Mybatis

环境：

- jDK1.8
- Mysql 5.7
- maven 3.6.1
- IDEA

回顾：

- JDBC
- Mysql
- Java基础
- Maven
- Junit

SSM框架：配置文件的。看官方文档

官网：https://mybatis.org/mybatis-3/zh/index.html

1、简介

1.1、什么是Mybatis

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

如何获得Mybatis?

- maven仓库

```
<!-- https://mvnrepository.com/artifact/org.mybatis/mybatis -->
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis</artifactId>
    <version>3.5.7</version>
</dependency>

```

- GitHub:

1.2、持久化

数据持久化

- 持久化就是将程序的数据在持久状态和瞬时状态转化的过程
- 内存：**断电即失**
- 数据库(JDBC)，io文件持久化

为什么需要持久化

- 有一些对象，不能让他丢掉
- 内存太贵

1.3、持久层

Dao层，Servlet层，Controller层....

- 完成持久化工作的代码块
- 层界限十分明显。

1.4、为什么需要Mybatis?

- 帮助程序员将数据存入到数据库中。
- 方便
- 传统的JDBC代码太复杂了，简化。框架。自动化。
- 优点
  - 简单易学
  - 灵活
  - sql和代码的分离，提高了可维护性
  - 提供映射标签，支持对象与数据库的orm字段关系映射
  - 提供对象关系映射标签，支持对象关系组维护
  - 提供xml标签，支持编写动态sql。

2、第一个Mybatis程序

2.1、环境搭建

搭建数据库

```
CREATE DATABASE `mybatis`
USE mybatis

CREATE TABLE `user`(
   `id` INT(20) NOT NULL PRIMARY KEY,
   `name` VARCHAR(20) DEFAULT NULL,
   `pwd`  VARCHAR(20) DEFAULT NULL
)ENGINE=INNODB DEFAULT CHARSET=utf8;

INSERT INTO `user`(`id`,`name`, `pwd`) VALUES(1,'咚咚','123456')
,(2,'咚咚1','123456'),(3,'咚咚2','123456');
```

新建项目

1. 新建一个普通的maven项目
2. 删除src目录

3. 导入maven依赖

```
  <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>8.0.16</version>
        </dependency>
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis</artifactId>
            <version>3.5.6</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>
    </dependencies>
```

2.2、创建一个模块

- 编写mybaits的核心配置文件

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.jdbc.Driver"/>
                <property name="url"
value="jdbc:mysql://localhost:3306/mybatis?
useSSL=true&amp;useUnicode=true&amp;characteEncoding=UTF-8"/>
                <property name="username" value="root"/>
                <property name="password" value="root"/>
            </dataSource>
        </environment>
    </environments>
</configuration>
```

- 编写mybatis工具类

```
package com.dongdong.utils;

import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;

import java.io.IOException;
import java.io.InputStream;

//SqlSessionFactory  --> SqlSession
```

```
public class MybatisUtils {
    private static SqlSessionFactory sqlSessionFactory;
    static {
        try {
            //获取SqlSessionFactory对象
            String resource = "mybatis-config.xml";
           InputStream inputStream =
Resources.getResourceAsStream(resource);
            sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    public static SqlSession getSqlSession() {
        return sqlSessionFactory.openSession();
    }
}

```

2.3、编写代码

- 实体类

```
package com.dongdong.pojo;

//实体类
public class User {
    private int id;
    private String name;
    private String pwd;

    public User() {
    }

    public User(int id, String name, String pwd) {
        this.id = id;
        this.name = name;
        this.pwd = pwd;
    }

    public int getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

```

```
    public String getPwd() {
        return pwd;
    }

    public void setPwd(String pwd) {
        this.pwd = pwd;
    }

}
```

- Dao接口

```
public interface UserDao {
    List<User> getUserList();
}
```

- 接口实现类 由原来的UserDaoImpl转化为一个Mapper配置文件

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<!--namespace=绑定一个对应的Dao/Mappar接口-->
<mapper namespace="org.mybatis.example.BlogMapper">
  <select id="getUserList" resultType="com.dongdong.pojo.User">   //如果设置别名就简化成User
    select * from mybatis.user

  </select>
</mapper>
```

2.4、测试

注意点：org.apache.ibatis.binding.BindingException: Type interface com.dongdong.dao.UserDao is not known to the MapperRegistry

MapperRegistry是什么?

核心配置文件中注册mapper

junit测试

3、CRUD

1、namespace

namespace中的包名要和Dao/mapper接口的包名要一致!

2、Select

选择，查询语句：

- id：对应的namespace中的方法名
- resultType：sql语句执行的返回值!　select查询语句才有
- parameterType：参数的类型!

