Mybatis

1、简介

1.1什么是Mybatis

- MyBatis 是一款优秀的**持久层框架**。
- 它支持自定义 SQL、存储过程以及高级映射。
- MyBatis 免除了几乎所有的 JDBC 代码以及设置参数和获取结果集的工作。
- MyBatis 可以通过简单的 XML 或注解来配置和映射原始类型、接口和 Java POJO（Plain Old Java Objects，普通老式 Java 对象）为数据库中的记录。

1.2 如何获得Mybatis

```xml
<!-- https://mvnrepository.com/artifact/org.mybatis/mybatis -->
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis</artifactId>
    <version>3.5.6</version>
</dependency>
```

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

1.3 持久化

数据持久化

- 持久化就是将程序的数据在持久状态和瞬时状态转化的过程
- 瞬时状态：内存：断电即失
- 持久状态：数据库（jdbc）、io文件持久化

为什么需要持久化

- 有一些对象，不能丢失它
- 内存太贵了

1.4 持久层

Dao 层、Service层、Controller层.....

- 完成持久化工作的代码块
- 层界十分明显

1.5 为什么需要Mybatis

- 帮助程序员将数据存放到数据库中
- 方便
- 传统的JDBC代码太复杂了，简化、框架、自动化
- 不用Mybatis也可以，更容易上手。技术没有高低之分
- 优点：
    - sql和代码分离

2、第一个Mybatis程序

思路：搭建环境--->导入Mybatis-->编写代码-->测试

2.1 搭建环境

搭建数据库

```
CREAT DATABASE mybatis
USE 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,'chenfei','123456'),
(2,'zhang','111111'),
(3,'wu','222222')
```

新建项目

1、新建一个普通的maven项目

2、删除src目 录

3、导入maven工程

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

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <!--父工程-->
    <groupId>org.example</groupId>
    <artifactId>Mybatis</artifactId>
    <version>1.0-SNAPSHOT</version>
    <!--子工程-->
    <!--导入依赖-->
    <dependencies>
        <!--mysql驱动-->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>8.0.21</version>
        </dependency>
        <!--mybatis-->
        <!-- https://mvnrepository.com/artifact/org.mybatis/mybatis -->
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis</artifactId>
            <version>3.5.5</version>
        </dependency>
        <!--junit-->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <scope>test</scope>
        </dependency>
    </dependencies>

</project>
```

2.2 创建一个模块

- 编写mybatis核心配置文件

```xml
<?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>
    <!--核心配置文件-->
    <environments default="development">
```

```
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.jdbc.Driver"/>
                <property name="url"
value="jdbc:mysql://localhost:3306/mybatis?useSSL = ture
&amp;useUnicode=true&amp;characterEncoding=UTF-8"/>
                <property name="username" value="root"/>
                <property name="password" value="cf19971101"/>
            </dataSource>
        </environment>
    </environments>
</configuration>
```

- 编写mybatis工具类

```
//构建 SqlSessionFactory-->SqlSession
public class MybatisUtils {
    private static SqlSessionFactory sqlSessionFactory;
    static {
        try {
            //使用Mybatis第一步
            //获取 SqlSessionFactory对象
            String resource = "org/mybatis/example/mybatis-config.xml";
            InputStream inputStream =
Resources.getResourceAsStream(resource);
            //提升作用域
            sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);
        }catch (IOException e) {
            e.printStackTrace();
        }
    }

    //既然有了 SqlSessionFactory，顾名思义，我们可以从中获得 SqlSession 的实例。
    // SqlSession 提供了在数据库执行 SQL 命令所需的所有方法

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

2.3 编写代码

- 实体类

```java
package com.chamfers.pojo;

//实体类
//POJO:“Plain Old Java Object”“简单java对象”。
// POJO的内在含义是指那些没有从任何类继承、也没有实现任何接口，更没有被其它框架侵入的java对象。
public class User {
    private int id;

    private String name;

    private String pwd;

    public User() {
    }

    public User(int id, String name, String pwd) {
        this.id = id;
        this.name = name;
        this.pwd = pwd;
    }

    public int getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getPwd() {
        return pwd;
    }

    public void setPwd(String pwd) {
        this.pwd = pwd;
    }
```

```
45
46      @Override
47      public String toString() {
48          return "User{" +
49                  "id=" + id +
50                  ", name='" + name + '\'' +
51                  ", pwd='" + pwd + '\'' +
52                  '}';
53      }
54  }
55
```

