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 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
- 内存：**断电即失**
- 数据库（Jdbc）,io文件持久化。

为什么要持久化？

- 有一些对象，不能让他丢掉
- 内存太贵

1.3 持久层

Dao层、Service层、Controller层

- 完成持久化工作的代码块
- 层界限十分明显

1.4 为什么需要MyBatis

- 帮助程序员将数据存入到数据库中
- 方便

- 传统的JDBC代码太复杂了，简化，框架，自动化
- 不用MyBatis也可以，技术没有高低之分
- 优点：
    - 简单易学
    - 灵活
    - sql和代码的分离，提高了可维护性。
    - 提供映射标签，支持对象与数据库的orm字段关系映射
    - 提供对象关系映射标签，支持对象关系组建维护
    - 提供xml标签，支持编写动态sql

2、第一个Mybatis程序

思路：搭建环境 --> 导入MyBatis --> 编写代码 --> 测试

2.1 搭建环境

准备工作： 创建数据库表

```sql
CREATE DATABASE `mybatis`;

use `mybatis`;

CREATE TABLE `user`(
`id` INT(20) NOT NULL AUTO_INCREMENT,
`name` VARCHAR(30) DEFAULT NULL,
`pwd` VARCHAR(30) DEFAULT NULL,
 PRIMARY KEY(`id`)
)ENGINE=INNODB DEFAULT CHARSET=utf8;

INSERT INTO `user` (`id`,`name`,`pwd`) VALUES (NULL,'oldwang','123456');
INSERT INTO `user` (`id`,`name`,`pwd`) VALUES (NULL,'test1','test2');
INSERT INTO `user` (`id`,`name`,`pwd`) VALUES (NULL,'test2','test2');
```

新建项目

1. 创建一个普通的maven项目
2. 删除src目录（就可以把此工程当做父工程了，然后创建子工程）
3. 导入maven依赖

```xml
<!--导入依赖-->
<dependencies>

    <!--mysqlq驱动-->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.12</version>
    </dependency>
```

```
    <!--mybatis-->
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.5.4</version>
    </dependency>

    <!--junit-->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

4. 新建一个model

2.2 快速开始

- 创建mybatis核心配置文件

```
<?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">
<!--核心配置文件-->
<configuration>
    <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?
userSSL=true&amp;useUnicode=true&amp;characterEncoding=UTF-8&amp;serverTimezone=UTC"/>
                <property name="username" value="root"/>
                <property name="username" value="root"/>
                <property name="password" value="root"/>
            </dataSource>
        </environment>
    </environments>
</configuration>
```

- 编写MybatisUtils工具类

```
package com.oldwang.utils;

import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;
```

```
import java.io.IOException;
import java.io.InputStream;

//SqlSessionFactory  --> SqlSession
public class MybatisUtils {
    private static SqlSessionFactory sqlSessionFactory = null;
    static {
        try {
            //使用mybatis第一步 获取SQLSessionFactory
            String resource = "org/mybatis/example/mybatis-config.xml";
            InputStream inputStream = Resources.getResourceAsStream(resource);
            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。你可以通过 SqlSession 实例来直接执行
已映射的 SQL 语句。例如：
    public static SqlSession SqlSession (){
        return sqlSessionFactory.openSession();
    }
}
```

2.3 编写代码

- 实体类

```
package com.oldwang.pojo;

import java.io.Serializable;

public class User implements Serializable {
    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;
    }

    @Override
    public String toString() {
        return "User{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", pwd='" + pwd + '\'' +
                '}';
    }
}
```

- Dao接口

```
package com.oldwang.dao;

import com.oldwang.pojo.User;

import java.util.List;

public interface UserMapper {

    List<User> getUserList();
}
```

- 接口实现类由原来的实现类转为mapper.xml

```
<?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">

<!--namespace=绑定一个对应的Mapper接口-->
```

```xml
<mapper namespace="com.oldwang.dao.UserMapper">

    <!--select查询语句-->
    <select id="getUserList" resultType="com.oldwang.pojo.User">
       select * from mybatis.user
    </select>

</mapper>
```

2.4 测试

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

该错误是Mapper.xml没有在mybatis-config.xml中注册 解决方案如下

```xml
在mybatis-config中指定Mapper.xml的位置
    <!--每一个Mapper都需要在mybatis核心配置文件中注册-->
    <mappers>
        <mapper resource="com/oldwang/dao/UserMapper.xml"></mapper>
    </mappers>
```

可能会遇见的错误

```
java.lang.ExceptionInInitializerError
    at com.oldwang.dao.UserMapperTest.test(UserMapperTest.java:15) <25 internal calls>
Caused by: org.apache.ibatis.exceptions.PersistenceException:
### Error building SqlSession.
### The error may exist in com/oldwang/dao/UserMapper.xml
### Cause: org.apache.ibatis.builder.BuilderException: Error parsing SQL Mapper Configuration. Cause: java.io.IOException: Could not find resource com/oldwang/dao/UserMap
    at org.apache.ibatis.exceptions.ExceptionFactory.wrapException(ExceptionFactory.java:30)
    at org.apache.ibatis.session.SqlSessionFactoryBuilder.build(SqlSessionFactoryBuilder.java:80)
    at org.apache.ibatis.session.SqlSessionFactoryBuilder.build(SqlSessionFactoryBuilder.java:64)
    at com.oldwang.utils.MybatisUtils.<clinit>(MybatisUtils.java:19)
    ... 26 more
```

maven由于它的约定大于配置 我们之后可能遇到我们写的配置文件无法导出或者生效的问题 解决方案在maven中加入如下配置

```xml
<build>
        <resources>
            <resource>
                <directory>src/main/resource</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>
```

测试代码

```
package com.oldwang.dao;
import com.oldwang.pojo.User;
import com.oldwang.utils.MybatisUtils;
import org.apache.ibatis.session.SqlSession;
import org.junit.Test;
import java.util.List;

public class UserMapperTest {

    @Test
    public void test(){
        //获取SQLSession对象
        SqlSession sqlSession = MybatisUtils.getSqlSession();
        //获取UserMapper对象
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
        //执行SQL
        List<User> userList = mapper.getUserList();
        userList.forEach(user -> {
            System.out.println(user);
        });
        //释放资源
        sqlSession.close();
    }
}
```

3 CURD

1 namespace

namespace中的包名要和DAO/Mapper接口的包名一致

2 增删改查语句

选择，查询语句

- id 就是对应namespace中的方法名
- resultType SQL语句的返回值
- parameterType 参数类型

1.编写接口

```
public interface UserMapper {

    //查询全部用户
    List<User> getUserList();
```

```
    //根据ID查询用户
    User getUserById(int id);

    //添加用户
    int addUser(User user);

