

# MyBatis

---

## 环境:

- JDK1.8
- MySQL 5.7
- Maven 3.6.1
- IDEA

## 回顾:

- JDBC
- MySQL
- Java基础
- Maven
- Junit

## 如何获得:

maven仓库:

```
1 <dependency>
2   <groupId>org.mybatis</groupId>
3   <artifactId>mybatis</artifactId>
4   <version>3.5.2</version>
5 </dependency>
```

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

Github: <https://github.com/mybatis/mybatis-3>

---

## 1. 简介

---

### 1.1 什么是MyBatis



# 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[iBatis](#), 2010年这个项目由apache software foundation 迁移到了google code, 并且改名为MyBatis。
- 2013年11月迁移到Github。

## 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；

为什么需要持久化？

- 有一些对象，不能丢失；
- 内存太贵了；

## 1.3 持久层

- 完成持久化工作的代码块；
- 层是界限十分明显的；

## 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；
- 简化、框架、自动化；
- 更容易上手；

## 1.5 优点

- 简单易学**:本身就很很小且简单。没有任何第三方依赖，最简单安装只要两个jar文件+配置几个sql映射文件易于学习，易于使用，通过文档和源代码，可以比较完全的掌握它的设计思路和实现。
- 灵活**:mybatis不会对应用程序或者数据库的现有设计强加任何影响。sql写在xml里，便于统一管理和优化。通过sql语句可以满足操作数据库的所有需求。
- 解除sql与程序代码的耦合:通过提供DAO层，将业务逻辑和数据访问逻辑分离，使系统的设计更清晰，更易维护，更易单元测试。**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

---

【工程1：mybatis-01】

### 2.1 编写配置文件

- 编写MyBatis的配置类

```
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  <!--configuration的核心配置文件。-->
6  <configuration>
7
8      <environments default="development">
9          <environment id="development">
10              <transactionManager type="JDBC"/>
11              <dataSource type="POOLED">
```

```

12         <property name="driver" value="com.mysql.jdbc.Driver"/>
13         <property name="url"
value="jdbc:mysql://localhost:3306/mybatis?
useSSL=true&useUnicode=true&characterEncoding=utf-8"/>
14         <property name="username" value="root"/>
15         <property name="password" value="123456"/>
16     </dataSource>
17 </environment>
18 </environments>
19 <!--每一个Mapper.xml都需要在Mabatis的核心配置文件中注册-->
20 <mappers>
21     <mapper resource="com.longg.dao.UserMapper.xml"/>
22 </mappers>
23
24 </configuration>

```

- 编写MyBatis的工具类

```

1 public class MybatisUtils {
2
3     private static SqlSessionFactory sqlSessionFactory;
4
5     /**
6      * 使用MyBatis的第一步: 获取sqlSessionFactory对象
7      */
8     static{
9         try {
10             String resource = "mybatis-config.xml";
11             InputStream inputStream =
Resources.getResourceAsStream(resource);
12             sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);
13         } catch (IOException e) {
14             e.printStackTrace();
15         }
16     }
17
18
19     /**
20      * 既然有了 SqlSessionFactory，顾名思义，我们可以从中获得 SqlSession 的实例。
SqlSession 提供了在数据库
21      * 执行 SQL 命令所需的所有方法。你可以通过 SqlSession 实例来直接执行已映射的 SQL
语句。
22      */
23     public static SqlSession getSqlSession(){
24         SqlSession sqlSession = sqlSessionFactory.openSession();
25
26         return sqlSession;
27     }
28 }

```

## 2.2 编写代码

- 实体类

```
1  public class User {
2      private int id;
3      private String name;
4      private String pwd;
5
6      public User() {
7      }
8
9      public User(int id, String name, String pwd) {
10         this.id = id;
11         this.name = name;
12         this.pwd = pwd;
13     }
14
15     public int getId() {
16         return id;
17     }
18
19     public void setId(int id) {
20         this.id = id;
21     }
22
23     public String getName() {
24         return name;
25     }
26
27     public void setName(String name) {
28         this.name = name;
29     }
30
31     public String getPwd() {
32         return pwd;
33     }
34
35     public void setPwd(String pwd) {
36         this.pwd = pwd;
37     }
38
39     @Override
40     public String toString() {
41         return "User{" +
42             "id=" + id +
43             ", name='" + name + '\'' +
44             ", pwd='" + pwd + '\'' +
45             '}';
46     }
47 }
48
```

- Mapper接口：由原来的UserDao.java变成现在的UserMapper接口

```

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

```

- 接口实现类(UserMapper.xml): 由原来的UserDaoImpl.java变成现在的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 绑定一个对应的Dao/Mapper接口-->
6 <mapper namespace="com.longg.dao.UserMapper">
7
8     <select id="getUserList" resultType="com.longg.pojo.User">
9         select * from mybatis.user;
10    </select>
11 </mapper>

```

- 测试类

```

1 public class UserDaoTest {
2     @Test
3     public void test(){
4
5         /**
6          * 第一步: 获得SqlSession 的对象
7          */
8         SqlSession sqlSession = MybatisUtils.getSqlSession();
9
10        /**
11         * 第二步: 执行sql
12         * 方式一: getMapper
13         */
14        UserMapper com.longg.mapper =
15        sqlSession.getMapper(UserMapper.class);
16        List<User> userList = com.longg.mapper.getUserList();
17
18        for (User user :
19            userList) {
20            System.out.println(user);
21        }
22
23        /**
24         * 关闭SqlSession
25         */
26        sqlSession.close();
27    }
28 }

```

- **注意点:**

1. 报错: org.apache.ibatis.binding.BindingException: Type interface com.com.longg.dao.UserMapper is not known to the MapperRegistry.

MapperRegistry: Mapper注册中心: 每一个Mapper.xml都需要在Mabatis的核心配置文件中注册

解决方法：在MyBatis的核心配置文件中添加以下代码

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

2. Maven导出资源问题：由于maven的约定大于配置，所有可能会遇见我们写的配置文件无法被导出或者生效的问题。

解决方法：在pom文件中添加以下代码

```
1 <!--在build中配置Resources，来防止我们的资源文件导出失败的问题-->
2 <build>
3     <resources>
4         <resource>
5             <directory>src/main/resources</directory>
6             <includes>
7                 <include>**/*.properties</include>
8                 <include>**/*.xml</include>
9             </includes>
10            <filtering>true</filtering>
11        </resource>
12        <resource>
13            <directory>src/main/java</directory>
14            <includes>
15                <include>**/*.properties</include>
16                <include>**/*.xml</include>
17            </includes>
18            <filtering>true</filtering>
19        </resource>
20    </resources>
21 </build>
```

---

## 3. CRUD

---

【工程1：mybatis-01】

### 3.1 select

选择、查询语句；

- id：就是对应命名空间namespace中的方法名；
- resultType：SQL语句执行的返回值；即实体类；
- parameterType：参数类型；

#### 1. 编写接口；

```
1 /**
2  * 根据ID查询用户
3  */
4 User getUserById(int id);
```

## 2. 编写对应的mapper中的sql语句;

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

## 3. 测试;

```
1 @Test
2 public void getUserById(){
3     SqlSession sqlSession = MybatisUtils.getSqlSession();
4
5     UserMapper mapper = sqlSession.getMapper(UserMapper.class);
6     User user = mapper.getUserById(1);
7     System.out.println(user);
8
9     sqlSession.close();
10 }
```

## 3.2 insert

### 1. 编写接口;

```
1 /**
2  * 插入一个用户
3  */
4 int addUser(User user);
```

## 2. 编写对应的mapper中的sql语句;

```
1 <insert id="addUser" parameterType="com.longg.pojo.User">
2     insert into mybatis.user (id,name,pwd) values (#{id},#{name},#{pwd});
3 </insert>
```

## 3. 测试;

```
1 @Test
2 public void addUser(){
3     SqlSession sqlSession = MybatisUtils.getSqlSession();
4
5     UserMapper mapper = sqlSession.getMapper(UserMapper.class);
6     int result = mapper.addUser(new User(6, "long", "123456"));
7     if(result > 0) {
8         System.out.println("插入成功! ");
9     }
10
11     // 提交事务（手动）
12     sqlSession.commit();
13
14     sqlSession.close();
15 }
```

## 3.3 update

### 1. 编写接口;

```
1  /**
2   * 修改用户
3   */
4  int updateUser(User user);
```

### 2. 编写对应的mapper中的sql语句;

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

### 3. 测试;

```
1  @Test
2  public void updateUser(){
3      SqlSession sqlSession = MybatisUtils.getSqlSession();
4
5      UserMapper mapper = sqlSession.getMapper(UserMapper.class);
6      mapper.updateUser(new User(4, "hhh", "654321"));
7
8      sqlSession.commit();
9
10     sqlSession.close();
11 }
```

## 3.4 delete

### 1. 编写接口;

```
1  /**
2   * 删除用户
3   */
4  int deleteUser(int id);
```

### 2. 编写对应的mapper中的sql语句;

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

### 3. 测试;



```

1  @Test
2  public void deleteUser(){
3      SqlSession sqlSession = MybatisUtils.getSqlSession();
4
5      UserMapper mapper = sqlSession.getMapper(UserMapper.class);
6      mapper.deleteUser(6);
7
8      sqlSession.commit();
9
10     sqlSession.close();
11 }

```

**注意点：增删改需要提交事务**

## 3.5 Map

假设实体类或者数据库中的字段过多，我们需要考虑使用Map来传数据；

```

1  /**
2   * 用万能的map插入数据
3   */
4  int addUserMap(Map<String ,Object> map);

```

```

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

```

```

1  @Test
2  public void addUserMap(){
3      SqlSession sqlSession = MybatisUtils.getSqlSession();
4
5      UserMapper mapper = sqlSession.getMapper(UserMapper.class);
6
7      HashMap<String, Object> map = new HashMap<String, Object>();
8      map.put("userId",6);
9      map.put("userName","long");
10     map.put("password","123456");
11
12     mapper.addUserMap(map);
13
14     sqlSession.commit();
15     sqlSession.close();
16 }

```

## 总结：

- 只有一个基本类型参数的情况下，可以直接在sql中取其参数，可以不用定义参数类型（也可以定义：【parameterType="int"】）；
- 对象传递参数的情况下，可以直接在sql中取其对象的属性；  
【parameterType="Object"（com.com.longg.pojo.User）】；
- Map传递参数的情况下，可以直接在sql中取其对应的key值；【parameterType="map"】

**如果是多个无规则的参数则可以用Map或者是注解；**

# 模糊查询

## 模糊查询怎么写？

1. 在java代码执行的时候传递通配符 % %; 【List users = com.longg.mapper.getUserLike("%李%");】

