

Mybatis_Day2

1. 输入映射和输出映射

Mapper.xml 映射文件中定义了操作数据库的 sql，每个 sql 是一个 statement，映射文件是 mybatis 的核心。

1.1. parameterType(输入类型)

1.2. 传递简单类型

参考第一天内容

1.3. 传递 pojo 对象

Mybatis 使用 ognl 表达式解析对象字段的值，#{ } 或者 \${ } 括号中的值为 pojo 属性名称。

1.4. 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，一般是使用包装类传入参数

Pojo 类中包含 Pojo。

需求：根据年级名称查询年级信息，查询条件放到 QueryVo 的 grade 属性中。

QueryVo

```
public class QueryVo {  
    private Grade grade;  
    public Grade getGrade() {  
        return grade;  
    }  
    public void setGrade(Grade grade) {  
        this.grade = grade;  
    }  
}
```

Sql 语句

```
SELECT * FROM Grade WHERE id LIKE #{grade.id} AND  
grade=${grade.grade}
```

Mapper 文件

```
<select id="findGradeByVo" parameterType="com.neuesoft.sz.vo.QueryVo"
resultType="com.neuesoft.sz.pojo.Grade">
    SELECT * FROM Grade WHERE id LIKE #{grade.id} AND grade=${grade.grade}
</select>
```

接口

```
public interface GradeMapper {
    public Grade findGradeById(Integer id);
    public List<Grade> findGradeByGradeName(String grade);
    public void saveGrade(Grade grade);
    public List<Grade> findGradeByVo(QueryVo vo);
}
```

测试方法

```
@Test
public void findGradeByV0() {
    SqlSession openSession = factory.openSession();
    GradeMapper mapper = openSession.getMapper(GradeMapper.class);
    QueryVo qv = new QueryVo();
    Grade grade = new Grade();
    grade.setId(1);
    grade.setGrade(2020);
    qv.setGrade(grade);
    List<Grade> grades = mapper.findGradeByVo(qv);
    System.out.println(grades);
}
```

2. resultType(输出类型)

2.1. 输出简单类型

Mapper.xml 文件

```
<!-- 获取年级列表总数 -->
<!-- 只有返回值类型为一行一列时，才能指定返回值类型为基本数据类型-->
<select id="findGradeCount" resultType="java.lang.Integer">
    select count(1) from Grade
</select>
```

Mapper 接口

```
public Integer findGradeCount();
```

测试

```
@Test
public void findGradeCount() {
    SqlSession openSession = factory.openSession();
    //通过 getMapper 方法来实例化接口
    GradeMapper mapper = openSession.getMapper(GradeMapper.class);
    Integer count = mapper.findGradeCount();
    System.out.println(count);
}
```

输出简单类型必须查询出来的结果集有一条记录，最终将第一个字段的值转换为输出类型。
使用 session 的 selectOne 可查询单条记录。

2.2. 输出 pojo 对象

参考第一天内容

2.3. 输出 pojo 列表（集合）

参考第一天内容。

3. resultMap

resultType 可以指定 pojo 将查询结果映射为 pojo，但需要 pojo 的属性名和 sql 查询的列名一致方可映射成功。

如果 sql 查询字段名和 pojo 的属性名不一致，可以通过 resultMap 将字段名和属性名作一个对应关系，resultMap 实质上还需要将查询结果映射到 pojo 对象中。

resultMap 可以实现将查询结果映射为复杂类型的 pojo，比如在查询结果映射对象中包括 pojo 和 list 实现一对一查询和一对多查询。

3.1. 简单使用

配置文件

```
<resultMap id="gradeMap" type="com.neuesoft.sz.pojo.Grade">
    <id property="id" column="id"/>
</resultMap>
```

```

    <result property="grade" column="grade"/>
</resultMap>
<select id="findAllGrade" resultMap="gradeMap">
    SELECT * FROM Grade
</select>

```

上面代码，<resultMap>的子元素<id>和<result>的 property 属性表示 Grade 类的属性名，column 属性表示数据表 grade 的列名
<select>元素的 resultMap 属性表示引用上面定义的 resultMap

