

Mybatis 第一天

框架课程

1. 课程计划

第一天：

- 1、Mybatis 的介绍
- 2、Mybatis 的入门
 - a) 使用 jdbc 操作数据库存在的问题
 - b) Mybatis 的架构
 - c) Mybatis 的入门程序
- 3、Dao 的开发方法
 - a) 原始 dao 的开发方法
 - b) 接口的动态代理方式
- 4、SqlMapConfig.xml 文件说明

第二天：

- 1、输入映射和输出映射
 - a) 输入参数映射
 - b) 返回值映射
- 2、动态 sql
- 3、关联查询
 - a) 一对一关联
 - b) 一对多关联
- 4、Mybatis 整合 spring

2. Mybatis 介绍

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

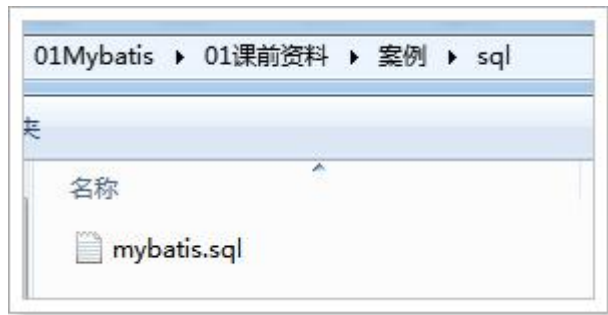
MyBatis 是一个优秀的持久层框架, 它对 jdbc 的操作数据库的过程进行封装, 使开发者只需要关注 SQL 本身, 而不需要花费精力去处理例如注册驱动、创建 connection、创建 statement、手动设置参数、结果集检索等 jdbc 繁杂的过程代码。

Mybatis 通过 xml 或注解的方式将要执行的各种 statement (statement、preparedStatemnt、CallableStatement) 配置起来, 并通过 java 对象和 statement 中的 sql 进行映射生成最终执行的 sql 语句, 最后由 mybatis 框架执行 sql 并将结果映射成 java 对象并返回。

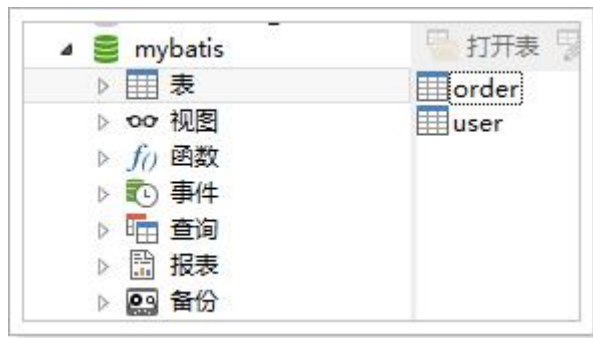
3. 使用 jdbc 编程问题总结

3.1. 创建 mysql 数据库

创建数据库，将下图所示的 sql 脚本导入到数据库中。



导入后效果如下图：



3.2. 创建工程

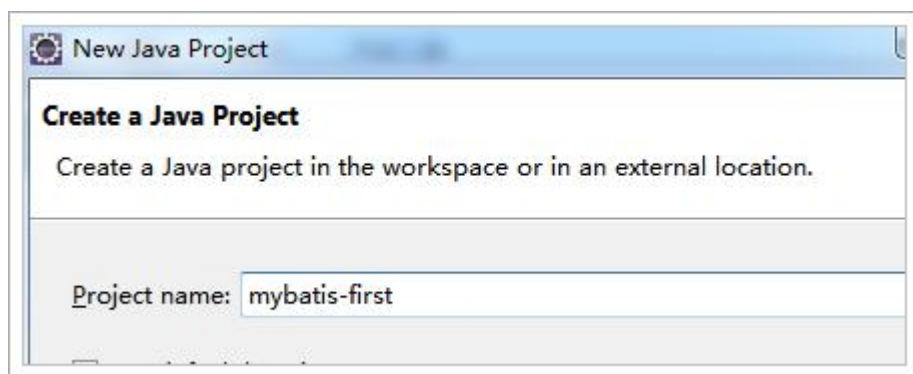
开发环境：

IDE: eclipse Mars2

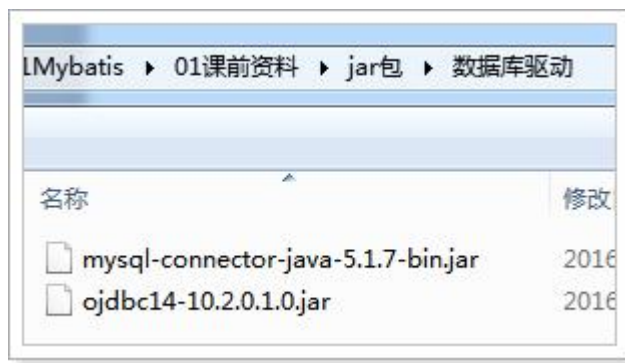
JDK: 1.7

1、创建一个 java 工程。

按下图进行创建



2、需要 mysql 的数据库驱动，如下图位置找到 jar 包。



3.3. jdbc 编程步骤:

- 1、加载数据库驱动
- 2、创建并获取数据库链接
- 3、创建 jdbc statement 对象
- 4、设置 sql 语句
- 5、设置 sql 语句中的参数(使用 preparedStatement)
- 6、通过 statement 执行 sql 并获取结果
- 7、对 sql 执行结果进行解析处理
- 8、释放资源(resultSet、preparedstatement、connection)

3.4. jdbc 程序

```
public static void main(String[] args) {  
    Connection connection = null;  
    PreparedStatement preparedStatement = null;  
    ResultSet resultSet = null;  
  
    try {  
        // 加载数据库驱动  
        Class.forName("com.mysql.jdbc.Driver");  
  
        // 通过驱动管理类获取数据库链接  
        connection =  
DriverManager.getConnection("jdbc:mysql://localhost:3306/mybatis?cha  
racterEncoding=utf-8", "root", "root");  
        // 定义 sql 语句 ?表示占位符  
        String sql = "select * from user where username = ?";  
        // 获取预处理 statement  
        preparedStatement = connection.prepareStatement(sql);  
    }  
}
```

```

        // 设置参数，第一个参数为 sql 语句中参数的序号（从 1 开始），第二个参数
        为设置的参数值
        preparedStatement.setString(1, "王五");
        // 向数据库发出 sql 执行查询，查询出结果集
        resultSet = preparedStatement.executeQuery();
        // 遍历查询结果集
        while (resultSet.next()) {
            System.out.println(resultSet.getString("id") + " " +
resultSet.getString("username"));
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        // 释放资源
        if (resultSet != null) {
            try {
                resultSet.close();
            } catch (SQLException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
        if (preparedStatement != null) {
            try {
                preparedStatement.close();
            } catch (SQLException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
        if (connection != null) {
            try {
                connection.close();
            } catch (SQLException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    }
}
}
}

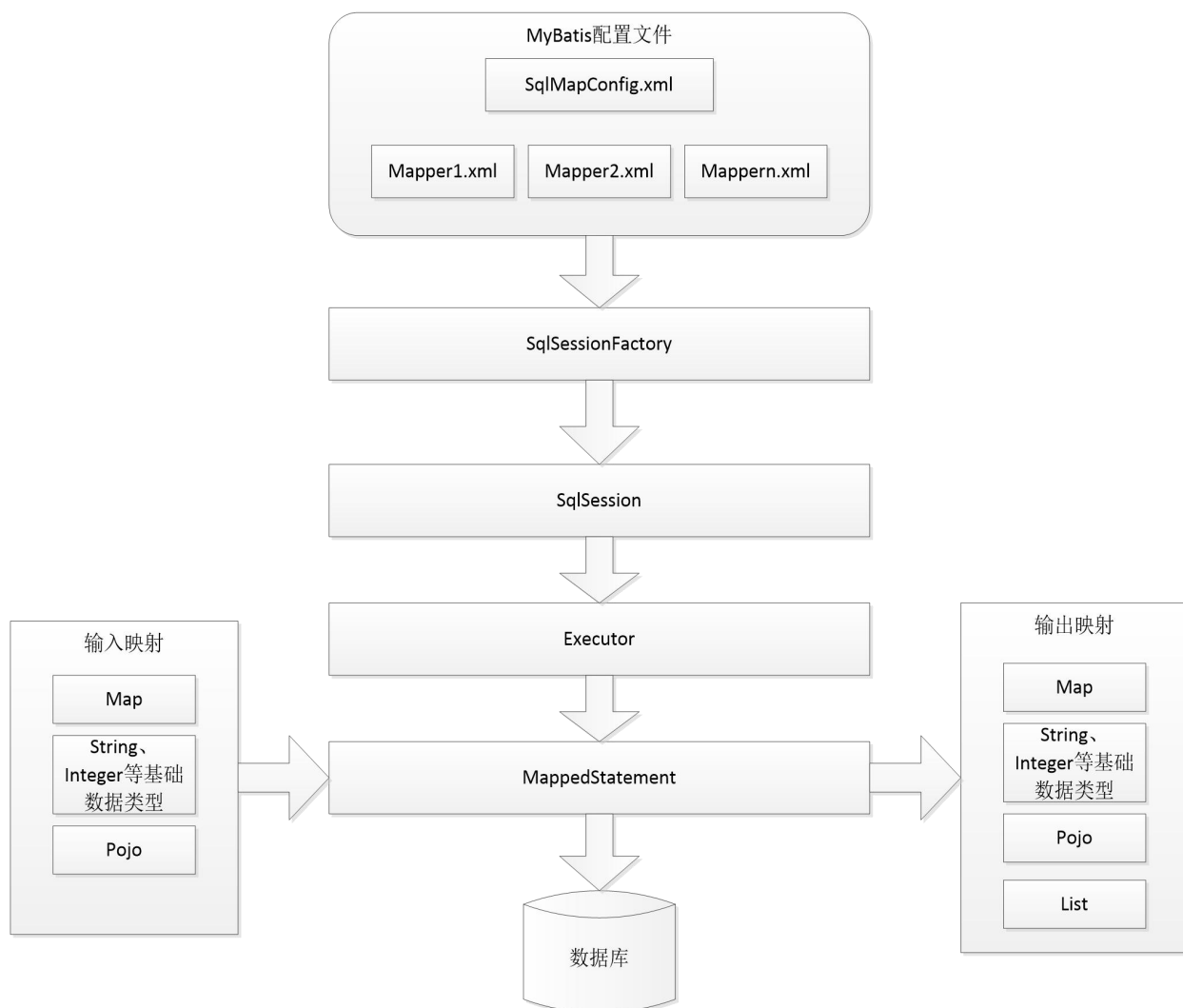
```

