# Mybatis是什么？

MyBatis 本是[apache](http://baike.baidu.com/view/28283.htm)的一个开源项目[iBatis](http://baike.baidu.com/view/628102.htm), 2010年这个项目由apache software foundation 迁移到了google code，并且改名为MyBatis，实质上Mybatis对ibatis进行一些改进。

MyBatis是一个优秀的持久层框架，它对jdbc的操作数据库的过程进行封装，使开发者只需要关注 SQL 本身，而不需要花费精力去处理例如注册驱动、创建connection、创建statement、手动设置参数、结果集检索等jdbc繁杂的过程代码。

Mybatis通过xml或注解的方式将要执行的各种statement（statement、preparedStatemnt、CallableStatement）配置起来，并通过java对象和statement中的sql进行映射生成最终执行的sql语句，最后由mybatis框架执行sql并将结果映射成java对象并返回。

# 分析原生态jdbc程序中存在的问题

## 原生态Jdbc程序代码

**public static void** main(String[] args) {

Connection connection = **null**;

PreparedStatement preparedStatement = **null**;

ResultSet resultSet = **null**;

**try** {

//1、加载数据库驱动

Class.*forName*("com.mysql.jdbc.Driver");

//2、通过驱动管理类获取数据库链接

connection = DriverManager.*getConnection*("jdbc:mysql://localhost:3306/mybatis?characterEncoding=utf-8", "root", "root");

//3、定义sql语句 ?表示占位符

String sql = "select \* from user where username = ?";

//4、获取预处理statement

preparedStatement = connection.prepareStatement(sql);

//5、设置参数，第一个参数为sql语句中参数的序号（从1开始），第二个参数为设置的参数值

preparedStatement.setString(1, "王五");

//6、向数据库发出sql执行查询，查询出结果集

resultSet = preparedStatement.executeQuery();

//7、遍历查询结果集

**while**(resultSet.next()){

System.*out*.println(resultSet.getString("id")+" "+resultSet.getString("username"));

}

} **catch** (Exception e) {

e.printStackTrace();

}**finally**{

//8、释放资源

**if**(resultSet!=**null**){

**try** {

resultSet.close();

} **catch** (SQLException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

**if**(preparedStatement!=**null**){

**try** {

preparedStatement.close();

} **catch** (SQLException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

**if**(connection!=**null**){

**try** {

connection.close();

} **catch** (SQLException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

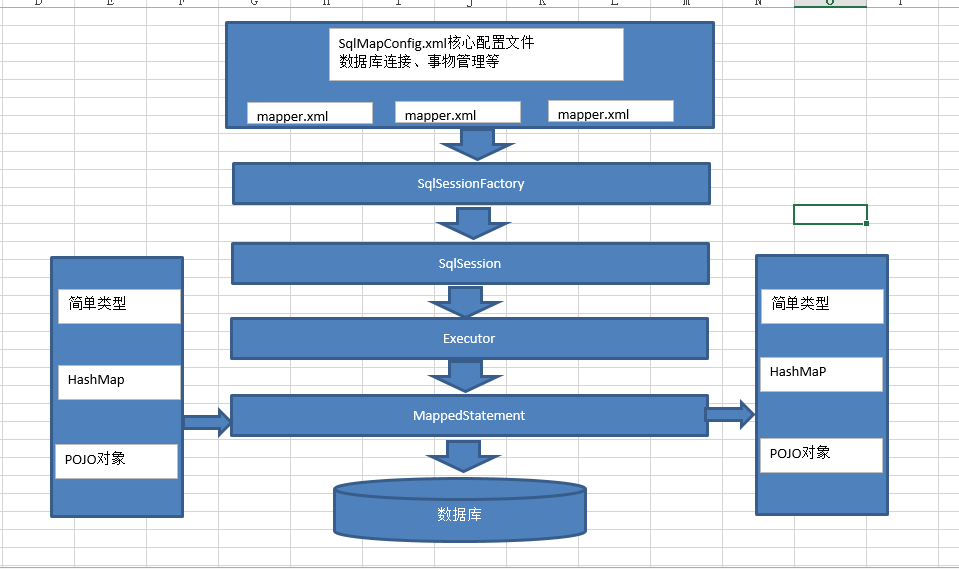
}

## Jdbc问题总结

1. 数据库连接频繁开启和关闭，会严重影响数据库的性能。
2. 代码中存在硬编码，分别是数据库部分的硬编码和SQL执行部分的硬编码。

# Mybatis框架原理（核心）

## 框架图



## 分析结论

1. mybatis配置文件，包括**Mybatis全局配置文件和Mybatis映射文件**，其中全局配置文件配置了数据源、事务等信息；映射文件配置了SQL执行相关的 信息。
2. mybatis通过读取配置文件信息（全局配置文件和映射文件），构造出**SqlSessionFactory，**即会话工厂。
3. 通过SqlSessionFactory，可以创建**SqlSession**即会话。Mybatis是通过SqlSession来操作数据库的。
4. SqlSession本身不能直接操作数据库，它是通过底层的**Executor**执行器接口来操作数据库的。Executor接口有两个实现类，一个是普通执行器，一个是**缓存执行器（默认）**。
5. Executor执行器要处理的SQL信息是封装到一个底层对象**MappedStatement**中。该对象包括：SQL语句、输入参数映射信息、输出结果集映射信息。其中输入参数和输出结果的映射类型包括java的简单类型、HashMap集合对象、POJO对象类型。

# Mybatis入门程序

Mybatis课程的所有代码程序将通过一个**订单商品案例**来进行讲解。

## 需求

对用户信息的增删改查操作。

1. 根据用户ID来查询用户信息；
2. 根据用户名称来模糊查询用户信息列表；
3. 添加用户
4. 删除用户（练习）
5. 修改用户（练习）

## 环境准备

* Jdk环境：jdk1.7.0\_72
* Ide环境：eclipse indigo
* 数据库环境：MySQL 5.1
* Mybatis：3.2.7

### 数据库初始化

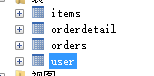
#### 数据库脚本



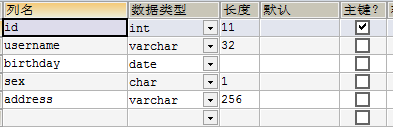
1. 执行sql\_table.sql脚本，创建数据库表。
2. 执行sql\_data.sql初始化测试数据。

#### 数据库表

订单商品案例的数据库脚本中，总共包含四张表，其中入门程序只使用**user表**：



**用户表**的表结构如下：



### 下载mybatis

mybaits的代码由github.com管理，下载地址：<https://github.com/mybatis/mybatis-3/releases>



Lib：mybatis的依赖包

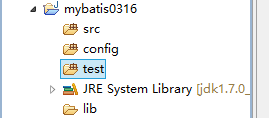
Mybatis-3.2.7.jar：mybatis的核心包

Mybatis-3.2.7.pdf：mybatis的使用指南

## 工程搭建（三步）

### 第一步：创建java工程

用eclipse创建一个java工程，jdk使用1.7.0\_72。



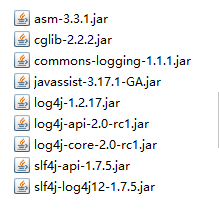
### 第二步：加入jar包

加入以下四部分jar包，其中junit的jar包，是非必须的。

* Mybatis核心包



* Mybatis依赖包



* MySQL驱动包



* Junit单元测试包（单元测试需要的包）



### 第三步：添加log4j.properties文件

Mybatis使用的日志包是log4j的，所以需要添加log4j.properties。

在classpath下创建log4j.properties如下：

文件内容可以从mybatis-3.2.7.pdf中拷贝

# Global logging configuration

log4j.rootLogger=DEBUG, stdout

# Console output...

log4j.appender.stdout=org.apache.log4j.ConsoleAppender

log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n

日志级别在开发阶段设置成DEBUG，在生产阶段设置成INFO或者ERROR。

## 编程步骤

1. 创建PO类，根据需求创建；
2. 创建全局配置文件SqlMapConfig.xml；
3. 编写映射文件；
4. 加载映射文件，在SqlMapConfig.xml中进行加载；
5. 编写测试程序，即编写Java代码，连接并操作数据库。

思路：

1. 读取配置文件；
2. 通过SqlSessionFactoryBuilder创建SqlSessionFactory会话工厂。
3. 通过SqlSessionFactory创建SqlSession。
4. 调用SqlSession的操作数据库方法。
5. 关闭SqlSession。

## 代码开发

### 创建PO类

创建的po类的属性要和数据库中表的列名一致（如果表中的列名是带有下划线，那么po类中对应的的属性名要采用驼峰式命名）

User.java类如下：

**Public class** User {

**private int** id;

**private** String username;// 用户姓名

**private** String sex;// 性别

**private** Date birthday;// 生日

**private** String address;// 地址

get/set……

### 创建SqlMapConfig.xml配置文件

在classpath下，创建SqlMapConfig.xml文件

SqlMapConfig.xml（文件头可以从mybatis-3.2.7.pdf文档的2.1.2小节中拷贝）：

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<!DOCTYPE configuration

PUBLIC "-//mybatis.org//DTD Config 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-config.dtd">

<configuration>

<!-- 配置mybatis的环境信息 -->

<environments default=*"development"*>

<environment id=*"development"*>

<!-- 配置JDBC事务控制，由mybatis进行管理 -->

<transactionManager type=*"JDBC"*></transactionManager>

<!-- 配置数据源，采用dbcp连接池 -->

<dataSource type=*"POOLED"*>

<property name=*"driver"* value=*"com.mysql.jdbc.Driver"*/>

<property name=*"url"* value=*"jdbc:mysql://localhost:3306/mybatis?useUnicode=true&amp;characterEncoding=utf8"*/>

<property name=*"username"* value=*"root"*/>

<property name=*"password"* value=*"root"*/>

</dataSource>

</environment>

</environments>

</configuration>

### 需求开发

在classpath下，创建sqlmap文件夹。在sqlmap目录下，创建User.xml映射文件。

**Mybatis的映射文件头**（可以从mybatis-3.2.7.pdf文件中拷贝）：

<?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">

#### 根据用户ID来查询用户信息

##### 编写映射文件

在User.xml中添加以下代码：

<!--

namespace：命名空间，它的作用就是对SQL进行分类化管理，可以理解为SQL隔离

注意：使用mapper代理开发时，namespace有特殊且重要的作用

-->

<mapper namespace=*"test"*>

<!-- 根据用户ID，查询用户信息 -->

<!--

[id]：statement的id，要求在命名空间内唯一

[parameterType]：入参的java类型

[resultType]：查询出的单条结果集对应的java类型

[#{}]： 表示一个占位符?

[#{id}]：表示该占位符待接收参数的名称为id。注意：如果参数为简单类型时，#{}里面的参数名称可以是任意定义

-->

<select id=*"findUserById"* parameterType=*"int"* resultType=*"cn.itcast.mybatis.po.User"*>

SELECT \* FROM USER WHERE id = #{id}

</select>

</mapper>

##### 加载映射文件

在SqlMapConfig.xml中，添加以下代码：

<!-- 加载mapper -->

<mappers>

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

</mappers>

##### 编写测试程序

**public** **class** MybatisFirst {

@Test

**public** **void** findUserByIdTest() **throws** Exception{

//1、读取配置文件

String resource = "SqlMapConfig.xml";

InputStream inputStream = Resources.*getResourceAsStream*(resource);

//2、根据配置文件创建SqlSessionFactory

SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(inputStream);

//3、SqlSessionFactory创建SqlSession

SqlSession sqlSession = sqlSessionFactory.openSession();

//4、SqlSession执行statement，并返回映射结果

//第一个参数：statement的id，建议：namespace.statementId（确保唯一）

//第二个参数：入参的值，它的类型要和映射文件中对应的statement的入参类型一致

User user = sqlSession.selectOne("findUserById", 1);

//打印输出结果集

System.*out*.println(user);

//5、关闭SqlSession

sqlSession.close();

}

}

#### 根据用户名称来模糊查询用户信息列表

##### 编写映射文件

在User.xml中，添加以下内容：

<!-- 根据用户名称模糊查询用户信息列表 -->

<!--

[${}]：表示拼接SQL字符串

[${value}]：表示要拼接的是简单类型参数。

注意：

1、如果参数为简单类型时，${}里面的参数名称必须为value

2、${}会引起SQL注入，一般情况下不推荐使用。但是有些场景必须使用${}，比如order by ${colname}

-->

<select id=*"findUsersByName"* parameterType=*"String"* resultType=*"cn.itcast.mybatis.po.User"*>

SELECT \* FROM USER WHERE username LIKE '%${value}%'

select \* from user where username like "%"#{username}"%"

</select>

##### 加载映射文件

已配置，此处无需再次配置。

##### 编写测试程序

@Test

**public** **void** findUsersByNameTest() **throws** Exception {

// 1、读取配置文件

String resource = "SqlMapConfig.xml";

InputStream inputStream = Resources.*getResourceAsStream*(resource);

// 2、根据配置文件创建SqlSessionFactory

SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(inputStream);

// 3、SqlSessionFactory创建SqlSession

SqlSession sqlSession = sqlSessionFactory.openSession();

// 4、SqlSession执行statement，并返回映射结果

// 第一个参数：statement的id，建议：namespace.statementId（确保唯一）

// 第二个参数：入参的值，它的类型要和映射文件中对应的statement的入参类型一致

List<User> users = sqlSession.selectList("test.findUsersByName", "小明");

// 打印输出结果集

System.*out*.println(users);