1. 编写接口

```java
package com.dongdong.dao;

import com.dongdong.pojo.User;

import java.util.List;

public interface UserMapper {
    List<User> getUserList();
    //查询id
    User getUserById(int id);
    //insert插入
    int addUser(User user);
}

```

2. 编写对应的mapper中的sql语句

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<!--namespace=绑定一个对应的Dao/Mappar接口-->
<mapper namespace="com.dongdong.dao.UserMapper">
  <select id="getUserList" resultType="com.dongdong.pojo.User">
    select * from mybatis.user

  </select>
  <select id="getUserById" parameterType="int"
resultType="com.dongdong.pojo.User">
    select * from mybatis.user where id = #{id}

  </select>
  <insert id="addUser" parameterType="com.dongdong.pojo.User"  >

    insert into mybatis.user (id, name, pwd) values (#{id},#{name},#{pwd})
  </insert>
</mapper>
```

3. 测试

```java
 public void addUser() {
        SqlSession sqlSession = MybatisUtils.getSqlSession();
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
        int i = mapper.addUser(new User(4,"哈哈","123456"));
        System.out.println(i > 0?"成功":"失败");
     // 提交事务  Mybaits默认把自动提交事务给关闭了
        sqlSession.commit();
        sqlSession.close();

    }
```

3、Insert

4、update

5、Delete

注意：

- 增删改需要提交事务！

6、分析错误

- 标签不要匹配错
- resource 绑定mapper，需要使用路径
- 程序配置文件必须符合规范！

7、万能Map

假设，我们的实体类，或者数据库中的表，字段或者参数过多，我们应当考虑使用Map!

```
@Test
  public void addUser2() {
      SqlSession sqlSession = MybatisUtils.getSqlSession();
      UserMapper mapper = sqlSession.getMapper(UserMapper.class);
      HashMap<String, Object> map = new HashMap<String, Object>();
      map.put("userid",5);
      map.put("username","哈哈");
      map.put("password","123456");
      int i = mapper.addUser2(map);
      System.out.println(i > 0?"成功":"失败");
      sqlSession.commit();
      sqlSession.close();
  }
```

Map传递参数，直接在sql中取出key即可！ parameterType "map"

对象传递参数，直接在sql中取对象的属性即可！parameterType "Object"

只有一个基本类型参数的情况下，可以直接在sql中取到！

7.1、模糊查询

1. Java代码执行的时候，传递通配符% %
2. 在sql拼接中使用通配符！

4、配置解析

1、核心配置文件

- mybatis-config.xml
- MyBatis 的配置文件包含了会深深影响MyBatis行为的设置和属性信息

```
configuration（配置）
properties（属性）
settings（设置）
typeAliases（类型别名）
typeHandlers（类型处理器）
objectFactory（对象工厂）
plugins（插件）
environments（环境配置）
environment（环境变量）
transactionManager（事务管理器）
dataSource（数据源）
databaseIdProvider（数据库厂商标识）
mappers（映射器）2、环境配置 （environments）
```

MyBatis 可以配置成适应多种环境

不过要记住：尽管可以配置对个环境，但每个SqlSessionFactory 实例只能选择一种环境。

Mybatis默认的事务管理器就是JDBC，连接池：POOLED

3、属性 （properties）

我们可以通过properties属性来实现引用配置文件

这些属性可以在外部进行配置，并可以进行动态替换。你既可以在典型的 Java 属性文件中配置这些属性，也可以在 properties 元素的子元素中设置。

编写配置文件

```
driver=com.mysql.jdbc.Driver
url=jdbc:mysql://localhost:3306/mybatis?
useSSL=true&;useUnicode=true&;characterEncoding=UTF-8
username=root
password=root
```

在核心配置文件中引入

```
<configuration>

<!--引入外部配置文件-->
    <properties resource="db.properties">
        <property name="username" value="root"/>
        <property name="password" value="root"/>
    </properties>
```

- 可以直接引入外部文件
- 可以在其中增加一些属性配置
- 如果两个文件有同一个字段，优先使用外部配置文件的!

4、类型别名（typeAliases）

类型别名可为 Java 类型设置一个缩写名字。 它仅用于 XML 配置，意在降低冗余的全限定类名书写

```
<!--可以给实体类取别名-->
    <typeAliases>
        <typeAlias type="com.dongdong.pojo.User" alias="User"/>
    </typeAliases>
```

也可以指定一个包名，MyBatis 会在包名下面搜索需要的 Java Bean

扫描实体类的包，它的默认别名就为这个类的 类名，首字母小写！

```
<!--可以给实体类取别名-->
    <typeAliases>
        <package name="com.dongdong.pojo"/>
    </typeAliases>
```

在实体类比较少的时候，使用第一种方式。

如何实体类十分多，建议使用第二种。

第一种可以DIY别名，第二种则不可以 ，如果非要改，需要在实体类增加注解

```
//实体类
@Alias("user")
```

5、其他配置

- **类型处理器（typeHandlers）**
- **对象工厂（objectFactory）**
- **插件（plugins）**
  - mybatis-generator-core
  - mybatis-plus
  - 通用mapper

6、映射器 （mappers）

MapperRegisity：注册绑定我们的Mapper文件；

方式一：

