

# MyBatis

MyBatis是一款优秀的持久层框架，它支持定制化SQL，存储过程以及高级映射。

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

MyBatis可以使用简单的XML或注解来配置和映射原生类型、接口和Java的POJO (Plain Old Java, 普通式java对象) 为数据库库中的记录。

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。

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

持久层：完成持久化工作的代码块，像Dao层，Service层，Controller层一样，层的界限十分明显。

为什么需要MyBatis：帮助我们将数据存入到数据库中，

- 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  
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。

第一个Mybatis程序：思路：搭建环境-->导入Mybatis-->编写代码-->测试

## 2.1 搭建环境

搭建数据库：

```
CREATE DATABASE `mysbatis`;
```

```
use `mybatis`;
```

```
CREATE TABLE `user` (
    `id` int(20) NOT NULL PRIMARY KEY,
    `name` VARCHAR(30) DEFAULT NULL,
    `pwd` VARCHAR(30) NOT NULL
) ENGINE=INNODB DEFAULT CHARSET=UTF8;
```

```
INSERT INTO `user` (`id`, `name`, `pwd`)
VALUES ('1', '张三', '123'), ('2', '李四', '456'), ('3', '王五', '789')
```

在IDEA新建Maven项目

删除src目录

导入maven依赖:

```
--导入依赖-->
<dependencies>
    <!--mysql驱动-->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.26</version>
    </dependency>
    <!--mybatis-->
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.5.2</version>
    </dependency>
    <!--junit-->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.11</version>
    </dependency>
</dependencies>
```

3. 创建一个模块

编写mybatis的核心配置文件（连接数据库）

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
    PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-config.dtd">
<!--configuration核心配置文件-->
<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=UTF8"/>
                <property name="username" value="root"/>
                <property name="password" value="123456"/>
            </dataSource>
        </environment>
    </environments>
</configuration>
```

编写mybatis工具类构建SqlSessionFactory

```
package com.xiang.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;
```

```

//从XML文件中构建SqlSessionFactory实例
public class MybatisUtils {
    private static SqlSessionFactory sqlSessionFactory;
    static {
        try{
            //使用mybatis的第一步：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();
        }
    }
    //有了SqlSessionFactory对象，就可以从中获得SqlSession的实例了
    //SqlSession完全包含了面向数据库执行SQL命令所需的所有方法
    public static SqlSession getSqlSession(){
        return sqlSessionFactory.openSession();
    }
}

```

## 2.3 编写代码

### 实体类

```

package com.xiang.pojo;
//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

```

package com.xiang.dao;

import com.xiang.pojo.User;

import java.util.List;

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

```

接口实现类：由原来的UserDaoImpl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="com.xiang.dao.UserDao">
    <!--select查询-->
    <select id="getUserList" resultType="com.xiang.pojo.User">
        select * from mybatis.user
    </select>
</mapper>

```

## 2.4 测试

Junit测试

注意报错：org.apache.ibatis.binding.BindingException: Type interface

com.xiang.dao.UserDao is not known to the MapperRegistry。 配置文件导出错误

解决办法：在pom.xml文件中加上

```

<!--在build中配置resources，来防止资源导出失败的问题-->
<build>
    <resources>
        <resource>
            <directory>src/main/resources</directory>
            <includes>
                <include>**/*.properties</include>
                <include>**/*.xml</include>
            </includes>
            <filtering>true</filtering> <!--开启资源过滤-->
        </resource>
        <resource>
            <directory>src/main/java</directory>
            <includes>
                <include>**/*.properties</include>
                <include>**/*.xml</include>
            </includes>
            <filtering>true</filtering>
        </resource>
    </resources>

```

```
</resource>
</resources>
</build>
```

## SqlSessionFactoryBuilder

这个类可以被实例化、使用和丢弃，一旦创建了 SqlSessionFactory，就不再需要它了。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）。你可以重用 SqlSessionFactoryBuilder 来创建多个 SqlSessionFactory 实例，但是最好还是不要让其一直存在，以保证所有的 XML 解析资源可以被释放给更重要的事情。

## SqlSessionFactory

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。使用 SqlSessionFactory 的最佳实践是在应用运行期间不要重复创建多次，多次重建 SqlSessionFactory 被视为一种代码“坏味道（bad smell）”。因此 SqlSessionFactory 的最佳作用域是应用作用域。有很多方法可以做到，最简单的就是使用单例模式或者静态单例模式。

## SqlSession

每个线程都应该有它自己的 SqlSession 实例。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。绝对不能将 SqlSession 实例的引用放在一个类的静态域，甚至一个类的实例变量也不行。也绝不能将 SqlSession 实例的引用放在任何类型的托管作用域中，比如 Servlet 框架中的 HttpSession。如果你现在正在使用一种 Web 框架，要考虑 SqlSession 放在一个和 HTTP 请求对象相似的作用域中。换句话说，每次收到的 HTTP 请求，就可以打开一个 SqlSession，返回一个响应，就关闭它。这个关闭操作是很重要的，你应该把这个关闭操作放到 finally 块中以确保每次都能执行关闭。下面的示例就是一个确保 SqlSession 关闭的标准模式：

