

课堂主题

Mybatis基于XML和注解方式的开发应用专题

课堂目标

- 主键返回（mybatis的自增主键或者非自增主键）
- 批量查询
- 动态传参
- 查询缓存（
• 、二级缓存）
- 延迟加载（侵入式延迟加载、深度延迟加载）
- 关联查询（一对一、一对映射）
- 逆向工程
- PageHelper分页插件
- 注解开发

知识要点

[TOC]

介绍篇

认识自己

开发人员 OR 研发人员？

主要是使用成熟的框架去开发应用功能，还是使用JavaEE、JVM、并发编程、NIO/Netty等知识点实现编写自定义框架或者解决高并发场景下的非功能性需求，比如如何提高并发能力等？

如何进行接下来的学习呢？

他山之石，可以攻玉！！学习人家的框架，写出自己的框架。

认识框架

什么是框架

摘自百度：

可以说，一个框架是一个可复用的设计构件，它规定了应用的体系结构，阐明了整个设计、协作构件之间的依赖关系、责任分配和控制流程，表现为一组抽象类以及其实例之间协作的方法，它为构件复用提供了上下文(Context)关系。因此构件库的大规模重用也需要框架。

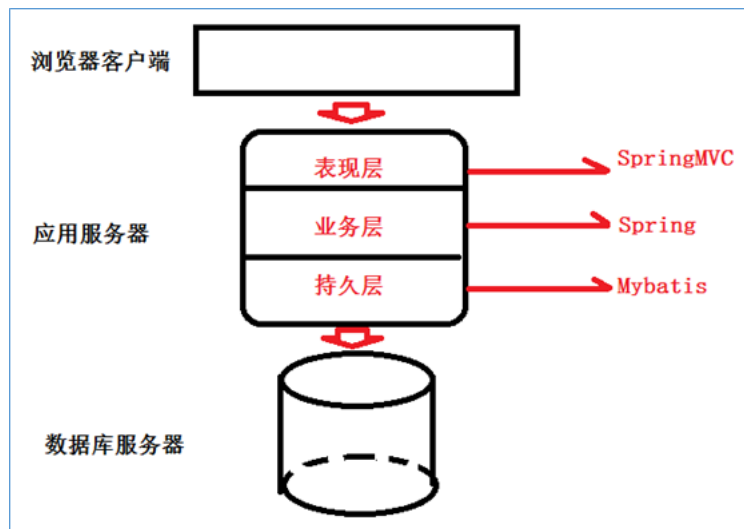
为什么使用框架

- 因为软件系统发展到今天已经很复杂了，特别是服务器端软件，涉及到的知识，内容，问题太多。在某些方面使用别人成熟的框架，就相当于让别人帮你完成一些基础工作，你只需要集中精力完成系统的业务逻辑设计。
- 而且框架一般是成熟，稳健的，它可以处理系统很多细节问题，比如，事务处理，安全性，数据流控制等问题。
- 还有框架一般都经过很多人使用，所以结构很好，所以扩展性也很好，而且它是不断升级的，你可以直接享受别人升级代码带来的好处。

软件开发的三层结构

我们用三层结构主要是使项目结构更清楚，分工更明确，有利于后期的维护和升级。

三层结构包含：表现层，业务层，持久层



认识设计模式

设计模式概述

- 设计模式（Design pattern）代表了最佳的实践，通常被有经验的面向对象的软件开发人员所采用。
- 设计模式是软件开发人员在软件开发过程中面临的一般问题的解决方案。这些解决方案是众多软件开发人员经过相当长的一段时间的试验和错误总结出来的。
- 设计模式是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。
- 设计模式不是一种方法和技术，而是一种思想。
- 设计模式和具体的语言无关，学习设计模式就是要建立面向对象的思想，尽可能的面向接口编程，低耦合，高内聚，使设计的程序可复用。
- 学习设计模式能够促进对面向对象思想的理解，反之亦然。它们相辅相成。

设计模式的类型

总体来说，设计模式分为三类23种：

- 创建型（5种）：工厂模式、抽象工厂模式、单例模式、原型模式、构建者模式
- 结构型（7种）：适配器模式、装饰模式、代理模式、外观模式、桥接模式、组合模式、享元模式
- 行为型（11种）：模板方法模式、策略模式、观察者模式、中介者模式、状态模式、责任链模式、命令模式、迭代器模式、访问者模式、解释器模式、备忘录模式

认识MyBatis

mybatis参考网址：<http://www.mybatis.org/mybatis-3/zh/index.html>

Github源码地址：<https://github.com/mybatis/mybatis-3>

Mybatis是什么

MyBatis 是一款优秀的持久层框架，它支持定制化SQL、存储过程以及高级映射。MyBatis 避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集，它可以使用简单的XML或注解来配置和映射SQL信息，将接口和 Java 的 POJOs(Plain Old Java Objects,普通的 Java对象)映射成数据库中的记录。

Mybatis的由来

- MyBatis 本是apache的一个开源项目iBatis。
- 2010年这个项目由apache software foundation 迁移到了google code，并且改名为MyBatis。
- 2013年11月迁移到Github。