    //修改用户
    int updateUser(User user);

    //删除用户
    int deleteUser(int id);
}
```

2. 编写对应的mapper中的sql语句

```
<?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">

<!--namespace=绑定一个对应的Mapper接口-->
<mapper namespace="com.oldwang.dao.UserMapper">

    <!--select查询语句-->
    <select id="getUserList" resultType="com.oldwang.pojo.User">
       select * from mybatis.user
    </select>

    <select id="getUserById" parameterType="int" resultType="com.oldwang.pojo.User">
       select * from mybatis.user where id = #{id}
    </select>

    <!--对象中的属性可以直接取出来-->
    <insert id="addUser" parameterType="com.oldwang.pojo.User" >
        insert into mybatis.user (id,name,pwd) values (#{id},#{name},#{pwd})
    </insert>

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

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

3. 测试

```
public class UserMapperTest {
```

```
    @Test
    public void getUserList(){
        //获取SQLSession对象
        SqlSession sqlSession = MybatisUtils.getSqlSession();
        //获取UserMapper对象 方式一
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
        //执行SQL
        List<User> userList = mapper.getUserList();
        userList.forEach(System.out::println);
        //获取UserMapper对象 方式二
        List<User> users = sqlSession.selectList("com.oldwang.dao.UserMapper.getUserList",
User.class);
        users.forEach(System.out::println);
        //释放资源
        sqlSession.close();
    }


    @Test
    public void getUserById(){
        SqlSession sqlSession = MybatisUtils.getSqlSession();
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
        User userById = mapper.getUserById(1);
        System.out.println(userById);
        sqlSession.close();
    }

    @Test  //增删改需要提交事务
    public void addUser(){
        SqlSession sqlSession = MybatisUtils.getSqlSession();
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
        User user = new User();
        user.setId(4);
        user.setName("张三");
        user.setPwd("123");
        mapper.addUser(user);
        //提交事务
        sqlSession.commit();
        sqlSession.close();
    }

    @Test
    public void updateUser(){
        SqlSession sqlSession = MybatisUtils.getSqlSession();
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
        User user = new User();
        user.setId(4);
        user.setName("张三");
        user.setPwd("123345");
        mapper.updateUser(user);
        //提交事务
        sqlSession.commit();
```

```
        sqlSession.close();
    }

    @Test
    public void deleteUser(){
        SqlSession sqlSession = MybatisUtils.getSqlSession();
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
        mapper.deleteUser(4);
        //提交事务
        sqlSession.commit();
        sqlSession.close();
    }
}
```

注意 增删改需要提交事务

3 Map类型

假设，我们的实体类，或者数据库中的表，字段或者参数过多，我们应该考虑使用Map!

- UserMapper接口

```
 //map插入用户
int addUserByMap(Map<String,Object> map);
```

- UserMapper.xml

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

- 测试

```
  @Test
    public void addUserByMap(){
        SqlSession sqlSession = MybatisUtils.getSqlSession();
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
        Map<String,Object> map = new HashMap<>();
        map.put("id",4);
        map.put("name","张三");
        map.put("pwd","123456");
        mapper.addUserByMap(map);
        //提交事务
        sqlSession.commit();
        sqlSession.close();
    }
```

> Map传递参数，直接在sql中取出key即可！【parameter="map"】
>
> 对象传递参数，直接在sql中取出对象的属性即可！【parameter="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，**或者注解！**

4 模糊查询

模糊查询这么写？

1. Java代码执行的时候，传递通配符% %

   Mapper接口

```
//模糊查询
List<User> getUserByLike(String userName);
```

   Mapper.xml

```
    <!--模糊查询-->
    <select id="getUserByLike" parameterType="java.lang.String"
resultType="com.oldwang.pojo.User">
        select * from mybatis.user where name like #{userName}
    </select>
```

   测试

```
@Test
    public void getUserByLike(){
        SqlSession sqlSession = MybatisUtils.getSqlSession();
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
        List<User> userByLike = mapper.getUserByLike("%test%");
        userByLike.forEach(System.out::println);
        sqlSession.close();
    }
```

2. 在sql拼接中使用通配符

   Mapper接口

```
//模糊查询
List<User> getUserByLike(String userName);
```

   Mapper.xml

```
    <!--模糊查询-->
    <select id="getUserByLike" parameterType="java.lang.String"
resultType="com.oldwang.pojo.User">
        select * from mybatis.user where name like "%"#{userName}"%"
    </select>
```

   测试

```
    @Test
    public void getUserByLike(){
        SqlSession sqlSession = MybatisUtils.getSqlSession();
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
        List<User> userByLike = mapper.getUserByLike("test");
        userByLike.forEach(System.out::println);
        sqlSession.close();
    }
```

4 配置解析

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）
```

2. 环境配置 environments

MyBatis 可以配置成适应多种环境

不过要记住：尽管可以配置多个环境，但每个 SqlSessionFactory 实例只能选择一种环境

学会使用配置多套运行环境！

MyBatis默认的事务管理器就是JDBC ，连接池：POOLED

3. 属性 properties

我们可以通过properties属性来实现引用配置文件

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

1. 编写一个配置文件

   db.properties

```
driver=com.mysql.cj.jdbc.Driver
url=jdbc:mysql://localhost:3306/mybatis?
userSSL=true&useUnicode=true&characterEncoding=UTF-8&serverTimezone=UTC
username=root
password=root
```

2. 在核心配置文件中引入

```
<!--引用外部配置文件-->
<properties resource="db.properties">
    <property name="username" value="root"/>
    <property name="password" value="root"/>
</properties>
```

- 可以直接引入外部文件
- 可以在其中增加一些属性配置
- 如果两个文件有同一个字段，优先使用外部配置文件的

4. 类型别名 typeAliases

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

```
<!--可以给实体类起别名-->
<typeAliases>
    <typeAlias type="com.oldwang.pojo.User" alias="User"/>
</typeAliases>
```

也可以指定一个包，每一个在包 `domain.blog` 中的 Java Bean，在没有注解的情况下，会使用 Bean 的首字母小写的非限定类名来作为它的别名。 比如 `domain.blog.Author` 的别名为 `author`；若有注解，则别名为其注解值。见下面的例子：

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

> 在实体类比较少的时候，使用第一种方式。
>
> 如果实体类十分多，建议用第二种扫描包的方式。
>
> 第一种可以DIY别名，第二种不行，如果非要改，需要在实体上增加注解。

```
@Alias("author")
public class Author {
    ...
}
```

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。

配置官网地址：https://mybatis.org/mybatis-3/zh/configuration.html#settings

6. 其他配置

- typeHandlers（类型处理器）
- objectFactory（对象工厂）
- plugins 插件
  - mybatis-generator-core
  - mybatis-plus
  - 通用mapper

7. 映射器 mappers

MapperRegistry：注册绑定我们的Mapper文件；

方式一：【推荐使用】

