一、简介

1、MyBatis简介

MyBatis 是一款优秀的持久层框架，它支持自定义 SQL、存储过程以及高级映射。MyBatis 免除了几乎所有的 JDBC 代码以及设置参数和获取结果集的工作。MyBatis 可以通过简单的 XML 或注解来配置和映射原始类型、接口和 Java POJO（Plain Old Java Objects，普通老式 Java 对象）为数据库中的记录。

2、MyBatis特性

1） MyBatis 是支持定制化 SQL、存储过程以及高级映射的优秀的持久层框架

2） MyBatis 避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集

3） MyBatis可以使用简单的XML或注解用于配置和原始映射，将接口和Java的POJO（Plain Old Java Objects，普通的Java对象）映射成数据库中的记录

4） MyBatis 是一个 半自动的ORM（Object Relation Mapping）框架

3、和其它持久化层技术对比

- 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
- 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
  - 内部自动生产的 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
- 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，但是完全能够接受

二、搭建MyBatis

1、创建Maven工程

引入依赖

```
<dependencies>
    <!-- Mybatis核心 -->
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.5.7</version>
```

```
    </dependency>
    <!-- junit测试 -->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>
    <!-- MySQL驱动 -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.3</version>
    </dependency>
</dependencies>
```

2、创建MyBatis的核心配置文件

核心配置文件存放的位置是src/main/resources目录下

```
<?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>

    <!--设置日志-->
    <settings>
        <setting name="logImpl" value="STDOUT_LOGGING"/>
    </settings>

    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <!-- 配置数据源：创建Connection对象 -->
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.jdbc.Driver"/>
                <property name="url" value="jdbc:mysql:///mybatis"/>
                <property name="username" value="root"/>
                <property name="password" value="root"/>
            </dataSource>
        </environment>
    </environments>

    <!-- 指定其他mapper文件的位置 -->
    <mappers>
        <mapper resource="mapper/UserMapper.xml"/>
    </mappers>
</configuration>
```

3、创建mapper接口

MyBatis中的mapper接口相当于以前的dao。但是区别在于，mapper仅仅是接口，我们不需要 提供实现类。

```java
public interface UserMapper {

    /**
     * MyBatis面向接口编程的两个一致：
     * 1.映射文件的namespace要和mapper接口的全类名保持一致
     * 2.映射文件中的SQL语句的id要和mapper接口中的方法一致
     */


    /**
     * 添加用户
     * @return
     */
    int insertUser();
}
```

4、创建MyBatis的映射文件

相关概念：

- ORM（Object Relationship Mapping）对象关系映射。
- 对象：Java的实体类对象
- 关系：关系型数据库
- 映射：二者之间的对应关系

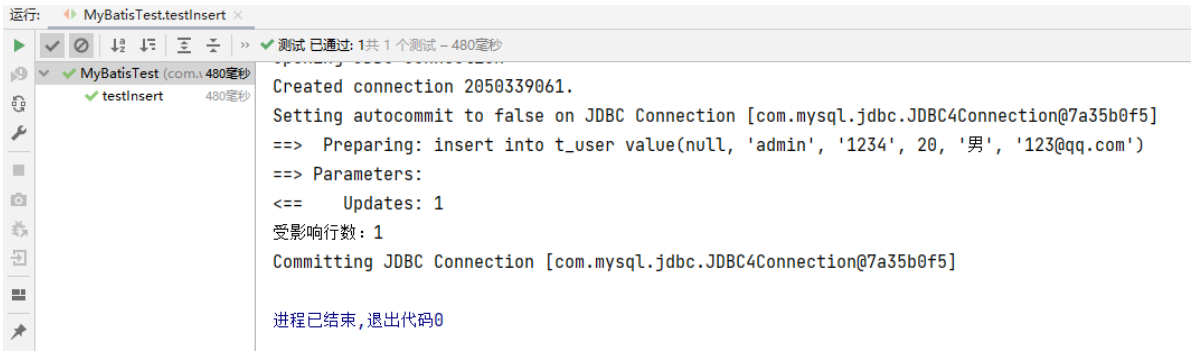
Java概念	数据库概念
类	表
属性	字段/列
对象	记录/行

> 1、映射文件的命名规则：
>
> 表所对应的实体类的类名+Mapper.xml
>
> 例如：表t_user，映射的实体类为User，所对应的映射文件为UserMapper.xml
>
> 因此一个映射文件对应一个实体类，对应一张表的操作
>
> MyBatis映射文件用于编写SQL，访问以及操作表中的数据 MyBatis映射文件存放的位置是src/main/resources/mappers目录下
>
> 2、MyBatis中可以面向接口操作数据，要保证两个一致：
>
> a>mapper接口的全类名和映射文件的命名空间（namespace）保持一致
>
> b>mapper接口中方法的方法名和映射文件中编写SQL的标签的id属性保持一致

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.ww.mybatis.mapper.UserMapper">
    <!-- 使用select, update, delete, insert 标签写sql -->
    <insert id="insertUser">
        insert into t_user value(null, 'admin', '1234', 20, '男',
'123@qq.com')
    </insert>

</mapper>
```

5、测试

```java
public class MyBatisTest {

    @Test
    public void testInsert() throws IOException {
        // 1.加载核心配置文件
        InputStream inputStream = Resources.getResourceAsStream("mybatis-
config.xml");
        // 2.获取SqlSessionFactoryBuilder
        SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new
SqlSessionFactoryBuilder();
        // 3.获取SqlSessionFactory
        SqlSessionFactory sqlSessionFactory =
sqlSessionFactoryBuilder.build(inputStream);
        // 4.获取SqlSession
        SqlSession sqlSession = sqlSessionFactory.openSession();
        // 5.获取mapper接口对象
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
        // 6.测试功能
        // 调用UserMapper接口中的方法，就可以根据UserMapper的全类名匹配元素文件，通过
调用的方法名匹配映射文件中的SQL标签，并执行标签中的SQL语句
        int result = mapper.insertUser();
        System.out.println("受影响行数：" + result);
        // 7.提交事务
        sqlSession.commit();
    }
}
```



> SqlSession：代表Java程序和数据库之间的会话。（HttpSession是Java程序和浏览器之间的 会话）

> SqlSessionFactory：是“生产”SqlSession的“工厂”。
>
> 工厂模式：如果创建某一个对象，使用的过程基本固定，那么我们就可以把创建这个对象的 相关代码封装到一个“工厂类”中，以后都使用这个工厂类来“生产”我们需要的对象。

6、优化

6.1、自动提交

SqlSession默认不自动提交事务，若需要自动提交事务，则进行以下修改即可：

```
SqlSession sqlSession = sqlSessionFactory.openSession();
SqlSession sqlSession = sqlSessionFactory.openSession(true);
```

6.2、加入log4j日志功能

日志级别

> FATAL(致命)>ERROR(错误)>WARN(警告)>INFO(信息)>DEBUG(调试)
>
> 左到右打印的内容越来越详细

配置步骤

1. 引入依赖

```
<!-- log4j日志 -->
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
</dependency>
```

2. 在src/main/resources目录下创建log4j.xml文件，并添加以下内容

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
    <appender name="STDOUT" class="org.apache.log4j.ConsoleAppender">
        <param name="Encoding" value="UTF-8" />
        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern" value="%-5p %d{MM-dd
HH:mm:ss,SSS}
%m (%F:%L) \n" />
        </layout>
    </appender>
    <logger name="java.sql">
        <level value="debug" />
    </logger>
    <logger name="org.apache.ibatis">
        <level value="info" />
    </logger>
    <root>
        <level value="debug" />
```

```
        <appender-ref ref="STDOUT" />
    </root>
</log4j:configuration>
```

> 注：http://jakarta.apache.org/log4j/爆红是不影响使用的

3. 优化结果显示

```
D:\JDK\bin\java.exe ...
DEBUG 07-05 19:31:51,261 ==>  Preparing: insert into t_user value(null, 'admin', '1234', 20, '男', '123@qq.com') (BaseJdbcLogger.java:137)
DEBUG 07-05 19:31:51,284 ==> Parameters:  (BaseJdbcLogger.java:137)
DEBUG 07-05 19:31:51,289 <==    Updates: 1 (BaseJdbcLogger.java:137)
受影响行数：1
```

7、测试查询功能

1. 编写接口方法

```
User getUserById();
```

2. 编写sql语句

```
<select id="getUserById" resultType="com.ww.mybatis.pojo.User">
    select * from t_user where id = 3
</select>
```

3. 编写测试方法

```
@Test
public void testGetById() throws IOException {
    // 1.加载核心配置文件
    InputStream inputStream = Resources.getResourceAsStream("mybatis-config.xml");
    // 2.获取SqlSessionFactoryBuilder
    SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new SqlSessionFactoryBuilder();
    // 3.获取SqlSessionFactory
    SqlSessionFactory sqlSessionFactory = sqlSessionFactoryBuilder.build(inputStream);
    // 4。获取SqlSession
    SqlSession sqlSession = sqlSessionFactory.openSession(true);
    // 5.获取mapper接口对象
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);
    // 6.测试功能
    User result = mapper.getUserById();
    System.out.println("查询结果：" + result);
}
```

4. 结果

```
DEBUG 07-05 19:50:13,732 ==>  Preparing: select * from t_user where id = 3 (BaseJdbcLogger.java:137)
DEBUG 07-05 19:50:13,756 ==> Parameters:  (BaseJdbcLogger.java:137)
DEBUG 07-05 19:50:13,774 <==      Total: 1 (BaseJdbcLogger.java:137)
查询结果：User{id=3, username='admin', password='1234', age=20, sex='男', email='123@qq.com'}
```

5. 注意

> 查询功能的标签必须设置resultType或resultMap

> resultType：设置默认的映射关系，字段名和属性名一致的时候使用
>
> resultMap：设置自定义的映射关系，字段名和属性名不一致的时候使用
>
> 当查询的数据为多条时，不能使用实体类作为返回值，只能使用集合，否则会抛出异常TooManyResultsException；但是若查询的数据只有一条，可以使用实体类或集合作为返回值

8、封装SqlSessionUtils工具类

```
public class SqlSessionUtils {

    public static SqlSession getSqlSession(){
        SqlSession sqlSession = null;
        try {
            // 1.加载核心配置文件
            InputStream inputStream =
Resources.getResourceAsStream("mybatis-config.xml");
            // 2.获取SqlSessionFactoryBuilder
            SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new
SqlSessionFactoryBuilder();
            // 3.获取SqlSessionFactory
            SqlSessionFactory sqlSessionFactory =
sqlSessionFactoryBuilder.build(inputStream);
            // 4。获取SqlSession
            sqlSession = sqlSessionFactory.openSession(true);
        } catch (IOException e) {
            e.printStackTrace();
        }
        return sqlSession;
    }
}
```

测试

```
@Test
public void testGetAllUser(){
    SqlSession sqlSession = SqlSessionUtils.getSqlSession();
    ParameterMapper mapper = sqlSession.getMapper(ParameterMapper.class);
    List<User> userList = mapper.getAllUser();
    userList.forEach(user -> System.out.println(user));
}
```

```
D:\JDK\bin\java.exe ...
DEBUG 07-05 21:44:03,499 ==>  Preparing: select * from t_user (BaseJdbcLogger.java:137)
DEBUG 07-05 21:44:03,525 ==> Parameters:  (BaseJdbcLogger.java:137)
DEBUG 07-05 21:44:03,546 <==      Total: 2 (BaseJdbcLogger.java:137)
User{id=3, username='admin', password='1234', age=20, sex='男', email='123@qq.com'}
User{id=4, username='张三', password='1234', age=20, sex='男', email='123@qq.com'}
```

三、核心配置文件详解

核心配置文件中的标签必须按照固定的顺序：
properties?,settings?,typeAliases?,typeHandlers?,objectFactory?,objectWrapperFactory?,reflectorFactory?,plugins?,environments?,databaseIdProvider?,mappers?