上边使用 jdbc 的原始方法（未经封装）实现了查询数据库表记录的操作。

3.5. jdbc 问题总结如下：

- 1、数据库连接创建、释放频繁造成系统资源浪费，从而影响系统性能。如果使用数据库连接池可解决此问题。
- 2、Sql 语句在代码中硬编码，造成代码不易维护，实际应用中 sql 变化的可能较大，sql 变动需要改变 java 代码。
- 3、使用 preparedStatement 向占有位符号传参数存在硬编码，因为 sql 语句的 where 条件不一定，可能多也可能少，修改 sql 还要修改代码，系统不易维护。
- 4、对结果集解析存在硬编码（查询列名），sql 变化导致解析代码变化，系统不易维护，如果能将数据库记录封装成 pojo 对象解析比较方便。

4. Mybatis 架构



1、mybatis 配置

SqlMapConfig.xml，此文件作为 mybatis 的全局配置文件，配置了 mybatis 的运行环境等信息。

mapper.xml 文件即 sql 映射文件，文件中配置了操作数据库的 sql 语句。此文件需要在 SqlMapConfig.xml 中加载。

2、通过 mybatis 环境等配置信息构造 SqlSessionFactory 即会话工厂

3、由会话工厂创建 sqlSession 即会话，操作数据库需要通过 sqlSession 进行。

4、mybatis 底层自定义了 Executor 执行器接口操作数据库，Executor 接口有两个实现，一个是基本执行器、一个是缓存执行器。

5、Mapped Statement 也是 mybatis 一个底层封装对象，它包装了 mybatis 配置信息及 sql 映射信息等。mapper.xml 文件中一个 sql 对应一个 Mapped Statement 对象，sql 的 id 即是 Mapped statement 的 id。

6、Mapped Statement 对 sql 执行输入参数进行定义，包括 HashMap、基本类型、pojo，Executor 通过 Mapped Statement 在执行 sql 前将输入的 java 对象映射至 sql 中，输入参数映射就是 jdbc 编程中对 preparedStatement 设置参数。

7、Mapped Statement 对 sql 执行输出结果进行定义，包括 HashMap、基本类型、pojo，Executor 通过 Mapped Statement 在执行 sql 后将输出结果映射至 java 对象中，输出结果映射过程相当于 jdbc 编程中对结果的解析处理过程。

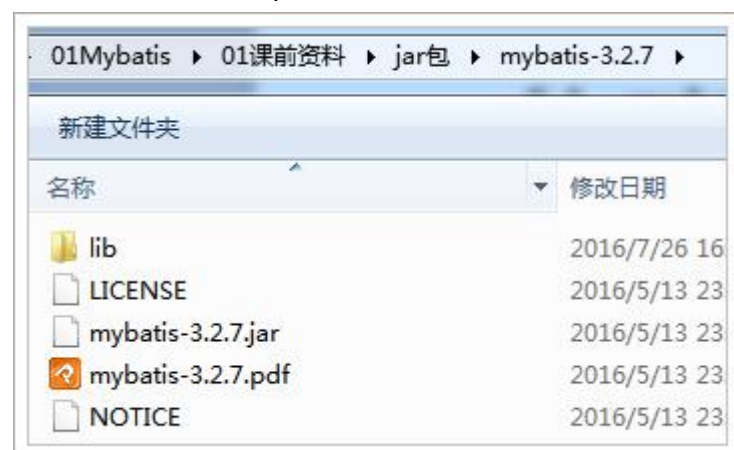
5. Mybatis 入门程序

5.1. mybatis 下载

mybaitis 的代码由 github.com 管理

下载地址：<https://github.com/mybatis/mybatis-3/releases>

课前资料提供的 mybatis 如下：



mybatis-3.2.7.jar mybatis 的核心包
lib 文件夹 mybatis 的依赖包所在
mybatis-3.2.7.pdf mybatis 使用手册

5.2. 业务需求

使用 MyBatis 实现以下功能：
根据用户 id 查询一个用户
根据用户名称模糊查询用户列表
添加用户
更新用户
删除用户

5.3. 环境搭建

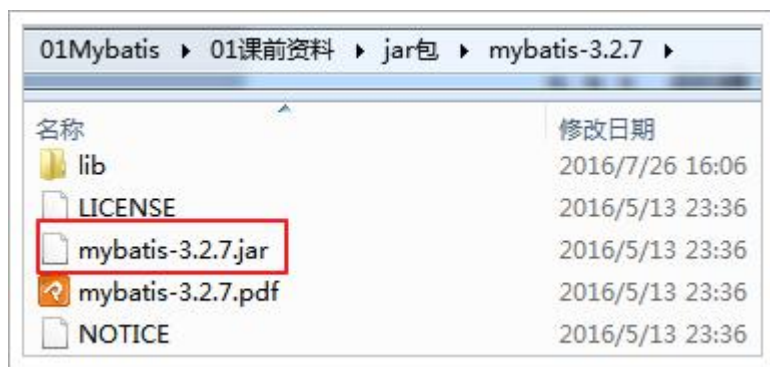
5.3.1. 创建 java 工程

使用之前创建的 mybatis-first 工程










5.3.2. 加入 jar 包

加入 mybatis 核心包、依赖包、数据驱动包。



mybatis 核心包














mybatis 依赖包

01Mybatis ▶ 01课前资料 ▶ jar包 ▶ mybatis-3.2.7 ▶ lib	
新建文件夹	
名称	修改日期
 asm-3.3.1.jar	2016/5/13 23:36
 cglib-2.2.2.jar	2016/5/13 23:36
 commons-logging-1.1.1.jar	2016/5/13 23:36
 javassist-3.17.1-GA.jar	2016/5/13 23:36
 log4j-1.2.17.jar	2016/5/13 23:36
 log4j-api-2.0-rc1.jar	2016/5/13 23:36
 log4j-core-2.0-rc1.jar	2016/5/13 23:36
 slf4j-api-1.7.5.jar	2016/5/13 23:36
 slf4j-log4j12-1.7.5.jar	2016/5/13 23:36