```
<!--每一个Mapper.XML都需要在Mybatis核心配置文件中注册！-->
    <mappers>
        <mapper resource="com/dongdong/dao/UserMapper.xml"></mapper>
    </mappers>

```

方式二：使用class文件绑定注册

```
<!--每一个Mapper.XML都需要在Mybatis核心配置文件中注册！-->
<mappers>
        <mapper class="com.dongdong.dao.UserMapper"></mapper>
 </mappers>
```

注意点：

- 接口和他的Mapper配置文件必须同名！
- 接口和他的Mapper配置文件必须在同一个包下！

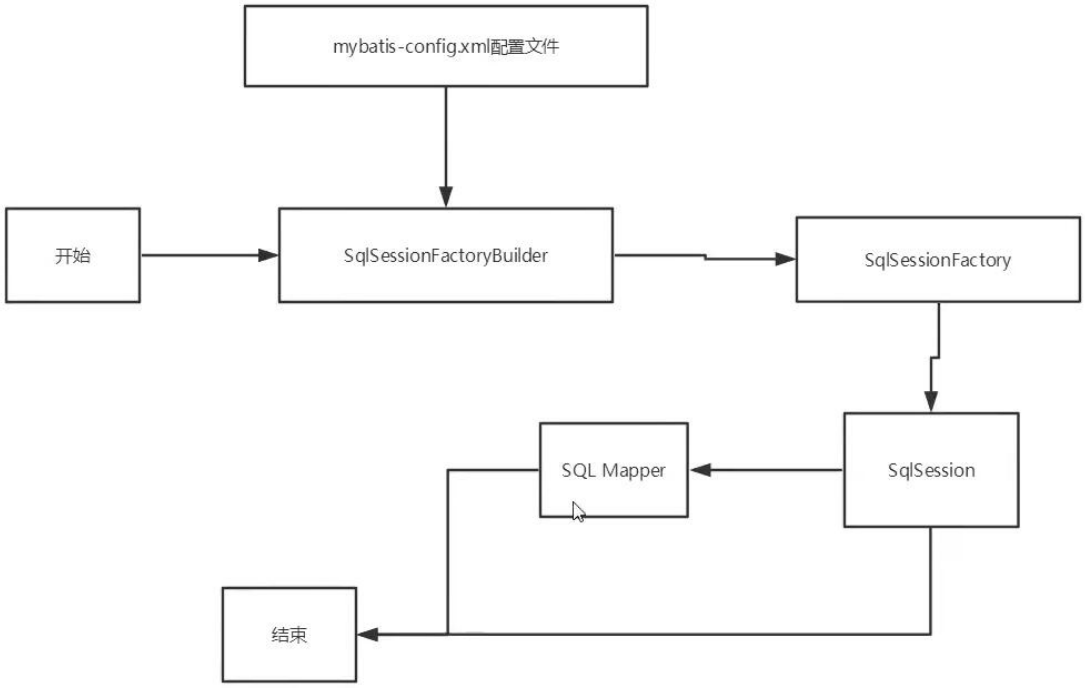
方式三：使用扫描包进行注册绑定

