Mybatis

一、环境配置

(一)、Mybatis 下载:

https://github.com/mybatis/mybatis-3/releases

(二)、Mybatis 的 Maven:

```
<!-- https://mvnrepository.com/artifact/org.mybatis/mybatis -->

<dependency>

    <groupId>org.mybatis</groupId>

    <artifactId>mybatis</artifactId>

    <version>3.5.7</version>

</dependency>
```

(三)、中文文档:

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

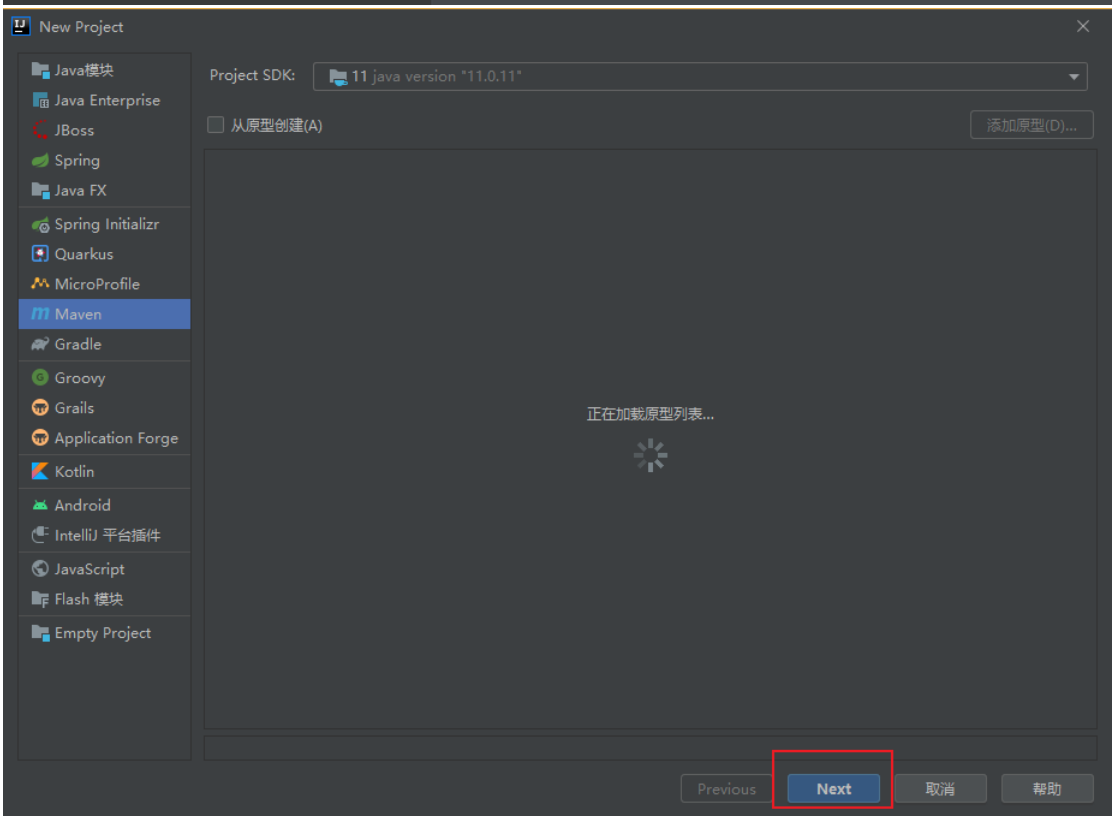
二、第一个 Mybatis 搭建

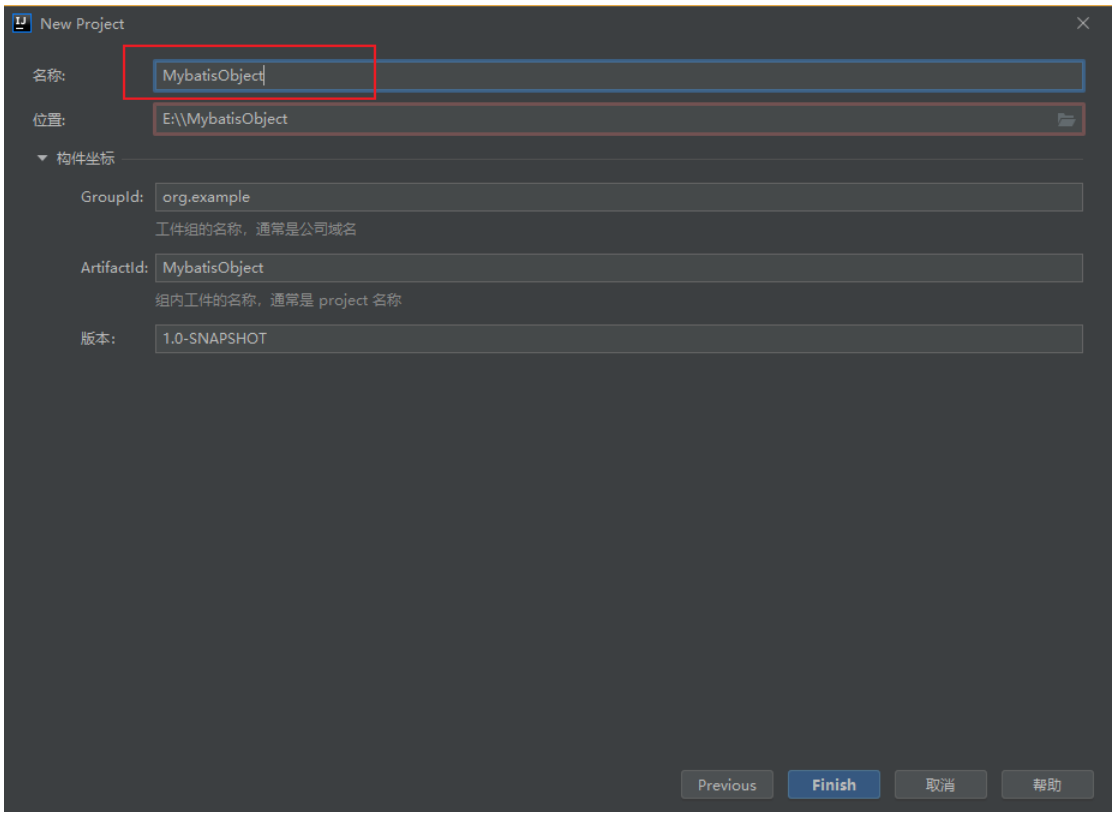
(一)、搭建数据库

```
CREATE mybatis;

CREATE TABLE `user`(

        `id` INT(20) NOT NULL PRIMARY KEY,

        `name` VARCHAR(30) DEFAULT NULL,

        `pwd` VARCHAR(30) DEFAULT NULL,



)ENGINE=INNODB DEFAULT CHARSET=utf8;

INSERT INTO `user` (`id`,`name`,`pwd`) VALUES

(1,'马佳盛','12345'),

(2,'王子泓','12345'),

(3,'王宝延','12345');
```

(二)、创建 IDEA 项目


Welcome to IntelliJ IDEA
SpringMVC_demo1
E:\SpringMVC_demo1
Spring
E:\Spring
java
E:\java
IntelliJ IDEA
Version 2020.1.1
创建新项目
打开或导入...
从版本控制中获取
Configure
Get Help


New Project
Java模块
Java Enterprise
JBoss
Spring
Java FX
Spring Initializr
Quarkus
MicroProfile
Maven
Gradle
Groovy
Grails
Application Forge
Kotlin
Android
IntelliJ 平台插件
JavaScript
Flash 模块
Empty Project
Project SDK:
11 java version "11.0.11"
从原型创建(A)
添加原型(D)...
正在加载原型列表...
Previous
Next
取消
帮助