数据库驱动包（已添加）

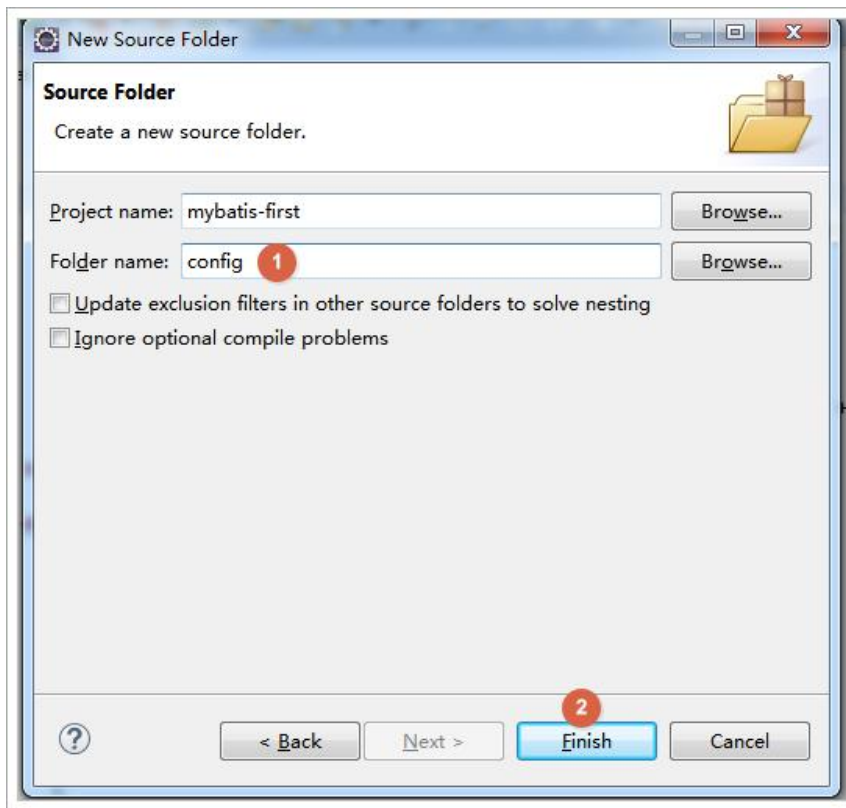
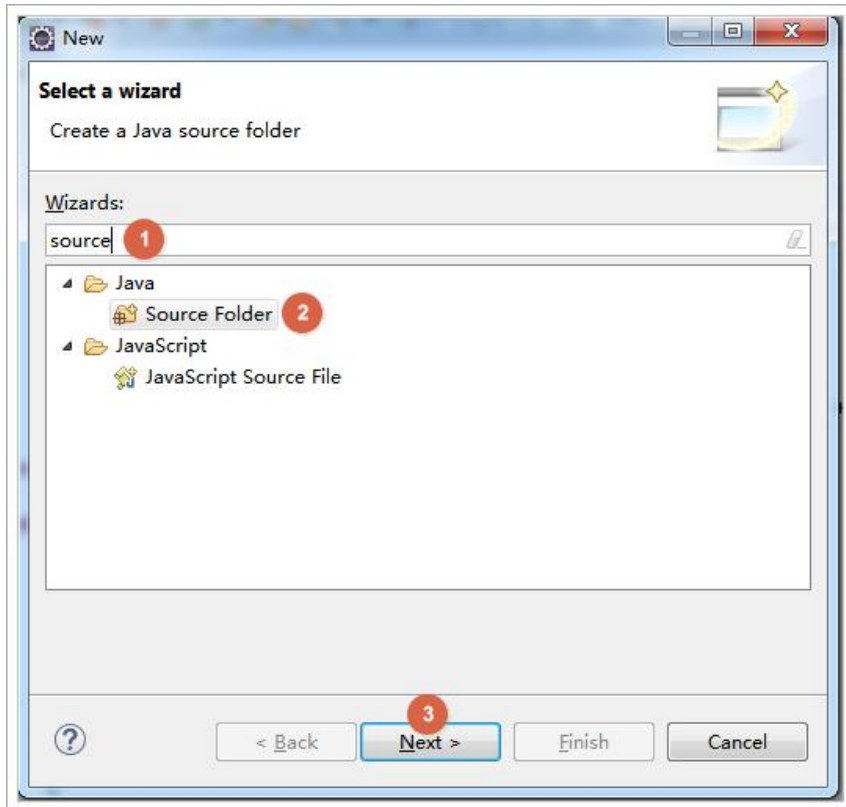
1Mybatis ▶ 01课前资料 ▶ jar包 ▶ 数据库驱动	
名称	修改
 mysql-connector-java-5.1.7-bin.jar	2016
 ojdbc14-10.2.0.1.0.jar	2016

效果：

mybatis-first
▶ src
▶ JRE System Library [JavaSE-1.7]
▶ Referenced Libraries
▶  mysql-connector-java-5.1.7-bin.jar
▶  asm-3.3.1.jar
▶  cglib-2.2.2.jar
▶  commons-logging-1.1.1.jar
▶  javassist-3.17.1-GA.jar
▶  log4j-1.2.17.jar
▶  log4j-api-2.0-rc1.jar
▶  log4j-core-2.0-rc1.jar
▶  mybatis-3.2.7.jar
▶  slf4j-api-1.7.5.jar
▶  slf4j-log4j12-1.7.5.jar

5.3.3. 加入配置文件

如下图创建资源文件夹 config，加入 log4j.properties 和 SqlMapConfig.xml 配置文件



5.3.3.1. log4j.properties

在 config 下创建 log4j.properties 如下：

```
# Global logging configuration
log4j.rootLogger=DEBUG, stdout
# Console output...
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n
```

mybatis 默认使用 log4j 作为输出日志信息。

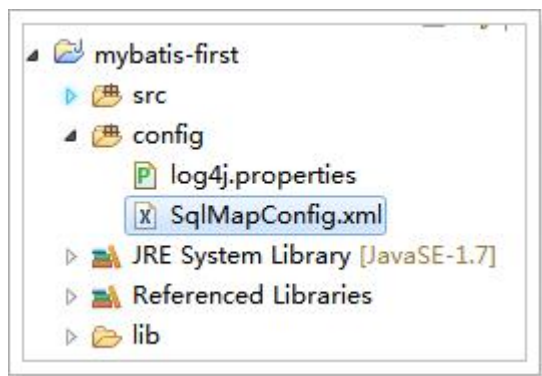
5.3.3.2. SqlMapConfig.xml

在 config 下创建 SqlMapConfig.xml，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>
  <!-- 和 spring 整合后 environments 配置将废除 -->
  <environments default="development">
    <environment id="development">
      <!-- 使用 jdbc 事务管理 -->
      <transactionManager type="JDBC" />
      <!-- 数据库连接池 -->
      <dataSource type="POOLED">
        <property name="driver" value="com.mysql.jdbc.Driver" />
        <property name="url"
          value="jdbc:mysql://localhost:3306/mybatis?characterEncoding=utf-8" />
        <property name="username" value="root" />
        <property name="password" value="root" />
      </dataSource>
    </environment>
  </environments>
</configuration>
```

SqlMapConfig.xml 是 mybatis 核心配置文件，配置文件内容为数据源、事务管理。

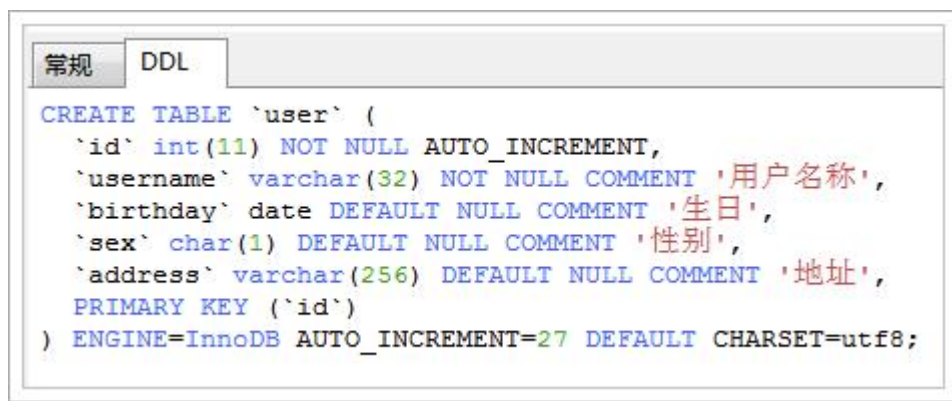
5.3.3.3. 效果



5.3.4. 创建 pojo

pojo 类作为 mybatis 进行 sql 映射使用，po 类通常与数据库表对应，

数据库 user 表如下图：



User.java 如下：

