

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
- MyBatis 本是apache的一个开源项目iBatis, 2010年这个项目由apache software foundation 迁移到了google code，并且改名为MyBatis
- 2013年11月迁移到Github。

如何获得mybatis?

- maven仓库:

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

- Github: <https://github.com/mybatis/mybatis-3/releases>
- 中文文档: <https://mybatis.org/mybatis-3/zh/index.html>

1.2、持久化

数据持久化

- 持久化就是将程序的数据从临时状态向持久状态转化的过程
- 实现方式: 数据库 (jdbc) 、 io文件
- 内存**断电即失**

为什么需要持久化?

- 有一些对象, 不能丢弃
- 内存价格昂贵

1.3、持久层

如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

- 方便
- 帮助程序员将数据存入到数据库中
- 传统的jdbc代码太复杂了。我们需要通过框架来简化
- 不采用Mybatis框架也可，更容易上手
- 优点
 - 简单易学
 - 灵活
 - 解除sql与程序代码的耦合，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。

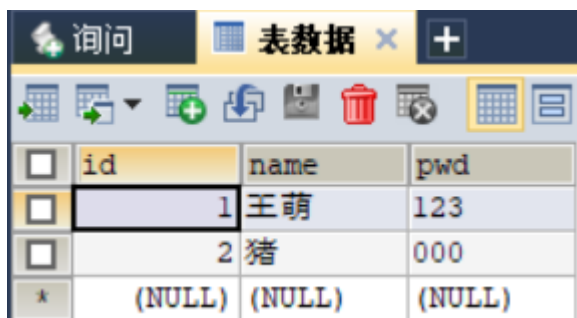
最重要的一点：使用的人多！

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

新建数据库



id	name	pwd
1	王萌	123
2	猪	000
(NULL)	(NULL)	(NULL)

新建项目

1. 新建一个普通的maven项目
2. 删除src目录

3. 导入依赖

```
<!--mysql-->
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.21</version>
</dependency>
<!--junit-->
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.11</version>
</dependency>
<!--mybatis-->
<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis</artifactId>
  <version>3.5.6</version>
</dependency>
```

2.2、创建一个模块

- 编写mybatis核心配置文件，连接数据库

```
<!--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">

<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?
useSSL=true&useUnicode=true&characterEncoding=UTF-
8&serverTimezone=UTC"/>
        <property name="username" value="root"/>
        <property name="password" value="7295wangmeng"/>
      </dataSource>
    </environment>
  </environments>

  <mappers>
    <mapper resource="UserMapper.xml"/>
  </mappers>
</configuration>
```

- 编写mybatis工具类

```
//工具类
public class MybatisUtils {
    //提升作用域
    private static SqlSessionFactory sqlSessionFactory;

    static{
        try {
            //使用mybatis第一步：获取sqlSessionFactory对象
            String resource = "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
    public static SqlSession getSqlSession(){
        return sqlSessionFactory.openSession();
    }
}
```

2.3、编写代码

- 实体类

```
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;
    }

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

```

- Dao接口

```

public interface UserDao {
    public List<User> getUserList();
}

```

- 接口实现类由原来的UserDaoDemo转变为一个Mapper配置文件

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
    PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<!--namespace绑定一个对应的Dao/Mapper接口-->
<mapper namespace="dao.UserDao">

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

```

2.4、测试

注意点：一定要注意路径！路径！

Mapperegistry是什么？

核心配置文件中注册mappers

- junit测试

```

public class UserDaoTest {
    @Test
    public void test(){
        //1.获得sqlSession对象
        SqlSession sqlSession = MybatisUtils.getSqlSession();
        //2.执行sql
        //方式一:
        UserDao mapper = sqlSession.getMapper(UserDao.class);
        List<User> userList = mapper.getUserList();

        for (User user : userList) {
            System.out.println(user);
        }

        //方式二:
        List<User> objects = sqlSession.selectList("dao.UserDao.getUserList");

