1、持久层框架对比

1）JDBC：SQL 夹杂在 Java 代码中**耦合度高**，导致硬编码内伤；维护不易且实际开发需求中 SQL 有变化，频繁修改的情况多见；**代码冗长**，开发效率低。

2）Hibernate 和 JPA：操作简便，开发效率高；程序中的长难复杂 SQL 需要绕过框架；**内部自动生成的 SQL，不容易做特殊优化**；基于**全映射的全自动框架**，大量字段的 POJO 进行**部分映射**时**比较困难**；**反射操作太多**，导致数据库性能下降。

3）MyBatis：轻量级，性能出色；SQL 和 Java 编码分开，功能边界清晰。**Java 代码**专注**业务、SQL 语句**专注**数据**；开发效率稍逊于 Hibernate，但是完全能够接收；**开发效率**：Hibernate>Mybatis>JDBC；**运行效率**：JDBC>Mybatis>Hibernate。

2、mybatis 日志输出配置

在 mybatis 的配置文件使用 **settings 标签**设置输出运过程 SQL 日志

```
<settings>
  <!-- SLF4J 选择slf4j输出！  -->
  <setting name="logImpl" value="SLF4J"/>
</settings>
```

3、向 SQL 语句传参

1）**#{}**形式：Mybatis 会将 SQL 语句中的#{}转换为**问号占位符**。

2）**${}**形式：Mybatis 做的是**字符串拼接**操作。

总结：**#{}**形式传值可以**防止【注入攻击】**问题；实际开发中，能用#{}实现的，肯定不用${}。

特殊情况：动态的不是值，是**列名或者关键字**，需要使用**${}拼接**。

4、数据输入

1）单个简单类型参数

在#{}中可以随意命名，但是通常还是使用和**接口方法参数同名**。

Mapper接口中抽象方法的声明

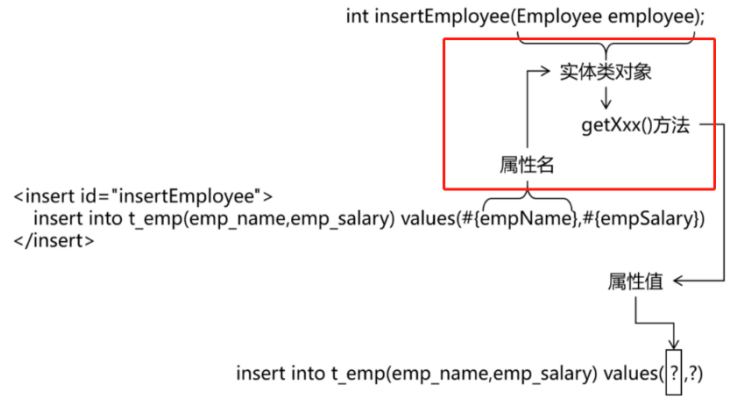
```
Employee selectEmployee(Integer empId);
```

SQL语句

```
<select id="selectEmployee" resultType="com.atguigu.mybatis.entity.Employee">
  select emp_id empId,emp_name empName,emp_salary empSalary from t_emp where emp_id=#{empId}
</select>
```

2）实体类类型参数

Mybatis 会根据#{}中**传入的数据，加工成 getXxx()方法，通过反射**在实体类对象中**调用这个方法**，从而**获取到对应的数据**。**填充**到#{}解析后的**问号占位符**这个位置。



Mapper接口中抽象方法的声明

```
int insertEmployee(Employee employee);
```

SQL语句

对应属性名

```
<insert id="insertEmployee">
  insert into t_emp(emp_name,emp_salary) values(#{empName},#{empSalary})
</insert>
```

3）多个简单类型参数（不可以随便写，也不可以按照形参名称获取）

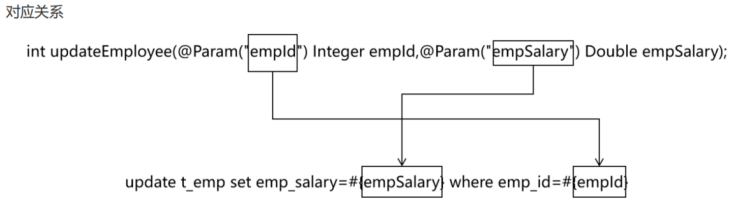
方案 1：**注解指定**

@Param 注解，指定多个简单参数的 key，key=@Param(“value 值”)；

方案 2：**mybatis 默认机制**

形参从左到右依次对应 arg0,arg1,…或 param1,param2,…(索引 0/1)

对应关系



Mapper接口中抽象方法的声明

```
int updateEmployee(@Param("empId") Integer empId,@Param("empSalary") Double empSalary);
```

SQL语句

```
<update id="updateEmployee">
  update t_emp set emp_salary=#{empSalary} where emp_id=#{empId}
</update>
```

也可以用 #{arg0} #{arg1}

4）Map 类型参数

#{}中写 Map 中的 key，使用场景：有**很多零散的参数**需要传递，但是**没有对应的实体类类型**可以使用。使用**@Param 注解**一个一个传入又**太麻烦**了，所以都**封装到 Map 中**。

Mapper接口中抽象方法的声明

```
int updateEmployeeByMap(Map<String, Object> paramMap);
```

SQL语句

```
<update id="updateEmployeeByMap">

  update t_emp set emp_salary=#{empSalaryKey} where emp_id=#{empIdKey}

</update>
```

```
@Test
public void testUpdateEmpNameByMap() {
  EmployeeMapper mapper = session.getMapper(EmployeeMapper.class);
  Map<String, Object> paramMap = new HashMap<>();
  paramMap.put("empSalaryKey", 999.99);
  paramMap.put("empIdKey", 5);
  int result = mapper.updateEmployeeByMap(paramMap);
  log.info("result = " + result);
}
```

5、数据输出

数据输出总体上有两种形式：

>**增删改**操作返回的**受影响行数**：直接使用 int 或 long 类型接收即可；

>**查询**操作返回**查询结果**，我们需要**指定**查询的**输出数据类型**。

1）单个简单类型（通过 **resultType** 指定**查询返回值类型+别名**）

```
int selectEmpCount();
```

SQL语句

```
<select id="selectEmpCount" resultType="int">
  select count(*) from t_emp
</select>
```

select 标签，通过 **resultType** 指定**查询返回值类型**；

resultType=”类全限定符 | 别名 | 返回集合类型时写泛型类型即可”

