

介绍

MyBatis是一个优秀的基于Java的持久层框架，它内部封装了JDBC，使开发者只需要关注SQL语句本身，而不用在花费精力去处理诸如注册驱动，创建Connection，配置Statement等过程。

MyBatis通过xml或注解的方式将要执行的各种statement（statement，preparedStatement）配置起来，并通过Java对象和Statement中的SQL的动态参数进行映射生成最终执行的SQL语句，最后有MyBatis框架执行SQL并将结果映射成Java对象并返回

与Hibernate的对比

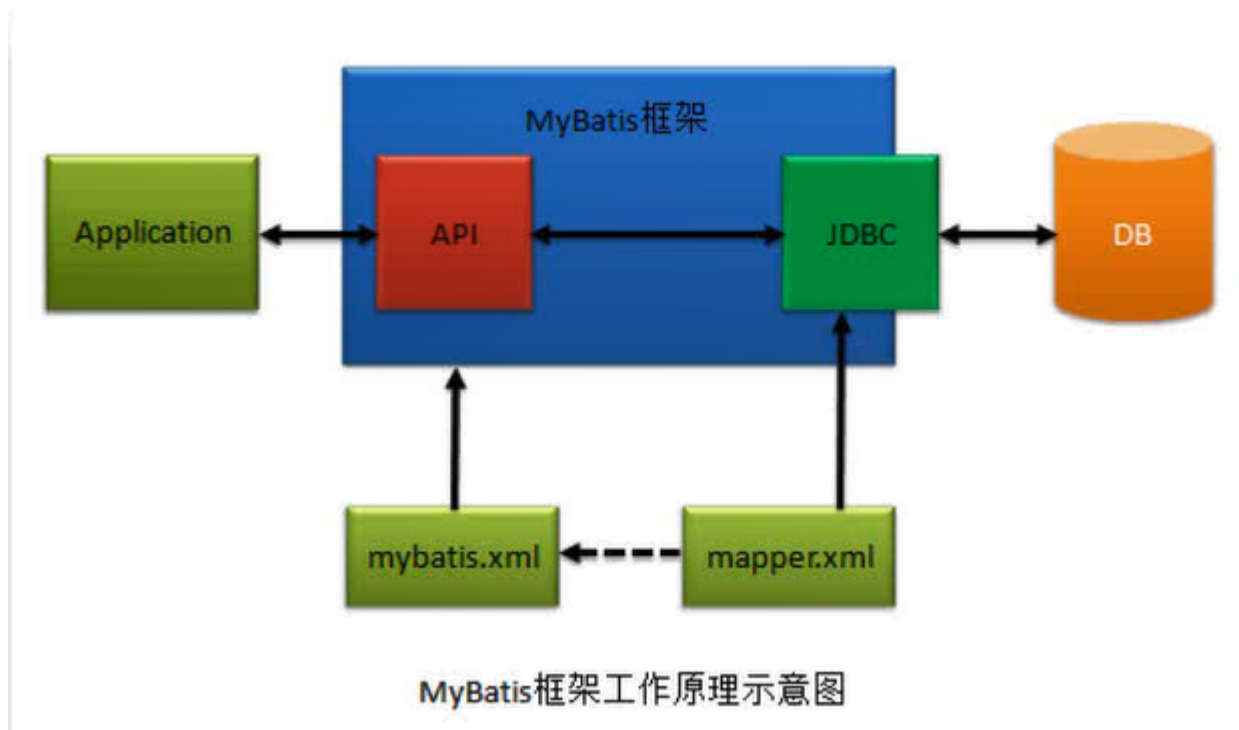
Hibernate框架是提供了全面的数据库封装机制的“全自动”ORM，即实现了POJO和数据库表之间的映射，以及SQL的自动生成和执行

相对于此，MyBatis只能算是“半自动”ORM。其着力点，实在POJO类，与SQL语句之间的映射关系。也就是说，MyBatis并不会为程序员自动生成SQL语句，具体的SQL语句需要程序员自己编写，然后通过SQL语句映射文件，将SQL所需的参数，以及返回的结果字段映射到只当的POJO。因此，MyBatis成为了“全自动”ORM的一种有益补充

与Hibernate相比，MyBatis具有以下几个特点

- （1）在XML文件中配置SQL语句，实现了SQL语句与代码的分离，给程序员的维护带来了很大的遍历
- （2）因为程序员需要自己去编写SQL语句，程序员可以结合数据库自身的特点灵活控制SQL语句，因此能够实现比Hibernate等全自动ORM框架更高的查询效率，能够完成复杂的查询
- （3）简单，易于学习，易于使用，上手快

MyBatis工作原理



MyBatis框架对操作数据库的JDBC进行了封装，但并不是完全进行封装，而是半自动封装，另外一半也就是用户需要写的SQL语句

Application(用户)通过MyBatis提供的API（增删改查...）操作数据库

MyBatis将用户需要进行操作数据库的SQL语句放在了XML文件（mapper.xml）上，其中文件名可以自定义而进行连接数据库需要的配置，比如链接的驱动名，连接数据库名，密码，端口号...也同样写在了XML文件（mybatis.xml）上，在上面写的SQL映射文件也需要在该文件上进行注册

配置

步骤

1. 在Maven项目上添加MyBatis依赖

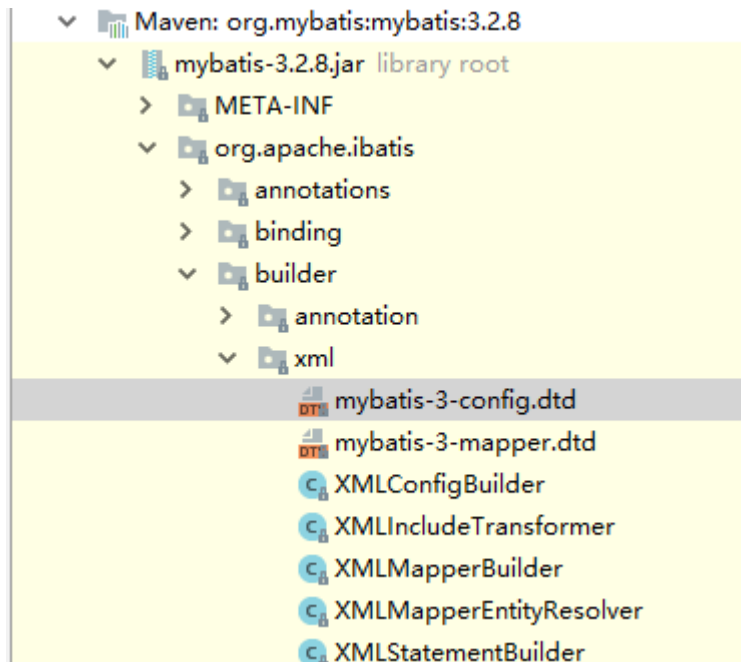
```
<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis</artifactId>
  <version>3.2.8</version>
</dependency>
```

2. 定义实体类

MyBatis通过反射将实体类中的成员变量中的值映射到对应的SQL语句中

3. 创建mapper.xml文件

如果XML标签不提示，需要手动配置DTD文件



4. 配置操作数据库SQL语句

```
<mapper namespace="命名空间">
    <insert id="该条SQL语句的ID" parameterType="bean类">
        insert into Student(id,name,age) values(#{成员变量名1},#{成员变量名2},#{成员变量名3});
    </insert>
</mapper>
```

5. 创建配置文件 (config.xml)

6. 配置连接参数和注册mapper.xml文件

```
<?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>
    <!--添加配置参数信息-->
    <properties resource="properties/jdbc_mysql">
        <!--<property name="jdbc.mysql.driver" value="com.mysql.jdbc.Driver"></property>-->
        <!--<property name="jdbc.mysql.ur" value="jdbc:mysql://127.0.0.1:3306/test"></property>-->
    ->
        <!--<property name="jdbc.mysql.user" value="root"></property>-->
        <!--<property name="jdbc.mysql.password" value="123456"></property>-->
    </properties>

    <!--设置类别名-->
    <typeAliases>
        <!--指定特定bean类别名-->
        <!--<typeAlias type="com.jr.exmaple.bean.Student" alias="student"></typeAlias>-->
        <!--指定在该bean包下的bean对象的别名都为bean类名-->
        <package name="com.jr.exmaple.bean"></package>
    </typeAliases>
```

```

<!-- 对事务的管理和连接池的配置 -->
<environments default="默认数据库连接配置id">
    <environment id="单个数据库连接配置id">
        <transactionManager type="采用事务器"/>
        <dataSource type="采用连接池">
            <property name="driver" value="启动名"/>
            <property name="url" value="连接数据url"/>
            <property name="username" value="数据库名"/>
            <property name="password" value="数据库密码"/>
        </dataSource>
    </environment>
</environments>

<!-- mapping 文件路径配置 -->
<mappers>
    <mapper resource="注册的mapper.xml文件路径"/>
</mappers>
</configuration>

```

什么是连接池技术

通常WEB应用在进行数据库连接时，往往数据库并不是跟WEB应用在同一个服务器，所以每次在进行数据库连接操作是，都是一个相当消耗资源的操作。

所以连接池技术是一个将连接数据库的所有对象保存在WEB服务器缓存中，下次要想再次连接数据库就不需要重新连接数据库，而是直接拿到该连接对象进行数据库操作，从而节省了耗费的时间。

7. 添加log4j日志框架

如果需要查看到MyBatis对数据进行的操作的日志的话，需要添加log4j的jar包，否则查看不到任何操作的信息

```

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

```

配置标签

- **<properties resoures="路径"> </properties>**

在配置MyBatis环境时，可以将一些配置参数统一放到一个文件（如：properties文件），然后在配置文件中引用该文件，便于管理

- **<typeAliases></typeAliases>**

在配置SQL语句的映射文件中，若每次指定的bean类的parameterType类型名很长，那么可以给该bean类取个别名，parameterType中也就填写该别名替代复杂的完整路径。

1. 指定特定bean类别名

```
<typeAliases>
<typeAlias type="com.jr.exmaple.bean.Student" alias="student"></typeAlias>
</typeAliases>
```

