Mybatis

学习目标

1. 能够理解三层架构及框架
2. 能够配置mybatis的环境
3. 能够使用mybatis的基本增删改查
4. 能够使用mybatis的条件查询
5. 能够使用mybatis模糊查询
6. 能够完成主键返回

一、框架

1.1 三层架构

软件开发常用的架构是三层架构，之所以流行是因为有着清晰的任务划分。一般包括以下三层：

持久层：主要完成与数据库相关的操作，即对数据库的增删改查。

因为数据库访问的对象一般称为Data Access Object（简称DAO），所以有人把持久层叫做DAO层。

业务层：主要根据功能需求完成业务逻辑的定义和实现。

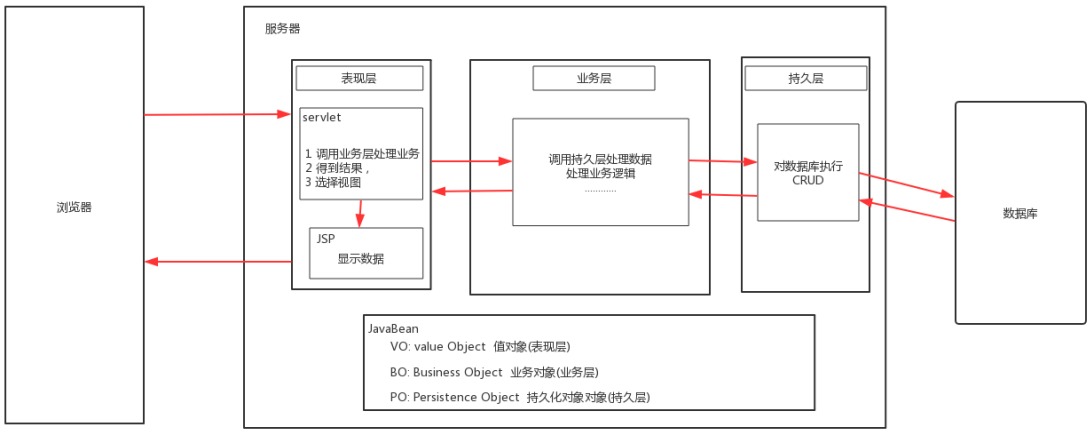
因为它主要是为上层提供服务的，所以有人把业务层叫做Service层或Business层。

表现层：主要完成与最终软件使用用户的交互，需要有交互界面（UI）。

因此，有人把表现层称之为web层或View层。

三层架构之间调用关系为:表现层调用业务层，业务层调用持久层。

各层之间必然要进行数据交互，我们一般使用java实体对象来传递数据。



1.2 框架

1.2.1 什么是框架?

框架（Framework）是整个或部分系统的可重用设计，表现为一组抽象构件及构件实例间交互的方法;另一种定义认为，框架是可被应用开发者定制的应用骨架。前者是从应用方面而后者是从目的方面给出的定义。

简而言之，框架其实就是某种应用的半成品，就是一组组件，供你选用完成你自己的系统。简单说就是使用别人搭好的舞台，你来做表演。而且，框架一般是成熟的，不断升级的软件。

1.2.2 框架要解决的问题

框架要解决的最重要的一个问题是技术整合的问题，在 J2EE 的 框架中，有着各种各样的技术，不同的软件企业需要从 J2EE 中选择不同的技术，这就使得软件企业最终的应用依赖于这些技术，技术自身的复杂性和技术的风险性将会直接对应用造成冲击。而应用是软件企业的核心，是竞争力的关键所在，因此应该将应用自身的设计和具体的实现技术解耦。这样，软件企业的研发将集中在应用的设计上，而不是具体的技术实现，技术实现是应用的底层支撑，它不应该直接对应用产生影响。

1.2.3 常见的框架

Java世界中的框架非常的多，每一个框架都是为了解决某一部分或某些问题而存在的。

下面列出在目前企业中流行的几种框架（一定要注意他们是用来解决哪一层问题的）：

持久层框架：专注于解决数据持久化的框架。常用的有mybatis、hibernate、spring jdbc等等。

表现层框架：专注于解决与用户交互的框架。常见的有struts2、spring mvc等等。

全栈框架:　　能在各层都给出解决方案的框架。比较著名的就是spring。

```
这么多框架，我们怎么选择呢?
    我们以企业中最常用的组合为准来学习Spring  +  Spring MVC  +  mybatis（SSM）
```

二、Mybatis 概述

2.1 ORM概述

```
ORM（object  Relational Mapping）对象关系映射,是一个针对持久层的理论思想。
    O----对象----类
    R----关系----数据表
    M----映射----在类和数据表之间建立的一一对应的关系（类名-->表名    属性名-->字段名）

ORM用来解决什么问题呢?
    一句话说，就是ORM可以让我们以面向对象的形式操作数据库

总结:ORM就是建立实体类和数据库表之间的关系，从而达到操作实体类就相当于操作数据库表的目的。
```

```
常见的ORM框架有哪些?
    Hibernate
    JPA（SUN公司的规范，只有接口名，没有实现）
    Mybatis(半ORM框架，让我们既可以使用ORM的思想，又不受ORM的严格约束，表现为可以手动书写SQL)
```

2.2 Mybatis介绍

历史

- MyBatis本是apache的一个开源项目，名为iBatis。
- 2010年这个项目由apache迁移到了google，并且改名为MyBatis。
- 2013年迁移到Github。