```
1  /**
2   * 模糊查询
3   */
4  List<User> getUserLike(String str);
```

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

```
1  @Test
2  public void getUserLike() {
3      SqlSession sqlSession = MybatisUtils.getSqlSession();
4
5      UserMapper mapper = sqlSession.getMapper(UserMapper.class);
6      List<User> users = mapper.getUserLike("%李%");
7      for (User user : users) {
8          System.out.println(user);
9      }
10
11     sqlSession.close();
12 }
```

2. 在sql中拼接使用通配符% %; 【select \* from user where name like "%"#{value}"%";】

```
1  /**
2   * 模糊查询
3   */
4  List<User> getUserLike(String str);
```

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

```
1  @Test
2  public void getUserLike() {
3      SqlSession sqlSession = MybatisUtils.getSqlSession();
4
5      UserMapper mapper = sqlSession.getMapper(UserMapper.class);
6      List<User> users = mapper.getUserLike("龙");
7      for (User user : users) {
8          System.out.println(user);
9      }
10
11     sqlSession.close();
12 }
```

## 4. 配置解析

【工程2: mybatis-02】

### 4.1 核心配置文件

- mybatis-config.xml文件;
- MyBatis的配置文件包含并会深深的影响MyBatis的行为设置和属性信息;

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)

注: 标红的是需要掌握的;

### 4.2 环境配置 (environment)

MyBatis可以配置多个不同的环境来适应多种环境的需要; 通过【default=""】来选择不同的环境;

**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。**

学会使用配置来配置多套不同的运行环境:

```
1 <environments default="development">      <!--默认使用的环境 ID (如:
   default="test") -->
2     <environment id="development">
3         <transactionManager type="JDBC"/>
4         <dataSource type="POOLED">
5             <property name="driver" value="com.mysql.jdbc.Driver"/>
6             <property name="url" value="jdbc:mysql://localhost:3306/mybatis?
   useSSL=true&useUnicode=true&characterEncoding=utf-8"/>
7             <property name="username" value="root"/>
8             <property name="password" value="123456"/>
9         </dataSource>
10    </environment>
11
12    <environment id="test"> <!--每个environment元素定义的环境ID-->
13        <transactionManager type="JDBC"/>
14        <dataSource type="POOLED">
15            <property name="driver" value="com.mysql.jdbc.Driver"/>
16            <property name="url" value="jdbc:mysql://localhost:3306/mybatis?
   useSSL=true&useUnicode=true&characterEncoding=utf-8"/>
17            <property name="username" value="root"/>
18            <property name="password" value="123456"/>
19        </dataSource>
20    </environment>
```

```
21
22 </environments>
```

Mybatis默认的事务管理器是：JDBC，连接池是：POOLED；

**注意点：**

- 默认使用的环境 ID（比如：default="development"）。
- 每个 environment 元素定义的环境 ID（比如：id="development"）。
- 事务管理器的配置（比如：type="JDBC"）。
- 数据源的配置（比如：type="POOLED"）。

## 4.3 属性 (properties)

我们可以通过properties属性来实现引用配置文件；

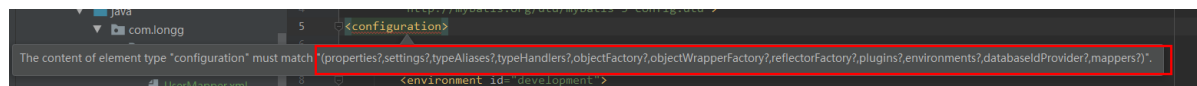
这些属性可以在外部进行配置，并可以进行动态替换。你既可以在典型的 Java 属性文件中配置这些属性，也可以在 properties 元素的子元素中设置。【db.properties】

- 编写配置文件 db.properties

```
1 driver = com.mysql.jdbc.Driver
2 url = jdbc:mysql://localhost:3306/mybatis?
  useSSL=true&useUnicode=true&characterEncoding=utf-8"
3 username = root
4 password = 123456
```

- 在核心配置文件中映入

**注意：**在configuration中写这些配置的顺序必须是按照图中的顺序啦，否则将报错！！



```
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 <configuration>
6
7     <!--引入外部配置文件-->
8     <properties resource="db.properties"/>
9
10    <environments default="test">
11        </environment>
12        <environment id="test">
13            <transactionManager type="JDBC"/>
14            <dataSource type="POOLED">
15                <property name="driver" value="${driver}"/>
16                <property name="url" value="${url}"/>
17                <property name="username" value="${username}"/>
18                <property name="password" value="${password}"/>
19            </dataSource>
20        </environment>
21    </environments>
22
23    <mappers>
24        <mapper resource="com/longg/mapper/UserMapper.xml"/>
25    </mappers>
26 </configuration>
```

```
25     </mappers>
26
27 </configuration>
```

## 4.4 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。

- 在核心配置文件中取别名
  - 给实体类取别名

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

```
1  <select id="getUserList" resultType="User">
2      select * from mybatis.user;
3  </select>
```

- 给对应的包取别名：可以指定一个包名，MyBatis 会在包名下面搜索需要的 Java Bean；每一个在包 `com.long.pojo` 中的 Java Bean，在没有注解的情况下，会使用 Bean 的首字母小写的非限定类名来作为它的别名。比如 `com.long.pojo.User` 的别名为 `user`；若有注解，则别名为其注解值。

```
1  <!--给对应的包取别名-->
2  <typeAliases>
3      <package name="com.longg.pojo"/>
4  </typeAliases>
```

```
1  <select id="getUserList" resultType="user">
2      select * from mybatis.user;
3  </select>
```

区别：

- 在实体类比较少的时候使用第一种方法；
- 如果实体类比较多的情况下推荐使用第二种方法；
- 第一种可以使用自定义别名，第二种只能使用注解自定义别名；【@Alias(" ")】
- 下面是一些为常见的 Java 类型内建的类型别名。它们都是不区分大小写的，注意，为了应对原始类型的命名重复，采取了特殊的命名风格。

别名	映射的类型
_byte	byte
_long	long
_short	short
_int	int
_integer	int
_double	double
_float	float
_boolean	boolean
string	String
byte	Byte
long	Long
short	Short
int	Integer
integer	Integer
double	Double
float	Float
boolean	Boolean
date	Date
decimal	BigDecimal
bigdecimal	BigDecimal
object	Object
map	Map
hashmap	HashMap
list	List
arraylist	ArrayList
collection	Collection
iterator	Iterator

## 4.5 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。

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

一个配置完整的 settings 元素的示例如下：

```
1 <settings>
2   <setting name="cacheEnabled" value="true"/>
3   <setting name="lazyLoadingEnabled" value="true"/>
4   <setting name="multipleResultSetsEnabled" value="true"/>
5   <setting name="useColumnLabel" value="true"/>
6   <setting name="useGeneratedKeys" value="false"/>
7   <setting name="autoMappingBehavior" value="PARTIAL"/>
8   <setting name="autoMappingUnknownColumnBehavior" value="WARNING"/>
9   <setting name="defaultExecutorType" value="SIMPLE"/>
10  <setting name="defaultStatementTimeout" value="25"/>
11  <setting name="defaultFetchSize" value="100"/>
12  <setting name="safeRowBoundsEnabled" value="false"/>
13  <setting name="mapUnderscoreToCamelCase" value="false"/>
14  <setting name="localCacheScope" value="SESSION"/>
15  <setting name="jdbcTypeForNull" value="OTHER"/>
16  <setting name="lazyLoadTriggerMethods"
    value="equals,clone,hashCode,toString"/>
17 </settings>
```

## 4.6 映射器 (mappers)

**MapperRegistry: Mapper注册中心：每一个Mapper.xml文件都需要在Mabatis的核心配置文件中注册**

- 注册方式一：使用资源文件绑定注册

```
1 <!-- 使用相对于类路径的资源引用 -->
2 <mappers>
3   <com.longg.mapper resource="org/mybatis/builder/AuthorMapper.xml"/>
4   <com.longg.mapper resource="org/mybatis/builder/BlogMapper.xml"/>
5   <com.longg.mapper resource="org/mybatis/builder/PostMapper.xml"/>
6 </mappers>
7
8 例：
9 <mappers>
10   <com.longg.mapper resource="com/longg/mapper/UserMapper.xml"/>
11 </mappers>
```

- 注册方式二：使用class文件绑定注册

```

1  <!-- 使用映射器接口实现类的完全限定类名 -->
2  <mappers>
3      <mapper class="org.mybatis.builder.AuthorMapper"/>
4      <mapper class="org.mybatis.builder.BlogMapper"/>
5      <mapper class="org.mybatis.builder.PostMapper"/>
6  </mappers>
7
8  例：
9  <mappers>
10     <mapper class="com.longg.mapper.UserMapper"/>
11 </mappers>

```

#### 注意点：

1. 接口和它的Mapper配置文件必须同名；
  2. 接口和它的Mapper配置文件必须在同一个包下；
- 注册方式三：使用扫描包进行注入绑定

