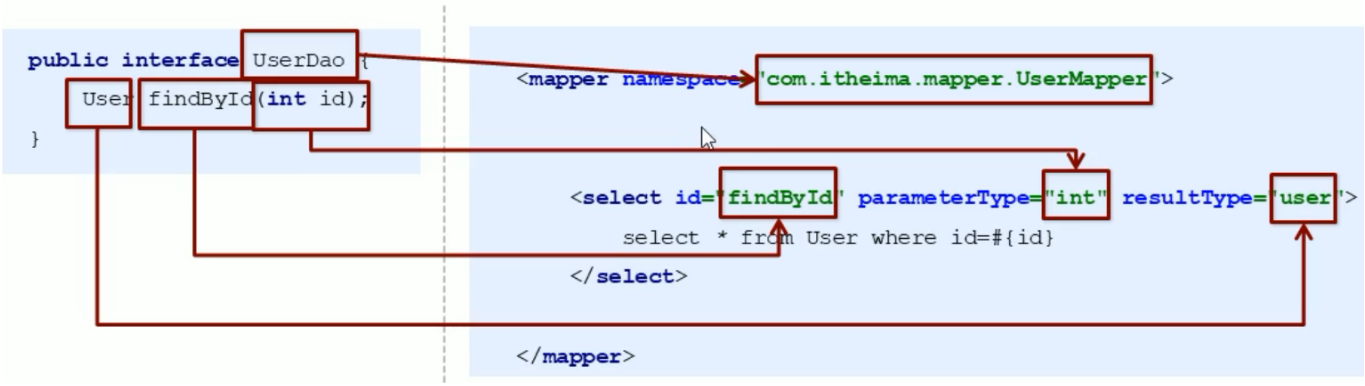
1. 代理开发方式介绍

采用 Mybatis 的代理开发方式实现 DAO 层的开发，这种方式是我们后面进入企业的主流。
Mapper 接口开发方法只需要程序员编写Mapper 接口（相当于Dao 接口），由Mybatis 框架根据接口定义创建接口的动态代理对象，代理对象的方法体同上边Dao接口实现类方法。
Mapper 接口开发需要遵循以下规范:

1、 Mapper.xml文件中的namespace与mapper接口的全限定名相同
2、 Mapper接口方法名和Mapper.xml中定义的每个statement的id相同
3、 Mapper接口方法的输入参数类型和mapper.xml中定义的每个sql的parameterType的类型相同
4、 Mapper接口方法的输出参数类型和mapper.xml中定义的每个sql的resultType的类型相同

1.2 代理开发方式

2. 编写UserMapper接口



```xml
<select id="findByCondition" parameterType="user" resultType="user">
    select * from user
    <where>
        <if test="id!=0">
            and id=#{id}
        </if>
        <if test="username!=null">
            and username=#{username}
        </if>
        <if test="password!=null">
            and password=#{password}
        </if>
    </where>
</select>
```

抽取

```xml
<!--sql语句抽取-->
<sql id="selectUser">select * from user</sql>

<select id="findByCondition" parameterType="user" resultType="user">
    <include refid="selectUser"></include>
    <where>
        <if test="id!=0">
            and id=#{id}
        </if>
        <if test="username!=null">
            and username=#{username}
        </if>
        <if test="password!=null">
            and password=#{password}
        </if>
    </where>
</select>
```

MyBatis映射文件配置：

<select>：查询

<insert>：插入

<update>：修改

<delete>：删除

<where>：where条件

<if>：if判断

<foreach>：循环

<sql>：sql片段抽取

1.1 typeHandlers标签

你可以重写类型处理器或创建你自己的类型处理器来处理不支持的或非标准的类型。具体做法为：实现 org.apache.ibatis.type.TypeHandler 接口，或继承一个很便利的类 org.apache.ibatis.type.BaseTypeHandler，然后可以选择性地将它映射到一个JDBC类型。例如需求：一个Java中的Date数据类型，我想将之存到数据库的时候存成一个1970年至今的毫秒数，取出来时转换成java的Date，即java的Date与数据库的varchar毫秒值之间转换。

开发步骤：

① 定义转换类继承类BaseTypeHandler<T>

② 覆盖4个未实现的方法，其中setNonNullParameter为java程序设置数据到数据库的回调方法，getNullableResult为查询时 mysql的字符串类型转换成java的Type类型的方法

