

MyBatis

- 文件关系 ——> 环境搭建
- 类关系 ——> 使用 | 原理

<https://mybatis.org/mybatis-3/zh/getting-started.html>

概述

- mybatis是一个持久层框架，用java编写的。
- 封装了jdbc操作的很多细节
 - 使开发者只需要关注sql语句本身
 - 无需关注注册驱动，创建连接等繁杂过程
- 使用了ORM思想，实现了结果集的封装
 - 解决了实体和数据库映射的问题，对jdbc进行了封装，屏蔽了jdbc api底层访问细节
- ORM：(Object Relational Mapping 对象关系映射)
- 把数据库表和实体类及实体类的属性对应起来
- 我们可以操作实体类就实现操作数据库表

user表	User类
id	userId
user_name	userName

- 需要做到实体类中的属性和数据库表的字段名称保持一致。
- 通过xml或注解的方式将要执行的各种statement配置起来
 - 每一个操作就是一个statement，其实就是对应的SQL语句
- 通过java对象和statement中sql的动态参数进行映射生成最终执行的sql语句
 - 占位符替换
- 最后由mybatis框架执行sql
 - 将结果映射为java对象并返回

三层架构

表现层

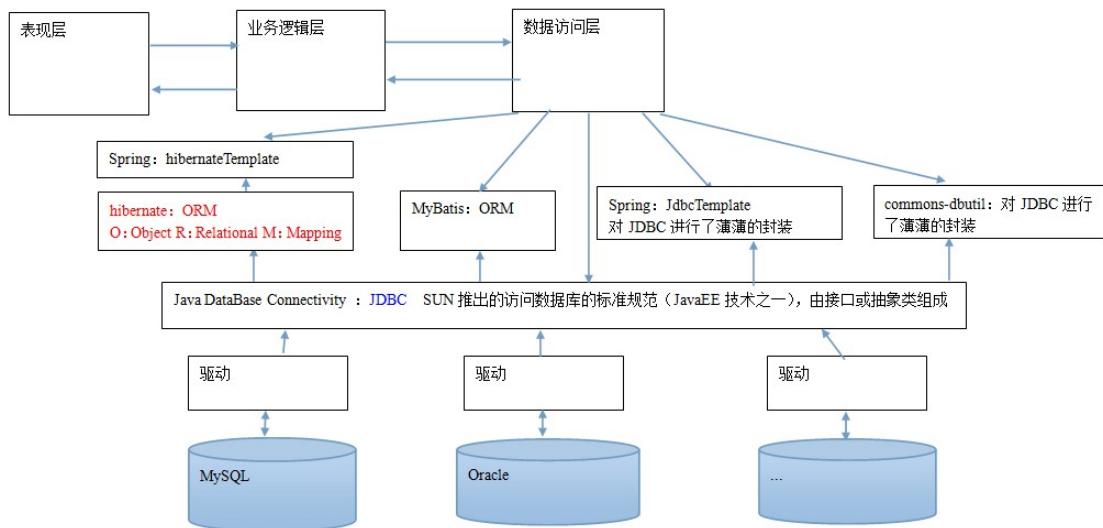
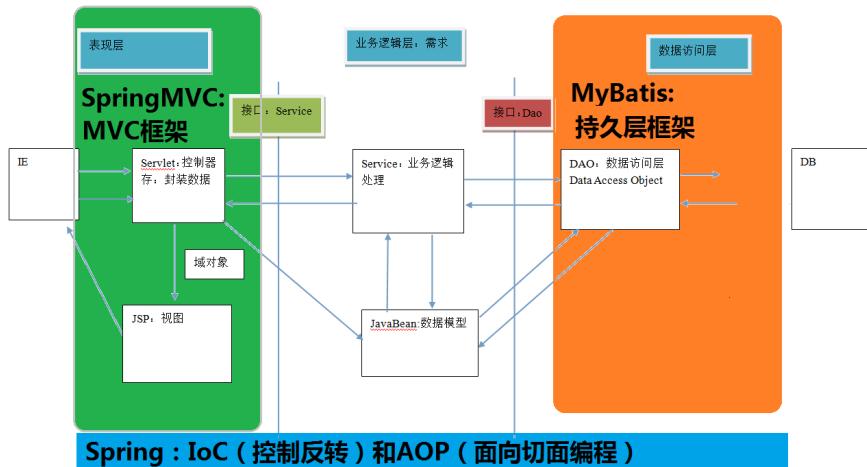
- 展示数据

业务层

- 处理业务

持久层

- 和数据库交互



持久层技术解决方案

- JDBC是规范。
- Spring的JdbcTemplate 和 Apache的DBUtils 是工具类。
- 以上都不是框架。

JDBC技术

- Connection
- PreparedStatement
- ResultSet

Spring的JdbcTemplate

- Spring中对 JDBC 的简单封装

Apache的DBUtils

- 对 JDBC 的简单封装

JDBC编程分析

jdbc程序

```
1 public static void main(String[] args) {
2     Connection connection = null;
3     PreparedStatement preparedStatement = null;
4     ResultSet resultSet = null;
5     try {
6         //加载数据库驱动
7         Class.forName("com.mysql.jdbc.Driver");
8         //通过驱动管理类获取数据库链接
9         connection =
10        DriverManager.getConnection("jdbc:mysql://localhost:3306/mybatis?
11        characterEncoding=utf-8","root", "root");
12         //定义 sql 语句 ?表示占位符
13         String sql = "select * from user where username = ?";
14         //获取预处理 statement
15         preparedStatement = connection.prepareStatement(sql);
16         //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
17         //的参数值
18         preparedStatement.setString(1, "王五");
19         //向数据库发出 sql 执行查询, 查询出结果集
20         resultSet = preparedStatement.executeQuery();
21         //遍历查询结果集
22         while(resultSet.next()){
23             System.out.println(resultSet.getString("id") +
24             resultSet.getString("username"));
25         }
26     } catch (Exception e) {
27         e.printStackTrace();
28     } finally {
29         //释放资源
30         if(resultSet!=null){
31             try {
32                 resultSet.close();
33             } catch (SQLException e) {
34                 e.printStackTrace();
35             }
36         }
37         if(preparedStatement!=null){
38             try {
39                 preparedStatement.close();
40             } catch (SQLException e) {
```

```

37         e.printStackTrace();
38     }
39   }
40   if(connection!=null){
41     try {
42       connection.close();
43     } catch (SQLException e) {
44       // TODO Auto-generated catch block
45       e.printStackTrace();
46     }
47   }
48 }
49 }
```

jdbc问题

问题：

- 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。
 - 在 SqlMapConfig.xml 中配置数据链接池，使用连接池管理数据库链接。
- Sql 语句在代码中 硬编码，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。
 - 将 Sql 语句配置在 XXXXmapper.xml 文件中与 java 代码分离。
- 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 数对应。使用 preparedStatement 向 占有位符号 传参数存在 硬编码
 - 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。
- 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。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。
 - 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。

快速入门

- 创建 maven工程 并 导入 pom坐标
- 创建 实体类 和 dao的接口
- 创建 Mybatis的 主配置文件
 - SqlMaoConfig.xml
 - 指定映射 位置，每个dao独有一个
- 创建 映射配置 文件
 - xml方式
 - UserDao.xml
 - 注解方式
 - UserDao + 注解
 - 实现类

- 传递 factory给构造方法
- session 调用 操作数据方法

新建maven工程

- 什么都不要选
 - 最简单的maven工程即可
- archetype
 - 选择普通的java工程

pom文件依赖

- mybatis
- mysql-connection-java
- (log4j) 日志
- (junit)

可以去官网去找。<https://mybatis.org/mybatis-3/zh/getting-started.html>

- 入门中有pom坐标

```
1 <dependency>
2   <groupId>org.mybatis</groupId>
3   <artifactId>mybatis</artifactId>
4   <version>3.5.1</version>
5 </dependency>
6
7 <dependency>
8   <groupId>mysql</groupId>
9   <artifactId>mysql-connector-java</artifactId>
10  <version>8.0.15</version>
11 </dependency>
12
13 <dependency>
14   <groupId>log4j</groupId>
15   <artifactId>log4j</artifactId>
16   <version>1.2.12</version>
17 </dependency>
```

建包&文件？？？

- 没有说明项目结构
- 映射文件不是必须的，这里有歧义，放到xml实现那里吧
- 这就写写实体类

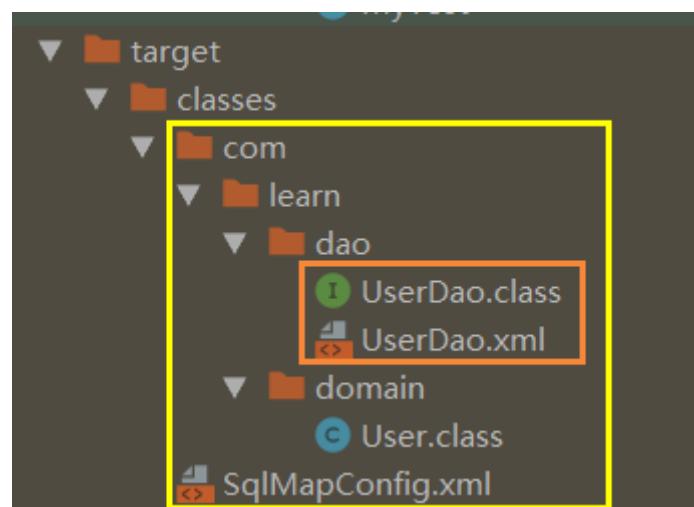
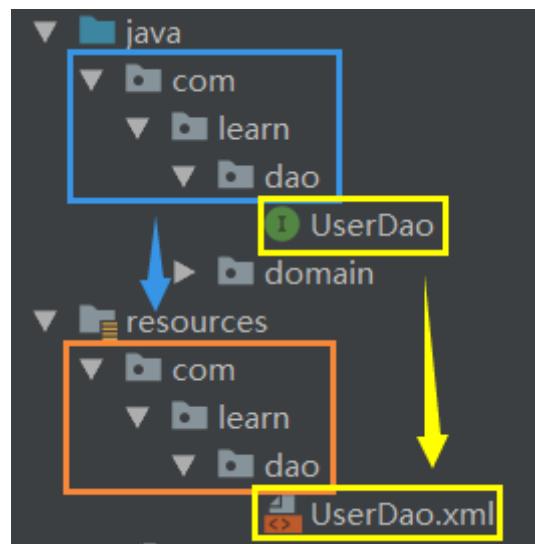
映射文件UserDao.xml

- resources 新建 file

- 使用注解时 不能有此文件

要求

- 创建位置：必须和持久层接口在相同的包中。
 - dao接口：`java/com.learn.dao.UserDao`
 - 映射文件：`resources/com/learn/dao/UserDao.xml`
- 名称：必须以持久层接口名称命名文件名，扩展名是.xml
 - dao接口：`java/com.learn.dao.UserDao`
 - 映射文件：`resources/com/learn/dao/UserDao.xml`



- 映射配置文件的 mapper标签 namespace属性 的取值 必须是
 - 接口的全限定类名
- 映射配置文件的操作配置（ select标签 ），id属性必须是dao接口的方法名

文件约束

```

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

完整xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE mapper
3     PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4     "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5
6 <mapper namespace="com.learn.dao.UserDao"><!-- UserDao接口的全限定类名 -->
7     <!-- 配置查询所有 id是方法名 -->
8     <select id="findAll" resultType="com.learn.domain.User">
9         <!-- SQL语句 -->
10        select * from user;
11    </select>
12 </mapper>
```

主配置文件 SqlMapConfig.xml

- resources 新建 file
- 叫什么都行
 - 习惯上 SqlMapConfig.xml
- configuration
 - environments
 - environment
 - transactionManage : 事务
 - dataSource
 - property
 - dirver、url、username、password
 - mappers
 - mapper
 - "resource" : 写映射xml文件位置
 - "class" : 写接口的文件位置
 - "package"

文件约束

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE configuration
3     PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4     "http://mybatis.org/dtd/mybatis-3-config.dtd">
```

完整xml

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE configuration
3     PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4     "http://mybatis.org/dtd/mybatis-3-config.dtd">
5
6 <!-- mybatis的主配置文件:和环境、连数据库相关的 -->
7 <configuration>
8     <!-- 配置mysql环境 -->
9     <environments default="mysql"><!-- 起个名 -->
10    <!-- 配置mysql的环境 -->
11    <environment id="mysql"><!-- 上面那名 -->
12        <!-- 配置事务的类型 -->
13        <transactionManager type="JDBC"></transactionManager>
14        <!-- 配置数据源(连接池) 连接数据库的信息 -->
15        <dataSource type="POOLED"><!-- 取值有3个 -->
16            <!-- 配置连接数据库的4个基本信息 -->
17            <property name="driver" value="com.mysql.cj.jdbc.Driver"/>
18            <property name="url"
value="jdbc:mysql://localhost:3306/mybatis"/>
19            <property name="username" value="root"/>
20            <property name="password" value="xxxx"/>
21        </dataSource>
22    </environment>
23 </environments>
24
25 <!-- 指定映射配置文件的位置 告知mybatis映射配置的位置 -->
26 <!-- 映射配置文件指的是每个dao独立的配置文件 -->
27 <mappers>
28     <mapper resource="com/learn/dao/UserDao.xml"></mapper>
29 </mappers>
30 </configuration>

```

Navicat导入sql建表

user表

名	类型	长度	小数点	不是 null	虚拟	键	注释
id	int	0	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>	 1	
username	varchar	32	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>		用户名
birthday	datetime	0	0	<input type="checkbox"/>	<input type="checkbox"/>		生日
sex	char	1	0	<input type="checkbox"/>	<input type="checkbox"/>		性别
address	varchar	256	0	<input type="checkbox"/>	<input type="checkbox"/>		地址

```

1 CREATE TABLE `user` (
2     `id` int NOT NULL AUTO_INCREMENT,
3     `username` varchar(32) NOT NULL COMMENT '用户名',
4     `birthday` datetime DEFAULT NULL COMMENT '生日',
5     `sex` char(1) DEFAULT NULL COMMENT '性别',
6     `address` varchar(256) DEFAULT NULL COMMENT '地址',
7     PRIMARY KEY (`id`)
8 ) ENGINE=InnoDB AUTO_INCREMENT=49 DEFAULT CHARSET=utf8;

```

注意事项

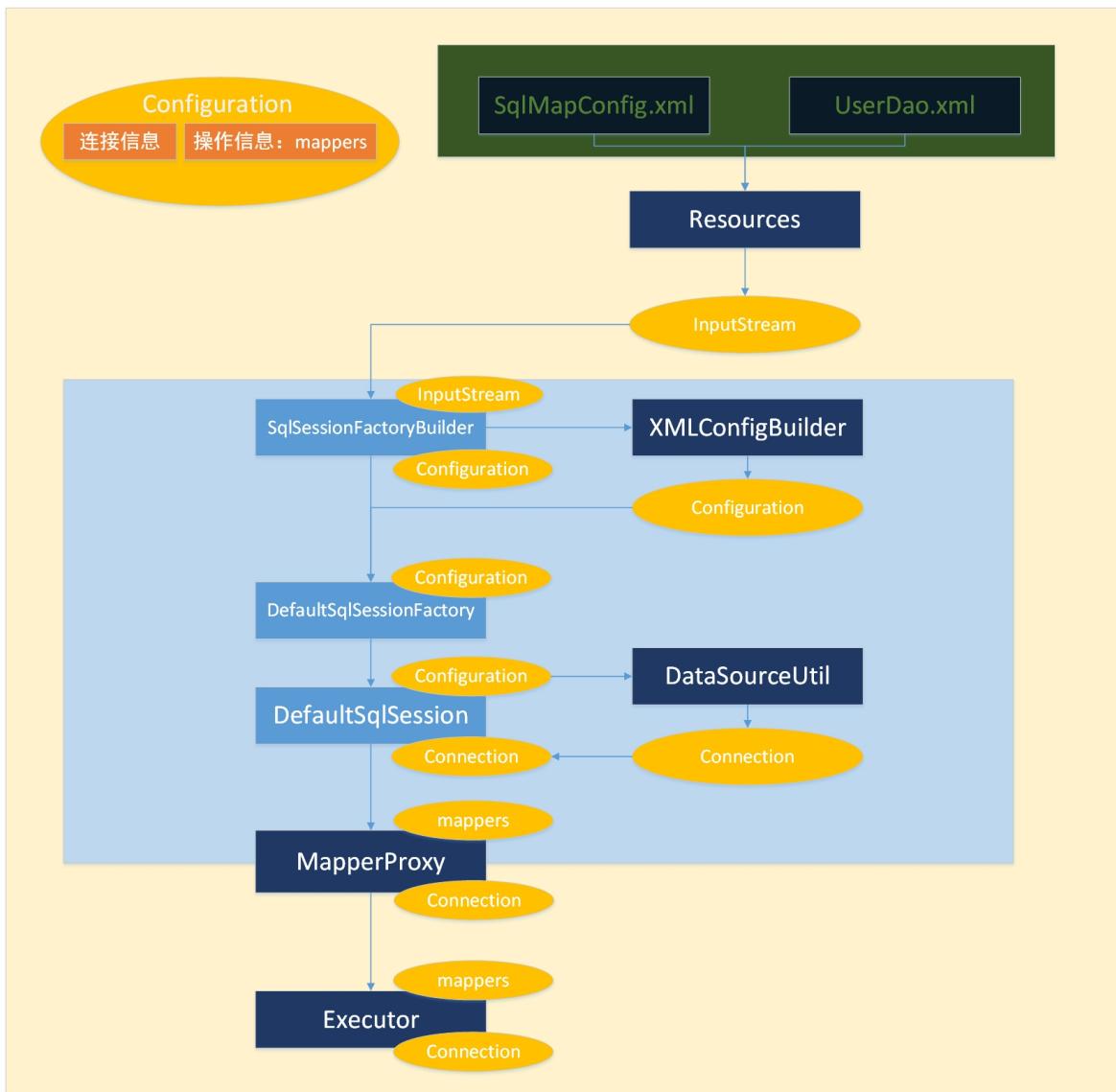
习惯惯例

- 创建UserDao.xml 和 UserDao.java 时名称 是为了和我们之前的知识保持一致。
 - 在Mybatis中它把持久层的操作接口名称和映射文件也叫做：Mapper。
 - 所以，UserDao 和 UserMapper是一样的。
- 在idea中创建 目录 的时候，它和 包 是不一样的。
 - 包在创建时：com.learn.dao是三级结构
 - 目录在创建时：com.learn.dao是一级目录，所以要逐级，一目录一目录的创建

必须遵守

- mybatis的映射配置文件位置必须和dao接口的包结构相同
- 映射配置文件的mapper标签namespace属性的取值必须是接口的全限定类名
- 映射配置文件的操作配置，id属性必须是dao接口的方法名

使用示例



	xml&注解	实现类
Resource	测试类	测试类
SqlSessionFactoryBuilder	测试类	测试类
SqlSessionFactory	测试类	测试类 -> 实现类(factory)
SqlSession	测试类	实现类

UserDao接口

```
1 /**
2  * 用户的持久层接口
3 */
4 public interface UserDao {
5
6     /**
7      * 查询所有
8      * @return
9      */
10    List<User> findAll();
11
12 }
```

xml方式

映射配置 | 操作信息(SQL、参数返回值类型) 由XML文件配置

```
1 <mapper resource="com/learn/dao/UserDao.xml">
```

配置xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE configuration
3   PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4   "http://mybatis.org/dtd/mybatis-3-config.dtd">
5
6 <!-- mybatis的主配置文件:和环境、连数据库相关的 -->
7 <configuration>
8   <!-- 配置mysql环境 -->
9   <environments default="mysql"><!-- 起个名 -->
10  <!-- 配置mysql的环境 -->
11  <environment id="mysql"><!-- 上面那名 -->
12    <!-- 配置事务的类型 -->
13    <transactionManager type="JDBC"></transactionManager>
14    <!-- 配置数据源(连接池) 连接数据库的信息 -->
15    <dataSource type="POOLED"><!-- 取值有3个 -->
16      <!-- 配置连接数据库的4个基本信息 -->
17      <property name="driver" value="com.mysql.cj.jdbc.Driver"/>
18      <property name="url"
19        value="jdbc:mysql://localhost:3306/mybatis"/>
20      <property name="username" value="root"/>
21      <property name="password" value="xxxx"/>
22    </dataSource>
23  </environment>
24 </environments>
25
26 <!-- 指定映射配置文件的位置 告知mybatis映射配置的位置 -->
27 <!-- 映射配置文件指的是每个dao独立的配置文件 -->
28 <mappers>
29   <mapper resource="com/learn/dao/UserDao.xml"></mapper>
30 </mappers>
31 </configuration>
```

映射xml

- namespace
 - 接口的全限定类名
- 须在映射配置中告知 mybatis要封装到哪个实体类中
 - 指定实体类的全限定类名

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE mapper
3     PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4     "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5
6 <mapper namespace="com.learn.dao.UserDao"><!-- UserDao接口的全限定类名 -->
7     <!-- 配置查询所有 id是方法名 -->
8     <select id="findAll" resultType="com.learn.domain.User">
9         <!-- SQL语句 -->
10        select * from user;
11    </select>
12 </mapper>
```

测试类

- 读取配置文件
- 创建 SqlSessionFactory 工厂
- 使用工厂生产 SqlSession 对象
- 使用SqlSession 创建Dao接口的 代理对象
- 使用代理对象执行方法
- 释放资源

```
1 public class MyTest {
2
3     public static void main(String[] args) throws Exception {
4         //1.读取配置文件
5         InputStream in = Resources.getResourceAsStream("SqlMapConfig.xml");
6         //2.创建SqlSessionFactory工厂
7         SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
8         SqlSessionFactory factory = builder.build(in);
9         //3.使用工厂生产SqlSession对象
10        SqlSession session = factory.openSession();
11        //4.使用SqlSession创建Dao接口的代理对象
12        UserDao dao = session.getMapper(UserDao.class);
13        //5.使用代理对象执行方法
14        List<User> users = dao.findAll();
15        for (User user : users) {
16            System.out.println(user);
17        }
18        //6.释放资源
19        session.close();
20        in.close();
21    }
22 }
```

注解方式

映射配置|操作信息(SQL、参数返回值类型)由接口上注解完成。

- 配置xml
 - mapper 标签 class属性 指定dao接口的全限定类名
- 接口dao
 - 方法上使用@注解，指定SQL语句
- 不需要映射xml了。

```
1 | <mapper class="com.learn.dao.UserDao"></mapper>
```

配置xml

mapper

- 指定接口

```
1 | <?xml version="1.0" encoding="UTF-8"?>
2 | <!DOCTYPE configuration
3 |     PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4 |     "http://mybatis.org/dtd/mybatis-3-config.dtd">
5 |
6 | <configuration>
7 |     <environments default="mysql">
8 |         <environment id="mysql">
9 |             <transactionManager type="JDBC"></transactionManager>
10 |             <dataSource type="POOLED">
11 |                 <property name="driver" value="com.mysql.cj.jdbc.Driver"/>
12 |                 <property name="url" value="jdbc:mysql://localhost:3306/mybatis"/>
13 |                 <property name="username" value="root"/>
14 |                 <property name="password" value="xxxx"/>
15 |             </dataSource>
16 |         </environment>
17 |     </environments>
18 |
19 |     <!-- 如果是用注解来配置的话，此处应该使用class属性指定被注解的dao全限定类名 -->
20 |     <mappers>
21 |         <mapper class="com.learn.dao.UserDao"></mapper>
22 |     </mappers>
23 | </configuration>
```

UserDao

- @注解，指定SQL语句

```
1 public interface UserDao {  
2  
3     /**  
4      * 查询所有  
5      * @return  
6      */  
7     @Select("select * from user")  
8     List<User> findAll();  
9  
10 }
```

测试类

同上

- 读取配置文件
- 创建 SqlSessionFactory 工厂
- 使用工厂生产 SqlSession 对象
- 使用SqlSession 创建Dao接口的 代理对象
- 使用代理对象执行方法
- 释放资源

```
1 public class MyTest {  
2  
3     public static void main(String[] args) throws Exception {  
4         //1.读取配置文件  
5         InputStream in = Resources.getResourceAsStream("SqlMapConfig.xml");  
6         //2.创建SqlSessionFactory工厂  
7         SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();  
8         SqlSessionFactory factory = builder.build(in);  
9         //3.使用工厂生产SqlSession对象  
10        SqlSession session = factory.openSession();  
11        //4.使用SqlSession创建Dao接口的代理对象  
12        UserDao dao = session.getMapper(UserDao.class);  
13        //5.使用代理对象执行方法  
14        List<User> users = dao.findAll();  
15        for (User user : users) {  
16            System.out.println(user);  
17        }  
18        //6.释放资源  
19        session.close();  
20        in.close();  
21    }  
22 }
```

实现类方式

MyBatis的核心类 SqlSession 提供了 两种方式：

- selectList等 直接调用方法（参数指定操作信息位置）
 - 操作还是 MyBatis 去实现的
- getMapper 提供代理对象，相当于dao 实现类

注解 和 XML都使用session的 getMapper 拿到 代理对象 , 这个代理对象就是MyBatis 为我们生成的 实现类 , daoImpl。

我们也可以自己 定义实现类 , 虽然一般不这么做。

提要 :

- 映射xml 同 xml方式
- 配置xml 同 xml方式
- 实现类 新增 , 实现功能

- ```
SqlSession session = factory.openSession();
```
- ```
List<User> users =
```
- ```
session.selectList("com.learn.dao.UserDao.findAll");
```