2. 指定在该bean包下的bean对象的别名都为bean类名

```
<typeAliases>
<package name="com.jr.exmaple.bean"></package>
</typeAliases>
```

- **<mappers></mappers>**

注册SQL映射文件

1. 指定SQL映射文件路径

```
<mappers>
  <mapper resource="xml/student_mapper.xml"/>
</mappers>
```

2. 注册指定url文件下的SQL映射路径

```
<mappers>
  <mapper url="file:///Program Files/mapper.xml"/>
</mappers>
```

3. 在使用注解式开发时，注册指定完整类接口

```
<mappers>
  <mapper class="com.jr.exmaple.dao.IStudentDao2"/>
</mappers>
```

4. 注册包名下的所有Dao的映射文件，注意前提是需要映射文件名需要和Dao类名一致，而且SQL映射文件必须与Dao类放在同一个包下

```
<mappers>
  <package name="com.jr.exmaple.dao"></package>
</mappers>
```

注意：如果一个接口被多次注册会出现错误，如在namespace中指定了接口路径后，在通过<package/>自动注册包下的所有接口，无意中可能该接口就会被注册了多次。

Maven工程注意

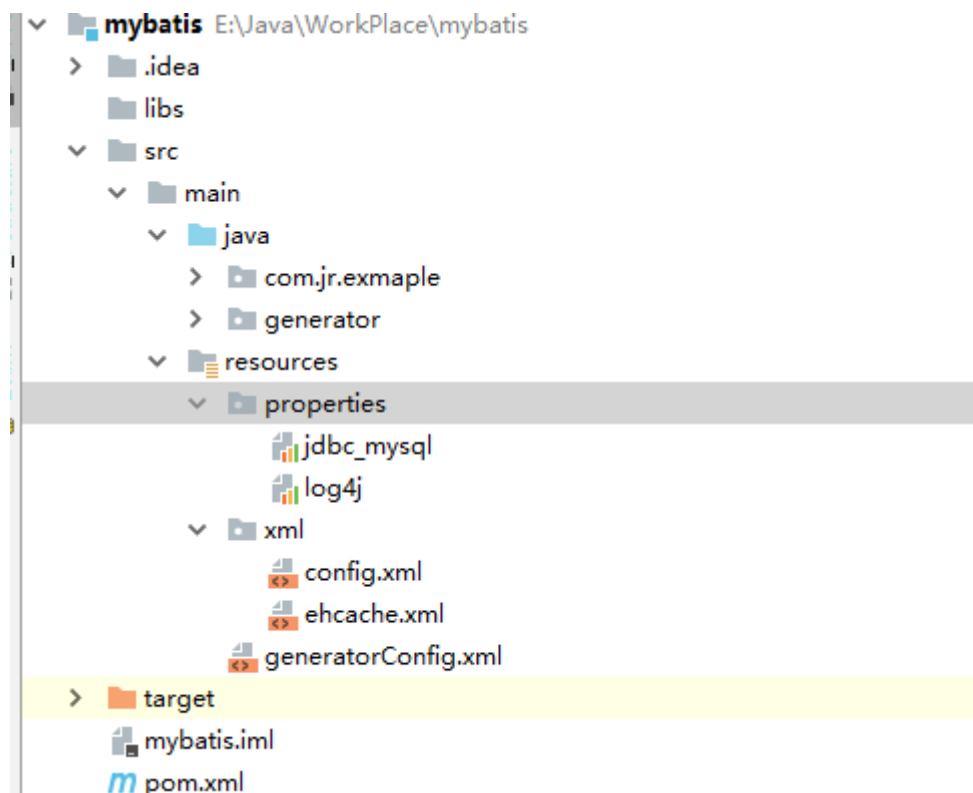
- Maven 工程中，.xml 或 .properties 配置文件无法找到

在Maven工程中, xml、properties等资源不会自动被编译, 需要在配置文件中指定编译资源

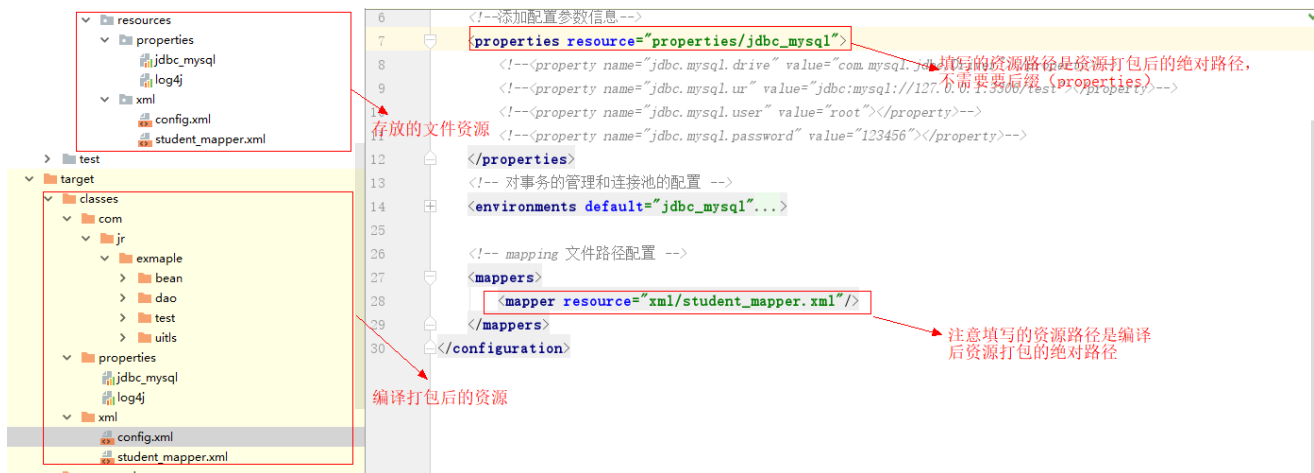
```
<build>
  <resources>
    <!-- 编译之后包含xml和properties -->
    <resource>
      <directory>src/main/java/resources</directory>
      <includes>
        <include>**/*</include><!-- **:表示所有上级文件文件夹; *:表示所有文件-->
        <include>*.xml</include><!-- 表示编译所有xml文件-->
      </includes>
      <filtering>true</filtering>
    </resource>
  </resources>
</build>
```

- 将资源统一存放问题

将xml, properties等配置文件统一放到resources文件目录



注意引用文件的路径是编译后的文件路径



CURD(增删改查)

前言

使用MyBatis进行增删改查，C(create)、U(update)、R(read)、D(delete)

操作步骤

1. 创建SQL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="student">

<mapper/>
```

2. 创建SQL映射语句

```
<insert id="">
    insert into Student(name,age) values(#{},#{ });
</insert>
```

3. 创建MyBatis配置文件

```
<?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>

</configuration>
```

4. 配置环境和注册SQL映射文件

```
<!--添加配置参数信息-->
<properties resource="properties/jdbc_mysql">
    <!--<property name="jdbc.mysql.driver" value="com.mysql.jdbc.Driver"></property>-->
    <!--<property name="jdbc.mysql.ur" value="jdbc:mysql://127.0.0.1:3306/test">
</property>-->
    <!--<property name="jdbc.mysql.user" value="root"></property>-->
    <!--<property name="jdbc.mysql.password" value="123456"></property>-->
</properties>
<!--设置类别名-->
<typeAliases>
    <!--指定特定bean类别名-->
    <!--<typeAlias type="com.jr.exmaple.bean.Student" alias="student"></typeAlias>-->
    <!--指定在该bean包下的bean对象的别名都为bean类名-->
    <package name="com.jr.exmaple.bean"></package>
</typeAliases>
<!-- 对事务的管理和连接池的配置 -->
<environments default="jdbc_mysql">
    <environment id="jdbc_mysql">
        <transactionManager type="JDBC"/>
        <dataSource type="POOLED">
            <property name="driver" value="${jdbc.driver}"/>
            <property name="url" value="${jdbc.url}"/>
            <property name="username" value="${jdbc.user}"/>
            <property name="password" value="${jdbc.password}"/>
        </dataSource>
    </environment>
</environments>

<!-- mapping 文件路径配置 -->
<mapppers>
    <mapper resource="xml/student_mapper.xml"/>
</mapppers>
```

5. 初始化载入log4j配置文件

```
PropertyConfigurator.configure("src/main/java/resources/properties/log4j");
```

6. 获取SqlSession对象

```
public static SqlSession getSqlSession() {
```



```

        if (sqlSession == null) {
            synchronized (SqlSessionUtil.class) {
                if (sqlSession == null) {
                    try {
                        InputStream inputStream =
Resources.getResourceAsStream("xml/config.xml");
                        SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);
                        sqlSession = sqlSessionFactory.openSession();
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
            }
        }
        return sqlSession;
    }
}

```

标签

在进行数据库操作时，携带的参数或者产生的数据有时需要封装成对象，那么<resultMap/>就是将数据封装成映射对象的标签

```

<resultMap id="resultMapID" type="对象类型">
    <id column="id" property="id" javaType="int" jdbcType="INTEGER"></id>
    <result column="name" property="name" javaType="String" jdbcType="CHAR"></result>
    <result column="age" property="age" javaType="String" jdbcType="CHAR"></result>
</resultMap>

```

有时不希望将参数封装成对象类型，而是直接将参数封装成Map集合，那么可以通过parameterMap自定义参数映射关系

```

<parameterMap id="studentMap" type="java.util.Map">
<!--property:对应在上面封装数据映射的property    ;    resultMap:映射数据对象的id-->
<parameter property="name" resultMap="map"></parameter>
<parameter property="age" resultMap="map"></parameter>
</parameterMap>

```

增

<insert/>标签

```
<insert id="" parameterType="">
    insert into Student(name,age) values(#{},#{});
</insert>
```

将参数使用bean对象封装

SQL映射文件

```
<insert id="insert_bean" parameterType="com.jr.exmaple.bean.Student">
    insert into Student(name,age) values(#{name},#{age});
</insert>
```

通过#{成员变量名}, 获取到封装对象中的数据

接口实现方法

```
@Override
public void insert_bean(Student student) {
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    sqlSession.insert("student.insert_bean", student);
    sqlSession.commit();
    sqlSession.close();
}
```

调用SqlSession的**insert()**、**commit()**、**close()**进行数据传递, 事务的提交和关闭

将参数封装成Map集合

SQL映射文件

