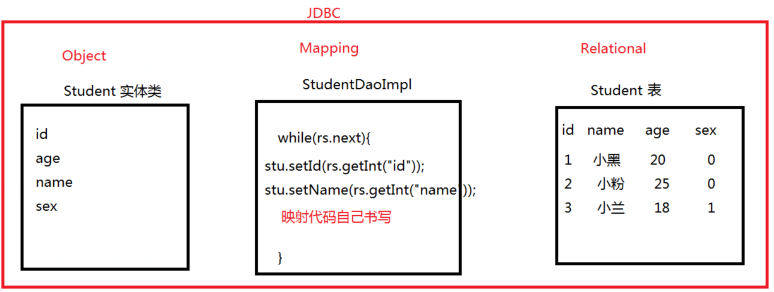
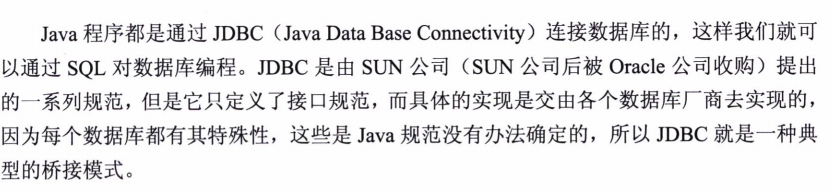
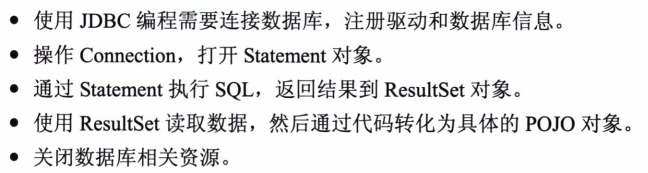
# 一:MyBatis简介

## 传统的JDBC编程







存在的弊端：

1. 工作量大，我们需要获取数据库四要素，连接，操作各种JDBC API，

之后还要正确的关闭这个对象;

1. 需要对JDBC编程产生的异常进行捕捉和处理
2. 如果项目业务复杂，使用JDBC也会非常复杂

针对于上述的问题，就引入了ORM框架！

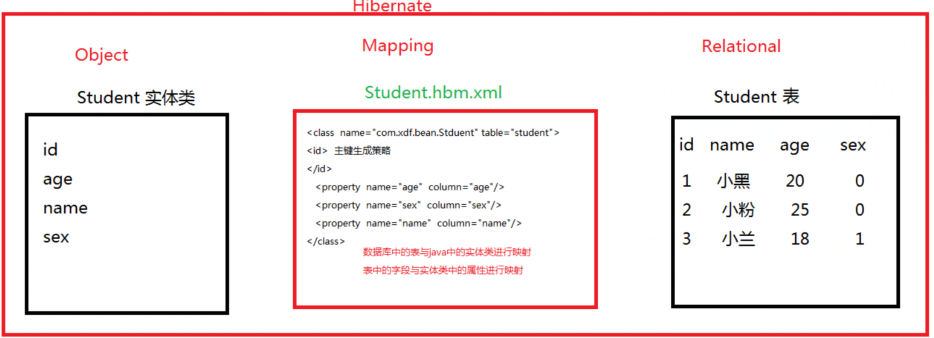
## ORM模型

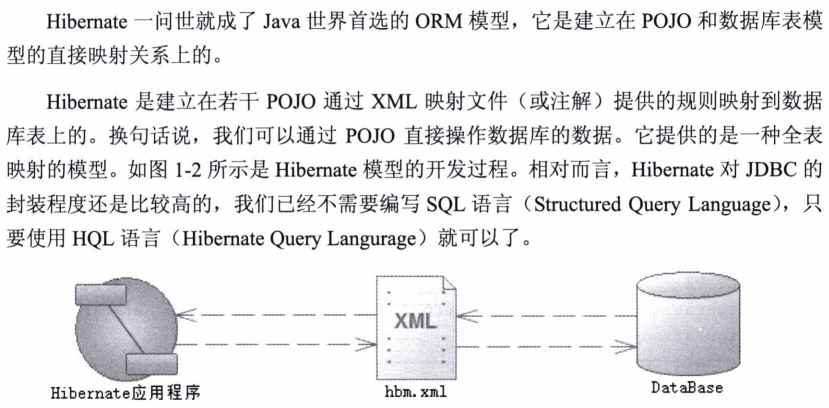
ORM（Object Relational Mapping）对象关系映射模型！

就是让数据库中的表和我们的JAVA中的对象（Plain Ordinary Java Object,简称POJO）产生关系！



## Hibernate





相比JDBC的优点：

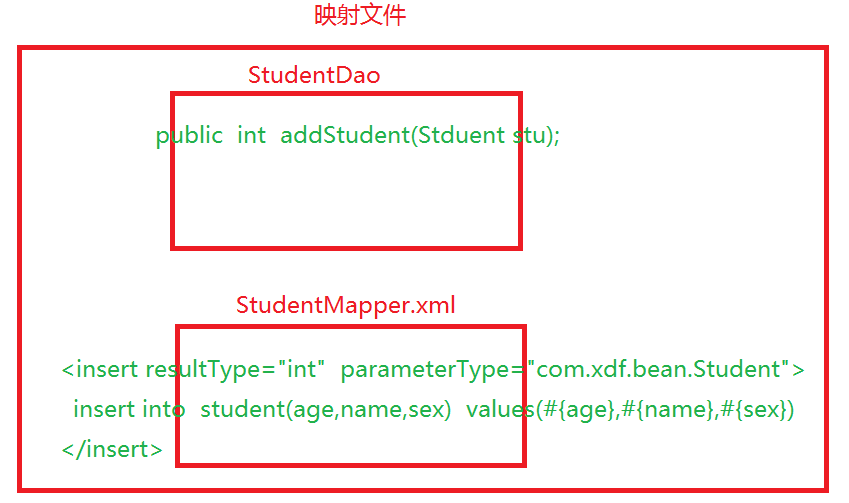
1. 不需要管理数据库的连接，配置在hibernate.cfg.xml文件中
2. 消除了代码的映射规则，也被配置在了xx.hbm.xml文件中
3. 释放资源，只需要关闭session即可

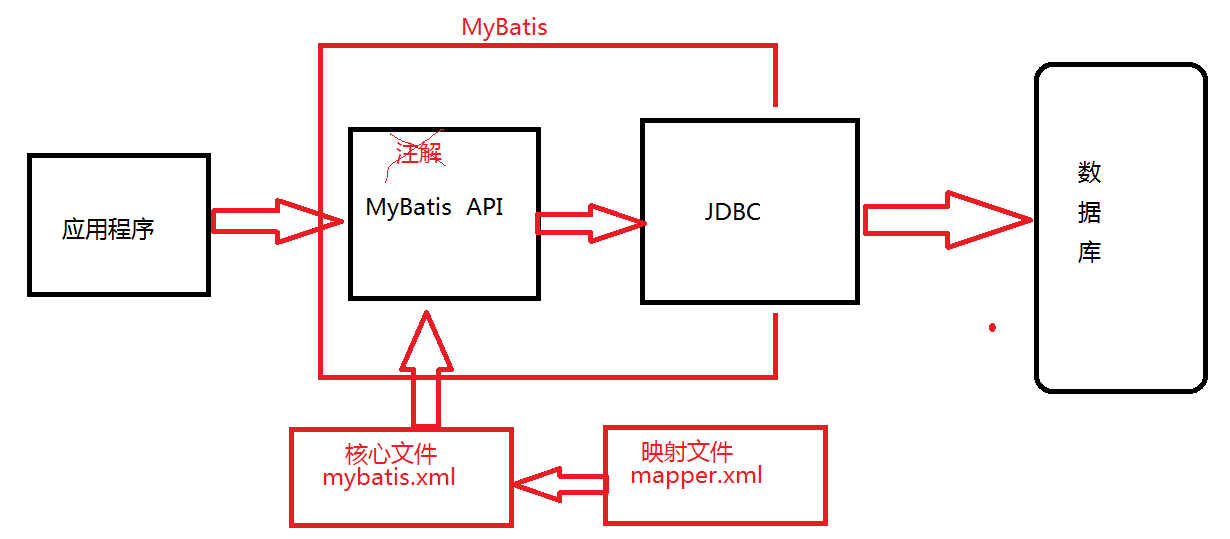
缺点：

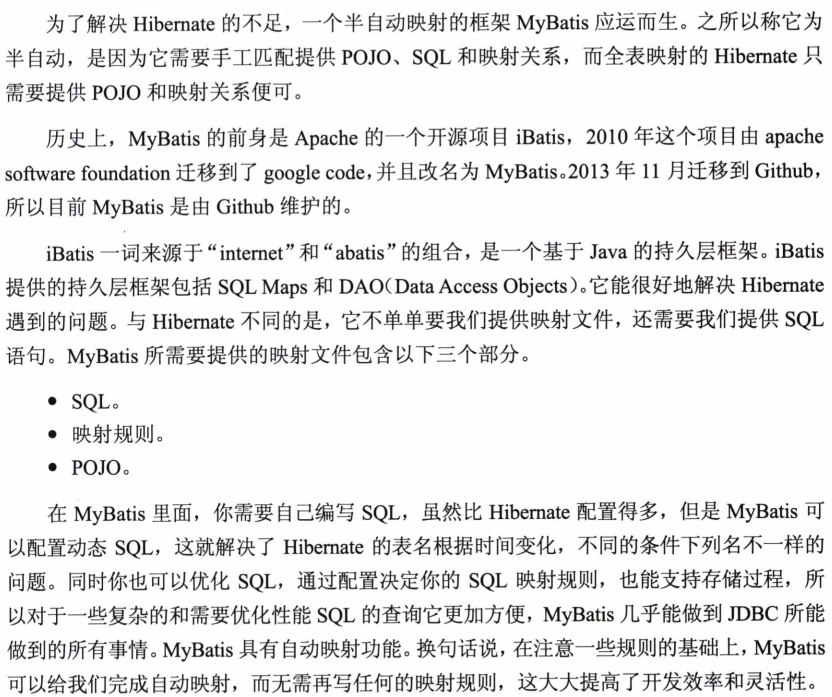
1. Hibernate屏蔽了sql语句，只能做全表映射，如果一张表很多字段，我们只关注2个，是没办法实现的；
2. 多表联查和复杂的sql语句支持较差；
3. 不支持存储过程
4. 不支持sql优化