【补充】别名问题：

类型别名可为 Java 类型设置一个缩写名字。它仅用于 XML 配置，意在**降低冗余的全限定类名书写**。

Mybatis 提供了 72 中默认的别名[Java 中常用的数据类型]

- 基本数据类型 int,double,…→_int,_double,…
- 包装数据类型 Integer,Double,…→int,double,…
- 集合数据类型 Map,List,HashMap→map,list,hashmap,…

如果没有提供的需要自己定义：

- 给类单独定义别名：

<**typeAlias alias**=“别名”,**type**=“全限定类名”/>

```
<typeAliases>
  <typeAlias alias="Author" type="domain.blog.Author"/>
  <typeAlias alias="Blog" type="domain.blog.Blog"/>
</typeAliases>
```

- **批量设置**包下所有类的别名，即类的首字母小写：

<package name=“包名”>

```
<typeAliases> <package name="domain.blog"/> </typeAliases>
```

同时还可以使用**@Alias(”value 值”)**对该包下**某个类设置别名**：

```
@Alias("author")
public class Author {
    ...
}
```

2）返回实体类对象(自动映射驼峰式命名规则时可以不用别名)
通过**给数据库表字段加别名**，让查询结果的每一列都和 Java 实体类中属性对应起来。可以在 **settings 中**将 **mapUnderscoreToCamelCase 属性**配置为 true，表示**开启自动映射驼峰式命名规则**，就可以不用别名。

Mapper接口的抽象方法

```
Employee selectEmployee(Integer empId);
```

SQL语句

```
<!-- 编写具体的SQL语句，使用id属性唯一的标记一条SQL语句 -->
<!-- resultType属性：指定封装查询结果的Java实体类的全类名 -->
<select id="selectEmployee" resultType="com.atguigu.mybatis.entity.Employee">

  <!-- Mybatis负责把SQL语句中的#{}部分替换成“?”占位符 -->
  <!-- 给每一个字段设置一个别名，让别名和Java实体类中属性名一致 -->
  select emp_id empId,emp_name empName,emp_salary empSalary from t_emp where emp_id=#{maomi}

</select>
```

增加全局配置自动识别对应关系：在 Mybatis 全局配置文件中，做了下面的配置，select 语句中可以不给字段设置别名。

```
<!-- 在全局范围内对Mybatis进行配置 -->
<settings>

  <!-- 具体配置 -->
  <!-- 从org.apache.ibatis.session.Configuration类中可以查看能使用的配置项 -->
  <!-- 将mapUnderscoreToCamelCase属性配置为true，表示开启自动映射驼峰式命名规则 -->
  <!-- 规则要求数据库表字段命名方式：单词_单词 -->
  <!-- 规则要求Java实体类属性名命名方式：首字母小写的驼峰式命名 -->
  <setting name="mapUnderscoreToCamelCase" value="true"/>

</settings>
```

3）返回 Map 类型（没法封装到实体类对象中）
适用于 SQL 查询返回的各个字段综合起来并不和任何一个现有的实体类对应，**没法封装到实体类对象中**。能够封装成实体类类型的，就不使用 Map 类型。

Mapper接口的抽象方法

```
Map<String,Object> selectEmpNameAndMaxSalary();
```

SQL语句

```
<!-- Map<String,Object> selectEmpNameAndMaxSalary(); -->
<!-- 返回工资最高的员工的姓名和他的工资 -->
<select id="selectEmpNameAndMaxSalary" resultType="map">
  SELECT
    emp_name 员工姓名,
    emp_salary 员工工资,
    (SELECT AVG(emp_salary) FROM t_emp) 部门平均工资
  FROM t_emp WHERE emp_salary=(
    SELECT MAX(emp_salary) FROM t_emp
  )
</select>
```

4）返回 List 类型（查询结果返回多个实体类对象）
查询结果**返回多个实体类对象**，希望把多个实体类对象放在 List 集合中返回。此时不需要任何特殊处理，在 **resultType 属性**中还是**设置实体类类型**即可。

```
<!-- List<Employee> selectAll(); -->
<select id="selectAll" resultType="com.atguigu.mybatis.entity.Employee">
  select emp_id empId,emp_name empName,emp_salary empSalary
  from t_emp
</select>
```

5）返回主键值
1> 自增长类型主键
useGeneratedKeys="true"和 **keyProperty**="主键对应的实体属性名" Mybatis 是将自增主键的值**设置到实体类对象中**，而**不是**以 **Mapper 接口方法返回值**的形式返回。

SQL语句

```
<!-- int insertEmployee(Employee employee); -->
<!-- useGeneratedKeys属性字面意思就是“使用生成的主键” -->
<!-- keyProperty属性可以指定主键在实体类对象中对应的属性名，Mybatis会将拿到的主键值存入这个属性 -->
<insert id="insertEmployee" useGeneratedKeys="true" keyProperty="empId">
  insert into t_emp(emp_name,emp_salary)
  values(#{empName},#{empSalary})
</insert>
```

2> 非自增长类型主键
对于不支持自增型主键的数据库(例如 Oracle)或者字符串类型主键，则可以使用 **selectKey 子元素**：selectKey 元素将**会首先运行**，**id 会被设置**，然后**插入语句会被调用**。

```
<insert id="insertUser" parameterType="User">
    <selectKey keyProperty="id" resultType="java.lang.String"
        order="BEFORE">
        SELECT UUID() as id SELECT REPLACE(UUID(),'-','')
    </selectKey>
    INSERT INTO user (id, username, password)
    VALUES (
        #{id},
        #{username},
        #{password}
    )
</insert>
```

我们定义了一个 insertUser 的插入语句来将 User 对象插入到 user 表中。我们**使用 selectKey 来查询 UUID** 并**设置到 id 字段中**。
keyProperty 属性：指定查询到的 UUID 赋值给**对象中的 id 属性**，
resultType 属性：指定 UUID 的类型为 **java.lang.String**，
order 属性:指定 SQL 语句在插入语句之前或之后执行(BEFORE|AFTER)。
【注意】我们将 selectKey 放在了插入语句的前面，这是因为 MySQL 在 insert 语句中只支持一个 select 子句，而 selectKey 中查询 UUID 的语句就是一个 select 子句，因此我们需要将其放在前面。
最后，在将 User 对象插入到 user 表中时，我们直接使用对象中的 id 属性来插入主键值。