接口

```
public List<Grade> findAllGrade();
```

测试代码

```

@Test
public void findAllGrade() {
    SqlSession openSession = factory.openSession();
    //通过 getMapper 方法来实例化接口
    GradeMapper mapper = openSession.getMapper(GradeMapper.class);
    List<Grade> grades = mapper.findAllGrade();
    System.out.println(grades);
}

```

复杂使用请看关联查询

4. 动态 sql

通过 mybatis 提供的各种标签方法实现动态拼接 sql。

4.1. if

类似于 Java 中的 if 语句，主要用于实现某些简单的条件选择

where 1=1; 此条件始终为 true，在不定数量查询条件情况下，1=1 可以很方便的规范语句。

```

<!-- 传递 pojod 多条件查询信息 -->
<select id="findGradeByIdAndName"
parameterType="com.neuesoft.sz.pojo.Grade"
resultType="com.neuesoft.sz.pojo.Grade">
    SELECT * FROM Grade
    WHERE 1=1
    <if test="id != 0">
        AND id=#{id}
    </if>
    <if test="grade != null and grade != ''">
        AND grade LIKE '%${grade}%'

```

```
</if>
</select>
```

注意要做不等于空字符串校验。

接口

```
public List<Grade> findGradeByIdAndName(Grade grade);
```

测试

```
@Test
public void findGradeByIdAndName() {
    SqlSession openSession = factory.openSession();
    //通过 getMapper 方法来实例化接口
    GradeMapper mapper = openSession.getMapper(GradeMapper.class);
    Grade grade = new Grade();
    //      grade.setId(1);
    grade.setGrade(20);
    List<Grade> grades = mapper.findGradeByIdAndName(grade);
    System.out.println(grades);
}
```

4.2. where

普通示例

```
<!-- 传递 pojod 多条件查询信息 -->
<select id="findGradeByIdAndName"
parameterType="com.neuesoft.sz.pojo.Grade"
resultType="com.neuesoft.sz.pojo.Grade">
    SELECT * FROM Grade WHERE
    <if test="id != 0">
        id = #{id}
    </if>
    <if test="grade != null and grade != ''">
        AND grade LIKE '%${grade}%'
    </if>
</select>
```

如果这些条件没有一个能匹配上将会怎样？最终这条 SQL 会变成这样：

```
SELECT * FROM Grade WHERE
```

这会导致查询失败。如果仅仅第二个条件匹配又会怎样？这条 SQL 最终会是这样：

```
SELECT * FROM Grade WHERE AND grade like '20'
```

这个查询也会失败。为了解决这个问题，MyBatis 提供了一个简单的处理

```
<!-- 传递 pojod 多条件查询信息 -->
<select id="findGradeByIdAndName"
parameterType="com.neuesoft.sz.pojo.Grade"
resultType="com.neuesoft.sz.pojo.Grade">
    SELECT * FROM Grade
    <where>
        <if test="id != 0">
            id=#{id}
        </if>
        <if test="grade != null and grade != ''">
            AND grade LIKE '%${grade}%'
        </if>
    </where>
</select>
```

where 元素知道只有在一个以上的 if 条件有值的情况下才去插入“WHERE”子句。而且，若最后的内容是“AND”或“OR”开头的，where 元素也知道如何将他们去除。

4.3. foreach

向 sql 传递数组或 List，mybatis 使用 foreach 解析，如下：

◆ 需求

传入多个 id 查询年级信息，用下边两个 sql 实现：

```
SELECT * FROM Grade WHERE grade LIKE '%20%' AND (id =1 OR id =8 OR id=6)
SELECT * FROM Grade WHERE grade LIKE '%20%' AND id IN (1,8,6)
```

◆ 在 pojo 中定义 list 属性 ids 存储多个年级 id，并添加 getter/setter 方法

```
public class QueryVo {
    private Grade grade;
    private List<Integer> ids;
    public Grade getGrade() {
        return grade;
    }
}
```

