8. MyBatis 操作数据库(入门)

本节目标

1. 使用MyBatis完成简单的增删改查操作, 参数传递.
2. 掌握MyBatis的两种写法: 注解 和 XML方式
3. 掌握MyBatis 相关的日志配置

前言

在应用分层学习时, 我们了解到web应用程序一般分为三层，即：Controller、Service、Dao .

之前的案例中，请求流程如下: 浏览器发起请求, 先请求Controller, Controller接收到请求之后, 调用Service进行业务逻辑处理, Service再调用Dao, 但是Dao层的数据是Mock的, 真实的数据应该从数据库中读取.

我们学习MySQL数据库时，已经学习了JDBC来操作数据库, 但是JDBC操作太复杂了.

JDBC 操作示例回顾

我们先来回顾一下 JDBC 的操作流程:

1. 创建数据库连接池 DataSource
2. 通过 DataSource 获取数据库连接 Connection
3. 编写要执行带？占位符的 SQL 语句
4. 通过 Connection 及 SQL 创建操作命令对象 Statement
5. 替换占位符：指定要替换的数据库字段类型，占位符索引及要替换的值
6. 使用 Statement 执行 SQL 语句
7. 查询操作：返回结果集 ResultSet，更新操作：返回更新的数量
8. 处理结果集
9. 释放资源

下面的一个完整案例，展示了通过 JDBC 的 API 向数据库中添加一条记录，修改一条记录，查询一条记录的操作。

```
1 -- 创建数据库
2 create database if not exists library default character set utf8mb4;
3 --  使用数据库
```

```
use library;
-- 创建表
create table if not exists soft_bookrack (
  book_name varchar(32) NOT NULL,
  book_author varchar(32) NOT NULL,
  book_isbn varchar(32) NOT NULL primary key
);
```

以下是 JDBC 操作的具体实现代码：

```
package com.example.demo.mapper;
import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class SimpleJdbcOperation {

    private final DataSource dataSource;

    public SimpleJdbcOperation(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    /**
     * 添加一本书
     */
    public void addBook() {
        Connection connection = null;
        PreparedStatement stmt = null;
        try {
            //获取数据库连接
            connection = dataSource.getConnection();
            //创建语句
            stmt = connection.prepareStatement(
                    "insert into soft_bookrack (book_name, book_author, book_isbn) values (?,?,?);"
            );
            //参数绑定
            stmt.setString(1, "Spring in Action");
            stmt.setString(2, "Craig Walls");
            stmt.setString(3, "9787115417305");
            //执行语句
            stmt.execute();
```

```
35          } catch (SQLException e) {
36              //处理异常信息
37          } finally {
38              //清理资源
39              try {
40                  if (stmt != null) {
41                      stmt.close();
42                  }
43                  if (connection != null) {
44                      connection.close();
45                  }
46              } catch (SQLException e) {
47                  //
48              }
49          }
50      }
51
52      /**
53       * 更新一本书
54       */
55      public void updateBook() {
56          Connection connection = null;
57          PreparedStatement stmt = null;
58          try {
59              //获取数据库连接
60              connection = dataSource.getConnection();
61              //创建语句
62              stmt = connection.prepareStatement(
63                      "update soft_bookrack set book_author=? where book_isbn=?;"
64              );
65              //参数绑定
66              stmt.setString(1, "张卫滨");
67              stmt.setString(2, "9787115417305");
68              //执行语句
69              stmt.execute();
70          } catch (SQLException e) {
71              //处理异常信息
72          } finally {
73              //清理资源
74              try {
75                  if (stmt != null) {
76                      stmt.close();
77                  }
78                  if (connection != null) {
79                      connection.close();
80                  }
81              } catch (SQLException e) {
```

```
82                  //
83              }
84          }
85      }
86
87      /**
88       * 查询一本书
89       */
90      public void queryBook() {
91          Connection connection = null;
92          PreparedStatement stmt = null;
93          ResultSet rs = null;
94          Book book = null;
95          try {
96              //获取数据库连接
97              connection = dataSource.getConnection();
98              //创建语句
99              stmt = connection.prepareStatement(
100                     "select book_name, book_author, book_isbn from
    soft_bookrack where book_isbn =?"
101             );
102             //参数绑定
103             stmt.setString(1, "9787115417305");
104             //执行语句
105             rs = stmt.executeQuery();
106             if (rs.next()) {
107                 book = new Book();
108                 book.setName(rs.getString("book_name"));
109                 book.setAuthor(rs.getString("book_author"));
110                 book.setIsbn(rs.getString("book_isbn"));
111             }
112             System.out.println(book);
113         } catch (SQLException e) {
114             //处理异常信息
115         } finally {
116             //清理资源
117             try {
118                 if (rs != null) {
119                     rs.close();
120                 }
121                 if (stmt != null) {
122                     stmt.close();
123                 }
124                 if (connection != null) {
125                     connection.close();
126                 }
127             } catch (SQLException e) {
```

```
128                 //
129             }
130         }
131     }
132 
133     public static class Book {
134         private String name;
135         private String author;
136         private String isbn;
137         //省略 setter getter 方法
138     }
139 }
```

从上述代码和操作流程可以看出，对于 JDBC 来说，整个操作非常的繁琐，我们不但要拼接每一个参数，而且还要按照模板代码的方式，一步步的操作数据库，并且在每次操作完，还要手动关闭连接等，而所有的这些操作步骤都需要在每个方法中重复书写. 那有没有一种方法，可以更简单、更方便的操作数据库呢?

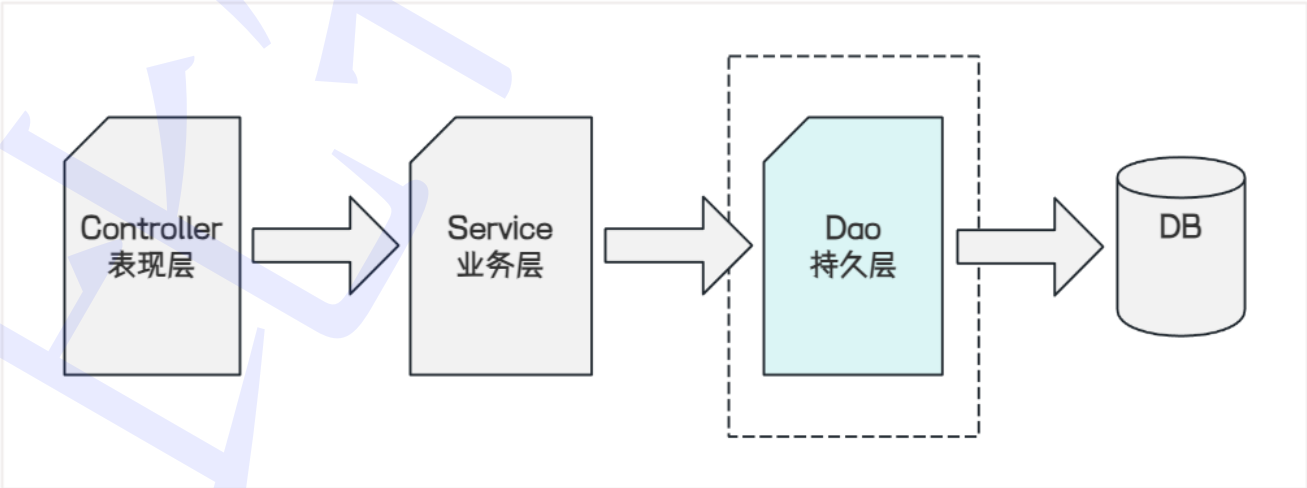
答案是肯定的，这就是我们要学习 MyBatis 的真正原因，它可以帮助我们更方便、更快速的操作数据库.

1. 什么是MyBatis?

- MyBatis是一款优秀的 **持久层** 框架，用于简化JDBC的开发。
- MyBatis本是 Apache的一个开源项目iBatis，2010年这个项目由apache迁移到了google code，并且改名为MyBatis 。2013年11月迁移到Github.
- 官网：MyBatis中文网

在上面我们提到一个词：持久层

- 持久层：指的就是持久化操作的层, 通常指数据访问层(dao), 是用来操作数据库的.