```
try (SqlSession session = sqlSessionFactory.openSession()) {
    // 你的应用逻辑代码
}
```

在你的所有的代码中一致地使用这种模式来保证所有数据库资源都能被正确地关闭。

## 3 CRUD

1. namespace （UserMapper.xml 文件 Mapper 标签中的 namespace 属性）

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 属性：**就是对应的 namespace 中的方法名

resultType：sql 语句执行的返回值

parameterType：参数类型

带参数的查询语句只需修改

UserMapper 接口

```
package com.xiang.dao;
```

```
import com.xiang.pojo.User;
```

```

import java.util.List;

public interface UserMapper {
    //查询全部用户
    List<User> getUserList();
    //根据ID查询用户
    User getUserById(int id); //insert一个用户
    int addUser(User user);
    //修改用户
    int updateUser(User user);
    //删除一个用户
    int deleteUser(int id);
}

```

### UserMapper.xml配置

```

<?xml version="1.0" encoding="UTF8" ?>
<!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.xiang.dao.UserMapper">
    <!--select查询-->
    <select id="getUserList" resultType="com.xiang.pojo.User">
        select * from mybatis.user
    </select>

    <select id="getUserById" parameterType="int" resultType="com.xiang.pojo.User">
        select * from mybatis.user where id = #{id}
    </select> <!--对象中的属性，可以直接取出来放到values里面-->
    <insert id="addUser" parameterType="com.xiang.pojo.User">
        insert into mybatis.user (id, name, pwd) values(#{id}, #{name}, #{pwd})
    </insert>

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

    <delete id="deleteUser" parameterType="int">
        delete from mybatis.user where id=#{id};
    </delete>
</mapper>

```

### 测试文件

```

package com.xiang.dao;

import com.xiang.pojo.User;
import com.xiang.utils.MybatisUtils;
import org.apache.ibatis.session.SqlSession;
import org.junit.Test;

import java.util.List;

public class UserDaoTest {
    @Test
    public void test() {
        //第一步：获取SqlSession对象
        SqlSession sqlSession = MybatisUtils.getSqlSession();
        //执行SQL方式一：getMapper
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
        List<User> userList = mapper.getUserList();
    }
}

```

```

//老方法，方式二：
//      List<User> userList =
sqlSession.selectList("com.xiang.dao.UserDao.getUserList");

for (User user : userList) {
    System.out.println(user);
}
//关闭SqlSession
sqlSession.close();
}

@Test
public void getUserById() {
    SqlSession sqlSession = MybatisUtils.getSqlSession();
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);
    User userById = mapper.getUserById(2);
    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);
    int res = mapper.addUser(new User(4, "赵六", "234567"));
    if(res > 0) {
        System.out.println("插入成功!!");
    }
    //提交事务
    sqlSession.commit();
    sqlSession.close();
}

@Test
public void updateUser() {
    SqlSession sqlSession = MybatisUtils.getSqlSession();
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);
    int res = mapper.updateUser(new User(4, "赵七", "101"));
    if(res>0) {
        System.out.println("更新成功!!");
    }

    sqlSession.commit();
    sqlSession.close();
}

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

```

3. insert

4. update

5. delete

注意点：增删改需要提交事务sqlSession.commit();

使用map：若我们的实体类或者数据库中的表、字段、参数过多，我们应当考虑使用map代替对象  
UserMapper接口：

```
int addUser2(Map<String, Object> map);
```

UserMapper.xml文件：

```
<!--传递map的key-->
<insert id="addUser2" parameterType="map">
    insert into mybatis.user (id, pwd)
    values (#{}{userid}, #{}{userpwd});
</insert>
```

测试：

```
@Test
public void addUser2() {
    SqlSession sqlSession = MybatisUtils.getSqlSession();
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);

    Map<String, Object> map = new HashMap<String, Object>();
    map.put("userid", 6);
    //      map.put("username", "wang");
    map.put("userpwd", "102");
    mapper.addUser2(map);
    sqlSession.commit();
    sqlSession.close();
}
```

获得指定id和pwd的User

UserMapper接口：

```
User getUserById2(Map<String, Object> map);
```

UserMapper.xml文件：

```
<select id="getUserById2" parameterType="map" resultType="com.xiang.pojo.User">
    select *
    from mybatis.user
    where id = #{}{idd} and pwd = #{}{namee};
</select>
```

测试：

```
@Test
```

```
public void getUserById2() {
```

```

SqlSession sqlSession = MybatisUtils.getSqlSession();
UserMapper mapper = sqlSession.getMapper(UserMapper.class);
Map<String, Object> map = new HashMap<>();
map.put("id", 6);
map.put("name", "李");
User user = mapper.getUserById2(map);
System.out.println(user);
sqlSession.close();
}

```

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

对象传递参数，直接在sql中取对象的属性即可parameterType="com.xiang.pojo.User"

只有一个基本类型参数的情况下，可以直接在sql中取到，不用写parameterType

多个参数用Map，或者注解

模糊查询时，为了防止SQL注入：

1、Java代码执行的时候，传递通配符%，直接写死

```

List<User> getUserList2(String value);
<!--模糊查询中，为了防止SQL注入，在sql拼接中使用通配符-->
<select id="getUserList2" resultType="com.xiang.pojo.User">
    select * from mybatis.user where name like #{value}
</select>
@Test
public void getUserList2() {
    SqlSession sqlSession = MybatisUtils.getSqlSession();
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);
    List<User> user = mapper.getUserList2("%李%");
    for (User user1 : user) {
        System.out.println(user1);
    }
}

```

2、在sql拼接中使用通配符：

