一、MyBatis

1、MyBatis简介

1.1、MyBatis历史

MyBatis最初是Apache的一个开源项目iBatis, 2010年6月这个项目由Apache Software Foundation迁移到了Google Code。随着开发团队转投Google Code旗下， iBatis3.x正式更名为MyBatis。代码于2013年11月迁移到Github。

iBatis一词来源于“internet”和“abatis”的组合，是一个基于Java的持久层框架。 iBatis提供的持久层框架包括SQL Maps和Data Access Objects（DAO）。

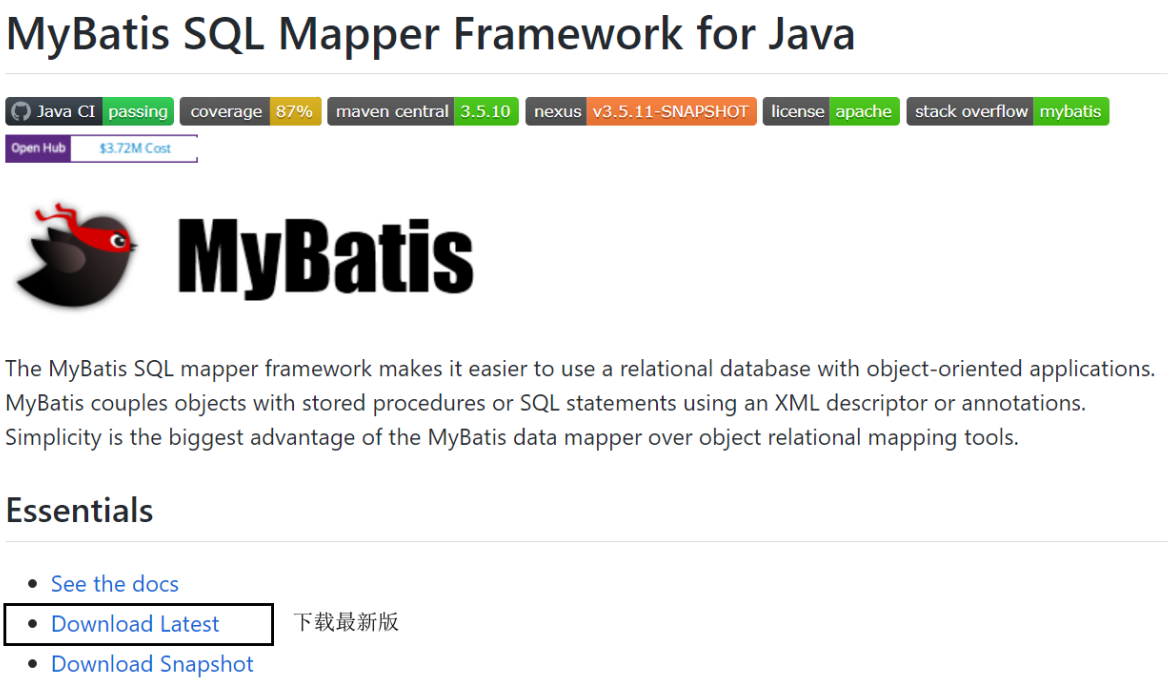
1.2、MyBatis特性

1. MyBatis 是支持定制化 SQL、存储过程以及高级映射的优秀的持久层框架
2. MyBatis 避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集
3. MyBatis可以使用简单的XML或注解用于配置和原始映射，将接口和Java POJO（Plain Old Java Objects，普通的Java对象）映射成数据库中的记录
4. MyBatis 是一个 半自动的ORM（Object Relation Mapping）框架

1.3、MyBatis下载

MyBatis下载地址：https://github.com/mybatis/mybatis-3

> 一般是直接使用Maven下载。



下载最新版

1.4、和其它持久化层技术对比

- JDBC
  - SQL夹杂在Java代码中耦合度高，导致硬编码内伤
  - 维护不易且实际开发需求中SQL有变化，频繁修改的情况多见
  - 代码冗长，开发效率低
- Hibernate和JPA
  - 操作简便，开发效率高
  - 程序中的长难复杂SQL需要绕过框架
  - 内部自动生产的SQL，不容易做特殊优化
  - 基于全映射的全自动框架，大量字段的POJO进行部分映射时比较困难
  - 反射操作太多，导致数据库性能下降
- MyBatis
  - 轻量级，性能出色
  - SQL和Java编码分开，功能边界清晰。Java代码专注业务、SQL语句专注数据
  - 开发效率稍逊于HIbernate，但是完全能够接受

2、搭建MyBatis

2.1、开发环境

IDE：idea 2022.1

构建工具：maven 3.6.3

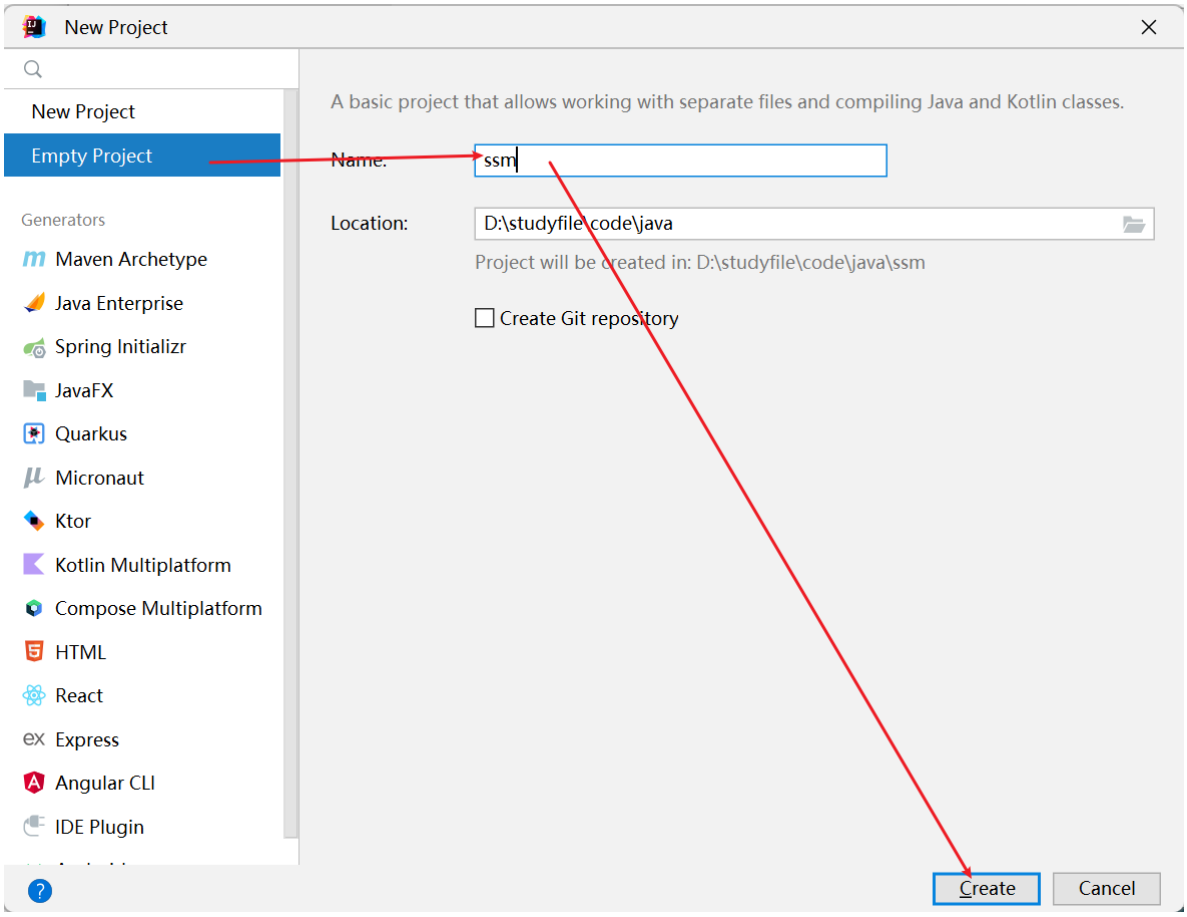
MySQL版本：MySQL 8

MyBatis版本：MyBatis 3.5.7

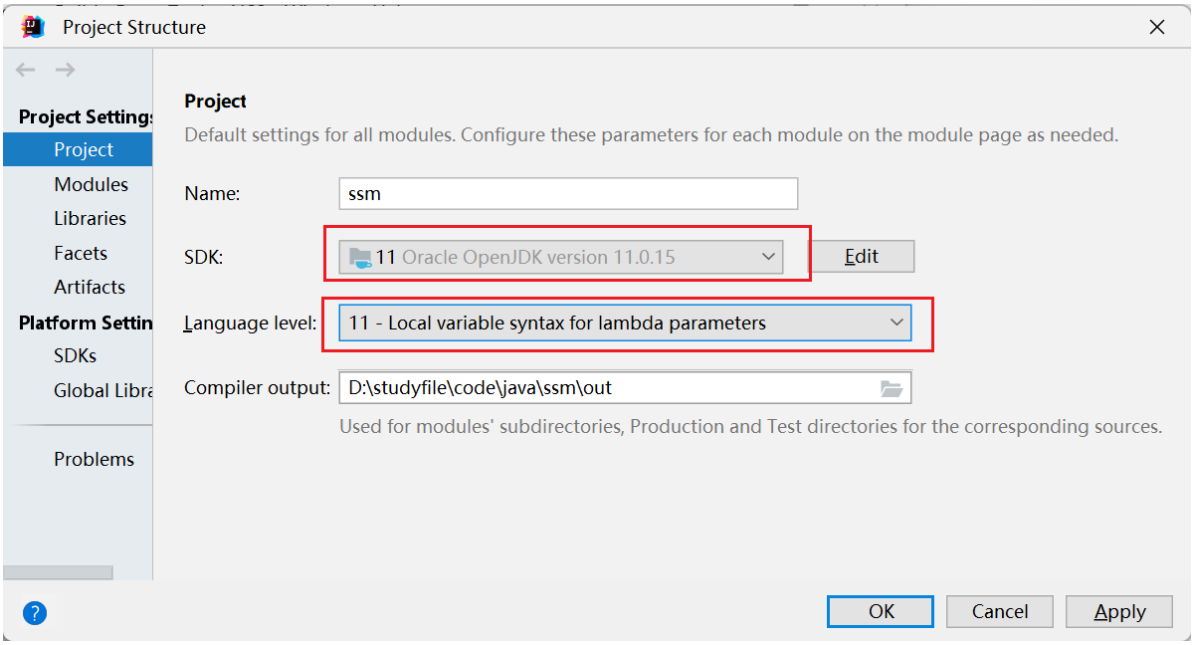
> MySQL不同版本的注意事项：
>
> 1. 驱动类driver-class-name
>
>   MySQL 5版本使用jdbc5驱动，驱动类使用：`com.mysql.jdbc.Driver`
>
>   MySQL 8版本使用jdbc8驱动，驱动类使用：`com.mysql.cj.jdbc.Driver`
>
> 2. 连接url
>
>   MySQL 5版本的url：
>
>   `jdbc:mysql://localhost:3306/ssm`
>
>   MySQL 8版本的url，需要添加时区：
>
>   `jdbc:mysql://localhost:3306/ssm?serverTimezone=UTC`

2.2、创建Maven工程

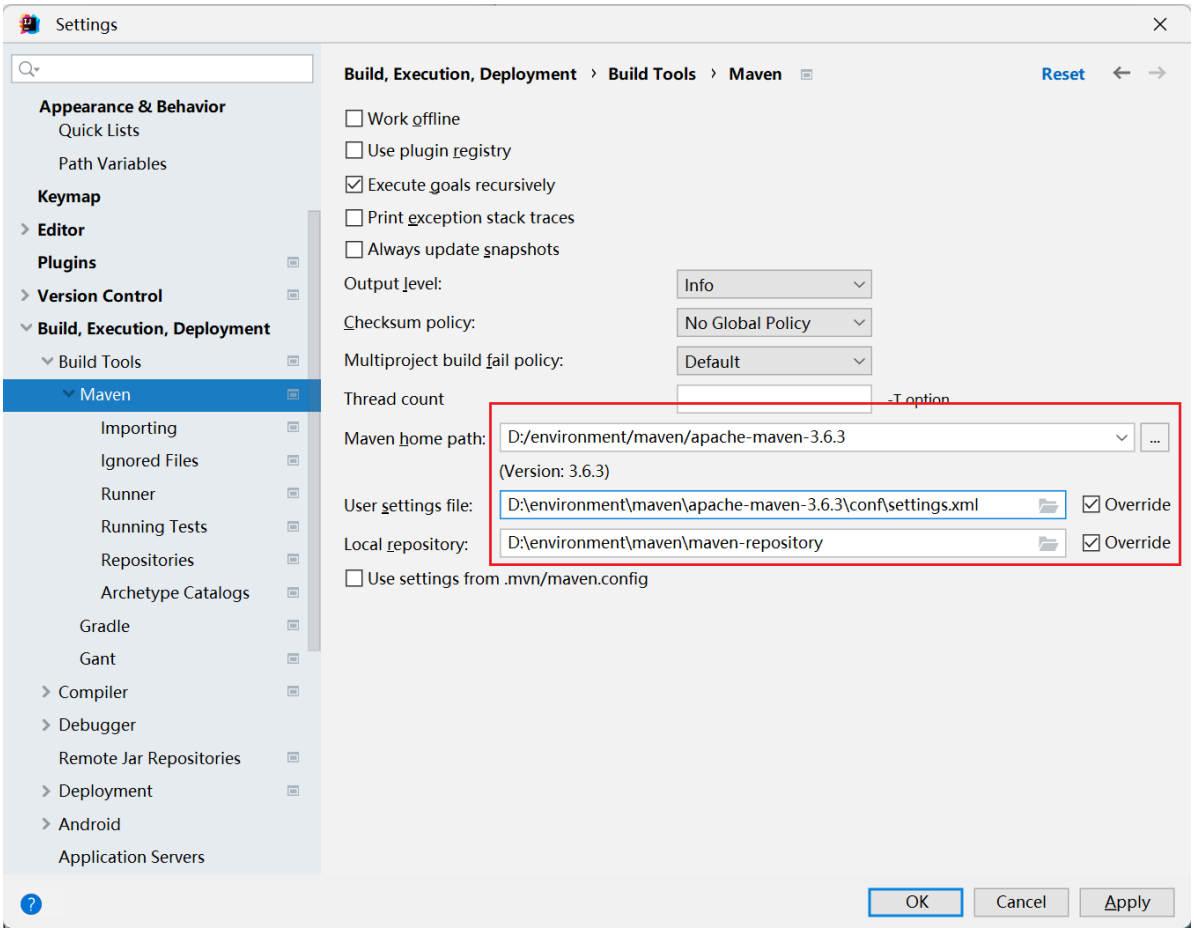
1、首先创建一个maven工程（这里因为是在学习SSM，因此先创建一个空的Java项目，命名为ssm，后续在创建Module），如下图所示。