```
<insert id="insert_map" parameterType="java.util.Map">
    insert into Student(name,age) values(#{name},#{age});
</insert>
```

通过#{集合中的key}, 获取到集合中的数据

接口实现方法

```

@Override
public void insert_map(Map studentMap) {
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    sqlSession.insert("student.insert_map", studentMap);
    sqlSession.commit();
    sqlSession.close();
}

```

将参数封装成数组

SQL映射文件

```

<insert id="inser_array" >
    insert into Student(name,age) values("#{array[0]}",#{array[1]});
</insert>

```

传入参数如果是数据，那么获取数据需要根据array获取到数据引用，然后通过index获取到数组中的数据

接口实现方法

```

@Override
public void insert_array(String[] data) {
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    sqlSession.insert("student.inser_array", data);
    sqlSession.commit();
    sqlSession.close();
}

```

获取到Mysql分配的id字段值

有时候添加一条字段数据是不需要自定该字段的ID，而是由数据库系统自动生成。那么在添加数据到数据库之后想要获取到该ID并封装到之前的Bean对象的话。可通过<selectKey/>标签实现

SQL映射文件

```

<insert id="insert_get_id" parameterType="Student">
    insert into Student(name,age) values("#{name}",#{age});
    <selectKey resultType="int" keyProperty="id" order="AFTER">
        select @@identity
    </selectKey>
</insert>

```

```

insert into student(name,age) value('SB',18);

select @@identity;

```

每次新增字段后，在insert语句结束后，在查询identity变量，会获取到新增字段的主键ID值

接口实现方法

```
@Override
public void inser_get_id(Student student) {
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    sqlSession.insert("insert_get_id", student);
    sqlSession.commit();
    sqlSession.close();
}
```

调用SqlSession的insert()、commit()、close()进行数据传递，事务的提交和关闭

删

<delete/>标签

```
<delete id="" parameterType="">
    delete from Student where id=#{};
</delete>
```

根据一个int参数id删除数据

SQL映射文件

```
<delete id="deleteByID" parameterType="int">
    delete from Student where id=#{id};
</delete>
```

接口实现方法

```
@Override
public void deleteByid(int id) {
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    sqlSession.delete("student.deleteByID", id);
    sqlSession.commit();
    sqlSession.close();
}
```

调用SqlSession的**delete()**、commit()、close()进行数据传递，事务的提交和关闭

根据对象参数删除数据

SQL映射文件

```
<delete id="deleteByBean" parameterType="Student">
    delete from Student where id=#{id}
</delete>
```

通过#{成员变量名}，获取到封装对象中的数据

接口实现方法

```
@Override
public void deleteByBean(Student student) {
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    sqlSession.delete("student.deleteByBean", student);
    sqlSession.commit();
    sqlSession.close();
}
```

调用SqlSession的**delete()**、**commit()**、**close()**进行数据传递，事务的提交和关闭

改

<update/>标签

将修改数据封装到bean对象

SQL映射文件

```
<update id="updateByBean" parameterType="Student">
    update Student SET name=#{name},age=#{age} where id=#{id};
</update>
```

通过#{成员变量名}，获取到封装对象中的数据

接口实现方法

```

@Override
public void updata(Student student) {
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    sqlSession.update("student.updataByBean", student);
    sqlSession.commit();
    sqlSession.close();
}

```

调用SqlSession的update()、commit()、close()进行数据传递，事务的提交和关闭

查

<select/>标签

```

<select id="selectStudentByArgs" resultType="Student" >
    select * from student where id=#{0};
</select>

```

查询所有数据到List集合

SQL映射文件

```

<select id="selectAllStudentList" resultType="Student">
    select * from student
</select>

```

接口实现方法

```

@Override
public List<Student> selectAllStudentList() {
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    List<Student> students = sqlSession.selectList("student.selectAllStudentList");
    sqlSession.commit();
    sqlSession.close();
    return students;
}

```

调用SqlSession的selectList()、commit()、close()进行数据传递，事务的提交和关闭

查询所有数据到Map集合

SQL映射文件

```
<select id="selectAllStudentMap" resultType="Student">
    select * from student
</select>
```

接口实现方法

```
@Override
public Map<Object, Student> selectAllStudentMap() {
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    Map<Object, Student> studentMap = sqlSession.selectMap("student.selectAllStudentMap",
"id");
    sqlSession.commit();
    sqlSession.close();
    return studentMap;
}
```

调用SqlSession的selectMap()、commit()、close()进行数据传递，事务的提交和关闭

指定特定参数查询

SQL映射文件

```
<select id="selectStudentByMap" parameterType="java.util.Map" resultType="Student">
    select * from student where id=#{id} and age >#{age};
</select>
```

通过#{集合中的key}，获取到集合中的数据

接口实现方法

```
@Override
public List<Student> selectStudentByMap(Map studentMap) {
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    List<Student> students = sqlSession.selectList("student.selectStudentByMap", studentMap);
    sqlSession.commit();
    sqlSession.close();
    return students;
}
```

调用SqlSession的selectList()、commit()、close()进行数据传递，事务的提交和关闭

查询映射实体类问题：由于查询的字段数据映射到实体类要求字段名和实体类的成员变量名一致，那么有些情况下会出现不一致的问题，那么可以通过下面的手段将字段数据映射到实体类

1. 给查询出来的字段名进行重名，命名与实体类成员变量名一致

2. 通过<resultMap/>标签自定义映射关系

总结：在SQL映射文件中，指定映射数据到对象类型后，调用SqlSession的不同方法，MyBatis即可以将查询到的对象封装到List集合或Map集合

注意：MyBatis底层通过反射，根据数据字段名拼接set方法将数据封装到指定Bean对象，需要注意的是，在实例化bean对象时，要求该对象有一个无参的构造方法

获取方法参数

#获取到动态参数值

1. 单个方法参数：#{成员变量名}
2. 多个方法参数：#{0(方法参数位置)}
3. 再注解式开发中的多个方法参数：在方法参数中使用@Param("key"),后通过#{key}获取方法参数值

```
@Delete("delete from student where id=#{myId}")
void delete(@Param("myId") int id);
```

4. Map集合参数：#{Map集合Key}
5. 数组参数：#{array[数组指针]}

数据查询和提交到数据库乱码问题

有时候数据查询的结果与数据库内的不一致，或者数据提交到数据库也可能出现乱码不一致的现象。

如果出现乱码的现象，解决方法是在连接数据库的url上添加参数，如：

```
jdbc:mysql://localhost:3306/数据库名? useUnicode=true&characterEncoding=utf8
```

Mapper代理

前言


```

@Override
public void insert_bean(Student student) {
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    sqlSession.insert("student.insert_bean", student);
    sqlSession.commit();
    sqlSession.close();
}

```

我们观察到，dao的接口实现类并没有实际实现了什么方法，都是MyBatis实现了数据操作的方法
所以，MyBatis想到或许自己可以直接创建接口实现类。这就是通过Mapper动态代理生成接口实现对象。

实现步骤

1. SQL映射文件中的namespace属性必须是实现接口的完整路径

```

<mapper namespace="com.jr.exmaple.dao.IStudentDao">
</mapper>

```

2. SQL语句的id必须是接口的方法名

接口实现方法

```

@Override
public List<Student> selectStudentByMap(Map studentMap) {
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    List<Student> students = sqlSession.selectList("student.selectStudentByMap", studentMap);
    sqlSession.commit();
    sqlSession.close();
    return students;
}

```

SQL映射文件

```

<select id="selectStudentByMap" parameterType="java.util.Map" resultType="Student">
    select * from student where id=#{id} and age >#{age};
</select>

```

原理

底层通过动态代理然后通过指定class文件反射生成实例对象并，根据id中的方法名，在通过方法获取接口中的方法并实现该方法。

动态SQL

前言

动态SQL，主要用于解决查询条件不确定的情况。在程序运行期间，根据用户提交的查询条件进行查询。提交的查询条件不同，执行的SQL语句不同。若将每种可能的情况均逐一列出，将所有条件进行排列组合，将会出现大量的SQL语句。此时，可使用动态SQL来解决这样的问题。

动态SQL，即通过MyBatis提供的各种标签对条件做出判断以实现动态拼接SQL语句。

用户自定义查询	
姓名:	<input type="text"/>
学号	<input type="text"/>
年龄:	从 <input type="text"/> 到 <input type="text"/>
成绩	从 <input type="text"/> 到 <input type="text"/>
<input type="button" value="查询"/>	

if

判断传入参数的条件，若不为空的等条件下拼接SQL条件语句。

```
<select id="selectStudentByMap" resultType="Student">
    SELECT * from Student where
    <if test="id !=null and id !=''">      判断传入的参数id不为null and 不等于''的情况下,
        Student.id >= #{id}              拼接该条SQL条件语句。
    </if>
    <if test="name !=null && name !=''">
        and Student.name like '%' #{name} '%'
    </if>
    <if test="age != null && age != ''">
        and Student.age=#{age}
    </if>
</select>
```

where

在单独使用<if/>动态SQL标签时，会存在一个问题。假如上面参数传入时，id为null的情况下，则where条件判断语句直接拼接成**where and Student.name like '%' #{name} '%'**，这样的SQL语句无疑时错误的。

所以，<where/>动态SQL标签也就解决了这个问题

```
<select id="selectStudentByWhere" resultType="Student" parameterType="Student">
    SELECT * from Student
    <where>
        <if test="id !=null and id !=''">
            Student.id >= #{id}
        </if>
        <if test="name !=null && name !=''"> 若第一个if标签为false 则该条条件语句的
```

```

        and Student.name like '%' #{name} '%' and会被自动去除
    </if>
    <if test="age != null && age != ''">
        and Student.age=#{age}
    </if>
</where>
</select>

```

在不满足条件时，如果直接拼接**and**条件语句，那么<where/>动态标签会自动将**and**去除，

choose

对于<choose/>标签，其会从第一个<when/>开始逐个向后进行条件判断。若出现<when/>中的test属性为true的情况下，则直接结果<choose/>标签，不再向后进行判断查找，若所有<when/>的test判断结果均为false，则最后会执行<otherwise/>标签