- Dao接口

```
1  //DAO = Data Access Object = 数据存取对象
2  public interface UserDao {
3      List<User> getUserList();
4  }
```

- 接口实现类

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <!DOCTYPE mapper
3          PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4          "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5
6  <!--namespace 绑定一个对应的Dao/Mapper接口-->
7  <mapper namespace="com.chamfers.dao.UserDao">
8      <select id="getUserList" resultType="com.chamfers.pojo.User">
9          select * from mybatis.user
10     </select>
11 </mapper>
```

2.4 测试

注意点

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

MapperRegistry是什么?

每一个Mappers.xml都需要在核心配置文件中进行注册

```
1  <!--每一个Mappers.xml都需要在核心配置文件中进行注册-->
2      <mappers>
3          <mapper resource="com/chamfers/dao/UserMapper.xml"/>
4      </mappers>
```

- junit测试

```java
public class UserDaoTest {
    @Test
    public void test(){
        //第一步：获取SqlSession对象
        SqlSession sqlSession = MybatisUtils.getSqlSession();
        //第二步：执行SQL
        ////方式1：getMapper()
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
        List<User> userList = mapper.getUserList();
        for (User user : userList) {
            System.out.println(user);
        }
        sqlSession.close();
    }
}
```

遇到的问题

- 1、配置文件没有注册
- 2、绑定接口错误
- 3、方法名不对
- 4、返回类型不对
- 5、Maven导出资源问题:

```
Caused by: org.apache.ibatis.exceptions.PersistenceException:
### Error building SqlSession.
### The error may exist in com/chamfers/dao/UserMapper.xml
### Cause: org.apache.ibatis.builder.BuilderException: Error parsing SQL Mapper Configuration. Cause: java.io.IOException: Could not find resource com/chamfers/dao/UserMapper.xml
```

```xml
<!--在build中配置resources，来防止我们资源导出失败的问题-->
    <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>
```

3、CRUD

3.1 namespace

namespace中的包名要和Dao/mapper接口包名一致！

3.2 select

选择，查询语句

- id：就是对应的namespace中的方法名；
- resultType：Sql语句执行的返回值！
- parameterType：方法中传入参数的类型！

1、编写接口

```
    //查询所有用户
    List<User> getUserList();
```

2、编写对应的mapper中的sql语句

```
<!--sql查询语句，id=UserDao中的接口方法，resulrType=接口方法的返回类型（<>泛型类型）-->
    <select id="getUserList" resultType="com.chamfers.pojo.User">
        select * from mybatis.user
    </select>
```

3、测试

```
public class UserDaoTest {
    @Test
    public void selectUser(){
        //第一步：获取SqlSession对象
```

```
        SqlSession sqlSession = MybatisUtils.getSqlSession();
        //第二步：执行SQL
        ////方式1：getMapper()
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
        List<User> userList = mapper.getUserList();
        for (User user : userList) {
            System.out.println(user);
        }
        sqlSession.close();
    }
}
```

3.3 Insert

```
    <!--增加用户-->
    <insert id="addUser" parameterType="com.chamfers.pojo.User">
        insert into mybatis.user (id,name,pwd) values (#{id},#{name},#{pwd});
    </insert>
```

3.4 Update

```
    <!--修改一个用户-->
    <update id="updateUser" parameterType="com.chamfers.pojo.User" >
        update mybatis.user set name = #{name},pwd = #{pwd} where id = #{id};
    </update>
```

3.5 Delete

```
    <!--删除一个用户-->
    <delete id="deleteUserById" parameterType="int">
        delete from mybatis.user where id = #{id};
    </delete>
```

- 注意点：增删改需要提交事物

  session.commit()

3.6 错误

- 标签不要匹配错误
- resource绑定mapper时，需要使用路径!(要使用“/”：com/chamfers/dao/UserMapper.xml)
- 配置文件必须符合要求
- 空指针异常
- 输出的xml文件中存在中文乱码

- maven资源没有导出问题

3.7 万能Map

如果，我们的实体类，或者数据库中规定表，字段或者参数过多，我们应当考虑使用Map！

```
<insert id="addUser2" parameterType="map">
  insert into mybatis.user (id,pwd) values (#{userId},#{passWord});
</insert>
```

Map传递参数，直接在sql中取出key即可！【parameterType-"map"】

对象传递参数，直接在sql中取对象的属性即可！【parameterType-"Object"】

只有一个基本类型参数的情况下，可以直接在sql中取到！

多个参数用Map，**或者用注解**！

3.8模糊查询

- Java代码执行的时候，传递通配符% %