简单来说 MyBatis 是更简单完成程序和数据库交互的框架，也就是更简单的操作和读取数据库工具

接下来，我们就通过一个入门程序，让大家感受一下通过Mybatis如何来操作数据库

2. MyBatis入门

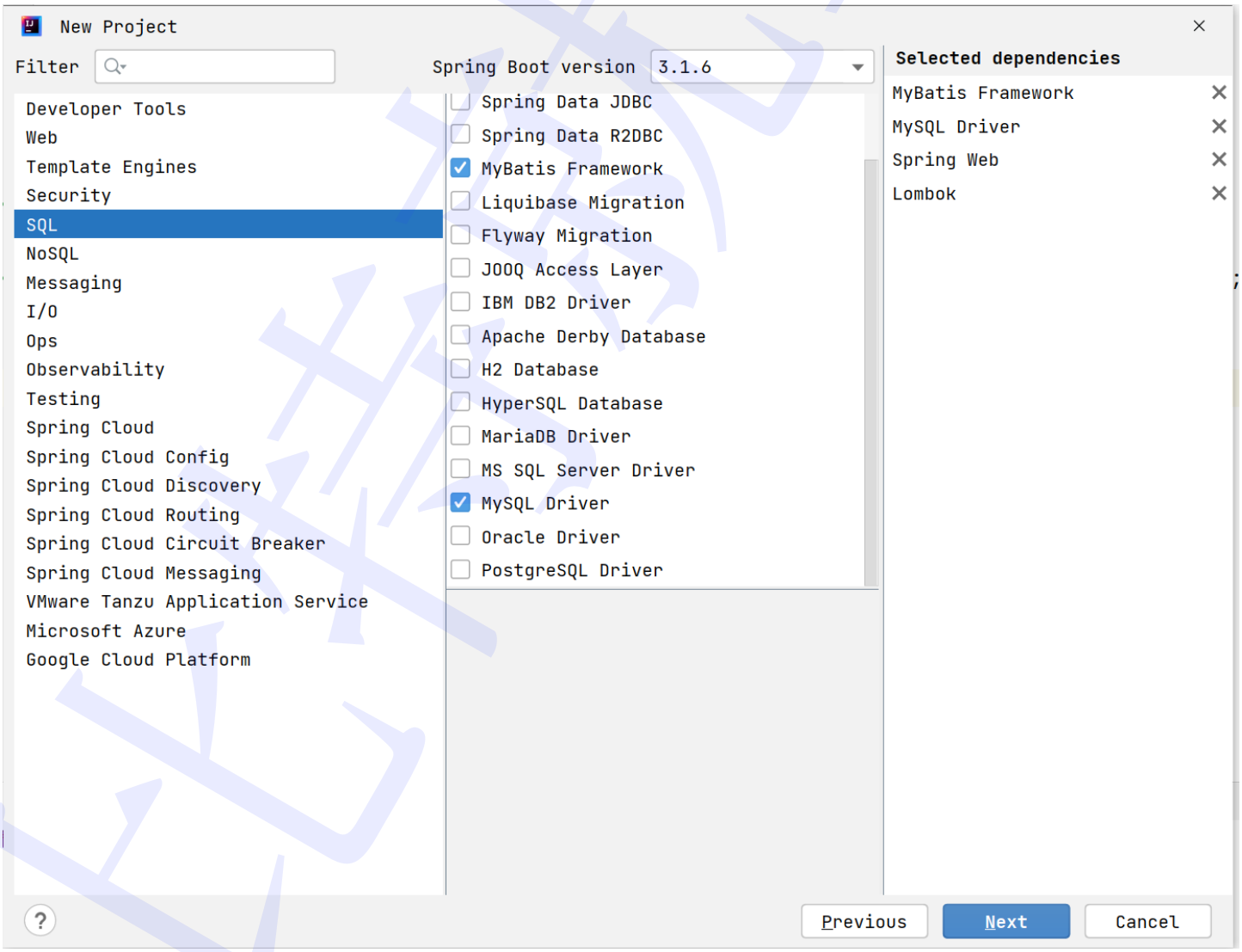
Mybatis操作数据库的步骤：

1. 准备工作(创建springboot工程、数据库表准备、实体类)
2. 引入Mybatis的相关依赖，配置Mybatis(数据库连接信息)
3. 编写SQL语句(注解/XML)
4. 测试

2.1 准备工作

2.1.1 创建工程

创建springboot工程，并导入 mybatis的起步依赖、mysql的驱动包



> Mybatis 是一个持久层框架, 具体的数据存储和数据操作还是在MySQL中操作的, 所以需要添加MySQL驱动

项目工程创建完成后，自动在pom.xml文件中，导入Mybatis依赖和MySQL驱动依赖

> 版本会随着SpringBoot 版本发生变化
>
> SpringBoot 3.X对用MyBatis版本为3.X
>
> 对应关系参考:https://mybatis.org/spring-boot-starter/mybatis-spring-boot-autoconfigure/

Requirements

The MyBatis-Spring-Boot-Starter requires following versions:

MyBatis-Spring-Boot-Starter	MyBatis-Spring	Spring Boot	Java
3.0	3.0	3.0 - 3.1	17 or higher
2.3	2.1	2.5 - 2.7	8 or higher
~~2.2 (EOL)~~	~~2.0 (need 2.0.6+ for enable all features)~~	~~2.5 - 2.7~~	~~8 or higher~~
~~2.1 (EOL)~~	~~2.0 (need 2.0.6+ for enable all features)~~	~~2.1 - 2.4~~	~~8 or higher~~
~~2.0 (EOL)~~	~~2.0~~	~~2.0 or 2.1~~	~~8 or higher~~
~~1.3 (EOL)~~	~~1.3~~	~~1.5~~	~~6 or higher~~
~~1.2 (EOL)~~	~~1.3~~	~~1.4~~	~~6 or higher~~
~~1.1 (EOL)~~	~~1.3~~	~~1.3~~	~~6 or higher~~
~~1.0 (EOL)~~	~~1.2~~	~~1.3~~	~~6 or higher~~

```xml
<!--Mybatis 依赖包-->
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>3.0.3</version>
</dependency>
<!--mysql驱动包-->
<dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <scope>runtime</scope>
</dependency>
```

2.1.2 数据准备

创建用户表, 并创建对应的实体类User

```sql
-- 创建数据库
DROP DATABASE IF EXISTS mybatis_test;

CREATE DATABASE mybatis_test DEFAULT CHARACTER SET utf8mb4;
```

```sql

-- 使用数据数据
USE mybatis_test;

-- 创建表[用户表]
DROP TABLE IF EXISTS user_info;
CREATE TABLE `user_info` (
        `id` INT ( 11 ) NOT NULL AUTO_INCREMENT,
        `username` VARCHAR ( 127 ) NOT NULL,
        `password` VARCHAR ( 127 ) NOT NULL,
        `age` TINYINT ( 4 ) NOT NULL,
        `gender` TINYINT ( 4 ) DEFAULT '0' COMMENT '1-男 2-女 0-默认',
        `phone` VARCHAR ( 15 ) DEFAULT NULL,
        `delete_flag` TINYINT ( 4 ) DEFAULT 0 COMMENT '0-正常, 1-删除',
        `create_time` DATETIME DEFAULT now(),
        `update_time` DATETIME DEFAULT now() ON UPDATE now(),
        PRIMARY KEY ( `id` )
) ENGINE = INNODB DEFAULT CHARSET = utf8mb4;

-- 添加用户信息
INSERT INTO mybatis_test.user_info( username, `password`, age, gender, phone )
VALUES ( 'admin', 'admin', 18, 1, '18612340001' );
INSERT INTO mybatis_test.user_info( username, `password`, age, gender, phone )
VALUES ( 'zhangsan', 'zhangsan', 18, 1, '18612340002' );
INSERT INTO mybatis_test.user_info( username, `password`, age, gender, phone )
VALUES ( 'lisi', 'lisi', 18, 1, '18612340003' );
INSERT INTO mybatis_test.user_info( username, `password`, age, gender, phone )
VALUES ( 'wangwu', 'wangwu', 18, 1, '18612340004' );
```

创建对应的实体类 UserInfo

> 实体类的属性名与表中的字段名一一对应

```java
import lombok.Data;
import java.util.Date;

@Data
public class UserInfo {
    private Integer id;
    private String username;
    private String password;
    private Integer age;
    private Integer gender;
    private String phone;
    private Integer deleteFlag;
    private Date createTime;
```

```
14      private Date updateTime;
15  }
```

2.2 配置数据库连接字符串

Mybatis中要连接数据库，需要数据库相关参数配置

- MySQL驱动类
- 登录名
- 密码
- 数据库连接字符串

如果是application.yml文件, 配置内容如下:

```
1  # 数据库连接配置
2  spring:
3    datasource:
4      url: jdbc:mysql://127.0.0.1:3306/mybatis_test?
   characterEncoding=utf8&useSSL=false
5      username: root
6      password: root
7      driver-class-name: com.mysql.cj.jdbc.Driver
```

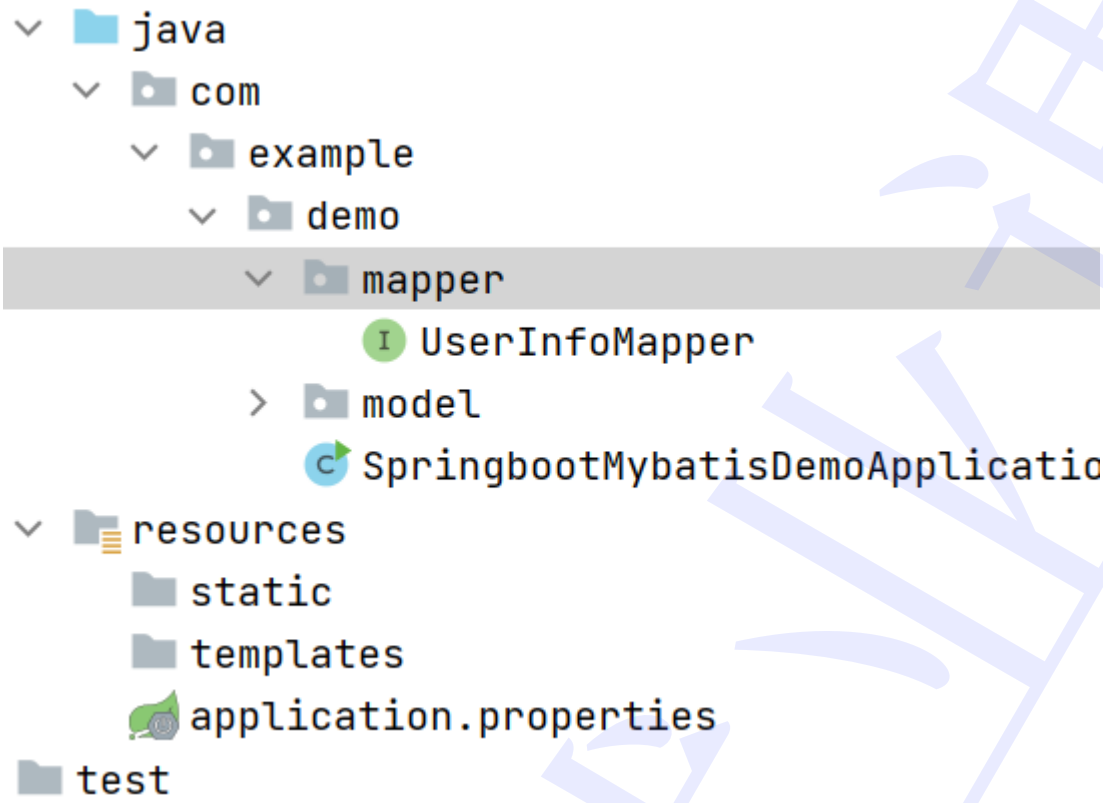
> 注意事项:
>
> 如果使用 MySQL 是 5.x 之前的使用的是"com.mysql.jdbc.Driver"，如果是大于 5.x 使用的是“com.mysql.cj.jdbc.Driver”.

如果是application.properties文件, 配置内容如下:

```
1  #驱动类名称
2  spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
3  #数据库连接的url
4  spring.datasource.url=jdbc:mysql://127.0.0.1:3306/mybatis_test?
   characterEncoding=utf8&useSSL=false
5  #连接数据库的用户名
6  spring.datasource.username=root
7  #连接数据库的密码
8  spring.datasource.password=root
```

2.3 写持久层代码

在项目中, 创建持久层接口UserInfoMapper




```java
import com.example.demo.model.UserInfo;
import org.apache.ibatis.annotations.Mapper;
import org.apache.ibatis.annotations.Select;

import java.util.List;

@Mapper
public interface UserInfoMapper {
    //查询所有用户
    @Select("select username, `password`, age, gender, phone from user_info")
    public List<UserInfo> queryAllUser();
}
```

> Mybatis的持久层接口规范一般都叫 XxxMapper
>
> @Mapper注解：表示是MyBatis中的Mapper接口
>
> - 程序运行时, 框架会自动生成接口的实现类对象(代理对象)，并给交Spring的IOC容器管理
> - @Select注解：代表的就是select查询，也就是注解对应方法的具体实现内容.