简介

- MyBatis是一款优秀的持久层框架，它不需要像JDBC那样去写复杂代码、手动设置参数、繁琐的处理结果集
- 它采用简单的XML配置 + 接口方法的形式实现对数据库的增删改查，使得让程序员只关注sql本身

总结

也就是说使用了Mybatis以后，程序员要做的事情变成了这样:

1. 在接口中定义方法
2. 在xml中书写SQL语句

三、Mybatis入门案例

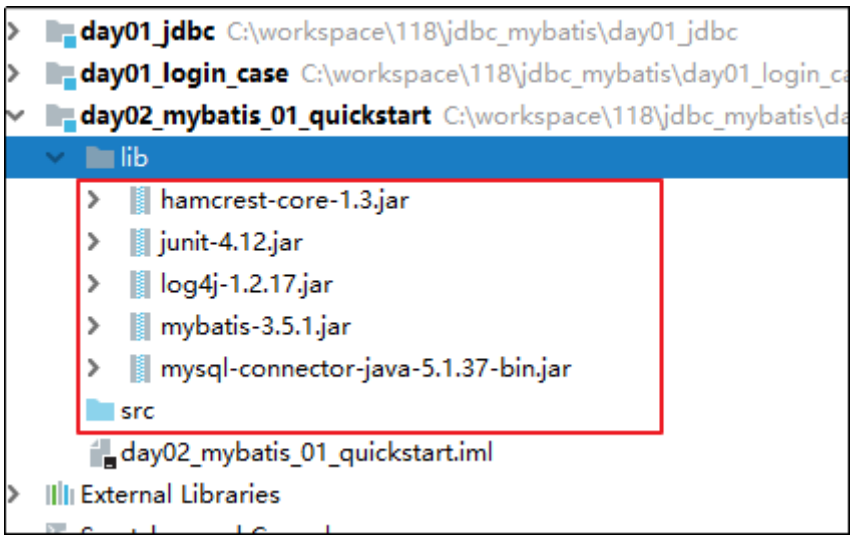
目标：使用一个user对象的保存操作来体验mybatis的使用

3.1 创建数据库 数据表

```
CREATE TABLE USER(
      uid INT PRIMARY KEY AUTO_INCREMENT,
      name VARCHAR(100) NOT NULL,
      password VARCHAR(50) NOT NULL,
      email VARCHAR(50),
      birthday DATE
);
```

3.2 创建工程，引入jar包

我们要导入的jar包有: mysql mybatis junit log4j



3.3 创建实体类

```
//创建User表
public class User {
    private Integer uid;
    private String name;
    private String password;
    private String email;
    private Date birthday;
    //省略set和get方法...
}
```

3.4 加入Mapper映射文件

主配置文件可以在官网下载: https://mybatis.org/mybatis-3/zh/getting-started.html

在src下添加一个UserMapper.xml, 内容如下:

```
<?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">
<!--namespace命名空间 可以认为是这个配置文件的标识-->
<mapper namespace="UserMapper">
    <!--
        保存的sql语句,mybatis把这段称为一个statement
        insert  标签表示新增
        id sql语句的标识,也称为statementId
        parameterType  传入的参数类型
    -->
    <insert id="save" parameterType="com.itheima.domain.User">
        insert into user values (
            null,#{name},#{password},#{email},#{birthday}
        )
    </insert>
</mapper>
```

3.5 加入mybatis的主配置文件

主配置文件可以在官网下载: https://mybatis.org/mybatis-3/zh/getting-started.html

在src下添加一个mybatis-config.xml, 内容如下:

```
<?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>

    <!--数据环境配置-->
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.jdbc.Driver"/>
                <property name="url" value="jdbc:mysql:///mybatis_118"/>
                <property name="username" value="root"/>
                <property name="password" value="adminadmin"/>
            </dataSource>
        </environment>
    </environments>

    <!--引入映射文件(sql)-->
    <mappers>
        <mapper resource="UserMapper.xml"/>
    </mappers>
</configuration>
```

3.6 加入日志配置文件

在src下添加log4j.properties,内容如下:

```
### 设置###
log4j.rootLogger = debug,stdout

### 输出信息到控制抬 ###
log4j.appender.stdout = org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target = System.out
log4j.appender.stdout.layout = org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern = [%-5p] %d{yyyy-MM-dd
HH:mm:ss,SSS} method:%l%n%m%n
```

3.7 测试

```
public class MybatisTest {

    //使用一个user对象的保存操作来体验mybatis的使用
    @Test
    public void testSave() throws IOException {
        //准备一个User对象
        User user = new User();
        user.setName("传智播客1");
        user.setPassword("admin");
        user.setEmail("admin1@itcast.cn");
        user.setBirthday(new Date());

        //1 读取配置文件,读成流的形式
        InputStream stream = Resources.getResourceAsStream("mybatis-
config.xml");

        //2 创建sqlSessionFactory
        SqlSessionFactory factory = new
SqlSessionFactoryBuilder().build(stream);

        //3 获取sqlSession[这是mybatis的重要API, 使用它可以操作数据库crud]
        SqlSession sqlSession = factory.openSession();

        //4 执行保存操作
        //第一参数:指的是sql语句的位置,使用namespace.statementId表示
        //第二参数:指的是sql语句需要的参数
        sqlSession.insert("UserMapper.save",user);

        //5 提交事务(mybatis默认是不自动提交事务的)
        sqlSession.commit();

        //6 释放资源
        sqlSession.close();
    }
}
```