1、properties

```
<!--引入properties文件，此时就可以${属性名}的方式访问属性值-->
<properties resource="jdbc.properties"></properties>
```

jdbc.properties：

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql:///mybatis
jdbc.username=root
jdbc.password=root
```

2、settings

```
<settings>
    <!--将表中字段的下划线自动转换为驼峰-->
    <setting name="mapUnderscoreToCamelCase" value="true"/>
    <!--开启延迟加载-->
    <setting name="lazyLoadingEnabled" value="true"/>
</settings>
```

3、typeAliases

```
<typeAliases>
<!-- typeAlias：设置某个具体的类型的别名
     属性：
         type：需要设置别名的类型的全类名
         alias：设置此类型的别名，若不设置此属性，该类型拥有默认的别名，即类名且不区分大小写若设置此属性，此时该类型的别名只能使用alias所设置的值
-->
<!--<typeAlias type="com.ww.mybatis.bean.User"></typeAlias>-->
<!--<typeAlias type="com.ww.mybatis.bean.User" alias="user"></typeAlias>-->
<!--以包为单位，设置改包下所有的类型都拥有默认的别名，即类名且不区分大小写-->
<package name="com.ww.mybatis.bean"/>
</typeAliases>
```

4、environments

```
<!--
environments：设置多个连接数据库的环境
    属性：
        default：设置默认使用的环境的id
-->
<environments default="mysql_test">
    <!--
```

```
        environment：设置具体的连接数据库的环境信息
        属性：
            id：设置环境的唯一标识，可通过environments标签中的default设置某一个环境的id，表示默认使用的环境
    -->
    <environment id="mysql_test">
        <!--
            transactionManager：设置事务管理方式
            属性：
                type：设置事务管理方式，type="JDBC|MANAGED"
                type="JDBC"：设置当前环境的事务管理都必须手动处理
                type="MANAGED"：设置事务被管理，例如spring中的AOP
        -->
        <transactionManager type="JDBC"/>
            <!--
                dataSource：设置数据源
                属性：
                type：设置数据源的类型，type="POOLED|UNPOOLED|JNDI"
                type="POOLED"：使用数据库连接池，即会将创建的连接进行缓存，下次使用可以从缓存中直接获取，不需要重新创建
                type="UNPOOLED"：不使用数据库连接池，即每次使用连接都需要重新创建
                type="JNDI"：调用上下文中的数据源
            -->
            <dataSource type="POOLED">
                <!--设置驱动类的全类名-->
                <property name="driver" value="${jdbc.driver}"/>
                <!--设置连接数据库的连接地址-->
                <property name="url" value="${jdbc.url}"/>
                <!--设置连接数据库的用户名-->
                <property name="username" value="${jdbc.username}"/>
                <!--设置连接数据库的密码-->
                <property name="password" value="${jdbc.password}"/>
            </dataSource>
    </environment>
</environments>
```

5、mappers

```
<!--引入映射文件-->
<mappers>
    <mapper resource="UserMapper.xml"/>
    <!--
        以包为单位，将包下所有的映射文件引入核心配置文件
        注意：此方式必须保证mapper接口和mapper映射文件必须在相同的包下
    -->
    <package name="com.ww.mybatis.mapper"/>
</mappers>
```

四、MyBatis获取参数值的两种方式

MyBatis获取参数值的两种方式：`${}`和`#{}`

`${}`的本质就是字符串拼接，`#{}`的本质就是占位符赋值

`${}`使用字符串拼接的方式拼接sql，若为字符串类型或日期类型的字段进行赋值时，需要手动加单引号；但是`#{}`使用占位符赋值的方式拼接sql，此时为字符串类型或日期类型的字段进行赋值时，可以自动添加单引号

1、单个字面量类型的参数

若mapper接口中的方法参数为单个的字面量类型，此时可以使用`${}`和`#{}`以任意的名称获取参数的值，注意${}需要手动加单引号

实现按用户名查询用户信息

1. 编写mapper接口方法

```
/**
     * 根据用户名查询用户信息
     * @param username
     * @return
     */
    User getUserByUsername(String username);
```

2. 编写sql语句

`#{}`方式：

```
<select id="getUserByUsername" resultType="com.itww.mybatis.pojo.User">
   select * from t_user where username = #{username}
</select>
```

`${}`方式

```
<select id="getUserByUsername" resultType="com.itww.mybatis.pojo.User">
   select * from t_user where username = '${username}'
</select>
```

3. 编写测试方法

```
@Test
public void GetUserByUsername(){
    SqlSession sqlSession = SqlSessionUtils.getSqlSession();
    ParameterMapper mapper = sqlSession.getMapper(ParameterMapper.class);
    User user = mapper.getUserByUsername("张三");
    System.out.println(user);
}
```

4. 运行结果

`#{}`方式：

```
D:\JDK\bin\java.exe ...
DEBUG 07-05 22:05:43,298 ==>  Preparing: select * from t_user where username = ? (BaseJdbcLogger.java:137)
DEBUG 07-05 22:05:43,323 ==> Parameters: 张三(String) (BaseJdbcLogger.java:137)
DEBUG 07-05 22:05:43,338 <==      Total: 1 (BaseJdbcLogger.java:137)
User{id=4, username='张三', password='1234', age=20, sex='男', email='123@qq.com'}
```

${}方式：

```
D:\JDK\bin\java.exe ...
DEBUG 07-05 22:06:20,052 ==>  Preparing: select * from t_user where username = '张三' (BaseJdbcLogger.java:137)
DEBUG 07-05 22:06:20,080 ==> Parameters:  (BaseJdbcLogger.java:137)
DEBUG 07-05 22:06:20,097 <==      Total: 1 (BaseJdbcLogger.java:137)
User{id=4, username='张三', password='1234', age=20, sex='男', email='123@qq.com'}
```

2、多个字面量类型的参数

若mapper接口中的方法参数为多个时

此时MyBatis会自动将这些参数放在一个map集合中，有以下两种方式：

a>以arg0,arg1...为键，以参数为值

b>以 param1,param2...为键，以参数为值

因此只需要通过${}和#{}访问map集合的键就可以获取相对应的值，注意${}需要手动加单引号

实现多个子面量类型的参数

1. 编写mapper接口方法

```java
/**
 * 验证登录
 * @param username
 * @param password
 * @return
 */
User checkLogin(String username, String password);
```

2. 编写sql语句

```xml
<select id="checkLogin" resultType="com.itww.mybatis.pojo.User">
   select * from t_user where username = #{arg0} and password = #{arg1}
</select>
```

3. 编写测试方法

```java
@Test
public void checkLogin(){
    SqlSession sqlSession = SqlSessionUtils.getSqlSession();
    ParameterMapper mapper = sqlSession.getMapper(ParameterMapper.class);
    User user = mapper.checkLogin("张三","1234");
    System.out.println(user);
}
```

4. 运行结果

```
D:\JDK\bin\java.exe ...
DEBUG 07-06 21:08:13,198 ==>  Preparing: select * from t_user where username = ? and password = ? (BaseJdbcLogger.java:137)
DEBUG 07-06 21:08:13,223 ==> Parameters: 张三(String), 1234(String) (BaseJdbcLogger.java:137)
DEBUG 07-06 21:08:13,240 <==      Total: 1 (BaseJdbcLogger.java:137)
User{id=4, username='张三', password='1234', age=20, sex='男', email='123@qq.com'}

进程已结束,退出代码0
```

3、map集合类型的参数

若mapper接口中的方法需要的参数为多个时

此时可以手动创建map集合，将这些数据放在map中只需要通过`${}`和`#{}`访问map集合的键就可以获取相对应的值，注意`${}`需要手动加单引号

实现map集合类型的参数

1. 编写mapper接口方法

```
/**
* 验证登录，参数为map
* @param map
* @return
*/
User checkLoginByMap(Map<String, Object> map);
```

2. 编写sql语句

```
<select id="checkLoginByMap" resultType="com.itww.mybatis.pojo.User">
    select * from t_user where username = #{username} and password = #{password}
</select>
```

3. 编写测试方法

```
@Test
public void checkLoginByMap(){
    SqlSession sqlSession = SqlSessionUtils.getSqlSession();
    ParameterMapper mapper = sqlSession.getMapper(ParameterMapper.class);
    Map<String, Object> map = new HashMap<>();
    map.put("username", "张三");
    map.put("password", "1234");
    User user = mapper.checkLoginByMap(map);
    System.out.println(user);
}
```

4. 运行结果

```
D:\JDK\bin\java.exe ...
DEBUG 07-06 21:20:43,250 ==>  Preparing: select * from t_user where username = ? and password = ? (BaseJdbcLogger.java:137)
DEBUG 07-06 21:20:43,275 ==> Parameters: 张三(String), 1234(String) (BaseJdbcLogger.java:137)
DEBUG 07-06 21:20:43,291 <==      Total: 1 (BaseJdbcLogger.java:137)
User{id=4, username='张三', password='1234', age=20, sex='男', email='123@qq.com'}

进程已结束,退出代码0
```

4、实体类类型的参数

若mapper接口中的方法参数为实体类对象时

此时可以使用`${}`和`#{}`，通过访问实体类对象中的属性名获取属性值，注意`${}`需要手动加单引号

实现实体类类型的参数

1. 编写mapper接口方法

```java
/**
 * 添加用户
 * @param user
 * @return
 */
int insertUser(User user);
```

2. 编写sql语句

```xml
<insert id="insertUser">
   insert into t_user values(null, #{username}, #{password}, #{age}, #{sex}, #{email})
</insert>
```

3. 编写测试方法

```java
@Test
public void insertUser(){
    SqlSession sqlSession = SqlSessionUtils.getSqlSession();
    ParameterMapper mapper = sqlSession.getMapper(ParameterMapper.class);
    User user = new User(null, "王五", "1234", 18, "男" , "123@qq.com");
    int rows = mapper.insertUser(user);
    System.out.println(rows);
}
```

4. 运行结果

```
D:\JDK\bin\java.exe ...
DEBUG 07-06 21:28:31,811 ==>  Preparing: insert into t_user values(null, ?, ?, ?, ?, ?) (BaseJdbcLogger.java:137)
DEBUG 07-06 21:28:31,839 ==> Parameters: 王五(String), 1234(String), 18(Integer), 男(String), 123@qq.com(String) (BaseJdbcLogger.java:137)
DEBUG 07-06 21:28:31,842 <==    Updates: 1 (BaseJdbcLogger.java:137)
1

进程已结束,退出代码0
```

id	username	password	age	sex	email
3	admin	1234	20	男	123@qq.com
4	张三	1234	20	男	123@qq.com
6	王五	1234	18	男	123@qq.com

5、使用@Param标识参数

可以通过@Param注解标识mapper接口中的方法参数

此时，会将这些参数放在map集合中，有两种方式：

a>以@Param注解的value属性值为键，以参数为值

b>以 param1,param2...为键，以参数为值；

只需要通过 `${}` 和 `#{}` 访问map集合的键就可以获取相对应的值，注意 `${}` 需要手动加单引号

实现使用@Param标识参数

1. 编写mapper接口方法

```java
/**
 * 验证登录，使用@Param
 * @param username
 * @param password
 * @return
 */
User checkLoginByParam(@Param("username") String username,
@Param("password") String password);
```

2. 编写sql语句

```xml
<select id="checkLoginByParam" resultType="com.itww.mybatis.pojo.User">
   select * from t_user where username = #{username} and password = #
{password}
</select>
```

3. 编写测试方法

```java
@Test
public void checkLoginByParam(){
    SqlSession sqlSession = SqlSessionUtils.getSqlSession();
    ParameterMapper mapper = sqlSession.getMapper(ParameterMapper.class);
    User user = mapper.checkLoginByParam("张三", "1234");
    System.out.println(user);
}
```

4. 运行结果

```
D:\JDK\bin\java.exe ...
DEBUG 07-06 21:35:53,728 ==>  Preparing: select * from t_user where username = ? and password = ? (BaseJdbcLogger.java:137)
DEBUG 07-06 21:35:53,757 ==> Parameters: 张三(String), 1234(String) (BaseJdbcLogger.java:137)
DEBUG 07-06 21:35:53,774 <==      Total: 1 (BaseJdbcLogger.java:137)
User{id=4, username='张三', password='1234', age=20, sex='男', email='123@qq.com'}

进程已结束,退出代码0
```

6、总结

建议将任何类型的参数分为两种情况进行处理：

- 实体类类型的参数
- 使用@Param标识参数

五、MyBatis的各种查询功能

1、查询一个实体类对象

若查询出的数据只有一条，可以通过实体类对象接收

若查询出的数据有多条，一定不能通过实体类对象接收，此时会抛TooManyResultsException异常

1. 编写mapper接口方法

```java
/**
 * 根据id查询用户信息
 * @return
 */
User getUserById(@Param("id") Integer id);
```

2. 编写sql语句

```xml
<select id="getUserById" resultType="com.itww.mybatis.pojo.User">
    select * from t_user where id = #{id}
</select>
```

3. 编写测试方法

```java
@Test
public void getUserById(){
    SqlSession sqlSession = SqlSessionUtils.getSqlSession();
    SelectMapper mapper = sqlSession.getMapper(SelectMapper.class);
    User user = mapper.getUserById(3);
    System.out.println(user);
}
```

2、查询一个list集合

若查询出的数据有多条，可以通过List集合接收

1. 编写mapper接口方法

```java
/**
* 查询所有用户信息
* @return
*/
List<User> getUserList();
```

2. 编写sql语句

```xml
<select id="getUserList" resultType="com.itww.mybatis.pojo.User">
    select * from t_user
</select>
```

3. 编写测试方法

```java
@Test
public void getUserList(){
    SqlSession sqlSession = SqlSessionUtils.getSqlSession();
    SelectMapper mapper = sqlSession.getMapper(SelectMapper.class);
    List<User> userList = mapper.getUserList();
    userList.forEach(user -> System.out.println(user));
}
```

3、查询单个数据

在MyBatis中，对于Java中常用的类型都设置了类型别名，例如：

- java.lang.Integer-->int|integer
- int-->*int*|integer
- Map-->map,List-->list

1. 编写mapper接口方法

```
/**
 * 查询用户的总记录数
 * @return
 */
int getCount();
```

2. 编写sql语句

```
<select id="getCount" resultType="int">
    select count(id) from t_user
</select>
```

3. 编写测试方法

```
@Test
public void getCount(){
    SqlSession sqlSession = SqlSessionUtils.getSqlSession();
    SelectMapper mapper = sqlSession.getMapper(SelectMapper.class);
    int count = mapper.getCount();
    System.out.println("总记录数为：" + count);
}
```

4、查询一条数据为map集合

1. 编写mapper接口方法

```
/**
 * 根据用户id查询用户信息为map集合
 * @param id
 * @return
 */
Map<String, Object> getUserToMap(@Param("id") int id);
```

2. 编写sql语句

```
<select id="getUserToMap" resultType="map">
    select * from t_user where id = #{id}
</select>
```

3. 编写测试方法

```java
@Test
public void getUserToMap(){
    SqlSession sqlSession = SqlSessionUtils.getSqlSession();
    SelectMapper mapper = sqlSession.getMapper(SelectMapper.class);
    Map<String, Object> userToMap = mapper.getUserToMap(3);
    System.out.println(userToMap);
}
```

4. 运行结果

```
DEBUG 07-06 22:51:32,027 ==>  Preparing: select * from t_user where id = ? (BaseJdbcLogger.java:137)
DEBUG 07-06 22:51:32,055 ==> Parameters: 3(Integer) (BaseJdbcLogger.java:137)
DEBUG 07-06 22:51:32,071 <==      Total: 1 (BaseJdbcLogger.java:137)
{password=1234, sex=男, id=3, age=20, email=123@qq.com, username=admin}

进程已结束,退出代码0
```

5、查询多条数据为map集合

方式一

将表中的数据以map集合的方式查询，一条数据对应一个map；若有多条数据，就会产生多个map集合，此时可以将这些map放在一个list集合中获取

1. 编写mapper接口方法