```

1  <!-- 将包内的映射器接口实现全部注册为映射器 -->
2  <mappers>
3      <package name="org.mybatis.builder"/>
4  </mappers>
5
6  例：
7  <mappers>
8      <package name="com.longg.mapper"/>
9  </mappers>

```

#### 注意点：

1. 接口和它的Mapper配置文件必须同名；
2. 接口和它的Mapper配置文件必须在同一个包下；

## 4.7 作用域（Scope）和生命周期

理解我们之前讨论过的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

- 简单的说就可以理解为：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；
- 有很多方法可以做到，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;
- 用完之后即可关闭, 否则容易导致资源占用;

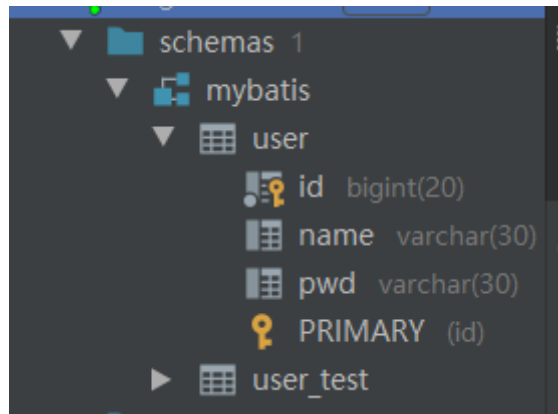
---

## 5. 解决属性名与字段名不一致的问题

---

### 【工程3: mybatis-03】

数据库中的字段:



新建一个项目, 拷贝之前的, 测试实体类字段不一致的情况:

```
1 public class User {
2     private int id;
3     private String name;
4     private String password;
5 }
```

测试出现问题:

```
6ms "D:\Program Files\Java\jdk1.8.0_152\bin\java.exe" ...
6ms
User{id=1, name='龙龙', password='null'}
```

sql语句:

```
1 <select id="getUserById" parameterType="int"
2     responseType="com.longg.pojo.User">
3     select * from mybatis.user where id = #{id};
4 </select>
5 原始sql: select id,name,pwd from mybatis.user where id = #{id};
```

解决方法: (最不可取)

- 给pwd取别名:

```

1 <select id="getUserById" parameterType="int"
  resultType="com.longg.pojo.User">
2     select id,name,pwd as password from mybatis.user where id = #{id};
3 </select>

```

- 用结果集映射，见5.1;

## 5.1 resultMap

结果集映射

```

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.longg.mapper.UserMapper">
7
8     <!--结果集映射-->
9     <resultMap id="UserMap" type="User">
10         <!--column为数据库中的字段，property为实体类中的字段-->
11         <result column="id" property="id"/>
12         <result column="name" property="name"/>
13         <result column="pwd" property="password"/>
14     </resultMap>
15
16     <select id="getUserById" parameterType="int" resultMap="UserMap">
17         select * from mybatis.user where id = #{id};
18     </select>
19
20 </mapper>

```

- 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;
- 如果数据库字段和实体类字段名相同的情况下可以不用映射相同的字段，只映射不同的字段，也可以成功;

```

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.longg.mapper.UserMapper">
7
8     <!--结果集映射-->
9     <resultMap id="UserMap" type="User">
10         <!--column为数据库中的字段，property为实体类中的字段-->
11         <result column="pwd" property="password"/>
12     </resultMap>
13
14     <select id="getUserById" parameterType="int" resultMap="UserMap">
15         select * from mybatis.user where id = #{id};
16     </select>
17

```

## 6. 日志

### 【工程4: mybatis-04】

### 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，日志就是最好的帮手；

原来：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   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中具体使用哪一个日志实现，在设置中设定；

### 6.2 STDOUT\_LOGGING 标准日志输出

在mybatis的核心配置文件中配置日志：

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

输出结果：

```
PoolDataSource forcibly closed/removed all connections.
Opening JDBC Connection
Created connection 1702143276.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@6574a52c]
==> Preparing: select * from mybatis.user where id = ?;
==> Parameters: 1(Integer)
<==      Columns: id, name, pwd
<==      Row: 1, 龙龙, 12345678
<==      Total: 1
User{id=1, name='龙龙', password='12345678'}
Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@6574a52c]
Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@6574a52c]
Returned connection 1702143276 to pool.
```

## 6.3 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；
- 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 <dependency>
2   <groupId>apache-log4j</groupId>
3   <artifactId>log4j</artifactId>
4   <version>1.2.15</version>
5 </dependency>
```

### 2. log4j.properties文件配置

```
1  ### 设置输出sql的级别，其中logger后面的内容全部为jar包中所包含的包名 ###
2  log4j.logger.org.apache=DEBUG
3  log4j.logger.java.sql.Connection=DEBUG
4  log4j.logger.java.sql.Statement=DEBUG
5  log4j.logger.java.sql.PreparedStatement=DEBUG
6  log4j.logger.java.sql.ResultSet=DEBUG
7
8  ### 配置输出到控制台 ###
9  log4j.appender.console = org.apache.log4j.ConsoleAppender
10 log4j.appender.console.Target = System.out
11 log4j.appender.console.Threshold = DEBUG
12 log4j.appender.console.layout = org.apache.log4j.PatternLayout
13 log4j.appender.console.layout.ConversionPattern = %d{ABSOLUTE} %5p %c{ 1
   }:%L - %m%n
14
15 ### 配置输出到文件 ###
16 log4j.appender.file = org.apache.log4j.FileAppender
17 log4j.appender.file.File = ./logs/long.log
18 log4j.appender.file.MaxFileSize = 100KB
19 log4j.appender.file.Append = true
20 log4j.appender.file.Threshold = DEBUG
21 log4j.appender.file.layout = org.apache.log4j.PatternLayout
22 log4j.appender.file.layout.ConversionPattern = %-d{yyyy-MM-dd HH:mm:ss} [
   %t:%r ] - [ %p ] %m%n
```

### 3. 在mybatis的核心配置文件中配置 LOG4J 日志：

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

简单使用：

1. 在要使用Log4j 的类中，导入对应的包：import org.apache.log4j.Logger;

2. 日志对象，参数为当前类的class;

```
1 | Logger logger = Logger.getLogger(UserMapperTest.class);
```

3. 日志级别

```
1 | logger.info("info:进入了testlog4j");
2 | logger.debug("debug:进入了testlog4j");
3 | logger.error("error:进入了testlog4j");
```

4. 测试输出

```
ms
log4j:WARN No such property [maxFileSize] in org.apache.log4j.FileAppender.
[ INFO ] 2020-09-24 19:20:19 UserMapperTest:30 - info:进入了testlog4j
[ DEBUG ] 2020-09-24 19:20:19 UserMapperTest:31 - debug:进入了testlog4j
[ ERROR ] 2020-09-24 19:20:19 UserMapperTest:32 - error:进入了testlog4j
```

## 7. 分页

【工程4: mybatis-04】

思考：为什么要分页？

- 减少数据的处理量；

### 7.1 Limit分页的方法

- 传统使用Limit分页：

```
1 | 语法：select * from user limit startIndex,PageSize;
2 | select * from user limit 2,3;
```

- 使用MyBatis实现分页，核心为SQL；

- 接口

```
1 | /**
2 |  * 分页查询
3 |  */
4 | List<User> getUserLimit(Map<String,Integer> map);
```

- mapper.xml

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

- 测试

```
1 | @Test
```

```

2 public void getUserLimit(){
3     SqlSession sqlSession = MybatisUtils.getSqlSession();
4
5     UserMapper mapper = sqlSession.getMapper(UserMapper.class);
6     HashMap<String, Integer> map = new HashMap<String, Integer>();
7     map.put("startIndex",2);
8     map.put("pageSize",3);
9     List<User> users = mapper.getUserLimit(map);
10    for (User user : users) {
11        System.out.println(user);
12    }
13
14    sqlSession.close();
15 }

```

## 7.2 RowBounds分页的方法

不再使用SQL实现分页，在Java层实现分页

- 接口

```

1 /**
2  * 分页查询:RowBounds
3  */
4 List<User> getUserRowBounds();

```

- mapper.xml

```

1 <select id="getUserRowBounds" resultMap="UserMap" >
2     select * from user ;
3 </select>

```

- 测试

```

1 @Test
2 public void getUserRowBounds(){
3     SqlSession sqlSession = MybatisUtils.getSqlSession();
4
5     // RowBounds实现分页
6     RowBounds rowBounds = new RowBounds(2, 3);
7
8     List<User> users =
9     sqlSession.selectList("com.longg.mapper.UserMapper.getUserRowBounds",null,row
10    Bounds);
11
12    for (User user : users) {
13        System.out.println(user);
14    }
15
16    sqlSession.close();
17 }

```

## 7.3 分页插件

# 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](#)

maven central 5.2.0

## 8. 注解开发

【工程5：mybatis-05】

### 8.1 面向接口编程

#### 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(名实分离的原则)的分离。
- 接口的本身反映了系统设计人员对系统的抽象理解。
- 接口应有两类：
  - 第一类是对一个体的抽象，它可对应为一个抽象体(abstract class);
  - 第二类是对一个体某一方面的抽象，即形成一个抽象面(interface);
- 一个体有可能有多个抽象面。抽象体与抽象面是有区别的。

#### 三个面向的区别

- **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

### 8.2 使用注解开发

1. 注解在接口上实现；

```
1 public interface UserMapper {
2
3     /**
4      * 查询全部用户
5      */
6     @Select("select * from user")
7     List<User> getUsers();
8
9 }
```