```
<!--每一个Mapper.xml都需要在MyBatis核心配置文件中注册-->
<mappers>
    <mapper resource="com/kuang/dao/UserMapper.xml"/>
</mappers>
```

方式二：使用class文件绑定注册

```
<!--每一个Mapper.xml都需要在MyBatis核心配置文件中注册-->
<mappers>
    <mapper class="com.kuang.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

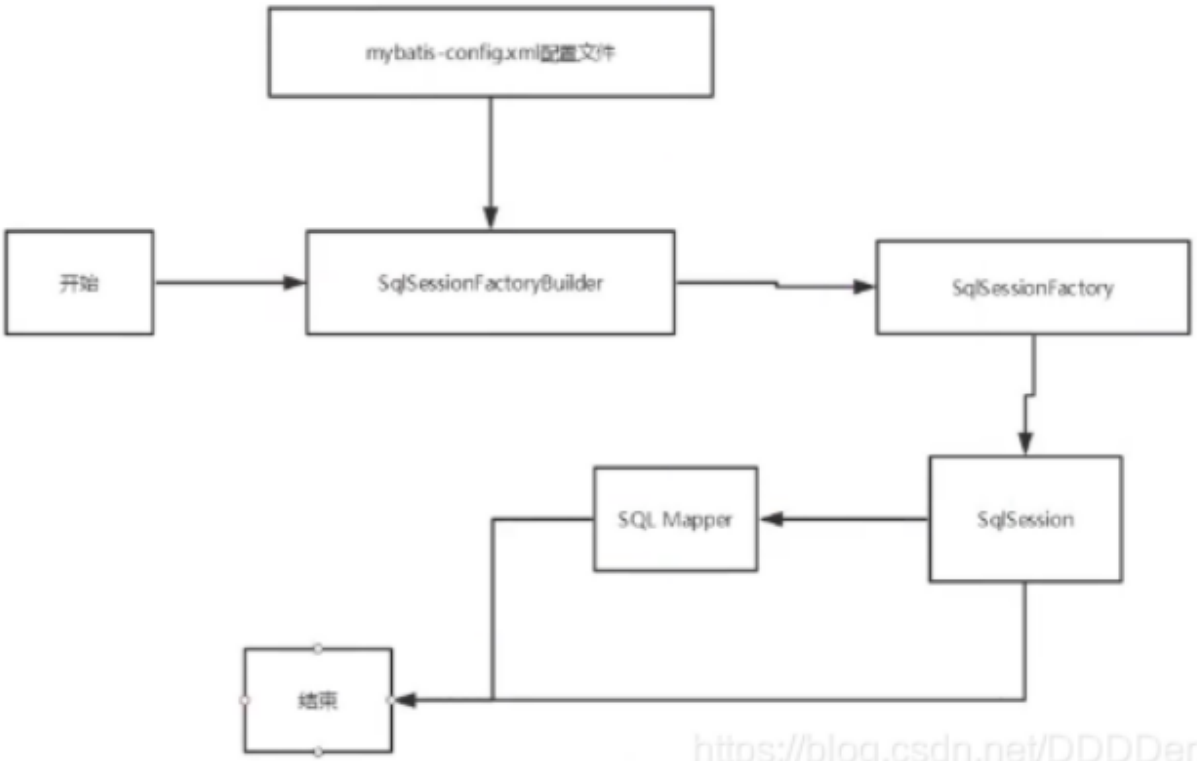
方式三：使用包扫描进行注入

```
<mappers>
    <package name="com.kuang.dao"/>
</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

8. 作用域和生命周期



声明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:

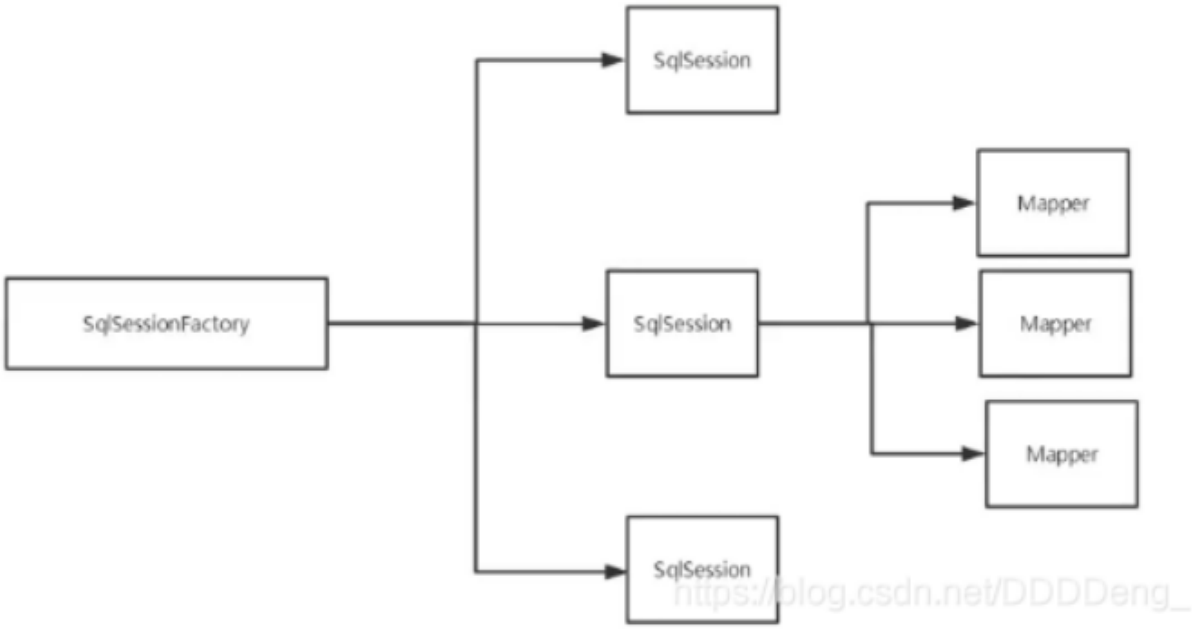
- 一旦创建了SqlSessionFactory，就不再需要它了
- 局部变量

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，**没有任何理由丢弃它或重新创建一个实例。**
- 因此SqlSessionFactory的最佳作用域是应用作用域（ApplocationContext）。
- 最简单的就是使用**单例模式**或静态单例模式。

SqlSession：

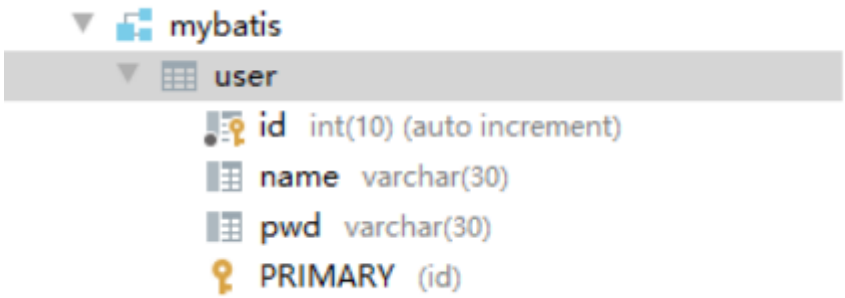
- 连接到连接池的一个请求
- SqlSession 的实例不是线程安全的，因此是不能被共享的，所以它的最佳的作用域是请求或方法作用域。
- 用完之后需要赶紧关闭，否则资源被占用！



5、ResultMap

1. 问题

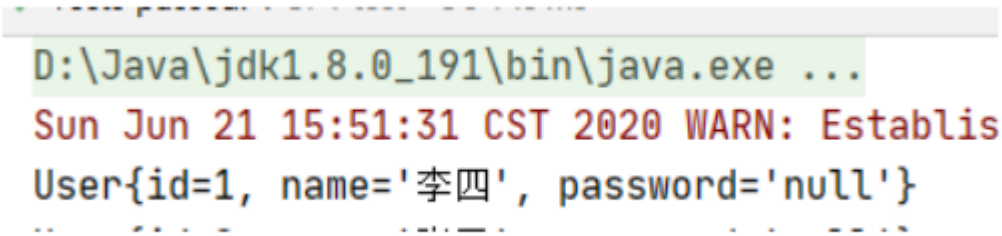
数据库中的字段



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;
```