四、基于传统方式实现dao层

4.1 添加dao层接口

```
public interface UserDao {
    //保存
    void save(User user);
}
```

4.2 添加dao层实现类

```
public class UserDaoImpl implements UserDao {
    @Override
    public void save(User user) {
        try {
            //1 读取配置文件,读成流的形式
            InputStream stream = Resources.getResourceAsStream("mybatis-
config.xml");

            //2 创建sqlSessionFactory
            SqlSessionFactory factory = new
SqlSessionFactoryBuilder().build(stream);

            //3 获取sqlSession[这是mybatis的重要API, 使用它可以操作数据库crud]
            SqlSession sqlSession = factory.openSession();

            //4 执行保存操作
            //第一参数:指的是sql语句的位置,使用namespace.statementId表示
            //第二参数:指的是sql语句需要的参数
            sqlSession.insert("UserMapper.save",user);

            //5 提交事务(mybatis默认是不自动提交事务的)
            sqlSession.commit();

            //6 释放资源
            sqlSession.close();
        }catch (Exception e){
            e.printStackTrace();
        }
    }
}
```

4.3 测试

```
public class MybatisTest {

    //使用一个user对象的保存操作来体验mybatis的使用
    @Test
    public void testSave() throws IOException {
        //准备一个User对象
        User user = new User();
        user.setName("传智播客2");
        user.setPassword("admin2");
        user.setEmail("admin2@itcast.cn");
        user.setBirthday(new Date());

        UserDao userDao = new UserDaoImpl();
        userDao.save(user);
    }
```

```
}
```

五、基于接口代理实现dao层

采用 Mybatis 的代理开发方式实现 DAO 层的开发，这种方式是我们后面进入企业的主流方式，所以我们在前面的入门程序实现的 CRUD 就是采用这种方式来做的。现在一起再使用这种方式来开发我们的 DAO。

代理方式开发 DAO 规范总结：

Mapper 接口开发方法只需要程序员编写 Mapper 接口（相当于 Dao 接口），由 Mybatis 框架根据接口定义创建接口的动态代理对象，代理对象的方法体同上边 Dao 接口实现类方法。

Mapper 接口开发需要遵循以下规范：

1、 Mapper.xml 文件中的 namespace 与 mapper 接口的类路径相同。

2、 Mapper 接口方法名和 Mapper.xml 中定义的每个 statement 的 id 相同

3、 Mapper 接口方法的输入参数类型和 mapper.xml 中定义的每个 sql 的 parameterType 的类型相同

4、 Mapper 接口方法的输出参数类型和 mapper.xml 中定义的每个 sql 的 resultType 的类型相同

5.1 创建dao层接口

删除上面案例中的实现类

```
public interface UserDao {
    //保存
    void save(User user);
}
```

5.2 修改Mapper映射文件

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.itheima.dao.UserDao">
    <select id="save" parameterType="com.itheima.domain.User">
        insert into user values (
            null,#{name},#{password},#{email},#{birthday}
        )
    </select>
</mapper>
```

5.3 测试

```
public class MybatisTest {

    //使用一个user对象的保存操作来体验mybatis的使用
    @Test
    public void testSave() throws IOException {
```

```
        //准备一个User对象
        User user = new User();
        user.setName("传智播客2");
        user.setPassword("admin2");
        user.setEmail("admin2@itcast.cn");
        user.setBirthday(new Date());

        //1 读取配置文件,读成流的形式
        InputStream stream = Resources.getResourceAsStream("mybatis-
config.xml");

        //2 创建sqlSessionFactory
        SqlSessionFactory factory = new
SqlSessionFactoryBuilder().build(stream);

        //3 获取sqlSession[这是mybatis的重要API, 使用它可以操作数据库crud]
        SqlSession sqlSession = factory.openSession();

        //4 通过Mybatis代理产生userdao对象
        UserDao userDao = sqlSession.getMapper(UserDao.class);
        userDao.save(user);

        //5 提交事务(mybatis默认是不自动提交事务的)
        sqlSession.commit();

        //6 释放资源
        sqlSession.close();
    }
}
```

六、Mybatis的API介绍和原理分析

6.1 API介绍

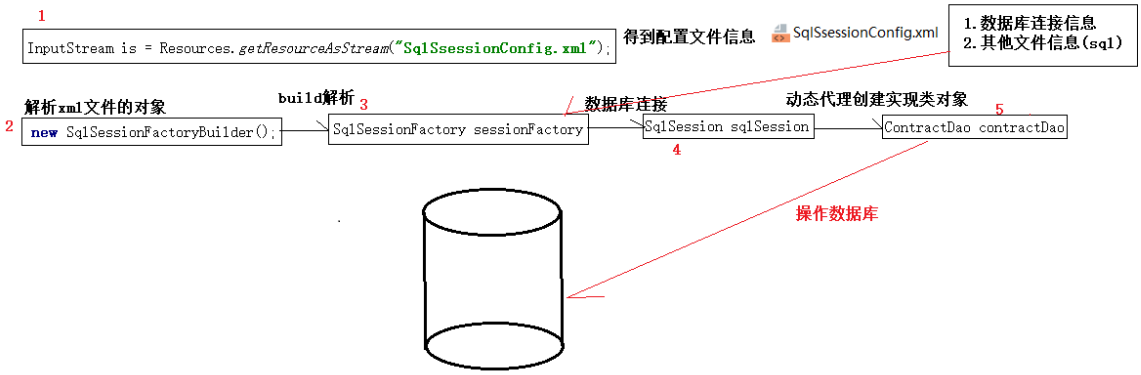
```
Resources
    加载mybatis的配置文件