```

List<User> getUserList2(String value);
<!--模糊查询中，为了防止SQL注入，在sql拼接中使用通配符-->
<select id="getUserList2" resultType="com.xiang.pojo.User">
    select * from mybatis.user where name like "%" #{value} "%"
</select>
@Test
public void getUserList2() {
    SqlSession sqlSession = MybatisUtils.getSqlSession();
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);
    List<User> user = mapper.getUserList2("李");
    for (User user1 : user) {
        System.out.println(user1);
    }
}

```

```
}
```

## 4、配置解析

### 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)

#### 事务管理器 (transactionManager)

[搜索] [帮助]

在 MyBatis 中有两种类型的事务管理器 (也就是 type="JDBC|MANAGED") :

- JDBC – 这个配置就是直接使用了 JDBC 的提交和回滚设置，它依赖于从数据源得到的连接来管理事务作用域。
- MANAGED – 这个配置几乎没做什么。它从来不提交或回滚一个连接，而是让容器来管理事务的整个生命周期 (比如 JEE 应用服务器的上下文)。默认情况下它会关闭连接，然而一些容器并不希望这样，因此需要将 closeConnection 属性设置为 false 来阻止它默认的关闭行为。例如：

```
<transactionManager type="MANAGED">
  <property name="closeConnection" value="false"/>
</transactionManager>
```

**提示** 如果你正在使用 Spring + MyBatis，则没有必要配置事务管理器，因为 Spring 模块会使用自带的管理器来覆盖前面的配置。

这两种事务管理器类型都不需要设置任何属性。它们其实是类型别名，换句话说，你可以使用 TransactionFactory 接口的实现类的完全限定名或类型别名代替它们。

```
public interface TransactionFactory {
  default void setProperties(Properties props) { // Since 3.5.2, change to default method
    // NOP
  }
  Transaction newTransaction(Connection conn);
  Transaction newTransaction(DataSource dataSource, TransactionIsolationLevel level, boolean autoCommit);
}
```

#### 数据源 (dataSource) 连接数据库 dbcp c3p0 druid

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。

- 许多 MyBatis 的应用程序会按示例中的例子来配置数据源。虽然这是可选的，但为了使用延迟加载，数据源是必须配置的。

有三种内建的数据源类型 (也就是 type="UNPOOLED|POOLED|JNDI") :

**UNPOOLED**– 这个数据源的实现只是每次被请求时打开和关闭连接。虽然有点慢，但对于在数据库连接可用性方面没有太高要求的简单应用程序来说，是一个很好的选择。不同的数据库在性能方面的表现也是不一样的，对于某些数据库来说，使用连接池并不重要，这个配置就很适合这种情形。UNPOOLED 类型的数据源具有以下属性。：

- driver – 这是 JDBC 驱动的 Java 类的完全限定名 (并不是 JDBC 驱动中可能包含的数据源类)。
- url – 这是数据库的 JDBC URL 地址。
- username – 登录数据库的用户名。
- password – 登录数据库的密码。
- defaultTransactionIsolationLevel – 默认的连接事务隔离级别。
- defaultNetworkTimeout – The default network timeout value in milliseconds to wait for the database operation to complete. See the API documentation of `java.sql.Connection#setNetworkTimeout()` for details.

池：用完可以回收

作为可选项，你也可以传递属性给数据库驱动。只需在属性名加上“driver.”前缀即可，例如：

- `driver.encoding=UTF8`

这将通过 `DriverManager.getConnection(url,driverProperties)` 方法传递值为 `UTF8` 的 `encoding` 属性给数据库驱动。

**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，避免了创建新的连接实例时所必需的初始化和认证时间。这是一种使得并发 Web 应用快速响应请求的流行处理方式。

## (2) 环境配置 (environments)

Mybatis可以配置成适应多种环境，学会配置多套环境，default=”选择环境id”

```
<environments default="development">
```

```
    <environment id="development">
```

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

Mybatis默认的事务管理器是JDBC，连接池：POOLED

## (3) 属性 (properties)

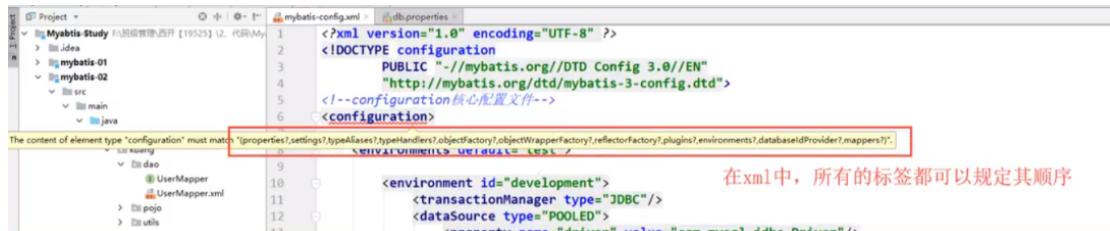
我们可以通过properties属性来实现引用配置文件

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，亦可以通过properties元素的子元素来传递【db.properties】

编写一个配置文件：

```
driver=com.mysql.jdbc.Driver  
url=jdbc:mysql://localhost:3306/mybatis?  
useSSL=true&useUnicode=true&characterEncoding=UTF-8  
username=root  
password=123456
```

在核心配置文件中引入，标签在文件中的顺序如下：