// 5、关闭SqlSession

sqlSession.close();

}

#### 添加用户

##### 编写映射文件

<!-- 添加用户 -->

<!-- 如果主键的值是通过MySQL自增机制生成的，那么我们此处不需要再显示的给ID赋值 -->

<insert id=*"insertUser"* parameterType=*"cn.itcast.mybatis.po.User"*>

INSERT INTO USER(username,sex,birthday,address) VALUES (#{username},#{sex},#{birthday},#{address})

</insert>

##### 加载映射文件

已配置，此处无需再次配置。

##### 编写测试程序

注意：**增删改操作要对SqlSession执行commit操作。**

@Test

**public** **void** insertUserTest() **throws** Exception {

// 1、读取配置文件

String resource = "SqlMapConfig.xml";

InputStream inputStream = Resources.*getResourceAsStream*(resource);

// 2、根据配置文件创建SqlSessionFactory

SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder()

.build(inputStream);

// 3、SqlSessionFactory创建SqlSession

SqlSession sqlSession = sqlSessionFactory.openSession();

// 4、SqlSession执行statement，并返回映射结果

//构建user参数，没有赋值的属性采取默认值

User user = **new** User();

user.setUsername("东哥1");

user.setAddress("清河宝盛西里");

// 第一个参数：statement的id，建议：namespace.statementId（确保唯一）

// 第二个参数：入参的值，它的类型要和映射文件中对应的statement的入参类型一致

sqlSession.insert("insertUser", user);

//切记：增删改操作时，要执行commit操作

sqlSession.commit();

// 5、关闭SqlSession

sqlSession.close();

}

##### 主键返回之MySQL自增主键

思路：

* MySQL自增主键，是指在insert之前MySQL会自动生成一个自增的主键。
* 我们可以通过MySQL的函数获取到刚插入的自增主键:

LAST\_INSERT\_ID()

* 这个函数是在insert语句之后去调用。

修改映射文件：

<!-- 添加用户之自增主键返回（selectKey方式） -->

<!--

[selectKey标签]：通过select查询来生成主键

[keyProperty]：指定存放生成主键的属性

[resultType]：生成主键所对应的Java类型

[order]：指定该查询主键SQL语句的执行顺序，相对于insert语句

[last\_insert\_id]：MySQL的函数，要配合insert语句一起使用

-->

<insert id=*"insertUser"* parameterType=*"cn.itcast.mybatis.po.User"*>

<selectKey keyProperty=*"id"* resultType=*"int"* order=*"AFTER"*>

SELECT LAST\_INSERT\_ID()

</selectKey>

INSERT INTO USER(username,sex,birthday,address) VALUES (#{username},#{sex},#{birthday},#{address})

</insert>

##### 主键返回之MySQL函数UUID（自己练习）

注意：**使用mysql的uuid()函数生成主键，需要修改表中id字段类型为string，长度设置成35位。**

<!-- 添加用户之UUID主键返回 -->

<!--

[uuid]：MySQL的函数，生成的主键是35位的字符串，所以使用它时要修改id的类型为字符类型

注意：

1、此时order采用BEFORE，因为需要先生成出主键，再执行insert语句

2、显式的给ID赋值

-->

<insert id=*"insertUser"* parameterType=*"cn.itcast.mybatis.po.User"*>

<selectKey keyProperty=*"id"* resultType=*"string"* order=*"BEFORE"*>

SELECT UUID()

</selectKey>

INSERT INTO USER(id,username,sex,birthday,address) VALUES (#{id},#{username},#{sex},#{birthday},#{address})

</insert>

##### 主键返回之Oracle序列返回（自己练习）

<!-- 添加用户之sequence返回 -->

<!--

通过Oracle的sequence获取主键方式与MySQL的uuid方式基本一致

-->

<insert id=*"insertUser"* parameterType=*"cn.itcast.mybatis.po.User"*>

<selectKey keyProperty=*"id"* resultType=*"int"* order=*"BEFORE"*>

SELECT user\_seq.nextval() FROM dual

</selectKey>

INSERT INTO USER(id,username,sex,birthday,address) VALUES (#{id},#{username},#{sex},#{birthday},#{address})

</insert>

#### 删除用户

##### 编写映射文件

<!-- 根据ID删除用户 -->

<delete id=*"deleteUser"* parameterType=*"int"*>

DELETE FROM USER WHERE id= #{id}

</delete>

##### 加载映射文件

已配置，此处无需再次配置。

##### 编写测试程序

@Test

**public** **void** deleteUserTest() **throws** Exception{

// 1、读取配置文件

String resource = "SqlMapConfig.xml";

InputStream inputStream = Resources.*getResourceAsStream*(resource);

// 2、根据配置文件创建SqlSessionFactory

SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder()

.build(inputStream);

// 3、SqlSessionFactory创建SqlSession

SqlSession sqlSession = sqlSessionFactory.openSession();

// 4、SqlSession执行statement，并返回映射结果

// 第一个参数：statement的id，建议：namespace.statementId（确保唯一）

// 第二个参数：入参的值，它的类型要和映射文件中对应的statement的入参类型一致

sqlSession.delete("test.deleteUser", 30);

//切记：增删改操作时，要执行commit操作

sqlSession.commit();

// 5、关闭SqlSession

sqlSession.close();

}

#### 修改用户

##### 编写映射文件

<!-- 根据传入的用户信息修改用户 -->

<update id=*"updateUser"* parameterType=*"cn.itcast.mybatis.po.User"*>

UPDATE USER SET username = #{username},sex=#{sex} WHERE id=#{id}

</update>

##### 加载映射文件

已配置，此处无需再次配置。

##### 编写测试程序

@Test

**public** **void** updateUserTest() **throws** Exception{

// 1、读取配置文件

String resource = "SqlMapConfig.xml";

InputStream inputStream = Resources.*getResourceAsStream*(resource);

// 2、根据配置文件创建SqlSessionFactory

SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(inputStream);

// 3、SqlSessionFactory创建SqlSession

SqlSession sqlSession = sqlSessionFactory.openSession();

// 4、SqlSession执行statement，并返回映射结果

//构建user参数，没有赋值的属性采取默认值

User user = **new** User();

user.setId(28);

user.setUsername("东哥11");

user.setAddress("清河宝盛西里");

// 第一个参数：statement的id，建议：namespace.statementId（确保唯一）

// 第二个参数：入参的值，它的类型要和映射文件中对应的statement的入参类型一致

sqlSession.update("test.updateUser", user);

//切记：增删改操作时，要执行commit操作

sqlSession.commit();

// 5、关闭SqlSession

sqlSession.close();

}

## 小结

### parameterType和resultType

parameterType指定输入参数的java类型，可以填写或Java类的全限定名。

resultType指定输出结果的java类型，可以填写别名或Java类的全限定名。

### #{}和${}

#{}：相当于预处理中的占位符？。

#{}里面的参数表示接收java输入参数的名称。

#{}可以接受HashMap、简单类型、POJO类型的参数。

当接受简单类型的参数时，#{}里面可以是value，也可以是其他。

#{}可以防止SQL注入。

${}：相当于拼接SQL串，对传入的值不做任何解释的原样输出。

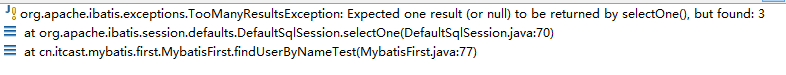
${}会引起SQL注入，所以要谨慎使用。

${}可以接受HashMap、简单类型、POJO类型的参数。

当接受简单类型的参数时，${}里面只能是value。

### selectOne和selectList

selectOne：只能查询0或1条记录，大于1条记录的话，会报错：



selectList：可以查询0或N条记录

# Mybatis开发dao

Mybatis在项目中主要使用的地方就是开发dao（数据访问层），所以下面讲解一下mybatis开发dao的方法。有两种方式：**原始dao开发方式**、**mapper代理开发方式（推荐）**。

## 需求

1. 根据用户ID来查询用户信息；
2. 根据用户名称来模糊查询用户信息列表；
3. 添加用户；

## 原始dao开发方式

### 思路

程序员需要写dao接口和dao实现类。

### 编程步骤

1. 根据需求创建po类
2. 编写全局配置文件
3. 根据需求编写映射文件
4. 加载映射文件
5. 编写dao接口
6. 编写dao实现类
7. 编写测试代码

### 程序编写

**步骤中的1、2、3、4都在入门程序中进行了编写，此处不需要重新编写。**

#### 开发dao接口

**public** **interface** UserDao {

//根据用户ID来查询用户信息

**public** User findUserById(**int** id);

//根据用户名称来模糊查询用户信息列表

**public** List<User> findUsersByName(String username);

//添加用户

**public** **void** insertUser(User user);

}

#### 开发dao实现类

##### SqlSession使用范围

通过入门程序，大家可以看出，在测试代码中，有大量的重复代码。所以我们第一反应就是想给它抽取出共性的部分，但是SqlSession、SqlSessionFactory、SqlSessionFactoryBuilder有着各自的生命周期，因为这些生命周期的不同，抽取时要有针对性的处理。

所以在抽取之前，我们先来了解并总结下它们三个的生命周期。

###### SqlSessionFactoryBuilder

它的作用只是通过配置文件创建SqlSessionFactory，所以只要创建出SqlSessionFactory，它就可以销毁了。**所以说，它的生命周期是在方法之内。**

###### SqlSessionFactory

它的作用是创建SqlSession的工厂，工厂一旦创建，除非应用停掉，不要销毁。

**所以说它的生命周期是在应用范围内。**这里可以通过单例模式来管理它。

在mybatis整合spring之后，最好的处理方式是把SqlSessionFactory交由spring来做单例管理。

###### SqlSession

SqlSession是一个面向用户（程序员）的接口，它的默认实现是DefaultSqlSession。

Mybatis是通过SqlSession来操作数据库的。SqlSession中不仅包含要处理的SQL信息，还包括一些数据信息，所以说它是线程不安全的，**因此它最佳的生命周期范围是在方法体之内。**

##### Dao实现类代码

需要向dao实现类中注入SqlSessionFactory，在方法体内通过SqlSessionFactory创建SqlSession

要注意SqlSession和SqlSessionFactory的生命周期。

**public** **class** UserDaoImpl **implements** UserDao {

//注入SqlSessionFactory

**private** SqlSessionFactory sqlSessionFactory;

//使用构造方法来初始化SqlSessionFactory

**public** UserDaoImpl(SqlSessionFactory sqlSessionFactory){

**this**.sqlSessionFactory = sqlSessionFactory;

}

@Override

**public** User findUserById(**int** id) {

//通过工厂，在方法内部获取SqlSession，这样就可以避免线程不安全

SqlSession sqlSession = sqlSessionFactory.openSession();

//返回结果集

**return** sqlSession.selectOne("test.findUserById", id);

}

@Override

**public** List<User> findUsersByName(String username) {

//通过工厂，在方法内部获取SqlSession，这样就可以避免线程不安全

SqlSession sqlSession = sqlSessionFactory.openSession();

**return** sqlSession.selectList("test.findUsersByName", username);

}

@Override

**public** **void** insertUser(User user) {

//通过工厂，在方法内部获取SqlSession，这样就可以避免线程不安全

SqlSession sqlSession = sqlSessionFactory.openSession();

sqlSession.insert("test.insertUser", user);

}

}

#### 编写测试代码

**public** **class** UserDaoTest {

//声明全局的SqlSessionFactory

**private** SqlSessionFactory sqlSessionFactory;

@Before

**public** **void** setUp() **throws** Exception {

// 1、读取配置文件

String resource = "SqlMapConfig.xml";

InputStream inputStream = Resources.*getResourceAsStream*(resource);

// 2、根据配置文件创建SqlSessionFactory

sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(inputStream);

}

@Test

**public** **void** testFindUserById() {

//构造UserDao对象

UserDao userDao = **new** UserDaoImpl(sqlSessionFactory);

//调用UserDao对象的方法

User user = userDao.findUserById(1);

System.*out*.println(user);

}

@Test

**public** **void** testFindUsersByName() {

//构造UserDao对象

UserDao userDao = **new** UserDaoImpl(sqlSessionFactory);

//调用UserDao对象的方法

List<User> list = userDao.findUsersByName("小明");

System.*out*.println(list);

}

@Test

**public** **void** testInsertUser() {

//构造UserDao对象

UserDao userDao = **new** UserDaoImpl(sqlSessionFactory);

//构造User对象

User user = **new** User();

user.setUsername("东哥3");

user.setAddress("清河宝盛西里3");

//调用UserDao对象的方法

userDao.insertUser(user);

System.*out*.println(user.getId());

}

}

### 问题总结

原始dao开发存在一些问题：

* **存在一定量的模板代码**。比如：通过SqlSessionFactory创建SqlSession；调用SqlSession的方法操作数据库；关闭Sqlsession。
* **存在一些硬编码**。调用SqlSession的方法操作数据库时，需要指定statement的id，这里存在了硬编码。

## Mapper代理开发方式（推荐）

Mapper代理的开发方式，程序员只需要编写mapper接口（相当于dao接口）即可。Mybatis会自动的为mapper接口生成动态代理实现类。

**不过要实现mapper代理的开发方式，需要遵循一些开发规范。**

### 开发规范

1. mapper接口的全限定名要和mapper映射文件的namespace的值相同。
2. mapper接口的方法名称要和mapper映射文件中的statement的id相同；
3. mapper接口的方法参数只能有一个，且类型要和mapper映射文件中statement的parameterType的值保持一致。
4. mapper接口的返回值类型要和mapper映射文件中statement的resultType值或resultMap中的type值保持一致；

通过规范式的开发mapper接口，可以解决原始dao开发当中存在的问题：

1. 模板代码已经去掉；
2. 剩下去不掉的操作数据库的代码，其实就是一行代码。这行代码中硬编码的部分，通过第一和第二个规范就可以解决。

### 编程步骤

1. 根据需求创建po类
2. 编写全局配置文件
3. 根据需求编写映射文件
4. 加载映射文件
5. 编写mapper接口
6. 编写测试代码

### 程序编写

**步骤中的1、2都在入门程序中进行了编写，此处不需要重新编写。**

#### 编写mapper映射文件

重新定义mapper映射文件UserMapper.xml（内容同Users.xml，**除了namespace的值**），放到新创建的目录mapper下。

<?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">

<!-- namespace：此时用mapper代理方式，它的值必须等于对应mapper接口的全限定名 -->

<mapper namespace=*"cn.itcast.mybatis.mapper.UserMapper"*>

<!-- 根据用户ID，查询用户信息 -->

<!--

[id]：statement的id，要求在命名空间内唯一

[parameterType]：入参的java类型，可是是简单类型、POJO、HashMap

[resultType]：查询出的单条结果集对应的java类型

[#{}]： 表示一个占位符?