SqlSessionFactoryBuilder
    利用Resources指定的资源，将配置信息加载到内存中，还会加载mybatis配置文件中指定的所有映射
配置信息，      并用特定的对象实例进行保存，从而创建SqlSessionFactory对象

SqlSessionFactory
    这是一个工厂对象，对于这种创建和销毁都非常耗费资源的重量级对象，一个项目中只需要存在一个即可
    也就是说，它的生命周期跟项目的生命周期是一致的
    它的任务是创建SqlSession

SqlSession
    这是Mybatis的一个核心对象。我们基于这个对象可以实现对数据的CRUD操作
    对于这个对象应做到每个线程独有，每次用时打开，用完关闭
```

6.2 Mybatis执行过程



1. 通过Resources读取主配置文件，里面含有数据库的连接信息以及sql相关信息
2. 通过SqlSessionFactoryBuilder 对象加载流信息，解析后获得SqlSessionFactory工厂对象
3. SqlSessionFactory工厂对象能够帮助我们生成SqlSession会话连接对象
4. SqlSession会话对象可以帮助我们构建接口的动态代理对象
5. 通过动态代理对象，我们可以操作数据库

七、抽取工具类

7.1 抽取基本代码工具类

```
public class MybatisUtil {
    private static SqlSessionFactory sqlSessionFactory = null;

    static {
        try {
            InputStream inputStream = Resources.getResourceAsStream("mybatis-
config.xml");
            sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);
        } catch (Exception e) {
            throw new RuntimeException("加载配置文件失败");
        }
    }

    public static SqlSession openSession() {
        return sqlSessionFactory.openSession();
    }
}
```

7.2 抽取测试基类

```
public class BaseMapperUtil {
    protected SqlSession sqlSession = null;

    @Before
    public void init() {
        sqlSession = MybatisUtil.openSession();
    }

    @After
    public void destory() {
        sqlSession.commit();
        sqlSession.close();
```

```
    }
}
```

7.3 测试

```
//注意: 这里直接继承BaseUtil,从中获取sqlSession
public class MyBatisTest extends BaseUtil {
    //使用一个user对象的保存操作来体验mybatis的使用
    @Test
    public void testSave() throws IOException {

        //创建user对象
        User user = new User();
        user.setName("传智播客");
        user.setEmail("admin@itcast.cn");
        user.setPassword("admin");
        user.setBirthday(new Date());

        //4 执行操作
        UserDao userDao = sqlSession.getMapper(UserDao.class);
        userDao.save(user);
    }
}
```

八、完成增删改操作

8.1 增加

需求: 向数据库user表新增一条记录

8.1.1 在 UserMapper 类中添加新增方法

我们在入门案例中已经完成了用户新增的功能，所以不用再做，仅仅在这里整理一下思路即可。

```
public interface UserDao {
    //保存
    void save(User user);
}
```

8.1.2 在 UserMapper.xml 文件中加入新增配置

```
    <!--
        insert : 标签表示新增
        id : 表示方法的名称，一定要与接口的方法名称对应
        parameterType : 请求参数类型
    -->
    <insert id="save" parameterType="com.itheima.domain.User">
        insert into user(name,password,email,birthday)
        values(#{name},#{password},#{email},#{birthday})
    </insert>
```

我们可以发现，这个 sql 语句中使用#{}字符，#{}代表占位符，我们可以理解是原来 jdbc 部分所学的?，它们都是代表占位符， 具体的值是由 User 类的 username 属性来决定的。 parameterType 属性：代表参数的类型，因为我们要传入的是一个类的对象，所以类型就写类的全名称。

8.1.3 在测试类中的测试方法

```
@Test
public void testSave() throws IOException {

    //创建user对象
    User user = new User();
    user.setName("传智播客");
    user.setEmail("admin@itcast.cn");
    user.setPassword("admin");
    user.setBirthday(new Date());

    //执行操作
    UserDao userDao = sqlSession.getMapper(UserDao.class);
    userDao.save(user);
}
```

测试结果如下：

```
==>  Preparing: insert into user(name,password,email,birthday) values(?,?,?,?)
[DEBUG] 2019-12-14 18:34:00,023 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debug(BaseJdbcLogger.j
==> Parameters: 传智播客(String), admin(String), admin@itcast.cn(String), 2019-12-14 18:33:58.773(Timestamp)
[DEBUG] 2019-12-14 18:34:00,066 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debug(BaseJdbcLogger.j
<==    Updates: 1
```

8.2 修改

需求: 根据user的主键修改其他属性

8.2.1 在 UserMapper 类中添加更新方法

在UserMapper.java类中添加一个用于 update()的方法,如下：

```
//修改
void update(User user);
```

8.2.2 在 UerMapper.xml文件中加入更新操作配置

在 UserMapper.xml 文件中加入更新用户的配置，如下：