        //关闭sqlSession
        sqlSession.close();
    }
}

```

可能会遇到的问题:

1. 配置文件没有注册

```

<mappers>
    <mapper resource="UserMapper.xml"/>
</mappers>
</configuration>

```

2. 绑定接口错误
3. 方法名不对
4. 返回类型错误
5. Maven到处资源问题 (xml文件放置的路径)

3、CRUD

1、namespace

namespace中的包名要和 Dao/Mapper接口的包名一致

2、select

选择、查询语句

- id: 对应的namespace中的方法名
- resultType: sql语句执行的返回值
- parameterType: 参数类型

1. 编写接口

```
public User getUserById(int id);
```

2. 编写对应Mapper中的sql语句

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

3. 写测试类

```
@Test
public void getUserById(){
    //获得sql执行对象sqlSession
    SqlSession sqlSession = MybatisUtils.getSqlSession();
    //执行sql
    UserDao mapper = sqlSession.getMapper(UserDao.class);
    User user = mapper.getUserById(1);

    System.out.println(user);
    sqlSession.close();
}
```

3、Insert

4、Delete

5、Update

- 增删改记得提交事务!

6、错误分析

- 注册Mapper时路径用'/'，其余时候都用'.'
- 标签不要匹配错
- 注意mysql的时区: &serverTimezone=UTC
- Maven资源无法导出
- 输出的xml文件中存在中文乱码问题

7、万能Map

假设实体类或数据库中表字段或参数过多，我们应当考虑使用万能Map

接口中:

```
public int addUser(Map<String, Object> map);
```

UserMapper中:

```
<!--对象中的属性，可以直接取出来
传递map的key，通过map控制参数-->
<insert id="id">
    insert into mybatis.user(id,name,pwd) values(#{userid},#{userName},#
{password})
</insert>
```

测试类：

```
public void addUser(){
    SqlSession sqlSession = Mybatisutils.getSqlSession();
    UserDao mapper = sqlSession.getMapper(UserDao.class);

    Map<String, Object> map = new HashMap<String, Object>();
    map.put("userid", 3);
    map.put("userName", "猫猫");
    map.put("password", "520");

    mapper.addUser(map);
    sqlSession.commit();
    sqlSession.close();
}
```

8、模糊查询怎么写？

预防sql注入

1. Java代码执行的时候，传递通配符%

```
List<User> users = mapper.getUser("%猫%");
```

2. 在sql拼接中，使用通配符%

```
<select id="getUser" resultType="pojo.User">
    select * from mybatis.user where name like #{value};
</select>
```

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

尽管可以配置多个环境，但每个 SqlSessionFactory 实例只能选择一种环境。

environments 元素定义了如何配置环境。

```
<environments default="development">
  <environment id="development">
    <transactionManager type="JDBC">
      <property name="..." value="..." />
    </transactionManager>
    <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>
</environments>
```

事务管理器，有JDBC、MANGLED两种
默认JDBC

数据源，有dbcp、c3p0、druid等
作用是连接数据库，类型三种：POOLED、UNPOOLED、JNDI，默认POOLED

3、属性 (properties)

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。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】

编写一个配置文件：db.properties

```
driver = com.mysql.jdbc.Driver
url = jdbc:mysql://localhost:3306/mybatis?
useSSL=true&useUnicode=true&characterEncoding=UTF-8&serverTimezone=UTC
username = root
password = 7295wangmeng
```

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

这是properties里面全部都配好的情况，所以可以直接使用

也可以一样写一半。当两者同时存在时，计算机先读取db.properties里面的数据，然后被mybatis-config.xml里的数据覆盖了。

宏观上存在**优先级**

