MyBatis

1. MyBatis简介

1.1 MyBatis

- MyBatis 是一款优秀的**持久层**框架
- MyBatis 避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集的过程
- MyBatis 可以使用简单的XML或注解来配置和映射原生信息，将接口和 Java的实体类 【Plain Old Java Objects,普通的Java对象】映射成数据库中的记录。
- Mybatis官方文档 : http://www.mybatis.org/mybatis-3/zh/index.html
- GitHub : https://github.com/mybatis/mybatis-3

1.2 持久化

持久化是将程序数据在持久状态和瞬时状态间转换的机制。

- **即把数据（如内存中的对象）保存到可永久保存的存储设备中（如磁盘）**。持久化的主要应用是将内存中的对象存储在数据库中，或者存储在磁盘文件中、XML数据文件中等等。
- 内存：**断电即失**
- JDBC就是一种持久化机制。文件IO也是一种持久化机制。
- 在生活中 : 将鲜肉冷藏，吃的时候再解冻的方法也是。将水果做成罐头的方法也是。

为什么需要持久化服务呢？那是由于内存本身的缺陷引起的

- 内存断电后数据会丢失，但有一些对象是无论如何都不能丢失的，比如银行账号等，遗憾的 是，人们还无法保证内存永不掉电。
- 内存过于昂贵，与硬盘、光盘等外存相比，内存的价格要高2~3个数量级，而且维持成本也 高，至少需要一直供电吧。所以即使对象不需要永久保存，也会因为内存的容量限制不能一直呆在内存中，需要持久化来缓存到外存。

1.3 持久层

- 完成持久化工作的代码块 . ----> dao层 【DAO (Data Access Object) 数据访问对象】
- 大多数情况下特别是企业级应用，数据持久化往往也就意味着将内存中的数据保存到磁盘上加以固化，而持久化的实现过程则大多通过各种关系数据库来完成。
- 不过这里有一个字需要特别强调，也就是所谓的“层”。对于应用系统而言，数据持久功能大多 是必不可少的组成部分。也就是说，我们的系统中，已经天然的具备了“持久层”概念？也许 是，但也许实际情况并非如此。之所以要独立出一个“持久层”的概念,而不是“持久模块”，“持久 单元”，也就意味着，我们的系统架构中，应该有一个相对独立的逻辑层面，专著于数据持久 化逻辑的实现.
- 与系统其他部分相对而言，这个层面应该具有一个较为清晰和严格的逻辑边界。 【说白了就 是用来操作数据库存在的！ 】

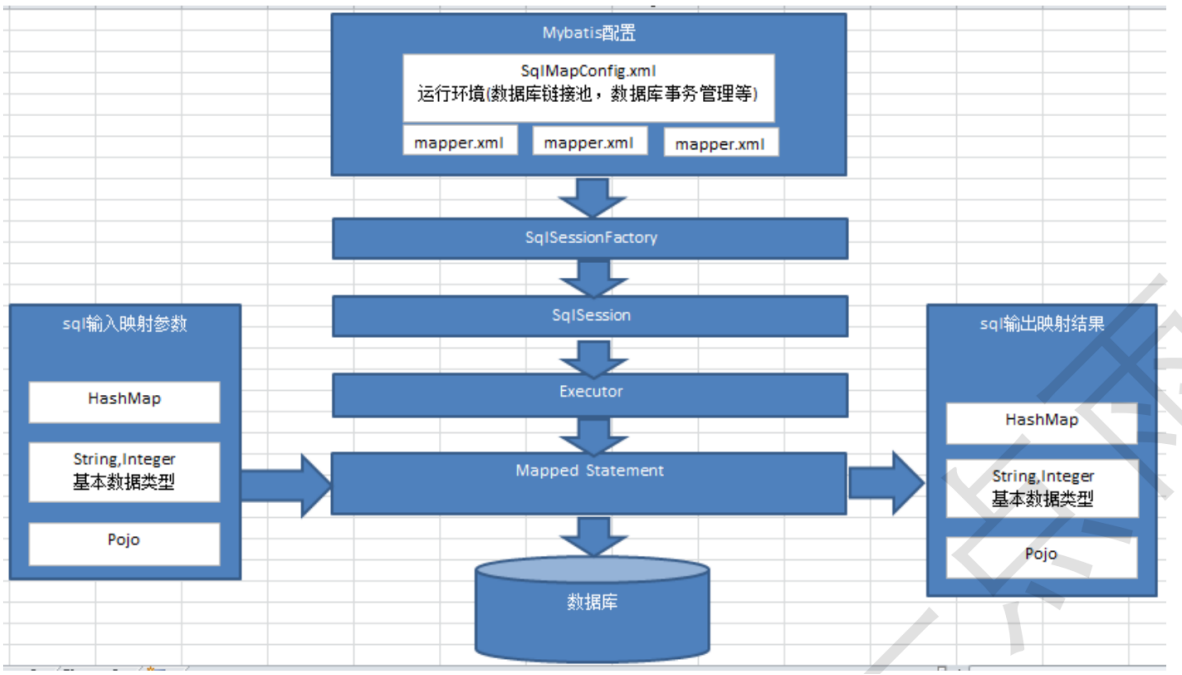
1.4 为什么需要Mybatis

- Mybatis就是帮助程序猿将数据存入数据库中，和从数据库中取数据.
- 传统的jdbc操作, 有很多重复代码块. 比如 : 数据取出时的封装 , 数据库的建立连接等等... , 通过框架可以减少重复代码,提高开发效率 .
- MyBatis 是一个半自动化的ORM框架 (Object Relationship Mapping) -->对象关系映射
- 所有的事情，不用Mybatis依旧可以做到，只是用了它，所有实现会更加简单！技术没有高低之分，只有使用这个技术的人有高低之别

MyBatis的优点

- 简单易学：本身就很小且简单。没有任何第三方依赖，最简单安装只要两个jar文件+配置几个 sql 映射文件就可以了，易于学习，易于使用，通过文档和源代码，可以比较完全的掌握它的设计思路和实现。
- 灵活：mybatis不会对应用程序或者数据库的现有设计强加任何影响。 sql写在xml里，便于统一管理和优化。通过sql语句可以满足操作数据库的所有需求。
- 解除sql与程序代码的耦合：通过提供DAO层，将业务逻辑和数据访问逻辑分离，使系统的设 计更清晰，更易维护，更易单元测试。sql和代码的分离，提高了可维护性。
- 提供xml标，支持编写动态sql

1.5 MyBatis 架构



1. MyBatis: 配置mybatis-config.xml 这个文件是mybatis的全局配置文件，配置了mybatis的运行环境等信息。另外一个具体的mapper.xml是sql的映射文件，文件中配置了操作数据库的sql语句。并且需要在mybatis-config.xml中加载使用
2. 通过mybatis环境等配置信息构造SqlSessionFactory会话工厂
3. 由会话工厂创建sqlSession即会话，我们通过sqlSession来操作数据库
4. mybatis底层自定义了Executor执行器接口操作数据库，Executor接口有两个实现，一个是基本执行器，一个是缓存执行器
5. Mapped Statement是mybatis一个底层的封装对象，它包装了mybatis配置信息及sql映射信息等。mapper.xml文件中的一个sql对应一个Mapped Statement对象，sql的id是mapped statement的id.
6. Mapped Statement对sql执行输入参数进行定义，包括HashMap，基本类型，pojo, Executor通过Mapped Statement执行sql前将输入的java对象映射至sql中，输入参数映射就是jdbc编程中对

preparedStatement设置参数

7. Mapped Statement对sql执行输出结果进行定义，包括HashMap, 基本类型，pojo, Executor通过Mapped Statement执行sql后将输出结果映射至java对象中，输出结果映射过程相当于jdbc编程中对结果的解析处理过程

2. helloMyBatis

思路：**搭建环境->导入MyBatis->编写代码->测试**

2.1 代码

1. 搭建数据库

```
CREATE DATABASE `mybatis`;
USE `mybatis`;
DROP TABLE IF EXISTS `user`;

CREATE TABLE `user` (
    `id` INT(20) NOT NULL,
    `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 (1,'狂神','123456'),(2,'张
三','abcdef'),(3,'李四','987654');
```

2. 新建项目

```
<!--注入依赖-->
<dependencies>
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.5.2</version>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.47</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.13.2</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

3. 编写MyBatis核心配置文件

```
//sqlSessionFactory --> sqlSession
```

```
public class MyBatisUtils {

    private static SqlSessionFactory sqlSessionFactory;

    static{
        try {
            //使用MyBatis获取SqlSessionFactory对象
            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，  我们可以从中获得SqlSession的实例了
    //SqlSession完全包含了面向数据库执行SQL命令所需要的所有方法
    public static SqlSession getSqlSession() {
        SqlSession sqlSession = sqlSessionFactory.openSession();
        return sqlSession;
    }
}
```

4. 编写代码

实体类

```
public class User {
    private int id; //id
    private String name; //姓名
    private String pwd; //密码
    //构造,有参,无参
    //set/get
    //toString()
}
```

Mapper接口类

```
import com.kuang.pojo.User;
import java.util.List;

public interface UserMapper {
    List<User> selectUser();
}
```

编写Mapper.xml文件

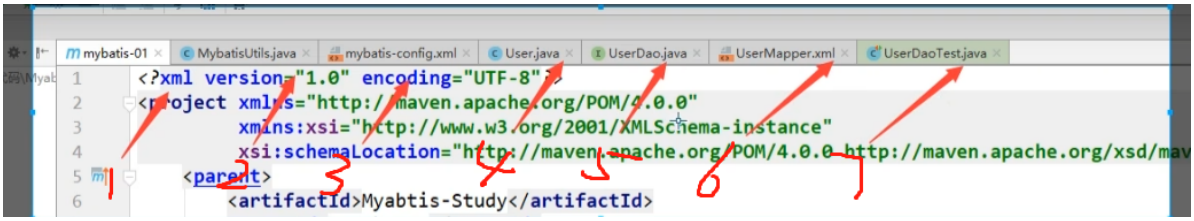
```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<!--namespace绑定一个对应的DAO/Mapper接口-->
<mapper namespace="com.robin.dao.UserMapper">
    <select id="getUserList" resultType="com.robin.pojo.User">
        select * from user
    </select>
</mapper>
```

Test

