MyBatis

一、MyBatis 基础

1. 课程目标

- 掌握 MyBatis 概念
- 掌握 MyBatis 接口和映射文件配置

2. 知识架构树

- MyBatis 简介
- MyBatis 基础配置
- MyBatis 实现增删改查

3. 理论知识

3.1 MyBatis 简介

> MyBatis 是一款优秀的**持久层**框架，它支持自定义 SQL、存储过程以及高级映射。**MyBatis 免除了几乎所有的 JDBC 代码以及设置参数和获取结果集的工作。**MyBatis 可以通过简单的 XML 或注解来配置和映射原始类型、接口和 Java POJO（Plain Old Java Objects，普通老式 Java 对象）为数据库中的记录。
>
> 一款优秀的**半自动化**的基于**ORM**思想开发的**持久层**框架；
>
> **ORM（Object Relational Mapping）对象关系映射：** 类和表对应：一张表要对应一个实体类。 类中的属性和表字段对应：表中的字段要在实体类中有所体现。 类的对象和表的一条记录对应：表中的每一条记录，对应到ava中就是一个实体类的实例。
>
> 中文官网：https://mybatis.org/mybatis-3/zh/

3.2 MyBatis 基础配置

1、导包

```xml
<!-- MyBatis 依赖 -->
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis</artifactId>
    <version>3.5.11</version>
</dependency>
<!-- 数据库驱动依赖 -->
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.27</version>
</dependency>
```

2、编写持久层接口

```java
public int queryCount();
```

3、编写持久层接口的映射文件（代替接口的实现类）

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<!--
    namespace：MyBtatis3之后是必填项，用于标识这个文件和哪个接口对应；填接口的全限定名称
-->
<mapper namespace="com.soft.dao.StudentDao">
    <!--
        id：持久层接口的方法名，区分大小写
        resultType：
            返回值类型，要写类型的全限定名称，由于MyBatis对JDK中一些类型进行了封装，可以直接写类名；
            自定义的类型要写全限定名称，配置后也可以写类名
            如果返回值是list类型，填List的泛型
    -->
    <select id="queryCount" resultType="integer">
        select count(1) from student
    </select>
</mapper>
```

4、编写MyBatis配置文件，名字没有要求，但是要见名知意，例如：mybatis-config.xml

```properties
# mysql
mysql.driver=com.mysql.cj.jdbc.Driver
mysql.url=jdbc:mysql://localhost:3306/riu
mysql.username=账号
mysql.password=密码

# oracle
oracle.driver=com.mysql.cj.jdbc.Driver
oracle.url=jdbc:oracle:thin:@10.87.221.199:1521:orcl
oracle.username=账号
oracle.password=密码
```

```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">
<!--MyBatis的配置文件-->
<configuration>
    <!--
        加载properties文件
        resource：文件的路径，以/分隔开
    -->
    <properties resource="jdbc.properties"></properties>

    <!--
        配置数据库的信息
```

```
        default：默认使用哪个数据库
    -->
    <environments default="dev">
        <!--开发环境-->
        <environment id="dev">
            <!--事务管理器，使用JDBC的事务管理器-->
            <transactionManager type="JDBC"></transactionManager>
            <!--数据库链接配置-->
            <dataSource type="POOLED">
                <property name="driver" value="${mysql.driver}"/>
                <property name="url" value="${mysql.url}"/>
                <property name="username" value="${mysql.username}"/>
                <property name="password" value="${mysql.password}"/>
            </dataSource>
        </environment>
    </environments>

    <!--配置映射文件-->
    <mappers>
        <!--resource是资源文件，目录是以斜杠分割的-->
        <mapper resource="com/soft/xml/StudentDaoXml.xml"/>
    </mappers>
</configuration>
```

5、编写测试类

```
public class MyBatisTest {

    @Test
    public void testQueryForList() throws IOException {
        // 配置文件路径
        String mybatisXML = "mybatis.xml";
        // 配置文件路径创建流对象
        InputStream inputStream = Resources.getResourceAsStream(mybatisXML);
        // 根据流对象创建Session工厂
        SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);

        // 开启数据库session会话
        SqlSession sqlSession = sqlSessionFactory.openSession();

        // 创建dao实例
        QueryDao mapper = sqlSession.getMapper(QueryDao.class);
        System.out.println(mapper.queryCount());
    }
}
```

3.3 MyBaits 实现增删改查

MyBatis的配置注意事项：

1. 持久层接口中不能有重载方法
2. 持久层映射 xml 文件必须添加 `namespace` 属性
3. 持久层映射 xml 文件中的标签的 id 值要和持久层接口中的方法名对应，包括大小写
4. 持久层映射 xml 文件的所有入参类型都可以省略不写
5. 持久层接口中方法的参数最好不要超过 1 个

3.3.1 参数绑定

> 多数业务需求下，SQL的操作都需要参数支持。根据需求可以传入单个参数、多个参数、对象参数等。MyBatis对入参的绑定有一定的要求。

- 单一参数

```
// 持久层接口方法只有一个参数时
Student queryForObject(String no);
```

```
<select id="queryForObject" resultType="com.soft.entity.Student">
    <!--
        SQL接收参数使用#{参数名}的形式，#{参数名}本质含义是占位符，类似于原生JDBC中的问号。
        MyBatis在给占位符赋值时采用的也是类似远程JDBC中按照问号先后顺序赋值的形式，只匹配占位符的顺序。
        所以当入参只有一个时，#{}中的内容可以任意填，但为了见名知意，这里写了和入参一样的名称
    -->
    select * from student where sno = #{sno}
</select>
```

- 多参数

```
// 持久层接口方法有多个个参数时
Student queryForObject(String sex, String clazz);
```

```xml
<select id="queryForObject" resultType="com.soft.entity.Student">
    <!--
        SQL接收参数使用#{参数名}的形式，#{参数名}本质含义是占位符，类似于原生JDBC中的问号。
        MyBatis在给占位符赋值时采用的也是类似远程JDBC中按照问号先后顺序赋值的形式，只匹配占位符的顺序。
        所以当入参只有多个时，#{}中的内容不可以任意填，为了明确参数和占位符的顺序关系，这里需要使用
            #{arg0} #{arg1} arg0代表第一个参数，arg1代表第二个参数，依此类推
            #{param1} #{param2} param1代表第一个参数，param2代表第二个参数，以此类推
        这种带有明确顺序的名称
    -->
    select * from student where sex = #{arg0} and class = #{arg1}
    select * from student where sex = #{param1} and class = #{param2}
</select>
```

```java
// 持久层接口方法有多个个参数时
// 为了明确参数和占位符的对应关系，MyBatis提供了对参数设置别名的方式。
// 通过@Param("名称")设置参数名称
Student queryForObject(@Param("sex") String sex, @Param("clazz") String clazz);
```

```xml
<select id="queryForObject" resultType="com.soft.entity.Student">
    <!--
        设置了参数名称时，可直接通过名称获取值
    -->
    select * from student where sex = #{sex} and class = #{clazz}
</select>
```

- 对象参数

单对象

```java
// 持久层接口方法时对象参数时
Student queryForObject(Student student);
```

```xml
<select id="queryForObject" resultType="com.soft.entity.Student">
    <!--
        入参是一个对象，#{} 就相当于调用了属性的get 方法。
        MyBatis对入参是对象的情况进行了处理，SQL接收参数直接写实体类中属性的get方法去掉get之后首字母小写即可。
            例如：getSno -> sno
        由于实体类中的set、get方法是通过工具自动生成的，具有规范性，所以有些地方也会说直接填写属性名称。
    -->
    select * from student where sex = #{sex} and class = #{clazz}
</select>
```

单 + 字面量

```java
// 持久层接口方法时对象参数时
Student queryForObject(@Param("student") Student student, @Param("index") int index, @Param("limit") int limit);
```

```xml
<select id="queryForObject" resultType="com.soft.entity.Student">
    <!--
        入参是一个对象，#{} 就相当于调用了属性的get 方法。
        MyBatis对入参是对象的情况进行了处理，SQL接收参数直接写实体类中属性的get方法去掉get之后首字母小写即可。
            例如：getSno -> sno
        由于实体类中的set、get方法是通过工具自动生成的，具有规范性，所以有些地方也会说直接填写属性名称。
    -->
    select * from student where sex = #{student.sex} and class = #{student.clazz}
    limit #{index} #{limit}
</select>
```

- Map 参数

```java
// 持久层接口方法时对象参数时
Student queryForObject(Map<String, Object> map);
```

```xml
<select id="queryForObject" resultType="com.soft.entity.Student">
    <!--
        入参是一个Map，#{} 就相当于调用了 map 的get() 方法。所以 #{} 中的值要写 map 的 key。
    -->
    select * from student where sex = #{sex} and class = #{clazz}
</select>
```

下面是一些为常见的 Java 类型内建的类型别名。它们都是不区分大小写的，注意，为了应对原始类型的命名重复，采取了特殊的命名风格。

别名	映射的类型
_byte	byte
_char (since 3.5.10)	char

别名	映射的类型
_character (since 3.5.10)	char
_long	long
_short	short
_int	int
_integer	int
_double	double
_float	float
_boolean	boolean
string	String
byte	Byte
char (since 3.5.10)	Character
character (since 3.5.10)	Character
long	Long
short	Short
int	Integer
integer	Integer
double	Double
float	Float
boolean	Boolean
date	Date
decimal	BigDecimal
bigdecimal	BigDecimal
biginteger	BigInteger
object	Object
date[]	Date[]
decimal[]	BigDecimal[]
bigdecimal[]	BigDecimal[]
biginteger[]	BigInteger[]
object[]	Object[]
map	Map
hashmap	HashMap
list	List
arraylist	ArrayList
collection	Collection
iterator	Iterator

3.3.2 基于 XML 配置

3.3.2.1 查询

- **查询返回对象（单参数）**

