

# Mybatis

## 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。
- MyBatis 本是apache的一个[开源项目](https://github.com/mybatis/mybatis-3)iBatis, 2010年这个[项目](https://github.com/mybatis/mybatis-3)由apache software foundation 迁移到了[google code](https://github.com/mybatis/mybatis-3)，并且改名为MyBatis。
- 2013年11月迁移到[Github](https://github.com/mybatis/mybatis-3)。

如何获得Mybatis

- Maven

```
1 <!-- https://mvnrepository.com/artifact/org.mybatis/mybatis -->
2 <dependency>
3     <groupId>org.mybatis</groupId>
4     <artifactId>mybatis</artifactId>
5     <version>3.5.6</version>
6 </dependency>
7
```

- 中文文档 <https://mybatis.org/mybatis-3/zh/index.html>
- GitHub <https://github.com/mybatis/mybatis-3>

### 1.2 持久化

数据持久化

- 持久化就是将程序的数据在持久化和瞬时状态转化的过程
- 内存：断电即失
- 数据库，io文件持久化
- 生活：冷藏，罐头

为什么需要持久化？

- 有一些对象，不能让他丢失
- 内存太贵了

### 1.3 持久层

- 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
- 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，支持对象关系组建维护
  - 提供xml标签，支持编写动态sql。

## 2、第一个Mybatis程序

思路：搭建环境-->导入Mybatis--->编程代码--->测试！

### 2.1 搭建环境

搭建数据库

```

1 CREATE DATABASE `mybatis`;
2 USE `mybatis`;
3
4 CREATE TABLE `user`(
5   `id` INT(20) NOT NULL PRIMARY KEY,
6   `name` VARCHAR(20) DEFAULT NULL,
7   `pwd` VARCHAR(30) DEFAULT NULL
8 )ENGINE=INNODB DEFAULT CHARSET utf8;
9
10 INSERT INTO `user`(`id`,`name`,`pwd`)VALUES
11 (1,'雅艺','1234'),
12 (2,'米亚','123'),
13 (3,'西西','1233')
14

```

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

```

1 <!--mysql驱动-->
2     <dependency>
3         <groupId>mysql</groupId>
4         <artifactId>mysql-connector-java</artifactId>
5         <version>8.0.22</version>
6     </dependency>
7
8     <!-- Mybatis -->
9     <dependency>
10         <groupId>org.mybatis</groupId>
11         <artifactId>mybatis</artifactId>
12         <version>3.5.6</version>
13     </dependency>

```

```

14      <!-- junit -->
15      <dependency>
16          <groupId>junit</groupId>
17          <artifactId>junit</artifactId>
18          <version>4.13</version>
19      </dependency>

```

## 2.2 创建一个模块

- 编写mybatis的核心配置文件

mybatis-config.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  <!--mybatis核心配置文件-->
6  <configuration>
7      <environments default="development">
8          <environment id="development">
9              <transactionManager type="JDBC"/>
10             <dataSource type="POOLED">
11                 <property name="driver"
12 value="com.mysql.cj.jdbc.Driver"/>
13                 <property name="url"
14 value="jdbc:mysql://localhost:3306/mybatis?
15 useSSL=true&useUnicode=true&characterEncoding=UTF-
16 8&serverTimezone=UTC"/>
17                 <property name="username" value="root"/>
18                 <property name="password" value="123"/>
19             </dataSource>
20         </environment>
21     </environments>
22     <!--每个Mapper.xml都需要在Mybatis核心配置文件中注册-->
23     <mappers>
24         <mapper resource="com/yzp/dao/UserMapper.xml"/>
25     </mappers>
26 </configuration>

```

- 1 XML 配置文件中包含了对 MyBatis 系统的核心设置，包括获取数据库连接实例的数据源（DataSource）以及决定事务作用域和控制方式的事务管理器（TransactionManager）
- 2 environment 元素体中包含了事务管理和连接池的配置。mappers 元素则包含了一组映射器（mapper），这些映射器的 XML 映射文件包含了 SQL 代码和映射定义信息

- 编写Mybatis工具类

```

1  public class MybatisUtils {
2      private static SqlSessionFactory sqlSessionFactory;
3      static {
4
5          try {
6              //使用mybatis第一步：获取SqlSessionFactory对象

```

```

7         String resource = "mybatis-config.xml";
8         InputStream inputStream =
Resources.getResourceAsStream(resource);
9         sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);
10        } catch (IOException e) {
11            e.printStackTrace();
12        }
13    }
14    //既然有了 SqlSessionFactory，顾名思义，我们可以从中获得 SqlSession 的
实例。
15    // SqlSession 提供了在数据库执行 SQL 命令所需的所有方法。
16    // 你可以通过 SqlSession 实例来直接执行已映射的 SQL 语句。
17
18    public static SqlSession getSqlSession(){
19        return sqlSessionFactory.openSession();
20    }
21
22 }

```

- 1 每个基于 MyBatis 的应用都是以一个 SqlSessionFactory 的实例为核心的。SqlSessionFactory 的实例可以通过 SqlSessionFactoryBuilder 获得。而 SqlSessionFactoryBuilder 则可以从 XML 配置文件或一个预先配置的 Configuration 实例来构建出 SqlSessionFactory 实例。
- 2
- 3 从 XML 文件中构建 SqlSessionFactory 的实例非常简单，建议使用类路径下的资源文件进行配置。
- 4 MyBatis 包含一个名叫 Resources 的工具类，它包含一些实用方法，使得从类路径或其它位置加载资源文件更加容易。

## 2.3 编写代码

### 1. 实体类

```

1 public class User {
2     private int id;
3     private String name;
4     private String pwd;
5     ....
6 }

```

### 2. Dao接口

```

1 public interface UserDao {
2     List<User> getUserList();
3 }

```

### 3. 接口实现类(由原来的UserDaoImpl转化为一个Mapper配置文件)

```

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 <!--namespace绑定一个对应得Dao/Mapper接口-->
6 <mapper namespace="com.yzp.dao.UserDao">
7     <!--select查询语句-->
8     <select id="getUserList" resultType="com.yzp.pojo.User">
9         select * from mybatis.user;
10    </select>
11 </mapper>

```

## 2.4 测试

```

1 @Test
2     public void test(){
3         //第一步：获得SqlSession对象
4         SqlSession sqlSession = MybatisUtils.getSqlSession();
5         //方式一： 执行SQL
6         UserDao mapper = sqlSession.getMapper(UserDao.class);
7         List<User> userList = mapper.getUserList();
8
9         for (User user:userList) {
10             System.out.println(user);
11         }
12
13         //关闭SqlSession
14         sqlSession.close();
15     }

```

注意点：

1. 在pom.xml文件中的build中配置resources，防止资源导出失败的问题

```

1 <build>
2     <resources>
3         <resource>
4             <directory>src/main/resources</directory>
5             <includes>
6                 <include>**/*.properties</include>
7                 <include>**/*.xml</include>
8             </includes>
9             <filtering>true</filtering>
10        </resource>
11        <resource>
12            <directory>src/main/java</directory>
13            <includes>
14                <include>**/*.properties</include>
15                <include>**/*.xml</include>
16            </includes>
17            <filtering>true</filtering>
18        </resource>
19    </resources>
20 </build>

```

2. 每个Mapper.xml都需要在Mybatis核心配置文件中注册
3. mybatis配置文件中需配置serverTimezone=UTC

## 3、CURD

### 1.namespace

namespace中的包名要和Dao/Mapper接口的包名一致!

### 2.select

选择, 查询语句

- id: 就是对应的namespace中的方法
- resultType: Sql语句执行的返回值
- parameterType: 参数类型

#### 1. 编写接口

```
1 //根据id查询
2 user getUserById(int id);
```

#### 2. 编写对应的mapper中的sql语句

```
1 <select id="getUserById" resultType="com.yzp.pojo.User"
2     parameterType="int">
3     select * from mybatis.user where id=#{id};
4 </select>
```

#### 3. 测试

```
1 //第一步: 获得SqlSession对象
2 SqlSession sqlSession = MybatisUtils.getSqlSession();
3 //方式一: 执行SQL
4 UserDao mapper = sqlSession.getMapper(UserDao.class);
5 User user = mapper.getUserById(2);
6
7
8 System.out.println(user);
9
10
11 //关闭SqlSession
12 sqlSession.close();
```

### 3.insert