6）实体类属性和数据库字段对应关系
1> 别名对应：将字段的别名设置成和实体类属性一致。

```
<!-- 编写具体的SQL语句，使用id属性唯一的标记一条SQL语句 -->
<!-- resultType属性：指定封装查询结果的Java实体类的全类名 -->
<select id="selectEmployee" resultType="com.atguigu.mybatis.entity.Employee">

  <!-- Mybatis负责把SQL语句中的#{}部分替换成“?”占位符 -->
  <!-- 给每一个字段设置一个别名，让别名和Java实体类中属性名一致 -->
  select emp_id empId,emp_name empName,emp_salary empSalary from t_emp where emp_id=#
{maomi}

</select>
```

2> 全局配置自动识别驼峰式命名规则
在 Mybatis 全局配置文件加入如下配置：

```
<!-- 使用settings对Mybatis全局进行设置 -->
<settings>

  <!-- 将xxx_xxx这样的列名自动映射到xxXxx这样驼峰式命名的属性名 -->
  <setting name="mapUnderscoreToCamelCase" value="true"/>

</settings>
```

SQL语句中可以不使用别名

```
<!-- Employee selectEmployee(Integer empId); -->
<select id="selectEmployee" resultType="com.atguigu.mybatis.entity.Employee">

  select emp_id,emp_name,emp_salary from t_emp where emp_id=#{empId}

</select>
```

3> 使用 resultMap（具有嵌套结构的复杂实体类）
使用 **resultMap 标签定义对应关系**，再在 SQL 语句中引用此对应关系。
id 标签：设置**主键列**和**主键属性**之间的对应关系；
result 标签：设置**普通字段**和 **Java 实体类属性**之间的关系
column 属性：用于指定**字段名**；
property 属性：用于指定 **Java 实体类属性名**。

```
<!-- 专门声明一个resultMap设定column到property之间的对应关系 -->
<resultMap id="selectEmployeeByRMResultMap" type="com.atguigu.mybatis.entity.Employee">

  <!-- 使用id标签设置主键列和主键属性之间的对应关系 -->
  <!-- column属性用于指定字段名；property属性用于指定Java实体类属性名 -->
  <id column="emp_id" property="empId"/>

  <!-- 使用result标签设置普通字段和Java实体类属性之间的关系 -->
  <result column="emp_name" property="empName"/>

  <result column="emp_salary" property="empSalary"/>

</resultMap>

<!-- Employee selectEmployeeByRM(Integer empId); -->
<select id="selectEmployeeByRM" resultMap="selectEmployeeByRMResultMap">

  select emp_id,emp_name,emp_salary from t_emp where emp_id=#{empId}

</select>
```

【注意】在 setting 中将 autoMappingBehavior 设置为 full，可以开启 resultMap 自动映射（列名=属性名），省略<result/>标签。(第 11)

6、单表 CRUD 实践

1）准备数据库数据

```
CREATE TABLE `user` (
  `id` INT(11) NOT NULL AUTO_INCREMENT,
  `username` VARCHAR(50) NOT NULL,
  `password` VARCHAR(50) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=INNODB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

2）实体类准备

接下来，我们需要定义一个实体类 `User`，来对应 user 表的一行数据。

```
@Data //lombok
public class User {
  private Integer id;
  private String username;
  private String password;
}
```

3）Mapper 接口定义

定义一个 Mapper 接口 `UserMapper`，并在其中添加 user 表的增、删、改、查方法。

```
public interface UserMapper {

  int insert(User user);

  int update(User user);

  int delete(Integer id);

  User selectById(Integer id);

  List<User> selectAll();
}
```

4）MapperXML 编写

在 resources/mappers 目录下创建一个名为 UserMapper.xml 的 XML 文件，包含与 Mapper 接口中相同的五个 SQL 语句，并在其中，将查询结果映射到 User 实体中。

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "https://mybatis.org/dtd/mybatis-3-mapper.dtd">
<!-- namespace等于mapper接口类的全限定名,这样实现对应 -->
<mapper namespace="com.atguigu.mapper.UserMapper">
  <!-- 定义一个插入语句，并获取主键值 -->
  <insert id="insert" useGeneratedKeys="true" keyProperty="id">
    INSERT INTO user(username, password)
                VALUES(#{username}, #{password})
  </insert>
```

```
  <update id="update">
    UPDATE user SET username=#{username}, password=#{password}
    WHERE id=#{id}
  </update>

  <delete id="delete">
    DELETE FROM user WHERE id=#{id}
  </delete>
```

```
  <!-- resultType使用user别名，稍后需要配置! -->
  <select id="selectById" resultType="user">
    SELECT id, username, password FROM user WHERE id=#{id}
  </select>

  <!-- resultType返回值类型为集合，所以只写范型即可!  -->
  <select id="selectAll" resultType="user">
    SELECT id, username, password FROM user
  </select>

</mapper>
```

5）MyBatis 配置文件

位置：resources: 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>
        <!-- 开启驼峰式映射-->
        <setting name="mapUnderscoreToCamelCase" value="true"/>
        <!-- 开启logback日志输出-->
        <setting name="logImpl" value="SLF4J"/>
    </settings>

    <typeAliases>
        <!-- 给实体类起别名 -->
        <package name="com.atguigu.pojo"/>
    </typeAliases>
```

```
    <!-- environments表示配置Mybatis的开发环境，可以配置多个环境，在众多具体环境中，使用default属性指定实际运行时使用的环境。default属性的取值是environment标签的id属性的值。 -->
    <environments default="development">
        <!-- environment表示配置Mybatis的一个具体的环境 -->
        <environment id="development">
            <!-- Mybatis的内置的事务管理器 -->
            <transactionManager type="JDBC"/>
            <!-- 配置数据源 -->
            <dataSource type="POOLED">
                <!-- 建立数据库连接的具体信息 -->
                <property name="driver" value="com.mysql.cj.jdbc.Driver"/>
                <property name="url" value="jdbc:mysql://localhost:3306/mybatis-example"/>
                <property name="username" value="root"/>
                <property name="password" value="root"/>
            </dataSource>
        </environment>
    </environments>
```

```
    <mappers>
        <!-- Mapper注册：指定Mybatis映射文件的具体位置 -->
        <!-- mapper标签：配置一个具体的Mapper映射文件 -->
        <!-- resource属性：指定Mapper映射文件的实际存储位置，这里需要使用一个以类路径根目录为基准的相对路径 -->
        <!--    对Maven工程的目录结构来说，resources目录下的内容会直接放入类路径，所以这里我们可以以resources目录为基准 -->
        <mapper resource="mappers/UserMapper.xml"/>
    </mappers>

