

1、Mybatis的发展史

MyBatis 本是apache的一个开源项目iBatis, 2010年这个项目由apache software foundation 迁移到了google code, 并且改名为MyBatis。2013年11月迁移到

Github, 通俗说法Ibatis3 = MyBatis,iBatis一词来源于“internet”和“abatis”的组合, 是一个基于Java的持久层框架。iBatis提供的持久层框架包括SQL Maps和

Data Access Objects (DAO)



MyBatis

2、什么是 Mybatis

MyBatis是一个数据持久层(ORM)框架。把实体类和SQL语句之间建立了映射关系, 是一种半自动化的ORM实现。

3、MyBatis的优点:

- 基于SQL语法, 简单易学
- 能了解底层组装过程
- SQL语句封装在配置文件中, 便于统一管理与维护, 降低了程序的耦合度
- 程序调试方便

MyBatis和JDBC的区别:

<pre>Class.forName("com.mysql.jdbc.Driver"); Connection con = DriverManager.getConnection("jdbc:mysql://127.0.0.1:3306/MyBatisDemo", "root", "root"); String sql = "select * from tuser where userId = ?"; PreparedStatement pstmt = con.prepareStatement(sql); pstmt.setInt(1, 1); ResultSet rs = pstmt.executeQuery(); while (rs.next()) { System.out.println(rs.getString("userId")); }</pre>	JDBC数据访问流程
<pre><select id="findAllUser" parameterType="int" resultType="com.chinasofti.etc.mybatisdemo.domain.UserEntity"> select * from tuser where userId = #{userId} </select></pre>	使用MyBatis之后对应的查询配置

- 减少了大量的代码量
- 最简单的持久化框架
- 架构级性能增强
- SQL代码从程序代码中彻底分离, 可重用
- 增强了项目中的分工
- 增强了移植性

4、Mybatis的结构特性

- MyBatis的功能架构分为三层：

- API接口层： 声明一些列方法： 添加、删除、修改

- 提供给外部使用的接口API，开发人员通过这些本地API来操纵数据库。接口层一接收到调用请求就会调用数据处理层来完成具体的数据处理

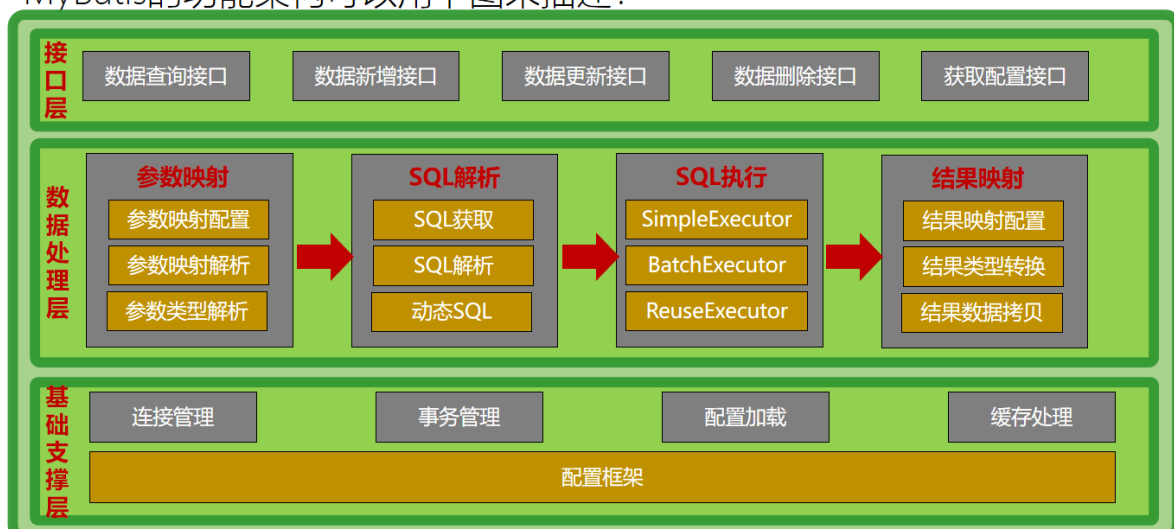
- 数据处理层： 就是具体的操作，执行数据库中的sql的

- 负责具体的SQL查找、SQL解析、SQL执行和执行结果映射处理等。它主要的目的是根据调用的请求完成一次数据库操作

- 基础支撑层： 用来连接数据库的

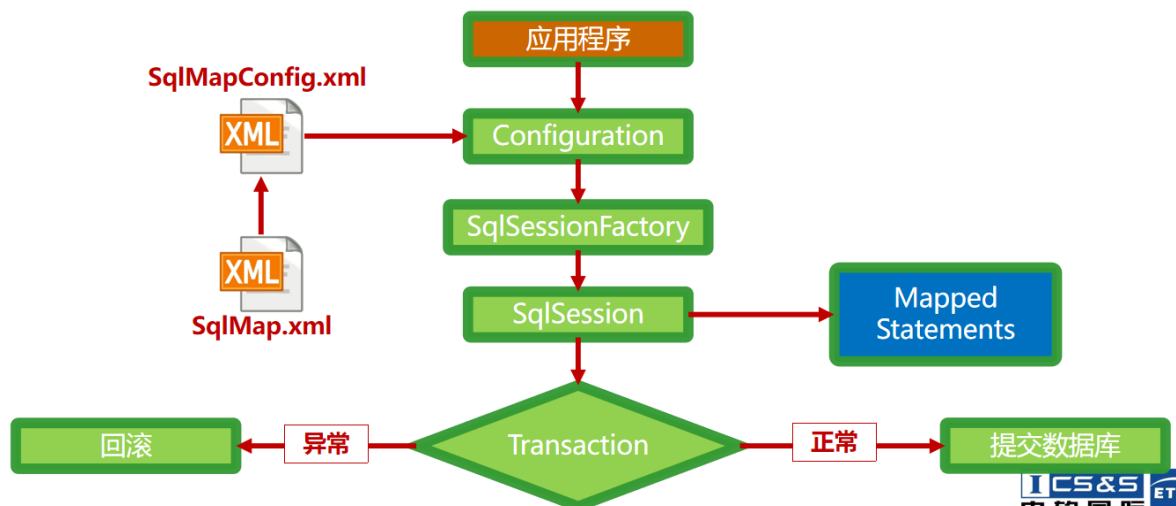
- 负责最基础的功能支撑，包括连接管理、事务管理、配置加载和缓存处理，这些都是共用的东西，将他们抽取出来作为最基础的组件。为上层的数据处理层提供最基础的支撑

- MyBatis的功能架构可以用下图来描述：



5、MyBatis基础代码结构

- 在代码中使用MyBatis的基础结构流程如下：



6、MyBatis核心配置文件

- configuration.xml (文件名可以自定义) 是系统的核心配置文件, 包含数据源和事务管理等设置和属性信息, XML文档结构如下:
- configuration 配置
 - properties 可以配置在Java 属性配置文件中
 - settings 修改 MyBatis 在运行时的行为方式
 - typeAliases 为 Java 类型命名一个短的名字
 - typeHandlers 类型处理器
 - objectFactory 对象工厂
 - plugins 插件
 - **environments 环境 (MyBatis的基础配置内容, 包含了JDBC数据源相关参数等)**
 - environment 环境变量
 - transactionManager 事务管理器
 - dataSource 数据源
 - mappers
 - 映射器

按照顺序写

基础环境配置示例:

```
<configuration>
  <environments default="development">
    <environment id="development">
      <transactionManager type="JDBC"/>
      <dataSource type="POOLED">
        <property name="driver" value="${driver}"/>
        <property name="url" value="${url}"/>
        <property name="username" value="${username}"/>
        <property name="password" value="${password}"/>
      </dataSource>
    </environment>
    <environment id="development2">
      .....
    </environment>
  </environments>
</configuration>
```

事务管理类型

数据源及类型

JDBC参数

• MyBatis 有两种事务管理类型:

• JDBC:

- 这个类型直接全部使用 JDBC 的提交和回滚功能。它依靠使用连接的数据源来管理事务的作用域

• MANAGED:

- 这个类型什么都不做, 它从不提交、回滚和关闭连接, 而是委托给第三方管理事务的全部生命周期 (如Spring或者JavaEE服务器)

数据源类型有三种：UNPOOLED，POOLED，JNDI：

- UNPOOLED：
 - 这个数据源实现只是在每次请求的时候简单的打开和关闭一个连接。虽然这有点慢，但作为一些不需要性能和立即响应的简单应用来说，不失为一种好选择
- POOLED：
 - 这个数据源缓存 JDBC 连接对象用于避免每次都要连接和生成连接实例而需要的验证时间。对于并发 WEB 应用，这种方式非常流行因为它有最快的响应时间
- JNDI：
 - 这个数据源实现是为了准备和JavaEE应用服务一起使用，可以在外部也可以在内部配置这个数据源，然后在 JNDI 上下文中引用它