测试出现问题




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

解决方法：

- 起别名

```
<select id="getUserById" resultType="com.kuang.pojo.User">
    select id,name,pwd as password from USER where id = #{id}
</select>
```

2. resultMap

结果集映射

```
<!--结果集映射-->
<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>
    <result column="name" property="name"></result>
    <result column="pwd" property="password"></result>
</resultMap>

<select id="getUserList" resultMap="UserMap">
    select * from USER
</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` 的优秀之处——你完全可以不用显式地配置它们。
- 如果这个世界总是这么简单就好了。

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，日志就是最好的助手！

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中具体使用哪一个日志实现，在设置中设定

STDOUT_LOGGING

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

```
Opening JDBC Connection
Sun Jun 21 16:43:09 CST 2020 WARN: Establishing SSL connection without server's identity verifica
Created connection 131635550.
Setting autocommit to false on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@7d8995e]
==>  Preparing: select * from USER
==> Parameters:
<==    Columns: id, name, pwd
<==        Row: 1, 李四, 123
<==        Row: 2, 张三, 233
<==        Row: 3, 黑子, 666
<==        Row: 4, 王虎, 789
<==      Total: 4
User{id=1, name='李四', password='123'}
User{id=2, name='张三', password='233'}
User{id=3, name='黑子', password='666'}
User{id=4, name='王虎', password='789'}
Resetting autocommit to true on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@7d8995e]
Closing JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@7d8995e]
Returned connection 131635550 to pool.
```


6.2 Log4j

什么是Log4j？

- Log4j是Apache的一个开源项目，通过使用Log4j，我们可以控制日志信息输送的目的地是控制台、文件、GUI组件；
- 我们也可以控制每一条日志的输出格式；
- 通过定义每一条日志信息的级别，我们能够更加细致地控制日志的生成过程；
- 最令人感兴趣的就是，这些可以通过一个配置文件来灵活地进行配置，而不需要修改应用的代码。

1. 先导入log4j的包

```
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
</dependency>
```

2. log4j.properties

```
#将等级为DEBUG的日志信息输出到console和file这两个目的地，console和file的定义在下面的代码
log4j.rootLogger=DEBUG,console,file
#控制台输出的相关设置
log4j.appender.console = org.apache.log4j.ConsoleAppender
log4j.appender.console.Target = System.out
log4j.appender.console.Threshold=DEBUG
log4j.appender.console.layout = org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=[%c]-%m%n
#文件输出的相关设置
log4j.appender.file = org.apache.log4j.RollingFileAppender
log4j.appender.file.File=./log/rzp.log
log4j.appender.file.MaxFileSize=10mb
log4j.appender.file.Threshold=DEBUG
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=[%p][%d{yy-MM-dd}][%c]%m%n
#日志输出级别
log4j.logger.org.mybatis=DEBUG
log4j.logger.java.sql=DEBUG
log4j.logger.java.sql.Statement=DEBUG
log4j.logger.java.sql.ResultSet=DEBUG
log4j.logger.java.sq1.PreparedStatement=DEBUG
```

1. 配置settings为log4j实现
2. 测试运行

Log4j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对象

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

3. 日志级别

```
logger.info("info: 测试log4j");
logger.debug("debug: 测试log4j");
logger.error("error:测试log4j");
```

7、分页

思考：为什么分页？

- 减少数据的处理量

7.1 使用Limit分页

```
SELECT * from user limit startIndex,pageSize
```

使用MyBatis实现分页，核心SQL

1. 接口

```
//分页
List<User> getUserByLimit(Map<String,Integer> map);
```

2. Mapper.xml

```
<!--分页查询-->
<select id="getUserByLimit" parameterType="map" resultMap="UserMap">
    select * from user limit #{startIndex},#{pageSize}
</select>
```

3. 测试

```
    @Test
    public void getUserByLimit(){
        SqlSession sqlSession = MybatisUtils.getSqlSession();
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
        HashMap<String, Integer> map = new HashMap<String, Integer>();
        map.put("startIndex",1);
        map.put("pageSize",2);
        List<User> list = mapper.getUserByLimit(map);
        for (User user : list) {
            System.out.println(user);
        }
    }
```

7.2 RowBounds分页

不再使用SQL实现分页

1. 接口

```
//分页2
List<User> getUserByRowBounds();
```

2. Mapper.xml

```xml
<select id="getUserByRowBounds"  resultType="User">
    select * from user
</select>
```

3. 测试

```java
@Test
public void getUserByRowBounds(){
    SqlSession sqlSession = MybatisUtils.getSqlSession();
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);
    //RowBounds实现
    RowBounds rowBounds = new RowBounds(1,2);
    List<User> userByRowBounds = mapper.getUserByRowBounds(rowBounds);
    userByRowBounds.forEach(System.out::println);
    sqlSession.close();
}
```

7.3 分页插件

（自己研究）

8、使用注解开发

8.1 面向接口开发

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，考虑它的属性和方法；

- 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

