

# 修改驱动名重点!!!

在5.0版本之前 是 com.mysql.jdbc.Driver  
在8.0 就需要 加 cj 即com.mysql.cj.jdbc.Driver

## 打包

```
m pom.xml (Mybatis2) x
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema
4      xsi:schemaLocation="http://maven.apache.o
5      <modelVersion>4.0.0</modelVersion>
6
7      <groupId>org.example</groupId>
8      <artifactId>Mybatis2</artifactId>
9      <version>1.0-SNAPSHOT</version>
10     <packaging>jar</packaging>
11     <properties>
```

## 引入依赖

```
<dependencies>
<!--      mybatis-->
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.5.9</version>
    </dependency>
<!--      测试-->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
    </dependency>
<!--      mysql 驱动-->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.29</version>
    </dependency>
<!--      日志-->
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.12</version>
    </dependency>
</dependencies>
```

# 核心配置文件

习惯命名为mybatis-config.xml

配置文件去官方文档找

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
<!-- 连接jdbc.properties-->
  <properties resource="jdbc.properties"></properties>
<!-- 类型别名-->
  <typeAliases>
    <package name="mappers"/>
  </typeAliases>
<!-- 配置连接数据库的环境-->
<!-- environments:配置多个连接数据库的环境
    default: 设置默认使用环境-->
  <environments default="development">
    <!-- environment: 配置某个具体环境
        id: 表示连接数据库环境的唯一标识, 不能重复
    -->
    <environment id="development">
      <!-- 设置事务管理方式
          type=JDBC/MANAGED
          JDBC标识当前环境执行sql使用jdbc原生的事务管理方式, 事务提交回
          滚要手动配置
          MANAGED: 被管理, 例如Spring
        -->
      <transactionManager type="JDBC"/>
      <!-- 配置数据源
          type: 设置数据源类型
          type=POOLED/UNPOOLED/JNDI
          POOLED: 表示使用数据库连接池缓存数据库连接
          UNPOOLED: 表示不使用数据库连接池
          JNDI: 表示使用上下文中的数据源
        -->
      <dataSource type="POOLED">
        <!-- 设置连接数据库的驱动-->
        <property name="driver" value="${jdbc.driver}"/>
        <!-- 设置连接数据库的连接地址-->
        <property name="url" value="${jdbc.url}"/>
        <!-- 设置连接数据库的用户名-->
        <property name="username" value="${jdbc.username}"/>
        <!-- 设置连接数据库的密码-->
        <property name="password" value="${jdbc.password}"/>
      </dataSource>
    </environment>
  </environments>
<!-- 引入映射文件-->
  <mappers>
    <package name="mappers"/>
  </mappers>
```

```
</configuration>
```

注意driver是com.mysql.cj.jdbc.Driver

加载类“com.mysql.jdbc.Driver”。这是弃用。新的驱动程序类是' com.mysql.cj.jdbc.Driver'。驱动程序是通过SPI自动注册的，手动加载驱动程序类通常是不必要的。

url是jdbc: mysql: //localhost: 3306/databaseName

## 引入properties文件

jdbc.properties

```
jdbc.driver=com.mysql.cj.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/jiaoxue?characterEncoding=utf-8
jdbc.username=root
jdbc.password=mdjfbzyj515
```

核心配置文件

mybatis-config.xml

连接properties

```
<properties resource="jdbc.properties"/>
```

引用

```
<dataSource type="POOLED">
<!--      设置连接数据库的驱动-->
    <property name="driver" value="${jdbc.driver}"/>
<!--      设置连接数据库的连接地址-->
    <property name="url" value="${jdbc.url}"/>
    <!--      设置连接数据库的用户名-->
    <property name="username" value="${jdbc.username}"/>
<!--      设置连接数据库的密码-->
    <property name="password" value="${jdbc.password}"/>
</dataSource>
```

## 标签顺序

```
<!--      标签顺序
The content of element type "configuration" must match
"(properties?,settings?,typeAliases?,typeHandlers?,
objectFactory?,objectWrapperFactory?,reflectorFactory?,
plugins?,environments?,databaseIdProvider?,mappers?)".-->
```

## 设置类型别名，不区分大小写！！

如果不写别名alias="User"，则默认别名为类名User

```

<!-- 设置类型别名-->
<typeAliases>
    <typeAlias type="object.User" alias="User"></typeAlias>
</typeAliases>

```

也可以给标签typeAliases的属性package以包为单位，将包下所有类设置默认类型别名

```

<typeAliases>
    <package name="object"/>
</typeAliases>

```

映射文件中

```

<!-- resultMap设置默认的映射关系，resultMap设置自定义的映射关系-->
<select id="findId" resultType="object.User">
    select * from t_user where id=7
</select>

```

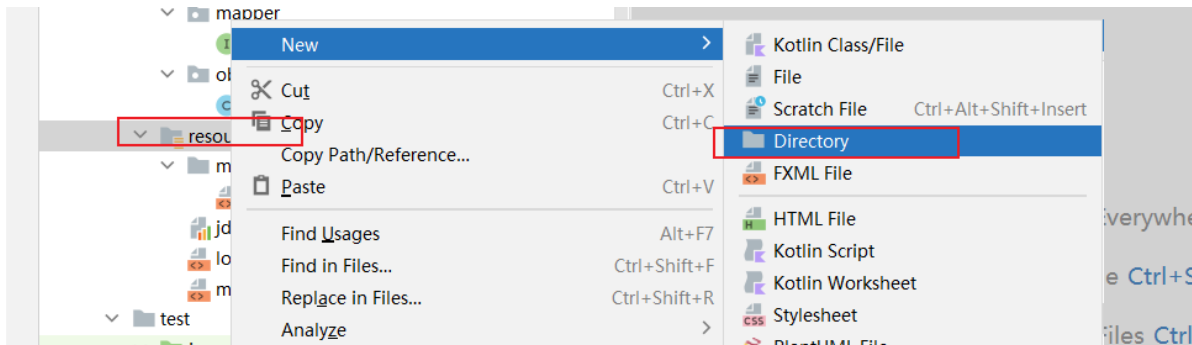
可修改为

resultType="User"或者"user"，不区分大小写！！

对应的是配置文件中类型别名的类型

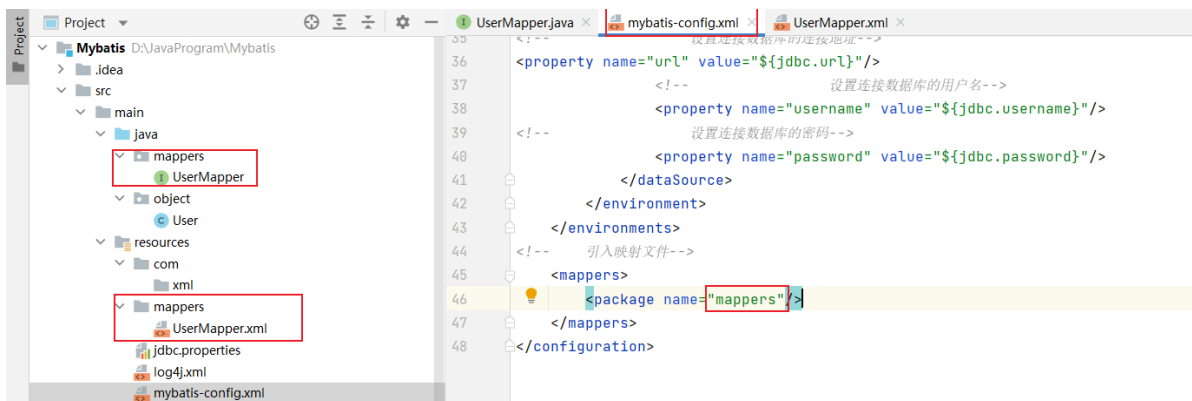
## 映射文件导包

resources没用包，只能导入文件



则不能包.包.包.文件，否则就只是一个文件不是多个文件下的文件

得是包/包/包/文件



也可以导入整个包

①要求mapper接口所在的包要和映射文件所在的包一致！！！即包名一样

②mapper接口要和映射文件的名字一致