```
Public class User {  
    private int id;  
    private String username; // 用户姓名  
    private String sex; // 性别  
    private Date birthday; // 生日  
    private String address; // 地址
```

get/set.....

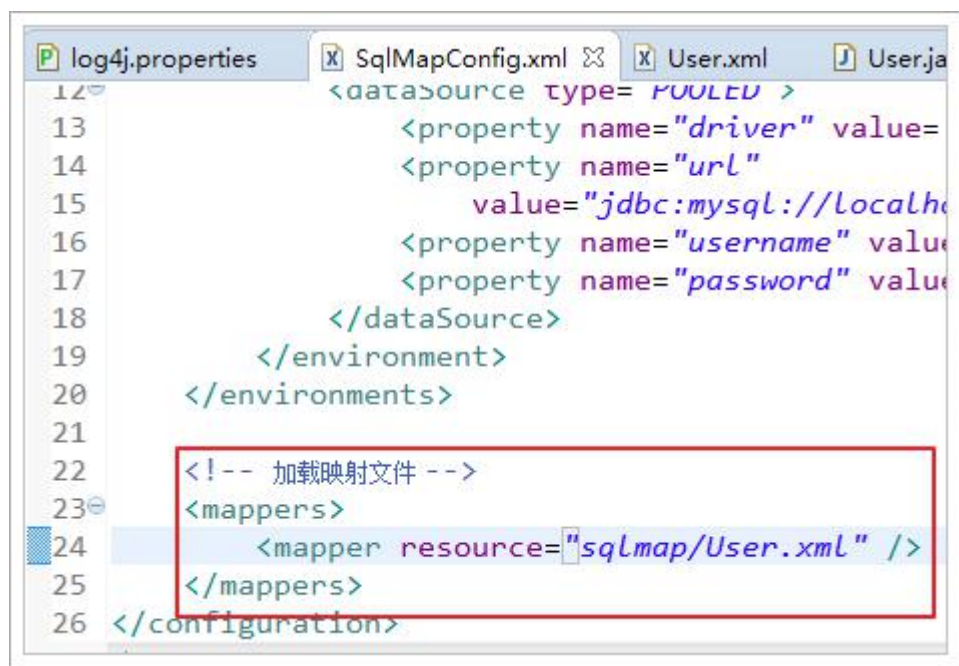
5.3.5. 第六步：sql 映射文件

在 config 下的 sqlmap 目录下创建 sql 映射文件 User.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">
<!-- namespace: 命名空间，用于隔离sql，还有一个很重要的作用，后面会讲 -->
<mapper namespace="test">
</mapper>
```

5.3.6. 第七步：加载映射文件

mybatis 框架需要加载 Mapper.xml 映射文件
将 users.xml 添加在 SqlMapConfig.xml，如下：



5.4. 实现根据 id 查询用户

使用的 sql:

```
SELECT * FROM `user` WHERE id = 1
```

5.4.1. 映射文件:

在 user.xml 中添加 select 标签, 编写 sql:

```
<?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, 用于隔离 sql, 还有一个很重要的作用, 后面会讲 -->
<mapper namespace="test">

    <!-- id:statement 的 id 或者叫做 sql 的 id-->
    <!-- parameterType:声明输入参数的类型 -->
    <!-- resultType:声明输出结果的类型, 应该填写 pojo 的全路径 -->
    <!-- #{}: 输入参数的占位符, 相当于 jdbc 的? -->
    <select id="queryUserById" parameterType="int"
        resultType="com.cloudtcc.mybatis.pojo.User">
        SELECT * FROM `user` WHERE id = #{id}
    </select>

</mapper>
```

5.4.2. 测试程序:

测试程序步骤:

1. 创建 SqlSessionFactoryBuilder 对象
2. 加载 SqlMapConfig.xml 配置文件
3. 创建 SqlSessionFactory 对象
4. 创建 SqlSession 对象
5. 执行 SqlSession 对象执行查询, 获取结果 User
6. 打印结果
7. 释放资源

MybatisTest 编写测试程序如下:

```
public class MybatisTest {
    private SqlSessionFactory sqlSessionFactory = null;

    @Before
    public void init() throws Exception {
        // 1. 创建 SqlSessionFactoryBuilder 对象
        SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new
        SqlSessionFactoryBuilder();
```

```

        // 2. 加载 SqlMapConfig.xml 配置文件
        InputStream inputStream =
Resources.getResourceAsStream("SqlMapConfig.xml");

        // 3. 创建 SqlSessionFactory 对象
        this.sqlSessionFactory =
sqlSessionFactoryBuilder.build(inputStream);
    }

    @Test
    public void testQueryUserById() throws Exception {
        // 4. 创建 SqlSession 对象
        SqlSession sqlSession = sqlSessionFactory.openSession();

        // 5. 执行 SqlSession 对象执行查询，获取结果 User
        // 第一个参数是 User.xml 的 statement 的 id，第二个参数是执行 sql 需
        要的参数；
        Object user = sqlSession.selectOne("queryUserById", 1);

        // 6. 打印结果
        System.out.println(user);

        // 7. 释放资源
        sqlSession.close();
    }
}

```

5.4.3. 效果

测试结果如下图

```

DEBUG [main] - Setting autocommit to false on JDBC Connection [co
DEBUG [main] - ==> Preparing: SELECT * FROM `user` WHERE id = ?
DEBUG [main] - ==> Parameters: 1(Integer)
DEBUG [main] - <==      Total: 1
User [id=1, username=王五, sex=2, birthday=null, address=null]

```

5.5. 实现根据用户名模糊查询用户

查询 sql:

```
SELECT * FROM `user` WHERE username LIKE '%王%'
```


5.5.1. 方法一

5.5.1.1. 映射文件

在 User.xml 配置文件中添加如下内容：

```
<!-- 如果返回多个结果，mybatis 会自动把返回的结果放在 list 容器中 -->
<!-- resultMap 的配置和返回一个结果的配置一样 -->
<select id="queryUserByUsername1" parameterType="string"
        resultMap="com.cloudtcc.mybatis.pojo.User">
    SELECT * FROM `user` WHERE username LIKE #{username}
</select>
```

5.5.1.2. 测试程序

MybatisTest 中添加测试方法如下：

```
@Test
public void testQueryUserByUsername1() throws Exception {
    // 4. 创建 SqlSession 对象
    SqlSession sqlSession = sqlSessionFactory.openSession();

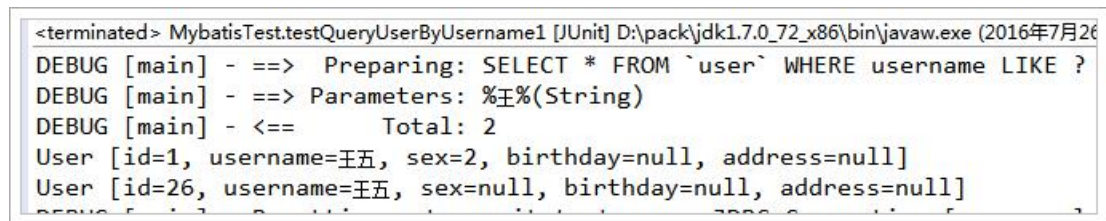
    // 5. 执行 SqlSession 对象执行查询，获取结果 User
    // 查询多条数据使用 selectList 方法
    List<Object> list =
sqlSession.selectList("queryUserByUsername1", "%王%");

    // 6. 打印结果
    for (Object user : list) {
        System.out.println(user);
    }

    // 7. 释放资源
    sqlSession.close();
}
```

5.5.1.3. 效果

测试效果如下图：



```
<terminated> MybatisTest.testQueryUserByUsername1 [JUnit] D:\pack\jdk1.7.0_72_x86\bin\javaw.exe (2016年7月26
DEBUG [main] - ==> Preparing: SELECT * FROM `user` WHERE username LIKE ?
DEBUG [main] - ==> Parameters: %王%(String)
DEBUG [main] - <==          Total: 2
User [id=1, username=王五, sex=2, birthday=null, address=null]
User [id=26, username=王五, sex=null, birthday=null, address=null]
```


