



# MyBatis入门操作

# 目 录

## Contents

- ◆ MyBatis的简介
- ◆ MyBatis的快速入门
- ◆ MyBatis的映射文件概述
- ◆ MyBatis的增删改查操作
- ◆ MyBatis的核心配置文件概述
- ◆ MyBatis的相应API

# 1. Mybatis简介

## 1.1 原始jdbc操作（查询数据）

```
//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: "select id,username,password from user");
//执行查询
ResultSet resultSet = statement.executeQuery();
//遍历结果集
while(resultSet.next()){
    //封装实体
    User user = new User();
    user.setId(resultSet.getInt(columnLabel: "id"));
    user.setUsername(resultSet.getString(columnLabel: "username"));
    user.setPassword(resultSet.getString(columnLabel: "password"));
    //user实体封装完毕
    System.out.println(user);
}
//释放资源
resultSet.close();
statement.close();
connection.close();
```

# 1. Mybatis简介

## 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();
```

## 1.2 原始jdbc操作的分析

原始jdbc开发存在的问题如下：

- ① 数据库连接创建、释放频繁造成系统资源浪费从而影响系统性能
- ② sql语句在代码中硬编码，造成代码不易维护，实际应用sql变化的可能较大，sql变动需要改变java代码。
- ③ 查询操作时，需要手动将结果集中的数据手动封装到实体中。插入操作时，需要手动将实体的数据设置到sql语句的占位符位置

应对上述问题给出的解决方案：

- ① 使用数据库连接池初始化连接资源
- ② 将sql语句抽取到xml配置文件中
- ③ 使用反射、内省等底层技术，自动将实体与表进行属性与字段的自动映射

## 1.3 什么是Mybatis



- mybatis 是一个优秀的基于java的持久层框架，它内部封装了 jdbc，使开发者只需要关注sql语句本身，而不需要花费精力去处理加载驱动、创建连接、创建statement等繁杂的过程。
- mybatis通过xml或注解的方式将要执行的各种 statement配置起来，并通过java对象和statement中sql的动态参数进行映射生成最终执行的sql语句。
- 最后mybatis框架执行sql并将结果映射为java对象并返回。采用ORM思想解决了实体和数据库映射的问题，对jdbc 进行了封装，屏蔽了jdbc api 底层访问细节，使我们不用与jdbc api打交到，就可以完成对数据库的持久化操作。

# 目 录

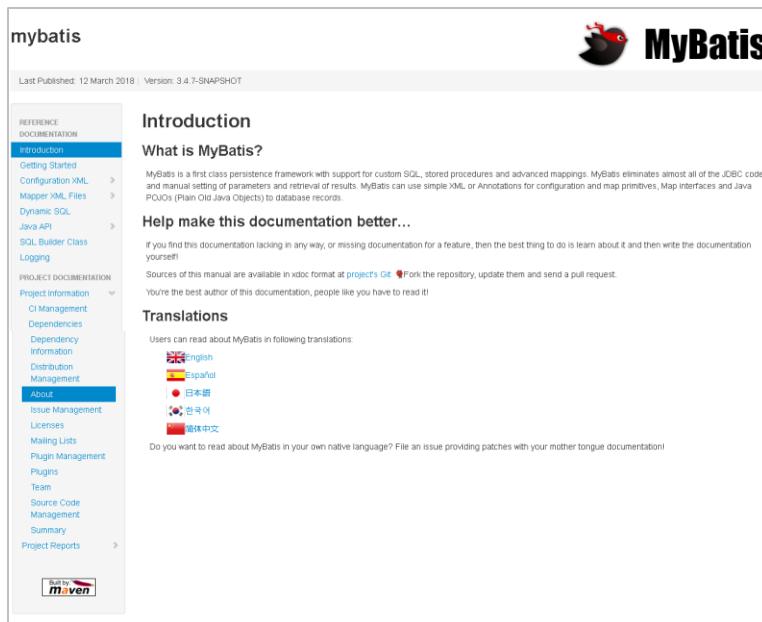
# Contents

- ◆ MyBatis的简介
- ◆ MyBatis的快速入门
- ◆ MyBatis的映射文件概述
- ◆ MyBatis的增删改查操作
- ◆ MyBatis的核心配置文件概述
- ◆ MyBatis的相应API

# 2. Mybatis的快速入门

## 2.1 MyBatis开发步骤

MyBatis官网地址：<http://www.mybatis.org/mybatis-3/>



mybatis

Last Published: 12 March 2018 | Version: 3.4.7-SNAPSHOT

REFERENCE DOCUMENTATION

- Introduction
- Getting Started
- Configuration XML
- Mapper XML Files
- Dynamic SQL
- Java API
- SQL Builder Class
- Logging

PROJECT DOCUMENTATION

- CI Management
- Dependencies
- Dependency Information
- Distribution Management
- About
- Issue Management
- Licenses
- Mailing Lists
- Plugin Management
- Plugins
- Team
- Source Code Management
- Summary
- Project Reports

Translations

MyBatis is a first-class persistence framework with support for custom SQL, stored procedures and advanced mappings. MyBatis eliminates almost all of the JDBC code and manual setting of parameters and retrieval of results. MyBatis can use simple XML or Annotations for configuration and map primitives, Map interfaces and Java POJOs (Plain Old Java Objects) to database records.

If you find this documentation lacking in any way, or missing documentation for a feature, then the best thing to do is learn about it and then write the documentation yourself!

Sources of this manual are available in docx format at [projects Git](#). Fork the repository, update them and send a pull request.

You're the best author of this documentation, people like you have to read it!

Do you want to read about MyBatis in your own native language? File an issue providing patches with your mother tongue documentation!

Built by 

MyBatis开发步骤：

- ① 添加MyBatis的坐标
- ② 创建user数据表
- ③ 编写User实体类
- ④ 编写映射文件UserMapper.xml
- ⑤ 编写核心文件SqlMapConfig.xml
- ⑥ 编写测试类

# 2. Mybatis的快速入门

## 2.2 环境搭建

### 1. 导入MyBatis的坐标和其他相关坐标

```
<!--mybatis坐标-->
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis</artifactId>
    <version>3.4.5</version>
</dependency>
<!--mysql驱动坐标-->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.6</version>
    <scope>runtime</scope>
</dependency>
```

# 2. Mybatis的快速入门

## 2.2 环境搭建

### 1. 导入MyBatis的坐标和其他相关坐标

```
<!--单元测试坐标-->  
  
<dependency>  
    <groupId>junit</groupId>  
    <artifactId>junit</artifactId>  
    <version>4.12</version>  
    <scope>test</scope>  
</dependency>  
  
<!--日志坐标-->  
  
<dependency>  
    <groupId>log4j</groupId>  
    <artifactId>log4j</artifactId>  
    <version>1.2.12</version>  
</dependency>
```

# 2. Mybatis的快速入门

## 2.2 环境搭建

### 2. 创建user数据表

名	类型	长度	小数点	允许空值 (	)
id	int	11	0	<input type="checkbox"/>	 1
username	varchar	50	0	<input checked="" type="checkbox"/>	
password	varchar	50	0	<input checked="" type="checkbox"/>	

### 3. 编写User实体

```
public class User {  
    private int id;  
    private String username;  
    private String password;  
    //省略get个set方法  
}
```

# 2. Mybatis的快速入门

## 2.2 环境搭建

### 4. 编写UserMapper映射文件

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

# 2. Mybatis的快速入门

## 2.2 环境搭建

### 5. 编写MyBatis核心文件

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

    <typeAliases>
        <package name="org.snbo.domain"/>
    </typeAliases>

    <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>
    <mappers>
        <package name="org.snbo.dao"/>
    </mappers>
</configuration>
```

# 2. Mybatis的快速入门

## 2.3 编写测试代码

```
//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();
```

## 2. Mybatis的快速入门

### 2.4 知识小结

#### MyBatis开发步骤：

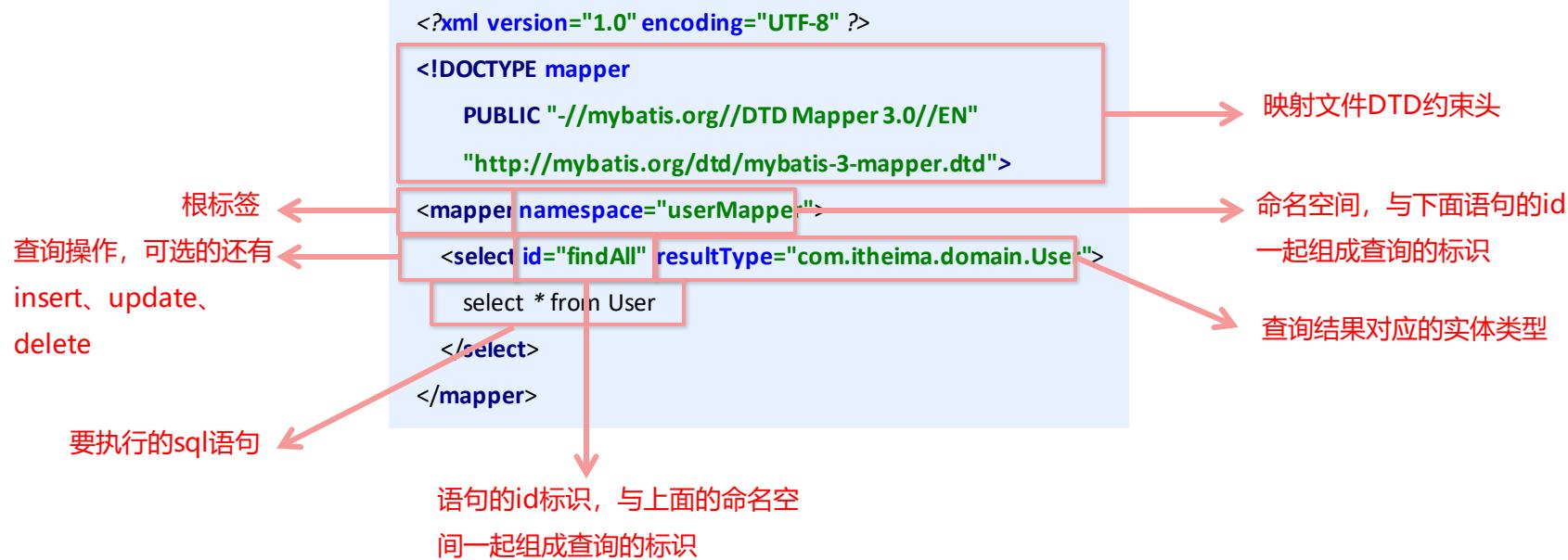
- ① 添加MyBatis的坐标
- ② 创建user数据表
- ③ 编写User实体类
- ④ 编写映射文件UserMapper.xml
- ⑤ 编写核心文件SqlMapConfig.xml
- ⑥ 编写测试类