- 测试类 修改 ,

- 删除 session获取
- 删除 代理dao对象创建
- 改用 dao实现类

- ```
UserDao dao = new UserDaoImpl(factory);
```

配置xml

同xml方式

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE configuration
3     PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4     "http://mybatis.org/dtd/mybatis-3-config.dtd">
5
6 <!-- mybatis的主配置文件:和环境、连数据库相关的 -->
7 <configuration>
8     <!-- 配置mysql环境 -->
9     <environments default="mysql"><!-- 起个名 -->
10    <!-- 配置mysql的环境 -->
11    <environment id="mysql"><!-- 上面那名 -->
12        <!-- 配置事务的类型 -->
13        <transactionManager type="JDBC"></transactionManager>
14        <!-- 配置数据源(连接池) 连接数据库的信息 -->
15        <dataSource type="POOLED"><!-- 取值有3个 -->
16            <!-- 配置连接数据库的4个基本信息 -->
17            <property name="driver" value="com.mysql.cj.jdbc.Driver"/>
18            <property name="url"
19                value="jdbc:mysql://localhost:3306/mybatis"/>
20            <property name="username" value="root"/>
21            <property name="password" value="xxxx"/>
22        </dataSource>
23    </environment>
24 </environments>
25
26 <!-- 指定映射配置文件的位置 告知mybatis映射配置的位置 -->
27 <!-- 映射配置文件指的是每个dao独立的配置文件 -->
28 <mappers>
```

```
28     <mapper resource="com/learn/dao/UserDao.xml"></mapper>
29   </mappers>
30 </configuration>
```

映射xml

同xml方式

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE mapper
3     PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4     "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5
6 <mapper namespace="com.learn.dao.UserDao"><!-- UserDao接口的全限定类名 -->
7   <!-- 配置查询所有 id是方法名 -->
8   <select id="findAll" resultType="com.learn.domain.User">
9     <!-- SQL语句 -->
10    select * from user;
11  </select>
12 </mapper>
```

实现类

- factory——> session——>执行方法
- 参数：根据映射xml， UserDao.xml的namespace属性来的
 - namespace.id

```
1 public class UserDaoImpl implements UserDao {
2
3     private SqlSessionFactory factory;
4
5     public UserDaoImpl(SqlSessionFactory factory) {
6         this.factory = factory;
7     }
8
9     public List<User> findAll() {
10        //1.使用工厂创建SqlSession对象
11        SqlSession session = factory.openSession();
12        //2.使用session执行查询所有方法
13        //根据UserDao.xml的namespace属性来的 namespace.id
14        List<User> users =
15        session.selectList("com.learn.dao.UserDao.findAll");
16        session.close();
17        //3.返回查询结果
18        return users;
19    }
}
```

测试类

- 使用工厂生产SqlSession对象
 - `SqlSession session = factory.openSession();`
- 使用SqlSession创建Dao接口的代理对象
 - `UserDao dao = session.getMapper(UserDao.class)`

获取dao的方式改为：

- `UserDao dao = new UserDaoImpl(factory)`

```

1  public static void main(String[] args) throws Exception {
2      //1.读取配置文件
3      InputStream in = Resources.getResourceAsStream("SqlMapConfig.xml");
4      //2.创建SqlSessionFactory工厂
5      SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
6      SqlSessionFactory factory = builder.build(in);
7
8      //3.使用工厂生产SqlSession对象
9      //    SqlSession session = factory.openSession();
10     //4.使用SqlSession创建Dao接口的代理对象
11     //    UserDao dao = session.getMapper(UserDao.class);
12
13     //3.使用工厂创建dao对象
14     UserDao dao = new UserDaoImpl(factory);
15
16     //5.使用代理对象执行方法
17     List<User> users = dao.findAll();
18     for (User user : users) {
19         System.out.println(user);
20     }
21     //6.释放资源
22     //    session.close();
23     in.close();
24 }
```

案例分析

```

public static void main(String[] args) throws Exception {
    //1.读取配置文件
    InputStream in = Resources.getResourceAsStream("SqlMapConfig.xml");  
第一个：使用类加载器，它只能读取类路径的配置文件  
绝对路径：d:/xxx/xxx.xml X  
相对路径：src/java/main/xxx.xml X  
//2.创建SqlSessionFactory工厂
    SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();  
创建工厂 mybatis使用了构建者模式  
builder.build(in); builder就是构建者
    //3.使用工厂生产SqlSession对象
    SqlSession session = factory.openSession();  
生产SqlSession使用了工厂模式  
//4.使用SqlSession创建Dao接口的代理对象
    IUserDao userDao = session.getMapper(IUserDao.class);  
创建Dao接口实现类使用了代理
    //5.使用代理对象执行方法
    List<User> users = userDao.findAll();  
模式  
for (User user : users) {  
    System.out.println(user);
}
//6.释放资源
session.close();
in.close();
}
```

- 读取文件方式
 - 相对路径：`src/java/main/xxx.xml`
 - web工程一部署，src目录就没了
 - 绝对路径：`D:/xxx/xxx.xml`

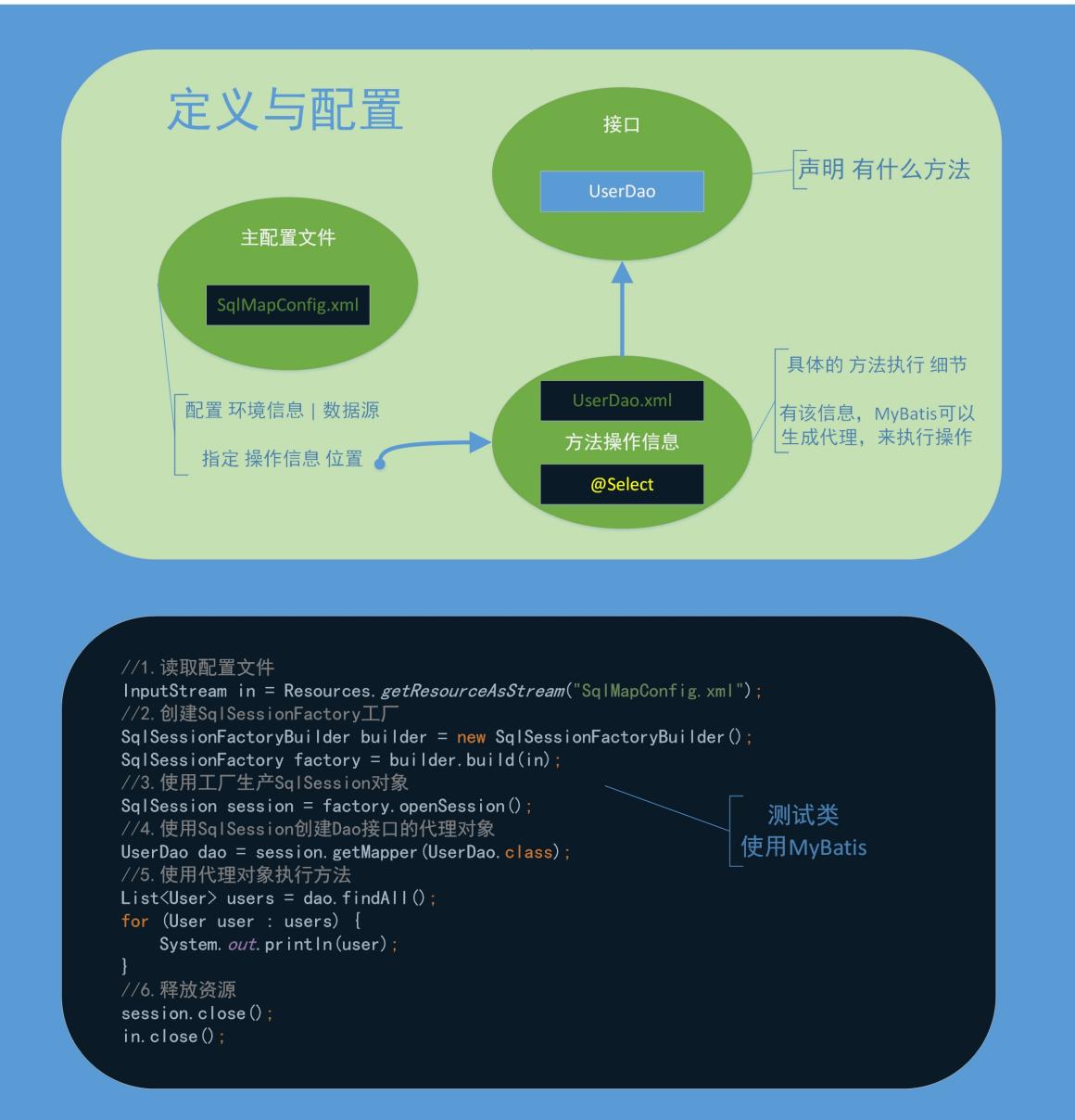
- 系统未必有盘符
- 类加载器
 - 只能读取类路径的配置文件。
 - `InputStream in = Resources.getResourceAsStream("SqlMapConfig.xml")`
 - `ServletContext.getRealPath()`
 - 得到当前应用部署的 绝对路径
- 构建者模式：把对象创建的细节隐藏，直接调用方法即可拿到对象
 - 创建工厂：`sqlSessionFactory factory = builder.build(in);`
- 工厂模式
 - 生成SqlSession对象(工厂模式，可以生产一类对象)
- 代理模式
 - 不用实现类UserDaoImpl实现查询
 - `UserDao dao = session.getMapper(UserDao.class);`
 - 生成了代理对象

环境搭建

- 建项目|工程
 - 单独使用，maven下简单工程
- 主配置文件（必需的）
- 操作信息配置
 - xml文件
 - 注解

主要4文件

- 主配置文件
- 接口：声明方法
- xml|注解：方法操作信息
- 测试类使用



新建maven工程

单纯学习MyBatis的项目，不需要什么其他的。

- 什么都不要选
 - 最简单的maven工程即可
- archetype
 - 选择普通的java工程

pom文件依赖

- mybatis
- mysql-connection-java
- (log4j) 日志
- (junit)

可以去官网去找。<https://mybatis.org/mybatis-3/zh/getting-started.html>

- 入门中有pom坐标

```

1 <dependency>
2   <groupId>org.mybatis</groupId>
3   <artifactId>mybatis</artifactId>
4   <version>3.5.1</version>
5 </dependency>
6
7 <dependency>
8   <groupId>mysql</groupId>
9   <artifactId>mysql-connector-java</artifactId>
10  <version>8.0.15</version>
11 </dependency>
12
13 <dependency>
14   <groupId>log4j</groupId>
15   <artifactId>log4j</artifactId>
16   <version>1.2.12</version>
17 </dependency>

```

主配置文件

应该放在这里。

位置

- resources 新建 file
- resources / SqlMapConfig.xml

命名

- 叫什么都行
 - 习惯上 SqlMapConfig.xml

标签

- `<configuration>`
 - `<properties>` : 属性。也可以通过属性引用外部配置文件信息
 - resource :
 - 用于指定 配置文件 的位置
 - 按照 类路径的写法来写
 - 必须存在于类路径下
 - url :
 - 要求按照Url的写法来写地址
 - file:///E:/code/MyBatis202Dao/src/main/resources/jdbcConfig.properties
 - URL : Uniform Resource Locator。统一资源定位符，它是可以唯一标识一个资源的位置。
 - 写法： 协议 主机 端口 URI
 - <http://localhost:8080/mybatisserver/demoServlet>

- URI : Uniform Resource Identifier 统一资源标识符。它是在应用中可以唯一定位一个资源的。
- <property> : 一般不在这配置信息，都写properties文件里
- <settings> : 全局参数配置
 - <setting>
- <typeAliases> : 配置domain中的类|实体类的 别名，再用就不再需要写类的 全限定类名了
 - type : 指定实体类 全限定类名
 - alias : 指定别名；指定了别名就不再区分大小写
 - <typeAlias>
 - <package> 标签 用于指定 要配置别名的包
 - 指定后，该包下的 实体类都会注册别名，并且类名就是别名，不再区分大小写
- <typeHandlers> : 类型处理器
- <objectFactory> : 对象工厂
- <plugins> : 插件
- <environments> default="mysql" 环境集合属性对象
 - <environment> id="mysql" 这里应该不固定 环境子属性对象
 - <transactionManager> : 事务管理
 - type :
 - JDBC
 - <dataSource> : 配置数据源(连接池) 连接数据库的信息
 - type :
 - UNPOOLED
 - POOLED
 - JNDI
 - <property>
 - dirver、url、username、password
- <mappers> : 映射器
 - <mapper>
 - "resource" : 映射xml文件 位置；xml方式
 - "class" : 接口 的文件位置；注解方式
 - <package> :
 - 指定 映射配置 的包
 - xml 和 class都支持

properties

<properties> : 配置连接数据库的信息。也可以通过属性引用外部配置文件信息

- resource :
- 用于指定 配置文件 的位置
- 按照 类路径的写法来写
- 必须存在于类路径下

- url :
 - 要求按照Url的写法来写地址
 - file:///E:/code/MyBatis202Dao/src/main/resources/jdbcConfig.properties
 - URL : Uniform Resource Locator。统一资源定位符，它是可以唯一标识一个资源的位置。
 - 写法： 协议 主机 端口 URI
 - <http://localhost:8080/mybatisserver/demoServlet>
 - URI : Uniform Resource Identifier 统一资源标识符。它是在应用中可以唯一定位一个资源的。
- <property> : 一般不在这配置信息，都写properties文件里

```

1 <properties resource="jdbcConfig.properties">
2   <!--
3     url="file:///E:/code/MyBatis202Dao/src/main/resources/jdbcConfig.properties"
4   -->
5 </properties>
6
7 <properties resource="jdbcConfig.properties">
8   <!--
9     url="file:///E:/code/MyBatis202Dao/src/main/resources/jdbcConfig.properties"
10    -->
11   <property name="driver" value="com.mysql.cj.jdbc.Driver"/>
12   <property name="url" value="jdbc:mysql://localhost:3306/mybatis"/>
13   <property name="username" value="root"/>
14   <property name="password" value="xxxx"/>
15 </properties>

```

typeAliases

- <typeAliases> : 配置domain中的类|实体类的 别名，再用就不需要写类的 全限定类名了
- type : 指定实体类 全限定类名
 - alias : 指定别名；指定了别名就不再区分大小写
 - <package> 标签 用于指定 要配置别名的包
 - 指定后，该包下的 实体类都会注册别名，并且类名就是别名，不再区分大小写

```

1 <!-- 使用typeAliases 配置别名。它只能配置domain中类的别名 -->
2 <typeAliases>
3   <!-- type属性指定的是实体类全限定类名。alias属性指定别名 指定了别名就不再区分大小
4   写 -->
5     <typeAlias type="com.learn.domain.User" alias="user"></typeAlias>
6
7   <!-- 用于指定要配置别名的包，当指定之后，该包下的实体类都会注册别名，并且类名就是别
8   名，不再区分大小写 -->
9     <package name="com.learn.domain"></package>
10 </typeAliases>
11
12 <typeAliases>
13   <!-- 单个别名定义 -->
14   <typeAlias alias="user" type="com.itheima.domain.User"/>

```

```
13 |     <!-- 批量别名定义，扫描整个包下的类，别名为类名（首字母大写或小写都可以） -->
14 |     <package name="com.itheima.domain"/>
15 |     <package name=" 其它包 "/>
16 | </typeAliases>
```

environments

environment

transactionManage

dataSource

mappers

```
1 <mappers>
2     <mapper resource="com/learn/dao/UserDao.xml">使用相对于类路径的资源</mapper>
3
4     <!-- 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 -->
5     <mapper class="com.itheima.dao.UserDao">使用 mapper 接口类路径</mapper>
6
7     <!-- 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 -->
8     <package name="cn.itcast.mybatis.mapper">注册指定包下的所有 mapper 接口
9 </package>
</mappers>
```

mapper

package

指定包

- name

```
1 | <package name="com.learn.dao"></package>
```

property

文件约束

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE configuration
3   PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4   "http://mybatis.org/dtd/mybatis-3-config.dtd">
```

完整xml

常规示例

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE configuration
3   PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4   "http://mybatis.org/dtd/mybatis-3-config.dtd">
5
6 <!-- mybatis的主配置文件:和环境、连数据库相关的 -->
7 <configuration>
8   <!-- 配置mysql环境 -->
9   <environments default="mysql"><!-- 起个名 -->
10    <!-- 配置mysql的环境 -->
11    <environment id="mysql"><!-- 上面那名 -->
12      <!-- 配置事务的类型 -->
13      <transactionManager type="JDBC"></transactionManager>
14      <!-- 配置数据源(连接池) 连接数据库的信息 -->
15      <dataSource type="POOLED"><!-- 取值有3个 -->
16        <!-- 配置连接数据库的4个基本信息 -->
17        <property name="driver" value="com.mysql.cj.jdbc.Driver"/>
18        <property name="url"
19          value="jdbc:mysql://localhost:3306/mybatis"/>
20        <property name="username" value="root"/>
21        <property name="password" value="xxxx"/>
22      </dataSource>
23    </environment>
24  </environments>
25
26  <!-- 指定映射配置文件的位置 告知mybatis映射配置的位置 -->
27  <!-- 映射配置文件指的是每个dao独立的配置文件 -->
28  <mappers>
29    <mapper resource="com/learn/dao/UserDao.xml"></mapper>
30  </mappers>
31 </configuration>
```

properties示例

引入外部

```
1 <properties resource="jdbcConfig.properties">
2   <!--
3     url="file:///E:/code/MyBatis202Dao/src/main/resources/jdbcConfig.properties"
4   -->
5 </properties>
```

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE configuration
3      PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4      "http://mybatis.org/dtd/mybatis-3-config.dtd">
5
6  <configuration>
7      <!-- 配置properties 一般不写这 放外面的配置文件里 -->
8      <!-- 可以在标签内部配置连接数据库的信息。也可以通过属性引用外部配置文件信息
9          resource属性：
10         用于指定配置文件的位置，是按照类路径的写法来写，并且必须存在于类路径下
11         url属性： 通常不用
12         是要求按照URL的写法来写地址
13         URL: Uniform Resource Locator 统一资源定位符。它可以唯一标识一个
14         资源的位置。
15         写法：
16             协议   主机   端口   URI
17             http://localhost:8080/mybatisserver/demoServlet
18
19         URI: Uniform Resource Identifier 统一资源标识符。它是在应用中可
20         以唯一定位一个资源的。
21
22         -->
23     <properties resource="jdbcConfig.properties">
24         <!--
25             url="file:///E:/code\MyBatis202Dao\src\main\resources\jdbcConfig.properties
26         " -->
27     </properties>
28
29
30         <!-- 使用typeAliases 配置别名。它只能配置domain中类的别名 -->
31     <typeAliases>
32         <!-- type属性指定的是实体类全限定类名。alias属性指定别名 指定了别名就不再区
33         分大小写 -->
34         <typeAlias type="com.learn.domain.User" alias="user"></typeAlias>
35
36         <!-- 用于指定要配置别名的包，当指定之后，该包下的实体类都会注册别名，并且类名
37         就是别名，不再区分大小写 -->
38         <package name="com.learn.domain"></package>
39     </typeAliases>
40
41
42         <!-- 配置环境 -->
43     <environments default="mysql">
44         <environment id="mysql">
45             <!-- 配置事务 -->
46             <transactionManager type="JDBC"></transactionManager>
47             <!-- 配置连接池 -->
48             <dataSource type="POOLED">
49                 <!-- 配置连接数据库的4个基本信息 -->
50                 <property name="driver" value="${jdbc.driver}"/>
51                 <property name="url" value="${jdbc.url}"/>
52                 <property name="username" value="${jdbc.username}"/>
53                 <property name="password" value="${jdbc.password}"/>
54             </dataSource>
55         </environment>
56     </environments>
57         <!-- 指定映射配置文件的位置 -->
58     <mappers>
59         <!--     <mapper resource="com/learn/dao/UserDao.xml"></mapper>-->
```

```

52
53      <!-- 用于指定dao接口所在的包,当指定了之后就不需要再写mapper以及resource或者
54      class了 -->
55      <package name="com.learn.dao"></package>
56
57  </configuration>

```

内部配置

```

1  <properties resource="jdbcConfig.properties">
2      <!--
3      url="file:///E:/code\MyBatis202Dao\src\main\resources\jdbcConfig.properties"
-->
4      <property name="driver" value="com.mysql.cj.jdbc.Driver"/>
5      <property name="url" value="jdbc:mysql://localhost:3306/mybatis"/>
6      <property name="username" value="root"/>
7      <property name="password" value="xxxx"/>
8  </properties>

```

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE configuration
3      PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4      "http://mybatis.org/dtd/mybatis-3-config.dtd">
5
6  <configuration>
7      <!-- 配置properties 一般不写这 放外面的配置文件里 -->
8      <!-- 可以在标签内部配置连接数据库的信息。也可以通过属性引用外部配置文件信息
9          resource属性:
10         用于指定配置文件的位置，是按照类路径的写法来写，并且必须存在于类路径下
11         url属性：通常不用
12         是要求按照URL的写法来写地址
13         URL: Uniform Resource Locator 统一资源定位符。它可以唯一标识一个
14         资源的位置。
15         写法:
16             协议   主机   端口   URI
17             http://localhost:8080/mybatisserver/demoServlet
18
19             URI: Uniform Resource Identifier 统一资源标识符。它是在应用中可
20             以唯一定位一个资源的。
21
22     <!--
23     <properties resource="jdbcConfig.properties">
24         <!--
25         url="file:///E:/code\MyBatis202Dao\src\main\resources\jdbcConfig.properties"
26         -->
27         <property name="driver" value="com.mysql.cj.jdbc.Driver"/>
28         <property name="url" value="jdbc:mysql://localhost:3306/mybatis"/>
29         <property name="username" value="root"/>
30         <property name="password" value="xxxx"/>
31     </properties>
32
33     <!-- 使用tyoeAliases 配置别名。它只能配置domain中类的别名 -->

```

```

30     <typeAliases>
31         <!-- type属性指定的是实体类全限定类名。alias属性指定别名 指定了别名就不再区分大小写 -->
32         <typeAlias type="com.learn.domain.User" alias="user"></typeAlias>
33
34         <!-- 用于指定要配置别名的包，当指定之后，该包下的实体类都会注册别名，并且类名就是别名，不再区分大小写 -->
35         <package name="com.learn.domain"></package>
36     </typeAliases>
37
38     <!-- 配置环境 -->
39     <environments default="mysql">
40         <environment id="mysql">
41             <!-- 配置事务 -->
42             <transactionManager type="JDBC"></transactionManager>
43             <!-- 配置连接池 -->
44             <dataSource type="POOLED">
45                 <!-- 配置连接数据库的4个基本信息 -->
46                 <property name="driver" value="${jdbc.driver}"/>
47                 <property name="url" value="${jdbc.url}"/>
48                 <property name="username" value="${jdbc.username}"/>
49                 <property name="password" value="${jdbc.password}"/>
50             </dataSource>
51         </environment>
52     </environments>
53     <!-- 指定映射配置文件的位置 -->
54     <mappers>
55         <!-- <mapper resource="com/learn/dao/UserDao.xml"></mapper>-->
56             <!-- 用于指定dao接口所在的包，当指定了之后就不再需要再写mapper以及resource或者class了 -->
57             <package name="com.learn.dao"></package>
58         </mappers>
59
60     </configuration>

```

jdbcConfig.properties

位置：

- resources / jdbcConfig.properties

```

1  jdbc.driver=com.mysql.cj.jdbc.Driver
2  jdbc.url=jdbc:mysql://localhost:3306/mybatis
3  jdbc.username=root
4  jdbc.password=xxxx

```

typeAliases示例

```
1 <!-- 使用typeAliases 配置别名。它只能配置domain中类的别名 -->
2 <typeAliases>
3     <!-- type属性指定的是实体类全限定类名。alias属性指定别名 指定了别名就不再区分大小写
-->
4     <typeAlias type="com.learn.domain.User" alias="user"></typeAlias>
5
6     <!-- 用于指定要配置别名的包，当指定之后，该包下的实体类都会注册别名，并且类名就是别名，不再区分大小写 -->
7     <package name="com.learn.domain"></package>
8 </typeAliases>
```

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE configuration
3     PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4     "http://mybatis.org/dtd/mybatis-3-config.dtd">
5
6 <configuration>
7     <!-- 配置properties 一般不写这 放外面的配置文件里 -->
8     <properties resource="jdbcConfig.properties">
9         <!--
10        url="file:///E:/code\MyBatis202Dao\src\main\resources\jdbcConfig.properties
11        " -->
12         <property name="driver" value="com.mysql.cj.jdbc.Driver"/>
13         <property name="url" value="jdbc:mysql://localhost:3306/mybatis"/>
14         <property name="username" value="root"/>
15         <property name="password" value="xxxx"/>
16     </properties>
17
18     <!-- 使用typeAliases 配置别名。它只能配置domain中类的别名 -->
19     <typeAliases>
20         <!-- type属性指定的是实体类全限定类名。alias属性指定别名 指定了别名就不再区分大小写 -->
21         <typeAlias type="com.learn.domain.User" alias="user"></typeAlias>
22
23         <!-- 用于指定要配置别名的包，当指定之后，该包下的实体类都会注册别名，并且类名就是别名，不再区分大小写 -->
24         <package name="com.learn.domain"></package>
25     </typeAliases>
26
27     <!-- 配置环境 -->
28     <environments default="mysql">
29         <environment id="mysql">
30             <!-- 配置事务 -->
31             <transactionManager type="JDBC"></transactionManager>
32             <!-- 配置连接池 -->
33             <dataSource type="POOLED">
34                 <!-- 配置连接数据库的4个基本信息 -->
35                 <property name="driver" value="${jdbc.driver}"/>
36                 <property name="url" value="${jdbc.url}"/>
37                 <property name="username" value="${jdbc.username}"/>
38                 <property name="password" value="${jdbc.password}"/>
39             </dataSource>
40         </environment>
41     </environments>
```