2. 需要在核心配置文件中绑定接口；

```

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

```

### 3. 测试使用

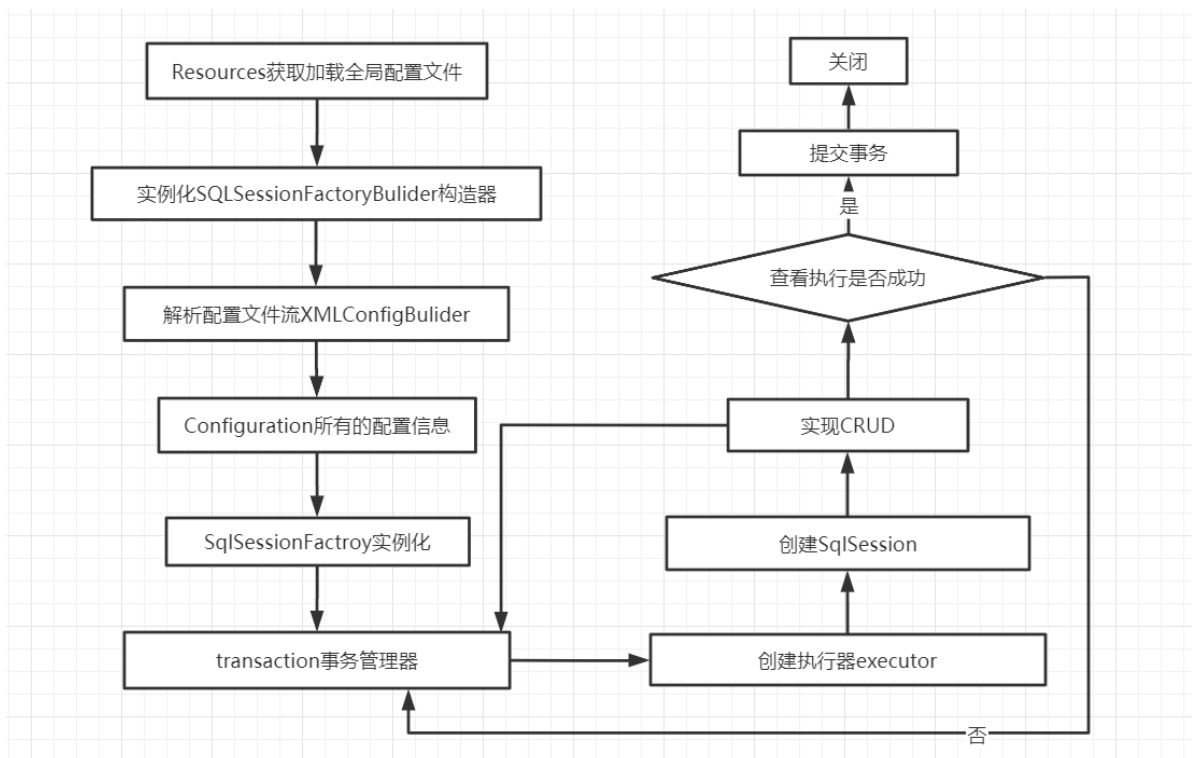
```

1 @Test
2 public void getUsers(){
3     sqlSession = MybatisUtils.getSqlSession();
4
5     // 底层主要运用反射
6     UserMapper mapper = sqlSession.getMapper(UserMapper.class);
7     List<User> users = mapper.getUsers();
8     for (User user : users) {
9         System.out.println(user);
10    }
11
12    sqlSession.close();
13 }

```

- 本质：反射机制实现；
- 底层：动态代理；

## MyBatis详细的执行流程：





## 8.3 自动提交事务

在MyBatis工具类MyBatisUtils中的实例化SqlSession的getSqlSession()方法中加上 `true` 即可自动提交事务，不用在执行SQL语句之后再手动提交事务，但是通常情况下不推荐使用自动提交事务的方法；  
可对比3.2-3.4节的代码

手动提交事务：工具类和测试类

```
public static SqlSession getSqlSession(){
    SqlSession sqlSession = sqlSessionFactory.openSession();

    return sqlSession;
}
```

```
@Test
public void deleteUser(){
    SqlSession sqlSession = MybatisUtils.getSqlSession();

    UserMapper mapper = sqlSession.getMapper(UserMapper.class);
    System.out.println(mapper.deleteUser(id: 4));

    sqlSession.commit();

    sqlSession.close();
}
```

自动提交事务：工具类和测试类

```
public static SqlSession getSqlSession(){
    SqlSession sqlSession = sqlSessionFactory.openSession(b: true);

    return sqlSession;
}
```

```
@Test
public void deleteUser(){
    SqlSession sqlSession = MybatisUtils.getSqlSession();

    UserMapper mapper = sqlSession.getMapper(UserMapper.class);
    System.out.println(mapper.deleteUser(id: 4));

    sqlSession.close();
}
```

## 8.4 使用注解实现CRUD开发

- 编写接口,增加注解 (增删改查)

```
1 public interface UserMapper {
2
3     /**
4      * 使用注解查询全部用户
5      */
6     @Select("select * from user")
7     List<User> getUsers();
8
9     /**
10      * 使用注解根据id查询用户, 参数获取使用注解 @Param("userId");
11      * SQL语句中的 id = #{userId} 部分的“userId”参数由注解 @Param("userId") 装配;
12      * 如果方法存在多个参数, 则在所有的参数前面使用注解 @Param(" ") 即可;
13      */
14     @Select("select * from user where id = #{userId};")
15     User getUserById(@Param("userId") int id);
16
17     /**
18      * 使用注解插入对象
19      */
20     @Insert("insert into user(id,name,pwd) values (#{id},#{name},#{password})")
21     int insertUser(User user);
22
23     /**
24      * 使用注解修改用户
25      */
26     @Update("update user set name = #{name},pwd = #{password} where id = #{id};")
27     int updateUser(User user);
28
29     /**
30      * 使用注解删除用户
31      */
32     @Delete("delete from user where id = #{id};")
33     int deleteUser(@Param("id") int id);
34
35 }
```

### 关于@Param(" ")注解说明

- 基本类型的参数或者String类型, 需要加上; 【如上例代码中的“int”类型】
- 引用类型不需要加; 【如上例中的“User”类型】
- 如果只有一个基本类型的话, 可以忽略, 但是建议加上;
- 如果方法存在多个参数, 则在所有的参数前面使用注解 @Param(" ") 即可;
- 上述查询 SQL语句中的 id = #{userId} 部分的“userId”参数由注解 @Param("userId") 装配;

- 测试实现

```
1 public class UserMapperTest {
2
3     @Test
4     public void getUsers(){
5         SqlSession sqlSession = MybatisUtils.getSqlSession();
```

```
6
7 // 底层主要运用反射
8 UserMapper mapper = sqlSession.getMapper(UserMapper.class);
9 List<User> users = mapper.getUsers();
10 for (User user : users) {
11     System.out.println(user);
12 }
13
14 sqlSession.close();
15 }
16
17 @Test
18 public void getUserById(){
19     SqlSession sqlSession = MybatisUtils.getSqlSession();
20
21     UserMapper mapper = sqlSession.getMapper(UserMapper.class);
22     System.out.println(mapper.getUserById(2));
23
24     sqlSession.close();
25 }
26
27 @Test
28 public void insertUser(){
29     SqlSession sqlSession = MybatisUtils.getSqlSession();
30
31     UserMapper mapper = sqlSession.getMapper(UserMapper.class);
32     System.out.println(mapper.insertUser(new
33 User(8,"longlong","6666")));
34
35     sqlSession.close();
36 }
37
38 @Test
39 public void updateUser(){
40     SqlSession sqlSession = MybatisUtils.getSqlSession();
41
42     UserMapper mapper = sqlSession.getMapper(UserMapper.class);
43     System.out.println(mapper.updateUser(new
44 User(8,"long","66668888")));
45
46     sqlSession.close();
47 }
48
49 @Test
50 public void deleteUser(){
51     SqlSession sqlSession = MybatisUtils.getSqlSession();
52
53     UserMapper mapper = sqlSession.getMapper(UserMapper.class);
54     System.out.println(mapper.deleteUser(4));
55
56     sqlSession.close();
57 }
```

- 注意：必须将我们的接口类注册绑定到配置文件中，详见8.1节

## 9. Lombok

### 【工程6：mybatis-06】

Lombok项目是一个Java库，它会自动插入编辑器和构建工具中，Lombok提供了一组有用的注释，用来消除Java类中的大量样板代码。仅五个字符(@Data)就可以替换数百行代码从而产生干净，简洁且易于维护的Java类。

### 9.1 常用注解

- @Setter：注解在类或字段，注解在类时为所有字段生成setter方法，注解在字段上时只为该字段生成setter方法。
- @Getter：使用方法同上，区别在于生成的是getter方法。
- @ToString：注解在类，添加toString方法。
- @EqualsAndHashCode：注解在类，生成hashCode和equals方法。
- @NoArgsConstructor：注解在类，生成无参的构造方法。
- @RequiredArgsConstructor：注解在类，为类中需要特殊处理的字段生成构造方法，比如final和被@NonNull注解的字段。
- @AllArgsConstructor：注解在类，生成包含类中所有字段的构造方法。
- @Data：注解在类，生成无参构造方法、setter/getter、equals、canEqual、hashCode、toString方法，如为final属性，则不会为该属性生成setter方法。【最重要】
- @Slf4j：注解在类，生成log变量，严格意义来说是常量。