◆ mapper.xml

```
<!-- 传递 pojod 多条件查询信息 -->
<select id="findGradeByIds" parameterType="com.neuesoft.sz.vo.QueryVo"
resultType="com.neuesoft.sz.pojo.Grade">
    SELECT * FROM Grade
    <where>
        <if test="ids != null">
            <!--
```

```

        item: 配置的是循环中当前的元素
        index: 配置的是当前元素在集合中的位置下标
        collection: 配置的是传递过来的参数类型
        open: 循环开始时拼接的字符串
        close: 循环结束后拼接的字符串
        separator: 循环中拼接的字符串
    -->
    <foreach collection="ids" item="id" open="id in(" separator=","
close=")">
        #{id}
    </foreach>
</if>
</where>
</select>

```

◆ 测试代码:

```

@Test
public void testFindGradeByIds() {
    SqlSession openSession = factory.openSession();
    //通过 getMapper 方法来实例化接口
    GradeMapper mapper = openSession.getMapper(GradeMapper.class);
    QueryVo vo = new QueryVo();
    ArrayList<Integer> ids = new ArrayList<>();
    ids.add(1);
    ids.add(2);
    ids.add(3);
    vo.setIds(ids);
    List<Grade> grades = mapper.findGradeByIds(vo);
    System.out.println(grades);
}

```

4.4.Sql 片段

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

```

<select id="findGradeByIdAndName"
parameterType="com.neuesoft.sz.pojo.Grade"
resultType="com.neuesoft.sz.pojo.Grade">
    SELECT * FROM Grade
    <where>
        <if test="id != 0">

```

```
        AND id=#{id}
    </if>
    <if test="grade != null and grade != ''">
        AND grade LIKE '%${grade}%'
    </if>
</where>
</select>
```

◆ 将 where 条件抽取出来：

```
<sql id="find_grade_where">
    <if test="id != 0">
        AND id=#{id}
    </if>
    <if test="grade != null and grade != ''">
        AND grade LIKE '%${grade}%'
    </if>
</sql>
```

◆ 使用 include 引用：

```
<select id="findGradeByIdAndName"
parameterType="com.neuesoft.sz.pojo.Grade"
resultType="com.neuesoft.sz.pojo.Grade">
    SELECT * FROM Grade
    <where>
        <include refid="find_grade_where" />
    </where>
</select>
```

注意：如果引用其它 Mapper.xml 的 sql 片段，则在引用时需要加上 namespace，如下：

```
<include refid="namespace.sql 片段"/>
```

5. 关联查询

5.1. 查询类型（【嵌套】【结果】）

嵌套查询

指的是通过执行另外一个 SQL 映射语句来返回所关联的对象类型。

嵌套结果

指的是通过联表查询将所需的所有字段内容先查询出来，再通过级联属性映射来创建复杂类型的结果对象。

5.2. 一对一查询

嵌套结果

在<resultMap>元素中，包含了一个<association>子元素，MyBatis 就是通过该元素来处理一对一关联关系的。

<association>元素使用时，有两种配置方式，如下所示：

```
<!-- 方式一：嵌套查询 -->
<association property="grade" column="gid"
    javaType="com.neuesoft.sz.pojo.Grade"
        select="com.neuesoft.sz.mapper.GradeMapper.findGradeById" />
<!-- 方式二：嵌套结果 -->
<association property="grade" javaType="com.neuesoft.sz.pojo.Grade">
    <id column="gid" property="id"/>
    <result column="grade" property="grade"/>
</association>
```

<association>元素中，通常可以配置以下属性：

property: 指定映射到的实体类对象属性，与表字段一一对应

column: 指定表中对应的字段

javaType: 指定映射到实体对象属性的类型

select: 指定引入嵌套查询的子 SQL 语句，该属性用于关联映射中的嵌套查询

fetchType: 指定在关联查询时是否启用延迟加载，fetchType 属性有 lazy 和 eager 两个属性值，默认值为 lazy（即默认关联映射延迟加载）

案例：查询班级信息，关联查询班级所在的年级信息

SQL:

指定 id:

```
SELECT c.id,c.cno,g.id gid,g.grade FROM Class c, Grade g WHERE c.id = g.id AND c.id = 1;
```

不指定 id:

```
SELECT c.id,c.cno,g.id gid,g.grade FROM Class c, Grade g WHERE c.id = g.id;
```