```java
public class MyTest {
    @Test
    public void selectUser() {
        SqlSession session = MybatisUtils.getSession();
        //方法一:
        //List<User> users =
session.selectList("com.kuang.mapper.UserMapper.selectUser");
        //方法二:
        UserMapper mapper = session.getMapper(UserMapper.class);
        List<User> users = mapper.selectUser();

        for (User user: users){
            System.out.println(user);
        }
        session.close();
    }
}
```

潜在问题： Maven静态资源过滤问题

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



3. CRUD operation

1. namespace

配置文件中namespace中的名称为对应Mapper接口或者Dao接口的完整包名,必须一致!

2. select

select标签是mybatis中最常用的标签之一

select语句有很多属性可以详细配置每一条SQL语句

- id

命名空间中唯一的标识符 接口中的方法名与映射文件中的SQL语句ID, 对应namespace中的方法名

- parameterType

传入SQL语句的参数类型 。【万能的Map，可以多尝试使用】

- resultType

SQL语句返回值类型。【完整的类名或者别名】

根据id查询用户

1. 在UserMapper中添加对应方法

```
public interface UserMapper {
    //查询全部用户
    List<User> selectUser();
    //根据id查询用户
    User selectUserById(int id);
}
```

2. 在UserMapper.xml中添加select语句

```
<select id="selectUserById" resultType="com.robin.pojo.User">
    select * from user where id = #{id}
</select>
```

3. Test

```java
@Test
public void tsetSelectUserById() {
    SqlSession session = MybatisUtils.getSession(); //获取SqlSession连接
    UserMapper mapper = session.getMapper(UserMapper.class);
    User user = mapper.selectUserById(1);
    System.out.println(user);
    session.close();
}
```

3. Insert

```java
//添加一个用户
int addUser(User user);
```

```xml
<insert id="addUser" parameterType="com.robin.pojo.User">
    insert into user (id,name,pwd) values (#{id},#{name},#{pwd})
</insert>
```

```java
@Test
public void testAddUser() {
    SqlSession session = MybatisUtils.getSession();
    UserMapper mapper = session.getMapper(UserMapper.class);
    User user = new User(5,"王五","zxcvbn");
    int i = mapper.addUser(user);
    System.out.println(i);
    session.commit(); //提交事务,重点!不写的话不会提交到数据库
    session.close();
}
```

增、删、改操作需要提交事务!

3.4 Update

修改用户信息

1. 编写接口

```java
//修改一个用户
int updateUser(User user);
```

2. 编写对应的配置文件SQL

```xml
<update id="updateUser" parameterType="com.robin.pojo.User">
    update user set name=#{name},pwd=#{pwd} where id = #{id}
</update>
```

3. Test

```java
@Test
public void testUpdateUser() {
    SqlSession session = MybatisUtils.getSession();
    UserMapper mapper = session.getMapper(UserMapper.class);
    User user = mapper.selectUserById(1);
    user.setPwd("asdfgh");
    int i = mapper.updateUser(user);
    System.out.println(i);
    session.commit(); //提交事务,重点!不写的话不会提交到数据库
    session.close();
}
```

3.5 Delete

1. 编写接口

```java
//根据id删除用户
int deleteUser(int id);
```

2. 编写对应的配置文件SQL

```xml
<delete id="deleteUser" parameterType="int">
    delete from user where id = #{id}
</delete>
```

3. Test

```java
@Test
public void testDeleteUser() {
    SqlSession session = MybatisUtils.getSession();
    UserMapper mapper = session.getMapper(UserMapper.class);
    int i = mapper.deleteUser(5);
    System.out.println(i);
    session.commit(); //提交事务,重点!不写的话不会提交到数据库
    session.close();
}
```

Summary

- 所有的**增删改**操作都需要提交事务!
- 接口所有的普通参数，尽量都写上@Param参数，尤其是多个参数时，必须写上!
- 有时候根据业务的需求，可以考虑使用map传递参数!
- resource绑定mapper, 要使用路径
- 为了规范操作，在SQL的配置文件中，我们尽量将Parameter参数和resultType都写上!

3.6 万能map

```
User selectUserByNP2(Map<String,Object> map);
```

```
<select id="selectUserByNP2" parameterType="map"
    resultType="com.robin.pojo.User">
    select * from user where name = #{username} and pwd = #{pwd}
</select>
```

```
Map<String, Object> map = new HashMap<String, Object>();
map.put("username","小明");
map.put("pwd","123456");
User user = mapper.selectUserByNP2(map);
//在使用方法的时候，Map的 key 为 sql中取的值即可，没有顺序要求
```

如果参数过多，我们可以考虑直接使用Map实现，如果参数比较少，直接传递参数即可

模糊查询

第1种：在Java代码中添加sql通配符。

```
string wildcardname = “%smi%”;
list<name> names = mapper.selectlike(wildcardname);
<select id=”selectlike”>
    select * from foo where bar like #{value}
</select>
```

第2种：在sql语句中拼接通配符，会引起sql注入.

```
string wildcardname = “smi”;
list<name> names = mapper.selectlike(wildcardname);
<select id=”selectlike”>
    select * from foo where bar like "%"#{value}"%"
</select>
```

3.7 @Param

两种默认名字

```
<update id=”selectlike”>
    update user set username = #{arg0} from foo where id = #{arg1}
</update>
<update id=”selectlike”>
    update user set username = #{param1} from foo where id = #{param2}
</update>
```

但是默认的名字并不好记

我们可以通过@Param来指定参数名，一旦指定了参数类型之后，可以省略掉参数类型，也就是说在xml文件中，不用定义parameterType了

```
Integer updateUsernameById(@Param("username") String username, @Param("id")
Integer id);
```

也可以对对象使用@Param

4. 配置解析

4.1 核心配置文件（mybatis-config.xml）

mybatis-config.xml 系统核心配置文件

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）
<!-- 注意元素节点的顺序！顺序不对会报错 -->
```

4.2 enviroments

```
<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>
```

配置MyBatis的多套运行环境，将SQL映射到多个不同的数据库上，必须指定其中一个为默认运行 环境（通过**default**指定）

子元素节点：**environment**

- 具体的一套环境，通过设置id进行区别，id保证唯一!
- 子元素节点：transactionManager - [事务管理器]

```
<!-- 语法 -->
<transactionManager type="[ JDBC | MANAGED ]"/>
```

这两种事务管理器类型都不需要设置任何属性。

- 子元素节点：数据源（dataSource）
  - dataSource 元素使用标准的 JDBC 数据源接口来配置 JDBC 连接对象的资源。
  - 数据源是必须配置的。
  - 有三种内建的数据源类型

```
type="[UNPOOLED|POOLED|JNDI]")
```

- **池: 用完可以回收**
  - unpooled： 这个数据源的实现只是每次被请求时打开和关闭连接。
  - **pooled**： 这种数据源的实现利用“池”的概念将 JDBC 连接对象组织起来 , 这是一种使得 并发 Web 应用快速响应请求的流行处理方式。
  - jndi：这个数据源的实现是为了能在如 Spring 或应用服务器这类容器中使用，容器可以 集中或在外部配置数据源，然后放置一个 JNDI 上下文的引用。
  - 数据源也有很多第三方的实现，比如dbcp，c3p0，druid等等....

4.3 Properties

数据库这些属性都是可外部配置且可动态替换的，既可以在典型的 Java 属性文件中配置，亦可通过 properties 元素的子元素来传递。【db,.properties】

通过properties来实现引用配置文件

顺序:

首先读取properties元素体内指定的属性

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

最低优先级的是properties元素中指定的属性

1. 编写一个配置文件

useSSL=false

```
driver=com.mysql.jdbc.Driver
url=jdbc:mysql://localhost:3306/mybatis?
useSSL=false&useUnicode=true&characterEncoding=utf8
username=root
password=991214
```

2. 核心文件中引入properties 配置文件

```
<configuration>
    <!--导入properties文件-->
    <properties resource="db.properties"/>

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

可以直接引入外部文件，也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

4.4 typeAliases

类型别名是为 Java 类型设置一个短的名字。它只和 XML 配置有关，存在的意义仅在于用来减少类完全限定名的冗余。

```
<!--配置别名,注意顺序, no.3-->
<typeAliases>
    <typeAlias alias="user" type="com.robin.pojo.User" alias = "user">
</typeAlias>
</typeAliases>
```

也可以指定一个包名，MyBatis 会在包名下面搜索需要的 Java Bean, 批量定义，通过扫描包来做，批量定义默认的类的别名，是类名首字母小写

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

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

```
@Alias("hello")
public class User {
...
}
```

Alias不区分大小写

4.5 setting

- 懒加载 lazyLoadingEnabled
- 日志实现 logImpl
- 缓存开启关闭 cacheEnabled

4.6 Mapper

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

</mapper>
```

namespace中文意思：命名空间

- namespace和子元素的id联合保证唯一 , 区别不同的mapper
- 绑定DAO接口
    1. **namespace的命名必须跟某个接口同名** （类路径原则）
    2. **接口中的方法与映射文件中sql语句id应该一一对应**
- namespace命名规则 : **包名+类名**

MyBatis 的真正强大在于它的映射语句，这是它的魔力所在。由于它的异常强大，映射器的 XML 文件就显得相对简单。如果拿它跟具有相同功能的 JDBC 代码进行对比，你会立即发现省掉了将近 95% 的代码。MyBatis 为聚焦于 SQL 而构建，以尽可能地为你减少麻烦。