[#{id}]：表示该占位符待接收参数的名称为id。注意：如果参数为简单类型时，#{}里面的参数名称可以是任意定义

-->

<select id=*"findUserById"* parameterType=*"int"* resultType=*"cn.itcast.mybatis.po.User"*>

SELECT \* FROM USER WHERE id = #{id}

</select>

<!-- 根据用户名称模糊查询用户信息列表 -->

<!--

[${}]：表示拼接SQL字符串，即不加解释的原样输出

[${value}]：表示要拼接的是简单类型参数。

注意：

1、如果参数为简单类型时，${}里面的参数名称必须为value

2、${}会引起SQL注入，一般情况下不推荐使用。但是有些场景必须使用${}，比如order by ${colname}

-->

<select id=*"findUsersByName"* parameterType=*"java.lang.String"* resultType=*"cn.itcast.mybatis.po.User"*>

SELECT \* FROM USER WHERE username LIKE '%${value}%'

</select>

<!-- 添加用户之自增主键返回（selectKey方式） -->

<!--

[selectKey标签]：通过select查询来生成主键

[keyProperty]：指定存放生成主键的属性

[resultType]：生成主键所对应的Java类型

[order]：指定该查询主键SQL语句的执行顺序，相对于insert语句，此时选用AFTER

[last\_insert\_id]：MySQL的函数，要配合insert语句一起使用

-->

<insert id=*"insertUser"* parameterType=*"cn.itcast.mybatis.po.User"*>

<selectKey keyProperty=*"id"* resultType=*"int"* order=*"AFTER"*>

SELECT LAST\_INSERT\_ID()

</selectKey>

INSERT INTO USER(username,sex,birthday,address) VALUES (#{username},#{sex},#{birthday},#{address})

</insert>

</mapper>

#### 加载mapper映射文件

<!-- 加载mapper -->

<mappers>

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

<mapper resource=*"mapper/UserMapper.xml"*/>

</mappers>

#### 编写mapper接口

内容同UserDao接口一样：

**public** **interface** UserMapper {

//根据用户ID来查询用户信息

**public** User findUserById(**int** id);

//根据用户名称来模糊查询用户信息列表

**public** List<User> findUsersByName(String username);

//添加用户

**public** **void** insertUser(User user);

}

#### 编写测试代码

**public** **class** UserMapperTest {

// 声明全局的SqlSessionFactory

**private** SqlSessionFactory sqlSessionFactory;

@Before

**public** **void** setUp() **throws** Exception {

// 1、读取配置文件

String resource = "SqlMapConfig.xml";

InputStream inputStream = Resources.*getResourceAsStream*(resource);

// 2、根据配置文件创建SqlSessionFactory

sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(inputStream);

}

@Test

**public** **void** testFindUserById() {

// 创建SqlSession

SqlSession sqlSession = sqlSessionFactory.openSession();

// 通过SqlSession，获取mapper接口的动态代理对象

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

// 调用mapper对象的方法

User user = userMapper.findUserById(1);

System.*out*.println(user);

// 关闭SqlSession

sqlSession.close();

}

@Test

**public** **void** testFindUsersByName() {

// 创建SqlSession

SqlSession sqlSession = sqlSessionFactory.openSession();

// 通过SqlSession，获取mapper接口的动态代理对象

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

// 调用mapper对象的方法

List<User> list = userMapper.findUsersByName("小明");

System.*out*.println(list);

// 关闭SqlSession

sqlSession.close();

}

@Test

**public** **void** testInsertUser() {

// 创建SqlSession

SqlSession sqlSession = sqlSessionFactory.openSession();

// 通过SqlSession，获取mapper接口的动态代理对象

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

//构造User对象

User user = **new** User();

user.setUsername("东哥4");

user.setAddress("清河宝盛西里4");

// 调用mapper对象的方法

userMapper.insertUser(user);

System.*out*.println(user.getId());

//执行SqlSession的commit操作

sqlSession.commit();

// 关闭SqlSession

sqlSession.close();

}

}

# Mybatis全局配置文件

SqlMapConfig.xml是mybatis的全局配置文件，它的名称可以是任意命名的。

## 全部配置内容

**SqlMapConfig.xml的配置内容和顺序如下（顺序不能乱）：**

Properties（属性）

Settings（全局参数设置）

typeAliases（类型别名）

typeHandlers（类型处理器）

objectFactory（对象工厂）

plugins（插件）

environments（环境信息集合）

environment（单个环境信息）

transactionManager（事物）

dataSource（数据源）

mappers（映射器）

## 常用配置详解

### Properties

SqlMapConfig.xml文件中可以引用java属性文件中的配置信息

db.properties配置信息如下：

db.driver=com.mysql.jdbc.Driver

db.url=jdbc:mysql://localhost:3306/mybatis?useUnicode=true&characterEncoding=utf8

db.username=root

db.password=root

SqlMapConfig.xml使用properties标签后，如下所示：

<!-- 通过properties标签，读取java配置文件的内容 -->

<properties resource=*"db.properties"* />

<!-- 配置mybatis的环境信息 -->

<environments default=*"development"*>

<environment id=*"development"*>

<!-- 配置JDBC事务控制，由mybatis进行管理 -->

<transactionManager type=*"JDBC"*></transactionManager>

<!-- 配置数据源，采用dbcp连接池 -->

<dataSource type=*"POOLED"*>

<property name=*"driver"* value=*"${db.driver}"*/>

<property name=*"url"* value=*"${db.url}"*/>

<property name=*"username"* value=*"${db.username}"*/>

<property name=*"password"* value=*"${db.password}"*/>

</dataSource>

</environment>

</environments>

**使用${}，可以引用已经加载的java配置文件中的信息。**

注意：mybatis将按照下面的顺序加载属性：

* Properties标签体内定义的属性首先被读取
* Properties引用的属性会被读取，如果发现上面已经有同名的属性了，那后面会覆盖前面的值
* parameterType接收的值会最后被读取，如果发现上面已经有同名的属性了，那后面会覆盖前面的值

所以说，mybatis读取属性的顺序**由高到低分别是：parameterType接收的属性值、properties引用的属性、properties标签内定义的属性。**

### Settings

mybatis全局配置参数，全局参数将会影响mybatis的运行行为。

详细参见“mybatis学习资料/mybatis-settings.xlsx”文件







### typeAliases

别名是使用是为了在映射文件中，更方便的去指定入参和结果集的类型，不再用写很长的一段全限定名。

#### mybatis支持的别名

|  |  |
| --- | --- |
| 别名 | 映射的类型 |
| \_byte | byte |
| \_long | long |
| \_short | short |
| \_int | int |
| \_integer | int |
| \_double | double |
| \_float | float |
| \_boolean | boolean |
| string | String |
| byte | Byte |
| long | Long |
| short | Short |
| int | Integer |
| integer | Integer |
| double | Double |
| float | Float |
| boolean | Boolean |
| date | Date |
| decimal | BigDecimal |
| bigdecimal | BigDecimal |

#### 自定义别名

SqlMapConfig.xml配置信息如下：

<!-- 定义别名 -->

<typeAliases>

<!-- 单个定义别名 -->

<typeAlias type=*"cn.itcast.mybatis.po.User"* alias=*"user"*/>

<!-- 批量定义别名（推荐） -->

<!-- [name]：指定批量定义别名的类包，别名为类名（首字母大小写都可） -->

<package name=*"cn.itcast.mybatis.po"*/>

</typeAliases>

### mappers

#### <mapper resource=’’/>

使用相对于类路径的资源

如：<mapper resource="sqlmap/User.xml" />

#### <mapper url=’’/>

使用完全限定路径

如：<mapper url="file:///D:\workspace\_spingmvc\mybatis\_01\config\sqlmap\User.xml" />

#### <mapper class=’’/>

使用mapper接口的全限定名

如：<mapper class="cn.itcast.mybatis.mapper.UserMapper"/>

**注意：此种方法要求mapper接口和mapper映射文件要名称相同，且放到同一个目录下**；

#### <package name=’’/>（推荐）

注册指定包下的所有映射文件

如：<package name="cn.itcast.mybatis.mapper"/>

**注意：此种方法要求mapper接口和mapper映射文件要名称相同，且放到同一个目录下**；

# Mybatis映射文件（核心）

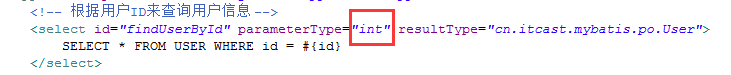
## 输入映射

### ParameterType

指定输入参数的java类型，可以使用别名或者类的全限定名。它可以接收简单类型、POJO、HashMap。

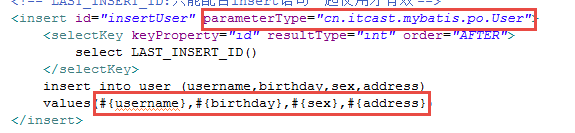
#### 传递简单类型

参考入门需求：根据用户ID查询用户信息。



#### 传递POJO对象

参考入门需求：添加用户。



#### 传递POJO包装对象

开发中通过pojo传递查询条件 ，查询条件是综合的查询条件，不仅包括用户查询条件还包括其它的查询条件（比如将用户购买商品信息也作为查询条件），这时可以使用包装对象传递输入参数。

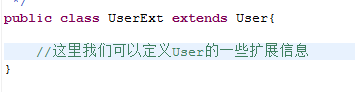
##### 需求

综合查询用户信息，需要传入查询条件复杂，比如（用户信息、订单信息、商品信息）。

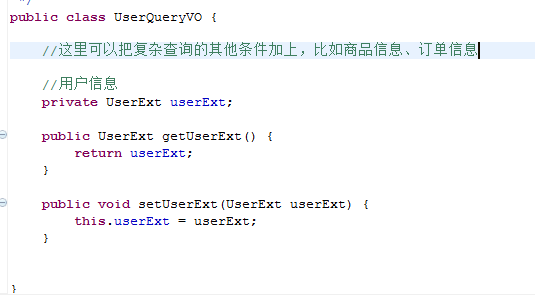
##### 定义包装对象

一般User.java类要和数据表表字段一致，最好不要在这里面添加其他字段，第二天学习mybatis的逆向工程时，会根据表结构，生成po类，如果在po类中扩展字段，此时会被覆盖掉。

所以针对要扩展的po类，我们需要创建一个扩展类，来继承它。



定义POJO包装类：



##### 编写Mapper接口

//通过包装类来进行复杂的用户信息综合查询

**public** List<UserExt> findUserList(UserQueryVO userQueryVO);

##### 编写mapper映射文件

<!-- 通过包装类来进行复杂的用户信息综合查询 -->

<select id=*"findUserList"* parameterType=*"userQueryVO"* resultType=*"userExt"*>

SELECT \* FROM USER WHERE sex=#{userExt.sex} AND username LIKE '%${userExt.username}%'

</select>

注意：入参的类型变为UserQueryVO、结果集的类型变为UserExt，#{}里面的参数变为UserQueryVO对象中的userExt属性的sex和username子属性。

##### 编写测试代码

@Test

**public** **void** findUserListTest() {

// 创建SqlSession

SqlSession sqlSession = sqlSessionFactory.openSession();

// 通过SqlSession，获取mapper接口的动态代理对象

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

//构造userQueryVO对象

UserQueryVO userQueryVO = **new** UserQueryVO();

// 构造UserExt对象

UserExt userExt = **new** UserExt();

userExt.setSex("1");

userExt.setUsername("小明");

userQueryVO.setUserExt(userExt);

// 调用mapper对象的方法

List<UserExt> list = userMapper.findUserList(userQueryVO);

System.*out*.println(list);

// 关闭SqlSession

sqlSession.close();

}

#### 传递HashMap（练习）

同传递POJO对象一样，**map的key相当于pojo的属性**。

##### 映射文件

<!-- 传递hashmap综合查询用户信息 -->

<select id=*"findUserByHashmap"* parameterType=*"hashmap"* resultType=*"user"*>

select \* from user where id=#{id} and username like '%${username}%'

</select>

上边红色标注的是hashmap的key。

##### 测试代码

**Public void** testFindUserByHashmap()**throws** Exception{

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获限mapper接口实例

UserMapper userMapper = session.getMapper(UserMapper.**class**);