```

<select id="selectStudentByChoose" resultType="Student" parameterType="Student">
    SELECT * from Student
    <where>
        <choose>
            <when test="id !=null && id !=''"> 若该when中的test属性为true，则不在执行
                id=#{id}                                下面的判断
            </when>
            <when test="age !=null && age !=''">
                and age <=#{age}
            </when>
            <otherwise>
                <if test="name !=null && name !=''">
                    and name like '%' #{name} '%'
                </if>
            </otherwise>
        </choose>
    </where>
</select>

```

foreach

有一种情形，就是查询在该id范围集合内的数据，也就是id in (? , ?) 这种SQL语句拼接情况。

那么<foreach/>标签也就能满足该拼接条件

```

<select id="selectStudentByForeach" resultType="Student" parameterType="java.util.List">
    SELECT * from Student
    <where>
        <choose>
            <when test="list!=null && list.size!=0">
                id in
                <foreach collection="list" item="id" open="(" separator="," close=")">

```

```

        #{id}
    </foreach>
</when>
<otherwise>
    1<0
</otherwise>
</choose>
</where>
</select>

```

<foreach/>标签对传入的参数集合或者数组进行了遍历，遍历的每一个元素进行SQL语句拼接。

Set

MyBatis在生成update语句时若使用if标签，如果前面的if没有执行，则可能导致有多余逗号的错误。如：

```

<update <upda id="updateByPrimaryKeySelective" parameterType="RecruitmentConfBanner">
    UPDATE conf_banner
    set
        <if test="bannerName != null">
            t.banner_name = #{bannerName},
        </if>
        <if test="bannerUrl != null">
            t.banner_url = #{bannerUrl},
        </if>
    where t.banner_id = #{bannerId}
</update>

```

使用set标签可以将动态的配置SET 关键字，和剔除追加到条件末尾的任何不相关的逗号

```

<update <upda id="updateByPrimaryKeySelective" parameterType="RecruitmentConfBanner">
    UPDATE conf_banner
    <set>
        <if test="bannerName != null">
            t.banner_name = #{bannerName},
        </if>
        <if test="bannerUrl != null">
            t.banner_url = #{bannerUrl},
        </if>
    <set/>
    where t.banner_id = #{bannerId}
</update>

```

trim

trim元素的主要功能是:

1. 在自己包含的内容前加上某些前缀。与之对应的属性是prefix
2. 可以在其后加上某些后缀。与之对应的属性是suffix
3. 可以把包含内容的首部某些内容覆盖，即忽略。与之对应的属性是prefixOverrides
4. 也可以把尾部的某些内容覆盖，与之对应的属性是suffixOverrides。

正因为trim有这样的功能，所以我们可以非常简单的利用trim来代替where元素等其它标签元素的功能。
如：

```
<select id="selectUsersTrim" resultMap="resultListUsers" parameterType="Users">
    select * from users
    <trim prefix="where" prefixOverrides="and">
        <if test="name!=null">
            name=#{name}
        </if>
        <if test="address!=null">
            and address=#{address}
        </if>
    </trim>
</select>
```

执行SQL语句

```
select * from users where name=xxx and address=xxx;
```

或

```
select * from users where address=xxx;
```

prefix属性定义拼接的SQL语句内容包含where前缀

prefixOverrides设置定义拼接的SQL语句内容的省略and前缀

注意事项

在mappre的动态SQL中若出现大于号(>)、小于号(<)、大于等于号(>=)、小于等于号(<=)、最好将其转换成事宜符号。否则，xml可能出现解析出错问题。

特别是对于小于号(<)，在XML中是绝对不能出现的。否则，一定出现错误。

原符号	<	<=	>	>=	&	'	"
替换符号	<	<=	>	>=	&	'	"

关联关系查询

前言

当查询内容涉及到具有关联关系的多个表时,就需要使用关联查询< 根据表与表间的关联关系的不同, 关联查询分为四种:

(1) 一对一关联查询

(2) 一对多关联查询

(3) 多对一关联查询

(4) 多对多关联查询 由于日常工作中最常见的关联关系是一对多、多对一与多对多, 所以这里就不专门至讲解一对一关联查询了, 其解决方案与多对一解决方案是相同的。

一对多

介绍

这里的一对多关联查询是指, 在查询一方对象的时候, 同时将其所关联的多方对象也都查询出来。

这里以国家Country与部长Minister间的一对多关系进行演示。

定义实体

```
public class Country {  
    private Integer cid;  
    private String cname;  
    private Set<Minister> ministers;  
  
    // getter and setter  
    // toString()  
}
```

```
public class Minister {  
    private Integer mid;  
    private String mname;  
  
    // getter and setter  
    // toString()  
}
```

定义数据库表

The diagram illustrates a relationship between two database tables. On the left is the 'country' table with columns 'cid' and 'cname'. It contains two rows: (1, USA) and (2, England). On the right is the 'minister' table with columns 'mid', 'mname', and 'countryId'. It contains five rows: (1, aaa, 1), (2, bbb, 1), (3, ccc, 2), (4, ddd, 2), and (5, eee, 2). A red arrow points from the 'country' table to the 'minister' table, indicating a foreign key relationship where 'countryId' in the 'minister' table references 'cid' in the 'country' table.

cid	cname
1	USA
2	England

mid	mname	countryId
1	aaa	1
2	bbb	1
3	ccc	2
4	ddd	2
5	eee	2

定义Dao层接口

```
public interface ICuntryDao {
    void insertCuntry(Cuntry cuntry);

    //多表连接查询方式
    List<Cuntry> selectAllCountryOneSelect();

    //多表单独查询方式
    List<Cuntry> selectAllCountryTwoSelect();
}
```

定义测试类

```
public class Associsational {
    SqlSession sqlSession;
    ICuntryDao iCuntryDao;

    @Before
    public void before() {
        PropertyConfigurator.configure("src/main/java/resources/properties/log4j");
        sqlSession = SqlSessionUtil.getSqlSession();
        iCuntryDao = sqlSession.getMapper(ICuntryDao.class);
        iMinistorDao = sqlSession.getMapper(IMinistorDao.class);
    }

    @After
    public void after() {
        sqlSession.commit();
        sqlSession.close();
    }

    @Test
    public void selectAllCountry() {
```

```

        List<Cuntry> cuntrys = iCuntryDao.selectAllCountryOneSelect();
        for (Cuntry cuntry : cuntrys) {
            System.out.println(cuntry);
        }
    }

    @Test
    public void selectAllCountryTwoSelect() {
        List<Cuntry> cuntrys = iCuntryDao.selectAllCountryTwoSelect();
        for (Cuntry cuntry : cuntrys) {
            System.out.println(cuntry);
        }
    }
}

```

定义映射文件

多表连接查询方式

多表连接查询方式是将多张表进行连接，连为一张表后进行查询。其查询的本事一张表。

```

<resultMap id="cuntry1" type="Cuntry">
    <id column="cid" property="cId"/>
    <result column="cname" property="cName"/>
    关系属性的映射关系
    <collection property="ministers" ofType="Minister">
        <id column="mid" property="mId"></id>
        <result column="mname" property="mName"></result>
    </collection>
</resultMap>
<select id="selectAllCountryOneSelect" resultMap="cuntry1">
    select cuntry.cid,cuntry.cname ,minister.mname from cuntry,minister
    <where>
        cuntry.cid=minister.cid
    </where>
</select>

```

注意，此时即使字段名与属性名相同，在<resultMap/>中要写出它们的映射关系，因为框架是一句这个<resultMap/>封装对象的

多表单独查询方式

多表单独查询方式是多张表各自查询各自的相关内容，需要多张表的联合数据，则主表的查询结果联合其他表的查询结果，然后封装为一个对象。

当然，这多个查询是可以跨多个映射文件的，即是可以跨多个namespace的。在使用其它namespace的查询时，添加上其所在的namespace即可


```

<mapper namespace="com.abc.dao.ISomeDao">

    <!-- 根据外键countryId查询minister表 -->
    <select id="selectMinisterByCountry" resultType="Minister">
        select mid,mname from minister where countryId=#{ooo}
    </select>

    <resultMap type="Country" id="countryMapper">
        <id column="cid" property="cid"/>
        <result column="cname" property="cname"/>
        <!-- 关联属性的映射关系 -->
        <!-- 集合的数据来自于指定的select查询。
            而该select查询的动态参数来自于column指定的字段值 -->
        <collection property="ministers" ofType="Minister"
            select="selectMinisterByCountry"
            column="cid"/>
    </resultMap>
    <!-- 根据id查询country表 -->
    <select id="selectCountryById" resultMap="countryMapper">
        select cid,cname from country where cid=#{xxx}
    </select>
</mapper>

```

关联属性<collection/>的数据来自另一个查询<selectMinisterByCountry/>。而该查询的
 <selectMinisterByCountry/>的动态参数countryid=#{ooo}的值来自与查询<selectMinisterByCountryById/>
 的查询结果字段cid

也就是说，接口先查询主查询Country表，而后将主查询中的cid作为参数传给关联查询Minister表

多对一

介绍

多对一关联查询是指，在查询多方对象的时候，同时将其所关联的一方对象也查询出来

由于在查询多方对象是也是一个一个查询，所以多对一关联查询，其实也就是一对一关联查询。即一对一关联查询的实现方式与多对一方式是相同的

定义实体

```

public class Country {
    private Integer cid;
    private String cname;
    private Set<Minister> ministers;