```
<!--每一个Mapper.XML都需要在Mybatis核心配置文件中注册！-->
<mappers>
        <package name="com.dongdong.dao"/>
    </mappers>
```

注意点：

- 接口和他的Mapper配置文件必须同名！
- 接口和他的Mapper配置文件必须在同一个包下！

8、生命周期和作用域



不同作用域和生命周期类别是至关重要的，因为错误的使用会导致非常严重的并发问题。

SqlSessionFactoryBuilder

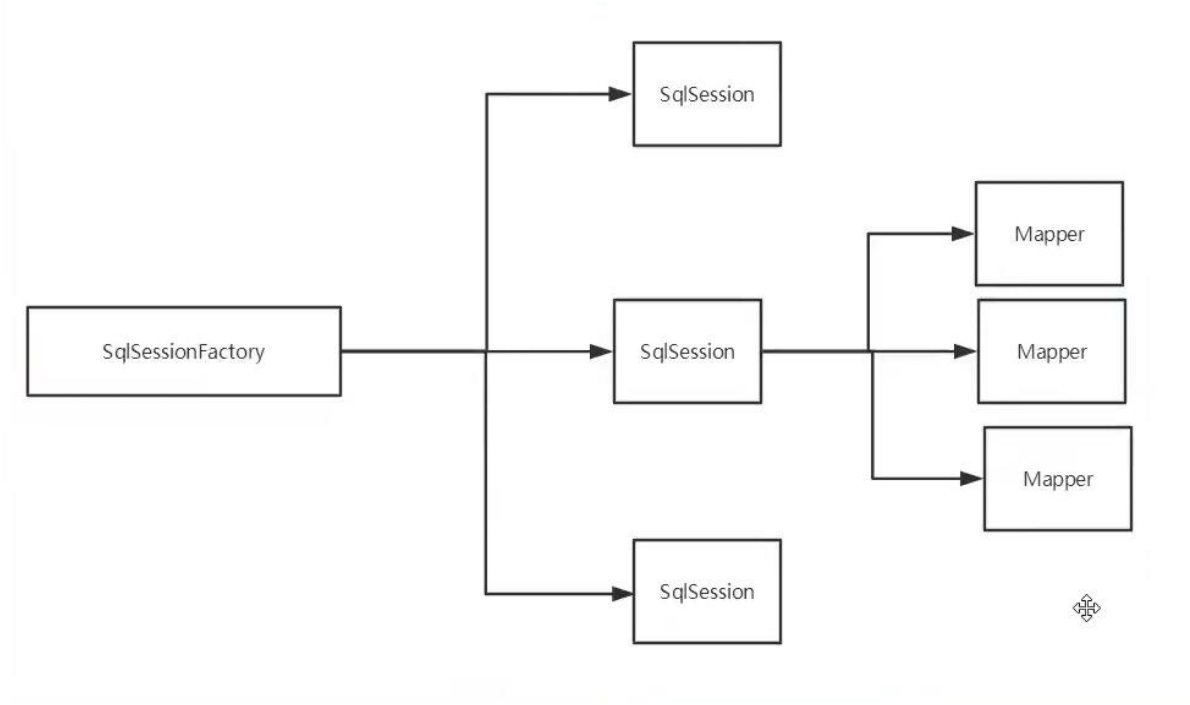
- 一旦创建了SqlSessionFactory，就不再需要它了
- 局部变量

SqlSessionFactory

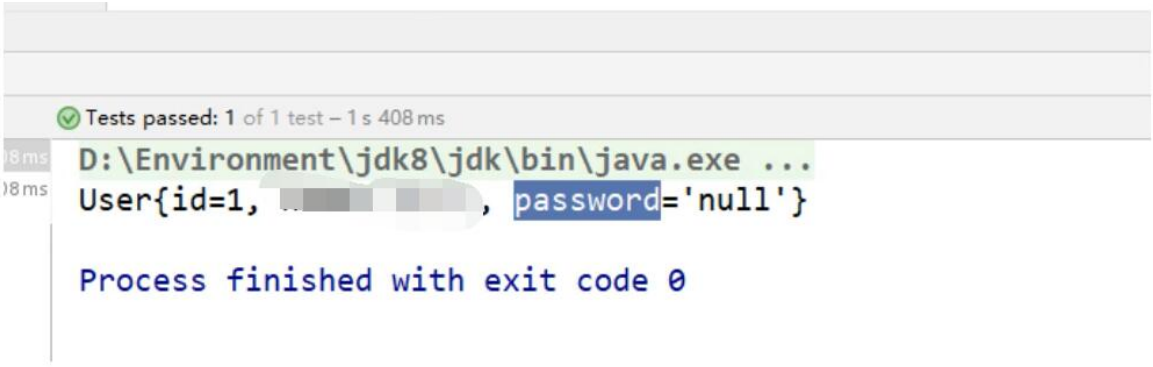
- 说白了就是可以想象为：数据库连接池
- SqlSessionFactory 一旦被创建就应该在应用的运行期间一直存在，没有任何理由丢弃它或重新创建另一个实例
- 因此 SqlSessionFactory 的最佳作用域是应用作用域
- 最简单的就是使用**单例模式**或者**静态单例模式**

SqlSession

- 连接到连接池的一个请求！
- SqlSession 的实例不是线程安全的，因此是不能被共享的，所以它的最佳的作用域是请求或方法作用域
- 用完之后需要赶紧关闭，否则资源被占用！



5、解决属性名和字段名不一致问题




```
//select * from mybatis.user where id = #{id}
//类型处理器
//select id, name, pwd from mybatis.user where id=#{id}
```

解决办法：

- 起别名：

```
<select id="getUserList" resultType="user">
   select id, name, pwd as password from mybatis.user where id = #{id}
 </select>
```

2、resultMap

结果集映射

```
id  name    pwd
id  name    password
```

```
<!--结果集映射-->
  <resultMap id="UserMap" type="User">
      <!--colum数据库中的字段，property实体类中的属性-->
    <result column="id" property="id"/>
    <result column="name" property="name"/>
    <result column="pwd" property="password"/>
  </resultMap>
  <select id="getUserById" resultMap="UserMap">
    select * from mybatis.user where id = #{id}

  </select>
```

- `resultMap` 元素是 MyBatis 中最重要最强大的元素
- ResultMap 的设计思想是，对简单的语句做到零配置，对于复杂一点的语句，只需要描述语句之间的关系就行了。

6、日志

6.1、日志工厂

如果一个数据库操作， 出现了异常， 我们需要排错。 日志就是最好的助手！

曾经：sout、debug

现在：日志复制

logImpl	指定 MyBatis 所用日志的具体实现，未指定时将自动查找。	SLF4J | LOG4J | LOG4J2 | JDK_LOGGING | COMMONS_LOGGING | STDOUT_LOGGING | NO_LOGGING	未设置

- SLF4J |
- LOG4J | 掌握
- LOG4J2 |
- JDK_LOGGING |
- COMMONS_LOGGING |
- STDOUT_LOGGING | 掌握
- NO_LOGGING

在Mybatis中具体实现使用那个日志实现，在设置中设定！

STDOUT_LOGGING

在Mybatis核心配置文件中，配置我们的日志！