ORM是什么

对象-关系映射（OBJECT/RELATIONALMAPPING，简称ORM），是随着面向对象的软件开发方法发展而产生的。用来把对象模型表示的对象映射到基于SQL的关系模型数据库结构中去。这样，我们在具体的操作实体对象的时候，就不需要再去和复杂的SQL语句打交道，只需简单的操作实体对象的属性和方法。ORM技术是在对象和关系之间提供了一条桥梁，前台的对象型数据和数据库中的关系型的数据通过这个桥梁来相互转化。

ORM框架和MyBatis的区别

| 对比项 | Mybatis | Hibernate | |-----|-----|-----| 市场占有率 | 高 | 高 | 适合的行业 | 互联网 电商 项目 | 传统的(ERP CRM OA) | 性能 | 高 | 低 | Sql灵活性 | 高 | 低 | 学习门槛 | 低 | 高 | Sql配置文件 | 全局配置文件、映射文件 | 全局配置文件、映射文件 | ORM | 半自动化 | 完

入门篇

编码流程

1. 编写全局配置文件: SqlMapConfig.xml
2. 映射文件: xxxMapper.xml
3. 编写dao代码: xxxDao接口、xxxDaoImpl实现类
4. POJO类
5. 单元测试类

需求

- 1、根据用户id查询一个用户信息
- 2、根据用户名称模糊查询用户信息列表
- 3、添加用户

项目搭建

- 创建maven工程: mybatis-demo
- POM文件

```xml org.mybatis mybatis 3.4.6 mysql mysql-connector-java 5.1.35

```
<!-- 单元测试 -->
<dependency>
 <groupId>junit</groupId>
 <artifactId>junit</artifactId>
 <version>4.12</version>
</dependency>
</dependencies>
```

...

- 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">

...

- UserMapper.xml

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"> <map

- PO类

```java public class User {

```
private int id;
private String username;
private Date birthday;
private String sex;
private String address;
// getter\setter方法
```

} ```

## 需求实现

查询用户

映射文件

```xml



配置说明：

`` - parameterType: 定义输入参数的Java类型，

- resultType: 定义结果映射类型。

• **{}**: 相当于JDBC中的? 占位符

• **{id}**表示使用**preparedstatement**设置占位符号并将输入变量id传到sql。

- \${value}: 取出参数名为value的值。将\${value}占位符替换。

注意：如果是取简单数量类型的参数，括号中的参数名称必须为value ``

dao接口和实现类

```
java public interface UserDao { public User findUserById(int id) throws Exception; public List<User> findUsersByName(String name) throws Exception; }
```

- 生命周期（作用范围）
- sqlSession: 方法级别
- sqlSessionfactory: 全局范围（应用级别）
- sqlSessionfactoryBuilder: 方法级别

```
``java public class UserDaoImpl implements UserDao { //注入SqlSessionFactory public UserDaoImpl(SqlSessionFactory sqlSessionFactory){ this.sqlSessionFactory = sqlSessionFactory; }
```

```
private SqlSessionFactory sqlSessionFactory;

@Override
public User findUserById(int id) throws Exception {
    SqlSession session = sqlSessionFactory.openSession();
    User user = null;
    try {
        //通过sqlsession调用selectOne方法获取一条结果集
        //参数1: 指定定义的statement的id,参数2: 指定向statement中传递的参数
        user = session.selectOne("test.findUserById", id);
        System.out.println(user);
    } finally{
        session.close();
    }
    return user;
}

@Override
public List<User> findUsersByName(String name) throws Exception {
    SqlSession session = sqlSessionFactory.openSession();
    List<User> users = null;
    try {
        users = session.selectList("test.findUsersByName", name);
        System.out.println(users);
    } finally{
        session.close();
    }
    return users;
}
```

```
} ``
```

测试代码

```
``java public class MybatisTest {
```

```
private SqlSessionFactory sqlSessionFactory;
```

```

@Before
public void init() throws Exception {
    SqlSessionFactoryBuilder sessionFactoryBuilder = new SqlSessionFactoryBuilder();
    InputStream inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");
    sqlSessionFactory = sessionFactoryBuilder.build(inputStream);
}

@Test
public void testFindUserById() {
    UserDao userDao = new UserDaoImpl(sqlSessionFactory);
    User user = userDao.findUserById(22);
    System.out.println(user);
}

@Test
public void testFindUsersByName() {
    UserDao userDao = new UserDaoImpl(sqlSessionFactory);
    List<User> users = userDao.findUsersByName("老郭");
    System.out.println(users);
}
}

```

```

}'''

```

#{}和\${}区别

- 区别1

```

...

```

{ }：相当于JDBC SQL语句中的占位符? (PreparedStatement)

`${}`：相当于JDBC SQL语句中的连接符合 + (Statement)'''

- 区别2

```

...

```

`{ }`：进行输入映射的时候，会对参数进行类型解析（如果是**String**类型，那么**SQL**语句会自动加上”）

`${}`：进行输入映射的时候，将参数原样输出到SQL语句中'''

- 区别3

