链滴

Mybatis 从 0 到 1（基础进阶教程）

作者：sirwsl

原文链接：https://ld246.com/article/1596109434751

来源网站：链滴

又是实训SSM，有关MyBatis有些东西还是不是那么熟悉，今天就来进阶一下吧
oy：

引入依赖

在使用mybatis肯定是需要引入依赖的

```
<!-- https://mvnrepository.com/artifact/org.mybatis/mybatis -->
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis</artifactId>
            <version>3.4.6</version>
        </dependency>
```

一、mybatis-config.xml

mybatis-config.xml

1.首先引入固定的标签：

```
<?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">
```

2.引入根标签（接下来的所有内容都在根标签中写）

```
<configuration>
... ...
</configuration>
```

3.配置使用LOG4J输出日志（非必须）：

引入依赖：

```
<dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.17</version>
    </dependency>
```

在resource文件夹下添加lo4j.properties文件，内容如下

```
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
```

在xml中添加setting设置

```
<settings>
<setting name = "logImpl" value="LOG4J" />
</settings>
```

4.自定义控制台输出（非必须）：

```
<appender name="console" class="">
    <encoder>
      <pattern>[%thread] %d{HH:mm:ss.SSS} %-5level %logger{36} - %msg%n</pattern>
    </encoder>
</appender>
  <root level="debug">
    <appender-ref ref="console"/>
  </root>
```

5.别名定义：

<typeAlias type="" alias="" > 当个别名定义

<package name="" > 批量别名定义，包下所有类名作为别名，大小写不敏感

```
<typeAliases>
      <typeAlias type="com.wsl.entity.User" alias="user"/>
  <package name="com.cai.mybatis.pojo" />
</typeAliases>
```

6.配置db.properties (db.yml)数据源（可直接在xml中配置）：

```
<properties resource="db.yml"></preperties>
```

7.配置开发环境:

```
<environments default="dev">
  <environment id="dev">
    <transactionManager type="JDBC"></transactionManager>
```

```
        <dataSource type="POOLED">
            <property/>
        </dataSource>
    </environment>
        <environment id="test"></environment>
            ... ...
<environments>
```

<environment id="test" > </enviroment > 设置开发环境id唯一标识

<transactionManager type = "JDBC" > </transactionManager > 开启mybatis自带事务管理

<dataSource type = "POOLED" >

< property name = "dirver|url|username|password" value=""/> 配置数据库

< / dataSource>

8.配置mapping.xml文件映射

```
<mappers>
    <mapper resource="mapper/TestUserMapper.xml"/>
    <mapper class="com.wsl.mapper.UserMapper"/>
    <package name=""/>
</mappers>
```

<mapper resource = "" > 使用相对类路径的资源

< mapper class="">使用相对包路径资源

< package name = "">注册指定包下的所有mapper接口

end：汇总起来大致如下:

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

<configuration>
    <properties resource="jdbc.yml"></properties>
    <settings>
        <setting name="logImpl" value="LOG4J"/>
    </settings>
    <typeAliases>
        <typeAlias type="com.wsl.entity.User" alias="user"/>
        <package name="com.cai.mybatis.pojo" />
    </typeAliases>

    <environments default="dev">
        <environment id="dev">
            <transactionManager type="JDBC"></transactionManager>
            <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>
    <environment id="rel">
        <transactionManager type="JDBC"></transactionManager>
        <dataSource type="POOLED">
            <property name="driver" value="com.mysql.jdbc.Driver"/>
            <property name="url" value=""/>
            <property name="username" value="root"/>
            <property name="password" value=""/>
        </dataSource>
    </environment>
</environments>

<mappers>
    <mapper resource="mapper/TestUserMapper.xml"/>
    <mapper class="com.wsl.mapper.UserMapper"/>
    <package name=""/>
</mappers>