```
driver = com.mysql.cj.jdbc.Driver
url = jdbc:mysql://localhost:3306/mybatis?
useSSL=true&useUnicode=true&characterEncoding=UTF-8&serverTimezone=UTC
password = 7295wangmeng
```

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

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="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

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

实体类数量较少时，可使用第一种方式；若数量较多，可使用第二种方式

第二种方式如果非要自己取名字，则需要在实体上增加注解

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

5、设置

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 <code>fetchType</code> 属性来覆盖该项的开关状态。	true false	false

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:

```
<mappers>
  <mapper resource="UserMapper.xml"/>
</mappers>
```

方式二: 使用class文件进行绑定注册

```
<mappers>
  <mapper class="UserMapper"/>
</mappers>
```

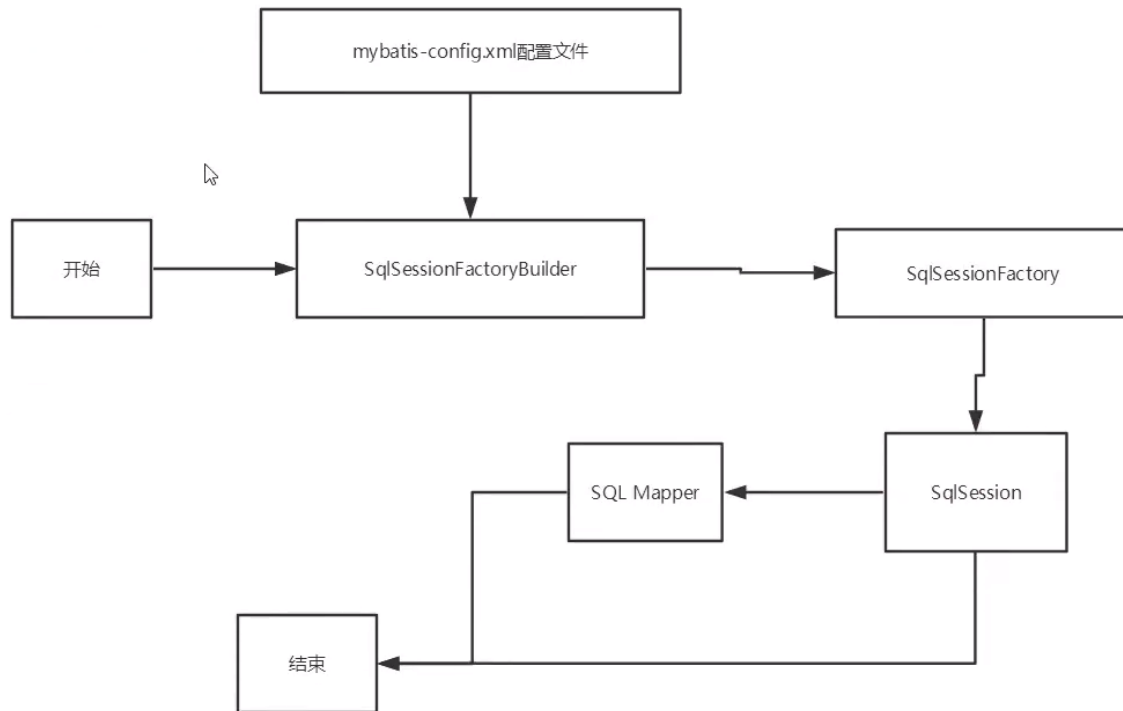
方式三: 扫描包进行绑定