```
    <!--引入外部配置文件-->  
    <properties resource="db.properties">  
        <property name="username" value="root"/>  
        <property name="pwd" value="111111"/>    <!--里外都配置了的话，优先使用外部属性-->  
    </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="${username}"/>  
            <property name="password" value="${pwd}"/>  
        </dataSource>
```

```
</environment>  
</environments>  
可以直接引入外部文件  
可以在其中增加一些属性配置  
如果两个文件都有同一属性，优先使用外部配置文件的
```

#### (4) 类型别名 (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

在mybatis-config.xml文件中给实体类起别名

```
<!--给实体类起别名-->  
<typeAliases>  
    <typeAlias type="com.xiang.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，默认包名是这个类的类名（首字母小写）

当这样配置时，`Blog` 可以用在任何使用 `domain.blog.Blog` 的地方。

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

```
<typeAliases>  
    <package name="domain.blog"/>  
</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。见下面的例子：

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

<!--给实体类起别名-->

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

在实体类比较少的时候，使用第一种，如果实体类十分多，建议使用第二种，第一种可以DIY（自定义）别名，第二种不能（若一定要改别名，可以在实体类上使用注解`@Alias("user")`）

Java类型内建的相应的类型别名：八大基本类型：int。。。别名：`_int`；包装类：Integer。。。别名：`int`

#### (5) 设置 (settings)

这是Mybatis中极为重要的调整设置，他们会改变Mybatis的运行时行为

设置名	描述	有效值	默认值
<code>cacheEnabled</code>	全局地开启或关闭配置文件中的所有映射器已经配置的任何缓存。	true   false	true
<code>lazyLoadingEnabled</code>	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

mapUnderscoreToCamelCase  
是否开启自动驼峰命名规则 (camel case) 映射, 即从经典数据库列名 A\_COLUMN 到经典 Java 属性名 aColumn 的类似映射。

logImpl  
指定 MyBatis 所用日志的具体实现, 未指定时将自动查找。

last\_name | lastName

SLF4J | LOG4J  
LOG4J2  
JDK\_LOGGING  
COMMONS\_LOGGING  
STDOUT\_LOGGING  
NO\_LOGGING

一个配置完整的 settings 元素的示例如下:

```
<settings>
    <setting name="cacheEnabled" value="true"/>
    <setting name="lazyLoadingEnabled" value="true"/>
    <setting name="multipleResultSetsEnabled" value="true"/>
    <setting name="useColumnLabel" value="true"/>
    <setting name="useGeneratedKeys" value="false"/>
    <setting name="autoMappingBehavior" value="PARTIAL"/>
    <setting name="autoMappingUnknownColumnBehavior" value="WARNING"/>
    <setting name="defaultExecutorType" value="SIMPLE"/>
    <setting name="defaultStatementTimeout" value="25"/>
    <setting name="defaultFetchSize" value="100"/>
    <setting name="safeRowBoundsEnabled" value="false"/>
    <setting name="mapUnderscoreToCamelCase" value="false"/>
    <setting name="localCacheScope" value="SESSION"/>
    <setting name="jdbcTypeForNull" value="OTHER"/>
    <setting name="lazyLoadTriggerMethods" value="equals,clone,hashCode,toString"/>
</settings>
```

## (6) 其它配置

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

既然 MyBatis 的行为已经由上述元素配置完了, 我们现在就要定义 SQL 映射语句了。但是首先我们需要告诉 MyBatis 到哪里去找到这些语句。Java 在自动查找这方面没有提供一个很好的方法, 所以最佳的方式是告诉 MyBatis 到哪里去找映射文件。你可以使用相对于类路径的资源引用, 或完全限定资源定位符 (包括 file:/// 的 URL), 或类名和包名等。例如:

```

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

<!-- 使用完全限定资源定位符 (URL) -->
<mappers>
    <mapper url="file:///var/mappers/AuthorMapper.xml"/>
    <mapper url="file:///var/mappers/BlogMapper.xml"/>
    <mapper url="file:///var/mappers/PostMapper.xml"/>
</mappers>

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

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

```

### 方式一：【推荐使用】

```

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

```

### 方式二：使用class文件绑定注册

```

<!--每一个Mapper.xml文件都需要在Mybatis核心配置文件中注册-->
<mappers>
    <mapper class="com.xiang.dao.UserMapper"/>
</mappers>

```

使用class文件注册的注意点：接口和它的Mapper配置文件必须同名，必须在同一个包下。

### 方式三：使用扫描包进行注入绑定