```java
/**
 * 查询所有用户信息为map集合
 * @return
 */
List<Map<String, Object>> getAllUserToMap();
```

2. 编写sql语句

```xml
<select id="getAllUserToMap" resultType="java.util.Map">
    select * from t_user
</select>
```

3. 编写测试方法

```java
@Test
public void getAllUserToMap(){
    SqlSession sqlSession = SqlSessionUtils.getSqlSession();
    SelectMapper mapper = sqlSession.getMapper(SelectMapper.class);
    List<Map<String, Object>> allUserToMap = mapper.getAllUserToMap();
    allUserToMap.forEach(map -> System.out.println(map));
}
```

4. 运行结果

```
D:\JDK\bin\java.exe ...
DEBUG 07-06 22:56:47,674 ==>  Preparing: select * from t_user (BaseJdbcLogger.java:137)
DEBUG 07-06 22:56:47,699 ==> Parameters:  (BaseJdbcLogger.java:137)
DEBUG 07-06 22:56:47,716 <==      Total: 3 (BaseJdbcLogger.java:137)
{password=1234, sex=男, id=3, age=20, email=123@qq.com, username=admin}
{password=1234, sex=男, id=4, age=20, email=123@qq.com, username=张三}
{password=1234, sex=男, id=6, age=18, email=123@qq.com, username=王五}

进程已结束,退出代码0
```

方式二

将表中的数据以map集合的方式查询，一条数据对应一个map；若有多条数据，就会产生多个map集合，并且最终要以一个map的方式返回数据，此时需要通过@MapKey注解设置map集合的键，值是每条数据所对应的 map集合

1. 编写mapper接口方法

```java
/**
 * 查询所有用户信息为map集合
 * @return
 */
@MapKey("id")
Map<String, Object> getAllUserToMap();
```

2. 编写sql语句

```xml
<select id="getAllUserToMap" resultType="java.util.Map">
    select * from t_user
</select>
```

3. 编写测试方法

```java
@Test
public void getAllUserToMap(){
    SqlSession sqlSession = SqlSessionUtils.getSqlSession();
    SelectMapper mapper = sqlSession.getMapper(SelectMapper.class);
    Map<String, Object> allUserToMap = mapper.getAllUserToMap();
    System.out.println(allUserToMap);
}
```

4. 运行结果

```
D:\JDK\bin\java.exe ...
DEBUG 07-06 23:00:36,201 ==>  Preparing: select * from t_user (BaseJdbcLogger.java:137)
DEBUG 07-06 23:00:36,225 ==> Parameters:  (BaseJdbcLogger.java:137)
DEBUG 07-06 23:00:36,241 <==      Total: 3 (BaseJdbcLogger.java:137)
{3={password=1234, sex=男, id=3, age=20, email=123@qq.com, username=admin}, 4={password=1234, sex=男, id=4, age=20, email=123@qq.com, username=张三}, 6={password=1234, sex=男, id=6

进程已结束,退出代码0
```

六、特殊SQL的执行

1、模糊查询

1. 编写mapper接口方法

```java
/**
 * 根据用户名模糊查询用户信息
 * @param username
 * @return
 */
List<User> getUserByLike(@Param("username") String username);
```

2. 编写sql语句

```xml
<select id="getUserByLike" resultType="com.itww.mybatis.pojo.User">
    <!-- 方式一 -->
    select * from t_user where username like '%${username}%'
    <!-- 方式二 -->
    select * from t_user where username like concat('%', #{username}, '%')
    <!-- 方式三(最常用) -->
    select * from t_user where username like "%"#{username}"%"
</select>
```

3. 编写测试方法

```java
@Test
public void getUserByLike(){
    SqlSession sqlSession = SqlSessionUtils.getSqlSession();
    SQLMapper mapper = sqlSession.getMapper(SQLMapper.class);
    List<User> list = mapper.getUserByLike("a");
    System.out.println(list);
}
```

2、批量删除

1. 编写mapper接口方法

```java
/**
 * 批量删除
 * @param ids
 * @return
 */
int deleteMore(@Param("ids") String ids);
```

2. 编写sql语句

```xml
<!-- 此处不能使用#{},因为它会自动加单引号，在此处不合理 -->
<delete id="deleteMore">
    delete from t_user where id in (${ids})
</delete>
```

3. 编写测试方法

```java
@Test
public void deleteMore(){
    SqlSession sqlSession = SqlSessionUtils.getSqlSession();
    SQLMapper mapper = sqlSession.getMapper(SQLMapper.class);
    int result = mapper.deleteMore("1,2,6");
    System.out.println(result);
}

// SQL语句输出
delete from t_user where id in (1,2,6)
```

3、动态设置表名

1. 编写mapper接口方法

```java
/**
 * 查询指定表中的数据
 * @param tableName
 * @return
 */
List<User> getUserByTableName(@Param("tableName") String tableName);
```

2. 编写sql语句

```xml
<!-- 表名不能加单引号，所以不能用#{} -->
<select id="getUserByTableName" resultType="com.itww.mybatis.pojo.User">
    select * from ${tableName}
</select>
```

3. 编写测试方法

```java
@Test
public void getUserByTableName(){
    SqlSession sqlSession = SqlSessionUtils.getSqlSession();
    SQLMapper mapper = sqlSession.getMapper(SQLMapper.class);
    List<User> list = mapper.getUserByTableName("t_user");
    for (User user : list) {
        System.out.println(user);
    }
}
```

4、添加功能获取自增的主键

- useGeneratedKeys：设置使用自增的主键
- keyProperty：因为增删改有统一的返回值是受影响的行数，因此只能将获取的自增的主键放在传输的参数user对象的某个属性中

实现

1. 编写mapper接口方法

```java
/**
 * 添加用户信息
 * @param user
 * @return
 */
void insertUser(User user);
```

2. 编写sql语句

```xml
<insert id="insertUser" useGeneratedKeys="true" keyProperty="id">
    insert into t_user values(null,#{username},#{password},#{age},#{sex})
</insert>
```

3. 编写测试方法

```java
@Test
public void insertUser(){
    SqlSession sqlSession = SqlSessionUtils.getSqlSession();
    SQLMapper mapper = sqlSession.getMapper(SQLMapper.class);
    User user = new User(null, "王五", "1234", 18, "男" , "123@qq.com");
    mapper.insertUser(user);
    System.out.println(user);
}
```

4. 运作结果，可以看见user的id传入的是Null，但现在却有了值

```
DEBUG 07-07 19:10:32,547 ==>  Preparing: insert into t_user values(null,?,?,?,?,?) (BaseJdbcLogger.java:137)
DEBUG 07-07 19:10:32,574 ==> Parameters: 王五(String), 1234(String), 18(Integer), 男(String), 123@qq.com(String) (BaseJdbcLogger.java:137)
DEBUG 07-07 19:10:32,576 <==    Updates: 1 (BaseJdbcLogger.java:137)
User{id=7, username='王五', password='1234', age=18, sex='男', email='123@qq.com'}

进程已结束,退出代码0
```

七、自定义映射resultMap

1、resultMap处理字段和属性的映射关系

当数据库字段名与java实体类属性名不一致时，实现查找所有员工信息

1. 编写mapper接口方法

```java
/**
 * 查询所有员工信息
 * @return
 */
List<Emp> getAllEmp();
```

2. 编写sql语句

```xml
<select id="getAllEmp" resultType="com.ww.mybatis.pojo.Emp">
    select * from t_emp
</select>
```

3. 编写测试方法

```java
@Test
public void getAllEmp(){
    SqlSession sqlSession = SqlSessionUtils.getSqlSession();
    EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);
    List<Emp> list = mapper.getAllEmp();
    list.forEach(emp -> System.out.println(emp));
}
```

4. 运行结果，可以看到所查找到的empName属性全为null

```
DEBUG 07-07 19:36:22,740 ==>  Preparing: select * from t_emp (BaseJdbcLogger.java:137)
DEBUG 07-07 19:36:22,766 ==> Parameters:  (BaseJdbcLogger.java:137)
DEBUG 07-07 19:36:22,785 <==      Total: 5 (BaseJdbcLogger.java:137)
Emp{eid=1, empName='null' age=10, sex='男', email='123@qq.com'}
Emp{eid=2, empName='null' age=11, sex='男', email='123@qq.com'}
Emp{eid=3, empName='null' age=12, sex='男', email='123@qq.com'}
Emp{eid=4, empName='null' age=13, sex='女', email='123@qq.com'}
Emp{eid=5, empName='null' age=14, sex='女', email='123@qq.com'}
```

解决字段名与属性名不一致

方式一

- 在sql语句中为字段起别名，保持和属性名一致

```xml
<select id="getAllEmp" resultType="com.ww.mybatis.pojo.Emp">
    select eid, emp_name empName, age, sex, email from t_emp
</select>
```

方式二

- 设置全局配置，将下划线自动映射为驼峰 emp_name -> empName
- `mapUnderscoreToCamelCase` 默认是false，不开启自动映射，设为true则开启自动映射

```xml
<!-- 在mybatis-config.xml核心配置文件中添加以下代码 -->
<settings>
    <!-- 将下划线自动映射为驼峰 emp_name->empName-->
    <setting name="mapUnderscoreToCamelCase" value="true"/>
</settings>
```

```xml
<select id="getAllEmp" resultType="com.ww.mybatis.pojo.Emp">
    select * from t_emp
</select>
```

方式三

通过resultMap设置自定义映射(最常用)

```
<resultMap id="empResultMap" type="com.ww.mybatis.pojo.Emp">
    <id property="eid" column="id"></id>
    <result property="empName" column="emp_name"></result>
    <result property="age" column="age"></result>
    <result property="sex" column="sex"></result>
    <result property="email" column="email"></result>
</resultMap>

<select id="getAllEmp" resultMap="empResultMap">
    select * from t_emp
</select>
```

关键字说明

- `resultMap`：设置自定义映射
- `id`：唯一标识，不能重复
- `type`：设置映射关系中的实体类类型

子标签

- `id`：设置主键的映射关系
- `result`：设置普通字段的映射关系

属性

- `property`：设置映射关系中的属性名，必须是type属性所设置的实体类类型中的属性名
- `column`：设置映射关系中的属性名，必须是sql语句查询出的字段名

2、多对一映射处理

对一对应对象

方式一

- 级联属性赋值

1. 编写mapper接口

```
/**
 * 查询员工以及员工所对应的部门信息
 * @param eid
 * @return
 */
Emp getEmpAndDept(@Param("eid") Integer eid);
```

2. 编写mapper.xml

```xml
<resultMap id="empAndDeptResultMapOne" type="com.ww.mybatis.pojo.Emp">
    <id property="eid" column="eid"></id>
    <result property="empName" column="emp_name"></result>
    <result property="age" column="age"></result>
    <result property="sex" column="sex"></result>
    <result property="email" column="email"></result>
    <result property="dept.did" column="did"></result>
    <result property="dept.deptName" column="dept_name"></result>
</resultMap>
<select id="getEmpAndDept" resultMap="empAndDeptResultMapOne">
    select * from t_emp left join t_dept on t_emp.did = t_dept.did where t_emp.eid = #{eid}
</select>
```

3. 编写测试方法

```java
@Test
public void getEmpAndDept(){
    SqlSession sqlSession = SqlSessionUtils.getSqlSession();
    EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);
    Emp emp = mapper.getEmpAndDept(1);
    System.out.println(emp);
}
```

4. 运行结果

```
DEBUG 07-07 20:25:11,298 ==>  Preparing: select * from t_emp left join t_dept on t_emp.did = t_dept.did where t_emp.eid = ? (BaseJdbcLogger.java:137)
DEBUG 07-07 20:25:11,329 ==> Parameters: 1(Integer) (BaseJdbcLogger.java:137)
DEBUG 07-07 20:25:11,350 <==      Total: 1 (BaseJdbcLogger.java:137)
Emp{eid=1, empName='张三', age=10, sex='男', email='123@qq.com', dept=Dept{did=1, deptName='A'}}
```

方式二

- 使用association处理映射关系

```xml
<resultMap id="empAndDeptResultMapTwo" type="com.ww.mybatis.pojo.Emp">
    <id property="eid" column="eid"></id>
    <result property="empName" column="emp_name"></result>
    <result property="age" column="age"></result>
    <result property="sex" column="sex"></result>
    <result property="email" column="email"></result>
    <association property="dept" javaType="com.ww.mybatis.pojo.Dept">
        <id property="did" column="did"></id>
        <result property="deptName" column="dept_name"></result>
    </association>
</resultMap>

<select id="getEmpAndDept" resultMap="empAndDeptResultMapTwo">
    select * from t_emp left join t_dept on t_emp.did = t_dept.did where t_emp.eid = #{eid}
</select>
```

说明：

- `association`：处理多对一的映射关系
- `property`：需要处理多对一的映射关系属性名

- `javaType`：该属性的类型

方式三

- 分步查询

1. 查询员工信息

```
/**
 * 通过分步查询查询员工以及员工所对应的部门信息
 * 分步查询第一步：查询员工信息
 * @param eid
 * @return
 */
Emp getEmpAndDeptByStepOne(@Param("eid") Integer eid);
```

```
<resultMap id="empAndDeptByStepResultMap"
type="com.ww.mybatis.pojo.Emp">
    <id property="eid" column="id"></id>
    <result property="empName" column="emp_name"></result>
    <result property="age" column="age"></result>
    <result property="sex" column="sex"></result>
    <result property="email" column="email"></result>
    <association property="dept"
select="com.ww.mybatis.mapper.DeptMapper.getEmpAndDeptByStepTwo"
column="did">
    </association>
</resultMap>

<select id="getEmpAndDeptByStepOne"
resultMap="empAndDeptByStepResultMap">
    select * from t_emp where eid = #{eid}
</select>
```

2. 根据员工所对应的部门id查询部门信息