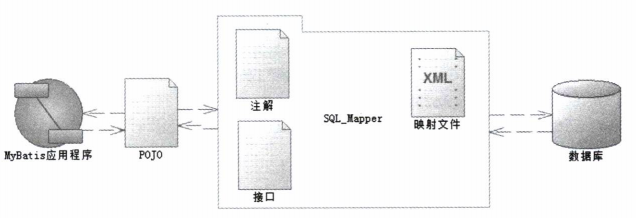
针对于Hibernate的这些问题，就产生了Mybatis框架来弥补Hibernate的这些不足！

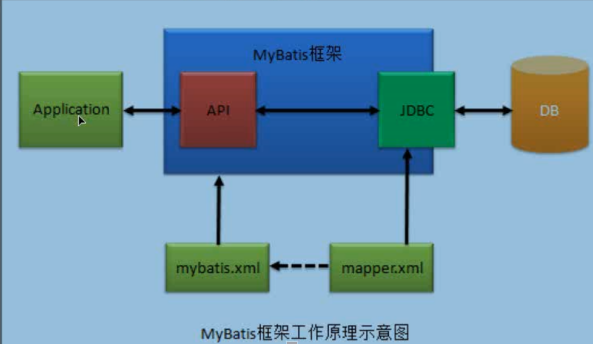
## MyBatis







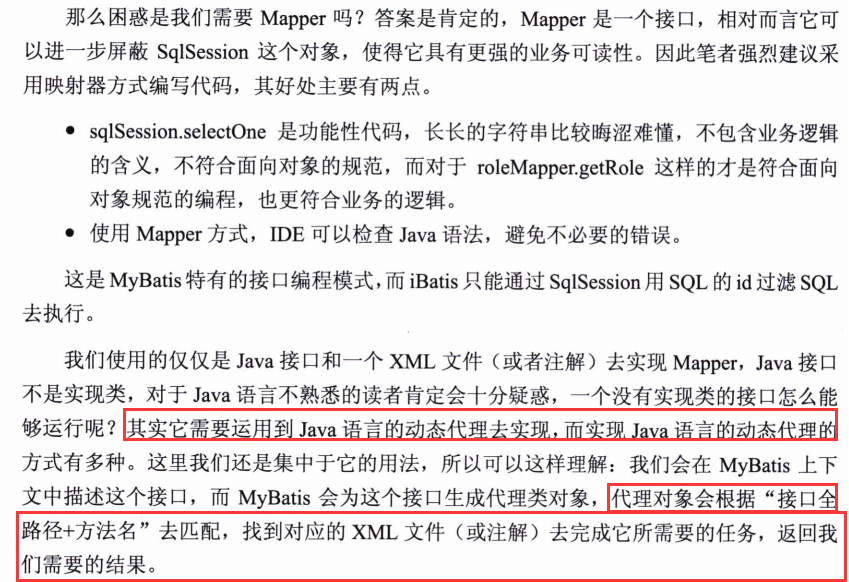




# 二:MyBatis入门

## MyBatis的环境搭建

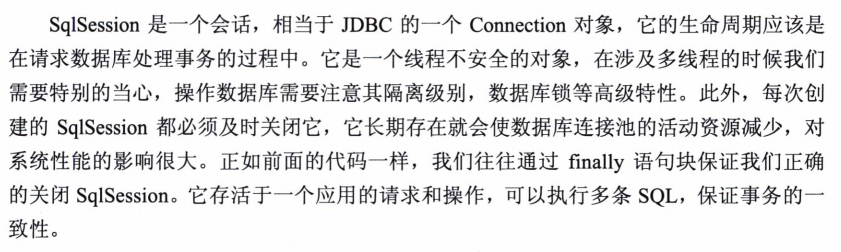
## MyBatis的基本构成



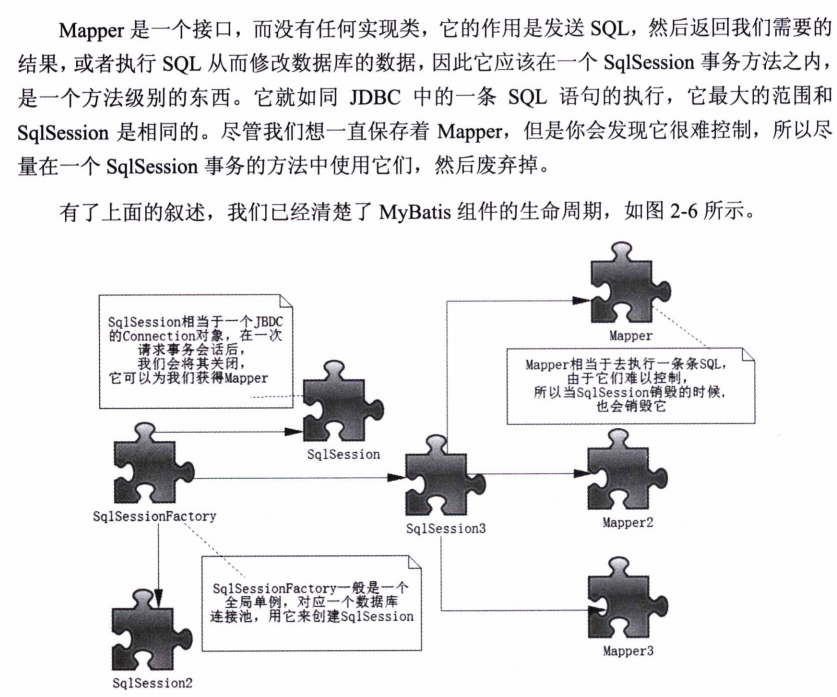
### 01.SqlSessionFactoryBuilder

### 02.SqlSessionFactory

### 03.SqlSession



### 04.Mapper



## MyBatis的配置文件



# 三:MyBatis简单的增删改查

## POM文件内容

<dependencies>  
 <dependency>  
 <groupId>junit</groupId>  
 <artifactId>junit</artifactId>  
 <version>4.1</version>  
 <scope>test</scope>  
 </dependency>  
  
 <!--父项目中引入log4j的配置-->  
 <dependency>  
 <groupId>log4j</groupId>  
 <artifactId>log4j</artifactId>  
 <version>1.2.17</version>  
 </dependency>  
  
 <!--引入mybatis需要的jar-->  
 <dependency>  
 <groupId>asm</groupId>  
 <artifactId>asm</artifactId>  
 <version>3.3.1</version>  
 </dependency>  
  
 <dependency>  
 <groupId>cglib</groupId>  
 <artifactId>cglib</artifactId>  
 <version>2.2.2</version>  
 </dependency>  
  
 <dependency>  
 <groupId>commons-logging</groupId>  
 <artifactId>commons-logging</artifactId>  
 <version>1.1.1</version>  
 </dependency>  
 <dependency>  
 <groupId>org.javassist</groupId>  
 <artifactId>javassist</artifactId>  
 <version>3.17.1-GA</version>  
 </dependency>  
  
 <dependency>  
 <groupId>org.slf4j</groupId>  
 <artifactId>slf4j-api</artifactId>  
 <version>1.7.5</version>  
 </dependency>  
  
 <dependency>  
 <groupId>org.slf4j</groupId>  
 <artifactId>slf4j-log4j12</artifactId>  
 <version>1.7.5</version>  
 <scope>test</scope>  
 </dependency>  
 <dependency>  
 <groupId>org.mybatis</groupId>  
 <artifactId>mybatis</artifactId>  
 <version>3.3.0</version>  
 </dependency>  
  
 <!--引入mysql数据库需要的jar-->  
 <dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <version>5.1.6</version>  
 </dependency>  
</dependencies>

<build> <!--加载所有src下xml文件-->  
 <resources>  
 <resource>  
 <directory>src/main/java</directory>  
 <includes>  
 <include>\*\*/\*.xml</include>  
 </includes>  
 </resource>  
 </resources>  
</build>

## mybatis.xml文件内容

<?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> <!--核心配置-->  
 <properties resource="jdbc.properties"/> <!-- 配置连接数据库的四要素-->  
 <typeAliases>  
 <package name="com.xdf.bean"/> <!--这个包下面的所有实体类的别名是 简写的类名-->  
 <!-- com.xdf.bean.User 简写User -->  
 </typeAliases>  
  
  
  