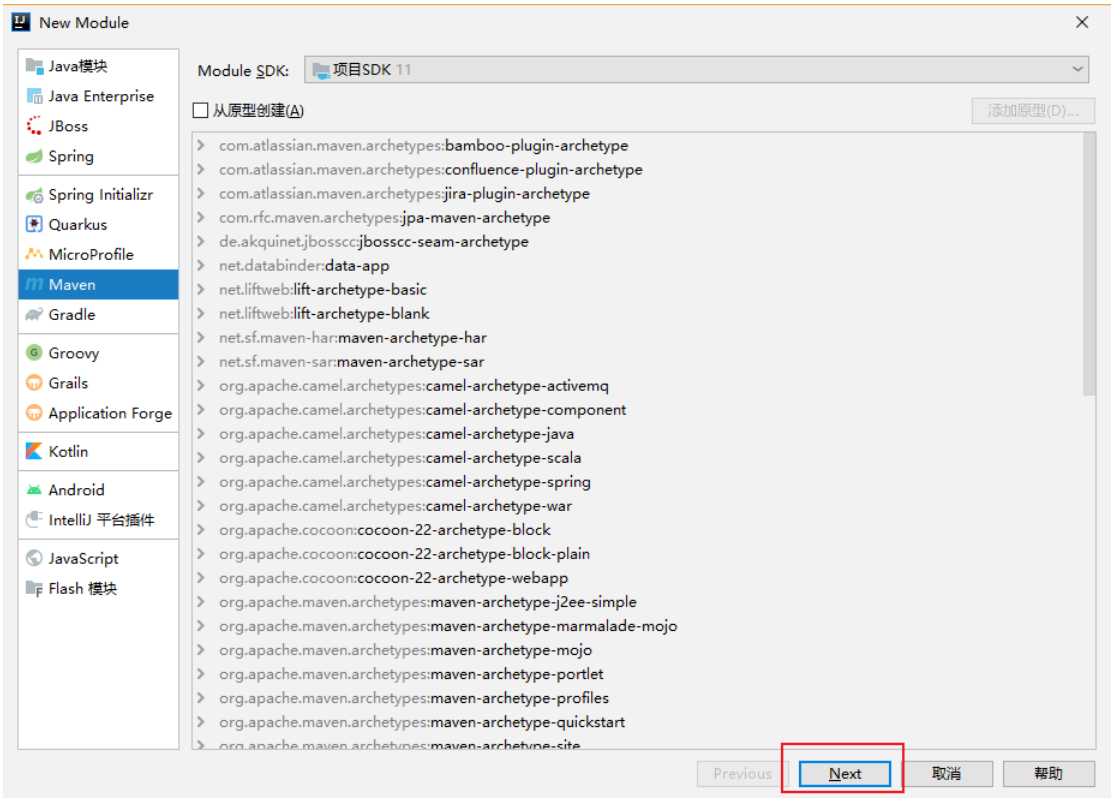
(三)、pom.xml 导入依赖

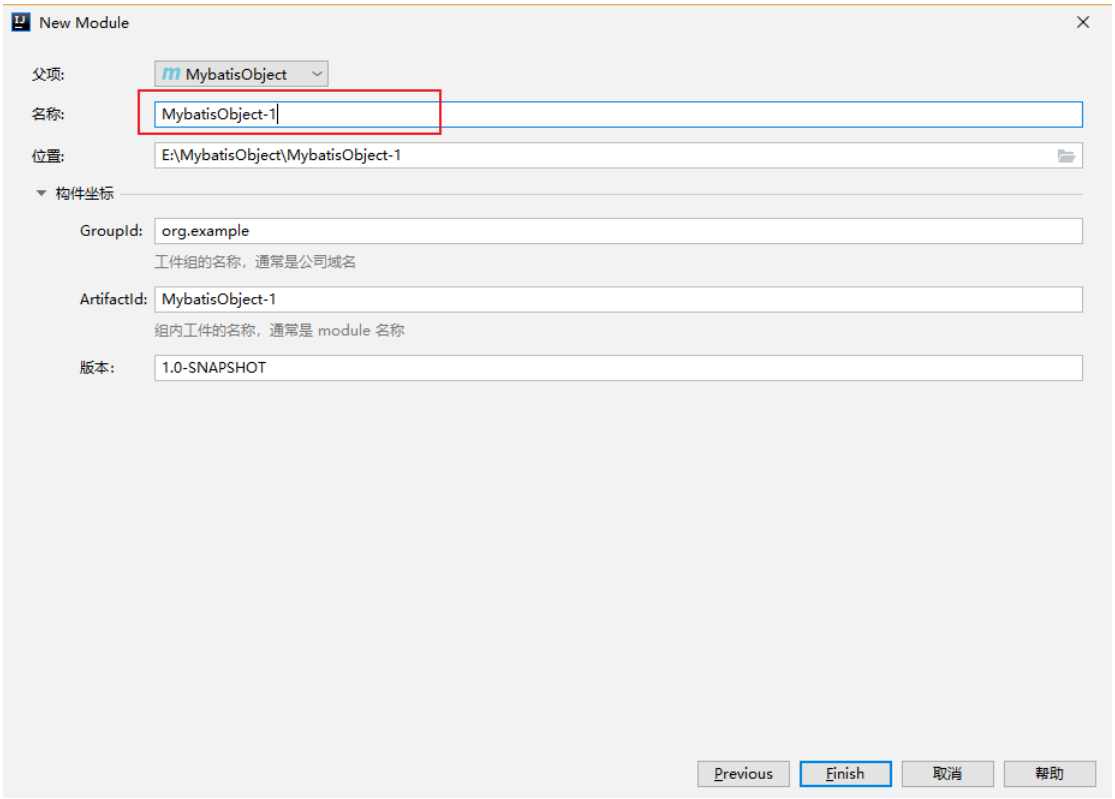
```
<dependencies>
    <!--mysql 驱动-->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.37</version>
    </dependency>
    <!--mybatis 驱动-->
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.5.7</version>
    </dependency>
    <!--junit-->
    <dependency>
```

```
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.11</version>
        <scope>test</scope>
    </dependency>
</dependencies>

<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.encoding>UTF-8</maven.compiler.encoding>
    <java.version>11</java.version>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
</properties>

<build>
    <resources>
        <resource>
            <directory>src/main/resources</directory>
            <includes>
                <include>**/*.properties</include>
                <include>**/*.xml</include>
            </includes>
            <filtering>true</filtering>
        </resource>
        <resource>
            <directory>src/main/java</directory>
            <includes>
                <include>**/*.properties</include>
                <include>**/*.xml</include>
            </includes>
            <filtering>true</filtering>
```

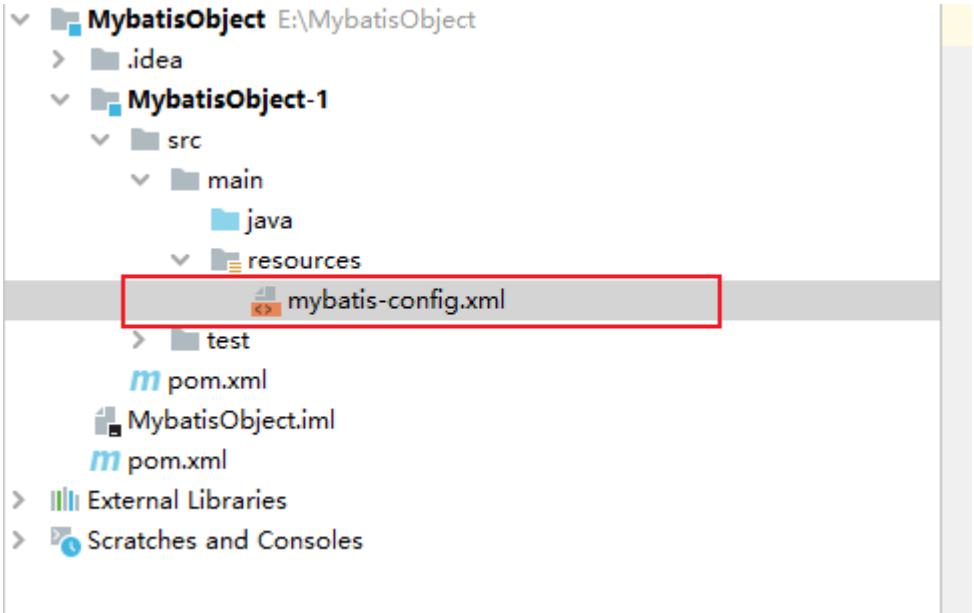
```
        </resource>
    </resources>
</build>
```

(四)、创建子项目





(五)、在 main—resource—创建 mybatis-config.xml



(六)、Mybatis-config.xml 配置

```
<!--Mybatis 核心配置-->
<configuration>
    <!--环境配置-->
    <environments default="development">
        <environment id="development">
            <!--事务使用的方式-->
            <transactionManager type="JDBC"/>
            <!--数据源-->
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.jdbc.Driver"/>
                <property name="url" value="jdbc:mysql://localhost:3306/mybatis?
useSSL=true&amp;useUniCode&amp;characterEncoding=UTF-8"/>
                <property name="username" value="root"/>
                <property name="password" value="Qwer1234"/>
            </dataSource>
        </environment>
    </environments>

   <mappers>
        <mapper resource="com/ma/dao/UserMappingImpl.xml"/>
    </mappers>


</configuration>
```

(七)、创建 pojo 实体类

```
public class User {


    private Integer id;
```

```
    private String name;
    private String pwd;
```

(八)、创建接口类

```
package com.ma.dao;

import com.ma.pojo.User;

import java.util.List;

public interface UserMapper {

    /**
     *  查询数据库所有用户信息
     * @return 返回用户信息 List
     */
    List<User> getUserAlls();
}
```

(九)、XML 方式实现接口

```
<!--创建映射关系指向接口-->
<mapper namespace="com.ma.dao.UserMapper">
    <!--
        查询所有用户信息
            id 为接口中的方法名    resultType    为返回值类型 返回一个
                                resultMap    为返回值类型 返回多个
    -->
```

```
    <select id="getUserAlls" resultType="com.ma.pojo.User">
        select id,name,pwd from user
  </select>
</mapper>
```

(十)、测试类

```
SqlSession sqlSession = ReadFileGetconnect.getSession();


@Test
public void test1(){
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);
    List<User> userAlls = mapper.getUserAlls();
    for (User userAll : userAlls) {
        System.out.println(userAll);
    }
}
```

三、增删查改

(一)接口

```
public interface UserMapper {


    /**
     *  查询数据库所有用户信息
     * @return 返回用户信息 List
     */
    List<User> getUserAlls();


    /**
```

```
     * 根据 id 查询员工信息
     * @param id 员工 id
     * @return 返回 user
     */
    User getUserById(Integer id);

    /**
     * 添加用户信息
     * @param user 添加的用户
     * @return 成功 1 失败 0
     */
    int insertUser(User user);

    /**
     * 更新用户信息
     * @param user 用户信息
     * @return 成功 1 失败 0
     */
    int updateUser(User user);

    /**
     * 根据 id 删除用户
     * @param id 用户 id
     * @return 成功 1 失败 0
     */
    int delUser(Integer id);
}
```