```
List <User> userList = mapper.getUserLike("%李%");
```

- 在sql拼接中使用通配符

```
select * from mybatis.user where name like "%"#{value}"%";
```

4、配置解析

4.1、核心配置文件

- mybatis-config.xml
- MyBatis 的配置文件包含了会深深影响 MyBatis 行为的设置和属性信息。

```
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）
```

4.2、环境配置（environments）

- MyBatis 可以配置成适应多种环境
- 学会使用配置多套运行环境！
- **不过要记住：尽管可以配置多个环境，但每个 SqlSessionFactory 实例只能选择一种环境。**
- mybatis默认的食物管理器时JDBC，连接池：POOLED

4.3、属性（properties）

我们可以通过属性（properties）来实现引用配置文件！！

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

编写一个db.properties配置文件

```
driver = com.mysql.cj.jdbc.Driver
url= jdbc:mysql://localhost:3306/mybatis?useSSL =
TRUE&useUnicode=TRUE&characterEncoding=UTF-8
username=root
password=cf19971101
```

在核心配置文件中引入

```
<properties resource="db.properties">
<!--或者-->
<properties resource="org/mybatis/example/config.properties">
  <property name="username" value="dev_user"/>
  <property name="password" value="F2Fa3!33TYyg"/>
</properties>
```

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

- 可以直接引入外部文件
- 可以在其中增加一些属性配置
- 如果两个文件有相同字段，会发生**覆盖**

如果一个属性在不只一个地方进行了配置，那么，MyBatis 将按照下面的顺序来加载：

- 首先读取在 properties 元素体内指定的属性。
- 然后根据 properties 元素中的 resource 属性读取类**路径下属性文件**，或根**据 url 属性指定的路径**读取属性文件，并**覆盖**之前读取过的同名属性。
- 最后读取作为方法参数传递的属性，并覆盖之前读取过的同名属性。

4.4、类型别名（typeAliases）

- 类型别名可为 Java 类型设置一个缩写名字
- 意在降低冗余的全限定类名书写

```
<!--可以给实体类或包中的实体类起别名-->
    <typeAliases>
        <!--用单个类的形式起别名-->
        <typeAlias type="com.chamfers.pojo.User" alias="User"/>
    </typeAliases>
```

也可以指定一个包名，MyBatis 会在包名下面搜索需要的 Java Bean

扫描实体类的包，它的默认别名就为这个类的 类名首字母小写

```
<!--可以给实体类或包中的实体类起别名-->
    <typeAliases>
        <!--用包形式起别名 别名为类名首字母小写 e.g. domain.blog.Author 的别名为 author-->
        <package name="com.chamfers.pojo"/>
    </typeAliases>
```

在实体类较少的时候，使用第一种方式

如果实体类十分多，建议使用第二种方式

第一种方式可以DIY别名，第二种则不行，如果非要改，需要在实体类上增加@Alias("alias")注解

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

4.5、设置（settings）

这是 MyBatis 中极为重要的调整设置，它们会改变 MyBatis 的运行时行为。

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

4.6、其他配置

- typeHandlers（类型处理器）
- objectFactory（对象工厂）
- plugins（插件）

4.7、映射器（mappers）

MapperRegistry:注册绑定我们的Mapper文件；

方式一：使用相对于类路径的资源引用

```
<!-- 使用相对于类路径的资源引用 -->
<mappers>
  <mapper resource="org/mybatis/builder/AuthorMapper.xml"/>
  <mapper resource="org/mybatis/builder/BlogMapper.xml"/>
  <mapper resource="org/mybatis/builder/PostMapper.xml"/>
</mappers>

<!--每一个mybatis.xml都需要在核心配置文件中进行注册-->
<mappers>
  <mapper resource="com/chamfers/dao/UserMapper.xml"/>
</mappers>
```

方式二：使用class文件绑定注册

```xml
<!-- 使用映射器接口实现类的完全限定类名 -->
<mappers>
  <mapper class="org.mybatis.builder.AuthorMapper"/>
  <mapper class="org.mybatis.builder.BlogMapper"/>
  <mapper class="org.mybatis.builder.PostMapper"/>
</mappers>