</configuration>
```

二、使用方式

xml形式

只使用xml文件会造成很大的不便,采用mapper与实体相关联，传参时候具有很大局限性

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/
ybatis-3-mapper.dtd">
<!-- 针对于数据库的sql语句(增删改查)操作，并且将sql语句的返回结果保存到相应类型中 -->
<!-- namespace是所有mapper.xml文件中的唯一标识，通常使用包名+类名 -->
<mapper namespace="com.wsl.entity.User">
    <!-- id表示在当前xml文件中的唯一标识 -->
    <insert id="insertUser">
        insert into user values(null,'test','321')
    </insert>
    <select id="selectUsers" resultType="com.wsl.entity.User">
        select  * from user
    </select>
    <update id="">... ...</update>
    <delete id="">... ...</delete>
</mapper>
```

@Test测试：

```
@Test
public void testInsert() {
    InputStream in = Test.class.getClassLoader().getResourceAsStream("mybatis-config.xml");
    //根据读取的配置文件创建工厂对象

    SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(in);
```

```
    //根据工厂对象获取sqlsession对象
    SqlSession sqlSession = sqlSessionFactory.openSession();
    //需要通过sqlSession对象来直接执行mapper.xml标签中的sql语句
    //参数为namespace+id，返回值表示本次插入影响的数据库行数
    int i = sqlSession.insert("com.wsl.entity.User.insertUser");
    //增删改都会影响数据库中的数据，最后都要手动提交
    sqlSession.commit();
    //将期望值和真实值做对比
    assertEquals(1, i);
}
```

接口+注解

使用@Update、@Insert、@select、@delect相关的注解+接口实现CUID

ps:在传参数时候，一个参数不用加@Parm注解，如果是多个参数需要加入@Parm注解

```
package com.wsl.mapper;

import com.wsl.entity.User;
import org.apache.ibatis.annotations.*;
import java.util.List;

public interface UserMapper {

    @Update("update user set name = #{name} where id = #{id}")
    int updataUser(@Param("name") String name,@Param("id") int id);

    @Insert("insert into user value (null,#{name},#{password})")
    int insertUser(@Param("name") String name, @Param("password") String password);

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

    @Delete("delete from user where id = #{id}")
    public User delectById(Integer id);

}
```

@Test测试

```
public class MybatisInterface {
    SqlSession sqlSession = getSqlSession();
    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

    SqlSession getSqlSession(){
        //读取mybatis-config配置文件，把配置文件转化为流
        InputStream in = Test.class.getClassLoader().getResourceAsStream("mybatis-config.xml")

        //根据读取的配置文件创建工厂对象
        SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(in);
        SqlSession session = sqlSessionFactory.openSession();
        return session;
    }
```

```
    @Test
    public void testInsert() {

        int num = userMapper.insertUser("123","123");
        sqlSession.commit();
        System.out.println(num);

    }
}
```

接口+xml

结合了xml与接口的特点，传参、解耦得到改善

xml文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/
ybatis-3-mapper.dtd">
<!-- 针对于数据库的sql语句(增删改查)操作，并且将sql语句的返回结果保存到相应类型中 -->
<!-- namespace是所有mapper.xml文件中的唯一标识，通常使用包名+类名 -->

<mapper namespace="com.wsl.mapper.TestUsermapper">
    <!--对key重新名命-->
    <resultMap id="testMap" type="com.wsl.entity.User">
        <id column="id" property="id" />
        <result column="username" property="name" />
    </resultMap>
    <!-- id表示在当前xml文件中的唯一标识 -->
    <insert id="insertUser" parameterType="int">
    <selectKey resultType="java.lang.Integer" keyProperty="id" order="AFTER" >
    insert into user values(null,'test','321')
    </insert>
    <select id="selectAllUsers" resultType="com.wsl.entity.User">
        select  * from user
    </select>

    <delete id=""></delete>
    <update id=""></update>
    <select id="selectUser" resultType="com.wsl.entity.User">
        select  * from user where id = #{id}
    </select>

</mapper>
```

mapper.java文件

```
public interface TestUsermapper {
    //添加学生
    int insertUser();
    //查询学生
    List<User> selectAllUsers();
    //查询学生
```

```
    User selectUser(Integer id);

}
```

@Test文件

```
public class TestUserMapper {
    SqlSession sqlSession = getSqlSession();
    TestUsermapper testUserMapper = sqlSession.getMapper( TestUsermapper.class);

    SqlSession getSqlSession(){
        //读取mybatis-config配置文件，把配置文件转化为流
        InputStream in = Test.class.getClassLoader().getResourceAsStream("mybatis-config.xml")