```

<!--每一个Mapper.xml文件都需要在Mybatis核心配置文件中注册-->
<mappers>
    <package name="com.xiang.dao"/>
</mappers>

```

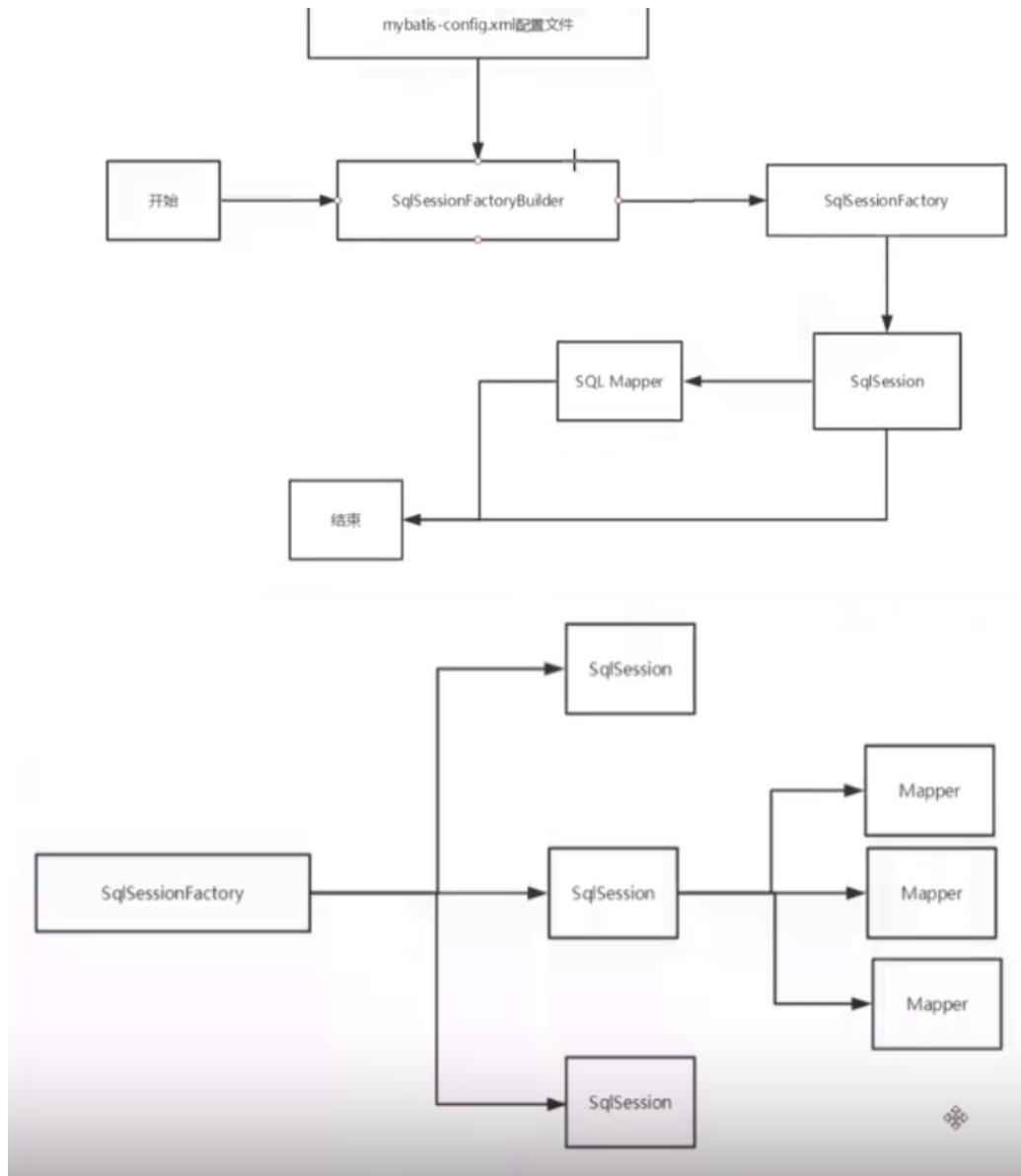
使用扫描包注入绑定的注意点：接口和它的Mapper配置文件必须同名，必须在同一个包下。

## (8) 生命周期和作用域(scope)

`SqlSessionFactoryBuilder`这个类可以被实例化、使用、丢弃，一旦创建了`SqlSessionFactory`，就不再需要它了，局部变量

`SqlSessionFactory`相当于数据库连接池，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，最简单的就是使用单例模式或者静态单例模式，应用作用域（全局）。

`SqlSession`每个线程都应该有它自己的`SqlSession`实例，可以理解为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，所以它的最佳作用域是请求或方法作用域，用完之后需要赶紧关闭，否则资源被占用



这里面的每一个Mapper，就代表一个具体的业务

## 5、解决属性名和字段名不一致的问题

数据库中的字段

tables	1
user	
columns	3

- id int
- name varchar
- pwd varchar

实体类中的属性

```
private int id;
private String name;
private String password;
```

测试结果

```
» ✓ Tests passed: 1 of 1 test - 1 sec 389 ms
0 ms "C:\Program Files\Java\jdk-16.0.1\bin\java.exe" ...
0 ms User{id=1, name='张三', pwd='null'}
```

Process finished with exit code 0

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

解决方法

(1) 方法一起别名

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

(2) 方法二: resultMap 结果集映射

```
<!--结果集映射-->
<resultMap id="UserMap" type="com.xiang.pojo.User">
    <!--column表示数据库中的字段, property表示实体类中的属性-->
    <result column="id" property="id"/>
    <result column="name" property="name"/>
    <result column="pwd" property="password"/>
</resultMap>
```