配置文件

```
<select id="findClazz" resultMap="clazz">
    SELECT c.id,c.cno,g.id gid,g.grade FROM Class c, Grade g WHERE c.id =
    g.id;
</select>
<resultMap id="clazz" type="com.neuesoft.sz.pojo.Clazz">
    <id property="id" column="id"/>
    <result property="cno" column="cno"/>
    <!-- 一对一 -->
    <association property="grade" javaType="com.neuesoft.sz.pojo.Grade">
```

```
        <id column="gid" property="id"/>
        <result column="grade" property="grade"/>
    </association>
</resultMap>
```

接口

```
public interface ClazzMapper {
    List<Clazz> findClazz();
}
```

测试

```
@Test
public void findAllGrade() {
    SqlSession openSession = factory.openSession();
    //通过 getMapper 方法来实例化接口
    ClazzMapper mapper = openSession.getMapper(ClazzMapper.class);
    List<Clazz> clazzs = mapper.findClazz();
    System.out.println(clazzs);
}
```

嵌套查询

所谓嵌套查询，指的是通过执行另外一个 SQL 映射语句来返回所关联的对象类型。这种方式与根据业务逻辑于动执行多次 SQL 的方式相像，最后会将结果组合成一个对象。

在上面的一对一示例中，查询班级（Clazz）对应的年级信息（Grade）时，可以使用嵌套查询，具体操作如下：

在 GradeMapper.xml 中添加通过 id 查询年级信息的配置，如下

```
<!-- 通过 id 查询指定的年级信息 -->
<select id="findGradeById" parameterType="int"
resultType="com.neuesoft.sz.pojo.Grade">
    SELECT * FROM grade WHERE id = #{id}
</select>
```

在 ClazzMapper.xml 文件中添加配置（通过 id 查询班级信息），如下

```
<!-- 通过 id 查询指定的班级 -->
<select id="findClazzById" resultMap="clazz2" parameterType="int">
    SELECT * FROM Class WHERE id = #{id};
</select>
<resultMap id="clazz2" type="com.neuesoft.sz.pojo.Clazz">
    <id property="id" column="id"/>
    <result property="cno" column="cno"/>
<!-- 一对一 -->
```

```

    <association property="grade" column="gid"
    javaType="com.neuesoft.sz.pojo.Grade"
    select="com.neuesoft.sz.mapper.GradeMapper.findGradeById"
    />
</resultMap>

```

执行流程

1. 先执行 findClazzById 对应的语句从 Clazz 表里获取到 ResultSet 结果集；
2. 取出 ResultSet 下一条有效记录，然后根据 resultMap 定义的映射规格，通过这条记录的数据来构建对应的一个 Clazz 对象。
3. 当要对 Clazz 中的 grade 属性进行赋值的时候，发现有一个关联的查询，此时 MyBatis 会先执行这个 select 查询语句，得到返回的结果，将结果设置到 Clazz 的 grade 属性上

5.3. 一对多查询

嵌套结果查询

<collection> MyBatis 通过该元素来处理一对多关联关系的，其中大部分属性和<association>相同，特殊属性如下：

```

<!-- 集合中的 property 须为 ofType 定义的 pojo 对象的属性-->
<collection property="pojo 的集合属性" ofType="集合中的 pojo 对象">
    <id column="集合中 pojo 对象对应的表的主键字段" jdbcType="字段类型"
    property="集合中 pojo 对象的主键属性" />
    <result column="可以为任意表的字段" jdbcType="字段类型" property="集合中
    的 pojo 对象的属性" />
</collection>

```

ofType: 该属性和 javaType 属性对应，用于指定实体对象中集合类属性所包含的元素类型。

案例：通过查询所有的年级以及年级对应的班级信息

POJO

```

public class Grade {
    private int id;
    private int grade;
    private ArrayList<Clazz> clazzs = new ArrayList<>();
}

```

配置文件（嵌套结果）

```

<resultMap id="gradeMap" type="com.neuesoft.sz.pojo.Grade">
    <id column="id" property="id"/>