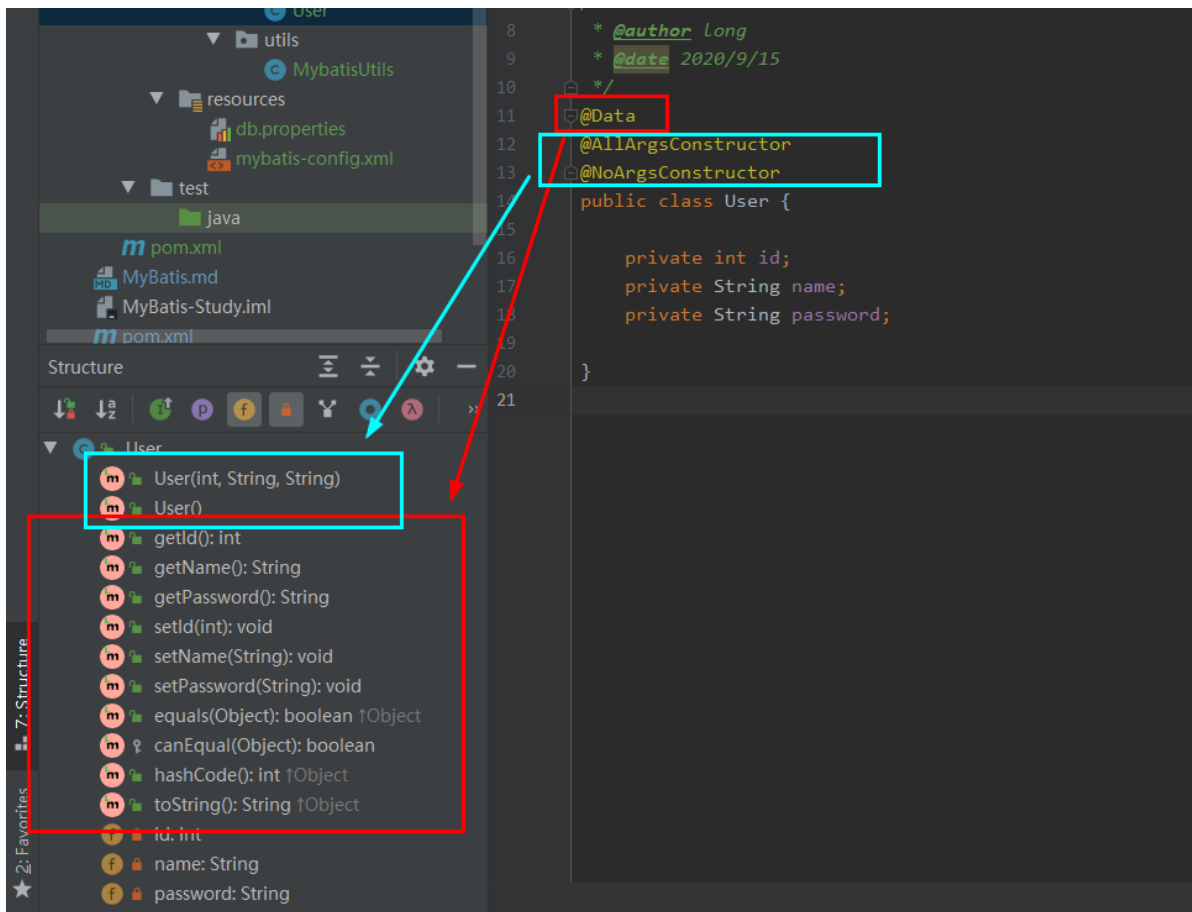
### 9.2 使用Lombok的步骤

1. 在 IDEA 中安装对应的插件
2. 在项目的pom文件中导入Lombok的jar包

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

3. 在实体类上添加其注解

```
1 @Data
2 @AllArgsConstructor
3 @NoArgsConstructor
4 public class User {
5
6     private int id;
7     private String name;
8     private String password;
9
10 }
```



## 10. 多对一情况处理

### 【工程7：mybatis-07】

#### 测试环境搭建

1. 导入Lombok;
2. 新建实体类：Teacher、Student;

```
1  @Data
2  public class Student {
3      private int id;
4      private String name;
5
6      /**
7       * 学生对应的老师对象
8       */
9      private Teacher teacher;
10 }
```

```
1  @Data
2  public class Teacher {
3      private int id;
4      private String name;
5  }
```

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接口或文件;

6. 测试查询是否成功;

## 10.1 按照查询嵌套处理

类似于Sql中的子查询:

mapper.xml文件:

```
1 <select id="getStudent" resultMap="StudentAndTeacher">
2     select * from student;
3 </select>
4
5 <resultMap id="StudentAndTeacher" type="Student">
6     <result property="id" column="id"/>
7     <result property="name" column="name"/>
8     <!--
9         复杂的属性需要单独处理
10        对象: association
11        集合: collection
12    -->
13     <association property="teacher" column="teacher_id" javaType="Teacher"
14     select="getTeacher"/>
15 </resultMap>
16
17 <select id="getTeacher" resultType="Teacher">
18     select * from teacher where id = #{teacher_id};
19 </select>
```

结果展示:

```
==> Preparing: select * from student;
==> Parameters:
<==    Columns: id, name, teacher_id
<==    Row: 1, 小明, 1
====> Preparing: select * from teacher where id = ?;
====> Parameters: 1(Long)
<====    Columns: id, name
<====    Row: 1, 秦老师
<====    Total: 1
<==    Row: 2, 小红, 1
<==    Row: 3, 小张, 1
<==    Row: 4, 小李, 1
<==    Row: 5, 小王, 1
<==    Total: 5
Student(id=1, name=小明, teacher=Teacher(id=1, name=秦老师))
Student(id=2, name=小红, teacher=Teacher(id=1, name=秦老师))
Student(id=3, name=小张, teacher=Teacher(id=1, name=秦老师))
Student(id=4, name=小李, teacher=Teacher(id=1, name=秦老师))
Student(id=5, name=小王, teacher=Teacher(id=1, name=秦老师))
```

## 10.2 按照结果嵌套处理

类似于Sql中的嵌套查询:

mapper.xml文件:

```
1 <select id="getStudent2" resultMap="StudentAndTeacher2">
2     select s.id sId, s.name sName, t.name tName
3     from teacher t, student s
4     where s.teacher_id = t.id;
5 </select>
```

```

6
7 <resultMap id="StudentAndTeacher2" type="Student">
8   <result property="id" column="sId"/>
9   <result property="name" column="sName"/>
10  <!--
11      复杂的属性需要单独处理
12      对象: association
13      集合: collection
14  -->
15  <association property="teacher" javaType="Teacher">
16      <result property="name" column="tName"/>
17  </association>
18 </resultMap>

```

结果展示:

```

==> Preparing: select s.id sId, s.name sName, t.name tName from teacher t, student s where s.teacher_id = t.id;
==> Parameters:
<== Columns: sId, sName, tName
<== Row: 1, 小明, 秦老师
<== Row: 2, 小红, 秦老师
<== Row: 3, 小张, 秦老师
<== Row: 4, 小李, 秦老师
<== Row: 5, 小王, 秦老师
<== Total: 5
Student(id=1, name=小明, teacher=Teacher(id=0, name=秦老师))
Student(id=2, name=小红, teacher=Teacher(id=0, name=秦老师))
Student(id=3, name=小张, teacher=Teacher(id=0, name=秦老师))
Student(id=4, name=小李, teacher=Teacher(id=0, name=秦老师))
Student(id=5, name=小王, teacher=Teacher(id=0, name=秦老师))
Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@5f71c76a]

```

## 11. 一对多情况处理

### 【工程8: mybatis-08】

#### 测试环境搭建

1. 导入Lombok;
2. 新建实体类: Teacher、Student;

```

1 @Data
2 public class Student {
3     private int id;
4     private String name;
5     private int teacherId;
6 }

```

```

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

```

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接口或文件;
6. 测试查询是否成功;

## 11.1 按照结果嵌套查询

类似于Sql中的嵌套查询:

mapper.xml文件:

```
1 <select id="getTeacherStudent" resultMap="TeacherAndStudent">
2     select s.id sId, s.name sName, t.name tName, t.id tId
3     from teacher t, student s
4     where s.teacher_id = t.id ;
5 </select>
6
7 <resultMap id="TeacherAndStudent" type="Teacher">
8     <result column="tId" property="id"/>
9     <result column="tName" property="name"/>
10    <!--
11        复杂的属性需要单独处理
12        对象: association
13        集合: collection
14        javaType="": 为指定属性的类型
15        ofType="": 为集合中的泛型信息
16    -->
17    <collection property="students" ofType="Student">
18        <result property="id" column="sId"/>
19        <result property="name" column="sName"/>
20        <result property="teacherId" column="tId"/>
21    </collection>
22 </resultMap>
```

结果展示:

```
Created connection 2275935.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@15b3e5b]
==> Preparing: select s.id sId, s.name sName, t.name tName, t.id tId from teacher t, student s where s.teacher_id = t.id ;
==> Parameters:
<== Columns: sId, sName, tName, tId
<== Row: 1, 小明, 蔡老师, 1
<== Row: 3, 小张, 蔡老师, 1
<== Row: 4, 小李, 蔡老师, 1
<== Row: 5, 小王, 蔡老师, 1
<== Row: 2, 小红, 李老師, 2
<== Row: 6, 龙龙, 李老師, 2
<== Row: 7, 小龙, 李老師, 2
<== Total: 7
Teacher(id=1, name=蔡老师, students=[Student(id=1, name=小明, teacherId=1), Student(id=3, name=小张, teacherId=1), Student(id=4, name=小李, teacherId=1), Student(id=5, name=小王, teacherId=1)])
Teacher(id=2, name=李老師, students=[Student(id=2, name=小红, teacherId=2), Student(id=6, name=龙龙, teacherId=2), Student(id=7, name=小龙, teacherId=2)])
Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@15b3e5b]
Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@15b3e5b]
```

## 11.2 按照查询嵌套查询

类似于Sql中的子查询:

mapper.xml文件:

```
1 <select id="getTeacherStudent2" resultMap="TeacherAndStudent2">
2     select * from teacher;
3 </select>
4
5 <resultMap id="TeacherAndStudent2" type="Teacher">
6     <result column="id" property="id"/>
7     <result column="name" property="name"/>
```



```

8      <!--
9          复杂的属性需要单独处理
10         对象: association
11         集合: collection
12         javaType="" : 为指定属性的类型
13         ofType="" : 为集合中的泛型信息
14     -->
15     <collection property="students" javaType="ArrayList" ofType="Student"
16     select="getStudentByTeacherId" column="id"/>
17 </resultMap>
18 <select id="getStudentByTeacherId" resultType="Student">
19     select * from student where teacher_id = #{teacherId};
20 </select>

```

结果展示:

```

Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@8b07145]
==> Preparing: select * from teacher;
==> Parameters:
<== Columns: id, name
<== Row: 1, 陈老师
==> Preparing: select * from student where teacher_id = ?;
==> Parameters: 1(Long)
<=== Columns: id, name, teacher_id
<=== Row: 1, 小明, 1
<=== Row: 3, 小张, 1
<=== Row: 4, 小李, 1
<=== Row: 5, 小王, 1
<=== Total: 4
<== Row: 2, 李老师
==> Preparing: select * from student where teacher_id = ?;
==> Parameters: 2(Long)
<=== Columns: id, name, teacher_id
<=== Row: 2, 小红, 2
<=== Row: 6, 龙龙, 2
<=== Row: 7, 小龙, 2
<=== Total: 3
<== Total: 2
Teacher(id=1, name=陈老师, students=[Student(id=1, name=小明, teacherId=0), Student(id=3, name=小张, teacherId=0), Student(id=4, name=小李, teacherId=0), Student(id=5, name=小王, teacherId=0),
Teacher(id=2, name=李老师, students=[Student(id=2, name=小红, teacherId=0), Student(id=6, name=龙龙, teacherId=0), Student(id=7, name=小龙, teacherId=0)])
Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@8b07145]

```

## 11.3 小结

1. 关联: association 【多对一】
2. 集合: collection 【一对多】
3. javaType & ofType
  - javaType 用来指定实体类中属性的类型;
  - 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;

掌握MySQL的引擎、InnoDB底层原理、索引、索引优化

## 12. 动态SQL

### 【工程9: mybatis-09】

- 动态 SQL 是 MyBatis 的强大特性之一!