 <!--设置mybatis运行环境 default默认运行环境是environment节点中的id -->  
 <environments default="mysql">  
 <environment id="mysql">  
 <!--设置事务管理 后续ssm整合就交给Spring去管理-->  
 <transactionManager type="JDBC"></transactionManager>  
 <!--配置数据源 POOLED：mybatis自带的数据源 JNDI:tomcat中的数据源-->  
 <dataSource type="POOLED">  
 <property name="driver" value="${driver}"/>  
 <property name="url" value="${url}"/>  
 <property name="username" value="${username}"/>  
 <property name="password" value="${password}"/>  
 </dataSource>  
 </environment>  
 </environments>  
  
 <mappers> <!--管理我们设置的映射文件-->  
 <mapper resource="com/xdf/dao/UserMapper.xml"/>  
 </mappers>  
</configuration>

## log4j文件内容

log4j.appender.console=org.apache.log4j.ConsoleAppender  
log4j.appender.console.Target=System.out  
log4j.appender.console.layout=org.apache.log4j.PatternLayout  
log4j.appender.console.layout.ConversionPattern=[%-5p] %c %L %m %n  
  
log4j.rootLogger=trace,console

## Jdbc文件内容

driver=com.mysql.jdbc.Driver  
url=jdbc:mysql://localhost:3306/mybatis  
username=root  
password=

## UserMapper.java接口内容

package com.xdf.dao;  
  
import com.xdf.bean.User;  
  
import java.io.Serializable;  
import java.util.List;  
  
public interface UserMapper { //用户的映射接口  
 */\*\*  
 \* 新增用户  
 \*/* void addUser(User user);  
 */\*\*  
 \* 新增用户 但是可以拿到 新增用户的id 主键回填  
 \*/* void addUserById(User user);  
  
 */\*\*  
 \* 删除  
 \*/* void deleteUser(Serializable id);  
  
 */\*\*  
 \* 修改  
 \*/* void updateUser(User user);  
  
 */\*\*  
 \* 查询指定的User  
 \*/* User selectUserById(Serializable id);  
  
 */\*\*  
 \* 查询所有  
 \*/* List<User> selectAllUsers();  
  
 */\*\*  
 \* 根据名称进行模糊查询  
 \*/* List<User> selectByNamelike(String name);  
}

## UserMapper.xml内容

<?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务必和接口的全类名一致 -->  
<mapper namespace="com.xdf.dao.UserMapper">  
 <!--id务必和接口中的方法名称对应  
 如果参数类型是一个对象，那么sql语句中#｛对象的属性名｝  
 -->  
 <insert id="addUser" parameterType="User">  
 insert into user(userName,password)  
 values(#{userName},#{password})  
 </insert>  
  
 <!--新增用户的同时 拿到数据库中的id 主键回填  
 01.完成新增操作之后并没有把连接还给连接池  
 02.而是接着使用连接去查询id  
 SELECT @@IDENTITY  
 SELECT LAST\_INSERT\_ID() 都可以获取刚刚插入数据的主键  
  
 mysql中使用 order="AFTER"， 主键自增，必须是插入数据成功之后才能获取id  
 oracle中使用 order="BEFORE" ,必须先从序列中获取id，才能新增！  
  
 useGeneratedKeys: mybatis会根据数据库的不同获取主键===》 主键回填  
 -->  
 <insert id="addUserById" parameterType="User" keyProperty="id" useGeneratedKeys="true">  
 insert into user(userName,password) values(#{userName},#{password})  
 <!-- <selectKey resultType="int" keyProperty="id" order="AFTER">  
 select @@IDENTITY  
 </selectKey>-->  
 </insert>  
  
  
 <!--删除功能 用户传递的是 一个变量！ 这时候sql语句中的#{xxx}  
 xxx只是一个占位符  
 只有一个参数的时候，可以省略parameterType-->  
 <delete id="deleteUser" parameterType="int">  
 delete from user where id=#{xxx}  
 </delete>  
  
 <!--修改 用户传递的是 一个对象！这时候sql语句中的#{属性值}  
 属性值必须和实体类中的属性一致 -->  
 <update id="updateUser" parameterType="User">  
 update user set userName=#{userName},password=#{password}  
 where id=#{id}  
 </update>  
  
 <!--查询指定的user对象-->  
 <select id="selectUserById" parameterType="int" resultType="User">  
 SELECT id,userName,password from user where id=#{yyy}  
 </select>  
  
 <!--查询所有的用户信息-->  
 <select id="selectAllUsers" resultType="User">  
 SELECT id,userName,password from user  
 </select>  
  
 <!--根据名称进行模糊查询-->  
 <select id="selectByNamelike" resultType="User">  
 SELECT id,userName,password from user  
 WHERE userName like concat('%',#{zzz},'%')  
 </select>  
  
</mapper>

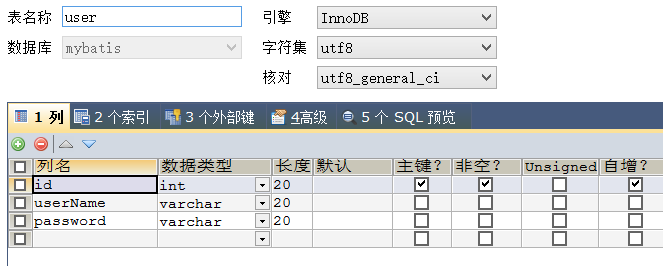
## UserMapperImpl.java内容

package com.xdf.dao;  
  
import com.xdf.bean.User;  
import com.xdf.util.SqlSessionFactoryUtil;  
import org.apache.ibatis.session.SqlSession;  
  
import java.io.Serializable;  
import java.util.ArrayList;  
import java.util.List;  
  
public class UserMapperImpl implements UserMapper{  
  
 //公共的session对象  
 //获取SqlSession  
 SqlSession session=null;  
  
 */\*\*  
 \** ***@param*** *user 需要新增的对象  
 \*  
 \* 通过观察insert的底层代码得知，所有的增删改其实都是执行的update  
 public int insert(String statement, Object parameter) {  
 return this.update(statement, parameter);  
 }  
 String statement：我们需要执行sql语句的id  
 Object parameter：sql语句需要的参数  
 \*/* public void addUser(User user) {  
  
 try {  
 //获取SqlSession  
 session= SqlSessionFactoryUtil.*getSession*();  
 session.insert("addUser",user); //新增用户  
 //提交事务  
 session.commit();  
 }catch (Exception e){  
 e.printStackTrace();  
 session.rollback(); //回滚  
 }finally {  
 session.close();  
 }  
 }  
  
 */\*\*  
 \* 新增用户并返回id  
 \*/* public void addUserById(User user) {  
  
 try {  
 //获取SqlSession  
 session= SqlSessionFactoryUtil.*getSession*();  
 session.insert("addUserById",user); //新增用户  
 */\*\*  
 \* 手动提交事务  
 \* DefaultSqlSession的中dirty=false; 不是脏数据  
 \* transaction.commit();  
 \*/* session.commit();  
 }catch (Exception e){  
 e.printStackTrace();  
 session.rollback(); //回滚  
 }finally {  
 session.close();  
 }  
 }  
  
 */\*\*  
 \*  
 \** ***@param*** *id 删除用户的id  
 \*/* public void deleteUser(Serializable id) {  
  
 try {  
 //获取SqlSession  
 session= SqlSessionFactoryUtil.*getSession*();  
 session.delete("deleteUser",id); //删除用户  
 session.commit();  
 }catch (Exception e){  
 e.printStackTrace();  
 session.rollback(); //回滚  
 }finally {  
 session.close();  
 }  
  
 }  
  
 */\*\*  
 \*修改用户  
 \*/* public void updateUser(User user) {  
 try {  
 //获取SqlSession  
 session= SqlSessionFactoryUtil.*getSession*();  
 session.update("updateUser",user); //修改用户  
 session.commit();  
 }catch (Exception e){  
 e.printStackTrace();  
 session.rollback(); //回滚  
 }finally {  
 session.close();  
 }  
 }  
  
 */\*\*  
 \* 查询指定的用户信息  
 \*/* public User selectUserById(Serializable id) {  
 User user=null;  
 try {  
 //获取SqlSession  
 session= SqlSessionFactoryUtil.*getSession*();  
 user= session.selectOne("selectUserById",id); //查询指定的用户信息  
 session.commit();  
 }catch (Exception e){  
 e.printStackTrace();  
 session.rollback(); //回滚  
 }finally {  
 session.close();  
 }  
 return user;  
 }  
  
 */\*\*  
 \* 查询所有用户信息  
 \** ***@return*** *\*/* public List<User> selectAllUsers() {  
 List<User> users=new ArrayList<User>();  
 try {  
 //获取SqlSession  
 session= SqlSessionFactoryUtil.*getSession*();  
 users= session.selectList("selectAllUsers"); //查询所有的用户信息  
 session.commit();  
 }catch (Exception e){  
 e.printStackTrace();  
 session.rollback(); //回滚  
 }finally {  
 session.close();  
 }  
 return users;  
  
 }  
  
 */\*\*  
 \*  
 \** ***@param*** *name 根据名称模糊查询  
 \** ***@return*** *\*/* public List<User> selectByNamelike(String name) {  
 List<User> users=new ArrayList<User>();  
 try {  
 //获取SqlSession  
 session= SqlSessionFactoryUtil.*getSession*();  
 users= session.selectList("selectByNamelike",name); //查询所有的用户信息  
 session.commit();  
 }catch (Exception e){  
 e.printStackTrace();  
 session.rollback(); //回滚  
 }finally {  
 session.close();  
 }  
 return users;  
 }  
}