```

```

<result column="grade" property="grade"/>
<collection property="clazzs" ofType="com.neuesoft.sz.pojo.Clazz">
    <id property="id" column="cid" />
    <result column="cno" property="cno"/>
</collection>
</resultMap>

<select id="findAllGradeAndClazz" resultMap="gradeMap">
    SELECT g.id,g.grade,c.id cid,c.cno FROM grade g,class c WHERE g.id = c.gid;
</select>

```

测试

```

@Test
public void testFindAllGradeAndClazz() {
    SqlSession openSession = factory.openSession();
    //通过 getMapper 方法来实例化接口
    GradeMapper mapper = openSession.getMapper(GradeMapper.class);
    List<Grade> grades = mapper.findAllGradeAndClazz();
    System.out.println(grades);
}

```

6. MyBatis 延迟加载

在使用 MyBatis 嵌套方式进行查询时，使用 MyBatis 延迟加载在一定程度上可以降低运行消耗并提高查询效率，MyBatis 默认没有开启延迟加载，需要在核心配置文件中的<setting>元素内进行配置

```

<settings>
    <!-- 打开延迟加载开关-->
    <setting name="lazyLoadingEnabled" value="true"/>
    <!-- 将积极加载改为消息加载，即按需加载-->
    <setting name="aggressiveLazyLoading" value="false"/>
</settings>

```

7. MyBatis 整合 Spring

7.1. 整合思路

1、数据库的连接以及数据库连接池事务管理都交给 Spring 容器来完成。

- 2、SqlSessionFactory 对象应该放到 Spring 容器中作为单例存在。
- 3、传统 dao 的开发方式中，应该从 Spring 容器中获得 sqlSession 对象。
- 4、Mapper 代理形式中，应该从 Spring 容器中直接获得 Mapper 的代理对象。

7.2. 整合需要的 jar 包

- 1、Spring 的 jar 包
- 2、Mybatis 的 jar 包
- 3、Spring+MyBatis 的整合包。
- 4、Mysql 的数据库驱动 jar 包。
- 5、数据库连接池的 jar 包。

7.3. 整合的步骤

第一步：创建一个工程。（普通 Java 项目使用 Maven 管理，或者 Web 项目均可）

第二步：导入 jar 包。（上面提到的 jar 包，或者 Maven 导入）

第三步：MyBatis 的配置文件 SqlMapConfig.xml

第四步：编写 Spring 的配置文件

- 1、数据库连接及连接池
- 2、事务管理（暂时可以不配置）
- 3、sqlSessionFactory 对象，配置到 Spring 容器中
- 4、Mapeer 代理对象或者是 Dao 实现类配置到 Spring 容器中。

第五步：编写 Dao 或者 Mapper 文件

第六步：测试。

7.3.1. 编写配置文件

分别创建 db.properties、applicationContext.xml、mybatis-config.xml 等文件

db.properties

```
druid.driverClassName=com.mysql.jdbc.Driver
druid.url=jdbc:mysql://localhost:3306/examsystem?characterEncoding=utf-8&useSSL=false
druid.username=root
druid.password=123456
```

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>
```

```
<typeAliases>
    <package name="com.neuesoft.sz.pojo"/>
</typeAliases>
<mappers>
    <mapper resource="Grade.xml"/>
</mappers>
</configuration>
```

applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
        http://www.springframework.org/schema/context
            http://www.springframework.org/schema/context/spring-context-4.0.xsd">
    <!-- 加载配置文件 -->
    <context:property-placeholder location="classpath:db.properties"/>
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource"
        destroy-method="close">
        <property name="url" value="${druid.url}"/>
        <property name="username" value="${druid.username}"/>
        <property name="password" value="${druid.password}"/>
        <property name="driverClassName"
value="${druid.driverClassName}"/>
    </bean>

    <!-- mapper 配置 -->
    <!-- 让spring 管理sqlSessionFactory 使用mybatis 和spring 整合包中的 -->
    <bean id="sqlSessionFactory"
class="org.mybatis.spring.SqlSessionFactoryBean">
        <!-- 数据库连接池 -->
        <property name="dataSource" ref="dataSource"/>
        <!-- 加载mybatis 的全局配置文件 -->
        <property name="configLocation"
value="classpath:SqlMapConfig.xml"/>
    </bean>

    <bean id="gradeDao" class="com.neuesoft.sz.dao.impl.GradeDaoImpl">
        <property name="sqlSessionFactory" ref="sqlSessionFactory"/>
    </bean>
```

```
</beans>
```