如果使用class/扫描包进行注入绑定：

- 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

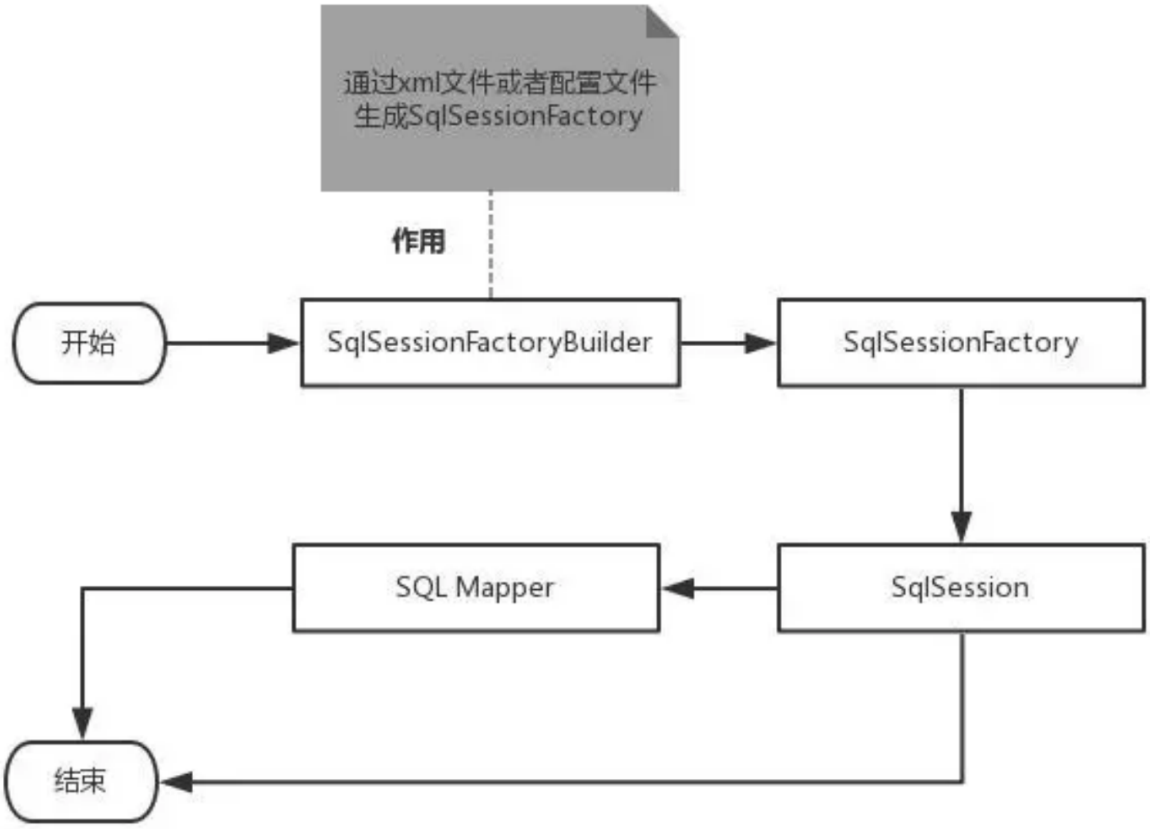
如果想分离，只需要在resource下建和接口所在相同的包

而mybatis的resource是通过classpath来找文件的

4.7 生命周期及作用域 lifetime&scope

理解我们目前已经讨论过的不同作用域和生命周期类是至关重要的，因为错误的使用会导致非常严重的**并发问题。**

MyBatis执行过程：



- **SqlSessionFactoryBuilder**：

创建 SqlSessionFactory，创建成功后， SqlSessionFactoryBuilder 就失去了作用，所以它只能存在于创建 SqlSessionFactory 的方法中， 而不要让其长期存在。因此**SqlSessionFactoryBuilder 实例的最佳作用域是方法作用域**（也就是： 局部方法变量）。

- **SqlSessionFactory**

理解为数据库连接池，它的作用是创建 SqlSession 接口对象。因为 **MyBatis 的本质就是 Java 对数据库的操作**，所以 SqlSessionFactory 的生命周期存在于整个 MyBatis 的应用之中，所以一旦创建了 SqlSessionFactory，就要长期保存它，直至不再使用 MyBatis 应用，所以可以认为 **SqlSessionFactory 的生命周期就等同于 MyBatis 的应用周期。**

由于 SqlSessionFactory 是一个对数据库的连接池，所以它占据着数据库的连接资源。如果创建多 个 SqlSessionFactory，那么就存在多个数据库连接池，这样不利于对数据库资源的控制，也会导 致数据库连接资源被消耗光，出现系统宕机等情况，所以尽量避免发生这样的情况。

因此在一般的应用中我们往往希望 SqlSessionFactory 作为一个单例，**让它在应用中被共享。所以说 SqlSessionFactory 的最佳作用域是应用作用域。**

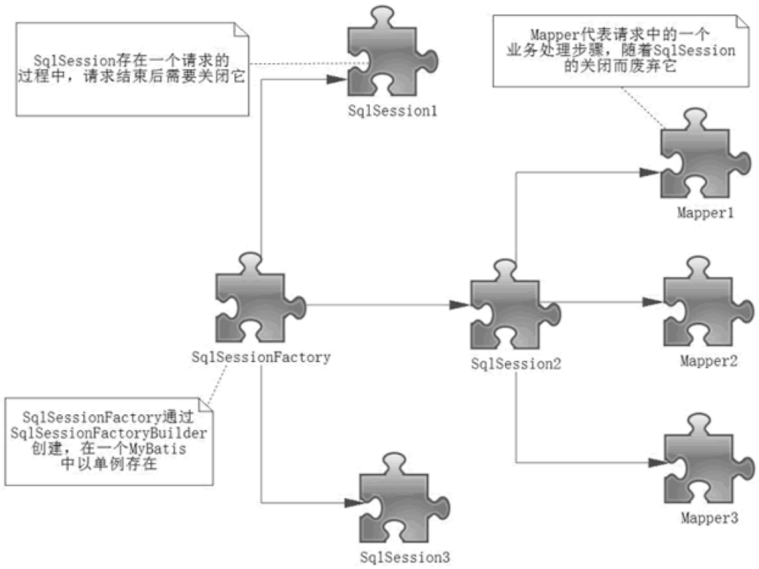
- **SqlSession**

如果说 SqlSessionFactory 相当于数据库连接池，那么 SqlSession 就相当于一个数据库连接（Connection 对象），你可以在一个事务里面执行多条 SQL，然后通过它的 commit、rollback 等方法，提交或者回滚事务。**需要开启和关闭。**

如果你现在正在使用一种 Web 框架，考虑将 SqlSession 放在一个和 HTTP 请求相似的作用域中。 换句话说，每次收到 HTTP 请求，就可以打开一个 SqlSession，返回一个响应后，就关闭它。 这个关闭操作很重要，为了确保每次都能执行关闭操作，你应该把这个关闭操作放到 finally 块中。

所以它应该存活在一个业务请求中，处理完整个请求后，应该关闭这条连接，让它归还给 SqlSessionFactory，否则数据库资源就很快被耗费精光，系统就会瘫痪，所以用 **try…catch…finally…** 语句来保证其正确关闭。

所以 SqlSession 的最佳的作用域是请求或方法作用域。



这里面的每一个mapper都代表一个具体的应用业务

5. ResultMap

属性名和字段名不一致

```
private int id;
private String name;
private String passwowrd;
```

```
Tests passed: 1 of 1 test – 310 ms
"C:\Program Files\Java\jdk1.8.0_131\bin\java.exe" ...
User{id=1, name='王五', passwowrd='null'}
```

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

解决方案：

1. Alias

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

2. ResultMap

结果集映射

resultMap 元素是 MyBatis 中最重要最强大的元素。

它可以让你从 90% 的 JDBC ResultSets 数据提取代码中解放出来。 实际上，在为一些比如连接的复杂语句编写映射代码的时候，一份 resultMap 能够代替实现同 等功能的长达数千行的代码。

ResultMap 的设计思想是，对于简单的语句根本不需要配置显式的结果映射，而对于复杂一点的语句只需要描述它们的关系就行了。

自动映射

```
<select id="selectUserById" resultType="map">
select id , name , pwd
from user
where id = #{id}
</select>
```

上述语句只是简单地将所有的列映射到 HashMap 的键上，这由 resultType 属性指定。

虽然在 大部分情况下都够用，但是 HashMap 不是一个很好的模型。

你的程序更可能会使用 JavaBean 或 POJO（Plain Old Java Objects，普通老式 Java 对象）作为模型。

ResultMap 最优秀的地方在于，虽然你已经对它相当了解了，但是根本就不需要显式地用到他们。

id为主键，column是数据库查出来的列名，property是对应实体类(对象)的属性名

手动映射

1. 使用resultMap

```
<select id="selectUserById" resultMap="UserMap">
    select id , name , pwd from user where id = #{id}
</select>
```

2. 构造resultMap

```
<resultMap id="UserMap" type="User">
    <!-- id为主键 -->
    <id column="id" property="id"/>
    <!-- column是数据库查出来的列名 ， property是对应实体类(对象)的属性名 -->
    <result column="name" property="name"/>
    <result column="pwd" property="password"/>
</resultMap>
```

6. 日志

6.1 日志工厂

如果一个 数据库相关的操作出现了问题，我们可以根据输出的SQL语句快速排查问题。

对于以往的开发过程，我们会经常使用到debug模式来调节，跟踪我们的代码执行过程。但是现在使用Mybatis是基于接口，配置文件的源代码执行过程。因此，我们必须选择日志工具来作为我们开发，调节程序的工具。

Mybatis内置的日志工厂提供日志功能，具体的日志实现有以下几种工具：

- SLF4J
- Apache Commons Logging
- Log4j 2
- Log4j
- JDK logging

具体选择哪个日志实现工具由MyBatis的内置日志工厂确定。它会使用最先找到的（按上文列举的顺序查找）。 如果一个都未找到，日志功能就会被禁用。

实现

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

6.2 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：控制台，文本，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. 先导包

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

2. 编写配置文件(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. setting设置日志实现

```
<settings>
    <setting name="logImpl" value="LOG4J"/>
</settings>
```

4. 程序中使用

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

@Test
public void selectUser() {
    logger.info("info: 进入selectUser方法");
    logger.debug("debug: 进入selectUser方法");
    logger.error("error: 进入selectUser方法");
    SqlSession session = MyBatisUtils.getSqlSession();
    UserMapper mapper = session.getMapper(UserMapper.class);
    List<User> users = mapper.getUserList();
    for (User user: users){
        System.out.println(user);
    }
    session.close();
}
```

使用Log4j 输出日志

可以看到还生成了一个日志的文件 【需要修改file的日志级别】

###

7. 分页

思考：为什么需要分页?

在学习mybatis等持久层框架的时候，会经常对数据进行增删改查操作，使用最多的是对数据库进行查询操作，如果查询大量数据的时候，我们往往使用分页进行查询，也就是每次处理小部分数据，这样对数据库压力就在可控范围内。

Limit实现分页