</configuration>
```

6）效果测试

```
public class MyBatisTest {
    private SqlSession session;
    // junit会在每一个@Test方法前执行@BeforeEach方法
    @BeforeEach
    public void init() throws IOException {
        session = new SqlSessionFactoryBuilder()
                .build(
                        Resources.getResourceAsStream("mybatis-config.xml"))
                .openSession();
    }
    @Test
    public void createTest() {
        User user = new User();
        user.setUsername("admin");
        user.setPassword("123456");
        UserMapper userMapper = session.getMapper(UserMapper.class);
        userMapper.insert(user);
        System.out.println(user);
    }
```

```
    @Test
    public void selectByIdTest() {
        UserMapper userMapper = session.getMapper(UserMapper.class);
        User user = userMapper.selectById(1);
        System.out.println("user = " + user);
    }

    @Test
    public void selectAllTest() {
        UserMapper userMapper = session.getMapper(UserMapper.class);
        List<User> userList = userMapper.selectAll();
        System.out.println("userList = " + userList);
    }

    // junit会在每一个@Test方法后执行@@AfterEach方法
    @AfterEach
    public void clear() {
        session.commit();
        session.close();
    }
}
```

7、mapperXML 标签总结

1）select 元素允许配置很多属性来配置每条语句的行为细节：

属性	描述
id	在命名空间中唯一的标识符，可以被用来引用这条语句。
resultType	期望从这条语句中返回结果的类全限定名或别名。 注意，如果返回的是集合，那应该设置为集合包含的类型，而不是集合本身的类型。 resultType 和 resultMap 之间只能同时使用一个。
resultMap	对外部 resultMap 的命名引用。结果映射是 MyBatis 最强大的特性，如果你对其理解透彻，许多复杂的映射问题都能迎刃而解。 resultType 和 resultMap 之间只能同时使用一个。
timeout	这个设置是在抛出异常之前，驱动程序等待数据库返回请求结果的秒数。默认值为未设置（unset）（依赖数据库驱动）。
statementType	可选 STATEMENT，PREPARED 或 CALLABLE。这会让 MyBatis 分别使用 Statement，PreparedStatement 或 CallableStatement，默认值：PREPARED。

2）insert，update 和 delete 标签：

属性	描述
id	在命名空间中唯一的标识符，可以被用来引用这条语句。
timeout	这个设置是在抛出异常之前，驱动程序等待数据库返回请求结果的秒数。默认值为未设置（unset）（依赖数据库驱动）。
statementType	可选 STATEMENT，PREPARED 或 CALLABLE。这会让 MyBatis 分别使用 Statement，PreparedStatement 或 CallableStatement，默认值：PREPARED。
useGeneratedKeys	（仅适用于 insert 和 update）这会令 MyBatis 使用 JDBC 的 getGeneratedKeys 方法来取出由数据库内部生成的主键（比如：像 MySQL 和 SQL Server 这样的关系型数据库管理系统的自动递增字段），默认值：false。
keyProperty	（仅适用于 insert 和 update）指定能够唯一识别对象的属性，MyBatis 会使用 getGeneratedKeys 的返回值或 insert 语句的 selectKey 子元素设置它的值，默认值：未设置（unset）。如果生成列不止一个，可以用逗号分隔多个属性名称。
keyColumn	（仅适用于 insert 和 update）设置生成键值在表中的列名，在某些数据库（像 PostgreSQL）中，当主键列不是表中的第一列的时候，是必须设置的。如果生成列不止一个，可以用逗号分隔多个属性名称。

8、对一映射

1）需求说明：根据 ID 查询订单，以及订单关联的用户的信息。

2）OrderMapper 接口

```
public interface OrderMapper {
  Order selectOrderWithCustomer(Integer orderId);
}
```

3）OrderMapper.xml 配置文件

```
<!-- 创建resultMap实现“对一”关联关系映射 -->
<!-- id属性：通常设置为这个resultMap所服务的那条SQL语句的id加上“ResultMap” -->
<!-- type属性：要设置为这个resultMap所服务的那条SQL语句最终要返回的类型 -->
<resultMap id="selectOrderWithCustomerResultMap" type="order">

  <!-- 先设置Order自身属性和字段的对应关系 -->
  <id column="order_id" property="orderId"/>
  <result column="order_name" property="orderName"/>

  <!-- 使用association标签配置“对一”关联关系 -->
  <!-- property属性：在Order类中对一的一端进行引用时使用的属性名 -->
  <!-- javaType属性：一的一端类的全类名 -->
  <association property="customer" javaType="customer">
    <!-- 配置Customer类的属性和字段名之间的对应关系 -->
    <id column="customer_id" property="customerId"/>
    <result column="customer_name" property="customerName"/>
  </association>

</resultMap>
```

```
<!-- Order selectOrderWithCustomer(Integer orderId); -->
<select id="selectOrderWithCustomer" resultMap="selectOrderWithCustomerResultMap">

  SELECT order_id,order_name,c.customer_id,customer_name
  FROM t_order o
  LEFT JOIN t_customer c
  ON o.customer_id=c.customer_id
  WHERE o.order_id=#{orderId}

</select>
```

4）Mybatis 全局注册 Mapper 文件

```
<!-- 注册Mapper配置文件：告诉Mybatis我们的Mapper配置文件的位置 -->
<mappers>

  <!-- 在mapper标签的resource属性中指定Mapper配置文件以“类路径根目录”为基准的相对路径 -->
  <mapper resource="mappers/OrderMapper.xml"/>

</mappers>
```

5）测试效果

```
@Slf4j
public class MyBatisTest {
    private SqlSession session;
    // junit会在每一个@Test方法前执行@BeforeEach方法
    @BeforeEach
    public void init() throws IOException {
        session = new SqlSessionFactoryBuilder()
                .build(
                        Resources.getResourceAsStream("mybatis-config.xml"))
                .openSession();
    }

    @Test
    public void testRelationshipToOne() {
      OrderMapper orderMapper = session.getMapper(OrderMapper.class);
      // 查询Order对象，检查是否同时查询了关联的Customer对象
      Order order = orderMapper.selectOrderWithCustomer(2);
      log.info("order = " + order);
    }