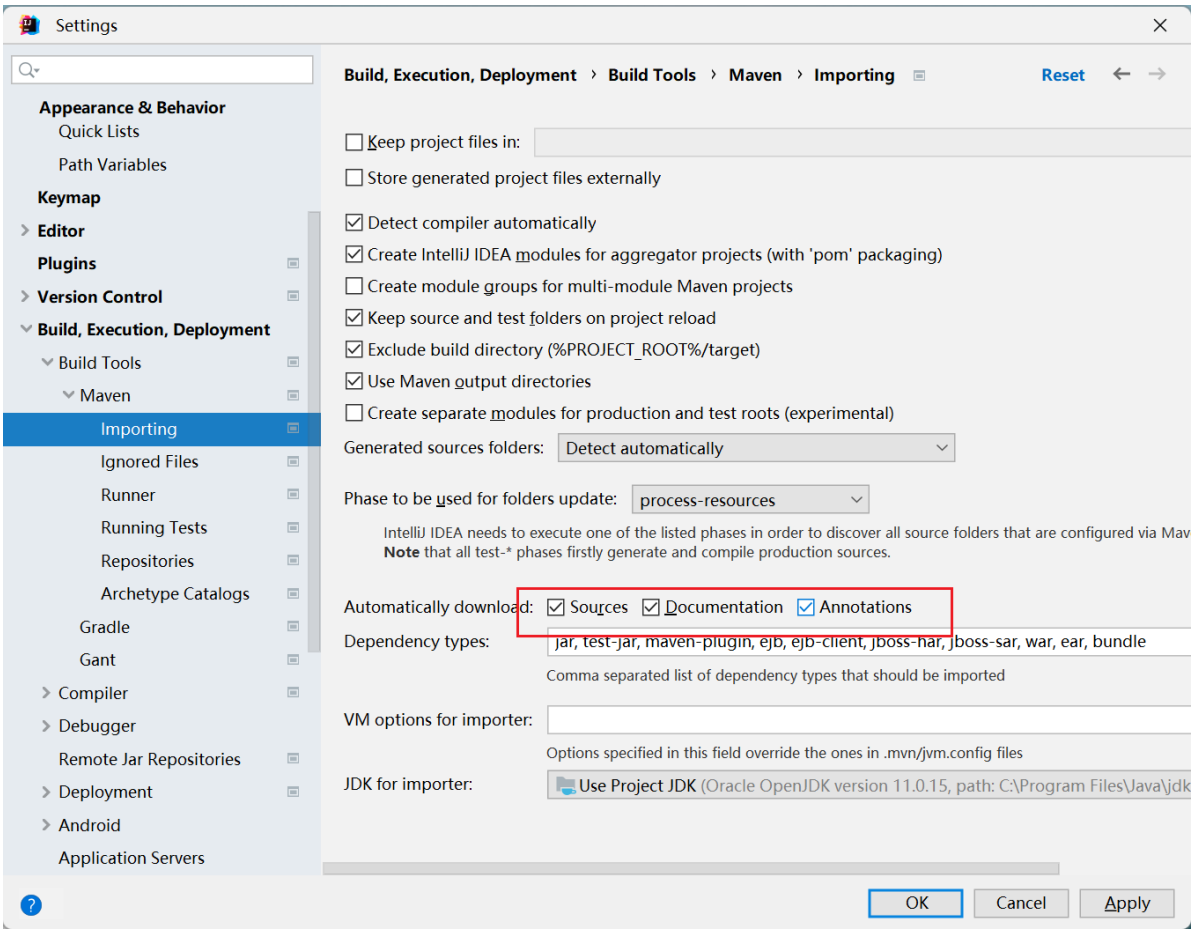
2、配置项目的结构，选择'File'→'Project Structure'，选择'SDK'和'Language level'为所安装的版本，这里选择版本11。



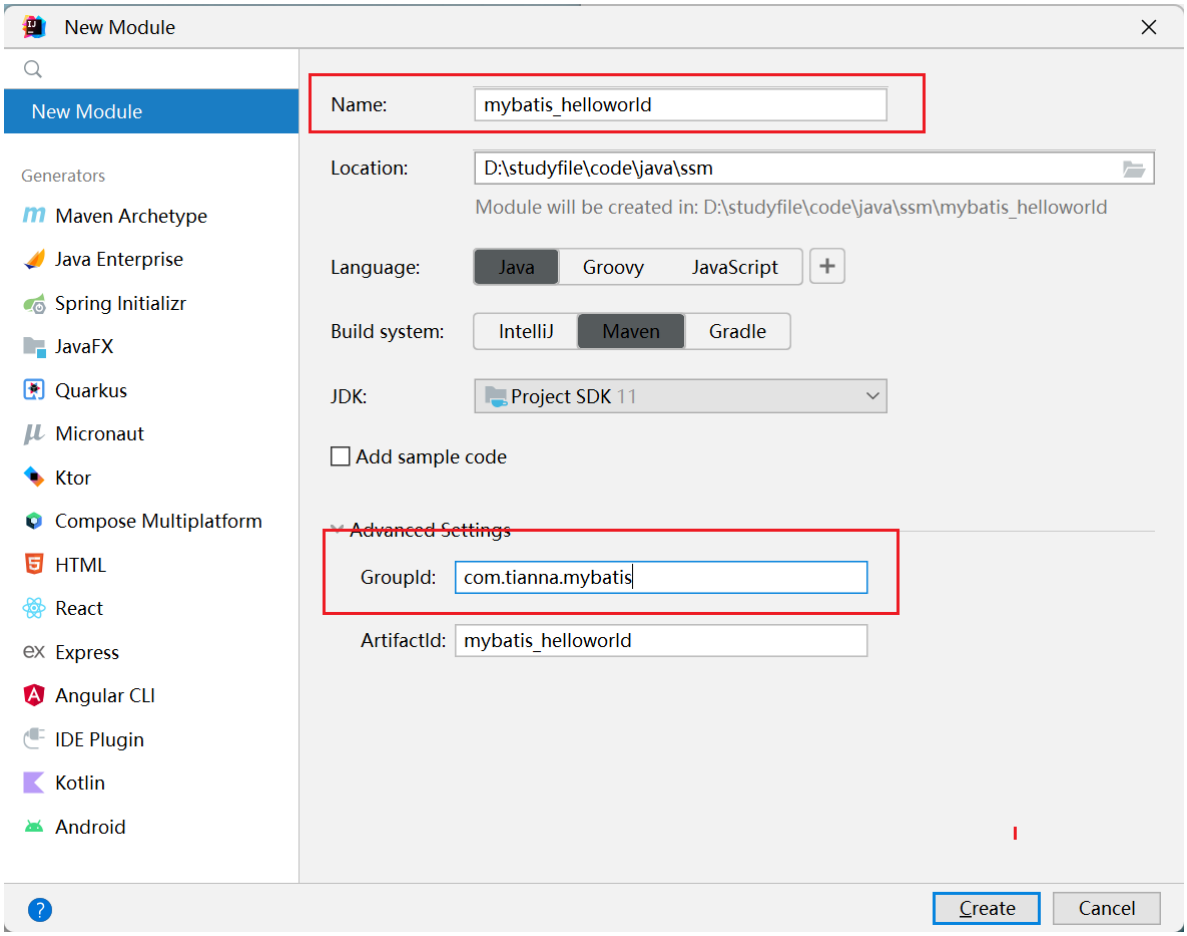
3、设置Maven，打开'File'→'Settings'→'Build,Execution,Deployment'→'Build Tools'→'Maven'，选择使用本地的Maven。



然后选择'Maven'下的'Importing'中的'Automatically download'，勾选所有的可选框，即自动下载Maven包。



4、创建一个Maven Module，项目名称→'New'→'Module'，设置Module名称和GroupId。



5、在pom.xml中设置项目的打包方式为jar包

```
<packaging>jar</packaging>
```

并引入以下相关依赖：

- Mybatis核心
- junit测试
- MySQL驱动

```
<dependencies>
    <!--MyBatis核心-->
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.5.7</version>
    </dependency>
    <!--junit测试-->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>
    <!--MySQL驱动-->
    <dependency>
        <groupId>mysql</groupId>
```

```
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.29</version>
    </dependency>
</dependencies>
```

2.3、在数据库中创建表并创建一个Java实体类

1、创建一个名为`ssm`的数据库：

```
create database ssm;
```

2、在`ssm`数据库中创建一个名为`t_user`的数据表：

```
CREATE TABLE IF NOT EXISTS `t_user`(
    `id` int NOT NULL AUTO_INCREMENT,
    `username` varchar(20),
    `password` varchar(20),
    `age` int,
    `gender` char,
    `email` varchar(50),
    PRIMARY KEY(`id`)
)ENGINE=INNODB DEFAULT CHARSET=utf8;
```

3、在文件夹src/main/java下创建一个名为com.tianna.mybatis.pojo的包，并在改包下创建一个名为`User`的实体类，并为其创建有参，无参，set，get，toString方法。

```
package com.tianna.mybatis.pojo;

/**
 * @author tiancn
 * @date 2022/7/22 22:12
 */
public class User {
    private Integer id;
    private String username;
    private String password;
    private Integer age;
    private String gender;
    private String email;

    public User() {
    }

    public User(Integer id, String username, String password, Integer age,
String gender, String email) {
        this.id = id;
        this.username = username;
        this.password = password;
        this.age = age;
        this.gender = gender;
```

```
        this.email = email;
    }

    public Integer getId() {
        return id;
    }

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

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }

    public String getGender() {
        return gender;
    }

    public void setGender(String gender) {
        this.gender = gender;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
```

```
    @Override
    public String toString() {
        return "User{" +
                "id=" + id +
                ", username='" + username + '\'' +
                ", password='" + password + '\'' +
                ", age=" + age +
                ", gender='" + gender + '\'' +
                ", email='" + email + '\'' +
                '}';
    }
}
```

2.4、创建MyBatis的核心配置文件

在src/main/resources目录下创建一个名为mybatis-config.xml的核心配置文件，核心配置文件主要用于配置连接数据库的环境以及MyBatis的全局配置信息。

> 该配置文件命名为mybatis-config.xml只是建议，并不是强制要求。

其一些简单的配置如下：

```
<?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>
    <!--设置连接数据库的环境-->
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.cj.jdbc.Driver"/>
                <property name="url"
value="jdbc:mysql://localhost:3306/ssm?
serverTimezone=UTC"/>
                <property name="username" value="root"/>
                <property name="password" value="123456"/>
            </dataSource>
        </environment>
    </environments>
    <!--引入MyBatis映射文件-->
    <mappers>
        <mapper resource=""/>
    </mappers>
</configuration>
```

上边只包含了最关键的部分，XML头部的声明，它用来验证XML文档的正确性。environment元素体中包含了事务管理和连接池的配置。mappers元素则包含了一组映射器（mapper），这些映射器的 XML 映射文件包含了SQL代码和映射定义信息。

根据MySQL的版本和相关配置设置了driver、url、username、password的相关值。

2.5、创建mapper接口

MyBatis中的mapper接口相当于以前的dao。区别在于，mapper仅仅是接口，不需要提供实现类。

其命名规则为，操作的类名+Mapper，如操作User类的mapper接口名为`UserMapper`。创建一个名为com.tianna.mybatis.mapper的包，在其下创建mappe接口。

```
package com.tianna.mybatis.mapper;

/**
 * @author tiancn
 * @date 2022/7/22 23:13
 */
public interface UserMapper {
    /**
     *添加用户信息
     */
    int insertUser();
}
```

2.6、创建MyBatis映射文件

相关概念：**ORM**（**O**bject **R**elationship **M**apping）对象关系映射。

- 对象：Java的实体类对象
- 关系：关系型数据库
- 映射：二者之间的对应关系

Java概念	数据库概念
类	表
属性	字段/列
对象	记录/行

Mybatis映射文件需要注意的地方：

1. MyBatis映射文件用于编写SQL，访问以及操作表中的数据。
2. 每一个映射文件对应一个实体类，对应一张数据表。其命名规则为：表所对应的实体类+Mapper.xml，例如实体类为User，其所对应的映射文件为`UserMapper.xml`。
3. MyBatis映射文件存放的位置为src/main/resources/mappers目录下。
4. Mybatis中使用映射文件使得可以面向接口操作数据，即只需要创建mapper接口，而不需要其实现，但是mapper接口要和映射文件保证两个一致：
    - mapper接口的全类名和映射文件的命名空间(namespace)保持一致。
    - mapper接口中方法的方法名和映射文件中编写的SQL的标签的id属性保持一致。

具体示例：

在src/main/resources目录下创建mappers目录，然后在该目录下创建名为`UserMapper.xml`的映射文件。以实现一个简单的插入SQL为例，**要注意保证两个一致。**`UserMapper.xml`的内容如下。

```
<?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="com.tianna.mybatis.mapper.UserMapper">

    <!--
        mapper接口要和映射文件保证两个一致：
        1、mapper接口的全类名和映射文件的命名空间(namespace)保持一致。
        2、mapper接口中方法的方法名和映射文件中编写的SQL的标签的id属性保持一
致。
    -->
    <!--int insertUser();-->
    <insert id="insertUser">
        insert into t_user values
(null,'admin','123456',23,'男','12345@qq.com')
    </insert>

</mapper>
```

> 还有一点不能忘记，需要在核心配置文件中引入mybatis的映射文件，即核心配置文件中`mappers`标签下需要改为以下内容：
>
> ```
> <!--引入MyBatis映射文件-->
>  <mappers>
>      <mapper resource="mappers/UserMapper.xml"/>
>  </mappers>
> ```

2.7、通过junit测试功能

在test/java目录下创建名为com.tianna.mybatis.test的包，并在该包下创建一个名为`MyBatisTest`的类，用于对功能进行测试。

```
/**
 * @author tiancn
 * @date 2022/7/23 22:33
 */
public class MyBatisTest {
    @Test
    public void testInsert() throws IOException {
        //获取MyBatis核心配置文件的输入流
        InputStream is = Resources.getResourceAsStream("mybatis-
config.xml");
        //创建SqlSessionFactoryBuilder对象
```

```
        SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new
SqlSessionFactoryBuilder();
        //获取SqlSessionFactory对象
        SqlSessionFactory sqlSessionFactory =
sqlSessionFactoryBuilder.build(is);
        //获取sql的会话对象SqlSession，是MyBatis提供的操作数据库的对象
        //创建SqlSession对象，此时通过SqlSession对象所操作的sql都必须手动提交
或回滚事务
        //SqlSession sqlSession = sqlSessionFactory.openSession();
        //创建SqlSession对象，此时通过SqlSession对象所操作的sql都会自动提交
        SqlSession sqlSession = sqlSessionFactory.openSession(true);
        //UserMapper是一个接口，不可以实例化，因此使用SqlSession对象获取
UserMapper的代理实现类对象
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
        //调用mapper接口中的方法，实现添加用户信息的功能
        int result = mapper.insertUser();
        System.out.println("结果："+result);
        //手动提交事务
        //sqlSession.commit();
        //关闭sqlSession
        sqlSession.close();
    }
}
```

输出为：

```
结果：1
```

在数据库中可以看到插入的结果，插入数据执行成功。

id	username	password	age	gender	email
1	admin	123456	23	男	12345@qq.