```
<select id="getUserById" resultMap="UserMap">
    select id, name, pwd 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，对于简单的语句根本不需要配置显示的结果映射，而对于复杂一点的语句只需要描述它们的关系就行了。

```
    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` 不是一个很好的领域模型。你的程序更可能会使用 JavaBean 或 POJO (Plain Old Java Objects, 普通老式 Java 对象) 作为领域模型。MyBatis 对两者都提供了支持。看看下面这个 JavaBean:

基于 JavaBean 的规范，上面这个类有 3 个属性：`id`, `username` 和 `hashedPassword`。这些属性会对应到 `select` 语句中的列名。

这样的一个 JavaBean 可以被映射到 `ResultSet`，就像映射到 `HashMap` 一样简单。

```
<select id="selectUsers" resultType="com.someapp.model.User">
    select id, username, hashedPassword
    from some_table
    where id = #{id}
</select>
```

类型别名是你的好帮手。使用它们，你就可以不用输入类的完全限定名称了。比如：

```
<!-- mybatis-config.xml 中 -->
<typeAlias type="com.someapp.model.User" alias="User"/>

<!-- SQL 映射 XML 中 -->
<select id="selectUsers" resultType="User">
    select id, username, hashedPassword
    from some_table
    where id = #{id}
</select>
```

这些情况下，MyBatis 会在幕后自动创建一个 `ResultMap`，再基于属性名来映射列到 JavaBean 的属性上。如果列名和属性名没有精确匹配，可以在 SELECT 语句中对列使用别名（这是一个基本的 SQL 特性）来匹配标签。比如：

```
<select id="selectUsers" resultType="User">
    select
        user_id          as "id",
        user_name        as "userName",
        hashed_password as "hashedPassword"
    from some_table
    where id = #{id}
</select>
```

`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。上面这些简单的示例根本不需要下面这些繁琐的配置。但出于示范的原因，让我们来看看最后一个示例中，如果使用外部的 `resultMap` 会怎样，这也是解决列名不匹配的另外一种方式。

```
<resultMap id="userResultMap" type="User">
    <id property="id" column="user_id" />
    <result property="username" column="user_name"/>
    <result property="password" column="hashed_password"/>
</resultMap>
```

`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。上面这些简单的示例根本不需要下面这些繁琐的配置。但出于示范的原因，让我们来看看最后一个示例中，如果使用外部的 `resultMap` 会怎样，这也是解决列名不匹配的另外一种方式。

```
<resultMap id="userResultMap" type="User">
    <id property="id" column="user_id" />
    <result property="username" column="user_name"/>
    <result property="password" column="hashed_password"/>
</resultMap>
```

而在引用它的语句中使用 `resultMap` 属性就行了（注意我们去掉了 `resultType` 属性）。比如：

```
<select id="selectUsers" resultMap="userResultMap">
    select user_id, user_name, hashed_password
    from some_table
    where id = #{id}
</select>
```

如果世界总是这么简单就好了。

## 6、日志

### (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

曾经用：sout、debug 现在用：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 Java自带的日志输出
- COMMONS\_LOGGING 工具包
- STDOUT\_LOGGING 控制台输出【掌握】
- NO\_LOGGING 没有日志输出

在Mybatis中具体使用那个日志实现，在设置中设定

STDOUT\_LOGGING 标准日志输出

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

测试添加了日志的输出

```
Opening JDBC Connection  Created connection 1533524862. Setting autocommit to false
on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@5b67bb7e]  ==> Preparing:
select id, name, pwd from mybatis.user where id = ?  ==> Parameters: 1(Integer)
<==   Columns: id, name, pwd <==       Row: 1, 张三, 123  <==       Total: 1
User{id=1, name='张三', pwd='123'}  Resetting autocommit to true on JDBC Connection
[com.mysql.cj.jdbc.ConnectionImpl@5b67bb7e]  Closing JDBC Connection
[com.mysql.cj.jdbc.ConnectionImpl@5b67bb7e]  Returned connection 1533524862 to
pool.
```

### (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
- 我们可以控制每一天日志的输出格式
- 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
- 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

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

2)创建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.appenders.console = org.apache.log4j.ConsoleAppender
log4j.appenders.console.Target = System.out
log4j.appenders.console.Threshold=DEBUG
log4j.appenders.console.layout = org.apache.log4j.PatternLayout
log4j.appenders.console.layout.ConversionPattern=[%c]-%m%n

#文件输出的相关设置
log4j.appenders.file = org.apache.log4j.RollingFileAppender
log4j.appenders.file.File =./log/xiang.log
log4j.appenders.file.MaxFileSize = 10mb
log4j.appenders.file.Threshold=DEBUG
log4j.appenders.file.layout=org.apache.log4j.PatternLayout
log4j.appenders.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
```

日志级别：程序会打印高于或等于所设置级别的日志，设置的日志等级越高，打印出来的日志就越少。

3)在mybatis-config.xml中配置log4j为日志的实现

```
<settings>
  <!--标准的日志工厂实现-->
  <!--<setting name="logImpl" value="STDOUT_LOGGING"/>-->
  <setting name="logImpl" value="LOG4J"/>
</settings>
```