③ 在MyBatis核心配置文件中进行注册

④ 测试转换是否正确

```java
public class DateTypeHandler extends BaseTypeHandler<Date> {
    //将java类型 转换成 数据库需要的类型
    public void setNonNullParameter(PreparedStatement preparedStatement, int i, Date date, JdbcType jdbcType
        long time = date.getTime();
        preparedStatement.setLong(i,time);
    }

    //将数据库中类型 转换成java类型
    //String参数  要转换的字段名称
    //ResultSet 查询出的结果集
    public Date getNullableResult(ResultSet resultSet, String s) throws SQLException {
        //获得结果集中需要的数据(long) 转换成Date类型 返回
        long aLong = resultSet.getLong(s);
        Date date = new Date(aLong);
        return date;
    }
```

```java
public void test2() throws IOException {
    InputStream resourceAsStream = Resources.getResourceAsStream("sqlMapConf
    SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().buil
    SqlSession sqlSession = sqlSessionFactory.openSession();
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);

    User user = mapper.findById(15);
    System.out.println("user中的birthday: "+user.getBirthday());

    sqlSession.commit();
    sqlSession.close();
}
```

1.2 plugins标签

MyBatis可以使用第三方的插件来对功能进行扩展，分页助手PageHelper是将分页的复杂操作进行封装，使用简单的方式即可获得分页的相关数据

开发步骤：

① 导入通用PageHelper的坐标

② 在mybatis核心配置文件中配置PageHelper插件

③ 测试分页数据获取

1.2 plugins标签

① 导入通用PageHelper坐标

```
<!-- 分页助手 -->
<dependency>
    <groupId>com.github.pagehelper</groupId>
    <artifactId>pagehelper</artifactId>
    <version>3.7.5</version>
</dependency>
<dependency>
    <groupId>com.github.jsqlparser</groupId>
    <artifactId>jsqlparser</artifactId>
    <version>0.9.1</version>
</dependency>
```

1.2 plugins标签

② 在mybatis核心配置文件中配置PageHelper插件

```
<!-- 注意：分页助手的插件  配置在通用馆mapper之前 -->
<plugin interceptor="com.github.pagehelper.PageHelper">
    <!-- 指定方言 -->
    <property name="dialect" value="mysql"/>
</plugin>
```

```java
public void test3() throws IOException {
    InputStream resourceAsStream = Resources.getResourceAsStream("sqlMapConfig.xml");
    SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(resourceAsStream);
    SqlSession sqlSession = sqlSessionFactory.openSession();
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);

    //设置分页相关参数   当前页+每页显示的条数
    PageHelper.startPage( pageNum: 2, pageSize: 3);

    List<User> userList = mapper.findAll();
    for (User user : userList) {
        System.out.println(user);
    }

    sqlSession.close();
}
```

```java
    //获得与分页相关参数
    PageInfo<User> pageInfo = new PageInfo<User>(userList);
    System.out.println("当前页："+pageInfo.getPageNum());
    System.out.println("每页显示条数："+pageInfo.getPageSize());
    System.out.println("总条数："+pageInfo.getTotal());
    System.out.println("总页数："+pageInfo.getPages());
    System.out.println("上一页："+pageInfo.getPrePage());
    System.out.println("下一页："+pageInfo.getNextPage());
    System.out.println("是否是第一个："+pageInfo.isIsFirstPage());
    System.out.println("是否是最后一个："+pageInfo.isIsLastPage());

    sqlSession.close();
}
```

1.3 知识小结

MyBatis核心配置文件常用标签：

1、properties标签：该标签可以加载外部的properties文件
2、typeAliases标签：设置类型别名
3、environments标签：数据源环境配置标签
4、typeHandlers标签：配置自定义类型处理器
5、plugins标签：配置MyBatis的插件

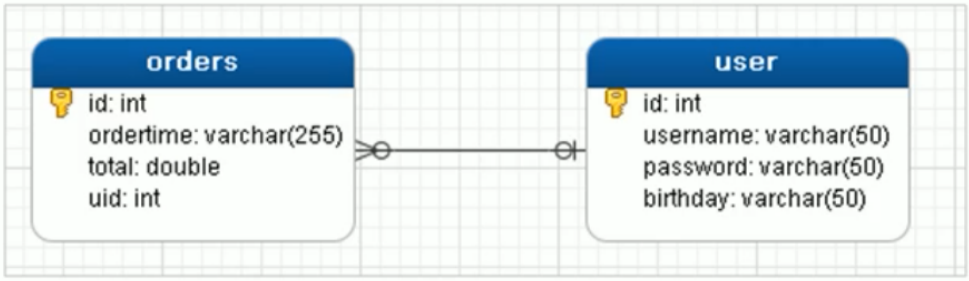
Mybatis

1.Mybatis多表查询

1.1 一对一查询

1. 一对一查询的模型

用户表和订单表的关系为，一个用户有多个订单，一个订单只从属于一个用户
一对一查询的需求：查询一个订单，与此同时查询出该订单所属的用户



多表查询，指定查询什么，把查询的东西封装到指定的mapper中

```xml
<resultMap id="orderMap" type="order">
    <!--手动指定字段与实体属性的映射关系
        column: 数据表的字段名称
        property: 实体的属性名称
    -->
    <id column="oid" property="id"></id>
    <result column="ordertime" property="ordertime"></result>
    <result column="total" property="total"></result>
    <result column="uid" property="user.id"></result>
    <result column="username" property="user.username"></result>
    <result column="password" property="user.password"></result>
    <result column="birthday" property="user.birthday"></result>
</resultMap>

<select id="findAll" resultMap="orderMap">
    SELECT *,o.id oid FROM orders o,USER u WHERE o.uid=u.id
</select>

</mapper>
```