7、Mybatis基本要素

MyBatis基本要素

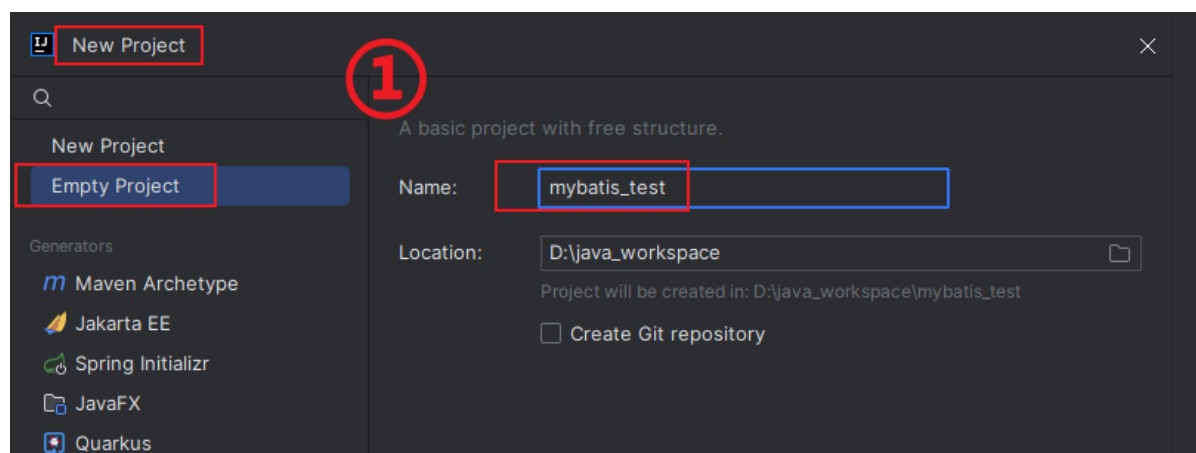
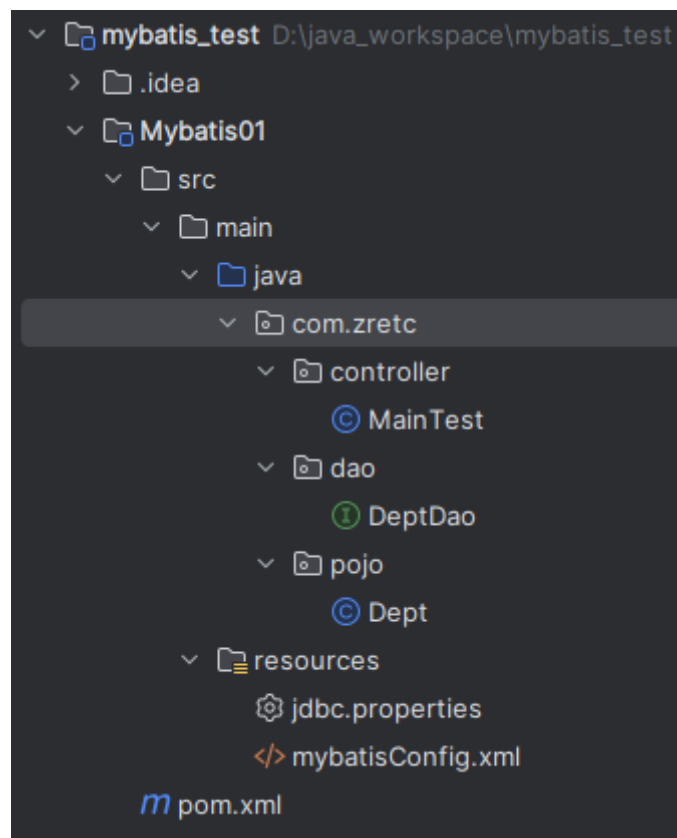
- 一、configuration.xml 全局配置文件
- 二、导入生成的文件
 - Mapper.xml存放sql语句
 - Mapper接口用于执行xml中的sql语句
 - Model：对应数据库表的对象，属性一致
- 三、获取SqlSession接口

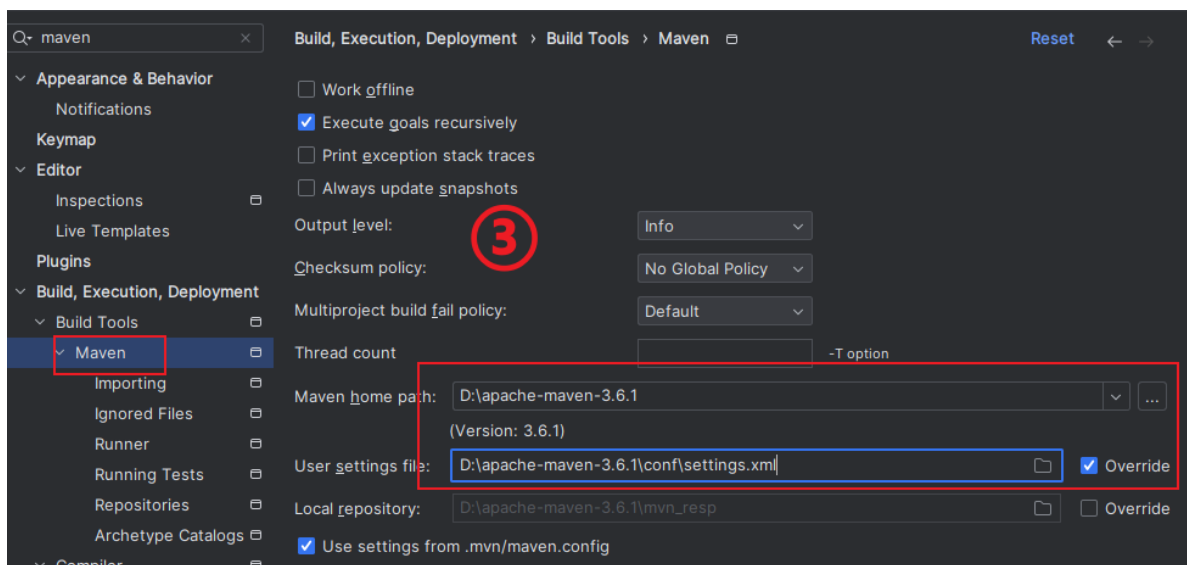
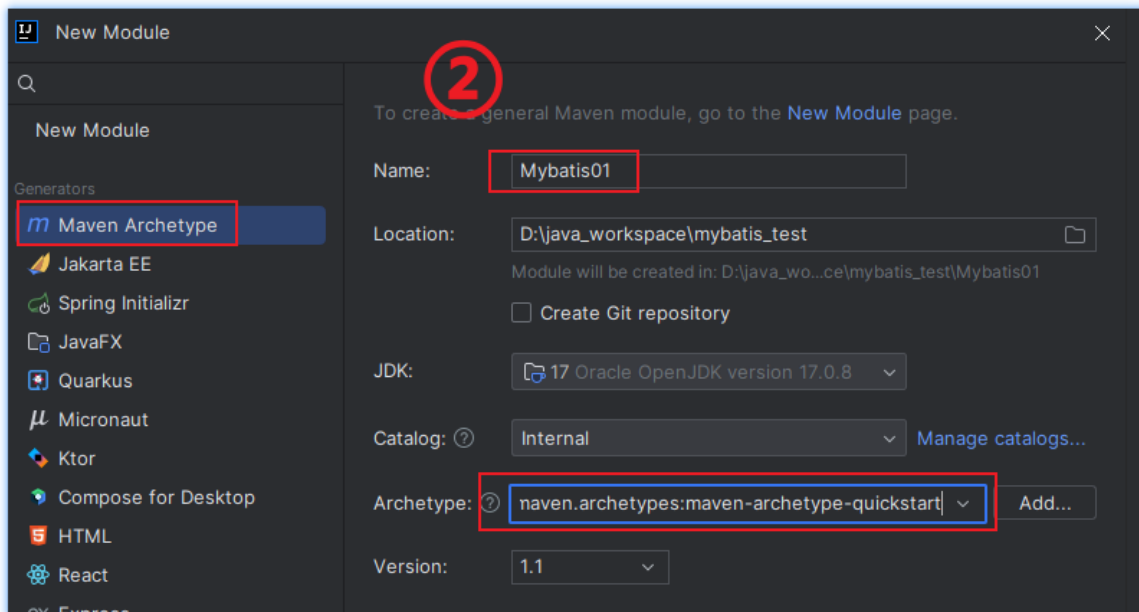
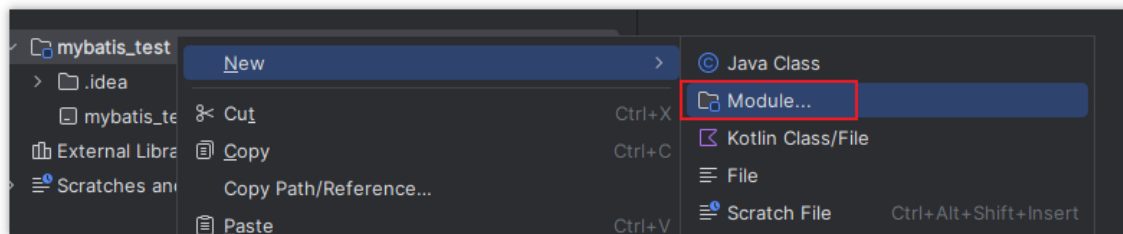
```
InputStream is = Resources.getResourceAsStream(filename);  
SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(is);  
SqlSession session = factory.openSession();
```