//构造查询条件Hashmap对象

HashMap<String, Object> map = **new** HashMap<String, Object>();

map.put("id", 1);

map.put("username", "管理员");

//传递Hashmap对象查询用户列表

List<User>list = userMapper.findUserByHashmap(map);

//关闭session

session.close();

}

异常测试：

传递的map中的key和sql中解析的key不一致。

测试结果没有报错，只是通过key获取值为空。

## 输出映射

### resultType

先带着同学们看下原先resultType作为输出结果映射时，它的特点，如何再把列名改为别名，看看是否还能不能映射成功。

#### 使用方法

使用resultType进行结果映射时，查询的列名和映射的pojo属性名完全一致，该列才能映射成功。

如果查询的列名和映射的pojo属性名全部不一致，则不会创建pojo对象；

如果查询的列名和映射的pojo属性名有一个一致，就会创建pojo对象。

#### 输出简单类型

当输出结果只有一列时，可以使用ResultType指定简单类型作为输出结果类型。

##### 需求

综合查询用户总数，需要传入查询条件复杂，比如（用户信息、订单信息、商品信息）。

##### Mapper映射文件

<!-- 综合查询用户信息总数，需要传入查询条件复杂，比如（用户信息、订单信息、商品信息） -->

<select id=*"findUsersCount"* parameterType=*"UserQueryVO"*

resultType=*"int"*>

SELECT count(1) FROM USER WHERE sex = #{userExt.sex} AND username LIKE '%${userExt.username}%'

</select>

##### Mapper接口

//综合查询用户信息总数。学习：resultType输出简单类型

**public** int findUsersCount(UserQueryVO vo);

##### 测试代码

@Test

**public** **void** testFindUsersCount() {

// 创建SqlSession

SqlSession sqlSession = sqlSessionFactory.openSession();

// 通过SqlSession，获取mapper接口的动态代理对象

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

//构造userQueryVO对象

UserQueryVO userQueryVO = **new** UserQueryVO();

// 构造UserExt对象

UserExt userExt = **new** UserExt();

userExt.setSex("1");

userExt.setUsername("小明");

userQueryVO.setUserExt(userExt);

**int** count = mapper.findUsersCount(userQueryVO);

System.*out*.println(count); // 关闭SqlSession

sqlSession.close();

}

#### 输出POJO单个对象和列表

注意：输出单个pojo对象和pojo列表（盛放pojo对象）时，mapper映射文件中的resultType的类型是一样的，mapper接口的方法返回值不同。

##### Mapper映射文件

Mapper映射文件是同一个

<select id=*"findUsersByName"* parameterType=*"java.lang.String"* resultType=*"cn.itcast.mybatis.po.User"*>

SELECT \* FROM USER WHERE username LIKE '%${value}%'

</select>

##### Mapper接口

下面看下mapper接口的不同之处

1. 输出单个pojo对象

//根据用户名称来模糊查询用户信息

**public** User findUsersByName(String username);

1. 输出pojo列表

//根据用户名称来模糊查询用户信息列表

**public** List<User> findUsersByName(String username);

**总结：**同样的mapper映射文件，返回单个对象和对象列表时，mapper接口在生成动态代理的时候，**会根据返回值的类型，决定调用selectOne方法还是selectList方法。**

### resultMap

resultMap可以进行高级结果映射（**一对一、一对多映射，第二天讲解**）。

#### 使用方法

如果查询出来的列名和属性名不一致，通过定义一个resultMap将列名和pojo属性名之间作一个映射关系。

1. 定义resultMap
2. 使用resultMap作为statement的输出映射类型。

#### 需求

把下面SQL的输出结果集进行映射

SELECT id id\_,username username\_,sex sex\_ FROM USER WHERE id = 1

#### Mapper映射文件

定义resultMap：

<!-- 定义resultMap -->

<!--

[id]：定义resultMap的唯一标识

[type]：定义该resultMap最终映射的pojo对象

[id标签]：映射结果集的唯一标识列，如果是多个字段联合唯一，则定义多个id标签

[result标签]：映射结果集的普通列

[column]：SQL查询的列名，如果列有别名，则该处填写别名

[property]：pojo对象的属性名

-->

<resultMap type=*"user"* id=*"userResultMap"*>

<id column=*"id\_"* property=*"id"*/>

<result column=*"username\_"* property=*"username"*/>

<result column=*"sex\_"* property=*"sex"*/>

</resultMap>

定义statement：

<!-- 根据ID查询用户信息（学习resultMap） -->

<select id=*"findUserByIdResultMap"* parameterType=*"int"* resultMap=*"userResultMap"*>

SELECT id id\_,username username\_,sex sex\_ FROM USER WHERE id = #{id}

</select>

#### Mapper接口定义

//根据ID查询用户信息（学习resultMap）

**public** User findUserByIdResultMap(**int** id);

定义Statement使用resultMap映射结果集时，Mapper接口定义**方法的返回值类型为mapper映射文件中resultMap的type类型。**

#### 测试代码

@Test

**public** **void** findUserByIdResultMapTest() {

// 创建SqlSession

SqlSession sqlSession = sqlSessionFactory.openSession();

// 通过SqlSession，获取mapper接口的动态代理对象

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

// 调用mapper对象的方法

User user = userMapper.findUserByIdResultMap(1);

System.*out*.println(user);

// 关闭SqlSession

sqlSession.close();

}

## 动态SQL（重点）

通过Mybatis提供的各种动态标签实现动态拼接sql，使得mapper映射文件在编写SQL时更加灵活，方便。常用动态SQL标签有：if、where、foreach；

### If和where

* If标签：作为判断入参来使用的，如果符合条件，则把if标签体内的SQL拼接上。

**注意：用if进行判断是否为空时，不仅要判断null，也要判断空字符串‘’；**

* Where标签：会去掉条件中的第一个and符号。

#### 需求

用户信息综合查询列表和用户信息综合查询总数这两个statement的定义使用动态SQL。

#### 映射文件

<!-- 综合查询用户信息，需要传入查询条件复杂，比如（用户信息、订单信息、商品信息） -->

<select id=*"findUsersByQueryVO"* parameterType=*"cn.itcast.mybatis.po.QueryUserVO"*

resultType=*"User"*>

SELECT \* FROM USER

<where>

<if test=*"userExt != null"*>

<if test=*"userExt.sex != null and userExt.sex != ''"*>

AND sex = #{userExt.sex}

</if>

<if test=*"userExt.username != null and userExt.username != ''"*>

AND username LIKE '%${userExt.username}%'

</if>

</if>

</where>

</select>

<!-- 综合查询用户信息总数，需要传入查询条件复杂，比如（用户信息、订单信息、商品信息） -->

<select id=*"findUsersCount"* parameterType=*"QueryUserVO"*

resultType=*"int"*>

SELECT count(1) FROM USER

<where>

<if test=*"userExt != null"*>

<if test=*"userExt.sex != null and userExt.sex != ''"*>

AND sex = #{userExt.sex}

</if>

<if test=*"userExt.username != null and userExt.username != ''"*>

AND username LIKE '%${userExt.username}%'

</if>

</if>

</where>

</select>

#### Mapper接口

//通过包装类来进行复杂的用户信息综合查询

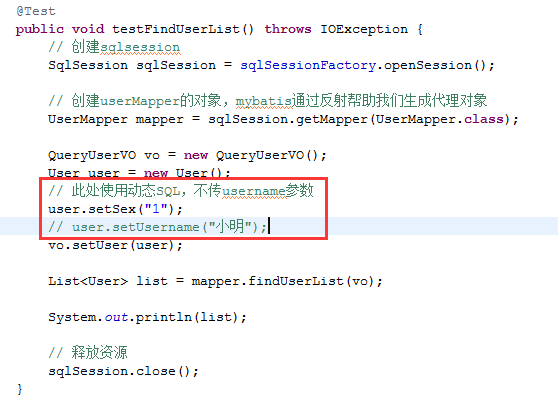
**public** List<UserExt> findUserList(UserQueryVO userQueryVO);

//综合查询用户总数

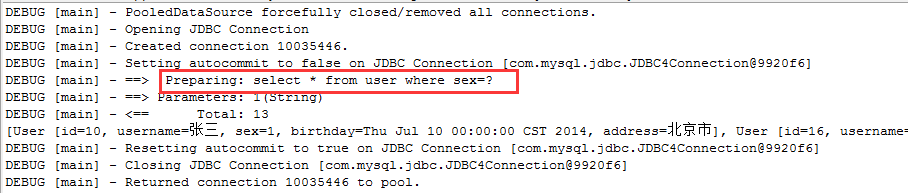
**public** int findUsersCount(UserQueryVO userQueryVO);

#### 测试代码

不传用户名：



输出的SQL如下（也不包含用户名）：



通过测试可以得知，打印出的SQL语句确实会随着条件的满足情况而不一样。

### SQL片段

Mybatis提供了SQL片段的功能，可以提高SQL的可重用性。

#### 定义SQL片段

使用sql标签来定义一个SQL片段：

<!-- 定义SQL片段 -->

<!--

[sql标签]：定义一个SQL片段

[id]：SQL片段的唯一标识

建议：

1、SQL片段中的内容最好是以单表来定义

2、如果是查询字段，则不要写上SELECT

3、如果是条件语句，则不要写上WHERE

-->

<sql id=*"select\_user\_where"*>

<if test=*"userExt != null"*>

<if test=*"userExt.sex != null and userExt.sex != ''"*>

AND sex = #{userExt.sex}

</if>

<if test=*"userExt.username != null and userExt.username != ''"*>

AND username LIKE '%${userExt.username}%'

</if>

</if>

</sql>

#### 引用SQL片段

使用**<include refid=’’ />** 来引用SQL片段：

<!-- 根据用户id来查询用户信息（使用SQL片段） -->

<!--

[include标签]：引用已经定义好的SQL片段

[refid]：引用的SQL片段id

-->

<select id=*"findUserList"* parameterType=*"userQueryVO"* resultType=*"userExt"*>

SELECT \* FROM USER

<where>

<include refid=*"select\_user\_where"*/>

</where>

</select>

<!-- 综合查询用户信息总数，需要传入查询条件复杂，比如（用户信息、订单信息、商品信息） -->

<select id=*"findUsersCount"* parameterType=*"QueryUserVO"*

resultType=*"int"*>

SELECT count(1) FROM USER

<where>

<include refid=*"select\_user\_where"*/>

</where>

</select>

### Foreach

向sql传递数组或List时，mybatis使用foreach解析数组里的参数并拼接到SQL中。

#### 传递pojo对象中的List集合

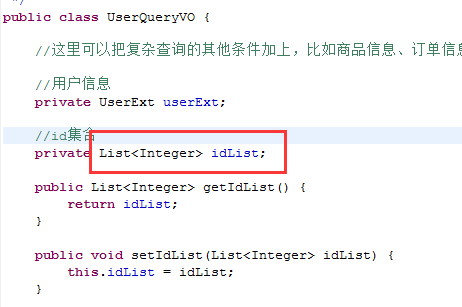
##### 需求

在用户查询列表和查询总数的statement中增加多个id输入查询。

##### SQL

SELECT \* FROM user WHERE **id IN (1,10,16)**

##### 定义pojo中的List属性



##### 映射文件

<!-- [foreach标签]：表示一个foreach循环 -->

<!-- [collection]：集合参数的名称，如果是直接传入集合参数，则该处的参数名称只能填写[list]。 -->

<!-- [item]：每次遍历出来的对象 -->

<!-- [open]：开始遍历时拼接的串 -->

<!-- [close]：结束遍历时拼接的串 -->

<!-- [separator]：遍历出的每个对象之间需要拼接的字符 -->

<if test=*"idList != null and idList.size > 0"*>

<foreach collection=*"idList"* item=*"id"* open=*"AND id IN ("* close=*")"* separator=*","*>

#{id}

</foreach>

</if>

##### Mapper接口

//根据用户ID的集合查询用户列表（学习foreach标签之通过POJO对象传ID集合）

**public** List<UserExt> findUserList(UserQueryVO vo);

##### 测试代码

@Test

**public** **void** testFindUserList() {

// 创建SqlSession

SqlSession sqlSession = sqlSessionFactory.openSession();

// 通过SqlSession，获取mapper接口的动态代理对象

UserMapper mapper = sqlSession.getMapper(UserMapper.**class**);

// 构造QueryUserVO对象

QueryUserVO vo = **new** QueryUserVO();

// UserExt ext = new UserExt();

// ext.setUsername("小明");

// ext.setSex("1");

// vo.setUserExt(ext);

// 创建用户ID集合，然后设置到QueryUserVO对象中

List<Integer> idList = **new** ArrayList<Integer>();

idList.add(1);

idList.add(10);

idList.add(16);

vo.setIdList(idList);

// 调用mapper代理对象的方法

List<UserExt> list = mapper.findUserList(vo);

System.*out*.println(list);

// 关闭SqlSession

sqlSession.close();

}

#### 直接传递List集合（自学）

##### 需求

根据用户ID的集合查询用户列表

##### SQL

SELECT \* FROM user WHERE id IN (1,10,16)

##### 映射文件

<!-- 根据用户ID的集合查询用户列表（学习foreach标签之直接传ID集合） -->

<!--

[foreach标签]：表示一个foreach循环

[collection]：集合参数的名称，如果是直接传入集合参数，则该处的参数名称只能填写[list]。

[item]：定义遍历集合之后的参数名称

[open]：开始遍历之前需要拼接的SQL串

[close]：结束遍历之后需要拼接的SQL串

[separator]：遍历出的每个对象之间需要拼接的字符

-->

<select id=*"findUsersByIdList"* parameterType=*"****java.util.List****"* resultType=*"user"*>