```
1 <!--对象中的属性可以直接读出来-->
2 <insert id="adduser" parameterType="com.yzp.pojo.User">
3     insert into mybatis.user(id,name,pwd)values (#{id},#{name},#{pwd});
4 </insert>
```

### 4.Delete

```

1 <delete id="deleteUser" parameterType="int">
2     delete from mybatis.user where id=#{id}
3 </delete>

```

## 5.update

```

1 <update id="updateUser" parameterType="com.yzp.pojo.User">
2     update mybatis.user set name=#{name},pwd=#{pwd} where id=#{id};
3 </update>

```

注意点:

- 增删改需要提交事务! sqlSession.commit();

## 6.Map

如果, 我们的实体类或者数据库中的表, 字段或者参数过多, 我们可以考虑用Map

```

1 <insert id="addUser1" parameterType="map">
2     insert into mybatis.user(id,pwd)values (#{userId},#{pwd});
3 </insert>

```

```

1 @Test
2     public void update1Test(){
3         SqlSession sqlSession = MybatisUtils.getSqlSession();
4         UserDao mapper = sqlSession.getMapper(UserDao.class);
5         Map<String, Object> map = new HashMap<String, Object>();
6         map.put("userId",4);
7         map.put("pwd",33333);
8         mapper.addUser1(map);
9         sqlSession.commit();
10        sqlSession.close();
11    }

```

## 7.模糊查询

1. java代码执行的时候传递通配符%%

```

1 List<User> users = mapper.getUserLike("%米%");

```

```

1 <select id="getUserLike" resultType="com.yzp.pojo.User">
2     select * from mybatis.user where name like #{value};
3 </select>

```

2. 在sql拼接中使用通配符

```

1 <select id="getUserLike" resultType="com.yzp.pojo.User">
2     select * from mybatis.user where name like "%#{value}%"
3 </select>

```

```
1 | List<User> users = mapper.getUserLike("米");
```

## 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。配置文档的顶层结构如下：

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：尽管可以配置多个环境，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)

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

编写一个配置文件 db.properties

```
1 | driver=com.mysql.cj.jdbc.Driver
2 | url=jdbc:mysql://localhost:3306/mybatis?
  | useSSL=true&useUnicode=true&characterEncoding=UTF-8&serverTimezone=UTC
3 | username=root
4 | password=123
```

在核心配置文件中引入

```
1 | <properties resource="db.properties"/>
```



也可以在 properties 元素的子元素中设置：

```
1 <properties resource="db.properties">
2   <property name="username" value="root"/>
3   <property name="password" value="123"/>
4 </properties>
```

- 如果外部配置文件和 properties 元素的子元素有相同的字段，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。

```
1 <!--给实体类起别名-->
2 <typeAliases>
3   <typeAlias type="com.yzp.entity.User" alias="User"/>
4 </typeAliases>
```

也可以指定一个包名，MyBatis 会在包下面搜索需要的 Java Bean，比如：扫描实体类的包，它的默认别名就是这个类的类名，首字母小写

```
1 <typeAliases>
2   <package name="com.yzp.entity"/>
3 </typeAliases>
```

在实体类比较少的时候，使用第一种方式

如果实体类十分多，建议使用第二种

第一种可以DIY别名（自定义），第二种不行，如果非要改，需要在实体上增加注解

```
@Alias("user")
public class User {}
```

## 5. 设置

设置名	描述	有效值	默认值
cacheEnabled	全局性地开启或关闭所有映射器配置文件中已配置的任何缓存。	true   false	true
lazyLoadingEnabled	延迟加载的全局开关。当开启时，所有关联对象都会延迟加载。特定关联关系中可通过设置 <code>fetchType</code> 属性来覆盖该项的开关状态。	true   false	false
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

## 6. 其他配置

- [typeHandlers \(类型处理器\)](#)
- [objectFactory \(对象工厂\)](#)
- plugins (插件)
  - mybatis-generator-core
  - mybatis-plus

- 通用mapper

## 7.映射器 (mappers)

MapperRegistry:注册绑定我们的Mapper文件

方式一：使用相对于类路径的资源引用（推荐）

```
1 <mappers>
2     <mapper resource="com/yzp/mapper/UserMapper.xml"/>
3 </mappers>
```

方式二：使用class文件绑定注册

```
1 <mappers>
2     <mapper class="com.yzp.mapper.UserMapper"/>
3 </mappers>
```

注意：

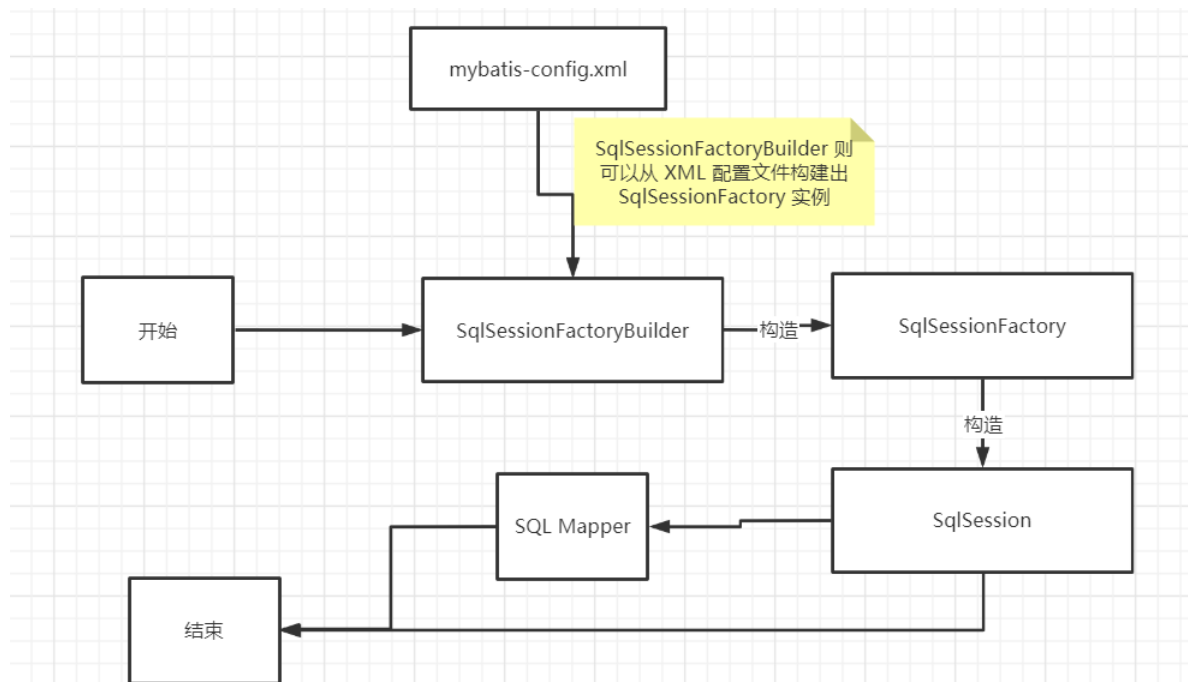
- 接口和它的Mapper配置文件必须同名
- 接口和他的Mapper配置文件必须在同一个包下

方式三：使用扫描包进行绑定注册(注意点同方式二)