- 四、根据mapper接口执行方法调用sql，获取执行结果

```
StudentMapper mapper = session.getMapper(StudentMapper.class);  
Student student = mapper.selectByPrimaryKey(1);  
System.out.println(student);
```

8、创建一个Mybatis项目





导入依赖:

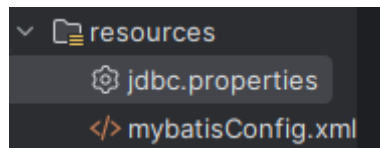
```
<dependencies>
  <!-- 导入mybatis依赖 -->
  <dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis</artifactId>
    <version>3.5.14</version>
  </dependency>
  <!-- 驱动 -->
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.33</version>
  </dependency>
</dependencies>
```

```

<!-- junit -->
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.13.2</version>
</dependency>
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <version>1.18.30</version>
</dependency>
</dependencies>

```

定义属性文件：

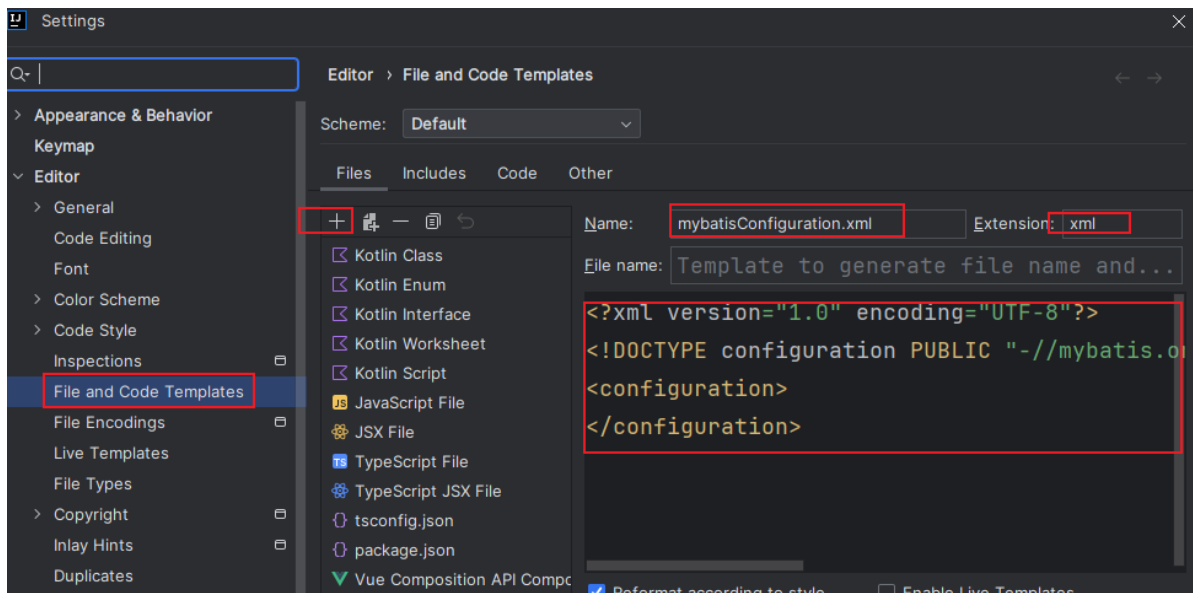


```

jdbc.driver=com.mysql.cj.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/zly
jdbc.user=root
jdbc.password=root

```

创建一个mybatis的配置文件：



```

<?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>

```

将属性配置文件引入到mybatis总配置文件中：

```
<?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>
    <!-- 引入属性文件 -->
    <properties resource="jdbc.properties"/>
    <!-- 基础环境 -->
    <environments default="development">
        <environment id="development">
            <!-- 事务管理器 -->
            <transactionManager type="JDBC"/>
            <!-- 数据源 -->
            <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>
</configuration>
```

实体类：

```
@Data
public class Dept {
    private Integer dept_id;
    private String dept_name;
    private String dept_loc;
}
```

在总配置文件中给实体类起别名：

```
mybatisConfig.xml x DeptDao.xml
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN" "http://
3 <configuration>
4     <!-- 引入属性文件 -->
5     <properties resource="jdbc.properties"/>
6     <!-- 给实体类起别名 -->
7     <typeAliases>
8         <!-- 挨个实体类起别名 -->
9         <!-- <typeAlias type="com.zretc.pojo.Dept" alias="dept"/>-->
10        <!-- 给com.zretc.pojo下的所有实体类起别名,默认就是类名首字母小写 -->
11        <package name="com.zretc.pojo"/>
12    </typeAliases>
13    <!-- 基础环境 -->
14    <environments default="development">
15        <environment id="development">
```



```

<!-- 给实体类起别名 -->
<typeAliases>
    <!-- 挨个实体类起别名 -->
    <!-- <typeAlias type="com.zretc.pojo.Dept" alias="dept"/>-->
    <!-- 给com.zretc.pojo下的所有实体类起别名,默认就是类名首字母小写 -->
    <package name="com.zretc.pojo"/>
</typeAliases>

```

dao层接口

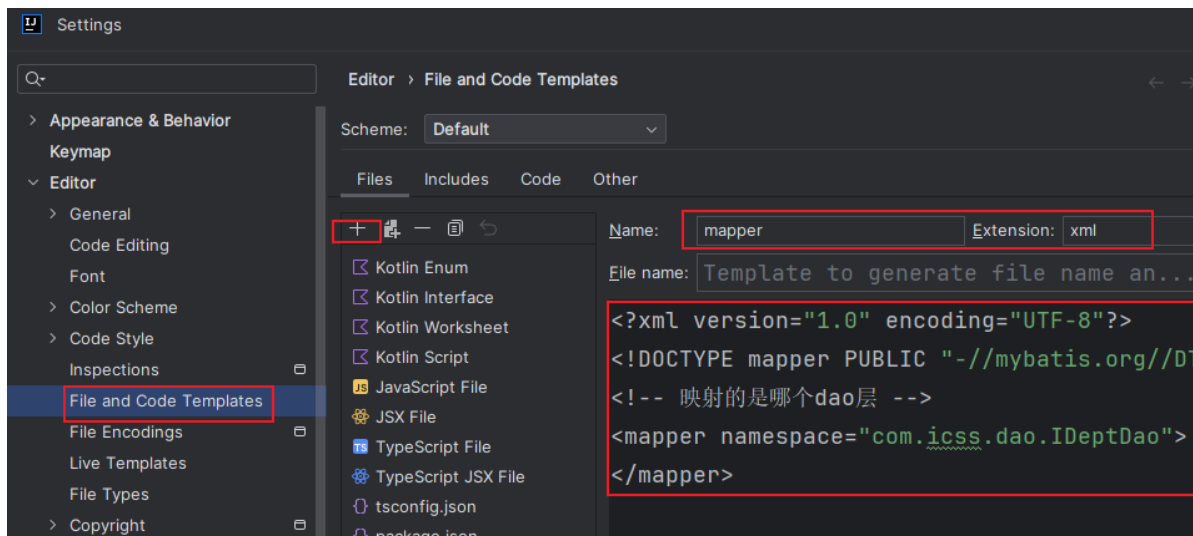
```

public interface DeptDao {
    int insertDept(Dept dept);
    int deleteDept(Integer dept_id);
    int updateDept(Dept dept);
    List<Dept> selectAll();
    Dept selectById(Integer dept_id);
}

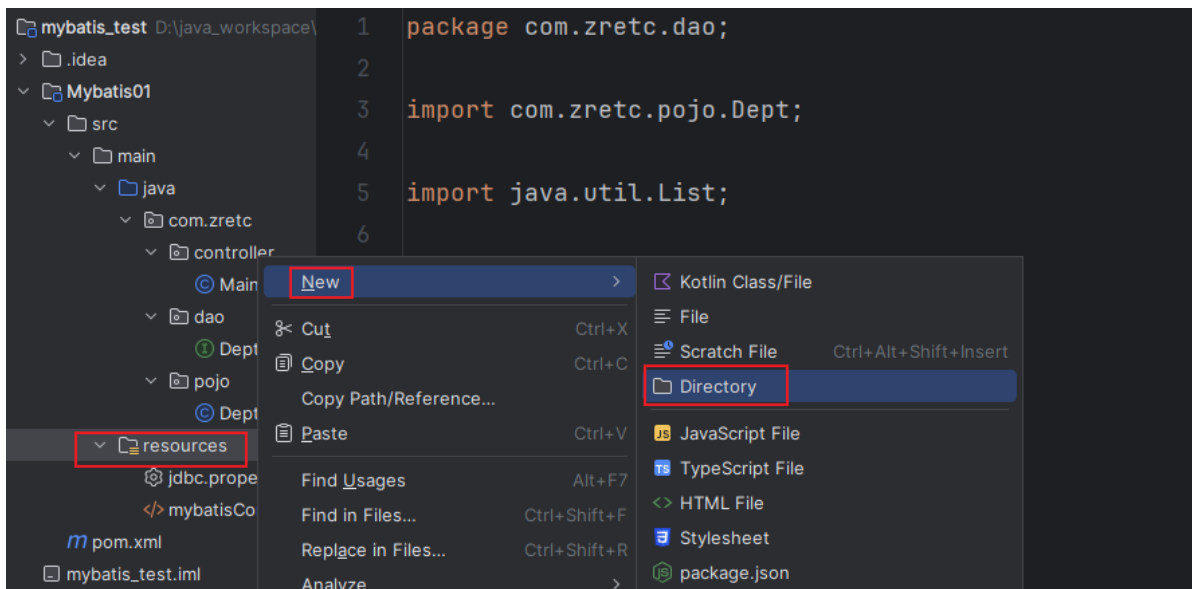
```