SELECT \* FROM USER

<where>

<if test=*"****list*** *!= null and* ***list****.size > 0"*>

<foreach collection=*"****list****"* item=*"id"* open=*"AND id IN ("* close=*")"* separator=*","*>

#{id}

</foreach>

</if>

</where>

</select>

##### Mapper接口

//根据用户ID的集合查询用户列表（学习foreach标签之直接传ID集合）

**public** List<User> findUsersByIdList (List<Integer> idList);

##### 测试代码

@Test

**public** **void** findUsersByIdListTest() {

// 创建SqlSession

SqlSession sqlSession = sqlSessionFactory.openSession();

// 通过SqlSession，获取mapper接口的动态代理对象

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

// 构造List<Integer>集合

List<Integer> idList = **new** ArrayList<Integer>();

idList.add(1);

idList.add(10);

idList.add(16);

// 调用mapper对象的方法

List<User> list = userMapper.findUsersByIdList (idList);

System.*out*.println(list);

// 关闭SqlSession

sqlSession.close();

}

# mybatis与hibernate的区别及各自应用场景

Mybatis技术特点：

1. 通过直接编写SQL语句，可以直接对SQL进行性能的优化；
2. 学习门槛低，学习成本低。只要有SQL基础，就可以学习mybatis，而且很容易上手；
3. 由于直接编写SQL语句，所以灵活多变，代码维护性更好。
4. 不能支持数据库无关性，即数据库发生变更，要写多套代码进行支持，移植性不好。
5. 需要编写结果映射。

Hibernate技术特点：

1. 标准的orm框架，程序员不需要编写SQL语句。
2. 具有良好的数据库无关性，即数据库发生变化的话，代码无需再次编写。
3. 学习门槛高，需要对数据关系模型有良好的基础，而且在设置OR映射的时候，需要考虑好性能和对象模型的权衡。
4. 程序员不能自主的去进行SQL性能优化。

Mybatis应用场景：

需求多变的互联网项目，例如电商项目。

Hibernate应用场景：

需求明确、业务固定的项目，例如OA项目、ERP项目等。

# 关联查询映射

## 分析数据模型

### 思路

1. 每张表记录的数据内容

分模块对每张表记录的内容进行熟悉，相当于你学习系统需求（功能）的过程。

1. 每张表重要的字段

主键、外键、非空字段

1. 数据库级别表与表的关系

外键关系

1. 表与表之间的业务关系

在分析表与表之间的业务关系时一定要建立 在某个业务意义基础上去分析。

### 图形分析

一对多

订单表

id：订单ID

Number：订单号

User\_id：用户id（外键）

订单明细表

Orders\_id：订单ID（外键）

Items\_id：商品ID（外键）

外键

一对一

外键

一对多

一对一

外键

一对一

一对多

多对多

多对多

商品表

Id：商品ID

用户表

Id：用户ID

### 数据库表之间有外键关系的业务关系

**user和orders：**

user---->orders：一个用户可以创建多个订单，一对多

orders--->user：一个订单只由一个用户创建，一对一

**orders和orderdetail：**

orders-🡪orderdetail：一个订单可以包括 多个订单明细，因为一个订单可以购买多个商品，每个商品的购买信息在orderdetail记录，一对多关系

orderdetail--> orders：一个订单明细只能包括在一个订单中，一对一

**orderdetail和itesm：**

orderdetail---》itesms：一个订单明细只对应一个商品信息，一对一

items--> orderdetail:一个商品可以包括在多个订单明细 ，一对多

### 数据库表之间没有外键关系的业务关系

**Orders和items：**

这两张表没有直接的外键关系，通过业务及数据库的间接关系分析出它们是多对多的关系。

Orders🡪 orderdetail –>items：一个订单可以有多个订单明细，一个订单明细对应一个商品，所以一个订单对应多个商品

Items-🡪orderdetail🡪orders：一个商品可以对应多个订单明细，一个订单明细对应一个订单，所以一个商品对应多个订单

**User和items：**

这两张表没有直接的外键关系，通过业务及数据库的间接关系分析出它们是多对多的关系。

User🡪orders🡪orderdetail🡪items：一个用户有多个订单，一个订单有多个订单明细、一个订单明细对应一个商品，所以一个用户对应多个商品

Items🡪orderdetail🡪orders🡪user：一个商品对应多个订单明细，一个订单明细对应一个订单，一个订单对应一个用户，所以一个商品对应多个用户

## 一对一查询

### 需求

查询订单信息，关联查询创建订单的用户信息

### SQL语句

确定查询的主表：订单表

确定查询的关联表：用户表

关联查询使用内链接？还是外链接？

Select

Orders.id,

Orders.user\_id,

orders.number,

orders.createtime,

orders.note,

user.username,

user.address

from orders,user

where orders.user\_id = user.id

### resultType

复杂查询时，单表对应的po类已不能满足输出结果集的映射。

所以要根据需求建立一个扩展类来作为resultType的类型。

#### 创建po类

//通过此类映射订单和用户查询的结果，让此类继承包括 字段较多的pojo类

**public** **class** OrdersExt **extends** Orders{

//添加用户属性

/\*USER.username,

USER.address \*/

**private** String username;

**private** String address;

**public** String getUsername() {

**return** username;

}

**public** **void** setUsername(String username) {

**this**.username = username;

}

**public** String getAddress() {

**return** address;

}

**public** **void** setAddress(String address) {

**this**.address = address;

}

}

#### 编写mapper接口

创建OrdersMapper接口类，在类中添加以下内容：

// 进行订单信息查询，包括用户的名称和地址信息

**public** List<OrdersExt> findOrdersUser();

#### 编写映射文件

<mapper namespace=*"cn.itcast.mybatis.mapper.OrdersMapper"*>

<!-- 定义查询订单表列名的SQL片段 -->

<sql id=*"select\_orders"*>

Orders.id,

Orders.user\_id,

orders.number,

orders.createtime,

orders.note

</sql>

<!-- 定义查询用户表列名的SQL片段 -->

<sql id=*"select\_user"*>

user.username,

user.address

</sql>

<!-- 进行订单信息查询，包括用户的名称和地址信息 -->

<select id=*"findOrdersUser"* resultType=*"OrdersExt"*>

Select

<include refid=*"select\_orders"* />

<include refid=*"select\_user"*></include>

from orders,user

where orders.user\_id = user.id

</select>

</mapper>

#### 加载映射文件

<!-- 批量加载mapper文件，需要mapper接口文件和mapper映射文件名称相同且在同一个包下 -->

<package name=*"cn.itcast.mybatis.mapper"*/>

#### 编写测试代码

@Test

**public** **void** testFindOrdersUser() {

// 创建sqlSession

SqlSession sqlSession = sqlSessionFactory.openSession();

// 通过SqlSession构造usermapper的代理对象

OrdersMapper ordersMapper = sqlSession.getMapper(OrdersMapper.**class**);

// 调用usermapper的方法

List<OrdersExt> list = ordersMapper.findOrdersUser();

System.*out*.println(list);

// 释放SqlSession

sqlSession.close();

}

### resultMap

#### 修改po类

在Orders类中，添加User对象

**public** **class** Orders {

**private** Integer id;

**private** Integer userId;

**private** String number;

**private** Date createtime;

**private** String note;

//用户信息

**private** User user;

#### 编写mapper接口

// 进行订单信息查询，包括用户的名称和地址信息（resultMap）

**public** List<OrdersExt> findOrdersUserRstMap();

#### 编写映射文件

<!-- 进行订单信息查询，包括用户的名称和地址信息 (ResultMap) -->

<select id=*"findOrdersUserRstMap"* resultMap=*"OrdersUserRstMap"*>

Select

<include refid=*"select\_orders"* />

,

<include refid=*"select\_user"*></include>

from orders,user

where orders.user\_id = user.id

</select>

<!-- 定义orderUserResultMap -->

<resultMap type=*"* *cn.itcast.mybatis.po.Orders"* id=*"OrdersUserRstMap"*>

<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"* />

<!-- 映射一对一关联关系的用户对象-->

<!--

property：指定关联对象要映射到Orders的哪个属性上

javaType：指定关联对象所要映射的java类型

-->

<!-- id标签：指定关联对象结果集的唯一标识，很重要，不写不会报错，但是会影响性能 -->

<association property=*"user"* javaType=*"cn.itcast.mybatis.po.User"*>

<id column=*"user\_id"* property=*"id"* />

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

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

</association>

</resultMap>

#### 编写测试代码

@Test

**public** **void** testFindOrdersUserRstMap() {

// 创建sqlSession

SqlSession sqlSession = sqlSessionFactory.openSession();

// 通过SqlSession构造usermapper的代理对象

OrdersMapper ordersMapper = sqlSession.getMapper(OrdersMapper.**class**);

// 调用usermapper的方法

List<Orders> list = ordersMapper.findOrdersUserRstMap();

//此处我们采用debug模式来跟踪代码，然后验证结果集是否正确

System.*out*.println(list);

// 释放SqlSession

sqlSession.close();

}

### 一对一小结

实现一对一查询：

**resultType**：使用resultType实现较为简单，如果pojo中没有包括查询出来的列名，需要增加列名对应的属性，即可完成映射。

如果没有查询结果的特殊要求建议使用resultType。

**resultMap**：需要单独定义resultMap，实现有点麻烦，如果对查询结果有特殊的要求，使用resultMap可以完成将关联查询映射pojo的对象属性中。

**resultMap可以实现延迟加载，resultType无法实现延迟加载。**

## 一对多查询

一对多查询和一对一查询的配置基本类似。只是如果使用resultMap的话，映射一对多关联关系要使用collection标签。

### 需求

查询订单信息及订单明细信息

### SQL语句

确定主查询表：订单表

确定关联查询表：订单明细表

在一对一查询基础上添加订单明细表关联即可。

Select

Orders.id,

Orders.user\_id,

orders.number,

orders.createtime,

orders.note,

user.username,

user.address,

orderdetail.id detail\_id,

orderdetail.items\_id,

orderdetail.items\_num

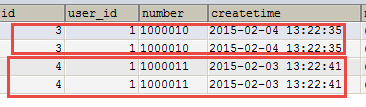
from orders,user,orderdetail

where orders.user\_id = user.id

and orders.id = orderdetail.orders\_id

### 分析

使用resultType将上边的 查询结果映射到pojo中，订单信息将会重复。



要求：

对orders映射不能出现重复记录。

**在orders.java类中添加List<Orderdetail> detailList属性。**

**最终会将订单信息映射到orders中，订单所对应的订单明细映射到orders中的detailList属性中。**



**映射成的orders记录数为两条（orders信息不重复）**

**每个orders中的detailList属性存储了该订单所对应的订单明细集合**。

### 修改PO类

在Orders类中添加以下属性，并提供get/set方法：

//订单明细

**private** List<Orderdetail> detailList;

### 编写mapper接口

// 查询订单信息及订单明细信息（一对多映射之使用resultMap）

**public** List<Orders> findOrdersAndOrderdetailRstMap();

### 编写映射文件

<!-- 定义OrdersAndOrderdetailRstMap -->

<!-- extends：继承已有的ResultMap，值为继承的ResultMap的唯一标示 -->

<resultMap type=*"Orders"* id=*"OrdersAndOrderdetailRstMap"*

extends=*"OrdersUserRstMap"*>

<!-- 映射关联关系（一对多） -->

<!-- collection标签：定义一个一对多关系

ofType：指定该集合参数所映射的类型

-->

<collection property=*"detailList"* ofType=*"Orderdetail"*>

<id column=*"detail\_id"* property=*"id"* />

<result column=*"items\_id"* property=*"itemsId"* />

<result column=*"items\_num"* property=*"itemsNum"* />

</collection>

</resultMap>

<!-- 查询订单信息，包括用户名称、用户地址，订单商品信息（嵌套结果） -->

<select id=*"findOrdersAndOrderdetailRstMap"* resultMap=*"OrdersAndOrderdetailRstMap"*>

Select

<include refid=*"select\_orders"* />

,

<include refid=*"select\_user"*/>

,

orderdetail.id detail\_id,

orderdetail.items\_id,

orderdetail.items\_num

from orders,user,orderdetail

where orders.user\_id = user.id

and

orders.id = orderdetail.orders\_id

</select>

**resultMap的extends属性：可以用此属性来继承一个已有的resultmap。但是它继承的resultMap的type和它本身的type要保持一致。**

### 编写测试代码

@Test

**public** **void** testFindOrdersAndOrderdetailRstMap() {

// 创建sqlSession

SqlSession sqlSession = sqlSessionFactory.openSession();

// 通过SqlSession构造usermapper的代理对象

OrdersMapper ordersMapper = sqlSession.getMapper(OrdersMapper.**class**);

// 调用usermapper的方法

List<Orders> list = ordersMapper.findOrdersAndOrderdetailRstMap();

//此处我们采用debug模式来跟踪代码，然后验证结果集是否正确

System.*out*.println(list);

// 释放SqlSession

sqlSession.close();