```
public Student queryForObject(String sno);
```

```
<select id="queryForObject" resultType="com.soft.entity.Student">
    <!-- SQL接收参数使用#{参数名}的形式 -->
    select * from student where sno = #{sno}
</select>
```

- **查询返回对象（多参数）**

方案一：按照参数顺序接收参数的值

```java
public Student queryForObject1(String sno, Integer age);
```

```xml
<select id="queryForObject1" resultType="com.soft.entity.Student">
    <!--
        多条件查询，SQL接收参数默认不识别参数名，在不指定参数名的情况下，
        使用arg0、arg1...来接收参数值：arg0代表第一个参数，arg1代表第二个参数，依此类推
        或者
        使用param1、param2...来接收参数值：param1代表第一个参数，param2代表第二个参数，以此类推
    -->
    select * from student where sno = #{arg0} and age = #{arg1}
    select * from student where sno = #{param1} and age = #{param2}
</select>
```

方案二：指定参数名称

```java
public Student queryForObject2(@Param("sno") String sno, @Param("age") Integer age);
```

```xml
<select id="queryForObject2" resultType="com.soft.entity.Student">
    <!--
        多条件查询可以在dao接口，@Param("name")的形式给参数指定名称
    -->
    select * from student where sno = #{sno} and age = #{age}
</select>
```

方案三：把参数封装为对象

```java
public Student queryForObject3(Student stu);
```

```xml
<select id="queryForObject3" resultType="com.soft.entity.Student">
    <!--
        入参是对象，SQL接收参数直接写实体类中属性的get方法去掉get之后首字母小写即可
        例如：getSno -> sno
    -->
    select * from student where sno = #{sno} and age = #{age}
</select>
```

方案四：把参数封装为`Map<String, String>`对象

```java
public Student queryForObject4(Map<String, String> map);
```

```xml
<select id="queryForObject4" resultType="com.soft.entity.Student">
    <!--
        入参是Map，SQL接收参数直接写Map中的key
        例如：{sno=100, age=18}
    -->
    select * from student where sno = #{sno} and age = #{age}
</select>
```

- **查询返回对象**`Map<String, Object>`

```java
public Map<String, Object> queryForMap(Student stu);
```

```xml
<select id="queryForMap" resultType="map">
    select * from student where sno = #{sno}
</select>
```

- **查询返回集合**`List<Object>`

```java
public List<Student> queryForList();
```

```xml
<!-- 方法的返回值类型如果是List加泛型，resultType应该写List的泛型 -->
<select id="queryForList" resultType="com.soft.entity.Student">
    select * from student
</select>
```

- **查询返回集合**`List<Map<String, Object>>`

```java
public List<Map<String, Object>> queryForListMap(Student stu);
```

```xml
<select id="queryForListMap" resultType="map">
    select * from student
</select>
```

- **模糊查询**

方案一：不能防止SQL注入

```
public List<Map<String, Object>> queryForListMap(Student stu);
```

```
<select id="queryForListMap" resultType="map">
    <!--MySQL、Oracle通用写法-->
    select * from student where sname like '%${sname}%'
</select>
```

```
Student student = new Student();
student.setSname("张");
```

方案二：可以防止SQL注入

```
public List<Map<String, Object>> queryForListMap(Student stu);
```

```
<select id="queryForListMap" resultType="map">
    <!--MySQL、Oracle通用写法-->
    select * from student where sname like #{sname}
</select>
```

```
Student student = new Student();
student.setSname("%张%");
```

方案三（*）：可以防止SQL注入

```
public List<Map<String, Object>> queryForListMap(Student stu);
```

```
<select id="queryForListMap" resultType="map">
    <!--MySQL写法-->
    select * from student where sname likt concat('%', #{sname}, '%')
</select>
```

```
<select id="queryForListMap" resultType="map">
    <!--Oracle写法-->
    select * from student where sname likt concat('%', concat(#{sname},'%'))
</select>
```

```
Student student = new Student();
student.setSname("张");
```

3.3.2.2 增删改

- **添加、删除、修改**

```
public int add(Student stu);
public int edit(Student stu);
public int del(Student stu);
```

```
<!-- 添加删除修改的返回类型是记录数，为int类型，所以可以省略resultType -->
<insert id="add">
    insert into student(sno, sname, sex, age, tel, native_place, class, pass, del)
    values (seq_student.nextval, #{sname}, #{sex}, #{age}, #{tel}, #{home}, #{clazz}, #{pass}, '1')
</insert>

<update id="edit">
    update student set sname = #{sname} where sno = #{sno}
</update>

<delete id="del">
    delete student where sno = #{sno}
</delete>
```

- **插入数据返回key**

```
public int add(Student stu);
```

使用 `<selectKey>`

```
<insert id="add">
    <!--
        resultType：返回值类型，当前标签中SQL的返回值类型，和入参中指定的类型一致
        order：在主SQL执行前执行或者执行后执行：
            BEFORE：在主SQL执行前执行
            AFTER：在主SQL执行后执行
```

```
            Oracle生成主键的策略：序列，通过调用序列生成，要优先于主SQL执行，所以要写BEFORE
            MySQL生成主键的策略：数据库自增，数据插入之后，数据库自动生成，所以要写AFTER

        keyProperty：SQL值要存到哪儿？主要取决于入参的类型，结果会自动封装到入参中
            1、入参类型是实体类：调用的是 set 方法值是实体类中对应属性的set方法去掉set之后，首字母小写的内容
            2、入参是Map类型：调用的是 put 方法，值可以随意，确保 key 在 Map 中不能重复

        插入数据之后返回主键，常用于注册成功之后返回主键信息，SQL执行成功之后会通过keyProperty把值设置到入参中
    -->
    <selectKey resultType="String" order="BEFORE" keyProperty="sno" >
        select seq_student.nextval sno from dual
    </selectKey>
    insert into student(sno, sname, sex, age, tel, native_place, class, pass, del)
    values (#{sno}, #{sname}, #{sex}, #{age}, #{tel}, #{home}, #{clazz}, #{pass}, '1')
</insert>
```

使用 `useGeneratedKeys`

```
<!--
    useGeneratedKeys：允许 JDBC 支持自动生成主键
    keyProperty：生成 key 对应的属性
        如果入参是 对象，调用的是 set 方法
        如果入参是 Map，调用的是 put 方法，确保 key 在 Map 中不能重复
-->
<insert id="add" useGeneratedKeys="true" keyProperty="sno">
    insert into student(sno, sname, sex, age, tel, native_place, class, pass, del)
    values (#{sno}, #{sname}, #{sex}, #{age}, #{tel}, #{home}, #{clazz}, #{pass}, '1')
</insert>
```

```
<settings>
    <!--
        允许 JDBC 支持自动生成主键，需要数据库驱动支持。
        如果设置为 true，将强制使用自动生成主键。
        尽管一些数据库驱动不支持此特性，但仍可正常工作（如 Derby）。
    -->
    <setting name="useGeneratedKeys" value="true"/>
</settings>
```

3.3.3 基于注解配置

> 设计初期的 MyBatis 是一个 XML 驱动的框架。配置信息是基于 XML 的，映射语句也是定义在 XML 中的。而在 MyBatis 3 中，我们提供了其它的配置方式。MyBatis 3 构建在全面且强大的基于 Java 语言的配置 API 之上。它是 XML 和注解配置的基础。**注解提供了一种简单且低成本的方式来实现简单的映射语句。**
>
> 不幸的是，Java 注解的表达能力和灵活性十分有限。尽管我们花了很多时间在调查、设计和试验上，但最强大的 MyBatis 映射并不能用注解来构建——我们真没开玩笑。而 C# 属性就没有这些限制，因此 MyBatis.NET 的配置会比 XML 有更大的选择余地。虽说如此，基于 Java 注解的配置还是有它的好处的。

> 使用注解来映射简单语句会使代码显得更加简洁，但对于稍微复杂一点的语句，Java 注解不仅力不从心，还会让本就复杂的 SQL 语句更加混乱不堪。 因此，如果你需要做一些很复杂的操作，最好用 XML 来映射语句。
>
> 选择何种方式来配置映射，以及是否应该要统一映射语句定义的形式，完全取决于你和你的团队。 换句话说，永远不要拘泥于一种方式，你可以很轻松地在基于注解和 XML 的语句映射方式间自由移植和切换。

- **注册接口**

```
// MyBatis 核心对象
SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
// 讲接口添加到映射
sqlSessionFactory.getConfiguration().addMapper(StudentMapperAnnotation.class);
// sqlSessionFactory.getConfiguration().addMappers("com.soft.mapper");
```

- **查询**

```
@Select("select * from student where sno = #{sno}")
public Student queryForObject(String sno);
```

- **增删改**

```
@Insert(
    "insert into student(sno, sname, sex, age, tel, native_place, class, pass, del) values " +
    "(seq_student.nextval, #{sname}, #{sex}, #{age}, #{tel}, #{home}, #{clazz}, #{pass}, '1')"
)
public int add(Student stu);

@Update("update student set sname = #{sname} where sno = #{sno}")
public int edit(Student stu);