(二)、XML 实现接口

```
<!--创建映射关系指向接口-->
<mapper namespace="com.ma.dao.UserMapper">
    <!--查询所有用户信息-->
    <select id="getUserAlls" resultType="com.ma.pojo.User">
        select id,name,pwd from mybatis.user;
    </select>

    <!--根据 id 查询用户信息:parameterType 参数类型-->
    <select id="getUserById" parameterType="integer" resultType="com.ma.pojo.Us
er">
        select id,name,pwd from mybatis.user where id = #{id}
    </select>

    <!--插入用户-->
    <insert id="insertUser" parameterType="com.ma.pojo.User">
        insert into mybatis.user (id,name,pwd) values (#{id},#{name},#{pwd})
    </insert>

    <!--更新用户-->
    <update id="updateUser" parameterType="com.ma.pojo.User">
        update mybatis.user set name = #{name},pwd=#{pwd} where id=#{id}
    </update>

    <!--删除用户-->
    <delete id="delUser" parameterType="int">
        delete from mybatis.user where id=#{id}
    </delete>
</mapper>
```

(三)、测试

```
public class TextUserMapper {

    SqlSession sqlSession = ReadFileGetconnect.getSession();

    //查询所有信息
    @Test
    public void test1(){
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
        List<User> userAlls = mapper.getUserAlls();
        for (User userAll : userAlls) {
            System.out.println(userAll);
        }
    }

    //根据 id 查询信息
    @Test
    public void test2() {
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
        User userById = mapper.getUserById(1);
        System.out.println(userById);
        sqlSession.close();
    }

    //添加测试
    @Test
    public void test3() {
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
        int result = mapper.insertUser(new User(4, "哈哈", "123456"));
        if(result > 0){
```

```
            System.out.println("添加成功");
        }
        sqlSession.commit();
        sqlSession.close();
    }

    //修改测试
    @Test
    public void test4() {
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
        int result = mapper.updateUser(new User(4, "hehe", "123123"));
        if(result > 0){
            System.out.println("修改成功");
        }
        sqlSession.commit();
        sqlSession.close();
    }

    //删除测试
    @Test
    public void test5() {
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
        int result = mapper.delUser(4);
        if(result > 0){
            System.out.println("删除成功");
        }
        sqlSession.commit();
        sqlSession.close();
    }
}
```

四、万能的 Map,Map 作为参数操作

(一) 、说明

当查询的条件为少数的话就可以使用 map,而不用创建对象,这样节省资源
以查询添加为例

(二)、接口

```
public interface UserMapper {



/**
     * 使用 map 根据 id 查询
     * @param map map
     * @return 返回用户信息
     */
    User getUserByIdTwo(Map<String,Object> map);


     /**
     * 使用 map 添加数据
     * @param map map
     * @return 成功 1 失败 0
     */
    int insertUserTwo(Map<String,Object> map);

}
```

(三)、XML 实现

```
<!--根据 id 查询用户信息:parameterType 参数类型-->
<select id="getUserByIdTwo" parameterType="map" resultType="com.ma.pojo.User">
    select id,name,pwd from mybatis.user where id = #{id}
</select>

<!--插入用户-->
<insert id="insertUserTwo" parameterType="map">
    insert into mybatis.user (id,name,pwd) values (#{id},#{name},#{pwd})
</insert>
```

(四)、测试

```
public class TextUserMapper {

    SqlSession sqlSession = ReadFileGetconnect.getSession();

    @Test
    //使用 Map 根据 id 查询
    public void test6() {
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
        Map<String,Object> map = new HashMap<>();
        map.put("id",2);
        User userById = mapper.getUserByIdTwo(map);
        System.out.println(userById);
        sqlSession.close();
    }

    @Test
```

```
    //使用 map 添加数据
    public void test7() {
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
        Map<String,Object> map = new HashMap<>();
        map.put("id",6);
        map.put("name","匹配");
        map.put("pwd","123444");
        int result = mapper.insertUserTwo(map);
        if(result > 0){
            System.out.println("添加成功");
        }
        sqlSession.commit();
        sqlSession.close();
    }
}
```

五、模糊查询

(一)、接口

List<User> mohuSel(String name);

(二)、实现 XML

```
<select id="mohuSel" parameterType="string" resultType="com.ma.pojo.User">
    select * from mybatis.user where name like "%"#{value}"%"
</select>
```

(三)、测试

```
@Test
public void mohuSel(){
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);
    List<User> user = mapper.mohuSel("王");
    for (User user1 : user) {
        System.out.println(user1);
    }
    sqlSession.close();
}
```

六、Mybatis 配置解析

(一)、环境

Mybatis 可以适用于多种环境

(二)、默认配置

mybatis 默认的事务管理器是 jdbc。连接池为 pooled

(三)、提取数据库连接属性

1、创建配置文件

```
drive=com.mysql.jdbc.Driver
url=jdbc:mysql://localhost:3306/mybatis?useSSL=true&useUniCode&characterEnc
oding=UTF-8
username=root
password=Qwer1234
```

2、XML 引入配置文件

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

3、使用${}匹配连接数据库属性

```
    <!--环境配置-->
    <environments default="development">
        <environment id="development">
            <!--事务使用的方式-->
            <transactionManager type="JDBC"/>
            <!--数据源-->
            <dataSource type="POOLED">
                <property name="driver" value="${drive}"/>
                <property name="url" value="${url}"/>
                <property name="username" value="${username}"/>
                <property name="password" value="${password}"/>
            </dataSource>
        </environment>
    </environments>
```

4、可以在< properties >中添加额外的属性

注意点:如果 name 重名,优先使用配置文件中的属性值

```
<properties resource="jdbcAttributes.properties">
    <property name="aaaa" value="11111"/>
</properties>
```

(四)、别名

有 2 种起别名的方式:
第一种为指定类分配别名
第二种:为包起别名,包下的类别名为类名首字母小写

1、第一种

```
<typeAliases>

  <typeAlias alias="user" type="com.ma.pojo.User "/>

</typeAliases>
```

2、第二种

```
<typeAliases>

  <package name="com.ma"/>

</typeAliases>
```

3、第一种使用

```
<!--根据id查询用户信息:parameterType参数类型-->
<select id="getUserById" parameterType="integer" resultType="userAlias">
    select id,name,pwd from mybatis.user where id = #{id}
</select>
```

原来的 com.ma.pojo.User 可以简写为别名形式

4、第二种使用

包下面类名首字母小写

```
<!--根据id查询用户信息:parameterType参数类型-->
<select id="getUserByIdTwo" parameterType="map" resultType="user">
    select id,name,pwd from mybatis.user where id = #{id}
</select>
```

(六)、设置事务自动提交

在返回连接的时候添加参数 true

```
//返回连接
public static SqlSession getSession(){
    return sqlSessionFactory.openSession(true);
}
```

(五)、Mybatis 属性设置

设置（settings）

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

设置名	描述	有效值	默认值
cacheEnabled	全局性地开启或关闭所有映射器配置文件中已配置的任何缓存。	true | false	true
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 `fetchType` 属性来覆盖该项的开关状态。	true | false	false
aggressiveLazyLoading	开启时，任一方法的调用都会加载该对象的所有延迟加载属性。 否则，每个延迟加载属性会按需加载（参考 `lazyLoadTriggerMethods`）。	true | false	false （在 3.4.1 及之前的版本中默认为 true）
multipleResultSetsEnabled	是否允许单个语句返回多结果集（需要数据库驱动支持）。	true | false	true
useColumnLabel	使用列标签代替列名。实际表现依赖于数据库驱动，具体可参考数据库驱动的相关文档，或通过对比测试来观察。	true | false	true
useGeneratedKeys	允许 JDBC 支持自动生成主键，需要数据库驱动支持。如果设置为 true，将强制使用自动生成主键。尽管一些数据库驱动不支持此特性，但仍可正常工作（如 Derby）。	true | false	False
autoMappingBehavior	指定 MyBatis 应如何自动映射列到字段或属性。 NONE 表示关闭自动映射；PARTIAL 只会自动映射没有定义嵌套结果映射的字段。 FULL 会自动映射任何复杂的结果集（无论是否嵌套）。	NONE, PARTIAL, FULL	PARTIAL
autoMappingUnknownColumnBehavior	指定发现自动映射目标未知列（或未知属性类型）的行为。 • `NONE`：不做任何反应 • `WARNING`：输出警告日志（`'org.apache.ibatis.session.AutoMappingUnknownColumnBehavior'` 的日志等级必须设置为 `WARN`） • `FAILING`：映射失败 (抛出 `SqlSessionException`)	NONE, WARNING, FAILING	NONE
defaultExecutorType	配置默认的执行器。SIMPLE 就是普通的执行器；REUSE 执行器会重用预处理语句（PreparedStatement）； BATCH 执行器不仅重用语句还会执行批量更新。	SIMPLE REUSE BATCH	SIMPLE
defaultStatementTimeout	设置超时时间，它决定数据库驱动等待数据库响应的秒数。	任意正整数	未设置 (null)
defaultFetchSize	为驱动的结果集获取数量（fetchSize）设置一个建议值。此参数只可以在查询设置中被覆盖。	任意正整数	未设置 (null)
defaultResultSetType	指定语句默认的滚动策略。（新增于 3.5.2）	FORWARD_ONLY | SCROLL_SENSITIVE | SCROLL_INSENSITIVE | DEFAULT（等同于未设置）	未设置 (null)