7.4.Dao 的开发

三种 Dao 的实现方式:

- 1、传统 Dao 的开发方式
- 2、使用 Mapper 代理、扫描包配置 Mapper 代理。

7.5.传统 dao 的开发方式

接口+实现类来完成。Dao 实现类需要继承 SqlSessionDaoSupport 类

Dao 接口

```
public interface GradeDao {  
    //通过id 查询  
    Grade findGradeById(Integer id);  
    //通过年级名模糊查询  
    List<Grade> findGradeByGradeName(String grade);  
}
```

配置文件

```
<mapper namespace="grade">  
    <select id="findGradeById" parameterType="int"  
resultType="com.neuesoft.sz.pojo.Grade">  
        SELECT * FROM grade WHERE id = #{id}  
    </select>  
  
    <select id="findGradeByName" parameterType="string"  
resultType="com.neuesoft.sz.pojo.Grade">  
        SELECT * FROM grade where grade like '%${value}%'  
    </select>  
</mapper>
```

Dao 实现类

```
public class GradeDaoImpl extends SqlSessionDaoSupport implements GradeDao  
{  
    @Override  
    public Grade findGradeById(Integer id) {  
        Grade grade = getSqlSession().selectOne("grade.findGradeById",  
id);  
        return grade;  
    }  
}
```

```

    }
    @Override
    public List<Grade> findGradeByGradeName(String grade) {
        List<Grade> list =
        getSqlSession().selectList("grade.findGradeByName", grade);
        return list;
    }
}

```

配置 Dao

把 Dao 实现类配置到 Spring 容器中

```

<bean id="gradeDao" class="com.neuesoft.sz.dao.impl.GradeDaoImpl">
    <property name="sqlSessionFactory" ref="sqlSessionFactory"/>
</bean>

```

测试

初始化:

```

private ApplicationContext act;
@Before
public void init() {
    act = new ClassPathXmlApplicationContext("applicationContext.xml");
}

```

测试:

```

@Test
public void findGradeById() {
    GradeDao gradeDao = (GradeDao) act.getBean("gradeDao");
    Grade grade = gradeDao.findGradeById(1);
    System.out.println(grade);
}
@Test
public void findGradeByGradeName() {
    GradeDao gradeDao = (GradeDao) act.getBean("gradeDao");
    List<Grade> grades = gradeDao.findGradeByGradeName("20");
    System.out.println(grades);
}

```

7.6.Mapper 代理形式开发 Dao

Mapper 接口

```

public interface GradeMapper {
    Grade findGradeById(Integer id);
}

```



```
List<Grade> findGradeByGradeName(String grade);  
}
```

配置文件（GradeMapper.xml）

```
<mapper namespace="com.neuesoft.sz.mapper.GradeMapper">  
  <select id="findGradeById" parameterType="int"  
resultType="com.neuesoft.sz.pojo.Grade">  
    SELECT * FROM grade WHERE id = #{id}  
  </select>  
  
  <select id="findGradeByGradeName" parameterType="string"  
resultType="com.neuesoft.sz.pojo.Grade">  
    SELECT * FROM grade where grade like '%${value}%'  
  </select>  
</mapper>
```

配置 Mapper 代理

MapperFactoryBean 是 MyBatis-Spring 团队提供的一个用于根据 Mapper 接口生成 Mapper 对象的类，该类在 Spring 配置文件中可以配置如下属性：

mapperInterface: 用于指定接口

SqlSessionFactory: 指定 SqlSessionFactory

SqlSessionTemplate: 指定 SqlSessionTemplate

```
<!-- 配置 mapper 代理对象 -->  
<bean class="org.mybatis.spring.mapper.MapperFactoryBean">  
  <property name="mapperInterface"  
value="com.neuesoft.sz.mapper.GradeMapper"/>  
  <property name="sqlSessionFactory" ref="sqlSessionFactory"/>  
</bean>
```