## SqlSessionFactoryUtil.java内容

package com.xdf.util;  
  
import org.apache.ibatis.io.Resources;  
import org.apache.ibatis.session.SqlSession;  
import org.apache.ibatis.session.SqlSessionFactory;  
import org.apache.ibatis.session.SqlSessionFactoryBuilder;  
  
import java.io.IOException;  
import java.io.InputStream;  
  
*/\*\*  
 \* 单例模式创建SqlSessionFactory  
 \*/*public class SqlSessionFactoryUtil {  
 //01.创建需要单例的静态对象  
 private static SqlSessionFactory *sessionFactory*;  
  
 //02.私有化构造  
 private SqlSessionFactoryUtil(){}  
  
 //03.提供对外访问的接口  
 public static synchronized SqlSession getSession(){  
 try {  
 //把xml文件读取到内存中 并且返回一个输入流对象  
 InputStream stream = Resources.*getResourceAsStream*("mybatis.xml");  
 if (*sessionFactory*==null){  
 //创建SqlSessionFactory  
 *sessionFactory*= new SqlSessionFactoryBuilder().build(stream);  
 }  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 */\*\*  
 \* 为什么不需要关闭输入流  
 \* 01.build(stream) 使用了输入流  
 \* 02.ctrl+左键 点击build  
 \*  
 \* public SqlSessionFactory build(InputStream inputStream) {  
 return this.build((InputStream)inputStream, (String)null, (Properties)null);  
 }  
 03.继续点击build  
 04.inputStream.close(); 底层以及关闭了  
 \*/* return *sessionFactory*.openSession(); //返回一个 SqlSession的实现类===》DefaultSqlSession  
 }  
}

## User.java内容 以及数据库表结构

package com.xdf.bean;  
  
public class User { //用户实体类  
  
 private Integer id;  
 private String userName;  
 private String password;  
  
 @Override  
 public String toString() {  
 return "User{" +  
 "id=" + id +  
 ", userName='" + userName + '\'' +  
 ", password='" + password + '\'' +  
 '}';  
 }  
  
 public Integer getId() {  
 return id;  
 }  
  
 public void setId(Integer id) {  
 this.id = id;  
 }  
  
 public String getUserName() {  
 return userName;  
 }  
  
 public void setUserName(String userName) {  
 this.userName = userName;  
 }  
  
 public String getPassword() {  
 return password;  
 }  
  
 public void setPassword(String password) {  
 this.password = password;  
 }  
  
 public User(Integer id, String userName, String password) {  
  
 this.id = id;  
 this.userName = userName;  
 this.password = password;  
 }  
 public User(String userName, String password) {  
 this.userName = userName;  
 this.password = password;  
 }  
  
 public User() {  
  
 }  
}  


## 10.需要的sql脚本

DROP TABLE IF EXISTS `user`;

CREATE TABLE `user` (

`id` int(10) NOT NULL AUTO\_INCREMENT,

`userName` varchar(20) DEFAULT NULL,

`password` varchar(20) DEFAULT NULL,

PRIMARY KEY (`id`)

) ENGINE=InnoDB AUTO\_INCREMENT=15 DEFAULT CHARSET=utf8;

## 11.UserDemo测试代码

package com.xdf;  
  
import com.xdf.bean.User;  
import com.xdf.dao.UserMapper;  
import com.xdf.dao.UserMapperImpl;  
import org.junit.Before;  
import org.junit.Test;  
  
import java.util.List;  
  
public class UserDemo {  
  
 UserMapper mapper=null;  
  
 @Before  
 public void before(){  
 mapper=new UserMapperImpl();  
 }  
  
  
 */\*\*  
 \* 新增用户  
 \*/* @Test  
 public void addUser(){  
 User user=new User("小拜拜","admin");  
 mapper.addUser(user);  
 //没有id是因为数据库中已经设置了 主键自增 我们能不能获取新增的id  
 System.*out*.println(user.getId());  
 }  
  
 */\*\*  
 \* 我们需要新增用户之后，接着对此用户进行操作！  
 \* 就是需要获取用户的id  
 \*/* @Test  
 public void addUserById(){  
 User user=new User("小白白","admin");  
 mapper.addUserById(user);  
 //没有id是因为数据库中已经设置了 主键自增 我们能不能获取新增的id  
 System.*out*.println(user.getId());  
 }  
  
  
 */\*\*  
 \* 删除  
 \*/* @Test  
 public void deleteUser(){  
 mapper.deleteUser(10);  
 }  
 */\*\*  
 \* 修改  
 \*/* @Test  
 public void updateUser(){  
 User user=new User();  
 user.setId(9); //需要修改的id  
 user.setUserName("小黑黑");  
 user.setPassword("adminssss");  
 mapper.updateUser(user);  
 }  
  
 */\*\*  
 \* 查询指定的User  
 \*/* @Test  
 public void selectById(){  
 User user= mapper.selectUserById(9);  
 System.*out*.println(user);  
 }  
 */\*\*  
 \* 查询所有的User  
 \*/* @Test  
 public void selectAll(){  
 List<User> user= mapper.selectAllUsers();  
 System.*out*.println(user);  
 }  
  
 */\*\*  
 \* 模糊查询  
 \*/* @Test  
 public void selectLike(){  
 List<User> user= mapper.selectByNamelike("a");  
 System.*out*.println(user);  
 }  
}

# 四:MyBatis使用Mapper动态代理的增删改查

Mapper动态代理

之前我们使用JDBC操作数据库

01.StudentDao 增删改查的接口

02.StudentDaoImpl 真正实现增删改查的类 有sql语句

现在我们使用mybatis

01.studentDao 增删改查的接口

02.StudentDaoImpl 获取session和sql语句

03.StudentMapper.xml 有sql语句 真正实现增删改查

针对于上面我们的分析，mybatis框架自身就抛开了DaoImpl,

让我们可以直接定位到Mapper.xml文件中的sql语句！

我们就可以通过sql来操作DB！

这种实现方式 我们称之为Mapper动态代理！

怎么实现这种Mapper动态代理呢？

01.删除StudentDaoImpl

02.修改StudentMapper.xml文件中mapper节点的namespace

为StudentDao接口的完整限定名

03.把StudentMapper.xml文件中所有节点的id

改变成 StudentDao接口中的方法名

这样就可以让我们接口中的方法和mapper文件中的sql产生映射关系

04.在需要使用的时候

　之前用 session.insert(),update(),delete(),selectList().....

现在用 session.getMapper(StudentDao.class)

因为底层使用的是反射机制！我们在使用的时候,

务必要传递接口的class对象！这样在类加载的时候，

我们就可以得到StudentDao接口中所有的方法名称！

又因为mapper.xml文件中的节点id和方法同名！

所以我们在调用StudentDao接口中方法的时候，

底层会默认调用mapper.xml文件中的执行节点中的sql语句！

Mapper动态代理 \*\*\*\*\*

之前使用JDBC操作数据库的时候 使用Dao

01.UserDao 增删改查的接口

02.UserDaoImpl 增删改查接口的实现类 有sql语句

现在使用mybatis

UserMapper.java 增删改查的接口

UserMapper.xml 增删改查接口的实现类 有sql语句

UserMapperImpl.java 获取了 sqlSession 以及从xml文件中获取sql语句

Mapper动态代理的注意点：

01.xml文件中mapper节点的namespace属性值 必须和对应接口的 全类名一致

02.xml文件中有很多insert update delete select 节点

这些节点中id属性值 必须和接口中方法名称一致

## 删除UserMapperImpl实现类

## 在UserDemo中增加代码

UserMapper mapper=null; // User对应的接口  
SqlSession session=null;  
  
@Before  
public void before(){  
 //通过工具类获取session  
 session= SqlSessionFactoryUtil.*getSession*();  
 //使用mapper动态代理  
 mapper= session.getMapper(UserMapper.class);  
}  
  
@After  
public void after(){  
 session.commit();  
}

## 运行代码测试即可

# 五:#和$ 的区别

## 1. $ 不安全 底层是statement对象 ，可以sql注入

select \* from user whe

re id =${id}

如果我们传入的是 9，编译之后的sql语句是

select \* from user where id =9

# 安全 底层是PreparedStatement对象 ，可以防止sql注入

select \* from user where id =#{id}

如果我们传入的是 9，编译之后的sql语句是

select \* from user where id =?