```
//注解开发
@Select("select * from user")
List<User> getListUser();
```

2. 需要在核心配置文件中绑定接口

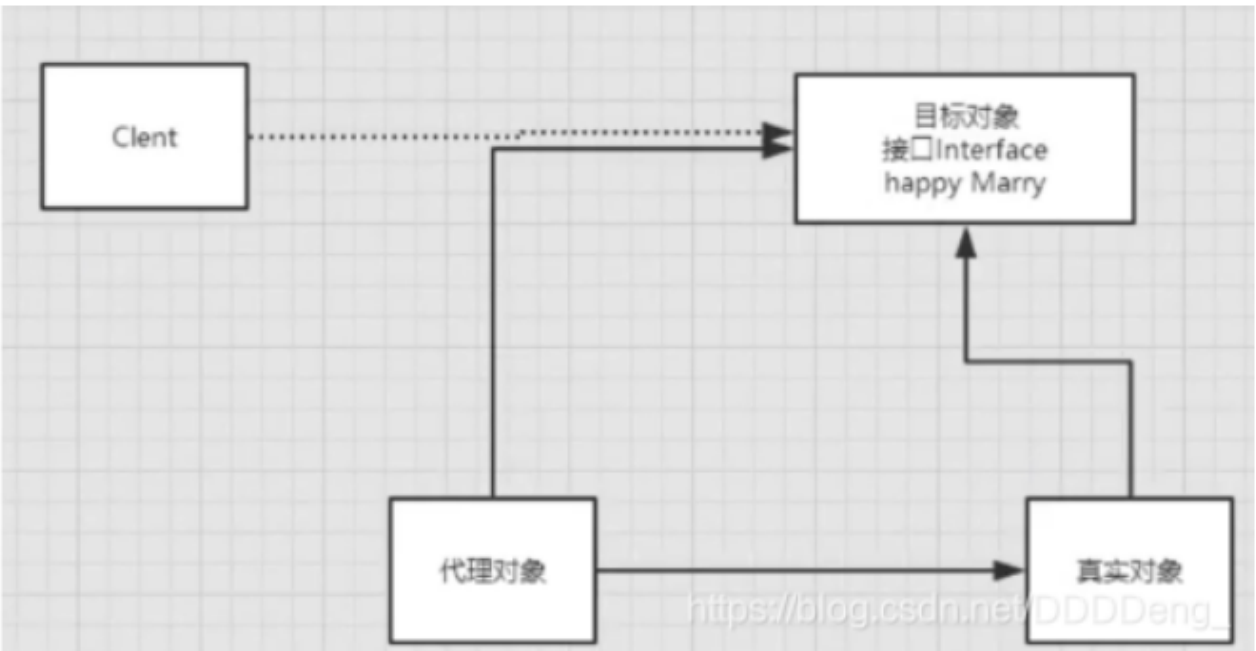
```
<mappers>
    <mapper class="com.kuang.dao.UserMapper"/>
</mappers>
```

3. 测试

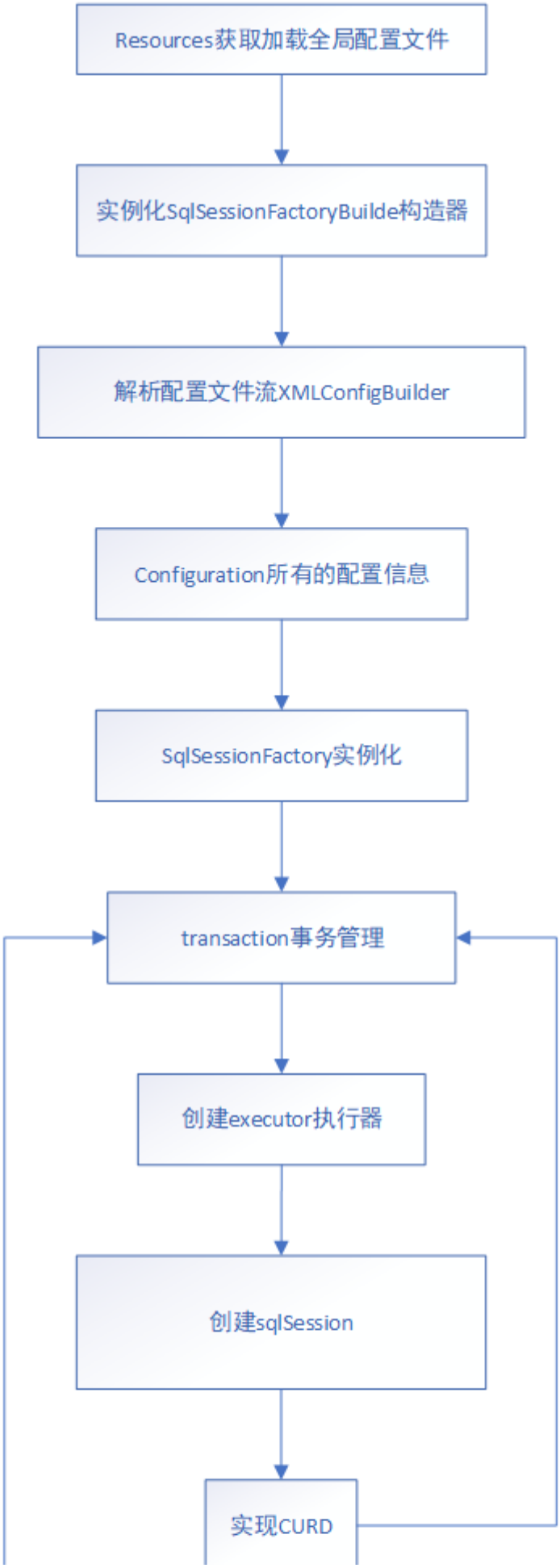
```
@Test
public void getListUser(){
    SqlSession sqlSession = MybatisUtils.getSqlSession();
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);
    List<User> listUser = mapper.getListUser();
    listUser.forEach(System.out::println);
}
```

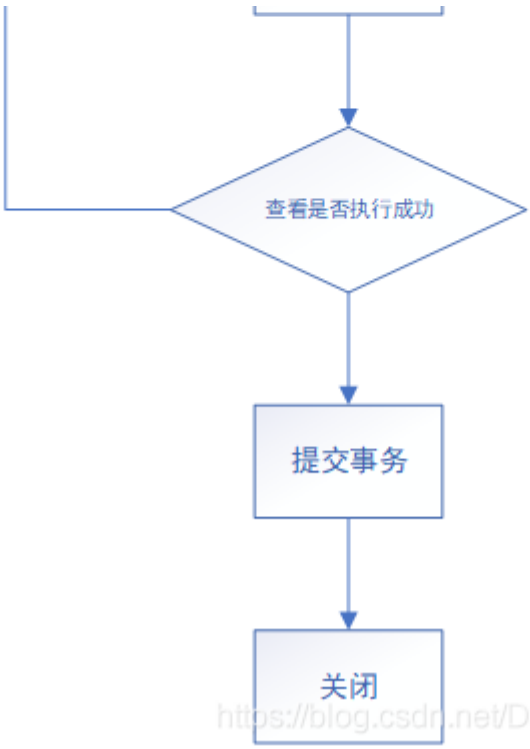
本质：反射机制实现

底层：动态代理



MyBatis详细执行流程


Resources获取加载全局配置文件
实例化SqlSessionFactoryBuilde构造器
解析配置文件流XMLConfigBuilder
Configuration所有的配置信息
SqlSessionFactory实例化
transaction事务管理
创建executor执行器
创建sqlSession
实现CURD



8.3 注解CURD

我们可以再创建工具类的时候实现自动提交事务