```
/**
 * 通过分步查询查询员工以及员工所对应的部门信息
 * 分步查询第二步：通过did查询员工所对应的部门
 * @param did
 * @return
 */
Dept getEmpAndDeptByStepTwo(@Param("did") Integer did);
```

```
<select id="getEmpAndDeptByStepTwo"
resultType="com.ww.mybatis.pojo.Dept">
    select * from t_dept where did = #{did}
</select>
```

3. 编写测试方法

```java
@Test
public void getEmpAndDeptByStep(){
    SqlSession sqlSession = SqlSessionUtils.getSqlSession();
    EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);
    Emp emp = mapper.getEmpAndDeptByStepOne(1);
    System.out.println(emp);
}
```

4. 运行结果，可以发现有两个sql

```
DEBUG 07-07 20:52:56,077 ==>  Preparing: select * from t_emp where eid = ? (BaseJdbcLogger.java:137)
DEBUG 07-07 20:52:56,104 ==> Parameters: 1(Integer) (BaseJdbcLogger.java:137)
DEBUG 07-07 20:52:56,125 ====>  Preparing: select * from t_dept where did = ? (BaseJdbcLogger.java:137)
DEBUG 07-07 20:52:56,126 ====> Parameters: 1(Integer) (BaseJdbcLogger.java:137)
DEBUG 07-07 20:52:56,132 <====      Total: 1 (BaseJdbcLogger.java:137)
DEBUG 07-07 20:52:56,133 <==      Total: 1 (BaseJdbcLogger.java:137)
Emp{eid=null, empName='张三', age=10, sex='男', email='123@qq.com', dept=Dept{did=1, deptName='A'}}
```

说明：

- `select`：设置分步查询的sql的唯一标识(namespace.SQLId或mapper接口的全类名.方法名)
- `column`：设置分步查询的条件
- `fetchType`：当**开启**了全局的延迟加载之后，可通过此属性手动控制延迟加载的效果，lazy(延迟加载)|eager(立即加载)

3、一对多映射处理

对多对应集合

方式一

collection

1. 编写mapper接口方法

```java
/**
 * 获取部门以及部门中所有的员工的信息
 * @param did
 * @return
 */
Dept getDeptAndEmp(@Param("did") Integer did);
```

2. 编写sql语句

```xml
<resultMap id="deptAndEmpResultMap" type="com.ww.mybatis.pojo.Dept">
    <id property="did" column="did"></id>
    <result property="deptName" column="dept_name"></result>
    <collection property="emps" ofType="com.ww.mybatis.pojo.Emp">
        <id property="eid" column="eid"></id>
        <result property="empName" column="emp_name"></result>
        <result property="age" column="age"></result>
        <result property="sex" column="sex"></result>
        <result property="email" column="email"></result>
    </collection>
</resultMap>

```

```
<select id="getDeptAndEmp" resultMap="deptAndEmpResultMap">
    select * from t_dept left join t_emp on t_dept.did = t_emp.did where t_dept.did = #{did}
</select>
```

3. 编写测试方法

```
@Test
public void getDeptAndEmp(){
    SqlSession sqlSession = SqlSessionUtils.getSqlSession();
    DeptMapper mapper = sqlSession.getMapper(DeptMapper.class);
    Dept dept = mapper.getDeptAndEmp(1);
    System.out.println(dept);
}
```

4. 运行结果

```
DEBUG 07-07 21:39:30,748 ==>  Preparing: select * from t_dept left join t_emp on t_dept.did = t_emp.did where t_dept.did = ? (BaseJdbcLogger.java:137)
DEBUG 07-07 21:39:30,773 ==> Parameters: 1(Integer) (BaseJdbcLogger.java:137)
DEBUG 07-07 21:39:30,791 <==      Total: 2 (BaseJdbcLogger.java:137)
Dept{did=1, deptName='A', emps=[Emp{eid=1, empName='张三', age=10, sex='男', email='123@qq.com', dept=null}, Emp{eid=4, empName='赵六', age=13, sex='女', email='123@qq.com', dept=n

进程已结束,退出代码0
```

说明：

- `collection`：处理一对多的映射关系
- `ofType`：表示该属性所对应的集合中存储数据的类型

方式二

- 分步查询

1. 查询部门信息

```
/**
 * 通过分步查询部门以及部门中的所有员工信息
 * 分步查询第一步：查询部门信息
 * @param id
 * @return
 */
Dept getDeptAndEmpByStepOne(@Param("did") Integer id);
```

```
<resultMap id="deptEmpStep" type="com.ww.mybatis.pojo.Dept">
    <id property="did" column="did"></id>
    <result property="deptName" column="dept_name"></result>
    <collection property="emps" select="com.ww.mybatis.mapper.EmpMapper.getEmpListByDid" column="did" fetchType="eager">
    </collection>
</resultMap>
<select id="getDeptAndEmpByStepOne" resultMap="deptEmpStep">
    select * from t_dept where did = #{did}
</select>
```

2. 根据部门id查询部门中的所有员工

```java
/**
 * 根据部门id查询员工信息
 * @param did
 * @return
 */
List<Emp> getEmpListByDid(@Param("did") int did);
```

```xml
<select id="getEmpListByDid" resultType="com.ww.mybatis.pojo.Emp">
    select * from t_emp where did = #{did}
</select>
```

3. 编写测试方法

```java
@Test
public void getDeptAndEmpByStepOne(){
    SqlSession sqlSession = SqlSessionUtils.getSqlSession();
    DeptMapper mapper = sqlSession.getMapper(DeptMapper.class);
    Dept dept = mapper.getDeptAndEmpByStepOne(1);
    System.out.println(dept);
}
```

4. 运行结果

```
7-07 21:54:25,952 ==>  Preparing: select * from t_dept where did = ? (BaseJdbcLogger.java:137)
7-07 21:54:25,981 ==> Parameters: 1(Integer) (BaseJdbcLogger.java:137)
7-07 21:54:25,997 ====>  Preparing: select * from t_emp where did = ? (BaseJdbcLogger.java:137)
7-07 21:54:25,998 ====> Parameters: 1(Integer) (BaseJdbcLogger.java:137)
7-07 21:54:26,001 <====      Total: 2 (BaseJdbcLogger.java:137)
7-07 21:54:26,001 <==      Total: 1 (BaseJdbcLogger.java:137)
d=1, deptName='A', emps=[Emp{eid=1, empName='张三', age=10, sex='男', email='123@qq.com', dept=null}, Emp{eid=4, empName='赵六', age=13, sex='女', email='123@qq.com', dept=null}]}
```

4、延迟加载

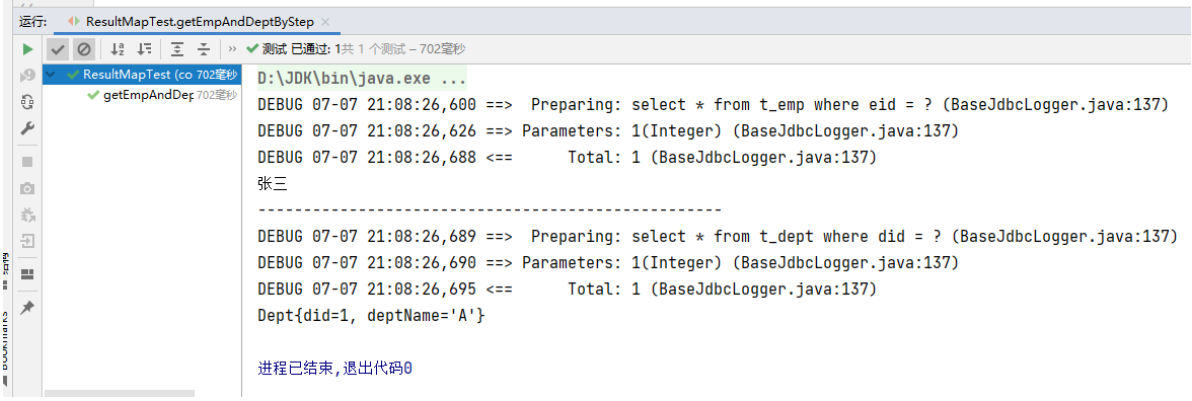
分步查询的优点：可以实现延迟加载，但是必须在核心配置文件中设置全局配置信息：

- lazyLoadingEnabled：延迟加载的全局开关。当开启时，所有关联对象都会延迟加载

```xml
<settings>
    <!-- 开启延迟加载-->
    <setting name="lazyLoadingEnabled" value="true"/>
</settings>
```

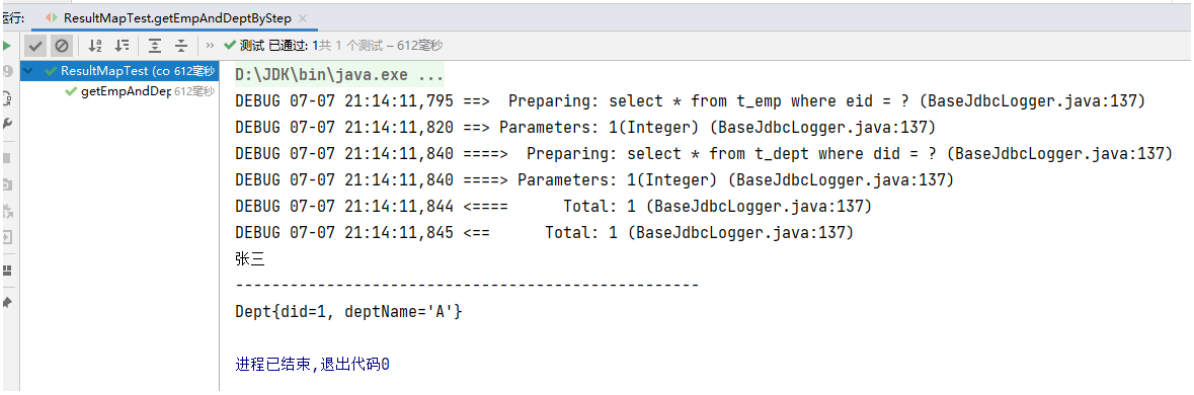
当开启延迟加载后，mybatis就会按需执行，比如

```java
@Test
public void getEmpAndDeptByStep(){
    SqlSession sqlSession = SqlSessionUtils.getSqlSession();
    EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);
    Emp emp = mapper.getEmpAndDeptByStepOne( eid: 1);
    System.out.println(emp.getEmpName());
    System.out.println("----------------------------------------------------");
    System.out.println(emp.getDept());
}
}
```



当关掉延迟加载时，mybatis会一次性执行完

```java
@Test
public void getEmpAndDeptByStep(){
    SqlSession sqlSession = SqlSessionUtils.getSqlSession();
    EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);
    Emp emp = mapper.getEmpAndDeptByStepOne( eid: 1);
    System.out.println(emp.getEmpName());
    System.out.println("----------------------------------------------------");
    System.out.println(emp.getDept());
}
}
```



如果想让某些方法不参加延迟加载，可以设置`fetchType="eager"`，使延迟加载变得可控

```xml
<resultMap id="empAndDeptByStepResultMap" type="com.ww.mybatis.pojo.Emp">
    <id property="eid" column="id"></id>
    <result property="empName" column="emp_name"></result>
    <result property="age" column="age"></result>
    <result property="sex" column="sex"></result>
    <result property="email" column="email"></result>
    <association property="dept"
select="com.ww.mybatis.mapper.DeptMapper.getEmpAndDeptByStepTwo" column="did"
fetchType="eager">
    </association>
</resultMap>
```

八、动态SQL

Mybatis框架的动态SQL技术是一种根据特定条件动态拼装SQL语句的功能，它存在的意义是为了解决 拼接SQL语句字符串时的痛点问题。

1、if

if标签可通过test属性的表达式进行判断，若表达式的结果为true，则标签中的内容会执行；反之标签中的内容不会执行

实现

1. 编写mapper接口方法

```
/**
 * 多条件查询
 * @param emp
 * @return
 */
List<Emp> getEmpByCondition(Emp emp);
```

2. 编写sql语句

```
<select id="getEmpByCondition" resultType="com.ww.mybatis.pojo.Emp">
   select * from t_emp where 1 = 1
   <if test="empName != null and empName != ''">
      and emp_name = #{empName}
   </if>
   <if test="age != null and age != ''">
      and age = #{age}
   </if>
   <if test="sex != null and sex != ''">
      and sex = #{sex}
   </if>
   <if test="email != null and email != ''">
      and email = #{email}
   </if>
</select>
```

3. 编写测试方法

```
@Test
public void getEmpByCondition(){
    SqlSession sqlSession = SqlSessionUtils.getSqlSession();
    DynamicSQLMapper mapper =
sqlSession.getMapper(DynamicSQLMapper.class);
    List<Emp> emp = mapper.getEmpByCondition(new Emp(null, null, 10,
"男", "123@qq.com"));
    System.out.println(emp);
}
```

说明

> 在sql语句的where后面加上1=1是因为防止传入的第一次参数为空导致语法错误，1=1是个恒成立条件，比如上面测试类就是给sql语句中第一个if的参数为null，却依然能得到查询结果

```
DEBUG 07-08 15:24:56,860 ==>  Preparing: select * from t_emp where 1 = 1 and age = ? and sex = ? and email = ? (BaseJdbcLogger.java:137)
DEBUG 07-08 15:24:56,884 ==> Parameters: 10(Integer), 男(String), 123@qq.com(String) (BaseJdbcLogger.java:137)
DEBUG 07-08 15:24:56,900 <==      Total: 1 (BaseJdbcLogger.java:137)
[Emp{eid=1, empName='张三', age=10, sex='男', email='123@qq.com', dept=null}]
```

2、where

where和if一般结合使用：

- 若where标签中的if条件都不满足，则where标签没有任何功能，即不会添加where关键字
- 若where标签中的if条件满足，则where标签会自动添加where关键字，并将条件最前方多余的and去掉
- **注意**：where标签不能去掉条件最后多余的and