- SqlSession：代表Java程序和数据库之间的会话。（HttpSession是Java程序和浏览器之间的会话）
- SqlSessionFactory：是“生产”SqlSession的“工厂”。
- 工厂模式：如果创建某一个对象，使用的过程基本固定，那么我们就可以把创建这个对象的相关代码封装到一个“工厂类”中，以后都使用这个工厂类来“生产”我们需要的对象。

2.8、加入log4j日志功能

1、在pom.xml文件中加入依赖。

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

2、加入log4j的配置文件

log4j配置文件的命名必须时`log4j.xml`，该文件存放的位置为src/main/resources下。

```
<?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>
```

> **日志的级别**
>
> FATAL(致命)>ERROR(错误)>WARN(警告)>INFO(信息)>DEBUG(调试)
>
> 从左到右打印的内容越来越详细，即设置级别为INFO，其左边所有的内容都会被打印出来。

再次运行测试方法，其运行结果如下所示，可见运行的详细信息都被打印出来了：

```
DEBUG 07-24 15:48:35,810 ==>  Preparing: insert into t_user values
(null,'admin','123456',23,'男','12345@qq.com') (BaseJdbcLogger.java:137)
DEBUG 07-24 15:48:35,830 ==> Parameters:  (BaseJdbcLogger.java:137)
DEBUG 07-24 15:48:35,834 <==    Updates: 1 (BaseJdbcLogger.java:137)
结果：1
```

3、MyBatis增删改查

在测试MyBatis功能时，需要创建SqlSession对象，这一过程比较麻烦，每一个测试方法中都需要写，因此创建一个工具类来获取SqlSession对象。

在src/main/java目录下创建一个名为com.tianna.mybatis.utils的包，然后在该包下创建一个名为`SqlSessionUtil`的类，用于创建SqlSession对象，该类下的内容如下：

```
/**
 * @author tiancn
 * @date 2022/7/24 16:39
```

```
 */
public class SqlSessionUtil {
    public static SqlSession getSqlSession(){
        SqlSession sqlSession =null;
        try {
            //获取核心配置文件的输入流
            InputStream is = Resources.getResourceAsStream("mybatis-
config.xml");
            //获取SqlSessionFactoryBuilder
            SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new
SqlSessionFactoryBuilder();
            //获取SqlSessionFactory
            SqlSessionFactory sqlSessionFactory =
sqlSessionFactoryBuilder.build(is);
            //获取SqlSession对象
            sqlSession = sqlSessionFactory.openSession(true);
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        return sqlSession;
    }
}
```

然后测试添加功能的测试方法就可以写成如下形式。

```
@Test
public void testInsert_new(){
    //创建SqlSession对象
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    //使用SqlSession对象获取UserMapper的代理实现类对象
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);
    //调用mapper接口中的方法，实现添加用户信息的功能
    mapper.insertUser();
    //关闭sqlSession
    sqlSession.close();
}
```

3.1、新增

新增功能在前面已经实现过，Mapper接口添加的代码为：

```
int insertUser();
```

MyBatis映射文件添加的代码为：

```
<!--int insertUser();-->
<insert id="insertUser">
    insert into t_user values
(null,'admin','123456',23,'男','12345@qq.com')
</insert>
```

测试过程参考上面介绍的步骤。

3.2、修改

下面进行一个简单的修改，将固定id的`username`及其`password`进行修改。

Mapper接口添加的代码为：

```
void updateUser();
```

MyBatis映射文件添加的代码为：

```
<!--int updateUser()-->
<update id="updateUser">
    update t_user set username = 'root',password = '123' where id = 3
</update>
```

测试代码为：

```
@Test
public void testUpdate(){
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);
    mapper.updateUser();
    sqlSession.close();
}
```

3.3、删除

将固定id的数据删除。

Mapper接口添加的代码为：

```
void deleteUser();
```

MyBatis映射文件添加的代码为：

```
<!--void deleteUser();-->
<delete id="deleteUser">
    delete from t_user where id = 3
</delete>
```

测试代码为：

```
@Test
public void testDelete(){
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);
    mapper.deleteUser();
    sqlSession.close();
}
```

3.4、查询一个实体类对象

由于还没有学到传参，因此所查询的id时固定的，Mapper接口添加的代码为：

```
User getUserById();
```

MyBatis映射文件添加的代码为：

```
<!--User getUserById();-->
<select id="getUserById" resultType="com.tianna.mybatis.pojo.User">
    select * from t_user where id = 1
</select>
```

> 查询的select必须设置属性resultType或resultMap，用于设置实体类和数据库表的映射关系：
>
> resultType：自动映射，用于属性名和表中字段名一致的情况
>
> resultMap：自定义映射，用于一对多或多对一或字段名和属性名不一致的情况

测试代码为：

```
@Test
public void testGetUserById(){
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);
    User user = mapper.getUserById();
    System.out.println(user);
    sqlSession.close();
}
```

3.5、查询list集合

查询该表中所有的数据，即返回类型应该是一个list集合。

Mapper接口添加的代码为：

```
List<User> getAllUser();
```

MyBatis映射文件添加的代码为：

```
<!--List<User> getAllUser();-->
<select id="getAllUser" resultType="com.tianna.mybatis.pojo.User">
    select * from t_user
</select>
```

其中resultType设置的类型还是`User`。

测试代码为：

```
@Test
public void testGetAllUser(){
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);
    List<User> list = mapper.getAllUser();
    list.forEach(System.out::println);
}
```

4、核心配置文件详解

MyBatis核心配置文件中的标签必须要按照指定的顺序配置：

- 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)

下面介绍几种常用的标签。

4.1、environments(环境配置)

MyBatis 可以配置成适应多种环境，这种机制有助于将SQL映射应用于多种数据库之中。下面为`environments`的详细介绍。

```
<!--设置连接数据库的环境-->
<!--
    environments：配置多个连接数据库的环境
    属性：
```

```
        default：设置默认使用的环境id
-->
<environments default="development">
    <!--
        environment：配置某个具体的环境
        属性：
            id：表示连接数据库的环境的唯一标识，不能重复
    -->
    <environment id="development">
        <!--
            transactionManager：设置事务管理模式
            属性：
                type=“JDBC|MANAGED”
                JDBC：表示当前环境中，执行SQL时，使用的是JDBC中原生的事务管理方式，事务的提交或回滚需要手动处理。
                MANAGED：被管理，例如Spring
        -->
        <transactionManager type="JDBC"/>
        <!--
            dataSource：配置数据源
            属性：
                type：设置数据源的类型
                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="com.mysql.cj.jdbc.Driver"/>
            <!--设置连接数据库的连接地址-->
            <property name="url" value="jdbc:mysql://localhost:3306/ssm?serverTimezone=UTC"/>
            <!--设置连接数据库的用户名-->
            <property name="username" value="root"/>
            <!--设置连接数据库的密码-->
            <property name="password" value="123456"/>
        </dataSource>
    </environment>
</environments>
```

4.2、properties(属性)

环境配置中的数据库连接驱动、数据库连接地址、用户名和密码等信息可以在外部进行配置，并可以进行动态替换。你既可以在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。例如：

```
<properties resource="org/mybatis/example/config.properties">
  <property name="username" value="root"/>
  <property name="password" value="123456"/>
</properties>
```

下面介绍如何在Java属性文件中配置这些属性并使用。

1、首先在src/main/resources目录下创建一个名为`jdbc.properties`的属性文件（名字不唯一）。然后在属性文件中输入属性及其值的键值对，键和值中间用等号连接。

```
jdbc.driver = com.mysql.cj.jdbc.Driver
jdbc.url = jdbc:mysql://localhost:3306/ssm?serverTimezone=UTC
jdbc.username = root
jdbc.password = 123456
```

2、在核心配置文件中引入`jdbc.properties`，在核心配置文件中添加以下内容（注意标签的顺序）。

```
<!--引入properties文件，此后就可以在当前文件中使用${key}的方式访问value-->
<properties resource="jdbc.properties"></properties>
```

3、在environment标签中使用${key}的方式获取value。

```
<environment id="development">
    <transactionManager type="JDBC"/>
    <dataSource type="POOLED">
        <property name="driver" value="${jdbc.driver}"/>
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
    </dataSource>
</environment>
```

4.3、typeAliases(类型别名)

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。

有三种设置类型别名的方式。

1. 给一个Java类型设置一个任意的别名，为`type`中的类型起一个`alias`对应的别名。

```
<!--
    typeAliases：设置类型别名，即为某个具体的类型设置一个别名
    在MyBatis的范围中，就可以使用别名表示一个具体的类型
-->
<typeAliases>
    <!--
        type：设置需要起别名的类型
        alias：别名的名称
    -->
    <typeAlias type="com.tianna.mybatis.pojo.User" alias="abc">
</typeAlias>
</typeAliases>
```

然后在MyBatis配置文件中就可以使用`alias`对应的别名，而不需要使用全类名。

```
<!--List<User> getAllUser();-->
<select id="getAllUser" resultType="abc">
    select * from t_user
</select>
```

2. 其中`alias`参数可以不设置值，那么别名则为类名,并且不区分大小写。

```
<typeAliases>
    <!--若不设置alias，当前类型的别名为类名即User，并且不区分大小写，即也可以写为user-->
    <typeAlias type="com.tianna.mybatis.pojo.User"></typeAlias>
</typeAliases>
```

3. 当需要起别名的Java类型过多时，以上两种方式写起来就比较麻烦，可以指定一个包名，MyBatis会在包下所有的类型起一个默认别名，即类名且不区分大小写。

> 一般使用该方式，比较简便。

```
<typeAliases>
    <!--通过包名设置类型别名，指定包下所有的类型将全部拥有默认的别民，即类名且不区分大小写-->
    <package name="com.tianna.mybatis.pojo"/>
</typeAliases>
```

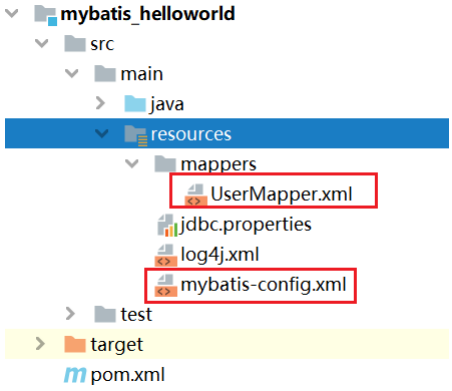
4.4、mappers(映射器)

mappers(映射器)的作用为告诉MyBatis到哪里去找映射文件，其也有两种方式。

1. 使用映射文件的路径，在这种方式下需要为每一个映射文件写一条`mapper`，比较麻烦。

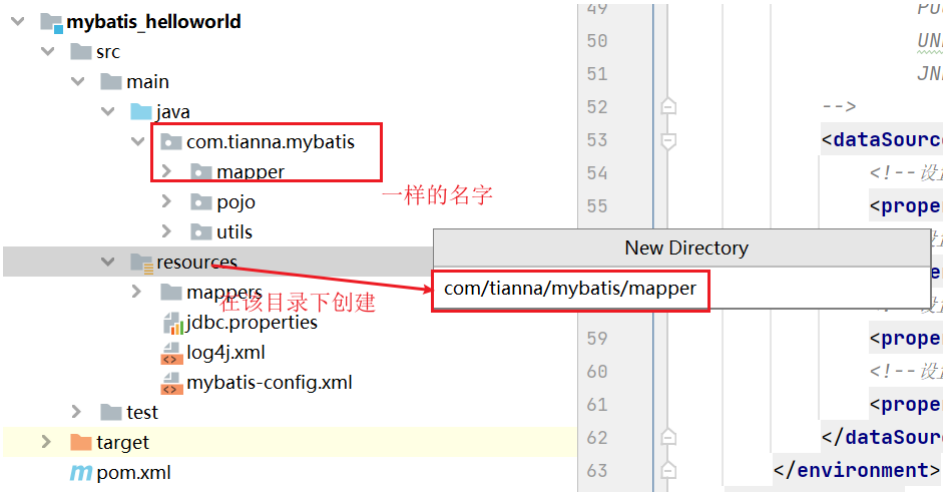
```
<!--引入MyBatis映射文件-->
<mappers>
    <mapper resource="mappers/UserMapper.xml"/>
</mappers>
```

其中MyBatis映射文件`UserMapper.xml`的位置与核心配置文件的关系为：



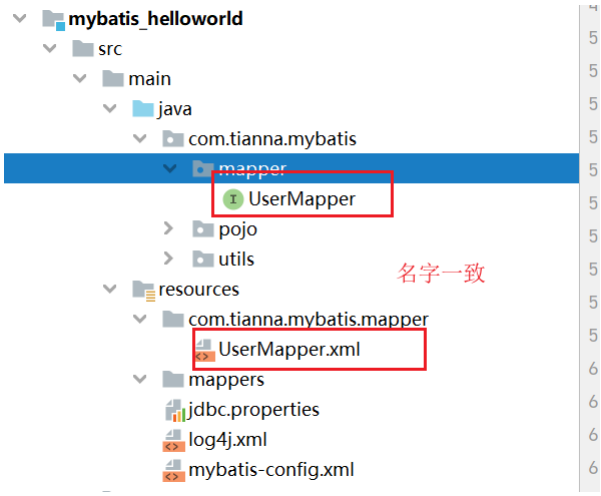
2. 上面一种方式需要为每一个映射文件写一条语句，因此比较麻烦，因此还可以以包为单位引入映射文件，即引入一个包下所有的映射文件，但是又两个要求：

    1. mapper接口所在的包要和映射文件所在的包一致，这就需要在src/main/resources目录下创建一个与mapper接口所在包一样的包。



    然后将`UserMapper.xml`映射文件移动到该包下即可。

    2. mapper接口要和映射文件的名字一致。



在满足以上两个条件时，就可以以包为单位引入映射文件，

```
<mappers>
    <!--
        以包为单位引入映射文件
        要求：
        1、mapper接口所在的包要和映射文件所在的包一致
        2、mapper接口要和映射文件的名字一致
    -->
    <package name="com.tianna.mybatis.mapper"/>
</mappers>
```