```
public openSession getSqlsession(){
    return sqlSessionFactory.openSession(true)
}
```

```
//方法存在多个参数，所有的参数前面必须加上@Param("id")注解
@Delete("delete from user where id = ${uid}")
int deleteUser(@Param("uid") int id);
```

关于@Param( )注解

- 基本类型的参数或者String类型，需要加上
- 引用类型不需要加
- 如果只有一个基本类型的话，可以忽略，但是建议大家都加上
- 我们在SQL中引用的就是我们这里的@Param()中设定的属性名

#{} 能够很大程度防止SQL注入

${} 无法防止SQL注入

9、Lombok

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

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>1.18.10</version>
    <scope>provided</scope>
</dependency>
```

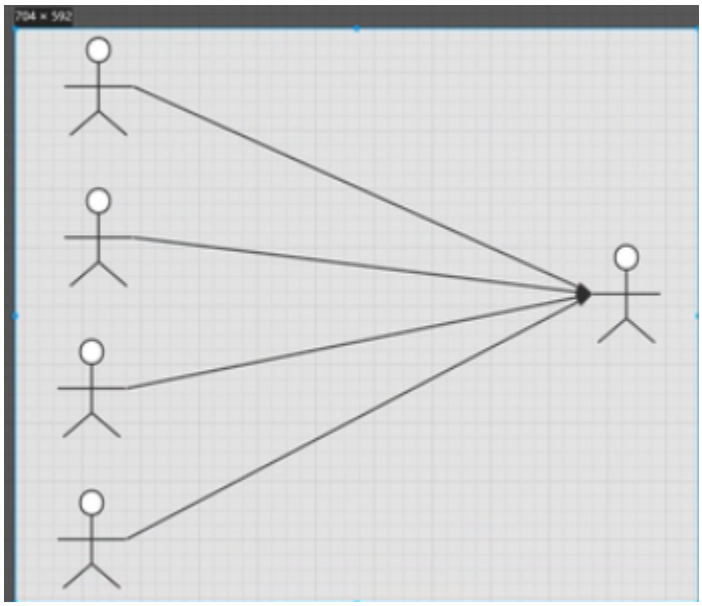
3. 在程序上加注解

```
@Getter and @Setter
@FieldNameConstants
@ToString
@EqualsAndHashCode
@AllArgsConstructor, @RequiredArgsConstructor and @NoArgsConstructor
@Log, @Log4j, @Log4j2, @Slf4j, @XSlf4j, @CommonsLog, @JBossLog, @Flogger, @CustomLog
@Data
@Builder
@SuperBuilder
@Singular
@Delegate
@Value
@Accessors
@Wither
@With
@SneakyThrows
@val
```

说明：

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class User {
    private int id;
    private String name;
    private String password;
}
```

10、多对一处理



> 多个学生 对应一个老师
>
> 对于学生而言，关联...多个学生 关联一个老师【多对一】
>
> 对于老师而言 集合 一个老师有很多学生 【一对多】

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

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

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

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. 测试查询是否能够成功

2. 按照查询嵌套处理

```
<!--
     思路：
        1. 查询所有的学生信息
        2. 根据查询出来的学生的tid寻找特定的老师 （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，我们需要单独出来 对象：association 集合：collection-->
    <collection property="teacher" column="tid" javaType="teacher" select="getTeacher"/>
</resultMap>
<select id="getTeacher" resultType="teacher">
    select * from teacher where id = #{id}
</select>
```

3.按照结果嵌套处理

```
    <!--按照结果进行查询-->
    <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>
    <!--结果封装，将查询出来的列封装到对象属性中-->
    <resultMap id="StudentTeacher2" type="student">
        <result property="id" column="sid"/>
        <result property="name" column="sname"/>
        <association property="teacher" javaType="teacher">
            <result property="name" column="tname"></result>
        </association>
    </resultMap>
```

回顾Mysql多对一查询方式:

- 子查询（按照查询嵌套）
- 联表查询（按照结果嵌套）

11、一对多处理

> 一个老师多个学生；

> 对于老师而言，就是一对多的关系；

1. 环境搭建

实体类

```
@Data
public class Student {
    private int id;
    private String name;
    private int tid;
}
```

```
@Data
public class Teacher {
    private int id;
    private String name;

    //一个老师拥有多个学生
    private List<Student> students;
}
```

2. 按照结果嵌套嵌套处理

```
<!--按结果嵌套查询-->
<select id="getTeacher" resultMap="StudentTeacher">
    SELECT s.id sid, s.name sname,t.name tname,t.id tid FROM student s, teacher t
    WHERE s.tid = t.id AND tid = #{tid}
</select>
<resultMap id="StudentTeacher" type="Teacher">
    <result property="id" column="tid"/>
    <result property="name" column="tname"/>
    <!--复杂的属性，我们需要单独处理 对象：association 集合：collection
    javaType=""指定属性的类型！
    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>
```

小结

1. 关联 - association 【多对一】
2. 集合 - collection 【一对多】
3. javaType & ofType

1. JavaType用来指定实体类中的类型
2. 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：

- 保证SQL的可读性，尽量保证通俗易懂
- 注意一对多和多对一，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就是根据不同的条件生成不同的SQL语句

所谓的动态SQL，本质上还是SQL语句，只是我们可以在SQL层面，去执行一个逻辑代码

> 动态 SQL 是 MyBatis 的强大特性之一。如果你使用过 JDBC 或其它类似的框架，你应该能理解根据不同条件拼接 SQL 语句有多痛苦，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。
>
> if
>
> choose (when otherwise)
>
> trim (where,set)
>
> foreach

```
CREATE TABLE `mybatis`.`blog`  (
  `id` varchar(50) NOT NULL COMMENT '博客id',
  `title` varchar(30) 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',
  PRIMARY KEY (`id`)
)
```

创建一个基础工程

1. 导包
2. 编写配置文件
3. 编写实体类

```
@Data
public class Blog {
    private  int 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