        //根据读取的配置文件创建工厂对象
        SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(in);
        SqlSession session = sqlSessionFactory.openSession();
        return session;
    }

    @Test
    public void TestSelectAll(){
        List<User> users  = testUserMapper.selectAllUsers();
        sqlSession.commit();
        for (User user:users) System.out.println(user);
    }
}
```

属性说明：

（随便写几个，找了几个，剩下的可能在下面会用到）

- **id** ： 用于设置主键字段与领域模型属性的映射关系
- **parameterType:** 输入参数，在配置的时候，配置相应的输入参数类型即可
- **resultType** 结果的类型。MyBatis 通常可以算出来,但是写上也没有问题。MyBatis 允许任何简单型用作主键的类型,包括字符串。
- **resultsMap**: 使用一个嵌套的结果映射来处理通过join查询结果集，映射成Java实体类型
- **namespace:** 是所有mapper.xml文件中的唯一标识，通常使用包名+类名
- **order:** 这可以被设置为 BEFORE 或 AFTER。如果设置为 BEFORE,那么它会首先选择主键,设置 keyProperty 然后执行插入语句。如果设置为 AFTER,那么先执行插入语句,然后是 selectKey 元素-这和如 Oracle 数据库相似,可以在插入语句中嵌入序列调用。
- **keyProperty:** selectKey 语句结果应该被设置的目标属性。
- statementType**:** 和前面的相 同,MyBatis 支持 STATEMENT ,PREPARED 和CALLABLE 语句的映射型,分别代表 PreparedStatement 和CallableStatement 类型。
- **useGeneratedKeys**：true为启用设置是否使用JDBC的getGenereatedKeys方法获取主键并赋值keyProperty设置的领域模型属性中。
- **keyProperty**：javabean中的属性名称。
- **keyColumn**：对应数据库的字段名称。
- **collection**:传入集合或数组的名称。

- **index**:遍历数组或集合的索引一般就写index。
- **item**:取出每一个元素的名称。
- **separator**:上一个元素与下一个元素之间填充的字符，如不需要则不添加此属性。
- **open**：表示在每个元素前填充字符。
- **close**:表示在每个元素后填充字符。

$与#的区别

默认情况下,使用#{}格式的语法会导致 MyBatis 创建预处理语句属性并安全地设置值（比如?）。这做更安全，更迅速，通常也是首选做法，不过有时你只是想直接在 SQL 语句中插入一个不改变的字串。当使用${} MyBatis 不会修改或转义字符串。以这种方式接受从用户输出的内容并提供给语句中变的字符串是不安全的，会导致潜在的 SQL 注入攻击，因此要么不允许用户输入这些字段，要么自转义并检验。

简单点说就是：#会进行一个预编译检验，但是$是直接赋值，不安全存在sql注入

三、动态sql

if

<if test="" > test后为选择条件为真执行

```
<select id="selectAllUsers" resultType="com.wsl.entity.User">
   select  * from user
   <if test="id != null"> id = #{id}</if>
</select>
```

foreach

遍历传入的数组、list、map。

xml：

```
<select id="selectUser" resultType="com.wsl.entity.User">
   select  * from user where name in
   <foreach collection="name" open="(" close=")" index="index" item="item" separator=",">
      #{item}
   </foreach>
</select>
```

interface:

```
List<User> selectUser(@Param("name") String[] name);
```

Test:

```
public class TestUserMapper {
   SqlSession sqlSession = getSqlSession();
   TestUsermapper testUserMapper = sqlSession.getMapper( TestUsermapper.class);
```

```
    SqlSession getSqlSession(){
        //读取mybatis-config配置文件，把配置文件转化为流
        InputStream in = Test.class.getClassLoader().getResourceAsStream("mybatis-config.xml")