```
<!-- 引入映射文件-->
<mappers>
  <package name="mappers"/>
</mappers>
```

## ORM

### object relationship mapper

### 对象关系映射

java概念	数据库概念
类	表
属性	字段/列
对象	记录/行

## Mapper映射

### 实体类

```
//字段名和属性名是一致的！
```

### 接口

实体类名+Mapper.java

这相当于DAO (data access object)对象访问数据的规范

```
/*
默认实现User类
*/
```

### 映射文件

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
  PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="org.mybatis.example.BlogMapper">
  <select id="selectBlog" resultType="Blog">
    select * from Blog where id = #{id}
  </select>
</mapper>
```

实体类名+Mapper.xml

这相当于DAOImplement, DAO的实现

```
/*
Mybatis面向接口编程的两个一致
1映射文件的namespace要和mapper接口的全类名保持一致
2映射文件的SQL语句的id要和mapper接口中的方法名一致
*/
```

## 映射文件连接接口

命名空间对应接口，id对应接口方法

```
<mapper namespace="mapper.UserMapper">
    <insert id="insertUser">
        insert into t_user values(null , 'admit', '123456', 23, '123456@qq.com')
    </insert>
```

## 映射文件连接表

sql语句连接表

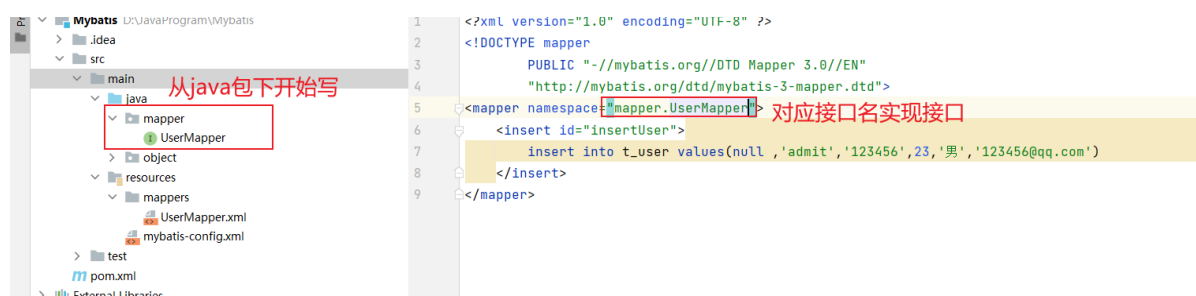
```
<insert id="insertUser">
    insert into t_user values(null , 'admit', '123456', 23, '123456@qq.com')
</insert>
```

## 映射文件连接实体类

```
<!--      resultType设置默认的映射关系，resultMap设置自定义的映射关系-->
<select id="findId" resultType="object.User">
    select * from t_user where id=7
</select>
```

## 流程

①映射文件的namespace命名空间为同级包下包名++（重复）+类名



②

方法名和id一致

```
public interface UserMapper
{
    /*
    Mybatis 面向接口编程的两个一致
    1映射文件的namespace要和mapper接口的全类名保持一致
    2映射文件的SQL 语句的id要和mapper接口中的方法名一致
    */
    int insertUser();
}
```

```
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="mapper.UserMapper">
6     <insert id="insertUser">
7         insert into t_user values(null , 'admit', '123456', 23, '男', '123456@qq.com')
8     </insert>
9 </mapper>
```

引入映射文件是以同级包名下忽略不写然后包名+/(重复) +映射文件名+.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 <configuration>
6     <!-- 配置连接数据库的环境 -->
7     <environments default="development">
8         <environment id="development">
9             <transactionManager type="JDBC"/>
10            <dataSource type="POOLED">
11                <property name="driver" value="com.mysql.jdbc.Driver"/>
12                <property name="url" value="jdbc:mysql://localhost:3306/jiaoxue"/>
13                <property name="username" value="root"/>
14                <property name="password" value="mdjfbzyj515"/>
15            </dataSource>
16        </environment>
17    </environments>
18    <!-- 引入映射文件 -->
19    <mappers>
20        <mapper resource="mappers/UserMapper.xml"/>
21    </mappers>
22 </configuration>
```

## 调试

```
//加载核心配置文件,以输入流形式获得
InputStream resourceAsStream = Resources.getResourceAsStream("mybatis-
config.xml");
//获取sqlSessionFactoryBuilder sql会话工厂建造者
SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new
SqlSessionFactoryBuilder();
//获取sqlSessionFactory sql会话工厂
SqlSessionFactory build = sqlSessionFactoryBuilder.build(resourceAsStream);
//获取sqlSession 打开sql会话
SqlSession sqlSession = build.openSession();
```

```
//获取mapper接口对象,代理模式获取接口实现类
UserMapper mapper = sqlSession.getMapper(UserMapper.class);
//测试功能
int i = mapper.insertUser();
//提交事务
sqlSession.commit();
```

## 自动提交

```
//获取sqlSession                打开sql会话                默认自动提交为false,想自动提交设
为true
sqlSession sqlSession = build.openSession(true);
```

## 日志

### 引入依赖

```
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.12</version>
</dependency>
```

### 配置文件log4j.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
  <appender name="STDOUT" class="org.apache.log4j.ConsoleAppender">
    <param name="Encoding" value="UTF-8" />
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern" value="%-5p %d{MM-dd HH:mm:ss,SSS}
%m (%F:%L) \n" />
    </layout>
  </appender>
  <logger name="java.sql">
    <level value="debug" />
  </logger>
  <logger name="org.apache.ibatis">
    <level value="info" />
  </logger>
  <root>
    <level value="debug" />
    <appender-ref ref="STDOUT" />
  </root>
</log4j:configuration>
```