动态SQL之IF语句

接口

```
//查询博客
List<Blog> queryBlog(Map map);
```

Mapper.xml

```
 <select id="queryBlog" parameterType="map" resultType="blog">
        select * from blog
          <where>
            <if test="title !=null" >
                and title = #{title}
            </if>
            <if test="author !=null">
                and author = #{author}
            </if>
          </where>
</select>
```

test

```
@Test
public void queryBlogIF(){
    SqlSession sqlSession = MybatisUtils.getSqlSession();
    BlogMapper mapper = sqlSession.getMapper(BlogMapper.class);
    Map<String,String> map = new HashMap<>();
    map.put("title","spring 如此简单");
    map.put("author","oldwang");
    List<Blog> blogs = mapper.queryBlog(map);
    blogs.forEach(System.out::println);
}
```

动态SQL常用标签

choose (when otherwise)

```
<select id="queryBlogChoose" resultType="blog" parameterType="map">
    select * from blog
    <where>
        <choose>
            <when test="title !=null">
                title = #{title}
            </when>
            <when test="author !=null">
                and autho = #{author}
            </when>
            <otherwise>
                and views = #{views}
```

```
            </otherwise>
        </choose>
    </where>
</select>
```

trim (where,set)

where

```
 <select id="queryBlog" parameterType="map" resultType="blog">
        select * from blog
          <where>
            <if test="title !=null" >
                and title = #{title}
            </if>
            <if test="author !=null">
                and author = #{author}
            </if>
          </where>
</select>
```

set

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

所谓的动态SQL本质还是SQL 只是我们可以再SQL层面执行一些逻辑代码

foreach

接口

```
//查询1,2,3记录的博客
List<Blog> getBlog(List<String> list);
```

mapper.xml

```
<select id="getBlog" resultType="blog" parameterType="java.util.List">
    select * from blog
    <where>
        id in
        <foreach collection="list" item="item" open="(" close=")" separator=",">
            #{item}
        </foreach>
    </where>
</select>
```

测试

```
@Test
public void test(){
    SqlSession sqlSession = MybatisUtils.getSqlSession();
    BlogMapper mapper = sqlSession.getMapper(BlogMapper.class);
    List<Blog> blog = mapper.getBlog(Arrays.asList("1","2","3"));
    blog.forEach(System.out::println);
}
```

SQL片段

有的时候我们可能会将公共的部分抽取出来方便复用

使用SQL标签抽取公共部分

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

在需要使用的地方使用include标签引入

```
    <select id="queryBlogIF" parameterType="map" resultType="blog">
        select * from blog
          <where>
            <include refid="if-title-author"></include>
          </where>
    </select>
```

注意事项：

- 最好基于单标来定义SQL片段
- 不要存在where标签

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：

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

13 缓存

13.1 简介

> 查询 ： 连接数据库，耗资源
>
> 一次查询的结果，给他暂存一个可以直接取到的地方 --> 内存：缓存
>
> 我们再次查询的相同数据的时候，直接走缓存，不走数据库了

1. 什么是缓存[Cache]？
    - 存在内存中的临时数据
    - 将用户经常查询的数据放在缓存（内存）中，用户去查询数据就不用从磁盘上（关系型数据库文件）查询，从缓存中查询，从而提高查询效率，解决了高并发系统的性能问题
2. 为什么使用缓存？
    - 减少和数据库的交互次数，减少系统开销，提高系统效率
3. 什么样的数据可以使用缓存？
    - 经常查询并且不经常改变的数据 【可以使用缓存】

13.2 MyBatis缓存

- MyBatis包含一个非常强大的查询缓存特性，它可以非常方便的定制和配置缓存，缓存可以极大的提高查询效率。
- MyBatis系统中默认定义了两级缓存：

  一级缓存

  和

  二级缓存
    - 默认情况下，只有一级缓存开启（SqlSession级别的缓存，也称为本地缓存）
    - 二级缓存需要手动开启和配置，他是基于namespace级别的缓存。
    - 为了提高可扩展性，MyBatis定义了缓存接口Cache。我们可以通过实现Cache接口来定义二级缓存。

13.3 一级缓存

- 一级缓存也叫本地缓存：SqlSession
    - 与数据库同一次会话期间查询到的数据会放在本地缓存中
    - 以后如果需要获取相同的数据，直接从缓存中拿，没必要再去查询数据库

测试步骤：

1. 开启日志
2. 测试在一个Session中查询两次记录

```java
@Test
    public void test(){
        SqlSession sqlSession = MybatisUtils.getSqlSession();
        BlogMapper mapper = sqlSession.getMapper(BlogMapper.class);
        List<Blog> blog = mapper.getBlog(Arrays.asList("1","2","3"));
        blog.forEach(System.out::println);
        System.out.println("========================");
        List<Blog> blog1 = mapper.getBlog(Arrays.asList("1","2","3"));
        blog1.forEach(System.out::println);

        System.out.println(blog == blog1);
    }
```

```
Opening JDBC Connection
Tue Jun 23 12:27:38 CST 2020 WARN: Establishing SS
Created connection 1108924067.
==>  Preparing: select * from user where id = ?
==> Parameters: 1(Integer)
<==    Columns: id, name, pwd
<==        Row: 1, 李四, 123
<==      Total: 1
User(id=1, name=李四, pwd=123)
========================================
User(id=1, name=李四, pwd=123)
true

Process finished with exit code 0
```


缓存失效的情况：

1. 查询不同的东西
2. 增删改操作，可能会改变原来的数据，所以必定会刷新缓存
3. 查询不同的Mapper.xml
4. 手动清理缓存

```java
sqlSession.clearCache();
```

5. 小结： 一级缓存默认是开启的 只在SQLsession开启和关闭的过程中有用

13.4 二级缓存

- 二级缓存也叫全局缓存，一级缓存作用域太低了，所以诞生了二级缓存
- 基于namespace级别的缓存，一个名称空间，对应一个二级缓存
- 工作机制
  - 一个会话查询一条数据，这个数据就会被放在当前会话的一级缓存中

- 如果会话关闭了，这个会员对应的一级缓存就没了；但是我们想要的是，会话关闭了，一级缓存中的数据被保存到二级缓存中
- 新的会话查询信息，就可以从二级缓存中获取内容
- 不同的mapper查询出的数据会放在自己对应的缓存（map）中

一级缓存开启（SqlSession级别的缓存，也称为本地缓存）

- 二级缓存需要手动开启和配置，他是基于namespace级别的缓存。
- 为了提高可扩展性，MyBatis定义了缓存接口Cache。我们可以通过实现Cache接口来定义二级缓存。