```
1 <package name="com.yzp.mapper"/>
```

## 8.生命周期和作用域

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

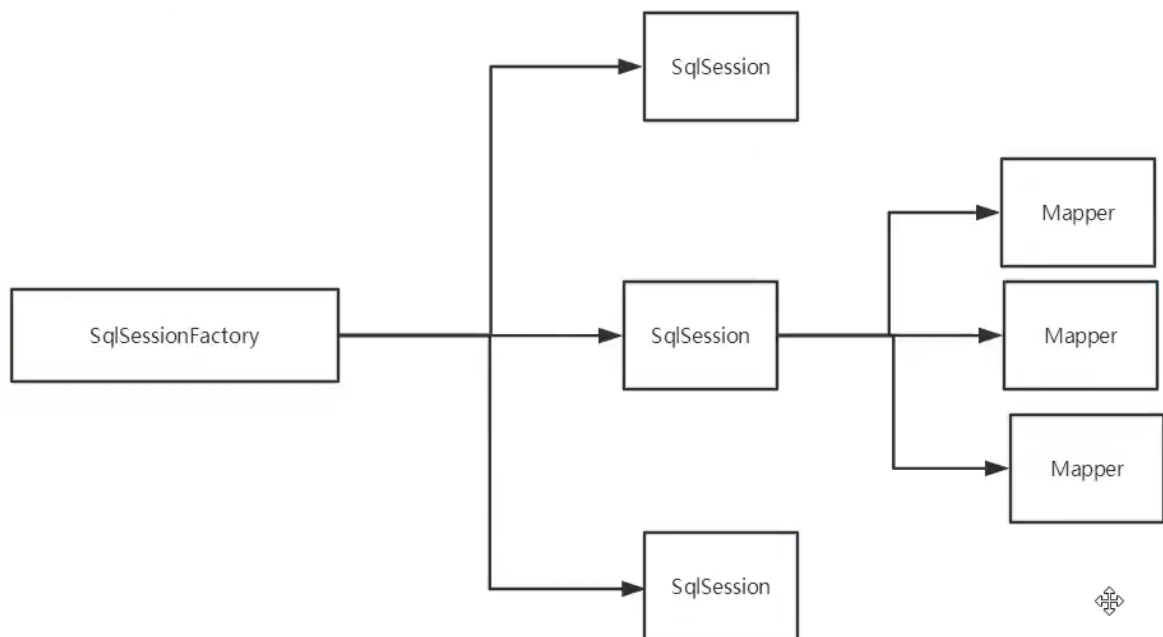
- 一旦创建了 `SqlSessionFactory`，就不再需要它了
- 最佳作用域是方法作用域（也就是局部方法变量）

### SqlSessionFactory

- 可以想象为：数据库连接池
- 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。**
- 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

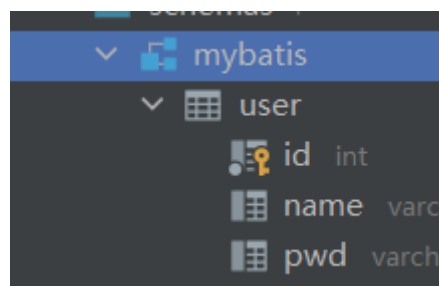


这里的每个Mapper，就代表每一个业务

## 5、解决属性名和字段名不一致的问题——resultMap(重点)

### 1.问题

数据库字段



类属性：

```
public class User {
    private int id;
    private String name;
    private String password;
}
```

测试结果：

```
D:\dev_soft\Java\jdk1.8.0_261\bin\java
User{id=2, name='米亚', pwd='null'}
```

```
1 select * from mybatis.user where id=#{id};
2 //以上通过类型处理器得到
3 select id,name,pwd from mybatis.user where id=#{id};
```

解决方法：

1. 起别名

```
1 select id,name,pwd as password from mybatis.user where id=#{id};
```

```
D:\dev_soft\Java\jdk1.8.0_261\bin\java
User{id=2, name='米亚', pwd='123'}
```

2. 使用resultMap

## 2.resultMap

结果集映射

```
1 <!--结果集映射-->
2 <resultMap id="UserMap" type="User">
3     <!--column数据库表中的字段 property实体类中的属性-->
4     <result column="id" property="id"/>
5     <result column="name" property="name"/>
6     <result column="pwd" property="password"/>
7 </resultMap>
8 <select id="getUserById" resultMap="UserMap" parameterType="int">
9     select * from mybatis.user where id=#{id};
10 </select>
```

- resultMap元素是Mybatis中最强大的元素
- 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。
- resultMap 的优秀之处——虽然你已经对它相当了解，但你完全可以不用显式地配置它们

```

1 <resultMap id="UserMap" type="User">
2     <!--column数据库表中的字段 property实体类中的属性-->
3     <!--<result column="id" property="id"/>
4     <result column="name" property="name"/>-->
5     <result column="pwd" property="password"/>
6 </resultMap>

```

语句中设置 `resultMap` 属性就行了（注意我们去掉了 `resultType` 属性）

## 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，日志是最好的助手！

曾经：sout\debug

现在：日志工厂

logImpl

指定 MyBatis 所用日志的具体实现，未指定时将自动查找。

SLF4J | LOG4J |

未设置

LOG4J2 |  
JDK\_LOGGING |  
COMMONS\_LOGGING  
| STDOUT\_LOGGING |  
NO\_LOGGING

- SLF4J
- LOG4J（掌握）
- LOG4J2
- JDK\_LOGGING
- COMMONS\_LOGGING
- STDOUT\_LOGGING（掌握）
- NO\_LOGGING

在Mybatis中具体使用哪一个由设置决定

**STDOUT\_LOGGING标准日志输出**

```

1 <!--在Mybatis核心配置文件中配置-->
2 <settings>
3     <setting name="logImpl" value="STDOUT_LOGGING"/>
4 </settings>

```

```

Opening JDBC Connection
Created connection 926434463.
Setting autocommit to false on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@3738449f]
==> Preparing: select * from mybatis.user where id=?;
==> Parameters: 2(Integer)
<==      Columns: id, name, pwd
<==      Row: 2, 米亚, 123
<==      Total: 1
User{id=2, name='米亚', pwd='123'}
Resetting autocommit to true on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@3738449f]
Closing JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@3738449f]
Returned connection 926434463 to pool.

```

### 6.2 LOG4J

## 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。
- 最令人感兴趣的就是，这些可以通过一个[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

```
1 <!-- https://mvnrepository.com/artifact/log4j/log4j -->
2 <dependency>
3   <groupId>log4j</groupId>
4   <artifactId>log4j</artifactId>
5   <version>1.2.17</version>
6 </dependency>
7
```

### 2. log4j.properties

```
1 #将等级为DEBUG的日志信息输出到console和file这个目的地，console和file的定义
  在下面的代码
2 log4j.rootLogger=DEBUG,console,file
3
4 #控制台输出的相关设置
5 log4j.appender.console = org.apache.log4j.ConsoleAppender
6 log4j.appender.console.Target = System.out
7 log4j.appender.console.Threshold=DEBUG
8 log4j.appender.console.layout = org.apache.log4j.PatternLayout
9 log4j.appender.console.layout.ConversionPattern=[%c]-%m%n
10
11 #文件输出的相关设置
12 log4j.appender.file = org.apache.log4j.RollingFileAppender
13 log4j.appender.file.File = ./log/you.log
14 log4j.appender.file.File.MaxFileSize = 10mb
15 log4j.appender.file.Threshold = DEBUG
16 log4j.appender.file.layout = org.apache.log4j.PatternLayout
17 log4j.appender.file.layout.ConversionPattern = [%p][%d{yy-MM-dd}]
  [%c]%m%n
18
19 #日志输出级别
20 log4j.logger.org.mybatis=DEBUG
21 log4j.logger.java.sql=DEBUG
22 log4j.logger.java.sql.Statement=DEBUG
23 log4j.logger.java.sql.ResultSet=DEBUG
24 log4j.logger.java.sql.PreparedStatement=DEBUG
```

### 3. 配置LOG4J 为日志的实现

```
1 <settings>
2   <setting name="logImpl" value="LOG4J"/>
3 </settings>
```

### 4. 测试

```
[org.apache.ibatis.transaction.jdbc.JdbcTransaction]-Opening JDBC Connection
[org.apache.ibatis.datasource.pooled.PooledDataSource]-Created connection 854487022.
[org.apache.ibatis.transaction.jdbc.JdbcTransaction]-Setting autocommit to false on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@32ee6fee]
[com.yzp.mapper.UserMapper.getUserById]==> Preparing: select * from mybatis.user where id=?;
[com.yzp.mapper.UserMapper.getUserById]==> Parameters: 2(Integer)
[com.yzp.mapper.UserMapper.getUserById]<== Total: 1
User{id=2, name='米亚', pwd='123'}
[org.apache.ibatis.transaction.jdbc.JdbcTransaction]-Resetting autocommit to true on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@32ee6fee]
[org.apache.ibatis.transaction.jdbc.JdbcTransaction]-Closing JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@32ee6fee]
[org.apache.ibatis.datasource.pooled.PooledDataSource]-Returned connection 854487022 to pool.
```

## 简单使用

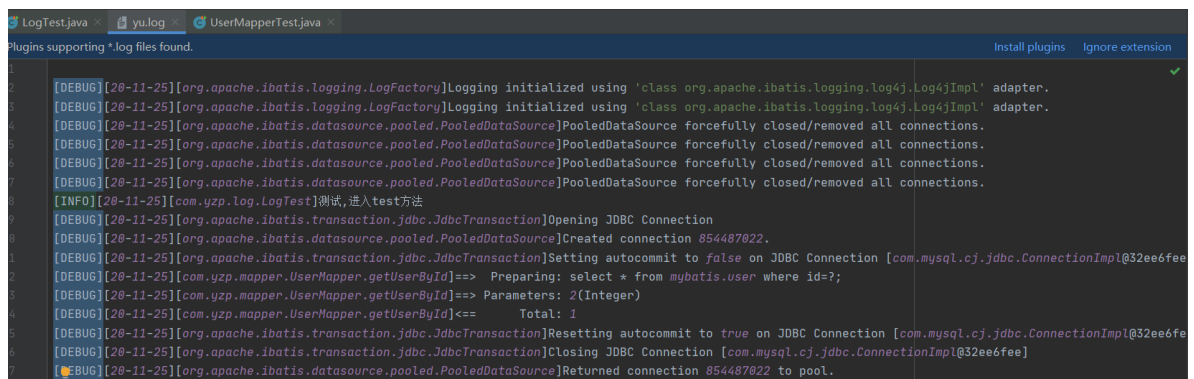
1. 日志对象 参数为当前类的class