# 目 录

# Contents

- ◆ MyBatis的简介
- ◆ MyBatis的快速入门
- ◆ MyBatis的映射文件概述
- ◆ MyBatis的增删改查操作
- ◆ MyBatis的核心配置文件概述
- ◆ MyBatis的相应API

# 3. MyBatis的映射文件概述



# 目 录

# Contents

- ◆ MyBatis的简介
- ◆ MyBatis的快速入门
- ◆ MyBatis的映射文件概述
- ◆ MyBatis的增删改查操作
- ◆ MyBatis的核心配置文件概述
- ◆ MyBatis的相应API

# 4. MyBatis的增删改查操作

## 4.1 MyBatis的插入数据操作

### 1. 编写UserMapper映射文件

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

# 4. MyBatis的增删改查操作

## 4.1 MyBatis的插入数据操作

### 2. 编写插入实体User的代码

```
InputStream resourceAsStream = Resources.getResourceAsStream("SqlMapConfig.xml");
SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(resourceAsStream);
SqlSession sqlSession = sqlSessionFactory.openSession();
int insert = sqlSession.insert("userMapper.add", user);
System.out.println(insert);
//提交事务
sqlSession.commit();
sqlSession.close();
```

# 4. MyBatis的增删改查操作

## 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()

# 4. MyBatis的增删改查操作

## 4.2 MyBatis的修改数据操作

### 1. 编写UserMapper映射文件

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

# 4. MyBatis的增删改查操作

## 4.2 MyBatis的修改数据操作

### 2. 编写修改实体User的代码

```
InputStream resourceAsStream = Resources.getResourceAsStream("SqlMapConfig.xml");
SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(resourceAsStream);
SqlSession sqlSession = sqlSessionFactory.openSession();
int update = sqlSession.update("userMapper.update", user);
System.out.println(update);
sqlSession.commit();
sqlSession.close();
```

# 4. MyBatis的增删改查操作

## 4.2 MyBatis的修改数据操作

### 3. 修改操作注意事项

- 修改语句使用update标签
- 修改操作使用的API是sqlSession.update(“命名空间.id”,实体对象);

# 4. MyBatis的增删改查操作

## 4.3 MyBatis的删除数据操作

### 1. 编写UserMapper映射文件

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

# 4. MyBatis的增删改查操作

## 4.3 MyBatis的删除数据操作

### 2. 编写删除数据的代码

```
InputStream resourceAsStream = Resources.getResourceAsStream("SqlMapConfig.xml");
SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(resourceAsStream);
SqlSession sqlSession = sqlSessionFactory.openSession();
int delete = sqlSession.delete("userMapper.delete",3);
System.out.println(delete);
sqlSession.commit();
sqlSession.close();
```

# 4. MyBatis的增删改查操作

## 4.3 MyBatis的删除数据操作

### 3. 删除操作注意问题

- 删除语句使用delete标签
- Sql语句中使用#{任意字符串}方式引用传递的单个参数
- 删除操作使用的API是sqlSession.delete(“命名空间.id”,Object);

# 4. MyBatis的增删改查操作

## 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>
```

# 目 录

# Contents

- ◆ MyBatis的简介
- ◆ MyBatis的快速入门
- ◆ MyBatis的映射文件概述
- ◆ MyBatis的增删改查操作
- ◆ MyBatis的核心配置文件概述
- ◆ MyBatis的相应API

# 5. MyBatis核心配置文件概述

## 5.1 MyBatis核心配置文件层级关系

- configuration 配置
  - properties 属性
  - settings 设置
  - typeAliases 类型别名
  - typeHandlers 类型处理器
  - objectFactory 对象工厂
  - plugins 插件
  - environments 环境
    - environment 环境变量
      - transactionManager 事务管理器
      - dataSource 数据源
  - databaseIdProvider 数据库厂商标识
  - mappers 映射器

# 5. MyBatis核心配置文件概述

## 5.2 MyBatis常用配置解析

### 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. MyBatis核心配置文件概述

## 5.2 MyBatis常用配置解析

### 1. environments标签

其中，事务管理器（transactionManager）类型有两种：

- JDBC：这个配置就是直接使用了JDBC的提交和回滚设置，它依赖于从数据源得到的连接来管理事务作用域。
- MANAGED：这个配置几乎没做什么。它从来不提交或回滚一个连接，而是让容器来管理事务的整个生命周期（比如JEE应用服务器的上下文）。默认情况下它会关闭连接，然而一些容器并不希望这样，因此需要将closeConnection属性设置为false来阻止它默认的关闭行为。

其中，数据源（dataSource）类型有三种：

- UNPOOLED：这个数据源的实现只是每次被请求时打开和关闭连接。
- POOLED：这种数据源的实现利用“池”的概念将JDBC连接对象组织起来。
- JNDI：这个数据源的实现是为了能在如EJB或应用服务器这类容器中使用，容器可以集中或在外部配置数据源，然后放置一个JNDI上下文的引用。

# 5. MyBatis核心配置文件概述

## 5.2 MyBatis常用配置解析

### 2. mapper标签

该标签的作用是加载映射的，加载方式有如下几种：

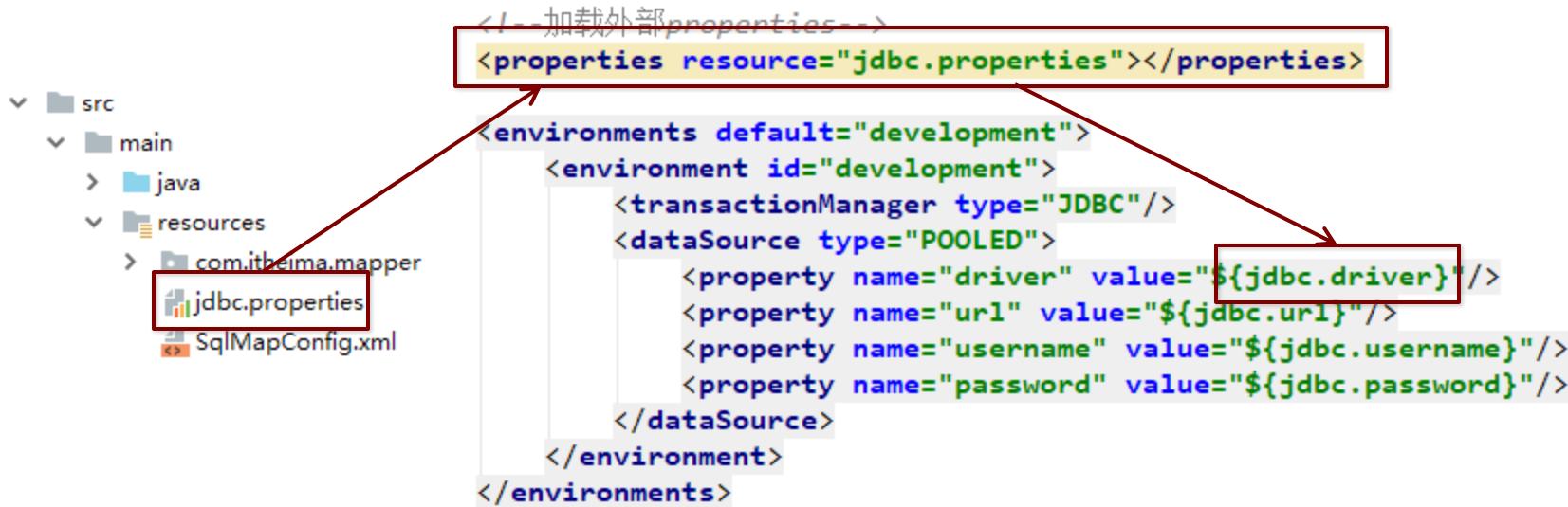
- 使用相对于类路径的资源引用，例如：<mapper resource="org/mybatis/builder/AuthorMapper.xml"/>
- 使用完全限定资源定位符（URL），例如：<mapper url="file:///var/mappers/AuthorMapper.xml"/>
- 使用映射器接口实现类的完全限定类名，例如：<mapper class="org.mybatis.builder.AuthorMapper"/>
- 将包内的映射器接口实现全部注册为映射器，例如：<package name="org.mybatis.builder"/>

# 5. MyBatis核心配置文件概述

## 5.2 MyBatis常用配置解析

### 3. Properties标签

实际开发中，习惯将数据源的配置信息单独抽取成一个properties文件，该标签可以加载额外配置的properties文件



# 5. MyBatis核心配置文件概述

## 5.2 MyBatis常用配置解析

### 4. typeAliases标签

类型别名是为Java类型设置一个短的名字。原来的类型名称配置如下

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

User全限定名称

配置typeAliases，为com.itheima.domain.User定义别名为user

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

```
<select id="findAll" resultType="user">  
    select * from User  
</select>
```

user为别名

# 5. MyBatis核心配置文件概述

## 5.2 MyBatis常用配置解析

### 4. typeAliases标签

上面我们是自定义的别名，mybatis框架已经为我们设置好的一些常用的类型的别名

别名	数据类型
string	String
long	Long
int	Integer
double	Double
boolean	Boolean
... ...	... ...

# 5. MyBatis核心配置文件概述

## 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>
```

# 5. MyBatis核心配置文件概述

## 5.3 知识小结

### 核心配置文件常用配置：

4、environments标签：数据源环境配置标签

```
<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>
```