        //根据读取的配置文件创建工厂对象
        SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(in);
        SqlSession session = sqlSessionFactory.openSession();
        return session;
    }
@Test
    public void TestSelect(){
        String[] array = {"wsl","test"};
        List<User> user  = testUserMapper.selectUser(array);
        sqlSession.commit();
        System.out.println(user);
    }
}
```

参数说明：

- collection：collection 属性的值有三个分别是 list、array、map 三种
- item ： 表示在迭代过程中每一个元素的别名
- index ： 表示在迭代过程中每次迭代到的位置（下标）
- open ： 前缀
- close ： 后缀
- separator ： 分隔符，每个元素之间以什么分隔

choose-when

当我们在传入值得时候进行选择判断，类似java中switch...case...default...。如果其中一个when标签足则执行所在<when > 标签的sql如果都不满足则执行<otherwise > 中的内容。

这里需要与<if > 标签做区别，<if > 中的依次判断，如果满足就不断拼接，但是<choose > 标签中只选择一个<when > ，如果遇到满足（为ture）的时候，就再执行下面的标签。

```
<select id="selectUser" resultType="com.wsl.entity.User">
    select  * from user where
    <choose>
        <when test="name != null and name !=''">name = #{name}</when>
        <when test="id != null ">id = #{id}</when>
        <otherwise>1=1</otherwise>
    </choose>
</select>
```

where

自动添加where关键字、剔除and、or关键字

```
<select>SELECT * from user where
        <if test="name!=null and name!='' ">
            name =#{name}
```

```
    </if>
        <if test="password!= null and password!= '' ">
            AND password = #{password}
        </if>
</select>
```

上面例子中，如果第一个if不满足，但是第二个if满足的时候就会出现 ...where and ...错误sql的情况为了避免这种情况采用<where>标签自动判断是否添加where关键字，并且自动剔除and、or关键。

```
<select id="selectAllUsers" resultType="com.wsl.entity.User">
    select  * from user
    <where>
        <if test="password != null and password != '' != null"> password = #{password}</if>
        <if test="name != null and name !=''">and name = #{name}</if>
    </where>
</select>
```

set

当我们在执行update语句修改多字段的时候，最后一个if不执行将会导致sql出错（多了个,）

```
<select id="selectAllUsers" resultType="com.wsl.entity.User">
    update user set
    <if test="name != null and name !=''">name = #{name},</if>
    <if test="password != null and password != '' != null"> password = #{password}</if>
    where id = #{id}
</select>
```

当passowrd字段为null时候将会导致错误出现，为了避免这种情况，采用<set>标签自动剔除末尾号。

```
<select id="selectAllUsers" resultType="com.wsl.entity.User">
    update user set
    <set>
        <if test="name != null and name !=''">name = #{name},</if>
        <if test="password != null and password != '' != null"> password = #{password}</if>
    </set>
    where id = #{id}
</select>
```

trim

格式化标签，主要用于拼接sql语句（前后缀等）

trim属性：

- prefix：在trim标签内sql语句加上前缀
- suffix：在trim标签内sql语句加上后缀
- prefixOverrides：指定去除多余的前缀内容，如：prefixOverrides=“AND | OR”，去除trim标内sql语句多余的前缀"and"或者"or"
- suffixOverrides：指定去除多余的后缀内容

```
<select id="selectUser" resultType="com.wsl.entity.User">
   select  * from user where
   <trim prefixOverrides="and|or">
      <if test="name != null and name !=''"> and name = #{name}</if>
      <if test="id != null ">and id = #{id}</if>
   </trim>
</select>
```

sql-include

当多种类型的查询语句的查询字段或者查询条件相同时，可以将其定义为常量，方便调用。为求 <select> 结构清晰也可将 sql 语句分解。通过<include>标签包含其中

```
<sql id="userInfo">
   select name,password from user
</sql>
<select id="selectAllUsers" resultType="com.wsl.entity.User">
   <include refid="userInfo"></include>
   where id = #{id}
</select>
```

bind

bind做字符串拼接，方便后续使用。

xml:

```
<select id="selectUser" resultType="com.wsl.entity.User">
   <bind name="name" value="'%'+name+'%'"/>
   select  * from user where name like #{name}
</select>
```

interface:

```
List<User> selectUser( @Param("name") String name);#必须使用@Parm注解
```

四、关联关系映射

association与 collection

association：通常用来映射多对一、一对一的关系。eg：一个学生只能属于一个班

collection：通常用来映射一对多的关系。eg：一个班里有多个学生

- property: 对象集合属性名称
- javaType: 该对象属性的类
- ofType: 集合的泛型
- coumn: select属性查询字段

案例

查询一个班的基本信息与在这个班的学生基本信息（association用法类似，更容易理解，此处不举例）

多表单独查询xml:

```
<select id="testselect" resultMap="testMap" >
        select * from sc where classid= #{id}
    </select>
    <resultMap id="testMap" type="com.wsl.entity.sc">
        <id property="id" column="id"/>
        <result property="classid" column="classid"/>
        <result property="userid" column="userid"/>
        <collection property="users" ofType="com.wsl.entity.User" select="test" column="useri
"/>
    </resultMap>
    <select id="test" resultType="com.wsl.entity.User">
        select * from user where id = #{id}
    </select>
```

多表连接查询xml

```
<select id="testselect" resultMap="testMap" >
        select * from sc,student where where sc.userid=user.id and classid= #{id}
    </select>
    <resultMap id="testMap" type="com.wsl.entity.sc">
        <id property="id" column="id"/>
        <result property="classid" column="classid"/>
        <result property="userid" column="userid"/>
        <collection property="users" ofType="com.wsl.entity.User">
            <id property="id" column="id"/>
            <result property="name" column="name"/>
            <result property="password" column="password"/>
        </collection>
    </resultMap>
```

sc:

```
@Data
public class sc {
    Integer id;
    Integer userid;
    Integer classid;
    User[] users;
}
```

User:

```
@Data
public class User {
    private Integer id;
    private String name;
    private String password;
}
```

五、mybatis逆向工程：

mybatis官方提供了Mapper自动生成工具，可以通过配置generator.xml文件，根据数据库表自动成pojo类和mapper映射文件，接口文件。

PS：

1. 逆向工程生成的代码只能做单表查询

2. 不能在生成的代码上进行扩展，因为如果数据库变更，需要重新使用逆向工程生成代码，原来编写代码就被覆盖了。

逆向工程文件包：mybatisgeneratorcore1.3.2.zip

```
<?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>

<!--修改成自己本机中mysql驱动包jar包路径-->
<!--此插件需要连接到数据库，读取数据库中内容帮助开发者生成实体类，所以我们需要为其制定一个dbc的jar。他通过此jar包读取数据库 -->
    <classPathEntry location="E:\mybatis-generator-core-1.3.2\lib\mysql-connector-5.1.8.jar"
>
    <context id="sysGenerator" targetRuntime="MyBatis3">

    <!--去除注释-->
        <commentGenerator >
          <property name="suppressDate" value="true"/>
           <property name="suppressAllComments" value="true"/>
        </commentGenerator>

<!-- 数据源，指定我们要生成实体类的数据库路径，用户名密码 -->
        <jdbcConnection
        driverClass="com.mysql.jdbc.Driver"
           connectionURL="jdbc:mysql://121.19.0.0:3306/mytest"
           userId="root" password="">
        </jdbcConnection>

    <!--用于配置实体类的所在的包targetPackage是包名
    targetProject指定生成的java代码和包存放的路径
    -->
        <javaModelGenerator targetPackage="com.math.hpu.domain"
           targetProject="E:\mybatis-generator-core-1.3.2\mybatis">
           <property name="enableSubPackages" value="true" />
           <property name="trimStrings" value="true" />
        </javaModelGenerator>

        <!--xml文件存在包和路径-->
        <sqlMapGenerator targetPackage="com.math.hpu.domain"
           targetProject="E:\mybatis-generator-core-1.3.2\mybatis">
           <property name="enableSubPackages" value="true" />
        </sqlMapGenerator>

        <!--配置接口存在包和保存的路径targetPackage表示包名,targetProject生成的文件保存的位置
```

```
->
        <javaClientGenerator type="XMLMAPPER"
          targetPackage="com.math.hpu.mapper" targetProject="E:\mybatis-generator-core-1.3
2\mybatis">
            <property name="enableSubPackages" value="true" />
        </javaClientGenerator>

        <!--指定生成哪些表-->

        <!--每张表都需要对应一个table标签对他表示设置-->
        <!--tableName表示表名，domainObjectName 表示实体类的名字-->
        <table tableName="user" domainObjectName="User"

            enableCountByExample="false"
            enableUpdateByExample="false" enableDeleteByExample="false"
            enableSelectByExample="false" selectByExampleQueryId="false">
            <generatedKey column="ID" sqlStatement="MYSQL" identity="true" />
        </table>