<!--每一个mybatis.xml都需要在核心配置文件中进行注册-->
<mappers>
    <!-- <mapper resource="com/chamfers/dao/UserMapper.xml"/>-->
    <mapper class="com.chamfers.dao.UserMapper"/>
</mappers>
```

注意点：

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

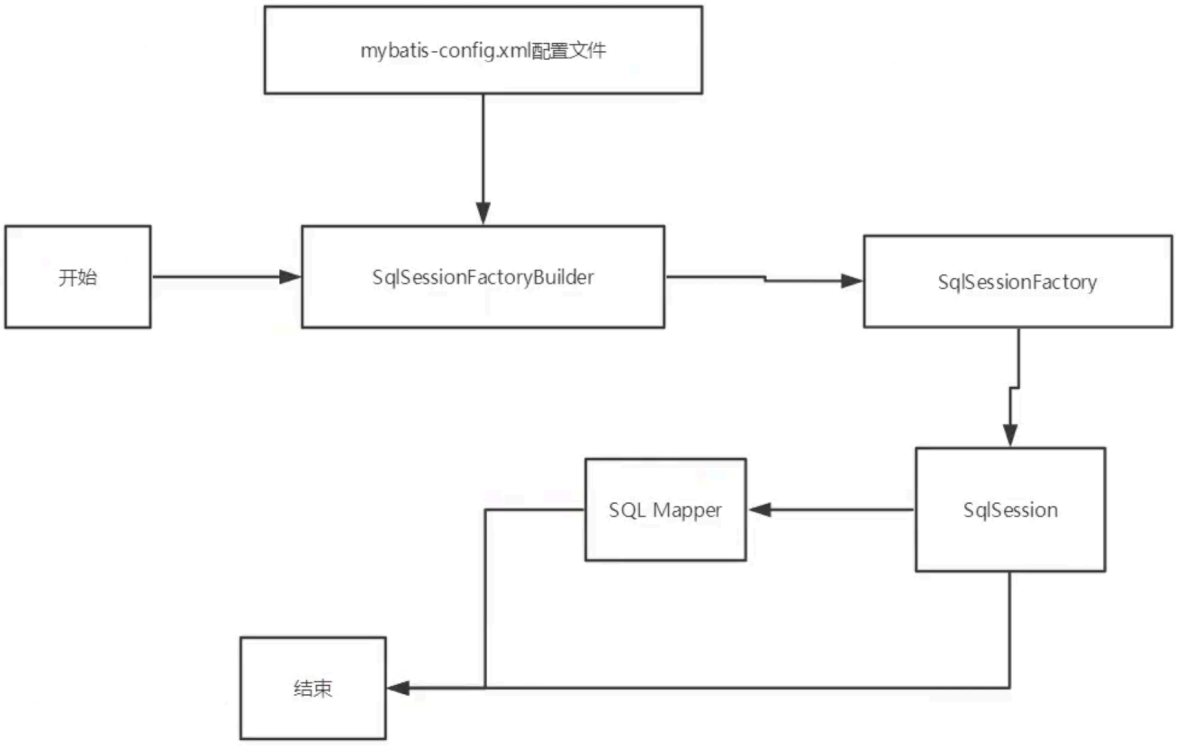
方式三：使用扫描包进行注入绑定