@Delete("delete student where sno = #{sno}")
public int del(Student stu);
```

注解	使用对象	XML 等价形式	描述
`@CacheNamespace`	类	`<cache>`	为给定的命名空间（比如类）配置缓存。属性：`implemetation`、`eviction`、`flushInterval`、`size`、`readwrite`、`blocki`
`@Property`	N/A	`<property>`	指定参数值或占位符（placeholder）（该占位符能被 `mybatis-config.xml` 内的配置属性替换）。属性：`name`、`value`。（仅
`@CacheNamespaceRef`	类	`<cacheRef>`	引用另外一个命名空间的缓存以供使用。注意，即使共享相同的全限定类名，在 XML 映射文件中声明的缓存仍被识别为一个独立 性用于指定能够表示该命名空间的 Java 类型（命名空间名就是该 Java 类型的全限定类名），`name` 属性（这个属性仅在 MyBatis
`@ConstructorArgs`	方法	`<constructor>`	收集一组结果以传递给一个结果对象的构造方法。属性：`value`，它是一个 `Arg` 数组。
`@Arg`	N/A	`<arg>` `<idArg>`	ConstructorArgs 集合的一部分，代表一个构造方法参数。属性：`id`、`column`、`javaType`、`jdbcType`、`typeHandler`、`sel` 从版本 3.5.4 开始，该注解变为可重复注解。
`@TypeDiscriminator`	方法	`<discriminator>`	决定使用何种结果映射的一组取值（case）。属性：`column`、`javaType`、`jdbcType`、`typeHandler`、`cases`。cases 属性是
`@Case`	N/A	`<case>`	表示某个值的一个取值以及该取值对应的映射。属性：`value`、`type`、`results`。results 属性是一个 `Results` 的数组，因此这
`@Results`	方法	`<resultMap>`	一组结果映射，指定了对某个特定结果列，映射到某个属性或字段的方式。属性：`value`、`id`。value 属性是一个 `Result` 注解
`@Result`	N/A	`<result>``<id>`	在列和属性或字段之间的单个结果映射。属性：`id`、`column`、`javaType`、`jdbcType`、`typeHandler`、`one`、`many`。id 属性 `<association>` 类似，而 many 属性则是集合关联，和 `<collection>` 类似。这样命名是为了避免产生名称冲突。
`@One`	N/A	`<association>`	复杂类型的单个属性映射。属性： `select`，指定可加载合适类型实例的映射语句（也就是映射器方法）全限定名； `fetchType` qualified name of a result map that map to a single container object from select result； `columnPrefix`(available since 3.5 于 Java 注解不允许产生循环引用。
`@Many`	N/A	`<collection>`	复杂类型的集合属性映射。属性： `select`，指定可加载合适类型实例集合的映射语句（也就是映射器方法）全限定名； `fetchT` qualified name of a result map that map to collection object from select result； `columnPrefix`(available since 3.5.5), whi 注解不允许产生循环引用。
`@MapKey`	方法		供返回值为 Map 的方法使用的注解。它使用对象的某个属性作为 key，将对象 List 转化为 Map。属性：`value`，指定作为 Map
`@Options`	方法	映射语句的属性	该注解允许你指定大部分开关和配置选项，它们通常在映射语句上作为属性出现。与在注解上提供大量的属性相比，`Options` 注 `resultSetType=DEFAULT`、`statementType=PREPARED`、`fetchSize=-1`、`timeout=-1`、`useGeneratedKeys=false`、`keyPr 你使用了 `Options` 注解，你的语句就会被上述属性的默认值所影响。要注意避免默认值带来的非预期行为。 The `databaseId`(A `databaseId` attribute or with a `databaseId` that matches the current one. If found with and without the `databaseId` the 于 `keyColumn` 和 `keyProperty` 可选值信息，请查看“insert, update 和 delete”一节。
`@Insert` `@Update` `@Delete` `@Select`	方法	`<insert>` `<update>` `<delete>` `<select>`	每个注解分别代表将会被执行的 SQL 语句。它们用字符串数组（或单个字符串）作为参数。如果传递的是字符串数组，字符串数 失空格”问题。当然，你也可以提前手动连接好字符串。属性：`value`，指定用来组成单个 SQL 语句的字符串数组。 The `databa` no `databaseId` attribute or with a `databaseId` that matches the current one. If found with and without the `databaseId` t
`@InsertProvider` `@UpdateProvider` `@DeleteProvider` `@SelectProvider`	方法	`<insert>` `<update>` `<delete>` `<select>`	允许构建动态 SQL。这些备选的 SQL 注解允许你指定返回 SQL 语句的类和方法，以供运行时执行。（从 MyBatis 3.4.6 开始，可 注解指定的方法。你可以通过 `ProviderContext` 传递映射方法接收到的参数、"Mapper interface type" 和 "Mapper method" `databaseId`。 `value` and `type` 属性用于指定类名 (The `type` attribute is alias for `value`, you must be specify either one. E 于指定该类的方法名（从版本 3.5.1 开始，可以省略 `method` 属性，MyBatis 将会使用 `ProviderMethodResolver` 接口解析方法 将会讨论该话题，以帮助你以更清晰、更便于阅读的方式构建动态 SQL。 The `databaseId`(Available since 3.5.5), in case there with a `databaseId` that matches the current one. If found with and without the `databaseId` the latter will be discarded.
`@Param`	参数	N/A	如果你的映射方法接受多个参数，就可以使用这个注解自定义每个参数的名字。否则在默认情况下，除 `RowBounds` 以外的参数会 `#{person}`。
`@SelectKey`	方法	`<selectKey>`	这个注解的功能与 `<selectKey>` 标签完全一致。该注解只能在 `@Insert` 或 `@InsertProvider` 或 `@Update` 或 `@UpdateProvid` 置的生成主键或设置（configuration）属性。属性：`statement` 以字符串数组形式指定将会被执行的 SQL 语句， `keyProperty` 语句应被在插入语句的之前还是之后执行。 `resultType` 则指定 `keyProperty` 的 Java 类型。 `statementType` 则用于选择语句类 `CallableStatement`。默认值是 `PREPARED`。 The `databaseId`(Available since 3.5.5), in case there is a configured `Database` the current one. If found with and without the `databaseId` the latter will be discarded.
`@ResultMap`	方法	N/A	这个注解为 `@Select` 或者 `@SelectProvider` 注解指定 XML 映射中 `<resultMap>` 元素的 id。这使得注解的 select 可以复用已 被此注解覆盖。
`@ResultType`	方法	N/A	在使用了结果处理器的情况下，需要使用此注解。由于此时的返回类型为 void，所以 Mybatis 需要有一种方法来判断每一行返回 定了，就不需要使用其它注解了。否则就需要使用此注解。比如，如果一个标注了 @Select 的方法想要使用结果处理器，那么它的 效。
`@Flush`	方法	N/A	如果使用了这个注解，定义在 Mapper 接口中的方法就能够调用 `SqlSession#flushStatements()` 方法。（Mybatis 3.3 以上可

4. 综合实验

- 编写学生管理系统持久层代码

5. 作业实践

- 配置 MyBatis
- 实现学生表增删改查

二、MyBatis 高级

1. 课程目标

- 掌握 MyBatis 配置文件
- 掌握 MyBatis 动态 SQL
- 掌握 MyBatis 高级映射
- 掌握 MyBatis 分页插件
- 了解 MyBatis 缓存

2. 知识架构树

- MyBatis 配置文件
- MyBatis 常见问题
- MyBatis 动态SQL
- MyBatis 高级映射
- MyBatis 存储过程
- MyBatis 配置日志
- MyBatis 配置缓存
- MyBatis 分页插件

3. 理论知识

3.1 MyBatis 配置文件

3.1.1 数据源配置

```
<!--
    数据库环境的集合，可以配置多个数据源，通过default配置默认数据源
    项目开发过程中有开发环境和生产环境，在开发过程中使用开发环境，项目上线时修改default可以切换为生产环境
    default：默认使用的数据库环境
-->
<environments default="mysql">
    <!--
        单个的数据库环境
        id：当前数据库的标识
    -->
    <!--MySQL环境：-->
    <environment id="mysql">
        <!--事务管理器，使用JDBC的事务管理器-->
        <transactionManager type="JDBC"></transactionManager>
        <!--数据源配置，使用连接池-->
        <dataSource type="POOLED">
            <!--驱动、url、账号、密码-->
            <property name="driver" value="${mysql.driver}"/>
            <property name="url" value="${mysql.url}"/>
            <property name="username" value="${mysql.username}"/>
            <property name="password" value="${mysql.password}"/>
        </dataSource>
    </environment>

    <!--Oracle环境-->
    <environment id="oracle">
        <!--事务管理器，使用JDBC的事务管理器-->
        <transactionManager type="JDBC"></transactionManager>
        <!--数据源配置，使用连接池-->
        <dataSource type="POOLED">
            <!--驱动、url、账号、密码-->
            <property name="driver" value="${oracle.driver}"/>
            <property name="url" value="${oracle.url}"/>
            <property name="username" value="${oracle.username}"/>
            <property name="password" value="${oracle.password}"/>
        </dataSource>
    </environment>
</environments>
```

3.1.2 自定义类型起别名

> 自定义类型在映射文件中需要写类的全限定名，而Java内置类型则不需要，是因为MyBatis底层分装了这些类型，我们可以给自定义类型起名字。

```
<typeAliases>
    <!-- 给单个实体类起别名 -->
    <typeAlias type="com.soft.entity.Student" alias="student"/>
</typeAliases>
```

```
<typeAliases>
    <!-- 扫描实体类路径给所有的实体类起别名，名称为类名或者类名首字母小写 -->
    <package name="com.soft.entity"/>
</typeAliases>
```

3.1.3 扫描映射文件

```xml
<!--配置映射文件-->
<mappers>
    <!--
        name：xml文件所在目录，使用/隔开，因为xml是资源文件，不是java文件。所以找文件值需要通过目录找，而不是包名
        1、xml和dao接口必须在同一个目录下
        2、xml的文件名和dao接口的文件名必须一致包含大小写
    -->
    <package name="com.soft.mapper"/>
</mappers>
```

3.2 MyBatis 常见问题

3.2.1 无效列类型：1111

> Oracle 数据库可能会出现的问题。输入参数为 null 会发生的问题（MyBatis 空指针异常）
>
> `org.apache.ibatis.type.TypeException: Error setting null for parameter #1 with JdbcType OTHER . Try setting a different JdbcType for this parameter or a different jdbcTypeForNull configuration property.`