5.5.2. 方法二

5.5.2.1. 映射文件：

在 User.xml 配置文件中添加如下内容：

```
<!-- 如果传入的参数是简单数据类型，${}里面必须写 value -->
<select id="queryUserByUsername2" parameterType="string"
        resultType="com.cloudtcc.mybatis.pojo.User">
    SELECT * FROM `user` WHERE username LIKE '%${value}%'
</select>
```

5.5.2.2. 测试程序：

MybatisTest 中添加测试方法如下：

```
@Test
public void testQueryUserByUsername2() throws Exception {
    // 4. 创建 SqlSession 对象
    SqlSession sqlSession = sqlSessionFactory.openSession();

    // 5. 执行 SqlSession 对象执行查询，获取结果 User
    // 查询多条数据使用 selectList 方法
    List<Object> list = sqlSession.selectList("queryUserByUsername2", "王");

    // 6. 打印结果
    for (Object user : list) {
        System.out.println(user);
    }

    // 7. 释放资源
    sqlSession.close();
}
```

5.5.2.3. 效果

测试结果如下图：

```
DEBUG [main] - ==> Preparing: SELECT * FROM `user` WHERE username LIKE '%王%'
DEBUG [main] - ==> Parameters:
DEBUG [main] - <==          Total: 2
User [id=1, username=王五, sex=2, birthday=null, address=null]
User [id=26, username=王五, sex=null, birthday=null, address=null]
```

5.6. 小结

5.6.1. #{}和\${}

#{}表示一个占位符号，通过#{}可以实现 preparedStatement 向占位符中设置值，自动进行 java 类型和 jdbc 类型转换。#{}可以有效防止 sql 注入。#{}可以接收简单类型值或 pojo 属性值。如果 parameterType 传输单个简单类型值，#{}括号中可以是 value 或其它名称。

\${}表示拼接 sql 串，通过\${}可以将 parameterType 传入的内容拼接在 sql 中且不进行 jdbc 类型转换，\${}可以接收简单类型值或 pojo 属性值，如果 parameterType 传输单个简单类型值，\${}括号中只能是 value。

5.6.2. parameterType 和 resultType

parameterType: 指定输入参数类型，mybatis 通过 ognl 从输入对象中获取参数值拼接在 sql 中。

resultType: 指定输出结果类型，mybatis 将 sql 查询结果的一行记录数据映射为 resultType 指定类型的对象。如果有多条数据，则分别进行映射，并把对象放到容器 List 中

5.6.3. selectOne 和 selectList

selectOne 查询一条记录，如果使用 selectOne 查询多条记录则抛出异常：

[org.apache.ibatis.exceptions_TOO_MANY_RESULTS](#): Expected one result (or null) to be returned by selectOne(), but found: 3
at
[org.apache.ibatis.session.defaults.DefaultSqlSession.selectOne\(DefaultSqlSession.java:70\)](#)

selectList 可以查询一条或多条记录。

5.7. 实现添加用户

使用的 sql:

```
INSERT INTO `user` (username,birthday,sex,address) VALUES  
('黄忠','2016-07-26','1','三国')
```

5.7.1. 映射文件:

在 User.xml 配置文件中添加如下内容:

```
<!-- 保存用户 -->  
<insert id="saveUser"  
parameterType="com.cloudtcc.mybatis.pojo.User">  
    INSERT INTO `user`  
    (username,birthday,sex,address) VALUES  
    (#{username},#{birthday},#{sex},#{address})  
</insert>
```

5.7.2. 测试程序

MybatisTest 中添加测试方法如下:

```
@Test  
public void testSaveUser() {  
    // 4. 创建 SqlSession 对象  
    SqlSession sqlSession = sqlSessionFactory.openSession();  
  
    // 5. 执行 SqlSession 对象执行保存  
    // 创建需要保存的 User  
    User user = new User();  
    user.setUsername("张飞");  
    user.setSex("1");  
    user.setBirthday(new Date());  
    user.setAddress("蜀国");  
  
    sqlSession.insert("saveUser", user);  
    System.out.println(user);  
  
    // 需要进行事务提交  
    sqlSession.commit();  
  
    // 7. 释放资源  
    sqlSession.close();  
}
```

5.7.3. 效果

```
DEBUG [main] - ==> Preparing: INSERT INTO `user` (username,birthday,sex,address) VALUES (?, ?, ?, ?)
DEBUG [main] - ==> Parameters: 张飞(String), 2016-07-26 22:04:47.495(Timestamp), 1(String), 蜀国(String)
DEBUG [main] - <== Updates: 1
User [id=0, username=张飞, sex=1, birthday=Tue Jul 26 22:04:47 CST 2016, address=蜀国]
```

如上所示，保存成功，但是 id=0，需要解决 id 返回不正常的问题。

5.7.4. mysql 自增主键返回

查询 id 的 sql

```
SELECT LAST_INSERT_ID()
```

通过修改 User.xml 映射文件，可以将 mysql 自增主键返回：

如下添加 selectKey 标签

```
<!-- 保存用户 -->
<insert id="saveUser" parameterType="com.cloudtcc.mybatis.pojo.User">
    <!-- selectKey 标签实现主键返回 -->
    <!-- keyColumn:主键对应的表中的哪一列 -->
    <!-- keyProperty: 主键对应的 pojo 中的哪一个属性 -->
    <!-- order: 设置在执行 insert 语句前执行查询 id 的 sql，孩纸在执行 insert
语句之后执行查询 id 的 sql -->
    <!-- resultType: 设置返回的 id 的类型 -->
    <selectKey keyColumn="id" keyProperty="id" order="AFTER"
        resultType="int">
        SELECT LAST_INSERT_ID()
    </selectKey>
    INSERT INTO `user`
    (username,birthday,sex,address) VALUES
    (#{username},#{birthday},#{sex},#{address})
</insert>
```

LAST_INSERT_ID():是 mysql 的函数,返回 auto_increment 自增列新记录 id 值。

效果如下图所示：

```
DEBUG [main] - ==> Preparing: INSERT INTO `user` (username,birthday,sex,address) VALUES (?, ?, ?, ?)
DEBUG [main] - ==> Parameters: 张飞(String), 2016-07-26 22:06:10.842(Timestamp), 1(String), 蜀国(String)
DEBUG [main] - <== Updates: 1
DEBUG [main] - ==> Preparing: SELECT LAST_INSERT_ID()
DEBUG [main] - ==> Parameters:
DEBUG [main] - <== Total: 1
User [id=48, username=张飞, sex=1, birthday=Tue Jul 26 22:06:10 CST 2016, address=蜀国]
```

返回的 id 为 48，能够正确的返回 id 了。

5.7.5. Mysql 使用 uuid 实现主键

需要增加通过 select uuid()得到 uuid 值

```
<!-- 保存用户 -->
<insert id="saveUser" parameterType="com.cloudtcc.mybatis.pojo.User">
    <!-- selectKey 标签实现主键返回 -->
    <!-- keyColumn:主键对应的表中的哪一列 -->
    <!-- keyProperty: 主键对应的 pojo 中的哪一个属性 -->
    <!-- order: 设置在执行 insert 语句前执行查询 id 的 sql, 孩纸在执行 insert
语句之后执行查询 id 的 sql -->
    <!-- resultType: 设置返回的 id 的类型 -->
    <selectKey keyColumn="id" keyProperty="id" order="BEFORE"
        resultType="string">
        SELECT LAST_INSERT_ID()
    </selectKey>
    INSERT INTO `user`
    (username,birthday,sex,address) VALUES
    (#{username},#{birthday},#{sex},#{address})
</insert>
```

注意这里使用的 order 是 “BEFORE”

5.8. 修改用户

根据用户 id 修改用户名

使用的 sql:

```
UPDATE `user` SET username = '赵云' WHERE id = 26
```

5.8.1. 映射文件

在 User.xml 配置文件中添加如下内容:

```
<!-- 更新用户 -->
<update id="updateUserById"
parameterType="com.cloudtcc.mybatis.pojo.User">
    UPDATE `user` SET
    username = #{username} WHERE id = #{id}
</update>
```