safeRowBoundsEnabled	是否允许在嵌套语句中使用分页（RowBounds）。如果允许使用则设置为 false。	true | false	False
safeResultHandlerEnabled	是否允许在嵌套语句中使用结果处理器（ResultHandler）。如果允许使用则设置为 false。	true | false	True
mapUnderscoreToCamelCase	是否开启驼峰命名自动映射，即从经典数据库列名 A_COLUMN 映射到经典 Java 属性名 aColumn。	true | false	False
localCacheScope	MyBatis 利用本地缓存机制（Local Cache）防止循环引用和加速重复的嵌套查询。 默认值为 SESSION，会缓存一个会话中执行的所有查询。 若设置值为 STATEMENT，本地缓存将仅用于执行语句，对相同 SqlSession 的不同查询将不会进行缓存。	SESSION | STATEMENT	SESSION
jdbcTypeForNull	当没有为参数指定特定的 JDBC 类型时，空值的默认 JDBC 类型。 某些数据库驱动需要指定列的 JDBC 类型，多数情况直接用一般类型即可，比如 NULL、VARCHAR 或 OTHER。	JdbcType 常量，常用值：NULL、VARCHAR 或 OTHER。	OTHER
lazyLoadTriggerMethods	指定对象的哪些方法触发一次延迟加载。	用逗号分隔的方法列表。	equals,clone,hashCode,toString
defaultScriptingLanguage	指定动态 SQL 生成使用的默认脚本语言。	一个类型别名或全限定类名。	org.apache.ibatis.scripting.xmltags.XMLLanguageDriver
defaultEnumTypeHandler	指定 Enum 使用的默认 `TypeHandler`。（新增于 3.4.5）	一个类型别名或全限定类名。	org.apache.ibatis.type.EnumTypeHandler
callSettersOnNulls	指定当结果集中值为 null 的时候是否调用映射对象的 setter（map 对象时为 put）方法，这在依赖于 Map.keySet() 或 null 值进行初始化时比较有用。注意基本类型（int、boolean 等）是不能设置成 null 的。	true | false	false
returnInstanceForEmptyRow	当返回行的所有列都是空时，MyBatis默认返回 `null`。 当开启这个设置时，MyBatis会返回一个空实例。 请注意，它也适用于嵌套的结果集（如集合或关联）。（新增于 3.4.2）	true | false	false
logPrefix	指定 MyBatis 增加到日志名称的前缀。	任何字符串	未设置
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
proxyFactory	指定 Mybatis 创建可延迟加载对象所用到的代理工具。	CGLIB | JAVASSIST	JAVASSIST （MyBatis 3.3 以上）
vfsImpl	指定 VFS 的实现	自定义 VFS 的实现的类全限定名，以逗号分隔。	未设置
useActualParamName	允许使用方法签名中的名称作为语句参数名称。 为了使用该特性，你的项目必须采用 Java 8 编译，并且加上 `-parameters` 选项。（新增于 3.4.1）	true | false	true
configurationFactory	指定一个提供 `Configuration` 实例的类。 这个被返回的 Configuration 实例用来加载被反序列化对象的延迟加载属性值。 这个类必须包含一个签名为 `static Configuration getConfiguration()` 的方法。（新增于 3.2.3）	一个类型别名或完全限定类名。	未设置
shrinkWhitespacesInSql	从SQL中删除多余的空格字符。请注意，这也会影响SQL中的文字字符串。 (新增于 3.5.5)	true | false	false
defaultSqlProviderType	Specifies an sql provider class that holds provider method (Since 3.5.6). This class apply to the `type` (or `value`) attribute on sql provider annotation(e.g. `@SelectProvider`), when these attribute was omitted.	A type alias or fully qualified class name	Not set

```
<settings>

  <setting name="cacheEnabled" value="true"/>

  <setting name="lazyLoadingEnabled" value="true"/>

  <setting name="multipleResultSetsEnabled" value="true"/>

  <setting name="useColumnLabel" value="true"/>
```

```
  <setting name="useGeneratedKeys" value="false"/>

  <setting name="autoMappingBehavior" value="PARTIAL"/>

  <setting name="autoMappingUnknownColumnBehavior" value="WARNING"/>

  <setting name="defaultExecutorType" value="SIMPLE"/>

  <setting name="defaultStatementTimeout" value="25"/>

  <setting name="defaultFetchSize" value="100"/>

  <setting name="safeRowBoundsEnabled" value="false"/>

  <setting name="mapUnderscoreToCamelCase" value="false"/>

  <setting name="localCacheScope" value="SESSION"/>

  <setting name="jdbcTypeForNull" value="OTHER"/>

  <setting name="lazyLoadTriggerMethods" value="equals,clone,hashCode,toString"
/>

</settings>
```

(六)、Mybatis 的插件

(七)、映射器(mapper)

1、第一种方式:通过 resource

resource 指向实现 XML 的全类名

```
<mappers>
    <mapper resource="com/ma/dao/UserMappingImpl.xml"/>
</mappers>
```

2、第二种方式:通过 Class 查找

Class 指向实现接口的全类名

```
<mappers>
    <mapper class="com.ma.dao.UserMapper"/>
</mappers>
```

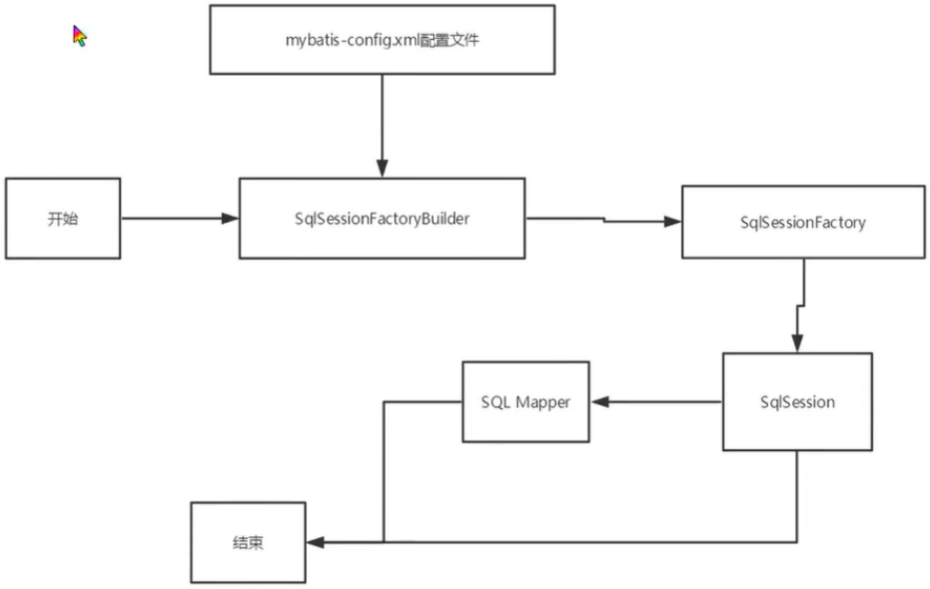
3、第三种方式:通过包扫描

```
<mappers>

  <package name="com.ma.dao"/>

</mappers>
```

4、注意点:

使用第二种和第三种方式,接口和实现接口的名字必须和接口同名并且在同一个包下

(八)、作用域（Scope）和生命周期



1、SqlSessionFactoryBuilder

为了创建 SqlSessionFactory 工厂，一旦创建了 SqlSessionFactory，就不再需要它了。

2、SqlSessionFactory

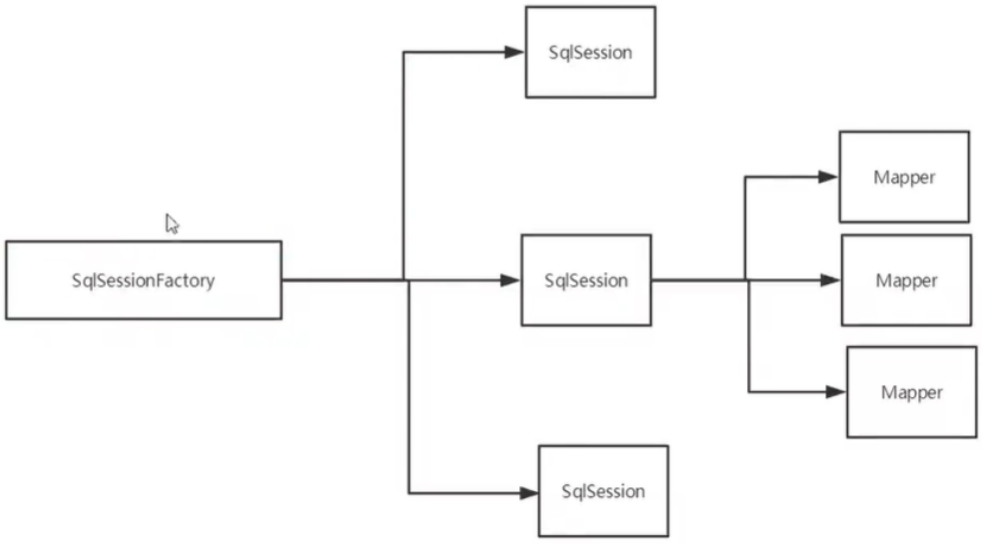
为了创建 SqlSession,SqlSessionFactory 一旦被创建就应该在应用的运行期间一直存在，没有任何理由丢弃它或重新创建另一个实例。