```
1 static Logger logger = Logger.getLogger(LogTest.class);
```

2. 日志级别

```
1 logger.info("info: 进入到了Log4jTest方法");
2     logger.debug("debug: 进入到了Log4jTest方法");
3     logger.error("error: 进入到了Log4jTest方法");
```

```
1 public class LogTest {
2     static Logger logger = Logger.getLogger(LogTest.class);
3     @Test
4     public void Log4jTest(){
5         logger.info("info: 进入到了Log4jTest方法");
6         logger.debug("debug: 进入到了Log4jTest方法");
7         logger.error("error: 进入到了Log4jTest方法");
8     }
9
10    @Test
11    public void test(){
12        //第一步: 获得SqlSession对象
13        SqlSession sqlSession = MybatisUtils.getSqlSession();
14        logger.info("测试, 进入test方法");
15        //方式一: 执行SQL
16        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
17        User user = mapper.getUserById(2);
18        System.out.println(user);
19        //关闭SqlSession
20        sqlSession.close();
21    }
22 }
```



```
LogTest.java x yu.log x UserMapperTest.java x
Plugins supporting *.log files found. Install plugins Ignore extension
1 [DEBUG][20-11-25][org.apache.ibatis.logging.LogFactory]Logging initialized using 'class org.apache.ibatis.logging.log4j.Log4jImpl' adapter.
2 [DEBUG][20-11-25][org.apache.ibatis.logging.LogFactory]Logging initialized using 'class org.apache.ibatis.logging.log4j.Log4jImpl' adapter.
3 [DEBUG][20-11-25][org.apache.ibatis.datasource.pooled.PooledDataSource]PooledDataSource forcefully closed/removed all connections.
4 [DEBUG][20-11-25][org.apache.ibatis.datasource.pooled.PooledDataSource]PooledDataSource forcefully closed/removed all connections.
5 [DEBUG][20-11-25][org.apache.ibatis.datasource.pooled.PooledDataSource]PooledDataSource forcefully closed/removed all connections.
6 [DEBUG][20-11-25][org.apache.ibatis.datasource.pooled.PooledDataSource]PooledDataSource forcefully closed/removed all connections.
7 [INFO][20-11-25][com.yzp.log.LogTest]测试, 进入test方法
8 [DEBUG][20-11-25][org.apache.ibatis.transaction.jdbc.JdbcTransaction]Opening JDBC Connection
9 [DEBUG][20-11-25][org.apache.ibatis.datasource.pooled.PooledDataSource]Created connection 854487022.
10 [DEBUG][20-11-25][org.apache.ibatis.transaction.jdbc.JdbcTransaction]Setting autocommit to false on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@32ee6fee]
11 [DEBUG][20-11-25][com.yzp.mapper.UserMapper.getUserById]==> Preparing: select * from mybatis.user where id=?;
12 [DEBUG][20-11-25][com.yzp.mapper.UserMapper.getUserById]==> Parameters: 2(Integer)
13 [DEBUG][20-11-25][com.yzp.mapper.UserMapper.getUserById]<== Total: 1
14 [DEBUG][20-11-25][org.apache.ibatis.transaction.jdbc.JdbcTransaction]Resetting autocommit to true on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@32ee6fee]
15 [DEBUG][20-11-25][org.apache.ibatis.transaction.jdbc.JdbcTransaction]Closing JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@32ee6fee]
16 [DEBUG][20-11-25][org.apache.ibatis.datasource.pooled.PooledDataSource]Returned connection 854487022 to pool.
```

## 7、分页

思考：为什么要分页？

- 减少数据的处理量

### 7.1 使用Limit分页（重点）

```
1 语法: select * from user limit startIndex,pageSize;  
2  select * from user limit 3;#[0,3]
```

使用Mybatis实现分页，核心SQL

#### 1. 接口

```
1  //分页查询  
2  List<User> getUserByLimit(Map<String,Integer> map);
```

#### 2. Mapper.xml

```
1  <select id="getUserByLimit" resultMap="UserMap" parameterType="map">  
2      select * from mybatis.user limit #{startIndex},#{pageSize}  
3  </select>
```

#### 3. 测试

```
1  @Test  
2      public void getUserByLimitTest(){  
3          SqlSession sqlSession = MybatisUtils.getSqlSession();  
4          UserMapper mapper = sqlSession.getMapper(UserMapper.class);  
5          Map<String,Integer> map = new HashMap<String, Integer>();  
6          map.put("startIndex",0);  
7          map.put("pageSize",2);  
8          List<User> users = mapper.getUserByLimit(map);  
9          for (User user:users  
10              ) {  
11              System.out.println(user);  
12          }  
13      }
```

### 7.2 RowBounds分页（了解）

#### 1. 接口

```
1  //RowBounds分页  
2  List<User> getUserByRowBounds();
```

#### 2. Mapper.xml



```
1 <select id="getUserByRowBounds" resultMap="UserMap"
  parameterType="map">
2     select * from mybatis.user
3 </select>
```

### 3. 测试