```xml
<!-- 将包内的映射器接口实现全部注册为映射器 -->
<mappers>
  <package name="org.mybatis.builder"/>
</mappers>
```

注意点：

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

4.8、生命周期和作用域



生命周期和作用域是至关重要的，因为错误的使用会导致非常严重的**并发问题**

SqlSessionFactoryBuilder:

- 一旦创建了SqlSessionFactory，就不再需要它了
- 局部变量

SqlSessionFactory

- 说白了就是可以想象为：数据库连接池
- SqlSessionFactory一旦被创建就应该在应用的运行期间一直存在，**没有任何理由丢弃或重新创建另一个实例。**
- 因此，SqlSessionFactory的最佳作用域是应用作用域
- 最简单的就是使用**单例模式**或者**静态单例模式**。

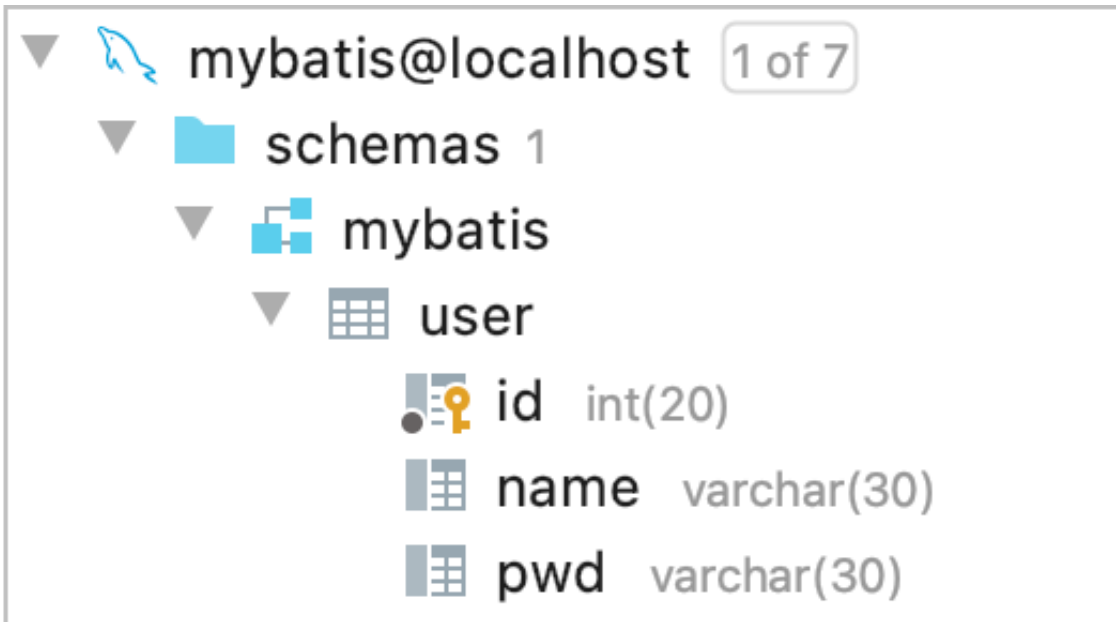
SqlSession：

- 连接到数据库连接池的一个请求
- SqlSession的实例不是线程安全的，因此是不能被共享的，所以他的最佳作用域是请求或方法作用域
- 用完之后需要赶紧关闭，否则资源被占用！

这里面的每一个Mapper就代表一个业务！

5、解决字段名和属性名不一致的问题

数据库中的字段



新建一个项目，拷贝之前的，测试实体类字段不一致的情况

```
public class User {
    private int id;

    private String name;

    private String password;
}
```

问题：password为空null

原因：类型处理器处理后，字段名和实体了属性名不一致

```
//select * from mybatis.user where id = #{id}
//类型处理器
//select id,name,pwd from mybatis.user where id = #{id}
```

解决方法：

1、起别名

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

2、Result Map 结果映射

结果集映射

```
id  name  pwd  //数据库中的字段名
id  name  password  //实体类中的属性名
//不一致！！
```

```
<!--创建结果集映射resultMap-->
    <resultMap id="UserMap" type="User">
        <!--column：数据库中的字段，property：实体类中的属性-->
        <result column="id" property="id"/>
        <result column="name" property="name"/>
        <result column="pwd" property="password"/>
    </resultMap>

    <!--按照id查询user-->
    <select id="getUserById" parameterType="int" resultMap="UserMap">
        select * from mybatis.user where id = #{id}
    </select>
```

- `resultMap` 元素是 MyBatis 中最重要最强大的元素。
- ResultMap 的设计思想是，对简单的语句做到零配置，对于复杂一点的语句，只需要描述语句之间的关系就行了。
- ResultMap最优秀的地方在于，虽然你已经对他相当了解了，但是根本就不需要显式地调用到他们。

6、日志

6.1、日志工厂

如果一个数据库操作出现了异常，我们需要排错，日志就是很好的帮手

曾经：sout、debug

现在：日志工厂

logImpl	指定 MyBatis 所用日志的具体实现，未指定时将自动查找。	SLF4J | LOG4J | LOG4J2 | JDK_LOGGING | COMMONS_LOGGING | STDOUT_LOGGING | NO_LOGGING	未设置

- SLF4J
- LOG4J 【掌握】
- LOG4J2
- JDK_LOGGING
- COMMONS_LOGGING
- STDOUT_LOGGING 【掌握】
- NO_LOGGING

在Mybatis中具体使用哪个日志实现，在设置中设定！

STDOUT_LOGGING 标准日志输出

在mybatis-config.xml核心配置文件中，配置我们的日志！

6.2、LOG4J

- 可以控制日志信息输送的目的地是控制台、文件、GUI组件
- 可以控制每一条日志的输出格式
- 定义每一条日志信息的级别
- 可以通过一个配置文件来灵活地进行配置

步骤：

- 先导入LOG4J包