}

### 一对多小结

mybatis使用resultMap的collection对关联查询的多条记录映射到一个list集合属性中。

**使用resultType实现：**

需要对结果集进行二次处理。

将订单明细映射到orders中的orderdetails中，需要自己处理，使用双重循环遍历，去掉重复记录，将订单明细放在orderdetails中。

## 多对多查询

### 需求

查询用户信息及用户购买的商品信息，**要求将关联信息映射到主pojo的pojo属性中**

### SQL语句

查询主表：user

查询关联表：orders、orderdetail、items

Select

Orders.id,

Orders.user\_id,

orders.number,

orders.createtime,

orders.note,

user.username,

user.address,

orderdetail.id detail\_id,

orderdetail.items\_id,

orderdetail.items\_num

items.name items\_name,

items.detail items\_detail

FROM

orders,

USER,

orderdetail,

items

WHERE user.`id` = orders.`user\_id`

AND orders.`id` = orderdetail.`orders\_id`

AND orderdetail.`items\_id` = items.`id`

### 映射思路

将用户信息映射到user中。

在user类中添加订单列表属性List<Orders> orderslist，将用户创建的订单映射到orderslist

在Orders中添加订单明细列表属性List<Orderdetail> detailList，将订单的明细映射到detailList

在Orderdetail中添加Items属性，将订单明细所对应的商品映射到Items

### 修改PO类

在user类中添加List<Orders> ordersList 属性

// 订单信息

**private** List<Orders> ordersList;

在Orders类中添加List<Orderdetail>属性

//订单明细

**private** List<Orderdetail> detailList;

在Orderdetail类中添加Items属性

//商品信息

**private** Items items;

### 编写mapper接口

//查询用户及用户购买商品信息（多对多映射之使用resultMap）

**public** List<User> findUserAndItemsRstMap();

### 编写映射文件

<!-- 定义UserAndItemsRstMap -->

<resultMap type=*"User"* id=*"UserAndItemsRstMap"*>

<!-- 用户信息 -->

<!-- id：关联查询用户的唯一标示 -->

<id column=*"user\_id"* property=*"id"* />

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

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

<!-- 订单信息 （一个用户有多个订单） -->

<collection property=*"ordersList"* ofType=*"orders"*>

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

<result column=*"user\_id"* property=*"userId"* />

<result column=*"number"* property=*"number"* />

<result column=*"createtime"* property=*"createtime"* />

<result column=*"note"* property=*"note"* />

<!-- 订单明细信息（一个订单有多个订单明细） -->

<collection property=*"detailList"* ofType=*"orderdetail"*>

<id column=*"detail\_id"* property=*"id"* />

<result column=*"items\_id"* property=*"itemsId"* />

<result column=*"items\_num"* property=*"itemsNum"* />

<!-- 商品信息 （一个订单明细对应一个商品） -->

<association property=*"items"* javaType=*"cn.itcast.mybatis.po.Items"*>

<id column=*"items\_id"* property=*"id"* />

<result column=*"items\_name"* property=*"name"* />

<result column=*"items\_detail"* property=*"detail"* />

</association>

</collection>

</collection>

</resultMap>

<!-- 查询用户及用户购买商品信息（多对多映射之使用resultMap） -->

<select id=*"findUserAndItemsRstMap"* resultMap=*"UserAndItemsRstMap"*>

Select

<include refid=*"select\_orders"* />

,

<include refid=*"select\_user"* />

,

<include refid=*"select\_orderdetail"*></include>

,

items.name items\_name,

items.detail items\_detail

from

orders,user,orderdetail,items

where orders.user\_id = user.id

and

orders.id = orderdetail.orders\_id

and orderdetail.items\_id = items.id

</select>

### 编写测试代码

@Test

**public** **void** testFindUserAndItemsRstMap() {

// 创建sqlSession

SqlSession sqlSession = sqlSessionFactory.openSession();

// 通过SqlSession构造usermapper的代理对象

OrdersMapper ordersMapper = sqlSession.getMapper(OrdersMapper.**class**);

// 调用usermapper的方法

List<User> list = ordersMapper.findUserAndItemsRstMap();

// 此处我们采用debug模式来跟踪代码，然后验证结果集是否正确

System.*out*.println(list);

// 释放SqlSession

sqlSession.close();

}

### 多对多查询小结

将查询用户购买的商品信息明细清单，（用户名、用户地址、购买商品名称、购买商品时间、购买商品数量）

针对上边的需求就使用resultType将查询到的记录映射到一个扩展的pojo中，很简单实现明细清单的功能。

一对多是多对多的特例，如下需求：

查询用户购买的商品信息，用户和商品的关系是多对多关系。

需求1：

查询字段：用户账号、用户名称、用户性别、商品名称、商品价格(最常见)

企业开发中常见明细列表，用户购买商品明细列表，

使用resultType将上边查询列映射到pojo输出。

需求2：

查询字段：用户账号、用户名称、购买商品数量、商品明细（鼠标移上显示明细）

使用resultMap将用户购买的商品明细列表映射到user对象中。

总结：

使用resultMap是针对那些对查询结果映射有特殊要求的功能，，比如特殊要求映射成list中包括 多个list。

## 高级映射总结

resultType：

作用：

将查询结果按照sql列名pojo属性名一致性映射到pojo中。

场合：

常见一些明细记录的展示，比如用户购买商品明细，将关联查询信息全部展示在页面时，此时可直接使用resultType将每一条记录映射到pojo中，在前端页面遍历list（list中是pojo）即可。

resultMap：

使用association和collection完成一对一和一对多高级映射（对结果有特殊的映射要求）。

association：

作用：

将关联查询信息映射到一个pojo对象中。

场合：

为了方便查询关联信息可以使用association将关联订单信息映射为用户对象的pojo属性中，比如：查询订单及关联用户信息。

使用resultType无法将查询结果映射到pojo对象的pojo属性中，根据对结果集查询遍历的需要选择使用resultType还是resultMap。

collection：

作用：

将关联查询信息映射到一个list集合中。

场合：

为了方便查询遍历关联信息可以使用collection将关联信息映射到list集合中，比如：查询用户权限范围模块及模块下的菜单，可使用collection将模块映射到模块list中，将菜单列表映射到模块对象的菜单list属性中，这样的作的目的也是方便对查询结果集进行遍历查询。

如果使用resultType无法将查询结果映射到list集合中。

# 延迟加载

## 什么是延迟加载

resultMap中的association和collection标签具有延迟加载的功能。

延迟加载的意思是说，在关联查询时，利用延迟加载，先加载主信息。需要关联信息时再去按需加载关联信息。这样会大大提高数据库性能，因为查询单表要比关联查询多张表速度要快。

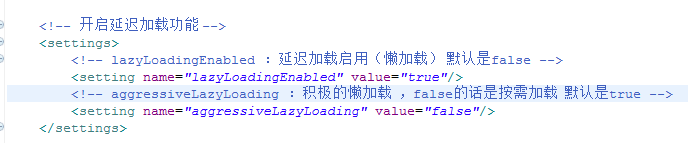
## 设置延迟加载

Mybatis默认是不开启延迟加载功能的，我们需要手动开启。

需要在SqlMapConfig.xml文件中，在<settings>标签中开启延迟加载功能。

**lazyLoadingEnabled、aggressiveLazyLoading**

|  |  |  |  |
| --- | --- | --- | --- |
| 设置项 | 描述 | 允许值 | 默认值 |
| lazyLoadingEnabled | 全局性设置懒加载。如果设为‘false’，则所有相关联的都会被初始化加载。 | true | false | true |
| aggressiveLazyLoading | 当设置为‘true’的时候，懒加载的对象可能被任何懒属性全部加载。否则，每个属性都按需加载。 | true | false | true |



## 使用association进行延迟加载

### 需求

查询订单并且关联查询用户信息（对用户信息的加载要求是按需加载）

### 编写映射文件

需要定义两个mapper的方法对应的statement。

1、只查询订单信息

SELECT \* FROM orders

在查询订单的statement中使用**association去延迟加载（执行）下边的satatement(**关联查询用户信息**)**

<!-- 定义OrdersUserLazyLoadingRstMap -->

<resultMap type=*"cn.itcast.mybatis.po.Orders"* id=*"OrdersUserLazyLoadingRstMap"*>

<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"* />

<!-- 延迟加载用户信息 -->

<!-- select：指定延迟加载需要执行的statement的id（是根据user\_id查询用户信息的statement）

我们使用UserMapper.xml中的findUserById完成根据用户ID（user\_id）查询用户信息

如果findUserById不在本mapper中，前边需要加namespace

-->

<!-- column：主信息表中需要关联查询的列，此处是user\_id -->

<association property=*"user"* select=*"cn.itcast.mybatis.mapper.UserMapper.findUserById"* column=*"user\_id"*></association>

</resultMap>

<!-- 查询订单信息，延迟加载关联查询的用户信息 -->

<select id=*"findOrdersUserLazyLoading"* resultMap=*"OrdersUserLazyLoadingRstMap"*>

SELECT \* FROM orders

</select>

2、关联查询用户信息

通过上边查询到的订单信息中user\_id去关联查询用户信息

使用UserMapper.xml中的findUserById

<select id=*"findUserById"* parameterType=*"int"*

resultType=*"cn.itcast.mybatis.po.User"*>

SELECT \* FROM user WHERE id = #{id}

</select>

上边先去执行findOrdersUserLazyLoading，当需要去查询用户的时候再去执行findUserById，通过resultMap的定义将延迟加载执行配置起来。

### 加载映射文件

<!-- 批量加载mapper文件，需要mapper接口文件和mapper映射文件名称相同且在同一个包下 -->

<package name=*"cn.itcast.mybatis.mapper"*/>

### 编写mapper接口

// 查询订单信息，延迟加载关联查询的用户信息

**public** List<Orders> findOrdersUserLazyLoading();

### 编写测试代码

思路：

1、执行上边mapper方法（findOrdersUserLazyLoading），内部去调用cn.itcast.mybatis.mapper.OrdersMapper中的findOrdersUserLazyLoading只查询orders信息（单表）。

2、在程序中去遍历上一步骤查询出的List<Orders>，当我们调用Orders中的getUser方法时，开始进行延迟加载。

3、执行延迟加载，去调用UserMapper.xml中findUserbyId这个方法获取用户信息。

@Test

**public** **void** testFindOrdersUserLazyLoading() {

// 创建sqlSession

SqlSession sqlSession = sqlSessionFactory.openSession();

// 通过SqlSession构造usermapper的代理对象

OrdersMapper ordersMapper = sqlSession.getMapper(OrdersMapper.**class**);

// 调用usermapper的方法

List<Orders> list = ordersMapper.findOrdersUserLazyLoading();

**for**(Orders orders : list){

System.*out*.println(orders.getUser());

}

// 释放SqlSession

sqlSession.close();

}

## 延迟加载思考

**不使用mybatis提供的association及collection中的延迟加载功能，如何实现延迟加载？？**

实现方法如下：

**定义两个mapper方法：**

1、查询订单列表

2、根据用户id查询用户信息

**实现思路：**

先去查询第一个mapper方法，获取订单信息列表

在程序中（service），按需去调用第二个mapper方法去查询用户信息。

**总之：**

使用延迟加载方法，先去查询**简单的sql（**最好单表，也可以关联查询），再去按需要加载关联查询的其它信息。

# 查询缓存

## mybatis缓存分析

mybatis提供查询缓存，如果缓存中有数据就不用从数据库中获取，用于减轻数据压力，提高系统性能。

sqlSession1

sqlSession 2

sqlSession3...

一级缓存

一级缓存

一级缓存

Mapper(namespace)二级缓存

一级缓存是SqlSession级别的缓存。在操作数据库时需要构造 sqlSession对象，在对象中有一个数据结构（HashMap）用于存储缓存数据。不同的sqlSession之间的缓存数据区域（HashMap）是互相不影响的。

二级缓存是mapper级别的缓存，多个SqlSession去操作同一个Mapper的sql语句，多个SqlSession可以共用二级缓存，二级缓存是跨SqlSession的。

## 一级缓存

### 原理

第一次查询id为1的用户

SqlSession

一级缓存区域

Map

SqlSession

写入

修改、添加、删除用户执行commit

清空

第二次查询id为1的用户

读取

第一次发起查询用户id为1的用户信息，先去找缓存中是否有id为1的用户信息，如果没有，从数据库查询用户信息。

得到用户信息，将用户信息存储到一级缓存中。

如果sqlSession去执行commit操作（执行插入、更新、删除），清空SqlSession中的一级缓存，这样做的目的为了让缓存中存储的是最新的信息，避免脏读。

第二次发起查询用户id为1的用户信息，先去找缓存中是否有id为1的用户信息，缓存中有，直接从缓存中获取用户信息。

**Mybatis默认支持一级缓存。**

### 测试1

@Test

**public** **void** testOneLevelCache() {

SqlSession sqlSession = sqlSessionFactory.openSession();

UserMapper mapper = sqlSession.getMapper(UserMapper.**class**);

// 第一次查询ID为1的用户，去缓存找，找不到就去查找数据库

User user1 = mapper.findUserById(1);

System.*out*.println(user1);

// 第二次查询ID为1的用户

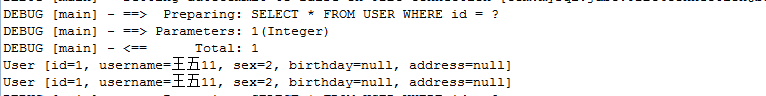
User user2 = mapper.findUserById(1);

System.*out*.println(user2);

sqlSession.close();

}

只输出一次SQL：



### 测试2

@Test

**public** **void** testOneLevelCache() {

SqlSession sqlSession = sqlSessionFactory.openSession();

UserMapper mapper = sqlSession.getMapper(UserMapper.**class**);