```

...

```

`{ }`：如果进行简单类型（**String**、**Date**、8种基本类型的包装类）的输入映射时，`#{}` 中参数名称可以任意

`${}`：如果进行简单类型（**String**、**Date**、8种基本类型的包装类）的输入映射时，`${}`中参数名称必须是value'''

- 区别4

`${}`：存在SQL注入问题，使用**OR 1=1** 关键字将查询条件忽略

添加用户

`#{}` ：是通过反射获取数据的---**StaticSqlSource**

`${}`：是通过**OGNL**表达式会随着对象的嵌套而相应地发生层级变化 --**DynamicSqlSource**

映射文件

```

'''xml

```

```

<insert id="insertUser" parameterType="com.kkb.mybatis.po.User">
    insert into user(username,birthday,sex,address)
    values(#{username},#{birthday},#{sex},#{address})
</insert>

```

...

dao接口和实现类

```
java public interface UserDao { public void insertUser(User user) throws Exception; }
```

```
'''java public class UserDaoImpl implements UserDao { //注入SqlSessionFactory public UserDaoImpl(SqlSessionFactory sqlSessionFactory){ this.sqlSessionFactory = sqlSessionFactory; }
```

```
private SqlSessionFactory sqlSessionFactory;

@Override
public void insertUser(User user) throws Exception {
    SqlSession sqlSession = sqlSessionFactory.openSession();
    try {
        sqlSession.insert("test.insertUser", user);
        sqlSession.commit();
    } finally{
        session.close();
    }
}
```

}

...

测试代码

```
'''java @Override public void insertUser(User user) throws Exception { SqlSession sqlSession = sqlSessionFactory.openSession(); try {
sqlSession.insert("insertUser", user); sqlSession.commit(); } finally{ session.close(); }
```

}

...

主键返回

```
'''xml select LAST/INSERTID() insert into user(username,birthday,sex,address) values(#{username},#{birthday},#{sex},#{address});
```

...

添加selectKey标签实现主键返回。

- **keyProperty**:指定返回的主键，存储在pojo中的哪个属性
- **order**: selectKey标签中的sql的执行顺序，是相对与insert语句来说。由于mysql的自增原理，执行完insert语句之后才将主键生成，所以这里selectKey的执行顺序为after。
- **resultType**:返回的主键对应的JAVA类型
- **LAST_INSERTID()**:是mysql的函数，返回autoincrement自增列新记录id值。

OGNL

对象导航图语言

```
|-User (参数值对象)
```

```
|-username--张三
```

```
|-birthday
```

```
|-sex--男
```

```
|-dept -- Department
```

```
|-name
```

```
|-no
```

OGNL表达式去获取Department对象的名字属性: dept.name

基础应用篇

mapper代理开发方式

此处使用的是JDK的动态代理方式，延迟加载使用的cglib动态代理方式

代理理解

代理分为静态代理和动态代理。此处先不说静态代理，因为Mybatis中使用的代理方式是动态代理。

动态代理分为两种方式：

- 基于JDK的动态代理--针对有接口的类进行动态代理
- 基于CGLIB的动态代理--通过子类继承父类的方式去进行代理。

XML方式

- 开发方式

只需要开发Mapper接口（dao接口）和Mapper映射文件，不需要编写实现类。

- 开发规范

Mapper接口开发方式需要遵循以下规范：

- 1、Mapper接口的类路径与Mapper.xml文件中的namespace相同。
- 2、Mapper接口方法名称和Mapper.xml中定义每个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接口方法的返回值类型和mapper.xml中定义的每个sql的resultType的类型相同。

- mapper映射文件

```
```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">
```

```
<select id="findUserId" parameterType="int" resultType="com.kkb.mybatis.po.User">
 select * from user where id = #{id}
</select>
```

...

- mapper接口

```
java /** * 用户管理mapper */ public interface UserMapper { //根据用户id查询用户信息 public User findUserId(int id) throws Exception; }
```

- 全局配置文件中加载映射文件

```
```xml
```

...

- 测试代码

```
```java public class UserMapperTest{
```

```
 private SqlSessionFactory sqlSessionFactory;
```

```
@Before public void setUp() throws Exception { //mybatis配置文件 String resource = "SqlMapConfig.xml"; InputStream inputStream =
Resources.getResourceAsStream(resource); //使用SqlSessionFactoryBuilder创建sessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream); } @Test public void testFindUserId() throws Exception { //获取session SqlSession session =
sqlSessionFactory.openSession(); //获取mapper接口的代理对象 UserMapper userMapper = session.getMapper(UserMapper.class); //调用代理对象方法
User user = userMapper.findUserId(1); System.out.println(user); //关闭session session.close();
```

```
}
```

```
}
```

...