4)Log4j的使用，直接测试运行刚才的查询

```
[org.apache.ibatis.transaction.jdbc.JdbcTransaction]-Opening JDBC Connection
[org.apache.ibatis.datasource.pooled.PooledDataSource]-Created connection 693267461.
[org.apache.ibatis.transaction.jdbc.JdbcTransaction]-Setting autocommit to false on
```

```
JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@29526c05]
[com.xiang.dao.UserMapper.getUserList]-> Preparing: select * from mybatis.user
[com.xiang.dao.UserMapper.getUserList]-> Parameters:
[com.xiang.dao.UserMapper.getUserList]-<== Total: 5    User{id=1, name='张三',
pwd='123'}    User{id=2, name='李四', pwd='456'}  User{id=3, name='王五',
pwd='789'}    User{id=5, name='wang', pwd='102'}  User{id=6, name='null',
pwd='102'}  [org.apache.ibatis.transaction.jdbc.JdbcTransaction]-Resetting
autocommit to true on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@29526c05]
[org.apache.ibatis.transaction.jdbc.JdbcTransaction]-Closing JDBC Connection
[com.mysql.cj.jdbc.ConnectionImpl@29526c05]
[org.apache.ibatis.datasource.pooled.PooledDataSource]-Returned connection 693267461
to pool.  Process finished with exit code 0
```

## Log4j的简单使用

1. 在要使用Log4j的类中，导入包import org.apache.log4j.Logger
2. 创建日志对象，参数为当前类的class

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

## 第四步

在要输出日志的类中加入相关语句：

定义属性： static Logger logger = Logger.getLogger(LogDemo.class); //LogDemo为相关的类

在相应的方法中：

```
if (logger.isDebugEnabled()){
    logger.debug("System ....");
}
```

译

### 3. 日志级别

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

## 7、分页

为什么要分页：减少数据的处理量

使用Limit分页

语法： SELECT \* FROM mybatis.user limit startIndex, pageSize;

```
SELECT * FROM mybatis.user limit 3; #[0, 3]
```

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="user">  
    select * from mybatis.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<>();  
    map.put("startIndex", 0);  
    map.put("pageSize", 2);  
    List<User> userList = mapper.getUserByLimit(map);  
    for (User user : userList) {  
        System.out.println(user);  
    }  
    sqlSession.close();  
}
```

使用RowBounds分页

不再使用SQL实现分页

### 1. 接口

```
//分页查询，老办法
```

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

### 2. Mapper.xml

```
<select id="getUserByRowBounds" resultType="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【了解即可】  
    List<User> userList =  
    sqlSession.selectList("com.xiang.dao.UserMapper.getUserByRowBounds", null, rowBounds);  
  
    for (User user : userList) {  
        System.out.println(user);  
    }  
    sqlSession.close();  
}
```

分页插件【了解即可】

# MyBatis 分页插件 PageHelper

如果你也在用 MyBatis，建议尝试该分页插件，这一定是最方便使用的分页插件。分页插件支持任何复杂的单表、多表分页。

[View on Github](#)

[View on GitOsc](#)

maven central 5.3.10

## 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多不同的不同对象协作完成的。在这种情况下，各个对象内部是如何实现自己的，对系统设计人员来讲就不那么重要了；
- 而各个对象之间的协作关系则成为系统设计的关键。小到不同类之间的通信，大到各模块之间的交互，在系统设计之初都是要着重考虑的，这也是系统设计的主要工作内容。面向接口编程就是指按照这种思想来编程。

#### 关于接口的理解

- 接口从更深层次的理解，应是定义（规范，约束）与实现（名实分离的原则）的分离。
- 接口的本身反映了系统设计人员对系统的抽象理解。
- 接口应有两类：
  - 第一类是对一个个体的抽象，它可对应为一个抽象体(abstract class);
  - 第二类是对一个个体某一方面的抽象，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。

#### 三个面向区别

- 面向对象是指，我们考虑问题时，以对象为单位，考虑它的属性及方法。
- 面向过程是指，我们考虑问题时，以一个具体的流程（事务过程）为单位，考虑它的实现。
- 接口设计与非接口设计是针对复用技术而言的，与面向对象（过程）不是一个问题.更多的体现就是对系统整体的架构

对于像 BlogMapper 这样的映射器类来说，还有另一种方法来处理映射。它们映射的语句可以不用 XML 来配置，而可以使用 Java 注解来配置。比如，上面的 XML 示例可被替换如下：

```
package org.mybatis.example;
public interface BlogMapper {
    @Select("SELECT * FROM blog WHERE id = #{id}")
    Blog selectBlog(int id);
}
```

使用注解来映射简单语句会使代码显得更加简洁，然而对于稍微复杂一点的语句，Java 注解就力不从心了，并且会显得更加混乱。因此，如果你需要完成很复杂的事情，那么最好使用 XML 来映射语句。

选择哪种方式来配置映射，以及认为映射语句定义的一致性是否重要，这些完全取决于你和你的团队。换句话说，永远不要拘泥于一种方式，你可以很轻松的在基于注解和 XML 的语句映射方式间自由移植和切换。

## 使用注解开发

### 1. 注解在接口上实现

```
@Select("select * from user")
```

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

### (2) 在mybatis核心配置文件中绑定接口

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

```
</mappers>
```

### (3) 测试

```
@Test
public void test() {
    SqlSession sqlSession = MybatisUtils.getSqlSession();
    //底层主要应用反射
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);
    List<User> users = mapper.getUsers();
    for (User user : users) {
        System.out.println(user);
    }

    sqlSession.close();
}
```

本质：反射机制实现

底层：动态代理

CRUD：

我们可以在工具类创建的时候实现自动提交事务，不设置默认为false

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

编写接口，增加注解