```
40     <!-- 指定映射配置文件的位置 -->
41     <mappers>
42         <package name="com.learn.dao"></package>
43     </mappers>
44
45 </configuration>
```

映射配置操作信息

同一接口的 映射配置(我喜欢叫执行信息) , 不能同时存在xml 和 注解方式。会报错

xml标签 和 注解 放这 其实不错

xml方式配置

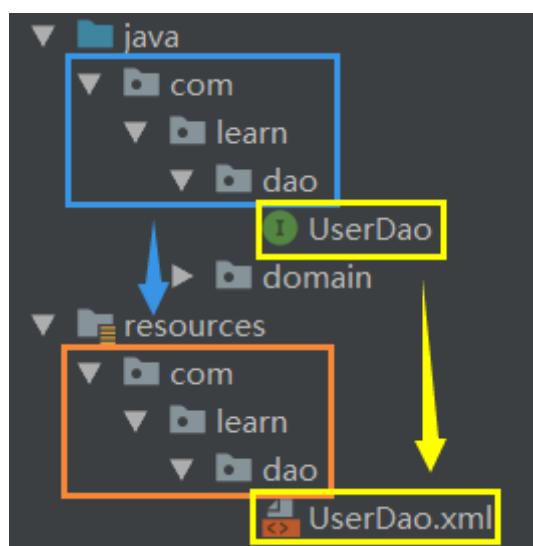
- 使用注解时 不能有此文件
- resources 新建 file

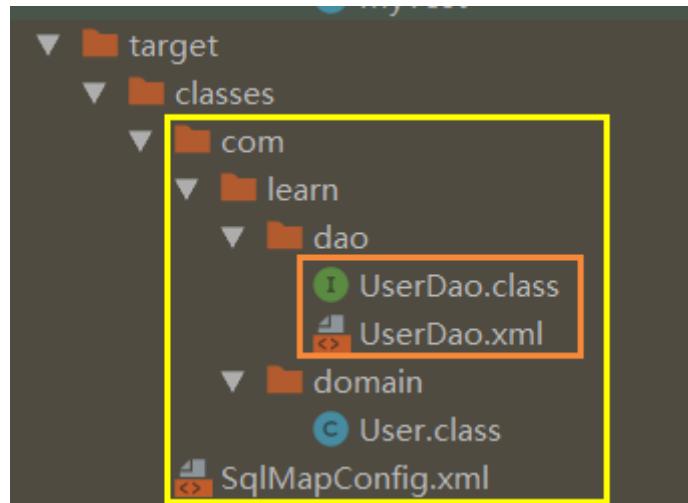
位置 :

- resources / com/learn/dao/ UserDao.xml

要求

- 创建位置 : 必须和持久层接口在相同的包中。
 - dao接口 : java/com.learn.dao.UserDao
 - 映射文件 : resources com/learn/dao/UserDao.xml
- 名称 : 必须以持久层接口名称命名文件名 , 扩展名是.xml
 - dao接口 : java/com.learn.dao.UserDao
 - 映射文件 : resources com/learn/dao/UserDao.xml





- 映射配置文件的 mapper标签 namespace属性 的取值 必须是
 - 接口的全限定类名
- 映射配置文件的操作配置 (select标签) , id属性必须是dao接口的方法名

标签

- <mapper>
 - **namespace** : 对应的是哪个 接口 ; 填写接口的全限定类名即可
 - <select> ...
 - id : 对应哪个 方法
 - resultType : 封装到哪里去 ; 填写全限定类名即可

mapper

- **namespace** : 对应的是哪个 接口 ; 填写接口的全限定类名即可

select

if

where

for-each

sql

resultMap

collection

assocation

cache

文件约束

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

完整xml

常规示例

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE mapper
3   PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4   "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5
6 <mapper namespace="com.learn.dao.UserDao"><!-- UserDao接口的全限定类名 -->
7   <!-- 配置查询所有 id是方法名 -->
8   <select id="findAll" resultType="com.learn.domain.User">
9     <!-- SQL语句 -->
10    select * from user;
11  </select>
12 </mapper>
```

注解方式配置

- 主配置文件 `<mappers> / <mapper>` 中指定 配置 位置

```
1 | <mapper class="com.learn.dao.UserDao"></mapper>
```

- 指定接口 方法上 写注解即可

```
1 | @Select("select * from user")
2 | List<User> findAll();
```

注解

@Select

实现查询

@Insert

实现新增

@Update

实现更新

@Delete

实现删除

@Result

实现结果集封装

@Results

可以与@Result一起使用，封装多个结果集

@One

实现一对一结果集封装

@Many

实现一对多结果集封装

@ResultMap

实现引用@Results 定义的封装

@SelectProvider

实现动态 SQL 映射

@CacheNamespace

实现注解二级缓存的使用

(测试类)

使用配置好的MyBatis

简单使用

```
1 public static void main(String[] args) throws Exception {
2     //1.读取配置文件
3     InputStream in = Resources.getResourceAsStream("SqlMapConfig.xml");
4     //2.创建SqlSessionFactory工厂
5     SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
6     SqlSessionFactory factory = builder.build(in);
7     //3.使用工厂生产SqlSession对象
8     SqlSession session = factory.openSession();
9     //4.使用SqlSession创建Dao接口的代理对象
10    UserDao dao = session.getMapper(UserDao.class);
11    //5.使用代理对象执行方法
12    List<User> users = dao.findAll();
13    for (User user : users) {
14        System.out.println(user);
15    }
16    //6.释放资源
17    session.close();
18    in.close();
19 }
```

比较成熟的测试类

```

1  public class MybatisTest {
2
3      private InputStream in;
4      private SqlSessionFactory factory;
5      private SqlSession session;
6      private UserDao dao;
7
8      @Before //用于在测试方法执行之前执行
9      public void init() throws Exception {
10          //1.读取配置文件，生成字节输出流
11          in = Resources.getResourceAsStream("SqlMapConfig.xml");
12          //2.获取SqlSessionFactory
13          factory = new SqlSessionFactoryBuilder().build(in);
14          //3.获取SqlSession对象
15          session = factory.openSession();
16          //4.获取dao的代理对象
17          dao = session.getMapper(UserDao.class);
18      }
19
20      @After //用于在测试方法之后执行
21      public void destroy() throws Exception {
22          //事务提交
23          session.commit();
24          //6.释放资源
25          session.close();
26          in.close();
27      }
28
29      @Test
30      public void testFindOne() {
31          User user = dao.findById(48);
32          System.out.println(user);
33      }
34
35  }

```

使用 | 原理

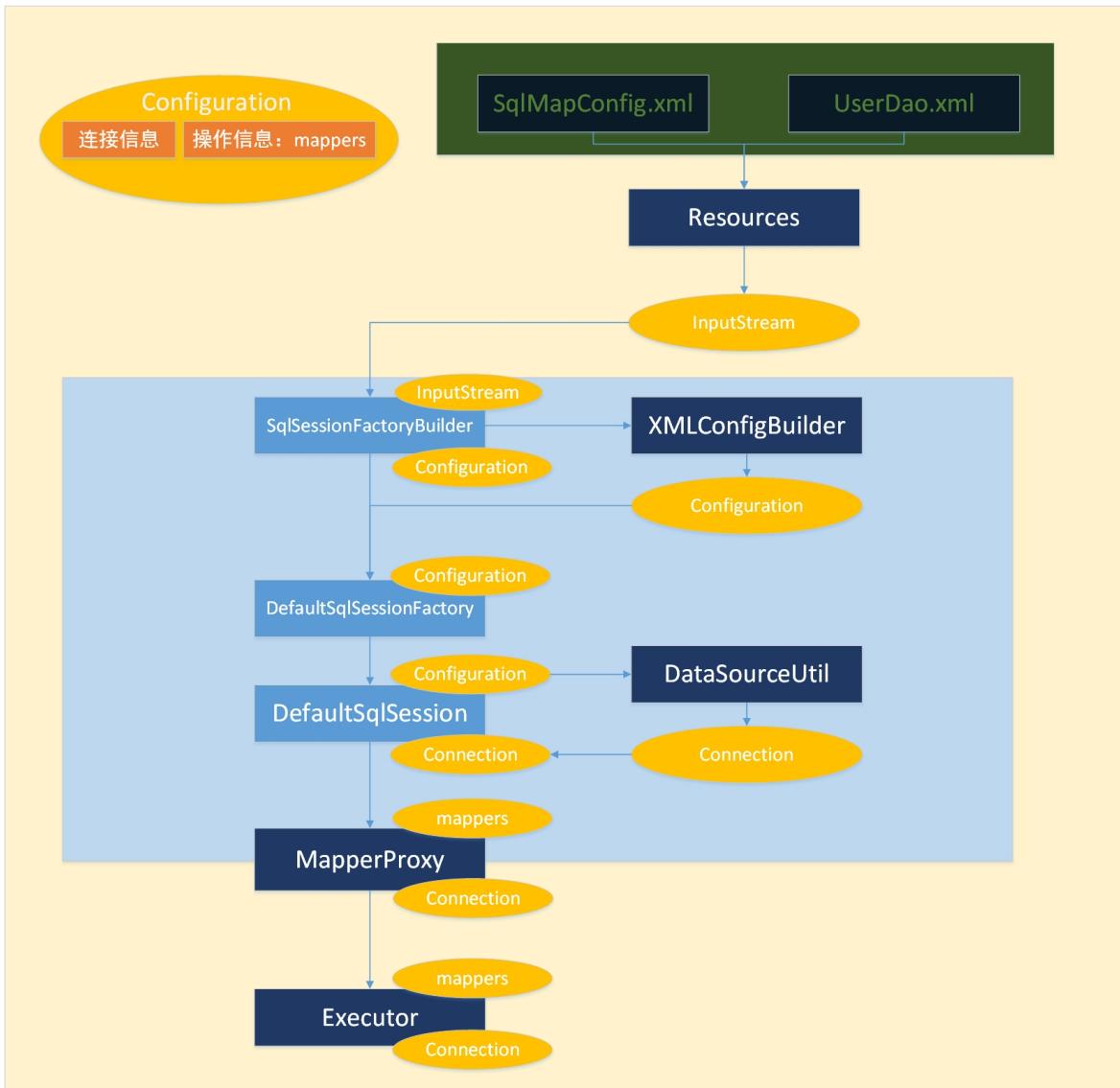
MyBatis 操作流程

- `InputStream in = Resources.getResourceAsStream("SqlMapConfig.xml")`
 - 读取 配置文件
- `SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder()`
 - 创建 工厂Builder
- `SqlSessionFactory factory = builder.build(in)`
 - 创建 工厂

下面两操作 可提取到实现类里：

- new UserDaoImpl(factory)
- 实现类 完成具体操作

- `session.selectList("com.learn.dao.UserDao.findAll")` 等方法
- `SqlSession session = factory.openSession()`
 - 创建 SqlSession
 - session 可创建 指定接口的 代理类，相当于实现类
 - session 可直接 操作方法 进行数据操作
- `UserDao dao = session.getMapper(UserDao.class)`
 - 创建 Dao 接口的 代理对象，功能相当于实现类



```
Resources.getResourceAsStream("SqlMapConfig.xml")
```

根据主配置文件信息 创建工厂

XMLConfigBuilder
连接信息 操作信息 mapper cfg

SqlSessionFactoryBuilder

```
SqlSessionFactory build(InputStream config)
```

```
new DefaultSqlSessionFactory(cfg)
```

连接信息 操作信息 mapper cfg

<<接口>>
SqlSessionFactory

```
SqlSession openSession()
```

创建SqlSession对象

DefaultSqlSessionFactory

Configuration cfg
DefaultSqlSessionFactory(Configuration cfg)
SqlSession openSession()

```
new DefaultSqlSession(cfg)
```

连接信息 操作信息 mapper cfg

<<接口>>
SqlSession

<T> T getMapper(Class<T> daoInterfaceClass)
void close()

DataSourceUtil

生成 接口 代理对象
定义通用CRUD方法

DefaultSqlSession

Configuration cfg
Connection conn
DefaultSqlSession(Configuration cfg)
<T> T getMapper(Class<T> daoInterfaceClass)
void close()

```
(T) Proxy.newProxyInstance(daoInterfaceClass.getClassLoader(),  
    new Class[]{daoInterfaceClass},  
    new MapperProxy(cfg.getMappers(), conn))  
)
```

conn

操作信息 mapper

代理对象

MapperProxy

Map<String, Mapper> mappers
Connection conn
MapperProxy(mappers, conn)
Object invoke(Object proxy, Method method, Object[] args)

```
new Executor().selectList(mapper, conn)
```

conn

操作信息 mapper

执行SQL语句

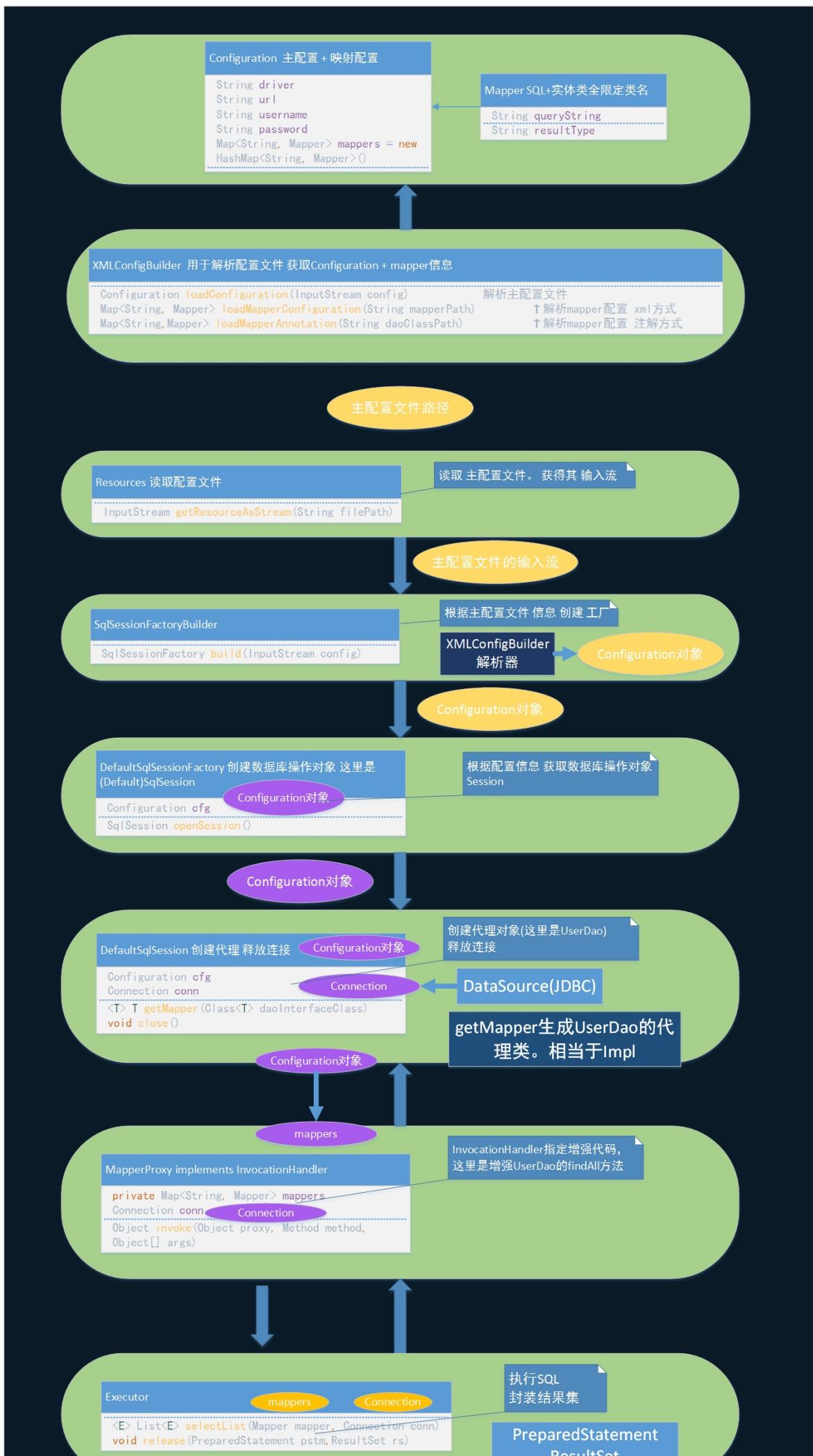
封装结果集

Executor

<E> List<E> selectList(Mapper mapper, Connection conn)
void release(PreparedStatement pstm, ResultSet rs)

```
pstm = conn.prepareStatement(queryString)
```

```
rs = pstm.executeQuery()
```



相关类

Resources

SqlSessionFactoryBuilder

XMLConfigBuilder

SqlSessionFactory

SqlSession

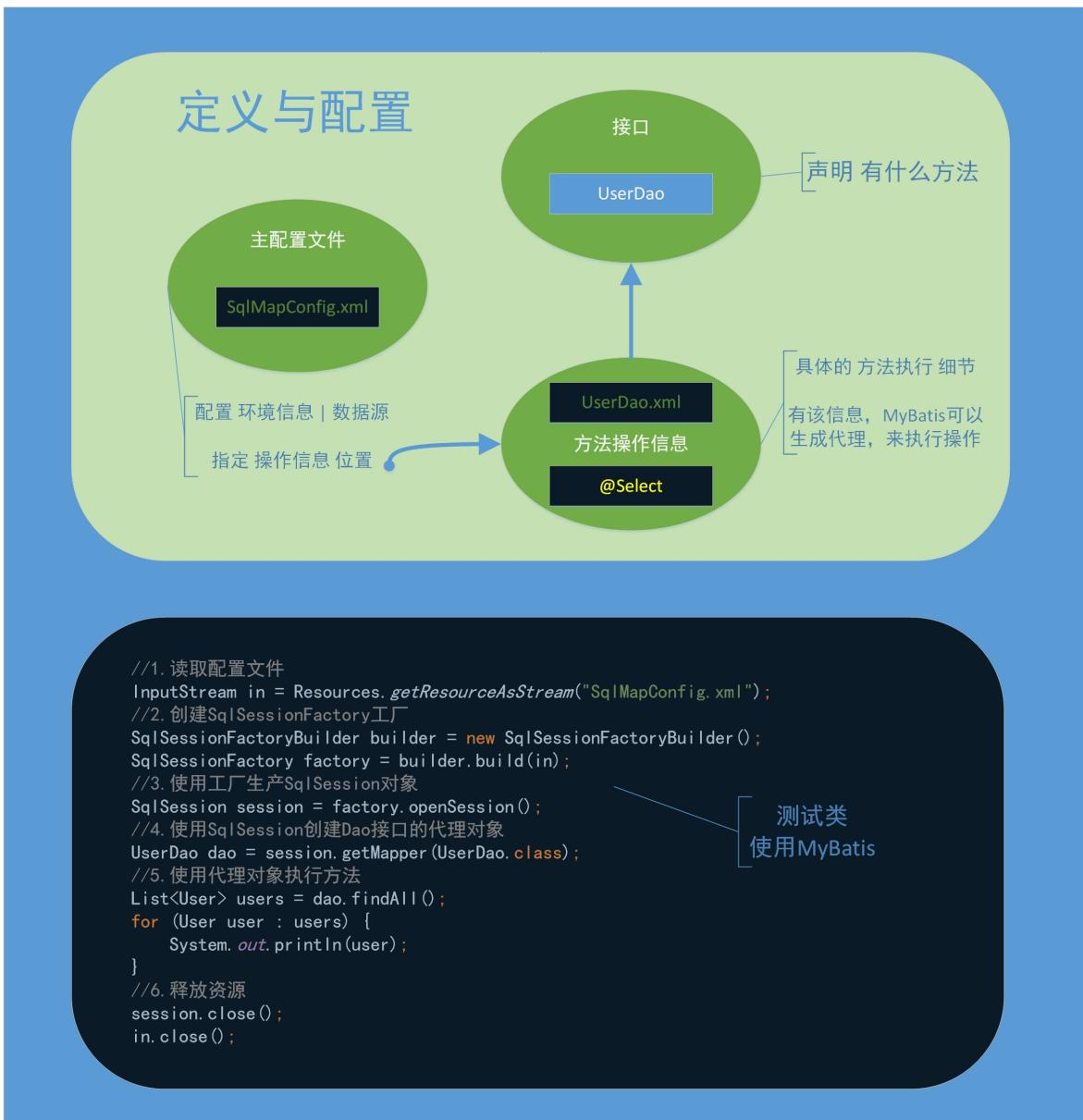
DataSourceUtil

MapperProxy

Executor

主要 4文件 关系

- 主配置文件
 - 接口：声明方法
 - xml|注解：方法操作信息
-
- 测试类 使用



默认配置 | 注意事项

MyBatis默认 `autocommit` 为 `false`；表示开启事务

- 手动提交
- 设置为`true`，自动提交

注释只支持 `<!-- -->`

单表CRUD

xml方式

- `#`{实体类中属性名} 其实是`getXxx`

查询:基础查询

映射文件

```
1 <!-- 配置查询所有 id是方法名 -->
2 <select id="findAll" resultType="com.learn.domain.User"><!-- 实体类 全限定类名
  封装成啥 -->
3   <!-- SQL语句 -->
4     select * from user;
5 </select>
```

接口

```
1 List<User> findAll();
```

测试类

```
1 @Test
2 public void testfindAll() {
3   //5.执行查询所有方法
4   List<User> users = dao.findAll();
5   for (User user : users) {
6     System.out.println(user);
7   }
8 }
```

查询:条件查询

映射文件

```
1 <!-- 根据id查询用户 -->
2 <select id="findById" parameterType="int" resultType="com.learn.domain.User">
3   select * from user
4   where id = #{suibianxie}
5 </select>
```

接口

```
1 User findById(Integer userId);
```

测试类

```
1 @Test
2 public void testFindone() {
3   User user = dao.findById(48);
4   System.out.println(user);
5 }
```

查询:复杂条件查询

映射文件

```
1 <!-- 根据QueryVo的条件查询用户 -->
2 <select id="findUserByVo" parameterType="com.learn.domain.QueryVo"
resultType="com.learn.domain.User">
3   select * from user where username like #{user.username}
4 </select>
```

接口

```
1 List<User> findUserByVo(QueryVo vo);
```

封装类

```
1 public class QueryVo {
2
3   private User user;
4
5   public User getUser() {
6     return user;
7   }
8
9   public void setUser(User user) {
10    this.user = user;
11  }
12 }
```

测试类

```
1 @Test
2 public void testFindByVo() {
3   QueryVo vo = new QueryVo();
4   User user = new User();
5   user.setUsername("%王%");
6   vo.setUser(user);
7
8   List<User> users = dao.findUserByVo(vo);
9
10  for (User u : users) {
11    System.out.println(u);
12  }
13 }
```

查询:模糊查询

- MyBatis有两种写法
 - 字符串拼接 value是源码中写死的，一般不这么写
 - 预处理方式，我们使用需要提供%

映射文件

```
1 <!-- 根据名称模糊查询 -->
2 <select id="findByName" parameterType="string"
resultType="com.learn.domain.User">
3   <!-- 字符串拼接 这里接收的值 必须写value 源码写死了 '${}' '#' -->
```

```
4     select * from user
5         where username like '%${value}%''
6 </select>
7
8 <!-- 根据名称模糊查询 -->
9 <select id="findByName" parameterType="string"
resultType="com.learn.domain.User">
10    <!-- 只支持这种注释 -->
11    <!-- 预处理方式 %调用时传 这里 这好像不能加 -->
12    select * from user
13        where username like #{username}%
14 </select>
```

接口

```
1 | List<User> findByName(String username);
```

测试类

```
1 | @Test
2 | public void testFindByName() {
3 |     //List<User> users = dao.findByName("%王%");
4 |     List<User> users = dao.findByName("王");//'${value}'%
5 |     for (User user : users) {
6 |         System.out.println(user);
7 |     }
8 |
9 |
10 | @Test
11 | public void testFindByName() {
12 |     List<User> users = dao.findByName("%王%");//#{唯一参数这里叫什么都可以}
13 |     for (User user : users) {
14 |         System.out.println(user);
15 |     }
16 | }
```

查询:聚合查询

映射文件

```
1 | <!-- 获取用户总记录条数 -->
2 | <select id="findTotal" resultType="int">
3 |     select count(id) from user;
4 | </select>
```

接口

```
1 | int findTotal();
```

测试类

```
1  @Test
2  public void testFindTotal() {
3      int total = dao.findTotal();
4      System.out.println(total);
5  }
```

保存

- `#{}{实体类中属性名 其实是getXXX}`
- 如何获取 保存 操作的 id , 其实是sql的一条语句 , 这里讲这个可能是比较常用吧
 - `order="AFTER"` : 什么时候执行该操作 , 查询之后