## 注解方式

- 开发方式

只需要编写mapper接口文件接口。

- mapper接口

```
``java public interface AnnotationUserMapper { // 查询 @Select("SELECT * FROM user WHERE id = #{id}") public User findUserById(int id);
```

```
// 模糊查询用户列表
@Select("SELECT * FROM user WHERE username LIKE '%${value}%'")
public List<User> findUserList(String username);

// 添加并实现主键返回
@Insert("INSERT INTO user (username,birthday,sex,address) VALUES (#{username},#{birthday},#{sex},#{address})")
@SelectKey(statement = "SELECT LAST_INSERT_ID()", keyProperty = "id", resultType = int.class, before = false)
public void insertUser(User user);
```

```
} ``
```

- 测试代码

```
``java public class AnnotationUserMapperTest {
```

```
private SqlSessionFactory sqlSessionFactory;

/**
 * @Before注解的方法会在@Test注解的方法之前执行
 *
 * @throws Exception
 */
@Before
public void init() throws Exception {
 // 指定全局配置文件路径
 String resource = "SqlMapConfig.xml";
 // 加载资源文件（全局配置文件和映射文件）
 InputStream inputStream = Resources.getResourceAsStream(resource);
 // 还有构建者模式，去创建SqlSessionFactory对象
 sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
}

@Test
public void testFindUserById() {
 SqlSession sqlSession = sqlSessionFactory.openSession();
 AnnotationUserMapper userMapper = sqlSession.getMapper(AnnotationUserMapper.class);

 User user = userMapper.findUserById(1);
 System.out.println(user);
}

@Test
public void testFindUserList() {
 SqlSession sqlSession = sqlSessionFactory.openSession();
 AnnotationUserMapper userMapper = sqlSession.getMapper(AnnotationUserMapper.class);

 List<User> list = userMapper.findUserList("老郭");
 System.out.println(list);
}

@Test
public void testInsertUser() {
 SqlSession sqlSession = sqlSessionFactory.openSession();
 AnnotationUserMapper userMapper = sqlSession.getMapper(AnnotationUserMapper.class);

 User user = new User();
 user.setUsername("开课吧-2");
 user.setSex("1");
 user.setAddress("致真大厦");
 userMapper.insertUser(user);
 System.out.println(user.getId());
}
}
```

```
} ``
```

## 全局配置文件



## 配置内容

SqlMapConfig.xml中配置的内容和顺序如下：

`` properties（属性）

settings（全局配置参数）

typeAliases（类型别名）

typeHandlers（类型处理器）--Java类型--JDBC类型-->数据库类型转换

objectFactory（对象工厂）

plugins（插件）--可以在Mybatis执行SQL语句的流程中，横叉一脚去实现一些功能增强，比如PageHelper分页插件，就是第三方实现的一个插件

environments（环境集合属性对象）

environment（环境子属性对象） transactionManager（事务管理） dataSource（数据源） mappers（映射器） ``

## properties标签

SqlMapConfig.xml可以引用java属性文件中的配置信息。

1、在classpath下定义db.properties文件，

```
properties jdbc.driver=com.mysql.jdbc.Driver jdbc.url=jdbc:mysql://localhost:3306/ssm?characterEncoding=utf-8 jdbc.username=root jdbc.password=root
```

2、在SqlMapConfig.xml文件中，引用db.properties中的属性，具体如下：

```
xml <properties resource="db.properties"/> <environments default="development"> <environment id="development"> <transactionManager type="JDBC"/> <dataSource
```

properties标签除了可以使用resource属性，引用properties文件中的属性。还可以在properties标签内定义property子标签来定义属性和属性值，具体如下：

```
xml <properties> <property name="driver" value="com.mysql.jdbc.Driver"/> </properties>
```

注意： **MyBatis** 将按照下面的顺序来加载属性：

- 读取properties 元素体内定义的属性。
- 读取properties 元素中resource或 url 加载的属性，它会覆盖已读取的同名属性。

## typeAlias标签

别名的作用：就是为了简化映射文件中parameterType和ResultType中的POJO类型名称编写。

默认支持别名

| 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 |

自定义别名

在SqlMapConfig.xml中进行如下配置：

```
xml <typeAliases> <!-- 单个别名定义 --> <typeAlias alias="user" type="com.kkb.mybatis.po.User"/> <!-- 批量别名定义，扫描整个包下的类，别名为类名（首字母大写或小写者
```

## mappers标签

\

使用相对于类路径的资源

如：

```
<mapper resource="sqlmap/User.xml" />
```

\

使用绝对路径加载资源

如：

```
<mapper url="file:///d:/sqlmap/User.xml" />
```

\

使用mapper接口类路径，加载映射文件。

如：

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

注意：此种方法要求**mapper**接口名称和**mapper**映射文件名称相同，且放在同一个目录中。

\

注册指定包下的所有**mapper**接口，来加载映射文件。

如：

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

注意：此种方法要求**mapper**接口名称和**mapper**映射文件名称相同，且放在同一个目录中。

## 输入映射和输出映射

### parameterType(输入类型)

parameterType属性可以映射的输入参数Java类型有：简单类型、**POJO**类型、**Map**类型、**List**类型（数组）。

- Map类型和POJO类型的用法类似，本课程只讲POJO类型的相关配置。
- List类型在动态SQL部分进行讲解。

#### 传递简单类型

参考入门案例中用户查询的案例。

#### 传递pojo对象

参考入门案例中的添加用户的案例。

#### 传递pojo包装对象

包装对象：pojo类中嵌套pojo。

#### 需求

通过包装POJO传递参数，完成用户查询。

#### QueryVO

定义包装对象QueryVO

```
java public class QueryVO { private User user; }
```

#### SQL语句

```
XML SELECT * FROM user where username like '%小明%'
```

#### Mapper文件

```
```xml
```

...

Mapper接口

```
java /** * 用户管理mapper */ public interface UserMapper { //综合查询用户列表 public List<User> findUserList(QueryVo queryVo)throws Exception; }
```

测试方法

在UserMapperTest测试类中，添加以下测试代码：