```
<dependencies>
<!--https://mvnrepository.com/artifact/log4j/log4j -->
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
</dependency>
</dependencies>
```

- log4j.properties

```
### 配置根 ###
log4j.rootLogger = debug,console ,file

### 设置输出sql的级别，其中logger后面的内容全部为jar包中所包含的包名 ###
log4j.logger.org.mybatis=dubug
log4j.logger.java.sql.Connection=dubug
log4j.logger.java.sql.Statement=dubug
log4j.logger.java.sql.PreparedStatement=dubug
log4j.logger.java.sql.ResultSet=dubug

### 配置输出到控制台 ###
log4j.appender.console = org.apache.log4j.ConsoleAppender
log4j.appender.console.Target = System.out
log4j.appender.console.layout = org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern =  %d{ABSOLUTE} %5p %c{ 1
}:%L - %m%n

### 配置输出到文件 ###
log4j.appender.fileAppender = org.apache.log4j.FileAppender
```

```
log4j.appender.fileAppender.File = logs/log.log
log4j.appender.fileAppender.Append = true
log4j.appender.fileAppender.Threshold = DEBUG
log4j.appender.fileAppender.MaxFileSize=10KB
log4j.appender.fileAppender.layout = org.apache.log4j.PatternLayout
log4j.appender.fileAppender.layout.ConversionPattern = %-d{yyyy-MM-dd HH:mm:ss}  [ %t:%r ] - [ %p ]  %m%n
```

- 配置log4j为日志的实现

```
static Logger logger = Logger.getLogger(UserDaoTest.class);
```

- Log4j的使用！直接测试运行刚才的查询

简单使用

1、在要使用Log4j的类中，导入包 import org.apache.log4j.Logger;

2、日志对象，参数为当前类的class

```
static Logger logger = Logger.getLogger(UserDaoTest.class);
```

3、日志级别

```
logger.info("info:进入testLog4j");
logger.debug("debug:进入testLog4j");
logger.error("error:进入testLog4j");
```

7、分页

思考：为什么要分页?

- 减少数据的处理量

7.1、使用Limit分页

```
select * from user limit startIndex[default=0],pageSize;
select * from user limit 3;#[0,3)
```

使用mybatis实现分页，核心sql

1、接口

```java
//实现分页查询
List<User> getUserListLimit(Map<String,Integer> map);
```

2、Mapper.xml

```xml
<!--分页查询-->
<select id="getUserListLimit" parameterType="map" resultType="User">
    select * from mybatis.user limit #{startIndex},#{pageSize}
</select>
```

3、测试

7.2、RowBounds分页

公司中不常用，了解就可以

8、使用注解开发

8.1、面向接口编程

- 大家之前都学过面向对象编程，也学习过接口，但在真正的开发中，很多时候我们会选择面向接口编程
- **根本原因: 解耦，可拓展， 提高复用， 分层开发中，上层不用管具体的实现，大家都遵守共同的标准， 使得开发变得容易,规范性更好**
- 在一个面向对象的系统中，系统的各种功能是由许许多多的不同对象协作完成的。在这种情况下，各个对象内部是如何实现自己的,对系统设计人员来讲就不那么重要了:
- 而各个对象之间的协作关系则成为系统设计的关键。小到不同类之间的通信，大到各模块之间的交互，在系统设计之初都是要着重考虑的，这也是系统设计的主要工作内容。面向接口编程就是指按照这种思想来编程。

关于接口的理解

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

三个面向区别

- 面向对象是指，我们考虑问题时，以对象为单位，考虑它的属性及方法.
- 面向过程是指，我们考虑问题时，以一个具体的流程(事务过程)为单位，考虑它的实现.

- 接口设计与非接口设计是针对复用技术而言的，与面向对象(过程)不是一个问题更多的体现就是对系统整体的架构

8.2、使用注解开发

1、注解在接口上实现

2、需要在核心配置文件中绑定接口

3、测试

本质：反射机制实现

底层：动态代理！ （动态代理、静态代理、工厂模式、单例模式.....）

Mybatis详细执行流程！

8.3、用注解实现CRUD

可以在工具类创建的时候实现自动提交事务