```
<mappers>
  <package name="dao"/>
</mappers>
```

方式二、三注意点:

- 接口和它的Mapper文件必须同名
- 接口和它的Mapper文件必须在同一个包下 (所以需要配置resources)

8、生命周期和作用域 (Scope)



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:

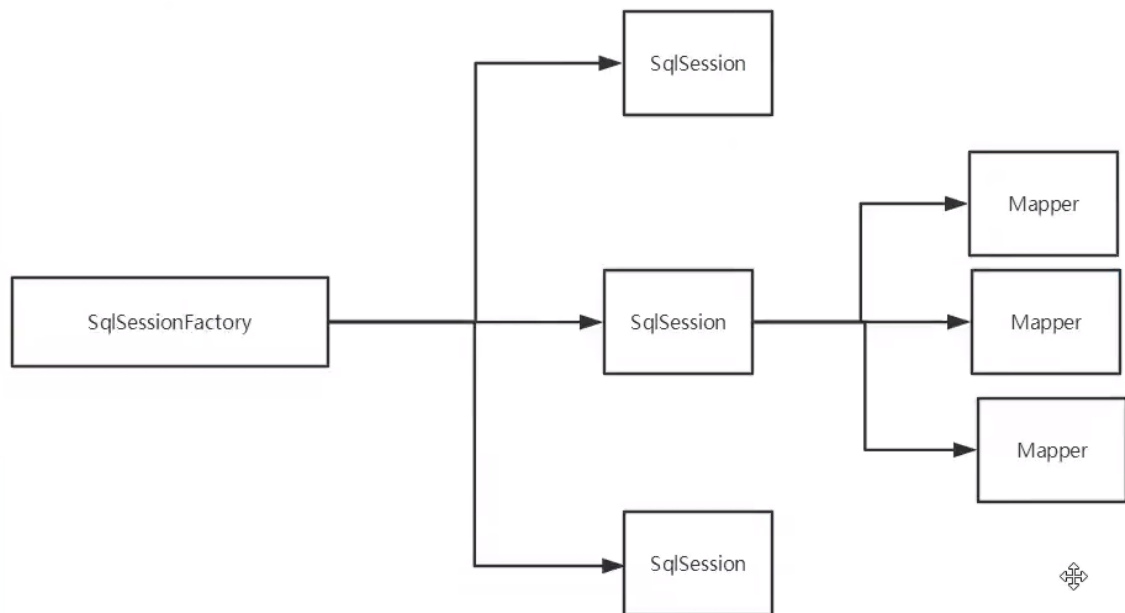
- 一旦创建了SqlSessionFactory，就不再需要它了->局部变量

SqlSessionFactory:

- 可以类比为：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另一个实例**
- 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，所以它的最优的作用域是请求或方法作用域
- 用完之后需要赶紧关闭，否则会占用资源

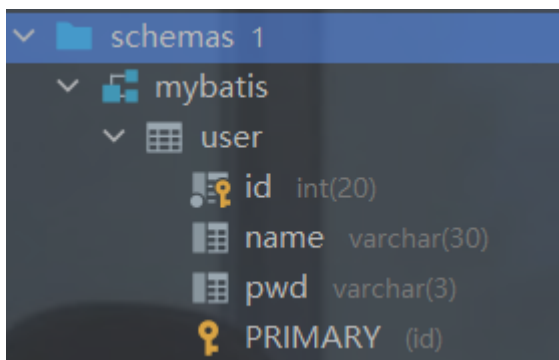


这里的每一个mapper，都代表一个具体业务

5、解决属性名和字段名不一致的问题

1、问题

数据库中的字段：



```
private int id;
private String name;
private String password; //与数据库中的pwd不同
```

```
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：起别名

```
<select id="getUserById" parameterType="int" resultType="User">
    select id,name,pwd as password from mybatis.user where id=#{id}
</select>
```

2、 resultMap

结果集映射

```
<!--结构及映射-->
<resultMap id="UserMap" type="User">
    <!--column为数据库中的字段，property为实体类中的属性-->
    <result column="id" property="id"/>
    <result column="username" property="username"/>
    <result column="pws" property="password"/>
</resultMap>

<select id="getUserById" 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，根本不需要配置显式的结果映射，对于复杂一点的语句，只需要描述语句之间的关系就行了。
- 如果字段名和属性相同，就不用显式的使用它们

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。日志便是最好的助手

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 ☆
- LOG4J2
- DK_LOGGING
- COMMONS_LOGGING
- STDOUT_LOGGING ☆
- NO_LOGGING

在mybatis中具体使用哪一个日志实现，在设置中设定

STDOUT_LOGGING标准日志输出

在mybatis核心配置文件中，配置日志