```
```java @Test public void testFindUserList() throws Exception { SqlSession sqlSession = sqlSessionFactory.openSession(); //获得mapper的代理对象 UserMapper userMapper = sqlSession.getMapper(UserMapper.class); //创建QueryVo对象 QueryVo queryVo = new QueryVo(); //创建user对象 User user = new User(); user.setUsername("小明");
```

```
queryVo.setUser(user);
//根据queryvo查询用户
List<User> list = userMapper.findUserList(queryVo);
System.out.println(list);
```

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

...

## resultType(输出类型)

resultType属性可以映射的java类型有：简单类型、POJO类型、Map类型。

不过Map类型和POJO类型的使用情况类型，所以只需讲解POJO类型即可。

### 使用要求

使用resultType进行输出映射时，要求sql语句中查询的列名和要映射的pojo的属性名一致。

### 映射简单类型

#### 案例需求

查询用户记录总数。

#### Mapper映射文件

```
```xml
```

```
<select id="findUserCount" resultType="int">  
    select count(1) from user  
</select>
```

...

Mapper接口

```
java //查询用户总数 public int findUserCount() throws Exception;
```

测试代码

在UserMapperTest测试类中，添加以下测试代码：

```
```java @Test public void testFindUserCount() throws Exception { SqlSession sqlSession = sessionFactory.openSession(); //获得mapper的代理对象  
UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
```

```
 int count = userMapper.findUserCount(queryVo);
 System.out.println(count);

 sqlSession.close();
}
```

...

注意：输出简单类型必须查询出来的结果集只有一列。

### 映射pojo对象

注意：不管是单个POJO还是POJO集合，在使用resultType完成映射时，用法一样。

参考入门程序之根据用户ID查询用户信息和根据名称模糊查询用户列表的案例

## resultMap

### 使用要求

如果sql查询列名和pojo的属性名可以不一致，通过resultMap将列名和属性名作一个对应关系，最终将查询结果映射到指定的pojo对象中。

注意：resultType底层也是通过resultMap完成映射的。

### 需求

将以下sql的查询结果进行映射：

```
SELECT id id_,username username_,birthday birthday_ FROM user
```

#### Mapper接口

```
java // resultMap入门 public List<User> findUserListResultMap() throws Exception;
```

Mapper映射文件

由于sql查询列名和User类属性名不一致，所以不能使用resultType进行结构映射。

需要定义一个resultMap将sql查询列名和User类的属性名对应起来，完成结果映射。

```xml

```
<!-- resultMap入门 -->
<select id="findUserListResultMap" resultMap="userListResultMap">
    SELECT id id_,username username_,birthday birthday_ FROM user
</select>
```

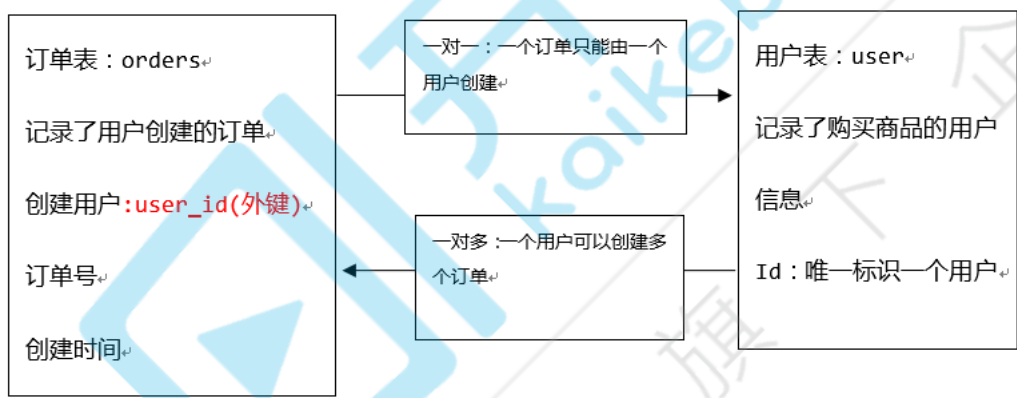
...

- \: 表示查询结果集的唯一标识，非常重要。如果是多个字段为复合唯一约束则定义多个
 - Property: 表示User类的属性。
 - Column: 表示sql查询出来的字段名。
 - Column和property放在一块儿表示将sql查询出来的字段映射到指定的pojo类属性上。
- \: 普通结果，即pojo的属性。

高级应用篇

关联查询

商品订单数据模型



注意：因为一个订单信息只会是一个人的订单，所以从查询订单信息出发，关联查询用户信息为一对一查询。如果从用户信息出发，查询用户下的订单信息则为一对多查询，因为一个用户可以下多个订单。

一对一查询

需求

查询所有订单信息，关联查询下单用户信息。

SQL 语句

```
```mysql SELECT orders.*, user.username, user.address FROM orders LEFT JOIN user ON orders.user_id = user.id
```

...