2.4 单元测试

在创建出来的SpringBoot工程中，在src下的test目录下，已经自动帮我们创建好了测试类，我们可以直接使用这个测试类来进行测试.

```java
import com.example.demo.mapper.UserInfoMapper;
import com.example.demo.model.UserInfo;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

import java.util.List;

@SpringBootTest
class DemoApplicationTests {

    @Autowired
    private UserInfoMapper userInfoMapper;

    @Test
    void contextLoads() {
        List<UserInfo> userInfoList = userInfoMapper.queryAllUser();
        System.out.println(userInfoList);
    }
}
```

> 测试类上添加了注解 @SpringBootTest，该测试类在运行时，就会自动加载Spring的运行环境.
>
> 我们通过@Autowired这个注解, 注入我们要测试的类, 就可以开始进行测试了

运行结果如下:

```
[UserInfo(id=null, username=admin, password=admin, age=18, gender=1, phone=18612340001, deleteFlag=null, createTime=null,
 updateTime=null), UserInfo(id=null, username=zhangsan, password=zhangsan, age=18, gender=1, phone=18612340002, deleteFlag=null,
 createTime=null, updateTime=null), UserInfo(id=null, username=lisi, password=lisi, age=18, gender=1, phone=18612340003,
 deleteFlag=null, createTime=null, updateTime=null), UserInfo(id=null, username=wangwu, password=wangwu, age=18, gender=1,
 phone=18612340004, deleteFlag=null, createTime=null, updateTime=null)]
```

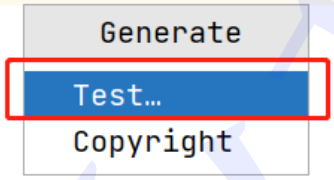
> 返回结果中, 可以看到, 只有SQL语句中查询的列对应的属性才有赋值

使用Idea 自动生成测试类

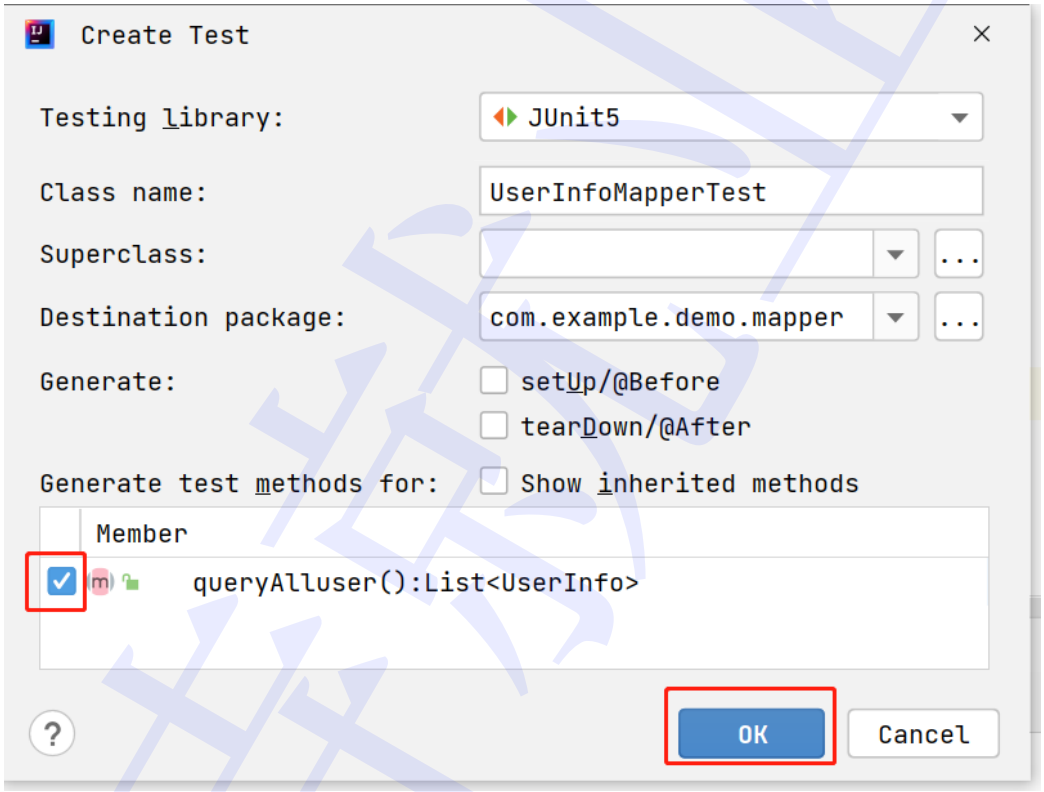
除此之外, 也可以使用Idea自动生成测试类

1. 在需要测试的Mapper接口中, 右键 -> Generate -> Test

```java
@Mapper
public interface UserInfoMapper {
    1 usage
    @Select("select username, age, gender, phone from userinfo")
    List<UserInfo> queryAlluser();
```



2. 选择要测试的方法, 点击 OK



3. 书写测试代码

```java
import com.example.demo.model.UserInfo;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

import java.util.List;

@SpringBootTest
class UserInfoMapperTest {
    @Autowired
    private UserInfoMapper userInfoMapper;

```

```
    @Test
    void queryAllUser() {
        List<UserInfo> userInfoList = userInfoMapper.queryAllUser();
        System.out.println(userInfoList);
    }
}
```

> 记得加 `@SpringBootTest` 注解, 加载Spring运行环境

运行结果:

```
[UserInfo(id=null, username=admin, password=admin, age=18, gender=1, phone=18612340001, deleteFlag=null, createTime=null,
 updateTime=null), UserInfo(id=null, username=zhangsan, password=zhangsan, age=18, gender=1, phone=18612340002, deleteFlag=null,
 createTime=null, updateTime=null), UserInfo(id=null, username=lisi, password=lisi, age=18, gender=1, phone=18612340003,
 deleteFlag=null, createTime=null, updateTime=null), UserInfo(id=null, username=wangwu, password=wangwu, age=18, gender=1,
 phone=18612340004, deleteFlag=null, createTime=null, updateTime=null)]
```

3. MyBatis的基础操作

上面我们学习了Mybatis的查询操作, 接下来我们学习MyBatis的增, 删, 改操作

在学习这些操作之前, 我们先来学习MyBatis日志打印

3.1 打印日志

在Mybatis当中我们可以借助日志, 查看到sql语句的执行、执行传递的参数以及执行结果

在配置文件中进行配置即可

```yaml
mybatis:
  configuration: # 配置打印 MyBatis日志
    log-impl: org.apache.ibatis.logging.stdout.StdOutImpl
```

如果是application.properties, 配置内容如下:

```properties
#指定mybatis输出日志的位置, 输出控制台
mybatis.configuration.log-impl=org.apache.ibatis.logging.stdout.StdOutImpl
```

> 注意: 后续配置项, 默认只提供一种, 请自行进行配置项转换.

重新运行程序, 可以看到SQL执行内容, 以及传递参数和执行结果

```
==>  Preparing: select username, `password`, age, gender, phone from userinfo ①
==> Parameters: ②
<==    Columns: username, password, age, gender, phone
<==        Row: admin, admin, 18, 1, 18612340001
<==        Row: zhangsan, zhangsan, 18, 1, 18612340002
<==        Row: lisi, lisi, 18, 1, 18612340003
<==        Row: wangwu, wangwu, 18, 1, 18612340004
<==      Total: 4
```

③

> ①: 查询语句
>
> ②: 传递参数及类型
>
> ③: SQL执行结果

3.2 参数传递

需求: 查找id=4的用户,对应的SQL就是: select * from user_info where id=4

```
1 @Select("select username, `password`, age, gender, phone from user_info where
  id= 4 ")
2 UserInfo queryById();
```

但是这样的话, 只能查找id=4 的数据, 所以SQL语句中的id值不能写成固定数值，需要变为动态的数值

解决方案：在queryById方法中添加一个参数(id)，将方法中的参数，传给SQL语句

使用 `#{}` 的方式获取方法中的参数

```
1 @Select("select username, `password`, age, gender, phone from user_info where
  id= #{id} ")
2 UserInfo queryById(Integer id);
```

> 如果mapper接口方法形参只有一个普通类型的参数，#{…} 里面的属性名可以随便写，如：#{id}、#{value}。建议和参数名保持一致

添加测试用例

```
1 @Test
2 void queryById() {
3     UserInfo userInfo = userInfoMapper.queryById(4);
4     System.out.println(userInfo);
5 }
```

运行结果:

```
==>  Preparing: select username, `password`, age, gender, phone from userinfo where id= ?
==> Parameters: 4(Integer)
<==    Columns: username, password, age, gender, phone
<==        Row: wangwu, wangwu, 18, 1, 18612340004
<==      Total: 1
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@642413d4]
UserInfo(id=null, username=wangwu, password=wangwu, age=18, gender=1, phone=18612340004, deleteFlag=null, createTime=null,
 updateTime=null)
```

也可以通过 `@Param` , 设置参数的别名, 如果使用 `@Param` 设置别名, #{...}里面的属性名必须和 `@Param` 设置的一样

```
1 @Select("select username, `password`, age, gender, phone from user_info where
  id= #{userid} ")
2 UserInfo queryById(@Param("userid") Integer id);
```

3.3 增(Insert)

SQL 语句:

```
1 insert into user_info (username, `password`, age, gender, phone) values
  ("zhaoliu","zhaoliu",19,1,"18700001234")
2
```

把SQL中的常量替换为动态的参数

Mapper接口

```
1 @Insert("insert into user_info (username, `password`, age, gender, phone)
  values (#{username},#{password},#{age},#{gender},#{phone})")
2 Integer insert(UserInfo userInfo);
```

直接使用UserInfo对象的属性名来获取参数

测试代码:

```
1 @Test
2 void insert() {
3     UserInfo userInfo = new UserInfo();
```

```
    userInfo.setUsername("zhaoliu");
    userInfo.setPassword("zhaoliu");
    userInfo.setGender(2);
    userInfo.setAge(21);
    userInfo.setPhone("18612340005");
    userInfoMapper.insert(userInfo);
}
```

运行后, 观察数据库执行结果

> 如果设置了 `@Param` 属性, #{...} 需要使用 参数.属性 来获取
>
> ```
> @Insert("insert into user_info (username, `password`, age, gender, phone)
> values (#{userInfo.username},#{userInfo.password},#{userInfo.age},#
> {userInfo.gender},#{userInfo.phone})")
> Integer insert(@Param("userInfo") UserInfo userInfo);
> ```