```
1 @Test
2     public void getUserByRowBoundsTest(){
3         SqlSession sqlSession = MybatisUtils.getSqlSession();
4         //RowBounds实现分页
5         RowBounds rowBounds = new RowBounds(1, 2);
6         List<User> userList =
7         sqlSession.selectList("com.yzp.mapper.UserMapper.getUserByRowBounds", null, rowBounds);
8         for (User user:userList) {
9             System.out.println(user);
10        }
11        sqlSession.close();
12    }
```

## 7.3分页插件

(了解)

# MyBatis 分页插件 PageHelper

如果你也在用 MyBatis，建议尝试该分页插件，这一定是最方便使用的分页插件。分页插件支持任何复杂的单表、多表分页。

[View on Github](#)[View on GitOsc](#)

## 8、使用注解开发

### 8.1 面向接口编程

我们学过面向对象编程,也学过接口,但在真正的开发中,很多时候我们会选择面向接口编程。

**根本原因:**解耦,可扩展,提供复用,分层开发中,上层不用管具体的实现,大家都遵循共同的标准使得开发变得容易,规范性好

在一个面向对象的系统中,系统的各种功能是由许许多多的不同对象协作完成的.在这种情况下,各个对象内部是如何实现自己的,对系统设计人员来讲就不那么重要了;

各个对象之间的协作关系则成为系统设计关键.小到不同类之间的通信,大到各个模块之间的交互,在系统设计之初都是要着重考虑的,这也是系统设计的主要工作内容.面向接口编程就是按照这种思想来编程。

## 关于接口的理解

- 接口从更深层次的理解,应是定义(规范,约束)与实现的分离
- 接口本身反映了系统设计人员对系统的抽象理解
- 接口有两类
  - 第一类是对一个个体的抽象,它对应为一个抽象体(abstract class)
  - 第二类是对一个个体某个方面的抽象,即形成一个抽象面 (interface)
- 一个个体有可能有对个抽象面,抽象体与抽象面是有区别的

## 三个面向区别

- 面向对象是指,我们考虑问题时,以对象为单位,考虑它的属性及方法
- 面向过程是指,我们考虑问题时,以一个具体的流程(事务过程)为单位,考虑它的实现
- 接口设计与非接口设计是针对复用技术而言的,与面向对象(过程)不是一个问题,更多体现的是对系统整体的架构

## 8.2 使用注解开发

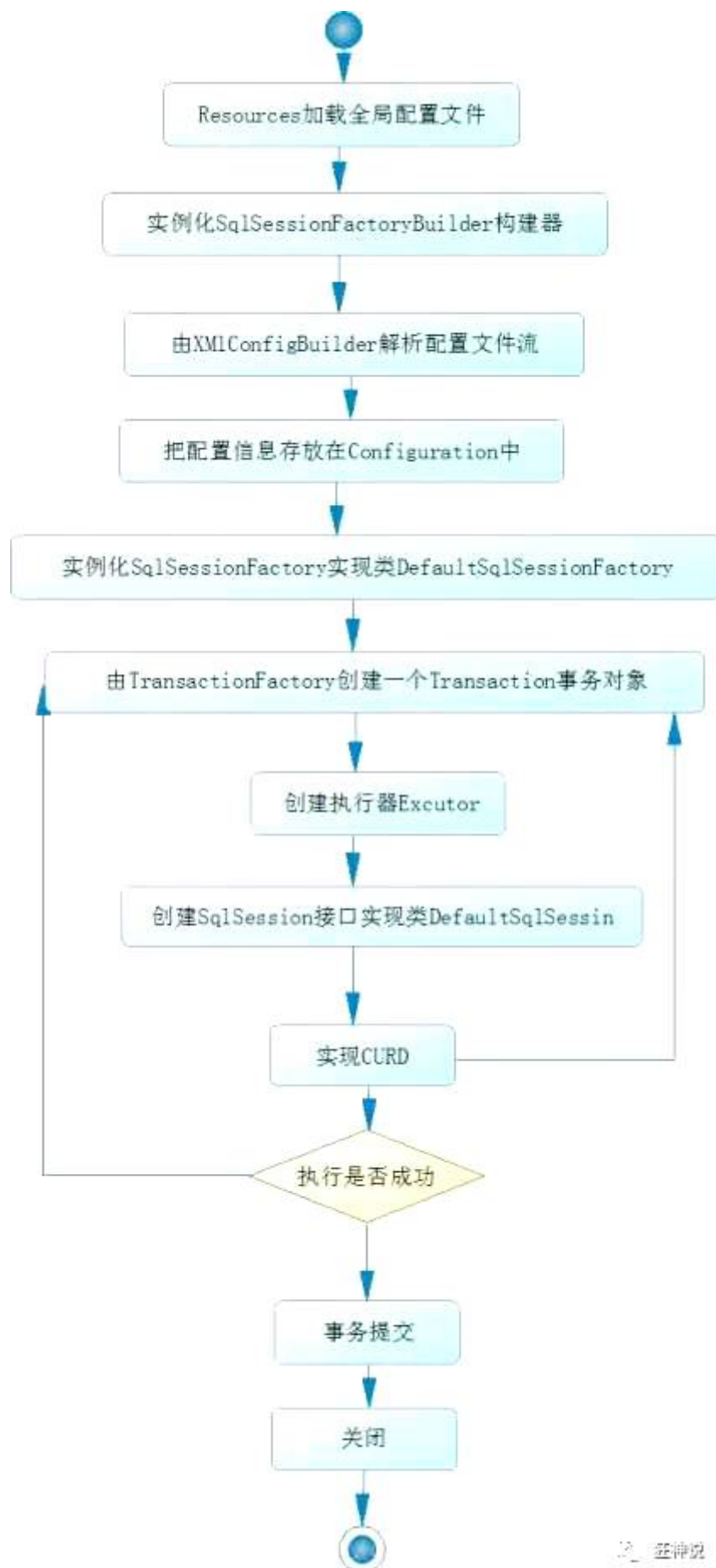
### 1. 注解直接在接口上实现

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

### 2. 需要在核心配置文件中绑定接口

```
1 | <mappers>
2 |     <mapper class="com.yzp.mapper.UserMapper"/>
3 | </mappers>
```

## mybatis详细执行过程



狂神说

## 8.3 CRUD

1. 编写接口
2. 绑定接口

```

1 <!-- 绑定接口-->
2 <mappers>
3     <mapper class="com.yzp.mapper.UserMapper"/>
4 </mappers>

```

## 查询

```

1 // 有多个参数时必须加上@param
2 @Select("select * from user where id = #{id}")
3 User getUserById(@Param("id") int id);

```

## 增加

```

1 @Insert("insert into user(id,name,pwd)values(#{id},#{name},#{password})")
2 int addUser(User user);

```

```

1 public static SqlSession getSqlSession(){
2
3     return sessionFactory.openSession(true); //true自动提交事务
4 }

```

## 修改

```

1 @Update("update user set name=#{name},pwd=#{password} where id=#{id}")
2 int updateUser(User user);

```

## 删除

```

1 @Delete("delete from user where id=#{id}")
2 int deleteUser(@Param("id") int id);

```

## @Param()注解

- 基本类型的参数或者String类型，需要加上
- 应用类型不需要加
- 如果只有一个基本类型可以加也可以不加
- 我们在SQL中引用的（#{id}）就是@Param("id")中设定的属性名

#{ } 可以防止sql注入

\${ } 不可以防止sql注入

## 9、Lombok

```

1 Features
2 @Getter and @Setter
3 @FieldNameConstants
4 @ToString
5 @EqualsAndHashCode

```

```
6 @AllArgsConstructor, @RequiredArgsConstructor and @NoArgsConstructor
7 @Log, @Log4j, @Log4j2, @Slf4j, @XSlf4j, @CommonsLog, @JBossLog, @Flogger,
  @CustomLog
8 @Data
9 @Builder
10 @SuperBuilder
11 @Singular
12 @Delegate
13 @Value
14 @Accessors
15 @With
16 @With
17 @SneakyThrows
18 @val
19 @var
20 experimental @var
21 @UtilityClass
22 Lombok config system
23 Code inspections
24 Refactoring actions (lombok and delombok)
```

### 使用步骤:

1. 在idea中下载Lombok
2. 在项目导入lombok依赖

```
1 <dependency>
2   <groupId>org.projectlombok</groupId>
3   <artifactId>lombok</artifactId>
4   <version>1.18.12</version>
5 </dependency>
6
```

3. 在实体类上加注解

@Data :无参构造, get, set, toString, hashCode, equals方法

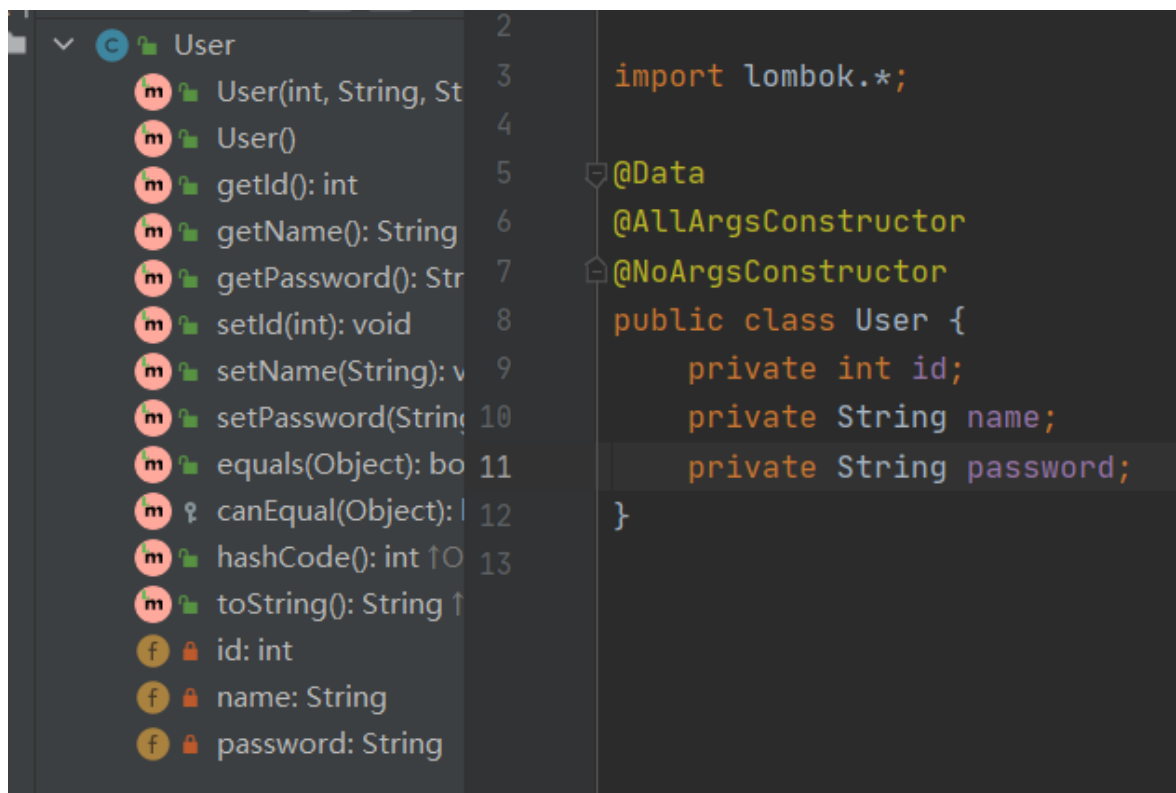
@AllArgsConstructor

@NoArgsConstructor

@EqualsAndHashCode

@ToString

@Getter and @Setter



## 10、多对一处理

- 多个学生，对应一个老师
- 对于学生这边而言，关联，多个学生，**关联**一个老师【多对一】
- 对于老师而言，集合，一个老师，有很多学生【一对多】

SQL