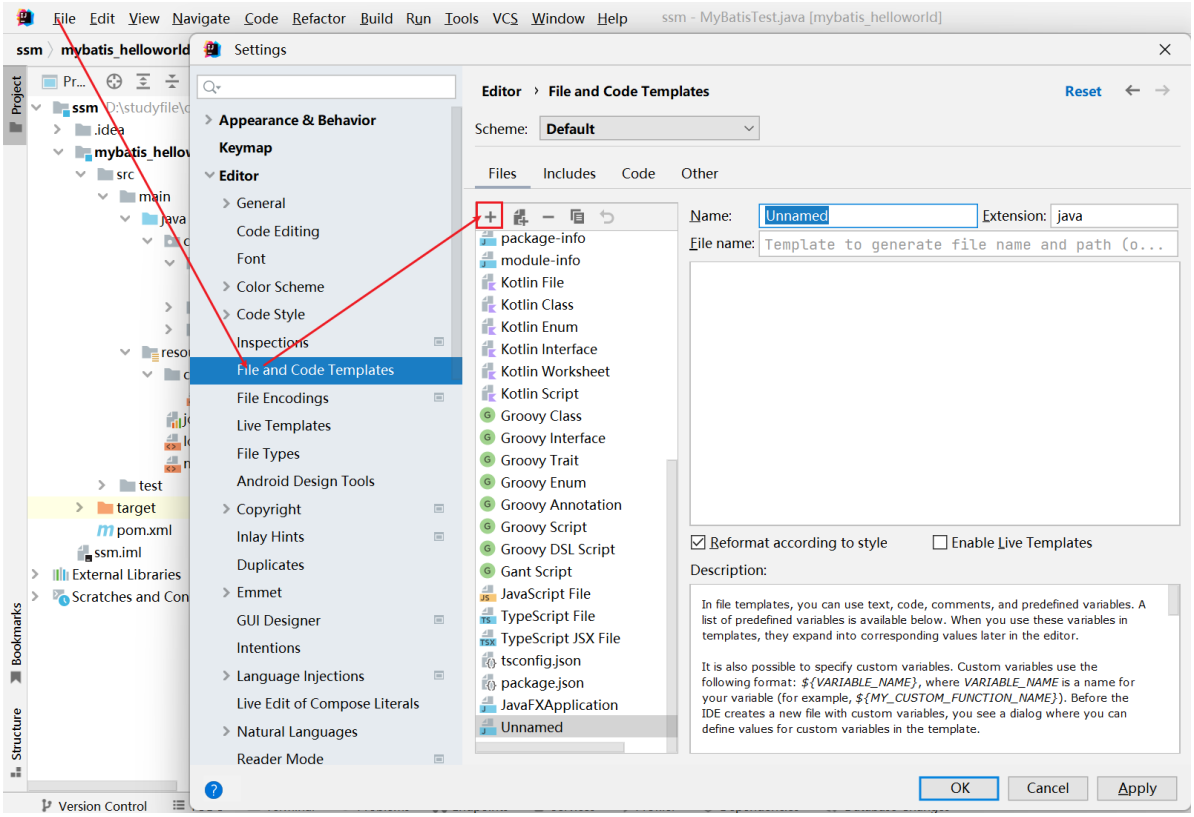
> 一般使用以包为单位引入的方法，比较简便。

还有一些其他的标签在后面使用的时候再介绍。可看

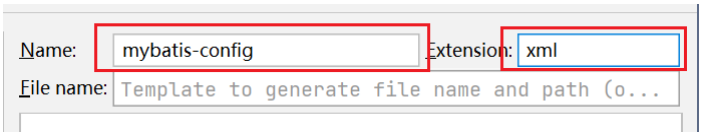
5、IDEA创建MyBatis核心配置文件和映射文件模板

5.1、添加核心配置文件模板

1、选择File→Settings→Editor→File and Code Templates，然后点击Files一栏下的➕添加一个新的模板。



2、添加核心配置文件的模板，模板名称取名为`mybatis-config`(该名称为模板名称创建时还需要起名)，模板类型为`xml`。



3、核心配置文件模板内容如下，然后应用：

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

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

    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <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MyBatis映射文件-->
    <mappers>
        <package name=""/>
    </mappers>
</configuration>
```

5.2、添加映射文件模板

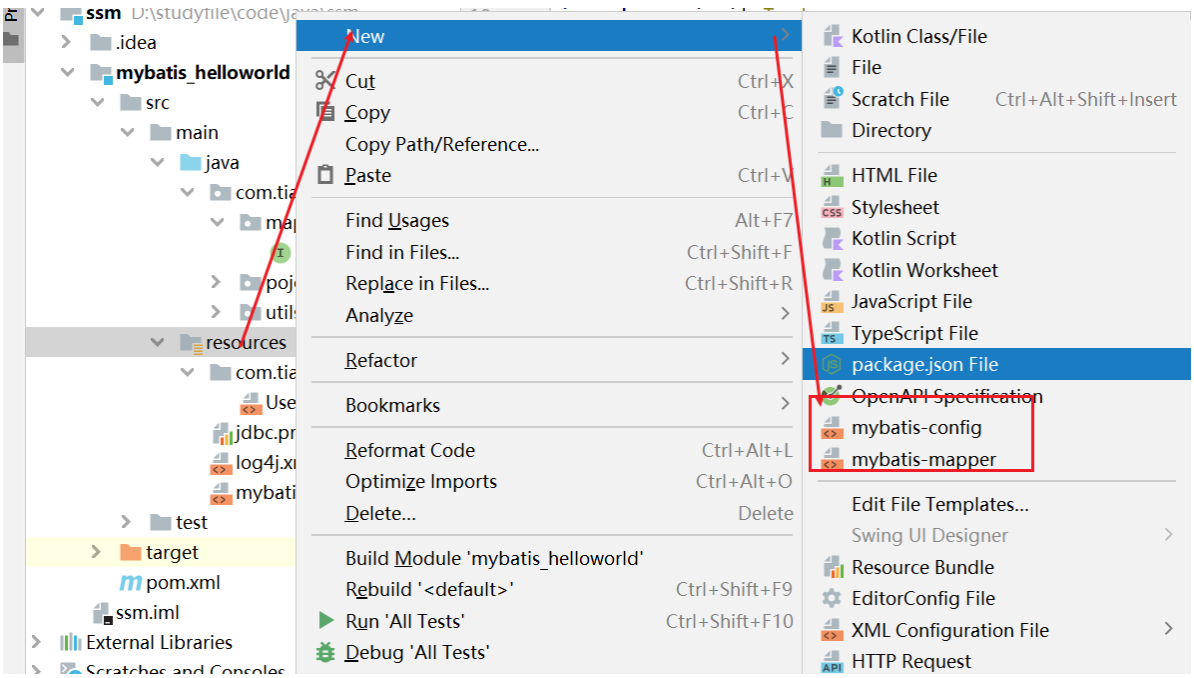
1、与上述添加核心配置文件模板相同，添加一个名为`mybatis-mapper`类型为xml的模板，其内容如下：

```
<?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="">

</mapper>
```

然后应用即可添加改模板。

5.3、如何使用

使用时，只需要在需要添加模板文件的目录下右键→New，选择刚才添加的模板文件，然后输入新的文件名，就可以使用模板文件。



6、MyBatis获取参数值的两种方式

在MyBatis中有两种获取参数值的方式：${}和#{}。

- ${}：其本质为字符串拼接，若为字符串类型或日期类型的字段进行赋值的时候，需要手动加单引号。
- #{}：其本质为占位符赋值，此时为字符串类型或者日期类型进行赋值时`会自动添加单引号`，并且可以避免SQL注入的问题。`因此一般使用该方法比较多`。

6.1、单个字面量类型的参数

先看看什么是字面量和变量？

以`int a = 1`为例，其中`a`为变量，`1`为字面量。其中，基本数据类型、字符串和包装类都属于字面量类型。

当mapper接口中的方面为单个字面量类型时，以根据用户名查询数据为例，mapper接口内容为：

```
/**
 * 根据用户名查询用户信息
 * @param username
 * @return
 */
User getUserByUsername(String username);
```

这是可以使用${}和#{}以任意名称获取参数的值，`虽然可以取任意名称，但是最好还是与参数名保持一致`，当使用${}时一定注意要加`单引号`。

MyBatis映射文件内容为：

```
<!--User getUserByUsername(String username);-->
<select id="getUserByUsername" resultType="user">
    <!--1、使用${}时需要加单引号-->
    <!--select * from t_user where username = '${username}'-->
    <!--2、使用#{}时不加单引号，一般使用该方法-->
    select * from t_user where username = #{username}
</select>
```

其测试方法为：

```
@Test
public void testGetUserByUsername(){
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);
    User user = mapper.getUserByUsername("admin");
    System.out.println(user);
    sqlSession.close();
}
```

6.2、多个字面量类型的参数

下面看看mapper接口中的方法参数为多个时，以验证登录时为例，这时的参数为username和password两个，mapper接口中的内容为：

```
/**
 * 验证登录
 * @param username
 * @param password
 * @return
 */
User checkLogin(String username,String password);
```

在MyBatis中会将这些参数放在一个map集合中，它会以两种方式存储数据：

1. 以arg0,arg1,...为键，以参数为值。
2. 以param1,param2,...为键，以参数为值。

然后只需要通过${}或#{}访问map集合的键就可以获取相对应的值。

MyBatis映射文件内容如下

```
<!--User checkLogin(String username,String password);-->
<select id="checkLogin" resultType="user">
    <!--1、使用arg0,arg1,...-->
    <!--select * from t_user where username = #{arg0} and password = #
{arg1}-->
    <!--2、使用param1,param2,...-->
    select * from t_user where username = #{param1} and password = #
{param2}
</select>
```

其中这两种方式也可以混合使用：

```
<!--User checkLogin(String username,String password);-->
<select id="checkLogin" resultType="user">
    <!--3、混合使用-->
    select * from t_user where username = #{arg0} and password = #{param2}
</select>
```

> ${}的使用方法与#{}类似，只是在使用${}时需要手动加上单引号。

测试方法为：

```
@Test
public void testCheckLogin(){
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);
    User user = mapper.checkLogin("admin", "123456");
    System.out.println(user);
    sqlSession.close();
}
```

6.3、map集合类型的参数

若mapper接口中的方法需要的参数为多个时，此时可以手动创建mapper集合，将这些数据放在map中。还是以验证登录为例，在mapper中创建接口为：

```
/**
 * 通过map参数传递参数
 * @param map
 * @return
 */
User checkLoginByMap(Map<String,Object> map);
```

这样在调用时，就需要传递一个map作为参数，其中键的值就可以自定义，在测试方法中创建一个map作为参数：

```
@Test
public void testCheckLoginByMap(){
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);
    Map<String,Object> map = new HashMap<>();
    map.put("username","admin");
    map.put("password","123456");
    User user = mapper.checkLoginByMap(map);
    System.out.println(user);
    sqlSession.close();
}
```

然后就可以使用自己设置的键来访问相对应的数据，MyBatis映射文件的内容为：

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

> 同样，${}的使用方法与#{}类似，只是在使用${}时需要手动加上单引号。

6.4、实体类类型参数

当mapper接口中的方法参数为实体类对象时，以添加一个用户为例，mapper接口代码为

```
/**
 * 添加用户信息
 * @param user
 */
void insertUser(User user);
```

此时就可以使用${}或者#{}，`通过访问实体类对象中的属性名获取属性值`。

> 属性名不一定是成员变量的名称，而是将get和set方法的方法名去掉get和set后，然后将剩余部分首字母小写作为属性名，一般属性名与成员变量名相同。

MyBatis映射文件的内容为：

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

测试方法为：

```
@Test
public void testInsertUser(){
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);
    User user = new User(null, "a123", "123456", 19, "女",
"123450@qq.com");
    mapper.insertUser(user);
    sqlSession.close();
}
```

6.5、使用@Param标识参数（推荐使用）

可以通过@Param注解标识mapper接口中方法的参数，还是以检查登录为例，mapper接口中添加@Param注解的方式为：

```
/**
 *验证登录，使用@Param注解
 * @param username
 * @param password
 * @return
 */
User checkLoginByParam(@Param("username") String username,
@Param("password") String password);
```

此时MyBatis会将这些参数放在map中，以两种方式存储：

1. 以@Param注解的value值为键，以参数为值。
2. 以param1,param2,...为键，以参数为值。

然后只需要通过${}或者#{}访问map集合中的键，就可以获取相对应的值。

MyBatis映射文件内容为：

```
<!--User checkLoginByParam(@Param("username") String username,
@Param("password") String password);-->
<select id="checkLoginByParam" resultType="user">
    <!--1、以param1,param2,...为键，以参数为值。-->
    <!--select * from t_user where username = #{param1} and password = #
{param2}-->
    <!--2、以@Param注解的value值为键，以参数为值。-->
    select * from t_user where username = #{username} and password = #
{password}
</select>
```

> 同样，${}的使用方法与#{}类似，只是在使用${}时需要手动加上单引号。

测试方法为:

```
@Test
public void testCheckLoginByParam(){
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);
    User user = mapper.checkLoginByParam("admin","123456");
    System.out.println(user);
    sqlSession.close();
}
```

6.6、接口参数为list和数组时（不常用）

> 以这两种作为参数并不常用，因为可以使用`map`集合或`@Param`标识参数来替代

参数为list和数组时，mybatis也会将其放在一个map中：

- 参数为list集合时，以`list`为键，以集合位为值放在map中。
- 参数为一个数组时，以`array`为键，以数组为值放在map中。

1、参数为list时：

还是以验证登录为例，将用户名和密码两个参数放到一个list集合中。mapper接口定义的方法为：

```
/**
 * 验证登录，以List作为参数
 * @param loginInfo
 * @return
 */
User checkLoginByPList(List<String> loginInfo);
```

MyBatis映射文件中，通过`list`键名来获取值，代码如下：

```
<!--User checkLoginByPList(List<String> loginInfo);-->
<select id="checkLoginByPList" resultType="user">
    select * from t_user where username = #{list[0]} and password = #
{list[1]}
</select>
```

测试方法和结果如下：

```
@Test
public void testCheckLoginByList(){
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);
    List<String> list = new ArrayList<>();
    list.add("admin");
    list.add("123456");
    User user = mapper.checkLoginByPList(list);
    System.out.println(user);
    sqlSession.close();
}
```

```
User{id=1, username='admin', password='123456', age=18, gender='男',
email='123456@qq.com'}
```

2、参数为数组时：

还是以验证登录为例，将用户名和密码两个参数放到一个数组中。mapper接口定义的方法为：

```
/**
 * 验证登录，以数组作为参数
 * @param loginInfo
 * @return
 */
User checkLoginByArray(String[] loginInfo);
```

MyBatis映射文件中，通过`array`键名来获取值，代码如下：

```
<!--User checkLoginByArray(String[] loginInfo);-->
<select id="checkLoginByArray" resultType="user">
    select * from t_user where username = #{array[0]} and password = #
{array[1]}
</select>
```

测试方法和结果如下：

```
@Test
public void testCheckLoginByArray(){
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);
    String[] array1 = {"admin","123456"};
    User user = mapper.checkLoginByArray(array1);
    System.out.println(user);
    sqlSession.close();
}
```

```
User{id=1, username='admin', password='123456', age=18, gender='男',
email='123456@qq.com'}
```

7、MyBatis各种查询功能

7.1、查询一个实体类对象

当查询的结果为一个实体类对象时，在定义mapper接口时，返回类型为该实体类。

```
/**
 * 根据id查询用户信息
 * @param id
 * @return
 */
User getUserById(@Param("id") Integer id);
```

在MyBatis映射文件中，需要指定`resultType`的值为返回的实体类（因为在核心配置文件中已经为User类起了别名，因此这里不需要写全类名，直接写user即可）。

```
<!--User getUserById(@Param("id") Integer id);-->
<select id="getUserById" resultType="user">
    select * from t_user where id = #{id}
</select>
```

7.2、查询一个list集合

当查询的结果为一个list集合时，在定义mapper接口时，返回类型应为一个集合。

```
/**
 * 查询所有用户信息
 * @return
 */
List<User> getAllUser();
```

在MyBatis映射文件中，与查询一个实体类对象一样，只需要指定`resultType`的值为返回的实体类即可。

```
<!--List<User> getAllUser();-->
<select id="getAllUser" resultType="user">
    select * from t_user
</select>
```

> 当查询的数据为多条时，不能使用实体类作为返回值，应使用一个集合作为返回值，否则会抛出异常。
>
> 若查询的数据只有一条，可以使用实体类或集合作为返回值。

7.3、查询单个数据

在MyBatis中，对Java中常用的数据类型都设置了别名，并且别名时不区分大小写的。常用的Java类型内建的类型别名为如下。全部可参考官方文档。

别名	映射的类型
_int,_integer	int
_double	double
_boolean	boolean
int，integer	Integer
double	Double
string	String
map	Map
list	List

下面以查询所有数据的个数为例进行演示，mapper接口代码为：

```
/**
 * 查询用户的总数量
 * @return
 */
Integer getCount();
```

返回类型为`Integer`，因此MyBatis映射文件中`resultType`的值可以设置为`int`、`integer`、`java.lang.Integer`、`_int`等都可以。

```
<!--Integer getCount();-->
<select id="getCount" resultType="int">
    select count(*) from t_user
</select >
```

7.4、查询一条数据为map集合

当查询出来的结果没有相对应的实体类，例如要查询所有用户的最大年龄、最小年龄以及平均年龄，那么这时候就可以使用`map`集合作为查询结果。

当查询的结果为`map`集合时，`map`集合应以字段为键，以字段的值为值。

下面以查询指定id的数据存放到`map`集合为例，mapper接口中的内容为：

```
/**
 * 根据id查询用户信息为一个map集合
 * @param id
 * @return
 */
Map<String,Object> getUserByIdToMap(@Param("id") Integer id);
```

MyBatis映射文件中，`resultType`的值应设置为`map`，这样返回的值的类型才为`map`。

```
<!--Map<String,Object> getUserByIdToMap(@Param("id") Integer id);-->
<select id="getUserByIdToMap" resultType="map">
    select * from t_user where id = #{id}
</select>
```

测试类的代码为：

```
@Test
public void testGetUserByIdToMap(){
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    SelectMapper mapper = sqlSession.getMapper(SelectMapper.class);
    Map<String, Object> map = mapper.getUserByIdToMap(1);
    System.out.println(map);
    sqlSession.close();
}
```

输出结果：

```
{password=123456, gender=男, id=1, age=18, email=123456@qq.com,
username=admin}
```

> 如果某一条数据某一个字段为空，例如age字段为空，那么当查询结果为`map`类型时将不会显示该字段的结果；而如果查询结果为实体类时，则会显示该字段结果为null。

7.5、查询多条数据为map集合

当查询的信息为map集合时，例如查询所有的用户信息为map集合。此时查询的数据有多条，并且要将每条数据转换称map结合。这种情况有两种解决方案：

1. 将mapper接口方法的返回值设置为泛型为map的list集合，`这种方法使用比较多`。此时mapper接口的方法可以写成：

   ```
   /**
    * 查询所有的用户信息为map集合
    * @return
    */
   List<Map<String,Object>> getAllUserToMap();
   ```

   MyBatis映射文件中的`resultType`值应为`map`。

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

测试类如下

```
@Test
public void testGetAllUserToMap(){
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    SelectMapper mapper = sqlSession.getMapper(SelectMapper.class);
    List<Map<String, Object>> allUserToMap = mapper.getAllUserToMap();
    System.out.println(allUserToMap);
    sqlSession.close();
}
```

运行结果如下,其结果是一个包含多个`map`的`list`集合。

```
[{password=123456, gender=男, id=1, age=18, email=123456@qq.com,
username=admin}, {password=123456, gender=女, id=2, age=23,
email=234567@qq.com, username=root}, {password=123456, gender=女, id=3,
age=19, email=123450@qq.com, username=a123}]
```

2. 可以将每条数据转换的map集合放在一个大的map中，但是必须要通过`@MapKey`注解，将查询的某个字段的值作为大的`map`的键。

   下面将`id`字段的值作为大的`map`的键，mapper接口中的方法为