```
<!--按照id 修改其他的-->
<!--
    update : 标签表示修改
    id : 表示方法的名称，一定要与接口的方法名称对应
    parameterType : 请求参数类型
-->
<update id="update" parameterType="com.itheima.domain.User">
    update user set
        name = #{name},
        password = #{password},
        email = #{email},
        birthday = #{birthday}
    where uid = #{uid}
</update>
```

8.2.3 测试

为了更好的实现测试效果，我们加入更新的测试方法 testUpdateUser()方法，具体实现代码如下：

```java
//修改
@Test
public void testUpdate() throws IOException {
    //创建user对象
    User user = new User();
    user.setName("传智播客1");
    user.setEmail("admin@itcast1.cn");
    user.setPassword("admin1");
    user.setBirthday(new Date());
    user.setUid(1);

    //执行操作
    UserDao userDao = sqlSession.getMapper(UserDao.class);
    userDao.update(user);
}
```

测试后，控制台输出结果如下：

```
==>  Preparing: update user set name = ?, password = ?, email = ?, birthday = ? where uid = ?
[DEBUG] 2019-12-14 18:39:17,146 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debug(BaseJdbc
==> Parameters: 传智播客1(String), admin1(String), admin@itcast1.cn(String), 2019-12-14 18:39:16.718
[DEBUG] 2019-12-14 18:39:17,203 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debug(BaseJdbc
<==    Updates: 1
```

8.3 删除

需求:根据主键删除一个用户

8.3.1 在 UserMapper 类中加入删除方法

在 UserMapper 类中加入删除 delete()方法，用于实现用户的删除。

```java
//删除 按照主键删除
void delete(Integer uid);
```

8.3.2 在 UserMapper.xml 文件中加入删除操作

加入的删除的映射配置信息如下：

```xml
<!--
    delete : 标签表示删除
    id : 表示方法的名称，一定要与接口的方法名称对应
    parameterType : 请求参数类型
    #{} 代表占位符,相当于JDBC中的?
    当传入的参数是一个JavaBean 的时候,#{}中填写的是JavaBean的属性名
    当传入的参数是一个简单类型, #{} 中可以随便写 但是不推荐,推荐写成方法的形参
-->
<delete id="delete" parameterType="java.lang.Integer">
    delete from user where uid = #{随便写}
</delete>
```

8.3.3 加入删除的测试方法

在原有的 UserTest 类中加入测试方法，如下：

```
    //删除
    @Test
    public void testDelete() throws IOException {
        UserDao userDao = sqlSession.getMapper(UserDao.class);
        userDao.delete(1);
    }
```

下面是测试的结果：

```
==>  Preparing: delete from user where uid = ?
[DEBUG] 2019-12-14 18:46:23,982 method:org.apache.ibatis.loggir
==> Parameters: 4(Integer)
[DEBUG] 2019-12-14 18:46:24,003 method:org.apache.ibatis.loggir
<==    Updates: 1
```

九、查询

9.1 ResultType的使用

当数据库返回的结果集中的字段和实体类中的属性名一一对应时, resultType可以自动将结果封装到实体中

9.1.1 在UserMapper类中加入方法

在UserMapper.java类中添加一个用于 findAll()的方法,如下：

```
    //查询所有
    List<User> findAll();
```

9.1.2 在 UserMapper.xml 文件中加入配置

在 UserMapper.xml 文件中加入 findAll的配置，如下：

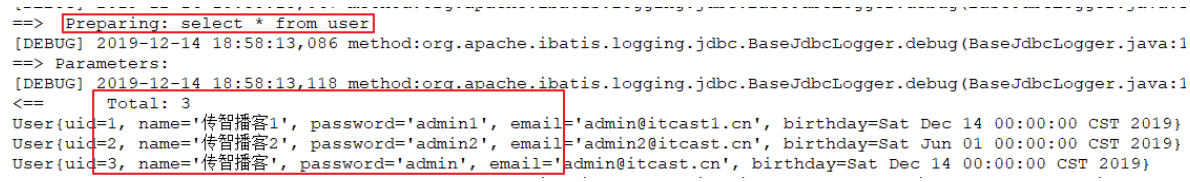
```
    <!--查询所有-->
    <!--
       当方法返回值类型为一个集合时   resultType中写的是集合中实体的类型
    -->
    <select id="findAll" resultType="com.itheima.domain.User">
        select * from user
    </select>
```

9.1.3 在测试类中的测试方法

在原有的 UserTest 类中加入测试方法，如下：

```
    //查询所有
    @Test
    public void testFindAll() {
        UserDao userDao = sqlSession.getMapper(UserDao.class);
        List<User> users = userDao.findAll();
        for (User user : users) {
            System.out.println(user);
        }
    }
```

测试结果如下:



9.2 ResultMap的使用

当数据库返回的结果集中的字段和实体类中的属性名存在不对应情况时,可以使用resultMap自定义映射关系

9.2.1 在UserMapper类中加入方法

在UserMapper.java类中添加一个用于 findAllMap()的方法,如下:

```
 //查询所有
List<User> findAllMap();
```

9.2.2 在 UserMapper.xml 文件中加入配置

在 UserMapper.xml 文件中加入 findAllMap的配置，如下:

```
    <!--
      resultMap : 手动指定数据库字段与实体关系映射配置
      id 属性 :  给当前配置命名, 方便其他地方进行引用
      type 属性: 表示最后返回的实体类对象的全限定类名
        id标签 : 配置主键
        result标签 : 配置普通字段
            column 用于指定数据库返回结果集中的字段
            property 用于指定实体类中的属性名称
    -->
    <resultMap id="userMap" type="com.itheima.domain.User">
        <id column="id" property="uid"/>
        <result column="username" property="name"/>
        <result column="password" property="password"/>
        <result column="email" property="email"/>
        <result column="birthday" property="birthday"/>
    </resultMap>