# 目 录

# Contents

- ◆ MyBatis的简介
- ◆ MyBatis的快速入门
- ◆ MyBatis的映射文件概述
- ◆ MyBatis的增删改查操作
- ◆ MyBatis的核心配置文件概述
- ◆ MyBatis的相应API

# 6. MyBatis相应API

## 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。

# 6. MyBatis相应API

## 6.2 SqlSession工厂对象SqlSessionFactory

SqlSessionFactory 有多个个方法创建 SqlSession 实例。常用的有如下两个：

方法	解释
openSession()	会默认开启一个事务，但事务不会自动提交，也就意味着需要手动提交该事务，更新操作数据才会持久化到数据库中
openSession(boolean autoCommit)	参数为是否自动提交，如果设置为true，那么不需要手动提交事务

# 6. MyBatis相应API

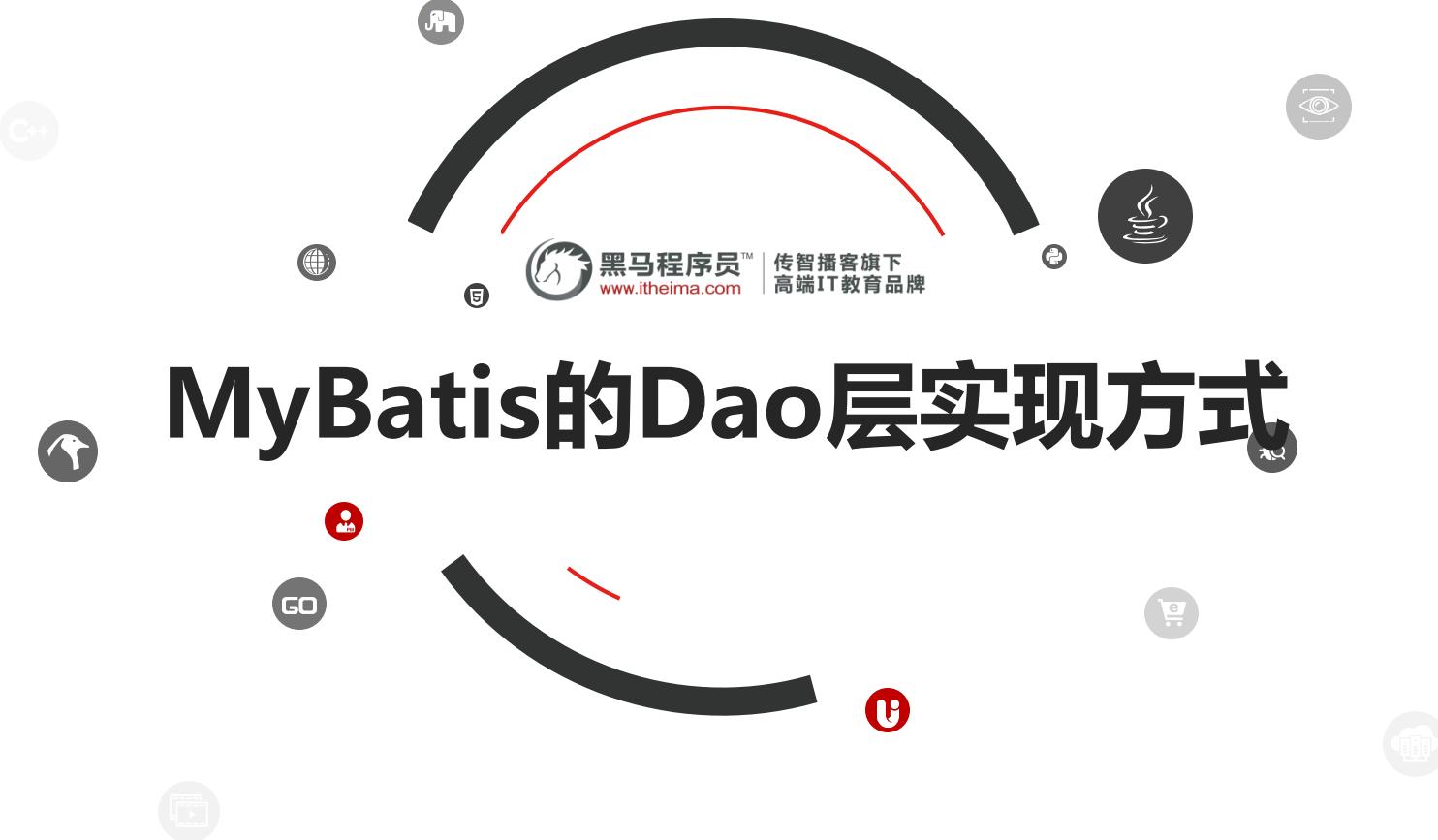
## 6.3 SqlSession会话对象

`SqlSession` 实例在 `MyBatis` 中是非常强大的一个类。在这里你会看到所有执行语句、提交或回滚事务和获取映射器实例的方法。执行语句的方法主要有：

```
<T> T selectOne(String statement, Object parameter)
<E> List<E> selectList(String statement, Object parameter)
int insert(String statement, Object parameter)
int update(String statement, Object parameter)
int delete(String statement, Object parameter)
```

操作事务的方法主要有：

```
void commit()
void rollback()
```



# MyBatis的Dao层实现方式

黑马程序员™ | 传智播客旗下  
高端IT教育品牌  
[www.itheima.com](http://www.itheima.com)

# 目 录

# Contents

◆ MyBatis的Dao层实现

# 1. Mybatis的Dao层实现

## 1.1 传统开发方式

### 1. 编写UserDao接口

```
public interface UserDao {  
    List<User> findAll() throws IOException;  
}
```

# 1. Mybatis的Dao层实现

## 1.1 传统开发方式

### 2. 编写UserDaoImpl实现

```
public class UserDaoImpl implements UserDao {  
    public List<User> findAll() throws IOException {  
        InputStream resourceAsStream = Resources.getResourceAsStream("SqlMapConfig.xml");  
        SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(resourceAsStream);  
        SqlSession sqlSession = sqlSessionFactory.openSession();  
        List<User> userList = sqlSession.selectList("userMapper.findAll");  
        sqlSession.close();  
        return userList;  
    }  
}
```

# 1. Mybatis的Dao层实现

## 1.1 传统开发方式

### 3. 测试传统方式

```
@Test  
public void testTraditionDao() throws IOException {  
    UserDao userDao = new UserDaoImpl();  
    List<User> all = userDao.findAll();  
    System.out.println(all);  
}
```

# 1. Mybatis的Dao层实现

## 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。

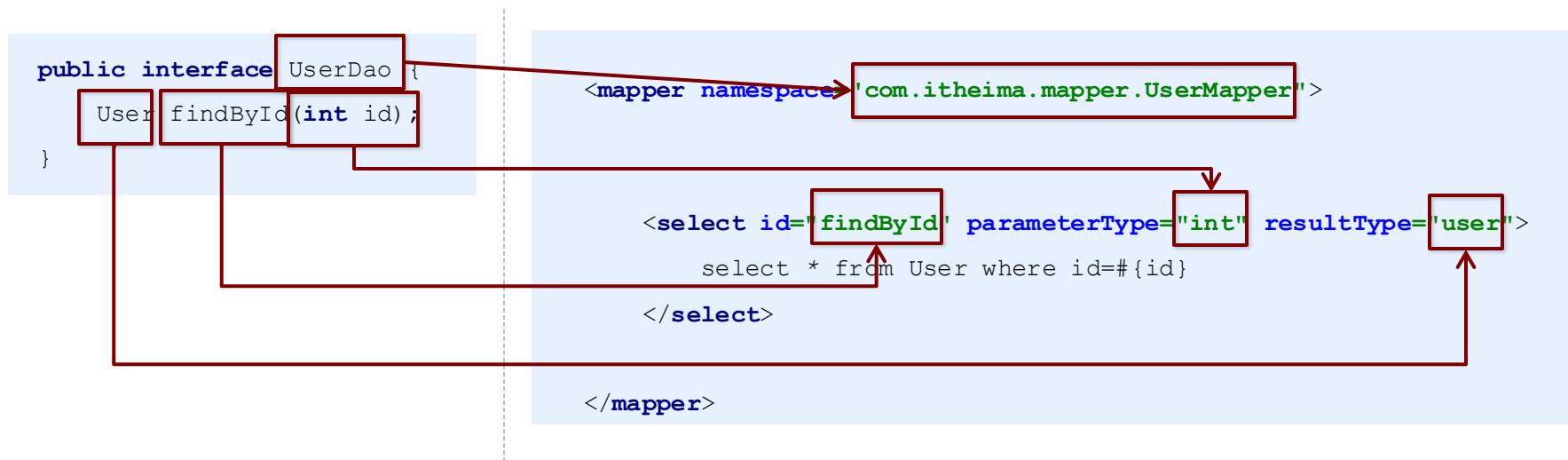
Mapper 接口开发需要遵循以下规范：

- 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的结果resultType的类型相同

# 1. Mybatis的Dao层实现

## 1.2 代理开发方式

### 2. 编写UserMapper接口



# 1. Mybatis的Dao层实现

## 1.2 代理开发方式

### 3. 测试代理方式

```
@Test
public void testProxyDao() throws IOException {
    InputStream resourceAsStream = Resources.getResourceAsStream("SqlMapConfig.xml");
    SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(resourceAsStream);
    SqlSession sqlSession = sqlSessionFactory.openSession();
    //获得MyBatis框架生成的UserMapper接口的实现类
    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
    User user = userMapper.findById(1);
    System.out.println(user);
    sqlSession.close();
}
```