映射文件

```
1  <!-- 保存用户 -->
2  <insert id="saveUser" parameterType="com.learn.domain.User"> <!-- 参数类型 --
->
3      <!-- 配置插入操作后, 获取插入数据的id -->
4      <!--          数据库          -->
5      <selectKey keyProperty="id" keyColumn="id" resultType="int"
order="AFTER">
6          select last_insert_id();
7      </selectKey>
8
9      <!-- #{getXXX 的xxx} -->
10     insert into user(username, address, sex, birthday)
11         values(#{username},#{address},#{sex},#{birthday});
12 </insert>
```

接口

```
1  void saveUser(User user);
```

测试类

```
1  @Test
2  public void testSave() {
3      User user = new User();
4      user.setUsername("save1 username");
5      user.setAddress("save1 address");
6      user.setSex("男");
7      user.setBirthday(new Date());
8
9      System.out.println(user); //id=null
10
11     //5.执行查询所有方法
12     dao.saveUser(user);
13
14     System.out.println(user); //id有值
15
16     // log4j 事务没提交
17     //测试类@After 中 session.commit()可以不写这么多次
18     //session.commit();
```

19 | }

更新

映射文件

```
1 <!-- 更新用户 -->
2 <update id="updateUser" parameterType="com.learn.domain.User">
3   update user set username=#{username}, address=#{address}, sex=#{sex},
4   birthday=#{birthday}
5   where id=#{id}
6 </update>
```

接口

```
1 void updateUser(User user);
```

测试类

```
1 @Test
2 public void testUpdate() {
3   User user = new User();
4   user.setId(51);
5   user.setUsername("update1 username");
6   user.setAddress("update1 address");
7   user.setSex("女");
8   user.setBirthday(new Date());
9
10  //5.执行查询所有方法
11  dao.updateUser(user);
12 }
```

删除

映射文件

```
1 <!-- 删除用户 -->
2 <delete id="deleteUser" parameterType="Integer"><!-- int INT Integer
3   java.lang.Integer -->
4   <!-- 只有一个参数 写什么都可以 -->
5   delete from user
6   where id = #{userId}doukeyi
7 </delete>
```

接口

```
1 void deleteUser(Integer userId);
```

测试类

```
1  @Test
2  public void testDelete() {
3      dao.deleteUser(51);
4  }
```

注解方式

查询@Select

基础查询

```
1 /**
2  * 查询所有用户
3  * @return
4  */
5 @Select(value = "select * from user")
6 List<User> findAll();
```

条件查询

```
1 /**
2  * 根据id查询用户
3  * @param userId
4  * @return
5  */
6 @Select(value = "select * from user where id=#{id}")
7 User findById(Integer userId);
```

模糊查询

```
1 /**
2  * 根据用户名称模糊查询
3  * @param username
4  * @return
5  */
6 //@Select("select * from user where username like #{username}")
7 @Select("select * from user where username like '%${value}%' ")
8 List<User> findUserByName(String username);
```

聚合查询

```
1 @Select("select count(*) from user")
2 int findTotalUser();
```

保存@Insert

```
1 /**
2  * 保存用户
3  * @param user
4  */
5 @Insert(value = "insert into user(username, address, sex, birthday) values(#{username}, #{address}, #{sex}, #{birthday})")
6 void saveUser(User user);
```

更新@Update

```
1 /**
2  * 更新用户
3  * @param user
4  */
5 @Update(value = "update user set username=#{username}, sex=#{sex}, birthday=#{birthday}, address=#{address} where id=#{id} ")
6 void updateUser(User user);
```

删除@Delete

```
1 /**
2  * 删除用户
3  * @param userId
4  */
5 @Delete(value = "delete from user where id=#{id}")
6 void deleteUser(Integer userId);
```

参数 & 返回值 绑定 | 映射

参数

MyBatis 使用 **ognl 表达式** 解析对象字段的值。#{ } 或 \${ } 括号中的值为 pojo 属性名称

- Object Graphic Navigation Language , 对象图导航语言。是 apache 提供的一种表达式语言，按照一定的语法格式来获取数据的。
 - 语法格式就是使用 #{对象.对象} 的方式
 - \${user.username} 它会先去找 user 对象，然后在 user 对象中找到 username 属性，并调用 getUsername() 方法把值取出来。但是我们在 parameterType 属性上

指定了实体类名称，所以可以省略 `user.`
而直接写 `username`。

- 通过对对象的取值方法来获取数据。在写法上把get给省略了
- eg：获取用户名
 - 类中写法：`user.getUsername()`
 - OGNL写法：`#{{user.username}}`
- ?: MyBatis中为什么能直接写`username`，而不用`user.` 呢
 - 因为在 `paramType` 中已经提供了属性所属的类，所以此时不需要写 对象名
 - `paramType.getXXX()`
 - `#{{xxx}}`
 - 可以连续打点
 - `#{{xxx.bbb}}`
- sql 语句中使用 `#{{}}` 字符，代表占位符，相当于原来 jdbc 部分所学的?，都是用于执行语句时替换实际的数据。

`#{{}}`

- `#{{}}` 表示一个占位符
- 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。

`${{}}`

- `${{}}` 表示拼接 sql
- 通过 `${{}}` 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`。

xml方式

- 使用标签 的 `parameterType` 属性来设定。
 - 基本类型
 - 引用类型 (`String`)
 - 实体类 类型 (`pojo`)
 - 实体类 的 包装类
- mybaits 在加载时已经把常用的数据类型注册了别名，从而我们在使用时可以不写包名
- 基本类型和 `String` 我们可以直接写 类型名称，也可以使用 包名 . 类名 的方式
 - `String`

- java.lang.String
- 实体类类型，目前我们只能使用全限定类名
- 只有一个参数时，写什么都可以调用，不一定非要同名

简单类型

```

1 <select id="findByName" parameterType="string"
resultType="com.learn.domain.User">
2 <select id="findByName" parameterType="java.lang.String"
resultType="com.learn.domain.User">
3
4 <select id="findById" parameterType="int"
resultType="com.learn.domain.User">
5
6 <!-- 不区分大小写 int INT Integer INTEGER java.lang.Integer -->
7 <delete id="deleteUser" parameterType="Integer"><!-- int INT Integer
INTEGER java.lang.Integer -->
8     <!-- 只有一个参数 写什么都可以 -->
9     delete from user
10    where id = #{userId}
11 </delete>
```

POJO对象

MyBatis 使用 ognl表达式解析对象字段的值，`#{}` 或者 `${} 括号中的值为 pojo属性名称。`

```

1 <!-- 更新用户 -->
2 <update id="updateUser" parameterType="com.learn.domain.User">
3     update user set username=#{username}, address=#{address}, sex=#{sex},
birthday=#{birthday}
4     where id=#{id}
5 </update>
```

包装对象

```

1 <select id="findUserByVo" parameterType="com.learn.domain.QueryVo"
resultType="com.learn.domain.User">
2     select * from user where username like #{user.username}
3 </select>
```

注解方式

返回值

- 基本类型
- 引用类型 (String)
- 实体类 类型 (pojo)

- 实体类的包装类

(实体类.属性名 == 数据库.列名) 对应结果对象的属性为 null

xml 方式

(实体类.属性名 == 数据库.列名) 时

- 使用标签的 resultType 属性来设定。

(实体类.属性名 != 数据库.列名) 时，别名

- <resultMap> 标签 别名
- SQL语句 as 别名

简单类型

```

1 <select id="findTotal" resultType="int">
2   select count(*) from user;
3 </select>
```

pojo 对象

```

1 <select id="findAll" resultType="com.learn.domain.User">
2   select * from user
3   where id = 1
4 </select>
```

pojo 集合

```

1 <select id="findAll" resultType="com.learn.domain.User">
2   select * from user
3 </select>
```

SQL as 属性别名

(实体类.属性名 != 数据库.列名) as 别名

- 效率最高
- 书写麻烦
- 不建议

```

1 <select id="findAll" resultType="com.learn.domain.User">
2   select id as userId, username as userName, address as useAddress,
3         sex as userSex, birthday as userBirthday
4   from user
5 </select>
```

resultMap标签 属性别名

(实体类.属性名 != 数据库.列名) resultMap标签别名

- id 标签：用于指定主键字段
- result 标签：用于指定非主键字段
- column 属性：用于指定数据库列名
- property 属性：用于指定实体类属性名称

```
1 <!-- 配置 查询结果的列名 和 实体类的属性名 的对应关系 -->
2 <resultMap id="userMap" type="com.learn.domain.User">
3     <!-- property类中属性名 column数据库列名 -->
4     <!-- 主键字段的对应 -->
5     <id property="userId" column="id"></id>
6     <!-- 非主键字段的对应 -->
7     <result property="userName" column="username"></result>
8     <result property="useAddress" column="address"></result>
9     <result property="userSex" column="sex"></result>
10    <result property="userBirthday" column="birthday"></result>
11 </resultMap>
12
13 <select id="findAll" resultMap="userMap">
14     <!-- SQL语句 -->
15     select * from user;
16 </select>
```

注解方式

这里主要考虑别名。实现复杂关系映射之前我们可以在映射文件中通过配置 `<resultMap>` 来实现，在使用注解开发时我们需要借助`@Results`注解，`@Result`注解，`@One`注解，`@Many`注解。

- `@Results`
 - `@Result`
 - `@One`
 - `@Many`
- `@ResultMap`

@Results

代替的是标签 `<resultMap>`

该注解中可以使用单个`@Result`注解，也可以使用`@Result`集合

`@Results ({ @Result(), @Result() })` 或 `@Results (@Result())`

@Result

代替了 `<id>` 标签和 `<result>` 标签

`@Result` 中 属性介绍：

- `id` 是否是主键字段
- `column` 数据库的列名

- property 需要装配的属性名
- one 需要使用的@One 注解 (@Result(one=@One) ()))
- many 需要使用的@Many 注解 (@Result (many=@many) ()))

@One

代替了 `<assocation>` 标签，是多表查询的关键，在注解中用来指定子查询返回单一对象

@One 注解属性介绍：

- select 指定用的来多表查询的sqlmapper
- fetchType 会覆盖全局的配置参数 lazyLoadingEnabled

使用格式：

- `@Result(column="",property="",one=@One(select=""))`

@Many

代替了 `<collection>` 标签, 是多表查询的关键，在注解中用来指定子查询返回对象集合

使用格式：

- `@Result(property="",column="",many=@Many(select=""))`

@ResultMap

一对一

```

1 public interface AccountDao {
2
3     /**
4      * 查询所有账户，并且获取每个账户所属的用户信息
5      * @return
6      */
7     @Select("select * from account")
8     @Results(
9         id = "accountMap",
10        value = {
11            @Result(id = true, column = "id", property = "id"),
12            @Result(column = "uid", property = "uid"),
13            @Result(column = "money", property = "money"),
14            @Result(property = "user", column = "uid",
15                   one =
16                   @One(select="com.learn.dao.UserDao.findById", fetchType= FetchType.EAGER)
17               ),
18        }
19    )

```

```

19     List<Account> findAll();
20
21     /**
22      * 根据用户id查询账户信息
23      * @param userId
24      * @return
25      */
26     @Select("select * from account where uid = #{userId}")
27     List<Account> findAccountByUserId(Integer userId);
28 }

```

一对多

```

1 public interface UserDao {
2
3     /**
4      * 查询所有用户
5      * @return
6      */
7     @Select(value = "select * from user")
8     @Results( id = "userMap",
9             value = {
10                @Result(id =true, column = "id", property = "id"),
11                @Result(column = "username", property = "username"),
12                @Result(column = "address", property = "address"),
13                @Result(column = "sex", property = "sex"),
14                @Result(column = "birthday", property = "birthday"),
15                @Result(property = "accounts", column = "id",
16                      many = @Many(select =
"com.learn.dao.AccountDao.findAccountByUserId",
17                           fetchType = FetchType.LAZY)
18                ),
19            }
20        )
21     List<User> findAll();
22
23     /**
24      * 根据id查询用户
25      * @param userId
26      * @return
27      */
28     @Select(value = "select * from user where id=#{id}")
29     @ResultMap(value = {"userMap"})
30     User findById(Integer userId);
31
32     /**
33      * 根据用户名模糊查询
34      * @param username
35      * @return
36      */
37     @Select("select * from user where username like #{username}")
38     //    @Select("select * from user where username like '%${value}%' ")
39     @ResultMap(value = {"userMap"})
40     List<User> findUserByName(String username);
41

```

连接池

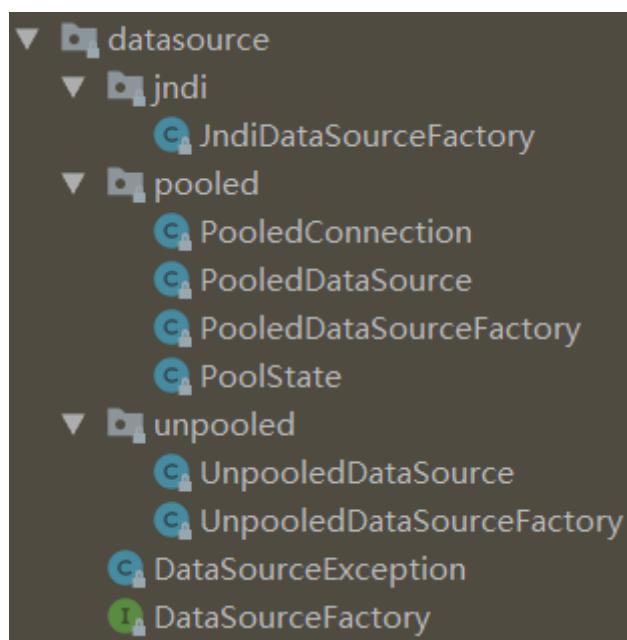
- MyBatis 采用的是自己的连接池技术
- 主配置文件 SqlMapConfig.xml , 通过 <dataSource type="pooled"> 来实现 Mybatis 中连接池的配置。
- MyBatis 是通过工厂模式来创建数据源 DataSource 对象的

MyBatis连接池分类

- MyBatis 数据源 : (共 3 类)

MyBatis 在初始化时 , 根据 <dataSource> 的 type 属性来创建相应类型的的数据源 DataSource。

- **UNPOOLED** 不使用连接池的数据源
 - 传统获取连接的方式
 - 虽然实现了 javax.sql.DataSource 接口 , 但并没有使用池的思想
 - MyBatis 会创建 UnpooledDataSource 实例
- **POOLED** 使用连接池的数据源
 - 采用传统的 javax.sql.DataSource 规范的连接池
 - MyBatis 中有针对规范的实现
 - MyBatis 会创建 PooledDataSource 实例
- **JNDI** 使用 JNDI 实现的数据源
 - 采用服务器提供的 JNDI 技术实现 , 来获取 DataSource 对象
 - 不同的服务器 所能拿到的 DataSource 是不一样的
 - MyBatis 会从 JNDI 服务上查找 DataSource 实例
 - 注 : 如果不是 web 或 maven 的 war 工程 , 是不能使用的
 - tomcat 采用 dbcp 连接池



- 数据源 就是为了更好的 管理数据库连接 , 也就是我们所说的 连接池技术
 - 连接池 就是一个 存储连接 的容器|集合
 - 该集合必须 线程安全 , 不能两个线程拿到同一连接
 - 该集合必须实现队列的特性 : 先进先出

MyBatis数据源配置

数据源配置就是在 SqlMapConfig.xml 文件中。

MyBatis 在初始化时 , 根据 `<dataSource>` 的 type 属性来创建相应类型的的数据源 DataSource , 即 :

- type="POOLED" : MyBatis 会创建 PooledDataSource 实例
- type="UNPOOLED" : MyBatis 会创建 UnpooledDataSource 实例
- type="JNDI" : MyBatis 会从 JNDI 服务上查找 DataSource 实例 , 然后返回使用

```
1 <!-- 配置数据源(连接池)信息 -->
2 <dataSource type="POOLED">
3   <property name="driver" value="${jdbc.driver}"/>
4   <property name="url" value="${jdbc.url}"/>
5   <property name="username" value="${jdbc.username}"/>
6   <property name="password" value="${jdbc.password}"/>
7 </dataSource>
```

MyBatis中DataSource的存取

MyBatis 是通过工厂模式来创建数据源 DataSource 对象的。

MyBatis 定义了抽象的工厂接口:org.apache.ibatis.datasource.DataSourceFactory,通过其 `getDataSource()`方法返回数据源DataSource。

MyBatis 创建了 DataSource 实例后 , 会将其放到 Configuration 对象内的 Environment 对象中 , 供以后使用。

- 先进入 XMLConfigBuilder 类中
- 分析 configuration 对象的 environment 属性

MyBatis中连接的获取

- 当我们需要创建 SqlSession 对象并需要执行 SQL 语句时 , 这时候 MyBatis 才会去调用 dataSource 对象来创建java.sql.Connection对象。
- java.sql.Connection对象的创建一直延迟到执行SQL语句的时候。

数据库连接是我们最为宝贵的资源 , 只有在要用到的时候 , 才去获取并打开连接 , 当我们用完了就再立即将数据库连接归还到连接池中。

UNPOOLED

UNPOOLED 不使用连接池的数据源

- 传统获取连接的方式
- 虽然实现了javax.sql.DataSource接口，但并没有使用池的思想

POOLED

POOLED 使用连接池的数据源

- 采用传统的 javax.sql.DataSource 规范的连接池
- MyBatis中有针对规范的实现

1. 空闲连接 有吗 ? `state.idleConnections.isEmpty()`

- 直接拿连接

2. 活动连接 数量 < 设定的最大值

- 新建连接

3. 获取 活动连接 中 最老的连接。设置清理后给我们用

- `state.activeConnections.get(0)`

有两个池：

- 空闲池 : idleConnections
- 活动池 : activeConnections

JNDI

- Java Naming and Directory Interface。Java命名 和 目录接口。
- Sun公司提供的一种 标准的 Java命名系统接口。
- 属于JavaEE 技术之一
- 目的：模仿windows系统的注册表

Map结构		JNDI	
注册表		tomcat服务器一启动	
key : 存的是路径+名称	value : 存的就是数据 在jndi中存的就是对象	key : 是一个字符串	value : 是一个Object
HKEY_USERS\DEFAULT \Control Panel\Accessibility \SoundSentry Flags	2	directory是固定的 name是可以自己指定的	要存放什么对象是可以指定的，指定的方式是通过配置文件的方式
HKEY_USERS\DEFAULT \Control Panel\Accessibility \StickyKeys Flags	510		

创建maven工程

- maven的webapp工程

pom

- servlet-api

- jsp-api

```
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <project xmlns="http://maven.apache.org/POM/4.0.0"
4   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
6   http://maven.apache.org/xsd/maven-4.0.0.xsd">
7   <modelVersion>4.0.0</modelVersion>
8
9   <groupId>com.grape</groupId>
10  <artifactId>MyBatis601JNDI</artifactId>
11  <version>1.0-SNAPSHOT</version>
12  <packaging>war</packaging>
13
14  <name>MyBatis601JNDI Maven Webapp</name>
15  <!-- FIXME change it to the project's website -->
16  <url>http://www.example.com</url>
17
18  <properties>
19    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
20    <maven.compiler.source>1.7</maven.compiler.source>
21    <maven.compiler.target>1.7</maven.compiler.target>
22  </properties>
23
24  <dependencies>
25    <dependency>
26      <groupId>org.mybatis</groupId>
27      <artifactId>mybatis</artifactId>
28      <version>3.5.1</version>
29    </dependency>
30    <dependency>
31      <groupId>mysql</groupId>
32      <artifactId>mysql-connector-java</artifactId>
33      <version>8.0.15</version>
34    </dependency>
35    <dependency>
36      <groupId>log4j</groupId>
37      <artifactId>log4j</artifactId>
38      <version>1.2.12</version>
39    </dependency>
40    <dependency>
41      <groupId>junit</groupId>
42      <artifactId>junit</artifactId>
43      <version>4.12</version>
44    </dependency>
45
46    <dependency>
47      <groupId>javax.servlet</groupId>
48      <artifactId> servlet-api</artifactId>
49      <version>2.5</version>
50    </dependency>
51    <dependency>
52      <groupId>javax.servlet.jsp</groupId>
53      <artifactId>jsp-api</artifactId>
54      <version>2.0</version>
55    </dependency>
```

```

54 </dependencies>
55
56 <build>
57   <finalName>MyBatis601JNDI</finalName>
58   <pluginManagement><!-- lock down plugins versions to avoid using Maven
defaults (may be moved to parent pom) -->
59     <plugins>
60       <plugin>
61         <artifactId>maven-clean-plugin</artifactId>
62         <version>3.1.0</version>
63       </plugin>
64       <!-- see http://maven.apache.org/ref/current/maven-core/default-
bindings.html#Plugin_bindings_for_war_packaging -->
65       <plugin>
66         <artifactId>maven-resources-plugin</artifactId>
67         <version>3.0.2</version>
68       </plugin>
69       <plugin>
70         <artifactId>maven-compiler-plugin</artifactId>
71         <version>3.8.0</version>
72       </plugin>
73       <plugin>
74         <artifactId>maven-surefire-plugin</artifactId>
75         <version>2.22.1</version>
76       </plugin>
77       <plugin>
78         <artifactId>maven-war-plugin</artifactId>
79         <version>3.2.2</version>
80       </plugin>
81       <plugin>
82         <artifactId>maven-install-plugin</artifactId>
83         <version>2.5.2</version>
84       </plugin>
85       <plugin>
86         <artifactId>maven-deploy-plugin</artifactId>
87         <version>2.8.2</version>
88       </plugin>
89     </plugins>
90   </pluginManagement>
91 </build>
92 </project>

```

建文件夹

- src/test/java
- main/java
- main/resources
- webapp/META-INF
 - context.xml
- make directory as

文件

context.xml

- 位置：META-INF/context.xml
- 配置tomcat相关信息，这里是配置了Tomcat的数据源

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <Context>
3 <!--
4 <Resource
5 name="jdbc/eesy_mybatis"          数据源的名称
6 type="javax.sql.DataSource"        数据源类型
7 auth="Container"                 数据源提供者
8 maxActive="20"                   最大活动数
9 maxWait="10000"                  最大等待时间
10 maxIdle="5"                     最大空闲数
11 username="root"                 用户名
12 password="1234"                 密码
13 driverClassName="com.mysql.jdbc.Driver" 驱动类
14 url="jdbc:mysql://localhost:3306/eesy_mybatis" 连接url字符串
15 />
16 -->
17 <Resource
18 name="jdbc/mybatis"
19 type="javax.sql.DataSource"
20 auth="Container"
21 maxActive="20"
22 maxWait="10000"
23 maxIdle="5"
24 username="root"
25 password="xxxx"
26 driverClassName="com.mysql.cj.jdbc.Driver"
27 url="jdbc:mysql://localhost:3306/mybatis"
28 />
29 </Context>
```

SqlMapConfig.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!-- 导入约束 -->
3 <!DOCTYPE configuration
4     PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
5     "http://mybatis.org/dtd/mybatis-3-config.dtd">
6 <configuration>
7     <typeAliases>
8         <package name="com.learn.domain"></package>
9     </typeAliases>
10    <!-- 配置mybatis的环境 -->
11    <environments default="mysql">
12        <!-- 配置mysql的环境 -->
13        <environment id="mysql">
14            <!-- 配置事务控制的方式 -->
15            <transactionManager type="JDBC"></transactionManager>
16            <!-- 配置连接数据库的必备信息 type属性表示是否使用数据源（连接池）-->
```

```

17      <dataSource type="JNDI">
18          <!-- value前缀固定 后缀是context里的name -->
19          <property name="data_source"
20              value="java:comp/env/jdbc/mybatis"/>
21      </dataSource>
22  </environment>
23</environments>
24  <!-- 指定mapper配置文件的位置 -->
25  <mappers>
26      <mapper resource="com/learn/dao/UserDao.xml"/>
27  </mappers>
28</configuration>

```

index.jsp

```

1 <html>
2 <body>
3 <h2>Hello world!</h2>
4<%
5     InputStream in = Resources.getResourceAsStream("SqlMapConfig.xml");
6     SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
7     SqlSessionFactory factory = builder.build(in);
8     SqlSession session1 = factory.openSession();
9     UserDao userDao = session1.getMapper(UserDao.class);
10    List<User> users = userDao.findAll();
11    for (User user : users) {
12        System.out.println(user);
13    }
14    session1.close();
15    in.close();
16 %>
17 </body>
18 </html>