### 日志级别

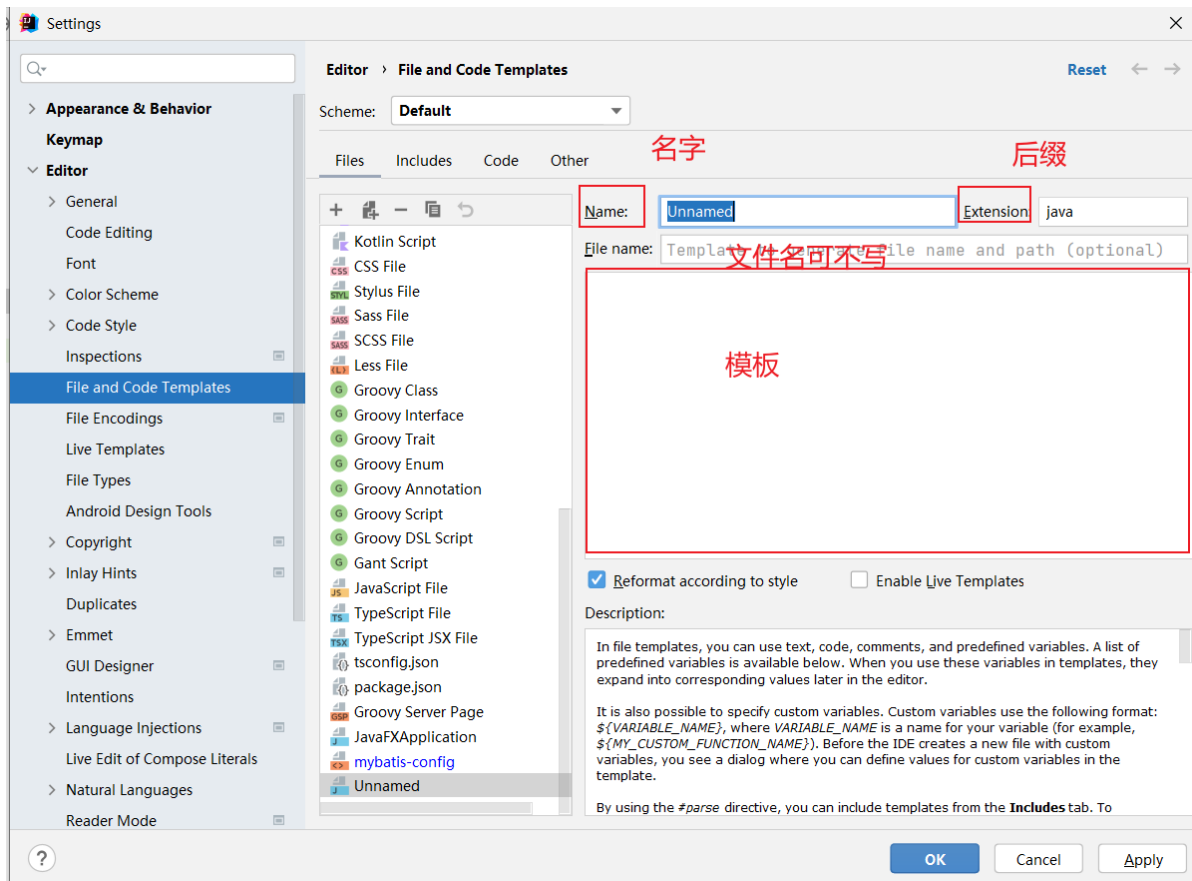
从左到右打印内容越来越详细

fatal致命, error错误, warn警告, info信息, debug调试



# 配置 核心配置/映射文件文件 模板

settings-----editor-----codeStyle-----file and code templates



## 重点

## 原理

获取连接

```
//原先获取连接的方式
public class getConnection {
    static Connection getConnection() throws Exception {
        //1读取jdbc.properties基本信息
        InputStream resourceAsStream =
        ClassLoader.getSystemClassLoader().getResourceAsStream("jdbc.properties");
        Properties properties = new Properties();
        //输入流读取
        properties.load(resourceAsStream);
        //加载驱动
        Class.forName(properties.getProperty("jdbc.driver"));
        //获取连接
        return
        DriverManager.getConnection(properties.getProperty("jdbc.url"), properties.getProperty("jdbc.username"), properties.getProperty("jdbc.password"));
    }
}
```

执行sql语句

```

public class CRUD
{
    @Test
    public void update() throws Exception {
        //获取连接
        Connection connection = getConnection.getConnection();
        //编写sql语句
        String sql="update t_user set username=? where id=6";
        //获取预编译实例
        PreparedStatement preparedStatement = connection.prepareStatement(sql);
        //填充占位符
        preparedStatement.setObject(1,"庄消津");
        //执行
        preparedStatement.execute();
    }
}

```

### 获取返回结果

```

@Test
public void getUser() throws Exception
{
    //获取连接
    Connection connection = getConnection.getConnection();
    //编写sql语句
    String sql="select id,username,password,age,email from t_user where id=6";
    //获取预编译实例
    PreparedStatement preparedStatement = connection.prepareStatement(sql);
    //编译返回结果集
    ResultSet resultSet = preparedStatement.executeQuery();
    //获取结果集的元数据
    ResultSetMetaData metaData = resultSet.getMetaData();
    //通过元数据得到列数
    int columnCount = metaData.getColumnCount();
    while (resultSet.next()) //判断结果集是否有下一个数据，有的话指针下移且返回true
    {
        //创建对象
        User user=new User();
        for (int i = 0; i < columnCount; i++)
        {
            //通过结果集获取列值，一定要配合resultSet.next () 使用
            Object value = resultSet.getObject(i + 1);
            //通过元数据获取sql语句中列名getColumnName,
            // 注意getColumnLabel获取的是如果sql语句列名有as则获取别名
            String columnName = metaData.getColumnName(i + 1);
            //通过反射获得属性
            Field declaredField = user.getClass().getDeclaredField(columnName);
            //关闭安全检查提高反射速度
            declaredField.setAccessible(true);
            //设置属性
            declaredField.set(user, value);
        }
        System.out.println(user);
    }
}

```

## 获取参数两种方式

`${}`本质字符串拼接，注意如果是字符串sql语句要有"，注意单引号问题

```
10 <select id="findUser" resultType="User">
11     select * from t_user where id=${zdasdasasf}
mapper > select

> Tests passed: 1 of 1 test - 1 sec 176 ms
"C:\Program Files\Java\jdk1.8.0_241\bin\java.exe" ...
5 DEBUG 07-24 16:04:59,282 ==> Preparing: select * from t_user where id=6 (BaseJdbcLogger.java:137)
```

`#{}` 本质占位符赋值

```
10 <select id="findUser" resultType="User">
11     select * from t_user where id=#{zdasdasasf}
12 </select>
mapper > select

>> Tests passed: 1 of 1 test - 1 sec 131 ms
"C:\Program Files\Java\jdk1.8.0_241\bin\java.exe" ...
ns DEBUG 07-24 16:08:41,515 ==> Preparing: select * from t_user where id=? (BaseJdbcLogger.java:137)
```

## 一个参数

mapper接口方法的参数为单个字面量类型

可以通过任意的字符串获取参数值，也就是只有一个参数可以瞎写，但是不能不写

```
<!-- User findUser(int lid);-->
<select id="findUser" resultType="User">
    select * from t_user where id=${zdasdasasf}
</select>
</mapper>
```

## 多个参数

mapper接口方法的参数为多个字面量类型

则`${}`或者`#{}` 填arg+数字或者param+数字，arg自变量argument是以0开始，param参数parameter是以1开始，参数是值，arg或者param为键即本质是利用接口Map

```
<!-- User CheckUser(int userId,String passWord);-->
<select id="CheckUser" resultType="User">
    select * from t_user where id=#{arg0} and password=#{param2}
</select>
</mapper>
```

利用注解给map赋值key，用@Param注解的值，也可以用param+数字

`[pw, id, param1, param2]`

```

<!--      User CheckUserByParam(@Param("id") int userId, @Param("pw") String
password);-->
<select id="CheckUserByParam" resultType="User">
    select * from t_user where id=#{id} and password=#{pw}
</select>

```

也可以利用Map引用数据类型

```

HashMap<String, Object> map = new HashMap<>();
map.put("id", 6);
map.put("passWord", "123456");
System.out.println(mapper.Map(map));

<!--      User Map(Map<String, Object> map);-->
<select id="Map" resultType="User">
    select * from t_user where id=#{id} and password=#{passWord}
</select>

```

区分大小写

## 参数是对象

mapper接口方法的参数是实体类类型的参数，则用属性访问属性值

```

<!--      int insertUser(User user);-->
<insert id="insertUser">
    insert into t_user values(null,#{username},#{password},#{age},#{email})
</insert>

```

## 返回值基本数据类型

mybatis中设置了默认的类型别名

Alias	Mapped Type
<code>_byte</code>	<code>byte</code>
<code>_char</code> (since 3.5.10)	<code>char</code>
<code>_character</code> (since 3.5.10)	<code>char</code>
<code>_long</code>	<code>long</code>
<code>_short</code>	<code>short</code>
<code>_int</code>	<code>int</code>
<code>_integer</code>	<code>int</code>
<code>_double</code>	<code>double</code>
<code>_float</code>	<code>float</code>
<code>_boolean</code>	<code>boolean</code>
<code>string</code>	<code>String</code>
<code>byte</code>	<code>Byte</code>
<code>char</code> (since 3.5.10)	<code>Character</code>
<code>character</code> (since 3.5.10)	<code>Character</code>
<code>long</code>	<code>Long</code>
<code>short</code>	<code>Short</code>
<code>int</code>	<code>Integer</code>
<code>integer</code>	<code>Integer</code>
<code>double</code>	<code>Double</code>
<code>float</code>	<code>Float</code>
<code>boolean</code>	<code>Boolean</code>
<code>date</code>	<code>Date</code>

decimal	BigDecimal
bigdecimal	BigDecimal
biginteger	BigInteger
object	Object
date[]	Date[]
decimal[]	BigDecimal[]
bigdecimal[]	BigDecimal[]
biginteger[]	BigInteger[]
object[]	Object[]
map	Map
hashmap	HashMap
list	List
arraylist	ArrayList
collection	Collection
iterator	Iterator

## 返回值map

字段是键，列值是值

查询一条，若是返回类型是map<String,object>

```
<!--      Map<String,Object> returnMap(int id);-->
<select id="returnMap" resultType="map">
    select * from t_user where id=#{arg0}
</select>
```

## 返回多条数据

注意resultType是单条数据的类型！！！！！！

查询多条，则返回值是List<map<String,object>>

```
<!--      List<Map<String,Object>> returnAllMap(int id);-->
<select id="returnAllMap" resultType="map">
    select * from t_user
</select>
```

也可以用

```

<!--
@MapKey("id")          //id作为key
Map<String,Object> returnMap(int id);
-->
<select id="returnMap" resultType="map">
    select * from t_user
</select>

```

模糊查询则

① '%#{value}%' 会解析成 '%? %'，即字符串为 '%? %'，则报错

则应该用

② '%\${value}%' 会解析成 '%value%'，即正确

③ 可以用拼接 concat ('%',#{value},'%')

④ "%"#{value}%" 推荐使用这一种!!!