```
Logging initialized using 'class org.apache.ibatis.logging.stdout.StdOutImpl' adapter.
PooledDataSource forcefully closed/removed all connections.
PooledDataSource forcefully closed/removed all connections.
PooledDataSource forcefully closed/removed all connections.
PooledDataSource forcefully closed/removed all connections.
Opening JDBC Connection
Created connection 1771243284.
Setting autocommit to false on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@69930714]
==> Preparing: select * from mybatis.user where name like ?;
==> Parameters: %猫%(String)
<==      Columns: id, name, pwd
<==      Row: 3, 猫猫, 520
<==      Total: 1
[User{id=3, name='猫猫', pwd='520'}]
Resetting autocommit to true on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@69930714]
Closing JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@69930714]
Returned connection 1771243284 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
- 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，能够更加细致地控制日志的生成过程
- 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/kuang.log
log4j.appender.file.MaxFileSize=10mb
log4j.appender.file.Threshold=DEBUG
log4j.appender.file.layout=org.apache.log4j.PatternLayout
```

```
log4j.appender.file.layout.ConversionPattern=【%p】 【%d{yy-MM-dd}】 【%c】 %m%n

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

3. 设置log4j为日志的实现

```
<settings>
  <!--Log4j日志实现-->
  <setting name="logImpl" value="LOG4J"/>
</settings>
```

4. log4j的使用

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

```
@Test
public void testlog4j(){
    logger.info("info");
    logger.debug("debug");
    logger.error("error");
}
```

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,padgeSize;

select * from user limit 3; #[0,n]
```

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

```
<select id="getUserByLimit" parameterType="map" resultType="pojo.User">
    select * from mybatis.user limit #{startIndex},#{pageSize}
</select>
```

3. 测试

```
@Test
public void getUserByLimit(){
    SqlSession sqlSession = MybatisUtils.getSqlSession();
    UserDao mapper = sqlSession.getMapper(UserDao.class);

    HashMap<String, Integer> map = new HashMap<>();
    map.put("startIndex",0);
    map.put("pageSize",2);

    List<User> userList = mapper.getUserByLimit(map);

    for(User user:userList){
        System.out.println(user);
    }

    sqlSession.close();
}
```

7.2、RowBounds实现分页

不再使用sql实现分页

1. 接口

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

2. mapper.xml

```
<select id="getUserByRowBounds" resultType="pojo.User">
    select * from mybatis.user
</select>
```

3. 测试

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

    //Rowbounds实现
    RowBounds rowBounds = new RowBounds(1, 2);
```

```
//通过java代码层面实现分页
//这里的路径有问题
List<User> userList =
sqlSession.selectList("UserMapper.getUserByRowBounds",null,rowBounds);

for (User user : userList) {
    System.out.println(user);
}

sqlSession.close();
}
```

7.3、分页插件

mybatis pagehelper

和Rowbounds的实现一模一样，了解即可

8.使用注解开发

8.1、面向接口编程

- 真正的开发中，我们选择面向接口开发
- **根本原因：解耦**，可拓展，提高复用，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

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，它对应为一个抽象体（abstract class）
 - 第二类是对一个个体某一方面的抽象，即形成一个抽象面（interface）
- 一个个体有可能有多个抽象面，抽象体与抽象面是有区别的

8.2、使用注解开发

1. 注解在接口上实现

```
public interface UserDao {
    @Select("select * from user")
    List<User> getusers();
}
```