# 1. Mybatis的Dao层实现

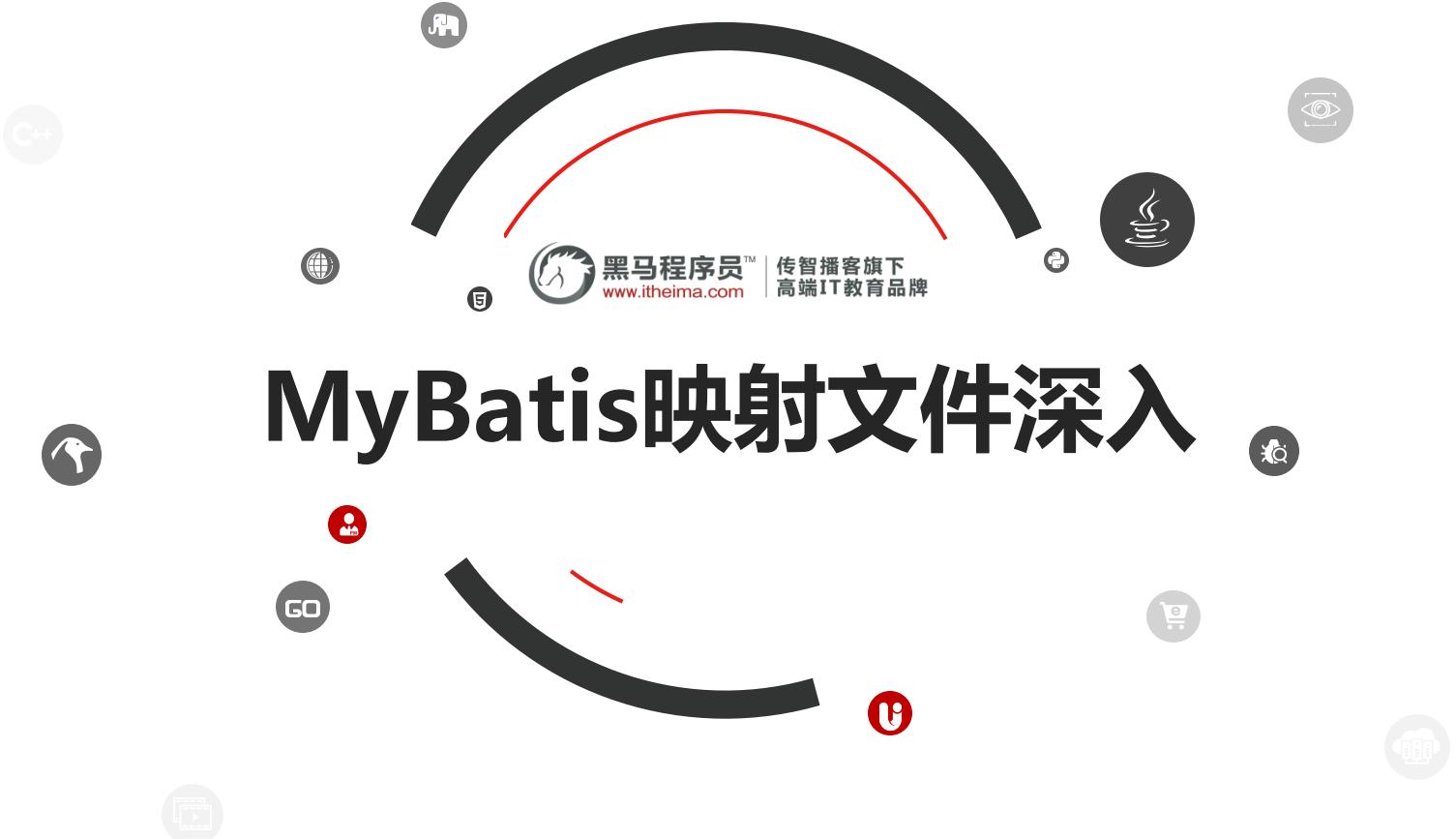
## 1.3 知识小结

MyBatis的Dao层实现的两种方式：

- 手动对Dao进行实现：传统开发方式
- 代理方式对Dao进行实现：

```
UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
```





# MyBatis映射文件深入

黑马程序员™ | 传智播客旗下  
高端IT教育品牌  
[www.itheima.com](http://www.itheima.com)

# 目 录

# Contents

◆ MyBatis映射文件深入

# 1. MyBatis映射文件深入

## 1.1 动态sql语句

### 1. 动态sql语句概述

Mybatis 的映射文件中，前面我们的 SQL 都是比较简单的，有些时候业务逻辑复杂时，我们的 SQL是动态变化的，此时在前面的学习中我们的 SQL 就不能满足要求了。

参考的官方文档，描述如下：

## Dynamic SQL

One of the most powerful features of MyBatis has always been its Dynamic SQL capabilities. If you have any experience with JDBC or any similar framework, you understand how painful it is to conditionally concatenate strings of SQL together, making sure not to forget spaces or to omit a comma at the end of a list of columns. Dynamic SQL can be downright painful to deal with.

While working with Dynamic SQL will never be a party, MyBatis certainly improves the situation with a powerful Dynamic SQL language that can be used within any mapped SQL statement.

The Dynamic SQL elements should be familiar to anyone who has used JSTL or any similar XML based text processors. In previous versions of MyBatis, there were a lot of elements to know and understand. MyBatis 3 greatly improves upon this, and now there are less than half of those elements to work with. MyBatis employs powerful OGNL based expressions to eliminate most of the other elements:

- if
- choose (when, otherwise)
- trim (where, set)
- foreach

# 1. MyBatis映射文件深入

## 1.1 动态sql语句

### 2. 动态 SQL 之<if>

我们根据实体类的不同取值，使用不同的 SQL语句来进行查询。比如在 id如果不为空时可以根据id查询，如果 username 不同空时还要加入用户名作为条件。这种情况在我们的多条件组合查询中经常会碰到。

```
<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>
    </where>
</select>
```

# 1. MyBatis映射文件深入

## 1.1 动态sql语句

### 2. 动态 SQL 之<if>

当查询条件id和username都存在时，控制台打印的sql语句如下：

```
.... ...
//获得MyBatis框架生成的UserMapper接口的实现类
UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
User condition = new User();
condition.setId(1);
condition.setUsername("lucy");
User user = userMapper.findByCondition(condition);
.... ...
```

```
- Created connection 586084331.
- Setting autocommit to false on JDBC Connection [com.mysql.jdbc. ....
- ==> Preparing: select * from User WHERE id=? and username=?
- ==> Parameters: 1(Integer), lucy(String)
- <==      Total: 1
```

# 1. MyBatis映射文件深入

## 1.1 动态sql语句

### 2. 动态 SQL 之<if>

当查询条件只有id存在时，控制台打印的sql语句如下：

```
.... ...
//获得MyBatis框架生成的UserMapper接口的实现类
UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
User condition = new User();
condition.setId(1);
User user = userMapper.findByCondition(condition);
.... ...
```

```
- Setting autocommit to false on JDBC Connection [com.mysql.
- ==> Preparing: select * from User WHERE id=?
- ==> Parameters: 1(Integer)
- <==      Total: 1
```

# 1. MyBatis映射文件深入

## 1.1 动态sql语句

### 3. 动态 SQL 之<foreach>

循环执行sql的拼接操作，例如：SELECT \* FROM USER WHERE id IN (1,2,5)。

```
<select id="findByIds" parameterType="list" resultType="user">
    select * from User
    <where>
        <foreach collection="array" open="id in(" close=")" item="id" separator=", "}>
            #{id}
        </foreach>
    </where>
</select>
```

# 1. MyBatis映射文件深入

## 1.1 动态sql语句

### 3. 动态 SQL 之<foreach>

测试代码片段如下：

```
.... ...
//获得MyBatis框架生成的UserMapper接口的实现类
UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
int[] ids = new int[]{2,5};
List<User> userList = userMapper.findByIds(ids);
System.out.println(userList);
.... ...
```

```
11:21:02,237 DEBUG findByIds:159 - ==> Preparing: select * from User WHERE id in( ? , ? )
11:21:02,262 DEBUG findByIds:159 - ==> Parameters: 2(Integer), 5(Integer)
11:21:02,280 DEBUG findByIds:159 - <==      Total: 2
[User{id=2, username='tom', password='123'}, User{id=5, username='haohao', password='123'}]
```

# 1. MyBatis映射文件深入

## 1.1 动态sql语句

### 3. 动态 SQL 之<foreach>

foreach标签的属性含义如下：

<foreach>标签用于遍历集合，它的属性：

- collection：代表要遍历的集合元素，注意编写时不要写#{}  
• open：代表语句的开始部分  
• close：代表结束部分  
• item：代表遍历集合的每个元素，生成的变量名  
• sperator：代表分隔符

# 1. MyBatis映射文件深入

## 1.2 SQL片段抽取

Sql 中可将重复的 sql 提取出来，使用时用 include 引用即可，最终达到 sql 重用的目的

```
<!--抽取sql片段简化编写-->
<sql id="selectUser" select * from User</sql>
<select id="findById" parameterType="int" resultType="user">
    <include refid="selectUser"></include> where id=#{id}
</select>
<select id="findByIds" parameterType="list" resultType="user">
    <include refid="selectUser"></include>
    <where>
        <foreach collection="array" open="id in(" close=")" item="id" separator=",">
            #{id}
        </foreach>
    </where>
</select>
```