集中到一起配置

```
    <id column="oid" property="id"></id>
    <result column="ordertime" property="ordertime"></result>
    <result column="total" property="total"></result>
    <!--<result column="uid" property="user.id"></result>
    <result column="username" property="user.username"></result>
    <result column="password" property="user.password"></result>
    <result column="birthday" property="user.birthday"></result>-->

    <!--
        property: 当前实体(order)中的属性名称(private User user)
        javaType: 当前实体(order)中的属性的类型(User)
    -->
    <association property="user" javaType="user">
        <id column="uid" property="id"></id>
        <result column="username" property="username"></result>
        <result column="password" property="password"></result>
        <result column="birthday" property="birthday"></result>
    </association>

</resultMap>
```

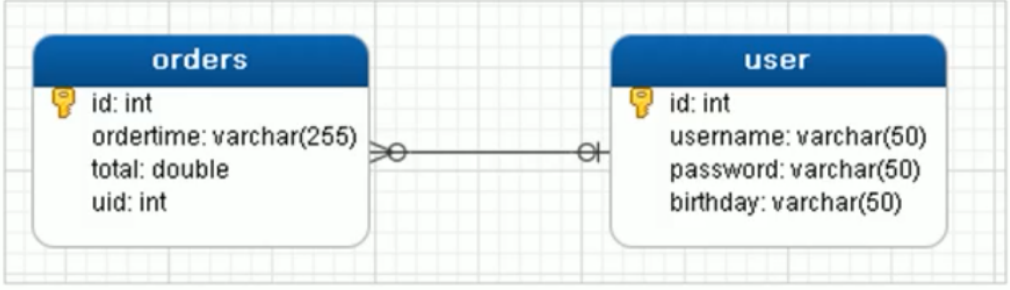
多表操作

1.2 一对多查询

1. 一对多查询的模型

用户表和订单表的关系为，一个用户有多个订单，一个订单只从属于一个用户

一对多查询的需求：查询一个用户，与此同时查询出该用户具有的订单



```java
public class User {

    private int id;
    private String username;
    private String password;
    private Date birthday;

    public List<Order> getOrderList() {
        return orderList;
    }
```

```xml
<resultMap id="userMap" type="user">
    <id column="uid" property="id"></id>
    <result column="username" property="username"></result>
    <result column="password" property="password"></result>
    <result column="birthday" property="birthday"></result>
    <!--配置集合信息
        property:集合名称
        ofType: 当前集合中的数据类型
    -->
    <collection property="orderList" ofType="order">
        <!--封装order的数据-->
        <id column="oid" property="id"></id>
        <result column="ordertime" property="ordertime"></result>
        <result column="total" property="total"></result>
    </collection>
</resultMap>
```

```xml
<select id="findAll" resultMap="userMap">
    SELECT *,o.id oid FROM USER u,orders o WHERE u.id=o.uid
</select>
```

多对多

```java
import java.util.Date;
import java.util.List;

public class User {

    private int id;
    private String username;
    private String password;
    private Date birthday;

    //描述的是当前用户存在哪些订单
    private List<Order> orderList;

    //描述的是当前用户具备哪些角色
    private List<Role> roleList;

    public List<Role> getRoleList() {
```

```xml
<!--##########################################################################-->

<resultMap id="userRoleMap" type="user">
    <!--user的信息-->
    <id column="userId" property="id"></id>
    <result column="username" property="username"></result>
    <result column="password" property="password"></result>
    <result column="birthday" property="birthday"></result>
    <!--user内部的roleList信息-->
    <collection property="roleList" ofType="role">
        <id column="roleId" property="id"></id>
        <result column="roleName" property="roleName"></result>
        <result column="roleDesc" property="roleDesc"></result>
    </collection>
</resultMap>
```

XML tag has empty body more... (Ctrl+F1)

```xml
<select id="findUserAndRoleAll" resultMap="userRoleMap">
    SELECT * FROM USER u,sys_user_role ur,sys_role r WHERE u.id=ur.userId AND ur.roleId=r.id
</select>
```

```java
@Test
public void test3() throws IOException {
    InputStream resourceAsStream = Resources.getResourceAsStream("sqlMapConfig.xml");
    SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(resourceAsStream);
    SqlSession sqlSession = sqlSessionFactory.openSession();

    UserMapper mapper = sqlSession.getMapper(UserMapper.class);
    List<User> userAndRoleAll = mapper.findUserAndRoleAll();
    for (User user : userAndRoleAll) {
        System.out.println(user);
    }

    sqlSession.close();
}
```