```xml
<select id="getEmpByCondition" resultType="com.ww.mybatis.pojo.Emp">
   select * from t_emp
   <where>
      <if test="empName != null and empName != ''">
         emp_name = #{empName}
      </if>
      <if test="age != null and age != ''">
         and age = #{age}
      </if>
      <if test="sex != null and sex != ''">
         and sex = #{sex}
      </if>
      <if test="email != null and email != ''">
         and email = #{email}
      </if>
   </where>
</select>

<!-- 或者把and/or写在后面 -->

<select id="getEmpByCondition" resultType="com.ww.mybatis.pojo.Emp">
   select * from t_emp
   <where>
      <if test="empName != null and empName != ''">
         emp_name = #{empName} and
      </if>
      <if test="age != null and age != ''">
         age = #{age} and
      </if>
      <if test="sex != null and sex != ''">
         sex = #{sex} and
      </if>
      <if test="email != null and email != ''">
         email = #{email}
      </if>
   </where>
</select>
```

3、trim

trim用于去掉或添加标签中的内容

若标签中有内容时：

- prefix：在trim标签中的内容的前面添加某些内容
- prefixOverrides：在trim标签中的内容的前面去掉某些内容
- suffix：在trim标签中的内容的后面添加某些内容
- suffixOverrides：在trim标签中的内容的后面去掉某些内容

若标签中没有内容时，trim标签也没有任何效果

```
<select id="getEmpByCondition" resultType="com.ww.mybatis.pojo.Emp">
   select * from t_emp
       <trim prefix="where" suffixOverrides="and|or" >
         <if test="empName != null and empName != ''">
            emp_name = #{empName} and
         </if>
         <if test="age != null and age != ''">
            age = #{age} and
         </if>
         <if test="sex != null and sex != ''">
            sex = #{sex} and
         </if>
         <if test="email != null and email != ''">
            email = #{email}
         </if>
      </trim>
</select>
```

4、choose、when、otherwise

choose、when、otherwise相当于java中if...else if..else

- when至少要有一个，otherwise最多只能有一个

实现

1. 编写mapper接口方法

```
/**
 * 测试choose, when, otherwise
 * @param emp
 * @return
 */
List<Emp> getEmpByChoose(Emp emp);
```

2. 编写sql语句

```
<select id="getEmpByChoose" resultType="com.ww.mybatis.pojo.Emp">
   select * from t_emp
   <where>
      <choose>
         <when test="empName != null and empName != ''">
```

```
            emp_name = #{empName}
        </when>
        <when test="age != null and age != ''">
            age = #{age}
        </when>
        <when test="sex != null and sex != ''">
            sex = #{sex}
        </when>
        <when test="email != null and email != ''">
            email = #{email}
        </when>
        <otherwise>
            did = 1
        </otherwise>
    </choose>
</where>
</select>
```

3. 编写测试方法

```
@Test
public void getEmpByChoose(){
    SqlSession sqlSession = SqlSessionUtils.getSqlSession();
    DynamicSQLMapper mapper =
sqlSession.getMapper(DynamicSQLMapper.class);
    List<Emp> emps = mapper.getEmpByChoose(new Emp(null, "", null, "男",
null));
    for (Emp emp : emps) {
        System.out.println(emp);
    }
}
```

```
DEBUG 07-08 15:58:32,211 ==>  Preparing: select * from t_emp WHERE sex = ? (BaseJdbcLogger.java:137)
DEBUG 07-08 15:58:32,238 ==> Parameters: 男(String) (BaseJdbcLogger.java:137)
DEBUG 07-08 15:58:32,253 <==      Total: 3 (BaseJdbcLogger.java:137)
Emp{eid=1, empName='张三', age=10, sex='男', email='123@qq.com', dept=null}
Emp{eid=2, empName='李四', age=11, sex='男', email='123@qq.com', dept=null}
Emp{eid=3, empName='王五', age=12, sex='男', email='123@qq.com', dept=null}
```

```
@Test
public void getEmpByChoose(){
    SqlSession sqlSession = SqlSessionUtils.getSqlSession();
    DynamicSQLMapper mapper =
sqlSession.getMapper(DynamicSQLMapper.class);
    List<Emp> emps = mapper.getEmpByChoose(new Emp(null, "", null, null,
null));
    for (Emp emp : emps) {
        System.out.println(emp);
    }
}
```

```
DEBUG 07-08 16:01:35,064 ==>  Preparing: select * from t_emp WHERE did = 1 (BaseJdbcLogger.java:137)
DEBUG 07-08 16:01:35,088 ==> Parameters:  (BaseJdbcLogger.java:137)
DEBUG 07-08 16:01:35,103 <==      Total: 2 (BaseJdbcLogger.java:137)
Emp{eid=1, empName='张三', age=10, sex='男', email='123@qq.com', dept=null}
Emp{eid=4, empName='赵六', age=13, sex='女', email='123@qq.com', dept=null}
```

5、foreach

用于进行批量操作

属性

- collection：设置要循环的数组或集合
- item：表示集合或数组中的每一个数据
- separator：设置循环体之间的分隔符
- open：设置foreach标签中所有循环的所有内容的开始符
- close：设置foreach标签中所有循环的所有内容的结束符

5.1 通过数组实现批量删除

1. 编写mapper接口

```java
/**
 * 通过数组实现批量删除
 * @param eids
 * @return
 */
int deleteMoreByArray(@Param("eids") Integer[] eids);
```

2. 编写sql语句

```xml
<delete id="deleteMoreByArray">
   delete from t_emp where eid in
   (
      <foreach collection="eids" item="eid" separator=",">
          #{eid}
      </foreach>
   )
</delete>

<!-- 或 -->

<delete id="deleteMoreByArray">
    delete from t_emp where eid in
    <foreach collection="eids" item="eid" separator="," open="("
close=")">
        #{eid}
    </foreach>
</delete>

<!-- 或 -->
<!-- delete from t_emp where eid = ? or eid = ? or eid = ?  -->
<delete id="deleteMoreByArray">
    delete from t_emp where
    <foreach collection="eids" item="eid" separator="or">
        eid = #{eid}
    </foreach>
</delete>
```

3. 编写测试方法

```java
@Test
public void deleteMoreByArray(){
    SqlSession sqlSession = SqlSessionUtils.getSqlSession();
    DynamicSQLMapper mapper =
sqlSession.getMapper(DynamicSQLMapper.class);
    int rows = mapper.deleteMoreByArray(new Integer[]{6, 7, 8});
    System.out.println(rows);
}
```

4. 运行结果

```
DEBUG 07-08 16:16:30,993 ==>  Preparing: delete from t_emp where eid in ( ? , ? , ? ) (BaseJdbcLogger.java:137)
DEBUG 07-08 16:16:31,018 ==> Parameters: 6(Integer), 7(Integer), 8(Integer) (BaseJdbcLogger.java:137)
DEBUG 07-08 16:16:31,022 <==    Updates: 3 (BaseJdbcLogger.java:137)
3
```

5.2 通过list集合实现批量添加

1. 编写mapper接口

```java
/**
 * 通过list集合实现批量添加
 * @param emps
 * @return
 */
int insertMoreByList(@Param("emps") List<Emp> emps);
```

2. 编写sql语句

```xml
<insert id="insertMoreByList">
   insert into t_emp values
   <foreach collection="emps" item="emp" separator=",">
      (null, #{emp.empName}, #{emp.age}, #{emp.sex}, #{emp.email}, null)
   </foreach>
</insert>
```

3. 编写测试方法

```java
@Test
public void insertMoreByList(){
    SqlSession sqlSession = SqlSessionUtils.getSqlSession();
    DynamicSQLMapper mapper =
sqlSession.getMapper(DynamicSQLMapper.class);
    Emp emp1 = new Emp(null, "a1", 12, "男","123@qq.com");
    Emp emp2 = new Emp(null, "a2", 13, "男","123@qq.com");
    Emp emp3 = new Emp(null, "a3", 14, "男","123@qq.com");
    List<Emp> list = Arrays.asList(emp1, emp2, emp3);
    int rows = mapper.insertMoreByList(list);
    System.out.println(rows);
}
```

4. 运行结果

```
DEBUG 07-08 16:39:06,558 ==>  Preparing: insert into t_emp values (null, ?, ?, ?, ?, null) , (null, ?, ?, ?, ?, null) , (null, ?, ?, ?, ?, null) (BaseJdbcLogger.java:137)
DEBUG 07-08 16:39:06,583 ==> Parameters: a1(String), 12(Integer), 男(String), 123@qq.com(String), a2(String), 13(Integer), 男(String), 123@qq.com(String), a3(String), 14(Integer),
DEBUG 07-08 16:39:06,586 <==    Updates: 3 (BaseJdbcLogger.java:137)
3
```

6、SQL片段

sql片段，可以记录一段公共sql片段，在使用的地方通过include标签进行引入

```
<sql id="empColumns">
    eid,emp_name,age,sex,email,did
</sql>

select <include refid="empColumns"></include> from t_emp
```

九、MyBatis的缓存

1、MyBatis的一级缓存

一级缓存是SqlSession级别的，是默认开启的，通过同一个SqlSession查询的数据会被缓存，下次查询相同的数据，就会从缓存中直接获取，不会从数据库重新访问。

使一级缓存失效的四种情况：

1. 不同的SqlSession对应不同的一级缓存
2. 同一个SqlSession但是查询条件不同
3. 同一个SqlSession两次查询期间执行了任何一次增删改操作
4. 同一个SqlSession两次查询期间手动清空了缓存

```
/**
* 通过id查找员工信息
*/
@Test
public void getEmpByEid(){
    SqlSession sqlSession = SqlSessionUtils.getSqlSession();
    CacheMapper mapper = sqlSession.getMapper(CacheMapper.class);
    System.out.println("----------------------------第一次执行----------
----------");
    Emp emp1 = mapper.getEmpByEid(1);
    System.out.println(emp1);
    System.out.println("----------------------------第二次执行----------
----------");
    Emp emp2 = mapper.getEmpByEid(1);
    System.out.println(emp2);
}
```

```
----------------------------第一次执行----------------------------
DEBUG 07-08 17:33:22,546 ==>  Preparing: select * from t_emp where eid = ? (BaseJdbcLogger.java:137)
DEBUG 07-08 17:33:22,573 ==> Parameters: 1(Integer) (BaseJdbcLogger.java:137)
DEBUG 07-08 17:33:22,590 <==      Total: 1 (BaseJdbcLogger.java:137)
Emp{eid=1, empName='张三', age=10, sex='男', email='123@qq.com', dept=null}
----------------------------第二次执行----------------------------
Emp{eid=1, empName='张三', age=10, sex='男', email='123@qq.com', dept=null}
```

从上面运行结果可以看到通过同一个SqlSession查询相同的数据时，sql语句只会执行一次。

```java
@Test
public void getEmpByEid(){
    SqlSession sqlSession1 = SqlSessionUtils.getSqlSession();
    CacheMapper mapper1 = sqlSession1.getMapper(CacheMapper.class);
    System.out.println("----------------------------第一次执行----------
----------");
    Emp emp1 = mapper1.getEmpByEid(1);
    System.out.println(emp1);
    System.out.println("----------------------------第二次执行----------
----------");
    SqlSession sqlSession2 = SqlSessionUtils.getSqlSession();
    CacheMapper mapper2 = sqlSession2.getMapper(CacheMapper.class);
    Emp emp2 = mapper2.getEmpByEid(1);
    System.out.println(emp2);
}
```

```
----------------------------第一次执行----------------------------
DEBUG 07-08 17:38:21,193 ==>  Preparing: select * from t_emp where eid = ? (BaseJdbcLogger.java:137)
DEBUG 07-08 17:38:21,218 ==> Parameters: 1(Integer) (BaseJdbcLogger.java:137)
DEBUG 07-08 17:38:21,233 <==      Total: 1 (BaseJdbcLogger.java:137)
Emp{eid=1, empName='张三', age=10, sex='男', email='123@qq.com', dept=null}
----------------------------第二次执行----------------------------
DEBUG 07-08 17:38:21,295 ==>  Preparing: select * from t_emp where eid = ? (BaseJdbcLogger.java:137)
DEBUG 07-08 17:38:21,296 ==> Parameters: 1(Integer) (BaseJdbcLogger.java:137)
DEBUG 07-08 17:38:21,297 <==      Total: 1 (BaseJdbcLogger.java:137)
Emp{eid=1, empName='张三', age=10, sex='男', email='123@qq.com', dept=null}
```

从上面运行结果可以看到通过不同的SqlSession对象执行相同的操作，sql语句也会执行多次。

2、MyBatis的二级缓存

二级缓存是SqlSessionFactory级别，通过同一个SqlSessionFactory创建的SqlSession查询的结果会被缓存；此后若再次执行相同的查询语句，结果就会从缓存中获取。

二级缓存开启的条件：

1. 在核心配置文件中，设置全局配置属性cacheEnabled="true"，默认为true，不需要设置
2. 在映射文件中设置标签`<cache />`
3. 二级缓存必须在SqlSession关闭或提交之后有效
4. 查询的数据所转换的实体类类型必须实现序列化的接口

使二级缓存失效的情况：

- 两次查询之间执行了任意的增删改，会使一级和二级缓存同时失效

```java
@Test
public void TwoCache(){
    try {
        InputStream is = Resources.getResourceAsStream("mybatis-
config.xml");
        SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new
SqlSessionFactoryBuilder();
        SqlSessionFactory sqlSessionFactory =
sqlSessionFactoryBuilder.build(is);
        SqlSession sqlSession1 = sqlSessionFactory.openSession(true);
        CacheMapper mapper1 = sqlSession1.getMapper(CacheMapper.class);
```