```
org.apache.ibatis.exceptions.PersistenceException:
### Error querying database.  Cause: org.apache.ibatis.type.TypeException: Could not set parameters for mapping: ParameterMapping{property='no', mode=IN, javaType=class java.lang.Object, jdbcType=null, numericScale=null, resultMapId='null', jdbcTypeName='null', expression='null'}. Cause: org.apache.ibatis.type.TypeException: Error setting null for parameter #1 with JdbcType OTHER . Try setting a different JdbcType for this parameter or a different jdbcTypeForNull configuration property. Cause: java.sql.SQLException: 无效的列类型: 1111
### The error may exist in com/soft/mapper/CrudMapper.xml
### The error may involve defaultParameterMap
### The error occurred while setting parameters
### SQL: select no, name, sex, tel, address, class clazz, del_flg delFlg         from tb_student where no = ?
### Cause: org.apache.ibatis.type.TypeException: Could not set parameters for mapping: ParameterMapping{property='no', mode=IN, javaType=class java.lang.Object, jdbcType=null, numericScale=null, resultMapId='null', jdbcTypeName='null', expression='null'}. Cause: org.apache.ibatis.type.TypeException: Error setting null for parameter #1 with JdbcType OTHER . Try setting a different JdbcType for this parameter or a different jdbcTypeForNull configuration property. Cause: java.sql.SQLException: 无效的列类型: 1111

    at org.apache.ibatis.exceptions.ExceptionFactory.wrapException(ExceptionFactory.java:30)
    at org.apache.ibatis.session.defaults.DefaultSqlSession.selectList(DefaultSqlSession.java:153)
    at org.apache.ibatis.session.defaults.DefaultSqlSession.selectList(DefaultSqlSession.java:145)
    at org.apache.ibatis.session.defaults.DefaultSqlSession.selectList(DefaultSqlSession.java:140)
    at org.apache.ibatis.session.defaults.DefaultSqlSession.selectOne(DefaultSqlSession.java:76)
    at org.apache.ibatis.binding.MapperMethod.execute(MapperMethod.java:87)
    at org.apache.ibatis.binding.MapperProxy$PlainMethodInvoker.invoke(MapperProxy.java:145)
    at org.apache.ibatis.binding.MapperProxy.invoke(MapperProxy.java:86)
    at com.sun.proxy.$Proxy8.queryForObject(Unknown Source)
    at com.soft.test.CrudTest.queryForObjectTest(CrudTest.java:32)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:498)
    at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:50)
    at org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.java:12)
    at org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod.java:47)
    at org.junit.internal.runners.statements.InvokeMethod.evaluate(InvokeMethod.java:17)
    at org.junit.runners.ParentRunner.runLeaf(ParentRunner.java:325)
    at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:78)
    at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:57)
    at org.junit.runners.ParentRunner$3.run(ParentRunner.java:290)
    at org.junit.runners.ParentRunner$1.schedule(ParentRunner.java:71)
    at org.junit.runners.ParentRunner.runChildren(ParentRunner.java:288)
    at org.junit.runners.ParentRunner.access$000(ParentRunner.java:58)
    at org.junit.runners.ParentRunner$2.evaluate(ParentRunner.java:268)
    at org.junit.runners.ParentRunner.run(ParentRunner.java:363)
    at org.junit.runner.JUnitCore.run(JUnitCore.java:137)
    at com.intellij.junit4.JUnit4IdeaTestRunner.startRunnerWithArgs(JUnit4IdeaTestRunner.java:68)
    at com.intellij.rt.junit.IdeaTestRunner$Repeater.startRunnerWithArgs(IdeaTestRunner.java:33)
    at com.intellij.rt.junit.JUnitStarter.prepareStreamsAndStart(JUnitStarter.java:230)
    at com.intellij.rt.junit.JUnitStarter.main(JUnitStarter.java:58)
Caused by: org.apache.ibatis.type.TypeException: Could not set parameters for mapping: ParameterMapping{property='no', mode=IN, javaType=class java.lang.Object, jdbcType=null, numericScale=null, resultMapId='null', jdbcTypeName='null', expression='null'}. Cause: org.apache.ibatis.type.TypeException: Error setting null for parameter #1 with JdbcType OTHER . Try setting a different JdbcType for this parameter or a different jdbcTypeForNull configuration property. Cause: java.sql.SQLException: 无效的列类型: 1111
    at org.apache.ibatis.scripting.defaults.DefaultParameterHandler.setParameters(DefaultParameterHandler.java:89)
```

```
    at org.apache.ibatis.executor.statement.PreparedStatementHandler.parameterize(PreparedStatementHandler.java:94)
    at org.apache.ibatis.executor.statement.RoutingStatementHandler.parameterize(RoutingStatementHandler.java:64)
    at org.apache.ibatis.executor.SimpleExecutor.prepareStatement(SimpleExecutor.java:88)
    at org.apache.ibatis.executor.SimpleExecutor.doQuery(SimpleExecutor.java:62)
    at org.apache.ibatis.executor.BaseExecutor.queryFromDatabase(BaseExecutor.java:325)
    at org.apache.ibatis.executor.BaseExecutor.query(BaseExecutor.java:156)
    at org.apache.ibatis.executor.CachingExecutor.query(CachingExecutor.java:109)
    at org.apache.ibatis.executor.CachingExecutor.query(CachingExecutor.java:89)
    at org.apache.ibatis.session.defaults.DefaultSqlSession.selectList(DefaultSqlSession.java:151)
    ... 30 more
Caused by: org.apache.ibatis.type.TypeException: Error setting null for parameter #1 with JdbcType OTHER . Try setting a different JdbcType for this parameter or a different jdbcTypeForNull configuration property. Cause: java.sql.SQLException: 无效的列类型: 1111
    at org.apache.ibatis.type.BaseTypeHandler.setParameter(BaseTypeHandler.java:67)
    at org.apache.ibatis.scripting.defaults.DefaultParameterHandler.setParameters(DefaultParameterHandler.java:87)
    ... 39 more
Caused by: java.sql.SQLException: 无效的列类型: 1111
    at oracle.jdbc.driver.OracleStatement.getInternalType(OracleStatement.java:3916)
    at oracle.jdbc.driver.OraclePreparedStatement.setNullCritical(OraclePreparedStatement.java:4541)
    at oracle.jdbc.driver.OraclePreparedStatement.setNull(OraclePreparedStatement.java:4523)
    at oracle.jdbc.driver.OraclePreparedStatementWrapper.setNull(OraclePreparedStatementWrapper.java:1281)
    at org.apache.ibatis.type.BaseTypeHandler.setParameter(BaseTypeHandler.java:65)
    ... 40 more
```

```
<!--
    给映射文件的入参添加jdbcType：org.apache.ibatis.type.JdbcType枚举类
    jdbcType的值要全大写
-->
<select id="queryForObject2" resultType="student">
    select * from student where sno = #{sno, jdbcType=VARCHAR} and pass = #{pass, jdbcType=VARCHAR}
</select>
```

Jdbc数据类型	Java数据类型	说明
CHAR	String	字符串类型
VARCHAR	String	字符串类型
NUMBERIC	BigDecimal	大数据类型
BOOLEAN	boolean	布尔类型
INTEGER	int	Integer整形
DOUBLE	double	Double双精度类型
SMALLINT	short	Short短整形
VARBINARY	byte[]	二进制数组
DATE	java.sql.Date	日期类型
TIME	java.sql.Time	时间类型
TIMESTAMP	java.sql.Timestamp	时间戳类型

3.2.2 返回值为Map映射为空

> 主要解决数据库中字段为空，没有办法映射到Map集合中的问题

```
<settings>
    <setting name="callSettersOnNulls" value="true"/>
</settings>
```

3.2.3 #和$的区别

> `#{}`：可以理解为`PreparedStatement`，可以预编译，能有效的防止SQL注入。主要操作数据，参数一般为页面输入的值； `${}`：可以理解为`Statement`，不会预编译，不能防止SQL注入。主要操作数据库对象，比如根据不同的字段排序、表、视图等；

3.3 MyBatis 动态SQL

> 根据不同的需求，SQL的结构或者SQL的条件是不同的。

常见标签：

```
<where></where>
    代替SQL语句中的 where 关键字，会自动去掉前面多余的连接符（and、or）
```

```
<if test=""></if>
    条件判断，test属性为判断入参的表达式，多个表达式之间使用and、or拼接，表达式不用添加#{}

<trim prefix="" suffix="" suffixOverrides="" prefixOverrides=""></trim>
    prefix：SQL拼接的前缀
    suffix：SQL拼接的后缀
    prefixOverrides：去除sql语句前面的关键字或者字符，该关键字或者字符由prefixOverrides属性指定，假设该属性指定为"AND"，当sql语句的开头为"AND"，trim标签将会去除该"AND"
    suffixOverrides：去除sql语句后面的关键字或者字符，该关键字或者字符由suffixOverrides属性指定

<set></set>
    代替SQL语句中的set关键字，会自动去掉多余的逗号

<choose>
    <when test=""></when>
    <when test=""></when>
    <otherwise></otherwise>
</choose>
    选择，满足一个条件后其他条件不再判断，可以理解为if - else if - else

<foreach collection="" item="" index="" open="" close="" separator=""></foreach>
    遍历，多用于批量执行

    collection：遍历集合，
        1、如果入参是list、数组、map类型的可以直接写list、collection、array、map；
        2、或者按照参数的索引位置，arg0、arg1
    item：集合中的每一个对象
    index：下标
    open：循环以某个字符开头
    close：循环以某个字符结尾
    separator：循环内容之间的分隔符，会自动去掉多余的分隔符

<sql id=""></sql>
    定义SQL片段，多用于提取重复的SQL代码

<include refid=""/>
    引入SQL片段，refid是<sql>标签的id值
```

3.3.1 动态查询

```
/**
 * 根据学生的学号、姓名、班级、电话等信息查询
 * 可以根据以上的任意一个或者多个条件进行查询
 * 查询的时候学号、姓名、班级、电话等信息可能都知道，也可能知道一部分
 */
List<Student> query(Student student);
```