5.8.2. 测试程序

MybatisTest 中添加测试方法如下：

```
@Test
public void testUpdateUserById() {
    // 4. 创建 SqlSession 对象
    SqlSession sqlSession = sqlSessionFactory.openSession();

    // 5. 执行 SqlSession 对象执行更新
    // 创建需要更新的 User
    User user = new User();
    user.setId(26);
    user.setUsername("关羽");
    user.setSex("1");
    user.setBirthday(new Date());
    user.setAddress("蜀国");

    sqlSession.update("updateUserById", user);

    // 需要进行事务提交
    sqlSession.commit();

    // 7. 释放资源
    sqlSession.close();
}
```

5.8.3. 效果

测试效果如下图：

```
DEBUG [main] - ==> Preparing: UPDATE `user` SET username = ? WHERE id = ?
DEBUG [main] - ==> Parameters: 关羽(String), 26(Integer)
DEBUG [main] - <==      Updates: 1
```

5.9. 删除用户

根据用户 id 删除用户

使用的 sql

DELETE FROM `user` WHERE id = 47

5.9.1. 映射文件:

在 User.xml 配置文件中添加如下内容:

```
<!-- 删除用户 -->
<delete id="deleteUserById" parameterType="int">
    delete from user where
    id=#{id}
</delete>
```

5.9.2. 测试程序:

MybatisTest 中添加测试方法如下:

```
@Test
public void testDeleteUserById() {
    // 4. 创建 SqlSession 对象
    SqlSession sqlSession = sqlSessionFactory.openSession();

    // 5. 执行 SqlSession 对象执行删除
    sqlSession.delete("deleteUserById", 48);

    // 需要进行事务提交
    sqlSession.commit();

    // 7. 释放资源
    sqlSession.close();
}
```

5.9.3. 效果

测试效果如下图:



```
DEBUG [main] - ==> Preparing: delete from user where id=?
DEBUG [main] - ==> Parameters: 48(Integer)
DEBUG [main] - <== Updates: 1
DEBUG [main] - Committing JDBC Connection [com.mysql.jdbc.J
```

5.10. Mybatis 解决 jdbc 编程的问题

- 1、数据库连接创建、释放频繁造成系统资源浪费从而影响系统性能，如果使用数据库连接池可解决此问题。

解决：在 SqlMapConfig.xml 中配置数据连接池，使用连接池管理数据库链接。

- 2、Sql 语句写在代码中造成代码不易维护，实际应用 sql 变化的可能较大，sql 变动需要改变 java 代码。

解决：将 Sql 语句配置在 XXXXmapper.xml 文件中与 java 代码分离。

- 3、向 sql 语句传参数麻烦，因为 sql 语句的 where 条件不一定，可能多也可能少，占位符需要和参数一一对应。

解决：Mybatis 自动将 java 对象映射至 sql 语句，通过 statement 中的 parameterType 定义输入参数的类型。

- 4、对结果集解析麻烦，sql 变化导致解析代码变化，且解析前需要遍历，如果能将数据库记录封装成 pojo 对象解析比较方便。

解决：Mybatis 自动将 sql 执行结果映射至 java 对象，通过 statement 中的 resultType 定义输出结果的类型。

5.11. mybatis 与 hibernate 不同

Mybatis 和 hibernate 不同，它不完全是一个 ORM 框架，因为 **MyBatis 需要程序员自己编写 Sql 语句**。mybatis 可以通过 XML 或注解方式灵活配置要运行的 sql 语句，并将 java 对象和 sql 语句映射生成最终执行的 sql，最后将 sql 执行的结果再映射生成 java 对象。

Mybatis 学习门槛低，简单易学，程序员直接编写原生态 sql，可严格控制 sql 执行性能，灵活度高，非常适合对关系数据模型要求不高的软件开发，例如互联网软件、企业运营类软件等，因为这类软件需求变化频繁，一旦需求变化要求成果输出迅速。但是灵活的前提是 **mybatis 无法做到数据库无关性**，如果需要实现支持多种数据库的软件则需要自定义多套 sql 映射文件，工作量大。

Hibernate 对象/关系映射能力强，**数据库无关性好**，对于关系模型要求高的软件（例如需求固定的定制化软件）如果用 hibernate 开发可以节省很多代码，提高效率。但是 Hibernate 的学习门槛高，要精通门槛更高，而且怎么设计 O/R 映射，在性能和对象模型之间如何权衡，以及怎样用好 Hibernate 需要具有很强的经验和能力才行。

总之，按照用户的需求在有限的资源环境下只要能做出维护性、扩展性良好的软件架构都是好架构，所以框架只有适合才是最好。

6. Dao 开发方法

使用 MyBatis 开发 Dao，通常有两个方法，即原始 Dao 开发方法和 Mapper 动态代理开发方法。

6.1. 需求

使用 MyBatis 开发 DAO 实现以下的功能：

根据用户 id 查询一个用户信息

根据用户名称模糊查询用户信息列表

添加用户信息

6.2. SqlSession 的使用范围

SqlSession 中封装了对数据库的操作，如：查询、插入、更新、删除等。

SqlSession 通过 SqlSessionFactory 创建。

SqlSessionFactory 是通过 SqlSessionFactoryBuilder 进行创建。

6.2.1. SqlSessionFactoryBuilder

SqlSessionFactoryBuilder 用于创建 SqlSessionFactory，SqlSessionFactory 一旦创建完成就不需要 SqlSessionFactoryBuilder 了，因为 SqlSession 是通过 SqlSessionFactory 创建的。所以可以将 SqlSessionFactoryBuilder 当成一个工具类使用，最佳使用范围是方法范围即方法体内局部变量。

6.2.2. SqlSessionFactory

SqlSessionFactory 是一个接口，接口中定义了 openSession 的不同重载方法，SqlSessionFactory 的最佳使用范围是整个应用运行期间，一旦创建后可以重复使用，通常以单例模式管理 SqlSessionFactory。

6.2.3. SqlSession

SqlSession 是一个面向用户的接口，sqlSession 中定义了数据库操作方法。

每个线程都应该有它自己的 SqlSession 实例。SqlSession 的实例不能共享使用，它也是线程不安全的。因此最佳的范围是请求或方法范围。绝对不能将 SqlSession 实例的引用放在一个类的静态字段或实例字段中。

打开一个 `SqlSession`；使用完毕就要关闭它。通常把这个关闭操作放到 `finally` 块中以确保每次都能执行关闭。如下：

```
SqlSession session = sqlSessionFactory.openSession();
try {
    // do work
} finally {
    session.close();
}
```

6.3. 原始 Dao 开发方式

原始 Dao 开发方法需要程序员编写 Dao 接口和 Dao 实现类。

6.3.1. 映射文件

编写映射文件如下：（也可以使用入门程序完成的映射文件）

```
<?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">
<!-- namespace: 命名空间，用于隔离 sql，还有一个很重要的作用，后面会讲 -->
<mapper namespace="test">

    <!-- 根据 id 查询用户 -->
    <select id="queryUserById" parameterType="int"
        resultType="com.cloudtcc.mybatis.pojo.User">
        select * from user where id = #{id}
    </select>

    <!-- 根据 username 模糊查询用户 -->
    <select id="queryUserByUsername" parameterType="string"
        resultType="com.cloudtcc.mybatis.pojo.User">
        select * from user where username like '%${value}%'
    </select>

    <!-- 保存用户 -->
    <insert id="saveUser"
parameterType="com.cloudtcc.mybatis.pojo.User">
        <selectKey keyProperty="id" keyColumn="id" order="AFTER"
            resultType="int">
            SELECT LAST_INSERT_ID()
        </selectKey>
        insert into user(username,birthday,sex,address)
        values(#{username},#{birthday},#{sex},#{address})
    </insert>
</mapper>
```

```
</insert>

</mapper>
```

6.3.2. Dao 接口

先进行 DAO 的接口开发，编码如下：

```
public interface UserDao {
    /**
     * 根据 id 查询用户
     *
     * @param id
     * @return
     */
    User queryUserById(int id);

    /**
     * 根据用户名模糊查询用户
     *
     * @param username
     * @return
     */
    List<User> queryUserByUsername(String username);

    /**
     * 保存用户
     *
     * @param user
     */
    void saveUser(User user);
}
```