```
/**
 * 查询所有的用户信息为map集合
 * @return
 */
@MapKey("id")
Map<String,Object> getAllUserToMap();
```

MyBatis映射文件中的`resultType`值应为`map`。

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

测试类如下

```
@Test
public void testGetAllUserToMap(){
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    SelectMapper mapper = sqlSession.getMapper(SelectMapper.class);
    /*List<Map<String, Object>> allUserToMap =
mapper.getAllUserToMap();*/
    Map<String, Object> allUserToMap = mapper.getAllUserToMap();
    System.out.println(allUserToMap);
    sqlSession.close();
}
```

运行结果如下,其结果是一个包含多个`map`的`map`集合，大的`map`的键为字段`id`的值。

```
{1={password=123456, gender=男, id=1, age=18, email=123456@qq.com,
username=admin}, 2={password=123456, gender=女, id=2, age=23,
email=234567@qq.com, username=root}, 3={password=123456, gender=女,
id=3, age=19, email=123450@qq.com, username=a123}}
```

8、特殊SQL的执行

在上面介绍了使用${}或 #{}来获取参数，其中${}的本质为字符串拼接； #{}为占位符赋值。

上面说到一般使用#{}占位符赋值来获取参数。但是在有些时候简单使用占位符赋值时行不通的。下面看一下以下几种情况。

8.1、模糊查询

模糊查询时我们在查询数据时经常用到的，以查询所有用户名包含`a`的用户为例，其查询SQL语句为：

```
select * from t_user where username like '%a%'
```

mapper接口中的方法为：

```
/**
 * 测试模糊查询
 * @param mohu
 * @return
 */
List<User> testMohu(@Param("mohu") String mohu);
```

假设使用普通的#{}占位符赋值来获取参数值，那么MyBatis配置文件的内容为

```
<!--List<User> testMohu(@Param("mohu") String mohu);-->
<select id="testMohu" resultType="user">
    select * from t_user where username like '%#{mohu}%'
</select>
```

下面测试以下这样写是否正确，测试类为：

```
@Test
public void testGetUserByLike(){
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    SpecialSQLMapper mapper = sqlSession.getMapper(SpecialSQLMapper.class);
    List<User> user = mapper.getUserByLike("a");
    user.forEach(System.out::println);
}
```

运行后会产生如下错误：

```
Caused by: java.sql.SQLException: Parameter index out of range (1 > number
of parameters, which is 0).
```

这个错误是参数索引超出范围，当前的参数为0，而我们操作的参数为1。造成这种的原因是#{}为占位符赋值，其被放在单引号中时（如上面的`'%#{mohu}%'`），`#{mohu}`就被看成为?，因此`'%#{mohu}%'`就成为`'%?%'`，其中的?就变成了一个字符而不是占位符，因此当前的参数个数为0。

`解决上面问题又三种方法：`

1. 使用${}进行字符串拼接来进行赋值，其可以将参数的值直接拼接到SQL语句中，其在映射文件中书写形式如下：

   ```
   <!--List<User> getUserByLike(@Param("mohu") String mohu);-->
   <select id="getUserByLike" resultType="user">
       <!--使用${}字符串拼接赋值-->
       select * from t_user where username like '%${mohu}%'
   </select>
   ```

2. 使用#{}和SQL中的字符串拼接函数concat()。`这种书写方式比较麻烦`

   ```
   <!--List<User> getUserByLike(@Param("mohu") String mohu);-->
   <select id="getUserByLike" resultType="user">
       <!--使用#{}和SQL中的字符串拼接函数concat()-->
       select * from t_user where username like concat('%',#{mohu},'%')
   </select>
   ```

3. 使用双引号和#{}进行拼接，`比较常用`

   ```
   <!--List<User> getUserByLike(@Param("mohu") String mohu);-->
   <select id="getUserByLike" resultType="user">
       <!--使用双引号+#{}进行拼接-->
       select * from t_user where username like "%"#{mohu}"%"
   </select>
   ```

8.2、批量删除

在MySQL中实现批量删除的方式有两种，以同时删除指定id的两条数据为例，SQL语句有如下两种书写方式。

```
#第一种方式
delete from t_user where id = 5 or id = 6
#第二种方式
delete from t_user where id in (6,7)
```

第一种SQL语句在MyBatis中使用${}和#{}都可以，下面主要条论使用第二种方式。在第二种方式中，主要是`in`后面括号中将要删除数据的id用`,`拼接起来，因此要传递的参数是一个形如`id1,id2,...`的字符串。mapper接口中的代码为：

```
/**
 * 批量删除
 * @param ids
 */
void deleteMoreUser(@Param("ids") String ids);
```

下面就是在MyBatis映射文件中数学SQL语句，一般情况下我们会使用#{}来获取参数，下面看一下在MyBatis中的SQL语句为：

```
delete from t_user where id in (#{ids})
```

> 这种写法是错误的，因为#{}是占位符赋值，会给参数添加一个单引号，假如我们想删除5和6两条数据，因此我们传的参数为`5,6`，那么我们如果使用#{}，那么SQL语句将变成：
>
> ```
> delete from t_user where id in ('5,6')
> ```
>
> 而不是我们所希望的：
>
> ```
> delete from t_user where id in (5,6)
> ```
>
> 因此执行会报错，那么我们可以使用${}字符串拼接的方式进行赋值。

使用${}拼接方式在MyBatis的实现为：

```
<!--void deleteMoreUser(@Param("ids") String ids);-->
<delete id="deleteMoreUser">
    delete from t_user where id in (#{ids})
</delete>
```

测试方法为

```
@Test
public void testDeleteMoreUser(){
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    SpecialSQLMapper mapper = sqlSession.getMapper(SpecialSQLMapper.class);
    mapper.deleteMoreUser("4,8");
    sqlSession.close();
}
```

执行成功。

8.3、动态设置表名

在查询时，当表名不确定的时候，我们需要输入要查询的表名，此时表名是不需要添加单引号的，如果使用#{}的方式会添加单引号，因此在这种情况下不能使用#{}，而应该使用${}。

mapper接口中的方法为：

```
/**
 * 动态设置表明，查询用户信息
 * @param tableName
 * @return
 */
List<User> getUserList(@Param("tableName") String tableName);
```

MyBatis映射文件内容为：

```
<!--List<User> getUserList(@Param("tableName") String tableName);-->
<select id="getUserList" resultType="user">
    select * from ${tableName}
</select>
```

8.4、添加功能获取自增的主键

在主键为自增的情况下，添加一条数据时不需要给主键传值，但是有时候我们希望在添加一条数据后，能够知道系统给的主键的值是多少。因此就需要使用MyBatis映射文件中`insert`标签的`useGeneratedKeys`和`keyProperty`两个属性。这两个属性的功能分别为：

- `useGeneratedKeys`：设置使用自增的主键。
- `keyProperty`：因为增删改的返回值是受影响的行数，因此只能将获取的自增的主键放在传输的参数user对象的某个属性中，使用`keyProperty`属性指定user对象的属性。

下面看看如何实现，mapper接口中的方法为：

```
/**
 * 添加用户信息并返回主键
 * @param user
 * @return
 */
int insertUser(User user);
```

MyBatis映射文件的内容为：

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

测试方法为：

```
@Test
public void testinsertUser(){
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    SpecialSQLMapper mapper = sqlSession.getMapper(SpecialSQLMapper.class);
    User user = new User(null, "zhangsan", "123456", 23, "男", "42343@qq.com");
    mapper.insertUser(user);
    System.out.println(user);
    sqlSession.close();
}
```

其输出为：

```
User{id=9, username='zhangsan', password='123456', age=23, gender='男', email='42343@qq.com'}
```

可以看出user对象在为被添加到数据库之前其id的值为`null`，在执行完添加之后，user对象的id值变为自增的主键值。

9、自定义映射resultMap

为演示本节的操作，创建员工表`t_emp`和部门表`t_dept`。其创建的SQL代码为

```
#员工表t_emp
CREATE TABLE IF NOT EXISTS `t_emp`(
    `emp_id` int NOT NULL AUTO_INCREMENT,
    `emp_name` varchar(20),
    `age` int,
    `gender` char,
    `dept_id` int,
    PRIMARY KEY(`emp_id`)
)ENGINE=INNODB DEFAULT CHARSET=utf8;
#部门表t_dept
CREATE TABLE IF NOT EXISTS `t_dept`(
    `dept_id` int NOT NULL AUTO_INCREMENT,
    `dept_name` varchar(20),
    PRIMARY KEY(`dept_id`)
)ENGINE=INNODB DEFAULT CHARSET=utf8;
```

本章主要介绍`resultMap`，下面看看其都有那些常用的属性和标签。

resultMap:设置自定义映射

- 属性：

  id:表示自定义映射的唯一标识

  type:查询的数据要映射的实体类的类型

- 子标签：

  id:设置主键的映射关系

  result:设置普通字段的映射关系

  association:设置多对一的映射关系

  collection:设置一对多的映射关系

  - 属性

    property:设置映射关系中实体类中的属性名

    column:设置映射关系中表中的字段名

9.1、字段名和属性名不一致的情况

在创建完表之后，就需要为其创建相对应的实体类。员工类`Emp`的属性(成员变量)如下所示(构造方法、get、set等方法自行加上)。

```
public class Emp {
    private Integer empId;
    private String empName;
    private Integer age;
    private String gender;
}
```

部门类`Dept`的属性如下：

```
public class Dept {
    private Integer deptId;
    private String deptName;
}
```

在java中属性的命名方式遵循驼峰命名，而SQL中的命名方式以下划线分开，例如SQL中的字段`emp_id`，在java中对应的属性名为`empId`。这就导致了字段名和属性名不一致的情况。

而在我们进行查询操作时，是通过查询得到的字段名，通过反射找对对应的属性名，然后通过属性名对属性进行赋值，也就是说需要`属性名和字段名保持一致`。下面看看当属性名和字段名不一致的情况下，我们查询出来的结果是什么样子的，以根据id查询员工信息为例，首先在mapper接口中定义方法如下：

```
/**
 * 根据empId查询员工信息
 * @param empId
 * @return
 */
Emp getEmpByEmpId(@Param("empId") Integer empId);
```

在当字段名和属性名一致的情况下，MyBatis映射文件中的内容为：

```
<!--Emp getEmpByEmpId(@Param("empId") Integer empId);-->
<select id="getEmpByEmpId" resultType="emp">
    select * from t_emp where emp_id = #{empId}
</select>
```

下面通过如下的测试方法看看这样写会不会报错

```
@Test
public void testGetEmpByEmpId(){
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);
    Emp emp = mapper.getEmpByEmpId(1);
    System.out.println(emp);
    sqlSession.close();
}
```

输出内容如下：

```
Emp{empId=null, empName='null', age=20, gender='男'}
```

由输出可以看出属性名`empId`和`empName`的属性所对应的值为`null`，而其他两个属性值是正确的。这是因为属性`empId`和`empName`与数据库中相对应的字段的名字是不一致的，因此导致属性的值为`null`。可见当属性名和字段名不一致的情况下，是无法将正确的查询结果赋值给java对象的。

下面介绍几种解决方法：

1. 在查询时，可以通过为字段起别名，保证字段名和实体类中的属性名保持一致，此时的MyBatis映射文件的内容为：

   ```
   <!--Emp getEmpByEmpId(@Param("empId") Integer empId);-->
   <select id="getEmpByEmpId" resultType="emp">
       <!--为字段起别名-->
       select emp_id empId,emp_name empName,age,gender from t_emp where
   emp_id = #{empId}
   </select>
   ```

2. 可以在核心配置文件中设置一个全局配置信息`mapUnderscoreToCamelCase`，将其对应的值设置为`true`。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。

核心配置文件添加的内容为(`推荐将其添加到模板中，自动映射会经常被使用`):

```
<!--开启驼峰命名自动映射-->
<settings>
    <setting name="mapUnderscoreToCamelCase" value="true"/>
</settings>
```

此时MyBatis映射文件的内容就不需要起别名:

```
<!--Emp getEmpByEmpId(@Param("empId") Integer empId);-->
<select id="getEmpByEmpId" resultType="emp">
    <!--开启驼峰命名自动映射后-->
    select * from t_emp where emp_id = #{empId}
</select>
```

3. 使用`resultMap`自定义映射处理，通过resultMap设置自定义映射，而不再使用resultType指定映射的类型。

   MyBatis映射文件中的内容为:

```
<!--
所用标签和属性
resultMap:设置自定义映射
    属性:
    id:表示自定义映射的唯一标识
    type:查询的数据要映射的实体类的类型
    子标签:
    id:设置主键的映射关系
    result:设置普通字段的映射关系
        属性
        property:设置映射关系中实体类中的属性名
        column:设置映射关系中表中的字段名
-->
<resultMap id="empResultMap" type="emp">
    <id column="emp_id" property="empId"/>
    <result column="emp_name" property="empName"/>
    <result column="age" property="age"/>
    <result column="gender" property="gender"/>
</resultMap>
<!--Emp getEmpByEmpId(@Param("empId") Integer empId);-->
<select id="getEmpByEmpId" resultMap="empResultMap">
    select * from t_emp where emp_id = #{empId}
</select>
```

9.2、多对一映射处理

多对一映射所使用的场景举例:

- 查询员工信息以及员工所对应的部门信息

首先需要在员工类(Emp)中添加一个部门(Dept)属性。

```
public class Emp {
    private Integer empId;
    private String empName;
    private Integer age;
    private String gender;
    private Dept dept;
}
```

我们可以通过以下SQL语句进行多表联查将员工及其所属的部门的信息查询出来

```
select t_emp.*,t_dept.* from t_emp left join t_dept on t_emp.dept_id =
t_dept.dept_id where t_emp.emp_id = 1
```

但是将查询的结果映射到员工类时，员工的信息可以与对象中的属性进行映射，但是部门的信息却无法进行映射。

下面由三种方法来解决多对一映射的问题。

9.2.1、级联方式处理映射关系

首先现在mapper接口中定义一个查询员工信息的方法。

```
/**
 * 获取员工以及所对应的部门信息
 * @param empId
 * @return
 */
Emp getEmpAndDeptByEmpId(@Param("empId") Integer empId);
```

在使用级联方式处理多对一映射关系时，需要使用`resultMap`，其方法是将查询出的部门对应的字段`dept_id`和`dept_name`分别与属性`dept.deptId`和`dept.deptName`对应，MyBatis映射文件的代码如下

```
<resultMap id="empAndDeptResultMap" type="emp">
    <id column="emp_id" property="empId"/>
    <result column="emp_name" property="empName"/>
    <result column="age" property="age"/>
    <result column="gender" property="gender"/>
    <result column="dept_id" property="dept.deptId"/>
    <result column="dept_name" property="dept.deptName"/>
</resultMap>
<!--Emp getEmpAndDeptByEmpId(@Param("empId") Integer empId);-->
<select id="getEmpAndDeptByEmpId" resultMap="empAndDeptResultMap">
    select t_emp.*,t_dept.*
    from t_emp left join t_dept on t_emp.dept_id = t_dept.dept_id
    where t_emp.emp_id = #{empId}
</select>
```

测试代码及测试结果如下:

```
@Test
public void testGetEmpAndDeptByEmpId(){
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);
    Emp emp = mapper.getEmpAndDeptByEmpId(1);
    System.out.println(emp);
    sqlSession.close();
}
```

```
Emp{empId=1, empName='张三', age=20, gender='男', dept=Dept{deptId=1,
deptName='A'}}
```

可正确的查询出结果。

9.2.2、使用`association`处理映射关系

第二种方法是使用`association`处理映关系。`association`处理多对一的映射关系(处理实体类类型的属性(如员工类中部门类的属性))。需要设置其两个属性：

- property设置需要处理映射关系的属性的属性名(在本例中为dept)。
- javaType:设置要处理的属性的类型(在本例中为dept)。

MyBatis映射文件的代码如下：