3、SqlSession

每个线程都应该有它自己的 SqlSession 实例。SqlSession 的实例不是线程安全的，因此是不能被共享的,一般写在方法中。

4、Mapping

映射器是一些绑定映射语句的接口。映射器接口的实例是从 SqlSession 中获得的。每一个 Mapping 都是一个业务



七、解决数据库字段和实体类属性不一致问题

(一)、第一种方式:起别名

select id,name,pwd as password from user;

(二)、第二种方式:使用 resultMap

```
<!--使用映射-->
<select id="getUserById" parameterType="integer" resultMap="MapUser">
    select id,name,pwd from mybatis.user where id = #{id}
</select>
```

resultMap 的属性值设置为映射匹配 id

```
<!--设置字段和属性匹配-->
<resultMap id="MapUser" type="user">
```

```
    <!—类属性:password    数据库字段:pwd-->
    <result property="password" column="pwd"/>
</resultMap>
```

八、日志

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)、STDOUT_LOGGING:

java 自带的日志

1、设置

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

2、输出

```
Opening JDBC Connection
Created connection 1020154737.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@3cce5371]
==>  Preparing: select id,name,pwd from mybatis.user;
==> Parameters:
<==    Columns: id, name, pwd
<==        Row: 1, 马佳盛, 12345
<==        Row: 2, 王子汎, 12345
<==        Row: 3, 王宝延, 12345
<==        Row: 6, 匹配, 123444
<==      Total: 4
User{id=1, name='马佳盛', password='12345'}
User{id=2, name='王子汎', password='12345'}
User{id=3, name='王宝延', password='12345'}
User{id=6, name='匹配', password='123444'}
```

(二)、Log4J

1、导入依赖

```
<!--日志-->
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
</dependency>
```

2、创建一个 log4j 的配置文件添加配置

```
#将等级为 DEBUG 的日志信息输出到 console 和 file 这两个目的地，console 和 file 的定义在下面的代码
log4j.rootLogger=DEBUG,console,file

#控制台输出的相关设置
log4j.appender.console = org.apache.log4j.ConsoleAppender
log4j.appender.console.Target = System.out
log4j.appender.console.Threshold=DEBUG
log4j.appender.console.layout = org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=[%c]-%m%n

#文件输出的相关设置
log4j.appender.file = org.apache.log4j.RollingFileAppender
log4j.appender.file.File=./log/ma.log
log4j.appender.file.MaxFileSize=10mb
log4j.appender.file.Threshold=DEBUG
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=[%p][%d{yy-MM-dd}][%c]%m%n
```

```
#日志输出级别
log4j.logger.org.mybatis=DEBUG
log4j.logger.java.sql=DEBUG
log4j.logger.java.sql.Statement=DEBUG
log4j.logger.java.sql.ResultSet=DEBUG
log4j.logger.java.sql.PreparedStatement=DEBUG
```

3、Mybatis 主配置文件设置

```
<!--日志设置-->
<settings>
    <setting name="logImpl" value="LOG4J"/>
</settings>
```

4、输出

```
[org.apache.ibatis.logging.LogFactory]-Logging initialized using 'class org.apache.ibatis.logging.log4j.Log4jImpl' adapter.
[org.apache.ibatis.logging.LogFactory]-Logging initialized using 'class org.apache.ibatis.logging.log4j.Log4jImpl' adapter.
[org.apache.ibatis.io.VFS]-Class not found: org.jboss.vfs.VFS
[org.apache.ibatis.io.JBoss6VFS]-JBoss 6 VFS API is not available in this environment.
[org.apache.ibatis.io.VFS]-Class not found: org.jboss.vfs.VirtualFile
[org.apache.ibatis.io.VFS]-VFS implementation org.apache.ibatis.io.JBoss6VFS is not valid in this environment.
[org.apache.ibatis.io.VFS]-Using VFS adapter org.apache.ibatis.io.DefaultVFS
[org.apache.ibatis.io.DefaultVFS]-Find JAR URL: file:/E:/MybatisObject/MybatisObject-1/target/classes/com/ma/pojo
[org.apache.ibatis.io.DefaultVFS]-Not a JAR: file:/E:/MybatisObject/MybatisObject-1/target/classes/com/ma/pojo
[org.apache.ibatis.io.DefaultVFS]-Reader entry: User.class
[org.apache.ibatis.io.DefaultVFS]-Listing file:/E:/MybatisObject/MybatisObject-1/target/classes/com/ma/pojo
[org.apache.ibatis.io.DefaultVFS]-Find JAR URL: file:/E:/MybatisObject/MybatisObject-1/target/classes/com/ma/pojo/User.class
[org.apache.ibatis.io.DefaultVFS]-Not a JAR: file:/E:/MybatisObject/MybatisObject-1/target/classes/com/ma/pojo/User.class
[org.apache.ibatis.io.DefaultVFS]-Reader entry: ????7:
[org.apache.ibatis.io.ResolverUtil]-Checking to see if class com.ma.pojo.User matches criteria [is assignable to Object]
[org.apache.ibatis.datasource.pooled.PooledDataSource]-PooledDataSource forcefully closed/removed all connections.
[org.apache.ibatis.datasource.pooled.PooledDataSource]-PooledDataSource forcefully closed/removed all connections.
[org.apache.ibatis.datasource.pooled.PooledDataSource]-PooledDataSource forcefully closed/removed all connections.
[org.apache.ibatis.datasource.pooled.PooledDataSource]-PooledDataSource forcefully closed/removed all connections.
[org.apache.ibatis.transaction.jdbc.JdbcTransaction]-Opening JDBC Connection
[org.apache.ibatis.datasource.pooled.PooledDataSource]-Created connection 1208532123.
[org.apache.ibatis.transaction.jdbc.JdbcTransaction]-Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@4808bc9b]
[com.ma.dao.UserMapper.getUserAlls]-==>  Preparing: select id,name,pwd from mybatis.user;
[com.ma.dao.UserMapper.getUserAlls]-==> Parameters:
[com.ma.dao.UserMapper.getUserAlls]-<==      Total: 4
User{id=1, name='马佳盛', password='12345'}
User{id=2, name='王子汎', password='12345'}
User{id=3, name='王宝延', password='12345'}
User{id=6, name='匹配', password='123444'}
```

九、分页查询

(一)、接口

```
/**
 * 分页查询
 * @param pageInt 分页起始位置和查询几个
 * @return 返回 User 数组
 */
List<User>pageUsers(Map<String,Integer> pageInt);
```

(二)、XML 实现

```
<!--分页查询-->
<select id="pageUsers" parameterType="map" resultMap="MapUser">
    select id,name,pwd from mybatis.user limit #{startIndex},#{endIndex}
</select>
```

(三)、测试

```
@Test
public void textPageUsers() {
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);
    Map<String,Integer> map = new HashMap<>();
    map.put("startIndex",0);
    map.put("endIndex",2);
    List<User> users = mapper.pageUsers(map);
    for (User user : users) {
        System.out.println(user);
    }
```

```
    sqlSession.close();
}
```

十、使用注解实现 SQl 语句增删查改

(一)、接口

```
public interface UserMapperOne {


    /**
     * 查询数据库所有用户信息
     *
     * @return 返回用户信息 List
     */
    @Select("select id,name,pwd as password from user")
    List<User> getUserAlls();


    /**
     * 根据 id 查询用户
     *
     * @param id id 号
     * @return 成功返回用户  失败 null
     */
    @Select("select id,name,pwd as password from user where id = #{id}")
    User getUserById(@Param("id") int id);


    /**
     * 根据 id,用户名查询用户
     *
     * @param id   id 号
     * @param name 用户名
```

```
     * @return 成功返回用户  失败 null
     */
    @Select("select id,name,pwd as password from user where id = #{id}")
    User getUserByIdAndName(@Param("id") int id, @Param("name") String name);

    /**
     * 添加用户
     *
     * @param user 用户
     * @return 成功 1 失败-1
     */
    @Insert("insert into user (id,name,pwd) values (#{id},#{name},#{password})
")
    int addUser(User user);

    /**
     * 更新用户
     * @param user 用户给
     * @return 成功 1 失败-1
     */
    @Update("update user set name=#{name},pwd=#{password} where id=#{id}")
    int updateUser(User user);

    /**
     * 根据 id 删除用户
     * @param id id
     * @return 成功 1 失败-1
     */
    @Delete("delete from user where id =#{id}")
    int deleteUser(Integer id);
}
```

(二)、Mybatis 配置文件添加映射

```
<mappers>
    <mapper class="com.ma.dao.UserMapper"/>
</mappers>
```

(三)、测试

```
public class TextUserMapper {


    SqlSession sqlSession = ReadFileGetconnect.getSession();


    //查询所有信息
    @Test
    public void test1(){
        UserMapperOne mapper = sqlSession.getMapper(UserMapperOne.class);
        List<User> userAlls = mapper.getUserAlls();
        for (User userAll : userAlls) {
            System.out.println(userAll);
        }
    }


    @Test
    public void test2(){
        UserMapperOne mapper = sqlSession.getMapper(UserMapperOne.class);
        User user = mapper.getUserById(1);
        System.out.println(user);
        sqlSession.close();
    }


    @Test
```

```
    public void test3(){
        UserMapperOne mapper = sqlSession.getMapper(UserMapperOne.class);
        int user = mapper.addUser(new User(8,"张三","9999"));
        sqlSession.close();
    }

    @Test
    public void test4(){
        UserMapperOne mapper = sqlSession.getMapper(UserMapperOne.class);
        mapper.updateUser(new User(8,"李四","8888"));
        sqlSession.close();
    }

    @Test
    public void test5(){
        UserMapperOne mapper = sqlSession.getMapper(UserMapperOne.class);
        mapper.deleteUser(8);
        sqlSession.close();
    }

}
```