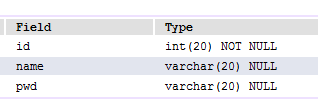
## 2.在sql语句中需要排序的时候使用$

order by ${id}

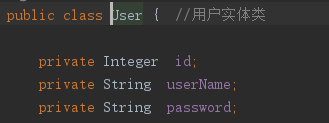
# 六:属性名和字段名不一致

案例：

查询User表中所有的数据



实体类的属性如下



## 使用别名

<select id="getAllUsers" resultType="User">  
 select id,`name` as userName,pwd as password from `user`  
 </select>-->

## 使用resultMap

第二种解决方式： 使用ResultMap按照我们自己设置的规则去映射

<select id="getAllUsers" resultMap="userMap">

select id,`name`,pwd from `user`

</select>

<!--可以按照我们自己的规则进行属性和字段的映射-->

<resultMap id="userMap" type="User">

<id property="id" column="id"/> <!--只针对于主键设置-->

<result property="userName" column="name"/>

<result property="password" column="pwd"/>

</resultMap>

<!--

类中可以存在的属性

int id; 主键

String name; 普通属性

Teacher teacher; 域属性association

List<Student> students; 集合属性collection

-->

如果我们类中存在域属性或者集合属性,或者需要延迟加载的时候 必须使用resultMap

# 七:多条件查询的问题

现在前台有个表单！

表单中有三项条件 来查询 符合条件的学生！

01.根据学生姓名查询

02.根据老师姓名查询

03.根据年级名称查询

针对于上述的情况，我们发现三个查询条件 不在同一个表中！

第一种情况：

把学生姓名，老师姓名，年级名称封装成一个Map集合

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

//把三个条件放进map

map.put("studentName",学生姓名);

map.put("teacherName",老师姓名);

map.put("gradeName",年级名称);

//调用dao层方法

dao.xxx(map);

然后去mapper.xml文件中执行sql语句

select s.name,s.id,s.age from student s,teacher t,grade g

where s.tId=t.id and s.gId=g.id

and s.name=#{studentName} //开始使用map中的key

and t.name=#{teacherName}

and g.name=#{gradeName}

第二种情况：

用户可以传递几个参数，我们在方法定义时就书写几个形参！

在接口中书写方法

List<Student> selectStduentsByCondition(String stuName,String teacherName,String gradeName);

在mapper.xml文件中书写sql

select s.name,s.id,s.age from student s,teacher t,grade g

where s.tId=t.id and s.gId=g.id

and s.name=#{0} //开始使用参数的下标

and t.name=#{1}

and g.name=#{2}

问题的引入：

如果前台传递的参数 不属于一个实体！！ 我们无法传递一个对象！

解决办法

## 1.使用Map集合

*/\*\*  
 \* 1.使用map  
 \* 模拟多条件查询  
 \* 查询姓名中包含a 并且id>10  
 \* 用户有可能只输入一个 姓名  
 \*/*List<User> findUserByParams(Map<String,Object> map);

*/\*\*1.使用map  
 \* 查询查询姓名中包含a 并且id>小白a 的数据  
 \*/*List<User> findUserByParams2(Map<String,Object> map);

## 2.使用索引

*/\*\*  
 \* 2.使用索引  
 \*/*List<User> findUserByParams3(String name,int id);

<!--使用索引 #{0} 指的是 参数在参数列表中的顺序 -->  
<select id="findUserByParams3" resultMap="userMap">  
 select id,`name`,pwd from `user`  
 where name like '%' #{0} '%'  
 and id >#{1}  
</select>

## 3.使用注解 @Param

*/\*\*  
 \* 3.使用Param注解  
 \*/*List<User> findUserByParams4(@Param("name") String name,@Param("id") int id);

<!--使用注解 #{name} 是Mapper接口中 @Param("name") -->  
<select id="findUserByParams4" resultMap="userMap">  
 select id,`name`,pwd from `user`  
 where name like '%' #{name} '%'  
 and id >#{id}  
</select>

# 八: #{} 中可以存放什么内容

## 1.只有一个单独的参数时候，#｛占位符｝

public void getUser(Integer id);

select \* from user where id=#{xx}

## 2.参数是一个对象的时候，#{必须是对象的属性名}

public void getUser(User user);

select \* from user where id=#{id}

## 3.有多个参数的时候，#{必须是参数对应的下标} 从0开始 也可以使用@Param注解

public void getUser(Integer id，String name);

select \* from user where id=#{0} and name=#{1}

## 4.参数是map集合的时候，传递的单独值，#{必须是map中的key}

public void getUser(Map<String,Object> map);

map.put("key",id);

select \* from user where id=#{key}

## 5.参数是map集合的时候，传递的对象，#{必须是map中的key.属性名}

public void getUser(Map<String,Object> map);

map.put("key",new User(1,"小黑"));

select \* from user where id=#{key.id}

# 九: 动态SQL

## If

### 1.1：UserMapper接口中的方法

*/\*\* if  
 \* 01.根据用户传递的参数 来确定执行的sql语句 用户传递一个对象  
 \*/*List<User> selectAllUsersByIf(User user);

### 1.2：userMapper.xml文件中的配置

<!--01.if的使用-->  
<select id="selectAllUsersByIf" resultType="User">  
 select id,userName,password from user where 1=1  
 <if test="userName!=null and userName!=''">  
 and userName like concat('%',#{userName},'%')  
 </if>  
 <if test="password!=null and password!=''">  
 and password=#{password}  
 </if>  
</select>

### 1.3：测试方法

@Test  
public void testIf(){  
 User user=new User();  
 //模拟用户给前台的输入框赋值  
 user.setUserName("a"); //模糊查询a  
 // user.setPassword("123");  
 List<User> users = mapper.selectAllUsersByIf(user);  
 System.*out*.println(users);  
}

## Where

### 2.1：UserMapper接口中的方法

*/\*\*  
 \* 02. where 标签替换上诉例子中的 where 1=1  
 \*/*List<User> selectAllUsersByWhere(User user);

### 2.2：userMapper.xml文件中的配置

<  
<!--02. where 替换 1=1-->  
 <select id="selectAllUsersByWhere" resultType="User">  
 select id,userName,password from user  
 <where>  
 <if test="userName!=null and userName!=''">  
 and userName like concat('%',#{userName},'%')  
 </if>  
 <if test="password!=null and password!=''">  
 and password=#{password}  
 </if>  
 </where>  
 </select>

### 2.3：测试方法

@Test  
public void testWhere(){  
 User user=new User();  
 //模拟用户给前台的输入框赋值  
 user.setUserName("a"); //模糊查询a  
 user.setPassword("123");  
 List<User> users = mapper.selectAllUsersByWhere(user);  
 System.*out*.println(users);  
}

## Choose

### 3.1：UserMapper接口中的方法

*/\*\* int a=50;  
 \* if(a>1) syso(1);  
 \* if(a>2) syso(2);  
 \* if(a>3) syso(3); 会输出3条数据  
 \*  
 \* switch(1){  
 \* case 1: syso(1); break;  
 \* case 2: syso(2); break;  
 \* case 3: syso(3); break;  
 \* default: syso(1111); break; 只执行一个  
 \* }  
 \* 如果多个条件都满足 默认值 只执行第一个满足的  
 \* 03. choose  
 \*/*List<User> selectAllUsersByChoose(User user);

### 3.2：UserMapper.xml文件中的配置

<!--03. choose-->  
<select id="selectAllUsersByChoose" resultType="User">  
 select id,userName,password from user  
 <where>  
 <choose>  
 <when test="userName!=null and userName!=''">  
 userName like concat('%',#{userName},'%')  
 </when>  
 <when test="password!=null and password!=''">  
 password=#{password}  
 </when>  
 <otherwise>  
 id=1  
 </otherwise>  
 </choose>  
 </where>  
</select>

### 3.3：测试方法

@@Test  
public void testChoose(){  
 User user=new User();  
 //模拟用户给前台的输入框赋值  
 // user.setUserName("a"); //模糊查询a  
 // user.setPassword("123");  
 List<User> users = mapper.selectAllUsersByChoose(user);  
 System.*out*.println(users);  
}

## Foreach 数组

### 4.1：UserMapper接口中的方法

*/\*\*  
 \* 04.foreach 遍历数组  
 \* select \* from user where id in(1,2,3) 不能这么写  
 \*/*List<User> selectAllUsersByForeachArray(int [] ids);

### 4.2：UserMapper.xml文件中的配置

<!--04.foreach 只要参数是 数组类型，mybatis默认的别名就是array -->  
<select id="selectAllUsersByForeachArray" resultType="User">  
 select id,userName,password from user  
 <if test="array.length>0">/\*证明数组有值\*/  
 where id in /\*(1,2,3)\*/  
 <foreach collection="array" item="myid" open="(" separator="," close=")">  
 #{myid}  
 </foreach>  
 </if>  
</select>

### 4.3：测试方法

@Test  
public void testForeachArray(){  
 int [] ids={2,3,4};  
 List<User> users = mapper.selectAllUsersByForeachArray(ids);  
 System.*out*.println(users);  
}

## Foreach List

### 5.1：UserMapper接口中的方法

*/\*\*  
 \* 05.foreach 遍历集合  
 \*/*List<User> selectAllUsersByForeachList(List<Integer> ids);

### 5.2：UserMapper.xml文件中的配置

<!--05.foreach list集合 只要参数是List类型，mybatis默认的别名就是list -->  
<select id="selectAllUsersByForeachList" resultType="User">  
 select id,userName,password from user  
 <if test="list.size>0">  
 where id in  
 <foreach collection="list" item="myid" open="(" separator="," close=")">  
 #{myid}  
 </foreach>  
 </if>  
</select>

### 5.3：测试方法

@Test  
public void testForeachList(){  
 List<Integer> list=new ArrayList<Integer>();  
 list.add(1);  
 list.add(3);  
 list.add(5);  
 List<User> users = mapper.selectAllUsersByForeachList(list);  
 System.*out*.println(users);  
}

## Foreach List<User>

### 6.1：UserMapper接口中的方法

*/\*\*  
 \* 06.foreach 遍历集合 集合中的元素是对象  
 \*/*List<User> selectAllUsersByForeachListUser(List<User> ids);

### 6.2：UserMapper.xml文件中的配置

<!--06.foreach list集合 元素是 对象 只要参数是List类型，mybatis默认的别名就是list -->  
<select id="selectAllUsersByForeachListUser" resultType="User">  
 select id,userName,password from user  
 <if test="list.size>0">  
 where id in  
 <foreach collection="list" item="user" open="(" separator="," close=")">  
 #{user.id} /\*一定是对象的属性名\*/  
 </foreach>  
 </if>  
</select>