```
<!-- 添加恒等式搭配if判断 -->
<select id="query" resultType="Student">
    select
    no, name, sex, tel, address, class clazz, del_flg delFlg
    from tb_student
    where 1 = 1
    <!-- 测试test中的表达式是否满足要求 -->
    <if test="no != null">
        and no = #{no}
    </if>
    <if test="name != null">
        and name = #{name}
    </if>
    <if test="tel != null">
        and tel = #{tel}
    </if>
    <if test="clazz != null">
        and class = #{clazz}
    </if>
</select>
```

```
<!-- where标签搭配if判断 -->
<select id="query" resultType="Student">
    select
    no, name, sex, tel, address, class clazz, del_flg delFlg
    from tb_student
    <!--把where关键字替换成标签，可以去掉多余拼接符-->
    <where>
        <!-- 测试test中的表达式是否满足要求 -->
        <if test="no != null">
            and no = #{no}
        </if>
        <if test="name != null">
            and name = #{name}
```

```
        </if>
        <if test="tel != null">
            and tel = #{tel}
        </if>
        <if test="clazz != null">
            and class = #{clazz}
        </if>
    </where>
</select>
```

```
<!-- trim标签搭配if判断 -->
<select id="query" resultType="Student">
    select
    no, name, sex, tel, address, class clazz, del_flg delFlg
    from tb_student
    where 1 = 1
    <!--目标只需要把多余的关键字去掉即可-->
    <trim suffixOverrides="and">
        <!-- 测试test中的表达式是否满足要求 -->
        <if test="no != null">
            no = #{no} and
        </if>
        <if test="name != null">
            name = #{name} and
        </if>
        <if test="tel != null">
            tel = #{tel} and
        </if>
        <if test="clazz != null">
            class = #{clazz} and
        </if>
    </trim>
</select>
```

3.3.2 动态插入

```
/**
 * 根据输入的学生信息进行添加，输入信息，可能有也可能没有
 */
int add(Student student);
```

```
<insert id="add">
    insert into tb_student
    <trim prefix="(" suffix=")" prefixOverrides=“, ”>
        <if test="name != null and name !=''">name,</if>
        <if test="sex != null and sex !=''">sex,</if>
        <if test="tel != null and tel !=''">tel,</if>
        <if test="address != null and address !=''">address,</if>
        <if test="clazz != null and clazz !=''">class,</if>
        <if test="delFlg != null and delFlg !=''">del_flg,</if>
    </trim>
    values
    <trim prefix="(" suffix=")" prefixOverrides=“, ”>
        <if test="name != null and name !=''">#{name},</if>
        <if test="sex != null and sex !=''">#{sex},</if>
        <if test="tel != null and tel !=''">#{tel},</if>
        <if test="address != null and address !=''">#{address},</if>
        <if test="clazz != null and clazz !=''">#{class},</if>
        <if test="delFlg != null and delFlg !=''">#{del_flg},</if>
    </trim>
</insert>
```

3.3.3 动态更新

```
/**
 * 根据输入的学生信息进行更新，输入信息，可能有也可能没有
 */
int edit(Student student);
```

```
<update id="edit">
    update tb_student
    <set>
        <if test="name != null">
            name = #{name},
        </if>
        <if test="tel != null">
            tel = #{tel},
        </if>
        <if test="clazz != null">
            class = #{clazz},
        </if>
```

```
    </set>
    where
    no = #{no}
</update>
```

3.3.4 选择

```
/**
 * 根据不同的需求（情况）对SQL进行不同的操作
 * 老师（flg = 1）：
 *      修改学生电话、班级等信息
 * 学生（flg = 2）：
 *      修改电话
 */
int choose(@Param("student") Student student, @Param("flg") int flg);
```

```
<update id="choose">
    update tb_student
    <set>
        <choose>
            <when test="flg == 1">
                <if test="student.tel != null and student.tel != ''">
                    tel = #{student.tel},
                </if>
                <if test="student.address != null and student.address != ''">
                    address = #{student.address},</if>
                <if test="student.clazz != null and student.clazz != ''">
                    class = #{student.clazz},
                </if>
                <if test="student.delFlg != null and student.delFlg != ''"
                    del_flg = #{student.delFlg},
                </if>
            </when>

            <when test="flg == 2">
                <if test="student.tel != null and student.tel != ''">
                    tel = #{student.tel},
                </if>
            </when>

            <otherwise>
                <if test="student.mno !=null ">mno = #{student.mno}</if>
            </otherwise>
        </choose>
    </set>
    where no = #{student.no}
</update>
```

3.3.5 批量删除

```
public int delForList(List<Student> list);
```

```
<!--
    批量删除的SQL：逻辑删除修改删除的标识
        in中的学号#{no}出现了多次，多次出现同一个内容需要用到循环
        循环体是什么？
            #{no}
        循环体之间的分隔符？
            ,
        for(Student stu : list)

    update tb_student set del_flg where no in (1, 2, 3, 4, 5, 6, 7, 8)
-->
<update id="del">
    update tb_student
    set del_flg = '0'
    where no in
    <foreach collection="list" item="stu" separator="," open="(" close=")">
        #{stu.no}
    </foreach>
</update>
```

3.3.6 批量插入

```
public int addForList(List<Student> stu);
```

> Oracle的批量插入，使用表的拷贝语句，s_c1, s_c2, s_c3应该是从另一张表中获取。
>
> 但是在实际的业务需求中，s_c1, s_c2, s_c3等数据是页面获取的，所以就不存在实际的表。Oracle的语法又要求from后面必须跟表名保证SQL的完整性，所以from后面要跟虚拟表dual。

> `select 数据1, 数据2, 数据3, ... from dual` 只能表示一条数据，想要实现多条数据，语句中间使用 `union all` 拼接即可。

```
insert into tableName(c1, c2, c3, ...)
(
    select s_c1, s_c2, s_c3, ... from subTableName
)


insert into tableName(c1, c2, c3, ...)
(
    select 数据1, 数据2, 数据3, ... from dual
    union all
    select 数据1, 数据2, 数据3, ... from dual
)
```

Oracle批量插入，带序列

```
<!--
    批量插入Oracle(不带序列)分析：
        1、表的拷贝语句，从一张表中，把数据查询出来，添加到表中
            insert into tb_student (
                select * from tb_student_bk
            )
        2、【select * from tb_student_bk】查询出来的是结果集
            以上的需求就可以转换为将一个结果集直接添加到表中
            insert into tb_student (
                结果集
            )
        3、结果集的构成：
            3.1 查询具体的表
                select no, sex, tel from tb_student
            3.2 从虚拟表中查询固定的值(数据并不是只能存在表中，还可以固定写死)
                select '1', '男' from dual
                select '2', '女' from dual
                select '3', '男' from dual

            3.3 结果集的合并，union all
                select '1', '男' from dual
                union all
                select '2', '女' from dual
                union all
                select '3', '男' from dual

        4、综上
            insert into tb_student (
                select '1', '男' from dual
                union all
                select '2', '女' from dual
                union all
                select '3', '男' from dual
            )

            4.1 结果集的分析
                select #{no}, #{sex} from dual
                union all
                select #{no}, #{sex} from dual
                union all
                select #{no}, #{sex} from dual

                循环体：
                    select #{no}, #{sex} from dual
                分隔符：
                    union all
-->
<insert id="add">
    insert into tb_student
        (no, name, sex, tel, address, class, del_flg)
        (
            select seq_student.nextval, t.* from (
                <foreach collection="list" item="stu" separator="union all">
                    select
                        #{stu.name}, #{stu.sex}, #{stu.tel}, #{stu.address}, #{stu.clazz}, #{stu.delFlg}
                    from
                        dual
                </foreach>
            ) t
        )
</insert>
```

Oracle批量插入，不带序列

```
<insert id="add">
    insert into tb_student
        (no, name, sex, tel, address, class, del_flg)
    (
        <foreach collection="list" item="stu" separator="union all">
            select
                #{stu.no}, #{stu.name}, #{stu.sex}, #{stu.tel}, #{stu.address}, #{stu.clazz}, #{stu.delFlg}
            from
                dual
        </foreach>
    )
</insert>
```

MySQL批量插入

```
insert into tableName(c1, c2, c3, ...) values
(v1, v2, v3, ...),
(v1, v2, v3, ...),
(v1, v2, v3, ...),
(v1, v2, v3, ...),
....
```

```
<!--
    批量插入MySQL分析:
        insert into tb_student values
            ('', '', '', '', ''),
            ('', '', '', '', ''),
            ('', '', '', '', ''),
            ('', '', '', '', ''),
            ('', '', '', '', '')
        循环体:
            ('', '', '', '', '')
        分隔符:
            ,
-->
<insert id="add">
    insert into tb_student
        (name, sex, tel, address, class, del_flg)
    values
        <foreach collection="list" item="stu" separator=",">
            (#{stu.name}, #{stu.sex}, #{stu.tel}, #{stu.address}, #{stu.clazz}, #{stu.delFlg})
        </foreach>
</insert>
```

3.3.7 SQL片段

> 将出现频次高的SQL内容，提取出来，供需要者调用。减少代码的开发和修改。

- 单个映射文件使用sql片段

```
<?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">
<mapper namespace="com.soft.mapper.QueryMapper">
    <sql id="query_sql">
        sno, sname, sex, age, tel, native_place home, class clazz, pass, head, del
    </sql>

    <select id="queryForList" resultType="student">
        select <include refid="query_sql"/> from student
    </select>
</mapper>
```

- 多个映射文件使用sql片段

1、创建全局的映射文件

```
<?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">
<!--全局的文件没有接口映射，所以namespace可以随意编写。-->
<mapper namespace="global">
    <sql id="query_sql">sno, sname, sex, age, tel, native_place home, class clazz, pass, head, del</sql>
</mapper>
```

2、由于全局的映射文件没有接口映射，所以需要单独配置管理全局的映射文件

```xml
<!--配置映射文件-->
<mappers>
    <mapper resource="com/soft/mapper/global.xml"/>
    <package name="com.soft.mapper"/>
</mappers>
```

3、引入SQL片段，通过全局映射文件的`namespace.sql`标签的id

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.soft.mapper.QueryMapper">
    <sql id="query_sql">sno, sname, sex, age, tel, native_place home, class clazz, pass, head, del</sql>
    <select id="queryForList" resultType="student">
        select <include refid="global.query_sql"/> from student
    </select>
</mapper>
```

3.4 MyBatis 高级映射

> 高级映射主要解决字段和属性对应不上和**多表**关联查询的问题。
>
> 表之间的关联关系有：一对一，一对多，那如何描述表之间的关系，就需要用到高级映射了。
>
> 实体类和DB结构是一一对应的。单表的结果对应一个对象。多表的结果怎么对应?