```
#语法
SELECT * FROM table LIMIT stratIndex, pageSize

SELECT * FROM table LIMIT 5,10; // 检索记录行 6-15

#为了检索从某一个偏移量到记录集的结束所有的记录行，可以指定第二个参数为 -1:
SELECT * FROM table LIMIT 95,-1; // 检索记录行 96-last.
//现在好像不行了

#如果只给定一个参数，它表示返回最大的记录行数目:
SELECT * FROM table LIMIT 5; //检索前 5 个记录行
#换句话说，LIMIT n 等价于 LIMIT 0,n。
```

实现

1. 修改mapper

```
<select id="getUserByLimit" parameterType="map" resultType="User"
resultMap="UserMap">
    select * from user limit #{startIndex}, #{endIndex}
</select>
```

2. 修改接口

```
//选择全部用户实现分页
List<User> getUserByLimit(Map<String, Integer> map);
```

3. test

```
public void getUserByLimit(){
    SqlSession sqlSession = MyBatisUtils.getSqlSession();
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);

    HashMap<String, Integer> map  = new HashMap<>();
    map.put("startIndex", 0);
    map.put("endIndex", 2);
    List<User> userByLimit = mapper.getUserByLimit(map);
    for (User user : userByLimit) {
        System.out.println(user);
    }

    sqlSession.close();
}
```

PageHelper

https://pagehelper.github.io/

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。

接口

- 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一方面的抽象，即形成一个抽象面（interface）；
- 一个体有可能有多个抽象面。抽象体与抽象面是有区别的。

三种面向

- 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

sql 类型主要分成 :

@select ()

@update ()

@Insert ()

@delete ()

1. 添加注解

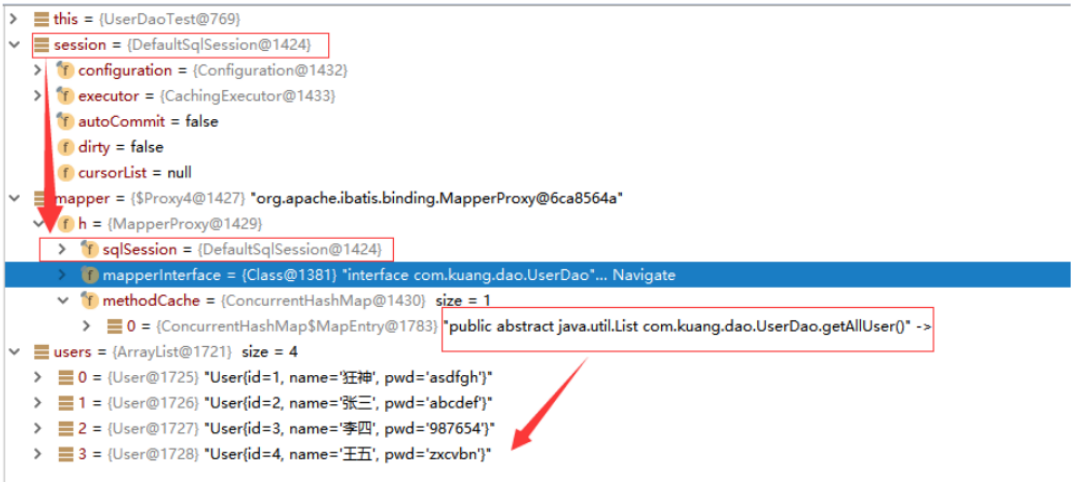
```
//查询全部用户
@Select("select id,name,pwd password from user")
public List<User> getAllUser();
```

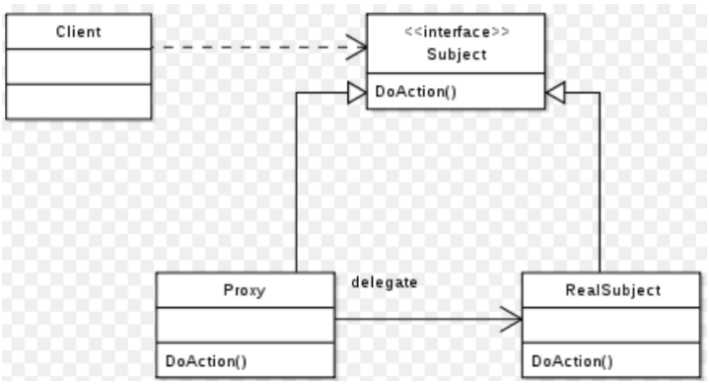
2. 在核心配置文件中注入

```
<!--使用class绑定接口-->
<mappers>
    <mapper class="com.kuang.mapper.UserMapper"/>
</mappers>
```

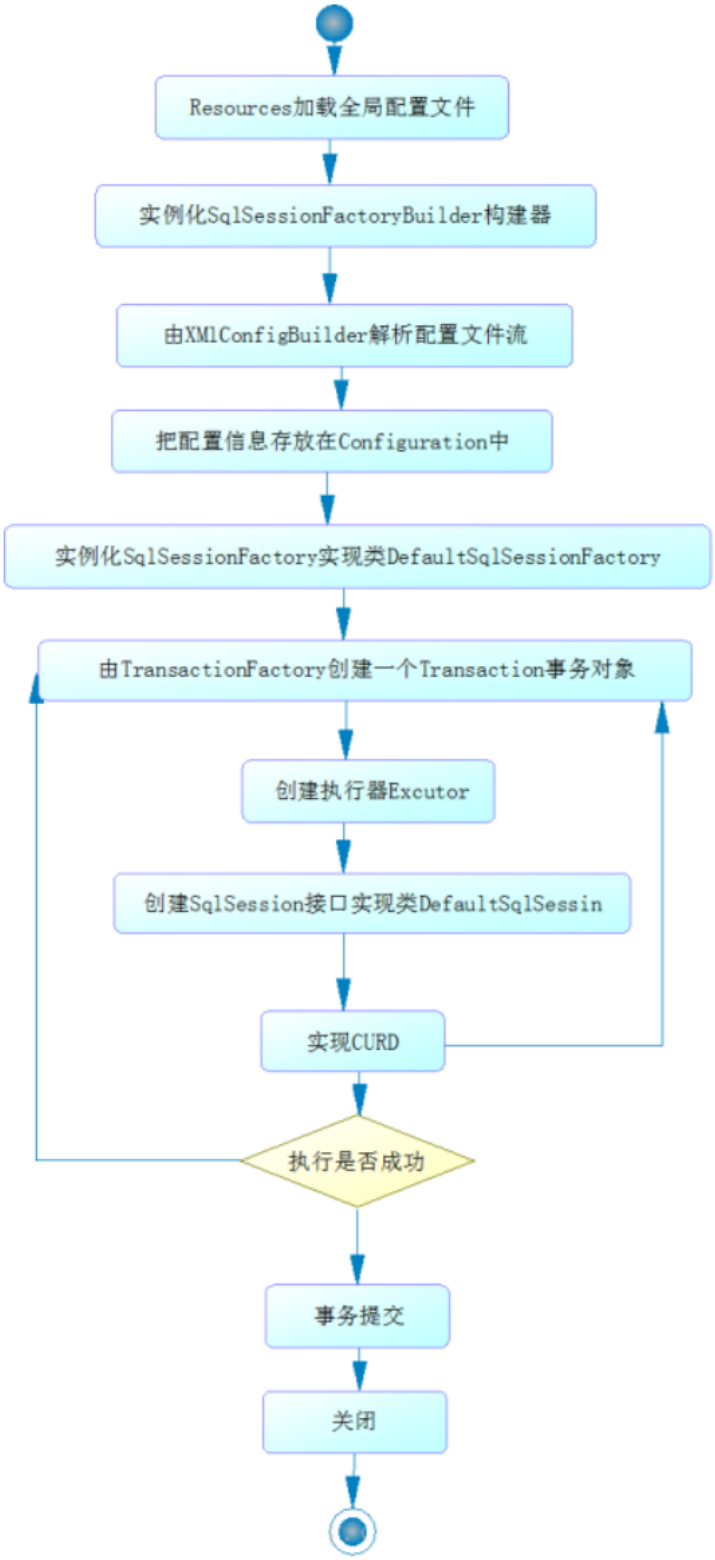
3. 测试

```
@Test
public void testGetAllUser() {
    SqlSession session = MybatisUtils.getSession();
    //本质上利用了jvm的动态代理机制，反射
    UserMapper mapper = session.getMapper(UserMapper.class);

    List<User> users = mapper.getAllUser();
    for (User user : users){
        System.out.println(user);
    }
    session.close();
}
```


this = {UserDaoTest@769}
session = {DefaultSqlSession@1424}
configuration = {Configuration@1432}
executor = {CachingExecutor@1433}
autoCommit = false
dirty = false
cursorList = null
mapper = {$Proxy4@1427} "org.apache.ibatis.binding.MapperProxy@6ca8564a"
h = {MapperProxy@1429}
sqlSession = {DefaultSqlSession@1424}
mapperInterface = {Class@1381} "interface com.kuang.dao.UserDao"... Navigate
methodCache = {ConcurrentHashMap@1430} size = 1
0 = {ConcurrentHashMap$MapEntry@1783} "public abstract java.util.List com.kuang.dao.UserDao.getAllUser()" ->
users = {ArrayList@1721} size = 4
0 = {User@1725} "User{id=1, name='狂神', pwd='asdfgh'}"
1 = {User@1726} "User{id=2, name='张三', pwd='abcdef'}"
2 = {User@1727} "User{id=3, name='李四', pwd='987654'}"
3 = {User@1728} "User{id=4, name='王五', pwd='zxcvbn'}"


Client
<<interface>>
Subject
DoAction()
Proxy
delegate
RealSubject
DoAction()
DoAction()

流程



8.3 CURD

改造MybatisUtils工具类的getSession( ) 方法，重载实现

```
public class User {
    private int id;
    private String name;
    private String password;
}
```

方法存在多个参数，所有参数前面必须加上@Param("id") 注解，要通过注解指定的类型来获取值

```java
@Select("select * from user")
List<User> getUsers();

@Select("select * from user where id = #{id}")
User getUserById(@Param("id") int id);

@Insert("insert into user(id, name, pwd) values (#{id}, #{name}, #{password})")
int addUser(User user);

@Update("update user set name = #{name}, pwd = #{password} where id = #{id}")
int updateUser(User user);