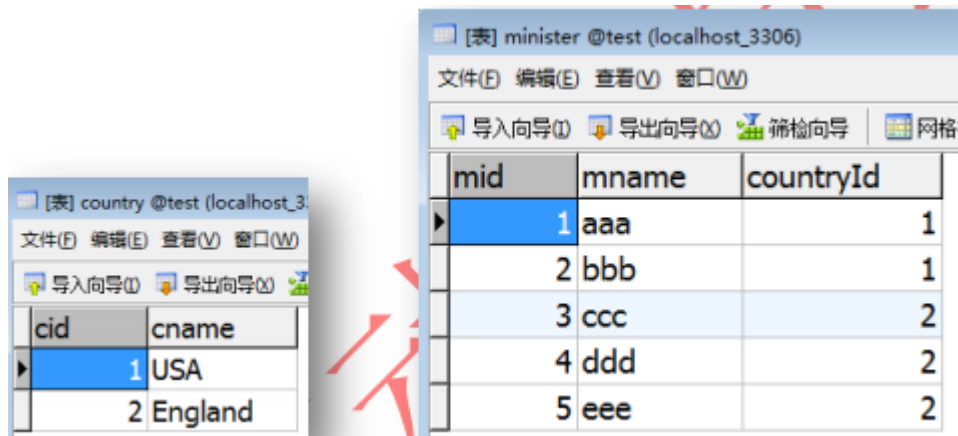
    // getter and setter
    // toString()
}

```

```
public class Minister {
    private Integer mid;
    private String mname;

    // getter and setter
    // toString()
}
```

定义数据库表



cid	cname
1	USA
2	England

mid	mname	countryId
1	aaa	1
2	bbb	1
3	ccc	2
4	ddd	2
5	eee	2

定义Dao层接口

```
public interface IMinistorDao {
    void insert_minister(Minister minister);

    //Minister
    List<Minister> selectAllMinisterOneSelect();

    //先主查询后关联查询
    List<Minister> selectAllMinisterTwoSelect();
}
```

定义测试类

```
public class Associsational {
    SqlSession sqlSession;
    IMinistorDao iMinistorDao;

    @Before
    public void before() {
        PropertyConfigurator.configure("src/main/java/resources/properties/log4j");
    }
}
```

```

        sqlSession = SqlSessionUtil.getSqlSession();
        iMinistorDao = sqlSession.getMapper(IMinistorDao.class);
    }

    @After
    public void after() {
        sqlSession.commit();
        sqlSession.close();
    }

    @Test
    public void insert_minister() {
        Minister minister = new Minister();
        minister.setmName("兆民");
        minister.setcId(3);
        iMinistorDao.insert_minister(minister);
    }

    @Test
    public void selectAllMinisterOneSelect() {
        List<Minister> ministers = iMinistorDao.selectAllMinisterOneSelect();
        for (Minister minister : ministers) {
            System.out.println(minister.toString());
        }
    }

    @Test
    public void selectAllMinisterTwoSelect() {
        List<Minister> ministers = iMinistorDao.selectAllMinisterTwoSelect();
        for (Minister minister : ministers) {
            System.out.println(minister.toString());
        }
    }
}

```

定义映射文件

多表连接查询方式

```

<resultMap id="minister1" type="Minister">
  <id column="mid" property="mId"></id>
  <result column="mname" property="mName"></result>
  <association property="cuntry" javaType="Cuntry">
    <id column="cid" property="cId"></id>
    <result column="cname" property="cName"></result>
  </association>
</resultMap>

<select id="selectAllMinisterOneSelect" resultMap="minister1">
  select minister.mid,minister.mname,minister.cid,cuntry.cname
  from minister,cuntry;
</select>

```

关联查询的数据，会通过<association/>映射关系封装成javaType中定义的对象类型并保存在property中的对象属性中

多表单独查询方式

```

<mapper namespace="com.abc.dao.IMinisterDao">

  <select id="selectCountryById" resultType="Country">
    select * from country where cid=#{ooo}
  </select>

  <resultMap type="Minister" id="ministerMapper">
    <id column="mid" property="mid"/>
    <result column="mname" property="mname"/>
    <association property="country"
      javaType="Country"
      select="selectCountryById"
      column="countryId"/>
  </resultMap>

  <select id="selectMinisterById" resultMap="ministerMapper">
    select mid,mname,countryId from minister
    where mid=#{xxx}
  </select>

</mapper>

```

通过主查询出的CountryId作为动态参数，在通过关联查询出Monister表中的其他关联数据

多对多

介绍

什么是多对多关联关系？一个学生可以选多门课程，而一门课程可以由多个学生选。这就是典型的多对多关联关系。多以，所谓多对多关系，其实是由两个互反的一对多关系组成。一般情况下，多对多关系都会通过一个中间表来建立。例如选课表。

定义实体

```
public class Country {  
    private Integer cid;  
    private String cname;  
    private Set<Minister> ministers;  
  
    // getter and setter  
    // toString()  
}
```

```
public class Minister {  
    private Integer mid;  
    private String mname;  
  
    // getter and setter  
    // toString()  
}
```

定义数据库表

[表] student @test (localhost_3306)

文件(F) 编辑(E) 查看(V)

导入向导(I) 导出向导(O)

sid	sname
1	张三
2	李四

[表] course @test (localhost_3306)

文件(F) 编辑(E) 查看(V)

导入向导(I) 导出向导(O)

cid	cname
1	JavaSE
2	JavaEE
3	Android

[表] middle @test (localhost_3306)

文件(F) 编辑(E) 查看(V) 窗口(W)

导入向导(I) 导出向导(O) 筛选向导(S)

id	studentId	courseId
1	1	1
2	1	2
3	2	1
4	2	3

定义Dao层接口

```
public interface IStudentDao {
    //多对多关联连接查询
    List<Student> selectStudentAndCourse1(int id);
}
```

定义测试类

```
@Test
public void selectStudentAndCourse() {
    List<Student> students = studentDao.selectStudentAndCourse1(1);
    if (students != null & !students.isEmpty()) {
        for (Student student : students) {
            System.out.println(student);
        }
    }
}
```

定义映射文件

```
<resultMap id="StudentMapper" type="Student">
    <id column="id" property="id"></id>
    <result column="name" property="name"></result>
    <result column="age" property="age"></result>

    <collection property="courses" ofType="Course">
        <id column="cid" property="cid"></id>
        <result column="cname" property="cName"></result>
    </collection>
</resultMap>

<select id="selectStudentAndCourse1" resultMap="StudentMapper">
    select stu.id,stu.name,stu.age,cou.cid,cou.cname
    from student as stu ,middle,course as cou
    WHERE
        stu.id=#{id}
        and middle.studentId=stu.id
        and cou.cid=middle.courseId;
</select>
```

多对多关联关系也是通过映射文件`<resultMap>`的`<collection/>`体现的，但是，需要注意的是SQL语句中是对三张表的连接查询

延迟加载

前言

MyBatis中的延迟加载，也称为懒加载，是指在进行关联查询时，按照设置延迟规则推迟对关联对象的select查询。延迟加载可以有效的减少数据库压力。

需要注意的时，MyBatis的延迟加载只是对关联对象的查询有迟延设置，对主加载对象都是直接执行查询语句的。

MyBatis根据关联对象查询的select语句的执行时机，分为三种类型：**直接加载**、**侵入式延迟加载**、**与深度延迟加载**

测试环境

1.关联关系查询

2.定义映射文件

```
关联查询
<select id="selectAllMinisterByCid" resultType="Minister">
    select mid,mname,cid
    from minister where cid =#{cid};
</select>
关联关系映射
```

```

<resultMap id="cuntry2" type="Cuntry">
    <id column="cid" property="cId"/>
    <result column="cname" property="cName"/>
    <collection
        property="ministers"
        ofType="Minister"
        select="selectAllMinisterByCid"
        column="cid">
    </collection>
</resultMap>
主查询
<select id="selectAllCountryTwoSelect" resultMap="cuntry2">
    select cuntry.cid,cuntry.cname from cuntry
</select>

```

3.定义Dao层

```

public interface ICuntryDao {
    List<Cuntry> selectAllCountryTwoSelect();
}

```

直接加载

解释

执行完对主加载对象的select语句，马上执行对关联对象的select查询。

若不开启延迟加载，默认采用直接加载查询

如下面中的SQL查询语句。

```

<resultMap id="minister1" type="Minister">
    <id column="mid" property="mId"></id>
    <result column="mname" property="mName"></result>
    <association property="cuntry" javaType="Cuntry">
        <id column="cid" property="cId"></id>
        <result column="cname" property="cName"></result>
    </association>
</resultMap>

<select id="selectAllMinisterOneSelect" resultMap="minister1">
    select minister.mid,minister.mname,minister.cid,cuntry.cname
    from minister,cuntry;
</select>

```

可观察到查询语句中只有一条select语句，一次直接查询主加载对象和关联对象中的数据。

测试类


```
@Test
public void selectAllCountryTwoSelect() {
    List<Cuntry> cuntrys = iCuntryDao.selectAllCountryTwoSelect();
}
```

查询结构log

```
15:10:18,554 DEBUG selectAllCountryTwoSelect:139 - ==> Preparing: select cuntry.cid,cuntry.cname
from cuntry
15:10:18,602 DEBUG selectAllCountryTwoSelect:139 - ==> Parameters:
15:10:18,628 DEBUG selectAllMinisterByCid:139 - ====> Preparing: select mid,mname,cid from
minister where cid =?;
15:10:18,628 DEBUG selectAllMinisterByCid:139 - ====> Parameters: 3(Integer)
15:10:18,632 DEBUG selectAllMinisterByCid:139 - <==== Total: 4
15:10:18,633 DEBUG selectAllMinisterByCid:139 - ====> Preparing: select mid,mname,cid from
minister where cid =?;
15:10:18,633 DEBUG selectAllMinisterByCid:139 - ====> Parameters: 4(Integer)
15:10:18,635 DEBUG selectAllMinisterByCid:139 - <==== Total: 0
15:10:18,635 DEBUG selectAllMinisterByCid:139 - ====> Preparing: select mid,mname,cid from
minister where cid =?;
15:10:18,635 DEBUG selectAllMinisterByCid:139 - ====> Parameters: 5(Integer)
15:10:18,637 DEBUG selectAllMinisterByCid:139 - <==== Total: 0
15:10:18,637 DEBUG selectAllCountryTwoSelect:139 - <== Total: 3
```

分析：可观察到直接加载一次性将主加载对象和关联对象都查询出来了

侵入式加载

解释

执行完对主加载对象的查询时，不会执行对关联对象的查询，但当要访问主加载对象的详情时，就会马上执行关联对象的select查询。即对关联对象的查询执行，侵入到了主加载对象的详情访问中。也可以这样理解：将关联对象的详情侵入到了主加载对象的详情中，即将关联对象的详情作为主加载对象的详情的一部分出现了

测试类