现在不需要dao层接口的实现类了，而是需要一个mapper文件

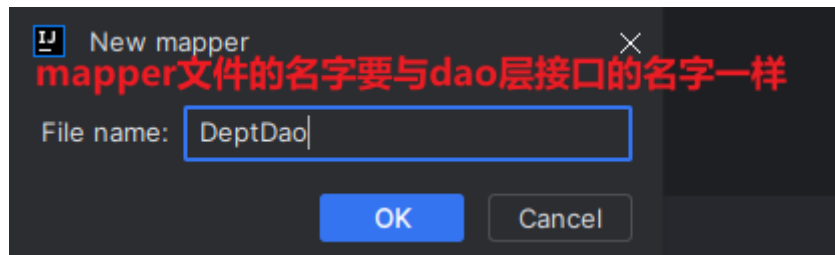
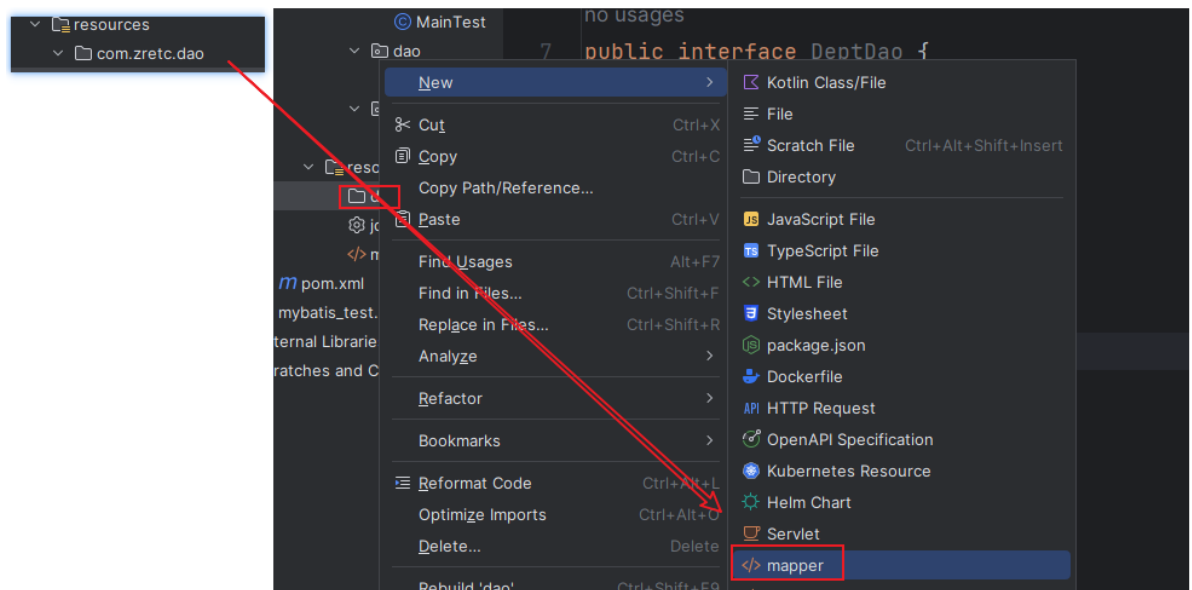
一个dao接口对应一个mapper文件



第一步，先创建一个目录，存储mapper文件：



在包下新建mapper文件



在mapper文件中编写接口中对应的sql语句

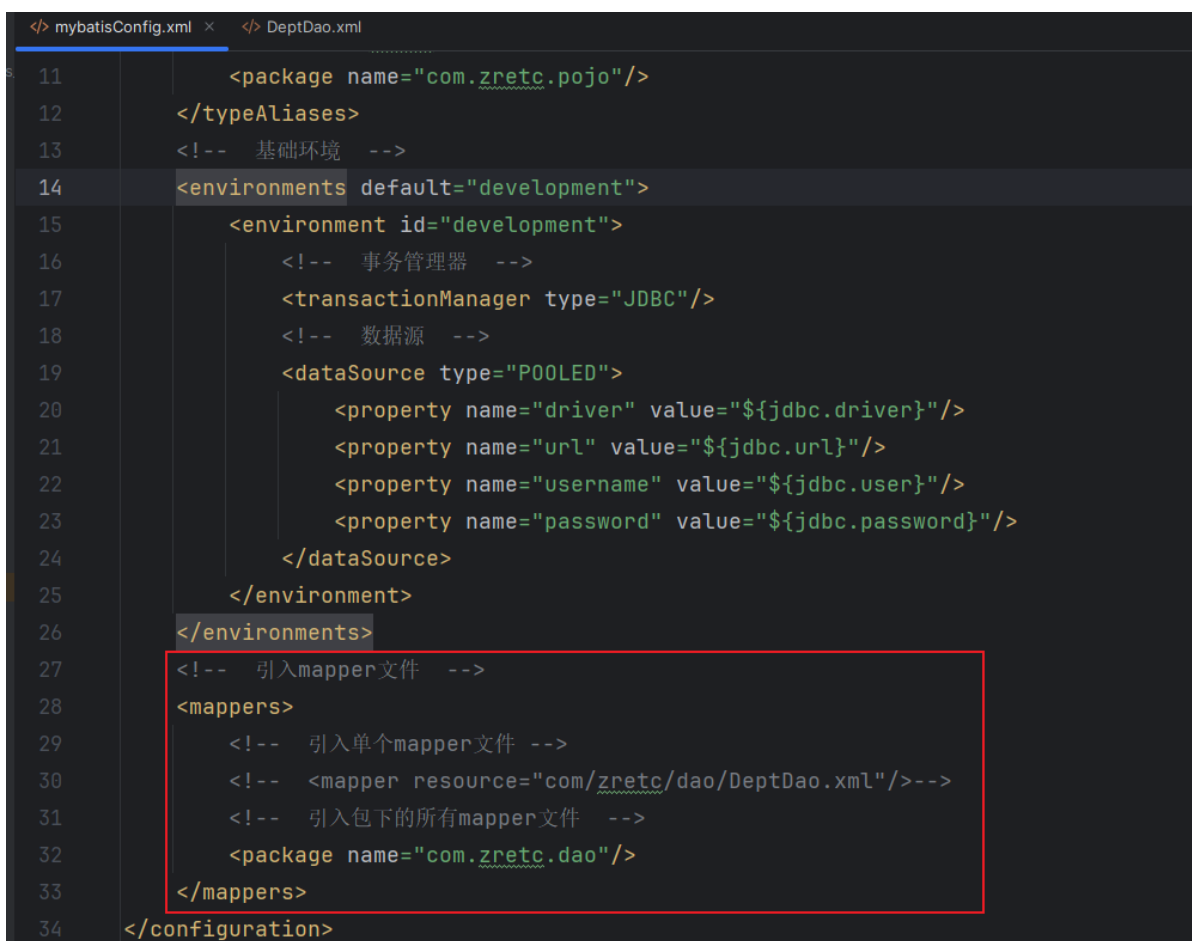
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd" >
<!-- 映射的是哪个dao层 -->
<mapper namespace="com.zretc.dao.DeptDao">
```

```

<!--
<insert> 表签代表的是添加操作
id 对应的是dao层接口中的函数名称
parameterType 函数的参数类型，默认是包名+类名映射，
                可以在总配置文件中使用typeAliases起别名，直接引用别名
                如果参数是一个基本类型，可以省略不写
#{ } 代表的是将实体类中的成员变量映射给表中的字段，相当于？
jdbcType=VARCHAR 该数据对应的数据库中的字段的类型，可以省略不写
-->
<insert id="insertDept" parameterType="dept">
    insert into dept
    values (default, #{dept_name,jdbcType=VARCHAR}, #{dept_loc})
</insert>
</mapper>

```

将mapper文件，引入到总配置文件中：



```

11      <package name="com.zretc.pojo"/>
12  </typeAliases>
13  <!-- 基础环境 -->
14  <environments default="development">
15      <environment id="development">
16          <!-- 事务管理器 -->
17          <transactionManager type="JDBC"/>
18          <!-- 数据源 -->
19          <dataSource type="POOLED">
20              <property name="driver" value="${jdbc.driver}"/>
21              <property name="url" value="${jdbc.url}"/>
22              <property name="username" value="${jdbc.user}"/>
23              <property name="password" value="${jdbc.password}"/>
24          </dataSource>
25      </environment>
26  </environments>
27  <!-- 引入mapper文件 -->
28  <mappers>
29      <!-- 引入单个mapper文件 -->
30      <!-- <mapper resource="com/zretc/dao/DeptDao.xml"/>-->
31      <!-- 引入包下的所有mapper文件 -->
32      <package name="com.zretc.dao"/>
33  </mappers>
34 </configuration>

```