# 1. MyBatis映射文件深入

## 1.3 知识小结

### MyBatis映射文件配置：

<select>：查询

<insert>：插入

<update>：修改

<delete>：删除

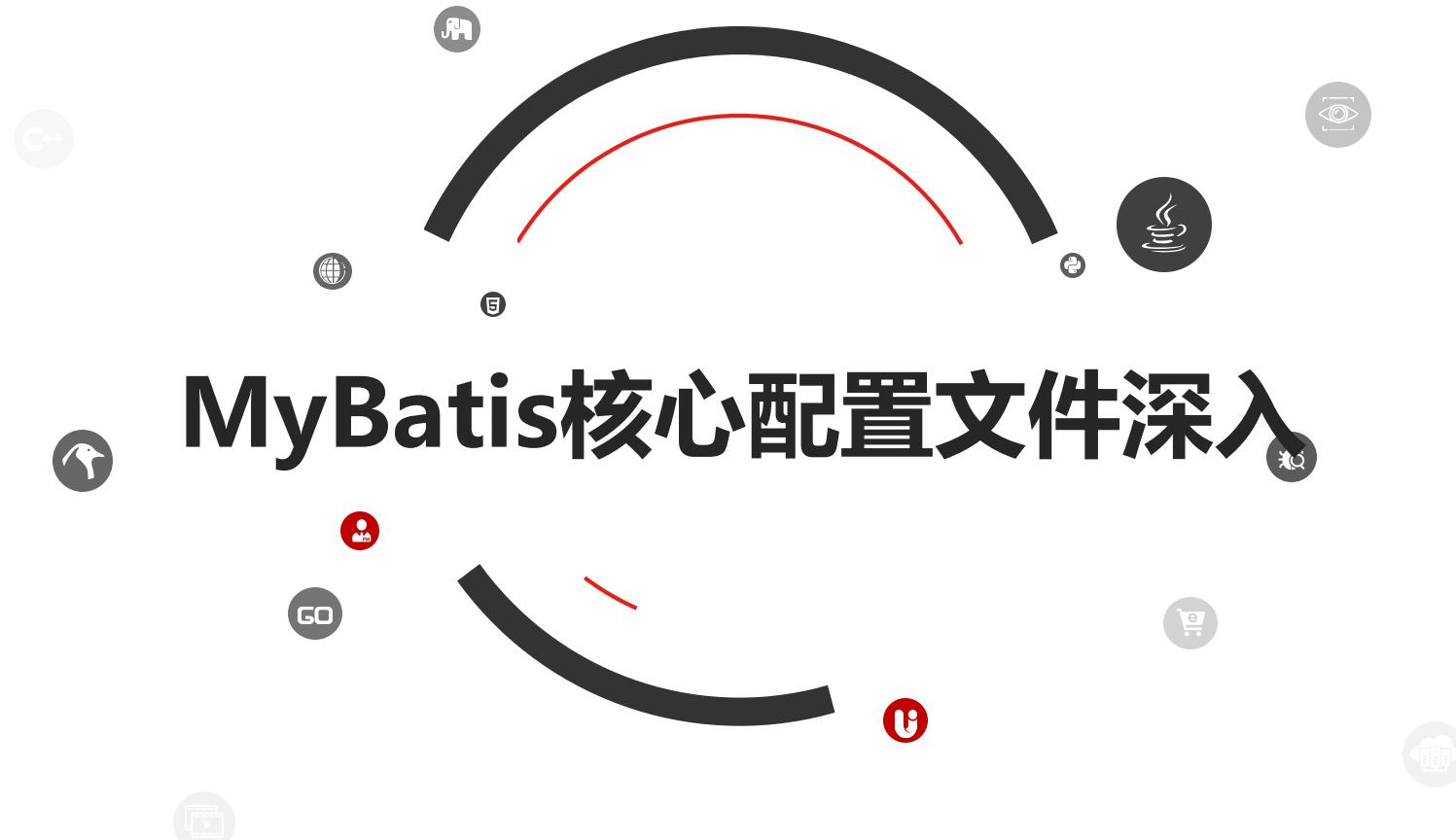
<where>：where条件

<if>：if判断

<foreach>：循环

<sql>：sql片段抽取

# MyBatis核心配置文件深入





# 目 錄

# Contents

◆ MyBatis核心配置文件深入

# 1. MyBatis核心配置文件深入

## 1.1 typeHandlers标签

无论是 MyBatis 在预处理语句（PreparedStatement）中设置一个参数时，还是从结果集中取出一个值时，都会用类型处理器将获取的值以合适的方式转换成 Java 类型。下表描述了一些默认的类型处理器（截取部分）。

类型处理器	Java 类型	JDBC 类型
BooleanTypeHandler	java.lang.Boolean, boolean	数据库兼容的 BOOLEAN
ByteTypeHandler	java.lang.Byte, byte	数据库兼容的 NUMERIC 或 BYTE
ShortTypeHandler	java.lang.Short, short	数据库兼容的 NUMERIC 或 SHORT INTEGER
IntegerTypeHandler	java.lang.Integer, int	数据库兼容的 NUMERIC 或 INTEGER
LongTypeHandler	java.lang.Long, long	数据库兼容的 NUMERIC 或 LONG INTEGER

# 1. MyBatis核心配置文件深入

## 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

# 1. MyBatis核心配置文件深入

## 1.1 typeHandlers标签

```
public class MyDateTypeHandler extends BaseTypeHandler<Date> {
    public void setNonNullParameter(PreparedStatement preparedStatement, int i, Date date, JdbcType type) {
        preparedStatement.setString(i, date.getTime() + " ");
    }
    public Date getNullableResult(ResultSet resultSet, String s) throws SQLException {
        return new Date(resultSet.getLong(s));
    }
    public Date getNullableResult(ResultSet resultSet, int i) throws SQLException {
        return new Date(resultSet.getLong(i));
    }
    public Date getNullableResult(CallableStatement callableStatement, int i) throws SQLException {
        return callableStatement.getDate(i);
    }
}
```

# 1. MyBatis核心配置文件深入

## 1.1 typeHandlers标签

```
<!--注册类型自定义转换器-->  
<typeHandlers>  
    <typeHandler handler="com.itheima.typeHandlers.MyDateTypeHandler"></typeHandler>  
</typeHandlers>
```

测试添加操作：

```
user.setBirthday(new Date());  
userMapper.add2(user);
```

数据库数据：

id	username	password	birthday
1	lucy	123	1539751863457
2	tom	123	1539751863457
5	haohao	123	1539751863457

# 1. MyBatis核心配置文件深入

## 1.1 typeHandlers标签

测试查询操作：

```
13:53:10,222 DEBUG findAll:159 - <==      Total: 9
[User{id=1, username='lucy', password='123', birthday=Wed Oct 17 12:51:03 GMT+08:00 2018}]
13:53:10,222 DEBUG JdbcTransaction:123 - Resetting autocommit to true on JDBC Connection
13:53:10,222 DEBUG JdbcTransaction:91 - Closing JDBC Connection [com.mysql.jdbc.JDBC4Conne
13:53:10,222 DEBUG PooledDataSource:363 - Returned connection 249155636 to pool.
```

# 1. MyBatis核心配置文件深入

## 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. MyBatis核心配置文件深入

## 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. MyBatis核心配置文件深入

## 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>
```

# 1. MyBatis核心配置文件深入

## 1.2 plugins标签

③ 测试分页代码实现

```
@Test  
public void testPageHelper(){  
    //设置分页参数  
    PageHelper.startPage(1,2);  
  
    List<User> select = userMapper2.select(null);  
    for(User user : select){  
        System.out.println(user);  
    }  
}
```

# 1. MyBatis核心配置文件深入

## 1.2 plugins标签

获得分页相关的其他参数

```
//其他分页的数据
PageInfo<User> pageInfo = new PageInfo<User>(select);
System.out.println("总条数: "+pageInfo.getTotal());
System.out.println("总页数: "+pageInfo.getPages());
System.out.println("当前页: "+pageInfo.getPageNum());
System.out.println("每页显示长度: "+pageInfo.getPageSize());
System.out.println("是否第一页: "+pageInfo.isIsFirstPage());
System.out.println("是否最后一页: "+pageInfo.isIsLastPage());
```

# 1. MyBatis核心配置文件深入

## 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的多表操作

# 目 录

# Contents

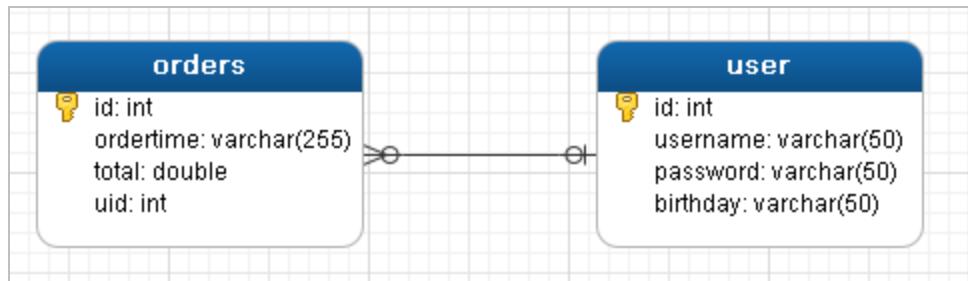
◆ MyBatis的多表操作

# 1. Mybatis多表查询

## 1.1 一对查询

### 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



# 1. Mybatis多表查询

## 1.1 一对—查询

### 2. 一对—查询的语句

对应的sql语句：select \* from orders o,user u where o.uid=u.id;

查询的结果如下：

信息	结果1	概况	状态					
	id	ordertime	total	uid	id1	username	password	birthday
▶	1	2018-12-12	3000	1	1	lucy	123	1539751863457
	2	2019-12-12	4000	1	1	lucy	123	1539751863457
	3	2020-12-12	5000	2	2	tom	123	1539751863457

# 1. Mybatis多表查询

## 1.1 一对查询

### 3. 创建Order和User实体

```
public class Order {  
  
    private int id;  
    private Date ordertime;  
    private double total;  
  
    //代表当前订单从属于哪一个客户  
    private User user;  
}
```

```
public class User {  
  
    private int id;  
    private String username;  
    private String password;  
    private Date birthday;  
}
```

# ■ 1.Mybatis多表查询

## 1.1 一对查询

### 4. 创建OrderMapper接口

```
public interface OrderMapper {  
    List<Order> findAll();  
}
```

# 1. Mybatis多表查询

## 1.1 一对查询

### 5. 配置OrderMapper.xml

```
<mapper namespace="com.itheima.mapper.OrderMapper">
    <resultMap id="orderMap" type="com.itheima.domain.Order">
        <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 * from orders o,user u where o.uid=u.id
    </select>
</mapper>
```

自己定义一个map来封装某个对象，并给这个map起个名字方便下面sql语句调用，这样做是为了下面sql语句封装时能识别清楚

# 1. Mybatis多表查询

## 1.1 一对查询

### 5. 配置OrderMapper.xml

其中<resultMap>还可以配置如下：

```
<resultMap id="orderMap" type="com.itheima.domain.Order">
    <result property="id" column="id"></result>
    <result property="ordertime" column="ordertime"></result>
    <result property="total" column="total"></result>
    <association property="user" javaType="com.itheima.domain.User">
        <result column="uid" property="id"></result>
        <result column="username" property="username"></result>
        <result column="password" property="password"></result>
        <result column="birthday" property="birthday"></result>
    </association>
</resultMap>
```

# 1. Mybatis多表查询

## 1.1 一对查询

### 6. 测试结果