```
@Test
public void selectAllCountryTwoSelect() {
    List<Cuntry> cuntrys = iCuntryDao.selectAllCountryTwoSelect();
}
```

查询结构log

```
15:12:30,088 DEBUG JdbcTransaction:98 - Setting autocommit to false on JDBC Connection
[com.mysql.jdbc.JDBC4Connection@3234e239]
15:12:30,091 DEBUG selectAllCountryTwoSelect:139 - ==> Preparing: select cuntry.cid,cuntry.cname
from cuntry
15:12:30,127 DEBUG selectAllCountryTwoSelect:139 - ==> Parameters:
15:12:30,261 DEBUG selectAllCountryTwoSelect:139 - <== Total: 3
```

分析：当不使用主加载对象中的数据时，不会进行关联查询

测试类

```
@Test
public void selectAllCountryTwoSelect() {
    List<Cuntry> cuntrys = iCuntryDao.selectAllCountryTwoSelect();
    System.out.println(cuntrys.get(1));
}
```

查询结果log

```
15:15:59,383 DEBUG selectAllCountryTwoSelect:139 - ==> Preparing: select cuntry.cid,cuntry.cname
from cuntry
15:15:59,418 DEBUG selectAllCountryTwoSelect:139 - ==> Parameters:
15:15:59,539 DEBUG selectAllCountryTwoSelect:139 - <==          Total: 3
15:15:59,540 DEBUG selectAllMinisterByCid:139 - ==> Preparing: select mid,mname,cid from
minister where cid =?;
15:15:59,540 DEBUG selectAllMinisterByCid:139 - ==> Parameters: 3(Integer)
15:15:59,545 DEBUG selectAllMinisterByCid:139 - <==          Total: 4
Cuntry{cId=3, cName='中国', ministers=[Minister{mId=1, mName='陈锦荣', cId=3, cuntry=null},
Minister{mId=2, mName='家声', cId=3, cuntry=null}, Minister{mId=3, mName='加成', cId=3,
cuntry=null}, Minister{mId=6, mName='兆民', cId=3, cuntry=null}]}
```

分析：System.out.println(cuntrys.get(1));使用了主加载对象中的数据时，所以菜进行了关联查询

深度加载

解释

执行对主加载对象的查询时，不会执行对关联对象的查询。访问主加载对象的详情时也不会执行关联对象的select查询。只有当真正访问关联对象的详情时，才会执行对关联对象的select查询。

测试类

```
@Test
public void selectAllCountryTwoSelect() {
    List<Cuntry> cuntrys = iCuntryDao.selectAllCountryTwoSelect();
    System.out.println(cuntrys.get(0).getcName());
}
```

查询结果log

```
15:23:48,699 DEBUG selectAllCountryTwoSelect:139 - ==> Preparing: select cuntry.cid,cuntry.cname
from cuntry
15:23:48,733 DEBUG selectAllCountryTwoSelect:139 - ==> Parameters:
15:23:48,833 DEBUG selectAllCountryTwoSelect:139 - <==          Total: 3
中国
```

分析：不使用关联对象中的数据时不会进行关联查询

测试类

```
@Test
public void selectAllCountryTwoSelect() {
    List<Cuntry> cuntrys = iCuntryDao.selectAllCountryTwoSelect();
    System.out.println(cuntrys.get(0).getMinisters());
}
```

查询结果log

```
15:25:41,499 DEBUG selectAllCountryTwoSelect:139 - ==> Preparing: select cuntry.cid,cuntry.cname
from cuntry
15:25:41,549 DEBUG selectAllCountryTwoSelect:139 - ==> Parameters:
15:25:41,678 DEBUG selectAllCountryTwoSelect:139 - <==          Total: 3
15:25:41,679 DEBUG selectAllMinisterByCid:139 - ==> Preparing: select mid,mname,cid from
minister where cid =?;
15:25:41,679 DEBUG selectAllMinisterByCid:139 - ==> Parameters: 3(Integer)
15:25:41,684 DEBUG selectAllMinisterByCid:139 - <==          Total: 4
[Minister{mId=1, mName='陈锦荣', cId=3, cuntry=null}, Minister{mId=2, mName='家声', cId=3,
cuntry=null}, Minister{mId=3, mName='加成', cId=3, cuntry=null}, Minister{mId=6, mName='兆民',
cId=3, cuntry=null}]15:23:48,699 DEBUG selectAllCountryTwoSelect:139 - ==> Preparing: select
cuntry.cid,cuntry.cname from cuntry
15:23:48,733 DEBUG selectAllCountryTwoSelect:139 - ==> Parameters:
15:23:48,833 DEBUG selectAllCountryTwoSelect:139 - <==          Total: 3
中国
```

分析：System.out.println(cuntrys.get(0).getMinisters());使用到了关联对象中的数据时，所以进行关联查询

注意

延迟加载的应用要求，关联对象的查询与主加载对象的查询必须是分别进行的select语句，不能是使用多表连接所进行的select 查询。因为，多表连接查询，其实质是对一张表的查询，对由多张表连接后形成的一张表的查询。会一次性将多张表的所有信息查询出来

MyBatis中对延迟加载设置，可以应用到一对一、一对多、多对一、多对多的所有关系查询中

应用

1. 导入Glib动态代理框架

```
<dependency>
    <groupId>cglib</groupId>
    <artifactId>cglib</artifactId>
    <version>2.2.2</version>
</dependency>
```

2. 在MyBatis配置文件中的<properties/>和<typeAliases/>标签之间配置<setting/>设置全局参数

```
<configuration>
  <!--添加配置参数信息-->
  <properties resource="properties/jdbc_mysql">
  </properties>
  <!--设置全局参数-->
  <settings>
    <setting name="lazyLoadingEnabled" value="true"/>
    <setting name="aggressiveLazyLoading" value="true"></setting>
  </settings>
  <!--设置类别名-->
  <typeAliases>
    <package name="com.jr.exmaple.bean"></package>
  </typeAliases>
</configuration>
```

- 3. 选择是否开启延迟加载
- 4. 选择延迟加载方式

延迟加载策列总结

加载策略	lazyLoadingEnabled	aggressiveLazyLoading
直接加载	false	false
侵入式延迟加载	true	true
深度延迟加载	true	false

查询缓存

前言

查询缓存的使用，主要是为了提高访问速度，将用户对同一数据的重复查询过程简化，不再每次均从数据库查询获取结果数据，从而提高访问速度。

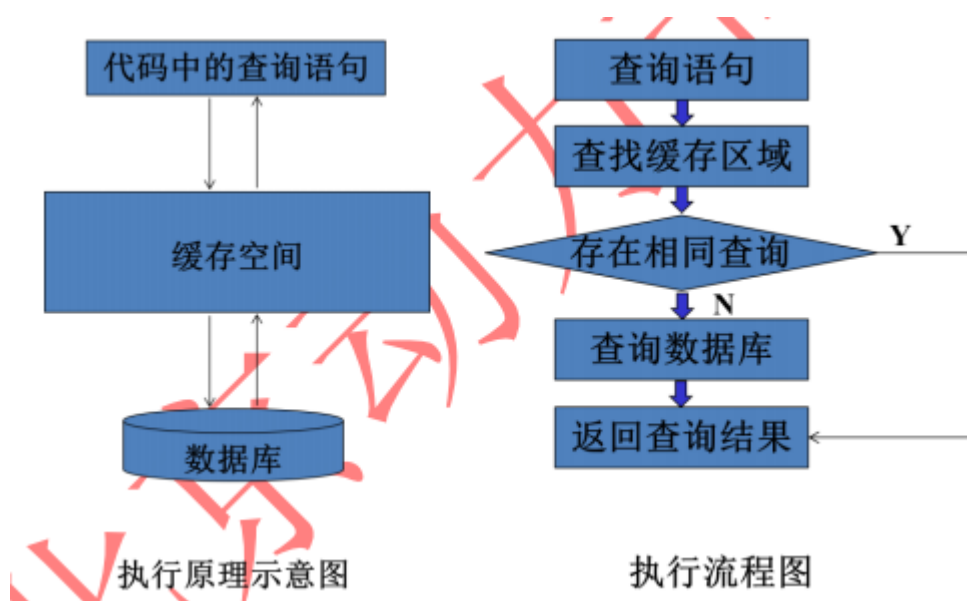
MyBatis的查询缓存机制，根据缓存区的作用域（生命周期）可划分为两种，一级查询缓存和二级查询缓存。

一级缓存

介绍

MyBatis一级查询缓存是基于org.apache.ibatis.cache.impl.PerpetualCache类的HashMap。一次执行完毕后，会将查询结果写入到缓存中，第二次会从缓存中直接获取数据，而不再到数据库中进行查询，从而提高查询效率。

当一个SqlSession结束后，该SqlSession中的一级查询缓存也就不存在了，MyBatis默认一级查询缓存是开启状态的，且不能关闭。



一级缓存依据

一级缓存中缓存的是相同的Sql映射id的查询结果，而非相同Sql语句的查询结果。因为MyBatis内部对于查询缓存，无论是一级缓存还是二级缓存，其底层使用一个HashMap实现：key为Sql的id相关内容（**可理解为sql的id+查询参数**），value为从数据库中查询出的结果。

增删改对一级缓存的影响

增、删、改操作，无论是否进行事务的提交(sqlSession.commit())，均会清空一级查询缓存，将对应key上的value置为null。

使之下次进行查询时会查找数据库。

二级缓存

介绍

MyBatis查询缓存的作用域是根据映射文件mapper的namespace划分的，相同namespace的mapper查询数据放在同一个缓存区域。不同namespace下的数据互不干扰。无论是一级缓存还是二级缓存，都是按照namespace进行分别存放的。

但一、二级缓存的不同之处在于，SqlSession一旦关闭，则SqlSession中的数据将不存在，即一级缓存就不覆存在。而二级缓存的生命周期会与整个应用同步，与SqlSession是否关闭无关。

使用二级缓存的目的，不是共享数据，因为MyBatis从缓存中读取数据的依据是SQL的id，而非查询出来的对象。所以，**二级缓存中的数据不是为了在多个查询之间共享**（所有查询中只要查询结果中存在该对象，就直接从缓存中读取，这就是数据的共享，Hibernate中的缓存就是为了共享，但是MyBatis不是），**而是为了延长该查询结果的保存时间**，提高系统性能

应用

1. 在mapper映射文件中的<mapper/>标签中添加<cache/>子标签