@Delete("delete from user where id = #{uid}")
int deleteUser(@Param("uid") int id);
```

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

增删改一定记得对事务的处理

@Param注解用于给方法参数起一个名字。

使用原则：

- 在方法只接受一个参数的情况下，可以不使用@Param。
- 在方法接受多个参数的情况下，建议一定要使用@Param注解给参数命名。
- 如果参数是 JavaBean ， 则不能使用@Param。
- SQL中引用的就是里面设置的属性名
- 不使用@Param注解时，参数只能有一个，并且是Javabean。

6.3 $ vs

#{} 的作用主要是替换预编译语句(PrepareStatement)中的**占位符**? 【推荐使用】

有预编译

```sql
INSERT INTO user (name) VALUES (#{name});
INSERT INTO user (name) VALUES (?);
```

${} 的作用是直接进行字符串替换, 相当于 **参数拼接的方式(有sql注入的风险)**

比如要传入一个动态字段，但是字段不确定的情况下，需要通过参数传入，这个时候就只能使用$

```sql
INSERT INTO user (name) VALUES ('${name}');
INSERT INTO user (name) VALUES ('kuangshen');
```

9. Lombok

Project Lombok is a java library that automatically plugs into your editor and build tools, spicing up your java.
Never write another getter or equals method again, with one annotation your class has a fully featured builder, Automate your logging variables, and much more.

https://projectlombok.org/features/all

使用步骤

1. 在iead中安装插件
2. 项目中导入jar包

```
<dependencies>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.18.22</version>
        </dependency>
    </dependencies>
```

3. 在实体类上加注解即可

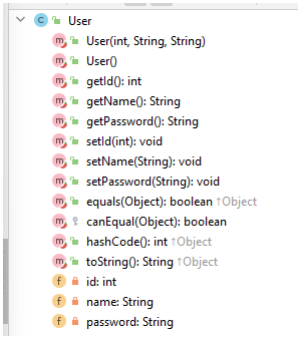
@Data: 无参构造，get, set, toString, equals, hashcode

@AllArgsConstructor

@NoArgsConstructor

@EqualsAndHashCode

@ToString



也可以自己加构造方法，注解本身不支持多种构造方法

10. 多对一的处理

多对一的理解：

多个学生对应一个老师 如果对于学生这边，就是一个多对一的现象，即从学生这边关联一个老师！

10.1 搭建数据库

```sql
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);
```

10.2 搭建测试环境

Student and Teacher class:

```java
package com.robin.pojo;
@Data // Introduce parameterless construction 、get、set、toString Other methods
@AllArgsConstructor // The parametric construction method is introduced
@NoArgsConstructor // Introduce the parameterless construction method
public class Teacher {
    private int id; // Teacher number
    private String name; // Teacher's name
}
```

```java
package com.robin.pojo;
@Data // Introduce parameterless construction 、get、set、toString Other methods
@AllArgsConstructor // The parametric construction method is introduced
@NoArgsConstructor // Introduce the parameterless construction method
public class Student {
    private int id; // Student number
    private String name; // The student's name
    // Students need to be associated with a teacher
    private Teacher teacher;
}
```

MyBatisUtils Class

```java
package com.robin.utils;
/** * SqlSessionFactoryBuilder( Build a factory ) * --> sqlSessionFactory(
production sqlSession) * --> sqlSession */

public class MybatisUtils {
```

```
    // Get static SQL Conversational factory
    private static SqlSessionFactory sqlSessionFactory;
    // Static method body
    static {
        try {
            // Read configuration file
            String resource = "mybatis-config.xml";
            // Parsing configuration file stream
            InputStream inputStream = Resources.getResourceAsStream(resource);
            // Get factory
            sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    /** * SqlSession Provides execution in the database SQL All methods required
for the command */
    public static SqlSession getSqlSession() {
        // Set the parameter to true, Realize automatic submission
        return sqlSessionFactory.openSession(true);
    }
}
```

Mapper Interface

```
package com.robin.dao;
import com.robin.pojo.Teacher;
import org.apache.ibatis.annotations.Param;
import org.apache.ibatis.annotations.Select;
public interface TeacherMapper {
/** * Use @Select annotation : Through users id Query specific user information *
( Be careful : SQL Statement id And @Param The agreement in the notes ) */
    @Select("Select * from teacher where id=#{tid}")
    Teacher getTeacher(@Param("tid") int id);
}
```

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

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

Core Config

```
<?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: Core profile -->
<configuration>
```

```
    <!-- Import external profile , Take advantage of external configuration files
-->
    <properties resource="db.properties"/>

    <!-- Set standard log output -->
    <settings>
        <setting name="logImpl" value="STDOUT_LOGGING"/>
    </settings>

    <!-- By aliasing the package -->
    <typeAliases>
        <package name="com.kuang.pojo"/>
    </typeAliases>

    <!-- Set the default environment to the development environment -->
    <environments default="development">
        <!-- Set an environment as the development environment -->
        <environment id="development">
            <!-- transactionManager: Represents a transaction manager , and
MyBatis The default manager is JDBC-->
            <transactionManager type="JDBC"/>
            <!-- dataSource: Presentation data source , The main role : Connect
to database , MyBatis The default data source type is POOLED, That is, there is a
pool connection -->
            <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>

    <!-- Binding interface : Use class File binding registration -->
    <mappers>
        <mapper class="com.robin.dao.TeacherMapper"/>
        <mapper class="com.robin.dao.StudentMapper"/>
    </mappers>
</configuration>
```

10.3 Improve the complex environment and build test

1. **按照查询嵌套处理**

```
package com.robin.dao;
public interface StudentMapper {
    /** * Query all student information , And the corresponding teacher
information */
    // Nested by query
    public List<Student> getStudent();
    // Nested according to the result
    public List<Student> getStudent2();
}
```

Nested by query

需求：获取所有学生及对应老师的信息

思路：

1. 获取所有学生的信息
2. 根据获取的学生信息的老师ID->获取该老师的信息
3. 思考问题，这样学生的结果集中应该包含老师，该如何处理呢，数据库中我们一般使用关联查询?
4. 做一个结果集映射：StudentTeacher
5. StudentTeacher结果集的类型为 Student
6. 学生中老师的属性为teacher，对应数据库中为tid。多个 [1,...）学生关联一个老师=> 一对一，一对多
7. 查看官网找到：association – 一个复杂类型的关联；使用它来处理关联查询

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

    <!-- 查询所有的学生信息 -->
    <select id="getStudent" resultMap="StudentTeacher">
        Select * from student
    </select>

    <!-- 结果映射集: 其中id对应select标签中resultMap的值, type对应Java实体类 -->
    <resultMap id="StudentTeacher" type="Student">
        <!-- column: 表示数据库中的字段名, property: 表示Java实体类中的属性名
             注意: 只需要显式定义不一致的字段和属性即可 -->
        <result column="id" property="id"/>
        <result column="name" property="name"/>
        <!-- 复杂的属性，需要单独处理，其中association代表对象，而collection代表集合;
             property的值teacher是对象，因此需要获取对象Teacher，同时property又是一个复杂类型，需要给它一个类型，因此使用javaType，值为对象Teacher;
             要让数据库列tid查出来是teacher，因此要使用嵌套查询select -->
        <association property="teacher" column="tid" javaType="Teacher" select="getTeacher"/>
    </resultMap>
    <!--
    这里传递过来的id，只有一个属性的时候，下面可以写任何值
    association中column多参数配置:
    column="{key=value,key=value}"
    其实就是键值对的形式，key是传给下个sql的取值名称，value是片段一中sql查询的字段名。
    -->
    <!-- 通过id查询指定的老师信息 -->
    <select id="getTeacher" resultType="Teacher">
        Select * from teacher where id = #{id}
    </select>

</mapper>
```

association：对象

collection：集合

property：注入给实体类的某个属性

column：在上次查询结果集中，用哪些列值作为条件去执行下一条SQL语句

javaType：把SQL语句查询出来的结果集，封装到某个类的对象(可以省略)

select：下一条要执行的SQL语句

2. **按照结果嵌套处理**

```
<!--
按查询结果嵌套处理
思路:
1. 直接查询出结果，进行结果集的映射
-->

 <!-- 按照结果嵌套处理 -->
    <select id="getStudent2" resultMap="StudentTeacher2">
         Select s.id as sid,s.name as sname,t.name as tname
         from student s,teacher t
         where  s.tid = t.id
    </select>

    <!-- 结果集映射 -->
    <resultMap id="StudentTeacher2" type="Student">
        <!-- property表示Java类属性名，而column表示数据库列名-->
        <result property="id" column="sid"/>
        <result property="name" column="sname"/>
        <!-- 复杂类型，需单独处理，这里使用association，代表集合;
             其他属性，如property表示注入实体类Student的属性teacher;
             javaType则表示返回的结果集，封装在Teacher类中-->
        <association property="teacher"  javaType="Teacher">
            <!-- 嵌套一个result
                 property表示Teacher类的属性名，例如name
                 column表示对应数据库字段别名，例如tname -->
            <result property="name" column="tname"/>
        </association>
    </resultMap>
```

Id 和 Result 的属性

属性	描述
`property`	映射到列结果的字段或属性。如果 JavaBean 有这个名字的属性（property），会先使用该属性。否则 MyBatis 将会寻找给定名称的字段（field）。 无论是哪一种情形，你都可以使用常见的点式分隔形式进行复杂属性导航。 比如，你可以这样映射一些简单的东西：“username”，或者映射到一些复杂的东西上：“address.street.number”。
`column`	数据库中的列名，或者是列的别名。一般情况下，这和传递给 `resultSet.getString(columnName)` 方法的参数一样。
`javaType`	一个 Java 类的全限定名，或一个类型别名（关于内置的类型别名，可以参考上面的表格）。 如果你映射到一个 JavaBean，MyBatis 通常可以推断类型。然而，如果你映射到的是 HashMap，那么你应该明确地指定 javaType 来保证行为与期望的相一致。
`jdbcType`	JDBC 类型，所支持的 JDBC 类型参见这个表格之后的“支持的 JDBC 类型”。 只需要在可能执行插入、更新和删除的且允许空值的列上指定 JDBC 类型。这是 JDBC 的要求而非 MyBatis 的要求。如果你直接面向 JDBC 编程，你需要对可以为空值的列指定这个类型。
`typeHandler`	我们在前面讨论过默认的类型处理器。使用这个属性，你可以覆盖默认的类型处理器。 这个属性值是一个类型处理器实现类的全限定名，或者是类型别名。