```
OrderMapper mapper = sqlSession.getMapper(OrderMapper.class);
List<Order> all = mapper.findAll();
for(Order order : all){
    System.out.println(order);
}
```

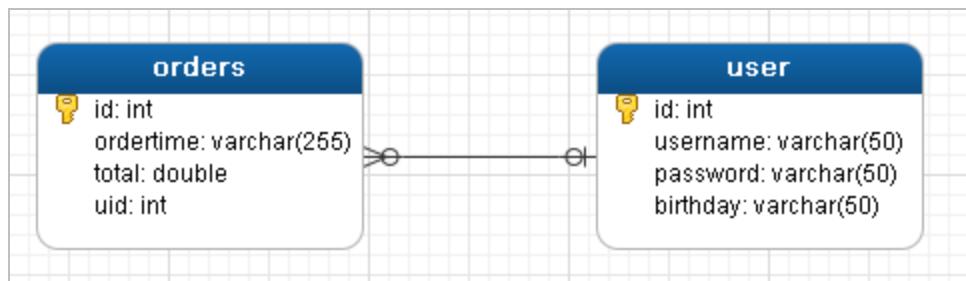
```
09:12:24,650 DEBUG findAll:54 - ==> Preparing: select * from orders o,user u where o.uid=u.id
09:12:24,672 DEBUG findAll:54 - ==> Parameters:
09:12:24,699 DEBUG findAll:54 - <== Total: 3
Order{id=1, ordertime=Wed Dec 12 00:00:00 GMT+08:00 2018, total=3000.0, user=User{id=1, username='lucy'},
Order{id=2, ordertime=Thu Dec 12 00:00:00 GMT+08:00 2019, total=4000.0, user=User{id=1, username='lucy'},
Order{id=3, ordertime=Sat Dec 12 00:00:00 GMT+08:00 2020, total=5000.0, user=User{id=2, username='tom'},
09:12:24,706 DEBUG JdbcTransaction:54 - Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.
09:12:24,706 DEBUG JdbcTransaction:54 - Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@28ac3dc3
09:12:24,706 DEBUG PooledDataSource:54 - Returned connection 682376643 to pool.
```

# 1. Mybatis多表查询

## 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



# 1. Mybatis多表查询

## 1.2 一对多查询

### 2. 一对多查询的语句

对应的sql语句：select \*,o.id oid from user u left join orders o on u.id=o.uid;

查询的结果如下：

信息	结果1	概况	状态							
	id	username	password	birthday	id1	ordertime	total	uid	oid	
▶	1	lucy	123	2018-12-12	1	2018-12-12	3000	1	1	
	1	lucy	123	2018-12-12	2	2019-12-12	4000	1	2	
	2	tom	123	2018-12-12	3	2020-12-12	5000	2	3	
	5	haohao	123	2018-12-12	(Null)	(Null)	(Null)	(Null)	(Null)	

# 1. Mybatis多表查询

## 1.2 一对多查询

### 3. 修改User实体

```
public class Order {  
  
    private int id;  
    private Date ordertime;  
    private double total;  
  
    //代表当前订单从属于哪一个客户  
    private User user;  
}
```

```
public class User {  
  
    private int id;  
    private String username;  
    private String password;  
    private Date birthday;  
    //代表当前用户具备哪些订单  
    private List<Order> orderList;  
}
```

# 1. Mybatis多表查询

## 1.2 一对多查询

### 4. 创建UserMapper接口

```
public interface UserMapper {  
    List<User> findAll();  
}
```

# 1. Mybatis多表查询

## 1.2 一对多查询

### 5. 配置UserMapper.xml

```
<mapper namespace="com.itheima.mapper.UserMapper">

    <resultMap id="userMap" type="com.itheima.domain.User">
        <result column="id" property="id"></result>
        <result column="username" property="username"></result>
        <result column="password" property="password"></result>
        <result column="birthday" property="birthday"></result>
        <collection property="orderList" ofType="com.itheima.domain.Order">
            <result column="oid" property="id"></result>
            <result column="ordertime" property="ordertime"></result>
            <result column="total" property="total"></result>
        </collection>
    </resultMap>
    <select id="findAll" resultMap="userMap">
        select *,o.id oid from user u left join orders o on u.id=o.uid
    </select>
</mapper>
```

# 1. Mybatis多表查询

## 1.2 一对多查询

### 6. 测试结果

```
UserMapper mapper = sqlSession.getMapper(UserMapper.class);
List<User> all = mapper.findAll();
for(User user : all){
    System.out.println(user.getUsername());
    List<Order> orderList = user.getOrderList();
    for(Order order : orderList){
        System.out.println(order);
    }
    System.out.println("-----");
}
```

# 1. Mybatis多表查询

## 1.2 一对多查询

### 6. 测试结果

```
10:02:27,817 DEBUG findAll:54 - ==> Preparing: select *,o.id oid from user u left join orders o on u.id=o.uid
10:02:27,843 DEBUG findAll:54 - ==> Parameters:
10:02:27,865 DEBUG findAll:54 - <==      Total: 4
lucy
Order{id=1, ordertime=Wed Dec 12 00:00:00 GMT+08:00 2018, total=3000.0, user=null}
Order{id=2, ordertime=Thu Dec 12 00:00:00 GMT+08:00 2019, total=4000.0, user=null}
-----
tom
Order{id=3, ordertime=Sat Dec 12 00:00:00 GMT+08:00 2020, total=5000.0, user=null}
-----
haohao
-----
10:02:27,868 DEBUG JdbcTransaction:54 - Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@289d1c02]
10:02:27,869 DEBUG JdbcTransaction:54 - Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@289d1c02]
10:02:27,869 DEBUG PooledDataSource:54 - Returned connection 681384962 to pool.
```

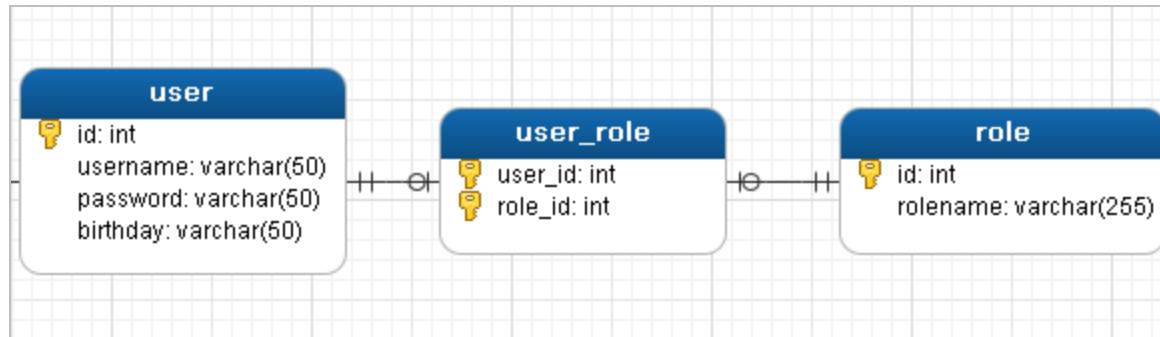
# 1. Mybatis多表查询

## 1.3 多对多查询

### 1. 多对多查询的模型

用户表和角色表的关系为，一个用户有多个角色，一个角色被多个用户使用

多对多查询的需求：查询用户同时查询出该用户的所有角色



# 1. Mybatis多表查询

## 1.3 多对多查询

### 2. 多对多查询的语句

对应的sql语句： select u.\* , r.\* , r.id rid from user u left join user\_role ur on u.id=ur.user\_id  
inner join role r on ur.role\_id=r.id;

查询的结果如下：

信息						
	id	username	password	birthday	id1	rolename
▶	1	lucy	123	2018-12-12	1	CEO
	1	lucy	123	2018-12-12	2	CFO
	2	tom	123	2018-12-12	2	CFO
	2	tom	123	2018-12-12	3	COO

# 1. Mybatis多表查询

## 1.3 多对多查询

### 3. 创建Role实体，修改User实体

```
public class User {  
    private int id;  
    private String username;  
    private String password;  
    private Date birthday;  
    //代表当前用户具备哪些订单  
    private List<Order> orderList;  
    //代表当前用户具备哪些角色  
    private List<Role> roleList;  
}
```

```
public class Role {  
  
    private int id;  
    private String rolename;  
}
```

# 1. Mybatis多表查询

## 1.3 多对多查询

### 4. 添加UserMapper接口方法

```
List<User> findAllUserAndRole();
```

# 1. Mybatis多表查询

## 1.3 多对多查询

### 5. 配置UserMapper.xml

```
<resultMap id="userRoleMap" type="com.itheima.domain.User">
    <result column="id" property="id"></result>
    <result column="username" property="username"></result>
    <result column="password" property="password"></result>
    <result column="birthday" property="birthday"></result>
    <collection property="roleList" ofType="com.itheima.domain.Role">
        <result column="rid" property="id"></result>
        <result column="rolename" property="rolename"></result>
    </collection>
</resultMap>
<select id="findAllUserAndRole" resultMap="userRoleMap">
    select u.*,r.* ,r.id rid from user u left join user_role ur on
    u.id=ur.user_id
    inner join role r on ur.role_id=r.id
</select>
```

# 1. Mybatis多表查询

## 1.3 多对多查询

### 6. 测试结果

```
UserMapper mapper = sqlSession.getMapper(UserMapper.class);
List<User> all = mapper.findAllUserAndRole();
for(User user : all){
    System.out.println(user.getUsername());
    List<Role> roleList = user.getRoleList();
    for(Role role : roleList){
        System.out.println(role);
    }
    System.out.println("-----");
}
```

# 1. Mybatis多表查询

## 1.3 多对多查询

### 6. 测试结果