```

1 CREATE TABLE `teacher`(
2   `id` INT(10) NOT NULL,
3   `name` VARCHAR(30) DEFAULT NULL,
4   PRIMARY KEY(`id`)
5 )ENGINE=INNODB DEFAULT CHARSET=utf8;
6
7 INSERT INTO teacher(`id`,`name`)VALUES(1,'游老师')
8
9 CREATE TABLE `student`(
10  `id` INT(10) NOT NULL,
11  `name` VARCHAR(30) DEFAULT NULL,
12  `tid` INT(10) DEFAULT NULL,
13  PRIMARY KEY(`id`),
14  CONSTRAINT `fk tid` FOREIGN KEY(`tid`) REFERENCES `teacher`(`id`)
15 )ENGINE=INNODB DEFAULT CHARSET=utf8;;
16
17 INSERT INTO `student` (`id`,`name`,`tid`)VALUES(1,'小张',1)
18 INSERT INTO `student` (`id`,`name`,`tid`)VALUES(2,'小李',1)
19 INSERT INTO `student` (`id`,`name`,`tid`)VALUES(3,'小云',1)
20 INSERT INTO `student` (`id`,`name`,`tid`)VALUES(4,'小华',1)

```

实体类：

```

1  @Data
2  public class Student {
3      private int id;
4      private String name;
5      //学生需要关联一个老师
6      private Teacher teacher;
7  }
8
9  @Data
10 public class Teacher {
11     private int id;
12     private String name;
13 }

```

业务需求：查询所有的学生信息，以及对应的老师信息

首次尝试：

首先在mybatis核心配置文件中绑定接口或配置文件

```

1  <mappers>
2      <mapper class="com.yzp.mapper.TeacherMapper"/>
3      <mapper class="com.yzp.mapper.StudentMapper"/>
4  </mappers>

```

### 1. 编写接口

```

1  public List<Student> getStudent();

```

### 2. 编写Mapper.xml

```

1  <mapper namespace="com.yzp.mapper.StudentMapper">
2      <select id="getStudent" resultType="Student">
3          select * from student s,teacher t where s.tid = t.id;
4      </select>
5  </mapper>

```

### 3. 测试

```

1  @Test
2      public void getStudentTest(){
3          SqlSession sqlSession = MybatisUtils.getSqlSession();
4          StudentMapper mapper =
5              sqlSession.getMapper(StudentMapper.class);
6          List<Student> studentList = mapper.getStudent();
7          for (Student stu:studentList) {
8              System.out.println(stu);
9          }
10         sqlSession.close();

```

测试结果：

```
Student(id=1, name=小张, teacher=null)
Student(id=2, name=小李, teacher=null)
Student(id=3, name=小云, teacher=null)
Student(id=4, name=小华, teacher=null)
```

teacher为null

解决方法：

按照查询嵌套处理

修改StudentMapper.xml为

```
1 <mapper namespace="com.yzp.mapper.StudentMapper">
2 <!-- 思路:
3     1. 查询所有学生的信息
4     2. 根据查询出来的学生的tid, 寻找对应得老师-->
5     <select id="getStudent" resultMap="StudentTeacher">
6         select * from student
7     </select>
8     <resultMap id="StudentTeacher" type="Student">
9 <!-- 复杂的属性我们需要单独处理 对象:association 集合:collection-->
10     <association property="teacher" column="tid" javaType="Teacher"
11     select="getTeacher"/>
12     </resultMap>
13     <select id="getTeacher" resultType="Teacher">
14         select * from teacher where id =#{id}
15     </select>
16 </mapper>
```

测试结果：

```
Student(id=1, name=小张, teacher=Teacher(id=1, name=游老师))
Student(id=2, name=小李, teacher=Teacher(id=1, name=游老师))
Student(id=3, name=小云, teacher=Teacher(id=1, name=游老师))
Student(id=4, name=小华, teacher=Teacher(id=1, name=游老师))
```

按照结果嵌套处理

修改StudentMapper.xml为

```
1 <!--按照结果嵌套处理-->
2 <select id="getStudent2" resultMap="StudentTeacher2">
3     select s.id sid,s.name sname,t.name tname from student s,teacher t
4     where s.tid = t.id
5 </select>
6 <resultMap id="StudentTeacher2" type="Student">
7     <result property="id" column="sid"/>
8     <result property="name" column="sname"/>
9     <association property="teacher" javaType="Teacher">
10         <result property="name" column="tname"/>
11     </association>
12 </resultMap>
```



## 11、一对多处理

比如：一个老师拥有多个学生

对于老师而言就一对多

实体类：

```
1  @Data
2  public class Student {
3      private int id;
4      private String name;
5      private int tid;
6  }
7
8  @Data
9  public class Teacher {
10     private int id;
11     private String name;
12     //一个老师拥有多个学生
13     private List<Student> students;
14 }
```

1 | 获取指定老师下的所有学生及老师的信息

### 1. 接口

```
1 | List<Teacher> getTeacher(@Param("tid") int id);
```

### 2. Mapper.xml

#### 方式一

```
1  <!-- 按结果嵌套查询-->
2  <select id="getTeacher" resultMap="TeacherStudent">
3      select t.id tid,t.name tname,s.id sid,s.name sname
4      from teacher t,student s
5      where t.id = s.tid and t.id=#{tid};
6  </select>
7  <resultMap id="TeacherStudent" type="Teacher">
8      <result property="id" column="tid"/>
9      <result property="name" column="tname"/>
10     <collection property="students" ofType="Student">
11         <result property="id" column="sid"/>
12         <result property="name" column="sname"/>
13         <result property="tid" column="tid"/>
14     </collection>
15
16 </resultMap>
```

#### 方式二

```

1  <!--按查询嵌套处理-->
2  <select id="getTeacher2" resultMap="TeacherStudent2">
3      select * from teacher where id =#{tid}
4  </select>
5  <resultMap id="TeacherStudent2" type="Teacher">
6      <collection property="students" javaType="ArrayList"
ofType="Student"
7          select="getStudentByTeacherId" column="id"/>
8  </resultMap>
9  <select id="getStudentByTeacherId" resultType="Student">
10     select * from student where tid =#{tid}
11 </select>

```

### 3. 测试结果

```

==> Parameters: 1(Integer)
<==      Columns: tid, tname, sid, sname
<==      Row: 1, 游老师, 1, 小张
<==      Row: 1, 游老师, 2, 小李
<==      Row: 1, 游老师, 3, 小云
<==      Row: 1, 游老师, 4, 小华
<==      Total: 4
Teacher(id=1, name=游老师, students=[Student(id=1, name=小张, tid=1), Student
Closing JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@6babf3bf]

```

### 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，属性名和字段的问题
- 如果问题不好排查，可以使用日志，建议使用LOG4j

### 面试高频

- Mysql引擎
- InnoDB底层原理
- 索引
- 索引优化

## 12、动态SQL (重点)

### 什么是动态SQL

动态SQL就是根据不同条件生成不同的SQL语句

其本质还是SQL语句，只是我们可以在SQL层面，去执行逻辑代码

- if (重点)
- choose (when, otherwise)
- trim (where, set)
- foreach

## where

*where* 元素只会在子元素返回任何内容的前提下才插入“WHERE”子句。而且，若子句的开头为“AND”或“OR”，*where* 元素也会将它们去除。

## if

希望通过“title”和“author”两个参数进行可选搜索该怎么办呢