主信息：订单信息

从信息：用户信息

方法一：resultType

学生们自己实现。

方法二：resultMap

使用resultMap进行结果映射，定义专门的resultMap用于映射一对一查询结果。

### 创建扩展po类

创建OrdersExt类（该类用于结果集封装），加入User属性，user属性中用于存储关联查询的用户信息，因为订单关联查询用户是一对一关系，所以这里使用单个User对象存储关联查询的用户信息。

```
```java public class OrdersExt extends Orders {
```

```
    private User user; // 用户对象
    // get/set。。。。
```

```
}```
```

Mapper映射文件

在UserMapper.xml中，添加以下代码：

```
```xml
```

```
<resultMap type="OrdersExt" id="ordersAndUserRstMap">
 <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"/>
 <!-- 一对一关联映射 -->
 <!--
 property:Orders对象的user属性
 javaType: user属性对应 的类型
 -->
 <association property="user" javaType="com.kkb.mybatis.po.User">
 <!-- column:user表的主键对应的列 property: user对象中id属性-->
 <id column="user_id" property="id"/>
 <result column="username" property="username"/>
 <result column="address" property="address"/>
 </association>
</resultMap>
<select id="findOrdersAndUserRstMap" resultMap="ordersAndUserRstMap">
 SELECT
 o.id,
 o.user_id,
 o.number,
 o.createtime,
 o.note,
 u.username,
 u.address
 FROM
 orders o
 JOIN `user` u ON u.id = o.user_id
</select>
```

```
```
```

association: 表示进行一对一关联查询映射

property: 表示关联查询的结果存储在com.kkb.mybatis.po.Orders的user属性中

javaType: 表示关联查询的映射结果类型

Mapper接口

在UserMapper接口中，添加以下接口方法：

```
java public List<OrdersExt> findOrdersAndUserRstMap() throws Exception;
```

测试代码

在UserMapperTest测试类中，添加测试代码：

```
java public void testfindOrdersAndUserRstMap()throws Exception{ //获取session SqlSession session = sqlSessionSessionFactory.openSession(); //获取mapper接口实例 UserM
```

小结

使用resultMap进行结果映射时，具体是使用association完成关联查询的映射，将关联查询信息映射到pojo对象中。

一对多查询

需求

查询所有用户信息及用户关联的订单信息。

SQL语句

```
mysql SELECT u.*, o.id oid, o.number, o.createtime, o.note FROM `user` u LEFT JOIN orders o ON u.id = o.user_id
```

主信息：用户信息

从信息：订单信息

分析

在一对多关联查询时，只能使用resultMap进行结果映射。

- 1、一对多关联查询时，sql查询结果有多条，而映射对象是一个。
- 2、resultType完成结果映射的方式的一条记录映射一个对象。
- 3、resultMap完成结果映射的方式是以[主信息]为主对象，[从信息]映射为集合或者对象，然后封装到主对象中。

修改po类

在User类中加入List orders属性

```
6 public class User {
7
8     private Integer id;
9     private String username;// 用户名
10    private String sex;// 性别
11    private Date birthday;// 生日
12    private String address;// 地址
13    private List<Orders> orders;
14
15    public List<Orders> getOrders() {
16        return orders;
17    }
18    public void setOrders(List<Orders> orders) {
19        this.orders = orders;
20    }
21    public Integer getId() {
22        return id;
23    }
24    public void setId(Integer id) {
```

Mapper映射文件

在UserMapper.xml文件中，添加以下代码：

```
<xml <resultMap type="user" id="userAndOrderRstMap"> <!-- 用户信息映射 --> <id property="id" column="id"/> <result property="username" column="username"/> <res
```

Collection标签：定义了一对多关联的结果映射。

property="orders"：关联查询的结果集存储在User对象的上哪个属性。

ofType="orders"：指定关联查询的结果集中的对象类型即List中的对象类型。此处可以使用别名，也可以使用全限定名。

Mapper接口

```
```java // resultMap入门 public List findUserAndOrdersRstMap() throws Exception;
...
```
```

测试代码