```
        System.out.println("--------------------------第一次执行-------------
--------------");
        System.out.println(mapper1.getEmpByEid(1));
        //sqlSession1.close();
        SqlSession sqlSession2 = sqlSessionFactory.openSession(true);
        CacheMapper mapper2 = sqlSession2.getMapper(CacheMapper.class);
        System.out.println("--------------------------第二次执行-------------
--------------");
        System.out.println(mapper2.getEmpByEid(1));
        //sqlSession2.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

```
--------------------------第一次执行--------------------------
DEBUG 07-08 19:09:37,722 Cache Hit Ratio [com.ww.mybatis.mapper.CacheMapper]: 0.0 (LoggingCache.java:60)
DEBUG 07-08 19:09:37,889 ==>  Preparing: select * from t_emp where eid = ? (BaseJdbcLogger.java:137)
DEBUG 07-08 19:09:37,915 ==> Parameters: 1(Integer) (BaseJdbcLogger.java:137)
DEBUG 07-08 19:09:37,932 <==      Total: 1 (BaseJdbcLogger.java:137)
Emp{eid=1, empName='张三', age=10, sex='男', email='123@qq.com', dept=null}
--------------------------第二次执行--------------------------
DEBUG 07-08 19:09:37,933 Cache Hit Ratio [com.ww.mybatis.mapper.CacheMapper]: 0.0 (LoggingCache.java:60)
DEBUG 07-08 19:09:37,942 ==>  Preparing: select * from t_emp where eid = ? (BaseJdbcLogger.java:137)
DEBUG 07-08 19:09:37,942 ==> Parameters: 1(Integer) (BaseJdbcLogger.java:137)
DEBUG 07-08 19:09:37,944 <==      Total: 1 (BaseJdbcLogger.java:137)
Emp{eid=1, empName='张三', age=10, sex='男', email='123@qq.com', dept=null}
```

SqlSession没有关闭或提交，可以看到sql语句执行了两次

```
@Test
public void TwoCache(){
    try {
        InputStream is = Resources.getResourceAsStream("mybatis-
config.xml");
        SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new
SqlSessionFactoryBuilder();
        SqlSessionFactory sqlSessionFactory =
sqlSessionFactoryBuilder.build(is);
        SqlSession sqlSession1 = sqlSessionFactory.openSession(true);
        CacheMapper mapper1 = sqlSession1.getMapper(CacheMapper.class);
        System.out.println("--------------------------第一次执行-------------
--------------");
        System.out.println(mapper1.getEmpByEid(1));
        sqlSession1.close();
        SqlSession sqlSession2 = sqlSessionFactory.openSession(true);
        CacheMapper mapper2 = sqlSession2.getMapper(CacheMapper.class);
        System.out.println("--------------------------第二次执行-------------
--------------");
        System.out.println(mapper2.getEmpByEid(1));
        sqlSession2.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

```
-------------------------第一次执行-------------------------
DEBUG 07-08 19:11:13,197 Cache Hit Ratio [com.ww.mybatis.mapper.CacheMapper]: 0.0 (LoggingCache.java:60)
DEBUG 07-08 19:11:13,360 ==>  Preparing: select * from t_emp where eid = ? (BaseJdbcLogger.java:137)
DEBUG 07-08 19:11:13,383 ==> Parameters: 1(Integer) (BaseJdbcLogger.java:137)
DEBUG 07-08 19:11:13,400 <==      Total: 1 (BaseJdbcLogger.java:137)
Emp{eid=1, empName='张三', age=10, sex='男', email='123@qq.com', dept=null}
-------------------------第二次执行-------------------------
WARN  07-08 19:11:13,410 As you are using functionality that deserializes object streams, it is recommend
DEBUG 07-08 19:11:13,413 Cache Hit Ratio [com.ww.mybatis.mapper.CacheMapper]: 0.5 (LoggingCache.java:60)
Emp{eid=1, empName='张三', age=10, sex='男', email='123@qq.com', dept=null}
```

当SqlSession关闭或提交后，sql语句只有一条

3、二级缓存的相关配置

在mapper配置文件中添加的cache标签可以设置一些属性：

- eviction属性：缓存回收策略
  - LRU（Least Recently Used） – 最近最少使用的：移除最长时间不被使用的对象。
  - FIFO（First in First out） – 先进先出：按对象进入缓存的顺序来移除它们。
  - SOFT – 软引用：移除基于垃圾回收器状态和软引用规则的对象。
  - WEAK – 弱引用：更积极地移除基于垃圾收集器状态和弱引用规则的对象。
  - 默认的是 LRU。
- flushInterval属性：刷新间隔，单位毫秒
  - 默认情况是不设置，也就是没有刷新间隔，缓存仅仅调用语句时刷新
- size属性：引用数目，正整数
  - 代表缓存最多可以存储多少个对象，太大容易导致内存溢出
- readOnly属性：只读，true/false
  - true：只读缓存；会给所有调用者返回缓存对象的相同实例。因此这些对象不能被修改。这提供了 很重要的性能优势。
  - false：读写缓存；会返回缓存对象的拷贝（通过序列化）。这会慢一些，但是安全，因此默认是 false。

4、MyBatis缓存查询的顺序

- 先查询二级缓存，因为二级缓存中可能会有其他程序已经查出来的数据，可以拿来直接使用
- 如果二级缓存没有命中，再查询一级缓存
- 如果一级缓存也没有命中，则查询数据库
- SqlSession关闭之后，一级缓存中的数据会写入二级缓存

二级缓存——>一级缓存——>数据库

当SqlSession没有关闭前，数据默认写入一级缓存，关闭后，一级缓存中的数据会写入到二级缓存中

5、整合第三方缓存EHCache

只能代替二级缓存，无法代替一级缓存

1. 引入依赖

```xml
<!-- Mybatis EHCache整合包 -->
<dependency>
    <groupId>org.mybatis.caches</groupId>
    <artifactId>mybatis-ehcache</artifactId>
    <version>1.2.1</version>
</dependency>
<!-- slf4j日志门面的一个具体实现 -->
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>1.2.3</version>
</dependency>
```

2. 各jar包功能

名称	作用
mybatis-ehcache	Mybatis和EHCache的整合包
ehcache	EHCache核心包
slf4j-api	SLF4J日志门面包
logback-classic	支持SLF4J门面接口的一个具体实现

3. 创建EHCache的配置文件ehcache.xml

```xml
<?xml version="1.0" encoding="utf-8" ?>
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="../config/ehcache.xsd">
    <!-- 磁盘保存路径 -->
    <diskStore path="D:\mybatisCache\ehcache"/>

    <defaultCache
            maxElementsInMemory="1000"
            maxElementsOnDisk="10000000"
            eternal="false"
            overflowToDisk="true"
            timeToIdleSeconds="120"
            timeToLiveSeconds="120"
            diskExpiryThreadIntervalSeconds="120"
            memoryStoreEvictionPolicy="LRU">
    </defaultCache>
</ehcache>
```

4. 设置二级缓存的类型

```xml
<cache type="org.mybatis.caches.ehcache.EhcacheCache"/>
```

5. 加入logback日志

存在SLF4J时，作为简易日志的log4j将失效，此时我们需要借助SLF4J的具体实现logback来打印日志。 创建logback的配置文件logback.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<configuration debug="true">
```

```
    <!-- 指定日志输出的位置 -->
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <!-- 日志输出的格式 -->
            <!-- 按照顺序分别是：时间、日志级别、线程名称、打印日志的类、日志主体内容、换行 -->
            <pattern>
                [%d{HH:mm:ss.SSS}] [%-5level] [%thread] [%logger] [%msg]%n
            </pattern>
        </encoder>
    </appender>

    <!-- 设置全局日志级别。日志级别按顺序分别是：DEBUG、INFO、WARN、ERROR -->
    <!-- 指定任何一个日志级别都只打印当前级别和后面级别的日志。-->
    <root level="DEBUG">
        <!-- 指定打印日志的appender，这里通过“STDOUT”引用了前面配置的appender -->
        <appender-ref ref="STDOUT" />
    </root>

    <!-- 根据特殊需求指定局部日志级别 -->
    <logger name="com.ww.mybatis.mapper" level="DEBUG"/>

</configuration>
```

6. 运行之前二级缓存代码

```
-------------------------第一次执行-------------------------
[19:48:59.861] [DEBUG] [main] [com.ww.mybatis.mapper.CacheMapper] [Cache Hit Ratio [com.ww.mybatis.mapper.CacheMapper]: 0.0]
[19:48:59.865] [DEBUG] [main] [org.apache.ibatis.transaction.jdbc.JdbcTransaction] [Opening JDBC Connection]
[19:49:00.039] [DEBUG] [main] [org.apache.ibatis.datasource.pooled.PooledDataSource] [Created connection 1942828992.]
[19:49:00.043] [DEBUG] [main] [com.ww.mybatis.mapper.CacheMapper.getEmpByEid] [==>  Preparing: select * from t_emp where eid = ?]
[19:49:00.068] [DEBUG] [main] [com.ww.mybatis.mapper.CacheMapper.getEmpByEid] [==> Parameters: 1(Integer)]
[19:49:00.085] [DEBUG] [main] [com.ww.mybatis.mapper.CacheMapper.getEmpByEid] [<==      Total: 1]
Emp{eid=1, empName='张三', age=10, sex='男', email='123@qq.com', dept=null}
[19:49:00.091] [DEBUG] [main] [org.apache.ibatis.transaction.jdbc.JdbcTransaction] [Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@73cd37c0]]
[19:49:00.091] [DEBUG] [main] [org.apache.ibatis.datasource.pooled.PooledDataSource] [Returned connection 1942828992 to pool.]
-------------------------第二次执行-------------------------
[19:49:00.093] [WARN ] [main] [org.apache.ibatis.io.SerialFilterChecker] [As you are using functionality that deserializes object streams, it is recommended to define the JEP-290
[19:49:00.100] [DEBUG] [main] [com.ww.mybatis.mapper.CacheMapper] [Cache Hit Ratio [com.ww.mybatis.mapper.CacheMapper]: 0.5]
Emp{eid=1, empName='张三', age=10, sex='男', email='123@qq.com', dept=null}
```

7. EHCache配置文件说明

属性名	是否必须	作用
maxElementsInMemory	是	在内存中缓存的element的最大数目
maxElementsOnDisk	是	在磁盘上缓存的element的最大数目，若是0表示无穷大
eternal	是	设定缓存的elements是否永远不过期。 如果为 true，则缓存的数据始终有效， 如果为false那么还 要根据timeToIdleSeconds、timeToLiveSeconds 判断
overflowToDisk	是	设定当内存缓存溢出的时候是否将过期的element 缓存到磁盘上
timeToIdleSeconds	否	当缓存在EhCache中的数据前后两次访问的时间超 过timeToIdleSeconds的属性取值时， 这些数据便 会删除，默认值是0,也就是可闲置时间无穷大
timeToLiveSeconds	否	缓存element的有效生命期，默认是0,也就是element存活时间无穷大
diskSpoolBufferSizeMB	否	DiskStore(磁盘缓存)的缓存区大小。默认是30MB。每个Cache都应该有自己的一个缓冲区
diskPersistent	否	在VM重启的时候是否启用磁盘保存EhCache中的数 据，默认是false
diskExpiryThreadIntervalSeconds	否	磁盘缓存的清理线程运行间隔，默认是120秒。每 个120s， 相应的线程会进行一次EhCache中数据的 清理工作
memoryStoreEvictionPolicy	否	当内存缓存达到最大，有新的element加入的时 候， 移除缓存中element的策略。 默认是LRU（最 近最少使用），可选的有LFU（最不常使用）和 FIFO（先进先出）

十、MyBatis的逆向工程

- 正向工程：先创建Java实体类，由框架负责根据实体类生成数据库表。Hibernate是支持正向工程的。
- 逆向工程：先创建数据库表，由框架负责根据数据库表，反向生成如下资源：
  - Java实体类
  - Mapper接口
  - Mapper映射文件

1、清晰简洁版

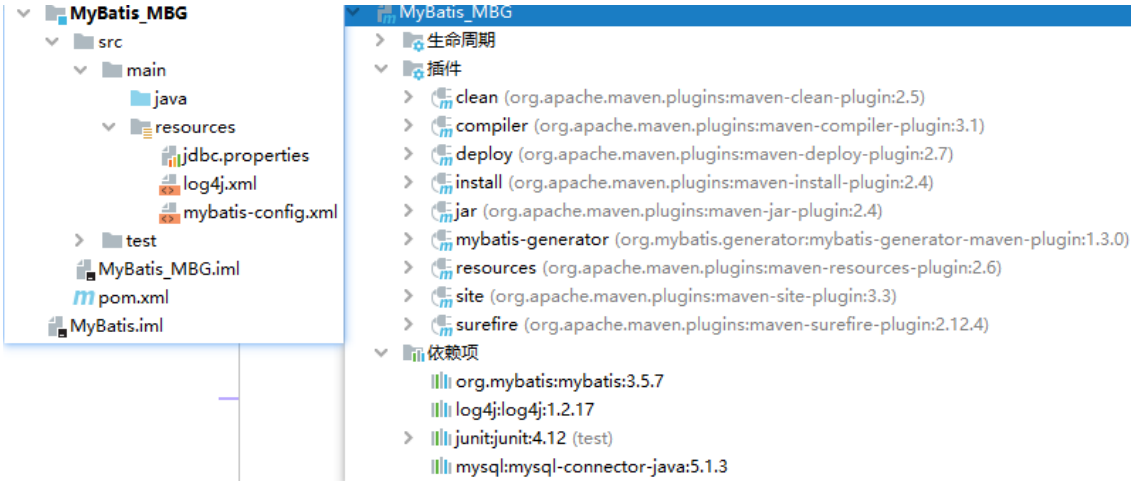
创建清晰简洁版逆向工程

1. 添加依赖和插件

```
<!-- 依赖MyBatis核心包 -->
<dependencies>
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.5.7</version>
    </dependency>
    <!-- log4j日志 -->
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.17</version>
    </dependency>
    <!-- junit测试 -->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>
    <!-- MySQL驱动 -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.3</version>
    </dependency>
</dependencies>

<!-- 控制Maven在构建过程中相关配置 -->
<build>
    <!-- 构建过程中用到的插件 -->
    <plugins>
        <!-- 具体插件，逆向工程的操作是以构建过程中插件形式出现的 -->
        <plugin>
            <groupId>org.mybatis.generator</groupId>
            <artifactId>mybatis-generator-maven-plugin</artifactId>
            <version>1.3.0</version>
            <!-- 插件的依赖 -->
            <dependencies>
                <!-- 逆向工程的核心依赖 -->
                <dependency>
                    <groupId>org.mybatis.generator</groupId>
                    <artifactId>mybatis-generator-core</artifactId>
                    <version>1.3.2</version>
                </dependency>
                <!-- 数据库连接池 -->
                <dependency>
                    <groupId>com.mchange</groupId>
                    <artifactId>c3p0</artifactId>
                    <version>0.9.2</version>
                </dependency>
```

```
                <!-- MySQL驱动 -->
                <dependency>
                    <groupId>mysql</groupId>
                    <artifactId>mysql-connector-java</artifactId>
                    <version>5.1.8</version>
                </dependency>
            </dependencies>
        </plugin>
    </plugins>
</build>
```