## 批量删除

批量删除用in的时候不能用#{ }会加上单引号

只能 in (\${ })

## 动态设置表名

表名没有单引号只能用\${ }

## 获取自动递增的主键

```

<!--          int insertUser(User user);-->
<insert id="insertUser" useGeneratedKeys="true" keyProperty="id">
    insert into t_user values(null,#{username},#{password},#{age},#{email})
</insert>

```

useGeneratedKeys="true"使用自动递增true后

keyProperty键注入到属性id中。注意会主键自增的值会传给方法参数user

的属性!!!!!!

## 自定义映射resultMap

### 字段名和属性名不一致

#### 方法一

依旧使用resultType但select中是字段名 as 属性名或者as可以不写 字段名+空格+属性名

```
<select id="findName" resultType="User">
    select user_name as username from t_user where username like "%#{arg0}%"
</select>

<select id="findName" resultType="User">
    select user_name username from t_user where username like "%#{arg0}%"
</select>
```

## 方法二

依旧使用resultType可将字段a\_name自动as成属性aName

首先核心配置文件配置驼峰映射

找到配置xml

### Table of Contents

Table of Contents .....	i
1. Introduction .....	1
2. Getting Started .....	2
3. Configuration XML .....	8
4. Mapper XML Files .....	31
5. Dynamic SQL .....	63
6. Java API .....	70
7. Statement Builders .....	92
8. Logging .....	100

找到settings标签

### 3.1 Configuration

The MyBatis configuration contains settings and properties that have a dramatic effect on how MyBatis behaves. The high level structure of the document is as follows:

- configuration
  - properties
  - settings
  - typeAliases
  - typeHandlers
  - objectFactory
  - plugins
  - environments
    - environment
      - transactionManager
      - dataSource
  - databaseIdProvider
  - mappers

了解settings标签属性mapUnderscoreToCamelCase

mapUnderscoreToCamelCase	Enables automatic mapping from classic database column names A_COLUMN to camel case classic Java property names aColumn.	true   false	False
--------------------------	--	--------------	-------



第一列是setting标签的name属性，value可以是第三列其一，mybatis默认是false。第二列是describe描述，即能够自动映射数据库列名a\_column

驼峰拼写成java属性名aColumn

```
<!--驼峰映射-->
<settings>
  <setting name="mapUnderscoreToCamelCase" value="true"/>
</settings>
```

驼峰拼写是因为

mysql表名习惯t\_开头

不像java一样有大小写，有新的大写时候是用\_下划线

字段名和属性名不一致时候，属性值为null，不会报错！！

## 方法三

利用resultMap

```
<!-- type是针对哪一个类的映射，等同resultType-->
<resultMap id="zyj" type="User">
<!-- 针对主键-->
  <id property="id" column="id"></id>
<!-- 针对普通字段，property是属性，column是列名-->
  <result property="username" column="username"></result>
<!-- 不需要修改的列名与属性名的关系可以不写-->
</resultMap>
<!-- List<User> findName(String name);-->
<!-- resultMap是针对某一个resultMap标签的id-->
<select id="findName" resultMap="zyj">
  select * from t_user where username like "%#{arg0}%"
</select>
```

## 处理多对一映射关系

例如员工和部门，一个部门对应多个员工。则在员工类是会有一个对象部门

### 方法一

级联属性赋值

针对属性是对象，利用对象.属性的方式注入值

```

<resultMap id="empAndDeptResultMapOne" type="Emp">
    <id property="eid" column="eid"></id>
    <result property="empName" column="emp_name"></result>
    <result property="age" column="age"></result>
    <result property="sex" column="sex"></result>
    <result property="email" column="email"></result>
    <result property="dept.did" column="did"></result>
    <result property="dept.deptName" column="dept_name"></result>
</resultMap>

```

## 方法二

association处理多对一的映射关系

property需要处理的属性名

javaType该属性的类型

通过resultMap标签的属性association联系员工对象的属性为对象的dept

```

<resultMap id="empAndDeptResultMapTwo" type="Emp">
    <id property="eid" column="eid"></id>
    <result property="empName" column="emp_name"></result>
    <result property="age" column="age"></result>
    <result property="sex" column="sex"></result>
    <result property="email" column="email"></result>
    <association property="dept" javaType="Dept">
        <id property="did" column="did"></id>
        <result property="deptName" column="dept_name"></result>
    </association>
</resultMap>

```

## 方法三/分步查询

分步查询

```

<resultMap id="empAndDeptResultMapThree" type="Emp">
    <id property="eid" column="eid"></id>
    <result property="empName" column="emp_name"></result>
    <result property="age" column="age"></result>
    <result property="sex" column="sex"></result>
    <result property="email" column="email"></result>
    <association property="dept"
        select="com.atguigu.mybatis.mapper.DeptMapper.getEmpAndDeptByStepTwo"
        column="did"></association>
</resultMap>

```

property值dept代表员工类里的对象dept部门

column是前一次查询对应的列，将会获取其列值。将其作为select查询条件的参数

select是分步查询，mapper接口的全类名.方法名

注意select要返回一个对象！！！！是员工类里部门属性所对应的类。

```
<!--Dept getEmpAndDeptByStepTwo (@Param( "did") Integer did)-->
<select id="getEmpAndDeptByStepTwo" resultType="Dept">
    select * from t_dept where did = #{did}
</select>
```

## 延迟加载

默认是false

执行对主加载对象的查询时，不会执行对关联对象的查询。但当要访问主加载对象的某个属性（该属性不是关联对象的属性）时，就会马上执行关联对象的select查询。如果侵入式延迟加载打开则懒加载怎么设都没用，只设懒加载的话不用管侵入式延迟加载，因为它默认是false。

延迟加载也称为**懒加载**，是指在关联查询时，按照设置延迟规则推迟对关联对象的select查询。延迟加载可以有效的减少数据库压力。

```
<!-- 开启延迟加载-->
<settings>
    <setting name="lazyLoadingEnabled" value="true"/>默认就是false
</settings>
```

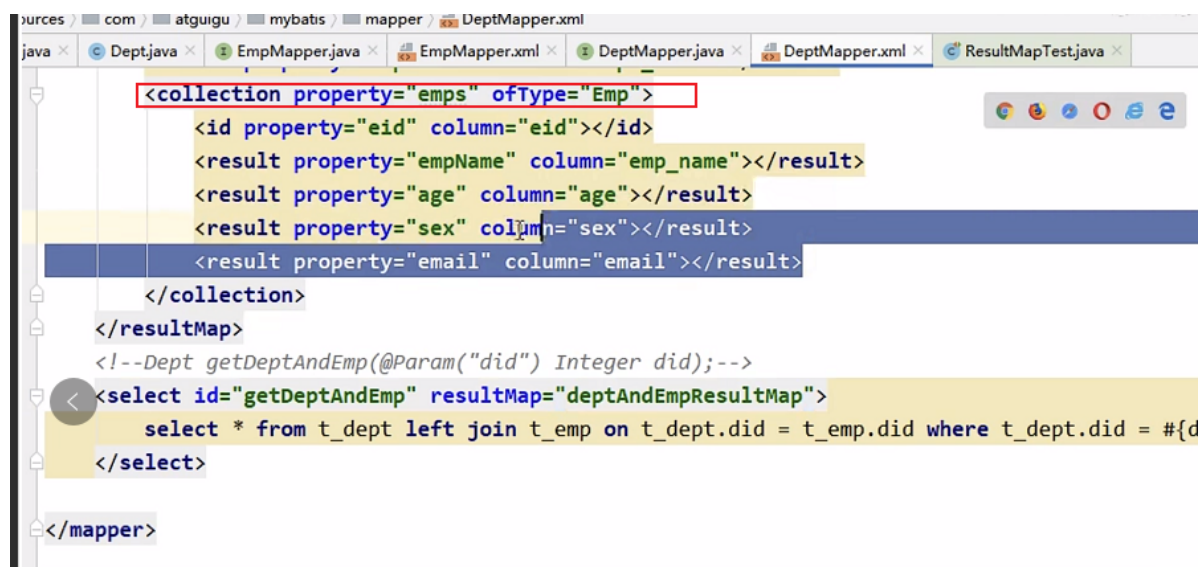
根据情况加载分步查询第二步

如果开启全局延迟加载但sql语句查询不想延迟加载则在标签的resultMap的association标签有个属性fetchType若是lazy则是延迟加载，若是eager则是立即加载，如果没有全局延迟加载则这个fetchType怎么设值都是eager