```
 public static SqlSession getSqlSession(){
   //openSession(true) 设置为true的时候 就为自动提交事务
        SqlSession sqlSession = sqlSessionFactory.openSession(true);
        return sqlSession;
    }
```

编写接口，增加注解

```
 //实现分页查询
@Select("select * from mybatis.user limit #{startIndex},#{pageSize}")
List<User> getUserListLimit(Map<String, Integer> map);
//方法存在多个参数，所有的参数前面必须加上@Param()注解
@Select("select * from mybatis.user where id = #{id}")
User getUserById(@Param("id") int id);
```

测试类

【注意：我们必须要将接口注册绑定到我们的核心配置文件中！！】

关于@Param()注解

- 基本类型的参数或者String类型，需要加上@Param()注解
- 引用类型不需要加
- 如果只有一个基本类型的话可以忽略，但是建议加上@Param()注解
- 在SQl中引用的就是@Param()中设定的属性名亦或称为**别名**

9、Lombok

- Java library
- plugs
- Build tools
- With one annotation your class

使用步骤：

1、在idea中安装Lombok插件！

2、在项目中导入Lombok的jar包

```
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <version>1.18.16</version>
</dependency>
```

3、在实体类上加入注解

@Data：无参构造、get、set、toString、hashcode、equals

```
@Getter and @Setter
@ToString
@EqualsAndHashCode
@AllArgsConstructor, @RequiredArgsConstructor and @NoArgsConstructor
@Data
@Accessors
```

10、多对一处理

- 多个学生对应一个老师

```
create table 'teacher'(
  'id' int(10) not null,
  'name' varchar(30) default null,
  primary key ('id')
) engine=innodb default charset=utf8
```

```
6
7  insert into teacher(id,name) VALUES (1,'陈老师');
8
9  create table student(
10   id int(10) not null,
11   name varchar(30) default null,
12   tid int(10) default null,
13   primary key(id),
14   key fktid(tid),
15   constraint fktid foreign key(tid) references teacher(id)
16 ) engine=innodb default charset=utf8
17
18 insert into student(id,name,tid) values (1,'小明', 1); insert into
   student(id,name,tid) values (2,'小张', 1); insert into student(id,name,tid)
   values (3,'小红', 1); insert into student(id,name,tid) values (4,'小李', 1);
   insert into student(id,name,tid) values (5,'小王', 1);
```

测试环境搭建

1、导入lombok

2、新建实体类Teacher、Student

3、建立Mapper接口

4、建立Mapper.xml文件

5、在核心配置文件中绑定注册我们的Mapper接口或者文件！【方式很多，随心选】

6、测试查询是否成功！

```
1  @Data
2  public class Student {
3      private int id;
4      private String name;
5
6      //tid关联到老师的实体
7      private Teacher teacher;
8  }
```

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

按照查询嵌套处理

```
<mapper namespace="com.chamfer.dao.StudentMapper">
<!--查询所有学生的信息，以及对应老师的信息
思路:
1、查询所有学生的信息
2、根据查询出来的学生tid信息，寻找对应老师的信息！嵌套子查询！！
-->
    <select id="getStudent" resultMap="StudentTeacher">
        select * from student
    </select>
    <resultMap id="StudentTeacher" type="Student">
        <result property="id" column="id"/>
        <result property="name" column="name"/>
        <!--复杂的属性，我们需要单独处理，对象：association；集合：collection-->
        <association property="teacher" column="tid" javaType="Student"
select="getTeacher"/>
    </resultMap>
    <select id="getTeacher" resultType="Teacher">
                select * from teacher where id = #{id}
    </select>
</mapper>
```

按照结果嵌套处理

```
<select id="getStudent2" resultMap="StudentTeacher2">
    select s.id sid,s.name sname,t.name tname
    from student s,teacher t
    where s.tid = t.id;
</select>
<resultMap id="StudentTeacher2" type="Student">
    <result property="id" column="sid"/>
    <result property="name" column="sname"/>
    <!--复杂的属性，我们需要单独处理，对象：association；集合：collection-->
    <association property="teacher" javaType="Teacher">
        <result property="name" column="tname"/>
    </association>
</resultMap>
```

回顾Mysql多对一查询方式：

- 子查询
- 连表查询

11、一对多

比如：一个老师拥有多个学生！

对于老师而言，就是一对多的关系

按照结果嵌套处理