2. 创建MyBatis的核心配置文件



3. 创建逆向工程的配置文件

文件名必须是：generatorConfig.xml

```
<?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>
    <!--
        targetRuntime: 执行生成的逆向工程的版本
            MyBatis3Simple: 生成基本的CRUD（清新简洁版）
            MyBatis3: 生成带条件的CRUD（奢华尊享版）
    -->
    <context id="DB2Tables" targetRuntime="MyBatis3Simple">
        <!-- 数据库的连接信息 -->
        <jdbcConnection driverClass="com.mysql.jdbc.Driver"

 connectionURL="jdbc:mysql://localhost:3306/mybatis"
                        userId="root"
                        password="root">
        </jdbcConnection>
        <!-- javaBean的生成策略-->
        <javaModelGenerator targetPackage="com.ww.mybatis.bean"
targetProject=".\src\main\java">
            <property name="enableSubPackages" value="true" />
            <property name="trimStrings" value="true" />
        </javaModelGenerator>
        <!-- SQL映射文件的生成策略 -->
        <sqlMapGenerator targetPackage="com.ww.mybatis.mapper"
targetProject=".\src\main\resources">
```

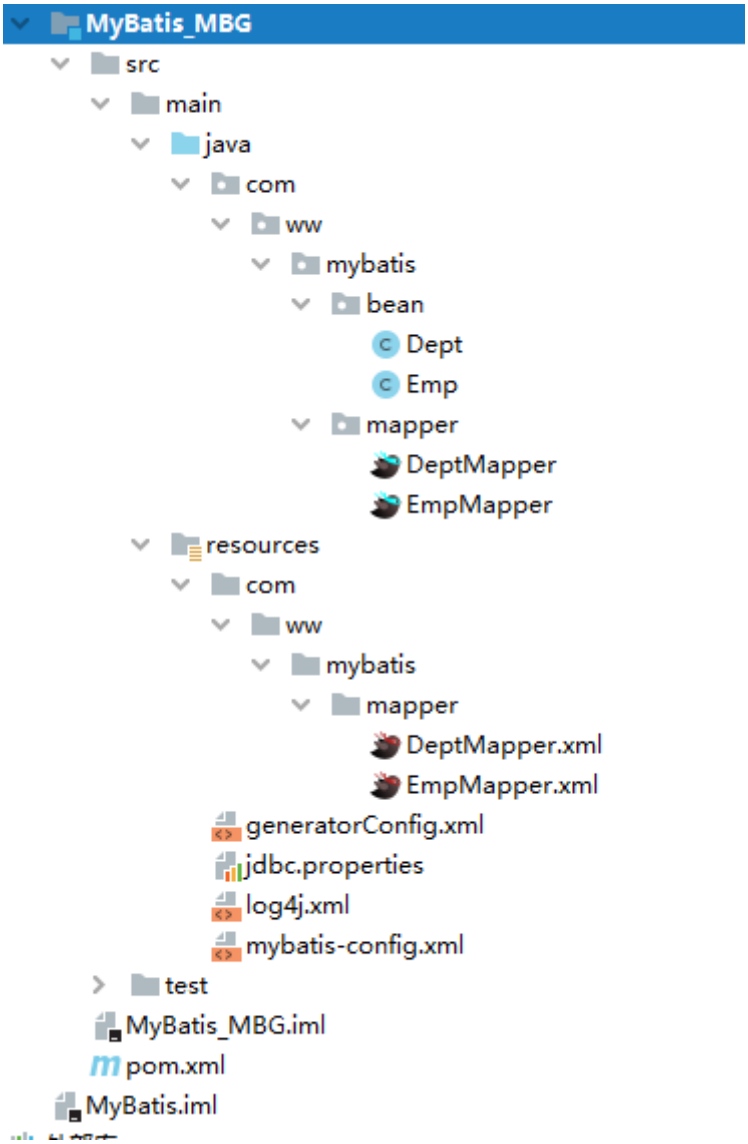
```
            <property name="enableSubPackages" value="true" />
        </sqlMapGenerator>
        <!-- Mapper接口的生成策略 -->
        <javaClientGenerator type="XMLMAPPER"
targetPackage="com.ww.mybatis.mapper" targetProject=".\src\main\java">
            <property name="enableSubPackages" value="true" />
        </javaClientGenerator>
        <!-- 逆向分析的表 -->
        <!-- tableName设置为*号，可以对应所有表，此时不写domainObjectName -->
        <!-- domainObjectName属性指定生成出来的实体类的类名 -->
        <table tableName="t_emp" domainObjectName="Emp"/>
        <table tableName="t_dept" domainObjectName="Dept"/>
    </context>
</generatorConfiguration>
```

4. 执行MBG插件的generate目标

   双击执行该插件


```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------< com.ww:MyBatis_MBG >------------------------
[INFO] Building MyBatis_MBG 1.0-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- mybatis-generator-maven-plugin:1.3.0:generate (default-cli) @ MyBatis_MBG ---
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  0.700 s
[INFO] Finished at: 2022-07-08T20:13:46+08:00
[INFO] ------------------------------------------------------------------------

进程已结束,退出代码0
```

5. 生成结果


MyBatis_MBG
src
main
java
com
ww
mybatis
bean
Dept
Emp
mapper
DeptMapper
EmpMapper
resources
com
ww
mybatis
mapper
DeptMapper.xml
EmpMapper.xml
generatorConfig.xml
jdbc.properties
log4j.xml
mybatis-config.xml
test
MyBatis_MBG.iml
pom.xml
MyBatis.iml

2、奢华尊享版

1. 修改generatorConfig.xml

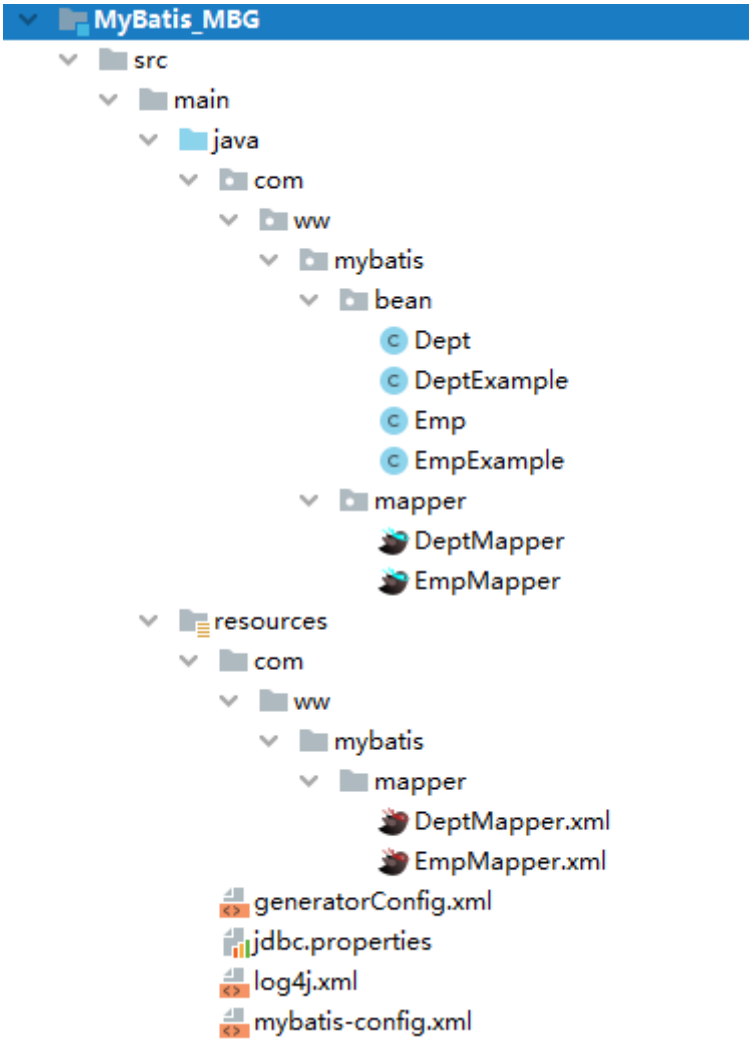
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE generatorConfiguration PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"
        "http://mybatis.org/dtd/mybatis-generator-config_1_0.dtd">
<generatorConfiguration>
    <!--
        targetRuntime: 执行生成的逆向工程的版本
            MyBatis3Simple: 生成基本的CRUD（清新简洁版）
            MyBatis3: 生成带条件的CRUD（奢华尊享版）
    -->
    <context id="DB2Tables" targetRuntime="MyBatis3">
        <!-- 数据库的连接信息 -->
        <jdbcConnection driverClass="com.mysql.jdbc.Driver"

 connectionURL="jdbc:mysql://localhost:3306/mybatis"
                        userId="root"
                        password="root">
        </jdbcConnection>
        <!-- javaBean的生成策略-->
        <javaModelGenerator targetPackage="com.ww.mybatis.bean" targetProject=".\src\main\java">
            <property name="enableSubPackages" value="true" />
```

```
            <property name="trimStrings" value="true" />
        </javaModelGenerator>
        <!-- SQL映射文件的生成策略 -->
        <sqlMapGenerator targetPackage="com.ww.mybatis.mapper"
targetProject=".\src\main\resources">
            <property name="enableSubPackages" value="true" />
        </sqlMapGenerator>
        <!-- Mapper接口的生成策略 -->
        <javaClientGenerator type="XMLMAPPER"
targetPackage="com.ww.mybatis.mapper" targetProject=".\src\main\java">
            <property name="enableSubPackages" value="true" />
        </javaClientGenerator>
        <!-- 逆向分析的表 -->
        <!-- tableName设置为*号，可以对应所有表，此时不写domainObjectName -->
        <!-- domainObjectName属性指定生成出来的实体类的类名 -->
        <table tableName="t_emp" domainObjectName="Emp"/>
        <table tableName="t_dept" domainObjectName="Dept"/>
    </context>
</generatorConfiguration>
```

2. 执行MBG插件的generate目标

```
[INFO] Scanning for projects...
[INFO]
[INFO] -------------------------< com.ww:MyBatis_MBG >-------------------------
[INFO] Building MyBatis_MBG 1.0-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- mybatis-generator-maven-plugin:1.3.0:generate (default-cli) @ MyBatis_MBG ---
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  0.743 s
[INFO] Finished at: 2022-07-08T20:21:07+08:00
[INFO] ------------------------------------------------------------------------
```

3. 结果，可发现实体类变多了，mapper接口方法也变多了


MyBatis_MBG
src
main
java
com
ww
mybatis
bean
Dept
DeptExample
Emp
EmpExample
mapper
DeptMapper
EmpMapper
resources
com
ww
mybatis
mapper
DeptMapper.xml
EmpMapper.xml
generatorConfig.xml
jdbc.properties
log4j.xml
mybatis-config.xml

```
     *
     * @mbggenerated Fri Jul 08 20:21:07 CST 2022
     */
    Emp selectByPrimaryKey(Integer eid);

    /**
     * This method was generated by MyBatis Generator.
     * This method corresponds to the database table t_emp
     *
     * @mbggenerated Fri Jul 08 20:21:07 CST 2022
     */
    int updateByExampleSelective(@Param("record") Emp record, @Param("example") EmpExample example);

    /**
     * This method was generated by MyBatis Generator.
     * This method corresponds to the database table t_emp
     *
     * @mbggenerated Fri Jul 08 20:21:07 CST 2022
     */
    int updateByExample(@Param("record") Emp record, @Param("example") EmpExample example);

    /**
     * This method was generated by MyBatis Generator.
     * This method corresponds to the database table t_emp
     *
     * @mbggenerated Fri Jul 08 20:21:07 CST 2022
     */
    int updateByPrimaryKeySelective(Emp record);

    /**
     * This method was generated by MyBatis Generator.
     * This method corresponds to the database table t_emp
     *
     * @mbggenerated Fri Jul 08 20:21:07 CST 2022
     */
```

4. 测试

```
@Test
public void testMBG(){
    try {
        InputStream is = Resources.getResourceAsStream("mybatis-config.xml");
        SqlSession sqlSession = new SqlSessionFactoryBuilder().build(is).openSession(true);
        EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);
        // 查询所有数据
        /*List<Emp> emps = mapper.selectByExample(null);
        emps.forEach(emp -> System.out.println(emp));*/
        // 根据条件查询
        EmpExample example = new EmpExample();
        example.createCriteria().andEmpNameEqualTo("张三");
        List<Emp> emps = mapper.selectByExample(example);
        emps.forEach(emp -> System.out.println(emp));
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

十一、分页插件

1、分页插件使用步骤

1. 引入依赖

```
<!-- https://mvnrepository.com/artifact/com.github.pagehelper/pagehelper
-->
<dependency>
    <groupId>com.github.pagehelper</groupId>
    <artifactId>pagehelper</artifactId>
    <version>5.2.0</version>
</dependency>
```

2. 配置分页插件

   在MyBatis的核心配置文件中配置插件

```
<plugins>
    <!--设置分页插件-->
    <plugin interceptor="com.github.pagehelper.PageInterceptor"></plugin>
</plugins>
```