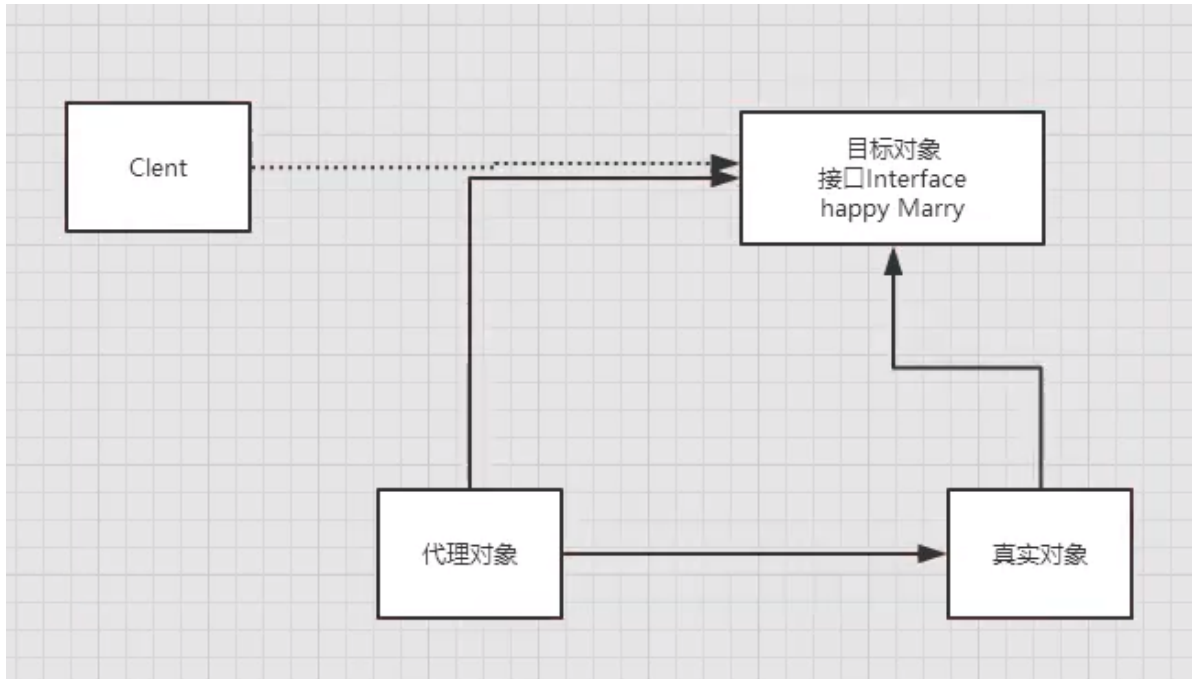
2. 需要在核心配置文件中绑定接口

```
<mappers>
  <mapper class="dao.UserDao"/>
</mappers>
```

3. 测试

本质：反射机制实现

底层：动态代理



mybatis执行流程剖析

8.3、CRUD

我们可以在工具类创建的时候实现自动提交事务

```
public static SqlSession getSqlSession(){
    return sessionFactory.openSession(true);
}
```

openSession里设置为true

编写接口，增加注释

```
//方法存在多个参数，所有的参数前面必须加上@param
@select("select * from user where id = #{id}")
User getUserById(@param("id") int id);
```

测试类

```

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

```

【注意，我们必须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
- 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
- 我们在sql中引用的就是@Param () 中设定的属性名

¥ {}和#{ }的区别

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

```

<!-- https://mvnrepository.com/artifact/org.projectlombok/lombok -->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.12</version>
    <scope>provided</scope>
</dependency>

```

3. @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
```

说明：

@Data: 无参构造、get、set、toString

10、多对一处理

- 多个学生对应一个老师
- 对于学生而言，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】

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
6. 测试查询是否能够成功

按照查询嵌套处理

```
<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，需要单独处理
        对象: association
        集合: collection
    -->
    <association property="teacher" column="tid" javaType="Teacher"
select="getTeacher"/>
</resultMap>
```

```
<select id="getTeacher" resultType="Teacher">
    select * from teacher where id = #{id}
</select>
```

按照结果嵌套处理