**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;**

如果你之前用过 JSTL 或任何基于类 XML 语言的文本处理器, 你对动态 SQL 元素可能会感觉似曾相识。在 MyBatis 之前的版本中, 需要花时间了解大量的元素。借助功能强大的基于 OGNL 的表达式, MyBatis 3 替换了之前的大部分元素, 大大精简了元素种类, 现在要学习的元素种类比原来的一半还要少。

- if
- choose (when, otherwise)

- trim (where, set)
- foreach

## 搭建环境

```
1 CREATE TABLE blog
2 (
3     id varchar(50) NOT NULL COMMENT '博客ID',
4     title varchar(100) NOT NULL COMMENT '博客标题',
5     author varchar(30) NOT NULL COMMENT '博客作者',
6     create_time datetime NOT NULL COMMENT '创建时间',
7     views int(30) NOT NULL COMMENT '浏览量'
8 )engine=InnoDB default charset=utf8;
```

## 创建基础工程：

1. 导入Lombok;
2. 新建实体类：Blog;

```
1 @Data
2 public class Blog {
3     private int id;
4     private String title;
5     private String author;
6     private Date createTime;
7     private int views;
8 }
```

3. 建立Mapper接口;
4. 建立Mapper.xml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 <mapper namespace="com.longg.mapper.BlogMapper">
6
7
8 </mapper>
```

5. 在核心配置文件中绑定注册我们的mapper接口或文件;

## 12.1 IF

官网：

if

使用动态 SQL 最常见情景是根据条件包含 where 子句的一部分。比如：

```
<select id="findActiveBlogWithTitleLike"
    resultType="Blog">
    SELECT * FROM BLOG
    WHERE state = 'ACTIVE'
    <if test="title != null">
        AND title like #{title}
    </if>
</select>
```

这条语句提供了可选的查找文本功能。如果不传入“title”，那么所有处于“ACTIVE”状态的 BLOG 都会返回；如果传入了“title”参数，那么就会对“title”一列进行模糊查找并返回对应的 BLOG 结果（细心的读者可能会发现，“title”的参数值需要包含查找编码或通配符字符）。

如果希望通过“title”和“author”两个参数进行可选搜索该怎么办呢？首先，我想先将语句名称修改成更名副其实的名称；接下来，只需要加入另一个条件即可。

```
<select id="findActiveBlogLike"
    resultType="Blog">
    SELECT * FROM BLOG WHERE state = 'ACTIVE'
    <if test="title != null">
        AND title like #{title}
    </if>
    <if test="author != null and author.name != null">
        AND author_name like #{author.name}
    </if>
</select>
```

mapper.xml：

```
1  <select id="queryBlogIF" parameterType="map" resultType="Blog">
2      select * from blog where 1=1    /*写 1=1 仅仅只为了拼接and语句*/
3      <if test="title != null">
4          and title = #{title}
5      </if>
6      <if test="author != null">
7          and author = #{author}
8      </if>
9  </select>
```

测试类：

```
1  @Test
2  public void queryBlogIF(){
3      SqlSession sqlSession = MybatisUtils.getSqlSession();
4
5      BlogMapper mapper = sqlSession.getMapper(BlogMapper.class);
6
7      HashMap map = new HashMap();
8      map.put("title", "Java如此简单");
9      map.put("author", "long");
10
11     List<Blog> blogs = mapper.queryBlogIF(map);
12     for (Blog blog : blogs) {
13         System.out.println(blog);
14     }
15
16     sqlSession.close();
17 }
```

结果：

```
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@4df828d7]
==> Preparing: select * from blog where 1=1 and title = ? and author = ?
==> Parameters: Java如此简单(String), long(String)
<== Columns: id, title, author, create_time, views
<== Row: 88e1be885fe94dd2845ca3f37b7e9117, Java如此简单, long, 2020-09-27 16:27:15.0, 88
<== Total: 1
Blog(id=88e1be885fe94dd2845ca3f37b7e9117, title=Java如此简单, author=long, createTime=Sun Sep 27 16:27:15 CST 2020, views=88)
Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@4df828d7]
```

注释map.put("title","Java如此简单");的结果：

```

Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@b59d31]
==> Preparing: select * from blog where 1=1 and author = ?
==> Parameters: long(String)
<== Columns: id, title, author, create_time, views
<== Row: 93b70c0a14e146d885e1531cd96ade61, MyBatis如此简单, long, 2020-09-27 16:27:15.0, 866
<== Row: 88e1be885fe94dd2845ca3f37b7e9117, Java如此简单, long, 2020-09-27 16:27:15.0, 88
<== Row: 19b4baa74c8a4cbc9c461d92e4ae2288, Spring从入门到放弃, long, 2020-09-27 16:27:15.0, 666
<== Row: ebceb80b40ef4027af8d6ff9a72c756e, SpringBoot从入门到放弃, long, 2020-09-27 16:27:15.0, 99
<== Total: 4
Blog(id=93b70c0a14e146d885e1531cd96ade61, title=MyBatis如此简单, author=long, createTime=Sun Sep 27 16:27:15 CST 2020, views=866)
Blog(id=88e1be885fe94dd2845ca3f37b7e9117, title=Java如此简单, author=long, createTime=Sun Sep 27 16:27:15 CST 2020, views=88)
Blog(id=19b4baa74c8a4cbc9c461d92e4ae2288, title=Spring从入门到放弃, author=long, createTime=Sun Sep 27 16:27:15 CST 2020, views=666)
Blog(id=ebceb80b40ef4027af8d6ff9a72c756e, title=SpringBoot从入门到放弃, author=long, createTime=Sun Sep 27 16:27:15 CST 2020, views=99)
Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@b59d31]
Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@b59d31]

```

## 12.2 choose (when, otherwise)

有时候，我们不想使用所有的条件，而只是想从多个条件中选择一个使用。针对这种情况，MyBatis 提供了 `choose` 元素，它有点像 Java 中的 `switch` 语句。

mapper.xml:

```

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

```

测试类:

```

1  /**
2   * 测试动态sql的Choose标签
3   */
4  @Test
5  public void queryBlogChoose(){
6      SqlSession sqlSession = MybatisUtils.getSqlSession();
7
8      BlogMapper mapper = sqlSession.getMapper(BlogMapper.class);
9
10     HashMap map = new HashMap();
11     //      map.put("title","Java如此简单");
12     //      map.put("author","long");
13     map.put("views",666);
14
15     List<Blog> blogs = mapper.queryBlogChoose(map);
16     for (Blog blog : blogs) {
17         System.out.println(blog);
18     }
19
20     sqlSession.close();
21 }

```

当map同时给title和author两个参数时，只选择第一个 `when` 中的语句执行，结果如下：

```

Created connection 1660794022.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@62fdb4a6]
==> Preparing: select * from blog where 1=1 and title = ?
==> Parameters: Java如此简单(String)
<== Columns: id, title, author, create_time, views
<== Row: 88e1be885fe94dd2845ca3f37b7e9117, Java如此简单, long, 2020-09-27 16:27:15.0, 88
<== Total: 1
Blog(id=88e1be885fe94dd2845ca3f37b7e9117, title=Java如此简单, author=long, createTime=Sun Sep 27 16:27:15 CST 2020, views=88)
Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@62fdb4a6]
Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@62fdb4a6]

```

当map只给title和author两个参数中的一个时，选择对应的 `when` 中的语句执行，结果如下：

```

Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@11e21d0e]
==> Preparing: select * from blog where 1=1 and author = ?
==> Parameters: long(String)
<== Columns: id, title, author, create_time, views
<== Row: 93b70c0a14e146d885e1531cd96ade61, MyBatis如此简单, long, 2020-09-27 16:27:15.0, 866
<== Row: 88e1be885fe94dd2845ca3f37b7e9117, Java如此简单, long, 2020-09-27 16:27:15.0, 88
<== Row: 19b4baa74c8a4cbc9c461d92e4ae2288, Spring从入门到放弃, long, 2020-09-27 16:27:15.0, 666
<== Row: ebceb80b40ef4027af8d6ff9a72c756e, SpringBoot从入门到放弃, long, 2020-09-27 16:27:15.0, 99
<== Total: 4
Blog(id=93b70c0a14e146d885e1531cd96ade61, title=MyBatis如此简单, author=long, createTime=Sun Sep 27 16:27:15 CST 2020, views=866)

```

```

Opening JDBC Connection
Created connection 1660794022.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@62fdb4a6]
==> Preparing: select * from blog where 1=1 and title = ?
==> Parameters: Java如此简单(String)
<== Columns: id, title, author, create_time, views
<== Row: 88e1be885fe94dd2845ca3f37b7e9117, Java如此简单, long, 2020-09-27 16:27:15.0, 88
<== Total: 1
Blog(id=88e1be885fe94dd2845ca3f37b7e9117, title=Java如此简单, author=long, createTime=Sun Sep 27 16:27:15 CST 2020, views=88)
Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@62fdb4a6]
Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@62fdb4a6]

```

当map不给参数参数时，choose会选择执行最后的 `otherwise` 中的语句执行，结果如下：(如果不给views参数则查询为空)

```

PooledDataSource forcefully closed/removed all connections.
Opening JDBC Connection
Created connection 300031246.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@11e21d0e]
==> Preparing: select * from blog where 1=1 and views = ?
==> Parameters: null
<== Total: 0
Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@11e21d0e]
Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@11e21d0e]

```