返回主键

Insert 语句默认返回的是 受影响的行数

但有些情况下, 数据插入之后, 还需要有后续的关联操作, 需要获取到新插入数据的id

> 比如订单系统
>
> 当我们下完订单之后, 需要通知物流系统, 库存系统, 结算系统等, 这时候就需要拿到订单ID

如果想要拿到自增id, 需要在Mapper接口的方法上添加一个Options的注解

```
@Options(useGeneratedKeys = true, keyProperty = "id")
@Insert("insert into user_info (username, age, gender, phone) values (#
{userInfo.username},#{userInfo.age},#{userInfo.gender},#{userInfo.phone})")
Integer insert(@Param("userInfo") UserInfo userInfo);
```

> - useGeneratedKeys：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：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）

测试数据:

```
@Test
```

```
void insert() {
    UserInfo userInfo = new UserInfo();
    userInfo.setUsername("zhaoliu");
    userInfo.setPassword("zhaoliu");
    userInfo.setGender(2);
    userInfo.setAge(21);
    userInfo.setPhone("18612340005");
    Integer count = userInfoMapper.insert(userInfo);
    System.out.println("添加数据条数:" +count +", 数据ID:" + userInfo.getId());
}
```

运行结果:

```
JDBC Connection [HikariProxyConnection@1320809135 wrapping com.mysql.cj.jdbc.ConnectionImpl@4a481728] will not be managed by Spring
==>  Preparing: insert into userinfo (username, `password`, age, gender, phone) values (?,?,?,?,?)
==> Parameters: zhaoliu(String), zhaoliu(String), 21(Integer), 2(Integer), 18612340005(String)
<==    Updates: 1
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@5a02bfe3]
添加数据条数:1, 数据ID:6
```

注意: 设置 `useGeneratedKeys=true` 之后, 方法返回值依然是受影响的行数, `自增id` 会设置在上述 `keyProperty` 指定的属性中.

3.4 删(Delete)

SQL 语句:

```
delete from user_info where id=6
```

把SQL中的常量替换为动态的参数

Mapper接口

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

3.5 改(Update)

SQL 语句:

```
update user_info set username="zhaoliu" where id=5
```

把SQL中的常量替换为动态的参数

Mapper接口

```
@Update("update user_info set username=#{username} where id=#{id}")
void update(UserInfo userInfo);
```

3.6 查(Select)

我们在上面查询时发现, 有几个字段是没有赋值的, 只有Java对象属性和数据库字段一模一样时, 才会进行赋值

接下来我们多查询一些数据

```
@Select("select id, username, `password`, age, gender, phone, delete_flag,
create_time, update_time from user_info")
List<UserInfo> queryAllUser();
```

查询结果:

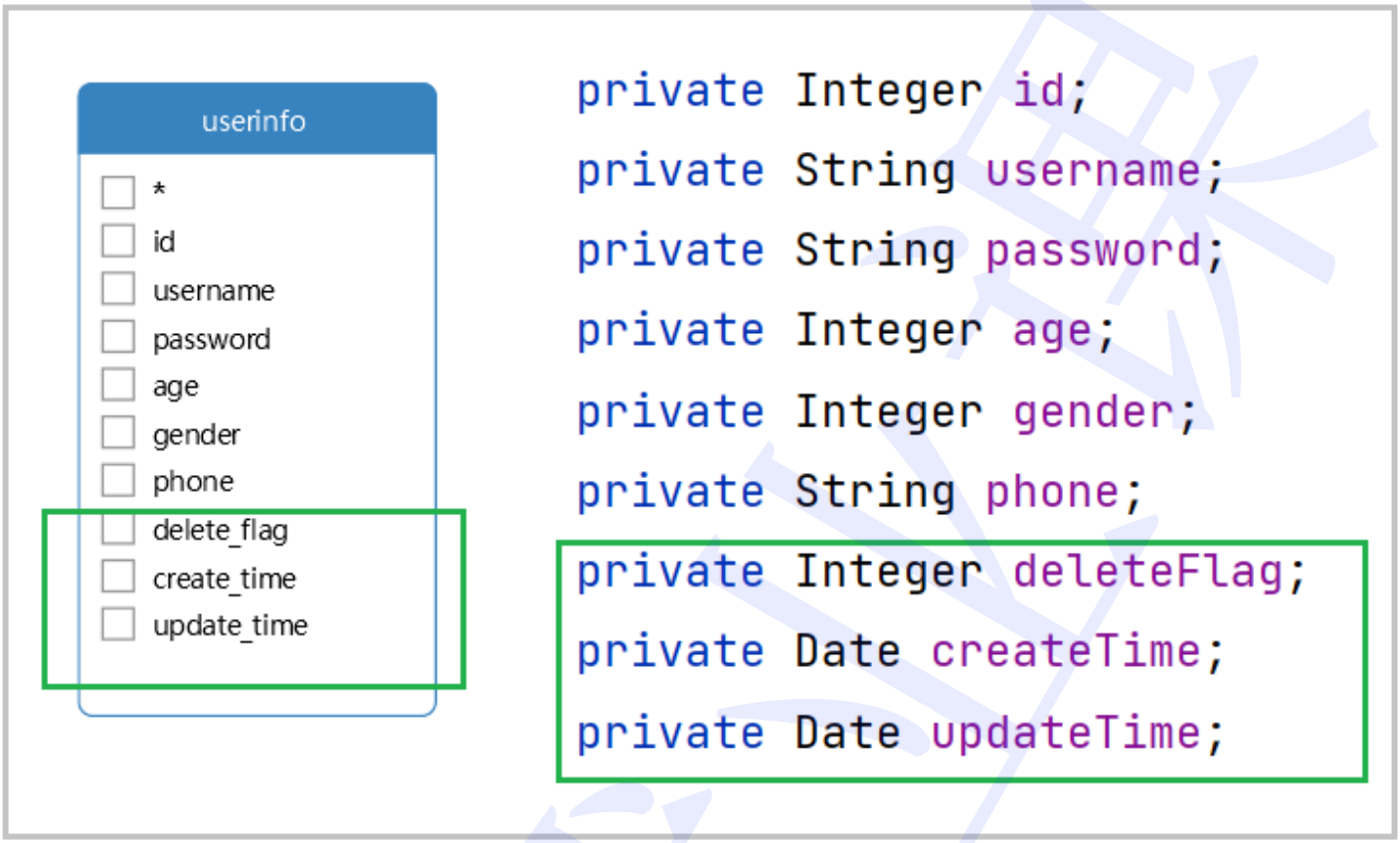
```
[UserInfo(id=1, username=admin, password=null, age=18, gender=1, phone=18612340001, deleteFlag=null, createTime=null,
updateTime=null), UserInfo(id=2, username=zhangsan, password=null, age=18, gender=1, phone=18612340002, deleteFlag=null,
createTime=null, updateTime=null), UserInfo(id=3, username=lisi, password=null, age=18, gender=1, phone=18612340003,
deleteFlag=null, createTime=null, updateTime=null), UserInfo(id=4, username=wangwu, password=null, age=18, gender=1,
phone=18612340004, deleteFlag=null, createTime=null, updateTime=null)]
```

从运行结果上可以看到, 我们SQL语句中, 查询了delete_flag, create_time, update_time, 但是这几个属性却没有赋值.

> MyBatis 会根据方法的返回结果进行赋值.
>
> 方法用对象 UserInfo接收返回结果, MySQL 查询出来数据为一条, 就会自动赋值给对象.
>
> 方法用List<UserInfo>接收返回结果, MySQL 查询出来数据为一条或多条时, 也会自动赋值给List.
>
> 但如果MySQL 查询返回多条, 但是方法使用UserInfo接收, MyBatis执行就会报错.

原因分析:

当自动映射查询结果时，MyBatis 会获取结果中返回的列名并在 Java 类中查找相同名字的属性（忽略大小写）。 这意味着如果发现了 ID 列和 id 属性，MyBatis 会将列 ID 的值赋给 id 属性



解决办法:

1. 起别名
2. 结果映射
3. 开启驼峰命名

3.6.1 起别名

在SQL语句中，给列名起别名，保持别名和实体类属性名一样

```
1 @Select("select id, username, `password`, age, gender, phone, delete_flag as
  deleteFlag, " +
2         "create_time as createTime, update_time as updateTime from user_info")
3 public List<UserInfo> queryAllUser();
```

> SQL语句太长时, 使用加号 `+` 进行字符串拼接

3.6.2 结果映射

```
1 @Select("select id, username, `password`, age, gender, phone, delete_flag,
  create_time, update_time from user_info")
```

```
@Results({
        @Result(column = "delete_flag",property = "deleteFlag"),
        @Result(column = "create_time",property = "createTime"),
        @Result(column = "update_time",property = "updateTime")
})
List<UserInfo> queryAllUser();
```

如果其他SQL, 也希望可以复用这个映射关系, 可以给这个Results定义一个名称

```
@Select("select id, username, `password`, age, gender, phone, delete_flag,
create_time, update_time from user_info")
@Results(id = "resultMap",value = {
        @Result(column = "delete_flag",property = "deleteFlag"),
        @Result(column = "create_time",property = "createTime"),
        @Result(column = "update_time",property = "updateTime")
})
List<UserInfo> queryAllUser();

@Select("select id, username, `password`, age, gender, phone, delete_flag,
create_time, update_time " +
        "from user_info where id= #{userid} ")
@ResultMap(value = "resultMap")
UserInfo queryById(@Param("userid") Integer id);
```

使用 `id` 属性给该 `Results` 定义别名, 使用 `@ResultMap` 注解来复用其他定义的 `ResultMap`

```
@Select("select id, username, `password`, age, gender, phone, delete_flag, create_time, update_time from userinfo")
@Results(id = "resultMap",value = {
        @Result(column = "delete_flag",property = "deleteFlag"),
        @Result(column = "create_time",property = "createTime"),
        @Result(column = "update_time",property = "updateTime")
})
List<UserInfo> queryAllUser();

@Select("select id, username, `password`, age, gender, phone, delete_flag, create_time, update_time " +
        "from userinfo where id= #{userid} ")
@ResultMap(value = "resultMap")
UserInfo queryById(@Param("userid") Integer id);
```

3.6.3 开启驼峰命名(推荐)

通常数据库列使用蛇形命名法进行命名(下划线分割各个单词), 而 Java 属性一般遵循驼峰命名法约定. 为了在这两种命名方式之间启用自动映射，需要将 `mapUnderscoreToCamelCase` 设置为 true。

```
mybatis:
```

```
  configuration:
    map-underscore-to-camel-case: true #配置驼峰自动转换
```