3.4.1 单表映射

> 单表映射主要解决字段和属性不能对应的问题。

实体类

```java
public class Student {
    private String no;
    private String name;
    private String sex;
    private String tel;
    private String address;
    private String clazz;
    private String delFlg;

    // 省略set/get方法
}
```

Dao接口

```java
/**
 * 单表的映射
 */
Student queryForObject(Student student);
```

映射文件

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.soft.dao.ResultMapDao">
    <!--
        id：resultMap的唯一标识
        type：完整的Java类或者别名
    -->
    <resultMap id="rm1" type="student">
        <!--
            column：需要映射的字段名，以sql查询结果的字段名为准
            property：需要把字段映射到实体类对应的属性
            jdbcType：数据库中字段对应的类型
            javaType：java中属性的类型
        -->
        <result column="no" property="no" javaType="String" jdbcType="VARCHAR"/>
        <result column="name" property="name" javaType="String" jdbcType="VARCHAR"/>
        <result column="sex" property="sex" javaType="String" jdbcType="VARCHAR"/>
        <result column="tel" property="tel" javaType="String" jdbcType="VARCHAR"/>
        <result column="address" property="address" javaType="String" jdbcType="VARCHAR"/>
        <result column="class" property="clazz" javaType="String" jdbcType="VARCHAR"/>
        <result column="del_flg" property="delFlg" javaType="String" jdbcType="VARCHAR"/>
    </resultMap>

    <select id="queryForObject" resultMap="rm1">
        select
            no, name, sex, tel, address, class, del_flg
```

```
        from
            tb_student
        where
            no = #{no}
    </select>
</mapper>
```

3.4.2 多表映射：一对一

> 需求：查询学生以及专业信息；以学生为主，专业为辅；

专业表SQL结构

```
create table tb_major(
  mno varchar2(20),
  mname varchar2(100)
);

insert into tb_major(
    select 'M10001', '软件工程' from dual
    union all
    select 'M10002', '土木工程' from dual
    union all
    select 'M10003', '信息工程' from dual
    union all
    select 'M10004', '电竞专业' from dual
    union all
    select 'M10005', '美术' from dual
    union all
    select 'M10006', '体育' from dual
);
```

实体类

```
public class Student {
    private String no;
    private String name;
    private String sex;
    private String tel;
    private String address;
    private String clazz;
    private String delFlg;
    // 专业的对象
    private Major major;

    // 省略set/get方法/toString方法
}
```

```
public class Major {
    private String mno;
    private String mname;

    // 省略set/get方法/toString方法
}
```

Dao接口

```
/**
 * 多表映射：学生和专业
 *  一个学生对应一个专业：一对一的关系
 *
 *  从表结构中，学生表中有专业的专业号
 */
Student queryForObjectWithMajor(Student student);
```

映射文件

```
<?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">
<mapper namespace="com.soft.dao.ResultMapDao">
    <!--
        一个学生对应一个专业
            在学生的角度去维护专业信息
            要在学生的实体类中去体现专业信息
    -->
    <resultMap id="rm2" type="student">
        <!--学生信息-->
        <result column="no" property="no" javaType="String" jdbcType="VARCHAR"/>
        <result column="name" property="name" javaType="String" jdbcType="VARCHAR"/>
```

```
        <result column="sex" property="sex" javaType="String" jdbcType="VARCHAR"/>
        <result column="tel" property="tel" javaType="String" jdbcType="VARCHAR"/>
        <result column="address" property="address" javaType="String" jdbcType="VARCHAR"/>
        <result column="class" property="clazz" javaType="String" jdbcType="VARCHAR"/>
        <result column="del_flg" property="delFlg" javaType="String" jdbcType="VARCHAR"/>

        <!--专业信息-->
        <!--
            association：用于描述一对一的关系
                property：主实体类中的属性
                javaType：当前属性对应的Java中的类型（实体类）
        -->
        <association property="major" javaType="Major">
            <result column="mno" property="mno" javaType="String" jdbcType="VARCHAR"/>
            <result column="mname" property="mname" javaType="String" jdbcType="VARCHAR"/>
        </association>
    </resultMap>
    <select id="queryForObjectWithMajor" resultMap="rm2">
        select
            t1.no, t1.name, t1.sex, t1.tel, t1.address, t1.class, t1.del_flg,
            t2.mno, t2.mname
        from
            tb_student t1
        left join
            tb_major t2
        on t1.mno = t2.mno
        where
            t1.no = #{no}
    </select>
</mapper>
```

3.4.3 多表映射：一对多

> 需求：查询专业对应的学生信息；以专业为主，学生为辅；

实体类

```
public class Student {
    private String no;
    private String name;
    private String sex;
    private String tel;
    private String address;
    private String clazz;
    private String delFlg;

    // 省略set/get方法
}
```

```
public class Major {
    private String mno;
    private String mname;

    // 一个专业很多学生信息
    List<Student> students;
}
```

Dao接口

```
/**
 * 多表映射：专业和学生
 *  一个专业有多个学生：一对多的关系
 */
Major queryForObjectWithStudent(Major major);
```

```
<?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">
<mapper namespace="com.soft.dao.ResultMapDao">
    <!--
        一个专业对应n个学生
            在专业的角度去维护学生信息
            要在专业的实体类中去体现学生的信息
    -->
    <resultMap id="rm3" type="major">
        <!--专业信息-->
        <id column="mno" property="mno" javaType="String" jdbcType="VARCHAR"/>
        <result column="mname" property="mname" javaType="String" jdbcType="VARCHAR"/>

        <!-- 学生信息 -->
        <!--
```

```
            collection：用于维护一对多的关系。如果多的一方数据有重复（每个字段的值都一样）会进行合并。一定要通过id给定主键关系
                property：属性名称
                ofType：集合依赖的泛型的类型
        -->
        <collection property="students" ofType="Student">
            <!--把查询到的结果和实体类的属性进行映射-->
            <id column="no" property="no" javaType="String" jdbcType="VARCHAR"/>
            <result column="name" property="name" javaType="String" jdbcType="VARCHAR"/>
            <result column="sex" property="sex" javaType="String" jdbcType="VARCHAR"/>
            <result column="tel" property="tel" javaType="String" jdbcType="VARCHAR"/>
            <result column="address" property="address" javaType="String" jdbcType="VARCHAR"/>
            <result column="class" property="clazz" javaType="String" jdbcType="VARCHAR"/>
            <result column="del_flg" property="delFlg" javaType="String" jdbcType="VARCHAR"/>
        </collection>
    </resultMap>

    <select id="queryForObjectWithStudent" resultMap="rm3">
        select
            t1.mno, t1.mname,
            t2.no, t2.name, t2.sex, t2.tel, t2.address, t2.class, t2.del_flg
        from
            tb_major t1
        left join
            tb_student t2
        on t1.mno = t1.mno
        where
            t1.mno = #{mno}
    </select>
</mapper>
```

3.5 MyBatis 存储过程

> 存储过程属于数据库编程的范畴。一个业务逻辑中，可能需要多次频繁的调用数据库，会对数据库带来压力。可以将逻辑写到数据库中的存储过程中，通过调用存储过程实现功能，减少了调用数据库的次数，减轻了数据库压力。

```
-- 创建存储过程，通过学号返回电话和姓名
create procedure pro_query_student(
    i_no in varchar2,
    o_tel out varchar2,
    o_name out varchar2
) as
begin
  select
       tel, name
  into
       o_tel, o_name
  from
       tb_student
  where
       no = i_no;
end;
```

```
/**
 *  存储过程特点：存储过程没有返回值，只有输入参数和输出参数
 *  根据以上特点：接口的方法也不需要返回值，存储过程的输出结果会封装到入参中
 */
void queryForObject(Student student);
```

```
<?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">
<mapper namespace="com.soft.dao.ProDao">
    <!--
        调用存储过程，根据学号查询学生的电话、姓名
            输入参数：学号
            输入参数：电话、姓名

        statementType：SQL执行器的类型
            statement：执行普通SQL，不能防止SQL注入
            prepared：执行普通SQL，可以防止SQL注入
            callable：执行存储过程或者函数

        useCache：使用缓存：false是不使用缓存，每次执行的时候都从数据库查询最新的

        mode：指定参数是输入还是输出参数。in和out
    -->
    <select id="queryForObject" useCache="false" statementType="CALLABLE">
        {call pro_query_student(
                #{no,mode=IN,jdbcType=VARCHAR},
```

```
                #{tel,mode=OUT,jdbcType=VARCHAR},
                #{name,mode=OUT,jdbcType=VARCHAR}
        )}
    </select>
</mapper>
```

3.6 MyBatis 配置日志

> 日志就是代码在执行过程中的状态或者结果进行展示。通过日志可以分析代码执行的异常情况和执行状况。日志一般都是系统维护人员去看。

1、导包

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

2、log4j 日志的组成部分

```
1、日志级别：信息详细程度
    由高到低：DEBUG -> INFO -> WARN -> ERROR
    级别越高日志的内容越多

2、日志的输出目的地（Appender）：信息展示的地方
    控制台：org.apache.log4j.ConsoleAppender
    文件：org.apache.log4j.RollingFileAppender、org.apache.log4j.DailyRollingFileAppender
    邮件：org.apache.log4j.net.SMTPAppender
    数据库：org.apache.log4j.jdbc.JDBCAppender