```
<!--按照结果嵌套处理-->
<select id="getStudent" resultMap="StudentTeacher">
    select s.id sid,s.name sname,t.name tname
    from student s,teacher t
    where s.tid = t.id
</select>

<resultMap id="StudentTeacher" type="Student">
    <result property="id" column="sid"/>
    <result property="name" column="sname"/>
    <association property="teacher" javaType="Teacher">
        <result property="name" column="tname"/>
    </association>
</resultMap>
```

11、一对多处理

1. 环境搭建

实体类

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

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

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

两种不同的查询方式：

```
<!--按结果嵌套查询-->
<select id="getTeacher" resultMap="TeacherStudent">
    select s.id sid,s.name sname,t.id tid,t.name tname
    from student s,teacher t
    where s.tid = t.id and t.id = #{tid}
</select>
<resultMap id="TeacherStudent" type="Teacher">
```

```
<result property="id" column="tid"/>
<result property="name" column="tname"/>
<!--复杂的属性需要单独处理，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>
```

```
<!--按照查询嵌套处理-->
<select id="getTeacher" resultMap="TeacherStudent">
    select * from mybatis.teacher where id = #{tid}
</select>
<resultMap id="TeacherStudent" type="Teacher">
    <collection property="students" javaType="ArrayList" ofType="Student"
select="getStudent" column="id"/>
</resultMap>
<select id="getStudent" resultType="Student">
    select * from mybatis.student where tid = #{tid}
</select>
```

总结

1. 关联 - association 【多对一】
2. 集合 - collection 【一对多】
3. javaType & ofType
 - javaType用来指定实体类中属性的类型
 - 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的可执行性，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
- 避免慢SQL
- 面试高频：MySQL引擎、InnoDB底层原理、索引、索引优化

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

利用动态sql这一特性可以避免拼接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

```
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
```

创建一个基础工程

1. 导包
2. 编写配置文件
3. 编写实体类
4. 编写实体类对应Mapper接口和Mapper.xml文件

IF

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

```
@Test
public void queryBlogIFTest(){
    SqlSession sqlSession = MybatisUtils.getSqlSession();
    BlogMapper mapper = sqlSession.getMapper(BlogMapper.class);
```



```

HashMap map = new HashMap();
map.put("title", "java如此简单");

List<Blog> blogs = mapper.queryBlogIF(map);
for (Blog blog : blogs) {
    System.out.println(blog);
}
sqlSession.close();
}

```

trim (where , set)

```

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

```

```

<update id="updateBlog" parameterType="map">
    update mybatis.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语句。只是我们可以在SQL层面上去执行一个逻辑代码

choose (when、otherwise)

```

<select id="queryBlogChoose" parameterType="map" resultType="blog">
    select * from mybatis.blog
    <where>
        <choose>
            <when test="title != null">
                title = #{title}
            </when>
            <when test="author != null">
                author = #{author}
            </when>
        </choose>
    </where>
</select>

```

```

        </when>
        <otherwise>
            and views = #{views}
        </otherwise>
    </choose>
</where>
</select>

```

SQL片段

将代码的复用部分进行封装，贴上标签

```

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

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

```

注意事项：

- 最好基于单表来定义sql片段
- 复用部分不要存在where标签

Foreach

```

<!--select * from mybatis.blog where 1=1 and (id=1 or id=2 or id=3)-->
<select id="queryBlogForeach" parameterType="map" resultType="blog">
    select * from mybatis.blog
    <where>
        <foreach collection="ids" item="id" open="and (" close=")"
        separator="or">
            id = #{id}
        </foreach>
    </where>
</select>

```

```

HashMap map = new HashMap();
ArrayList<Integer> ids = new ArrayList<>();
ids.add(1);
ids.add(2);
map.put("ids",ids);
List<Blog> blogs = mapper.queryBlogForeach(map);