```
```java @Test public void testFindUserAndOrdersRstMap() { SqlSession session = sqlSessionFactory.openSession(); UserMapper userMapper =
session.getMapper(UserMapper.class); List result = userMapper.findUserAndOrdersRstMap(); for (User user : result) { System.out.println(user); }
session.close(); }
...
```
```

延迟加载

什么是延迟加载

- MyBatis中的延迟加载，也称为**懒加载**，是指在进行关联查询时，按照设置延迟规则推迟对关联对象的select查询。延迟加载可以有效的减少数据库压力。
- Mybatis的延迟加载，需要通过resultMap标签中的association和collection子标签才能演示成功。
- Mybatis的延迟加载，也被称为是嵌套查询，对应的还有嵌套结果的概念，可以参考一对多关联的案例。
- 注意：**MyBatis**的延迟加载只是对关联对象的查询有延迟设置，对于主加载对象都是直接执行查询语句的。

延迟加载的分类

MyBatis根据对关联对象查询的select语句的执行时机，分为三种类型：直接加载、侵入式加载与深度延迟加载

- **直接加载**：执行完对主加载对象的select语句，马上执行对关联对象的select查询。
- **侵入式延迟**：执行对主加载对象的查询时，不会执行对关联对象的查询。但当要访问主加载对象的某个属性（该属性不是关联对象的属性）时，就会马上执行关联对象的select查询。
- **深度延迟**：执行对主加载对象的查询时，不会执行对关联对象的查询。访问主加载对象的详情时也不会执行关联对象的select查询。只有当真正访问关联对象的详情时，才会执行对关联对象的select查询。

延迟加载策略需要在Mybatis的全局配置文件中，通过标签进行设置。

案例准备

查询订单信息及它的下单用户信息。

直接加载

通过对全局参数：lazyLoadingEnabled进行设置，默认就是false。

```
xml <settings> <!-- 延迟加载总开关 --> <setting name="lazyLoadingEnabled" value="false"/> </settings>
```

侵入式延迟加载

```
xml <settings> <!-- 延迟加载总开关 --> <setting name="lazyLoadingEnabled" value="true"/> <!-- 侵入式延迟加载开关 --> <setting name="aggressiveLazyLoading" value
```

深度延迟加载

```
xml <settings> <!-- 延迟加载总开关 --> <setting name="lazyLoadingEnabled" value="true"/> <!-- 侵入式延迟加载开关 --> <setting name="aggressiveLazyLoading" value
```

N+1问题

- 深度延迟加载的使用会提升性能。
- 如果延迟加载的表数据太多，此时会产生N+1问题，主信息加载一次算1次，而从信息是会根据主信息传递过来的条件，去查询从表多次。

动态SQL

动态SQL的思想：就是使用不同的动态SQL标签去完成字符串的拼接处理、循环判断。

解决的问题是：

- 1、在映射文件中，会编写很多有重叠部分的SQL语句，比如SELECT语句和WHERE语句等这些重叠语句，该如何处理
- 2、SQL语句中的where条件有多个，但是页面只传递过来一个条件参数，此时会发生问题。

if标签

综合查询的案例中，查询条件是由页面传入，页面中的查询条件可能输入用户名称，也可能不输入用户名称。

```
xml <select id="findUserList" parameterType="queryVo" resultType="user"> SELECT * FROM user where 1=1 <if test="user != null"> <if test="user.username != nul
```

注意：要做『不等于空』字符串校验。

where标签

上边的sql中的1=1，虽然可以保证sql语句的完整性；但是存在性能问题。Mybatis提供where标签解决该问题。

代码修改如下：

```
xml <select id="findUserList" parameterType="queryVo" resultType="user"> SELECT * FROM user <!-- where标签会处理它后面的第一个and --> <where> <if test="user !=
```

sql片段

在映射文件中可使用sql标签将重复的sql提取出来，然后使用include标签引用即可，最终达到sql重用的目的，具体实现如下：

- 原映射文件中的代码：

```
```xml
```

```
...
```

- 将where条件抽取出来：

```
```xml AND username like '%${user.username}%'
```

```
...
```

- 使用include引用：

```
```xml SELECT * FROM user WHERE id = #{id}
```

```
...
```

## 刷新二级缓存

通过flushCache属性，可以控制select、insert、update、delete标签是否属性二级缓存

### 默认设置

- \* 默认情况下如果是select语句，那么flushCache是false。
- \* 如果是insert、update、delete语句，那么flushCache是true。

### 默认配置解读

- \* 如果查询语句设置成true，那么每次查询都是去数据库查询，即意味着该查询的二级缓存失效。
- \* 如果增删改语句设置成false，即使用二级缓存，那么如果在数据库中修改了数据，而缓存数据还是原来的，这个时候就会出现脏读。

flushCache设置如下：

```
xml <select id="findUserById" parameterType="int" resultType="com.kkb.mybatis.po.User" useCache="true" flushCache="true"> SELECT * FROM user WHERE id = #{id}
```

## 应用场景

- 使用场景：

对于访问响应速度要求高，但是实时性不高的查询，可以采用二级缓存技术。

- 注意事项：

在使用二级缓存的时候，要设置一下刷新间隔（cache标签中有一个flushInterval属性）来定时刷新二级缓存，这个刷新间隔根据具体需求来设置，比如设置30分钟、60分钟等，单位为毫秒。