```

Created connection 300031246.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@11e21d0e]
==> Preparing: select * from blog where 1=1 and views = ?
==> Parameters: 666(Integer)
<== Columns: id, title, author, create_time, views
<== Row: 93b70c0a14e146d885e1531cd96ade61, MyBatis如此简单, long, 2020-09-27 16:27:15.0, 666
<== Row: 19b4baa74c8a4cbc9c461d92e4ae2288, Spring从入门到放弃, long, 2020-09-27 16:27:15.0, 666
<== Total: 2
Blog(id=93b70c0a14e146d885e1531cd96ade61, title=MyBatis如此简单, author=long, createTime=Sun Sep 27 16:27:15 CST 2020, views=666)
Blog(id=19b4baa74c8a4cbc9c461d92e4ae2288, title=Spring从入门到放弃, author=long, createTime=Sun Sep 27 16:27:15 CST 2020, views=666)
Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@11e21d0e]
Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@11e21d0e]

```

## 12.3 trim (where, set)

- `where` 元素只会在子元素返回任何内容的情况下才插入“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。不需要用“1=1”来拼接where语句了

mapper.xml:

```

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

```

测试类：

```
1  /**
2   * 测试动态sql的where标签
3   */
4  @Test
5  public void queryBlogWhere(){
6      SqlSession sqlSession = MybatisUtils.getSqlSession();
7
8      BlogMapper mapper = sqlSession.getMapper(BlogMapper.class);
9
10     HashMap map = new HashMap();
11     map.put("title", "Java如此简单");
12     map.put("author", "long");
13
14     List<Blog> blogs = mapper.queryBlogWhere(map);
15     for (Blog blog : blogs) {
16         System.out.println(blog);
17     }
18
19     sqlSession.close();
20 }
```

两个条件均成立的情况，结果如下：

```
Created connection 500179317.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@1dd02175]
==> Preparing: select * from blog WHERE title = ? and author = ?
==> Parameters: Java如此简单(String), long(String)
<== Columns: id, title, author, create_time, views
<== Row: 88e1be885fe94dd2845ca3f37b7e9117, Java如此简单, long, 2020-09-27 16:27:15.0, 88
<== Total: 1
Blog(id=88e1be885fe94dd2845ca3f37b7e9117, title=Java如此简单, author=long, createTime=Sun Sep 27 16:27:15 CST 2020, views=88)
Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@1dd02175]
```

两个条件均不成立的情况，结果如下：

```
Created connection 824208363.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@31206beb]
==> Preparing: select * from blog
==> Parameters:
<== Columns: id, title, author, create_time, views
<== Row: 93b70c0a14e146d885e1531cd96ade61, MyBatis如此简单, long, 2020-09-27 16:27:15.0, 666
<== Row: 88e1be885fe94dd2845ca3f37b7e9117, Java如此简单, long, 2020-09-27 16:27:15.0, 88
<== Row: 19b4baa74c8a4cbc9c461d92e4ae2288, Spring从入门到放弃, long, 2020-09-27 16:27:15.0, 666
<== Row: ebceb80b40ef4027af8d6ff9a72c756e, SpringBoot从入门到放弃, long, 2020-09-27 16:27:15.0, 99
<== Total: 4
Blog(id=93b70c0a14e146d885e1531cd96ade61, title=MyBatis如此简单, author=long, createTime=Sun Sep 27 16:27:15 CST 2020, views=666)
Blog(id=88e1be885fe94dd2845ca3f37b7e9117, title=Java如此简单, author=long, createTime=Sun Sep 27 16:27:15 CST 2020, views=88)
Blog(id=19b4baa74c8a4cbc9c461d92e4ae2288, title=Spring从入门到放弃, author=long, createTime=Sun Sep 27 16:27:15 CST 2020, views=666)
Blog(id=ebceb80b40ef4027af8d6ff9a72c756e, title=SpringBoot从入门到放弃, author=long, createTime=Sun Sep 27 16:27:15 CST 2020, views=99)
Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@31206beb]
```

- 用于动态更新语句的类似于 `where` 标签的解决方案叫做 `set`。`set` 元素可以用于动态包含需要更新的列，忽略其它不更新的列；

mapper.xml：

```

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>

```

测试类：

```

1 /**
2  * 测试动态sql的set标签
3  */
4 @Test
5 public void updateBlogSet(){
6     SqlSession sqlSession = MybatisUtils.getSqlSession();
7
8     BlogMapper mapper = sqlSession.getMapper(BlogMapper.class);
9
10    HashMap map = new HashMap();
11    map.put("title", "Java如此简单2");
12    map.put("author", "long2");
13    map.put("id", "ebceb80b40ef4027af8d6ff9a72c756e");
14
15    mapper.updateBlogSet(map);
16
17    sqlSession.close();
18 }

```

两个条件均成立的情况，结果如下：

```

Opening JDBC Connection
Created connection 2142080121.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@7fad8c79]
==> Preparing: update blog SET title = ?, author = ? where id = ?
==> Parameters: Java如此简单2(String), long2(String), ebceb80b40ef4027af8d6ff9a72c756e(String)
<== Updates: 1
Rolling back JDBC Connection [com.mysql.jdbc.JDBC4Connection@7fad8c79]

```

只有第一个条件均成立的情况，结果如下：

```

Opening JDBC Connection
Created connection 1906808037.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@71a794e5]
==> Preparing: update blog SET title = ? where id = ?
==> Parameters: Java如此简单2(String), ebceb80b40ef4027af8d6ff9a72c756e(String)
<== Updates: 1
Rolling back JDBC Connection [com.mysql.jdbc.JDBC4Connection@71a794e5]
Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@71a794e5]
Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@71a794e5]

```

- 也可以通过自定义 trim 元素来定制 `where` 元素和 `set` 元素的功能

```

1 <trim prefix="WHERE" prefixOverrides="AND |OR ">
2     ...
3 </trim>

```

```

1 <trim prefix="SET" suffixOverrides=",">
2   ...
3 </trim>

```

## 12.4 Foreach

- 动态 SQL 的另一个常见使用场景是对集合进行遍历（尤其是在构建 IN 条件语句的时候）
- `foreach` 元素的功能非常强大，它允许你指定一个集合，声明可以在元素体内使用的集合项（item）和索引（index）变量。它也允许你指定开头与结尾的字符串以及集合项迭代之间的分隔符。这个元素也不会错误地添加多余的分隔符，看它多智能！
- 你可以将任何可迭代对象（如 List、Set 等）、Map 对象或者数组对象作为集合参数传递给 `foreach`。当使用可迭代对象或者数组时，index 是当前迭代的序号，item 的值是本次迭代获取到的元素。当使用 Map 对象（或者 Map.Entry 对象的集合）时，index 是键，item 是值。

mapper.xml：（map传参）

```

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

```

测试类：

```

1 /**
2  * 测试使用Foreach
3  */
4 @Test
5 public void queryBlogForeach(){
6     SqlSession sqlSession = MybatisUtils.getSqlSession();
7
8     BlogMapper mapper = sqlSession.getMapper(BlogMapper.class);
9
10    HashMap map = new HashMap();
11
12    ArrayList<Integer> ids = new ArrayList<Integer>();
13    ids.add(1);
14    ids.add(2);
15    ids.add(3);
16
17    map.put("ids",ids);
18
19    List<Blog> blogs = mapper.queryBlogForeach(map);
20    for (Blog blog : blogs) {
21        System.out.println(blog);
22    }
23
24    sqlSession.close();
25 }

```



结果如下:

```
Opening JDBC Connection
Created connection 1660794022.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@62fdb4a6]
==> Preparing: select * from blog WHERE ( id = ? or id = ? or id = ? )
==> Parameters: 1(Integer), 2(Integer), 3(Integer)
<== Columns: id, title, author, create_time, views
<== Row: 1, MyBatis如此简单, long, 2020-09-27 16:27:15.0, 666
<== Row: 2, Java如此简单, long, 2020-09-27 16:27:15.0, 88
<== Row: 3, Spring从入门到放弃, long, 2020-09-27 16:27:15.0, 666
<== Total: 3
Blog(id=1, title=MyBatis如此简单, author=long, createTime=Sun Sep 27 16:27:15 CST 2020, views=666)
Blog(id=2, title=Java如此简单, author=long, createTime=Sun Sep 27 16:27:15 CST 2020, views=88)
Blog(id=3, title=Spring从入门到放弃, author=long, createTime=Sun Sep 27 16:27:15 CST 2020, views=666)
Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@62fdb4a6]
```

mapper.xml: (int数组传参)

```
1 <select id="queryBlogForeach" parameterType="int[]" resultType="Blog">
2     select * from blog
3     <where>
4         <foreach collection="array" item="id" open="(" close=")"
separator="or">
5             id = #{id}
6         </foreach>
7     </where>
8 </select>
```

mapper.xml: (List传参)

```
1 <select id="queryBlogForeach" parameterType="list" resultType="Blog">
2     select * from blog
3     <where>
4         <foreach collection="list" item="id" open="(" close=")"
separator="or">
5             id = #{id}
6         </foreach>
7     </where>
8 </select>
```

mapper.xml: (List传参, SQL为in语句)

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

## 12.5 SQL片段

- 有时候，我们可能会将一些功能的部分抽取出来，实现SQL的复用；
- **用法：**使用SQL标签抽取公共部分，在需要使用地方使用include标签引用即可；
- **注意事项：**最好基于单表来定义SQL片段；不要存在where标签；

mapper.xml:

```

1  <select id="queryBlogSQL" resultType="Blog" parameterType="map">
2      select * from blog
3      <where>
4          <include refid="if-title-author"></include>
5      </where>
6  </select>
7
8  <sql id="if-title-author">
9      <if test="title != null">
10         and title = #{title}
11      </if>
12      <if test="author != null">
13         and author = #{author}
14      </if>
15  </sql>

```

测试类：

```

1  /**
2   * 测试使用动态SQL片段拼接SQL语句
3   */
4  @Test
5  public void queryBlogSQL(){
6      SqlSession sqlSession = MybatisUtils.getSqlSession();
7
8      BlogMapper mapper = sqlSession.getMapper(BlogMapper.class);
9
10     HashMap map = new HashMap();
11     map.put("title", "Java如此简单");
12     map.put("author", "long");
13
14     List<Blog> blogs = mapper.queryBlogSQL(map);
15     for (Blog blog : blogs) {
16         System.out.println(blog);
17     }
18
19     sqlSession.close();
20 }

```

结果如下：