```

<!-- 引入mapper文件 -->
<mappers>
    <!-- 引入单个mapper文件 -->
    <!-- <mapper resource="com/zretc/dao/DeptDao.xml"/>-->
    <!-- 引入包下的所有mapper文件 -->
    <package name="com.zretc.dao"/>
</mappers>

```

在测试类中，获取接口对象，调用添加方法：

```

public class MainTest {
    @Test
    public void test01() throws IOException {
        // 获取mybatis的总配置文件
        // 1. 读取mybatis总配置文件
        Reader reader = Resources.getResourceAsReader("mybatisConfig.xml");
        // 2.获取SqlSessionFactory工厂
        SqlSessionFactory sessionFactory = new
        SqlSessionFactoryBuilder().build(reader);
        // 3.获取SqlSession
        SqlSession sqlSession = sessionFactory.openSession(true); // true:自动提交
        // 4.获取接口对象
        DeptDao deptDao = sqlSession.getMapper(DeptDao.class);
        // 5.调用添加方法
        Dept dept = new Dept();
        dept.setDept_name("生产部");
        dept.setDept_loc("1楼101车间");
        deptDao.insertDept(dept);

        // 6.事务的提交
        // sqlSession.commit();
        // 7.关闭SqlSession
        sqlSession.close();
    }
}

```



dept_id	dept_name	dept_loc
1	销售部	一楼
2	市场部	二楼
12	406教室	406
13	生产部	1楼101车间

在mapper文件中配置其他方法:

```

public interface DeptDao {
    int insertDept(Dept dept);
    // @Param("id") 给参数起别名, 在mapper文件中映射
    // int deleteDept(@Param("id") Integer dept_id);
    int deleteDept(Integer dept_id);
    int updateDept(Dept dept);
    List<Dept> selectAll();
    //Dept selectById(@Param("id") Integer dept_id);
    Dept selectById(Integer dept_id);
    List<Dept> selectByLike(@Param("dept_name") String dept_name);
}

```

```

<!-- 删除功能
        当参数不是一个实体类时, 映射方式:
            1、利用@param()映射表中的字段
            2、使用参数的下标 (从0开始的) 映射
-->
<delete id="deleteDept" parameterType="java.lang.Integer">
    <!--delete from dept where dept_id=#{id}-->
    delete from dept where dept_id=#{0}
</delete>
<!-- 修改功能 -->
<update id="updateDept" parameterType="dept">
    update dept
    set dept_name=#{dept_name},
        dept_loc=#{dept_loc}
    where dept_id = #{dept_id}
</update>

<!-- 查询所有 -->
<select id="selectAll" resultType="dept">
    select * from dept
</select>

<!-- 根据部门编号查询所有 -->
<select id="selectById" resultType="dept">
    select * from dept where dept_id = #{dept_id}
</select>

<!-- 模糊查询
        使用 concat() 函数进行字符串拼接, 该函数的参数是可变的
-->
<select id="selectByLike" resultType="dept">
    select * from dept where dept_name like concat('%',#{dept_name},'%')
</select>

```

测试类中进行测试:

```

// 删除功能
//      deptDao.deleteDept(15);

// 修改

```

```
//      Dept dept = new Dept();
//      dept.setDept_id(14);
//      dept.setDept_name("生产部666");
//      dept.setDept_loc("1楼106车间");
//      deptDao.updateDept(dept);

// 查询所有
//      List<Dept> depts = deptDao.selectAll();
//      depts.forEach(dept -> System.out.println(dept));

// 根据部门编号查询
//      Dept dept = deptDao.selectById(14);
//      System.out.println(dept);

// 模糊查询
List<Dept> depts = deptDao.selectByLike("6");
depts.forEach(dept -> System.out.println(dept));
```

9、SqlSession对象

SqlSession在 MyBatis中是非常强大的一个核心类，类中包含了所有执行语句的方法,提交或回滚事务,还有获取映射器实例的方法

语句执行方法

- 这些方法被用来执行定义在 SQL 映射的XML文件中的 SELECT , INSERT , UPDAET和DELETE语句。执行时都使用语句的ID属性和参数对象,参数可以是原生类型(自动装箱或包装类),JavaBean,POJO或Map

语句执行方法
<T> T selectOne(String statement, [,Object parameter])
<E> List<E> selectList(String statement[,Object parameter])
<K,V> Map<K,V> selectMap(String statement[,Object parameter],String mapKey)
int insert(String statement[,Object parameter])
int update(String statement[,Object parameter])
int delete(String statement[,Object parameter])

- session.insert(“Mapper接口完整路径”， 参数);
- session.insert("cn. etc. Dao. StudentDao. insertStudent", stu);
- session.update(“Mapper接口完整路径”， 参数);
- session.update("cn. etc. Dao. StudentDao. updateStudent", stu);
- session.delete(“Mapper接口完整路径”， 参数);
- session.delete("cn. etc. Dao. StudentDao. deleteStudent", 1);

- 返回一个对象：`session.selectOne(" Mapper接口完整路径", 参数);`
- `Student stus=session.selectOne("cn.etc.Dao.StudentDao.SearchById", 1);`
- 返回一个集合：`session.selectList(" Mapper接口完整路径", 参数);`
- `List<Student> listAll=session.selectList("cn.etc.Dao.StudentDao.SearchAll");`
- `session.selectMap(" Mapper接口完整路径","实体类中某个String类型属性作为key");`
- `Map<String,Student> mapAll=session.selectMap("cn.etc.Dao.StudentDao.SearchAll","name");`
- 由于map的key不能重复，当name作为key值时，会自动将重名对象忽略
- `for(String stud:mapAll.keySet()){`
 - `System.out.println(mapAll.get(stud));`
- `}`

```

@Test
public void test02() throws IOException {

    // 获取mybatis的总配置文件
    // 1. 读取mybatis总配置文件
    Reader reader = Resources.getResourceAsReader("mybatisConfig.xml");
    // 2.获取SqlSessionFactory工厂
    SqlSessionFactory sessionFactory = new
    SqlSessionFactoryBuilder().build(reader);
    // 3.获取SqlSession
    SqlSession sqlSession = sessionFactory.openSession(true); // true:自动提交

    // 利用SqlSession来执行sql
    // 1.添加
    //      Dept dept = new Dept();
    //      dept.setDept_name("打包部");
    //      dept.setDept_loc("1楼102车间");
    //      sqlSession.insert("com.zretc.dao.DeptDao.insertDept",dept);

    // 2.删除
    //      sqlSession.delete("com.zretc.dao.DeptDao.deleteDept",16);

    // 3.修改
    //      Dept dept = new Dept();
    //      dept.setDept_id(14);
    //      dept.setDept_name("打包部");
    //      dept.setDept_loc("1楼102车间");
    //      sqlSession.update("com.zretc.dao.DeptDao.updateDept",dept);

    //      4. 根据id查询
    //      Dept dept =
    sqlSession.selectOne("com.zretc.dao.DeptDao.selectById",14);
    //      System.out.println(dept);

    // 5. 查询所有
    //      List<Dept> depts =
    sqlSession.selectList("com.zretc.dao.DeptDao.selectAll");
    //      depts.forEach(dept -> System.out.println(dept));

    // 6.模糊查询

```

```
//      List<Dept> depts =
sqlSession.selectList("com.zretc.dao.DeptDao.selectByLike","部");
//      depts.forEach(dept -> System.out.println(dept));

// 7. selectMap 将部门名称作为键
Map<String,Dept> deptMap =
sqlSession.selectMap("com.zretc.dao.DeptDao.selectAll","dept_name");
for (String s : deptMap.keySet()) {
    System.out.println(s + ":" + deptMap.get(s));
}

sqlSession.close();
}
```

- RowBounds 参数会告诉 MyBatis 略过指定数量的记录,还有限制返回结果的数量。 RowBounds 类有一个构造方法来接收 offset 和 limit:
- RowBounds rb=new RowBounds(略过的数量,返回的数量);
- RowBounds rb=new RowBounds(2,3);//略过两条数据， 返回3条数据

```
RowBounds rb=new RowBounds(1,4);//略过一条数据， 返回4条数据
List<Student> li=session.selectList("cn.etc.Dao.StudentDao.SearchAll",null, rb);

for (Student student : li) {
    System.out.println(student);
}
```

```
// 8、分页查询
RowBounds rb = new RowBounds(2,3);
List<Dept> depts =
sqlSession.selectList("com.zretc.dao.DeptDao.selectAll",null,rb);
depts.forEach(dept -> System.out.println(dept));
```