> 驼峰命名规则： abc_xyz => abcXyz
>
> - 表中字段名：abc_xyz
> - 类中属性名：abcXyz

Java 代码不做任何处理

```
@Select("select id, username, `password`, age, gender, phone, delete_flag, " +
        "create_time, update_time from user_info")
public List<UserInfo> queryAllUser();
```

添加上述配置, 运行代码:

```
[UserInfo(id=1, username=admin, password=admin, age=18, gender=1, phone=18612340001, deleteFlag=0, createTime=Sun Sep 24 10:53:45
 CST 2023, updateTime=Sun Sep 24 10:53:45 CST 2023), UserInfo(id=2, username=zhangsan, password=zhangsan, age=18, gender=1,
 phone=18612340002, deleteFlag=0, createTime=Sun Sep 24 10:53:45 CST 2023, updateTime=Sun Sep 24 10:53:45 CST 2023), UserInfo(id=3,
 username=lisi, password=lisi, age=18, gender=1, phone=18612340003, deleteFlag=0, createTime=Sun Sep 24 10:53:45 CST 2023,
 updateTime=Sun Sep 24 10:53:45 CST 2023), UserInfo(id=4, username=wangwu, password=wangwu, age=18, gender=1, phone=18612340004,
 deleteFlag=0, createTime=Sun Sep 24 10:53:45 CST 2023, updateTime=Sun Sep 24 10:53:45 CST 2023), UserInfo(id=5, username=zhaoliu,
 password=zhaoliu, age=21, gender=2, phone=18612340005, deleteFlag=0, createTime=Sun Sep 24 11:22:56 CST 2023, updateTime=Sun Sep 24
  11:22:56 CST 2023), UserInfo(id=6, username=zhaoliu, password=zhaoliu, age=21, gender=2, phone=18612340005, deleteFlag=0,
 createTime=Sun Sep 24 11:26:28 CST 2023, updateTime=Sun Sep 24 11:26:28 CST 2023)]
```

字段全部进行正确赋值.

4. MyBatis XML配置文件

Mybatis的开发有两种方式:

1. 注解
2. XML

上面学习了注解的方式, 接下来我们学习XML的方式

> 使用Mybatis的注解方式，主要是来完成一些简单的增删改查功能. 如果需要实现复杂的SQL功能，建议使用XML来配置映射语句，也就是将SQL语句写在XML配置文件中.

MyBatis XML的方式需要以下两步:

1. 配置数据库连接字符串和MyBatis
2. 写持久层代码

4.1 配置连接字符串和MyBatis

此步骤需要进行两项设置，数据库连接字符串设置和 MyBatis 的 XML 文件配置。

如果是application.yml文件, 配置内容如下:

```yaml
# 数据库连接配置
spring:
  datasource:
    url: jdbc:mysql://127.0.0.1:3306/mybatis_test?characterEncoding=utf8&useSSL=false
    username: root
    password: root
    driver-class-name: com.mysql.cj.jdbc.Driver
# 配置 mybatis xml 的文件路径，在 resources/mapper 创建所有表的 xml 文件
mybatis:
  mapper-locations: classpath:mapper/**Mapper.xml
```

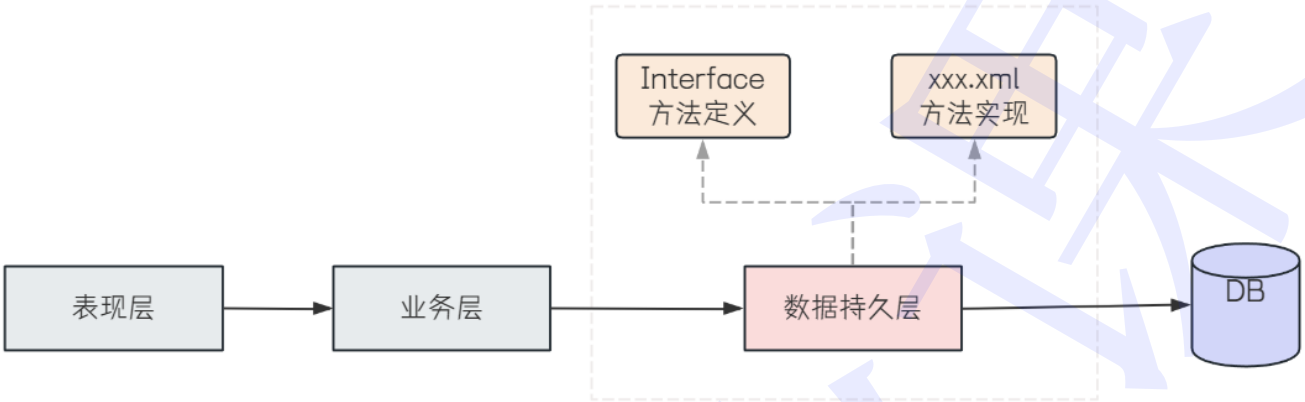
如果是application.properties文件, 配置内容如下:

```properties
#驱动类名称
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
#数据库连接的url
spring.datasource.url=jdbc:mysql://127.0.0.1:3306/mybatis_test?characterEncoding=utf8&useSSL=false
#连接数据库的用户名
spring.datasource.username=root
#连接数据库的密码
spring.datasource.password=root
# 配置 mybatis xml 的文件路径，在 resources/mapper 创建所有表的 xml 文件
mybatis.mapper-locations=classpath:mapper/**Mapper.xml
```

4.2 写持久层代码

持久层代码分两部分

1. 方法定义 Interface
2. 方法实现: XXX.xml



4.2.1 添加 mapper 接口

数据持久层的接口定义：

```
1 import com.example.demo.model.UserInfo;
2 import org.apache.ibatis.annotations.Mapper;
3
4 import java.util.List;
5
6 @Mapper
7 public interface UserInfoXMlMapper {
8     List<UserInfo> queryAllUser();
9 }
```

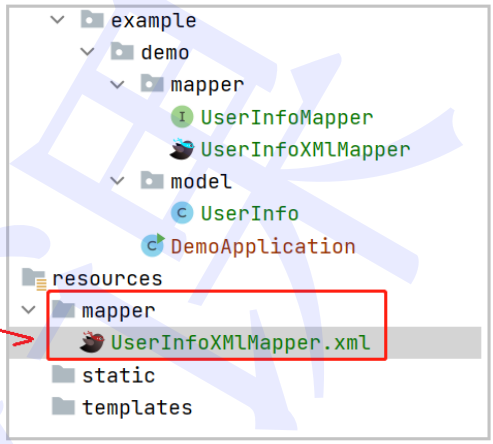
4.2.2 添加 UserInfoXMLMapper.xml

数据持久成的实现，MyBatis 的固定 xml 格式：

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
3 <mapper namespace="com.example.demo.mapper.UserInfoMapper">
4
5 </mapper>
```

创建UserInfoXMLMapper.xml, 路径参考yml中的配置

```yaml
mybatis:
  configuration: # 配置打印 MyBatis日志
    log-impl: org.apache.ibatis.logging.stdout.StdOutImpl
    map-underscore-to-camel-case: true #配置驼峰自动转换
  mapper-locations: classpath:mapper/**Mapper.xml
```



查询所有用户的具体实现：

```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.example.demo.mapper.UserInfoXMlMapper">
    <select id="queryAllUser" resultType="com.example.demo.model.UserInfo">
        select username,`password`, age, gender, phone from user_info
    </select>
</mapper>
```

以下是对以上标签的说明：

- `<mapper>` 标签：需要指定 `namespace` 属性，表示命名空间，值为 mapper 接口的全限定名，包括全包名.类名。
- `<select>` 查询标签：是用来执行数据库的查询操作的：
  - `id`：是和 `Interface` （接口）中定义的方法名称一样的，表示对接口的具体实现方法。
  - `resultType`：是返回的数据类型，也就是开头我们定义的实体类.

4.2.3 单元测试

```java
@SpringBootTest
class UserInfoMapperTest {
    @Autowired
    private UserInfoMapper userInfoMapper;

    @Test
    void queryAllUser() {
        List<UserInfo> userInfoList = userInfoMapper.queryAllUser();
        System.out.println(userInfoList);
    }

}
```

运行结果如下:

```
[UserInfo(id=null, username=admin, password=admin, age=18, gender=1, phone=18612340001, deleteFlag=null, createTime=null,
 updateTime=null), UserInfo(id=null, username=zhangsan, password=zhangsan, age=18, gender=1, phone=18612340002, deleteFlag=null,
 createTime=null, updateTime=null), UserInfo(id=null, username=lisi, password=lisi, age=18, gender=1, phone=18612340003,
 deleteFlag=null, createTime=null, updateTime=null), UserInfo(id=null, username=wangwu, password=wangwu, age=18, gender=1,
 phone=18612340004, deleteFlag=null, createTime=null, updateTime=null), UserInfo(id=null, username=zhaoliu, password=zhaoliu,
 age=21, gender=2, phone=18612340005, deleteFlag=null, createTime=null, updateTime=null), UserInfo(id=null, username=zhaoliu,
 password=zhaoliu, age=21, gender=2, phone=18612340005, deleteFlag=null, createTime=null, updateTime=null)]
```

4.3 增删改查操作

接下来，我们来实现一下用户的增加、删除和修改的操作.

4.3.1 增(Insert)

UserInfoMapper接口:

```java
Integer insertUser(UserInfo userInfo);
```

UserInfoMapper.xml实现:

```xml
<insert id="insertUser">
    insert into userinfo (username, `password`, age, gender, phone) values (#
  {username}, #{password}, #{age},#{gender},#{phone})
</insert>
```

> 如果使用@Param设置参数名称的话, 使用方法和注解类似
>
> UserInfoMapper接口:

```
Integer insertUser(@Param("userInfo") UserInfo userInfo);
```

UserInfoMapper.xml实现:

```
<insert id="insertUser">
    insert into user_info (username, `password`, age, gender, phone) values
    (#{userInfo.username},#{userInfo.password},#{userInfo.age},#
{userInfo.gender},#{userInfo.phone})
</insert>
```

返回自增 id

接口定义不变, Mapper.xml 实现 设置useGeneratedKeys 和keyProperty属性

```
<insert id="insertUser" useGeneratedKeys="true" keyProperty="id">
    insert into user_info (username, `password`, age, gender, phone) values
    (#{userInfo.username},#{userInfo.password},#{userInfo.age},#
{userInfo.gender},#{userInfo.phone})
</insert>
```

4.3.2 删(Delete)

UserInfoMapper接口:

```
Integer deleteUser(Integer id);
```

UserInfoMapper.xml实现:

```
<delete id="deleteUser">
    delete from user_info where id = #{id}
</delete>
```

4.3.3 改(Update)

UserInfoMapper接口:

```
Integer updateUser(UserInfo userInfo);
```

UserInfoMapper.xml实现:

```
<update id="updateUser">
    update user_info set username=#{username} where id=#{id}
</update>
```

4.3.4 查(Select)

同样的, 使用XML 的方式进行查询, 也存在数据封装的问题

我们把SQL语句进行简单修改, 查询更多的字段内容

```
<select id="queryAllUser" resultType="com.example.demo.model.UserInfo">
    select id, username,`password`, age, gender, phone, delete_flag,
create_time, update_time from user_info
</select>
```

运行结果:

```
[UserInfo(id=1, username=admin, password=admin, age=18, gender=1, phone=18612340001, deleteFlag=null, createTime=null,
 updateTime=null), UserInfo(id=2, username=zhangsan, password=zhangsan, age=18, gender=1, phone=18612340002, deleteFlag=null,
 createTime=null, updateTime=null), UserInfo(id=3, username=lisi, password=lisi, age=18, gender=1, phone=18612340003,
 deleteFlag=null, createTime=null, updateTime=null), UserInfo(id=4, username=wangwu, password=wangwu, age=18, gender=1,
 phone=18612340004, deleteFlag=null, createTime=null, updateTime=null), UserInfo(id=5, username=zhaoliu, password=zhaoliu, age=21,
 gender=2, phone=18612340005, deleteFlag=null, createTime=null, updateTime=null), UserInfo(id=6, username=zhaoliu, password=zhaoliu,
  age=21, gender=2, phone=18612340005, deleteFlag=null, createTime=null, updateTime=null), UserInfo(id=7, username=zhaoliu,
 password=zhaoliu, age=21, gender=2, phone=18612340005, deleteFlag=null, createTime=null, updateTime=null)]
```

结果显示: deleteFlag, createTime, updateTime 也没有进行赋值.

解决办法和注解类似:

1. 起别名
2. 结果映射
3. 开启驼峰命名

其中1,3的解决办法和注解一样,不再多说, 接下来看下xml如果来写结果映射

Mapper.xml

```
<resultMap id="BaseMap" type="com.example.demo.model.UserInfo">
    <id column="id" property="id"></id>
```

```
    <result column="delete_flag" property="deleteFlag"></result>
    <result column="create_time" property="createTime"></result>
    <result column="update_time" property="updateTime"></result>
</resultMap>

<select id="queryAllUser" resultMap="BaseMap">
    select id, username,`password`, age, gender, phone, delete_flag,
create_time, update_time from user_info
</select>
```

开发中使用注解还是XML的方式?

关于开发中使用哪种模式这个问题, 没有明确答案. 仁者见仁智者见智, 并没有统一的标准, 更多是取决于你的团队或者项目经理, 项目负责人.

5. 其他查询操作

5.1 多表查询

多表查询和单表查询类似, 只是SQL不同而已

5.1.1 准备工作

上面建了一张用户表, 我们再来建一张文章表, 进行多表关联查询.

文章表的uid, 对应用户表的id.

数据准备

```
-- 创建文章表
DROP TABLE IF EXISTS articleinfo;
```

```sql

CREATE TABLE articleinfo (
        id INT PRIMARY KEY auto_increment,
        title VARCHAR ( 100 ) NOT NULL,
        content TEXT NOT NULL,
        uid INT NOT NULL,
        delete_flag TINYINT ( 4 ) DEFAULT 0 COMMENT '0-正常, 1-删除',
        create_time DATETIME DEFAULT now(),
        update_time DATETIME DEFAULT now()
) DEFAULT charset 'utf8mb4';

-- 插入测试数据
INSERT INTO articleinfo ( title, content, uid ) VALUES ( 'Java', 'Java正文', 1
);
```

对应Model:

```java
import lombok.Data;
import java.util.Date;

@Data
public class ArticleInfo {
    private Integer id;
    private String title;
    private String content;
    private Integer uid;
    private Integer deleteFlag;
    private Date createTime;
    private Date updateTime;
}
```

5.1.2 数据查询

需求: 根据uid查询作者的名称等相关信息

SQL:

```sql
SELECT
        ta.id,
        ta.title,
        ta.content,
        ta.uid,
        tb.username,
        tb.age,
```

```
        tb.gender
FROM
        articleinfo ta
        LEFT JOIN user_info tb ON ta.uid = tb.id
WHERE
        ta.id =1
```

补充实体类:

```
@Data
public class ArticleInfo {
    private Integer id;
    private String title;
    private String content;
    private Integer uid;
    private Integer deleteFlag;
    private Date createTime;
    private Date updateTime;
    //用户相关信息
    private String username;
    private Integer age;
    private Integer gender;
}
```

接口定义:

```
import com.example.demo.model.ArticleInfo;
import org.apache.ibatis.annotations.Mapper;

@Mapper
public interface ArticleInfoMapper {

    @Select("SELECT
ta.id,ta.title,ta.content,ta.uid,tb.username,tb.age,tb.gender " +
            "FROM articleinfo ta LEFT JOIN user_info tb ON ta.uid = tb.id " +
            "WHERE ta.id = #{id}")
    ArticleInfo queryUserByUid(Integer id);
}
```

> 如果名称不一致的, 采用ResultMap, 或者别名的方式解决, 和单表查询一样
>
> Mybatis 不分单表还是多表, 主要就是三部分: SQL, 映射关系和实体类

> 通过映射关系, 把SQL运行结果和实体类关联起来.

5.2 #{} 和 ${}

MyBatis 参数赋值有两种方式, 咱们前面使用了 `#{}` 进行赋值, 接下来我们看下二者的区别

5.2.1 #{} 和${} 使用

1. 先看Interger类型的参数

```
1 @Select("select username, `password`, age, gender, phone from user_info where
  id= #{id} ")
2 UserInfo queryById(Integer id);
```

观察我们打印的日志

```
JDBC Connection [HikariProxyConnection@95980430 wrapping com.mysql.cj.jdbc.ConnectionImpl@58253c57] will not be managed by Spring
==>  Preparing: select username, `password`, age, gender, phone from userinfo where id= ?
==> Parameters: 4(Integer)
<==    Columns: username, password, age, gender, phone
<==        Row: wangwu, wangwu, 18, 1, 18612340004
<==      Total: 1
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@404eca05]
UserInfo(id=null, username=wangwu, password=wangwu, age=18, gender=1, phone=18612340004, deleteFlag=null, createTime=null,
 updateTime=null)
```

发现我们输出的SQL语句:

```
1 select username, `password`, age, gender, phone from user_info where id= ?
```

我们输入的参数并没有在后面拼接，id的值是使用 `?` 进行占位. 这种SQL 我们称之为"**预编译SQL**"

> MySQL 课程 JDBC编程使用的就是预编译SQL, 此处不再多说.

我们把 `#{}` 改成 `${}` 再观察打印的日志:

```
1 @Select("select username, `password`, age, gender, phone from user_info where
  id= ${id} ")
2 UserInfo queryById(Integer id);
```

```
JDBC Connection [HikariProxyConnection@1768471788 wrapping com.mysql.cj.jdbc.ConnectionImpl@5568c66f] will not be managed by Spring
==>  Preparing: select username, `password`, age, gender, phone from userinfo where id= 4
==> Parameters:
<==    Columns: username, password, age, gender, phone
<==        Row: wangwu, wangwu, 18, 1, 18612340004
<==      Total: 1
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@61a91c9b]
UserInfo(id=null, username=wangwu, password=wangwu, age=18, gender=1, phone=18612340004, deleteFlag=null, createTime=null,
 updateTime=null)
```

可以看到, 这次的参数是直接拼接在SQL语句中了.

2. 接下来我们再看String类型的参数

```
1 @Select("select username, `password`, age, gender, phone from user_info where
  username= #{name} ")
2 UserInfo queryByName(String name);
```

观察我们打印的日志, 结果正常返回

```
JDBC Connection [HikariProxyConnection@12006451 wrapping com.mysql.cj.jdbc.ConnectionImpl@1bf14704] will not be managed by Spring
==>  Preparing: select username, `password`, age, gender, phone from userinfo where username= ?
==> Parameters: zhangsan(String)
<==    Columns: username, password, age, gender, phone
<==        Row: zhangsan, zhangsan, 18, 1, 18612340002
<==      Total: 1
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@61a91c9b]
UserInfo(id=null, username=zhangsan, password=zhangsan, age=18, gender=1, phone=18612340002, deleteFlag=null, createTime=null,
 updateTime=null)
```

我们把 `#{}` 改成 `${}` 再观察打印的日志:

```
1 @Select("select username, `password`, age, gender, phone from user_info where
  username= ${name} ")
2 UserInfo queryByName(String name);
```

```
JDBC Connection [HikariProxyConnection@1286437308 wrapping com.mysql.cj.jdbc.ConnectionImpl@2c63762b] will not be managed by Spring
==>  Preparing: select username, `password`, age, gender, phone from userinfo where username= zhangsan
==> Parameters:
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@69afa141]

org.springframework.jdbc.BadSqlGrammarException:
### Error querying database.  Cause: java.sql.SQLSyntaxErrorException: Unknown column 'zhangsan' in 'where clause'
### The error may exist in com/example/demo/mapper/UserInfoMapper.java (best guess)
### The error may involve com.example.demo.mapper.UserInfoMapper.queryByName-Inline
### The error occurred while setting parameters
### SQL: select username, `password`, age, gender, phone from userinfo where username= zhangsan
### Cause: java.sql.SQLSyntaxErrorException: Unknown column 'zhangsan' in 'where clause'
```

可以看到, 这次的参数依然是直接拼接在SQL语句中了, 但是字符串作为参数时, 需要添加引号 `''` , 使用 `${}` 不会拼接引号 `''` , 导致程序报错.

修改代码如下:

```java
@Select("select username, `password`, age, gender, phone from user_info where
username= '${name}' ")
UserInfo queryByName(String name);
```

再次运行, 结果正常返回