    </context>
</generatorConfiguration>
```

六、mybatis二级缓存

一级缓存:

一级缓存默认开启，缓存范围为SqlSession会话，即一个session对象，范围太小，声明周期短。两个ession对象查询后，数据存储在不同的内存地址中。当我们在commit提交后会强制清空namespace存。高缓存命中率，提高查询速度，使用二级缓存。

二级缓存:

二级缓存的范围大，属于范围Mapper Namespace，周期长。需要设置catch标签。在一个namespae空间内，多个session对象执行同一个id的查询，查询后的数据放在一个缓存地址中。第一次从硬盘数据库中查询数据，后面再次查询不再执行sql语句，直接从缓存中提取数据，速度更快。

```
<cache eviction="LRU" flushInterval="600000" size="512" readOnly="true"/>
```

eviction:是缓存策略

- LRU：移除长时间未使用的对象
- FIFO:按照进入缓存的顺序移除,遵循先进先出
- SOFT（弱）WEAK（强）：移除基于垃圾回收器状态的对象

flushInterval：是间隔时间，单位毫秒

size：是二级缓存大小（缓存上限）

readOnly:

- true：返回只读缓存，每次取出缓存对象本身，效率较高
- false：每次取出缓存对象副本，每次取出对象都不同，安全性较高

使用规则：

- 二级开启后默认所有查询操作均使用缓存
- 写操作commit提交时对该namespace缓存强制清空
- 配置useCache=false可以不用缓存
- 配置flushCache=true代表强制清空缓存

```
<!-- 不建议把包含很多的list集合保存到缓存中，缓存命中率低 设置useCache="false"不使用缓存-->
<select id="selectAll" resultType="com.wsl.pojo.user" useCache="false">
   select * from user
</select>
```

七、pageHelper使用

引入依赖：

```
<dependency>
   <groupId>com.github.pagehelper</groupId>
   <artifactId>pagehelper</artifactId>
   <version>5.1.2</version>
</dependency>

<!--开始引入2.0的jsqlparser会出现错误，不知道什么原因-->
<dependency>    <groupId>com.github.jsqlparser</groupId>
      <artifactId>jsqlparser</artifactId>
      <version>1.4</version>
</dependency>
```

mybatis-config.xml增加Plugin配置

```
<plugins>
   <!-- 配置拦截器插件，新版拦截器是 com.github.pagehelper.PageInterceptor-->
   <plugin interceptor="com.github.pagehelper.PageInterceptor">
      <!--设置数据库类型-->
      <property name="helperDialect" value="mysql"/>
      <!--分页合理化-->
      <property name="reasonable" value="true"/>
   </plugin>
</plugins>
```

service层运用：

```
PageHelper.startPage(2,10);
```

测试：

```
@Test
   public void test(){
      PageHelper.startPage(1,5);
      List<User> users = testUserMapper.selectAllUsers();
      sqlSession.commit();
      for (User user:users)System.out.println(user);
   }
```

```
}
```

八、mybatis整合c3p0连接池

引入依赖：

```
<dependency>
    <groupId>com.mchange</groupId>
    <artifactId>c3p0</artifactId>
    <version>0.9.5.4</version>
</dependency>
```

创建C3P0数据源：

```
public class C3P0Data extends UnpooledDataSourceFactory {
    public C3P0Data(){
        this.dataSource = new ComboPooledDataSource();
    }
}
```

修改mybatis-config.xml数据连接：

```
<dataSource type="com.wsl.datesource.C3P0Data">
    <property name="driverClass" value="com.mysql.jdbc.Driver"/>
    <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/mytest?useUnicode=true&
haracterEncoding=UTF-8&SSL=false"/>
    <property name="user" value="root"/>
    <property name="password" value=""/>
    <property name="initialPoolSize" value="5"/>
    <property name="maxPoolSize" value="20"/>
    <property name="minPoolSize" value="5"/>
</dataSource>
```

文章到此结束，写了一天，实在是写不动了。