```
10:34:36,884 DEBUG findAllUserAndRole:54 - ==> Preparing: select u.*,r.* ,r.id rid from user u left
10:34:36,903 DEBUG findAllUserAndRole:54 - ==> Parameters:
lucy
Role{id=1, rolename='CEO'}
Role{id=2, rolename='CFO'}
-----
tom
Role{id=2, rolename='CFO'}
Role{id=3, rolename='COO'}
-----
10:34:36,937 DEBUG findAllUserAndRole:54 - <==      Total: 4
10:34:36,939 DEBUG JdbcTransaction:54 - Resetting autocommit to true on JDBC Connection [com.mysql.]
```

# 1. Mybatis多表查询

## 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



# MyBatis注解开发

# 目 录

# Contents

◆ MyBatis的注解开发

# 1. Mybatis的注解开发

## 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: 实现一对结果集封装

@Many: 实现一对多结果集封装

# 1. Mybatis的注解开发

## 1.2 MyBatis的增删改查

我们完成简单的user表的增删改查的操作

```
private UserMapper userMapper;

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

# 1.Mybatis的注解开发

## 1.2 MyBatis的增删改查

```
@Test
public void testAdd() {
    User user = new User();
    user.setUsername("测试数据");
    user.setPassword("123");
    user.setBirthday(new Date());
    userMapper.add(user);
}

@Test
public void testUpdate() throws IOException {
    User user = new User();
    user.setId(16);
    user.setUsername("测试数据修改");
    user.setPassword("abc");
    user.setBirthday(new Date());
    userMapper.update(user);
}
```

# 1. Mybatis的注解开发

## 1.2 MyBatis的增删改查

```
@Test
public void testDelete() throws IOException {
    userMapper.delete(16);
}

@Test
public void testFindById() throws IOException {
    User user = userMapper.findById(1);
    System.out.println(user);
}

@Test
public void testfindAll() throws IOException {
    List<User> all = userMapper.findAll();
    for(User user : all) {
        System.out.println(user);
    }
}
```

# 1. Mybatis的注解开发

## 1.2 MyBatis的增删改查

修改MyBatis的核心配置文件，我们使用了注解替代的映射文件，所以我们只需要加载使用了注解的Mapper接口即可

```
<mappers>
    <!--扫描使用注解的类-->
    <mapper class="com.itheima.mapper.UserMapper"></mapper>
</mappers>
```

或者指定扫描包含映射关系的接口所在的包也可以

```
<mappers>
    <!--扫描使用注解的类所在的包-->
    <package name="com.itheima.mapper"></package>
</mappers>
```

# 1. Mybatis的注解开发

## 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 () )
@Result	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) () )

# 1. Mybatis的注解开发

## 1.3 MyBatis的注解实现复杂映射开发

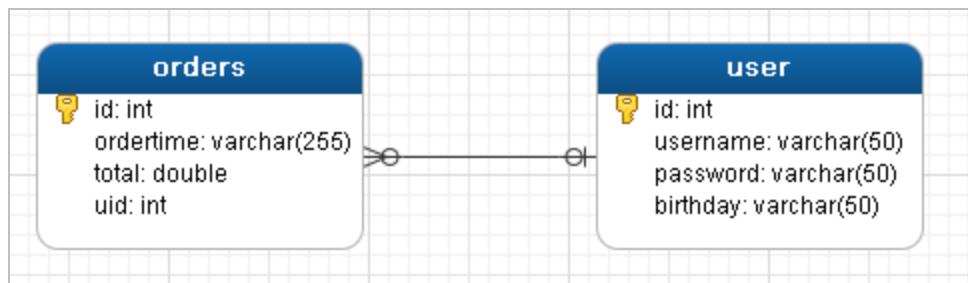
注解	说明
@One (一对一)	<p>代替了&lt;assocation&gt; 标签，是多表查询的关键，在注解中用来指定子查询返回单一对象。</p> <p>@One注解属性介绍：</p> <p>select: 指定用来多表查询的 sqlmapper</p> <p>使用格式: @Result(column=" ",property="",one=@One(select=""))</p>
@Many (多对一)	<p>代替了&lt;collection&gt;标签，是多表查询的关键，在注解中用来指定子查询返回对象集合。</p> <p>使用格式: @Result(property="",column=" ",many= @Many(select=""))</p>

# 1. Mybatis的注解开发

## 1.4 一对查询

### 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



# 1. Mybatis的注解开发

## 1.4 一对查询

### 2. 一对一查询的语句

对应的sql语句：

```
select * from orders;  
select * from user where id=查询出订单的uid;
```

查询的结果如下：

信息								
	id	ordertime	total	uid	id1	username	password	birthday
▶	1	2018-12-12	3000	1	1	lucy	123	1539751863457
	2	2019-12-12	4000	1	1	lucy	123	1539751863457
	3	2020-12-12	5000	2	2	tom	123	1539751863457

# 1. Mybatis的注解开发

## 1.4 一对查询

### 3. 创建Order和User实体

```
public class Order {  
  
    private int id;  
    private Date ordertime;  
    private double total;  
  
    //代表当前订单从属于哪一个客户  
    private User user;  
}
```

```
public class User {  
  
    private int id;  
    private String username;  
    private String password;  
    private Date birthday;  
}
```

# 1. Mybatis的注解开发

## 1.4 一对—查询

### 4. 创建OrderMapper接口

```
public interface OrderMapper {  
    List<Order> findAll();  
}
```

# 1. Mybatis的注解开发

## 1.4 一对查询

### 5. 使用注解配置Mapper

```
public interface OrderMapper {  
  
    @Select("select * from orders")  
  
    @Results({  
  
        @Result(id=true,property = "id",column = "id"),  
        @Result(property = "ordertime",column = "ordertime"),  
        @Result(property = "total",column = "total"),  
        @Result(property = "user",column = "uid",  
               javaType = User.class,  
               one = @One(select =  
"com.itheima.mapper.UserMapper.findById"))  
    })  
  
    List<Order> findAll();  
}
```

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

# 1. Mybatis的注解开发

## 1.4 一对查询

### 6. 测试结果

```
@Test
public void testSelectOrderAndUser() {
    List<Order> all = orderMapper.findall();
    for(Order order : all){
        System.out.println(order);
    }
}
```

```
12:18:29,699 DEBUG findById:54 - =====> Preparing: select * from user where id=?
12:18:29,699 DEBUG findById:54 - =====> Parameters: 2(Integer)
12:18:29,701 DEBUG findById:54 - <===== Total: 1
12:18:29,701 DEBUG findAll:54 - <== Total: 3
Order{id=1, ordertime=Wed Dec 12 00:00:00 GMT+08:00 2018, total=3000.0, user=User{id=1, username='lucy'},
Order{id=2, ordertime=Thu Dec 12 00:00:00 GMT+08:00 2019, total=4000.0, user=User{id=1, username='lucy'},
Order{id=3, ordertime=Sat Dec 12 00:00:00 GMT+08:00 2020, total=5000.0, user=User{id=2, username='tom'}}
```

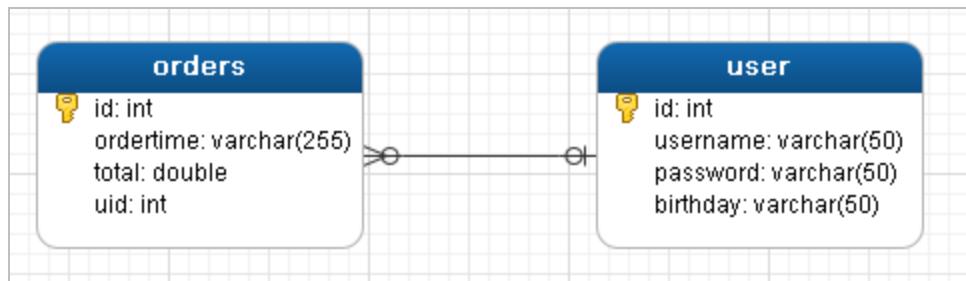
# 1. Mybatis的注解开发

## 1.5 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



# 1. Mybatis的注解开发

## 1.5 一对多查询

### 2. 一对多查询的语句

对应的sql语句：

```
select * from user;  
select * from orders where uid=查询出用户的id;
```

查询的结果如下：

信息									
	结果1		概况		状态				
	id	username	password	birthday	id1	ordertime	total	uid	oid
▶	1	lucy	123	2018-12-12	1	2018-12-12	3000	1	1
	1	lucy	123	2018-12-12	2	2019-12-12	4000	1	2
	2	tom	123	2018-12-12	3	2020-12-12	5000	2	3
	5	haohao	123	2018-12-12	(Null)	(Null)	(Null)	(Null)	(Null)

# 1. Mybatis的注解开发

## 1.5 一对多查询

### 3. 修改User实体

```
public class Order {  
  
    private int id;  
    private Date ordertime;  
    private double total;  
  
    //代表当前订单从属于哪一个客户  
    private User user;  
}
```

```
public class User {  
  
    private int id;  
    private String username;  
    private String password;  
    private Date birthday;  
    //代表当前用户具备哪些订单  
    private List<Order> orderList;  
}
```

# 1. Mybatis的注解开发

## 1.5 一对多查询

### 4. 创建UserMapper接口

```
List<User> findAllUserAndOrder();
```

# 1. Mybatis的注解开发

## 1.5 一对多查询

相当于select嵌套select

### 5. 使用注解配置Mapper

```
public interface UserMapper {  
    @Select("select * from user")  
    @Results({  
        @Result(id = true, property = "id", column = "id"),  
        @Result(property = "username", column = "username"),  
        @Result(property = "password", column = "password"),  
        @Result(property = "birthday", column = "birthday"),  
        @Result(property = "orderList", column = "id",  
               javaType = List.class,  
               many = @Many(select =  
                           "com.itheima.mapper.OrderMapper.findByUid"))  
    })  
    List<User> findAllUserAndOrder();  
}
```

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

# 1. Mybatis的注解开发

## 1.5 一对多查询

### 6. 测试结果

```
List<User> all = userMapper.findAllUserAndOrder();
for(User user : all){
    System.out.println(user.getUsername());
    List<Order> orderList = user.getOrderList();
    for(Order order : orderList){
        System.out.println(order);
    }
    System.out.println("-----");
}
```