```
<!--namespace 绑定一个对应的Dao/Mapper接口-->
<mapper namespace="com.chamfer.dao.TeacherMapper07">
    <select id="getTeacherById" resultMap="TeacherStudent">
        select t.id tid,t.name tname,s.id sid,s.name sname from teacher t,student s where t.id = s.tid and t.id = #{tid}
    </select>
    <resultMap id="TeacherStudent" type="Teacher">
        <result property="id" column="tid"/>
        <result property="name" column="tname"/>
        <!--
        复杂属性，我们需要单独处理，对象：association 集合：collection
        javaType=“”指定属性的类型
        ofType：集合中的泛型信息，我们使用ofType获取
        -->
        <collection property="students" ofType="Student">
            <result property="id" column="sid"/>
            <result property="name" column="sname"/>
            <result property="tid" column="tid"/>
        </collection>
    </resultMap>
</mapper>
```

按照查询嵌套处理

```
<select id="getTeacherById2" resultMap="TeacherStudent2">
    select * from mybatis.teacher
</select>
<resultMap id="TeacherStudent2" type="Teacher">
    <collection property="students" column="id" javaType="ArrayList" ofType="Student" select="getStudents"/>
</resultMap>
<select id="getStudents" resultType="Student">
    select * from mybatis.student where tid = #{id}
</select>
```

小结

1、关联：association【多对一】

2、集合：collection【一对多】

3、javaTpye：用来指定实体类中属性的类型

4、ofType：用来指定映射到List或者集合中的pojo类型，泛型中的约束类型

注意点：

- 保持SQL的可读性，尽量保证通俗易懂
- 注意一对多和多对一中，属性名和字段的问题！
- 如果问题不好排查错误，可以使用日志，建议使用log4j

12、动态SQL

什么是动态SQL：动态SQL就是指根据不同的条件生成不同的SQL语句

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

if
choose (when, otherwise)
trim (where, set)
foreach
```

搭建环境

```sql
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
```

1、导包

2、编写配置文件

3、编写实体类

```java
@Data
public class Blog {
    private String id;
    private String title;
    private String author;
    private Date createTime;
    private int views;
}
```

4、编写实体类对应Mapper接口和Mapper.xml文件

if

```xml
<select id="getBlogByIf" parameterType="map" resultType="Blog">
    select * from mybatis.blog where 1=1
    <if test="title != null">
        and title like #{title}
    </if>
</select>
```

choose,when,otherwise

```xml
<select id="getBlogByChoose" resultType="Blog" parameterType="map">
    select * from mybatis.blog
    <where>
        <choose>
            <when test="title != null">
                and title = #{title}
            </when>
            <when test="author != null">
                and author = #{author}
            </when>
            <otherwise>
                and views = 787
            </otherwise>
        </choose>
    </where>
</select>
```

tirm,where,set

```
<select id="getBlogByIf" parameterType="map" resultType="Blog">
    select * from mybatis.blog
  <where>
    <if test="title != null">
      title like #{title}
    </if>
    <if>
      and author = #{author}
    </if>
  </where>
</select>
<!--等价于-->
<trim prefix = "where" prefixOverrides="and|or">

</trim>
```

```
<update id="updateBlog" parameterType="map">
    update mybatis.blog
    <set>
        <if test="title != null">
            title = #{title},
        </if>
        <if test="author != null">
            author = #{author},
        </if>
    </set>
  <!--注意! where语句要放在<set></set>标签结束之后-->
    where id = #{bid}
</update>
<!--等价于-->
<trim prefix="set" suffixOverrides=",">
  ...
</trim>
```

所谓的动态SQL，本质还是SQL语句，指示我们可以子啊SQL层面去执行一些逻辑代码

SQL片段

有的时候，我们可能会将一些功能的部分抽取出来，方便复用。

1、使用sql标签 抽取出公共的部分

2、在需要使用的地方使用include标签，引用即可

注意事项：

- 最好基于单表来定义SQL标签
- 不要存在where标签

Foreach

```
<select id="getBlogByForeach" parameterType="map" resultType="Blog">
    select * from mybatis.blog
    <where>
        <foreach collection="ids" open="and id in (" item="id" close=")"
separator=",">
            #{id}
        </foreach>
    </where>
</select>
```

小结

动态SQL就是在拼接SQL语句，我们只要保证SQL的正确性，按照SQL的格式去排列组合就可以了！

建议：

- 现在Mysql中写出完整的SQL，再对应的去修改成为我们的动态SQL实现通用即可。