```
==> Parameters:
<==      Columns: dept_id, dept_name, dept_loc
<==      Row: 1, 销售部, 一楼
<==      Row: 2, 市场部, 二楼
<==      Row: 12, 406教室, 406
<==      Row: 13, 生产部, 1楼101车间
<==      Row: 14, 打包部, 1楼102车间
Dept(dept_id=12, dept_name=406教室, dept_loc=406)
Dept(dept_id=13, dept_name=生产部, dept_loc=1楼101车间)
Dept(dept_id=14, dept_name=打包部, dept_loc=1楼102车间)
```

10、动态sql

- <trim prefix="(" suffix=")" suffixOverrides=",">
 - trim 代表的是空格

- prefix="(" 以指定字符开头
- suffix=")" 以指定字符结尾
- suffixOverrides="," 去掉最后一个指定的字符
- <if test="">

- test 是条件表达式

```
<insert id="insertEmp" parameterType="emp">
  <!-- insert into emp(ename,hire_date,job_id,dept_id,interest)
  values('tom',now(),1,1,'唱歌,跳舞')-->
  <!--
    trim 代表的是空格
    prefix="(" 以指定字符开头
    suffix=")" 以指定字符结尾
    suffixOverrides="," 去掉最后一个指定的字符
  -->
  insert into emp
  <trim prefix="(" suffix=")" suffixOverrides=",">
    <if test="ename!=null">ename,</if>
    <if test="hire_date!=null">hire_date,</if>
    <if test="job_id!=null">job_id,</if>
    <if test="dept_id!=null">dept_id,</if>
    <if test="interest!=null">interest,</if>
  </trim>
  <trim prefix="values(" suffix=")" suffixOverrides=",">
    <if test="ename!=null">#{ename},</if>
    <if test="hire_date!=null">#{hire_date},</if>
    <if test="job_id!=null">#{job_id},</if>
    <if test="dept_id!=null">#{dept_id},</if>
    <if test="interest!=null">#{interest},</if>
  </trim>

  <!-- insert into emp(ename,hire_date) values('soso',now())-->
</insert>
```

- <set> 代替修改语句中的set关键字

```
<update id="updateEmp" parameterType="emp">
  update emp
  <set>
    <if test="ename!=null">ename=#{ename},</if>
    <if test="hire_date!=null">hire_date=#{hire_date},</if>
    <if test="job_id!=null">job_id=#{job_id},</if>
    <if test="dept_id!=null">dept_id=#{dept_id},</if>
    <if test="interest!=null">interest=#{interest},</if>
  </set>
  where empno = #{empno}
</update>
```

- <foreach> 遍历循环

- `collection` 代表的是集合
`item` 集合中的元素
`open` 以指定字符开始
`close` 以指定字符结束
`separator` 分隔符

```
<foreach collection="list" item="empno" open="(" close=")"
separator=",">
    #{empno}
</foreach>
```

- `<![CDATA[sql]]>`

- mapper文件中写大于号或者小于号时，需要将sql语句放在 `<![CDATA[sql]]>` 标记中


```
<![CDATA[
          select * from emp where empno < #{0}
      ]]>
```

- `<where>` 代替条件中的where 关键字,可以自动去掉前缀或后缀

- ```
<choose>
<when>
<otherwise>
<!-- 根据任意条件查询 -->
 <select id="selectByOther" resultType="emp" parameterType="emp">
 select * from emp
 <where>
 <!-- 相当于 if... else if... else... -->
 <choose>
 <when test="empno!=null">empno=#{empno}</when>
 <when test="ename!=null">ename=#{ename}</when>
 <when test="hire_date!=null">hire_date=#{hire_date}
 </when>

 <when test="job_id!=null">job_id=#{job_id}</when>
 <when test="dept_id!=null">dept_id=#{dept_id}</when>
 <when test="interest!=null">interest=#{interest}</when>
 <otherwise>1=1</otherwise>
 </choose>
 </where>
 </select>
```

- `<sql>`

- ```
<sql id="usually">
    empno,ename,hire_date,job_id,dept_id,interest
</sql>
select <include refid="usually"/> from emp
```

pojo实体类:

```

@Data
public class Emp {
    private Integer empno;
    private String ename;
    private Date hire_date;
    private Integer job_id;
    private Integer dept_id;
    private String interest;
}

```

dao层接口:

```

package com.zretc.dao;

import com.zretc.pojo.Emp;

import java.util.List;

public interface EmpDao {
    // 添加
    int insertEmp(Emp emp);
    // 修改
    int updateEmp(Emp emp);
    // 删除
    int deleteEmp(Integer empno);
    // 查询所有
    List<Emp> selectAll();
    // 根据编号查询
    Emp selectByNo(Integer empno);

    // 查询员工信息, 员工编号等于值列表中的一个 (例如: 编号在1/2/6的员工)
    List<Emp> selectByIn(List<Integer> empno);

    // 查询员工编号是6以内的员工信息
    List<Emp> selectByBetween(Integer empno);

    // 根据任意条件查询
    List<Emp> selectByOther(Emp emp);

    // 多个条件
    List<Emp> selectByAllOther(Emp emp);
}

```

mapper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" >
<!-- 映射的是哪个dao层 -->
<mapper namespace="com.zretc.dao.EmpDao">
    <!-- 添加 -->
    <insert id="insertEmp" parameterType="emp">
        <!-- insert into emp(ename,hire_date,job_id,dept_id,interest)

```

```

values('tom',now(),1,1,'唱歌,跳舞')-->
<!--
    trim 代表的是空格
    prefix="(" 以指定字符开头
    suffix=*)" 以指定字符结尾
    suffixOverrides=", " 去掉最后一个指定的字符
-->
insert into emp
<trim prefix="(" suffix=*)" suffixOverrides=", ">
    <if test="ename!=null">ename,</if>
    <if test="hire_date!=null">hire_date,</if>
    <if test="job_id!=null">job_id,</if>
    <if test="dept_id!=null">dept_id,</if>
    <if test="interest!=null">interest,</if>
</trim>
<trim prefix="values(" suffix=*)" suffixOverrides=", ">
    <if test="ename!=null">#{ename},</if>
    <if test="hire_date!=null">#{hire_date},</if>
    <if test="job_id!=null">#{job_id},</if>
    <if test="dept_id!=null">#{dept_id},</if>
    <if test="interest!=null">#{interest},</if>
</trim>

<!-- insert into emp(ename,hire_date) values('soso',now())-->
</insert>

<!--修改-->
<update id="updateEmp" parameterType="emp">
    update emp
    <set>
        <if test="ename!=null">ename=#{ename},</if>
        <if test="hire_date!=null">hire_date=#{hire_date},</if>
        <if test="job_id!=null">job_id=#{job_id},</if>
        <if test="dept_id!=null">dept_id=#{dept_id},</if>
        <if test="interest!=null">interest=#{interest},</if>
    </set>
    where empno = #{empno}
</update>

<!--删除-->
<delete id="deleteEmp">
    delete from emp where empno = #{0}
</delete>

<!-- 查询所有 -->
<select id="selectAll" resultType="emp">
    select * from emp
</select>

<!-- 根据id查询 -->
<select id="selectByNo" resultType="emp">
    select * from emp where empno=#{0}
</select>

<!--查询员工信息, 员工编号等于值列表中的一个(例如: 编号在1/2/6的员工)-->
<select id="selectByIn" resultType="emp">
    select * from emp where empno in
    <!--

```

```

        collection 代表的是集合
        item 集合中的元素
        open 以指定字符开始
        close 以指定字符结束
        separator 分隔符
    -->
    <foreach collection="list" item="empno" open="(" close=")"
separator=",">
        #{empno}
    </foreach>
</select>

<!-- 查询员工编号是6以内的员工信息 -->
<select id="selectByBetween" resultType="emp">
    <!-- mapper文件中写大于号或者小于号时，需要将sql语句放在 <![CDATA[ sql ]]> 标记
中-->
    <![CDATA[
        select * from emp where empno < #{0}
    ]]>
</select>

<!-- 根据任意条件查询 -->
<select id="selectByOther" resultType="emp" parameterType="emp">
    select * from emp
    <where>
        <!-- 相当于 if... else if... else... -->
        <choose>
            <when test="empno!=null">empno=#{empno}</when>
            <when test="ename!=null">ename=#{ename}</when>
            <when test="hire_date!=null">hire_date=#{hire_date}</when>
            <when test="job_id!=null">job_id=#{job_id}</when>
            <when test="dept_id!=null">dept_id=#{dept_id}</when>
            <when test="interest!=null">interest=#{interest}</when>
            <otherwise>1=1</otherwise>
        </choose>
    </where>
</select>

<!-- 通用的字段 -->
<sql id="usually">
    empno,ename,hire_date,job_id,dept_id,interest
</sql>

<!-- 多个条件 -->
<select id="selectByAllOther" parameterType="emp" resultType="emp">
    <!-- select * from emp where ename='tom' and hire_date='2025-3-7' -->
    <!-- select * from emp where ename='tom' or hire_date='2025-3-7' -->

    <!-- 引用通用的sql -->
    select <include refid="usually"/> from emp
    <!-- where会总去掉前缀和后缀 -->
    <where>
        <if test="empno!=null">and empno=#{empno}</if>
        <if test="ename!=null">and ename=#{ename}</if>
        <if test="hire_date!=null">and hire_date=#{hire_date}</if>
        <if test="job_id!=null">and job_id=#{job_id}</if>
        <if test="dept_id!=null">and dept_id=#{dept_id}</if>
        <if test="interest!=null">and interest=#{interest}</if>
    </where>