(四)、@param 属性

@param 属性如果是基本数据类型或 String 建议加上,可以规范 sql 语句中字段的名字

```
@Select("select id,name,pwd as password from user where id = #{id} and name=#{n
ame}")
User getUserByIdAndName(@Param("id") int id, @Param("name") String name);
```

十一、LomBok

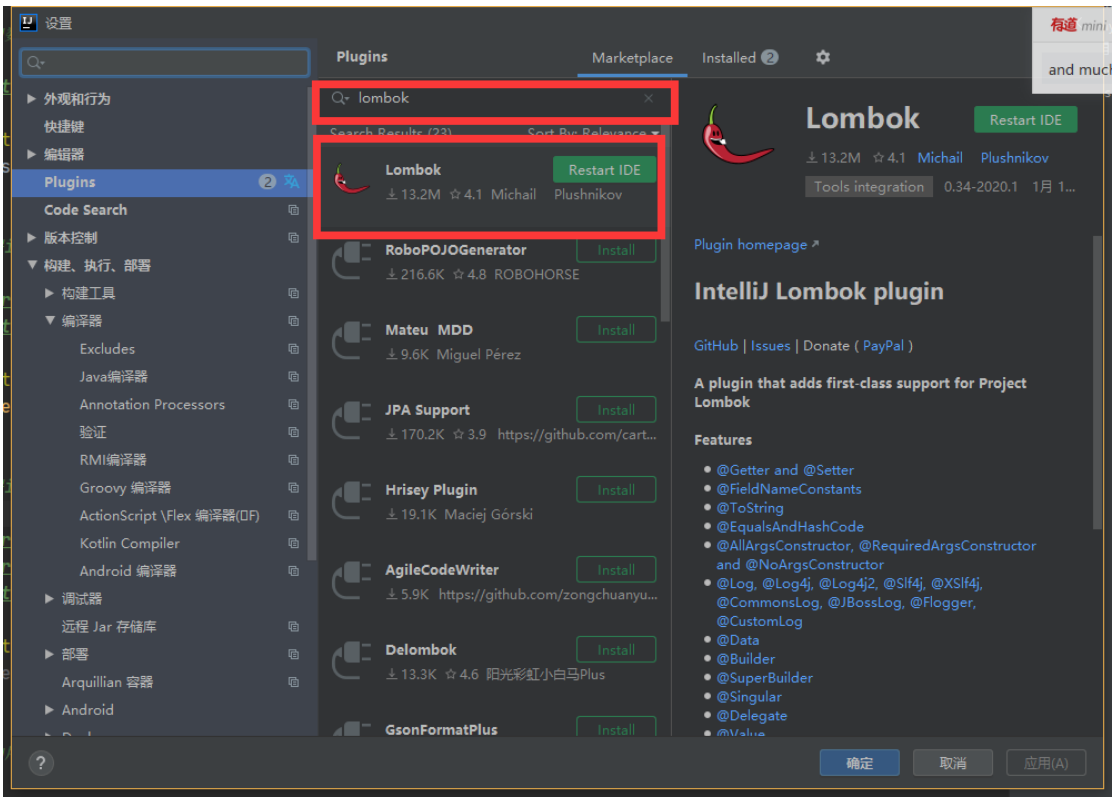
(一)、简介

Project Lombok is a java library that automatically plugs into your editor and build tools, spicing up your java.

Never write another getter or equals method again, with one annotation your class has a fully featured builder, Automate your logging variables, and much more.

(二)、使用步骤

1、安装插件



(二)、导入 jar 包

```
<!—LomBok 依赖-->
```

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

(三)、LomBok 中的注解

注解	使用的位置	说明	示例
@Getter	属性,类	放在属性上生成一个 get 方法,放在类上都生成	@Getter private int age = 10;
@Setter	属性,类	放在属性上生成一个 set 方法,放在类上都生成	@Setter private int age = 10;
@NonNull	属性	没有值赋值为空	public NonNullExample(@NonNull Person person) {
@Data	类	自动生成无参构造,setter/getter、equals、canEqual 、	@Data public class DataExample {

注解	使用的位置	说明	示例
		hashCode 、toString 方法，如为 final 属性，则不会为该属性生成 setter 方法。	
@Cleanup	需要关闭的变量前面	该注解能帮助我们自动调用 close()方法	@Cleanup InputStream in = new FileInputStream(args[0]);
@EqualsAndHashCode	类	自动生成 EqualsAndHashCode,(exclude)排除属性	@EqualsAndHashCode(exclude={"id", "shape"}) public class EqualsAndHashCodeExample {
@ToString	类	自动生成 toString 方法 ,(exclude)排除属性	@ToString(exclude="id") public class ToStringExample {
@NoArgsConstructor	类	无参构造	@NoArgsConstructor public static class NoArgsExample {
@RequiredArgsConstructor	类	部分属性构造	@RequiredArgsConstructor public static class NoArgsExample {

注解	使用的位置	说明	示例
@AllArgsConstructor	类	全部属性构造	@AllArgsConstructor public static class NoArgsExample {

(四)、使用

```
@Data
@AllArgsConstructor
public class User {

    private Integer id;
    private String name;
    private String password;
}
```

十二、一对多,多对一

(一)、说明

多个学生对应一个老师　　多对一

一个老师对应多个学生　　一对多

(二)、环境搭建

1、搭建数据库

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

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

CREATE TABLE `student` (
  `id` INT(10) NOT NULL,
  `name` VARCHAR(30) DEFAULT NULL,
  `tid` INT(10) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `fktid` (`tid`),
  CONSTRAINT `fktid` FOREIGN KEY (`tid`) REFERENCES `teacher` (`id`)
) ENGINE=INNODB DEFAULT CHARSET=utf8;
INSERT INTO `student` (`id`, `name`, `tid`) VALUES (1, "小明", 1);
INSERT INTO `student` (`id`, `name`, `tid`) VALUES (2, "小红", 1);
INSERT INTO `student` (`id`, `name`, `tid`) VALUES (3, "小张", 1);
INSERT INTO `student` (`id`, `name`, `tid`) VALUES (4, "小李", 1);
INSERT INTO `student` (`id`, `name`, `tid`) VALUES (5, "小王", 1);
```

2、创建工程

3、创建 pojo 类

(1)、Teacher

```
@Data
public class Teacher {
 //多对一
    private Integer id;
    private String name;

    //一对多
    private Integer id;
    private String name;
    private List<Student> student;
}
```

(2)、Student

```
@Data
public class Student {
 //多对一
    private Integer id;
    private String name;
    private Teacher teacher;

    //一对多
    private Integer id;
    private String name;
    private Intege teacher;
}
```

4、创建对应 pojo 接口

5、创建对应实现 XML,并添加头文件

Student

```
<?xml version="1.0" encoding="UTF8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<!--Mybatis 核心配置-->
<mapper namespace="com.ma.dao.TeacherMapping">


</mapper>
```

Teacher

```
<?xml version="1.0" encoding="UTF8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<!--Mybatis 核心配置-->
<mapper namespace="com.ma.dao.StudentMapping">


</mapper>
```

6.Mybatis 中映射 XML

```
<!--映射实现方式-->
<mappers>
    <mapper resource="com/ma/dao/StudentMapping.xml"/>
```

```
    <mapper resource="com/ma/dao/TeacherMapping.xml"/>
</mappers>
```

7.测试

(1)、接口

```
public interface TeacherMapping {


    @Select("select * from teacher where id = #{id}")
    Teacher getTeacher(@Param("id") Integer id);
}
```

(2)、Test

```
public class text {


    public static void main(String[] args) {
        SqlSession session = ReadFileGetconnect.getSession();
        TeacherMapping mapper = session.getMapper(TeacherMapping.class);
        Teacher teacher = mapper.getTeacher(1);
        System.out.println(teacher);
        session.close();
    }
}
```

(三)、多对一查询

1、方式一:子查询标签嵌套方式

(1)、接口

```
public interface StudentMapping {

    List<Student> getStudentsAndTeacherNames();

    Teacher getTeacherById(Integer id);
}
```

(二)、XML

```
<!--映射-->
<resultMap id="StudentAndTeacher" type="Student">
    <result property="id" column="id"/>
    <result property="name" column="name"/>
    <!--对象属性使用 association    集合使用 Collection-->
    <!--javaType:java 的类    select:子查询-->
    <association property="teacher" column="tid" javaType="Teacher"
select="getTeacherById"/>
</resultMap>

<!--使用 Map 映射属性-->
<select id="getStudentsAndTeacherNames" resultMap="StudentAndTeacher">
    select * from student;
</select>

<select id="getTeacherById" resultType="Teacher">
    select id,name from teacher where id = #{tid}
</select>
```

2、方式二:联表查询

(一)、接口

```
public interface StudentMapping {

    List<Student> getStudentsAndTeacherNames();

    Teacher getTeacherById(Integer id);
}
```

(二)、XML

```
<!--映射-->
<resultMap id="StudentAndTeacher" type="Student">
      <result property="id" column="sid"/>
      <result property="name" column="sname"/>
      <!--对象属性使用 association    集合使用 Collection-->
      <!--javaType:java 的类    select:子查询-->
      <association property="teacher" javaType="Teacher">
          <result property="name" column="tname"/>
      </association>
</resultMap>