```
<settings>
        <!--标准的日志工厂实现-->
    <setting name="logImpl" value="STDOUT_LOGGING"/>
</settings>
```

6.2、Log4j

1、什么是Log4j?

- **Log4j是Apache的一个开源项目，通过使用Log4j，我们可以控制日志信息输送的目的地是控制台、文件、GUI组件**
- **我们也可以控制每一条日志的输出格式**
- **通过定义每一条日志信息的级别，我们能够更加细致地控制日志的生成过程**
- **通过一个配置文件来灵活地进行配置，而不需要修改应用的代码。**

1. 先导入log4j的包

```
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.17</version>
    </dependency>
```

2. log4j.properties

```

```

3. 配置配置文件

```
<settings>
        <setting name="logImpl" value="Log4j"/>
    </settings>
```

4. 使用!

简单使用

1. 在要使用Log4j的类中，导入包 import org.apache.log4j.Logger;
2. 日志对象，参数为当前类的class

```
static Logger logger = Logger.getLogger(UserDaoTest.class);
```

3. 日志级别

```
info
debug
error
```

7、分页

7.1、使用Limit分页

思考：为什么要分页?

- 减少数据的处理量

实现Mybatis实现分页，核心SQL

1. 接口
2. Mapper.xml
3. 测试

7.2、RowBounds分页

不再使用SQL实现分页

1. 接口
2. Mapper.xml
3. 测试

8、使用注解开发

8.1、使用注解开发

1. 注解在接口上实现!

```
@Select("select * from user")
List<User> getUserList();
```

2. 需要在核心的配置文件中绑定接口

```
<!--绑定接口-->
<mappers>
    <mapper class="com.dongdong.dao.UserMapper"/>
 </mappers>
```

本质：反射机制实现!

底层：动态代理!

8.2、CRUD

我们可以在工具类创建的时候实现自动提交事务!

```
public static SqlSession getSqlSession() {
        return sql.openSession(true);
    }
```

编写接口，增加注解

```
package com.dongdong.dao;

```

```java
import com.dongdong.pojo.User;
import org.apache.ibatis.annotations.Param;
import org.apache.ibatis.annotations.Select;

import java.util.List;

public interface UserMapper {
    @Select("select * from user")
    List<User> getUserList();
    // 方法存在多个参数， 所有的参数前面必须加上 @Param("id")注解
    @Select("select * from user where id = #{id}")
    User getUserById(@Param("id") int id);
}

```

【注意：我们必须要将接口注册绑定到我们的核心配置文件中！】

关于@Param()注解

- 基本类型的参数或者String类型，需要加上
- 引用类型不需要加
- 如果只有一个基本类型的话，可以忽略，但是建议大家都加上！
- 我们在SQL中引用的就是我们这里的@Param()中设定的属性名！

#{} 和 #{}区别

- 一个安全，一个不安全
- ${} 会拼接sql，会产生Sql注入问题

9、Lombok

使用步骤：

1. 在IDEA中安装Lombok插件
2. 在项目中导入lombok的jar包

```xml
<dependencies>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.18.20</version>
        </dependency>
    </dependencies>
```

3. 在实体类上加注解！

说明：

```
@Data：无参构造，get，set，toString，hashcode，equals
@AllArgsConstructor
@NoArgsConstructor
@Getter
@Setter
@ToString
```

10、多对一处理

sql：

```sql
CREATE TABLE `teacher`(
  `id` INT(10)  NOT NULL,
  `name` VARCHAR(30) DEFAULT NULL,
  PRIMARY KEY (id)
)ENGINE=INNODB DEFAULT CHARSET=utf8

INSERT INTO teacher(`id`,`name`) VALUES(1,"咚老师");


CREATE TABLE `student`(
  `id` INT(10) NOT NULL,
  `name` VARCHAR(30) DEFAULT NULL,
  `tid` INT(10) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `fktid` (`tid`),
  CONSTRAINT `fktid` FOREIGN KEY (`tid`) REFERENCES `teacher` (`id`)
)ENGINE=INNODB DEFAULT CHARSET=utf8


INSERT INTO `student` (`id`,`name`, `tid`) VALUES('1',"小明","1")
INSERT INTO `student` (`id`,`name`, `tid`) VALUES('2',"小红","1")
INSERT INTO `student` (`id`,`name`, `tid`) VALUES('3',"小张","1")
INSERT INTO `student` (`id`,`name`, `tid`) VALUES('4',"小李","1")
INSERT INTO `student` (`id`,`name`, `tid`) VALUES('5',"小王","1")
```

测试环境

1. 导入lombok
2. 新建实体类 Teache，Student
3. 建立Mapper接口
4. 建立Mapper.xml文件
5. 在核心配置文件中绑定注册我们的Mapper接口或者文件！
6. 测试查询释是否成功！

按照查询嵌套处理