上述配置，指定了 SqlSessionFactory 和接口

测试方法

```
public class GradeMapperTest {  
  private ApplicationContext act;  
  @Before  
  public void init() {  
    act = new  
ClassPathXmlApplicationContext("classpath:applicationContext.xml");  
  }  
  @Test  
  public void findGradeById() {  
    GradeMapper gradeMapper = act.getBean(GradeMapper.class);  
    Grade grade = gradeMapper.findGradeById(1);  
  }  
}
```

```
        System.out.println(grade);
    }
}
```

MapperScannerConfigurer

在实际项目中，Dao 层包含很多接口，如果每一个接口都单独进行配置，那么会增加代码量，为此 MyBatis-Spring 提供了一种自动扫描的形式来配置 MyBatis 中的映射器 MapperScannerConfigurer

MapperScannerConfigurer 在 Spring 配置文件中使用时可以配置如下属性：

basePackage：指定映射接口文件所在的包路径，需要扫描多个包使用逗号分隔

annotationClass：指定要扫描的注解名称

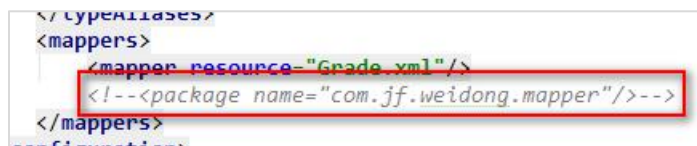
sqlSessionFactoryBeanName：指定在 Spring 中定义的 SqlSessionFactory 的 Bean 名称

sqlSessionTemplateBeanName：指定在 Spring 中定义的 SqlSessionTemplate 的 Bean 名称

```
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
    <property name="basePackage" value="com.neuesoft.sz.mapper"/>
</bean>
```

通常情况下，只需要使用 basePackage 属性指定扫描的包即可，Spring 会自动通过包中的接口来生成映射器

在 applicationContext.xml 中配置了扫描，那么在 SqlMapConfig.xml 配置文件中就不需要额外配置了。



```
</typeAliases>
<mapers>
    <mapper resource="Grade.xml"/>
    <!--<package name="com.jf.weidong.mapper"/>-->
</mapers>
</configuration>
```

7.7. 测试事务

在 MyBatis-Spring 的项目中，事务是由 Spring 来管理的，配置事务管理器如下

```
<!-- 配置事务管理器 -->
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
">
    <property name="dataSource" ref="dataSource"/>
</bean>
<!-- 开启事务控制的注解支持 -->
<tx:annotation-driven transaction-manager="transactionManager"/>
```

在项目中，业务层（Service 层）既是处理业务又是管理数据库事务的地方，要对事务进行测试，首先需要创建业务层，在业务层编写添加年级信息的代码，然后再添加操作的代码后，

认为添加一段异常代码，来模拟现实中的意外情况，最后调用业务中添加年级的方法，查看效果。

在 GradeMapper 接口中添加添加年级的方法

```
void addGrade(Grade grade);
```

添加配置信息

```
<insert id="addGrade" useGeneratedKeys="true" keyProperty="id"
parameterType="com.neuesoft.sz.pojo.Grade">
    INSERT INTO Grade (grade) VALUES ({grade})
</insert>
```

创建 Service 包，添加业务处理

```
@Service
@Transactional
public class GradeServiceImpl implements GradeService {
    @Autowired
    GradeMapper gradeMapper;
    @Override
    public void addGrade(Grade grade) {
        gradeMapper.addGrade(grade);
        //人为抛异常
        int i = 10 / 0;
    }
}
```

测试

```
@Test
public void transaction() {
    ApplicationContext act = new
    ClassPathXmlApplicationContext("applicationContext.xml");
    GradeService gradeService = act.getBean(GradeService.class);
    Grade grade = new Grade();
    grade.setGrade(2010);
    gradeService.addGrade(grade);
}
```

推荐链接

[MyBatis 相关工具](#)

德鲁伊数据库连接池的配置