1.4 知识小结

MyBatis多表配置方式：

一对一配置：使用<resultMap>做配置

一对多配置：使用<resultMap>+<collection>做配置

多对多配置：使用<resultMap>+<collection>做配置

常用开发注解

1.1 MyBatis的常用注解

这几年来注解开发越来越流行，Mybatis也可以使用注解开发方式，这样我们就可以减少编写Mapper映射文件了。我们先围绕一些基本的CRUD来学习，再学习复杂映射多表操作。

@Insert：实现新增
@Update：实现更新
@Delete：实现删除
@Select：实现查询
@Result：实现结果集封装
@Results：可以与@Result一起使用，封装多个结果集
@One：实现一对一结果集封装
@Many：实现一对多结果集封装

注解（不需要xml配置啦）

```
public interface UserMapper {

    @Insert("insert into user values(#{id},#{username},#{password},#{birthday})")
    public void save(User user);

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

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

    @Select("select * from user where id=#{id}")
    public User findById(int id);

    @Select("select * from user")
    public List<User> findAll();
```

1.3 MyBatis的注解实现复杂映射开发

实现复杂关系映射之前我们可以在映射文件中通过配置<resultMap>来实现，使用注解开发后，我们可以使用@Results注解，@Result注解，@One注解，@Many注解组合完成复杂关系的配置

注解	说明
@Results	代替的是标签<resultMap>该注解中可以使用单个@Result注解，也可以使用@Result集合。使用格式：@Results（{@Result（），@Result（）}）或@Results（@Result（））
@Resut	代替了<id>标签和<result>标签 @Result中属性介绍： column：数据库的列名 property：需要装配的属性名 one：需要使用的@One 注解（@Result（one=@One）（）） many：需要使用的@Many 注解（@Result（many=@many）（））

注解一对一查询

```
public interface OrderMapper {

    @Select("select *,o.id oid from orders o,user u where o.uid=u.id")
    @Results({
            @Result(column = "oid",property = "id"),
            @Result(column = "ordertime",property = "ordertime"),
            @Result(column = "total",property = "total"),
            @Result(column = "uid",property = "user.id"),
            @Result(column = "username",property = "user.username"),
            @Result(column = "password",property = "user.password")
    })
    public List<Order> findAll();

}
```

test

```java
public class MyBatisTest2 {

    private OrderMapper mapper;

    @Before
    public void before() throws IOException {
        InputStream resourceAsStream = Resources.getResourceAsStream("sqlMapConfig.xml");
        SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(resourceAsStream);
        SqlSession sqlSession = sqlSessionFactory.openSession( b: true);
        mapper = sqlSession.getMapper(OrderMapper.class);
    }
```

```java
@Test
public void testSave(){
    List<Order> all = mapper.findAll();
    for (Order order : all) {
        System.out.println(order);
    }
}
```

查询的另外一种方法

```java
public interface OrderMapper {

    @Select("select * from orders")
    @Results({
            @Result(column = "id",property = "id"),
            @Result(column = "ordertime",property = "ordertime"),
            @Result(column = "total",property = "total"),
            @Result(
                    property = "user", //要封装的属性名称
                    column = "uid", //根据那个字段去查询user表的数据
                    javaType = User.class, //要封装的实体类型
                    //select属性 代表查询那个接口的方法获得数据
                    one = @One(select = "com.itheima.mapper.UserMapper.findById")
            )
    })
    public List<Order> findAll();
```

一对多

在接口那写得

```
@Select("select * from orders where uid=#{uid}")
public List<Order> findByUid(int uid);
```

```
@Select("select * from user")
@Results({
        @Result(id=true ,column = "id",property = "id"),
        @Result(column = "username",property = "username"),
        @Result(column = "pasword",property = "password"),
        @Result(
                property = "orderList",
                column = "id",
                javaType = List.class,
                many = @Many(select = "com.itheima.mapper.OrderMapper.findByUid")
        )
})
```

```
        )
    })
    public List<User> findUserAndOrderAll();

}
```

```java
private UserMapper mapper;

@Before
public void before() throws IOException {
    InputStream resourceAsStream = Resources.getResourceAsStream("sqlMapConfig.xml");
    SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(resourceAsStream);
    SqlSession sqlSession = sqlSessionFactory.openSession( b: true);
    mapper = sqlSession.getMapper(UserMapper.class);
}


@Test
public void testSave(){
    List<User> userAndOrderAll = mapper.findUserAndOrderAll();
    for (User user : userAndOrderAll) {
        System.out.println(user);
    }
}
```