3、日志的布局（layout）：信息的内容格式
    %p：日志级别
    %c：类名
    %m：信息
    %n：换行
    %d：日期{yyyy-mm-dd hh:mm:ss}
```

3、log4j.properties 文件编写，注意文件名固定

```
#####################################################
# 日志就是功能，只要是功能，就需要类和对象，以及方法调用 #
#####################################################

# 日志目的地和全局日志的级别，
# 本文中设定的是全局的DEBUG,，为的是开发过程中方便查看调试信息
# 日志输入目的地设置了4个，分别是控制台，文件，邮件，数据库
# 全局的变量定义：日志级别和日志目的地
log4j.rootLogger=ERROR,console,logFile,jdbc,mail

##################################控制台打印#################################
# appender：目的地，一个appender就是一个目的地
# 配置目的地对象：控制台对象
# Console console = new ConsoleAppender();
log4j.appender.console=org.apache.log4j.ConsoleAppender
# 日志打印的类别，err：错误信息，字体颜色为红色；out：打印信息，
log4j.appender.console.target=System.err
# 自定义当前appender的日志级别
log4j.appender.console.threshold=DEBUG

# 配置控制台信息的布局
log4j.appender.console.layout=org.apache.log4j.PatternLayout
# %p：日志级别
# %c：类名
# %m：信息
# %n：换行
# %d：日期{yyyy-MM-dd hh:mm:ss}
log4j.appender.console.layout.conversionPattern=[%p] %d{yyyy-MM-dd hh:mm:ss} %c : %m%n

################################## 输出到文件 ################################
# 配置目的地对象：文件对象
# log4j.appender.logFile=org.apache.log4j.RollingFileAppender
# 配置文件大小
# log4j.appender.logFile.MaxFileSize=50MB
# file表示文件路径
# 可以是相对路径也可以是绝对路径
# log4j.appender.logFile.File=myLog.log

# 配置文件信息的布局
# log4j.appender.logFile.layout=org.apache.log4j.PatternLayout
# log4j.appender.logFile.layout.conversionPattern=[%p] %d{yyyy-MM-dd hh:mm:ss} %c : %m%n
# ---------------------------------------------------------------------------
```

```
log4j.appender.logFile=org.apache.log4j.DailyRollingFileAppender
log4j.appender.logFile.File=myLog.log
# 配置时间
log4j.appender.logFile.datePattern=yyyy-MM-dd

# 配置文件信息的布局
log4j.appender.logFile.layout=org.apache.log4j.PatternLayout
log4j.appender.logFile.layout.conversionPattern=[%p] %d{yyyy-MM-dd hh:mm:ss} %c : %m%n

################################# 输出到数据库 ################################
# 配置目的地对象：数据库对象
log4j.appender.jdbc=org.apache.log4j.jdbc.JDBCAppender
# 配置数据库相关信息
log4j.appender.jdbc.Driver=oracle.jdbc.driver.OracleDriver
log4j.appender.jdbc.URL=jdbc:oracle:thin:@localhost:1521:orcl
log4j.appender.jdbc.user=
log4j.appender.jdbc.password=
# SQL语句
log4j.appender.jdbc.sql=insert into tb_log values(seq_log.nextval, '%c', '%m', sysdate)
# 自定义当前appender的日志级别
log4j.appender.jdbc.threshold=INFO

# 配置数据库的信息布局
log4j.appender.jdbc.layout=org.apache.log4j.PatternLayout

################################# 输出到邮件 ################################
# java发送邮件需要导jar包：javax.mail
# 配置目的地对象：邮件对象
log4j.appender.mail=org.apache.log4j.net.SMTPAppender
# 配置发送邮件的信息
# 发件人登录邮箱的账号，一般为邮箱
log4j.appender.mail.SMTPUsername=
# 发件人登录邮箱的密码
log4j.appender.mail.SMTPPassword=
# 发件人邮箱
log4j.appender.mail.From=
# 发件人邮箱服务器，smtp.163.com
log4j.appender.mail.SMTPHost=
# 发件人邮箱服务器端口，25
log4j.appender.mail.SMTPPort=
# 邮件主题
log4j.appender.mail.Subject=
# 收件人邮箱
log4j.appender.mail.To=
# 配置邮件的布局，邮箱使用HTML布局
log4j.appender.mail.layout=org.apache.log4j.HTMLLayout

# 单独针对某个包设置打印级别
log4j.logger.com.soft=DEBUG
```

```
create table tb_log(
  no integer primary key,
  className varchar(200),
  msg varchar(200),
  create_time datetime
)

create sequence seq_log
start with 10000
increment by 1
```

4、MyBatis配置文件添加log4j配置

```
<settings>
    <!--固定写法-->
    <setting name="logImpl" value="log4j"/>
</settings>
```

3.7 MyBatis 配置缓存

> 作用：提高查询效率
>
> 缓存中放什么：**经常被使用，且不经常修改，且公共的信息需要放到缓存中。**

（一）一级缓存

> MyBatis的一级缓存默认是开启，不需要手动配置。MyBatis的一级缓存存在于**SqlSession的生命周期中**，在同一个SqlSession中查询时，MyBatis 会把执行的**方法**和**参数**通过算法生成缓存的**键**，将键和查询结果存入一个 Map 对象中。
>
> 如果同一个SqlSession中执行的方法和参数完全一致，那么通过算法会生成相同的**键**，当 Map 缓存对象中已经存在该键值时，则会返回缓存中的对象。也就是说 MyBatis 判断调用的方法和参数不变时会从缓存中取值。

> 如果在调用完查询操作之后，把数据库中的数据修改；再次调用查询操作，参数不变。那么，只会调用一次数据库，此时的结果就跟数据库的结果不对应。如何解决这个问题呢？可以在xml映射文件的查询操作添加【 flushCache="true" 】即可。

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.soft.dao.CacheDao">
    <select id="queryForMap" resultType="map">
        select * from tb_student where no = #{no}
    </select>

    <update id="edit">
        update tb_student set del_flg = 1 where no = #{no}
    </update>
</mapper>
```

```java
public class CacheTest {

    /**
     * 测试在同一个SqlSession中，方法调用2次的结果
     */
    @Test
    public void cache1() throws IOException {
        // 获取配置文件的输入流
        InputStream inputStream = Resources.getResourceAsStream("mybatis-config.xml");

        // 通过SQL的会话工厂的构建类，构建一个SQL的会话工厂（唯一工厂）
        SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);

        // 开启SQL会话，相当于获取了数据库的链接
        SqlSession sqlSession1 = sqlSessionFactory.openSession();

        // 获取实例
        CacheDao mapper = sqlSession1.getMapper(CacheDao.class);

        /*
           同一个方法，相同参数调用，
           第一次会调用数据库获取结果；
           第二次从缓存中共获取结果
        */
        Map<String, Object> map1 = mapper.queryForMap("4");
        Map<String, Object> map2 = mapper.queryForMap("4");

        System.out.println(map1 == map2); // true
    }

    /**
     * 测试在同一个SqlSession中
     * 调用查询方法
     * 调用修改方法
     * 调用查询方法
     */
    @Test
    public void cache1() throws IOException {
        // 获取配置文件的输入流
        InputStream inputStream = Resources.getResourceAsStream("mybatis-config.xml");

        // 通过SQL的会话工厂的构建类，构建一个SQL的会话工厂（唯一工厂）
        SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);

        // 开启SQL会话，相当于获取了数据库的链接
        SqlSession sqlSession1 = sqlSessionFactory.openSession();

        // 获取实例
        CacheDao mapper = sqlSession1.getMapper(CacheDao.class);

        /*
           1、调用查询方法，从数据库中获取，并存放缓存中
           2、调用修改方法会刷新缓存
           3、调用查询方法，从数据库中获取，并存放缓存中
        */
        Map<String, Object> map1 = mapper.queryForMap("4");

        int count = mapper.edit("6");
        System.out.println("修改结果：" + count)

        Map<String, Object> map2 = mapper.queryForMap("4");

        System.out.println(map1 == map2); // false
    }

```

```
    /**
     * 同一个SqlSessionFactory，不同SqlSession
     */
    @Test
    public void cache3() throws IOException {
        // 获取配置文件的输入流
        InputStream inputStream = Resources.getResourceAsStream("mybatis-config.xml");

        // 通过SQL的会话工厂的构建类，构建一个SQL的会话工厂（唯一工厂）
        SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);

        // SqlSession会话1
        SqlSession sqlSession1 = sqlSessionFactory.openSession();

        // 获取实例
        CacheDao mapper1 = sqlSession1.getMapper(CacheDao.class);
        // 调用查询方法
        Map<String, Object> map1 = mapper1.queryForMap("4");

        System.out.println("----------------------------------------------");
        // SqlSession会话2
        SqlSession sqlSession2 = sqlSessionFactory.openSession();

        // 获取实例
        CacheDao mapper2 = sqlSession2.getMapper(CacheDao.class);
        // 调用查询方法
        Map<String, Object> map2 = mapper2.queryForMap("4");

        System.out.println(map1 == map2); // false
    }
}
```