## 处理一对多映射关系

### 方法一

获取部门以及部门中所有的员工信息



```
<collection property="emps" ofType="Emp">
    <id property="eid" column="eid"></id>
    <result property="empName" column="emp_name"></result>
    <result property="age" column="age"></result>
    <result property="sex" column="sex"></result>
    <result property="email" column="email"></result>
</collection>
</resultMap>
<!--Dept getDeptAndEmp(@Param("did") Integer did);-->
<select id="getDeptAndEmp" resultMap="deptAndEmpResultMap">
    select * from t_dept left join t_emp on t_dept.did = t_emp.did where t_dept.did = #{did}
</select>
</mapper>
```

collection是处理部门类的对象数组emps属性

ofType是数组里的类型

### 方法二/分步查询

查部门及其部门的员工

先查部门

```
<!--Dept getDeptAndEmpByStepOne(@Param("did") Integer did);-->
<select id="getDeptAndEmpByStepOne" resultMap="">
    select * from t_dept where did = #{did}
</select>
```

再查员工

```
<resultMap id="deptAndEmpByStepResultMap" type="Dept">
    <id property="did" column="did"></id>
    <result property="deptName" column="dept_name"></result>
    <collection property="emps"
        select="com.atguigu.mybatis.mapper.EmpMapper.getDeptAndEmpByStepTwo"
        column="did"></collection>
</resultMap>
<!--Dept getDeptAndEmpByStepOne(@Param("did") Integer did);-->
<select id="getDeptAndEmpByStepOne" resultMap="deptAndEmpByStepResultMap">
    select * from t_dept where did = #{did}
</select>
```

select是分布查询，mapper接口的全类名.方法名

注意select要返回一个对象！！！！是部门类里对象数组员工属性所对应的类的！！！！数组！！！！

```
<!--List<Emp> getDeptAndEmpByStepTwo(@Param("did") Integer did);-->
<select id="getDeptAndEmpByStepTwo" resultType="Emp">
    select * from t_emp where did = #{did}
</select>
```

## 延迟加载

```
<!-- 开启延迟加载-->
<settings>
    <setting name="lazyLoadingEnabled" value="true"/>
</settings>
```

根据情况加载分步查询第二步

如果开启全局延迟加载但sql语句查询不想延迟加载则在标签的resultMap的association标签有个属性fetchType若是lazy则是延迟加载，若是eager则是立即加载，如果没有全局延迟加载则这个fetchType怎么设值都是lazy

## 动态sql

### if

if用的xml的and，而不是&&

加1=1是因为防止第一个if不进行，第二个进行，则会导致where and报错

```

<!--      List<User> getUser(User user);-->
<select id="getUser" resultType="User">
    select * from t_user where 1=1
    <if test="username !=null and username !=''">
        username=#{username}
    </if>
    <if test="id !=null and id !=' ' ">
        and id=#{id}
    </if>
</select>

```

## where

也可以换种写法，当where有内容时候自动生成where关键字，并且内容前多余的and或者or去掉，但不会去掉内容后的!!!，where没有内容，则不会自动生成where关键字

```

select * from t_user
<where>
    <if test="username !=null and username !=''">
        and这里写了也没啥事!!      username=#{username}
    </if>
    <if test="id !=null and id !=' ' ">
        and id=#{id}
    </if>
</where>

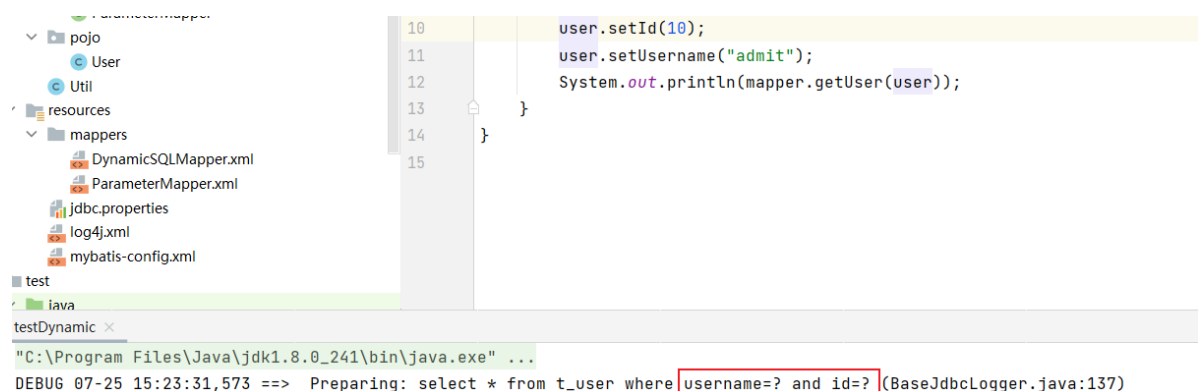
```

内容前多余的and或者or去掉，但不会去掉内容后的!!!

```

select * from t_user
<where>
    <if test="username !=null and username !=''">
        username=#{username} and 这里写了可能有事!!
    </if>
    <if test="id !=null and id !=' ' ">
        id=#{id}
    </if>
</where>

```



## trim

trim的属性



prefix: 给trim标签内sql语句加上前缀

suffix: 给trim标签内sql语句加上后缀

prefixOverrides: 去除多余的前缀内容, 如: prefixOverrides="OR", 去除trim标签内sql语句多余的前缀"OR"

suffixOverrides: 去除多余的后缀内容, 如: suffixOverrides=", ", 去除trim标签内sql语句多余的后缀", "

```
<select id="getUser" resultType="User">
  select * from t_user
  <trim prefix="where" suffixOverrides="and|or" >
    <if test="username !=null and username !=''">
      username=#{username} and
    </if>
    <if test="id !=null and id !=' ' ">
      id=#{id} or
    </if>
  </trim>
</select>
```

## 选择

也有

swith的case default类型

switch相当于choose

when相当于case

default相当于otherwise

也可以理解为if else if else if else

where至少要有有一个!

otherwise最多只有一个!

```
<select id="getUser" resultType="User">
  select * from t_user
  <where>
    <choose>
      <when test="id !=null and id !=' '">
        id=#{id}
      </when>
      <when test="username !=null and username !=' '">
        username=#{username}
      </when>
      <otherwise>
        id=9
      </otherwise>
    </choose>
  </where>
</select>
```

## foreach

collection设置需要循环的数组或者集合

item表示数组或者集合中的每一个数据

separator循环体之间的分割符

open表示foreach标签所循环的所有内容的开始符

close是结束符

## 批量删除

item是数组的遍历值,

separator是表示分隔符

以下代码可以理解为

```
StringBuffer temp=new StringBuffer("");
```

```
for (int id=0, id<array,id++)
```

```
{
```

```
    temp.append(id);
```

```
    if (id!=array-1)
```

```
        temp.append(",");
```

```
}
```

```
temp=new StringBuffer("");
```

Parameter 'aray' not found. Available parameters are [array, arg0]

```
<!-- int deleteMoreUser(Integer [] ids);-->
<delete id="deleteMoreUser" >
    delete from t_user where id in
    (
        //collection的属性值用array或者 arg0
        <foreach collection="array" item="id" separator=",">
            #{id}
        </foreach>
    )
</delete>

或者利用属性open和close
<delete id="deleteMoreUser" >
    delete from t_user where id in
        <foreach collection="arg0" item="id" separator="," open="("
close=")">
            #{id}
        </foreach>
</delete>

也可以不用逗号用or
<delete id="deleteMoreUser" >
    delete from t_user where id in
    (
        <foreach collection="array" item="id" separator="or">
            #{id}
        </foreach>
```

```
)  
</delete>
```

atis2 D:\JavaProgram\Mybatis2

dea

rc

main

java

mappers

DynamicSQLMapper

ParameterMapper

pojo

testDynamic

"C:\Program Files\Java\jdk1.8.0\_241\bin\java.exe" ...