```xml
<mapper namespace="com.dongdong.dao.StudentMapper">


<!--
        思路:
         1. 查询所有的学生信息
         2. 根据查询出来的学生的tid，寻找对应的老师

-->
    <select id="getStudent" resultMap="StudentTeacher">
        select * from student;
    </select>
    <resultMap id="StudentTeacher" type="Student">
        <result property="id" column="id"/>
        <result property="name" column="name"/>
            <!--复杂的属性，我们需要单独处理
                对象：association
```

```
                集合：collection
            -->
        <association property="teacher" column="tid" javaType="Teacher"
select="getTeacher"/>
    </resultMap>
    <select id="getTeacher" resultType="Teacher">
        select * from teacher where id = #{id}

    </select>
```

按照结果嵌套处理

```
  <!--按照结果嵌套处理-->
    <select id="getStudent2" resultMap="StudentTeacher2">

        select s.id sid,s.name sname,t.name tname from student s,teacher t
where s.tid = t.id;
    </select>
    <resultMap id="StudentTeacher2" type="Student">
        <result property="id" column="sid"/>
        <result property="name" column="sname"/>
        <association property="teacher" javaType="Teacher">
            <result property="name" column="tname"/>
        </association>
    </resultMap>
```

Mysql多对一查询方式：

- 子查询
- 联表查询

11、一对多处理

比如：一个老师有多个学生！

对于老师而言，就是一对多的关系！

1. 环境搭建

实体类

```
@Data
public class Student {
    private int id;
    private String name;
    private int tid;
}



@Data
public class Teacher {
    private int id;
```

```
    private String name;

    //一个老师拥有多个学生
    private List<Student> students;
}

```

按照结果嵌套处理

```
  <select id="getTeacher" resultMap="TeacherStudent">
        select s.id sid,s.name sname,t.id tid,t.name tname
        from student s, teacher t where s.tid = t.id and t.id = #{tid}
    </select>
    <resultMap id="TeacherStudent" type="Teacher">
        <result property="id" column="tid"/>
        <result property="name" column="tname"/>
        <!--复杂的属性，我们需要单独处理
            对象：association
            集合：collection
            javaType = "" 指定属性的类型!
            集合中的泛型信息，我们使用ofType获取
        -->
        <collection property="students" ofType="Student">
            <result property="id" column="sid"/>
            <result property="name" column="sname"/>
            <result property="tid" column="tid"/>
        </collection>
    </resultMap>
```

按照查询嵌套处理

```
<select id="getTeacher2" resultMap="TeacherStudent2">

        select * from mybatis.teacher where id = #{tid}
    </select>
    <resultMap id="TeacherStudent2" type="Teacher">
        <collection property="students" javaType="ArrayList" ofType="Student" select="getStudentByTeacherId" column="id"/>
    </resultMap>
    <select id="getStudentByTeacherId" resultMap="Student">
        select * from mybatis.student where tid = #{tid}

    </select>
```

小结

1. 关联-association 多对一
2. 集合-collection 一对多
3. javaType & ofType
    1. JavaType 用来指定实体类中属性的类型
    2. ofType 用来指定映射到List或者集合中的pojo类型，泛型中的约束类型!

注意点：

- 保证SQL的可读性， 尽量保证通俗易懂
- 注意一对多和多对一中，属性名和字段的问题！
- 如果问题不好排查错误，可以使用日志，建议使用Log4j

12、动态SQL

什么是动态SQl：动态SQL就是指根据不同的条件生成不同的SQL语句

搭建环境

```sql
CREATE TABLE `blog` (
 `id` VARCHAR(50) NOT NULL COMMENT '博客id',
 `title` VARCHAR(50) NOT NULL COMMENT '博客标题',
 `author` VARCHAR(50) NOT NULL COMMENT '博客作者',
 `create_time` DATETIME NOT NULL COMMENT '创造时间',
 `views` INT(30) NOT NULL COMMENT '浏览量'
)ENGINE=INNODB DEFAULT CHARSET=utf8
```

创建一个基础工程

1. 导包
2. 编写配置文件
3. 编写实体类

```java
@Data
public class Blog {
    private int id;
    private String title;
    private String author;
    private Date createTime;
    private int views;
}
```

4. 编写实体类对应Mapper接口和Mapper.xml文件

IF

```xml
<select id="queryBlogIF" resultType="Blog" parameterType="map" >

        select * from mybatis.blog where 1=1
        <if test="title != null">
           and title = #{title}
        </if>
        <if test="author!=null">
          and author = #{author}
        </if>
    </select>
```

choose(when,otherwise)

```
<select id="queryBlogChoose" resultType="Blog" parameterType="map">
       select * from mybatis.blog
       <where>
           <choose>
               <when test="title != null">
                   title = #{title}
               </when>
               <when test="title != null">
                  and author = #{author}
               </when>
                   <otherwise>
                       and views = #{views}
                   </otherwise>
           </choose>