```
JDBC Connection [HikariProxyConnection@1768471788 wrapping com.mysql.cj.jdbc.ConnectionImpl@5568c66f] will not be managed by Spring
==>  Preparing: select username, `password`, age, gender, phone from userinfo where username= 'zhangsan'
==> Parameters:
<==    Columns: username, password, age, gender, phone
<==        Row: zhangsan, zhangsan, 18, 1, 18612340002
<==      Total: 1
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@61a91c9b]
UserInfo(id=null, username=zhangsan, password=zhangsan, age=18, gender=1, phone=18612340002, deleteFlag=null, createTime=null,
 updateTime=null)
```

从上面两个例子可以看出:

`#{}` 使用的是预编译SQL, 通过 `?` 占位的方式, 提前对SQL进行编译, 然后把参数填充到SQL语句中. `#{}` 会根据参数类型, 自动拼接引号 `''` .

`${}` 会直接进行字符替换, 一起对SQL进行编译. 如果参数为字符串, 需要加上引号 `''` .

> 参数为数字类型时, 也可以加上, 查询结果不变, 但是可能会导致索引失效, 性能下降.

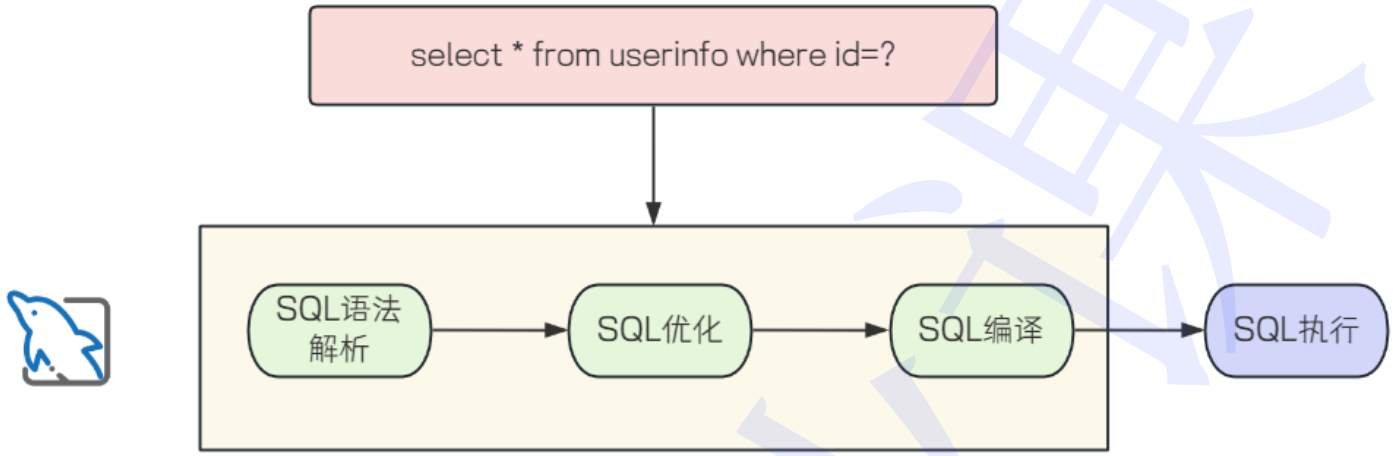
5.2.2 #{} 和 ${}区别

#{} 和 ${} 的区别就是**预编译SQL和即时SQL 的区别.**

> 简单回顾:
>
> 当客户发送一条SQL语句给服务器后, 大致流程如下:
>
> 1. 解析语法和语义, 校验SQL语句是否正确
> 2. 优化SQL语句, 制定执行计划
> 3. 执行并返回结果
>
> 一条 SQL如果走上述流程处理, 我们称之为 Immediate Statements(即时 SQL)

1. 性能更高

绝大多数情况下, 某一条 SQL 语句可能会被反复调用执行, 或者每次执行的时候只有个别的值不同(比如 select 的 where 子句值不同, update 的 set 子句值不同, insert 的 values 值不同). 如果每次都需要经过上面的语法解析, SQL优化、SQL编译等，则效率就明显不行了.



预编译SQL，编译一次之后会将编译后的SQL语句缓存起来，后面再次执行这条语句时，不会再次编译(只是输入的参数不同), 省去了解析优化等过程, 以此来提高效率

2. 更安全(防止SQL注入)

SQL注入：是通过操作输入的数据来修改事先定义好的SQL语句，以达到执行代码对服务器进行攻击的方法。

> 由于没有对用户输入进行充分检查，而SQL又是拼接而成，在用户输入参数时，在参数中添加一些SQL关键字，达到改变SQL运行结果的目的，也可以完成恶意攻击。

sql 注入代码：`' or 1='1`

先来看看SQL注入的例子

```
@Select("select username, `password`, age, gender, phone from user_info where
username= '${name}' ")
List<UserInfo> queryByName(String name);
```

测试代码:

正常访问情况:

```
@Test
void queryByName() {
    List<UserInfo> userInfos = userInfoMapper.queryByName("admin");
    System.out.println(userInfos);
}
```

结果运行正常

```
JDBC Connection [HikariProxyConnection@753853622 wrapping com.mysql.cj.jdbc.ConnectionImpl@12a470dd] will not be managed by Spring
==>  Preparing: select username, `password`, age, gender, phone from userinfo where username= 'admin'
==> Parameters:
<==    Columns: username, password, age, gender, phone
<==        Row: admin, admin, 18, 1, 18612340001
<==      Total: 1
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@404eca05]
[UserInfo(id=null, username=admin, password=admin, age=18, gender=1, phone=18612340001, deleteFlag=null, createTime=null,
 updateTime=null)]
```

SQL注入场景:

```java
@Test
void queryByName() {
    List<UserInfo> userInfos = userInfoMapper.queryByName("' or 1='1");
    System.out.println(userInfos);
}
```

结果依然被正确查询出来了, 其中参数 or被当做了SQL语句的一部分

```
JDBC Connection [HikariProxyConnection@1768471788 wrapping com.mysql.cj.jdbc.ConnectionImpl@5568c66f] will not be managed by Spring
==>  Preparing: select username, `password`, age, gender, phone from userinfo where username= '' or 1='1'
==> Parameters:
<==    Columns: username, password, age, gender, phone
<==        Row: admin, admin, 18, 1, 18612340001
<==        Row: zhangsan, zhangsan, 18, 1, 18612340002
<==        Row: lisi, lisi, 18, 1, 18612340003
<==        Row: wangwu, wangwu, 18, 1, 18612340004
<==        Row: zhaoliu, zhaoliu, 21, 2, 18612340005
<==        Row: zhaoliu, zhaoliu, 21, 2, 18612340005
<==        Row: zhaoliu, zhaoliu, 21, 2, 18612340005
<==      Total: 7
```

可以看出来, 查询的数据并不是自己想要的数据. 所以用于查询的字段，尽量使用 `#{}` 预查询的方式

> SQL注入是一种非常常见的数据库攻击手段, SQL注入漏洞也是网络世界中最普遍的漏洞之一.
>
> 如果发生在用户登录的场景中, 密码输入为 `' or 1='1` , 就可能完成登录(不是一定会发生的场景, 需要看登录代码如何写)
>
> 控制层: UserController

```java
import com.example.demo.model.UserInfo;
import com.example.demo.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

```

```java
@RestController
public class UserController {
    @Autowired
    private UserService userService;

    @RequestMapping("/login")
    public boolean login(String name, String password) {
        UserInfo userInfo = userService.queryUserByPassword(name, password);
        if (userInfo != null) {
            return true;
        }
        return false;
    }
}
```

业务层: UserService

```java
import com.example.demo.mapper.UserInfoMapper;
import com.example.demo.model.UserInfo;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class UserService {
    @Autowired
    private UserInfoMapper userInfoMapper;

    public UserInfo queryUserByPassword(String name, String password) {
        List<UserInfo> userInfos = userInfoMapper.queryUserByPassword(name,
password);
        if (userInfos != null && userInfos.size() > 0) {
            return userInfos.get(0);
        }
        return null;
    }
}
```

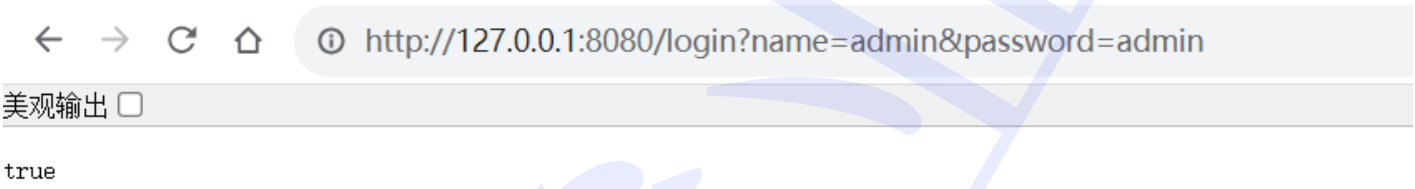
数据层: UserInfoMapper

```java
import com.example.demo.model.UserInfo;
import org.apache.ibatis.annotations.*;
```

```
3
4  import java.util.List;
5
6  @Mapper
7  public interface UserInfoMapper {
8      @Select("select username, `password`, age, gender, phone from user_info
   where username= '${name}' and password='${password}' ")
9      List<UserInfo> queryUserByPassword(String name, String password);
10 }
```

启动服务, 访问: http://127.0.0.1:8080/login?name=admin&password=admin

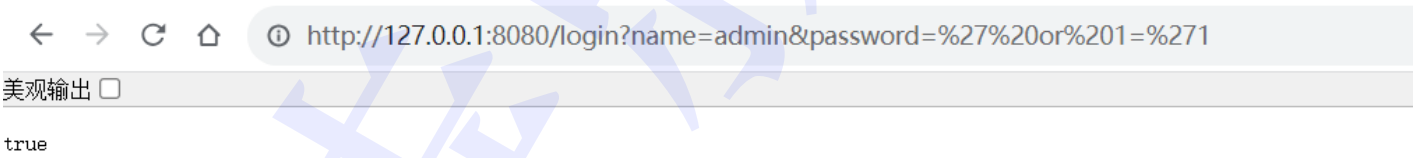
程序正常运行



接下来访问SQL注入的代码:

password 设置为 `' or 1='1`

http://127.0.0.1:8080/login?name=admin&password=%27%20or%201=%271



5.3 排序功能

从上面的例子中, 可以得出结论: **${} 会有SQL注入的风险, 所以我们尽量使用#{}完成查询**

既然如此, 是不是 ${} 就没有存在的必要性了呢?

当然不是.

接下来我们看下${}的使用场景

Mapper实现

```
1 @Select("select id, username, age, gender, phone, delete_flag, create_time,
  update_time " +
2         "from user_info order by id ${sort} ")
3 List<UserInfo> queryAllUserBySort(String sort);
```

使用 `${sort}` 可以实现排序查询, 而使用 `#{sort}` 就不能实现排序查询了.

> 注意: 此处 sort 参数为String类型, 但是SQL语句中, 排序规则是不需要加引号 `''` 的, 所以此时的 `${sort}` 也不加引号

我们把 `${}` 改成 `#{}`