Mysql多对一查询方式

- 子查询

```
Select id,name,tid
from student
where tid= (
    select ...);
```

- 联表查询

```
Select s.id,s.name,t.name
from student s,teacher t
where  s.tid = t.id
```

11 一对多

一对多的理解：

一个老师拥有多个学生 如果对于老师这边，就是一个一对多的现象，即从一个老师下面拥有一群学生（集合）！

11.1 实体类

```
package com.robin.pojo;
@Data // 引入无参构造、get、set、toString等方法
@AllArgsConstructor // 引入有参构造方法
@NoArgsConstructor // 再次引入无参构造方法，防止被有参构造覆盖
public class Student {
    private int id;
    private String name;
    private int tid;
}
```

```
package com.robin.pojo;

@Data // 引入无参构造、get、set、toString等方法
@AllArgsConstructor // 引入有参构造方法
@NoArgsConstructor // 再次引入无参构造方法，防止被有参构造覆盖
public class Teacher {
    private int id; // 教师编号
    private String name; // 教师姓名
    // 一个老师拥有多个学生，使用泛型为Student的List集合来实现一对多
    private List<Student> students;
}
```

11.2 Mapper & Interface

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

    <!-- 查询所有的老师信息 -->
    <select id="getTeacher" resultType="Teacher">
```

```
        Select * from mybatis.teacher
    </select>


</mapper>
```

```
public interface TeacherMapper {
    // 获取所有老师信息
    List<Teacher> getTeacher();
    /**
     * 获取指定老师下的所有学生及老师的信息
     */
    // 使用结果嵌套处理
    Teacher getTeacher2(@Param("tid") int id);

    // 使用查询嵌套处理
    Teacher getTeacher3(@Param("tid") int id);
}
```

11.3 按照结果嵌套处理

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

    <!-- 查询指定老师下的所有学生及老师信息 -->
    <!-- 按照结果嵌套查询 -->
    <select id="getTeacher2" 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结果集 -->
    <resultMap id="TeacherStudent" type="Teacher">
        <result property="id" column="tid"/>
        <result property="name" column="tname"/>
        <!-- 复杂属性，需要单独处理;
             其中association: 代表对象; collection: 代表集合
             javaType=""指定属性的类型; 而集合中的泛型信息，要使用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>
```

11.4 查询嵌套处理

```
    <!-- 按照查询嵌套处理 -->
    <select id="getTeacher3" resultMap="TeacherStudent2">
        Select * from mybatis.teacher where id = #{tid}
    </select>

    <!-- 结果集映射 -->
    <resultMap id="TeacherStudent2" type="Teacher">
        <!-- collection: 表示集合，一个老师包含多个学生，即一对多关系;
             property="students": 用于注入给Teacher实体类的属性students;
             javaType="ArrayList": 用于指定属性students的类型为ArrayList;
             ofType="Student": 集合中的泛型信息，使用ofType获取，这里的集合是值
List<Student>，因此值为Student
             select="getStudentByTeacherId": 下一条要执行的SQL语句，这里指下面的查询指
定老师的所有学生信息，column="id": 通过id查找对应的老师 -->
        <collection property="students" javaType="ArrayList" ofType="Student"
select="getStudentByTeacherId" column="id"/>
    </resultMap>

    <!-- 查询指定老师的所有学生信息 -->
    <select id="getStudentByTeacherId" resultType="Student">
        Select * from mybatis.student where tid = #{tid}
    </select>

</mapper>
```

11.5 summary

1. 关联-association
2. 集合-collection
3. 所以association是用于一对一和多对一，而collection是用于一对多的关系
4. JavaType和ofType都是用来指定对象类型的
    - JavaType是用来指定pojo中属性的类型
    - ofType指定的是映射到list集合属性中pojo的类型。

注意说明：

1. 保证SQL的可读性，尽量通俗易懂
2. 根据实际要求，尽量编写性能更高的SQL语句
3. 注意属性名和字段不一致的问题
4. 注意一对多和多对一 中：字段和属性对应的问题
5. 尽量使用Log4j，通过日志来查看自己的错误

12. 动态SQL

动态SQL指的是根据不同的查询条件，生成不同的Sql语句.

利用动态 SQL，可以彻底摆脱拼接时要确保忘记添加必要的空格和注意去掉列表最后一个列名的逗号等问题。

使用 mybatis 动态SQL，通过 **if, choose, when, otherwise, trim, where, set, foreach**等标签，可组合成非常灵活的SQL语句，从而在提高 SQL 语句的准确性的同 时，也大大提高了开发人员的效率。

```
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
```

创建基础工程：

1. 导入包
2. 编写配置文集
3. 编写实体类
4. 编写实体类对应的mapper接口和mapper xml文件

```
package com.robin.pojo;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import java.util.Date;

// 使用@Data注解，引入无参构造、get、set、toString等方法
@Data
// 使用@AllArgsConstructor注解，引入有参构造方法
@AllArgsConstructor
// 使用@NoArgsConstructor注解，引入无参构造方法
@NoArgsConstructor
public class Blog {

    private String id; // 博客id
    private String title; // 博客标题
    private String author; // 作者
    private Date createTime; // 创建时间
    private int views; // 浏览量

}
```

如何设置数据库

```
import com.robin.dao.blogMapper;
import com.robin.pojo.Blog;
import com.robin.utils.IDutils;
import com.robin.utils.MyBatisUtils;
import org.apache.ibatis.session.SqlSession;
import org.junit.Test;
import java.util.Date;

public class MyTest {

    @Test
```

```
    public void addBlog() {

        // 获取sqlSession对象
        SqlSession sqlSession = MyBatisUtils.getSqlSession();
        // 获取Mapper接口对象
        blogMapper mapper = sqlSession.getMapper(blogMapper.class);
        // 实例化Blog实体类
        Blog blog = new Blog();

        // 设置第一条数据信息
        blog.setId(IDutils.getId());
        blog.setTitle("MyBatis如此简单");
        blog.setAuthor("狂神说");
        blog.setCreateTime(new Date());
        blog.setViews(9999);
        // 插入第一条数据
        mapper.addBlog(blog);

        // 设置第二条数据信息
        blog.setId(IDutils.getId());
        blog.setTitle("Java如此简单");
        // 插入第二条数据
        mapper.addBlog(blog);

        // 设置第三条数据信息
        blog.setId(IDutils.getId());
        blog.setTitle("Spring如此简单");
        // 插入第三条数据
        mapper.addBlog(blog);

        // 设置第四条数据信息
        blog.setId(IDutils.getId());
        blog.setTitle("微服务如此简单");
        // 插入第四条数据
        mapper.addBlog(blog);

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

}
```

	id	title	author	create_time	views
1	3216bd68c5254155a028f68cada02e62	book1	robin	2022-03-29 18:55:42	9999
2	f9452880b75f469b80477cd7d22c01fb	book2	robin	2022-03-29 18:55:42	999
3	e646dcea2ea2475ebb04d0a60ae6fbef	book3	robin	2022-03-29 18:55:42	9999
4	9f2d65e04c7f454d8079b9fed481b283	book4	yijian	2022-03-29 18:55:42	9999

12.1 if

blogMapper接口

```
<!--需求1:
根据作者名字和博客名字来查询博客!
如果作者名字为空，那么只根据博客名字查询，反之，则根据作者名来查询
```

```
select * from blog where title = #{title} and author = #{author}
-->
<mapper namespace="com.kuang.dao.BlogMapper">

    <!-- 查询博客信息(加上resultType结果返回类型)-->
    <select id="queryBlogIF" parameterType="map" resultType="blog">
        <!-- Select * from mybatis.blog where title = #{title} and author = #
{author} -->
        <!-- where 1=1 首先不会妨碍到正常的查询，其次方便了下面and条件的拼接 -->
        Select * from mybatis.blog where 1=1
        <!-- 判断标题是否为空 -->
        <!--test表示判断条件，如果为true才会生效-->
        <if test="title != null">
            <!-- 若不为空，进行下列的and条件拼接 -->
            and title = #{title}
        </if>
        <!-- 判断作者是否为空 -->
        <if test="author !=null">
            <!-- 若不为空，进行下列的and条件拼接 -->
            and author = #{author}
        </if>
    </select>

</mapper>
```

Test

```
HashMap map = new HashMap();
// 设置标题信息
/*map.put("title", "book1");*/
map.put("author", "robin");

// 调用queryBlogIF方法，将map放入，获取List集合
List<Blog> blogs = mapper.queryBlogIF(map);
// 遍历List集合
for (Blog blog : blogs) {
    System.out.println(blogs);
}
```

12.2 where

使用where标签后:

- 如果只满足第二个条件（即标题信息为空，但作者信息不为空），它会自动去掉多余的and及前面的条件
- 如果两个条件都满足（即标题和作者都不为空），它又会自动加上之前多余的and连同前面的条件
- 如果条件都不满足（即标题和作者都为空），它还会自动删除后面的where及连同的条件

```
<select id="queryBlogIF" 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>
```

- **where元素只会在至少有一个子元素的条件返回SQL子句（即至少满足一个条件）的情况，才会自动插入“where”子句**
- **如果语句的开头为“and”或“or”，where元素也会将它们自动去除**

12.3 choose(when, otherwise)

choose(when,otherwise)相当于Java中的switch...case...语句，当标题满足一个条件（即标题为“Spring如此简单”）时，因为又使用了where标签，便自动去除and后面的条件，然后便结束查询

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

12.4 Set

虽然未设置作者信息，但是使用了set标签后，**自动去除标题条件后面的逗号以及作者信息条件**

```
<mapper namespace="com.kuang.dao.BlogMapper">

    <!-- 更新博客信息：使用set标签 -->
    <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>

</mapper>
```

```
public void testSelect3(){
    SqlSession sqlSession = MyBatisUtils.getSqlSession();
    blogMapper mapper = sqlSession.getMapper(blogMapper.class);

    Map map = new HashMap();
    //map.put("title", "book5");
    map.put("author", "yijian");
    map.put("id", "3216bd68c5254155a028f68cada02e62");

    int blogs = mapper.updateBlog(map);
    System.out.println(blogs);
    sqlSession.close();
}
```