6.3.3. Dao 实现类

编写的 Dao 实现类如下

```
public class UserDaoImpl implements UserDao {
    private SqlSessionFactory sqlSessionFactory;

    public UserDaoImpl(SqlSessionFactory sqlSessionFactory) {
        super();
        this.sqlSessionFactory = sqlSessionFactory;
    }
}
```

```

@Override
public User queryUserById(int id) {
    // 创建 SqlSession
    SqlSession sqlSession = this.sqlSessionFactory.openSession();
    // 执行查询逻辑
    User user = sqlSession.selectOne("queryUserById", id);
    // 释放资源
    sqlSession.close();

    return user;
}

@Override
public List<User> queryUserByUsername(String username) {
    // 创建 SqlSession
    SqlSession sqlSession = this.sqlSessionFactory.openSession();

    // 执行查询逻辑
    List<User> list = sqlSession.selectList("queryUserByUsername",
username);
    // 释放资源
    sqlSession.close();
    return list;
}

@Override
public void saveUser(User user) {
    // 创建 SqlSession
    SqlSession sqlSession = this.sqlSessionFactory.openSession();

    // 执行保存逻辑
    sqlSession.insert("saveUser", user);
    // 提交事务
    sqlSession.commit();
    // 释放资源
    sqlSession.close();
}
}

```

6.3.4. Dao 测试

创建一个 JUnit 的测试类，对 UserDao 进行测试，测试代码如下：

```

public class UserDaoTest {

```

```

private SqlSessionFactory sqlSessionFactory;

@Before
public void init() throws Exception {
    // 创建 SqlSessionFactoryBuilder
    SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new
SqlSessionFactoryBuilder();
    // 加载 SqlMapConfig.xml 配置文件
    InputStream inputStream =
Resources.getResourceAsStream("SqlMapConfig.xml");
    // 创建 SqlSessionFactory
    this.sqlSessionFactory =
sqlSessionFactoryBuilder.build(inputStream);
}

@Test
public void testQueryUserById() {
    // 创建 DAO
    UserDao userDao = new UserDaoImpl(this.sqlSessionFactory);
    // 执行查询
    User user = userDao.queryUserById(1);
    System.out.println(user);
}

@Test
public void testQueryUserByUsername() {
    // 创建 DAO

    UserDao userDao = new UserDaoImpl(this.sqlSessionFactory);
    // 执行查询
    List<User> list = userDao.queryUserByUsername("张");
    for (User user : list) {
        System.out.println(user);
    }
}

@Test
public void testSaveUser() {
    // 创建 DAO
    UserDao userDao = new UserDaoImpl(this.sqlSessionFactory);

    // 创建保存对象
    User user = new User();
    user.setUsername("刘备");
}

```

```
        user.setBirthday(new Date());
        user.setSex("1");
        user.setAddress("蜀国");
        // 执行保存
        userDao.saveUser(user);

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

6.3.5. 问题

原始 Dao 开发中存在以下问题：

- ◆ Dao 方法体存在重复代码：通过 SqlSessionFactory 创建 SqlSession，调用 SqlSession 的数据库操作方法
- ◆ 调用 sqlSession 的数据库操作方法需要指定 statement 的 id，这里存在硬编码，不得于开发维护。

6.4. Mapper 动态代理方式

6.4.1. 开发规范

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。

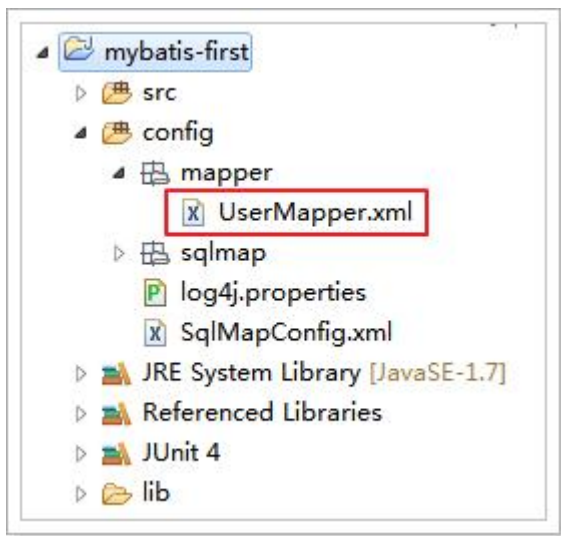
Mapper 接口开发需要遵循以下规范：

- 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

6.4.2. Mapper.xml(映射文件)

定义 mapper 映射文件 UserMapper.xml

将 UserMapper.xml 放在 config 下 mapper 目录下，效果如下：



UserMapper.xml 配置文件内容：

```
<?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，用于隔离 sql -->
<!-- 还有一个很重要的作用，使用动态代理开发 DAO，1. namespace 必须和 Mapper
接口类路径一致 -->
<mapper namespace="com.cloudtcc.mybatis.mapper.UserMapper">
    <!-- 根据用户 id 查询用户 -->
    <!-- 2. id 必须和 Mapper 接口方法名一致 -->
    <!-- 3. parameterType 必须和接口方法参数类型一致 -->
    <!-- 4. resultType 必须和接口方法返回值类型一致 -->
    <select id="queryUserById" parameterType="int"
        resultType="com.cloudtcc.mybatis.pojo.User">
        select * from user where id = #{id}
    </select>

    <!-- 根据用户名查询用户 -->
    <select id="queryUserByUsername" parameterType="string"
        resultType="com.cloudtcc.mybatis.pojo.User">
        select * from user where username like '%${value}%'
    </select>

    <!-- 保存用户 -->
```

```

    <insert id="saveUser"
parameterType="com.cloudtcc.mybatis.pojo.User">
    <selectKey keyProperty="id" keyColumn="id" order="AFTER"
        resultType="int">
        select last_insert_id()
    </selectKey>
    insert into user(username,birthday,sex,address) values
    (#{username},#{birthday},#{sex},#{address});
</insert>

</mapper>

```

6.4.3. UserMapper(接口文件)

创建 UserMapper 接口代码如下：

```

public interface UserMapper {
    /**
     * 根据 id 查询
     *
     * @param id
     * @return
     */
    User queryUserById(int id);

    /**
     * 根据用户名查询用户
     *
     * @param username
     * @return
     */
    List<User> queryUserByUsername(String username);

    /**
     * 保存用户
     *
     * @param user
     */
    void saveUser(User user);
}

```


6.4.4. 加载 UserMapper.xml 文件

修改 SqlMapConfig.xml 文件，添加以下所示的内容：

```
<!-- 加载映射文件 -->
<mappers>
    <mapper resource="sqlmap/User.xml" />
    <mapper resource="mapper/UserMapper.xml" />
</mappers>
```

6.4.5. 测试

编写的测试方法如下：

```
public class UserMapperTest {
    private SqlSessionFactory sqlSessionFactory;

    @Before
    public void init() throws Exception {
        // 创建 SqlSessionFactoryBuilder
        SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new
        SqlSessionFactoryBuilder();
        // 加载 SqlMapConfig.xml 配置文件
        InputStream inputStream =
        Resources.getResourceAsStream("SqlMapConfig.xml");
        // 创建 SqlSessionFactory
        this.sqlSessionFactory =
        sqlSessionFactoryBuilder.build(inputStream);
    }

    @Test
    public void testQueryUserById() {
        // 获取 sqlSession, 和 spring 整合后由 spring 管理
        SqlSession sqlSession = this.sqlSessionFactory.openSession();

        // 从 sqlSession 中获取 Mapper 接口的代理对象
        UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
        // 执行查询方法
        User user = userMapper.queryUserById(1);
        System.out.println(user);

        // 和 spring 整合后由 spring 管理
        sqlSession.close();
    }
}
```

```

@Test
public void testQueryUserByUsername() {
    // 获取 sqlSession, 和 spring 整合后由 spring 管理
    SqlSession sqlSession = this.sqlSessionFactory.openSession();

    // 从 sqlSession 中获取 Mapper 接口的代理对象
    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
    // 执行查询方法
    List<User> list = userMapper.queryUserByUsername("张");
    for (User user : list) {
        System.out.println(user);
    }

    // 和 spring 整合后由 spring 管理
    sqlSession.close();
}

@Test
public void testSaveUser() {
    // 获取 sqlSession, 和 spring 整合后由 spring 管理
    SqlSession sqlSession = this.sqlSessionFactory.openSession();