```
<!--
    association:处理多对一的映射关系(处理实体类类型的属性)
    - property设置需要处理映射关系的属性的属性名(在本例中为dept)。
    - javaType:设置要处理的属性的类型(在本例中为dept)。
-->
<resultMap id="empAndDeptResultMap" type="emp">
    <id column="emp_id" property="empId"/>
    <result column="emp_name" property="empName"/>
    <result column="age" property="age"/>
    <result column="gender" property="gender"/>
    <association property="dept" javaType="dept">
        <id column="dept_id" property="deptId"/>
        <result column="dept_name" property="deptName"/>
    </association>
</resultMap>
<!--Emp getEmpAndDeptByEmpId(@Param("empId") Integer empId);-->
<select id="getEmpAndDeptByEmpId" resultMap="empAndDeptResultMap">
    select t_emp.*,t_dept.*
    from t_emp left join t_dept on t_emp.dept_id = t_dept.dept_id
    where t_emp.emp_id = #{empId}
</select>
```

9.2.3、分步查询

分布查询就是分步骤查询，还是以查询员工信息以及员工所对应的部门信息为例，首先将员工信息查询出来，然后再用查询出来的部门ID去查询部门的信息。

第一步：查询员工信息

首先在`EmpMapper`接口中定义查询员工信息的方法：

```
/**
 * 通过分步查询 查询员工以及所对应的部门信息的第一步
 * @param empId
 * @return
 */
Emp getEmpAndDeptByStepOne(@Param("empId") Integer empId);
```

`EmpMapper`都对应的MyBatis映射文件中的内容为：

```
<resultMap id="empAndDeptByStepResultMap" type="Emp">
    <id column="emp_id" property="empId"/>
    <result column="emp_name" property="empName"/>
    <result column="age" property="age"/>
    <result column="gender" property="gender"/>
    <association property="dept"
select="com.tianna.mybatis.mapper.DeptMapper.getEmpAndDeptByStepTwo"
                 column="dept_id">
    </association>
</resultMap>
<!--Emp getEmpAndDeptByStepOne(@Param("empId") Integer empId);-->
<select id="getEmpAndDeptByStepOne" resultMap="empAndDeptByStepResultMap">
    select * from t_emp where emp_id = #{empId}
</select>
```

这种方法使用了`association`标签的`property`、`select`和`column`属性，其作用为：

- property：设置需要处理映射关系的属性的属性名
- select：设置分步查询的sql的唯一标识
- column：将查询出的某个字段作为分步查询的sql的条件

其中`select`的值为分步查询的sql的唯一标识，即查询部门信息的sql，需要在第二步中实现，`column`为分步查询的sql的条件，即将第一步查询出的部门id作为第二步的条件。

第二步：根据员工所对应的部门id查询部门信息

首先在`DeptMapper`接口中定义根据部门id查询部门信息的方法。

```
/**
 * 通过分步查询 查询员工以及所对应的部门信息的第二步
 * @return
 */
Dept getEmpAndDeptByStepTwo(@Param("deptId") Integer deptId);
```

然后再`DeptMapper`接口对应的MyBatis映射文件中实现根据部门id查询部门信息的sql。其文件的全类名+id名，即`com.tianna.mybatis.mapper.DeptMapper.getEmpAndDeptByStepTwo`作为第一步中`select`属性的值。

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

测试方法即输出结果如下：

```
@Test
public void testGetEmpAndDeptByStep(){
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);
    Emp emp = mapper.getEmpAndDeptByStepOne(1);
    System.out.println(emp);
    sqlSession.close();
}
```

```
Emp{empId=1, empName='张三', age=20, gender='男', dept=Dept{deptId=1,
deptName='A'}}
```

9.3、一对多映射处理

> 一对多的案例：一个部门包含多个员工；查询一个部门的信息，并将一个部门中所有员工的信息查询出来。

一个部门包含多个员工，就需要再部门类中添加一个员工List集合的一个属性，添加后的部门类的属性为：

```
public class Dept {
    private Integer deptId;
    private String deptName;
    private List<Emp> emps;
}
```

对于一对多映射关系，由两种处理方法：

- 使用collection标签
- 分步查询

9.3.1、collection

在`DeptMapper`接口中定义查询部门以及部门中员工信息的方法：

```
/**
 * 查询部门以及部门中的员工信息
 * @param deptId
 * @return
 */
Dept getDeptAndEmpByDeptId(@Param("deptId") Integer deptId);
```

一个部门中的员工有多个，因此使用`collection`来将查询到的多个员工信息映射到部门类中的员工列表属性中。

collection:处理一对多映射关系(处理集合类型属性)

- property:设置需要处理映射关系的属性的属性名
- ofType:设置集合类型的属性中存储的数据的类型

`DeptMapper`接口所对应的映射文件的内容为：

```
<resultMap id="deptAndEmpResultMap" type="dept">
    <id column="dept_id" property="deptId"/>
    <result column="dept_name" property="deptName"/>
    <!--
    collection:处理一对多映射关系(处理集合类型属性)
        property:设置需要处理映射关系的属性的属性名
        ofType:设置集合类型的属性中存储的数据的类型
    -->
    <collection property="emps" ofType="emp">
        <id column="emp_id" property="empId"/>
        <result column="emp_name" property="empName"/>
        <result column="age" property="age"/>
        <result column="gender" property="gender"/>
    </collection>
</resultMap>
<!--Dept getDeptAndEmpByDeptId(@Param("deptId") Integer deptId);-->
<select id="getDeptAndEmpByDeptId" resultMap="deptAndEmpResultMap">
    select *
    from t_dept left join t_emp on t_emp.dept_id = t_dept.dept_id
    where t_dept.dept_id = 1
</select>
```

测试方法以及测试结果为:

```
@Test
public void testGetDeptAndEmpByDeptId(){
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    DeptMapper mapper = sqlSession.getMapper(DeptMapper.class);
    Dept dept = mapper.getDeptAndEmpByDeptId(1);
    System.out.println(dept);
    sqlSession.close();
}
```

```
Dept{deptId=1, deptName='A', emps=[Emp{empId=1, empName='张三', age=20,
gender='男', dept=null}, Emp{empId=4, empName='赵六', age=23, gender='男',
dept=null}]}
```

9.3.2、分步查询

一对多的分步查询与多对一的分步查询类似。再本例中，首先通过部门id查询部门信息，然后再通过部门id查询员工信息。

第一步：查询部门信息

再`DeptMapper`接口中定义通过id查询部门信息的接口。

```
/**
 * 通过分步查询查询部门以及部门中的员工信息的第一步
 * @param deptId
 * @return
 */
Dept getDeptAndEmpByStepOne(@Param("deptId") Integer deptId);
```

然后在其对应的映射文件中将查询的结果与实体类进行映射，需要用到使用了`collection`标签的`property`、`select`和`column`属性，其作用为：

- property：设置需要处理映射关系的属性的属性名
- select：设置分步查询的sql的唯一标识
- column：将查询出的某个字段作为分步查询的sql的条件

其中`select`的值为分步查询的sql的唯一标识，即查询部门信息的sql，需要在第二步中实现，`column`为分步查询的sql的条件，即将第一步查询出的部门id作为第二步的条件。

MyBatis映射文件中的内容为：

```
<resultMap id="deptAndEmpResultMapByStep" type="dept">
    <id column="dept_id" property="deptId"/>
    <result column="dept_name" property="deptName"/>
    <collection property="emps"

 select="com.tianna.mybatis.mapper.EmpMapper.getDeptAndEmpByStepTwo"
                column="dept_id">
    </collection>
</resultMap>
<!--Dept getDeptAndEmpByStepOne(@Param("deptId") Integer deptId);-->
<select id="getDeptAndEmpByStepOne" resultMap="deptAndEmpResultMapByStep">
    select * from t_dept where dept_id = #{deptId}
</select>
```

第二步：根据部门id查询部门中的所有员工

首先在`EmpMapper`接口中定义根据部门id查询员工信息的方法。

```
/**
 * 通过分步查询查询部门以及部门中的员工信息的第二步
 * @param deptId
 * @return
 */
List<Emp> getDeptAndEmpByStepTwo(@Param("deptId") Integer deptId);
```

然后再`EmpMapper`接口对应的MyBatis映射文件中实现根据部门id查询员工信息的sql。其文件的全类名+id名，即`com.tianna.mybatis.mapper.EmpMapper.getDeptAndEmpByStepTwo`作为第一步中`select`属性的值。

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

测试方法即输出结果如下：

```
@Test
public void testGetDeptAndEmpByStep(){
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    DeptMapper mapper = sqlSession.getMapper(DeptMapper.class);
    Dept dept = mapper.getDeptAndEmpByStepOne(1);
    System.out.println(dept);
    sqlSession.close();
}
```

```
Dept{deptId=1, deptName='A', emps=[Emp{empId=1, empName='张三', age=20,
gender='男', dept=null}, Emp{empId=4, empName='赵六', age=23, gender='男',
dept=null}]}
```

9.4、延迟加载

分步查询的优点：可以实现延迟加载。

上面介绍的分步查询是通过两个SQL语句将最终的结果查询出来。当我们只需要第一个SQL所查询的信息，而不需要第二个SQL所查询的信息时，就可以只执行第一个SQL语句而不需要执行第二个SQL语句。我们可以利用`延迟加载`实现这一目的，可以减少内存的消耗。

不使用延迟加载

再默认的情况下时不适用延迟加载的，下面以查询员工的姓名为例，演示再不适用延迟加载时执行了几个SQL语句，测试方法如下：

```
@Test
public void testGetEmpAndDeptByStep(){
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);
    Emp emp = mapper.getEmpAndDeptByStepOne(1);
    System.out.println(emp.getEmpName());
    sqlSession.close();
}
```

日志信息以及输出结果如下：

```
DEBUG 07-31 12:44:05,742 ==>  Preparing: select * from t_emp where emp_id =
? (BaseJdbcLogger.java:137)
DEBUG 07-31 12:44:05,769 ==> Parameters: 1(Integer)
(BaseJdbcLogger.java:137)
DEBUG 07-31 12:44:05,789 ====>  Preparing: select * from t_dept where
dept_id = ? (BaseJdbcLogger.java:137)
DEBUG 07-31 12:44:05,790 ====> Parameters: 1(Integer)
(BaseJdbcLogger.java:137)
DEBUG 07-31 12:44:05,791 <====      Total: 1 (BaseJdbcLogger.java:137)
DEBUG 07-31 12:44:05,792 <==      Total: 1 (BaseJdbcLogger.java:137)
张三
```

由最终的结果可以看出，当只查询员工信息时，执行了两条SQL语句。然而只查询员工的信息只需要执行第一条SQL就可以了，没必要执行第二条SQL，这时就需要使用延迟加载。

使用延迟加载

使用延迟加载需要在核心配置文件中设置如下全局配置信息：

- 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。将其设置为true，其默认为false。
- aggressiveLazyLoading：当开启时，任何方法的调用都会加载该对象的所有属性。否则，每个属性会按需加载。需要将其设置为false，虽然其默认为false，但是延迟加载也与该项设置有关，最好还是设置以下。将lazyLoadingEnabled设置为true,该项设置为true，那么是没有开启延迟加载的。

核心配置文件中与延迟加载相关的全局配置信息的设置为：

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

开启后再执行上面的测试方法，得到的结果为：

```
DEBUG 07-30 20:21:05,170 ==>  Preparing: select * from t_emp where emp_id =
? (BaseJdbcLogger.java:137)
DEBUG 07-30 20:21:05,193 ==> Parameters: 1(Integer)
(BaseJdbcLogger.java:137)
DEBUG 07-30 20:21:05,247 <==      Total: 1 (BaseJdbcLogger.java:137)
张三
```

可以看出此时只执行了第一个SQL语句，实现了延迟加载的目的。

然而当再全局设置延迟加载后，对于所有的分步查询都会进行延迟加载。当某一个分步查询不需要使用延迟加载时应该怎么办?

这时就可以使用association和collection中的fetchType属性设置当前的分步查询是否使用延迟加载，其有两个参数：

- lazy(延迟加载)
- eager(立即加载)

当在全局设置延迟加载，并在某一个分步查询中使用eager(立即加载)时，当前的分步查询不会使用延迟加载。以association的设置为例，如下：

```
<association fetchType="eager" property="dept"
select="com.tianna.mybatis.mapper.DeptMapper.getEmpAndDeptByStepTwo"
             column="dept_id">
</association>
```

对其进行测试的结果为：

```
DEBUG 07-31 13:24:24,400 ==>  Preparing: select * from t_emp where emp_id =
? (BaseJdbcLogger.java:137)
DEBUG 07-31 13:24:24,430 ==> Parameters: 1(Integer)
(BaseJdbcLogger.java:137)
DEBUG 07-31 13:24:24,452 ====>  Preparing: select * from t_dept where
dept_id = ? (BaseJdbcLogger.java:137)
DEBUG 07-31 13:24:24,453 ====> Parameters: 1(Integer)
(BaseJdbcLogger.java:137)
DEBUG 07-31 13:24:24,454 <====      Total: 1 (BaseJdbcLogger.java:137)
DEBUG 07-31 13:24:24,456 <==      Total: 1 (BaseJdbcLogger.java:137)
张三
```

可见虽然全局设置了延迟加载，但将fetchType设置为eager后，当前分步查询并不会延迟加载。

10、动态SQL

MyBatis框架的动态SQL技术是一种根据特定条件动态拼装SQL语句的功能。 在使用JDBC或其它类似的框架时，根据不同条件拼接SQL语句是非常痛苦的，例如拼接时要确保不能忘记添加必要的空格，还要注意去掉列表最后一个列名的逗号。利用动态 SQL，可以彻底摆脱这种痛苦。

10.1、if

> if标签可以通过其test属性的表达式进行判断，若表达式的结果为true，则标签中的内容会执行；反之标签中的内容不执行。

下面通过多条件查询员工信息来演示if标签如何使用。Mapper接口定义查询员工信息的方法。

```
/**
 * 根据条件查询员工信息
 * @param emp
 * @return
 */
List<Emp> getEmpByCondition(Emp emp);
```

在MyBatis映射文件使用if标签判断某一属性的值是否不为null和""，来决定是否执行标签中包含的SQL语句。

```
<!--List<Emp> getEmpByCondition(Emp emp);-->
<select id="getEmpByCondition" resultType="emp">
    select * from t_emp where
    <if test="empName != null and empName != ''">
        emp_name = #{empName}
    </if>
    <if test="age != null and age != ''">
        and age = #{age}
    </if>
    <if test="gender != null and gender != ''">
        and gender = #{gender}
    </if>
</select>
```

测试代码及测试结果如下：

```
@Test
public void testGetEmpByCondition(){
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    DynamicSQLMapper mapper = sqlSession.getMapper(DynamicSQLMapper.class);
    Emp emp = new Emp(null,"张三",20,"男");
    List<Emp> list = mapper.getEmpByCondition(emp);
    list.forEach(System.out::println);
}
```

```
Emp{empId=1, empName='张三', age=20, gender='男'}
```

存在的问题

但只是用if标签还存在很多问题，

1、当empName、age、gender全为空或""时，我们期望能够查询没有任何条件限制的所有员工信息，SQL语句为：

```
select * from t_emp
```

但实际上的SQL语句比期望的SQL多了一个where：

```
select * from t_emp where
```

2、当第一个属性empName为空，后面属性age不为空时，SQL语句就会多出一个and，如下所示：

```
select * from t_emp where and age = ?
```

下面有几种解决方法：

在where关键字后添加一个恒成立条件(例如1=1)，然后再第一个if标签中的sql语句前加上and关键字，如下所示：

```
<!--List<Emp> getEmpByCondition(Emp emp);-->
<select id="getEmpByCondition" resultType="emp">
    select * from t_emp where 1=1
    <if test="empName != null and empName != ''">
        and emp_name = #{empName}
    </if>
    <if test="age != null and age != ''">
        and age = #{age}
    </if>
    <if test="gender != null and gender != ''">
        and gender = #{gender}
    </if>
</select>
```

10.2、where

10.1节中的问题还可以使用where关键字解决，`常用`。

where标签一般和if结合使用：

- 若where标签中有条件成立，会自动生成where关键字
- 会自动将where标签中内容前多余的and去掉，但是其中内容后多余的and无法去掉。
- 若where标签中没有任何一个条件成立，则where没有任何功能

where标签与if标签结合使用的用法如下所示：