DEBUG 07-25 16:47:18,012 ==> Preparing: delete from t\_user where id in ( ? or ? or ? ) (BaseJdbcLogger.java:137)

DEBUG 07-25 16:47:18,034 ==> Parameters: 6(Integer), 7(Integer), 8(Integer) (BaseJdbcLogger.java:137)

DEBUG 07-25 16:47:18,036 <== Updates: 0 (BaseJdbcLogger.java:137)

0

或者用@Parm

```
<!-- int deleteMoreUser(@Parm("idssss")Integer [] ids);-->  
<delete id="deleteMoreUser" >  
    delete from t_user where id in  
    (  
        //collection的属性值用array或者 arg0  
        <foreach collection="idssss" item="id" separator=",">  
            #{id}  
        </foreach>  
    )  
</delete>
```

## 批量添加

注意参数得用list或者arg0或者collection

```
<!-- int insertMoreUser(List<User> list);-->  
<insert id="insertMoreUser">  
    insert into t_user values  
    <foreach collection="list" item="i" separator="," >  
        (null,#{i.username}, #{i.password},#{i.age},#{i.email})  
    </foreach>  
</insert>
```

## 常用sql片段

```
<!-- sql片段-->  
<sql id="columns"> username,password,age,email</sql>  
  
<select id="">  
    select <include refid="columns"></include>  
</select>
```

## 缓存/只增对查询

从数据库查取后再次查询，从缓存中获取不会从数据库重新访问，查无再次去数据库搜索。

## 一级缓存范围SqlSession默认开启



通过同一个sql会话SqlSession查询的数据会被缓存。

失效：

- ①不同的SqlSession
- ②查询条件不同
- ③两次查询期间执行过增删改
- ④手动清空缓冲：代码SqlSession.clearCache();

## 二级缓存范围SqlSessionFactory要手动开启

通过同一个SqlSessionFactory创建的多个SqlSession

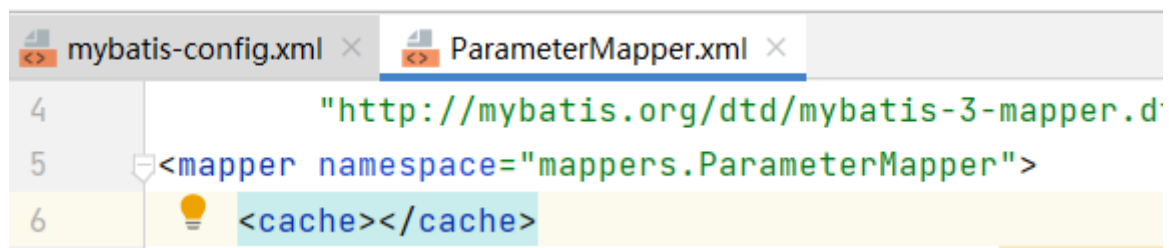
步骤：

- ①核心配置文件设置全局配置属性cacheEnabled="true",默认是true，即不用写这个配置，可跳过



```
13      <!-- 二级缓存，可不写-->
14      <setting name="cacheEnabled" value="true"/>
```

- ②在映射文件中设置标签



```
4      "http://mybatis.org/dtd/mybatis-3-mapper.d
5      <mapper namespace="mappers.ParameterMapper">
6      <cache></cache>
```

在mapper配置文件中添加的cache标签可以设置一些属性：

- eviction属性：缓存回收策略
  - LRU（Least Recently Used）- 最近最少使用的：移除最长时间不被使用的对象。
  - FIFO（First in First out）- 先进先出：按对象进入缓存的顺序来移除它们。
  - SOFT - 软引用：移除基于垃圾回收器状态和软引用规则的对象。
  - WEAK - 弱引用：更积极地移除基于垃圾收集器状态和弱引用规则的对象。默认的是 LRU。
- flushInterval属性：刷新间隔，单位毫秒
  - 默认情况是不设置，也就是没有刷新间隔，缓存仅仅调用语句时刷新
- size属性：引用数目，正整数
  - 代表缓存最多可以存储多少个对象，太大容易导致内存溢出
- readOnly属性：只读，true/false
  - true：只读缓存；会给所有调用者返回缓存对象的相同实例。因此这些对象不能被修改。这提供了很重要的性能优势。
  - false：读写缓存；会返回缓存对象的拷贝（通过序列化）。这会慢一些，但是安全，因此默认是 false。

eviction默认LRU

flushInterval调用语句时刷新是指增删改

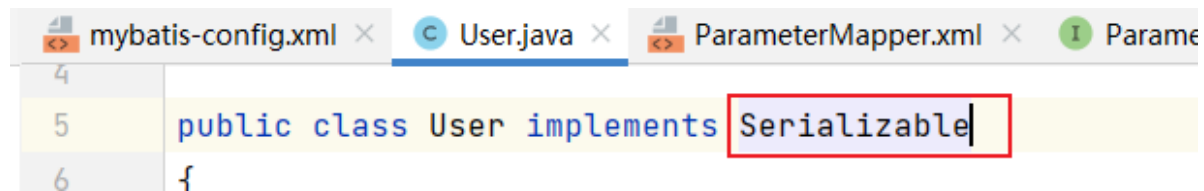
- ③二级缓存必须在sqlsession关闭或者提交之后才有效

不能用

```
SqlSession sqlSession = build.openSession( autoCommit: true);|
```

得用sqlSession对象.close()或者sqlSession.commit()

④查询的数据所转换的实体类类型必须实现序列化的接口



失效:

①两次查询的增删改

注意手动清空缓冲: 代码SqlSession.clearCache();没有用!!! 因为是针对sqlSession不是针对工厂SqlSessionFactory

## 缓存顺序

先大范围后小范围, 先查询二级缓存没有再查询一级范围 (二级可能没一级是因为如果没close或者commit的话二级没有缓存) 没有再查询数据库

## 整合第三方缓存EHCache代替二级缓存 (可不了解! )

### 引入依赖

```
<!--      Mybatis EHCACHE整合包-->
<dependency>
  <groupId>org.mybatis.caches</groupId>
  <artifactId>mybatis-ehcache</artifactId>
  <version>1.2.1</version>
</dependency>
简易日志的log4j可能会失效
<!--      slf4j日志门面的一个具体实现-->
<dependency>
  <groupId>ch.qos.logback</groupId>
  <artifactId>logback-classic</artifactId>
  <version>1.2.11</version>
</dependency>
```

### 配置文件名字必须叫ehcache.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd"
  updateCheck="false">
  <!--
    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:
    user.home - 用户主目录
    user.dir - 用户当前工作目录
    java.io.tmpdir - 默认临时文件路径
  -->
```

```
<diskStore path="java.io.tmpdir/Tmp_EhCache"/>
```

```
<defaultCache  
    eternal="false"  
    maxElementsInMemory="10000"  
    overflowToDisk="false"  
    diskPersistent="false"  
    timeToIdleSeconds="1800"  
    timeToLiveSeconds="259200"  
    memoryStoreEvictionPolicy="LRU"/>
```

```
<cache  
    name="cloud_user"  
    eternal="false"  
    maxElementsInMemory="5000"  
    overflowToDisk="false"  
    diskPersistent="false"  
    timeToIdleSeconds="1800"  
    timeToLiveSeconds="1800"  
    memoryStoreEvictionPolicy="LRU"/>
```

```
<!--
```

**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。

```
-->
```

```
<!--
```

**name:** 缓存名称。

**maxElementsInMemory:** 缓存最大数目

**maxElementsOnDisk:** 硬盘最大缓存个数。

**eternal:** 对象是否永久有效，一但设置了，**timeout**将不起作用。

**overflowToDisk:** 是否保存到磁盘，当系统宕机时

**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。

**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。

**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.**

**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。

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

**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）。

**clearOnFlush:** 内存数量最大时是否清除。

**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）。

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

**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。

**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。