```

事务控制

MyBatis中的事务

- 通过 SqlSession 对象的 commit 方法 和 rollback方法实现事务的提交和回滚

MyBatis默认 autocommit 为 false ; 表示开启事务

- 手动提交
- 设置为true , 自动提交

手动提交

```
1 | session.commit();
```

自动提交

```
1 | session = factory.openSession(true);
```

动态SQL

前面我们的 SQL 都是比较简单的，有些时候业务逻辑复杂时，我们的 SQL 是动态变化的，此时在前面的学习中我们的 SQL 就不能满足要求了。

xml方式

if

- 根据实体类的不同取值，使用不同的 SQL 语句来进行查询。
- 比如在 id 如果不为空时可以根据 id 查询，如果 username 不同空时还要加入用户名作为条件。
- 这种情况在我们的多条件组合查询中经常会碰到。
- 单独使用搭配 where 1=1

```
1 <select id="findUserByCondition" parameterType="com.learn.domain.User"
2   resultType="com.learn.domain.User">
3     select * from user where 1=1
4     <if test="username != null"><!-- 这的username是实体类中的属性 -->
5       and username=#{username}
6     </if>
7   </select>
```

where

- 简化 where 1=1 的条件拼装

```
1 <select id="findUserByCondition" parameterType="com.learn.domain.User"
2   resultType="com.learn.domain.User">
3     select * from user
4     <where> <!-- 不用写 1=1了 -->
5       <if test="username != null"><!-- 这的username是实体类中的属性 -->
6         and username=#{username}
7       </if>
8       <if test="sex != null">
9         and sex=#{sex}
10      </if>
11    </where>
12  </select>
```

for-each

- 在进行范围查询时，就要将一个集合中的值，作为参数动态添加进来。
- 用于遍历集合
 - collection:代表要遍历的集合元素，注意编写时不要写#{}，而是直接写\${}
 - open:代表语句的开始部分
 - close:代表结束部分
 - item:代表遍历集合的每个元素，生成的变量名
 - separator:代表分隔符

```

1 <!-- 根据queryvo中的id集合实现查询用户列表 -->
2 <!-- select * from user where id in (1,2,3,4,5); -->
3 <select id="findUserInIds" parameterType="com.learn.domain.QueryVo"
resultType="com.learn.domain.User">
4   select * from user
5   <include refid="defaultUser"></include>
6   <where>
7     <if test="ids != null and ids.size()>0">
8       <foreach collection="ids" open="and id in(" close=")" item="id"
separator=",">
9         #{id}
10        </foreach>
11      </if>
12    </where>
13 </select>
```

抽取重复SQL语句

```

1 <!-- 抽取重复的sql语句 -->
2 <sql id="defaultUser">
3   select * from user
4 </sql>
5
6 <select id="findUserInIds" parameterType="com.learn.domain.QueryVo"
resultType="com.learn.domain.User">
7   /*select * from user*/
8   <include refid="defaultUser"></include>
9   <where>
10    <if test="ids != null and ids.size()>0">
11      <foreach collection="ids" open="and id in(" close=")" item="id"
separator=",">
12        #{id}
13        </foreach>
14      </if>
15    </where>
16 </select>
```

注解方式

多表查询

需要图

表之间的关系：

- 一对多 | 多对一
 - 一个用户 多个订单
 - 一个用户 多个账户
 - 由于一个账户 只能 对应一个用户，MyBatis把多对一 看成 一对一
- 一对一
 - 一个人 一个身份证号
- 多对多
 - 学生 老师

xml方式:一对一

association

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE mapper
3 PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4 "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5 <mapper namespace="com.itheima.dao.IAccountDao">
6     <!-- 建立对应关系 -->
7     <resultMap type="account" id="accountMap">
8         <id column="aid" property="id"/>
9         <result column="uid" property="uid"/>
10        <result column="money" property="money"/>
11        <!-- 它是用于指定从表方的引用实体属性的 -->
12        <association property="user" javaType="user"
13             select="com.itheima.dao.IUserDao.findById"
14             column="uid">
15            </association>
16        </resultMap>
17        <select id="findAll" resultMap="accountMap">
18            select * from account
19        </select>
20    </mapper>
```

xml方式:一对多 | 多对一

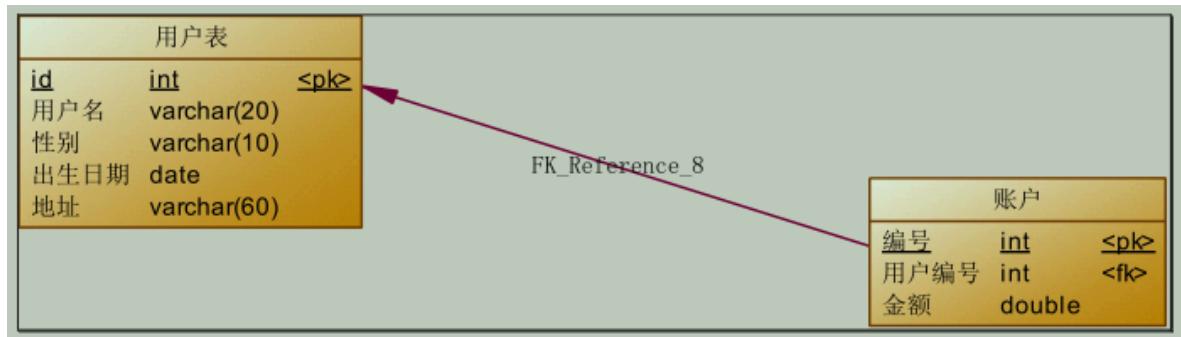
- 一个用户 多个账户
- 由于一个账户 只能 对应一个用户，MyBatis把多对一 看成 一对一

建表

- 数据库表：用户表、账户表
 - 一个用户 可以 有 多个 账户
 - 一个账户 只能 属于 一个 用户

- 让用户表 和 账户表之间具备一对多的关系。

- 需要使用外键 在账户表中添加



用户表

```

1 | CREATE TABLE `user` (
2 |   `id` int NOT NULL AUTO_INCREMENT,
3 |   `username` varchar(32) NOT NULL COMMENT '用户名',
4 |   `birthday` datetime DEFAULT NULL COMMENT '生日',
5 |   `sex` char(1) DEFAULT NULL COMMENT '性别',
6 |   `address` varchar(256) DEFAULT NULL COMMENT '地址',
7 |   PRIMARY KEY (`id`)
8 | ) ENGINE=InnoDB AUTO_INCREMENT=56 DEFAULT CHARSET=utf8;

```

账户表

```

1 | CREATE TABLE `account` (
2 |   `ID` int NOT NULL COMMENT '编号',
3 |   `UID` int DEFAULT NULL COMMENT '用户编号',
4 |   `MONEY` double DEFAULT NULL COMMENT '金额',
5 |   PRIMARY KEY (`ID`),
6 |   KEY `FK_Reference_8` (`UID`),
7 |   CONSTRAINT `FK_Reference_8` FOREIGN KEY (`UID`) REFERENCES `user` (`id`)
8 | ) ENGINE=InnoDB DEFAULT CHARSET=utf8;

```

建立实体类

- 实体类：用户类、账户类
 - 用户 和 账户 的实体类能体现出一对多 的关系

用户类

- 一对多关系映射：主表实体应该包含从表实体的引用集合
- `private List<Account> accounts;`

```

1 | public class User implements Serializable {
2 |
3 |   private Integer id;
4 |   private String username;

```

```
5     private String address;
6     private String sex;
7     private Date birthday;
8
9     //一对多关系映射：主表实体应该包含从表实体的引用集合
10    private List<Account> accounts;
11
12    public Integer getId() {
13        return id;
14    }
15
16    public void setId(Integer id) {
17        this.id = id;
18    }
19
20    public String getUsername() {
21        return username;
22    }
23
24    public void setUsername(String username) {
25        this.username = username;
26    }
27
28    public String getAddress() {
29        return address;
30    }
31
32    public void setAddress(String address) {
33        this.address = address;
34    }
35
36    public String getSex() {
37        return sex;
38    }
39
40    public void setSex(String sex) {
41        this.sex = sex;
42    }
43
44    public Date getBirthday() {
45        return birthday;
46    }
47
48    public void setBirthday(Date birthday) {
49        this.birthday = birthday;
50    }
51
52    public List<Account> getAccounts() {
53        return accounts;
54    }
55
56    public void setAccounts(List<Account> accounts) {
57        this.accounts = accounts;
58    }
59
60    @Override
61    public String toString() {
62        return "User{" +
```

```
63         "id=" + id +
64         ", username='" + username + '\'' +
65         ", address='" + address + '\'' +
66         ", sex='" + sex + '\'' +
67         ", birthday='" + birthday +
68         '}';
69     }
70 }
71 }
```

账户类

- 从表实体应该包含一个主表实体的对象引用
- `private User user;`

```
1 public class Account implements Serializable {
2
3     private Integer id;
4     private Integer uid;
5     private Double money;
6
7     //从表实体应该包含一个主表实体的对象引用
8     private User user;
9
10    public Integer getId() {
11        return id;
12    }
13
14    public void setId(Integer id) {
15        this.id = id;
16    }
17
18    public Integer getUserId() {
19        return uid;
20    }
21
22    public void setUserId(Integer uid) {
23        this.uid = uid;
24    }
25
26    public Double getMoney() {
27        return money;
28    }
29
30    public void setMoney(Double money) {
31        this.money = money;
32    }
33
34    public User getUser() {
35        return user;
36    }
37
38    public void setUser(User user) {
39        this.user = user;
40    }
41 }
```

```
41
42     @Override
43     public String toString() {
44         return "Account{" +
45             "id=" + id +
46             ", uid=" + uid +
47             ", money=" + money +
48             '}';
49     }
50 }
```

封装信息类

定义专门的 po 类作为输出类型，其中定义了 sql 查询结果集所有的字段。此方法较为简单，企业中使用普遍。

```
1  public class AccountUser extends Account {
2
3      private String username;
4      private String address;
5
6      public String getUsername() {
7          return username;
8      }
9
10     public void setUsername(String username) {
11         this.username = username;
12     }
13
14     public String getAddress() {
15         return address;
16     }
17
18     public void setAddress(String address) {
19         this.address = address;
20     }
21
22     @Override
23     public String toString() {
24         return super.toString() + " AccountUser{" +
25             "username='" + username + '\'' +
26             ", address='" + address + '\'' +
27             '}';
28     }
29 }
30 }
```

两个接口

持久层接口

UserDao.java

```
1 public interface UserDao {  
2  
3     /**  
4      * 查询所有用户  
5      *      同时获取 用户下 所有账户的信息  
6      * @return  
7      */  
8     List<User> findAll();  
9  
10    /**  
11     * 根据id查询用户信息  
12     * @param userId  
13     * @return  
14     */  
15     User findById(Integer userId);  
16  
17 }
```

AccountDao.java

```
1 public interface AccountDao {  
2  
3     /**  
4      * 查询所有账户 同时还要获取到当前账户的所属用户信息  
5      *      select u.*, a.id as aid, a.uid, a.money from account a, user u  
6      * where u.id=a.uid;  
7      * @return  
8      */  
9     List<Account> findAll();  
10  
11    /**  
12     *  
13     * 查询所有账户，  
14     *      并且带有用户名和地址信息  
15     *      账户信息+ username + address 需要封装个类啊  
16     * @return  
17     */  
18     List<AccountUser> findAllAccount();  
19 }
```

映射配置文件

- 用户的 配置文件
- 账户的 配置文件

UserDao.xml

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

```

5   <mapper namespace="com.learn.dao.UserDao">
6
7     <!-- 定义User的resultMap -->
8     <resultMap id="userAccountMap" type="user">
9       <id property="id" column="id"></id>
10      <result property="username" column="username"></result>
11      <result property="address" column="address"></result>
12      <result property="sex" column="sex"></result>
13      <result property="birthday" column="birthday"></result>
14      <!-- 配置user对象中accounts集合的映射 -->
15      <collection property="accounts" ofType="com.learn.domain.Account">
16        <!-- account别名 -->
17        <id property="id" column="aid"></id>
18        <result property="uid" column="uid"></result>
19        <result property="money" column="money"></result>
20      </collection>
21    </resultMap>
22
23    <select id="findAll" resultMap="userAccountMap">
24      select * from user u left outer join account a on u.id = a.uid
25    </select>
26
27    <select id="findById" parameterType="int"
28      resultType="com.learn.domain.User">
29      select * from user
30      where id = #{suibianxie}
31    </select>
32  </mapper>

```

AccountDao.xml

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE mapper
3    PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5
6  <mapper namespace="com.learn.dao.AccountDao">
7
8    <!-- 定义封装account和user的resultMap -->
9    <resultMap id="accountUserMap" type="com.learn.domain.Account">
10      <id property="id" column="aid"></id>
11      <result property="uid" column="uid"></result>
12      <result property="money" column="money"></result>
13      <!-- 一对一的关系映射，配置封装user的内容 -->
14      <association property="user" column="uid"
15        javaType="com.learn.domain.User">
16        <id property="id" column="id"></id>
17        <result column="username" property="username"></result>
18        <result column="address" property="address"></result>
19        <result column="sex" property="sex"></result>
20        <result column="birthday" property="birthday"></result>
21      </association>
22    </resultMap>

```

```

23     <select id="findAll" resultMap="accountUserMap">
24         select u.*, a.id as aid, a.uid, a.money from account a, user u
25         where u.id=a.uid;
26     </select>
27
28     <!-- 查询所有账户同时包含用户名和地址信息 -->
29     <select id="findAllAccount" resultType="com.learn.domain.AccountUser">
30         select a.*, u.username, u.address from account a, user u where
31         u.id=a.uid
32     </select>
33 </mapper>

```

AccountDao.findAll

接口

- 查询所有账户 同时还要获取到当前账户的所属用户信息

```

1 /**
2  * 查询所有账户 同时还要获取到当前账户的所属用户信息
3  *      select u.*, a.id as aid, a.uid, a.money from account a, user u where
4  *      u.id=a.uid;
5  * @return
6  */
6 List<Account> findAll();

```

映射文件

这里Account已经持有user引用

```

1 <select id="findAll" resultMap="accountUserMap">
2     select u.*, a.id as aid, a.uid, a.money from account a, user u where
3     u.id=a.uid;
4 </select>
5
6 <!-- 定义封装account和user的resultMap -->
7 <resultMap id="accountUserMap" type="com.learn.domain.Account">
8     <id property="id" column="aid"></id>
9     <result property="uid" column="uid"></result>
10    <result property="money" column="money"></result>
11    <!-- 一对一的关系映射，配置封装user的内容 -->
12    <!-- javaType用来提示 封装的是哪个对象 -->
13    <association property="user" column="uid"
14        javaType="com.learn.domain.User">
15        <id property="id" column="id"></id>
16        <result column="username" property="username"></result>
17        <result column="address" property="address"></result>
18        <result column="sex" property="sex"></result>
19        <result column="birthday" property="birthday"></result>
20    </association>
21 </resultMap>

```

测试类

```
1  @Test
2  public void testAccountfindAll() {
3      List<Account> accounts = accountDao.findAll();
4      for (Account account : accounts) {
5          System.out.println(account);
6          System.out.println(account.getUser());
7      }
8  }
```

AccountDao.findAllAccount

- 带有用户名称和地址信息

接口

```
1 /**
2  * 查询所有账户,
3  * 并且带有用户名称和地址信息
4  * 账户信息+ username + address 需要封装个类啊
5  * @return
6  */
7 List<AccountUser> findAllAccount();
```

映射文件

```
1 <!-- 查询所有账户同时包含用户名和地址信息 -->
2 <select id="findAllAccount" resultType="com.learn.domain.AccountUser">
3     select a.*, u.username, u.address from account a, user u where u.id=a.uid
4 </select>
```

测试类

```
1 @Test
2 public void testAccountfindAllAccount() {
3     List<AccountUser> aus = accountDao.findAllAccount();
4     for (AccountUser au : aus) {
5         System.out.println(au);
6     }
7 }
```

UserDao.findAll

- 当我们查询用户时，可以同时得到用户下所包含的账户信息

接口

```

1 /**
2  * 查询所有用户
3  *      同时获取 用户下 所有账户的信息
4  * @return
5 */
6 List<User> findAll();

```

映射文件

```

1 <select id="findAll" resultMap="userAccountMap">
2     select * from user u left outer join account a on u.id = a.uid
3 </select>
4
5 <!-- 定义User的resultMap -->
6 <resultMap id="userAccountMap" type="user">
7     <id property="id" column="id"></id>
8     <result property="username" column="username"></result>
9     <result property="address" column="address"></result>
10    <result property="sex" column="sex"></result>
11    <result property="birthday" column="birthday"></result>
12    <!-- 配置user对象中accounts集合的映射 -->
13    <!-- -->
14    <collection property="accounts" ofType="com.learn.domain.Account"><!--
15        account别名 -->
16        <id property="id" column="aid"></id>
17        <result property="uid" column="uid"></result>
18        <result property="money" column="money"></result>
19    </collection>
</resultMap>

```

测试类

```

1 @Test
2 public void testUserfindAll() {
3     List<User> users = userDao.findAll();
4     for (User user : users) {
5         System.out.println("每个用户的信息");
6         System.out.println(user);
7         System.out.println(user.getAccounts());
8     }
9 }

```

xml方式:多对多

建表

- 数据库表：用户表、角色表
 - 用户有多个角色
 - 角色可以赋予多个用户
- 用户表和角色表具有多对多的关系

- 使用中间表
- 中间表包含各表的主键，它们在中间表这里是外键



用户表

```

1 CREATE TABLE `user` (
2   `id` int NOT NULL AUTO_INCREMENT,
3   `username` varchar(32) NOT NULL COMMENT '用户名',
4   `birthday` datetime DEFAULT NULL COMMENT '生日',
5   `sex` char(1) DEFAULT NULL COMMENT '性别',
6   `address` varchar(256) DEFAULT NULL COMMENT '地址',
7   PRIMARY KEY (`id`)
8 ) ENGINE=InnoDB AUTO_INCREMENT=56 DEFAULT CHARSET=utf8;

```

角色表

```

1 CREATE TABLE `role` (
2   `ID` int NOT NULL COMMENT '编号',
3   `ROLE_NAME` varchar(30) DEFAULT NULL COMMENT '角色名称',
4   `ROLE_DESC` varchar(60) DEFAULT NULL COMMENT '角色描述',
5   PRIMARY KEY (`ID`)
6 ) ENGINE=InnoDB DEFAULT CHARSET=utf8;

```

中间表

```

1 CREATE TABLE `user_role` (
2   `UID` int NOT NULL COMMENT '用户编号',
3   `RID` int NOT NULL COMMENT '角色编号',
4   PRIMARY KEY (`UID`, `RID`),
5   KEY `FK_Reference_10` (`RID`),
6   CONSTRAINT `FK_Reference_10` FOREIGN KEY (`RID`) REFERENCES `role` (`ID`),
7   CONSTRAINT `FK_Reference_9` FOREIGN KEY (`UID`) REFERENCES `user` (`id`)
8 ) ENGINE=InnoDB DEFAULT CHARSET=utf8;

```

建实体类

- 用户和角色的实体类能体现多对多的关系
- 各自包含对方一个集合引用

用户类

- 多对多的关系映射：一个用户可以具备多个角色
- `private List<Role> roles`

```
1 public class User implements Serializable {
2
3     private Integer id;
4     private String username;
5     private String address;
6     private String sex;
7     private Date birthday;
8
9     //多对多的关系映射：一个用户可以具备多个角色
10    private List<Role> roles;
11
12    public Integer getId() {
13        return id;
14    }
15
16    public void setId(Integer id) {
17        this.id = id;
18    }
19
20    public String getUsername() {
21        return username;
22    }
23
24    public void setUsername(String username) {
25        this.username = username;
26    }
27
28    public String getAddress() {
29        return address;
30    }
31
32    public void setAddress(String address) {
33        this.address = address;
34    }
35
36    public String getSex() {
37        return sex;
38    }
39
40    public void setSex(String sex) {
41        this.sex = sex;
42    }
43
44    public Date getBirthday() {
45        return birthday;
46    }
47
48    public void setBirthday(Date birthday) {
49        this.birthday = birthday;
50    }
51
52    public List<Role> getRoles() {
53        return roles;
54    }
55
56    public void setRoles(List<Role> roles) {
57        this.roles = roles;
```

```

58 }
59
60     @Override
61     public String toString() {
62         return "User{" +
63             "id=" + id +
64             ", username='" + username + '\'' +
65             ", address='" + address + '\'' +
66             ", sex='" + sex + '\'' +
67             ", birthday='" + birthday +
68             '}';
69     }
70 }

```

角色类

- 多对多的关系映射：一个角色可以赋予多个用户
- `private List<User> users`

```

1  public class Role implements Serializable {
2
3      private Integer roleId;
4      private String roleName;
5      private String roleDesc;
6
7      //多对多的关系映射：一个角色可以赋予多个用户
8      private List<User> users;
9
10     public Integer getRoleId() {
11         return roleId;
12     }
13
14     public void setRoleId(Integer roleId) {
15         this.roleId = roleId;
16     }
17
18     public String getRoleName() {
19         return roleName;
20     }
21
22     public void setRoleName(String roleName) {
23         this.roleName = roleName;
24     }
25
26     public String getRoleDesc() {
27         return roleDesc;
28     }
29
30     public void setRoleDesc(String roleDesc) {
31         this.roleDesc = roleDesc;
32     }
33
34     public List<User> getUsers() {
35         return users;
36     }

```

```
37     public void setUsers(List<User> users) {
38         this.users = users;
39     }
40
41     @Override
42     public String toString() {
43         return "Role{" +
44             "roleId=" + roleId +
45             ", roleName='" + roleName + '\'' +
46             ", roleDesc='" + roleDesc + '\'' +
47             '}';
48     }
49 }
50 }
```

接口

UserDao

```
1 /**
2  * 用户的持久层接口
3 */
4 public interface UserDao {
5
6     /**
7      * 查询所有用户
8      *          同时获取 用户下 所有账户的信息
9      * @return
10     */
11     List<User> findAll();
12
13     /**
14      * 根据id查询用户信息
15      * @param userId
16      * @return
17      */
18     User findById(Integer userId);
19
20 }
21 }
```

RoleDao

```
1 public interface RoleDao {
2
3     /**
4      * 查询所有角色
5      * @return
6      */
7     List<Role> findAll();
8
9 }
```

映射配置文件

- 查询用户时，可以同时得到 用户下 所包含的角色信息
- 查询角色时，可以同时得到 角色的 所属 用户信息

UserDao.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE mapper
3     PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4     "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5
6 <mapper namespace="com.learn.dao.UserDao">
7
8     <!-- 定义User的resultMap -->
9     <resultMap id="userMap" type="user">
10        <id property="id" column="id"></id>
11        <result property="username" column="username"></result>
12        <result property="address" column="address"></result>
13        <result property="sex" column="sex"></result>
14        <result property="birthday" column="birthday"></result>
15        <!-- 配置角色集合的映射 -->
16        <collection property="roles" ofType="role">
17            <id property="roleId" column="rid"></id>
18            <result property="roleName" column="role_name"></result>
19            <result property="roleDesc" column="role_desc"></result>
20        </collection>
21    </resultMap>
22
23    <!-- 查询所有 -->
24    <select id="findAll" resultMap="userMap">
25        select u.*, r.id as rid, r.role_name, r.role_desc from user u
26            left outer join user_role ur on u.id = ur.uid
27            left outer join role r on r.id = ur.rid
28    </select>
29 </mapper>
```

RoleDao.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE mapper
3     PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4     "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5
6 <mapper namespace="com.learn.dao.RoleDao">
7     <!-- 定义role表的ResultMap -->
8     <resultMap id="roleMap" type="role">
9         <id property="roleId" column="rid"></id>
10        <result property="roleName" column="role_name"></result>
11        <result property="roleDesc" column="role_desc"></result>
12        <collection property="users" ofType="com.learn.domain.User">
13            <id property="id" column="id"></id>
14            <result property="username" column="username"></result>
```

```

15         <result property="address" column="address"></result>
16         <result property="sex" column="sex"></result>
17         <result property="birthday" column="birthday"></result>
18     </collection>
19 </resultMap>
20
21     <!-- 查询所有 -->
22     <select id="findAll" resultMap="roleMap">
23         select u.*, r.id as rid, r.role_name, r.role_desc from role r
24             left outer join user_role ur on r.id = ur.rid
25             left outer join user u on u.id = ur.uid
26     </select>
27
28 </mapper>

```

UserDao.findAll

接口

```

1 /**
2  * 查询所有用户
3  *      同时获取 用户下 所有账户的信息
4  * @return
5  */
6 List<User> findAll();

```

映射文件

```

1     <!-- 查询所有 -->
2     <select id="findAll" resultMap="userMap">
3         select u.*, r.id as rid, r.role_name, r.role_desc from user u
4             left outer join user_role ur on u.id = ur.uid
5             left outer join role r on r.id = ur.rid
6     </select>

```

测试类

```

1 @Test
2 public void testUserfindAll() {
3     List<User> users = userDao.findAll();
4     for (User user : users) {
5         System.out.println("用户信息");
6         System.out.println(user);
7         System.out.println(user.getRoles());
8     }
9 }

```

RoleDao.findAll

接口

```
1 /**
2  * 查询所有角色
3  * @return
4  */
5 List<Role> findAll();
```

映射文件

```
1 <!-- 查询所有 -->
2 <select id="findAll" resultMap="roleMap">
3     select u.*, r.id as rid, r.role_name, r.role_desc from role r
4     left outer join user_role ur on r.id = ur.rid
5     left outer join user u on u.id = ur.uid
6 </select>
```

测试类

```
1 @Test
2 public void testRolefindAll() {
3     List<Role> roles = roleDao.findAll();
4     for (Role role : roles) {
5         System.out.println("--每个角色信息--");
6         System.out.println(role);
7         System.out.println(role.getUsers());
8     }
9 }
```

注解方式:一对一

```
1 public interface AccountDao {
2
3     /**
4      * 查询所有账户，并且获取每个账户所属的用户信息
5      * @return
6      */
7     @Select("select * from account")
8     @Results(
9         id = "accountMap",
10        value = {
11            @Result(id = true, column = "id", property = "id"),
12            @Result(column = "uid", property = "uid"),
13            @Result(column = "money", property = "money"),
14            @Result(property = "user", column = "uid",
```

```

15         one =
16     @One(select="com.learn.dao.UserDao.findById", fetchType= FetchType.EAGER)
17             ),
18         }
19     List<Account> findAll();
20
21 /**
22 * 根据用户id查询账户信息
23 * @param userId
24 * @return
25 */
26 @Select("select * from account where uid = #{userId}")
27 List<Account> findAccountByUserId(Integer userId);
28 }
```