    // junit会在每一个@Test方法后执行@@AfterEach方法
    @AfterEach
    public void clear() {
        session.commit();
        session.close();
    }
}
```

9、对多映射

1）需求说明：查询客户和客户关联的订单信息。

2）CustomerMapper 接口

```
public interface CustomerMapper {
  Customer selectCustomerWithOrderList(Integer customerId);
}
```

3）CustomerMapper.xml 文件

```
<!-- 配置resultMap实现从Customer到OrderList的“对多”关联关系 -->
<resultMap id="selectCustomerWithOrderListResultMap" type="customer">

  <!-- 映射Customer本身的属性 -->
  <id column="customer_id" property="customerId"/>
  <result column="customer_name" property="customerName"/>

  <!-- collection标签：映射“对多”的关联关系 -->
  <!-- property属性：在Customer类中，关联“多”的一端的属性名 -->
  <!-- ofType属性：集合属性中元素的类型 -->
  <collection property="orderList" ofType="order">
    <!-- 映射Order的属性 -->
    <id column="order_id" property="orderId"/>
    <result column="order_name" property="orderName"/>
  </collection>

</resultMap>
```

```
<!-- Customer selectCustomerWithOrderList(Integer customerId); -->
<select id="selectCustomerWithOrderList"
resultMap="selectCustomerWithOrderListResultMap">
  SELECT c.customer_id,c.customer_name,o.order_id,o.order_name
  FROM t_customer c
  LEFT JOIN t_order o
  ON c.customer_id=o.customer_id
  WHERE c.customer_id=#{customerId}
</select>
```

4）Mybatis 全局注册 Mapper 文件

```
<!-- 注册Mapper配置文件：告诉Mybatis我们的Mapper配置文件的位置 -->
<mappers>
  <!-- 在mapper标签的resource属性中指定Mapper配置文件以“类路径根目录”为基准的相对路径 -->
  <mapper resource="mappers/OrderMapper.xml"/>
  <mapper resource="mappers/CustomerMapper.xml"/>
</mappers>
```

5）测试效果

```
@Test
public void testRelationshipToMulti() {

  CustomerMapper customerMapper = session.getMapper(CustomerMapper.class);
  // 查询Customer对象同时将关联的Order集合查询出来
  Customer customer = customerMapper.selectCustomerWithOrderList(1);
  log.info("customer.getCustomerId() = " + customer.getCustomerId());
  log.info("customer.getCustomerName() = " + customer.getCustomerName());
  List<Order> orderList = customer.getOrderList();
  for (Order order : orderList) {
    log.info("order = " + order);
  }
}
```

10、MyBatis 多表映射总结
多表关系实体类设计思路：
1）**对一**，属性中包含对方对象；（一个用户对应多个订单）
配置项关键词：**association 标签/javaType 属性/property 属性**；
2）**对多**，属性中包含对方对象集合；（一个订单对应一个用户）
配置项关键词：**collection 标签/ofType 属性/property 属性**；
3）只有真实发生多表查询时，才需要设计和修改实体类，否则不提前设计和修改实体类；
4）无论多少张表联查，实体类设计都是两两考虑；
5）在**查询映射的时候**，只需要**关注本次查询相关的属性**，例如：查询订单和对应的客户，就不要关注客户中的订单集合（不要形成死循环）。

```
public class Customer {
  private Integer customerId;
  private String customerName;
  private List<Order> orderList;// 体现的是对多的关系
}

public class Order {
  private Integer orderId;
  private String orderName;
  private Customer customer;// 体现的是对一的关系
}
```

11、多表映射优化

setting属性	属性含义	可选值	默认值
autoMappingBehavior	指定 MyBatis 应如何自动映射列到字段或属性。 NONE 表示关闭自动映射；PARTIAL 只会自动映射没有定义嵌套结果映射的字段。 FULL 会自动映射任何复杂的结果集（无论是否嵌套）。	NONE, PARTIAL, FULL	PARTIAL

将 autoMappingBehavior 设置为 full，进行多表 resultMap 映射的时候，可以省略符合列和属性命名映射规则（列名=属性名，或者开启驼峰映射也可以自定映射）的 result 标签。
修改 mybati-sconfig.xml：

```
<!--开启resultMap自动映射 -->
<setting name="autoMappingBehavior" value="FULL"/>
```

修改 teacherMapper.xml：

```
<resultMap id="teacherMap" type="teacher">
    <id property="tId" column="t_id" />
    <!-- 开启自动映射,并且开启驼峰式支持!可以省略 result!-->
<!--        <result property="tName" column="t_name" />-->
    <collection property="students" ofType="student" >
        <id property="sId" column="s_id" />
<!--            <result property="sName" column="s_name" />-->
    </collection>
</resultMap>
```

12、MyBatis 动态语句
1）if 和 where 标签
1> **<if></if>标签**：判断传入的参数，**有选择的添加 SQL 语句片段**；**test 属性**：内部**做比较运算**，**true** 则将标签内的 sql 语句进行**拼接**，**false** 则**不拼接**标签内的语句。
【注意】> **可以访问实体类的属性**，不能访问数据库表的字段；
> 判断语句：“key 比较符号 值 and|or key 比较符号 值”；
其中，**大于和小于符号**不推荐直接写符号，采用**实体符号&gt** 和**$lt**。
2> **<where></where>标签**作用有两个：
- **自动添加 where 关键字**，where 内部**有**任何一个 if **满足**，**自动添加** where 关键字，**不满足则去掉 where**；
- **自动去掉**多余的 **and 和 or 关键字**。

```
<!-- List<Employee> selectEmployeeByCondition(Employee employee); -->
<select id="selectEmployeeByCondition" resultType="employee">
    select emp_id,emp_name,emp_salary from t_emp
    <!-- where标签会自动去掉“标签体内前面多余的and/or” -->
    <where>
        <!-- 在if标签的test属性中，可以访问实体类的属性，不可以访问数据库表的字段 -->
        <if test="empName != null">
            <!-- 在if标签内部，需要访问接口的参数时还是正常写#{} -->
            or emp_name=#{empName}
        </if>
        <if test="empSalary &gt; 2000">
            or emp_salary>#{empSalary}
        </if>
        <!--
         第一种情况：所有条件都满足 WHERE emp_name=? or emp_salary>?
         第二种情况：部分条件满足 WHERE emp_salary>?
         第三种情况：所有条件都不满足 没有where子句
         -->
    </where>
</select>
```

2）set 标签：**自动添加 set 关键字**，并**去掉多余的逗号**