### 6.3：测试方法

@Test  
public void testForeachListUser(){  
 List<User> list=new ArrayList<User>();  
 User user1=new User();  
 user1.setId(1);  
 User user2=new User();  
 user2.setId(2);  
 list.add(user1);  
 list.add(user2);  
 List<User> users = mapper.selectAllUsersByForeachListUser(list);  
 System.*out*.println(users);  
}

## Foreach Map

### 7.1：UserMapper接口中的方法

*/\*\*  
 \* 07.foreach 遍历Map集合 底层没有对map进行规定别名  
 \*/*List<User> selectAllUsersByForeachMap(@Param("map") Map<String,Object> map);

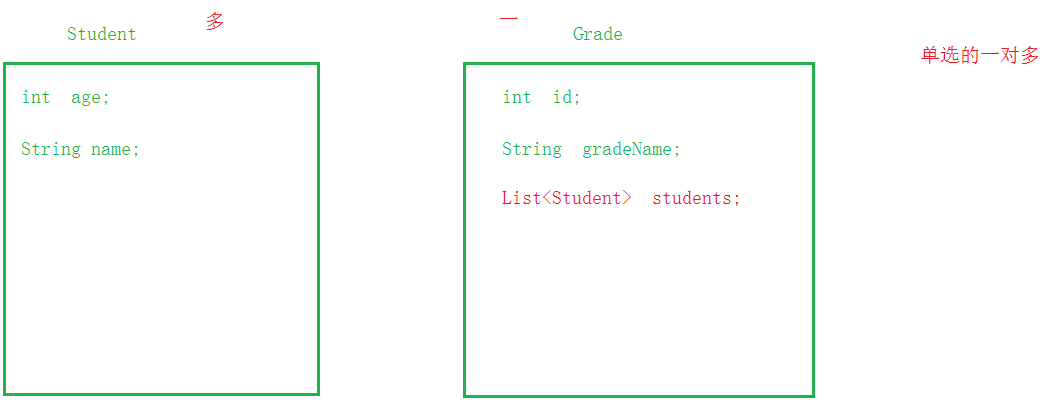
### 7.2：UserMapper.xml文件中的配置

<!--07.foreach Map集合 元素是 对象 只要参数是List类型，mybatis默认的别名就是list -->  
<select id="selectAllUsersByForeachMap" resultType="User">  
 select id,userName,password from user  
 <if test="map.keys.size>0">  
 where id in  
 <foreach collection="map.keys" item="mapkey" open="(" separator="," close=")">  
 #{mapkey}  
 </foreach>  
 </if>  
</select>

### 7.3：测试方法

@Test  
public void testForeachMap(){  
 Map<String, Object> map =new HashMap<String,Object>();  
 map.put("1","sas");  
 map.put("2","sas");  
 map.put("3","sas");  
 List<User> users = mapper.selectAllUsersByForeachMap(map);  
 System.*out*.println(users);  
}

# 十: 关联映射一对多



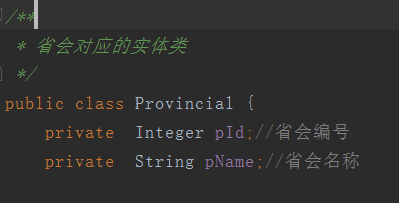
## 1.需要的sql创建表



## 2.Country实体类



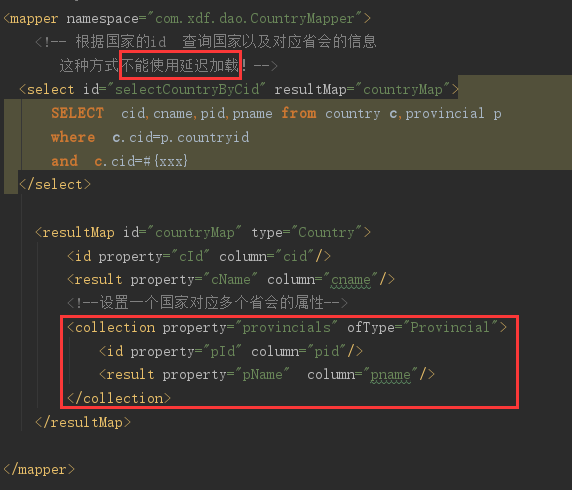
## Provincial实体类



## CountryMapper接口



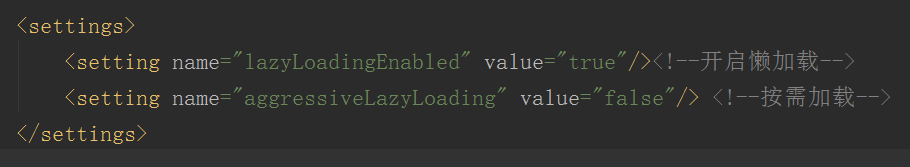
## CountryMapper.xml



## CountryMapper.xml 延迟加载



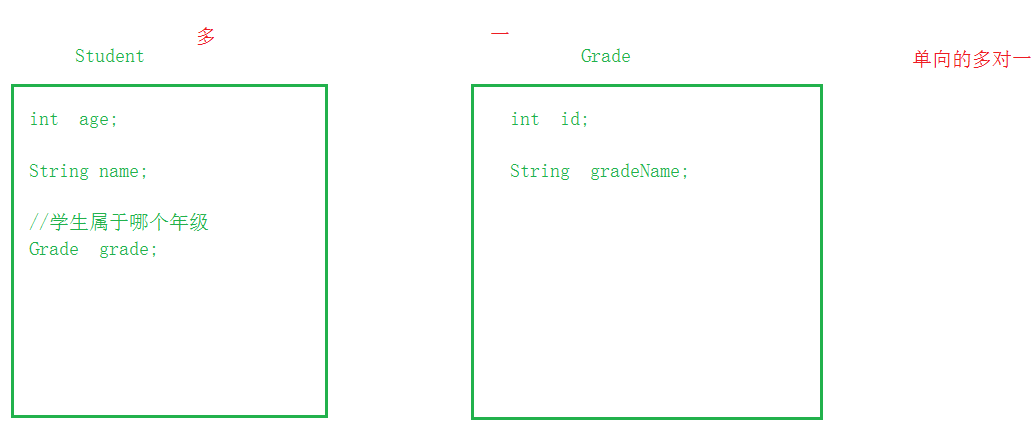
## mybatis.xml文件中的增加配置



## 测试类



# 十一: 关联映射多对一



## 1.需要修改的实体类

*/\*\*  
 \* 省会对应的实体类  
 \*/*public class Provincial {  
 private Integer pId;//省会编号  
 private String pName;//省会名称  
  
 //多个省会 对应一个国家 多对一  
 private Country country;

之前国家的那个省会集合属性就可以删除了

*/\*\*  
 \* 国家对应的实体类  
 \*/*public class Country {  
 private Integer cId;//国家编号  
 private String cName;//国家名称

## 需要的接口

public interface CountryMapper {  
 */\*\*  
 \* 根据省会的id 查询省会以及对应国家的信息  
 \*/* Provincial selectProvincialByCid(Integer id);  
}

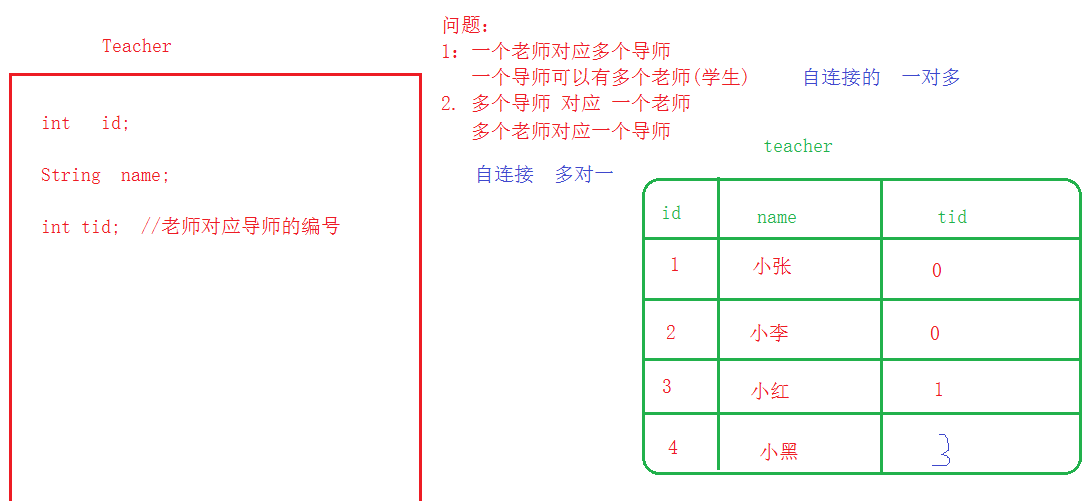
## 需要的xml文件

<!--namespace务必和接口的全类名一致 -->  
<mapper namespace="com.xdf.dao.CountryMapper">  
  
 <!--只查询 id对应的省会信息-->  
 <select id="selectProvincialByCid" resultMap="provincialMap">  
 SELECT pid,pname,countryid from provincial where pid=#{xxx}  
 </select>  
  