多对多 多了一张表

rMapper.java | RoleMapper.java | User.java

```java
package com.itheima.mapper;

import com.itheima.domain.Role;
import org.apache.ibatis.annotations.Select;

import java.util.List;

public interface RoleMapper {

    @Select("SELECT * FROM sys_user_role ur,sys_role r WHERE ur.roleId=r.id AND ur.userId=#{uid}")
    public List<Role> findByUid(int uid);

}
```

```
@Select("SELECT * FROM USER")
@Results({
        @Result(id = true, column = "id", property = "id"),
        @Result(id = true, column = "username", property = "username"),
        @Result(id = true, column = "password", property = "password"),
        @Result(
                property = "roleList",
                column = "id",
                javaType = List.class,
                many = @Many(select = "com.itheima.mapper.RoleMapper.findByUid")
        )
})
public List<User> findUserAndRoleAll();
```

```
    @Test
    public void testSave(){
        List<User> userAndRoleAll = mapper.findUserAndRoleAll();
        for (User user : userAndRoleAll) {
            System.out.println(user);
        }
    }


}
```

原始整合方式（源码是： iitheima_ssm）

1.1 准备工作

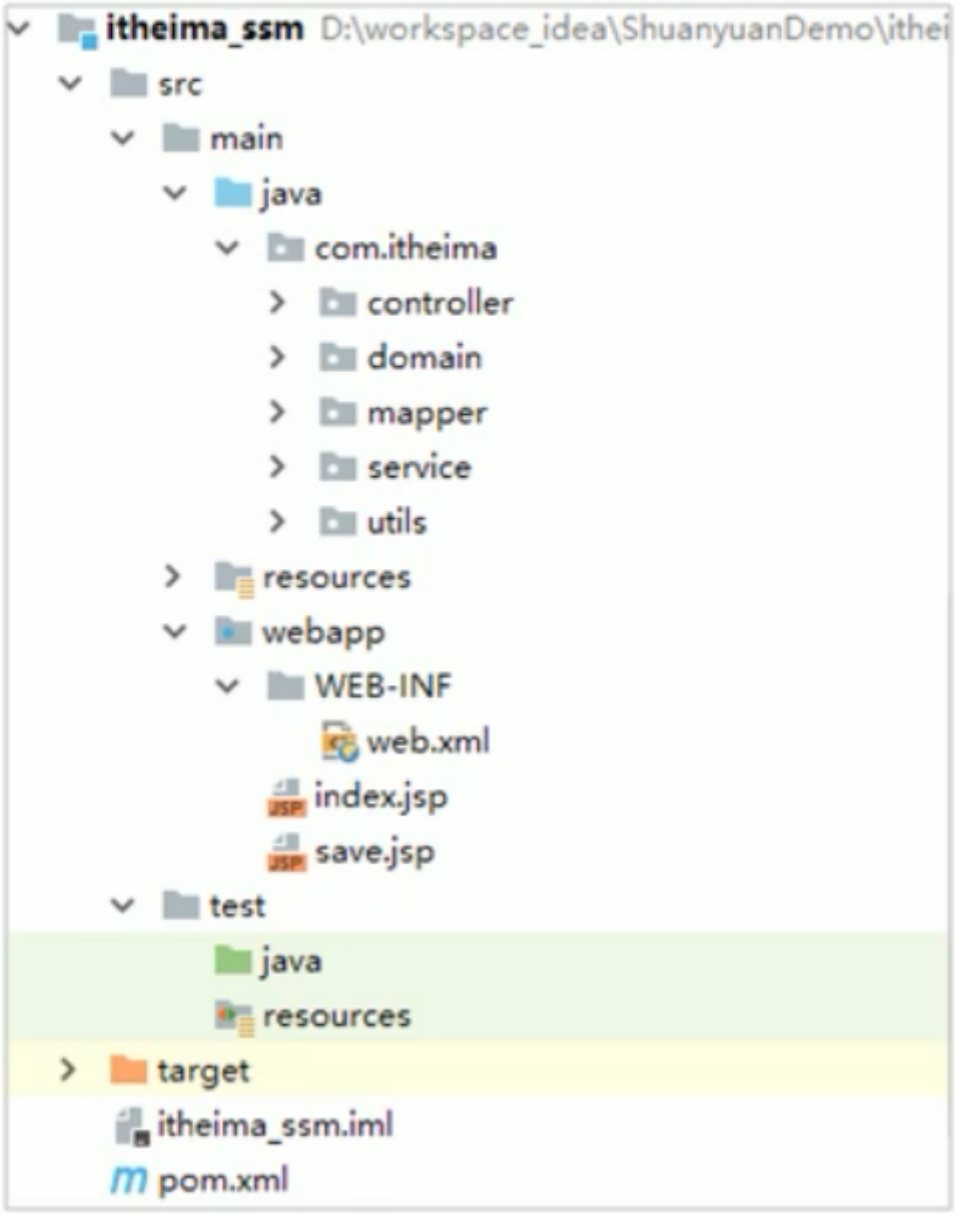
1. 原始方式整合

```
create database ssm;
create table account(
    id int primary key auto_increment,
    name varchar(100),
    money double(7,2)
);
```

id	name	money
1	tom	5000
2	lucy	5000

1.1原始方式整合

2. 创建Maven工程



1.1原始方式整合

3. 导入Maven坐标

点击打开坐标内容

1.1原始方式整合

4. 编写实体类

```
public class Account {
    private int id;
    private String name;
    private double money;
    //省略getter和setter方法
}
```