注解方式:一对多 | 多对一

```

1 public interface UserDao {
2
3     /**
4      * 查询所有用户
5      * @return
6      */
7     @Select(value = "select * from user")
8     @Results( id = "userMap",
9             value = {
10                 @Result(id =true, column = "id", property = "id"),
11                 @Result(column = "username", property = "username"),
12                 @Result(column = "address", property = "address"),
13                 @Result(column = "sex", property = "sex"),
14                 @Result(column = "birthday", property = "birthday"),
15                 @Result(property = "accounts", column = "id",
16                         many = @Many(select =
"com.learn.dao.AccountDao.findAccountByUserId",
17                                     fetchType = FetchType.LAZY)
18                     ),
19                 }
20             )
21     List<User> findAll();
22
23 /**
24 * 根据id查询用户
25 * @param userId
26 * @return
27 */
28     @Select(value = "select * from user where id=#{id}")
29     @ResultMap(value = {"userMap"})
30     User findById(Integer userId);
31
32 /**
33 * 根据用户名称模糊查询
34 * @param username
```

```

35     * @return
36     */
37     @Select("select * from user where username like #{username}")
38 //     @Select("select * from user where username like '%${value}%' ")
39     @ResultMap(value = {"userMap"})
40     List<User> findUserByName(String username);
41
42 }

```

注解方式:多对多

加载时机

MyBatis的 延迟加载 其实就是：

- 多表情况下，复杂SQL 可以看做是 多次单表查询
- 我们只提供第一次最简单的SQL单表查询
 - 第一次单表查询：简单属性封装到类中
 - 再次查询：封装到引用中
 - 简单对象
 - 集合
- 根据单表查询结果的再次查询，我们配置到resultMap中
- 再次查询由MyBatis自动执行
- 我们可以控制其执行时机
 - 立刻执行——立即
 - 使用到该数据在执行——延迟

使用：

- 主配置文件，开启支持
- 映射配置
 - xml
 - 注解

复杂SQL 多表查询

```

1  <!-- 定义封装account和user的resultMap -->
2  <resultMap id="accountUserMap" type="com.learn.domain.Account">
3      <id property="id" column="aid"></id>
4      <result property="uid" column="uid"></result>
5      <result property="money" column="money"></result>
6  <!-- 一对一的关系映射，配置封装user的内容 -->

```

```

7     <association property="user" column="uid"
8         javaType="com.learn.domain.User">
9             <id property="id" column="id"></id>
10            <result column="username" property="username"></result>
11            <result column="address" property="address"></result>
12            <result column="sex" property="sex"></result>
13            <result column="birthday" property="birthday"></result>
14        </association>
15    </resultMap>
16
17    <select id="findAll" resultMap="accountUserMap">
18        select u.*, a.id as aid, a.uid, a.money from account a, user u where
19        u.id=a.uid;
20    </select>

```

问题

查询结果的类：

- 基本属性
- 引用
 - 普通对象
 - 集合

? : 是否在一开始就给引用对象赋值呢，有时可能需要很多内存，但我们又不用

概念:立即 ? 延迟

- 立即加载
 - 不管用不用，只要一调用方法，马上发起查询。
- 延迟加载(懒加载)
 - 在真正使用数据时才发起查询，不用的时候不查询。按需加载
 - resultMap中设置再次查询的信息，需要数据时，会去根据信息生成SQL查询
 - 原SQL映射处，必然执行

主配置文件

配置延迟加载：

- lazyLoadingEnabled
 - 支持延迟加载
- aggressiveLazyLoading
 - 触发方法立即加载

```
1 <!-- 配置参数 -->
2 <settings>
3     <!-- 开启MyBatis支持延迟加载 -->
4     <setting name="lazyLoadingEnabled" value="true"/>
5     <!-- 触发方法立即加载 否则按需加载 -->
6     <setting name="aggressiveLazyLoading" value="false"/>
7 </settings>
```

完整版

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE configuration
3     PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4     "http://mybatis.org/dtd/mybatis-3-config.dtd">
5
6 <configuration>
7     <properties resource="jdbcConfig.properties"></properties>
8
9     <!-- 配置参数 -->
10    <settings>
11        <!-- 开启MyBatis支持延迟加载 -->
12        <setting name="lazyLoadingEnabled" value="true"/>
13        <!-- 触发方法立即加载 否则按需加载 -->
14        <setting name="aggressiveLazyLoading" value="false"/>
15    </settings>
16
17    <!-- 使用typeAliases 配置别名。它只能配置domain中类的别名 -->
18    <typeAliases>
19        <package name="com.learn.domain"></package>
20    </typeAliases>
21
22    <!-- 配置环境 -->
23    <environments default="mysql">
24        <environment id="mysql">
25            <!-- 配置事务 -->
26            <transactionManager type="JDBC"></transactionManager>
27            <!-- 配置连接池 -->
28            <dataSource type="POOLED">
29                <!-- 配置连接数据库的4个基本信息 -->
30                <property name="driver" value="${jdbc.driver}"/>
31                <property name="url" value="${jdbc.url}"/>
32                <property name="username" value="${jdbc.username}"/>
33                <property name="password" value="${jdbc.password}"/>
34            </dataSource>
35        </environment>
36    </environments>
37    <!-- 指定映射配置文件的位置 -->
38    <mappers>
39        <package name="com.learn.dao"></package>
40    </mappers>
41 </configuration>
```

xml方式

- 普通对象引用
- 集合对象引用

association延迟加载

普通对象引用 : (一对关系映射)

- select
 - 查询用户的唯一标志。使用什么方法进行 关联表的关联查询
- column
 - 用户根据id查询时 , 所需要的参数的值。给上面方法的传参
- property
 - 封装到类中的哪个属性 | 引用
- javaType
 - 封装成什么

```

1 <!-- 定义封装account和user的resultMap -->
2 <resultMap id="accountUserMap" type="com.learn.domain.Account">
3   <id property="id" column="id"></id>
4   <result property="uid" column="uid"></result>
5   <result property="money" column="money"></result>
6   <!-- 一对一的关系映射, 配置封装user的内容
7     select 属性指定的内容:
8       查询用户的唯一标志
9     column 属性指定的内容:
10       用户根据id查询时, 所需要的参数的值
11   -->
12   <association property="user" column="uid"
13     javaType="com.learn.domain.User"
14       select="com.learn.dao.UserDao.findById">
15   </association>
16 </resultMap>
17
18 <!-- 查询所有 -->
19 <select id="findAll" resultMap="accountUserMap">
20   select * from account
</select>
```

完整xml

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE mapper
3   PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4   "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5
6 <mapper namespace="com.learn.dao.AccountDao">
7
8   <!-- 定义封装account和user的resultMap -->
9   <resultMap id="accountUserMap" type="com.learn.domain.Account">
10    <id property="id" column="id"></id>
11    <result property="uid" column="uid"></result>
12    <result property="money" column="money"></result>
```

```

13      <!-- 一对一的关系映射，配置封装user的内容
14          select 属性指定的内容:
15              查询用户的唯一标志
16          column 属性指定的内容:
17              用户根据id查询时，所需要的参数的值
18      -->
19      <association property="user" column="uid" javaType="user"
20          select="com.learn.dao.UserDao.findById">
21      </association>
22  </resultMap>
23
24  <!-- 查询所有 -->
25  <select id="findAll" resultMap="accountUserMap">
26      select * from account
27  </select>
28
29  <!-- 根据用户id查询账户列表 -->
30  <select id="findAccountByUid" resultType="com.learn.domain.Account">
31      select * from account where uid=#{uid}
32  </select>
33 </mapper>

```

collection延迟加载

<collection>

- 主要用于加载关联的集合对象

集合引用：(一对多关系映射)

- select
 - 查询用户的唯一标志。使用什么方法进行关联表的关联查询
 - 用于指定查询 account 列表的 sql 语句，所以填写的是该 sql 映射的 id
- column
 - 用户根据id查询时，所需要的参数的值。给上面方法的传参
 - 用于指定 select 属性的 sql 语句的参数来源，上面的参数来自于 user 的 id 列，所以就写成 id 这一个字段名了
 - 是用于指定使用哪个字段的值作为条件查询
- property
 - 封装到类中的哪个属性 | 引用
- ofType
 - 封装成什么
 - 用于指定集合元素的数据类型

```

1  <!-- 定义User的结果集 -->
2  <resultMap id="userAccountMap" type="user">
3      <id property="id" column="id"></id>
4      <result property="username" column="username"></result>
5      <result property="address" column="address"></result>
6      <result property="sex" column="sex"></result>

```

```

7   <result property="birthday" column="birthday"></result>
8   
9   <collection property="accounts" ofType="com.learn.domain.Account"
10      select="com.learn.dao.AccountDao.findAccountById"
11      column="id">
9   <resultMap id="userAccountMap" type="user">
10     <id property="id" column="id"></id>
11     <result property="username" column="username"></result>
12     <result property="address" column="address"></result>
13     <result property="sex" column="sex"></result>
14     <result property="birthday" column="birthday"></result>
15     
16     <collection property="accounts" ofType="com.learn.domain.Account"
17       select="com.learn.dao.AccountDao.findAccountById"
18       column="id"><!-- account别名 --&gt;
19     &lt;/collection&gt;
20   &lt;/resultMap&gt;
21
22   &lt;select id="findAll" resultMap="userAccountMap"&gt;
23     select * from user
24   &lt;/select&gt;
25
26   &lt;select id="findById" parameterType="int"
27     resultType="com.learn.domain.User"&gt;
28     select * from user
29     where id = #{suibianxie}
30   &lt;/select&gt;
&lt;/mapper&gt;
</pre>

```

注解方式

- 普通对象引用：一对—
- 集合对象引用：一对多

一对多:@One

```
1 public interface AccountDao {  
2  
3     /**  
4      * 查询所有账户，并且获取每个账户所属的用户信息  
5      * @return  
6      */  
7     @Select("select * from account")  
8     @Results(  
9         id = "accountMap",  
10        value = {  
11            @Result(id = true, column = "id", property = "id"),  
12            @Result(column = "uid", property = "uid"),  
13            @Result(column = "money", property = "money"),  
14            @Result(property = "user", column = "uid",  
15            one =  
16            @One(select="com.learn.dao.UserDao.findById", fetchType= FetchType.EAGER)  
17            ),  
18        })  
19     List<Account> findAll();  
20  
21     /**  
22      * 根据用户id查询账户信息  
23      * @param userId  
24      * @return  
25      */  
26     @Select("select * from account where uid = #{userId}")  
27     List<Account> findAccountByUid(Integer userId);  
28 }
```

一对多:@Many

```
1 public interface UserDao {  
2  
3     /**  
4      * 查询所有用户  
5      * @return  
6      */  
7     @Select(value = "select * from user")  
8     @Results( id = "userMap",  
9             value = {  
10                @Result(id =true, column = "id", property = "id"),  
11                @Result(column = "username", property = "username"),  
12                @Result(column = "address", property = "address"),  
13                @Result(column = "sex", property = "sex"),  
14                @Result(column = "birthday", property = "birthday"),  
15                @Result(property = "accounts", column = "id",  
16                many = @Many(select =  
17                    "com.learn.dao.AccountDao.findAccountByUid",  
18                    fetchType = FetchType.LAZY)  
19            ),  
20        })  
21 }
```

```
19         }
20     )
21     List<User> findAll();
22
23 /**
24  * 根据id查询用户
25  * @param userId
26  * @return
27  */
28 @Select(value = "select * from user where id=#{id}")
29 @ResultMap(value = {"userMap"})
30 User findById(Integer userId);
31
32 /**
33  * 根据用户名模糊查询
34  * @param username
35  * @return
36  */
37 @Select("select * from user where username like #{username}")
38 //  @Select("select * from user where username like '%${value}%' ")
39 @ResultMap(value = {"userMap"})
40 List<User> findUserByName(String username);
41
42 }
```

缓存

什么？

- 存在于内存中的临时数据。

用处：

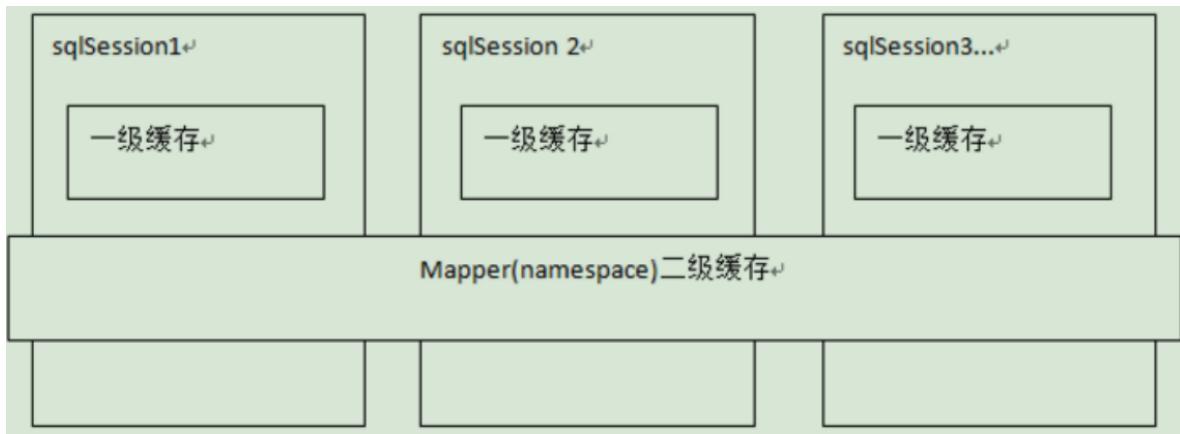
- 减少和数据库的交互次数，提高执行效率

适用：

- 经常查询 并且 不经常改变
- 数据的 正确与否 对最终结果 影响不大

像大多数的持久化框架一样，Mybatis 也提供了缓存策略，通过缓存策略来减少数据库的查询次数，从而提高性能。

- 一级缓存：默认开启使用。啥也不用管
- 二级缓存：需要配置开启。



一级缓存

- 一级缓存是 SqlSession 级别的缓存
 - 当我们执行查询之后，查询的结果会同时存入到SqlSession为我们提供的一块区域中。
 - 该区域的结构是一个Map
- 当调用 SqlSession 的修改(update)、添加、删除、commit flush 或 close时，就会清空一级缓存
 - SqlSession对象消失，缓存消失
 - close
 - clearCache 清空缓存
 - 目的为了让缓存中存储的是最新的信息，避免脏读。
- 默认就有，无需配置，什么都不需要做

xml方式

xml

```

1 <select id="findAll" resultType="com.learn.domain.User">
2   select * from user
3 </select>
4
5 <update id="updateUser" parameterType="user">
6   update user set username=#{username}, address=#{address}
7   where id=#{id}
8 </update>
```

测试类

```

1 //两次查询 一次SQL执行
2 //因为有缓存
3 @Test
4 public void testFirstLevelCache() {
5   User user1 = userDao.findById(41);
6   System.out.println(user1);
```

```
7     User user2 = userDao.findById(41);
8     System.out.println(user2);
9
10    System.out.println(user1 == user2); //true
11 }
12
13 //关闭再开启session
14 //会执行两次SQL，不是从缓存中获取
15 @Test
16 public void testFirstLevelCache() {
17     User user1 = userDao.findById(41);
18     System.out.println(user1);
19
20     //关闭 释放缓存
21     session.close();
22     //再次开启
23     session = factory.openSession();
24     userDao = session.getMapper(UserDao.class);
25     //此方法可以清空缓存
26     //session.clearCache();
27
28     User user2 = userDao.findById(41);
29     System.out.println(user2);
30
31     System.out.println(user1 == user2); //false
32 }
33
34
35 @Test
36 public void testClearCache() {
37     User user1 = userDao.findById(41);
38     System.out.println(user1);
39
40     //更新用户信息
41     user1.setUsername("update user clear cache");
42     user1.setAddress("xxxx市");
43     userDao.updateUser(user1);
44
45     User user2 = userDao.findById(41);
46     System.out.println(user2);
47
48     System.out.println(user1 == user2); //false
49 }
```

注解方式

接口(注解)

```

1 /**
2  * 查询所有用户
3  * @return
4 */
5 @Select(value = "select * from user")
6 List<User> findAll();
7
8 /**
9  * 更新用户
10 * @param user
11 */
12 @Update(value = "update user set username=#{username}, sex=#{sex},
13         birthday=#{birthday}, address=#{address} where id=#{id} ")
13 void updateUser(User user);

```

测试类

```

1 //两次查询 一次SQL执行
2 //因为有缓存
3 @Test
4 public void testFirstLevelCache() {
5     User user1 = userDao.findById(41);
6     System.out.println(user1);
7
8     User user2 = userDao.findById(41);
9     System.out.println(user2);
10
11     System.out.println(user1 == user2); //true
12 }
13
14 //关闭再开启session
15 //会执行两次SQL，不是从缓存中获取
16 @Test
17 public void testFirstLevelCache() {
18     User user1 = userDao.findById(41);
19     System.out.println(user1);
20
21     //关闭 释放缓存
22     session.close();
23     //再次开启
24     session = factory.openSession();
25     userDao = session.getMapper(UserDao.class);
26     //此方法可以清空缓存
27     //session.clearCache();
28
29     User user2 = userDao.findById(41);
30     System.out.println(user2);
31
32     System.out.println(user1 == user2); //false
33 }
34
35 @Test
36 public void testClearCache() {
37     User user1 = userDao.findById(41);
38     System.out.println(user1);

```

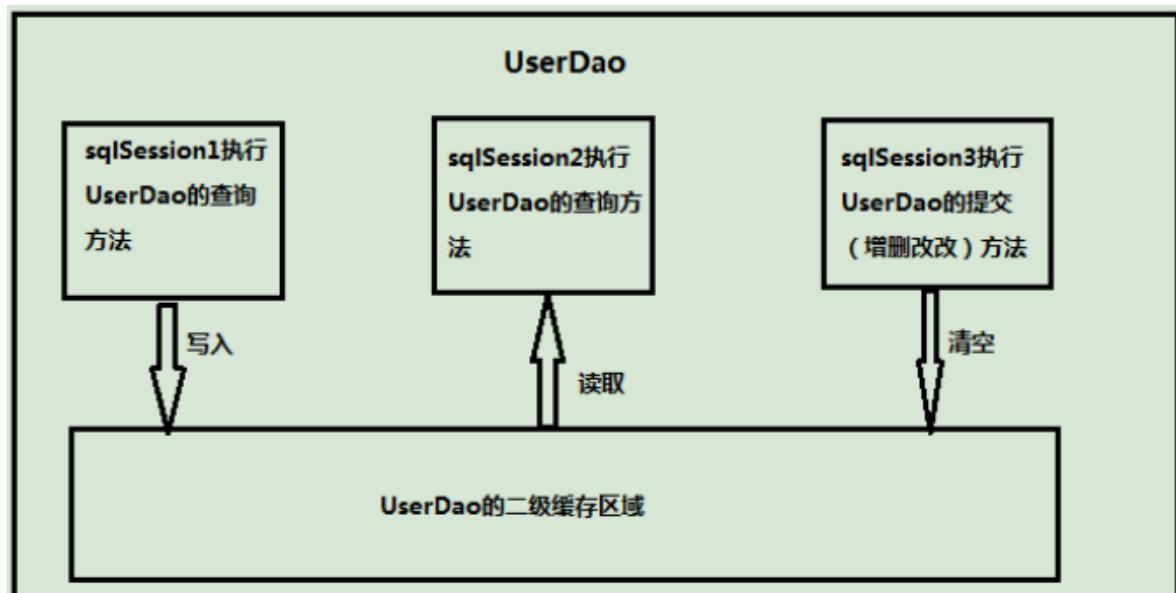
```

39
40     //更新用户信息
41     user1.setUsername("update user clear cache");
42     user1.setAddress("xxxxx市");
43     userDao.updateUser(user1);
44
45     User user2 = userDao.findById(41);
46     System.out.println(user2);
47
48     System.out.println(user1 == user2); //false
49 }

```

二级缓存

- MyBatis中 SqlSessionFactory对象 的缓存。 ? dao对应factory ?
- 二级缓存是 mapper 映射级别的缓存，多个 SqlSession 去操作同一个 Mapper 映射的 sql 语句，多个SqlSession 可以共用二级缓存，二级缓存是跨 SqlSession 的。
- 同一factory创建的 SqlSession 共享 该缓存。
- 二级缓存中存的是数据，不是对象，所以不会获取到同一对象了。



使用步骤

- 使用二级缓存，需要配置
 - 主配置文件，设置支持二级缓存
 - 映射文件，设置支持二级缓存
 - xml
 - 注解
 - 让当前的操作支持二级缓存，select...标签中配置
 - xml
 - 注解

主配置文件

- 主配置文件，设置支持二级缓存
 - 这里value默认就是true，不配置，全局二级缓存也是默认开启的

```
1 <settings>
2   <!-- 开启二级缓存的支持 -->
3   <setting name="cacheEnabled" value="true"/>
4 </settings>
5 因为 cacheEnabled 的取值默认就为 true，所以这一步可以省略不配置。
6 为 true 代表开启二级缓存；
7 为 false 代表不开启二级缓存。
```

完整版

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE configuration
3   PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4   "http://mybatis.org/dtd/mybatis-3-config.dtd">
5
6 <configuration>
7   <properties resource="jdbcConfig.properties"></properties>
8
9   <settings>
10    <setting name="cacheEnabled" value="true"/>
11 </settings>
12
13  <!-- 使用typeAliases 配置别名。它只能配置domain中类的别名 -->
14  <typeAliases>
15    <package name="com.learn.domain"></package>
16  </typeAliases>
17
18  <!-- 配置环境 -->
19  <environments default="mysql">
20    <environment id="mysql">
21      <!-- 配置事务 -->
22      <transactionManager type="JDBC"></transactionManager>
23      <!-- 配置连接池 -->
24      <dataSource type="POOLED">
25        <!-- 配置连接数据库的4个基本信息 -->
```

```

26      <property name="driver" value="${jdbc.driver}"/>
27      <property name="url" value="${jdbc.url}"/>
28      <property name="username" value="${jdbc.username}"/>
29      <property name="password" value="${jdbc.password}"/>
30    </dataSource>
31  </environment>
32</environments>
33  <!-- 指定映射配置文件的位置 -->
34  <mappers>
35    <package name="com.learn.dao"></package>
36  </mappers>
37</configuration>

```

xml方式

映射配置文件

- 开启支持二级缓存，相关Mapper映射文件层面
 - 标签：`<cache></cache>`
- 当前操作支持，statement层面。看来statement是操作相关的啊(配置 statement 上面的 useCache)
 - 属性：`useCache='true'`
 - 将 UserDao.xml 映射文件中的 `<select>` 标签中设置 `useCache="true"` 代表当前这个 statement 要使用二级缓存，如果不使用二级缓存可以设置为 false
 - 注意：针对每次查询都需要最新的数据 sql，要设置成 `useCache=false`，禁用二级缓存。

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE mapper
3   PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4   "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5
6 <mapper namespace="com.learn.dao.UserDao">
7   <!-- 开启二级缓存支持 -->
8   <cache></cache>
9
10  <select id="findAll" resultType="com.learn.domain.User" >
11    select * from user
12  </select>
13
14  <select id="findById" parameterType="int"
15    resultType="com.learn.domain.User" useCache="true">
16    select * from user
17    where id = #{suibianxie}
18  </select>
19
20  <update id="updateUser" parameterType="user">
21    update user set username=#{username}, address=#{address}
22    where id=#{id}
23  </update>
24</mapper>

```

测试类

- 两次查询
- 一次SQL执行
- 数据在factory缓存|二级缓存里
- 是数据，不是对象 false

```
1  @Test
2  public void testFirstLevelCache() {
3      SqlSession session1 = factory.openSession();
4      UserDao userDao1 = session1.getMapper(UserDao.class);
5      User user1 = userDao1.findById(41);
6      System.out.println(user1);
7
8      session1.close(); //一级缓存消失
9
10     SqlSession session2 = factory.openSession();
11     UserDao userDao2 = session2.getMapper(UserDao.class);
12     User user2 = userDao2.findById(41);
13     System.out.println(user2);
14
15     session2.close();
16
17     System.out.println(user1 == user2);
18 }
19
```

注解方式

- 接口上加注解
- 方法呢？？？，不需要，接口里方法全开启了
- 主配置文件，开启二级缓存。
 - 不配置其实也是默认开启的
- 接口上 @CacheNamespace(blocking = true)
 - 操作支持

自定义MyBatis框架

MyBatis 操作 流

- `InputStream in = Resources.getResourceAsStream("SqlMapConfig.xml")`
 - 读取 配置文件
- `SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder()`
 - 创建工厂Builder
- `SqlSessionFactory factory = builder.build(in)`
 - 创建工厂

下面两操作 可提取到实现类里：

- new UserDaoImpl(factory)
 - 实现类 完成具体操作
 - session.selectList("com.learn.dao.UserDao.findAll") 等方法
- `SqlSession session = factory.openSession()`
 - 创建 SqlSession
 - session可 创建 指定接口的 代理类 , 相当于实现类
 - session可 直接 操作方法 进行数据操作
 - `UserDao dao = session.getMapper(UserDao.class)`
 - 创建Dao接口的 代理对象 , 功能相当于实现类