```
<select id="getEmpByCondition" resultType="emp">
    select * from t_emp
    <where>
        <if test="empName != null and empName != ''">
            emp_name = #{empName}
        </if>
        <if test="age != null and age != ''">
            and age = #{age}
```

```
        </if>
        <if test="gender != null and gender != ''">
            and gender = #{gender}
        </if>
    </where>
</select>
```

10.3、trim

10.1节中的问题还可以使用trim标签解决。

trim用于去掉或添加标签中的内容。

常用属性：

- prefix：在trim标签中的内容的前面添加某些内容
- prefixOverrides：在trim标签中的内容的前面去掉某些内容
- suffix：在trim标签中的内容的后面添加某些内容
- suffixOverrides：在trim标签中的内容的后面去掉某些内容

trim标签的用法如下所示：

```
<select id="getEmpByCondition" resultType="emp">
    select * from t_emp
    <trim prefix="where" suffixOverrides="and">
        <if test="empName != null and empName != ''">
            emp_name = #{empName} and
        </if>
        <if test="age != null and age != ''">
            age = #{age} and
        </if>
        <if test="gender != null and gender != ''">
            gender = #{gender}
        </if>
    </trim>
</select>
```

10.4、choose、when、otherwise

choose、when、otherwise 相当于java中的if...else if...else。

choose标签包含when和otherwise标签，第一个when相当于if，后面的相当于else if，otherwise相当于else(即前面所有条件都不满足才会执行其下面的SQL)。

下面还是通过根据多条件查询员工信息来演示其用法，只是使用choose、when、otherwise时，当某一个条件满足时，其后面的语句将不再判断。首先在mapper接口中定义方法：

```java
/**
 * 使用choose查询员工信息
 * @param emp
 * @return
 */
List<Emp> getEmpByChoose(Emp emp);
```

在映射文件中使用choose、when、otherwise如下，其中没有使用otherwise标签。

```xml
<!-- List<Emp> getEmpByChoose(Emp emp);-->
<select id="getEmpByChoose" resultType="emp">
    select * from t_emp
    <where>
        <choose>
            <when test="empName != null and empName != ''">
                emp_name = #{empName}
            </when>
            <when test="age != null and age != ''">
                age = #{age}
            </when>
            <when test="gender != null and gender != ''">
                gender = #{gender}
            </when>
        </choose>
    </where>
</select>
```

测试当第一个条件empName不为空时的执行情况，测试方法如下：

```java
@Test
public void testGetEmpByChoose(){
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    DynamicSQLMapper mapper = sqlSession.getMapper(DynamicSQLMapper.class);
    Emp emp = new Emp(null,"张三",20,"男");
    List<Emp> list = mapper.getEmpByChoose(emp);
    list.forEach(System.out::println);
}
```

日志信息以及输出结果如下：

```
DEBUG 07-31 22:56:40,276 ==>  Preparing: select * from t_emp WHERE emp_name
= ? (BaseJdbcLogger.java:137)
DEBUG 07-31 22:56:40,299 ==> Parameters: 张三(String)
(BaseJdbcLogger.java:137)
DEBUG 07-31 22:56:40,319 <==      Total: 1 (BaseJdbcLogger.java:137)
Emp{empId=1, empName='张三', age=20, gender='男'}
```

由日志信息可以看出，当第一个条件满足时，将不会再执行其后的语句。

10.5、foreach

foreach用于对集合进行遍历，foreach元素的功能非常强大，它允许指定一个集合，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。

其常用的属性有以下几个：

- collection：设置要循环的数组或集合
- item：用一个字符串表示数组或集合中的每一个数据
- separator：设置每次循环的数据之间的分割符
- open：循环的所有内容以什么开始
- close：循环的所有内容以什么结束

1、使用foreach批量插入数据

批量插入数据的sql语句为：

```
insert into t_emp values (,,,),(,,,)....
```

在mapper接口定义批量插入的方法：

```
/**
 * 批量添加员工信息
 * @param emps
 */
void insertMoreEmp(@Param("emps") List<Emp> emps);
```

在MyBatis映射文件中使用foreach标签实现批量插入。

```
<!--void insertMoreEmp(@Param("emps") List<Emp> emps);-->
<insert id="insertMoreEmp">
    insert into t_emp values
    <foreach collection="emps" item="emp" separator=",">
        (null,#{emp.empName},#{emp.age},#{emp.gender},null)
    </foreach>
</insert>
```

测试方法如下：

```
@Test
public void testInsertMoreEmp(){
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    DynamicSQLMapper mapper = sqlSession.getMapper(DynamicSQLMapper.class);
    Emp emp1 = new Emp(null,"小明1",21,"男");
    Emp emp2 = new Emp(null,"小明2",22,"女");
    Emp emp3 = new Emp(null,"小明3",23,"男");
    List<Emp> list = Arrays.asList(emp1, emp2, emp3);
    mapper.insertMoreEmp(list);
    sqlSession.close();
}
```

2、使用foreach批量删除数据

批量删除数据又两种sql语句的写法:

```
#第一种写法
delete from t_emp where emp_id in (?,?,?,?)
```

```
#第二种写法
delete from t_emp where emp_id = ? or emp_id = ? ....
```

在mapper接口中定义批量删除的方法

```
/**
 *
 * @param empIds
 */
void deleteMoreEmp(@Param("empIds") Integer[] empIds);
```

在MyBatis映射文件中使用foreach标签实现批量删除的写法:

使用第一种方法批量删除。

```
<delete id="deleteMoreEmp">
    delete from t_emp where emp_id in
    <foreach collection="empIds" item="empId" separator="," open="("
close=")">
        #{empId}
    </foreach>
</delete>
```

使用第二种方法批量删除。

```
<delete id="deleteMoreEmp">
    delete from t_emp where
    <foreach collection="empIds" item="empId" separator="or">
        emp_id = #{empId}
    </foreach>
</delete>
```

测试方法：

```
@Test
public void testDeleteMoreEmp(){
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    DynamicSQLMapper mapper = sqlSession.getMapper(DynamicSQLMapper.class);
    Integer[] empIds = new Integer[]{7,8};
    mapper.deleteMoreEmp(empIds);
    sqlSession.close();
}
```

10.6、SQL片段

sql片段可以记录一段sql，在需要使用的地方使用include标签进行引用。

使用方法如下，以查询时将查询的字段名放入sql标签为例。

首先使用sql标签记录一段sql。

```
<sql id="empColumns">
    emp_id,emp_name,age,gender,dept_id
</sql>
```

使用时使用include标签进行引用即可

```
<select id="getEmpByCondition" resultType="emp">
    select <include refid="empColumns"></include> from t_emp
    <trim prefix="where" suffixOverrides="and">
        <if test="empName != null and empName != ''">
            emp_name = #{empName} and
        </if>
        <if test="age != null and age != ''">
            age = #{age} and
        </if>
        <if test="gender != null and gender != ''">
            gender = #{gender}
        </if>
    </trim>
</select>
```

执行时会将sql标签中的内容拼接到include标签所在处，如：

```
select emp_id,emp_name,age,gender,dept_id from t_emp
```

11、MyBatis的缓存

MyBatis的缓存分为两种：一级缓存和二级缓存。

11.1、MyBatis的一级缓存

一级缓存是SqlSession级别的，通过同一个SqlSession查询的数据会被缓存，`再次使用同一个SqlSession查询同一条数据，会从缓存中获取`，一级缓存是默认开启的。下面通过一个通过id查询员工的例子进行演示：

mapper接口中定义一个根据id查询员工信息的方法：

```
/**
 * 根据员工id查询员工信息
 * @param empId
 * @return
 */
Emp getEmpById(@Param("empId") Integer empId);
```

myBatis映射文件中代码如下：

```
<!--Emp getEmpById(@Param("empId") Integer empId);-->
<select id="getEmpById" resultType="emp">
    select * from t_emp where emp_id = #{empId}
</select>
```

下面测试以下通过同一个SqlSession查询同一条数据，测试方法如下：

```
@Test
public void testGetEmpById(){
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    CacheMapper mapper = sqlSession.getMapper(CacheMapper.class);
    Emp emp1 = mapper.getEmpById(1);
    System.out.println(emp1);
    Emp emp2 = mapper.getEmpById(1);
    System.out.println(emp2);
}
```

日志信息及输出信息如下：

```
DEBUG 08-01 16:57:44,144 ==>  Preparing: select * from t_emp where emp_id =
? (BaseJdbcLogger.java:137)
DEBUG 08-01 16:57:44,167 ==> Parameters: 1(Integer)
(BaseJdbcLogger.java:137)
DEBUG 08-01 16:57:44,188 <==      Total: 1 (BaseJdbcLogger.java:137)
Emp{empId=1, empName='张三', age=20, gender='男'}
Emp{empId=1, empName='张三', age=20, gender='男'}
```

由结果可以看出，查询出两条数据时，只执行了一次sql语句，这是因为第一条数据是从数据库查询出来的，而第二条数据是从一级缓存中查询出来的。

当出现以下四种情况的时候，一级缓存将会失效：

1. 不同的SqlSession对应不用的一级缓存
2. 同一个SqlSession但是查询条件不同
3. 同一个SqlSession两次查询期间执行了一次增删改操作
4. 同一个SqlSession两次查询期间手动清空了缓存

   使用`sqlSession1.clearCache();`方法可以清空SqlSession中的缓存。

11.2、MyBatis的二级缓存

二级缓存是SqlSessionFactory级别的，通过同一个SqlSessionFactory创建的SqlSession查询的结果会被缓存，在通过同一个SqlSessionFactory所获取的SqlSession查询相同的数据会从缓存中获取。

二级缓存开启的条件：

1. 在核心配置文件中，设置全局配置属性`cacheEnabled="true"`，默认为true，不需要设置。
2. 在相对应的映射文件中设置标签`<cache/>`。

   ```
   <!--在CacheMapper映射文件中设置-->
   <cache/>
   ```

3. 二级缓存必须在SqlSession关闭或提交之后有效，即需要执行`sqlSession.close()`方法。使用同一个SqlSessionFactory创建的SqlSession的测试二级缓存的方法如下：

   ```
   @Test
   public void testCache() throws IOException {
       InputStream is = Resources.getResourceAsStream("mybatis-
   config.xml");
       SqlSessionFactory sqlSessionFactory = new
   SqlSessionFactoryBuilder().build(is);
       SqlSession sqlSession1 = sqlSessionFactory.openSession(true);
       CacheMapper mapper1 = sqlSession1.getMapper(CacheMapper.class);
       Emp emp1 = mapper1.getEmpById(1);
       System.out.println(emp1);
       sqlSession1.close();
       SqlSession sqlSession2 = sqlSessionFactory.openSession(true);
       CacheMapper mapper2 = sqlSession2.getMapper(CacheMapper.class);
       Emp emp2 = mapper2.getEmpById(1);
       System.out.println(emp2);
       sqlSession2.close();
   }
   ```

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

```
public class Emp implements Serializable {
    ....
}
```

测试输出结果如下：

```
DEBUG 08-01 18:36:07,337 Cache Hit Ratio
[com.tianna.mybatis.mapper.CacheMapper]: 0.0 (LoggingCache.java:60)
DEBUG 08-01 18:36:07,781 ==>  Preparing: select * from t_emp where emp_id =
? (BaseJdbcLogger.java:137)
DEBUG 08-01 18:36:07,806 ==> Parameters: 1(Integer)
(BaseJdbcLogger.java:137)
DEBUG 08-01 18:36:07,828 <==      Total: 1 (BaseJdbcLogger.java:137)
Emp{empId=1, empName='张三', age=20, gender='男'}
WARN  08-01 18:36:07,841 As you are using functionality that deserializes
object streams, it is recommended to define the JEP-290 serial filter.
Please refer to https://docs.oracle.com/pls/topic/lookup?
ctx=javase15&id=GUID-8296D8E8-2B93-4B9A-856E-0A65AF9B8C66
(SerialFilterChecker.java:46)
DEBUG 08-01 18:36:07,844 Cache Hit Ratio
[com.tianna.mybatis.mapper.CacheMapper]: 0.5 (LoggingCache.java:60)
Emp{empId=1, empName='张三', age=20, gender='男'}
```

可以看出只执行了一次SQL语句，第一次查询是从数据库中查询的，第二次的查询输出是从二级缓存中查询的。输出的日志中的`Cache Hit Ratio`为缓存命中率，只要其值不为零，就说明从缓存中查询数据成功。

使二级缓存失效的情况：

两次查询之间执行了任意的增删改，会使一级和二级缓存同时失效。

11.3、二级缓存的相关配置

在mapper配置文件中添加的cache标签可以设置一些属性：

1. eviction属性：缓存回收策略，默认是LRU

   LRU(Least Recently Used)：最近最少使用的，移除最长时间不被只用的对象

   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地对象

2. flushIntervals属性：刷新间隔，单位毫秒

   默认情况下是不设置，也就是没有刷新间隔，缓存仅仅调用语句时刷新（执行增删改）。

3. 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，太大容易导致内存一出

4. readOnly属性：只读，true/fasle

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），因此可以对其进行修改。这会慢一些，但是安全。因此默认是`false`。

例如缓存得到一个emp对象，当设置true时，就不能对emp对象进行修改，设置为false时就可以对其进行修改。

11.4、MyBatis缓存查询的顺序

先查询二级缓存，然后再查询一级缓存。

如果二级缓存没有命中，在查询一级缓存

如果一级缓存也没有命中，则查询数据库

SqlSession关闭之后，一级缓存中的数据会写入二级缓存。

11.5、整合第三方缓存EHCache(针对二级缓存)

11.5.1、添加依赖

在pom.xml文件中添加如下依赖：

```
<!--Mybatis EHCache整合包-->
<dependency>
    <groupId>org.mybatis.caches</groupId>
    <artifactId>mybatis-ehcache</artifactId>
    <version>1.2.1</version>
</dependency>
<!--slf4j日志门面的一个具体实现-->
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>1.2.3</version>
</dependency>
```

11.5.2、各jar包的功能

jar包名称	作用
mybatis-ehcache	Mybatis和EHCacha的整合包
ehcache	EHCache核心包
slf4j-api	SLF4J日志门面包
logback-classic	支持SLF4J门面接口的一个具体实现

11.5.3、创建EHCache的配置文件ehcache.xml

在resources目录下创建，文件名必须为ehcache.xml，`diskStore`标签下为缓存保存的本地路径，根据需要自行修改。