```

```
</select>
</mapper>
```

测试类：

```
package com.zretc.controller;

import com.zretc.dao.EmpDao;
import com.zretc.pojo.Emp;
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.junit.Test;

import java.io.IOException;
import java.io.Reader;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

public class EmpTest {
    @Test
    public void test01() throws IOException {
        // 获取mybatis的总配置文件
        // 1. 读取mybatis总配置文件
        Reader reader = Resources.getResourceAsReader("mybatisConfig.xml");
        // 2. 获取SqlSessionFactory工厂
        SqlSessionFactory sessionFactory = new
        SqlSessionFactoryBuilder().build(reader);
        // 3. 获取SqlSession
        SqlSession sqlSession = sessionFactory.openSession(true);
        EmpDao empDao = sqlSession.getMapper(EmpDao.class);
        // 添加
        //      Emp emp = new Emp();
        //      emp.setName("soso2");
        //      emp.setHire_date(new Date());
        //      emp.setJob_id(1);
        //      emp.setDept_id(14);
        //      emp.setInterest("吃饭，睡觉，打豆豆");
        //      empDao.insertEmp(emp);

        // 修改
        //      Emp emp = new Emp();
        //      emp.setEmpno(6);
        //      emp.setName("soso666");
        //      emp.setHire_date(new Date());
        //      empDao.updateEmp(emp);

        // 查询员工信息，员工编号等于值列表中的一个（例如：编号在1/2/6的员工）
        //      List<Integer> list = new ArrayList<>();
        //      list.add(1);
        //      list.add(2);
        //      list.add(6);
        //      for (Emp emp : empDao.selectByIn(list)) {
        //          System.out.println(emp);
        //      }
    }
}
```

```

        // 查询员工编号是6以内的员工信息
        //      List<Emp> emps = empDao.selectByBetween(6);
        //      for (Emp emp : emps) {
        //          System.out.println(emp);
        //      }

        // 根据任意条件查询
        //      Emp emp = new Emp();
        //      emp.setEmpno(6);
        //      emp.setEname("soso666");
        //      emp.setHire_date(new Date());
        //      for (Emp emp1 : empDao.selectByOther(emp)) {
        //          System.out.println(emp1);
        //      }

        // 多个连接条件
        Emp emp = new Emp();
        //      emp.setEmpno(6);
        //      emp.setEname("soso666");
        emp.setJob_id(1);
        emp.setDept_id(1);
        for (Emp emp1 : empDao.selectByAllOther(emp)) {
            System.out.println(emp1);
        }
        sqlSession.close();
    }
}

```

11、类型处理器 TypeHandlers

- 无论是MyBatis在预处理语句（PreparedStatement）中设置一个参数时，还是从结果集中取出一个值时，都会用类型处理器（typeHandlers）将获取的值以合适的方式转换成Java类型，一些默认的类型处理器如下：

类型处理器	Java 类型	JDBC 类型
DateTypeHandler	java.util.Date	TIMESTAMP
DateOnlyTypeHandler	java.util.Date	DATE
TimeOnlyTypeHandler	java.util.Date	TIME
SqlDateTypeHandler	java.sql.Date	DATE

- 开发人员可以重写类型处理器或自定义类型处理器来处理不支持的或非标准的类型。
具体做法为：实现org.apache.ibatis.type.TypeHandler接口，或继承适配器类org.apache.ibatis.type.BaseTypeHandler（该类内部处理了NULL），然后可以选择性地将它映射到一个JDBC类型
- 映射类型可以采用两种方法：
 - 配置文件中指定
 - 在自定义类型处理器中标注特定的注解
- 接下来通过示例介绍类型处理器的编写和配置方法（在很多场景中，可能会将Java中的集合数据通过分隔符拼装成一个完整的字符串存放在数据库的一个字段中，如果地址列表以及在数据库中利用JSON存储Java对象信息）



- 定义类型处理，也就是定义java类型与数据库中的数据类型之间的转换关系。我们可以实现TypeHandler接口
 - 在设置PreparedStatement时调用 存数据 --- 将java实体类的类型转换成数据库中字段对应的类型

setParameter(PreparedStatement ps, int i, T parameter, JdbcType jdbcType) 方法

 - 参数ps 当前的PreparedStatement对象
 - 参数i 当前参数的位置
 - 参数parameter 当前参数的Java对象
 - 参数jdbcType 当前参数的数据库类型
 - 在获取结果级时调用getResult的三个重载方法

- getResult(ResultSet rs, String columnName)
 - getResult(ResultSet rs, int columnIndex)
 - getResult(CallableStatement cs, int columnIndex)

查询 --- 将数据库中的数据类型转换成java实体类需要的类型

BaseTypeHandler实现自TypeHandler接口

- 我们可以方便的利用BaseTypeHandler不用做null值判断
 - setNonNullParameter preparedStatement设置值调用
 - getNullableResult resultSet获取值时调用
- 使用BaseTypeHandler还有一个好处是它继承了另外一个叫做TypeReference的抽象类，通过TypeReference的getRawType()方法可以获取到当前TypeHandler所使用泛型的原始类型

11.1 实现步骤

继承BaseTypeHandler

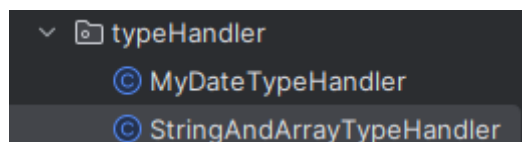
- 覆盖setNonNullParameter
 - 把数组变成字符串value
 - 用于设置参数值 ps.set**(index, value);
- getNullableResult
 - 用于检索结果：rs.get**(index);
 - 把结果变成数组

将emp中的兴趣爱好字段，在java中定义成字符串数组类型，
将java中的字符串数组 转成 数据库中的 字符串，用 逗号进行分割

Emp中的字段 改成是数组：

```
@Data
public class Emp {
    private Integer empno;
    private String ename;
    private Date hire_date;
    private Integer job_id;
    private Integer dept_id;
    private String[] interest;
}
```

定义类型转换器：



```
package com.zretc.typeHandler;

import org.apache.ibatis.type.BaseTypeHandler;
import org.apache.ibatis.type.JdbcType;

import java.sql.CallableStatement;
import java.sql.PreparedStatement;
```

```

import java.sql.ResultSet;
import java.sql.SQLException;

// 字符串和数组之间的类型转换
// java --- 数组
// mysql --- varchar
public class StringAndArrayTypeHandler extends BaseTypeHandler<String[]> {

    @Override
    public void setNonNullParameter(PreparedStatement ps, int i, String[]
parameter, JdbcType jdbcType) throws SQLException {
        // 往数据库中存储数据时，将java中的数组类型转成数据库中的varchar字符串
        // 数据库中的字符串：唱歌，跳舞，睡觉
        // 将数组中的元素按照逗号进行拼接
        String param = "";
        for(int j = 0;j < parameter.length;j++){
            // 最后一个元素不加逗号
            if(j == parameter.length - 1){
                param += parameter[j];
            }else{
                param += parameter[j] + ",";
            }
        }
        ps.setString(i,param);
    }

    @Override
    public String[] getNullableResult(ResultSet rs, String columnName) throws
SQLException {
        // 查询时，将数据库中的字符串，转成java中的数组
        String param = rs.getString(columnName);
        // 兴趣爱好的字段允许有空值，因此需要非空判断
        if (param != null){
            // 将数据库中的 param 转成java中的数组
            // 按照 逗号 进行切割
            return param.split(",");
        }
        return new String[0];
    }

    @Override
    public String[] getNullableResult(ResultSet rs, int columnIndex) throws
SQLException {
        // 查询时，将数据库中的字符串，转成java中的数组
        String param = rs.getString(columnIndex);
        // 兴趣爱好的字段允许有空值，因此需要非空判断
        if (param != null){
            // 将数据库中的 param 转成java中的数组
            // 按照 逗号 进行切割
            return param.split(",");
        }
        return new String[0];
    }

    @Override
    public String[] getNullableResult(CallableStatement cs, int columnIndex)
throws SQLException {
        // 查询时，将数据库中的字符串，转成java中的数组

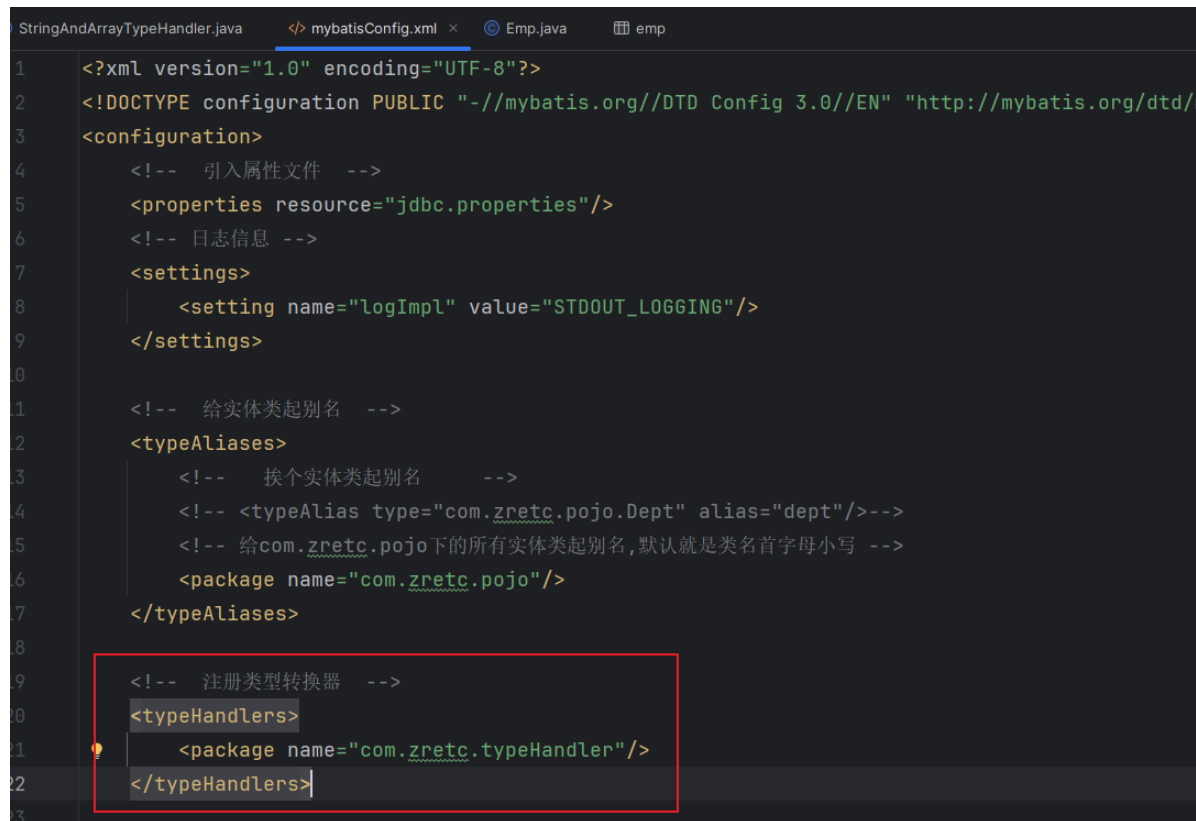
```

```

        // 查询时，将数据库中的字符串，转成java中的数组
        String param = cs.getString(columnIndex);
        // 兴趣爱好的字段允许有空值，因此需要非空判断
        if (param != null){
            // 将数据库中的 param 转成java中的数组
            // 按照 逗号 进行切割
            return param.split(",");
        }
        return new String[0];
    }
}