```
1 <select id="queryBlogIf" parameterType="map" resultType="blog">
2     select * from blog
3     <where>
4         <if test="title !=null">
5             title=#{title}
6         </if>
7         <if test="author !=null">
8             and author = #{author}
9         </if>
10    </where>
11 </select>
```

## choose

```
1 <select id="queryBlogChoose" parameterType="map" resultType="blog">
2     select * from blog
3     <where>
4         <choose>
5             <when test="title !=null">
6                 title = #{title}
7             </when>
8             <when test="author !=null">
9                 author = #{author}
10            </when>
11            <otherwise>
12                views = #{views}
13            </otherwise>
14        </choose>
15    </where>
16 </select>
```

相当于switch case

## set

```

1 <update id="updateBlogSet" parameterType="map">
2     update blog
3     <set>
4         <if test="title !=null">
5             title =#{title},
6         </if>
7         <if test="author !=null">
8             author =#{author}
9         </if>
10    </set>
11    where id = #{id}
12 </update>

```

## SQL片段

```

1 <sql id="if-title-author">
2     <if test="title !=null">
3         title =#{title},
4     </if>
5     <if test="author !=null">
6         author =#{author}
7     </if>
8 </sql>
9 <!--在需要使用sql片段是使用include标签引用即可-->
10 <update id="updateBlogSet" parameterType="map">
11     update blog
12     <set>
13         <include refid="if-title-author"></include>
14     </set>
15     where id = #{id}
16 </update>

```

注意事项：

- 最好基于单表来定义SQL片段（一般是if）
- 不要存在where标签

## foreach

```

1 <select id="queryBlogForeach" parameterType="map" resultType="blog">
2     select * from blog
3     <where>
4         id in
5         <foreach collection="ids" item="id" open="(" close=")"
separator="," >
6             #{id}
7         </foreach>
8     </where>
9 </select>

```

测试：

```

1  @Test
2      public void test1(){
3          SqlSession sqlSession = MybatisUtils.getSqlSession();
4          BlogMapper mapper = sqlSession.getMapper(BlogMapper.class);
5          Map<String, Object> map = new HashMap<String, Object>();
6          ArrayList<Integer> ids = new ArrayList<Integer>();
7          ids.add(1);
8          ids.add(3);
9          ids.add(4);
10         map.put("ids",ids);
11         mapper.queryBlogForeach(map);
12         sqlSession.close();
13     }

```

## 13、缓存（了解）

### 13.1 缓存简介

#### 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了从磁盘上（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

### 13.2 Mybatis缓存

- Mybatis包含一个非常强大的查询缓存特性，他可以非常方便地定制和配置缓存。缓存可以极大地提升查询效率。
- Mybatis系统默认定义了两级缓存：**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启和配置，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接口来定义二级缓存

### 13.3 一级缓存

- 一级缓存也叫本地缓存：SqlSession
  - 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，不用再去查询数据库

测试：

1. 开启日志
2. 测试

```

1  @Test
2      public void test(){

```

```

3      SqlSession sqlSession =
MybatisUtils.getSqlSession();
4      UserMapper mapper =
sqlSession.getMapper(UserMapper.class);
5      User user = mapper.queryUserById(1);
6      System.out.println(user);
7
8      System.out.println("=====");
9      User user2 = mapper.queryUserById(1);
10     System.out.println(user2);
11
12     System.out.println(user==user2);
13
14     sqlSession.close();
15 }

```

查看日志:

```

Opening JDBC Connection
Created connection 802243390.
==> Preparing: select * from user where id =?
==> Parameters: 1(Integer)
<==      Columns: id, name, pwd
<==      Row: 1, 雅艺, 1234
<==      Total: 1
User(id=1, name=雅艺, pwd=1234)
=====
User(id=1, name=雅艺, pwd=1234)
true
Closing JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@2fd1433e]
Returned connection 802243390 to pool.

```

从日志中可以看出第二次查询没有生成sql, 直接查出来了。并且两次查询的user是同一个

## 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

```

1  @Test
2  public void test(){
3      SqlSession sqlSession = MybatisUtils.getSqlSession();
4      UserMapper mapper = sqlSession.getMapper(UserMapper.class);
5      User user = mapper.queryUserById(1);
6      System.out.println(user);
7
8      System.out.println("=====");
9      mapper.updateUser(new User(2, "思妮", "yyy"));
10     System.out.println("=====");
11
12     User user2 = mapper.queryUserById(1);
13     System.out.println(user2);
14
15     system.out.println(user==user2);
16

```

```
17 |         sqlSession.close();
18 |     }
```

```
S ==> Preparing: select * from user where id =?
==> Parameters: 1(Integer)
<== Columns: id, name, pwd
<== Row: 1, 雅艺, 1234
<== Total: 1
User(id=1, name=雅艺, pwd=1234)
=====
==> Preparing: update user set name=?,pwd=? where id=?;
==> Parameters: 思妮(String), yyy(String), 2(Integer)
<== Updates: 1
=====
==> Preparing: select * from user where id =?
==> Parameters: 1(Integer)
<== Columns: id, name, pwd
<== Row: 1, 雅艺, 1234
<== Total: 1
User(id=1, name=雅艺, pwd=1234)
false
Closing JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@2fd1433e]
Returned connection 802243390 to pool.
```

```
1 | @Test
2 |     public void test(){
3 |         sqlSession = MybatisUtils.getSqlSession();
4 |         UserMapper mapper = sqlSession.getMapper(UserMapper.class);
5 |         User user = mapper.queryUserById(1);
6 |         System.out.println(user);
7 |
8 |         System.out.println("=====");
9 |         sqlSession.clearCache();//清理缓存
10 |
11 |         User user2 = mapper.queryUserById(1);
12 |         System.out.println(user2);
13 |
14 |         System.out.println(user==user2);
15 |
16 |         sqlSession.close();
17 |     }
```

```

Opening JDBC Connection
Created connection 802243390.
==> Preparing: select * from user where id =?
==> Parameters: 1(Integer)
<==      Columns: id, name, pwd
<==      Row: 1, 雅艺, 1234
<==      Total: 1
User(id=1, name=雅艺, pwd=1234)
=====
==> Preparing: select * from user where id =?
==> Parameters: 1(Integer)
<==      Columns: id, name, pwd
<==      Row: 1, 雅艺, 1234
<==      Total: 1
User(id=1, name=雅艺, pwd=1234)
false
Closing JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@2fd1433e]
Returned connection 802243390 to pool.

```

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这个连接

一级缓存就是一个Map

## 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，一个命名空间，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，这个会话对应的一级缓存就没了；但是我们想要的是，会话关闭了，一级缓存中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
  - 不同的mapper查出的数据会放在自己对应的缓存（map）中

二级缓存需要手动开启和配置，他是基于namespace级别的缓存。

### 设置 (settings)

这是 MyBatis 中极为重要的调整设置，它们会改变 MyBatis 的运行时行为。下表描述了设置中各项设置的含义、默认值等。

设置名	描述	有效值	默认值
cacheEnabled	全局性地开启或关闭所有映射器配置文件中已配置的任何缓存。	true   false	true

使用步骤:

1. 开启全局缓存

```
1 | <setting name="cacheEnabled" value="true"/>
```

2. 在要使用二级缓存的Mapper中开启



```

1 <mapper namespace="com.yzp.dao.UserMapper">
2 <!-- 在当前mapper中使用二级缓存-->
3     <cache />
4 </mapper>

```

也可以自定义参数

```

1 <mapper namespace="com.yzp.dao.UserMapper">
2 <!-- 在当前mapper中使用二级缓存-->
3     <cache eviction="FIFO"
4         flushInterval="60000"
5         size="512"
6         readOnly="true"/>
7 </mapper>

```

### 3. 测试

```

1 @Test
2     public void test(){
3         SqlSession sqlSession = MybatisUtils.getSqlSession();
4         UserMapper mapper = sqlSession.getMapper(UserMapper.class);
5         User user = mapper.queryUserById(1);
6         System.out.println(user);
7         sqlSession.close();
8         System.out.println("=====");
9
10        SqlSession sqlSession2 = MybatisUtils.getSqlSession();
11        UserMapper mapper2 =
sqlSession2.getMapper(UserMapper.class);
12        User user2 = mapper2.queryUserById(1);
13        System.out.println(user2);
14
15        System.out.println(user==user2);
16
17        sqlSession2.close();
18    }

```

出错:

```
org.apache.ibatis.cache.CacheException: Error serializing object. Cause: java.io.NotSerializableException: com.yzp.entity.User
```

需将实体类User序列化

修改后，测试结果

```

Cache Hit Ratio [com.yzp.dao.UserMapper]: 0.0
Opening JDBC Connection
Created connection 726181440.
==> Preparing: select * from user where id =?
==> Parameters: 1(Integer)
<== Columns: id, name, pwd
<== Row: 1, 雅艺, 1234
<== Total: 1
User(id=1, name=雅艺, pwd=1234)
Closing JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@2b48a640]
Returned connection 726181440 to pool.