抽取类

常规Mybatis使用流程

```
1 public static void main(String[] args) throws Exception {  
2     //1.读取配置文件  
3     InputStream in = Resources.getResourceAsStream("SqlMapConfig.xml");  
4     //2.创建SqlSessionFactory工厂  
5     SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();  
6     SqlSessionFactory factory = builder.build(in);  
7     //3.使用工厂生产SqlSession对象  
8     SqlSession session = factory.openSession();  
9     //4.使用SqlSession创建Dao接口的代理对象  
10    UserDao dao = session.getMapper(UserDao.class);  
11    //5.使用代理对象执行方法  
12    List<User> users = dao.findAll();  
13    for (User user : users) {  
14        System.out.println(user);  
15    }  
16    //6.释放资源  
17    session.close();  
18    in.close();  
19}
```

Resources

- 读取 配置文件

```
InputStream in = Resources.getResourceAsStream("SqlMapConfig.xml")  
builder.build(in)
```

SqlSessionFactoryBuilder

根据 配置信息 创建工厂

获取：`SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder()`

- 创建 工厂Builder

- 根据 配置信息 创建工厂。
- 配置信息 通过XMLConfigBuilder来解析

使用：`sqlSessionFactory factory = builder.build(in)`

XMLConfigBuilder

解析配置文件，生成Configuration对象

- 连接信息
- 操作信息

SqlSessionFactory

创建 SqlSession

获取：`sqlSessionFactory factory = builder.build(in)`

- 创建 SqlSession

使用：`sqlSession session = factory.openSession()`

SqlSession

获取 执行操作 的 代理对象

获取：`sqlSession session = factory.openSession()`

使用：

- 获得代理对象，用于执行操作。`getMapper`
- 直接调用方法，执行操作。`selectList`

MapperProxy

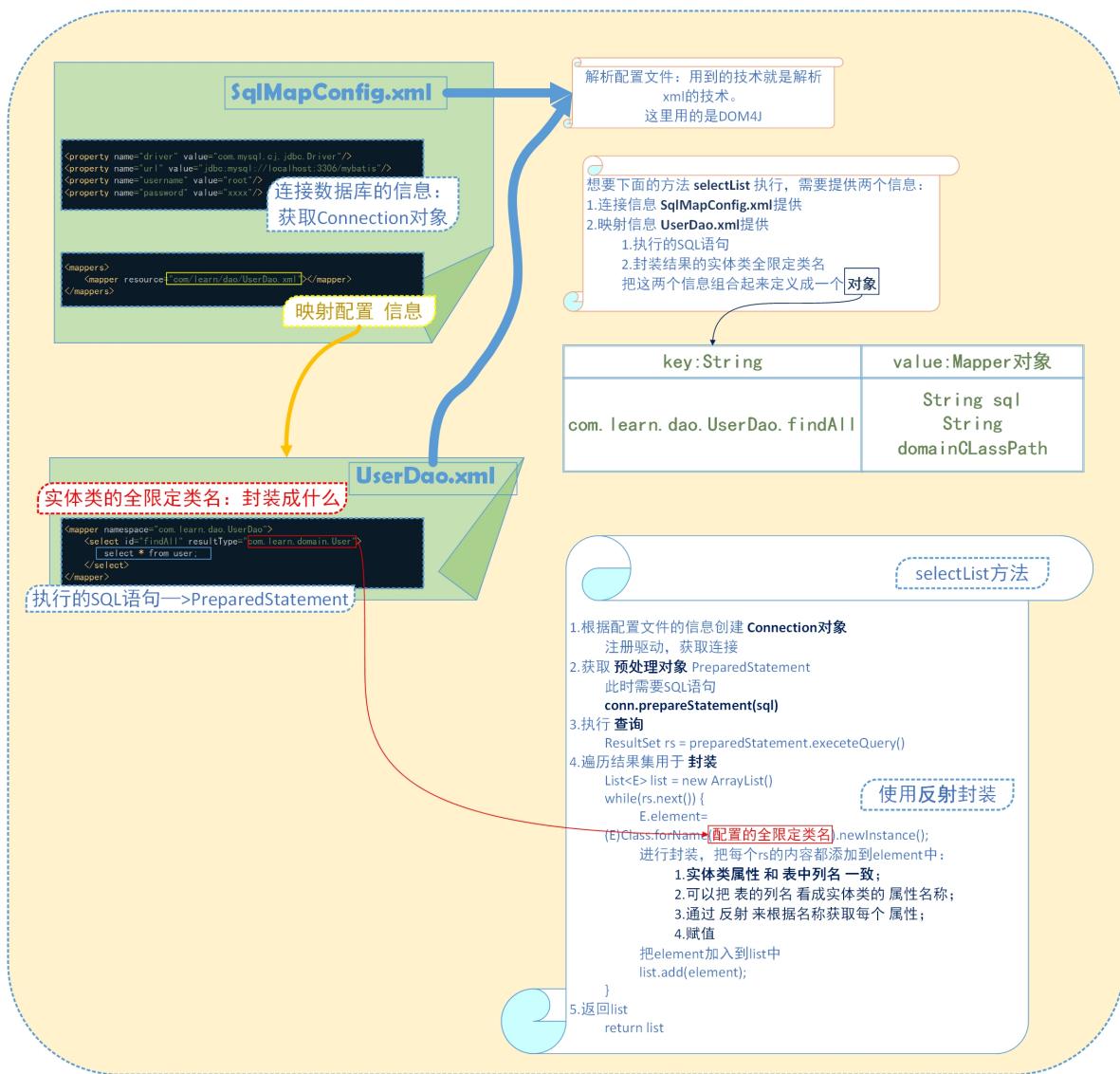
代理对象的增强方法代码

Executor

- 执行SQL
- 封装结果集

案例分析

功能简单分析



辅助类&文件图



主要类间关系图

```
Resources.getResourceAsStream("SqlMapConfig.xml")
```

根据主配置文件信息 创建工厂

XMLConfigBuilder
连接信息 操作信息 mapper cfg

SqlSessionFactoryBuilder

```
SqlSessionFactory build(InputStream config)
```

```
new DefaultSqlSessionFactory(cfg)
```

连接信息 操作信息 mapper cfg

SqlSessionFactory

```
SqlSession openSession()
```

创建SqlSession对象

DefaultSqlSessionFactory

Configuration cfg
DefaultSqlSessionFactory(Configuration cfg)
SqlSession openSession()

```
new DefaultSqlSession(cfg)
```

连接信息 操作信息 mapper cfg

SqlSession

<T> T getMapper(Class<T> daoInterfaceClass)
void close()

DataSourceUtil

生成接口代理对象
定义通用CRUD方法

DefaultSqlSession

Configuration cfg
Connection conn
DefaultSqlSession(Configuration cfg)
<T> T getMapper(Class<T> daoInterfaceClass)
void close()

```
(T) Proxy.newProxyInstance(daoInterfaceClass.getClassLoader(),  
    new Class[]{daoInterfaceClass},  
    new MapperProxy(cfg.getMappers(), conn))  
)
```

conn

操作信息 mapper

代理对象

MapperProxy

Map<String, Mapper> mappers
Connection conn
MapperProxy(mappers, conn)
Object invoke(Object proxy, Method method, Object[] args)

```
new Executor().selectList(mapper, conn)
```

conn

操作信息 mapper

执行SQL语句

封装结果集

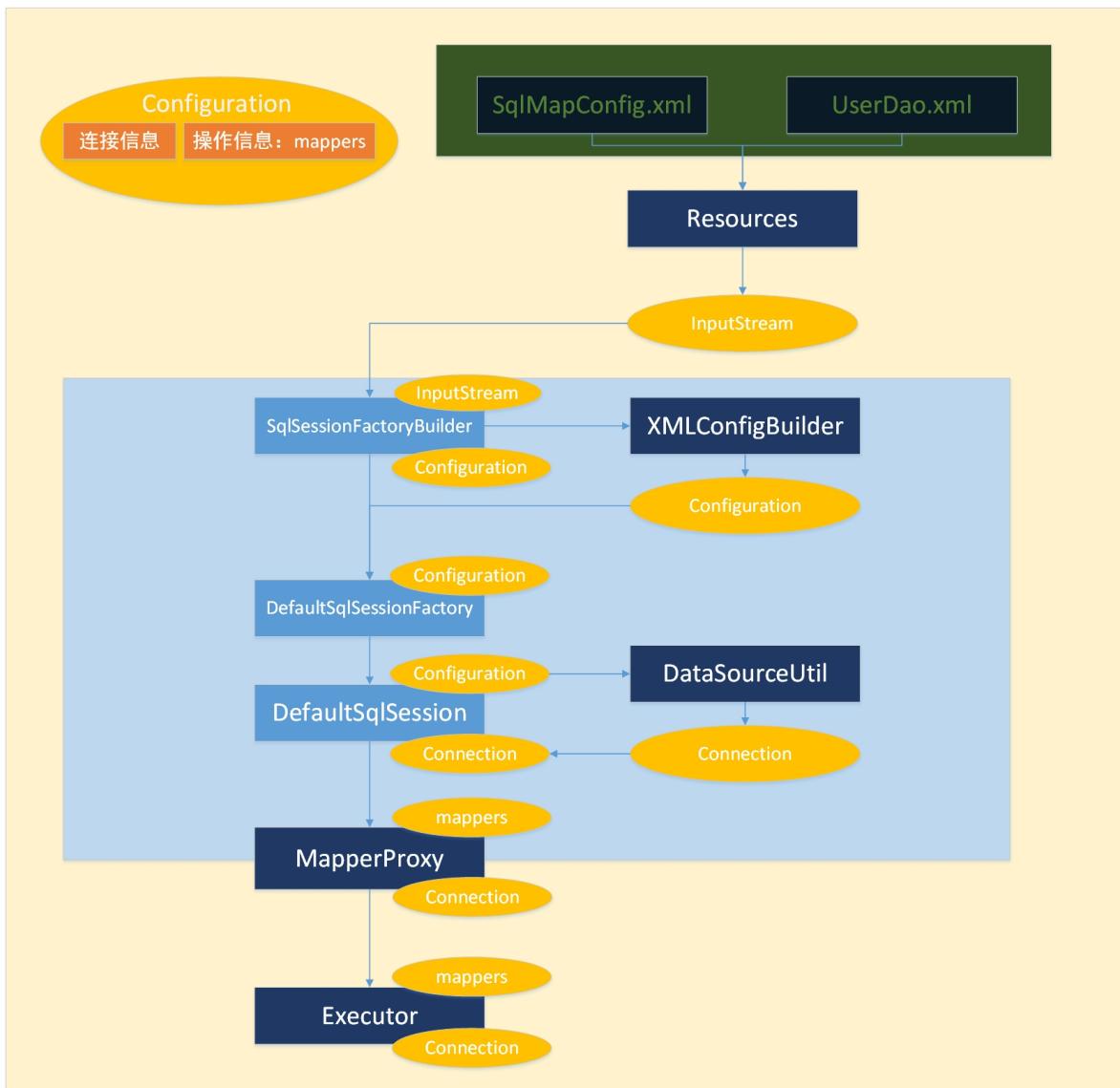
Executor

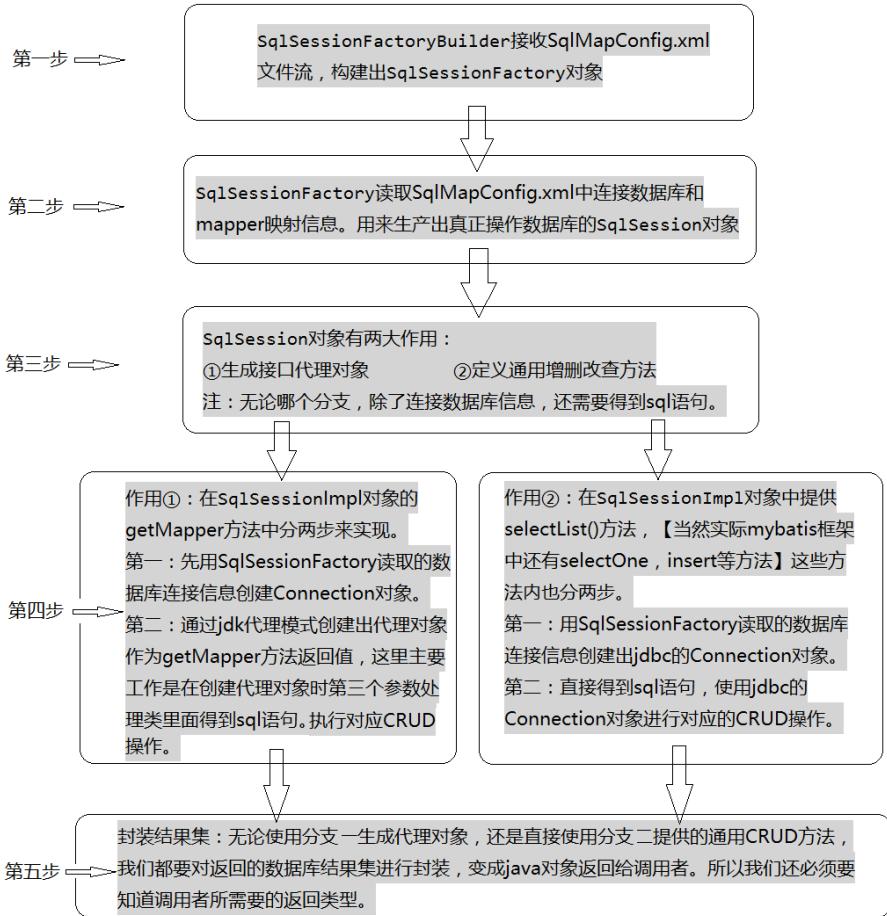
<E> List<E> selectList(Mapper mapper, Connection conn)
void release(PreparedStatement pstm, ResultSet rs)

```
pstm = conn.prepareStatement(queryString)
```

```
rs = pstm.executeQuery()
```

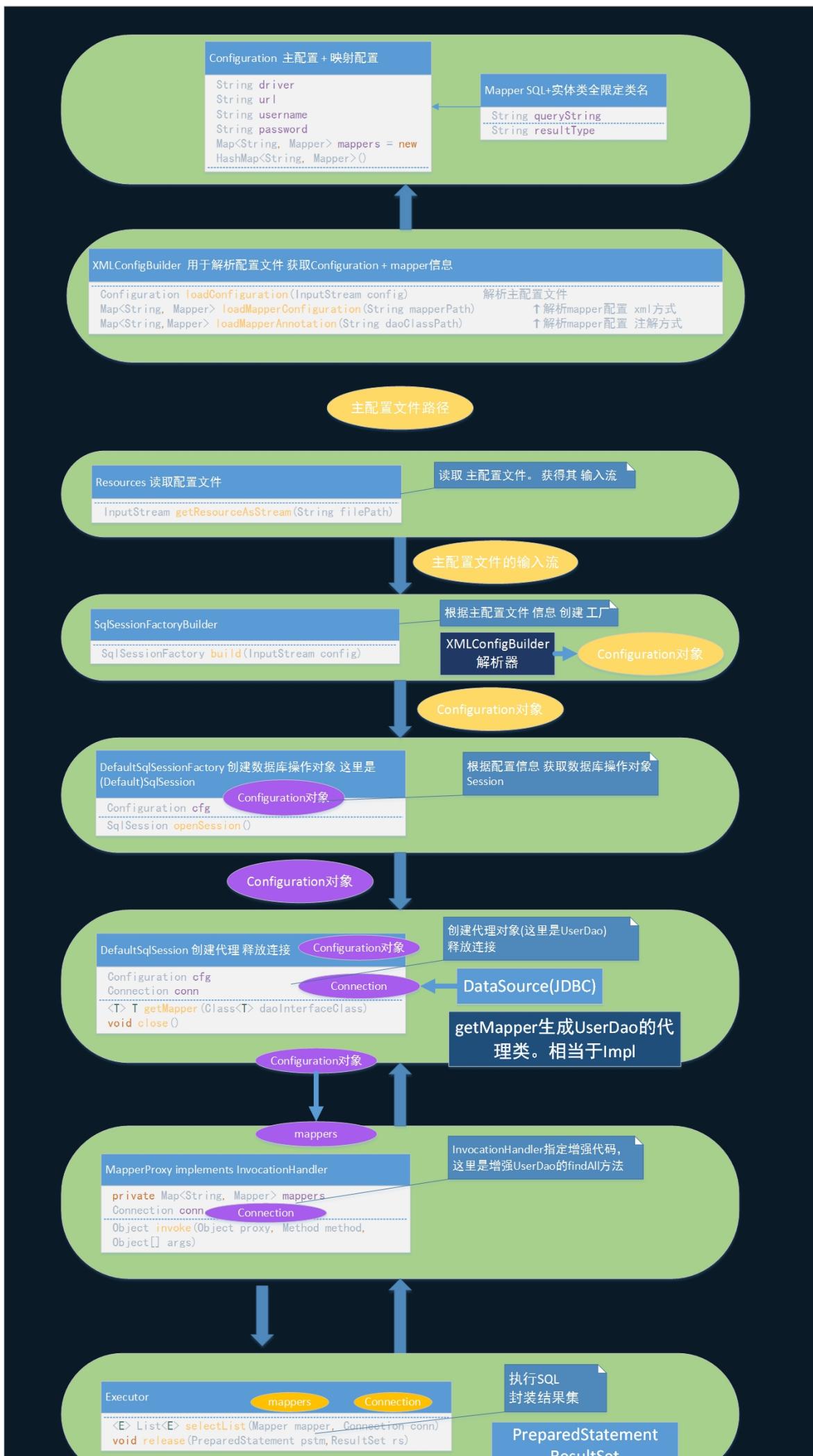
执行流程图





总结：通过以上流程我们不难看出，无论是让mybatis帮我们创建代理对象还是直接使用mybatis提供的CRUD方法，其本质都是得到jdbc的Connection对象，执行对应sql语句，最终封装结果集。只是注解和xml配置文件两种开发模式在传递sql语句和返回值类型的方式上有所差异而已。具体如下图所示：





创建代理对象分析

- 使用SqlSession创建Dao接口的代理对象：

```
1 UserDao dao = session.getMapper(UserDao.class);
```

- 根据dao接口的字节码创建dao的代理对象

```
1 Public <T> getMapper(Class<T> daoInterfaceClass) {
2     Proxy.newProxyInstance(类加载器, 代理对象要实现的接口字节码数组, 如何代理);
3 }
```

- 类加载器：
 - 它使用的和 被代理对象 是 相同的 类加载器
- 代理对象要实现的接口：
 - 和 被代理对象 实现 相同的接口
- 如何代理：
 - 它就是增强的方法，我们需要自己来提供
 - 此处是一个InvocationHandler的接口，我们需要写一个该接口的实现类
 - 在实现类中调用selectList方法

```
1 IProducer proxyProducer = (IProducer)Proxy.newProxyInstance(
2     producer.getClass().getClassLoader(),
3     producer.getClass().getInterfaces(),
4     new InvocationHandler() {
5         public Object invoke(Object proxy,
6                             Method method,
7                             Object[] args) throws Throwable {
8             //提供增强代码
9             Object returnValue = null;
10            //1.获取方法执行的参数
11            Float money = (Float)args[0];
12            //2.判断当前方法是不是销售
13            if ("saleProduct".equals(method.getName())) {
14                returnValue = method.invoke(producer, money * 0.8f);
15            }
16            return returnValue;
17        }
18    });
19
20 //这里的代理是全拦截，其实是有部分问题的
21 //原理已经大致清楚了 源码就交给msb吧
22 public <T> T getMapper(Class<T> daoInterfaceClass) {
23     /*
24         * 代理谁 就用谁的类加载器
25         *      daoInterfaceClass.getClassLoader()
26         * 代理谁 就要和谁实现相同的接口 这里本身就是接口
27         *      new Class[]{daoInterfaceClass}
28         * 如何代理？
29     */
30 }
```

```

29     *      自己的代理方式 MapperProxy implements InvocationHandler 传参处
理
30     */
31     return (T) Proxy.newProxyInstance (daoInterfaceClass.getClassLoader(),
32                                     new Class[]{daoInterfaceClass},
33                                     new MapperProxy(cfg.getMappers(),
34                                     conn)
35 );

```

环境搭建

和入门案例一样，但是这里pom不导入mybatis依赖，由我们手动自行实现。

- 删除mybatis
- 加入dom4j解析xml
- jaxen 用于使用XPATH语法 来查询标签

pom

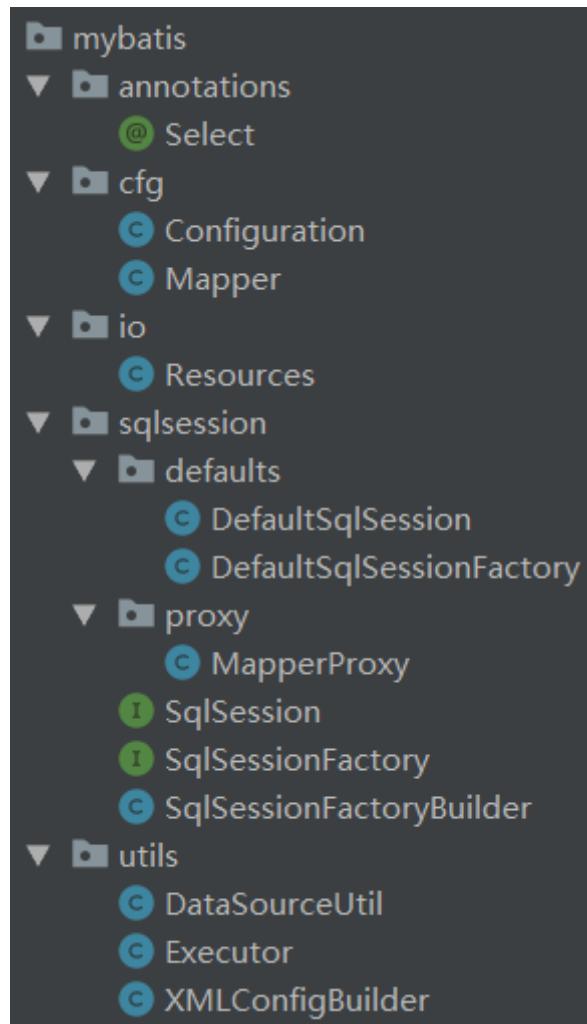
```

1 <dependencies>
2   <dependency>
3     <groupId>mysql</groupId>
4     <artifactId>mysql-connector-java</artifactId>
5     <version>8.0.15</version>
6   </dependency>
7   <dependency>
8     <groupId>log4j</groupId>
9     <artifactId>log4j</artifactId>
10    <version>1.2.12</version>
11  </dependency>
12  <dependency>
13    <groupId>junit</groupId>
14    <artifactId>junit</artifactId>
15    <version>4.12</version>
16  </dependency>
17
18  <!-- dom4j -->
19  <dependency>
20    <groupId>dom4j</groupId>
21    <artifactId>dom4j</artifactId>
22    <version>1.6.1</version>
23  </dependency>
24  <!-- xpath -->
25  <dependency>
26    <groupId>jaxen</groupId>
27    <artifactId>jaxen</artifactId>
28    <version>1.1.6</version>
29  </dependency>
30 </dependencies>

```

实现

整体结构



mybatis

- annotation
 - @Select
 - String value() 存SQL的
- cfg
 - Configuration
 - 连接信息 × 4
 - Map<String, Mapper> mappers
 - Mapper
 - queryString SQL
 - resultType 实体类的全限定类名
- io
 - Resources
 - 通过类加载器 获取 配置文件的 输入流





annotation

Select

- SQL

```
1  /**
2  * 查询的注解
3  */
4 @Retention(RetentionPolicy.RUNTIME)
5 @Target(ElementType.METHOD)
6 public @interface Select {
7
8     /**
9      * 配置SQL语句的
10     * @return
11     */
12     String value();
13 }
```

cfg

Configuration

- 数据库连接信息
- mappers 操作信息
 - key : 类名+方法名
 - value : mapper
 - SQL
 - 实体类的全限定类名

```
1 /**
2  * 自定义mybatis的配置类
3  */
4 public class Configuration {
5     private String driver;
6     private String url;
7     private String username;
8     private String password;
9
10    private Map<String, Mapper> mappers = new HashMap<String, Mapper>();
11
12    public String getDriver() {
13        return driver;
14    }
15
16    public void setDriver(String driver) {
17        this.driver = driver;
18    }
19
20    public String getUrl() {
21        return url;
22    }
23
24    public void setUrl(String url) {
25        this.url = url;
26    }
27
28    public String getUsername() {
```

```

29         return username;
30     }
31
32     public void setUsername(String username) {
33         this.username = username;
34     }
35
36     public String getPassword() {
37         return password;
38     }
39
40     public void setPassword(String password) {
41         this.password = password;
42     }
43
44     public Map<String, Mapper> getMappers() {
45         return mappers;
46     }
47
48     public void setMappers(Map<String, Mapper> mappers) {
49         this.mappers.putAll(mappers); //此处需要使用追加的方式
50     }
51 }
```