```
-->
```

```
</ehcache>
```

## 配置文件logback固定名字

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration debug="false">
    <!--定义日志文件的存储地址 -->
    <define name="hostname"
        class="com.jimmy.lai.route.gateway.config.HostnameConfig"/>
    <property name="LOG_HOME" value="../logs"/>
    <!-- 控制台输出 -->
    <appender name="STDOUT"
        class="ch.qos.logback.core.ConsoleAppender">
        <encoder
            class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
                <!--格式化输出：%d表示日期，%thread表示线程名，%-5level：级别从左显示5个字符宽度%msg：日志消息，%n是换行符 -->
                <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{50}: -
%msg%n</pattern>
            </encoder>
        </appender>

    <!-- 按照每天生成日志文件 -->
    <appender name="LOGFILE"
        class="ch.qos.logback.core.rolling.RollingFileAppender">
        <rollingPolicy
            class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
                <!--日志文件输出的文件名 -->
                <fileNamePattern>${LOG_HOME}/gateway.%d{yyyy-MM-dd}.${hostname}.log
            </fileNamePattern>
                <!--日志文件保留天数 -->
                <maxHistory>1</maxHistory>
            </rollingPolicy>
        <encoder
            class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
                <!--格式化输出：%d表示日期，%thread表示线程名，%-5level：级别从左显示5个字符宽度%msg：日志消息，%n是换行符 -->
                <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{50}: -
%msg%n</pattern>
            </encoder>
        </appender>

    <appender name="ACCESSFILE"
        class="ch.qos.logback.core.rolling.RollingFileAppender">
        <rollingPolicy
            class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
                <!--日志文件输出的文件名 -->
                <fileNamePattern>${LOG_HOME}/gateway_access.%d{yyyy-MM-dd}.${hostname}.log
            </fileNamePattern>
                <!--日志文件保留天数 -->
                <maxHistory>1</maxHistory>
            </rollingPolicy>
        <encoder
            class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
                <!--格式化输出：%d表示日期，%thread表示线程名，%-5level：级别从左显示5个字符宽度%msg：日志消息，%n是换行符 -->
                <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{50}: -
%msg%n</pattern>
```

```

    </encoder>
</appender>
<!--此日志文件只包含错误日志-->
<appender name="ERROR" class="ch.qos.logback.core.rolling.RollingFileAppender">
    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
        <fileNamePattern>${LOG_HOME}/gateway-error.%d{yyyy-MM-dd}.log</fileNamePattern>
    </rollingPolicy>
    <!-- 追加方式记录日志 -->
    <append>true</append>
    <!-- 日志文件的格式 -->
    <encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
        <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{50}: -
        %msg%n</pattern>
        <charset>utf-8</charset>
    </encoder>
    <!-- 此日志文件只记录error级别的 -->
    <filter class="ch.qos.logback.classic.filter.LevelFilter">
        <level>error</level>
        <onMatch>ACCEPT</onMatch>
        <onMismatch>DENY</onMismatch>
    </filter>
</appender>

<appender name="ACCESSFILE_ASYNC" class="ch.qos.logback.classic.AsyncAppender">
    <appender-ref ref="ACCESSFILE"/>
</appender>

<appender name="LOGFILE_ASYNC" class="ch.qos.logback.classic.AsyncAppender">
    <appender-ref ref="LOGFILE"/>
</appender>

<appender name="LOG_ERROR_FILE_ASYNC"
class="ch.qos.logback.classic.AsyncAppender">
    <appender-ref ref="ERROR"/>
</appender>
<logger name="reactor.netty.http.server.AccessLog" level="INFO"
additivity="false">
    <appender-ref ref="ACCESSFILE_ASYNC"/>
</logger>

<!--上报skywalking日志-->
<appender name="GRPC_LOG"
class="org.apache.skywalking.apm.toolkit.log.logback.v1.x.log.GRPCLogClientAppen
der">
    <encoder class="ch.qos.logback.core.encoder.LayoutWrappingEncoder">
        <layout
class="org.apache.skywalking.apm.toolkit.log.logback.v1.x.TraceIdPatternLogbackL
ayout">
            <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level
logger_name:%logger{36} - [%tid] - message:%msg%n</pattern>
        </layout>
    </encoder>
</appender>

<!-- 日志输出级别 -->
<root level="INFO">
    <!-- <appender-ref ref="STDOUT"/>-->

```

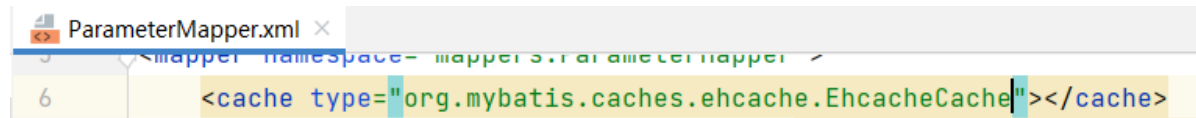
```

<appender-ref ref="LOGFILE_ASYNC"/>
<appender-ref ref="LOG_ERROR_FILE_ASYNC"/>
<appender-ref ref="GRPC_LOG" />
</root>
</configuration>

```

## 设置二级缓存类型

默认是使用mybatis



```
<cache type="org.mybatis.caches.ehcache.EhcacheCache"></cache>
```

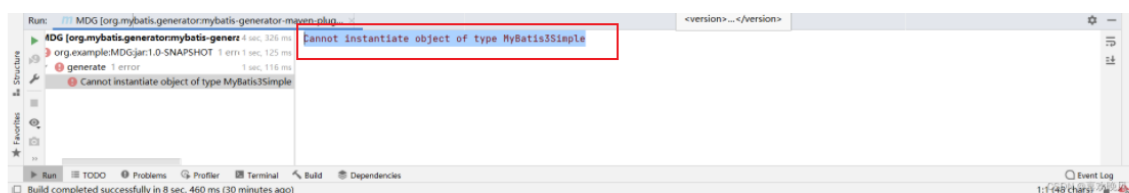
## 逆向工程

正向工程：先创建Java实体类再有框架负责实体类生成数据库表

逆向工程：先创建数据库表，由框架负责根据数据库表反向生成Java实体类，Mapper接口，Mapper映射文件

## 创建逆向工程步骤

## 问题



这个问题主要是mybatis-generator-core的版本太低了  
改成1.3.7就好了。

如果还不行就把mybatis-generator-maven-plugin的版本也改成1.3.7试试。

译

我是跟着b站视频走的，当时因为mybatis-generator-core的版本是1.3.0的时候下面c3p0的依赖没法引进来，我就把这依赖改高了改成1.3.6还是1.3.2来着，然后为了和视频一致，我也害怕出现版本问题，我就又把core的版本改回去了改成1.3.0。就出现了上面的问题。

## 引入依赖

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.example</groupId>
  <artifactId>Mybatis3</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <properties>
    <maven.compiler.source>8</maven.compiler.source>

```