 <!--只查询 国家-->  
 <select id="selectCountryByCid" resultType="Country">  
 select cid,cname from country where cid=#{xxx}  
 </select>  
 <resultMap id="provincialMap" type="Provincial">  
 <id property="pId" column="pid"/>  
 <result property="pName" column="pname"/>  
 <!--域属性-->  
 <association property="country" javaType="Country" select="selectCountryByCid" column="countryid"/>  
 </resultMap>  
</mapper>

## 测试类

*/\*\*  
 \* 有延迟加载  
 \*/*@Test  
public void getCountryLazyById(){  
 Provincial provincial = mapper.selectProvincialByCid(1);  
 System.*out*.println(provincial.getpName()); //获取省会名称 只会执行1条sql语句  
 System.*out*.println(provincial.getCountry().getcName()); //获取对应的名称 执行2条sql语句  
  
}

# 十二: 自连接一对多



## 1.需要的sql素材



## Teacher实体类

public class Teacher { //老师和导师对应的实体类  
 private Integer id; //老师或者导师的编号  
 private String name;//老师或者导师的姓名  
 //一个导师可以有多个老师  
 private Set<Teacher> teachers;

## TeacherMapper接口

public interface TeacherMapper {  
  
 */\*\*  
 \* 根据导师的编号查询出 所有老师的信息  
 \*/* List<Teacher> selectTeachersByTid(Serializable id);  
  
 */\*\*  
 \* 根据老师的编号，查询自身信息和所有的导师信息  
 \*/* Teacher selectByTeacherId(Serializable id);  
}

## TeacherMapper.xml文件

<mapper namespace="com.xdf.dao.TeacherMapper">  
 <!--1.根据导师的编号查询出 所有老师的信息-->  
 <select id="selectTeachersByTid" resultMap="teacherMap">  
 select id,name from teacher where tid=#{xxx}  
 </select>  
 <resultMap id="teacherMap" type="Teacher">  
 <id property="id" column="id"/>  
 <result property="name" column="name"/>  
 <!--设置 集合属性 我们不断的把查询出来的id当成tid再次查询 ，直到没有结果  
 SELECT id,NAME FROM teacher WHERE tid=2  
 SELECT id,NAME FROM teacher WHERE tid=4  
 SELECT id,NAME FROM teacher WHERE tid=5  
 SELECT id,NAME FROM teacher WHERE tid=6-->  
 <collection property="teachers" ofType="Teacher" select="selectTeachersByTid"  
 column="id"/>  
 </resultMap>  
  
  
 <!--2.根据老师的编号，查询自身信息和所有的导师信息-->  
 <select id="selectByTeacherId" resultMap="teacherIdMap">  
 select id,name,tid from teacher where id=#{xxx}  
 </select>  
  
  
 <resultMap id="teacherIdMap" type="Teacher">  
 <id property="id" column="id"/>  
 <result property="name" column="name"/>  
 <!--设置 集合属性 我们不断的把查询出来的tid当成id再次查询 ，直到没有结果  
 SELECT id,NAME,tid FROM teacher WHERE id=6  
 SELECT id,NAME,tid FROM teacher WHERE id=5  
 SELECT id,NAME,tid FROM teacher WHERE id=4  
 SELECT id,NAME,tid FROM teacher WHERE id=2  
 SELECT id,NAME,tid FROM teacher WHERE id=0  
 -->  
 <collection property="teachers" ofType="Teacher" select="selectByTeacherId"  
 column="tid"/>  
 </resultMap>

## 测试类

*/\*\*  
 \* 根据导师的编号查询出 所有老师的信息 递归查询  
 \*/*@Test  
public void selectTeacherByTid(){  
 List<Teacher> teachers = mapper.selectTeachersByTid(2);  
 System.*out*.println(teachers);  
}  
  
*/\*\*  
 \* 根据导师的编号查询出 所有老师的信息 递归查询  
 \*/*@Test  
public void selectTeacherById(){  
 Teacher teacher = mapper.selectByTeacherId(6);  
 System.*out*.println(teacher);  
}

# 十三: 自连接多对一

## 1.Teacher实体类

public class Teacher { //老师和导师对应的实体类  
 private Integer id; //老师或者导师的编号  
 private String name;//老师或者导师的姓名  
 // 多个老师对应一个导师  
 private Teacher teacher;

## 2.接口

*/\*\*  
 \* 根据老师的编号，查询自身信息和所有的导师信息  
 \*/*Teacher selectByTeacherId(Serializable id);

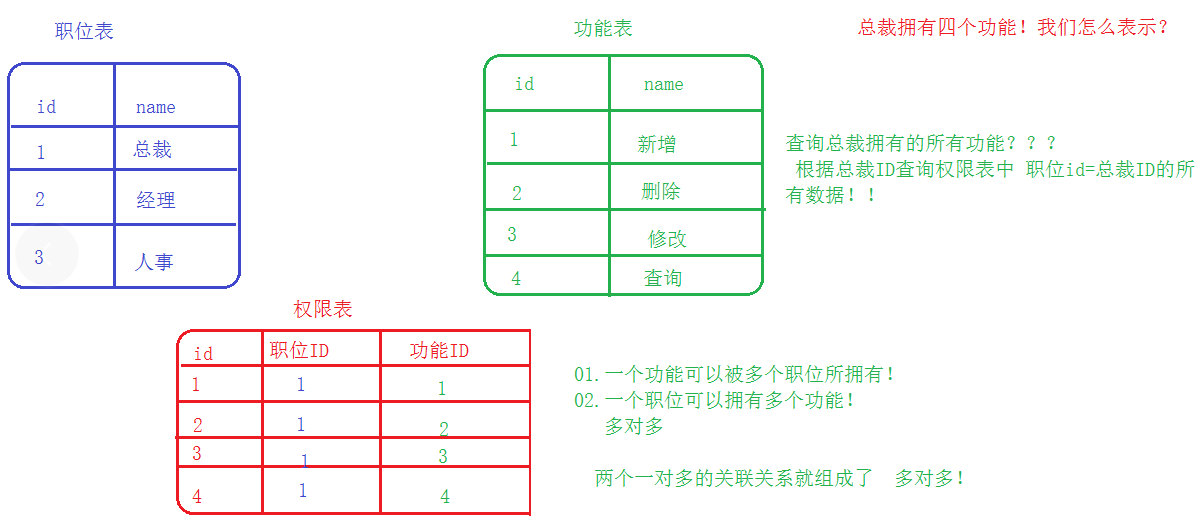
## 3.xml文件

<!--namespace务必和接口的全类名一致 -->  
<mapper namespace="com.xdf.dao.TeacherMapper">  
  
 <!--根据老师的编号，查询自身信息和所有的导师信息 -->  
 <select id="selectByTeacherId" resultMap="teacherMap">  
 select id,name,tid from teacher where id=#{xxx}  
 </select>  
  
 <resultMap id="teacherMap" type="Teacher">  
 <id property="id" column="id"/>  
 <result property="name" column="name"/>  
 <!--设置域属性-->  
 <association property="teacher" javaType="Teacher" select="selectByTeacherId" column="tid"/>  
 </resultMap>  
  
</mapper>

## 测试类

*/\*\*  
 \* 根据导师的编号查询出 所有老师的信息 递归查询  
 \*/*@Test  
public void selectTeacherById(){  
 Teacher teacher = mapper.selectByTeacherId(6);  
 System.*out*.println(teacher);  
}

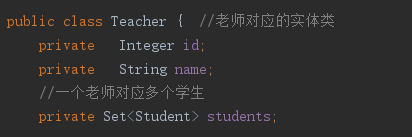
# 十四: 关联关系多对多

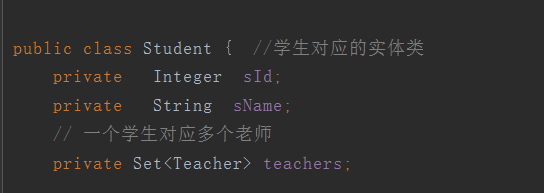


## 需要的sql素材

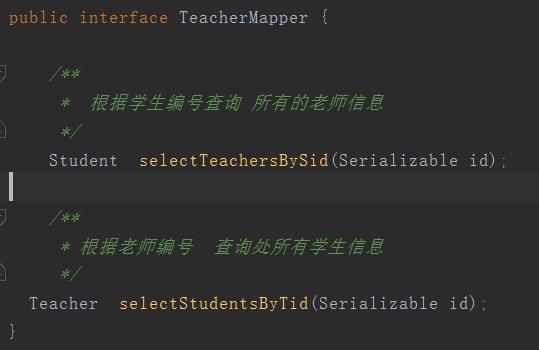


## 需要的实体类





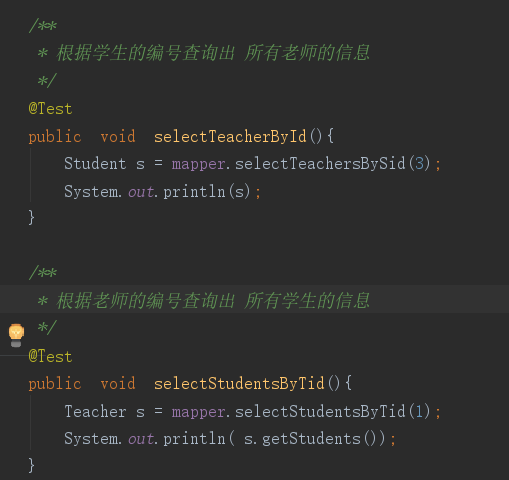
## 接口



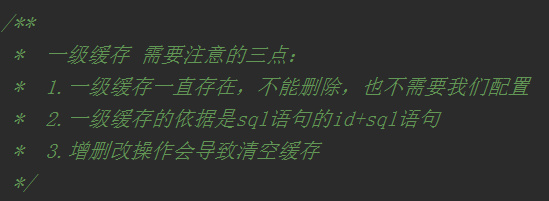
## Xml文件

<!--namespace务必和接口的全类名一致 -->  
<mapper namespace="com.xdf.dao.TeacherMapper">  
  
 <!--根据学生的编号，查询自身信息和所有的老师信息  
 # 查询 学号是3的学生 对应的 所有老师信息  
SELECT s.sid,t.id,sname,NAME FROM student s,teacher t,middle m  
WHERE s.`sid`=m.`sid` AND t.`id`=m.`tid`  
AND s.`sid`=3  
 -->  
 <select id="selectTeachersBySid" resultMap="studentMap">  
 SELECT s.sid,t.id,sname,NAME FROM student s,teacher t,middle m  
 WHERE s.`sid`=m.`sid` AND t.`id`=m.`tid`  
 AND s.`sid`=#{xxx}  
 </select>  
  