```
<mapper namespace="com.jr.exmaple.dao.IStudentDao">
    <cache></cache>
</mapper>
```

2. 二级缓存的配置

```
<cache type="" eviction="" flushInterval="" readOnly="" size=""></cache>
```

- FIFO: First In First Out 先进先出
- LRU: Least Recently Used 未被使用时间最长的

3. 实体类实现序列化接口

增删改对二级缓存的影响

增删改操作，无论是否进行事务提交（sqlSession.cimmit()）,均会清空一级、二级查询缓存

设置增删改不刷新二级缓存

若要使某个增、删、改操作不清空二级缓存，则需要在其<insert/>或<delete/>或<update/>中添加属性 flushCache="false", 默认为true。

```
<insert id="" parameterType="" flushCache="false">
</insert>
```

关闭二级缓存

全局关闭

所谓全局关闭是指，整个应用的二级缓存全部关闭，所有查询均不使用二级缓存。全局开关设置在主配置文件的全局设置<settings/>中，改属性为cacheEnabled，设置为false，则关闭；ture则开启。默认为true。即二级缓存默认时开启的。

```
<configuration>
    <settings>
        <setting name="cacheEnabled" value="false"></setting>
    </settings>
</configuration>
```

局部关闭

所谓局部关闭是指，整个应用的二级缓存是开启的，但是针对某个<select/>查询，不适用二级缓存。此时可以只单独关闭改<select/>标签的二级缓存。

在改要关闭二级缓存的<select/>标签中，将其属性useCahe设置为false，即可关闭改查询的二级缓存。改属性默认是true，即每个<select/>查询的二级缓存默认是开启的。

```
<select id="" resultType="" useCache="false">

</select>
```

二级缓存的使用原则

- 只能在一个命名空间下使用二级缓存

由于二级缓存中的数据是基于namespace的，即不同namespace中的数据互补干扰。在多个namespace中若均存在对同一个表的操作，那么这多个namespace中的数据可能会出现不一致现象。

- 在单表上使用二级缓存

如果一张表与其它表有关联关系，那么就非常有可能存在多个namespace对同一数据的操作。而不同namespace中的数据互不干扰，所以有可能出现这多个namespace中的数据不一致现象。

- 查询多于修改时使用二级缓存

在查询操作远远多于增删改操作的情况下可以使用二级缓存。因为任何增删改操作都将刷新二级缓存，对二级缓存的频繁刷新将降低系统性能。

使用第三方缓存

MyBatis的特长是SQL操作，缓存数据管理不是其特长，为了提高缓存的性能，MyBatis允许使用第三方缓存产品。ehCache就是其中的一种，它是Hibernate旗下的一款二级缓存产品。

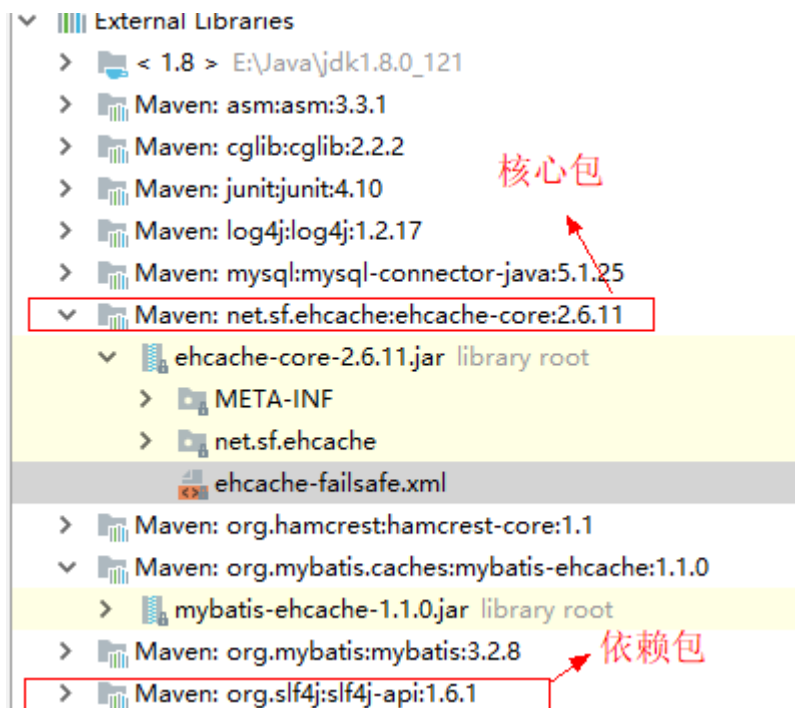
应用ehcache

1. 依赖jar包

```
<dependency>
    <groupId>org.mybatis.caches</groupId>
    <artifactId>mybatis-ehcache</artifactId>
    <version>1.1.0</version>
</dependency>
```

2. 添加ehcache.xml配置文件

解压ehcache的核心jar包ehcache-core.jar，将其中的一个配置文件ehcache-failsage.xml直接放到项目的src目录下，并更名为ehcache.xml



3. 配置文件参数

```
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../config/ehcache.xsd">
```

```
    <diskStore path="java.io.tmpdir"/>
```

```
    <defaultCache
```

```
        maxElementsInMemory="10000"
```

```
        eternal="false"
```

```
        timeToIdleSeconds="120"
```

```
        timeToLiveSeconds="120"
```

```
        overflowToDisk="true"
```

```
        maxElementsOnDisk="10000000"
```

```
        diskExpiryThreadIntervalSeconds="120"
```

```
        memoryStoreEvictionPolicy="LRU">
```

```
        <persistence strategy="localTempSwap"/>
```

```
    </defaultCache>
```

```
</ehcache>
```

- FIFO: First In First Out 先进先出
- LFU: Less Frequently Used 最少使用
- LRU: Least Recently Used 未被使用时间最长的

4. 启动ehcache缓存机制

在映射文件的mapper中的<cache/>中通过type指定缓存机制为ehcache缓存，默认为MyBatis内置的二级缓存org.apache.ibatis.cache.impl.PerpetualCache。


```
<mapper namespace="com.jr.exmaple.dao.ICourseDao">
    <cache type="org.mybatis.caches.ehcache.EhcacheCache">
</mapper>
```

5. ehcache在不同mapper中的个性化设置

在ehcache.xml中设置的属性值，会对该项目中所有使用ehcache缓存机制的缓存区域起作用。一个项目可以有多个mapper，不同的mapper有不同的缓存区域，对于不同缓存区域也可以进行专门针对当前区域的个性化设置，可以通过指定mapper的<cache>属性值来设置

```
<cache type="org.mybatis.caches.ehcache.EhcacheCache">
    <property name="maxElementsInMemory" value="10000"></property>
    <property name="eternal" value="false"></property>
    <property name="timeToIdleSeconds" value="120"></property>
    <property name="timeToLiveSeconds" value="120"></property>
</cache>
```

注解式开发

前言

MyBatis的注解，主要是用于替换映射文件。而映射文件中无非存放这增、删、改、查的SQL映射标签。所以，MyBatis注解，就是要替代映射文件中的个SQL标签。

MyBatis官方文档指出，若要真正发挥MyBatis功能，还是要用映射文件。即MyBatis官方并不建议通过注解方法来使用MyBatis。

增

```
@Insert(value = "insert into Student(name,age) values(#{name},#{age})")
@SelectKey(statement = "select @@identity", keyProperty = "id", before = false, resultType =
Integer.class)
void insert(Student student);
```

删

```
@Delete("delete from student where id=#{id}")
void delete(int id);
```

若标签属性只有一个，可不写属性名

改

```
@Update(value = "update student set name=#{name},age=#{age} where id=#{id}")
void update(Student student);
```

查

```
@Select(value = "select * from student where id=#{id}")
Student select(int id);
```

注册注解Dao

```
<!-- mapping 文件路径配置 -->
<mappers>
    <mapper class="com.jr.exmaple.dao.IStudentDao_Annotation"></mapper>
</mappers>
```

逆向工程

使用MyBatis最大的问题就是需要自己手动写Dao的接口，SQL的映射文件，和数据库对应的Bean类。

有了MyBatis提供的逆向工厂后，这些将会通过该插件对应到数据库中的表结构自动生成。

使用步骤

1. 配置Maven的配置文件

```
//添加逆向工厂插件
<build>
    <finalName>zsxt</finalName>
    <plugins>
        <plugin>
            <groupId>org.mybatis.generator</groupId>
            <artifactId>mybatis-generator-maven-plugin</artifactId>
            <version>1.3.2</version>
            <configuration>
                <verbose>true</verbose>
                <overwrite>true</overwrite>
            </configuration>
        </plugin>
    </plugins>
</build>
```

2. 配置逆向工厂

使用逆向工厂需要配置所需的文件，如指定生成的Dao接口类的路径，生成的SQL映射文件的路径...

需要注意的是，该配置文件需要放在resources资源目录下，并取名generatorConfiguration，否则执行该插件时会出现找不到文件错误。

```
<?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>
    <!--导入属性配置-->
    <properties resource="properties/jdbc_mysql"></properties>

    <!--指定特定数据库的jdbc驱动jar包的位置-->
    <classPathEntry location="C:\Users\Administrator\.m2\repository\mysql\mysql-connector-
java\5.1.25\mysql-connector-java-5.1.25.jar"/>

    <context id="default" targetRuntime="MyBatis3">

        <!-- optional, 旨在创建class时, 对注释进行控制 -->
        <commentGenerator>
            <property name="suppressDate" value="true"/>
            <property name="suppressAllComments" value="true"/>
        </commentGenerator>

        <!--jdbc的数据库连接 -->
        <jdbcConnection>
            <property name="driverClass" value="${jdbc.driver}" />
            <property name="connectionURL" value="${jdbc.url}" />
            <property name="userId" value="${jdbc.user}" />
            <property name="password" value="${jdbc.password}" />
        </jdbcConnection>