<!--使用 Map 映射属性-->
<select id="getStudentsAndTeacherNames" resultMap="StudentAndTeacher">
      SELECT s.id AS sid,s.name AS sname,t.name tname
      FROM student AS s,teacher AS t
      WHERE s.tid=t.id;
</select>
```

(四)、一对多

1、方式一:子查询标签嵌套方式

(1)、接口

```
public interface TeacherMapping {

    List<Teacher> getStudentAndTeachers(@Param("id") Integer id);

    Student getStudentInfo();

}
```

(2)、XML

```
<!--Mybatis 核心配置-->
<mapper namespace="com.ma.dao.TeacherMapping">
    <select id="getStudentAndTeachers" resultMap="getStudentAndTeachersMapping">
        select * from teacher where id=#{id};
    </select>
    <resultMap id="getStudentAndTeachersMapping" type="Teacher">
        <collection property="student" javaType="ArrayList" ofType="Student"
select="getStudentInfo" column="id"/>
    </resultMap>
    <select id="getStudentInfo" resultType="Student">
        select * from student where tid=#{id};
    </select>
</mapper>
```

(3)、测试

```
public class aa {


    @Test
    public void test1(){
        SqlSession session = ReadFileGetconnect.getSession();
        TeacherMapping mapper = session.getMapper(TeacherMapping.class);
        List<Teacher> studentAndTeachers = mapper.getStudentAndTeachers(1);
        for (Teacher studentAndTeacher : studentAndTeachers) {
            System.out.println(studentAndTeacher);
        }
        session.close();
    }
}
```

2、方式二:联表查询

(1)、接口

```
public interface TeacherMapping {


    List<Teacher> getStudentAndTeachers(@Param("id") Integer id);


}
```

(2)、XML

```
<select id="getStudentAndTeachers" resultMap="getStudentAndTeacherMapping">
    select s.id sid,s.name sname,t.name tname,t.id tid
    from student as s,teacher t
    where s.tid = t.id and t.id=#{id}
</select>
<resultMap id="getStudentAndTeacherMapping" type="Teacher">
```

```
    <result property="id" column="tid"/>
    <result property="name" column="tname"/>
    <collection property="student" ofType="Student">
        <result property="id" column="sid"/>
        <result property="name" column="sname"/>
        <result property="tid" column="tid"/>
    </collection>
</resultMap>
```

(3)、测试

```
public class aa {


    @Test
    public void test1(){
        SqlSession session = ReadFileGetconnect.getSession();
        TeacherMapping mapper = session.getMapper(TeacherMapping.class);
        List<Teacher> studentAndTeachers = mapper.getStudentAndTeachers(1);
        for (Teacher studentAndTeacher : studentAndTeachers) {
            System.out.println(studentAndTeacher);
        }
        session.close();
    }
}
```

十三、动态 SQL

(一)、概念

根据不通的条件生成不通的动态 SQL 语句,其中有 4 中标签

- if 判断
- choose (when, otherwise) switch
- trim (where, set) 尽量不让 sql 出错
- foreach 循环

(二)、环境搭建

1、数据库

```
CREATE TABLE `blog`(
  `id` VARCHAR(50) NOT NULL COMMENT '博客 id',
  `title` VARCHAR(100) NOT NULL COMMENT '博客标题',
  `author` VARCHAR(30) NOT NULL COMMENT '博客作者',
  `create_time` DATETIME NOT NULL COMMENT '创建时间',
  `views` INT(30) NOT NULL COMMENT '浏览量'

)ENGINE=INNODB DEFAULT CHARSET=utf8
```

2、创建对应的 pojo

```
@Data
public class Blog {

    private String id;
    private String title;
    private String author;
    private Date createTime;
    private Integer views;
}
```

3、创建接口

```
public interface BlogMapping {
      int addBlog(Blog blog);
}
```

4、创建接口对应 XML

```
<?xml version="1.0" encoding="UTF8" ?>
<!DOCTYPE mapper
         PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
         "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<!--Mybatis 核心配置-->
<mapper namespace="com.ma.dao.BlogMapping">
      <insert id="addBlog" parameterType="com.ma.pojo.Blog">
        insert into blog (id,title,author,create_time,views)
        values (#{id},#{title},#{author},#{createTime},#{views})
    </insert>
</mapper>
```

5、Mybatis 配置文件添加映射

```
<!--映射实现方式-->
<mappers>
    <mapper resource="com/ma/dao/BlogMapping.xml"/>
</mappers>
```

添加字段转换 如:数据库中 user_name java 中 userName 这样形式的自动转换

```
<settings>
    <setting name="mapUnderscoreToCamelCase" value="true"/>
</settings>
```

6、添加一个工具类生成不同的编号

```
public class GenerateUUIDUtils {

    public static String getUUID(){
        return UUID.randomUUID().toString().replaceAll("-","");
    }
}
```

7、数据库注入数据

```
@Test
public void test6() {
    SqlSession sqlSession = ReadFileGetconnect.getSession();
    com.ma.dao.BlogMapping mapper =
sqlSession.getMapper(com.ma.dao.BlogMapping.class);
    Blog blog = new Blog();
    blog.setId(GenerateUUIDUtils.getUUID());
    blog.setTitle("Mybatis");
    blog.setAuthor("狂神说");
    blog.setCreateTime(new Date());
    blog.setViews(9999);

    mapper.addBlog(blog);

    blog.setId(GenerateUUIDUtils.getUUID());
```

```
    blog.setTitle("Java");
    mapper.addBlog(blog);

    blog.setId(GenerateUUIDUtils.getUUID());
    blog.setTitle("Spring");
    mapper.addBlog(blog);

    blog.setId(GenerateUUIDUtils.getUUID());
    blog.setTitle("微服务");
    mapper.addBlog(blog);

    sqlSession.close();
}
```

(三)、标签

1、if 标签

(1)、语法:

```
<if title=""></if>
```

(2)、使用

接口

```
public interface BlogMapping {

    List<Blog> queryBlog(Map<String,Object> map);
}
```

XML

```xml
<select id="queryBlog" parameterType="map" resultType="Blog">
    <!--查询所有语句-->
    select * from blog where 1=1
    <!--如果 title 有值的话拼接 if 中语句-->
    <if test="title != null">
        and title = #{title}
    </if>
    <!--如果 author 有值的话拼接 if 中语句-->
    <if test="author != null">
        and author = #{author}
    </if>
</select>
```

测试

```
@Test
public void test1(){
    SqlSession session = ReadFileGetconnect.getSession();
    BlogMapping mapper = session.getMapper(BlogMapping.class);
    Map<String, Object> map   = new HashMap<>();
    map.put("title","java");
    //如果 map 中没有值查询全部,有 title 查找关于 title 的值
    List<Blog> blogs = mapper.queryBlog(map);
    for (Blog blog : blogs) {
        System.out.println(blog);
    }
    session.close();
}
```

2、where 标签

(1)、说明

sql 语句后面添加 where 标签,当满足第一个条件的时候自动添加 where,当满足第二个条件的时候 and 自动转换为 where

(2)、语法

```
<where>
 代码块
</where>
```

(3)、使用

```
<select id="queryBlog" parameterType="map" resultType="Blog">
    <!--查询所有语句-->
   select * from blog
    <where>
        <!--满足第一个条件在连接的时候往前面添加 where-->
        <if test="title != null">
            title = #{title}
        </if>
        <!--满足第二个条件连接的时候自动将 and 转换为 where-->
        <if test="author != null">
            and author = #{author}
        </if>
    </where>
    <!--如果 title 有值的话拼接 if 中语句-->
</select>
```

3、choose (when, otherwise)

(1)、说明

相当于 java 的 switch 当一个满足就不执行一下个

(2)、语法

```
<choose>
        <when test="title != null">
            title = #{title}
        </when>
        <when test="author">
            and author = #{author}
        </when>
        <!--如果添加 otherwise 就必须一个满足-->
        <otherwise>
            and views = #{views}
        </otherwise>
</choose>
```

(3)、使用

```
<select id="queryBlog" parameterType="map" resultType="Blog">
    select * from blog
     <where>
          <choose>
              <when test="title != null">
                  title = #{title}
              </when>
```

```
            <when test="author">
                and author = #{author}
            </when>
            <otherwise>
                and views = #{views}
            </otherwise>
        </choose>
    </where>
</select>
```

4、set 标签

(1)、说明

和 where 差不错,set 用于 update,会在首次判断成功添加 set,自动去掉末尾逗号

(2)、语法

```
<set>
    <if test="title != null">title = #{title},</if>
    <if test="author != null">author = #{author},</if>
</set>
```

(3)、使用

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

```
    </set>
    where id = #{id};
</update>
```

4、foreach 标签

(1)、语法

```
<!--collection:循环名    item:赋值项    open:开始    separator:中间  close:结尾-->
<foreach collection="listBlog" item="id" open="(" separator="or" close=")">
    <!--判断-->
    id = #{id}
</foreach>
```

(2)、使用

```
<select id="queryBlog" parameterType="map" resultType="Blog">
    select * from blog
    <where>
        <foreach collection="listBlog" item="id" open="(" separator="or" close=")">
            id = #{id}
        </foreach>
    </where>
</select>
```

(四)、SQL 片段(抽取公共部分)

1、使用 sql 标签把公共部分写入标签内

```
<sql id="title_author">
    <if test="title != null">title = #{title},</if>
    <if test="author != null">author = #{author},</if>
</sql>
```

2、然后再使用的地方添加 include 标签