```

        <maven.compiler.target>8</maven.compiler.target>
    </properties>
    <dependencies>
<!--      依赖MyBatis核心包-->
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis</artifactId>
            <version>3.5.9</version>
        </dependency>
<!--      逆向工程的核心包-->
        <dependency>
            <groupId>org.mybatis.generator</groupId>
            <artifactId>mybatis-generator-core</artifactId>
            <version>1.3.2</version>
        </dependency>
<!--      数据库连接池-->
        <dependency>
            <groupId>com.mchange</groupId>
            <artifactId>c3p0</artifactId>
            <version>0.9.2</version>
        </dependency>
<!--      MySQL驱动-->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>8.0.29</version>
        </dependency>
<!--      日志-->
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>1.2.12</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
<!--      具体插件，逆向工程的操作是以构建过程中插件的形式出现的-->
            <plugin>
                <groupId>org.mybatis.generator</groupId>
                <artifactId>mybatis-generator-maven-plugin</artifactId>
                <version>1.3.7</version>
                <configuration>
                    <verbose>true</verbose>
                    <overwrite>true</overwrite>
                </configuration>
<!--      插件的依赖-->
                <dependencies>
                    <dependency>
                        <groupId>mysql</groupId>
                        <artifactId>mysql-connector-java</artifactId>
                        <version>8.0.29</version>
                    </dependency>
                </dependencies>
            </plugin>
        </plugins>
    </build>
</project>

```



## 逆向工作的配置文件generatorConfig.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE generatorConfiguration
    PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"
    "http://mybatis.org/dtd/mybatis-generator-config_1_0.dtd">
<generatorConfiguration>
    <!-- 引入第三方依赖包 -->
    <!--<classPathEntry location=".\\lib\\mysql-connector-java-8.0.12.jar" />-->

    <!--
    targetRuntime常用值：
        MyBatis3Simple(只生成基本的CRUD和少量的动态SQL)
        MyBatis3(生成完整的CRUD，包含CriteriaAPI方法Example后缀的方法)
    -->
    <context id="localhost_mysql" targetRuntime="MyBatis3Simple">

        <!-- 不生成注释 -->
        <commentGenerator>
            <property name="suppressAllComments" value="true" />
        </commentGenerator>

        <jdbcConnection driverClass="com.mysql.jdbc.Driver"
            connectionURL="jdbc:mysql://localhost:3306/jiaoxue?
characterEncoding=utf8&serverTimezone=UTC&useSSL=true"
            userId="root"
            password="mdjfbzyj515">
            <!-- 如果连接的是mysql需要加上这个配置，应为mysql不支持catalog和schema，如果
            多个库中有相同名称的表，会重复生成代码 -->
            <!-- <property name="nullCatalogMeansCurrent" value="true" />-->
        </jdbcConnection>

        <javaTypeResolver >
            <property name="forceBigDecimals" value="false" />
        </javaTypeResolver>

        <!-- 生成实体类 -->
        <!-- 路径-->
        <javaModelGenerator targetPackage="pojo"
targetProject=".\\src\\main\\java">
            <!-- 是否生成子包-->
            <property name="enableSubPackages" value="false" />
            <!-- 去掉字段名转成属性的空格-->
            <property name="trimStrings" value="true" />
        </javaModelGenerator>

        <!-- 生成XML Mapper -->
        <sqlMapGenerator targetPackage="mappers"
targetProject=".\\src\\main\\resources">
            <!-- 是否生成子包-->
            <property name="enableSubPackages" value="false" />
        </sqlMapGenerator>

        <!-- 生成Mapper接口 -->
```



```

        <!-- 生成的Mapper类型: ANNOTATEDMAPPER (注解)、MIXEDMAPPER (混合)、XMLMAPPER (XML) -->
        <javaClientGenerator type="XMLMAPPER" targetPackage="mappers"
targetProject=".\\src\\main\\java">
            <!-- 是否将数据库中的schema作为包名的一部分, 默认就是false -->
            <property name="enableSubPackages" value="false" />
        </javaClientGenerator>

        <!-- 完全限定一张表: catalog名称.schema名称.表名(如果多个库中有同名的表必须配置)
        其实Mysql根本不支持catalog和schema, 建议不要配置这两个, 使用jdbcConnection标
        签中的nullCatalogMeansCurrent配置项解决
        -->
        <!--
        <table catalog="lanou" schema="lanou" tableName="book">
        -->
        <!-- 一个table标签对应着数据库中一个表 -->
<!--      表名和实体类类名-->
        <table tableName="t_user" domainObjectName="User">

            <!-- 是否用数据库中的字段名作为POJO属性名(不自动转小驼峰), 默认值是false -->
            <!--
            <property name="useActualColumnNames" value="true"/>
            -->
            <!-- 生成代码时支持获取插入数据后自增的ID, 需要通过sqlStatement配置数据库类
            型。 -->

            <!--      <generatedKey column="id" sqlStatement="mysql" identity="true"
            />-->

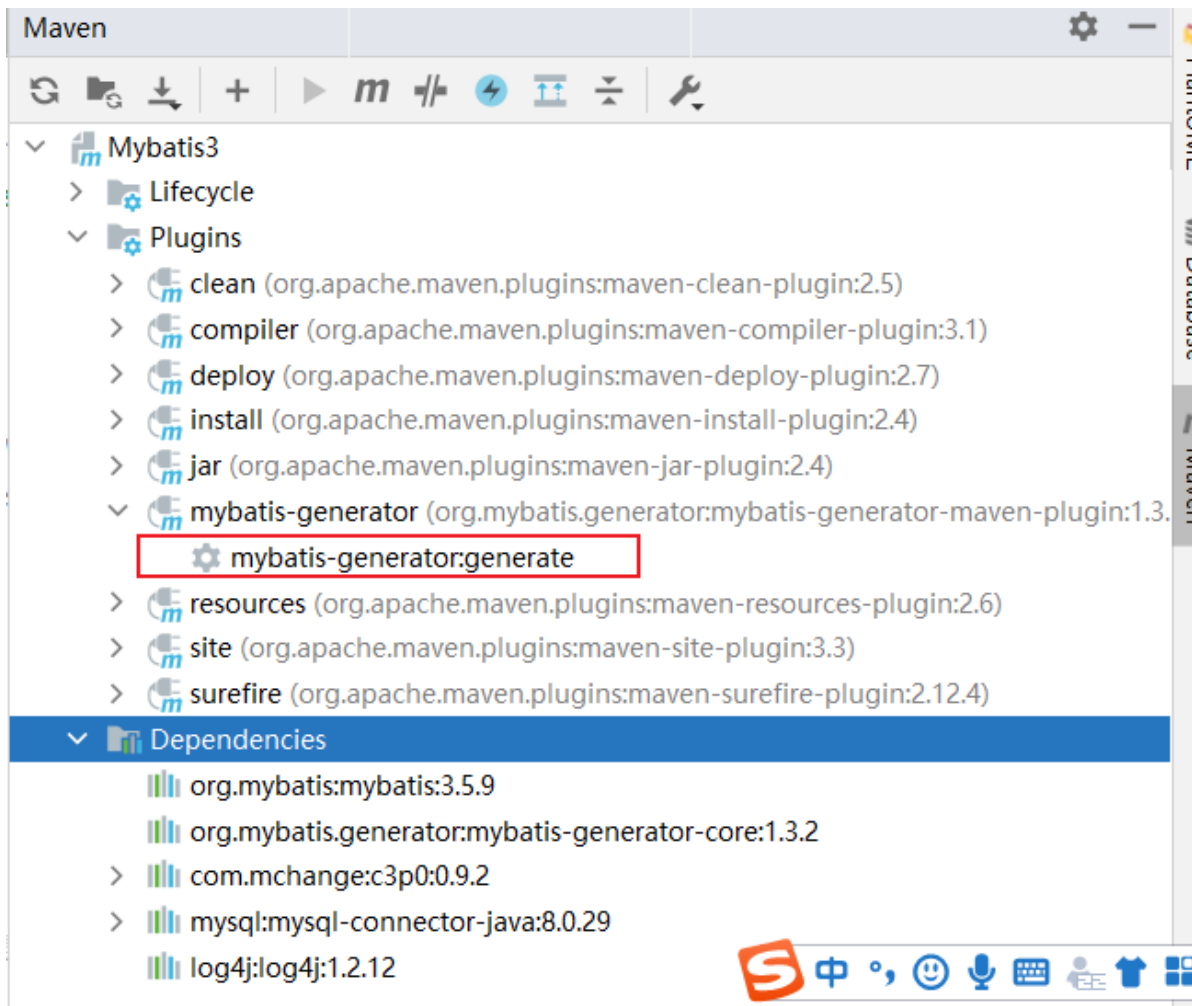
            <!-- 此标签用于在生成代码时忽略数据库中的某个字段 -->
            <!--
            <ignoreColumn column="FRED" />
            -->
            <!-- 通过此标签重写mybatis从数据库读到的元信息, 自定义列相关配置, 包括(名称、类
            型) -->

            <!--
            <columnOverride column="aa" property="sname" />
            -->

        </table>
    </context>
</generatorConfiguration>

```

## 调用插件生成代码



代码的方法调用以及分页的功能建议自己写代码