```
<!-- void updateEmployeeDynamic(Employee employee) -->
<update id="updateEmployeeDynamic">
    update t_emp
    <!-- set emp_name=#{empName},emp_salary=#{empSalary} -->
    <!-- 使用set标签动态管理set子句，并且动态去掉两端多余的逗号 -->
    <set>
        <if test="empName != null">
            emp_name=#{empName},
        </if>
        <if test="empSalary &lt; 3000">
            emp_salary=#{empSalary},
        </if>
    </set>
    where emp_id=#{empId}
    <!--
         第一种情况：所有条件都满足 SET emp_name=?, emp_salary=?
         第二种情况：部分条件满足 SET emp_salary=?
         第三种情况：所有条件都不满足 update t_emp where emp_id=?
            没有set子句的update语句会导致SQL语法错误
     -->
</update>
```

3）choose/when/otherwise 标签
在**多个分支条件**中，**仅执行一个**。
- 从上到下依次执行条件判断；
- 遇到的第一个满足条件的分支会被采纳，后面的分支都将不被考虑；
- 如果所有的 when 分支都不满足，那么就执行 otherwise 分支。

```
<!-- List<Employee> selectEmployeeByConditionByChoose(Employee employee) -->
<select id="selectEmployeeByConditionByChoose"
resultType="com.atguigu.mybatis.entity.Employee">
    select emp_id,emp_name,emp_salary from t_emp
    where
    <choose>
        <when test="empName != null">emp_name=#{empName}</when>
        <when test="empSalary &lt; 3000">emp_salary &lt; 3000</when>
        <otherwise>1=1</otherwise>
    </choose>
    <!--
     第一种情况：第一个when满足条件 where emp_name=?
     第二种情况：第二个when满足条件 where emp_salary < 3000
     第三种情况：两个when都不满足 where 1=1 执行了otherwise
     -->
</select>
```

4）foreach 标签
1> **collection 属性**：要**遍历的集合**；
2> **item 属性**：遍历集合的过程中能得到每一个具体对象，在 item 属性中设置一个名字，将来**通过 item 属性这个名字引用遍历出来的对象**；
3> **separator 属性**：指定当 foreach 标签的标签体**重复拼接字符串时**，各个标签体**字符串之间的分隔符**；
4> **open 属性**：**遍历之前**要添加的字符串；
5> **close 属性**：**遍历结束**要添加的字符串；
6> **index 属性**：这里起一个名字，便于后面引用；
遍历 List 集合，这里能够得到 List 集合的索引值；
遍历 Map 集合，这里能够得到 Map 集合的 key。

```
<!--int insertBatch(@Param("list")List<Employee> employeeList)-->
<insert id="insertBatch">
    insert into t_emp (emp_name,emp_salary) values
        <foreach collection="list" item="employee" separator="," index="myIndex">
        <!-- 在foreach标签内部如果需要引用遍历得到的具体的一个对象，需要使用item属性声明的名称 -->
            (#{emp.empName},#{emp.empSalary})
        </foreach>
</insert>
```

【注意】- **批量更新**时需要**多条 SQL 语句拼**起来，用**分号**分开。也就是一次性发送多条 SQL 语句让数据库执行。此时需要在数据库连接信息的 URL 地址中设置：**allowMultiQueries=true**

```
atguigu.dev.url=jdbc:mysql:///mybatis-example?allowMultiQueries=true
```

```
<!-- int updateEmployeeBatch(@Param("empList") List<Employee> empList) -->
<update id="updateEmployeeBatch">
    <foreach collection="empList" item="emp" separator=";">
        update t_emp set emp_name=#{emp.empName} where emp_id=#{emp.empId}
    </foreach>
</update>
```

- **collection 属性**：默认参数是[**arg0**|**collection**|**list**]，在开发中，建议使用**@Param** 注解**明确声明变量的名称**，然后在 foreach 标签的 collection 属性中按照@Param 注解指定的名称来引用传入的参数。

5）trim 标签(了解)
使用 trim 标签控制条件部分两端是否包含某些字符：
1> **prefix 属性**：指定要动态添加的**前缀**；
2> **suffix 属性**：指定要动态添加的**后缀**；
3> **prefixOverrides 属性**：指定要动态**去掉的前缀**，使用“|”分隔有可能的多个值；
4> **suffixOverrides 属性**：指定要动态**去掉的后缀**，使用“|”分隔有可能的多个值；

```
<!-- List<Employee> selectEmployeeByConditionByTrim(Employee employee) -->
<select id="selectEmployeeByConditionByTrim"
resultType="com.atguigu.mybatis.entity.Employee">
    select emp_id,emp_name,emp_age,emp_salary,emp_gender
    from t_emp
    <!-- 当前例子用where标签实现更简洁，但是trim标签更灵活，可以用在任何有需要的地方 -->
    <trim prefix="where" suffixOverrides="and|or">
        <if test="empName != null">
            emp_name=#{empName} and
        </if>
        <if test="empSalary &gt; 3000">
            emp_salary>#{empSalary} and
        </if>
        <if test="empAge &lt;= 20">
            emp_age=#{empAge} or
        </if>
        <if test="empGender=='male'">
            emp_gender=#{empGender}
        </if>
    </trim>
</select>
```

6）sql 片段
使用**<sql id=“标识”></sql>标签**抽取重复的 SQL 片段；
使用**<include refid=“上面 id 的标识”>标签**引用已抽取的 SQL 片段

```
<!-- 使用sql标签抽取重复出现的SQL片段 -->
<sql id="mySelectSql">
    select emp_id,emp_name,emp_age,emp_salary,emp_gender from t_emp
</sql>
```

```
<!-- 使用include标签引用声明的SQL片段 -->
<include refid="mySelectSql"/>
```

13、Mapper 批量映射优化
需求:Mapper 配置文件很多时,在全局配置文件中需要一个一个注册。
1）**配置方式**：Mybatis 允许在指定 Mapper 映射文件时，只**指定其所在的包**，此时这个包下的所有 Mapper 配置文件**将被自动加载、注册**，比较方便。