步骤：

1. 开启全局缓存

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

2. 在要是用二级缓存的Mapper.xml中使用缓存

```
<cache/>
<!--在当前Mapper.xml中使用二级缓存-->
<cache
       eviction="FIFO"
       flushInterval="60000"
       size="512"
       readOnly="true"/>
```

3. 测试

```
@Test
   public void test(){
       SqlSession sqlSession = MybatisUtils.getSqlSession();
       SqlSession sqlSession2 = MybatisUtils.getSqlSession();

       BlogMapper mapper = sqlSession.getMapper(BlogMapper.class);
       BlogMapper mapper2 = sqlSession2.getMapper(BlogMapper.class);

       List<Blog> blog = mapper.getBlog(Arrays.asList("1","2","3"));
       blog.forEach(System.out::println);
       sqlSession.close();

       System.out.println("========================");
       List<Blog> blog1 = mapper2.getBlog(Arrays.asList("1","2","3"));
       blog1.forEach(System.out::println);
       System.out.println(blog == blog1);

       sqlSession2.close();

   }
```

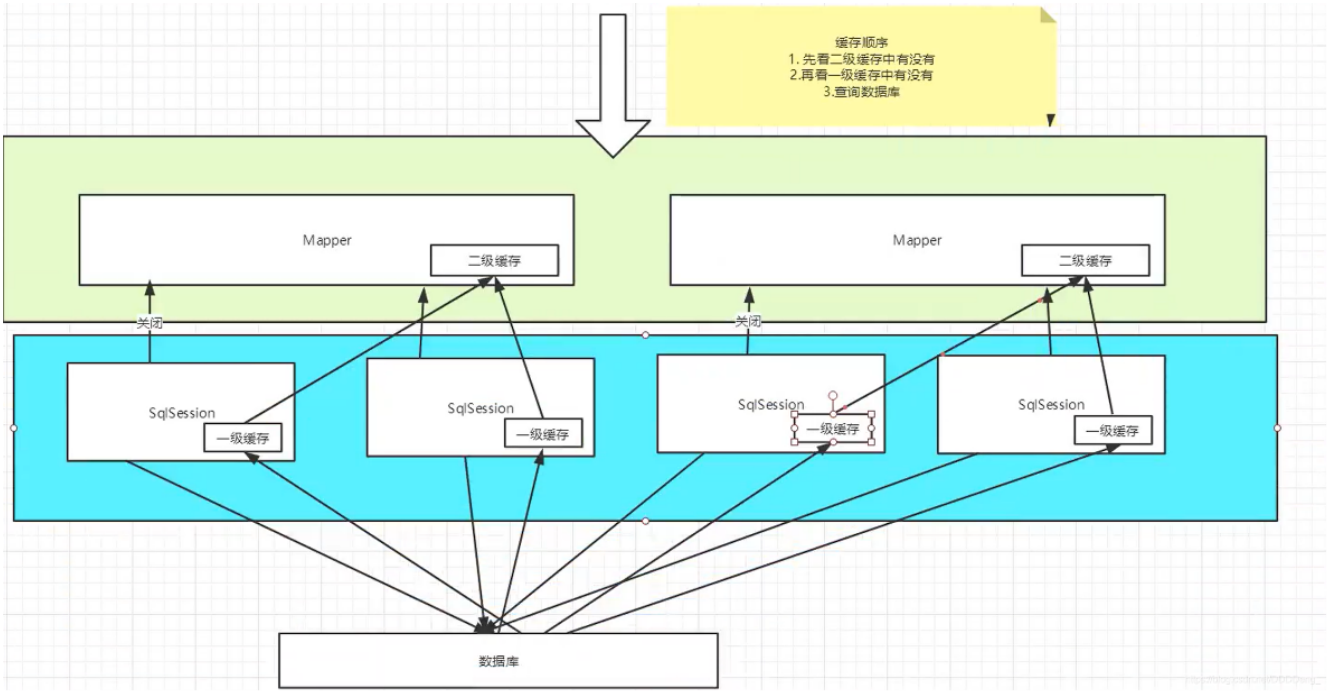
```
[org.apache.ibatis.datasource.pooled.PooledDataSource]-Created connection 384587033.
[org.apache.ibatis.transaction.jdbc.JdbcTransaction]-Setting autocommit to false on JDBC Connection [com.mysql.cj.jdbc.Connection]
[com.oldwang.dao.BlogMapper.getBlog]-==>  Preparing: select * from blog WHERE id in ( ? , ? , ? )
[com.oldwang.dao.BlogMapper.getBlog]-==> Parameters: 1(String), 2(String), 3(String)
[com.oldwang.dao.BlogMapper.getBlog]-<==      Total: 3
Blog(id=1, title=spring 如此简单2, author=oldwang, createTime=2020-10-09T12:14:49, views=9999)
Blog(id=2, title=mybatis 如此简单, author=oldwang, createTime=2020-10-09T12:14:49, views=9999)
Blog(id=3, title=java 如此简单, author=oldwang, createTime=2020-10-09T12:14:49, views=9999)
[org.apache.ibatis.transaction.jdbc.JdbcTransaction]-Resetting autocommit to true on JDBC Connection [com.mysql.cj.jdbc.Connectior
[org.apache.ibatis.transaction.jdbc.JdbcTransaction]-Closing JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@16ec5519]
[org.apache.ibatis.datasource.pooled.PooledDataSource]-Returned connection 384587033 to pool.
========================
[com.oldwang.dao.BlogMapper]-Cache Hit Ratio [com.oldwang.dao.BlogMapper]: 0.5只查询了一次
Blog(id=1, title=spring 如此简单2, author=oldwang, createTime=2020-10-09T12:14:49, views=9999)
Blog(id=2, title=mybatis 如此简单, author=oldwang, createTime=2020-10-09T12:14:49, views=9999)
Blog(id=3, title=java 如此简单, author=oldwang, createTime=2020-10-09T12:14:49, views=9999)
true
```

注意： 问题：我们需要将实体类序列化，否则就会报错

小结：

- 只要开启了二级缓存，在同一个Mapper下就有效
- 所有的数据都会放在一级缓存中
- 只有当前会话提交，或者关闭的时候，才会提交到二级缓存中

13.5 缓存原理



注意：

- 只有查询才有缓存，根据数据是否需要缓存（修改是否频繁选择是否开启）useCache="true"
- ```
  <select id="getUserById" resultType="user" useCache="true">
      select * from user where id = #{id}
  </select>
  ```

13.6 自定义缓存-ehcache
  ```

> Ehcache是一种广泛使用的开源Java分布式缓存。主要面向通用缓存

1. 导包

```
<dependency>
    <groupId>org.mybatis.caches</groupId>
    <artifactId>mybatis-ehcache</artifactId>
    <version>1.2.1</version>
</dependency>
```

2. 在mapper中指定使用我们的ehcache缓存实现

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