（二）二级缓存

> MyBatis 一级缓存最大的共享范围就是同一个SqlSession内部，那么如果多个 SqlSession 需要共享缓存，则需要使用二级缓存。当二级缓存开启后，同一个命名空间(namespace) 所有的操作语句，都影响着一个**共同的 cache**，也就是二级缓存被多个 SqlSession 共享，是一个**全局的变量**。当开启缓存后，数据的查询执行的流程就是 二级缓存 => 一级缓存 => 数据库。
>
> **实现二级缓存的时候，MyBatis要求返回的POJO（实体类）必须是可序列化的。**开启二级缓存的条件也是比较简单，通过直接在 MyBatis 配置文件配置，还需要在映射文件中添加`<cache>`标签。

```
<settings>
    <setting name="cacheEnabled" value="true"/>
</settings>
```

```
<?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">
<mapper namespace="com.soft.dao.CacheDao">
    <cache/>

    <select id="queryForMap" resultType="map">
        select * from tb_student where no = #{no}
    </select>

    <update id="edit">
        update tb_student set del_flg = 1 where no = #{no}
    </update>
</mapper>
```

```
public class CacheTest {
    /**
     * 同一个SqlSessionFactory，不同SqlSession
     */
    @Test
    public void cache3() throws IOException {
        // 获取配置文件的输入流
        InputStream inputStream = Resources.getResourceAsStream("mybatis-config.xml");

        // 通过SQL的会话工厂的构建类，构建一个SQL的会话工厂（唯一工厂）
        SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);

        // SqlSession会话1
        SqlSession sqlSession1 = sqlSessionFactory.openSession();

        // 获取实例
        CacheDao mapper1 = sqlSession1.getMapper(CacheDao.class);
        // 调用查询方法
        Map<String, Object> map1 = mapper1.queryForMap("4");
        // SqlSession使用之后一定要关闭会话
        sqlSession1.close();
```

```
        System.out.println("----------------------------------------------");
        // SqlSession会话2
        SqlSession sqlSession2 = sqlSessionFactory.openSession();

        // 获取实例
        CacheDao mapper2 = sqlSession2.getMapper(CacheDao.class);
        // 调用查询方法
        Map<String, Object> map2 = mapper2.queryForMap("4");
        // SqlSession使用之后一定要关闭会话
        sqlSession1.close();

        System.out.println(map1 == map2); // false
    }
}
```

（三）缓存使用

```
映射语句文件中的所有SELECT语句将会被缓存。
映射语句文件中的所有时INSERT、UPDATE、DELETE语句会刷新缓存。
缓存会使用LeastRecentlyUsed(LRU，最近最少使用的）算法来收回。
根据时间表（如noFlushInterval，没有刷新间隔），缓存不会以任何时间顺序来刷新。
缓存会存储集合或对象（无论查询方法返回什么类型的值）的1024个引用。
缓存会被视为read/write（可读 / 可写）的，意味着对象检索不是共享的，而且可以安全地被调用者修改，而不干扰其他调用者或线程所做的潜在修改。
```

缓存的执行流程：

```
当用户发送一个查询请求：
    判断二级缓存是否存在：
        存在：从缓存中读取
        不存在：
            判断一级缓存中是否存在数据
                存在：从缓存中读取
                不存在：发送查询语句到数据库查询。

一级缓存什么时候可以当作同一个查询：
    一个SqlSeession可以看作一个查询；
    一旦当前有增删改操作时，针对同一个mybatis会默认当作两个查询。
```

3.8 MyBatis 分页插件

> MyBatis 分页插件 **PageHelper** 可以让程序员专注 SQL 的编写，不用考虑不同数据库的分页语句，通过PageHelper可以自动识别数据库种类，进行分页语句的生成执行。
>
> 官网：https://pagehelper.github.io/

1、导包

```
<dependency>
  <groupId>com.github.pagehelper</groupId>
  <artifactId>pagehelper</artifactId>
  <version>5.3.1</version>
</dependency>
```

2、MyBatis配置文件中，添加分页插件

```
<plugins>
    <!-- com.github.pagehelper为PageHelper类所在包名 -->
    <plugin interceptor="com.github.pagehelper.PageInterceptor"/>
</plugins>
```

3、编写接口映射文件

```
package com.soft.dao;

import java.util.List;
import java.util.Map;

public interface PageDao {

    List<Map<String, String>> query();
}
```

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.soft.dao.PageDao">
    <select id="query" resultType="map">
        select * from tb_student order by sno
    </select>
</mapper>
```

4、编写测试类

```java
package com.soft.test;

import com.github.pagehelper.PageHelper;
import com.github.pagehelper.PageInfo;
import com.soft.dao.PageDao;
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 org.apache.log4j.Logger;
import org.junit.BeforeClass;
import org.junit.Test;

import java.io.IOException;
import java.io.InputStream;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;

public class PageTest {
    static SqlSession sqlSession;

    @BeforeClass
    public static void loadXML() throws IOException {
        // 获取配置文件的输入流
        InputStream inputStream = Resources.getResourceAsStream("mybatis.xml");
        // 使用工厂的创建类，根据配置文件的流生成SqlSessionFactory
        SqlSessionFactory ssf = new SqlSessionFactoryBuilder().build(inputStream);
        // 打开session
        sqlSession = ssf.openSession(true);
    }

    @Test
    public void page1() {
        PageDao pageDao = sqlSession.getMapper(PageDao.class);
        /*
            配置数据的开始位置和每页显示条数
            一定要紧紧的挨着查询操作的代码
         */

        // 每页显示条数
        int limit = 5;
        // 第几页
        int pageNum = 1;
        // 偏移量，数据的开始位置
        int offset = (pageNum - 1) * limit;

        // 分页代码，一定要紧紧的挨着查询操作的代码
        PageHelper.offsetPage(offset, limit);
        // 查询功能
        List<Map<String, String>> query = pageDao.query();

        // 分页的详细信息
        PageInfo pageInfo = new PageInfo(query);

        System.out.println(pageInfo);

        // 获取分页后的数据
        List list = pageInfo.getList();

        list.forEach(map -> {
            System.out.println(map);
        });
    }

    @Test
    public void page2() {
        PageDao pageDao = sqlSession.getMapper(PageDao.class);
        /*
            配置数据的开始位置和每页显示条数
```

```
            一定要紧紧的挨着查询操作的代码
         */
        int pageNum = 5; // 页码
        int pageSize = 5; // 每页显示条数

        // 分页代码，一定要紧紧的挨着查询操作的代码
        PageHelper.startPage(pageNum, pageSizes);
        // 查询功能
        List<Map<String, String>> query = pageDao.query();

        // 分页的详细信息
        PageInfo pageInfo = new PageInfo(query);

        System.out.println(pageInfo);

        // 获取分页后的数据
        List list = pageInfo.getList();

        list.forEach(map -> {
            System.out.println(map);
        });
    }
}
```

4. 综合实验

- 完善学生管理系统

5. 作业实践

- 优化配置文件
- 使用动态 SQL 实现多条件查询
- 优化映射文件，使用高级映射
- 实现 MyBatis 分页

三、Spring 和 MyBatis 整合

> Spring 和 MyBatis整合关键在于决定原来MyBatis管理的文件交给谁管理?
>
> 1、由于 Spring 管理起来更为合理，所以把数据源的管理和接口的映射文件交给spring管理。 2、MyBatis的核心是SqlSessionFactory，这个核心对象要交给 Spring 管理

```
Spring：轻量级框架（容器）
    1、管理控制层（Controller）、业务层（Service）、持久层（Mapper）对象创建和关系维护
    2、管理数据库相关（数据库链接、数据库连接池、事务）
    3、AOP切面管理

MyBatis：持久层框架
    1、管理数据库相关（数据库链接、数据库连接池、事务）  ==>  交给Spring管理
    2、管理持久层接口和映射文件  ==>  自己管理  ==>  通过Spring代管理用的还MyBatis核心
    3、缓存、日志、分页插件  ==>  自己管理  ==>  通过Spring代管理用的还MyBatis核心
```

1、导包

```
<!--Spring相关依赖-->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.2.9.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
    <version>5.2.9.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>5.2.9.RELEASE</version>
</dependency>
<!--mybatis依赖-->
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis</artifactId>
    <version>3.5.3</version>
</dependency>
<!--spring和mybatis集成的依赖-->
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis-spring</artifactId>
    <version>2.0.6</version>
</dependency>
```

```
<!--数据库驱动、log4j等-->
```

2、Spring持久层的配置文件中配置MyBatis，同时可以删除持久层接口对应的实体类，MyBatis是不需要实体类的；

如果有需要也可以编写mybatis的配置文件，比如实体类起别名等。

mybatis-config.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>
    <settings>
        <setting name="callSettersOnNulls" value="true"/>
        <setting name="logImpl" value="log4j"/>
    </settings>

    <!--给实体类起别名，方便映射文件设置对象类型-->
    <typeAliases>
        <!--
            扫描实体类所在的包，别名叫做类名或者类名的首字母小写
            Student student
        -->
        <package name="com.soft.entity"/>
    </typeAliases>
</configuration>
```

spring-dao.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/tx
        https://www.springframework.org/schema/tx/spring-tx.xsd
        http://www.springframework.org/schema/aop
        https://www.springframework.org/schema/aop/spring-aop.xsd">

    <!-- 驱动、url、账号、密码 -->
    <!--加载properties文件-->
    <context:property-placeholder location="jdbc.properties"/>
    <!-- 数据库连接池 -->
    <!--数据源-->
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
        <property name="driverClassName" value="${jdbc.driver}"/>
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.uname}"/>
        <property name="password" value="${jdbc.pwd}"/>
    </bean>

    <!--引入jdbc的事务管理器-->
    <!-- 可以理解为增强类 -->
    <bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <!--开启事务注解的开关-->
    <tx:annotation-driven transaction-manager="transactionManager" />

    <!-- 配置mybatis -->
    <bean id="sessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
        <!--加载mybatis配置文件-->
        <property name="configLocation" value="classpath:mybatis.xml"/>
        <!--加载数据源-->
        <property name="dataSource" ref="dataSource"/>
        <!--扫描映射文件所在包-->
        <property name="mapperLocations" value="classpath:com/soft/dao/*.xml"/>
        <!--实体类起别名-->
        <property name="typeAliasesPackage" value="com.soft.entity"/>
        <property name="configurationProperties">
            <props>
                <!--map映射为空配置-->
                <prop key="callSettersOnNulls">true</prop>
            </props>
        </property>
```

```
    </bean>

    <!--扫描接口所在包-->
    <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
        <!--扫描接口和映射文件对应-->
        <property name="basePackage" value="com.soft.dao"/>
        <!--和sessionFactory关联-->
        <property name="sqlSessionFactoryBeanName" value="sessionFactory"/>
    </bean>
</beans>
```

3、编写接口映射文件

4、测试，和Spring的测试方式一样