```
<mappers>
    <package name="com.atguigu.mapper"/>
</mappers>
```

2）**资源创建要求**
1> **Mapper 接口**和 **Mapper 配置文件名称**需要**一致**，即
- Mapper 接口：**EmployeeMapper.java**
- Mapper 配置文件：**EmployeeMapper.xml**
2> **Mapper 配置文件**放在 **Mapper 接口所在的包内**，即
-可以将 mapper.xml 文件放在 mapper 接口所在的包，pom.xml 配置：

```
<build>
    <resources>
        <resource>
            <directory>src/main/java</directory>
            <includes>
                <include>**/*.*</include>
            </includes>
        </resource>
    </resources>
</build>
```

- 可以**在 resources 下**创建 **mapper 接口包一致的文件夹结构**存放 mapper.xml 文件（使用/分割）
springboot 中的位置：resources/mapper/UserMapper.xml 即可。

14、PageHelper 插件使用
1）pom.xml **引入依赖** pageHelper

```
<dependency>
    <groupId>com.github.pagehelper</groupId>
    <artifactId>pagehelper</artifactId>
    <version>5.1.11</version>
</dependency>
```

2）在 MyBatis 的配置文件中**配置 PageHelper 的分页插件**

```
<plugins>
    <plugin interceptor="com.github.pagehelper.PageInterceptor">
        <property name="helperDialect" value="mysql"/>
    </plugin>
</plugins>
```

3）页插件使用，在查询方法中使用分页：
1> **先设置分页数据** PageHelper.startPage(currentPage,pageSize)；
2> **紧接着正常查询**获取所有数据；
3> 将查询数据**封装到**一个 **PageInfo** 的分页实体类中；
4> **使用 PageInfo** 获取分页数据。

```
@Test
public void testTeacherRelationshipToMulti() {

    TeacherMapper teacherMapper = session.getMapper(TeacherMapper.class);
    // 调用之前，先设置分页数据（当前是第几页，每页显示多少个）
    PageHelper.startPage(1,2);

    // 正常查询获取所有数据
    List<Teacher> allTeachers = teacherMapper.findAllTeachers();
    // 将查询数据封装到一个PageInfo的分页实体类中(一共有多少页，一共有多少条等等信息)
    PageInfo<Teacher> pageInfo = new PageInfo<>(allTeachers);

    //PageInfo获取分页数据
    long total = pageInfo.getTotal(); // 获取总记录数
    int pages = pageInfo.getPages();  // 获取总页数
    int pageNum = pageInfo.getPageNum(); // 获取当前页码
    int pageSize = pageInfo.getPageSize(); // 获取每页显示记录数
    List<Teacher> teachers = pageInfo.getList(); //获取查询页的数据集合
    teachers.forEach(System.out::println);