1.1原始方式整合

5. 编写Mapper接口

```
public interface AccountMapper {
    //保存账户数据
    void save(Account account);
    //查询账户数据
    List<Account> findAll();
}
```

1.1原始方式整合

6. 编写Service接口

```
public interface AccountService {
    void save(Account account); //保存账户数据
    List<Account> findAll(); //查询账户数据
}
```

1.1原始方式整合

7. 编写Service接口实现

```
@Service("accountService")
public class AccountServiceImpl implements AccountService {
    public void save(Account account) {
        SqlSession sqlSession = MyBatisUtils.openSession();
        AccountMapper accountMapper = sqlSession.getMapper(AccountMapper.class);
        accountMapper.save(account);
        sqlSession.commit();
        sqlSession.close();
    }
    public List<Account> findAll() {
        SqlSession sqlSession = MyBatisUtils.openSession();
        AccountMapper accountMapper = sqlSession.getMapper(AccountMapper.class);
        return accountMapper.findAll();
    }
}
```

1.1原始方式整合

8. 编写Controller

```
@Controller
public class AccountController {
    @Autowired
    private AccountService accountService;
    @RequestMapping("/save")
    @ResponseBody
    public String save(Account account){
        accountService.save(account);
        return "save success";
    }
    @RequestMapping("/findAll")
    public ModelAndView findAll(){
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.setViewName("accountList");
        modelAndView.addObject("accountList",accountService.findAll());
        return modelAndView;
    }
}
```

1.1原始方式整合

9. 编写添加页面

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
    <h1>保存账户信息表单</h1>
    <form action="${pageContext.request.contextPath}/save.action" method="post">
        用户名称<input type="text" name="name"><br/>
        账户金额<input type="text" name="money"><br/>
        <input type="submit" value="保存"><br/>
    </form>
</body>
</html>
```

1.1原始方式整合

10. 编写列表页面