    <select id="findAllMap" resultMap="userMap">
        <!--通过改别名的方式，现在返回结果集的列名已经与 User类的属性名不相同了-->
        select uid as id, name as username,password,email,birthday from user
    </select>
```

9.2.3 在测试类中的测试方法

在原有的 UserTest 类中加入测试方法，如下:

```
    @Test
    public void testFindAllMap() {
        UserDao userDao = sqlSession.getMapper(UserDao.class);
        List<User> users = userDao.findAllMap();
        for (User user : users) {
            System.out.println(user);
        }
    }
```

测试结果如下:

```
==>  Preparing: select uid as id, name as username,password,email,birthday from user
[DEBUG] 2019-12-14 18:59:14,775 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debug(BaseJdbcLogger.jav
==> Parameters:
[DEBUG] 2019-12-14 18:59:14,804 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debug(BaseJdbcLogger.jav
<==      Total: 3
User{uid=1, name='传智播客1', password='admin1', email='admin@itcast1.cn', birthday=Sat Dec 14 00:00:00 CST 20
User{uid=2, name='传智播客2', password='admin2', email='admin2@itcast.cn', birthday=Sat Jun 01 00:00:00 CST 20
User{uid=3, name='传智播客', password='admin', email='admin@itcast.cn', birthday=Sat Dec 14 00:00:00 CST 2019}
```

9.3 一个条件的查询

9.2.1 在UserMapper类中加入方法

在UserMapper.java类中添加一个方法,如下:

```
//单条件查询
List<User> findByName(String name);
```

9.2.2 在UserMapper.xml 文件中加入配置

在 UserMapper.xml 文件中加入配置，如下:

```
    <!--单条件查询-->
    <select id="findByName" parameterType="java.lang.String"
            resultType="com.itheima.domain.User">
        select * from user where name = #{name}
    </select>
```

9.2.3 在测试类中的测试方法

在原有的 UserTest 类中加入测试方法，如下:

```
    //一个条件查询
    @Test
    public void testFindByName() {
        UserDao userDao = sqlSession.getMapper(UserDao.class);
        List<User> users = userDao.findByName("传智播客");
        for (User user : users) {
            System.out.println(user);
        }
    }
```

测试结果如下:

```
==>  Preparing: select * from user where name = ?
[DEBUG] 2019-12-14 19:05:55,936 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debug(Ba
==> Parameters: 传智播客(String)
[DEBUG] 2019-12-14 19:05:55,958 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debug(Ba
<==      Total: 1
User{uid=3, name='传智播客', password='admin', email='admin@itcast.cn', birthday=Sat Dec 14 00
```

9.4 多个条件的查询

9.4.1 方式一

接口文件

在UserMapper.java类中添加一个方法,如下:

```
List<User> findByNameAndPassword1(String name, String password);
```

映射文件

在 UserMapper.xml 文件中加入配置，如下:

```
    <!--
      由于多条件查询的时候有两个参数 我们可以使用数组索引的方式查询
      mybatis默认提供的两个对象都可以使用 arg 或者 param
      arg0 arg1 arg2...
      param0 param1 param2
    -->
    <select id="findByNameAndPassword1"
            resultType="com.itheima.domain.User">
        <!--select * from user where name = #{arg0} and password = #{arg1}-->
        select * from user where name = #{param1} and password = #{param2}
    </select>
```

测试

在原有的 UserTest 类中加入测试方法，如下:

```
    @Test
    public void testFindByNameAndPassword1() {
        UserDao userDao = sqlSession.getMapper(UserDao.class);
        List<User> users = userDao.findByNameAndPassword3("传智播客", "admin");
        for (User user : users) {
            System.out.println(user);
        }
    }
```

测试结果如下:

```
==>  Preparing: select * from user where name = ? and password = ?
[DEBUG] 2019-12-14 19:20:15,625 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogg
==> Parameters: 传智播客(String), admin(String)
[DEBUG] 2019-12-14 19:20:15,646 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogg
<==      Total: 1
User{uid=3, name='传智播客', password='admin', email='admin@itcast.cn', birthday=Sat
```

9.4.2 方式二

接口文件

在UserMapper.java类中添加一个方法,如下:

```
//注解@Param用于和xml中的#{}中获取值的时候做个对应
//注意@Param("")中写的值一定要跟xml中引用时写的一致
List<User> findByNameAndPassword2(@Param("name") String name,
                                  @Param("password") String password);
```

映射文件

在 UserMapper.xml 文件中加入配置，如下：

```
    <select id="findByNameAndPassword2"
            resultType="com.itheima.domain.User">
        select * from user where name = #{name} and password = #{password}
    </select>
```

测试

在原有的 UserTest 类中加入测试方法，如下：

```
    @Test
    public void testFindByNameAndPassword2() {
        UserDao userDao = sqlSession.getMapper(UserDao.class);
        List<User> users = userDao.findByNameAndPassword2("传智播客", "admin");
        for (User user : users) {
            System.out.println(user);
        }
    }
```

测试结果如下:

```
==>  Preparing: select * from user where name = ? and password = ?
[DEBUG] 2019-12-14 19:21:17,896 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger
==> Parameters: 传智播客(String), admin(String)
[DEBUG] 2019-12-14 19:21:17,921 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger
<==      Total: 1
User{uid=3, name='传智播客', password='admin', email='admin@itcast.cn', birthday=Sat
```

9.4.2 方式三

接口文件

在UserMapper.java类中添加一个方法,如下：

```
    //直接使用一个对象传递多个参数
    List<User> findByNameAndPassword3(User user);
```

映射文件

在 UserMapper.xml 文件中加入配置，如下：

```
    <select id="findByNameAndPassword3" parameterType="com.itheima.domain.User"
            resultType="com.itheima.domain.User">
        select * from user where name = #{name} and password = #{password}
    </select>
```

测试

在原有的 UserTest 类中加入测试方法，如下：

```
    @Test
    public void testFindByNameAndPassword3() {
        UserDao userDao = sqlSession.getMapper(UserDao.class);
        User userParam = new User();
        userParam.setName("传智播客");
        userParam.setPassword("admin");

        List<User> users = userDao.findByNameAndPassword3(userParam);
        for (User user : users) {
            System.out.println(user);
        }
    }
```

测试结果如下:

```
==>  Preparing: select * from user where name = ? and password = ?
[DEBUG] 2019-12-14 19:21:59,203 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debug(
==> Parameters: 传智播客(String), admin(String)
[DEBUG] 2019-12-14 19:21:59,228 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debug(
<==      Total: 1
User{uid=3, name='传智播客', password='admin', email='admin@itcast.cn', birthday=Sat Dec 14 (
```

9.5 模糊查询

9.5.1 方式一

接口文件

在UserMapper.java类中添加一个方法,如下：

```
    //模糊查询
    List<User> findByNameLike1(String name);
```

映射文件

在 UserMapper.xml 文件中加入配置，如下：

```
<!--模糊查询-->
    <select id="findByNameLike1" parameterType="java.lang.String"
            resultType="com.itheima.domain.User">
        select * from user where name like #{name}
    </select>
```

测试

在原有的 UserTest 类中加入测试方法，如下：

```java
//模糊查询
@Test
public void testFindByNameLike1() {
    UserDao userDao = sqlSession.getMapper(UserDao.class);
    List<User> users = userDao.findByNameLike1("%传智%");
    for (User user : users) {
        System.out.println(user);
    }
}
```

测试结果如下:

```
==>  Preparing: select * from user where name like ?
[DEBUG] 2019-12-14 19:22:36,551 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger
==> Parameters: %传智%(String)
[DEBUG] 2019-12-14 19:22:36,573 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger
<==      Total: 3
User{uid=1, name='传智播客1', password='admin1', email='admin@itcast1.cn', birthday=Sa
User{uid=2, name='传智播客2', password='admin2', email='admin2@itcast.cn', birthday=Sa
User{uid=3, name='传智播客', password='admin', email='admin@itcast.cn', birthday=Sat D
```

9.5.2 方式二

接口文件

在UserMapper.java类中添加一个方法,如下:

```java
//模糊查询
List<User> findByNameLike2(String name);
```

映射文件

在 UserMapper.xml 文件中加入配置，如下:

```xml
<!--
    注意:在Oracle中, "" 只能用于起别名
-->
<select id="findByNameLike2" parameterType="java.lang.String"
        resultType="com.itheima.domain.User">
    select * from user where name like "%"#{name}"%"
</select>
```

测试

在原有的 UserTest 类中加入测试方法，如下:

```java
@Test
public void testFindByNameLike2() {
    UserDao userDao = sqlSession.getMapper(UserDao.class);
    List<User> users = userDao.findByNameLike2("传智");
    for (User user : users) {
        System.out.println(user);
    }
}
```

测试结果如下:

```
==>  Preparing: select * from user where name like "%"?"%"
[DEBUG] 2019-12-14 19:23:15,835 method:org.apache.ibatis.logging.jdbc.BaseJdbcLog
==> Parameters: 传智(String)
[DEBUG] 2019-12-14 19:23:15,854 method:org.apache.ibatis.logging.jdbc.BaseJdbcLog
<==      Total: 3
User{uid=1, name='传智播客1', password='admin1', email='admin@itcast1.cn', birthday
User{uid=2, name='传智播客2', password='admin2', email='admin2@itcast.cn', birthday
User{uid=3, name='传智播客', password='admin', email='admin@itcast.cn', birthday=Sa
```

9.5.3 方式三

接口文件

在UserMapper.java类中添加一个方法,如下:

```
    //模糊查询
    List<User> findByNameLike3(String name);
```

映射文件

在 UserMapper.xml 文件中加入配置，如下:

```
    <!--
        注意:在Oracle中, concat()只支持两个参数
    -->
    <select id="findByNameLike3" parameterType="java.lang.String"
            resultType="com.itheima.domain.User">
        select * from user where name like concat(concat('%',#{name}),'%')
    </select>
```

测试

在原有的 UserTest 类中加入测试方法，如下:

```
    @Test
    public void testFindByNameLike3() {
        UserDao userDao = sqlSession.getMapper(UserDao.class);
        List<User> users = userDao.findByNameLike3("传智");
        for (User user : users) {
            System.out.println(user);
        }
    }
```

测试结果如下:

```
==>  Preparing: select * from user where name like concat(concat('%',?),'%')
[DEBUG] 2019-12-14 19:25:09,264 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.
==> Parameters: 传智(String)
[DEBUG] 2019-12-14 19:25:09,285 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.
<==      Total: 3
User{uid=1, name='传智播客1', password='admin1', email='admin@itcast1.cn', birthday=Sat
User{uid=2, name='传智播客2', password='admin2', email='admin2@itcast.cn', birthday=Sat
User{uid=3, name='传智播客', password='admin', email='admin@itcast.cn', birthday=Sat De
```

9.5.4 方式四

接口文件

在UserMapper.java类中添加一个方法,如下:

```
    //模糊查询
    List<User> findByNameLike4(String name);
```

映射文件

在 UserMapper.xml 文件中加入配置，如下:

```
    <!-- 注意: ${} 的形式中间key必须填写value-->
    <select id="findByNameLike4" parameterType="java.lang.String"
            resultType="com.itheima.domain.User">
        select * from user where name like "%${value}%"
    </select>
```

测试

在原有的 UserTest 类中加入测试方法，如下:

```
    @Test
    public void testFindByNameLike4() {
        UserDao userDao = sqlSession.getMapper(UserDao.class);
        List<User> users = userDao.findByNameLike4("传智");
        for (User user : users) {
            System.out.println(user);
        }
    }
```

测试结果如下:

```
[DEBUG] 2019-12-14 19:25:53,914 method:org.apache.ibatis.logging.jdbc.
==>  Preparing: select * from user where name like "%传智%"
[DEBUG] 2019-12-14 19:25:53,947 method:org.apache.ibatis.logging.jdbc.
==> Parameters:
[DEBUG] 2019-12-14 19:25:53,975 method:org.apache.ibatis.logging.jdbc.
<==      Total: 3
User{uid=1, name='传智播客1', password='admin1', email='admin@itcast1.cn
User{uid=2, name='传智播客2', password='admin2', email='admin2@itcast.cn
User{uid=3, name='传智播客', password='admin', email='admin@itcast.cn',
```

9.6 #{}和${}的区别

```
# 表示占位符，相当于JDBC中的?，底层工作的是PreparedStetement对象，SQL只编译一次，而且没有SQL注入问题
# 当传入的参数为一个简单类型时，#{}可以随便写

$ 表示字符串拼接，底层工作的是Statement对象，每次都会重新编译，而且存在 SQL 注入问题
$ 当传入的参数为一个简单类型时，${}只能写value
```

十、返回主键

需求: 我们很多时候有这种需求，向数据库插入一条记录后，希望能立即拿到这条记录在数据库中的主键值。

10.1 接口文件

```
//返回主键
void save1(User user);
```

10.2 映射文件

方式一

此方式只适用于底层支持主键自增长的数据库(mysql DB2) oracle是不行的

```
<!--
    useGeneratedKeys="true" 声明需要返回主键
    keyProperty  声明返回的主键应该封装进输入参数(User对象)的哪一个属性(uid)
-->
<insert id="save1" parameterType="com.itheima.domain.User"
            useGeneratedKeys="true" keyProperty="uid">
        insert into user(name,password,email,birthday)
        values(#{name},#{password},#{email},#{birthday})
</insert>
```

方式二

```
    <!--
        selectKey 中间是自定义查询主键的sql语句
            order 表示selectKey中的sql语句在insert语句的什么位置执行
            resultType 将返回的结果转换成什么类型
            keyProperty 将转换之后的值赋值到输入属性的哪个参数上
    -->
    <insert id="save1" parameterType="com.itheima.domain.User">
        <selectKey order="AFTER" resultType="java.lang.Integer"
keyProperty="uid">
            <!--这句sql的意思是: 只要跟在一条插入语句之后,就能返回最后插入的主键的值-->
            SELECT LAST_INSERT_ID();
        </selectKey>
        insert into user(name,password,email,birthday)
        values(#{name},#{password},#{email},#{birthday})
    </insert>
```

10.3 测试

```
    @Test
    public void testSave1() throws IOException {
        //创建user对象
        User user = new User();
        user.setName("传智播客");
        user.setEmail("admin@itcast.cn");
        user.setPassword("admin");
        user.setBirthday(new Date());

        //4 执行操作
        UserDao userDao = sqlSession.getMapper(UserDao.class);
        userDao.save1(user);

        System.out.println(user);
    }
```

测试结果如下:

```
==>  Preparing: insert into user(name,password,email,birthday) values(?,?,?,?)
[DEBUG] 2019-12-14 19:28:52,495 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debug(BaseJdbcLogger.java
==> Parameters: 传智播客(String), admin(String), admin@itcast.cn(String), 2019-12-14 19:28:52.069(Timestamp)
[DEBUG] 2019-12-14 19:28:52,497 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debug(BaseJdbcLogger.java
<==    Updates: 1
[DEBUG] 2019-12-14 19:28:52,502 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debug(BaseJdbcLogger.java
==>  Preparing: SELECT LAST_INSERT_ID();
[DEBUG] 2019-12-14 19:28:52,503 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debug(BaseJdbcLogger.java
==> Parameters:
[DEBUG] 2019-12-14 19:28:52,519 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debug(BaseJdbcLogger.java
<==      Total: 1
User{uid=5, name='传智播客', password='admin', email='admin@itcast.cn', birthday=Sat Dec 14 19:28:52 CST 2019}
```