}
```

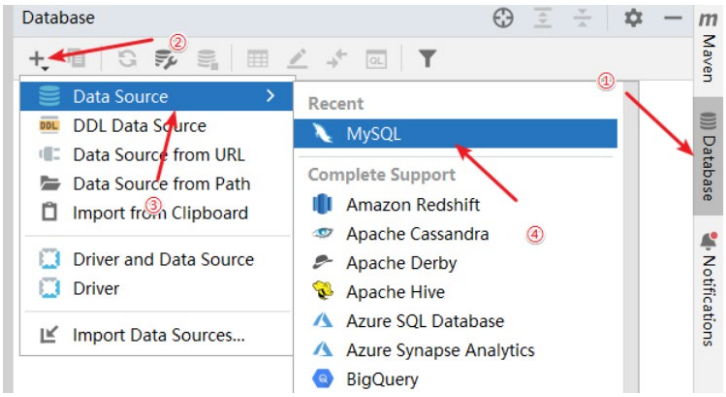
15、ORM 思维（让我们可以使用面向对象思维进行数据库操作）
1）ORM（Object-Relational Mapping，对象-关系映射）是一种将**数据库**和**面向对象编程语言中的对象**之间**进行转换**的技术。它将**对象**和**关系数据库的概念**进行**映射**，最后**通过方法调用**进行**数据库操作**。
2）ORM 框架通常有**半自动**和**全自动**两种方式：
- **半自动 ORM** 通常需要程序员**手动编写 SQL 语句或者配置文件**，将实体类和数据表进行映射，还需要**手动将查询的结果集转换成实体对象**。
- **全自动 ORM** 则是将实体类和数据表进行**自动映射**，使用 API 进行数据库操作时，**ORM 框架会自动执行 SQL 语句**并**将查询结果转换成实体对象**，程序员无需再手动编写 SQL 语句和转换代码。
3）下面是**半自动**和**全自动** ORM 框架的**区别**：
1> **映射方式**：（手动/自动**实体类**和**数据表**映射）
半自动 ORM 框架需要程序员**手动指定**实体类和数据表之间的**映射关系**，通常使用 **XML 文件**或**注解方式**来指定；
全自动 ORM 框架可以**自动**进行实体类和数据表的**映射**，无需手动干预。
2> **查询方式**：（手动/自动**编写** SQL 语句和**转换**成实体对象）
半自动 ORM 框架通常需要程序员**手动编写 SQL 语句**并将查询结果集**转换成实体对象**；
全自动 ORM 框架可以**自动组装 SQL 语句、执行查询操作**，并将查询结果**转换成实体对象**。
3> **性能**：（手动/自动**优化 SQL 语句**）
由于半自动 ORM 框架需要**手动编写 SQL 语句**，因此程序员必须对 SQL 语句和数据库的底层知识有一定的了解，才能编写高效的 SQL 语句；
全自动 ORM 框架通过**自动优化生成的 SQL 语句**来提高性能，程序员无需进行优化。
4> 常见**半自动** ORM 框架：**MyBatis** 等；
常见**全自动** ORM 框架:**Hibernate/Spring Data JPA/MyBatis-Plus** 等。

16、MyBatis 的逆向工程
MyBatis 的逆向工程是一种自动化生成持久层代码和映射文件的工具，它可以根据**数据库表结构**和**设置的参数生成**对应的**实体类**、**Mapper.xml 文件**、**Mapper 接口**等代码文件，简化了手动生成的过程。
MyBatis 的**逆向工程有两种方式**：通过 **MyBatis Generator 插件**实现和通过 **Maven 插件**实现。
【注意】逆向工程**只能生成单表 crud** 的操作，**多表查询**依然**需要编写**。

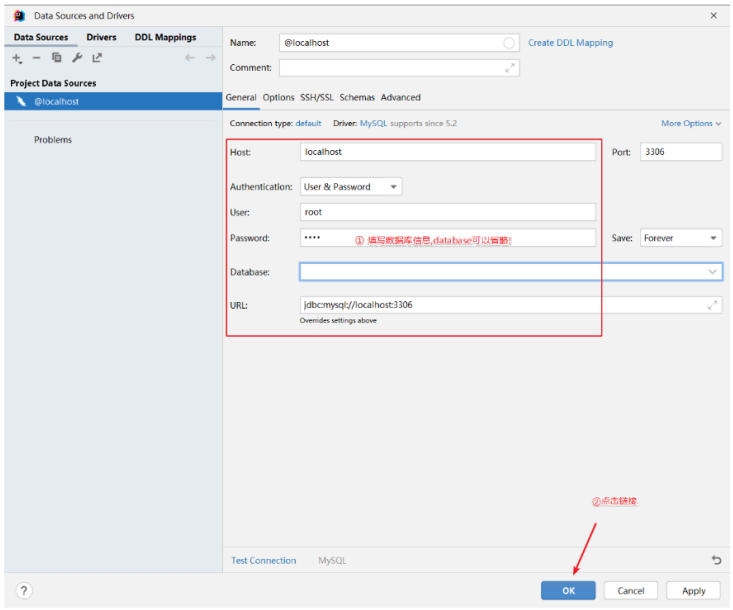
17、逆向工程插件 MyBatisX 使用

MyBatisX 是一个 MyBatis 的代码生成插件，可以通过简单的配置和操作快速生成 MyBatis Mapper、pojo 类和 Mapper.xml 文件。

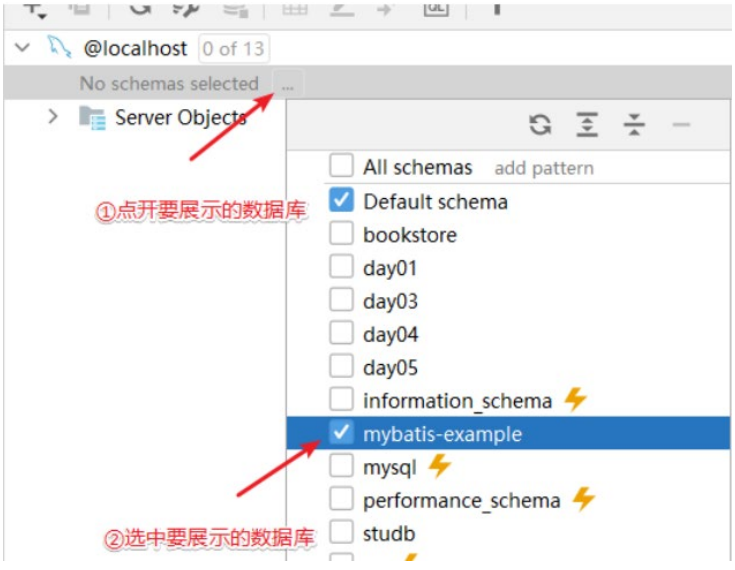
1）使用 IntelliJ IDEA 连接数据库



2）填写信息

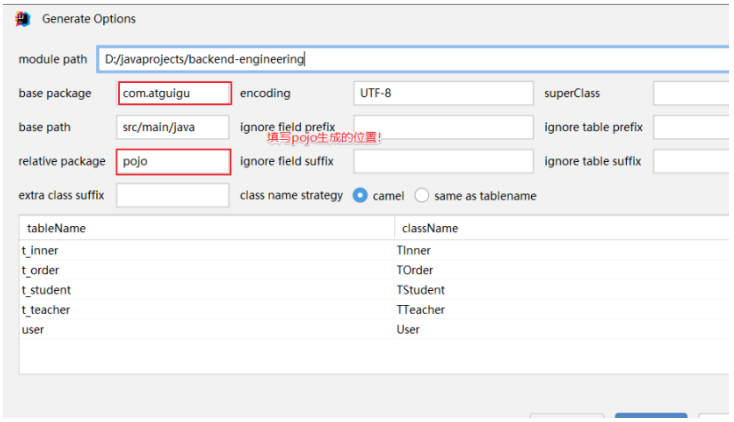


3）展示库表

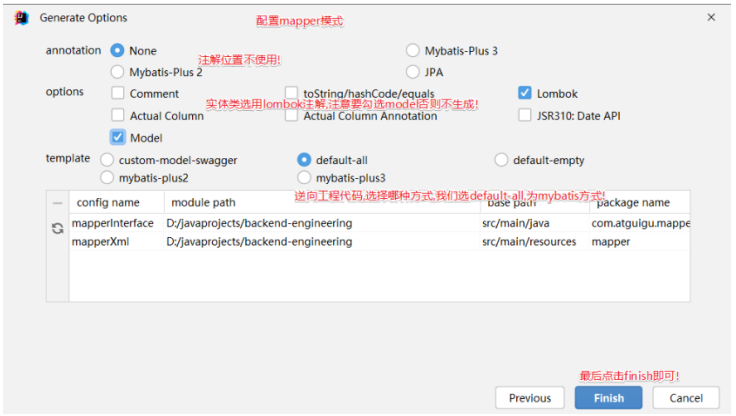


4）逆向工程使用

填写 pojo 生成的位置



配置 mapper 模式



【补充】

```
public class MyBatisTest {

    @Test
    public void testSelectEmployee() throws IOException {

        // 1.创建SqlSessionFactory对象
        // ②声明Mybatis全局配置文件的路径
        String mybatisConfigFilePath = "mybatis-config.xml";

        // ③以输入流的形式加载Mybatis配置文件
        InputStream inputStream =
Resources.getResourceAsStream(mybatisConfigFilePath);

        // ④基于读取Mybatis配置文件的输入流创建SqlSessionFactory对象
        SqlSessionFactory sessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);

        // 2.使用SqlSessionFactory对象开启一个会话
        SqlSession session = sessionFactory.openSession();

        // 3.根据EmployeeMapper接口的Class对象获取Mapper接口类型的对象(动态代理技术)
        EmployeeMapper employeeMapper = session.getMapper(EmployeeMapper.class);

        // 4. 调用代理类方法既可以触发对应的SQL语句
        Employee employee = employeeMapper.selectEmployee(1);

        System.out.println("employee = " + employee);

        // 4.关闭SqlSession
        session.commit(); //提交事务 [DQL不需要,其他需要]
        session.close(); //关闭会话

    }
}
```