2、分页插件的使用

1. 在查询功能之前使用`PageHelper.startPage(int pageNum, int pageSize)`开启分页功能
   - pageNum：当前页的页码
   - pageSize：每页显示的条数
2. 在查询获取list集合之后，使用`PageInfo pageInfo = new PageInfo<>(List list, int navigatePages)`获取分页相关数据
   - list：分页之后的数据
   - navigatePages：导航分页的页码数

常用数据

- pageNum：当前页的页码
- pageSize：每页显示的条数
- size：当前页显示的真实条数
- total：总记录数
- pages：总页数
- prePage：上一页的页码
- nextPage：下一页的页码
- isFirstPage/isLastPage：是否为第一页/最后一页
- hasPreviousPage/hasNextPage：是否存在上一页/下一页
- navigatePages：导航分页的页码数
- navigatepageNums：导航分页的页码，[1,2,3,4,5]

```java
@Test
public void testPageHelper(){
    try {
        InputStream is = Resources.getResourceAsStream("mybatis-config.xml");
        SqlSession sqlSession = new SqlSessionFactoryBuilder().build(is).openSession(true);
        EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);
        PageHelper.startPage(1,4);
        List<Emp> list = mapper.selectByExample(null);
        list.forEach(emp -> System.out.println(emp));
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

```
DEBUG 07-08 20:50:33,170 Cache Hit Ratio [SQL_CACHE]: 0.0 (LoggingCache.java:60)
DEBUG 07-08 20:50:33,216 ==>  Preparing: SELECT count(0) FROM t_emp (BaseJdbcLogger.java:137)
DEBUG 07-08 20:50:33,238 ==> Parameters:  (BaseJdbcLogger.java:137)
DEBUG 07-08 20:50:33,253 <==      Total: 1 (BaseJdbcLogger.java:137)
DEBUG 07-08 20:50:33,256 ==>  Preparing: select eid, emp_name, age, sex, email, did from t_emp LIMIT ? (BaseJdbcLogger.java:137)
DEBUG 07-08 20:50:33,257 ==> Parameters: 4(Integer) (BaseJdbcLogger.java:137)
DEBUG 07-08 20:50:33,259 <==      Total: 4 (BaseJdbcLogger.java:137)
Emp{eid=1, empName='张三', age=10, sex='男', email='123@qq.com', did=1}
Emp{eid=2, empName='李四', age=11, sex='男', email='123@qq.com', did=2}
Emp{eid=3, empName='王五', age=12, sex='男', email='123@qq.com', did=3}
Emp{eid=4, empName='赵六', age=13, sex='女', email='123@qq.com', did=1}
```

```java
@Test
public void testPageHelper(){
    try {
        InputStream is = Resources.getResourceAsStream("mybatis-config.xml");
        SqlSession sqlSession = new SqlSessionFactoryBuilder().build(is).openSession(true);
        EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);
        //PageHelper.startPage(1,4);
        PageHelper.startPage(1,2);
        List<Emp> list = mapper.selectByExample(null);
        PageInfo<Emp> page = new PageInfo<>(list, 3);
        System.out.println(page);
        //list.forEach(emp -> System.out.println(emp));
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

```
DEBUG 07-08 20:59:10,375 Cache Hit Ratio [SQL_CACHE]: 0.0 (LoggingCache.java:60)
DEBUG 07-08 20:59:10,419 ==>  Preparing: SELECT count(0) FROM t_emp (BaseJdbcLogger.java:137)
DEBUG 07-08 20:59:10,446 ==> Parameters:  (BaseJdbcLogger.java:137)
DEBUG 07-08 20:59:10,462 <==      Total: 1 (BaseJdbcLogger.java:137)
DEBUG 07-08 20:59:10,464 ==>  Preparing: select eid, emp_name, age, sex, email, did from t_emp LIMIT ? (BaseJdbcLogger.java:137)
DEBUG 07-08 20:59:10,465 ==> Parameters: 2(Integer) (BaseJdbcLogger.java:137)
DEBUG 07-08 20:59:10,467 <==      Total: 2 (BaseJdbcLogger.java:137)
PageInfo{pageNum=1, pageSize=2, size=2, startRow=1, endRow=2, total=11, pages=6, list=Page{count=true, pageNum=1, pageSize=2, startRow=0, endRow=2, total=11, pages=6, reasonable=f
```

十二、Spring整合MyBatis

1. 引入依赖

```xml
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.2.10.RELEASE</version>
    </dependency>
    <!-- 连接池-->
    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>druid</artifactId>
        <version>1.1.16</version>
    </dependency>
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.5.6</version>
    </dependency>
    <!-- mysql驱动 -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.47</version>
    </dependency>
    <!-- Spring操作数据库需要该jar包-->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-jdbc</artifactId>
        <version>5.2.10.RELEASE</version>
    </dependency>
    <!-- Spring与Mybatis整合的jar包，这个jar包mybatis在前面，是Mybatis提供的
-->
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis-spring</artifactId>
        <version>1.3.0</version>
    </dependency>
</dependencies>
```

2. 创建Spring的主配置类

```java
@Configuration
@ComponentScan("com.ww")
public class SpringConfig {
}
```

3. 创建数据源的配置类

   在配置类中完成数据源的创建

```java
public class JdbcConfig {
    @Value("${jdbc.driver}")
    private String driver;
    @Value("${jdbc.url}")
    private String url;
    @Value("${jdbc.username}")
    private String userName;
    @Value("${jdbc.password}")
```

```
    private String password;

    @Bean
    public DataSource dataSource(){
        DruidDataSource ds = new DruidDataSource();
        ds.setDriverClassName(driver);
        ds.setUrl(url);
        ds.setUsername(userName);
        ds.setPassword(password);
        return ds;
    }
}
```

4. 主配置类中读properties并引入数据源配置类

```
@Configuration
@ComponentScan("com.ww")
@PropertySource("classpath:jdbc.properties")
@Import(JdbcConfig.class)
public class SpringConfig {
}
```

5. 创建Mybatis配置类并配置SqlSessionFactory

```
public class MybatisConfig {
    //定义bean，SqlSessionFactoryBean，用于产生SqlSessionFactory对象
    @Bean
    public SqlSessionFactoryBean sqlSessionFactory(DataSource
dataSource){
        SqlSessionFactoryBean ssfb = new SqlSessionFactoryBean();
        ssfb.setTypeAliasesPackage("com.ww.pojo");
        ssfb.setDataSource(dataSource);
        return ssfb;
    }
    //定义bean，返回MapperScannerConfigurer对象
    @Bean
    public MapperScannerConfigurer mapperScannerConfigurer(){
        MapperScannerConfigurer msc = new MapperScannerConfigurer();
        msc.setBasePackage("com.ww.mapper");
        return msc;
    }
}
```

6. 主配置类中引入Mybatis配置类

```
@Configuration
@ComponentScan("com.ww")
@PropertySource("classpath:jdbc.properties")
@Import({JdbcConfig.class,MybatisConfig.class})
public class SpringConfig {
}
```

7. 编写运行类

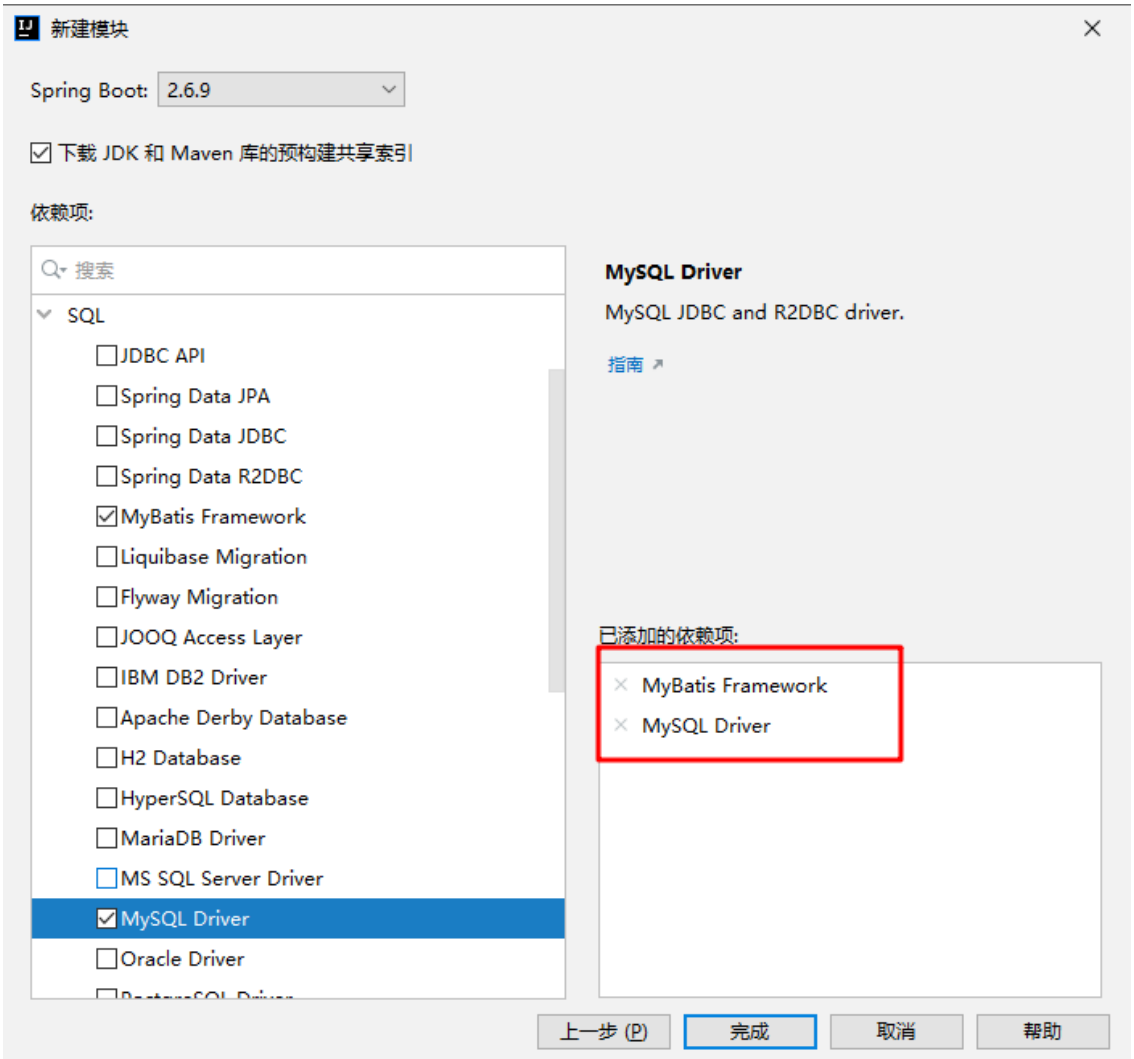
```java
public class App {
    public static void main(String[] args) {
        ApplicationContext context = new
AnnotationConfigApplicationContext(SpringConfig.class);
        UserService bean = context.getBean(UserService.class);
        User user = bean.findById(3);
        System.out.println(user);
    }
}
```

8. 运行结果

```
D:\JDK\bin\java.exe ...
7月 08, 2022 9:30:18 下午 com.alibaba.druid.support.logging.JakartaCommonsLoggingImpl info
信息: {dataSource-1} inited
User{id=3, username='admin', password='1234', age=20, sex='男', email='123@qq.com'}

进程已结束,退出代码0
```

十三、SpringBoot整合MyBatis

1. 创建SpringBoot项目，勾选当前模块需要的技术栈



2. 定义实体类

```java
public class User {
```

```
    private Integer id;

    private String username;

    private String password;

    private Integer age;

    private String sex;

    private String email;

    public User() {
    }

    public User(Integer id, String username, String password, Integer
age, String sex, String email) {
        this.id = id;
        this.username = username;
        this.password = password;
        this.age = age;
        this.sex = sex;
        this.email = email;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }

    public String getSex() {
```

```
        return sex;
    }

    public void setSex(String sex) {
        this.sex = sex;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    @Override
    public String toString() {
        return "User{" +
                "id=" + id +
                ", username='" + username + '\'' +
                ", password='" + password + '\'' +
                ", age=" + age +
                ", sex='" + sex + '\'' +
                ", email='" + email + '\'' +
                '}';
    }
}
```

3. 定义mapper接口

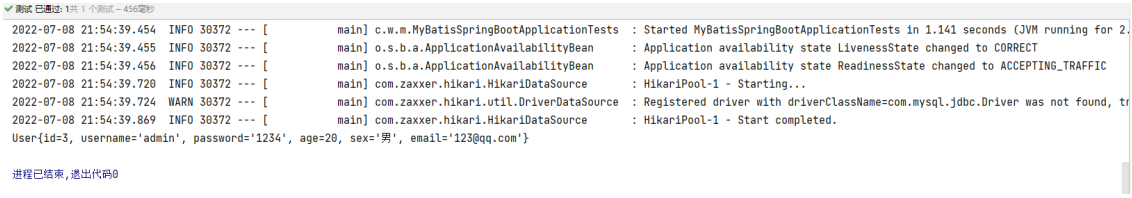
```
@Mapper
public interface UserMapper {

    @Select("select * from t_user where id = #{id})")
    User getById(Integer id);
}
```

4. 编写配置文件

```
spring:
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql:///mybatis
    username: root
    password: root
```

5. 编写测试方法

```java
@SpringBootTest
class MyBatisSpringBootApplicationTests {

    @Autowired
    private UserMapper userMapper;

    @Test
    void testGetById() {
        User user = userMapper.getById(3);
        System.out.println(user);
    }

}
```



6. 加入druid依赖

```xml
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.1.16</version>
</dependency>
```

7. 修改配置文件

```yaml
spring:
  datasource:
    type: com.alibaba.druid.pool.DruidDataSource
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql:///mybatis
    username: root
    password: root
```

结束