    // 从 sqlSession 中获取 Mapper 接口的代理对象
    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
    // 创建保存对象
    User user = new User();
    user.setUsername("刘备");
    user.setBirthday(new Date());
    user.setSex("1");
    user.setAddress("蜀国");
    // 执行查询方法
    userMapper.saveUser(user);
    System.out.println(user);

    // 和 spring 整合后由 spring 管理
    sqlSession.commit();
    sqlSession.close();
}
}

```

6.4.6. 小结

◆ selectOne 和 selectList

动态代理对象调用 `sqlSession.selectOne()` 和 `sqlSession.selectList()` 是根据 `mapper` 接口方法的返回值决定，如果返回 `list` 则调用 `selectList` 方法，如果返回单个对象则调用 `selectOne` 方法。

◆ namespace

mybatis 官方推荐使用 `mapper` 代理方法开发 `mapper` 接口，程序员不用编写 `mapper` 接口实现类，使用 `mapper` 代理方法时，输入参数可以使用 `pojo` 包装对象或 `map` 对象，保证 `dao` 的通用性。

7. SqlMapConfig.xml 配置文件

7.1. 配置内容

`SqlMapConfig.xml` 中配置的内容和顺序如下：

properties（属性）

settings（全局配置参数）

typeAliases（类型别名）

typeHandlers（类型处理器）

objectFactory（对象工厂）

plugins（插件）

environments（环境集合属性对象）

environment（环境子属性对象）

transactionManager（事务管理）

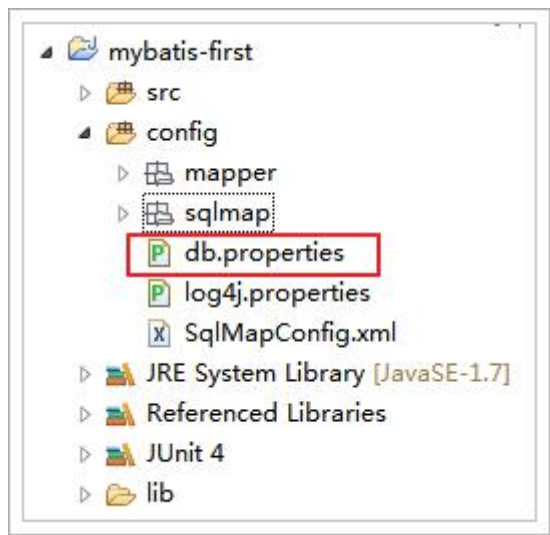
dataSource（数据源）

mappers（映射器）

7.2. properties（属性）

SqlMapConfig.xml 可以引用 java 属性文件中的配置信息如下：

在 config 下定义 db.properties 文件，如下所示：



db.properties 配置文件内容如下：

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/mybatis?characterEncoding
=utf-8
jdbc.username=root
jdbc.password=root
```

SqlMapConfig.xml 引用如下：

```
<?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>
  <!-- 是用 resource 属性加载外部配置文件 -->
  <properties resource="db.properties">
    <!-- 在 properties 内部用 property 定义属性 -->
    <!-- 如果外部配置文件有该属性，则内部定义属性被外部属性覆盖 -->
    <property name="jdbc.username" value="root123" />
    <property name="jdbc.password" value="root123" />
  </properties>

  <!-- 和 spring 整合后 environments 配置将废除 -->
```

```

<environments default="development">
  <environment id="development">
    <!-- 使用 jdbc 事务管理 -->
    <transactionManager type="JDBC" />
    <!-- 数据库连接池 -->
    <dataSource type="POOLED">
      <property name="driver" value="${jdbc.driver}" />
      <property name="url" value="${jdbc.url}" />
      <property name="username" value="${jdbc.username}" />
      <property name="password" value="${jdbc.password}" />
    </dataSource>
  </environment>
</environments>

<!-- 加载映射文件 -->
<mappers>
  <mapper resource="sqlmap/User.xml" />
  <mapper resource="mapper/UserMapper.xml" />
</mappers>
</configuration>

```

注意： MyBatis 将按照下面的顺序来加载属性：

- ◆ 在 `properties` 元素体内定义的属性首先被读取。
- ◆ 然后会读取 `properties` 元素中 `resource` 或 `url` 加载的属性，它会覆盖已读取的同名属性。

7.3. typeAliases（类型别名）

7.3.1. mybatis 支持别名：

别名	映射的类型
<code>_byte</code>	<code>byte</code>
<code>_long</code>	<code>long</code>
<code>_short</code>	<code>short</code>
<code>_int</code>	<code>int</code>
<code>_integer</code>	<code>int</code>
<code>_double</code>	<code>double</code>
<code>_float</code>	<code>float</code>
<code>_boolean</code>	<code>boolean</code>
<code>string</code>	<code>String</code>

byte	Byte
long	Long
short	Short
int	Integer
integer	Integer
double	Double
float	Float
boolean	Boolean
date	Date
decimal	BigDecimal
bigdecimal	BigDecimal
map	Map

7.3.2. 自定义别名:

在 SqlMapConfig.xml 中配置如下:

```
<?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>
    <!-- 是用 resource 属性加载外部配置文件 -->
    <properties resource="db.properties">
        <!-- 在 properties 内部用 property 定义属性 -->
        <property name="jdbc.username" value="root123" />
        <property name="jdbc.password" value="root123" />
    </properties>

    <typeAliases>
        <!-- 单个别名定义 -->
        <typeAlias alias="user" type="com.cloudtcc.mybatis.pojo.User" />
    </typeAliases>

    <!-- 批量别名定义，扫描整个包下的类，别名为类名（大小写不敏感） -->
    <package name="com.cloudtcc.mybatis.pojo" />
    <package name="其它包" />
</typeAliases>

    <!-- 和 spring 整合后 environments 配置将废除 -->
    <environments default="development">
        <environment id="development">
            <!-- 使用 jdbc 事务管理 -->
```

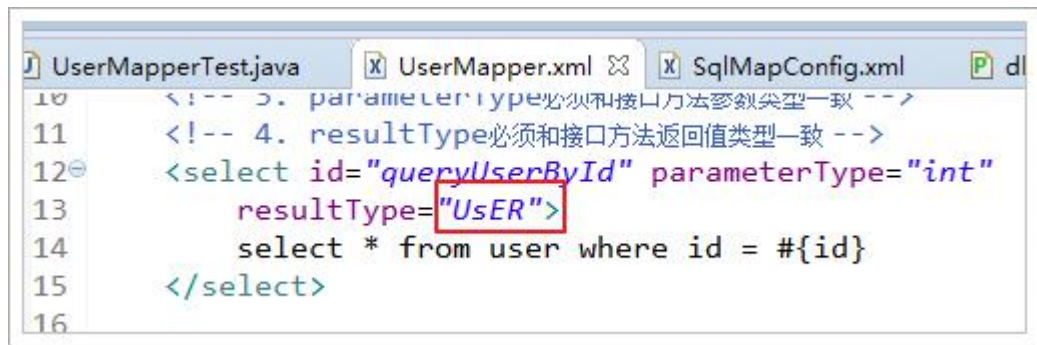
```

<transactionManager type="JDBC" />
<!-- 数据库连接池 -->
<dataSource type="POOLED">
    <property name="driver" value="${jdbc.driver}" />
    <property name="url" value="${jdbc.url}" />
    <property name="username" value="${jdbc.username}" />
    <property name="password" value="${jdbc.password}" />
</dataSource>
</environment>
</environments>

<!-- 加载映射文件 -->
<mapper>
    <mapper resource="sqlmap/User.xml" />
    <mapper resource="mapper/UserMapper.xml" />
</mapper>
</configuration>

```

在 mapper.xml 配置文件中，就可以使用设置的别名了
别名大小写不敏感



7.4. mappers（映射器）

Mapper 配置的几种方法：

7.4.1. <mapper resource=" " />

使用相对于类路径的资源（现在的使用方式）

如：<mapper resource="sqlmap/User.xml" />

7.4.2. <mapper class=" " />

使用 mapper 接口类路径

如：<mapper class="com.cloudtcc.mybatis.mapper.UserMapper"/>

注意：此种方法要求 **mapper** 接口名称和 **mapper** 映射文件名称相同，且放在同一个目录中。

7.4.3. <package name="" />

注册指定包下的所有 mapper 接口

如：<package name="com.cloudtcc.mybatis.mapper"/>

注意：此种方法要求 **mapper** 接口名称和 **mapper** 映射文件名称相同，且放在同一个目录中。