```
<?xml version="1.0" encoding="utf-8" ?>
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="../config/ehcache.xsd">
    <!-- 磁盘保存路径 -->
    <diskStore path="F:\mybatis\cache"/>
    <defaultCache
            maxElementsInMemory="1000"
            maxElementsOnDisk="10000000"
            eternal="false"
            overflowToDisk="true"
            timeToIdleSeconds="120"
            timeToLiveSeconds="120"
            diskExpiryThreadIntervalSeconds="120"
            memoryStoreEvictionPolicy="LRU">
    </defaultCache>
</ehcache>
```

11.5.4、设置二级缓存的类型

在mapper映射文件的cache标签下指定缓存的类型：

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

11.5.5、加入logback日志

存在SLF4j时，作为简易日志的log4j将失效，此时我们需要借助SLF4j的具体实现logback来打印日志。 创建logback的配置文件名必须为logback.xml，创建目录为resources，内容如下:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration debug="true">
    <!-- 指定日志输出的位置 -->
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <!-- 日志输出的格式 -->
            <!-- 按照顺序分别是： 时间、日志级别、线程名称、打印日志的类、日志主体内容、换行
            -->
            <pattern>[%d{HH:mm:ss.SSS}] [%-5level] [%thread] [%logger]
                [%msg]%n</pattern>
        </encoder>
    </appender>
    <!-- 设置全局日志级别。日志级别按顺序分别是： DEBUG、INFO、WARN、ERROR -->
    <!-- 指定任何一个日志级别都只打印当前级别和后面级别的日志。 -->
    <root level="DEBUG">
```

```
        <!-- 指定打印日志的appender，这里通过“STDOUT”引用了前面配置的appender
-->
        <appender-ref ref="STDOUT" />
    </root>
    <!-- 根据特殊需求指定局部日志级别 -->
    <logger name="com.tianna.mybatis.mapper" level="DEBUG"/>
</configuration>
```

logger标签的内容根据自己的路径进行设置，我这里是`com.tianna.mybatis.mapper`。

然后就可以使用第三方缓存了。。。。

12、MyBatis的逆向工程

正向工程：先创建Java实体类，由框架负责根据实体类生成数据表。Hibernate是支持正向工程的。

逆向工程：：先创建数据库表，由框架负责根据数据库表，反向生成如下资源：

- Java实体类
- Mapper接口
- Mapper映射文件

12.1、创建逆向工程的步骤

12.1.1、在pom.xml文件中添加依赖和插件

```
<packaging>jar</packaging>

<dependencies>
    <!--MyBatis核心-->
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.5.7</version>
    </dependency>
    <!--junit测试-->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>
    <!--MySQL驱动-->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.29</version>
    </dependency>
    <!--log4j日志-->
    <dependency>
        <groupId>log4j</groupId>
```

```
        <artifactId>log4j</artifactId>
        <version>1.2.17</version>
    </dependency>
</dependencies>
<!-- 控制Maven在构建过程中相关配置 -->
<build>
    <!-- 构建过程中用到的插件 -->
    <plugins>
        <!-- 具体插件，逆向工程的操作是以构建过程中插件形式出现的 -->
        <plugin>
            <groupId>org.mybatis.generator</groupId>
            <artifactId>mybatis-generator-maven-plugin</artifactId>
            <version>1.3.0</version>
            <!--插件的依赖-->
            <dependencies>
                <!--逆向工程的核心依赖-->
                <dependency>
                    <groupId>org.mybatis.generator</groupId>
                    <artifactId>mybatis-generator-core</artifactId>
                    <version>1.3.2</version>
                </dependency>
                <!--MySQL驱动-->
                <dependency>
                    <groupId>mysql</groupId>
                    <artifactId>mysql-connector-java</artifactId>
                    <version>8.0.29</version>
                </dependency>
            </dependencies>
        </plugin>
    </plugins>
</build>
```

12.1.2、创建MyBatis的核心配置文件

就使用之前的模板进行创建就好，文件名为`mybatis-config.xml`，内容如下：

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

    <!--开启驼峰命名自动映射-->
    <settings>
        <setting name="mapUnderscoreToCamelCase" value="true"/>
    </settings>

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

```
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <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MyBatis映射文件-->
    <mappers>
        <package name=""/>
    </mappers>
</configuration>
```

12.1.3、创建逆向工程的配置文件

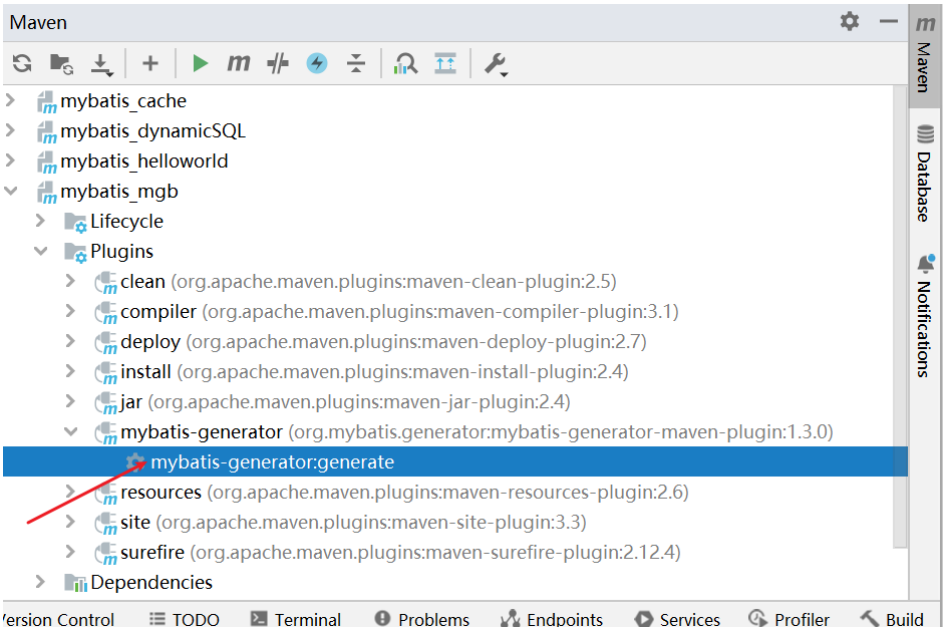
文件名必须是：generatorConfig.xml，内容如下所示：

```
<?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>
    <!--
    targetRuntime: 执行生成的逆向工程的版本
    MyBatis3Simple: 生成基本的CRUD，基本增删改查(单个，多个)五个方法（清新简洁
版）
    MyBatis3: 生成带条件的CRUD（奢华尊享版）
    -->
    <context id="DB2Tables" targetRuntime="MyBatis3Simple">
        <!-- 数据库的连接信息 -->
        <jdbcConnection driverClass="com.mysql.cj.jdbc.Driver"
                        connectionURL="jdbc:mysql://localhost:3306/ssm?
serverTimezone=UTC"
                        userId="root"
                        password="123456">
        </jdbcConnection>
        <!--targetPackage：根据自己的需要更改-->
        <!--enableSubPackages：是否使用子包，用.命名可以分成一层一层的目录-->
        <!-- javaBean的生成策略-->
        <javaModelGenerator targetPackage="com.tianna.mybatis.pojo"
```

```
                            targetProject=".\src\main\java">
            <property name="enableSubPackages" value="true" />
            <property name="trimStrings" value="true" />
        </javaModelGenerator>
        <!-- SQL映射文件的生成策略 -->
        <sqlMapGenerator targetPackage="com.tianna.mybatis.mapper"
                         targetProject=".\src\main\resources">
            <property name="enableSubPackages" value="true" />
        </sqlMapGenerator>
        <!-- Mapper接口的生成策略 -->
        <javaClientGenerator type="XMLMAPPER"
                             targetPackage="com.tianna.mybatis.mapper"
targetProject=".\src\main\java">
            <property name="enableSubPackages" value="true" />
        </javaClientGenerator>
        <!-- 逆向分析的表 -->
        <!-- tableName 为表中的字段名,设置为*号，可以对应所有表，此时不写
domainObjectName -->
        <!-- domainObjectName 实体类的名字 ，属性指定生成出来的实体类的类名 --
>
        <table tableName="t_emp" domainObjectName="Emp"/>
        <table tableName="t_dept" domainObjectName="Dept"/>
    </context>
</generatorConfiguration>
```

12.1.4、执行MBG插件的generate目标

双击如下图所示的位置，执行逆向工程插件的generate目标。

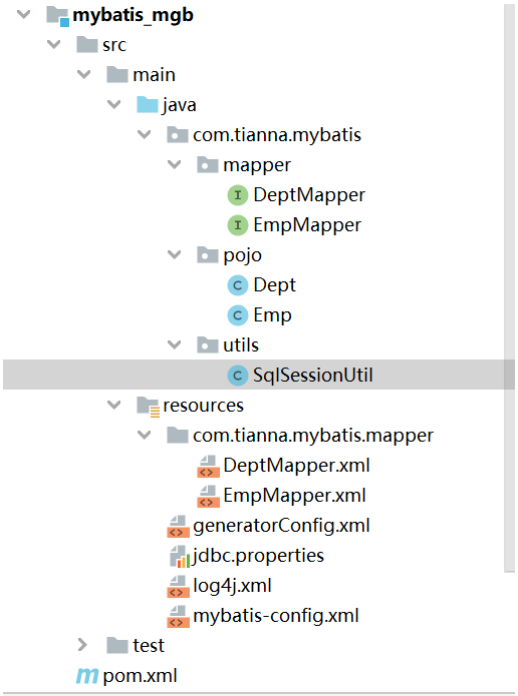


12.1.5、执行效果

将其他配置文件例如`jdbc.properties`和`log4j.xml`添加到项目中，然后再核心配之文件中修改起别名的实体类的包以及映射文件所对应的包，并将创建SqlSession的工具类放到项目中。

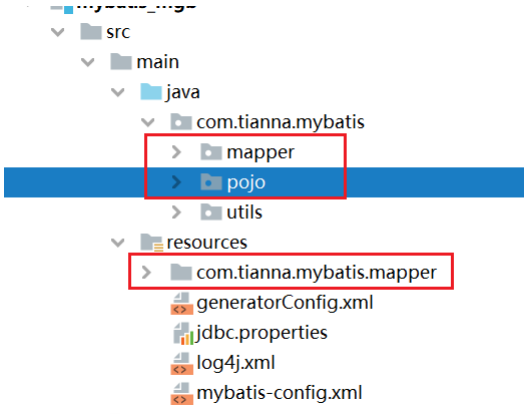
```
<typeAliases>
    <package name="com.tianna.mybatis.pojo"/>
</typeAliases>

<mappers>
    <package name="com.tianna.mybatis.mapper"/>
</mappers>
```

最终生成的内容如下：



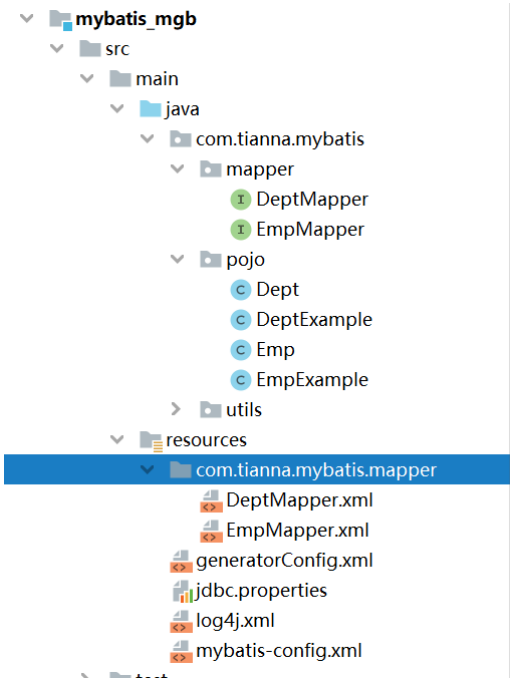
12.1.6、生成奢华尊享版

奢华尊享版为带条件的CRUD，由于上面生成了简单的CRUD，因此需要先删除生成的Java实体类、Mapper接口、Mapper映射文件，如下图所示的内容：



然后修改逆向工程的配置文件`generatorConfig.xml`中`targetRuntime`的值为`MyBatis3`。

然后执行MBG插件的generate目标，最终生成的内容如下：



12.2、QCB查询

由于自动生成的实体类没有构造方法个toString方法，因此首先为实体类添加有参构造方法、无参构造方法和toString方法。

1、根据主键查询，使用`selectByPrimaryKey`方法，代码如下

```
public void testMBG(){
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);
    1、根据主键查询
    Emp emp = mapper.selectByPrimaryKey(1);
}
```

2、查询所有的数据，使用`selectByExample`方法，其参数为nll。

```
public void testMBG(){
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);
    //2、查询所有数据
    List<Emp> emps = mapper.selectByExample(null);
    emps.forEach(System.out::println);
}
```

3、根据条件查询数据，使用`selectByExample`方法，需要创建一个`EmpExample`实例，用来构建我们查询的条件。多种条件之间可以通过`and`或`or`来连接。

```
public void testMBG(){
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);
    //3、根据条件查询，
    EmpExample example = new EmpExample();
    //以and连接
    example.createCriteria().andEmpNameEqualTo("张
三").andAgeGreaterThan(18);
    //以or连接
    example.or().andGenderEqualTo("男");
    List<Emp> emps = mapper.selectByExample(example);
    emps.forEach(System.out::println);
}
```

4、修改功能有两种（添加功能与其类似）：普通修改和选择型修改。

普通修改时，当修改的数据的值为null时，修改后的数据的值变为null。代码如下：

```
@Test
public void testMBG(){
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);
    Emp emp = new Emp(1,"小黑",null,"女",null);
    mapper.updateByPrimaryKey(emp);
}
```

```
DEBUG 08-03 17:12:14,912 ==>  Preparing: update t_emp set emp_name = ?, age
= ?, gender = ?, dept_id = ? where emp_id = ? (BaseJdbcLogger.java:137)
DEBUG 08-03 17:12:14,933 ==> Parameters: 小黑(String), null, 女(String),
null, 1(Integer) (BaseJdbcLogger.java:137)
DEBUG 08-03 17:12:14,952 <==    Updates: 1 (BaseJdbcLogger.java:137)
```

数据库修改结果如下：

emp_id	emp_name	age	gender	dept_id
1	小黑	(Null)	女	(Null)

选择性修改时，当修改的数据的值为null时，所对应的数据库的值不会被修改。代码如下：

```
@Test
public void testMBG(){
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);
    Emp emp = new Emp(1,"小黑",null,"女",null);
    mapper.updateByPrimaryKeySelective(emp);
}
```

```
DEBUG 08-03 17:17:54,625 ==>  Preparing: update t_emp SET emp_name = ?,
gender = ? where emp_id = ? (BaseJdbcLogger.java:137)
DEBUG 08-03 17:17:54,646 ==> Parameters: 小黑(String), 女(String),
1(Integer) (BaseJdbcLogger.java:137)
DEBUG 08-03 17:17:54,662 <==    Updates: 1 (BaseJdbcLogger.java:137)
```

数据库修改结果如下：

emp_id	emp_name	age	gender	dept_id
1	小黑	19	女	1

12、分页插件

分页用的sql语句为：

```
limit index,pageSIze
```

分页常用的一些参数为：

pageSize：每页显示的条数

pageNum：当前页的页码

index：当前页的起始索引，index = (pageNum - 1) * pageSize

count：总记录数

totalPage：总页数，totalPage = count/pageSize ，当count % pageSize != 0时 totalPage += 1

12.1、分页插件的使用步骤

12.1.1、添加依赖

在pom.xml文件中添加如下依赖：

```
<dependency>
    <groupId>com.github.pagehelper</groupId>
    <artifactId>pagehelper</artifactId>
    <version>5.2.0</version>
</dependency>
```

12.1.2、配置分页插件

在MyBatis核心配置文件中配置插件，`plugins`标签需要放在`environments`之前。

```
<plugins>
    <!--设置分页插件-->
    <plugin interceptor="com.github.pagehelper.PageInterceptor"></plugin>
</plugins>
```

12.1.3、开启分页进行查询

在查询功能之前需要先开启分页功能。

测试代码如下：

```
@Test
public void testPage(){
    SqlSession sqlSession = SqlSessionUtil.getSqlSession();
    EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);
    //查询功能之前开启分页功能
    PageHelper.startPage(1,2);
    List<Emp> list = mapper.selectByExample(null);
    list.forEach(System.out::println);
}
```

12.2、分页插件的使用

在查询获取list集合之后，使用`PageInfo<T> pageInfo = new PageInfo<>(List<T> list,int navigatePages);`来获取分页相关数据：

- list：分页之后的数据
- navigatePages：导航分页的页码数

分页相关数据如下：

```
PageInfo{pageNum=1, pageSize=2, size=2, startRow=1, endRow=2, total=6,
pages=3, list=Page{count=true, pageNum=1, pageSize=2, startRow=0, endRow=2,
total=6, pages=3, reasonable=false, pageSizeZero=false}[Emp{empId=1,
empName='小黑', age=19, gender='女', deptId=1}, Emp{empId=2, empName='李
四', age=21, gender='男', deptId=2}], prePage=0, nextPage=2,
isFirstPage=true, isLastPage=false, hasPreviousPage=false,
hasNextPage=true, navigatePages=2, navigateFirstPage=1, navigateLastPage=2,
navigatepageNums=[1, 2]}
```

> - pageNum：当前页的页码
> - pageSize：每页显示的条数
> - size：当前页显示的真实条数
> - total：总记录数
> - pages：总页数
> - prePage：上一页的页码
> - nextPage：下一页的页码
> - isFirstPage/isLastPage：是否为第一页/最后一页
> - hasPreviousPage/hasNextPage：是否存在上一页/下一页
> - navigatePages：导航分页的页码数
> - navigatepageNums：导航分页的页码，[1,2,3,4,5]