```

13、缓存

13.3、一级缓存

缓存失效的情况

1. 查询不同的内容
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

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

小结：一级缓存默认是开启的，只在一次sqlSession中有效，也就是连接到关闭连接这个区间段

一级缓存相当于一个Map

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，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
 - 如果当前会话关闭了，这个会话对应的一级缓存就没有了；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查出的数据会放在自己对应的缓存（map）中

步骤：

1. 开启全局缓存

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

2. 在要使用二级缓存的Mapper中开启

```
<!--在当前Mapper.xml中使用二级缓存-->  
<cache/>
```

也可以自定义参数

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

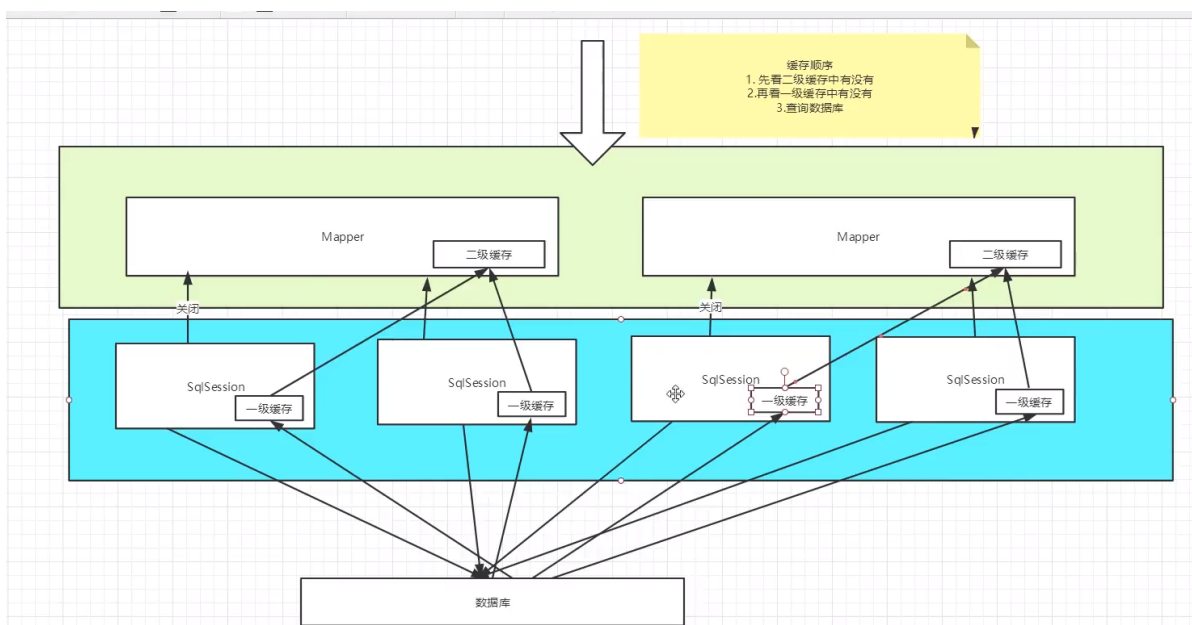
3. 测试

1. 问题：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，只要在同一个Mapper下就有效
- 所有的数据都会先放在一级缓存中，只有当会话提交或关闭时才会提交到二级缓存中

13.5、缓存原理



13.6、自定义缓存-ehcache

要在程序中使用ehcache，先要导包

```
<!-- https://mvnrepository.com/artifact/org.mybatis.caches/mybatis-ehcache -->  
<dependency>  
    <groupId>org.mybatis.caches</groupId>  
    <artifactId>mybatis-ehcache</artifactId>  
    <version>1.2.1</version>  
</dependency>
```

在mapper中指定使用ehcache缓存实现

```
<!--在当前Mapper.xml中使用二级缓存-->
<cache type="org.mybatis.caches.ehcache.EhcacheCache"/>
```

ehcache.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd"
  updateCheck="false">

  <diskStore path="./tmpdir/Tmp_EhCache"/>

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

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

</ehcache>
```

多用Redis数据库做缓存