Mapper

- SQL语句
- 实体类的全限定类名

```

1 /**
2 * 用于封装执行的SQL语句和结果类型的全限定类名
3 */
4 public class Mapper {
5
6     private String queryString;//SQL
7     private String resultType;//实体类的全限定类名
8
9     public String getQueryString() {
10         return queryString;
11     }
12
13     public void setQueryString(String queryString) {
14         this.queryString = queryString;
15     }
16
17     public String getResultType() {
18         return resultType;
19     }
20
21     public void setResultType(String resultType) {
22         this.resultType = resultType;
23     }
24 }
```

io

Resources

```
1 /**
2  * 使用类加载器读取配置文件的类
3 */
4 public class Resources {
5     /**
6      * 根据传入的参数，获取一个字节输入流
7      * @param filePath
8      * @return
9      */
10    public static InputStream getResourceAsStream(String filePath) {
11        //1.拿到当前类的字节码
12        //2.拿到当前类的字节码的类加载器
13        //3.同过类加载器读取配置文件
14        return
15        Resources.class.getClassLoader().getResourceAsStream(filePath);
16    }
17}
```

sqlsession

SqlSession

- 创建代理对象

```
1 /**
2  * 自定义MyBatis中和数据库交互的核心类
3  * 它可以创建dao接口的代理对象
4 */
5 public interface SqlSession {
6
7     /**
8      * 根据参数创建一个代理对象
9      * @param daoInterfaceClass dao的接口字节码
10     * @param <T>
11     * @return
12     */
13     <T> T getMapper(Class<T> daoInterfaceClass);
14
15     /**
16      * 释放资源
17      */
18     void close();
19 }
```

SqlSessionFactory

- 打开SqlSession

```
1 public interface SqlSessionFactory {  
2  
3     /**  
4      * 用于打开一个新的SqlSession对象  
5      * @return  
6      */  
7     SqlSession openSession();  
8  
9 }
```

SqlSessionFactoryBuilder

- 创建工厂对象

```
1  /**  
2   * 用于创建一个SqlSessionFactory对象  
3   */  
4  public class SqlSessionFactoryBuilder {  
5  
6      /**  
7       * 根据参数的字节码输入流来构建一个SqlSessionFactory工厂  
8       * @param config  
9       * @return  
10      */  
11     public SqlSessionFactory build(InputStream config) {  
12         Configuration cfg = XMLConfigBuilder.loadConfiguration(config);  
13         return new DefaultSqlSessionFactory(cfg);  
14     }  
15  
16 }
```

defaults

DefaultSqlSession

- 创建Dao的代理对象

```
1  /**  
2   * SqlSession接口的实现类  
3   */  
4  public class DefaultSqlSession implements SqlSession {  
5  
6      private Configuration cfg;  
7  
8      private Connection conn;  
9  
10     public DefaultSqlSession(Configuration cfg) {  
11         this.cfg = cfg;  
12         this.conn = DataSourceUtil.getConnection(cfg);  
13     }  
14  
15     /**  
16      * 用于创建代理对象
```

```

17     * @param daoInterfaceClass dao的接口字节码
18     * @param <T>
19     * @return
20     */
21     public <T> T getMapper(Class<T> daoInterfaceClass) {
22         /*
23             * 代理谁 就用谁的类加载器
24             *      daoInterfaceClass.getClassLoader()
25             * 代理谁 就要和谁实现相同的接口 这里本身就是接口
26             *      new Class[]{daoInterfaceClass}
27             * 如何代理?
28             *      自己的代理方式 MapperProxy implements InvocationHandler 传参处
理
29             */
30         return (T) Proxy.newProxyInstance(
31             daoInterfaceClass.getClassLoader(),
32                 new Class[]{daoInterfaceClass},
33                 new MapperProxy(cfg.getMappers(), conn)
34         );
35     }
36
37 /**
38 * 用于释放资源
39 */
40     public void close() {
41         if (conn != null) {
42             try {
43                 conn.close();
44             } catch (SQLException e) {
45                 e.printStackTrace();
46             }
47         }
48     }
49 }
```

DefaultSqlSessionFactory

- DefaultSqlSession

```

1 /**
2 * SqlSessionFactory接口的实现类
3 */
4 public class DefaultSqlSessionFactory implements SqlSessionFactory {
5
6     private Configuration cfg;
7
8     public DefaultSqlSessionFactory(Configuration cfg) {
9         this.cfg = cfg;
10    }
11
12 /**
13 * 用于创建一个新的操作数据库对象
14 * @return
15 */
```

```
16     public SqlSession openSession() {  
17         return new DefaultSqlSession(cfg);  
18     }  
19 }
```

proxy

MapperProxy

- 增强代码

```
1  public class MapperProxy implements InvocationHandler {  
2  
3     //map的key是全限定类名+方法名  
4     private Map<String, Mapper> mappers;  
5  
6     private Connection conn;  
7  
8     public MapperProxy(Map<String, Mapper> mappers, Connection conn) {  
9         this.mappers = mappers;  
10        this.conn = conn;  
11    }  
12  
13    /**  
14     * 用于对方法进行增强的，我们的增强就是调用selectList方法  
15     * @param proxy  
16     * @param method  
17     * @param args  
18     * @return  
19     * @throws Throwable  
20     */  
21    public Object invoke(Object proxy, Method method, Object[] args) throws  
Throwable {  
22        //1.获取方法名  
23        String methodName = method.getName();  
24        //2.获取方法所在类的名称  
25        String className = method.getDeclaringClass().getName();  
26        //3.组合key  
27        String key = className + "." + methodName;  
28        //4.获取mappers中的Mapper对象  
29        Mapper mapper = mappers.get(key);  
30        //5.判断是否有mapper  
31        if (mapper == null) {  
32            throw new IllegalArgumentException("传入的参数有误");  
33        }  
34        //6.调用工具类执行查询所有  
35  
36        return new Executor().selectList(mapper, conn);  
37    }  
38 }
```

utils

DataSourceUtil

```

1 /**
2  * 用于创建数据源的工具类
3 */
4 public class DataSourceUtil {
5
6     /**
7      * 用于获取一个连接
8      * @param cfg
9      * @return
10     */
11    public static Connection getConnection(Configuration cfg) {
12        try {
13            Class.forName(cfg.getDriver());
14            return DriverManager.getConnection(cfg.getUrl(),
15                cfg.getUsername(), cfg.getPassword());
16        } catch (Exception e) {
17            throw new RuntimeException(e);
18        }
19    }
20}

```

Executor

- 执行SQL语句，封装结果集
- mapper执行信息
- conn连接
- PreparedStatement ResultSet

```

1 /**
2  * 负责执行SQL语句，并且封装结果集
3 */
4 public class Executor {
5
6     public <E> List<E> selectList(Mapper mapper, Connection conn) {
7         PreparedStatement pstm = null;
8         ResultSet rs = null;
9         try {
10             //1.取出mapper中的数据
11             String queryString = mapper.getQueryString(); //select * from
12             user
13             String resultType =
14             mapper.getResultType(); //com.learn.domain.User
15             Class domainClass = Class.forName(resultType);
16             //2.获取PreparedStatement对象
17             pstm = conn.prepareStatement(queryString);
18             //3.执行SQL语句，获取结果集
19             rs = pstm.executeQuery();
20             //4.封装结果集
21             List<E> list = new ArrayList<E>(); //定义返回值
22             while(rs.next()) {
23                 //实例化要封装的实体类对象
24                 E obj = (E)domainClass.newInstance();

```

```

25             ResultSetMetaData rsmd = rs.getMetaData();
26             //取出总列数
27             int columnCount = rsmd.getColumnCount();
28             //遍历总列数
29             for (int i = 1; i <= columnCount; i++) {
30                 //获取每列的名称，列名的序号是从1开始的
31                 String columnName = rsmd.getColumnName(i);
32                 //根据得到列名，获取每列的值
33                 Object columnValue = rs.getObject(columnName);
34                 //给obj赋值：使用Java内省机制（借助PropertyDescriptor实现属性的封装）
35                 PropertyDescriptor pd = new
36                 PropertyDescriptor(columnName, domainClass); //要求：实体类的属性和数据库表的列名保持一种
37                     //获取它的写入方法
38                     Method writeMethod = pd.getWriteMethod();
39                     //把获取的列的值，给对象赋值
40                     writeMethod.invoke(obj, columnValue);
41                     }
42                     //把赋好值的对象加入到集合中
43                     list.add(obj);
44                     }
45             return list;
46         } catch (Exception e) {
47             throw new RuntimeException(e);
48         } finally {
49             release(pstm, rs);
50         }
51     }

52     private void release(PreparedStatement pstm, ResultSet rs){
53         if(rs != null){
54             try {
55                 rs.close();
56             }catch(Exception e){
57                 e.printStackTrace();
58             }
59         }
60
61         if(pstm != null){
62             try {
63                 pstm.close();
64             }catch(Exception e){
65                 e.printStackTrace();
66             }
67         }
68     }
69 }
```

XMLConfigBuilder

框架的实现原理，都体现出来了

- 解析配置文件
 - 数据库连接信息

- 操作信息

```
1  /**
2  * 用于解析配置文件
3  */
4  public class XMLConfigBuilder {
5
6      /**
7      * 解析主配置文件，把里面的内容填充到DefaultSqlSession所需要的地方
8      * 使用的技术：
9      *      dom4j+xpath
10     */
11    public static Configuration loadConfiguration(InputStream config){
12        try{
13            //定义封装连接信息的配置对象（mybatis的配置对象）
14            Configuration cfg = new Configuration();
15
16            //1.获取SAXReader对象
17            SAXReader reader = new SAXReader();
18            //2.根据字节输入流获取Document对象
19            Document document = reader.read(config);
20            //3.获取根节点
21            Element root = document.getRootElement();
22            //4.使用xpath中选择指定节点的方式，获取所有property节点
23            List<Element> propertyElements =
root.selectNodes("//property");
24            //5.遍历节点
25            for(Element propertyElement : propertyElements){
26                //判断节点是连接数据库的哪部分信息
27                //取出name属性的值
28                String name = propertyElement.getAttributeValue("name");
29                if("driver".equals(name)){
30                    //表示驱动
31                    //获取property标签value属性的值
32                    String driver =
propertyElement.getAttributeValue("value");
33                    cfg.setDriver(driver);
34                }
35                if("url".equals(name)){
36                    //表示连接字符串
37                    //获取property标签value属性的值
38                    String url = propertyElement.getAttributeValue("value");
39                    cfg.setUrl(url);
34                }
35                if("username".equals(name)){
36                    //表示用户名
37                    //获取property标签value属性的值
38                    String username =
propertyElement.getAttributeValue("value");
39                    cfg.setUsername(username);
34                }
35                if("password".equals(name)){
36                    //表示密码
37                    //获取property标签value属性的值
38                    String password =
propertyElement.getAttributeValue("value");
39                    cfg.setPassword(password);
34            }
35        }
36    }
37}
```

```

52         }
53     }
54     //取出mappers中的所有mapper标签，判断他们使用了resource还是class属性
55     List<Element> mapperElements =
56     root.selectNodes("//mappers/mapper");
57     //遍历集合
58     for(Element mapperElement : mapperElements){
59         //判断mapperElement使用的是哪个属性
60         Attribute attribute = mapperElement.attribute("resource");
61         if(attribute != null){
62             System.out.println("使用的是XML");
63             //表示有resource属性，用的是XML
64             //取出属性的值
65             String mapperPath = attribute.getValue(); //获取属性的
66             值 "com/learn/dao/IUserDao.xml"
67             //把映射配置文件的内容获取出来，封装成一个map
68             Map<String,Mapper> mappers =
69             loadMapperConfiguration(mapperPath);
70             //给configuration中的mappers赋值
71             cfg.setMappers(mappers);
72         }else{
73             System.out.println("使用的是注解");
74             //表示没有resource属性，用的是注解
75             //获取class属性的值
76             String daoClassPath =
77             mapperElement.attributeValue("class");
78             //根据daoClassPath获取封装的必要信息
79             Map<String,Mapper> mappers =
80             loadMapperAnnotation(daoClassPath);
81             //给configuration中的mappers赋值
82             cfg.setMappers(mappers);
83         }
84     }
85     //返回Configuration
86     return cfg;
87 }catch(Exception e){
88     throw new RuntimeException(e);
89 }finally{
90     try {
91         config.close();
92     }catch(Exception e){
93         e.printStackTrace();
94     }
95 }
96 /**
97 * XML 配置方式
98 * 根据传入的参数，解析XML，并且封装到Map中
99 * @param mapperPath 映射配置文件的位置
100 * @return map中包含了获取的唯一标识（key是由dao的全限定类名和方法名组成）
101 *         以及执行所需的必要信息（value是一个Mapper对象，里面存放的是执行的
102 *         SQL语句和要封装的实体类全限定类名）
103 */
104 private static Map<String, Mapper> loadMapperConfiguration(String
105 mapperPath) throws IOException {
106     InputStream in = null;

```

```
103     try{
104         //定义返回值对象
105         Map<String,Mapper> mappers = new HashMap<String,Mapper>();
106         //1.根据路径获取字节输入流
107         in = Resources.getResourceAsStream(mapperPath);
108         //2.根据字节输入流获取Document对象
109         SAXReader reader = new SAXReader();
110         Document document = reader.read(in);
111         //3.获取根节点
112         Element root = document.getRootElement();
113         //4.获取根节点的namespace属性取值
114         String namespace = root.getAttributeValue("namespace");//是组成map
  
中key的部分
115         //5.获取所有的select节点
116         List<Element> selectElements = root.selectNodes("//select");
117         //6.遍历select节点集合
118         for(Element selectElement : selectElements){
119             //取出id属性的值      组成map中key的部分
120             String id = selectElement.getAttributeValue("id");
121             //取出resultType属性的值  组成map中value的部分
122             String resultType =
123                 selectElement.getAttributeValue("resultType");
124             //取出文本内容          组成map中value的部分
125             String queryString = selectElement.getText();
126             //创建Key
127             String key = namespace+"."+id;
128             //创建value
129             Mapper mapper = new Mapper();
130             mapper.setQueryString(queryString);
131             mapper.setResultType(resultType);
132             //把key和value存入mappers中
133             mappers.put(key,mapper);
134         }
135         return mappers;
136     }catch(Exception e){
137         throw new RuntimeException(e);
138     }finally{
139         in.close();
140     }
141
142 /**
143 * 注解配置方式
144 * 根据传入的参数，得到dao中所有被select注解标注的方法。
145 * 根据方法名称和类名，以及方法上注解value属性的值，组成Mapper的必要信息
146 * @param daoclassPath
147 * @return
148 */
149 private static Map<String,Mapper> loadMapperAnnotation(String
daoclassPath) throws Exception{
150     //定义返回值对象
151     Map<String,Mapper> mappers = new HashMap<String, Mapper>();
152
153     //1.得到dao接口的字节码对象
154     Class daoclass = class.forName(daoclassPath);
155     //2.得到dao接口中的方法数组
156     Method[] methods = daoclass.getMethods();
157     //3.遍历Method数组
```

```

158     for(Method method : methods){
159         //取出每一个方法，判断是否有select注解
160         boolean isAnnotated =
161             method.isAnnotationPresent(Select.class);
162         if(isAnnotated){
163             //创建Mapper对象
164             Mapper mapper = new Mapper();
165             //取出注解的value属性值
166             Select selectAnno = method.getAnnotation(Select.class);
167             String queryString = selectAnno.value();
168             mapper.setQueryString(queryString);
169             //获取当前方法的返回值，还要求必须带有泛型信息
170             Type type = method.getGenericReturnType(); //List<User>
171             //判断type是不是参数化的类型
172             if(type instanceof ParameterizedType){
173                 //强转
174                 ParameterizedType ptype = (ParameterizedType)type;
175                 //得到参数化类型中的实际类型参数
176                 Type[] types = ptype.getActualTypeArguments();
177                 //取出第一个
178                 Class domainClass = (Class)types[0];
179                 //获取domainClass的类名
180                 String resultType = domainClass.getName();
181                 //给Mapper赋值
182                 mapper.setResultType(resultType);
183             }
184             //组装key的信息
185             //获取方法的名称
186             String methodName = method.getName();
187             String className = method.getDeclaringClass().getName();
188             String key = className+"."+methodName;
189             //给map赋值
190             mappers.put(key,mapper);
191         }
192     }
193 }
194
195 }

```

a

通用知识

读取(配置)文件方式

不用：

- 绝对路径
- 相对路径

用：

- 类加载器：
 - 只能读取类路径的配置文件
- ServletContext对象
 - `getRealPath()` 得到当前应用部署的 绝对路径

绝对路径

D:/xxx/xxx.xml

- 系统未必有盘符

相对路径

src/java/main/xxx.xml

- web工程一部署，src目录就没了

类加载器

只能读取类路径的配置文件。

```
InputStream in = Resources.getResourceAsStream("SqlMapConfig.xml")
```

ServletContext

`getRealPath()` 得到当前应用部署的 绝对路径。

什么是框架？

框架的定义：(Framework)

- 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
 - 应用方面 给出的定义
 - 可被应用开发者定制的 应用骨架
 - 目的方面 给出的定义
-
- 软件开发中的一套解决方案，不同的框架解决的是不同的问题。
 - 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。
 - 框架一般是成熟的，不断升级的软件。

使用框架的好处：

- 框架封装了很多的细节，使开发者可以使用极简的方式实现功能。大大提高开发效率。

断点调试&跟踪源码

断点 想看哪就打个断点

- 不确定接口 赋值 的 是什么类对象
- 分支 , 不确定走哪里 , 各分支打断点

debug执行

一行一行执行下去

会看到对象

右键show , 可以查看

URL ? URI

- URL : Uniform Resource Locator。统一资源定位符 , 它是可以唯一标识一个资源的位置。
- 写法 : 协议 主机 端口 URI
 - <http://localhost:8080/mybatisserver/demoServlet>
- URI : Uniform Resource Identifier 统一资源标识符。它是在应用中可以唯一定位一个资源的。

案例Demo

dao实现类

不用代理dao , 自己写实现类。主要是SqlSession提供了两种方式 , 但通常不这么做。

- InputStream in = Resources.getResourceAsStream("SqlMapConfig.xml")
 - 读取 配置文件
- SqlSessionFactory builder = new SqlSessionFactoryBuilder()
 - 创建 FactoryBuilder
- SqlSessionFactory factory = builder.build(in)
 - 创建 工厂

下面两操作 可提取到实现类里 :

- new UserDaoImpl(factory)
 - 实现类 完成具体操作
 - session.selectList("com.learn.dao.UserDao.findAll") 等方法
- SqlSession session = factory.openSession()
 - 创建 SqlSession
 - session可 创建 指定接口的 代理类 , 相当于实现类
 - session可 直接 操作方法 进行数据操作

- UserDao dao = session.getMapper(UserDao.class)

◦ 创建Dao接口的 代理对象 , 功能相当于实现类

接口

```
1 public interface UserDao {  
2  
3     /**  
4      * 查询所有用户  
5      * @return  
6      */  
7     List<User> findAll();  
8  
9     /**  
10      * 保存用户  
11      * @param user  
12      */  
13     void saveUser(User user);  
14  
15     /**  
16      * 更新用户  
17      * @param user  
18      */  
19     void updateUser(User user);  
20  
21     /**  
22      * 根据Id删除用户  
23      * @param userId  
24      */  
25     void deleteUser(Integer userId);  
26  
27     /**  
28      * 根据id查询用户信息  
29      * @param userId  
30      * @return  
31      */  
32     User findById(Integer userId);  
33  
34     /**  
35      * 根据名称模糊查询用户信息  
36      * @param username  
37      * @return  
38      */  
39     List<User> findByName(String username);  
40  
41     /**  
42      * 查询总用户数  
43      * @return  
44      */  
45     int findTotal();  
46  
47 }  
48 }
```

映射文件

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE mapper
3     PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4     "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5
6 <mapper namespace="com.learn.dao.UserDao"><!-- UserDao接口的全限定类名 -->
7
8     <!-- 配置 查询结果的列名 和 实体类的属性名 的对应关系 -->
9     <!-- 下边不用resultType 可以用resultMap -->
10    <resultMap id="userMap" type="com.learn.domain.User">
11        <!-- property类中属性名 column数据库列名 -->
12        <!-- 主键字段的对应 -->
13        <id property="id" column="id"></id>
14        <!-- 非主键字段的对应 -->
15        <result property="username" column="username"></result>
16        <result property="address" column="address"></result>
17        <result property="sex" column="sex"></result>
18        <result property="birthday" column="birthday"></result>
19    </resultMap>
20
21    <!-- 配置查询所有 id是方法名 -->
22    <select id="findAll" resultType="com.learn.domain.User"><!-- 实体类 全限
定类名 封装成啥 -->
23        <!-- SQL语句 -->
24        select * from user;
25    </select>
26
27    <!-- 保存用户 -->
28    <insert id="saveUser" parameterType="user"> <!-- 参数类型 -->
29        <!-- 配置插入操作后，获取插入数据的id -->
30        <!--          数据库          -->
31        <selectKey keyProperty="id" keyColumn="id" resultType="int"
order="AFTER">
32            select last_insert_id();
33        </selectKey>
34
35        <!-- #{getXXX 的xxx} -->
36        insert into user(username, address, sex, birthday)
37            values(#{$username},#${address},#${sex},#${birthday});
38    </insert>
39
40    <!-- 更新用户 -->
41    <update id="updateUser" parameterType="com.learn.domain.User">
42        update user set username=#{$username}, address=#{$address}, sex=#
{sex}, birthday=#{$birthday}
43        where id=#{$id}
44    </update>
45
46    <!-- 删除用户 -->
47    <delete id="deleteUser" parameterType="Integer"><!-- int INT Integer
48        INTEGER java.lang.Integer -->
49        <!-- 只有一个参数 写什么都可以 -->
50        delete from user
```

```

50         where id = #{userIdiddukeyi}
51     </delete>
52
53     <!-- 根据id查询用户 -->
54     <select id="findById" parameterType="int"
55         resultType="com.learn.domain.User">
56         select * from user
57         where id = #{suibianxie}
58     </select>
59
60     <!-- 根据名称模糊查询 -->
61     <select id="findByName" parameterType="string"
62         resultType="com.learn.domain.User">
63         <!-- 只支持这种注释 -->
64         <!-- 预处理方式 %调用时传 这里 这好像不能加 -->
65         <!-- select * from user
66         where username like #{username} -->
67
68         <!-- 字符串拼接 这里接收的值 必须写value 源码写死了 '${}' '#' -->
69         select * from user
70         where username like '%${value}%'
71     </select>
72
73     <!-- 获取用户总记录条数 -->
74     <select id="findTotal" resultType="int">
75         select count(id) from user;
76     </select>
77 </mapper>

```

实现类

- private SqlSessionFactory factory
- SqlSession session = factory.openSession()
 - session.xxx()

```

1  public class UserDaoImpl implements UserDao {
2
3     private SqlSessionFactory factory;
4
5     public UserDaoImpl(SqlSessionFactory factory) {
6         this.factory = factory;
7     }
8
9     public List<User> findAll() {
10        //1.根据factory获取SqlSession对象
11        SqlSession session = factory.openSession();
12        //2.调用SqlSession中的方法 实现查询列表
13        // 参数就是 配置文件中的信息 全限定类名+id方法名
14        List<User> users =
15        session.selectList("com.learn.dao.UserDao.findAll");
16        //3.释放资源
17        session.close();
18        return users;
19    }

```

```
18    }
19
20    public void saveUser(User user) {
21        //1.根据factory获取SqlSession对象
22        SqlSession session = factory.openSession();
23        //2.调用方法实现保存
24        session.insert("com.learn.dao.UserDao.saveUser", user); //参数就是能获
取配置信息的key
25        //3.提交事务
26        session.commit();;
27        //4.释放资源
28        session.close();
29    }
30
31    public void updateUser(User user) {
32        //1.根据factory获取SqlSession对象
33        SqlSession session = factory.openSession();
34        //2.调用方法实现保存
35        session.update("com.learn.dao.UserDao.updateUser", user);
36        //3.提交事务
37        session.commit();;
38        //4.释放资源
39        session.close();
40    }
41
42    public void deleteUser(Integer userId) {
43        //1.根据factory获取SqlSession对象
44        SqlSession session = factory.openSession();
45        //2.调用方法实现保存
46        session.update("com.learn.dao.UserDao.deleteUser", userId);
47        //3.提交事务
48        session.commit();;
49        //4.释放资源
50        session.close();
51    }
52
53    public User findById(Integer userId) {
54        //1.根据factory获取SqlSession对象
55        SqlSession session = factory.openSession();
56        //2.调用SqlSession中的方法 实现查询列表
57        // 参数就是 配置文件中的信息 全限定类名+id方法名
58        User user = session.selectOne("com.learn.dao.UserDao.findById",
userId);
59        //3.释放资源
60        session.close();
61        return user;
62    }
63
64    public List<User> findByName(String username) {
65        //1.根据factory获取SqlSession对象
66        SqlSession session = factory.openSession();
67        //2.调用SqlSession中的方法 实现查询列表
68        // 参数就是能获取 配置文件中的信息的key: 全限定类名+id方法名
69        List<User> users =
session.selectList("com.learn.dao.UserDao.findByName", username);
70        //3.释放资源
71        session.close();
72        return users;
```

```
73     }
74
75     public int findTotal() {
76         //1.根据factory获取SqlSession对象
77         SqlSession session = factory.openSession();
78         //2.调用SqlSession中的方法 实现查询列表
79         // 参数就是 配置文件中的信息 全限定类名+id方法名
80         Integer count =
81             session.selectOne("com.learn.dao.UserDao.findTotal");
82         //3.释放资源
83         session.close();
84         return count;
85     }
86 }
```

EOF