// 第一次查询ID为1的用户，去缓存找，找不到就去查找数据库

User user1 = mapper.findUserById(1);

System.*out*.println(user1);

User user = **new** User();

user.setUsername("东哥1");

user.setAddress("清河宝盛西里");

//执行增删改操作，清空缓存

mapper.insertUser(user);

// 第二次查询ID为1的用户

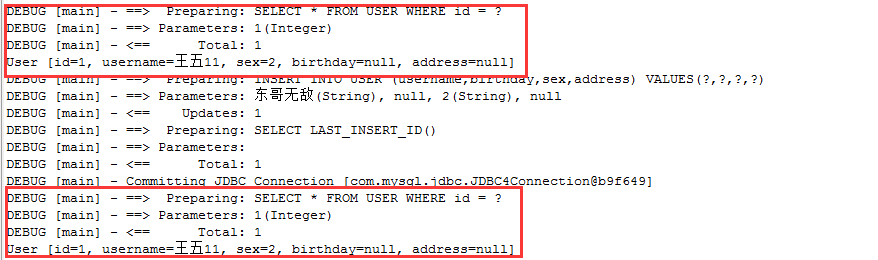
User user2 = mapper.findUserById(1);

System.*out*.println(user2);

sqlSession.close();

}

中间执行了commit操作，同样的查询SQL输出两次：



### 应用

正式开发，是将mybatis和spring进行整合开发，事务控制在service中。

一个service方法中包括 很多mapper方法调用。

service{

//开始执行时，开启事务，创建SqlSession对象

//第一次调用mapper的方法findUserById(1)

//第二次调用mapper的方法findUserById(1)，从一级缓存中取数据

//方法结束，sqlSession关闭

}

如果是执行两次service调用查询相同 的用户信息，不走一级缓存，因为session方法结束，sqlSession就关闭，一级缓存就清空。

## 二级缓存

### 原理

下图是多个sqlSession请求UserMapper的二级缓存图解。



二级缓存是mapper级别的。

第一次调用mapper下的SQL去查询用户信息。查询到的信息会存到该mapper对应的二级缓存区域内。

第二次调用相同namespace下的mapper映射文件中相同的SQL去查询用户信息。会去对应的二级缓存内取结果。

如果调用相同namespace下的mapper映射文件中的增删改SQL，并执行了commit操作。此时会清空该namespace下的二级缓存。

### 开启二级缓存

**Mybatis默认是没有开启二级缓存**

1. 在核心配置文件SqlMapConfig.xml中加入以下内容（开启二级缓存总开关）：

在settings标签中添加以下内容：

<!-- 开启二级缓存总开关 -->

<setting name=*"cacheEnabled"* value=*"true"*/>

1. 在UserMapper映射文件中，加入以下内容，开启二级缓存：

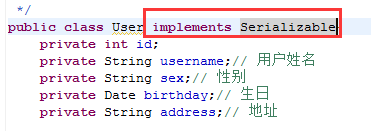
<!-- 开启本mapper下的namespace的二级缓存，默认使用的是mybatis提供的PerpetualCache -->

<cache></cache>

### 实现序列化

由于二级缓存的数据不一定都是存储到内存中，它的存储介质多种多样，所以需要给缓存的对象执行序列化。

如果该类存在父类，那么父类也要实现序列化。



### 测试1

@Test

**public** **void** testTwoLevelCache() {

SqlSession sqlSession1 = sqlSessionFactory.openSession();

SqlSession sqlSession2 = sqlSessionFactory.openSession();

SqlSession sqlSession3 = sqlSessionFactory.openSession();

UserMapper mapper1 = sqlSession1.getMapper(UserMapper.**class**);

UserMapper mapper2 = sqlSession2.getMapper(UserMapper.**class**);

UserMapper mapper3 = sqlSession3.getMapper(UserMapper.**class**);

// 第一次查询ID为1的用户，去缓存找，找不到就去查找数据库

User user1 = mapper1.findUserById(1);

System.*out*.println(user1);

// 关闭SqlSession1

sqlSession1.close();

// 第二次查询ID为1的用户

User user2 = mapper2.findUserById(1);

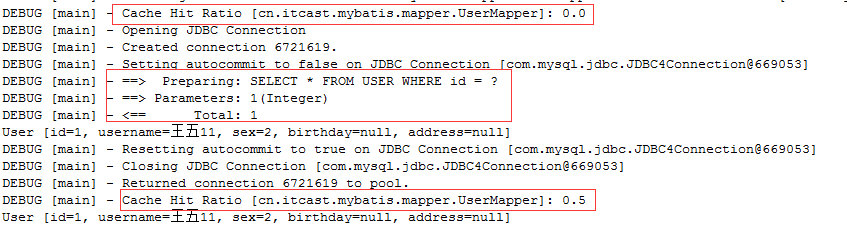
System.*out*.println(user2);

// 关闭SqlSession2

sqlSession2.close();

}

SQL输出结果：



Cache Hit Radio ： 缓存命中率

第一次缓存中没有记录，则命中率0.0；

第二次缓存中有记录，则命中率0.5（访问两次，有一次命中）

### 测试2

@Test

**public** **void** testTwoLevelCache() {

SqlSession sqlSession1 = sqlSessionFactory.openSession();

SqlSession sqlSession2 = sqlSessionFactory.openSession();

SqlSession sqlSession3 = sqlSessionFactory.openSession();

UserMapper mapper1 = sqlSession1.getMapper(UserMapper.**class**);

UserMapper mapper2 = sqlSession2.getMapper(UserMapper.**class**);

UserMapper mapper3 = sqlSession3.getMapper(UserMapper.**class**);

// 第一次查询ID为1的用户，去缓存找，找不到就去查找数据库

User user1 = mapper1.findUserById(1);

System.*out*.println(user1);

// 关闭SqlSession1

sqlSession1.close();

//修改查询出来的user1对象，作为插入语句的参数

user1.setUsername("东哥1");

user1.setAddress("清河宝盛西里");

mapper3.insertUser(user1);

// 提交事务

sqlSession3.commit();

// 关闭SqlSession3

sqlSession3.close();

// 第二次查询ID为1的用户

User user2 = mapper2.findUserById(1);

System.*out*.println(user2);

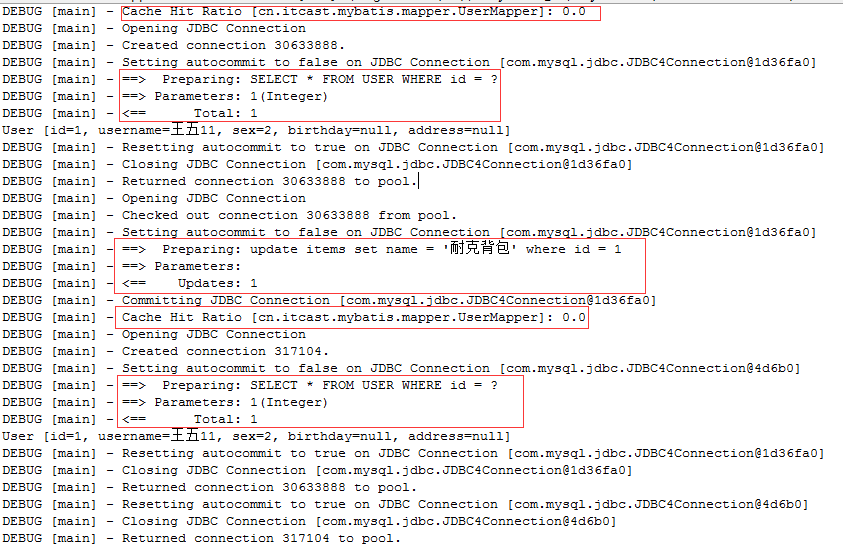
// 关闭SqlSession2

sqlSession2.close();

}

SQL输出结果：

根据SQL分析，确实是清空了二级缓存了。



### 禁用二级缓存

该statement中设置**userCache=false**，可以禁用当前select语句的二级缓存，即每次查询都是去数据库中查询，**默认情况下是true**，即该statement使用二级缓存。

<select id=*"findUserById"* parameterType=*"int"*

resultType=*"cn.itcast.mybatis.po.User"* useCache=*"true"*>

SELECT \* FROM user WHERE id = #{id}

</select>

### 刷新二级缓存

该statement中设置**flushCache=true**可以刷新当前的二级缓存，**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**。

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。

如果查询语句设置成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：

<select id=*"findUserById"* parameterType=*"int"*

resultType=*"cn.itcast.mybatis.po.User"* useCache=*"true"* flushCache=*"true"*>

SELECT \* FROM user WHERE id = #{id}

</select>

### 整合ehcache（了解）

Ehcache是一个分布式缓存。

#### 分布式缓存

系统为了提高性能，通常会对系统采用分布式部署（集群部署方式）

系统工程

springmvc

mybatis

服务器1

系统工程

springmvc

mybatis

服务器2

对缓存数据进行集中管理（redis集群）

使用分布式缓存框架

redis、memcached、ehcache。

不使用分布式缓存，缓存的数据在各个服务单独存储，不方便开发。所以要使用分布式缓存对缓存数据进行集中式管理。

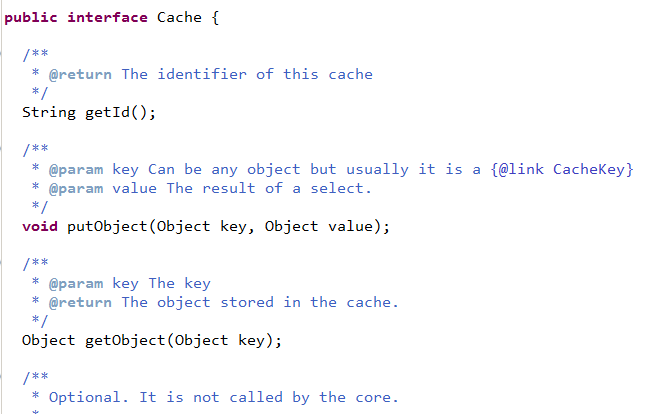
**Mybatis自身无法实现分布式缓存，需要和其它分布式缓存框架进行整合。**

#### 整合思路（重点）

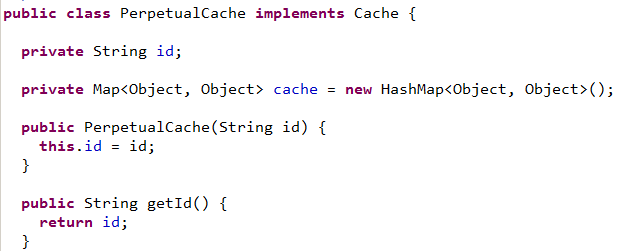
**Mybatis提供了一个cache接口，同时它自己有一个默认的实现类PerpetualCache。**

通过实现cache接口可以实现mybatis缓存数据通过其他缓存数据库整合，mybatis的特长是sql，缓存数据管理不是mybatis的特长，为了提高mybatis的性能，所以需要mybatis和第三方缓存数据库整合，比如ehcache、memcache、redis等

Mybatis提供接口如下：



Mybatis的默认实现类：



#### 整合ehcache的步骤

1. 引入ehcache的jar包；
2. 在mapper映射文件中，配置cache标签的type为ehcache对cache接口的实现类类型。
3. 加入ehcache的配置文件

#### 第一步：引入ehcache的jar包



#### 第二步：配置cache的type属性

<!-- 使用默认二级缓存 -->

<cache type=*"org.mybatis.caches.ehcache.EhcacheCache"* />

#### 第三步：添加ehcache的配置文件

在classpath下添加ehcache.xml

<ehcache xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:noNamespaceSchemaLocation=*"../config/ehcache.xsd"*>

<!-- 缓存数据要存放的磁盘地址 -->

<diskStore path=*"F:\develop\ehcache"* />

<!-- diskStore：指定数据在磁盘中的存储位置。  defaultCache：当借助CacheManager.add("demoCache")创建Cache时，EhCache便会采用<defalutCache/>指定的的管理策略

以下属性是必须的：  maxElementsInMemory - 在内存中缓存的element的最大数目  maxElementsOnDisk

- 在磁盘上缓存的element的最大数目，若是0表示无穷大  eternal - 设定缓存的elements是否永远不过期。如果为true，则缓存的数据始终有效，如果为false那么还要根据timeToIdleSeconds，timeToLiveSeconds判断

 overflowToDisk - 设定当内存缓存溢出的时候是否将过期的element缓存到磁盘上 以下属性是可选的：  timeToIdleSeconds

- 当缓存在EhCache中的数据前后两次访问的时间超过timeToIdleSeconds的属性取值时，这些数据便会删除，默认值是0,也就是可闲置时间无穷大

 timeToLiveSeconds - 缓存element的有效生命期，默认是0.,也就是element存活时间无穷大 diskSpoolBufferSizeMB

这个参数设置DiskStore(磁盘缓存)的缓存区大小.默认是30MB.每个Cache都应该有自己的一个缓冲区.  diskPersistent

- 在VM重启的时候是否启用磁盘保存EhCache中的数据，默认是false。  diskExpiryThreadIntervalSeconds

- 磁盘缓存的清理线程运行间隔，默认是120秒。每个120s，相应的线程会进行一次EhCache中数据的清理工作  memoryStoreEvictionPolicy

- 当内存缓存达到最大，有新的element加入的时候， 移除缓存中element的策略。默认是LRU（最近最少使用），可选的有LFU（最不常使用）和FIFO（先进先出） -->