## 局限性

Mybatis二级缓存对细粒度的数据级别的缓存实现不好。

- 场景：

对商品信息进行缓存，由于商品信息查询访问量大，但是要求用户每次查询都是最新的商品信息，此时如果使用二级缓存，就无法实现当一个商品发生变化只刷新该商品的缓存信息而不刷新其他商品缓存信息，因为二级缓存是mapper级别的，当一个商品的信息发送更新，所有的商品信息缓存数据都会清空。

- 解决方法

此类问题，需要在业务层根据需要对数据有针对性的缓存。

比如可以对经常变化的 数据操作单独放到另一个namespace的mapper中。

## Mybatis逆向工程

### 逆向工程介绍

使用官方网站的Mapper自动生成工具mybatis-generator-core-1.3.2来针对单表生成po类（Example）和Mapper接口和mapper映射文件

### 修改配置文件

在generatorConfig.xml中配置Mapper生成的详细信息，注意修改以下几点：



1. 修改要生成的数据库表

2. pojo文件所在包路径

3. Mapper所在的包路径

配置文件如下:

```
```xml <?xml version="1.0" encoding="UTF-8"?> <!DOCTYPE generatorConfiguration PUBLIC "-//mybatis.org/DTD MyBatis Generator Configuration 1.0//EN" "http://mybatis.org/dtd/mybatis-generator-config1.0.dtd">
```

```
<!-- 默认false, 把JDBC DECIMAL 和 NUMERIC 类型解析为 Integer, 为 true时把JDBC DECIMAL
和 NUMERIC 类型解析为java.math.BigDecimal -->
<javaTypeResolver>
  <property name="forceBigDecimals" value="false" />
</javaTypeResolver>

<!-- targetProject:生成PO类的位置 -->
<javaModelGenerator targetPackage="com.kkb.ms.po"
  targetProject=".\\src">
  <!-- enableSubPackages:是否让schema作为包的后缀 -->
  <property name="enableSubPackages" value="false" />
  <!-- 从数据库返回的值被清理前后的空格 -->
  <property name="trimStrings" value="true" />
</javaModelGenerator>
<!-- targetProject:mapper映射文件生成的位置 -->
<sqlMapGenerator targetPackage="com.kkb.ms.mapper"
  targetProject=".\\src">
  <!-- enableSubPackages:是否让schema作为包的后缀 -->
  <property name="enableSubPackages" value="false" />
</sqlMapGenerator>
<!-- targetPackage: mapper接口生成的位置 -->
<javaClientGenerator type="XMLMAPPER"
  targetPackage="com.kkb.ms.mapper" targetProject=".\\src">
  <!-- enableSubPackages:是否让schema作为包的后缀 -->
  <property name="enableSubPackages" value="false" />
</javaClientGenerator>
<!-- 指定数据库表 -->
<table schema="" tableName="user"></table>
<table schema="" tableName="order"></table>
</context>
```

...

注意事项

每次执行逆向工程代码之前, 先删除原来已经生成的mapper xml文件再进行生成。

- mapper.xml文件的内容不是被覆盖而是进行内容追加, 会导致mybatis解析失败。
- po类及mapper.java文件的内容是直接覆盖没有此问题。

PageHelper分页插件

PageHelper分页插件介绍

<https://github.com/pagehelper/Mybatis-PageHelper/blob/master/wikis/en/HowToUse.md>

* 如果你也在用Mybatis, 建议尝试该分页插件, 这个一定是最方便使用的分页插件。

* 目前几乎支持所有的关系型数据库

* 最新版本是5.1.6。

使用方法

添加依赖

```
xml <dependency> <groupId>com.github.pagehelper</groupId> <artifactId>pagehelper</artifactId> <version>5.1.6</version> </dependency>
```

配置PageHelper

- Mybatis全局配置文件

```
```xml
```

```
...
```

- spring配置文件

```
```xml
```

```
helperDialect=mysql
```

```
...
```

项目中使用PageHelper

```
```java //获取第1页，10条内容，默认查询总数count PageHelper.startPage(1, 10); List list = countryMapper.selectAll(); //用PageInfo对结果进行包装
PageInfo page = new PageInfo(list); //测试PageInfo全部属性 //PageInfo包含了非常全面的分页属性 assertEquals(1, page.getPageNum()); assertEquals(10,
page.getPageSize()); assertEquals(1, page.getStartRow()); assertEquals(10, page.getEndRow()); assertEquals(183, page.getTotal()); assertEquals(19,
page.getPages()); assertEquals(1, page.getFirstPage()); assertEquals(8, page.getLastPage()); assertEquals(true, page.isFirstPage()); assertEquals(false,
page.isLastPage()); assertEquals(false, page.isHasPreviousPage()); assertEquals(true, page.isHasNextPage());
...```
```

### 注意事项

1. 需要分页的查询语句，必须是处于PageHelper.startPage(1, 10);后面的第一条语句。
2. 如果查询语句是使用resultMap进行的嵌套结果映射，则无法使用PageHelper进行分页。

## 扩展点

- Mybatis Plus

## 总结

## 作业

- 使用resultType完成一对一的结果映射。
- foreach动态标签的使用：通过List集合作为参数传递id集合。

## 下节预告

Mybatis架构分析及手写Mybatis框架。