=====
As you are using functionality that deserializes object streams, it is recommended
Cache Hit Ratio [com.yzp.dao.UserMapper]: 0.5
User(id=1, name=雅艺, pwd=1234)
false

```

可以在mapper的某个查询中手动关闭或开启二级缓存

```

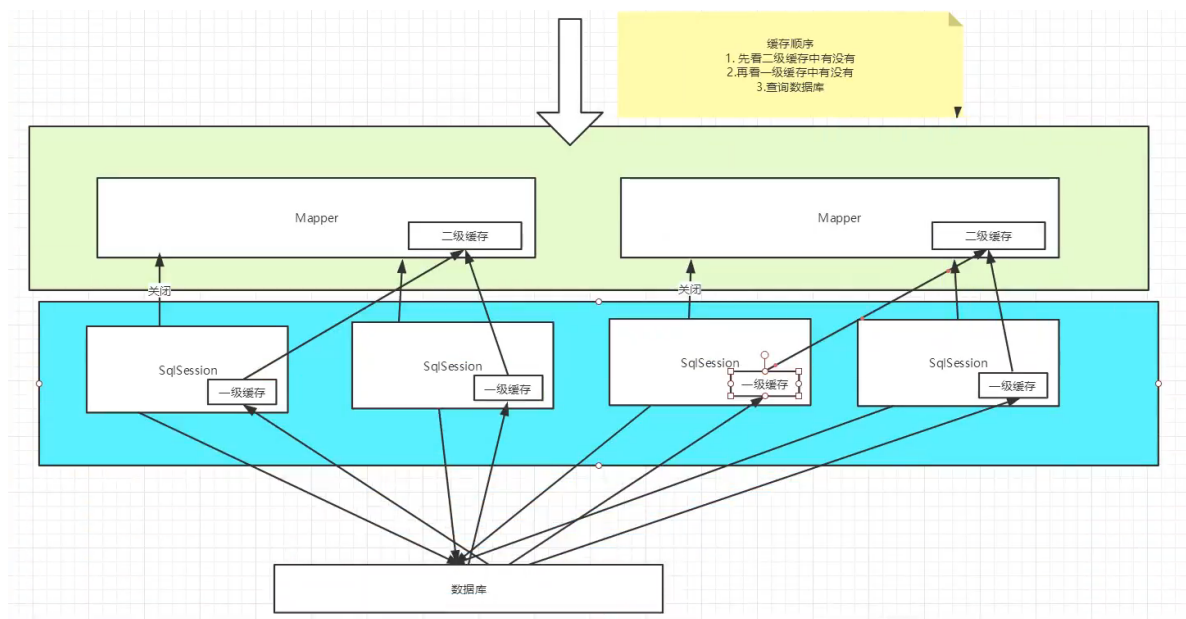
<mapper namespace="com.yzp.dao.UserMapper">
  <!-- 在当前mapper中使用二级缓存-->
  <cache/>
  <select id="queryUserById" resultType="user" useCache="true">
    select * from user where id =#{id}
  </select>

```

小结:

- 只要开启了二级缓存，同一个Mapper下就有效
- 所有数据都会先放在一级缓存中，只有当会话提交或关闭的时候，才会转存到二级缓存中

## 13.5 缓存原理



## 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。主要面向通用缓存

1. 在程序中使用ehcache,先要导包

```

1 <!-- https://mvnrepository.com/artifact/org.mybatis/mybatis-ehcache -->
2 <dependency>
3     <groupId>org.mybatis</groupId>
4     <artifactId>mybatis-ehcache</artifactId>
5     <version>1.0.0</version>
6 </dependency>
7

```

2.

ehcache.xml

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3     xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd"
4     updateCheck="false">
5     <!--
6         diskStore: 为缓存路径，ehcache分为内存和磁盘两级，此属性定义磁盘的缓存位置。参数
        解释如下：
7         user.home - 用户主目录
8         user.dir - 用户当前工作目录
9         java.io.tmpdir - 默认临时文件路径
10    -->
11    <diskStore path="./tmpdir/Tmp_EhCache"/>
12
13    <defaultCache
14        eternal="false"
15        maxElementsInMemory="10000"
16        overflowToDisk="false"
17        diskPersistent="false"
18        timeToIdleSeconds="1800"
19        timeToLiveSeconds="259200"
20        memoryStoreEvictionPolicy="LRU"/>
21
22    <cache
23        name="cloud_user"
24        eternal="false"
25        maxElementsInMemory="5000"
26        overflowToDisk="false"
27        diskPersistent="false"
28        timeToIdleSeconds="1800"
29        timeToLiveSeconds="1800"
30        memoryStoreEvictionPolicy="LRU"/>
31    <!--
32        defaultCache: 默认缓存策略，当ehcache找不到定义的缓存时，则使用这个缓存策略。只
        能定义一个。
33    -->
34    <!--
35        name: 缓存名称。
36        maxElementsInMemory: 缓存最大数目
37        maxElementsOnDisk: 硬盘最大缓存个数。
38        eternal: 对象是否永久有效，一但设置了，timeout将不起作用。
39        overflowToDisk: 是否保存到磁盘，当系统当机时
40        timeToIdleSeconds: 设置对象在失效前的允许闲置时间（单位：秒）。仅当
        eternal=false对象不是永久有效时使用，可选属性，默认值是0，也就是可闲置时间无穷大。

```

41       **timeToLiveSeconds**:设置对象在失效前允许存活时间（单位：秒）。最大时间介于创建时间和失效时间之间。仅当**eternal=false**对象不是永久有效时使用，默认是0.，也就是对象存活时间无穷大。

42       **diskPersistent**: 是否缓存虚拟机重启期数据 **whether the disk store persists between restarts of the virtual machine. The default value is false.**

43       **diskSpoolBufferSizeMB**: 这个参数设置**DiskStore**（磁盘缓存）的缓存区大小。默认是30MB。每个**Cache**都应该有自己一个缓冲区。

44       **diskExpiryThreadIntervalSeconds**: 磁盘失效线程运行时间间隔，默认是120秒。

45       **memoryStoreEvictionPolicy**: 当达到**maxElementsInMemory**限制时，**Ehcache**将会根据指定的策略去清理内存。默认策略是**LRU**（最近最少使用）。你可以设置为**FIFO**（先进先出）或是**LFU**（较少使用）。

46       **clearOnFlush**: 内存数量最大时是否清除。

47       **memoryStoreEvictionPolicy**:可选策略有：**LRU**（最近最少使用，默认策略）、**FIFO**（先进先出）、**LFU**（最少访问次数）。

48       **FIFO, first in first out**, 这个是大家最熟的，先进先出。

49       **LFU, Less Frequently Used**, 就是上面例子中使用的策略，直白一点就是讲一直以来最少被使用的。如上面所讲，缓存的元素有一个**hit**属性，**hit**值最小的将会被清出缓存。

50       **LRU, Least Recently Used**, 最近最少使用的，缓存的元素有一个时间戳，当缓存容量满了，而又需要腾出地方来缓存新的元素的时候，那么现有缓存元素中时间戳离当前时间最远的元素将被清出缓存。

51       -->

52

53 </ehcache>

54

```

3. <!-- 在当前mapper中使用二级缓存-->
   <cache type="org.mybatis.caches.ehcache"/>

```

Redis数据库来做缓存

## 13.7 Mybatis总结