       </where>
   </select>
```

trim(where,set)

```
select * from mybatis.blog
       <where>
           <if test="title != null">
               and title = #{title}
           </if>
           <if test="author!=null">
               and author = #{author}
           </if>
       </where>
```

```
<update id="updateBlog" parameterType="map" >
       update mybatis.blog
   <set>
       <if test="title != null">
           title = #{title},
       </if>
       <if test="author != null">
           author = #{author}
       </if>
   </set>
       where id = #{id}
   </update>
```

所谓的动态SQL，本质还是SQL语句，只是我们可以在SQL层面，去执行一个逻辑代码

SQL片段

有的时候，我们可能会将一些功能的部分抽取出来，方便复用！

1. 使用SQL标签抽取公共的部分

```
<sql id="if-titie-author">
        <if test="title != null">
            and title = #{title}
        </if>
        <if test="author!=null">
            and author = #{author}
        </if>
    </sql>
```

2. 在需要使用的地方使用include标签引用即可!

```
<select id="queryBlogIF" resultType="Blog" parameterType="map" >

        select * from mybatis.blog
        <where>
           <include refid="if-titie-author"></include>
        </where>

    </select>
```

注意事项:

- 最好基于单表定义SQL片段
- 不要存在where标签

Foreach

```
<!--我们现在传递一个map，这个map中可以存放一个集合-->
    <select id="queryBlogForeach" parameterType="map" resultType="blog">
        select * from mybatis.blog
        <where>
                <foreach collection="ids" item="id" open="and (" close=")"
separator="or">
                    id =#{id}
                </foreach>
        </where>
    </select>
```

```
public void Test3() {
        SqlSession sqlSession = MybatisUtils.getSqlSession();

        BlogMapper mapper = sqlSession.getMapper(BlogMapper.class);
        HashMap map = new HashMap();
        ArrayList ids = new ArrayList();
        map.put("ids",ids);
        List<Blog> blogs = mapper.queryBlogForeach(map);
        for (Blog blog : blogs) {
            System.out.println(blogs);
        }


        sqlSession.close();


    }
```

13、缓存

13.1、简介

```
查询        :      连接数据库，  耗资源！
      一次查询的结果，给他暂存在一个可以直接取到的地方！  -->内存     :  缓存

我们再次查询相同数据的时候，直接走缓存，就不用走数据库了
```

1. 什么是缓存[Cache]？

- 存在内存中的临时数据。
- 将用户经常查询的数据放在缓存(内存)中，用户去查询数据就不用从磁盘上(关系型数据库数据文件)查询，从缓存中查询,从而提高查询效率,解决了高并发系统的性能问题。

2. 为什么使用缓存？

- 减少和数据库的交互次数,减少系统开销,提高系统效率。

3. 什么样的数据能使用缓存？

- 经常查询并且不经常改变的数据。 【可以使用缓存】

13.2、Mybatis缓存

- MyBatis包含一个非常强大的查询缓存特性，它可以非常方便地定制和配置缓存，缓存可以极大的提升查询效率。
- MyBatis系统中默认定义了两及缓存：**一级缓存**和**二级缓存**
    - 默认情况下，只有一级缓存开启。（SqlSession级别的缓存，也称为本地缓存）
    - 二级缓存需要手动开始和配置，他是基于namespace级别的缓存。
    - 为了提高扩展性，MyBatis定义了缓存接口Cache。我们可以通过实现Cache接口来自定义二级缓存

13.3、一级缓存

- 一级缓存也叫本地缓存：
    - 与数据库同一次会话期间查询到的数据会放在本地缓存中。
    - 以后如果需要获取相同的数据，直接从缓存中拿，没必要再去查询数据库；

测试：

1. 开启日志！
2. 测试在一个Session中查询两次相同的记录
3. 查看日志输出

```
Checking to see if class com.dongdong.pojo.User matches criteria [is assignable to Object]
PooledDataSource forcefully closed/removed all connections.
PooledDataSource forcefully closed/removed all connections.
PooledDataSource forcefully closed/removed all connections.
PooledDataSource forcefully closed/removed all connections.
Opening JDBC Connection
Created connection 2007331442.
==>  Preparing: select * from mybatis.user where id =?
==> Parameters: 1(Integer)
<==    Columns: id, name, pwd
<==        Row: 1, 咚咚, 123456
<==      Total: 1
User(id=1, name=咚咚, pwd=123456)
User(id=1, name=咚咚, pwd=123456)
true
Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@77a57272]
Returned connection 2007331442 to pool.
```

•同一个地址，两个对象是相同的

缓存失效的情况：

1. 查询不同的东西
2. 增删改操作，可能会改变原来的数据，所以必定会刷新缓存！
3. 查询不同的Mapper.xml
4. 手动清理缓存！