```
<table border="1">
    <tr>
        <th>账户id</th>
        <th>账户名称</th>
        <th>账户金额</th>
    </tr>
    <c:forEach items="${accountList}" var="account">
        <tr>
            <td>${account.id}</td>
            <td>${account.name}</td>
            <td>${account.money}</td>
        </tr>
    </c:forEach>
</table>
```

1.1原始方式整合

11. 编写相应配置文件

- Spring配置文件：applicationContext.xml
- SprngMVC配置文件：spring-mvc.xml
- MyBatis映射文件：AccountMapper.xml
- MyBatis核心文件：sqlMapConfig.xml
- 数据库连接信息文件：jdbc.properties
- Web.xml文件：web.xml
- 日志文件：log4j.xml

1.1原始方式整合

12. 测试添加账户

id	name	money
1	tom	5000
2	lucy	5000
4	zhangsan	1000
5	zhangsan11	1000
6	测试数据	10000

sqlMap核心文件

b.xml × applicationContext.xml × spring-mcx.xml × AccountMapper.xml × sqlMapConfig.xml ×

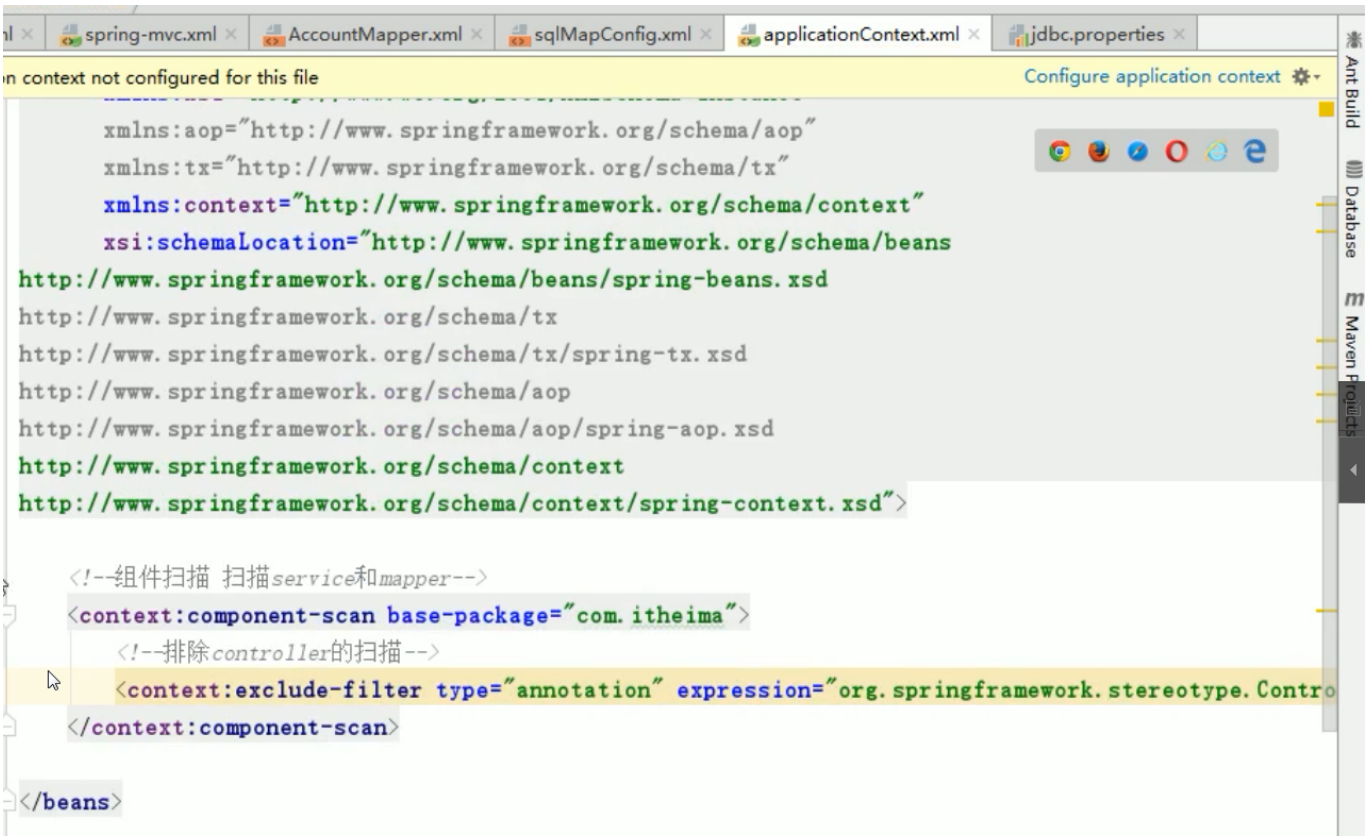
```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>

    <!--定义别名-->
    <typeAliases>
        <!--<typeAlias type="com.itheima.domain.Account" alias="account"></typeAlias>-->
        <package name="com.itheima.domian"></package>
    </typeAliases>

</configuration>
```

XML tag has empty body more... (Ctrl+F1)

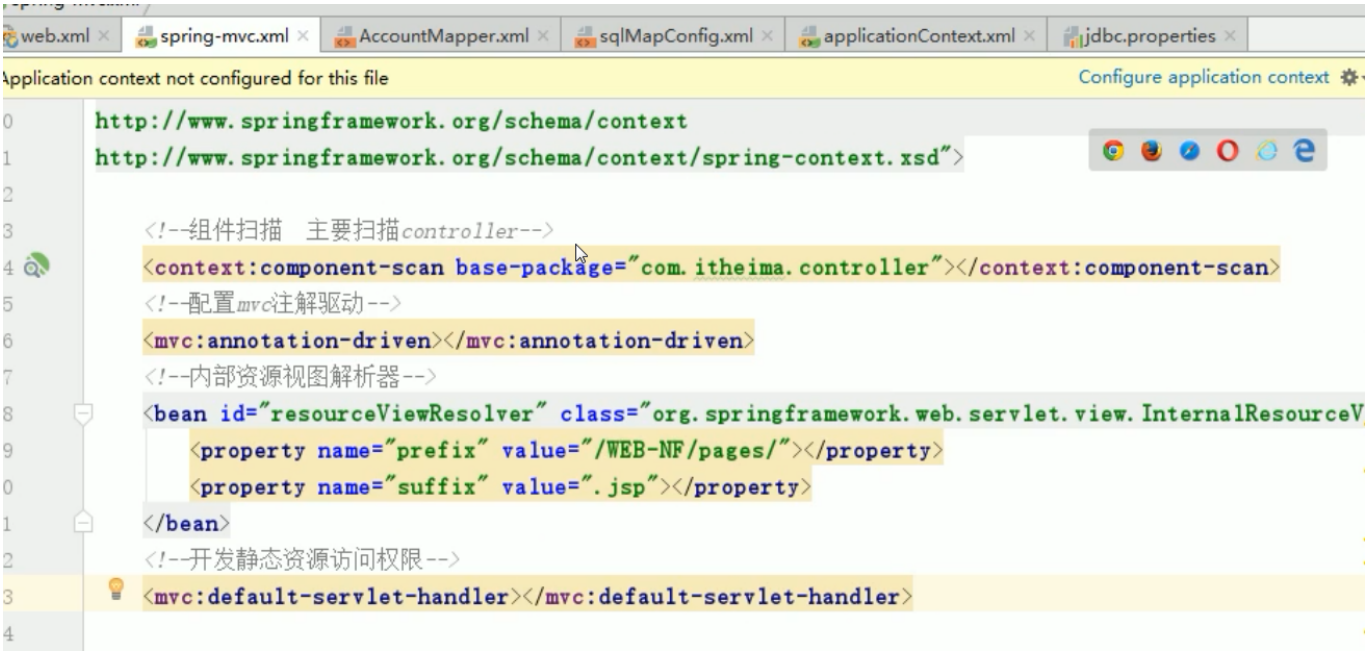
```xml
<!--加载properties文件-->
<properties resource="jdbc.properties"></properties>