```

把类型转换器在 mybatis 总配置文件中注册：



```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN" "http://mybatis.org/dtd/mybatis-3-config.dtd">
3  <configuration>
4      <!-- 引入属性文件 -->
5      <properties resource="jdbc.properties"/>
6      <!-- 日志信息 -->
7      <settings>
8          <setting name="logImpl" value="STDOUT_LOGGING"/>
9      </settings>
10
11     <!-- 给实体类起别名 -->
12     <typeAliases>
13         <!-- 挨个实体类起别名 -->
14         <!-- <typeAlias type="com.zretc.pojo.Dept" alias="dept"/>-->
15         <!-- 给com.zretc.pojo下的所有实体类起别名,默认就是类名首字母小写 -->
16         <package name="com.zretc.pojo"/>
17     </typeAliases>
18
19     <!-- 注册类型转换器 -->
20     <typeHandlers>
21         <package name="com.zretc.typeHandler"/>
22     </typeHandlers>
23

```

```

<!-- 注册类型转换器 -->
<typeHandlers>
    <package name="com.zretc.typeHandler"/>
</typeHandlers>

```

在测试类中测试 添加 和 查询功能

```
// 根据任意条件查询
Emp emp = new Emp();
//      emp.setEmpno(6);
//      emp.setEname("soso666");
emp.setHire_date(new Date());
for (Emp emp1 : empDao.selectByOther(emp)) {
    System.out.println(emp1);
}
```

```
✓ Tests passed: 1 of 1 test - 1sec 175ms

Checking to see if class com.zretc.dao.EmpDao matches criteria [is assignable to Object]
Opening JDBC Connection
Created connection 1174248013.
==> Preparing: select * from emp WHERE hire_date=?
==> Parameters: 2025-03-07(String)
<== Columns: empno, ename, hire_date, job_id, dept_id, interest
<== Row: 1, tom, 2025-03-07, 1, 1, 唱歌, 跳舞
<== Row: 2, lisa, 2025-03-07, 1, 1, 唱歌, 跳舞
<== Row: 3, jack, 2025-03-07, 1, 2, 唱歌, 跳舞
<== Row: 4, kiki, 2025-03-07, 1, 14, 吃饭, 睡觉, 打豆豆
<== Row: 5, soso, 2025-03-07, null, null, null
<== Row: 6, soso666, 2025-03-07, null, null, null
<== Row: 7, soso2, 2025-03-07, 1, 14, 唱歌, 睡觉, 玩, 敲代码
<== Row: 8, soso2, 2025-03-07, 1, 14, 唱歌, 睡觉, 玩, 敲代码
<== Total: 8
Emp(empno=1, ename=tom, hire_date=Fri Mar 07 00:00:00 CST 2025, job_id=1, dept_id=1, interest=[唱歌, 跳舞])
Emp(empno=2, ename=lisa, hire_date=Fri Mar 07 00:00:00 CST 2025, job_id=1, dept_id=1, interest=[唱歌, 跳舞])
Emp(empno=3, ename=jack, hire_date=Fri Mar 07 00:00:00 CST 2025, job_id=1, dept_id=2, interest=[唱歌, 跳舞])
Emp(empno=4, ename=kiki, hire_date=Fri Mar 07 00:00:00 CST 2025, job_id=1, dept_id=14, interest=[吃饭, 睡觉, 打豆豆])
Emp(empno=5, ename=soso, hire_date=Fri Mar 07 00:00:00 CST 2025, job_id=null, dept_id=null, interest=[])
Emp(empno=6, ename=soso666, hire_date=Fri Mar 07 00:00:00 CST 2025, job_id=null, dept_id=null, interest=[])
Emp(empno=7, ename=soso2, hire_date=Fri Mar 07 00:00:00 CST 2025, job_id=1, dept_id=14, interest=[唱歌, 睡觉, 玩, 敲代码])
Emp(empno=8, ename=soso2, hire_date=Fri Mar 07 00:00:00 CST 2025, job_id=1, dept_id=14, interest=[唱歌, 睡觉, 玩, 敲代码])
```

```
// 添加
Emp emp = new Emp();
emp.setEname("soso2");
emp.setHire_date(new Date());
emp.setJob_id(1);
emp.setDept_id(14);
emp.setInterest(new String[]{"唱歌", "睡觉", "玩", "敲代码"});
empDao.insertEmp(emp);
```

```
Created connection 13824077.
==> Preparing: insert into emp ( ename, hire_date, job_id, dept_id, interest ) values( ?, ?, ?, ?, ? )
==> Parameters: soso2(String), 2025-03-07(String), 1(Integer), 14(Integer), 唱歌, 睡觉, 玩, 敲代码(String)
<== Updates: 1
```

将日期转换成字符串

```
package com.zretc.typeHandler;

import org.apache.ibatis.type.BaseTypeHandler;
import org.apache.ibatis.type.JdbcType;

import java.sql.CallableStatement;
import java.sql.PreparedStatement;
```

```

import java.sql.ResultSet;
import java.sql.SQLException;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

public class MyDateTypeHandler extends BaseTypeHandler<Date> {
    // 将java中的Date 转成 年-月-日的日期
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
    @Override
    public void setNonNullParameter(PreparedStatement ps, int i, Date parameter,
        JdbcType jdbcType) throws SQLException {
        // 将日期转成字符串
        String hire_date = sdf.format(parameter);
        // 替换 i 位置的值
        ps.setString(i, hire_date);
    }

    @Override
    public Date getNullableResult(ResultSet rs, String columnName) throws
        SQLException {
        try {
            // 将数据库中的字符串, 转成java中的Date
            String hire_date = rs.getString(columnName);
            return sdf.parse(hire_date);
        } catch (ParseException e) {
            throw new RuntimeException(e);
        }
    }

    @Override
    public Date getNullableResult(ResultSet rs, int columnIndex) throws
        SQLException {
        try {
            // 将数据库中的字符串, 转成java中的Date
            String hire_date = rs.getString(columnIndex);
            return sdf.parse(hire_date);
        } catch (ParseException e) {
            throw new RuntimeException(e);
        }
    }

    @Override
    public Date getNullableResult(CallableStatement cs, int columnIndex) throws
        SQLException {
        try {
            // 将数据库中的字符串, 转成java中的Date
            String hire_date = cs.getString(columnIndex);
            return sdf.parse(hire_date);
        } catch (ParseException e) {
            throw new RuntimeException(e);
        }
    }
}

```