12.5 trim(where, set)

```
<trim prefix="where" prefixOverrides="and | or">
    ...
</trim>
```

refix="where"：是否加上前缀where
prefixOverrides="and | or"： 是否替换and或者or，如果不存在前缀where，则替换"and | or"
"and | or"在语句的前面

```
<trim prefix="set" suffixOverrides=",">
    ...
</trim>
```

prefix="set"：是否加上前缀set
suffixOverrides=",": 是否替换掉",", 如果不存在前缀set，则替换","
","在语句的后面

- prefix表示设置前缀，值通常可设为"where"或者"set"
- prefixOverrides表示句前覆盖，若前缀不存在，自动去除其设置值，通常可设置为"and | or"
- suffixOverrides表示句后覆盖，若后缀不存在，自动去除其设置值，通常可设置为","

动态SQL的本质就是可以在SQL层面执行一个逻辑代码

12.6 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

```
<sql id="if-title-author" >

    <!-- 判断标题是否为空 -->
    <if test="title != null">
        title = #{title}
    </if>

    <!-- 判断作者是否为空 -->
    <if test="author != null">
        and author = #{author}
    </if>

</sql>
```

使用

```
<!-- 查询博客信息：使用where标签 -->
<select id="queryBlogIF" parameterType="map" resultType="blog">
    Select * from mybatis.blog
    <where>
        <!-- refid: 引用的id，这里指之前写好的的sql标签的id -->
        <include refid="if-title-author"></include>
    </where>
</select>
```

- 最好基于单表来定义SQL片段
- 不要存在where标签

```
<sql id="Base_Column">
    id, username, address
</sql>
```

```
<select id = "getUserById" resultType = "user">
    select <include refid = "Base_Column"/> from user where id in
    <foreach collection = "ids" item = "id" open = "(" separator = ",">
        #{id}
    </foreach>
</select>
```

12.7 Foreach

```
SELECT * FROM user WHERE 1 = 1 and (id = 1 or id = 2 or id =3)
```

动态SQL中可以使用foreach语句对集合进行遍历，通常是在构建in条件语句的时候

```
<mapper namespace="com.kuang.dao.BlogMapper">
```

```
    <!-- 查询博客信息：使用foreach标签 -->
    <!-- 现在传递一个万能的map，这个map中可以存在一个集合-->
    <select id="queryBlogForeach" parameterType="map" resultType="blog">
        Select * from mybatis.blog
        <where>
            <!-- foreach: 遍历集合; item: 集合项（单个循环的变量）;  index: 索引(一般不使用);
            collection: 集合（数组变量）; open: 开头（循环结束后，左边的符号）; close: 结尾（循环结束后，右边的符号）; separator: 循环元素之间的分隔符 -->
            <foreach item="id" collection="ids" open="and (" separator="or" close=")">
                id = #{id}
            </foreach>
        </where>
    </select>

</mapper>
```

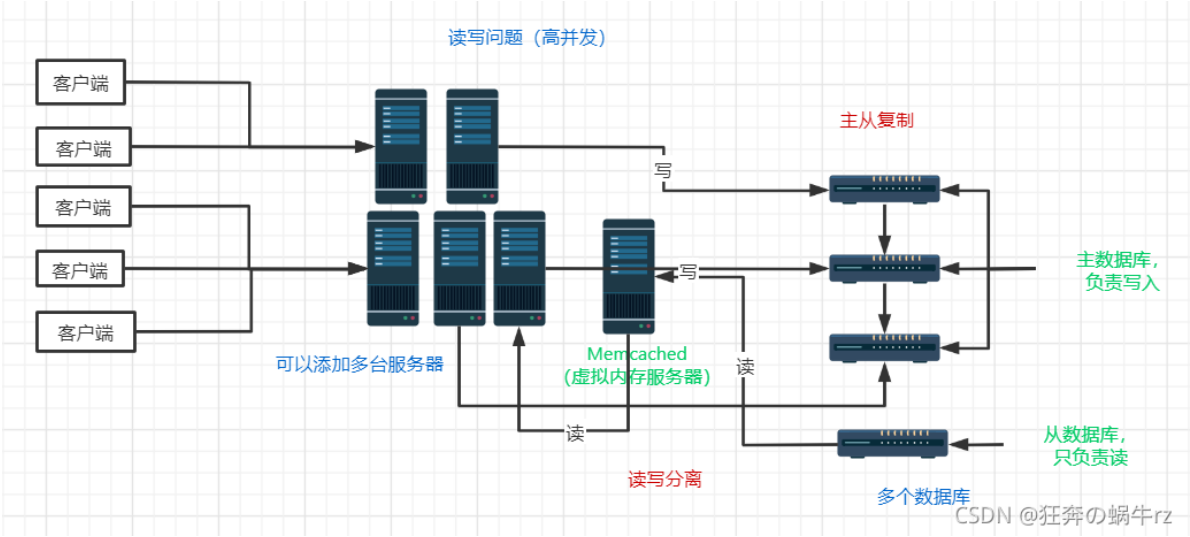
Test

```
// 实例化HashMap
HashMap map = new HashMap();
// 实例化ArrayList（存入ids这个集合）
ArrayList<Integer> ids = new ArrayList<Integer>();

// 添加id为1的集合项
ids.add(1);
// 添加id为2的集合项
ids.add(2);
// 把ids放入到map中
map.put("ids",ids);

// 获取Blog的List集合列表
List<Blog> blogs = mapper.queryBlogForeach(map);
// 遍历List集合
for (Blog blog : blogs) {
    System.out.println(blog);
}
```

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

先在MySQL中写出完整的SQL，再去对应的修改成为动态SQL，实现通用即可

13. 缓存



读写分离，主从复制

13.1 缓存概念

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(关系型数据库数据文件)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. 什么样的数据能使用缓存?
    - 经常查询并且不经常改变的数据。

13.2 MyBatis缓存

- MyBatis包含一个非常强大的查询缓存特性，它可以非常方便地定制和配置缓存。缓存可以极大的提升查询效率。
- 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。
    - 为了提高扩展性，MyBatis定义了缓存接口Cache。我们可以通过实现Cache接口来自定义二级缓存
- MyBatis中的清楚策略：
    - **LRU (最近最少使用）：移除最长时间不被使用的对象**
    - **FIFO (先进先出)：按对象进入缓存的顺序来移除它**
    - SOFT (软引用)：基于垃圾回收器状态和软引用规则移除对象
    - WEAK (弱引用)：更积极地基于垃圾收集器状态和弱引用规则移除对象
- Mybatis中默认的清除策略是LRU

13.3 一级缓存

一级缓存也叫本地缓存：

- 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，没必须再去查询数据库；

测试

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

```
//根据id查询用户
User queryUserById(@Param("id") int id);
```

```
<select id="queryUserById" resultType="user">
select * from user where id = #{id}
</select>
```

```
User user = mapper.queryUserById(1);
System.out.println(user);
User user2 = mapper.queryUserById(1);
System.out.println(user2);
System.out.println(user==user2);
```

```
Opening JDBC Connection
Created connection 1205555397.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@47db50c5]
==>  Preparing: select * from user where id = ?
==> Parameters: 1(Integer)
<==    Columns: id, name, pwd
<==        Row: 1, 秦疆, asdfgh
<==      Total: 1
User(id=1, name=秦疆, pwd=asdfgh)
User(id=1, name=秦疆, pwd=asdfgh)
true
Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@47db50c5]
Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@47db50c5]
Returned connection 1205555397 to pool.
```

SQL语句只查询了一次

第二次结果，没有进行数据库查询

用的是同一个对象

13.4 一级缓存失效的情况

一级缓存是SqlSession级别的缓存，是一直开启的，我们**关闭不了它**；

一级缓存失效情况：**没有使用到当前的一级缓存**，效果就是，**还需要再向数据库中发起一次查询请求**！

1. **sqlSession不同**

每个sqlSession中的缓存相互独立，查询不同的Mapper.xml

2. **sqlSession相同，查询条件不同**

当前缓存中，不存在这个数据

3. **sqlSession相同，两次查询之间执行了增删改操作！**

因为增删改操作可能会对当前数据产生影响

4. **sqlSession相同，手动清除一级缓存**

session.clearCache();//手动清除缓存

一级缓存就是一个map

13.5 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；
  - 如果当前会话关闭了，这个会话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查出的数据会放在自己对应的缓存（map）中；

使用

1. 开启全局缓存

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

2. Mapper.xml中开启二级缓存

- 在UserMapper.xml映射文件中使用`</cache>`标签开启二级缓存

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

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

cache标签参数解释：

eviction=“FIFO”：缓存清除策略为FIFO（即先进先出）
flushInterval=“60000”：刷新间隔为60秒
size=“512”：最多可以存储结果对象或列表的512个引用
readOnly=“true”：返回的对象是只读的

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

```
    <!-- 在当前的Mapper.xml映射文件中使用二级缓存，缓存只作用于cache标签所在的映射文件中的语句 -->
    <!-- eviction="FIFO": 缓存清除策略为FIFO(即先进先出)
       flushInterval="60000": 刷新间隔为60秒
       size="512": 最多可以存储结果对象或列表的512引用
       readOnly="true": 返回的对象是只读的 -->
    <cache eviction="FIFO" flushInterval="60000" size="512" readOnly="true"/>

    <!-- 通过id查询指定用户信息 -->
    <!-- parameterType="_int": int类型参数可以起别名“_id”，也可以省略
         resultType="user": 在核心配置文件中起别名才能直接使用user
         useCache="true": 使用缓存 -->
    <select id="queryUserById"  resultType="user" useCache="true">
        Select * from user where id = #{id}
    </select>

    <!-- 更新用户信息 -->
    <!-- flushCache="true": 其值设置为true，即为允许刷新缓存
         当需要调优的时候，也可以选择关闭，将其值设置为"false"即可-->
    <update id="updateUser" parameterType="User" flushCache="true">
        Update mybatis.user set name = #{name}, pwd = #{pwd} where id = #{id}
    </update>

</mapper>
```

```
package com.kuang.dao;

public class MyTest {

   // 通过id查询指定的用户信息
    @Test
    public void queryUserById2() {

        // 分别获取两个sqlSession对象
        SqlSession sqlSession = MybatisUtils.getSqlSession();
        SqlSession sqlSession2 = MybatisUtils.getSqlSession();

        // 获取第一个Mapper接口对象
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
        // 调用queryUserById方法，查询id为1的用户信息
        User user = mapper.queryUserById(1);
        // 打印用户信息
        System.out.println(user);
        // 关闭第一个sqlSession对象
        sqlSession.close();

        // 获取第二个Mapper接口对象
        UserMapper mapper2 = sqlSession2.getMapper(UserMapper.class);
        // 调用queryUserById方法，再次查询id为1的用户信息
        User user2 = mapper2.queryUserById(1);
        // 打印用户信息
        System.out.println(user2);
        // 判断两个用户信息是否相等
        System.out.println(user==user2);
        // 关闭第二个sqlSession对象
        sqlSession2.close();
```

```
    }

}
```