```
<include refid="title_author"></include>
```

十四、缓存

(一)、简介

1、默认情况

只启用了本地的会话缓存，它仅仅对一个会话中的数据进行缓存。 要启用全局的二级缓存，只需要在你的 SQL 映射文件中添加一行

2、缓存的方式

select 语句的结果将会被缓存。

insert、update 和 delete 语句会刷新缓存。

缓存会使用最近最少使用算法（LRU, Least Recently Used）算法来清除不需要的缓存。

缓存不会定时进行刷新

3、策略

可用的清除策略有：

- LRU – 最近最少使用：移除最长时间不被使用的对象。
- FIFO – 先进先出：按对象进入缓存的顺序来移除它们。
- SOFT – 软引用：基于垃圾回收器状态和软引用规则移除对象。
- WEAK – 弱引用：更积极地基于垃圾收集器状态和弱引用规则移除对象。

默认的清除策略是 LRU。

(二)、一级缓存(默认缓存)

默认开启,声明周期为每一个 SQLsession 的开启到关闭

1、测试

(1)、开启日志功能

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

(2)、测试中查询 23

```
@Test
public void test2(){
    UserMapperOne mapper = sqlSession.getMapper(UserMapperOne.class);
    User user1 = mapper.getUserById(1);
    User user2 = mapper.getUserById(1);
    User user3 = mapper.getUserById(1);
    System.out.println(user1);
    System.out.println(user2);
    System.out.println(user3);
```

```
    sqlSession.close();
}
```

(3)、结果

```
Find JAR URL: file:/E:/MybatisObject/MybatisObject-1/target/classes/com/ma/pojo/Blog.class
Not a JAR: file:/E:/MybatisObject/MybatisObject-1/target/classes/com/ma/pojo/Blog.class
Reader entry: ????7q
Find JAR URL: file:/E:/MybatisObject/MybatisObject-1/target/classes/com/ma/pojo/User.class
Not a JAR: file:/E:/MybatisObject/MybatisObject-1/target/classes/com/ma/pojo/User.class
Reader entry: ????7[ ?<  ?=  ?>?
Checking to see if class com.ma.pojo.Blog matches criteria [is assignable to Object]
Checking to see if class com.ma.pojo.User matches criteria [is assignable to Object]
PooledDataSource forcefully closed/removed all connections.
PooledDataSource forcefully closed/removed all connections.
PooledDataSource forcefully closed/removed all connections.
PooledDataSource forcefully closed/removed all connections.
Opening JDBC Connection
Created connection 216856121.
==>  Preparing: select id,name,pwd as password from user where id = ?
==> Parameters: 1(Integer)
<==    Columns: id, name, password
<==        Row: 1, 马佳盛, 12345
<==      Total: 1
User(id=1, name=马佳盛, password=12345)
User(id=1, name=马佳盛, password=12345)
User(id=1, name=马佳盛, password=12345)
Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@cecf639]
Returned connection 216856121 to pool.

Process finished with exit code 0
```

第一次

第二次

第三次

(三)、二级缓存

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。

默认情况下，只启用了本地的会话缓存，它仅仅对一个会话中的数据进行缓存。 要启用全局的二级缓存，只需要在你的 SQL 映射文件中添加一行：

```
<cache/>
```

基本上就是这样。这个简单语句的效果如下:

- 映射语句文件中的所有 select 语句的结果将会被缓存。
- 映射语句文件中的所有 insert、update 和 delete 语句会刷新缓存。
- 缓存会使用最近最少使用算法（LRU, Least Recently Used）算法来清除不需要的缓存。
- 缓存不会定时进行刷新（也就是说，没有刷新间隔）。
- 缓存会保存列表或对象（无论查询方法返回哪种）的 1024 个引用。
- 缓存会被视为读/写缓存，这意味着获取到的对象并不是共享的，可以安全地被调用者修改，而不干扰其他调用者或线程所做的潜在修改。

提示 缓存只作用于 cache 标签所在的映射文件中的语句。如果你混合使用 Java API 和 XML 映射文件，在共用接口中的语句将不会被默认缓存。你需要使用 @CacheNamespaceRef 注解指定缓存作用域。

这些属性可以通过 cache 元素的属性来修改。比如：

```
<cache
  eviction="FIFO"
  flushInterval="60000"
  size="512"
  readOnly="true"/>
```

这个更高级的配置创建了一个 FIFO 缓存，每隔 60 秒刷新，最多可以存储结果对象或列表的 512 个引用，而且返回的对象被认为是只读的，因此对它们进行修改可能会在不同线程中的调用者产生冲突。

可用的清除策略有：

- `LRU` – 最近最少使用：移除最长时间不被使用的对象。
- `FIFO` – 先进先出：按对象进入缓存的顺序来移除它们。
- `SOFT` – 软引用：基于垃圾回收器状态和软引用规则移除对象。
- `WEAK` – 弱引用：更积极地基于垃圾收集器状态和弱引用规则移除对象。

默认的清除策略是 LRU。

1、说明

二级缓存也叫全局缓存,一级缓存作用太低了,所以诞生了二级缓存,基于 namespace 级别的缓存,一个名称空间,对应一个二级缓存

2、工作机制

(1)、一个会话查询一条数据,这个数据就会被放在当前会话的一级缓存中;

(2)、如果会话关闭,一级缓存流没了,但是我们想要的是,会话关闭了,一级缓存中的数据就会保存到二级缓存中;

(3)、新的会话查询信息,就可以从二级缓存中获取内容;

(4)、不同的 mapper 查出的数据会放在自己对应的缓存中;

3、开启步骤

(1)、在 Mybatis 配置文件显示开启全局缓存

```
<settings>
    <setting name="cacheEnabled" value="true"/>
</settings>
```

(2)、Mapper 中开启二级缓存

```
<mapper namespace="com.ma.dao.UserMapping">


    <!--开启二级缓存-->
    <cache/> //如果不设置 eviction 的话 pojo 类需要序列化,实现 Serializable 接口
</mapper>
```

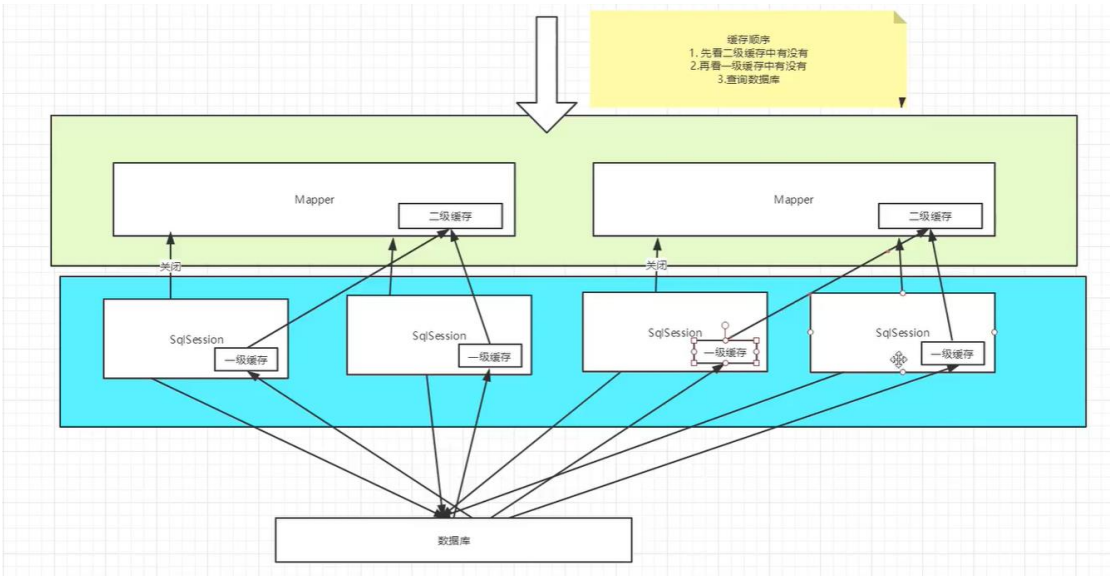
也可以自定义一些参数

```
<!--开启二级缓存-->
<cache eviction="FIFO" //先进先出,如果不设置的话 pojo 类需要序列化,实现 Serializable 接口
        flushInterval="60000" //60 秒刷新
        size="512" //最多 512 个引用
        readOnly="true"/> //只读开启
```

4、测试

```
@Test
public void test2(){
    //session1
    SqlSession sqlSession1 = ReadFileGetconnect.getSession();
    UserMapperOne mapper1 = sqlSession1.getMapper(UserMapperOne.class);
    User user1 = mapper1.getUserById(1);
    System.out.println(user1);
    //session1 关闭,将缓存存入二级缓存
    sqlSession1.close();

    //session2
    SqlSession sqlSession2 = ReadFileGetconnect.getSession();
    UserMapperOne mapper2 = sqlSession2.getMapper(UserMapperOne.class);
    //直接去二级缓存取数据
    User user2 = mapper2.getUserById(1);
    System.out.println(user2);
    sqlSession2.close();
}
```

(四)、缓存的流程


缓存顺序
1.先看二级缓存中有没有
2.再看一级缓存中有没有
3.查询数据库
Mapper
二级缓存
Mapper
二级缓存
关闭
关闭
SqlSession
一级缓存
SqlSession
一级缓存
SqlSession
一级缓存
SqlSession
一级缓存
数据库