```
PooledDataSource forcefully closed/removed all connections.
Opening JDBC Connection
Created connection 2007331442.
==>  Preparing: select * from mybatis.user where id =?
==> Parameters: 1(Integer)
<==    Columns: id, name, pwd
<==        Row: 1, 咚咚, 123456
<==      Total: 1
User(id=1, name=咚咚, pwd=123456)
==>  Preparing: select * from mybatis.user where id =?
==> Parameters: 1(Integer)
<==    Columns: id, name, pwd
<==        Row: 1, 咚咚, 123456
<==      Total: 1
User(id=1, name=咚咚, pwd=123456)
false
Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@77a57272]
```

小结：一级缓存默认是开启的，只在一次Sqlsession中有效，也就是拿到连接到关闭连接这个区间！

13.4、二级缓存

- 二级缓存也叫全局缓存，一级缓存作用域太低了，所以诞生了二级缓存
- 基于namespace级别的缓存，一个名称空间，对应一个二级缓存；
- 工作机制
  - 一个会话查询一条数据，这个数据就会被放在当前会话的一级缓存中。
  - 如果当前会话关闭了,这个会话对应的- -级缓存就没了;但是我们想要的是，会话关闭了，一级缓存中的数据被保存到二级缓存中;
  - 新的会话查询信息，就可以从二级缓存中获取内容;

步骤：

1. 开启全局缓存

```xml
<settings>
     <!--开启全局缓存-->
      <setting name="cacheEnabled" value="true"/>
   </settings>
```

2. 在要使用二级缓存的Mapper中开启

```xml
<!--在当前Mapper.xml中使用缓存-->
  <cache/>
```

也可以自定义一些参数

```xml
  <!--在当前Mapper.xml中使用缓存-->
  <cache
     eviction="FIFO"
     flushInterval="60000"
     size="512"
     readOnly="true"/>
```

3. 测试

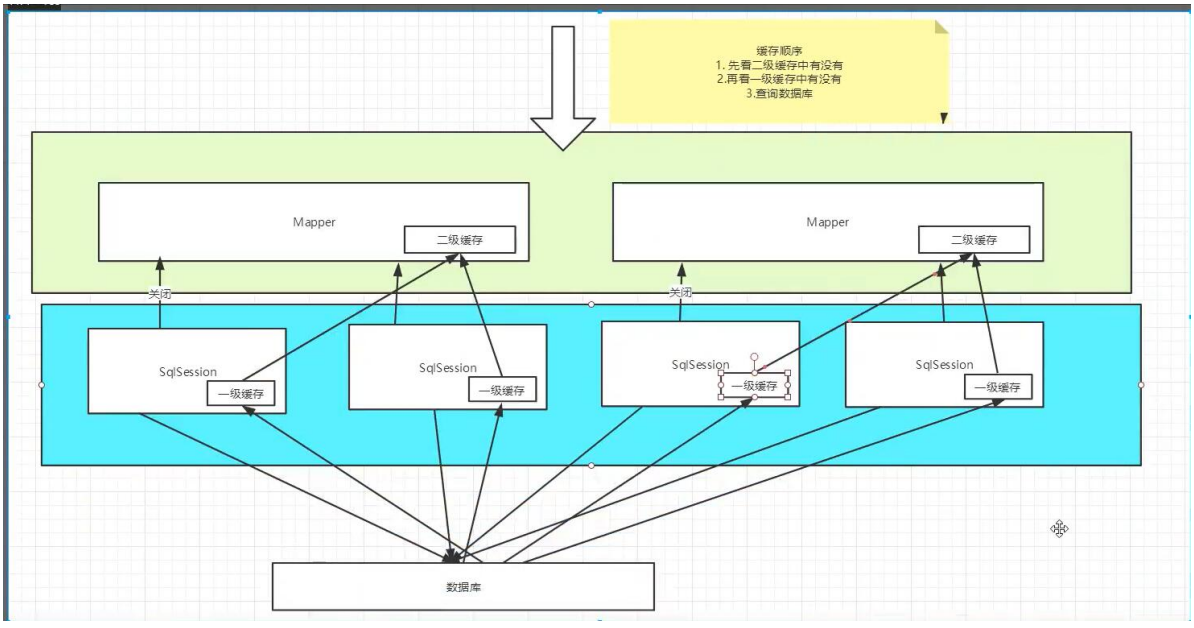
    1. 问题：我们需要将实体类序列化！否则就会报错！

    ```
    Cause: java.io.NotSerializableException: com.dongdong.pojo.User
    ```

小结：

- 只要开启了二级缓存，在同一个Mapper下就有效
- 所有的数据都会先放在一级缓存中；
- 只有当会话提交，或者关闭的时候，才会提交到二级缓存中！

13.5、缓存的原理



13.6、自定义缓存-ehcache

```
Ehcache是一种广泛使用的开源Java分布式缓存，主要面向用缓存
```

要使用程序，先导包！

```
<dependency>
          <groupId>org.mybatis.caches</groupId>
          <artifactId>mybatis-ehcache</artifactId>
          <version>1.1.0</version>
      </dependency>
```