使用了二级缓存后，预编译sql只执行了一次；当第一条用户信息查询完关闭sqlSession后，一级缓存虽然关闭了，但其数据被保存到了二级缓存中；因此第二次查询相同的用户信息，直接从二级缓存取出，不需要访问数据库和执行预编译sql语句了

将实体类序列化

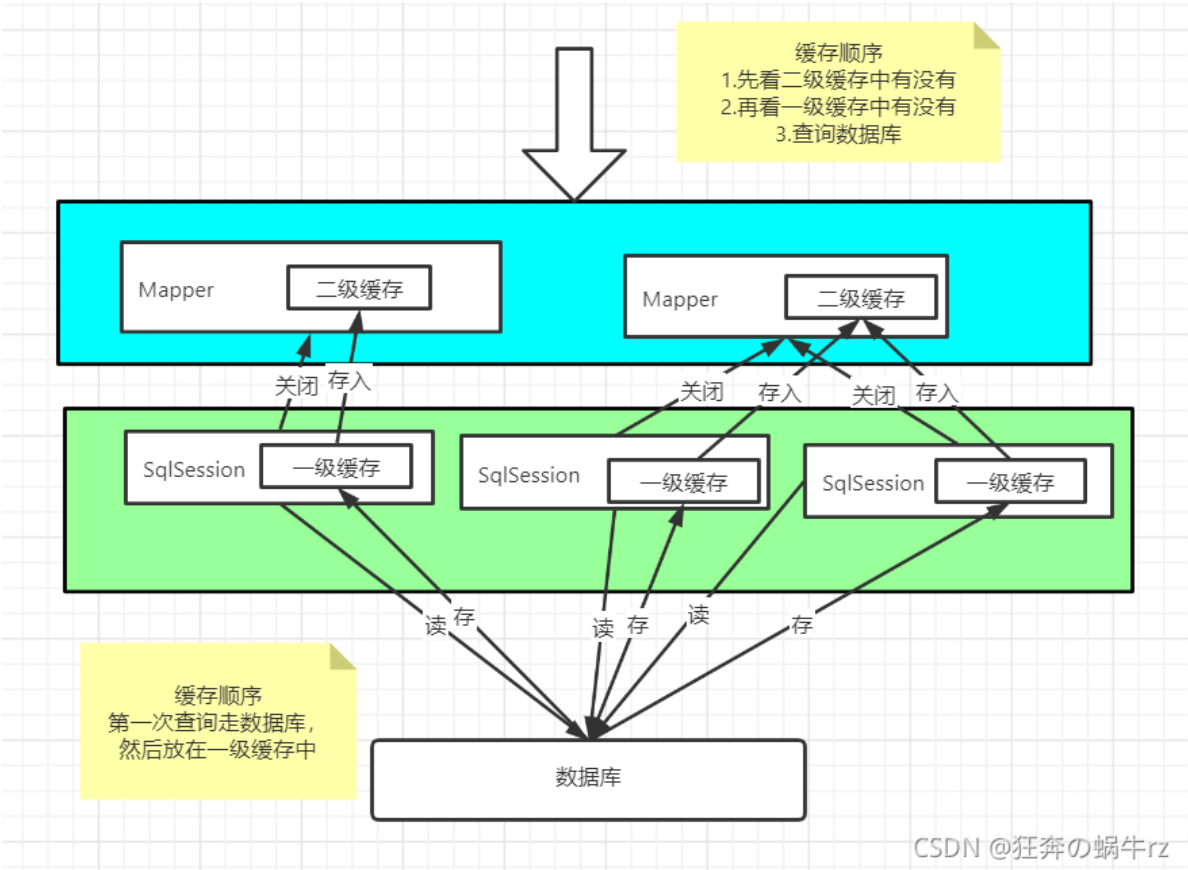
```
public class User implements Serializable {
    private int id;
    private String name;
    private String pwd;
}
```

第一次查询完id为1的用户信息后，本地缓存关闭，将数据存入二级缓存；再次查询此用户信息时，不用执行预编译SQL，查询二级缓存，直接取出该用户信息；但仔细观察后发现，"user==user2"的结果却为false了，是因为从二级缓存中取出该用户信息，内存地址发生了变化，因此结果为false

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

13.6 缓存原理

第一查询走数据库，然后放在一级缓存中，一次会话关闭后，一级缓存将数据存入二级缓存



- 第一次查询id为1的用户时，执行一次预编译SQL去查询数据库，然后将数据存入一级缓存

- 第一次查询结束后，关闭sqlSession会话时，一级缓存又将数据存入二级缓存中，因此，当第二次查询id为1的用户时，就直接查询二级缓存，不需要再次访问数据库了

开启和使用缓存

- 在使用二级缓存的Mapper.xml映射文件中设置useCache的值即可，一般用于查询数据信息

```
<mapper namespace="com.kuang.dao.UserMapper">
    <!-- 在当前的Mapper.xml文件中使用二级缓存 -->
    <!-- 使用简单的<cache/>标签 -->
    <cache/>

    <!-- 通过id查询指定用户信息-->
    <!-- useCache="true": 一般在查询数据时使用，值设置为"true“，即允许使用缓存； 当数据更新过于频繁时，可以将其值设置为false，禁止使用缓存 -->
    <select id="queryUserById"  resultType="user" useCache="true">
        Select * from user where id = #{id}
    </select>
</mapper>
```

开启和关闭刷新缓存

- 在使用二级缓存的Mapper.xml映射文件中设置 flushCache的值即可，一般在修改数据时使用

```
<mapper namespace="com.kuang.dao.UserMapper">

    <!-- 在当前的Mapper.xml映射文件中使用二级缓存 -->
    <!-- 使用简单的<cache/>标签 -->
    <cache/>

    <!-- 更新用户信息 -->
    <!-- flushCache="false": 一般在更新数据使用，其值设置为true，即为允许刷新缓存； 当需要调优的时候，也可以选择关闭，将其值设置为"false"即可 -->
    <update id="updateUser" parameterType="user" flushCache="false">
        Update mybatis.user set name = #{name}, pwd = #{pwd} where id = #{id}
    </update>
</mapper>
```

13.7 自定义缓存 Ehcache

1. Ehcache

- Ehcache是一个纯Java的进程内缓存框架，具有快速、精干的特点，是Hibernate中默认的CacheProvider
- 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、JavaEE和轻量级容器
- 它具有内存和磁盘存储、缓存加载器、缓存拓展、缓存异常处理程序、一个gzip缓存servlet过滤器、支持REST和SOAP API等特点

2. 特点

- 快速、简单，具有多种缓存策略
- 缓存数据有两级：内存和磁盘，因此无需担心容量问题
- 缓存数据会在虚拟机重启的过程中写入磁盘
- 可以通过RMI、可插入API等方式进行分布式缓存
- 具有缓存和缓存管理器的监听接口
- 支持多级缓存管理器实例，以及一个实例的多个缓存区域
- 提供Hibernate的缓存实现

使用

```
<!-- 导入mybatis-ehcache资源依赖 -->
<!-- https://mvnrepository.com/artifact/org.mybatis.caches/mybatis-ehcache -->
<dependency>
    <groupId>org.mybatis.caches</groupId>
    <artifactId>mybatis-ehcache</artifactId>
    <version>1.2.2</version>
</dependency>
```

在resources文件夹下创建ehcache.xml文件，然后编写具体的配置信息

```
<?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: 为缓存路径，ehcache分为内存和磁盘两级，此属性定义磁盘的缓存位置
   参数解释如下: user.home: 表示用户主目录: user.dir: 表示用户当前工作目录;
java.io.tmpdir: 表示默认临时文件路径-->
    <diskStore path="./tmpdir/Tmp_EhCache"/>

    <!-- defaultCache: 表示默认缓存策略，当ehcache找不到自定义的缓存时则使用这个缓存策略，
只能定义一个 -->
    <defaultCache
            eternal="false"
            maxElementsInMemory="10000"
            overflowToDisk="false"
            diskPersistent="false"
            timeToIdleSeconds="1800"
            timeToLiveSeconds="259200"
            memoryStoreEvictionPolicy="LRU"/>
    <!-- name: 表示缓存名称
         maxElementsInMemory: 缓存最大数目
         maxElementsOnDisk: 硬盘最大缓存个数
         eternal: 对象是否永久有效，一旦设置，timeout将不起作用
         overflowToDisk: 是否保存到硬盘，当系统宕机时
         timeToIdleSecond: 设置对象在失效前的允许闲置时间（单位：秒）
         timeToLiveSecond: 设置对象在失效前允许存活时间（单位：秒）
         diskPersistent: 是否缓存虚拟机重启数据
         diskSpoolBufferSizeMB: 这个参数设置DiskStore（磁盘缓存）的缓存区大小
         diskExpiryThreadIntervalSeconds: 磁盘失效线程运行时间间隔，默认是120秒
         memoryStoreEvictionPolicy: 当达到maxElementsInMemory限制时，Ehcache将会根据
指定的策略去清理内存，默认策略是LRU（最近最少使用）
```

```
        memoryStoreEvictionPolicy可选策略: LRU（最近最少使用，默认策略）、FIFO（先进先出）、LFU（最少访问次数）-->
    <cache
            name="cloud_user"
            eternal="false"
            maxElementsInMemory="5000"
            overflowToDisk="false"
            diskPersistent="false"
            timeToIdleSeconds="1800"
            timeToLiveSeconds="1800"
            memoryStoreEvictionPolicy="LRU"/>

</ehcache>
```

- 实际开发中，一般都会会用Redis数据库来做缓存：即K-V键值对
- Redis是NOSQL型 (即非关系型) 数据库的一种，非关系型数据库通常指数据以对象的形式存储在数据库中，而 对象之间的关系通过每个对象自身的属性来决定