 <resultMap id="studentMap" type="Student">  
 <id property="sId" column="sid"/>  
 <result property="sName" column="sname"/>  
 <!--一个学生对应多个老师-->  
 <collection property="teachers" ofType="Teacher">  
 <id property="id" column="id"/>  
 <result property="name" column="name"/>  
 </collection>  
 </resultMap>  
  
  
  
 <!--根据老师编号 查询处所有学生信息-->  
 <select id="selectStudentsByTid" resultMap="teacherMap">  
 SELECT s.sid,t.id,sname,NAME FROM student s,teacher t,middle m  
 WHERE s.`sid`=m.`sid` AND t.`id`=m.`tid`  
 AND t.id=#{xxx}  
 </select>  
  
 <resultMap id="teacherMap" type="Teacher">  
 <id property="id" column="id"/>  
 <result property="name" column="name"/>  
 <!--一个学生对应多个老师-->  
 <collection property="students" ofType="Student">  
 <id property="sId" column="sid"/>  
 <result property="sName" column="sname"/>  
 </collection>  
 </resultMap>  
  
</mapper>

## 测试类



# 十五: Mybatis一级缓存



## 1：验证缓存的存在

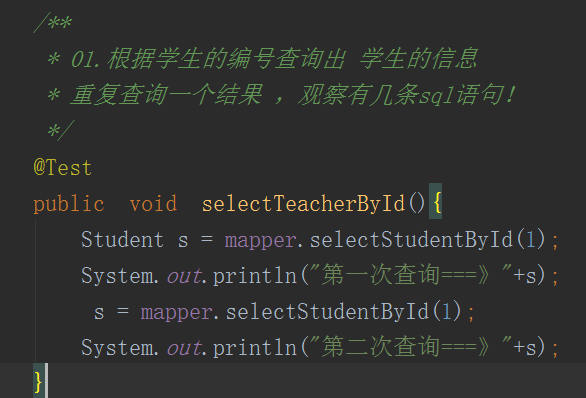
接口中的方法

*/\*\*  
 \* 根据学生编号查询 所有学生信息  
 \*/*Student selectStudentById(Serializable id);

Mapper.xml文件

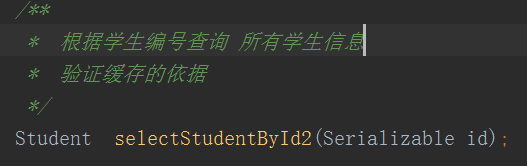
<!--根据学生编号查询 所有学生信息-->  
 <select id="selectStudentById" resultType="Student">  
 select sid,sname from student where sid=#{xxx}  
 </select>

测试类

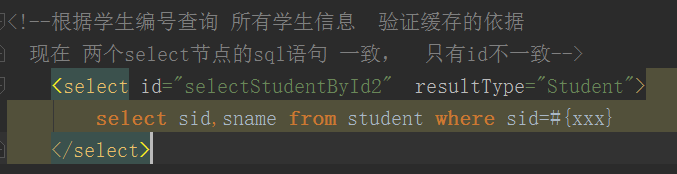


## 2：验证缓存的依据

接口的方法



Mapper.xml文件

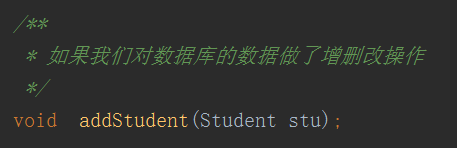


测试类

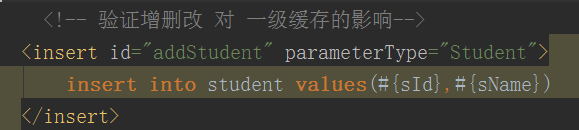


## 3：增删改对缓存的影响

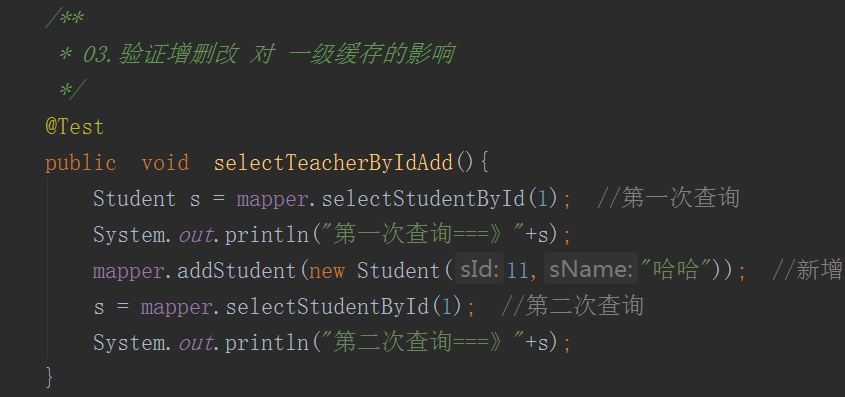
接口的方法



Mapper.xml文件



测试类



# 十六: Mybatis二级缓存

需要的xml文件所在的jar



## pom.xml文件配置

<!--引入需要的ehcache插件-->  
<dependency>  
 <groupId>net.sf.ehcache</groupId>  
 <artifactId>ehcache</artifactId>  
 <version>1.2.3</version>  
</dependency>  
<!--mybatis整合ehcache的jar-->  
<dependency>  
 <groupId>org.mybatis</groupId>  
 <artifactId>mybatis-ehcache</artifactId>  
 <version>1.0.0</version>  
</dependency>

## ehcache.xml文件配置

<ehcache>  
  
 <!--临时文件目录 可以自己设置-->  
 <diskStore path="java.io.tmpdir"/>  
  
 <!--  
 maxElementsInMemory：在内存中最大的存储量  
 eternal：是否在内存中永远不销毁  
 timeToIdleSeconds：在缓存中闲置多少时间之后销毁，默认单位是S  
 timeToLiveSeconds：在缓存中存活多少时间之后销毁，无论是否有人使用，默认单位是S  
 overflowToDisk：当缓存满的时候是否保存到磁盘中  
 diskPersistent：磁盘中的文件是否永久保存  
 diskExpiryThreadIntervalSeconds：检测线程运行时间间隔！默认单位是S  
  
  
 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 memoryStoreEvictionPolicy ： 缓存清理策略  
 1.LRU （least recently used） 最近最少使用  
 每个缓存的元素都有一个时间戳，当缓存容量满的时候，需要腾出来新位置给新缓存元素，  
 这时候，会比较时间戳然后删除符合条件的缓存元素  
  
 2.LFU （least frequently used ）最少使用  
 一直以来最少使用的缓存元素，缓存元素中有一个hit值，hit值最少的将会清除！  
  
 3.FIFO(first in first out) 先进先出  
 -->  
 <defaultCache  
 maxElementsInMemory="10000"  
 eternal="false"  
 timeToIdleSeconds="120"  
 timeToLiveSeconds="120"  
 overflowToDisk="true"  
 diskPersistent="false"  
 diskExpiryThreadIntervalSeconds="120"  
 memoryStoreEvictionPolicy="LRU"  
 />  
</ehcache>

## 接口的方法



## mapper.xml文件

<?xml version="1.0" encoding="UTF-8" ?>  
<!DOCTYPE mapper  
 PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  
 "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  
<!--namespace务必和接口的全类名一致 -->  
<mapper namespace="com.xdf.dao.StudentMapper">  
 <!--配置当前xml文件的2级缓存 下面的type使用一个即可  
 LoggingEhcache可以输出日志 -->  
 <cache type="org.mybatis.caches.ehcache.LoggingEhcache"/>  
 <!-- <cache type="org.mybatis.caches.ehcache.EhcacheCache"/>-->  
  
<!--根据学生编号查询 所有学生信息  
 01.局部关闭2级缓存 在对应的sql语句节点上 新增属性 useCache="false"  
 02.全局关闭2级缓存 在mybatis.xml文件中 新增  
 <setting name="cacheEnabled" value="false"/>-->  
 <select id="selectStudentById" resultType="Student">  
 select sid,sname from student where sid=#{xxx}  
 </select>  
  
 <!-- 验证增删改 对 二级缓存的影响 flushCache="false" 新增不刷新缓存-->  
 <insert id="addStudent" flushCache="false" parameterType="Student">  
 insert into student values(#{sId},#{sName})  
 </insert>  
</mapper>

## 测试文件