```

Created connection 500179317.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@1dd02175]
==> Preparing: select * from blog WHERE title = ? and author = ?
==> Parameters: Java如此简单(String), long(String)
<==      Columns: id, title, author, create_time, views
<==      Row: 88e1be885fe94dd2845ca3f37b7e9117, Java如此简单, long, 2020-09-27 16:27:15.0, 88
<==      Total: 1
Blog(id=88e1be885fe94dd2845ca3f37b7e9117, title=Java如此简单, author=long, createTime=Sun Sep 27 16:27:15 CST 2020, views=88)
Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@1dd02175]

```

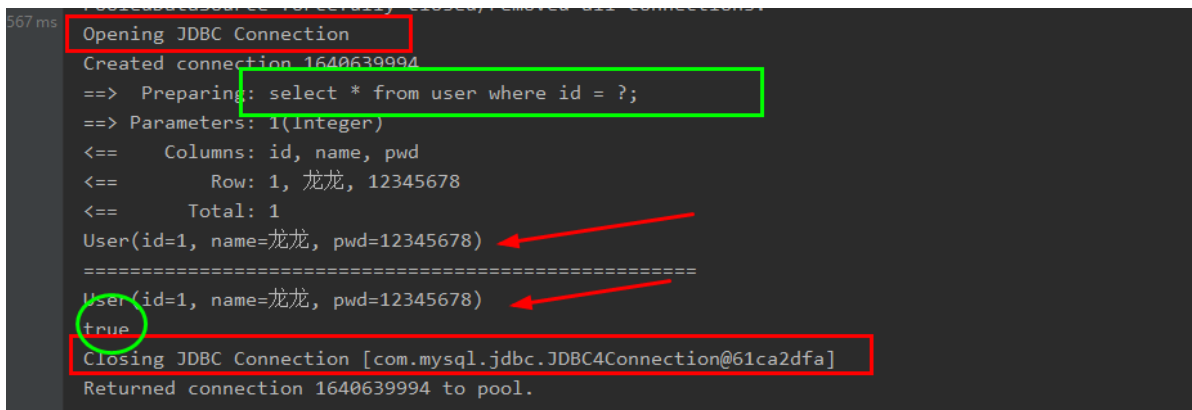
## 13. 缓存

### 13.1 MyBatis缓存

- MyBatis 内置了一个强大的事务性查询缓存机制，它可以非常方便地配置和定制；
- 为了使它更加强大而且易于配置，我们对 MyBatis 3 中的缓存实现进行了许多改进；
- MyBatis系统中默认定义了两级缓存：一级缓存和二级缓存；
  - 默认情况下，只启用了本地的会话缓存（也叫一级缓存，是SqlSession级别的缓存），它仅仅对一个会话中的数据进行缓存；
  - 二级缓存需要手动开启配置，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；

### 13.2 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；
  - 如果需要获取相同的数据，直接从缓存中拿走即可，没必要再到数据库中查询；
- 测试：开启日志，测试在一个session中查询两次相同的记录，查看日志输出情况；



```
607 ms
Opening JDBC Connection
Created connection 1640639994
==> Preparing: select * from user where id = ?;
==> Parameters: 1(Integer)
<== Columns: id, name, pwd
<== Row: 1, 龙龙, 12345678
<== Total: 1
User(id=1, name=龙龙, pwd=12345678)
=====
User(id=1, name=龙龙, pwd=12345678)
true
Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@61ca2dfa]
Returned connection 1640639994 to pool.
```

- 缓存失效的情况：
  - 查询不同的数据；
  - 增删改操作可能会改变数据库中原来的数据，所以必定会刷新缓存；

```
556ms PooledDataSource forcefully closed/removed all connections.
556ms Opening JDBC Connection
Created connection 1640639994.
==> Preparing: select * from user where id = ?;
==> Parameters: 1(Integer)
<== Columns: id, name, pwd
<== Row: 1, 龙龙, 12345678
<== Total: 1
User(id=1, name=龙龙, pwd=12345678)
=====
==> Preparing: update user set name = ?, pwd = ? where id = ?;
==> Parameters: longlong(String), 8888(String), 3(Integer)
<== Updates: 1
=====
==> Preparing: select * from user where id = ?;
==> Parameters: 1(Integer)
<== Columns: id, name, pwd
<== Row: 1, 龙龙, 12345678
<== Total: 1
User(id=1, name=龙龙, pwd=12345678)
false
```

- 查询不同的mapper.xml;
- 手动清理缓存; ;

```
1 sqlSession.clearCache(); // 手动清理缓存
```

```
PooledDataSource forcefully closed/removed all connections.
Opening JDBC Connection
Created connection 1640639994.
==> Preparing: select * from user where id = ?;
==> Parameters: 1(Integer)
<== Columns: id, name, pwd
<== Row: 1, 龙龙, 12345678
<== Total: 1
User(id=1, name=龙龙, pwd=12345678)
=====
==> Preparing: select * from user where id = ?;
==> Parameters: 1(Integer)
<== Columns: id, name, pwd
<== Row: 1, 龙龙, 12345678
<== Total: 1
User(id=1, name=龙龙, pwd=12345678)
false
Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@61ca2dfa]
```

- 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的这个过程有效；

## 13.3 二级缓存

- 二级缓存也叫全局缓存，一级缓存的作用域太低了，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以及缓存中；
  - 如果当前会话关闭了，这个会话对应的一级缓存就没有了，但是我们想要的是，会话关闭时对应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；
  - 不同的mapper查出的数据会放在自己对应的缓存中；
- 步骤：

- 开启全局缓存;

```
1 <!--显示的开启全局缓存-->
2 <setting name="cacheEnabled" value="true"/>
```

- 在mapper.xml中开启二级缓存

```
1 <!--开启二级缓存-->
2 <cache/>
3 <!--自定义二级缓存-->
4 <cache eviction="FIFO" flushInterval="60000" size="512" readOnly="true"/>
```

- 测试

```
Opening JDBC Connection
Created connection 1413378318.
==> Preparing: select * from user where id = ?;
==> Parameters: 1(Integer)
<== Columns: id, name, pwd
<== Row: 1, 龙龙, 12345678
<-- Total: 1
User(id=1, name=龙龙, pwd=12345678)
Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@543e710e]
Returned connection 1413378318 to pool.
Cache Hit Ratio [com.longg.mapper.UserMapper]: 0.5
User(id=1, name=龙龙, pwd=12345678)
```

- 注意: 我们需要将实体类序列化! 否则可能会报错!

```
1 Caused by: java.io.NotSerializableException: com.longg.pojo.User
```

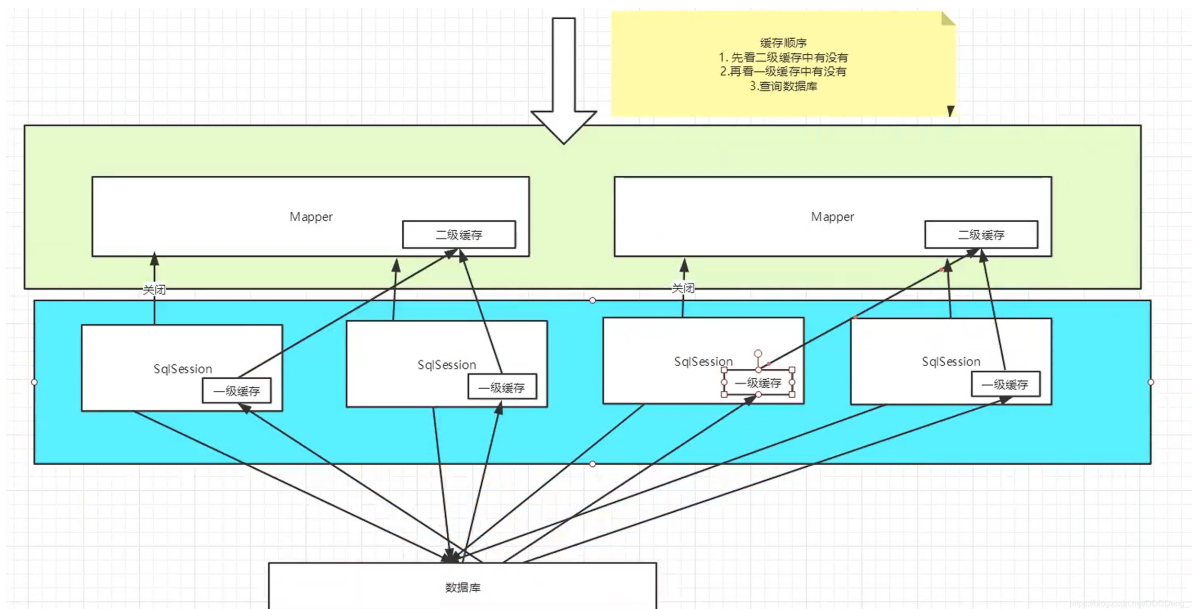
#### 序列化的方法:

```
public class User implements Serializable {
```

- 小结:
  - 只要开启了二级缓存, 在同一个mapper下就会有效;
  - 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, 才会被提交到二级缓存中;

## 13.4 缓存的原理【重点】

- 一级缓存只要SqlSession会话建立即随之建立, SqlSession在数据库中查询的书籍会自动保存在一级缓存中, 第二次查询的时候直接在一级缓存中取出即可, 在SqlSession被关闭的时候一级缓存也就随之关闭了; 【一级缓存对应SqlSession】
- 二级缓存在同一个mapper下都能生效, 再SqlSession建立时所有的查询数据都会先放在一级缓存中, 只有当会话被关闭的时候, 一级缓存内的数据才能被保存到二级缓存中, 之后的查询都能在二级缓存中去取对应的数据; 【二级缓存对应一个mapper】
- 数据存放的顺序: 数据库、一级缓存、二级缓存;
- 用户查询缓存的顺序: 用户先在二级缓存中是否有, 没有则进入一级缓存中找是否有, 最后才进入数据库中查询; 【与数据存放的顺序相反】



## 13.5 自定义缓存 (EhCache)

EhCache 是一个纯Java的进程内缓存框架，具有快速、精干等特点，是Hibernate中默认的CacheProvider。