<!--定义别名-->
<typeAliases>
    <!--<typeAlias type="com.itheima.domain.Account" alias="account"></typeAlias>-->
    <package name="com.itheima.domain"></package>
</typeAliases>

<!--环境-->
<environments default="developement">
    <environment id="developement">
        <transactionManager type="JDBC"></transactionManager>
        <dataSource type="POOLED">
            <property name="driver" value="${jdbc.driver}"></property>
            <property name="url" value="${jdbc.url}"></property>
            <property name="username" value="${jdbc.username}"></property>
            <property name="password" value="${jdbc.password}"></property>
        </dataSource>
```

```xml
<!--加载映射-->
<mappers>
    <!--<mapper resource="com/itheima/mapper/AccountMapper.xml"></mapper>-->
    <package name="com.itheima.mapper"></package>
</mappers>
```

spring核心文件



Spring MVC



web.xml

```xml
<!--spring 监听器-->
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:applicationContext.xml</param-value>
</context-param>
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

```xml
<!--springmvc的前端控制器-->
<servlet>
    <servlet-name>DispatcherServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:spring-mvc.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>DispatcherServlet</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

```xml
<!--乱码过滤器-->
<filter>
    <filter-name>CharacterEncodingFilter</filter-name>
    <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
    <init-param>
        <param-name>encoding</param-name>
        <param-value>UTF-8</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>CharacterEncodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

Spring整合MyBatis

就是用MyBatis包里面api，简化配置

1.2 Spring整合MyBatis

1. 整合思路

```
SqlSession sqlSession = MyBatisUtils.openSession();
AccountMapper accountMapper = sqlSession.getMapper(AccountMapper.class);
accountMapper.save(account);
sqlSession.commit();
sqlSession.close();
```

将Session工厂交给Spring容器管理，从容器中获得执行操作的Mapper实例即可

将事务的控制交给Spring容器进行声明式事务控制

spring-mvc.xml | accountList.jsp | AccountServiceImpl.java | applicationContext.xml | sqlMapConfig-spring.xml

MVC DispatcherServlet servlet context in module **itheima_ssm**. File is included in **4 contexts**.

```
<!--加载propeties文件-->
<context:property-placeholder location="classpath:jdbc.properties"></context:proper
<!--配置数据源信息-->
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
    <property name="driverClass" value="${jdbc.driver}"></property>
    <property name="jdbcUrl" value="${jdbc.url}"></property>
    <property name="user" value="${jdbc.username}"></property>
    <property name="password" value="${jdbc.password}"></property>
</bean>
<!--配置sessionFactory-->
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
    <property name="dataSource" ref="dataSource"></property>
    <!--加载mybatis核心文件-->
    <property name="configLocation" value="sqlMapConfig-spring.xml"></property>
</bean>
```

```
<!--扫描mapper所在的包 为mapper创建实现类-->
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
    <property name="basePackage" value="com.itheima.mapper"></property>
</bean>
```

service > impl > AccountServiceImpl

spring-mvc.xml | accountList.jsp | AccountServiceImpl.java | applicationContext.xml | sqlMapConfig-spring.xml

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

@Service("accountService")
public class AccountServiceImpl implements AccountService {

    @Autowired
    private AccountMapper accountMapper;

    @Override
    public void save(Account account) {
        accountMapper.save(account);
    }

    @Override
    public List<Account> findAll() {
        return accountMapper.findAll();
    }
}
```

事务

```xml
<!--平台事务管理器-->
<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionMan
    <property name="dataSource" ref="dataSource"></property>
</bean>

<!--配置事务增强-->
<tx:advice id="txAdvice">
    <tx:attributes>
        <tx:method name="*"/>
    </tx:attributes>
</tx:advice>

<!--事务的aop织入-->
<aop:config>
    <aop:advisor advice-ref="txAdvice" pointcut="execution(* com.itheima.service.impl.*.*(..))"></aop:
</aop:config>

</beans>
```