```
1 @Select("select id, username, age, gender, phone, delete_flag, create_time,
  update_time " +
2         "from user_info order by id #{sort} ")
3 List<UserInfo> queryAllUserBySort(String sort);
```

运行结果:

```
JDBC Connection [HikariProxyConnection@841721161 wrapping com.mysql.cj.jdbc.ConnectionImpl@502a4156] will not be managed by Spring
==>  Preparing: select id, username, age, gender, phone, delete_flag, create_time, update_time from userinfo order by id ?
==> Parameters: asc(String)
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@4483d35]

org.springframework.jdbc.BadSqlGrammarException:
### Error querying database.  Cause: java.sql.SQLSyntaxErrorException: You have an error in your SQL syntax; check the manual that
 corresponds to your MySQL server version for the right syntax to use near ''asc'' at line 1
### The error may exist in com/example/demo/mapper/UserInfoMapper1.java (best guess)
### The error may involve com.example.demo.mapper.UserInfoMapper1.queryAllUserBySort-Inline
### The error occurred while setting parameters
### SQL: select id, username, age, gender, phone, delete_flag, create_time, update_time from userinfo order by id ?
### Cause: java.sql.SQLSyntaxErrorException: You have an error in your SQL syntax; check the manual that corresponds to your MySQL
 server version for the right syntax to use near ''asc'' at line 1
; bad SQL grammar []; nested exception is java.sql.SQLSyntaxErrorException: You have an error in your SQL syntax; check the manual
 that corresponds to your MySQL server version for the right syntax to use near ''asc'' at line 1
```

可以发现, 当使用 `#{sort}` 查询时, asc 前后自动给加了引号, 导致 sql 错误

> `#{}` 会根据参数类型判断是否拼接引号 `''`
>
> 如果参数类型为String, 就会加上 引号.

除此之外, 还有表名作为参数时, 也只能使用 `${}`

5.4 like 查询

like 使用 #{} 报错

```
1 @Select("select id, username, age, gender, phone, delete_flag, create_time,
  update_time " +
2         "from user_info where username like '%#{key}%' ")
3 List<UserInfo> queryAllUserByLike(String key);
```

把 #{} 改成 ${} 可以正确查出来, 但是${}存在SQL注入的问题, 所以不能直接使用 ${}.

解决办法: 使用 mysql 的内置函数 concat() 来处理，实现代码如下：

```
1 @Select("select id, username, age, gender, phone, delete_flag, create_time,
  update_time " +
2         "from user_info where username like concat('%',#{key},'%')")
3 List<UserInfo> queryAllUserByLike(String key);
```

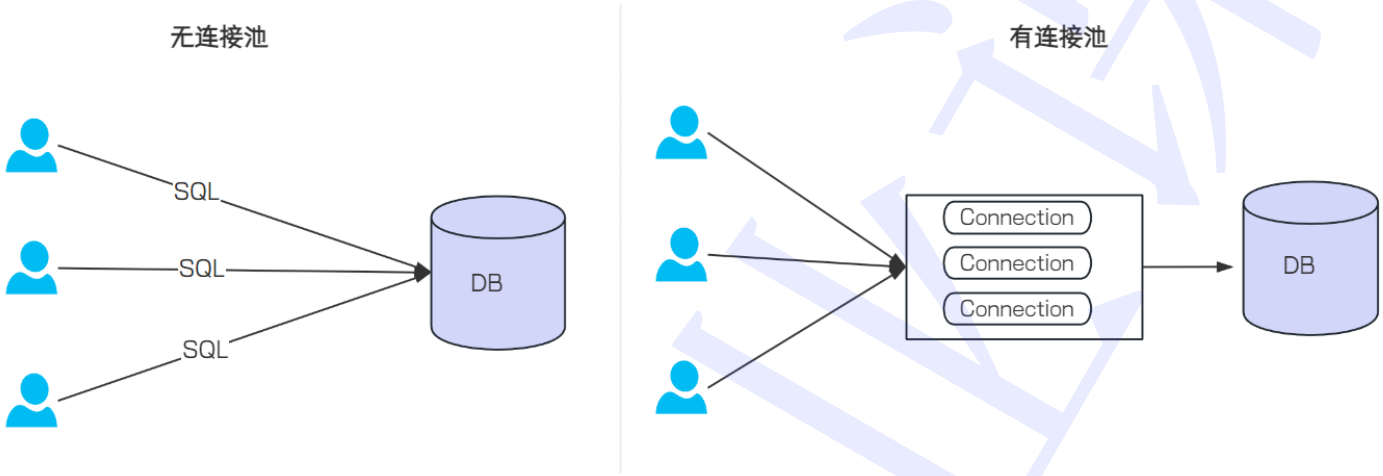
6. 数据库连接池

在上面Mybatis的讲解中, 我们使用了数据库连接池技术, 避免频繁的创建连接, 销毁连接

下面我们来了解下数据库连接池

6.1 介绍

数据库连接池负责分配、管理和释放数据库连接，它允许应用程序重复使用一个现有的数据库连接，而不是再重新建立一个.



没有使用数据库连接池的情况: 每次执行SQL语句, 要先创建一个新的连接对象, 然后执行SQL语句, SQL语句执行完, 再关闭连接对象释放资源. 这种重复的创建连接, 销毁连接比较消耗资源

使用数据库连接池的情况: 程序启动时, 会在数据库连接池中创建一定数量的Connection对象, 当客户请求数据库连接池, 会从数据库连接池中获取Connection对象, 然后执行SQL, SQL语句执行完, 再把Connection归还给连接池.

优点:

1. 减少了网络开销
2. 资源重用
3. 提升了系统的性能

6.2 使用

常见的数据库连接池:

- C3P0
- DBCP
- Druid
- Hikari

目前比较流行的是 Hikari, Druid

1. Hikari : SpringBoot默认使用的数据库连接池

```
2023-08-18 18:59:51.834  INFO 54568 --- [           main] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Starting...
2023-08-18 18:59:51.974  INFO 54568 --- [           main] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Start completed.
[UserInfo(id=null, username=admin, age=18, gender=1, phone=18612340001, status=null, createtime=null, updatetime=null), UserInfo
 (id=null, username=zhangsan, age=18, gender=1, phone=18612340002, status=null, createtime=null, updatetime=null), UserInfo(id=null,
  username=lisi, age=18, gender=1, phone=18612340003, status=null, createtime=null, updatetime=null), UserInfo(id=null,
 username=wangwu, age=18, gender=1, phone=18612340004, status=null, createtime=null, updatetime=null)]
```

> Hikari 是日语"光"的意思(ひかり), Hikari也是以追求性能极致为目标

2. Druid

如果我们想把默认的数据库连接池切换为Druid数据库连接池, 只需要引入相关依赖即可

```xml
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid-spring-boot-3-starter</artifactId>
    <version>1.2.21</version>
</dependency>
```

如果SpringBoot版本为2.X, 使用druid-spring-boot-starter 依赖

```xml
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid-spring-boot-starter</artifactId>
    <version>1.1.17</version>
</dependency>
```

运行结果:

```
2023-08-19 11:37:15.733  INFO 64692 --- [           main] c.a.d.s.b.a.DruidDataSourceAutoConfigure : Init DruidDataSource
2023-08-19 11:37:16.029  INFO 64692 --- [           main] com.alibaba.druid.pool.DruidDataSource   : {dataSource-1} inited
2023-08-19 11:37:16.301  INFO 64692 --- [           main] .d.SpringbootMybatisDemoApplicationTests : Started
SpringbootMybatisDemoApplicationTests in 2.381 seconds (JVM running for 3.319)
[UserInfo(id=null, username=admin, age=18, gender=1, phone=18612340001, status=null, createtime=null, updatetime=null), UserInfo
 (id=null, username=zhangsan, age=18, gender=1, phone=18612340002, status=null, createtime=null, updatetime=null), UserInfo(id=null,
  username=lisi, age=18, gender=1, phone=18612340003, status=null, createtime=null, updatetime=null), UserInfo(id=null,
 username=wangwu, age=18, gender=1, phone=18612340004, status=null, createtime=null, updatetime=null)]
2023-08-19 11:37:16.741  INFO 64692 --- [ionShutdownHook] com.alibaba.druid.pool.DruidDataSource   : {dataSource-1} closing ...
2023-08-19 11:37:16.750  INFO 64692 --- [ionShutdownHook] com.alibaba.druid.pool.DruidDataSource   : {dataSource-1} closed
```

> 参考官方地址：https://github.com/alibaba/druid/tree/master/druid-spring-boot-starter
>
> - Druid连接池是阿里巴巴开源的数据库连接池项目
> - 功能强大，性能优秀，是Java语言最好的数据库连接池之一

> - 学习文档: https://github.com/alibaba/druid/wiki/%E9%A6%96%E9%A1%B5

二者对比,参考: Hikaricp和Druid对比_数据库_晚风暖-华为云开发者联盟

7. 总结

7.1 MySQL 开发企业规范

1. 表名, 字段名使用小写字母或数字, 单词之间以下划线分割. 尽量避免出现数字开头或者两个下划线中间只出现数字. 数据库字段名的修改代价很大, 所以字段名称需要慎重考虑。

> MySQL 在 Windows 下不区分大小写, 但在 Linux 下默认是区分大小写. 因此, 数据库名, 表名, 字段名都不允许出现任何大写字母, 避免节外生枝
>
> 正例： aliyun_admin， rdc_config， level3_name
>
> 反例： AliyunAdmin， rdcConfig， level_3_name

2. 表必备三字段: id, create_time, update_time

> id 必为主键, 类型为 bigint unsigned, 单表时自增, 步长为 1
>
> create_time, update_time 的类型均为 datetime 类型, create_time表示创建时间, update_time表示更新时间
>
> ***有同等含义的字段即可, 字段名不做强制要求***

3. 在表查询中, 避免使用 * 作为查询的字段列表, 标明需要哪些字段(课堂上给大家演示除外).

> 1. 增加查询分析器解析成本
> 2. 增减字段容易与 resultMap 配置不一致
> 3. 无用字段增加网络消耗, 尤其是 text 类型的字段

7.2 #{} 和${} 区别

1. #{}：预编译处理, ${}：字符直接替换
2. #{} 可以防止SQL注入, ${}存在SQL注入的风险, 查询语句中, 可以使用 #{} ,推荐使用 #{}
3. 但是一些场景, #{} 不能完成, 比如 排序功能, 表名, 字段名作为参数时, 这些情况需要使用${}
4. 模糊查询虽然${}可以完成, 但因为存在SQL注入的问题,所以通常使用mysql内置函数concat来完成