# 1. Mybatis的注解开发

## 1.5 一对多查询

### 6. 测试结果

```
14:32:14,813 DEBUG findAllUserAndOrder:54 - ==> Preparing: select * from user
14:32:14,844 DEBUG findAllUserAndOrder:54 - ==> Parameters:
14:32:14,860 DEBUG findByUid:54 - ====> Preparing: select * from orders where uid=?
lucy
Order{id=1, ordertime=Wed Dec 12 00:00:00 GMT+08:00 2018, total=3000.0, user=null}
Order{id=2, ordertime=Thu Dec 12 00:00:00 GMT+08:00 2019, total=4000.0, user=null}
-----
tom
Order{id=3, ordertime=Sat Dec 12 00:00:00 GMT+08:00 2020, total=5000.0, user=null}
-----
haohao
-----
```

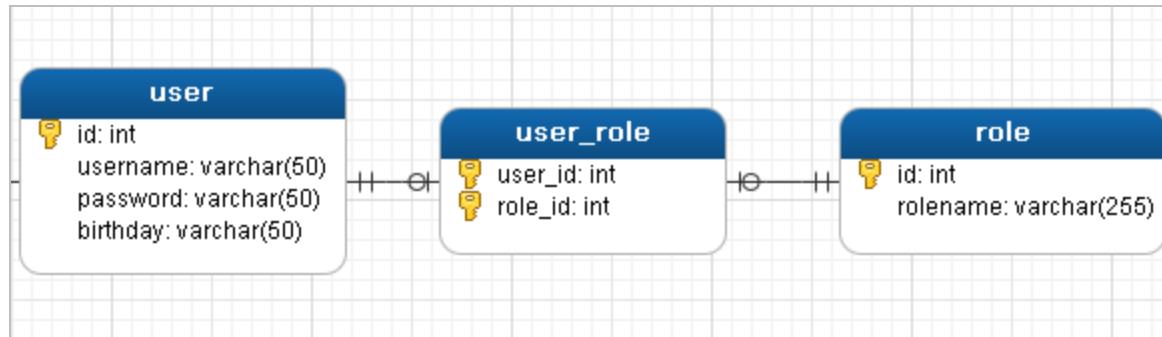
# 1. Mybatis的注解开发

## 1.6 多对多查询

### 1. 多对多查询的模型

用户表和角色表的关系为，一个用户有多个角色，一个角色被多个用户使用

多对多查询的需求：查询用户同时查询出该用户的所有角色



# 1. Mybatis的注解开发

## 1.6 多对多查询

### 2. 多对多查询的语句

对应的sql语句：

```
select * from user;
```

```
select * from role r,user_role ur where r.id=ur.role_id and ur.user_id=用户的id
```

查询的结果如下：

信息	结果1	概况	状态		
id	username	password	birthday	id1	rolename
▶	1 lucy	123	2018-12-12	1	CEO
	1 lucy	123	2018-12-12	2	CFO
	2 tom	123	2018-12-12	2	CFO
	2 tom	123	2018-12-12	3	COO

# 1. Mybatis的注解开发

## 1.6 多对多查询

### 3. 创建Role实体，修改User实体

```
public class User {  
    private int id;  
    private String username;  
    private String password;  
    private Date birthday;  
    //代表当前用户具备哪些订单  
    private List<Order> orderList;  
    //代表当前用户具备哪些角色  
    private List<Role> roleList;  
}
```

```
public class Role {  
  
    private int id;  
    private String rolename;  
}
```

# 1. Mybatis多表查询

## 1.6 多对多查询

### 4. 添加UserMapper接口方法

```
List<User> findAllUserAndRole();
```

# 1. Mybatis的注解开发

## 1.6 多对多查询

### 5. 使用注解配置Mapper

```
public interface UserMapper {  
    @Select("select * from user")  
    @Results({  
        @Result(id = true, property = "id", column = "id"),  
        @Result(property = "username", column = "username"),  
        @Result(property = "password", column = "password"),  
        @Result(property = "birthday", column = "birthday"),  
        @Result(property = "roleList", column = "id",  
               javaType = List.class,  
               many = @Many(select =  
                           "com.itheima.mapper.RoleMapper.findByUid"))  
    })  
    List<User> findAllUserAndRole();  
}
```

```
public interface RoleMapper {  
    @Select("select * from role  
            r,user_role ur where  
            r.id=ur.role_id and  
            ur.user_id=#{uid}")  
    List<Role> findByUid(int uid);  
}
```

# 1. Mybatis的注解开发

## 1.6 多对多查询

### 6. 测试结果

```
UserMapper mapper = sqlSession.getMapper(UserMapper.class);
List<User> all = mapper.findAllUserAndRole();
for(User user : all){
    System.out.println(user.getUsername());
    List<Role> roleList = user.getRoleList();
    for(Role role : roleList){
        System.out.println(role);
    }
    System.out.println("-----");
}
```

# 1. Mybatis的注解开发

## 1.6 多对多查询

### 6. 测试结果

```
14:52:12,823 DEBUG findAllUserAndRole:54 - ==> Preparing: select * from user
14:52:12,854 DEBUG findAllUserAndRole:54 - ==> Parameters:
14:52:12,870 DEBUG findByUid:54 - ====> Preparing: select * from role r,user_role ur where r.id=ur.role_id
14:52:12,870 DEBUG findByUid:54 - ====> Parameters: 1(Integer)
14:52:12,870 DEBUG findByUid:54 - <==== Total: 2
14:52:12,870 DEBUG findByUid:54 - ====> Preparing: select * from role r,user_role ur where r.id=ur.role_id
14:52:12,870 DEBUG findByUid:54 - ====> Parameters: 2(Integer)
14:52:12,870 DEBUG findByUid:54 - <==== Total: 2
14:52:12,870 DEBUG findByUid:54 - ====> Preparing: select * from role r,user_role ur where r.id=ur.role_id
14:52:12,870 DEBUG findByUid:54 - ====> Parameters: 5(Integer)
14:52:12,885 DEBUG findByUid:54 - <==== Total: 0
lucy
Role{id=1, rolename='CEO'}
Role{id=2, rolename='CFO'}
-----
tom
Role{id=2, rolename='CFO'}
Role{id=3, rolename='COO'}
-----
haohao
-----
```



# SSM整合



# 目 录

# Contents

◆ SSM框架整合

# 1. SSM框架整合

## 1.1准备工作

### 1. 原始方式整合

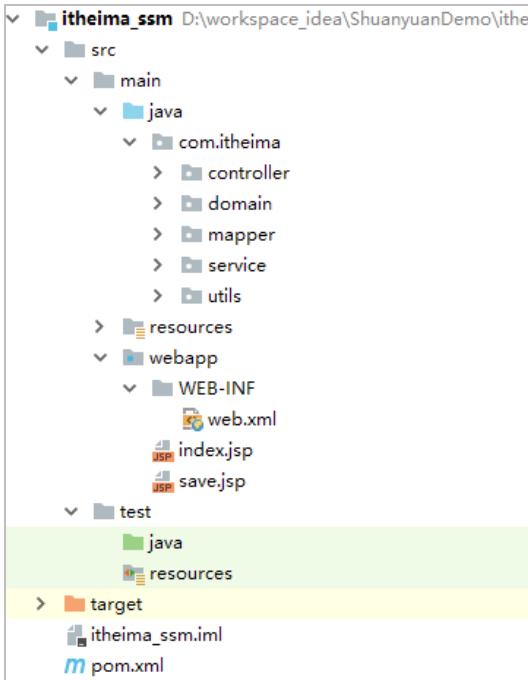
```
create database ssm;
create table account(
    id int primary key auto_increment,
    name varchar(100),
    money double(7,2)
);
```

	<b>id</b>	<b>name</b>	<b>money</b>
	1	tom	5000
	2	lucy	5000

# 1. SSM框架整合

## 1.1 原始方式整合

### 2. 创建Maven工程



# ■ 1. SSM框架整合

## 1.1 原始方式整合

### 3. 导入Maven坐标

[点击打开坐标内容](#)

# 1. SSM框架整合

## 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. SSM框架整合

## 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. SSM框架整合

## 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. SSM框架整合

## 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. SSM框架整合

## 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. SSM框架整合

## 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. SSM框架整合

## 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. SSM框架整合

## 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. SSM框架整合

## 1.1 原始方式整合

### 12. 测试添加账户

保存账户信息表单

用户名

账户金额

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

# 1. SSM框架整合

## 1.1 原始方式整合

### 13. 测试账户列表



The screenshot shows a web browser window with the following details:

- Address bar: localhost:8080/itheima\_ssm/findAll.action (highlighted with a red box).
- Toolbar icons: 应用 (Application), 微信 (WeChat), 百度 (Baidu), 邮箱 (Email), TLIAS, OA, 171 TB, Repository, mybatis3.
- Main content area:

### 账户列表

账户id	账户名称	账户金额
1	tom	5000.0
2	lucy	5000.0
4	zhangsan	1000.0
5	zhangsan11	1000.0
6	测试数据	10000.0

# 1. SSM框架整合

## 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

# 1. SSM框架整合

## 1.2 Spring整合MyBatis

### 2. 将SqlSessionFactory配置到Spring容器中

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

# 1. SSM框架整合

## 1.2 Spring整合MyBatis

### 3. 扫描Mapper，让Spring容器产生Mapper实现类

```
<!--配置Mapper扫描-->
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
    <property name="basePackage" value="com.itheima.mapper"/>
</bean>
```

# 1. SSM框架整合

## 1.2 Spring整合MyBatis

### 4. 配置声明式事务控制

```
<!--配置声明式事务控制-->

<bean id="transacionManager"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>

<tx:advice id="txAdvice" transaction-manager="transacionManager">
  <tx:attributes>
    <tx:method name="*" />
  </tx:attributes>
</tx:advice>

<aop:config>
  <aop:pointcut id="txPointcut" expression="execution(*
com.itheima.service.impl.*.*(..))"/>
  <aop:advisor advice-ref="txAdvice" pointcut-ref="txPointcut"/>
</aop:config>
```

# 1. SSM框架整合

## 1.2 Spring整合MyBatis

### 5.修改Service实现类代码

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

    @Autowired
    private AccountMapper accountMapper;

    public void save(Account account) {
        accountMapper.save(account);
    }
    public List<Account> findAll() {
        return accountMapper.findAll();
    }
}
```