        <!-- 非必需, 类型处理器, 在数据库类型和java类型之间的转换控制-->
        <javaTypeResolver>
            <property name="forceBigDecimals" value="false"/>
        </javaTypeResolver>

        <!-- Model模型生成器,用来生成含有主键key的类, 记录类 以及查询Example类
            targetPackage      指定生成的model生成所在的包名
            targetProject      指定在该项目下所在的路径
        -->
        <javaModelGenerator targetPackage="generator.pojo"
            targetProject="src/main/java">

            <!-- 是否允许子包, 即targetPackage.schemaName.tableName -->
            <property name="enableSubPackages" value="false"/>
            <!-- 是否对model添加 构造函数 -->
```

```

        <property name="constructorBased" value="true"/>
        <!-- 是否对类CHAR类型的列的数据进行trim操作 -->
        <property name="trimStrings" value="true"/>
        <!-- 建立的Model对象是否 不可改变 即生成的Model对象不会有 setter方法, 只有构造方法 -->
        <property name="immutable" value="false"/>
    </javaModelGenerator>

    <!--Mapper映射文件生成所在的目录 为每一个数据库的表生成对应的SqlMap文件 -->
    <sqlMapGenerator targetPackage="generator.dao"
        targetProject="src/main/java">
        <property name="enableSubPackages" value="false"/>
    </sqlMapGenerator>

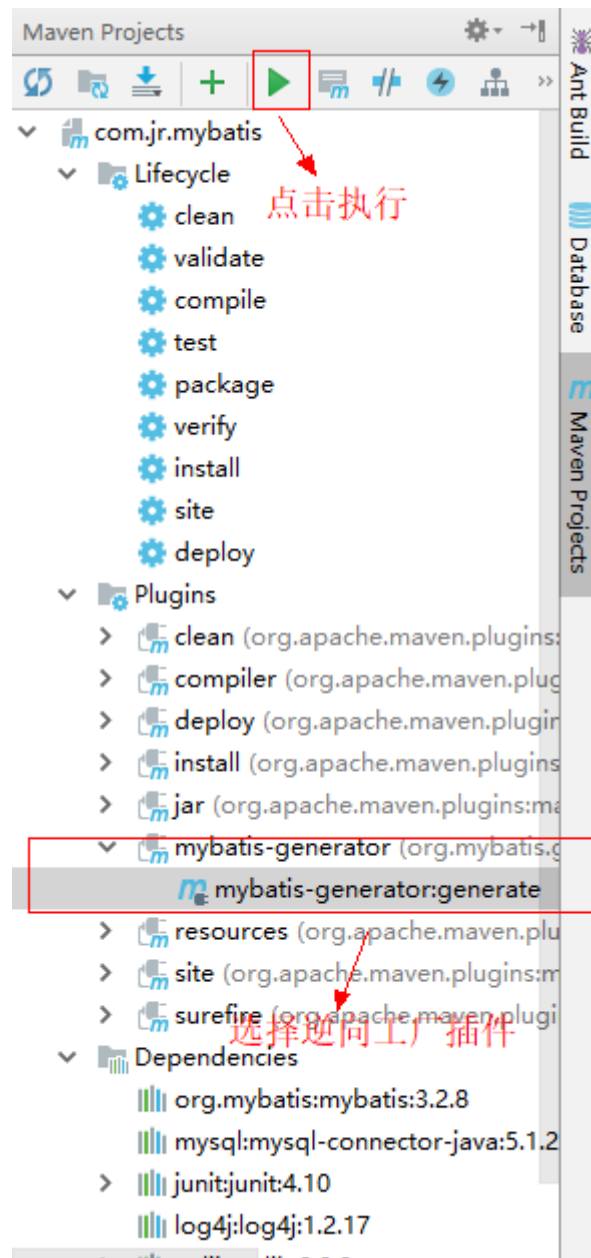
    <!-- 客户端代码, 生成易于使用的针对Model对象和XML配置文件 的代码
        type="ANNOTATEDMAPPER",生成Java Model 和基于注解的Mapper对象
        type="MIXEDMAPPER",生成基于注解的Java Model 和相应的Mapper对象
        type="XMLMAPPER",生成SQLMap XML文件和独立的Mapper接口
    -->
    <javaClientGenerator targetPackage="generator.dao"
        targetProject="src/main/java" type="XMLMAPPER">
        <property name="enableSubPackages" value="true"/>
    </javaClientGenerator>

    <table tableName="student" schema="">
    </table>
</context>
</generatorConfiguration>

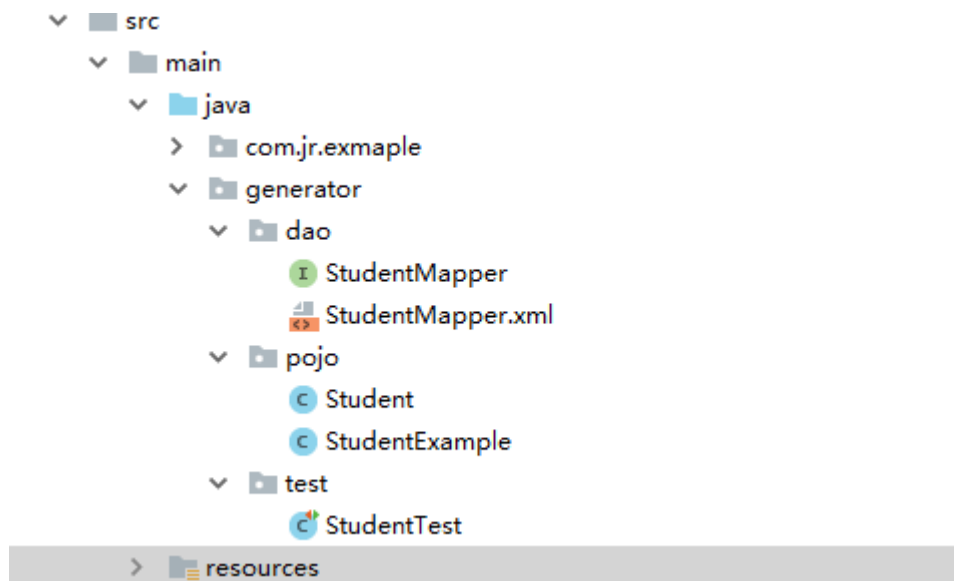
```

其中最要的配置是：

3. 执行逆向工厂插件



4. 生成的Java代码和映射文件



测试

1. 注册SQL映射文件

```
<mappers>
  <!--注册逆向工厂产生的映射文件-->
  <mapper resource="dao/StudentMapper.xml"></mapper>
</mappers>
```

2. 测试类

举例: mybatis工程包名generator

Selective

在insert和update的操作中都有出现Selective后缀的方法, 该方法的区别在于拼接SQL语句的时候, 会对实体类中的每个属性值进行判断是否为null(这也就是为什么逆向工程的实体类中的变量类型是每种基本数据类型对应的包装对象的原因), 如果为null的话该属性对应的字段将不拼接到SQL语句当中。如:

updateByPrimaryKey

```
@Test
public void updateByPrimaryKey() {
    Student student = new Student();
    student.setId(12);
    student.setName("SB");
    student.setAge((byte) 8);
    student.setTid(null);
    studentMapper.updateByPrimaryKeySelective(student);
}
```

拼接的SQL语句

```
11:11:00,328 DEBUG updateByPrimaryKey:139 - ==> Preparing: update student set name = ?, age = ?, tid = ? where id = ?
11:11:00,369 DEBUG updateByPrimaryKey:139 - ==> Parameters: SB(String), 8(Byte), null, 12(Integer)
11:11:00,371 DEBUG updateByPrimaryKey:139 - <= Updates: 1
```

updateByPrimaryKeySelective

```
@Test
public void updateByPrimaryKeySelective() {
    Student student = new Student();
    student.setId(12);
    student.setName("SB");
    student.setAge((byte) 7);
    student.setTid(null);
    studentMapper.updateByPrimaryKeySelective(student);
}
```

拼接的SQL语句

```
11:08:04,057 DEBUG updateByPrimaryKeySelective:139 - ==> Preparing: update student SET name = ?, age = ? where id = ?
11:08:04,106 DEBUG updateByPrimaryKeySelective:139 - ==> Parameters: SB (String), 7 (Byte), 12 (Integer)
11:08:04,108 DEBUG updateByPrimaryKeySelective:139 - <== Updates: 1
```

在进行更新操作时，如实体类中的属性值为null使用Selective的方法不会影响数据库中的字段。而普通的更新方法将会直接把其中的数据库字段值设置为null。

Exmaple

在生成数据库表结构对应的实体类时都附带生成一个xxxExmaple类，有时逆向工程生成的SQL文件并不能满足实际的需求，如需要进行高级的查询操作。这时xxxExmaple类就能够满足这类需求。

模糊查询

```
@Test
public void selectByExample() {
    StudentExample studentExample = new StudentExample();
    StudentExample.Criteria criteria = studentExample.createCriteria();
    criteria.andNameLike("S%");
    List<Student> students =studentMapper.selectByExample(studentExample);
    for (Student student : students) {
        System.out.println(student);
    }
}
```

其中Criteria是设置查询条件参数对象，其中对每个对象字段都有相应的高级操作方法，如：

andNameIsNull ()

andNameIsNotNull ()

andNameEqualTo ()

andNameNotEqualTo ()

andNameGreaterThan ()

...

每个查询条件都有一个Criteria对象，如果where条件中出现or，那么就需要有两个Criteria对象进行条件进行包装，最后再通过xxxExmaple对象进行串联，如：

```
@Test
public void selectByExample() {
    StudentExample studentExample = new StudentExample();
    StudentExample.Criteria criteria1 = studentExample.createCriteria();
    criteria1.andAgeBetween((byte) 1, (byte) 5);
    StudentExample.Criteria criteria2 = studentExample.createCriteria();
    criteria2.andNameLike("S%");
    studentExample.or(criteria2);
    List<Student> students = studentMapper.selectByExample(studentExample);
    for (Student student : students) {
        System.out.println(student);
    }
}
```