<defaultCache maxElementsInMemory=*"1000"*

maxElementsOnDisk=*"10000000"* eternal=*"false"* overflowToDisk=*"false"*

timeToIdleSeconds=*"120"* timeToLiveSeconds=*"120"*

diskExpiryThreadIntervalSeconds=*"120"* memoryStoreEvictionPolicy=*"LRU"*>

</defaultCache>

</ehcache>

### 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：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flashInterval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与spring集成

## 集成思路

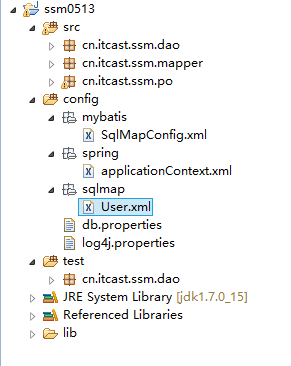
* 需要spring来管理数据源信息。
* 需要spring通过单例方式管理SqlSessionFactory。
* 使用SqlSessionFactory创建SqlSession。（spring和mybatis整合自动完成）
* 持久层的mapper都需要由spring进行管理，spring和mybatis整合生成mapper代理对象。

## 集成步骤

1. jar包集成；
2. 配置文件集成（数据源）；
3. SqlSessionFactory集成；
4. Mapper接口集成；

## 开始集成

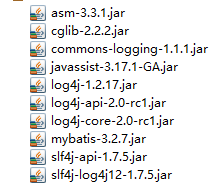
### 搭建工程结构



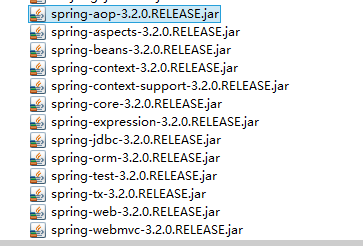
### Jar包集成

Jar包如下：

Mybatis3.2.7 的jar包（mybatis核心包、依赖包）



Spring3.2.0 的jar包



Spring与mybatis的集成包



数据库驱动包



Junit包



Dbcp连接池包



### 配置文件集成

注意：Mybatis的配置文件中的数据源配置去掉，由spring进行管理配置。

#### Mybatis的SqlMapConfig.xml

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<!DOCTYPE configuration

PUBLIC "-//mybatis.org//DTD Config 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-config.dtd">

<configuration>

<!-- 设置全局参数 -->

<settings>

<!-- lazyLoadingEnabled：延迟加载的开关，默认是false -->

<setting name=*"lazyLoadingEnabled"* value=*"true"*/>

<!-- aggressiveLazyLoading：默认为true，一旦为true上面的懒加载开关失效 -->

<setting name=*"aggressiveLazyLoading"* value=*"false"*/>

<!-- cacheEnabled：二级缓存的总开关 默认是false-->

<setting name=*"cacheEnabled"* value=*"true"*/>

</settings>

<!-- 定义别名 -->

<typeAliases>

<!-- 批量定义别名 -->

<!-- name：指定需要别名定义的包的名称 它的别名就是类名（类名的首字母大小写都可）-->

<package name=*"cn.itcast.ssm.po"*></package>

</typeAliases>

<!-- 注意：与spring集成后，数据源和事务交给spring来管理 -->

<!-- 加载mapper文件 -->

<mappers>

<mapper resource=*"mybatis/sqlmap/User.xml"*></mapper>

<!-- 批量加载mapper

注意：mapper接口文件和mapper映射文件，名称相同，在同一个包下

-->

<package name=*"cn.itcast.mybatis.mapper"*/>

</mappers>

</configuration>

#### Spring的applicationContext.xml

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns:mvc=*"http://www.springframework.org/schema/mvc"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:aop=*"http://www.springframework.org/schema/aop"* xmlns:tx=*"http://www.springframework.org/schema/tx"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-3.2.xsd*

*http://www.springframework.org/schema/mvc*

*http://www.springframework.org/schema/mvc/spring-mvc-3.2.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-3.2.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop-3.2.xsd*

*http://www.springframework.org/schema/tx*

*http://www.springframework.org/schema/tx/spring-tx-3.2.xsd "*>

<!-- 引用java配置文件 -->

<context:property-placeholder location=*"db.properties"*/>

<!-- 配置数据源，使用dbcp连接池 -->

<bean id=*"dataSource"* class=*"org.apache.commons.dbcp.BasicDataSource"*>

<property name=*"driverClassName"* value=*"${db.driver}"* />

<property name=*"url"* value=*"${db.url}"* />

<property name=*"username"* value=*"${db.username}"* />

<property name=*"password"* value=*"${db.password}"* />

<property name=*"maxActive"* value=*"10"* />

<property name=*"maxIdle"* value=*"5"* />

</bean>

</beans>

### Spring对SqlSessionFactory进行管理配置

<bean id=*"sqlSessionFactory"* class=*"org.mybatis.spring.SqlSessionFactoryBean"*>

<!-- mybatis的配置文件路径 -->

<property name=*"configLocation"* value=*"sqlMapConfig.xml"*></property>

<!-- SqlSessionFactory需要数据源信息，之前是写在sqlmapconfig.xml，现在需要重新指定 -->

<property name=*"dataSource"* ref=*"dataSource"*></property>

</bean>

### Mybatis程序编写

#### 原始dao方式

##### 编写dao接口

**public** **interface** UserDao {

// 1、 根据用户ID来查询用户信息；

**public** User findUserById(**int** id);

// 2、 根据用户名称来模糊查询用户信息列表；

**public** List<User> findUsersByName(String name);

// 3、 添加用户；

**public** **void** insertUser(User user);

}

##### 编写dao实现类（继承SqlSessionDaoSupport）

通过this.getSqlSession()获取sqlsession。

**public** **class** UserDaoImpl **extends** SqlSessionDaoSupport **implements** UserDao {

@Override

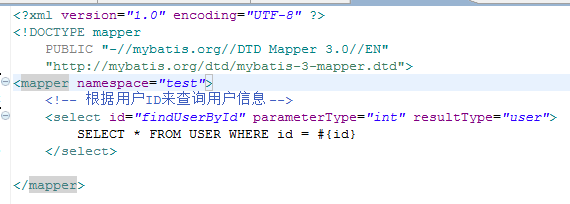
**public** User findUserById(**int** id) {

**return** **this**.getSqlSession().selectOne("test.findUserById", id);

}

}

##### 编写Mapper映射文件



##### Spring定义bean

<!-- 由spring管理原始dao的实现 -->

<bean id=*"userDao"* class=*"cn.itcast.mybatis.dao.UserDaoImpl"*>

<property name=*"sqlSessionFactory"* ref=*"sqlSessionFactory"*/>

</bean>

##### 编写测试代码

**public** **class** UserDaoTest {

//spring上下文

**private** ApplicationContext ctx;

@Before

**public** **void** setUp() **throws** Exception {

//读取spring的上下文，然后封装到ctx

ctx = **new** ClassPathXmlApplicationContext("spring/applicationContext.xml");

}

@Test

**public** **void** testFindUserById() {

//创建userdao对象

UserDao userDao = (UserDao) ctx.getBean("userDao");

//调用userdao对象的方法

User user = userDao.findUserById(1);

System.*out*.println(user);

}

}

#### Mapper代理方式

##### 编写mapper接口

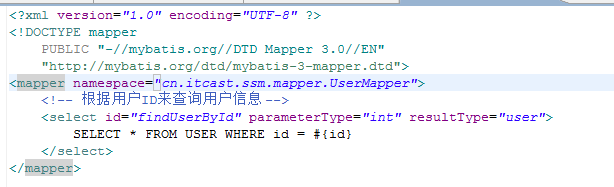
**public** **interface** UserMapper {

// 1、 根据用户ID来查询用户信息

**public** User findUserById(**int** id);

}

##### 编写mapper映射文件



##### Spring定义bean

Mapper代理开发方式有两种bean的定义方法，**一种是MapperFactoryBean，一种是MapperScannerConfigurer（推荐）。**

###### 通过MapperFactoryBean创建代理对象（了解）

<!-- mapper代理开发方式之单个mapper配置 -->

<bean id=*"userMapper"* class=*"org.mybatis.spring.mapper.MapperFactoryBean"*>

<property name=*"mapperInterface"* value=*"cn.itcast.mybatis.mapper.UserMapper"*></property>

<property name=*"sqlSessionFactory"* ref=*"sqlSessionFactory"*></property>

</bean>

###### 通过MapperScannerConfigurer批量扫描创建代理对象（掌握）

存在问题：一个mapper定义一个bean，很麻烦。

<!-- mapper代理开发方式之批量mapper配置 -->

<!-- bean的名字默认为mapper接口类名的首字母小写 -->

<bean class=*"org.mybatis.spring.mapper.MapperScannerConfigurer"*>

<!-- 指定批量mapper配置的包名 -->

<property name=*"basePackage"* value=*"cn.itcast.mybatis.mapper"*></property>

<!-- 指定使用的SqlSessionFactory -->

<property name=*"sqlSessionFactoryBeanName"* value=*"sqlSessionFactory"*></property>

</bean>

##### 编写测试代码

**private** ApplicationContext ctx;

@Before

**public** **void** setUp() **throws** Exception {

ctx = **new** ClassPathXmlApplicationContext(

"spring/applicationContext.xml");

}

@Test

**public** **void** testFindUserById() {

// 创建mapper对象

UserMapper userMapper = (UserMapper) ctx.getBean("userMapper");

// 调用mapper对象的方法

User user = userMapper.findUserById(1);

System.*out*.println(user);

}

# Mybatis的逆向工程（会用）

## 什么是逆向工程

**简单点说，就是通过数据库中的单表，自动生成java代码。**

Mybatis官方提供了逆向工程，可以针对单表自动生成mybatis代码（mapper.java\mapper.xml\po类）

企业开发中，逆向工程是个很常用的工具。

## 下载逆向工程

<https://github.com/mybatis/generator/releases/tag/mybatis-generator-1.3.2>

## 使用方法

1. 创建generator配置文件；
2. 使用java类来执行逆向工程；
3. 把生成的代码拷贝到项目中。
4. 在正式项目中使用逆向工程生成的代码

### 第一步：创建generator配置文件

在classpath下，创建generator.xml配置文件：

（文件内容可以从逆向工程的jar包中docs目录下的index.html中找到相关代码）

<?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-config\_1\_0.dtd">

<generatorConfiguration>

<context id=*"testTables"* targetRuntime=*"MyBatis3"*>

<commentGenerator>

<!-- 是否去除自动生成的注释 true：是 ： false:否 -->

<property name=*"suppressAllComments"* value=*"true"* />

</commentGenerator>

<!--数据库连接的信息：驱动类、连接地址、用户名、密码 -->

<jdbcConnection driverClass=*"com.mysql.jdbc.Driver"*

connectionURL=*"jdbc:mysql://localhost:3306/mybatis"* userId=*"root"*

password=*"mysql"*>

</jdbcConnection>

<!-- <jdbcConnection driverClass="oracle.jdbc.OracleDriver"

connectionURL="jdbc:oracle:thin:@127.0.0.1:1521:yycg"

userId="yycg"

password="yycg">

</jdbcConnection> -->

<!-- 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=*"cn.itcast.ssm.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=*"cn.itcast.ssm.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=*"cn.itcast.ssm.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 tableName=*"items"*></table>

<table tableName=*"orders"*></table>

<table tableName=*"orderdetail"*></table>

<table tableName=*"user"*></table>

</context>

</generatorConfiguration>

### 第二步：使用java类来执行逆向工程

**public** **class** Generator {

/\*\*

\* **@param** args

\*/

**public** **static** **void** main(String[] args) **throws** Exception{

List<String> warnings = **new** ArrayList<String>();

**boolean** overwrite = **true**;

File configFile = **new** File("config/generator.xml");

ConfigurationParser cp = **new** ConfigurationParser(warnings);

Configuration config = cp.parseConfiguration(configFile);

DefaultShellCallback callback = **new** DefaultShellCallback(overwrite);

MyBatisGenerator myBatisGenerator = **new** MyBatisGenerator(config,

callback, warnings);

myBatisGenerator.generate(**null**);

}

}

### 第三步：把生成的代码拷贝到项目中

如果正式项目中已经有po类所在的包了，那么就只需要拷贝po类到指定包下就可以。

如果正式项目中没有po包，那么就把逆向工程中整个po类的包拷贝过去。

Mapper.xml和mapper.java的拷贝与po类一样。

### 第四步：使用生成的代码

**public** **class** ItemsMapperTest {

// spring上下文

**private** ApplicationContext ctx;

@Before

**public** **void** setUp() **throws** Exception {

// 读取spring的上下文，然后封装到ctx

ctx = **new** ClassPathXmlApplicationContext(

"spring/applicationContext.xml");

}

@Test

**public** **void** testSelectByExample() {

ItemsMapper mapper = (ItemsMapper) ctx.getBean("itemsMapper");

ItemsExample example = **new** ItemsExample();

//使用它进行参数封装传递

Criteria criteria = example.createCriteria();

//设置参数

criteria.andNameEqualTo("背包");

List<Items> list = mapper.selectByExample(example);

System.*out*.println(list);

}

}

## 注意事项

Mapper.xml文件已经存在时，如果进行重新生成则mapper.xml文件时，内容不被覆盖而是进行内容追加，结果导致mybatis解析失败。

解决方法：**删除原来已经生成的mapper xml文件再进行生成。**

Mybatis自动生成的po及mapper.java文件不是内容而是直接覆盖没有此问题。