```
package com.xiang.dao;

import com.xiang.pojo.User;
import org.apache.ibatis.annotations.*;

import java.util.List;
import java.util.Map;

public interface UserMapper {

    @Select("select * from user")
    List<User> getUsers();
    //方法存在多个参数，所有的参数前面必须加上@Param("para")注解
    @Select("select * from user where id = #{id}")
    User getUserById(@Param("id") int id);

    @Insert("insert into user (id, name, pwd) values(#{id}, #{name}, #{pwd})")
    int addUser(User user); //引用类型无需加参数注解，基本类型才需要

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

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

测试类

```
import com.xiang.dao.UserMapper;
import com.xiang.pojo.User;
```

```

import com.xiang.utils.MybatisUtils;
import org.apache.ibatis.session.SqlSession;
import org.junit.Test;

import java.util.List;

public class UserMapperTest {
    @Test
    public void test() {
        SqlSession sqlSession = MybatisUtils.getSqlSession();
        //底层主要应用反射
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);

        // User userById = mapper.getUserById(1);
        // System.out.println(userById);

        // mapper addUser(new User(7, "tai", "111"));
        // mapper addUser(new User(8, "www", "111"));

        // mapper updateUser(new User(6, "xiang", "222"));

        mapper.deleteUser(8);

        sqlSession.close();
    }
}

```

注意：必须要将接口注册绑定到我们的核心配置文件中

```

<!--绑定接口-->
<mappers>
    <mapper class="com.xiang.dao.UserMapper"/>
    <!--<mapper resource="com/xiang/dao/*Mapper.xml"/>-->
</mappers>

```

关于@Param()注解：

- 基本类型的参数或者String类型，需要加上
- 引用类型参数不需要加
- 如果只有一个基本类型的话，可以忽略，建议加上
- 在SQL中引用的就是@Param(“uid”)中设定的属性名

#{} 和 \${} 的区别：

## 原sql语句:

```
delete from  
ups_role_permission_dataparams  
where role_id = #{roleId,jdbcType=INTEGER}
```

在这里用到了#{},使用#时:

1、用来传入参数，sql在解析的时候会加上“ ”,当成字符串来解析，如这里 role\_id = "roleid";

2、#{}能够很大程度上防止sql注入;

## 延伸:

1、用\${}传入数据直接显示在生成的sql中，如上面的语句，用role\_id = \${roleId,jdbcType=INTEGER},那么sql在解析的时候值为role\_id = roleId，执行时会报错；

2、\${}方式无法防止sql注入；

3、\$一般用入传入数据库对象，比如数据库表名；

4、能用#{},尽量用#{}；

**注意：Mybatis排序时使用order by 动态参数时需要注意，使用\${}而不用#{}。**

转载于<https://www.cnblogs.com/hskw/p/9957476.html>

## 1. PreparedStatement:

PreparedStatement是java.sql包下面的一个接口，用来执行SQL语句查询，通过调用connection.prepareStatement(sql)方法可以获得PreparedStatement对象。数据库系统会对sql语句进行预编译处理（如果JDBC驱动支持的话），预处理语句将被预先编译好，这条预编译的sql查询语句能在将来的查询中重用，这样一来，它比Statement对象生成的查询速度更快。

## 2. Statement

使用 Statement 对象。在对数据库只执行一次性存取的时候，用 Statement 对象进行处理。PreparedStatement 对象的开销比Statement大，对于一次性操作并不会带来额外的好处。

### 1、使用Statement而不是PreparedStatement对象

JDBC驱动的最佳化是基于使用的是什么功能. 选择PreparedStatement还是Statement取决于你要怎么使用它们. 对于只执行一次的SQL语句选择Statement是最好的. 相反，如果SQL语句被多次执行选用PreparedStatement是最好的. PreparedStatement的第一次执行消耗是很高的. 它的性能体现在后面的重复执行.

对于Statement，同一个查询只会产生一次网络到数据库的通讯.

对于使用PreparedStatement池的情况下，本指导原则有点复杂. 当使用PreparedStatement池时，如果一个查询很特殊，并且不太会再次执行到，那么可以使用Statement. 如果一个查询很少会被执行，但连接池中的Statement池可

能被再次执行，那么请使用PreparedStatement。在不是Statement池的同样情况下，请使用Statement。

### 1. 使用Statement对象

使用范围：当执行相似SQL(结构相同，具体值不同)语句的次数比较少

优点：语法简单

缺点：采用硬编码效率低，安全性较差。

原理：硬编码，每次执行时相似SQL都会进行编译

### 2. 预编译PreparedStatement

使用范围：当执行相似sql语句的次数比较多（例如用户登陆，对表频繁操作..）语句一样，只是具体的值不一样，被称为动态SQL

优点：语句只编译一次，减少编译次数。提高了安全性（阻止了SQL注入）

缺点：执行非相似SQL语句时，速度较慢。

原理：相似SQL只编译一次，减少编译次数

### 3. 使用PreparedStatement + 批处理

使用范围：一次需要更新数据库表多条记录

优点：减少和SQL引擎交互的次数，再次提高效率，相似语句只编译一次，减少编译次数。提高了安全性（阻止了SQL注入）

缺点：

原理：批处理：减少和SQL引擎交互的次数，一次传递给SQL引擎多条SQL。

名词解释：

PL/SQL引擎：在oracle中执行pl/sql代码的引擎，在执行中发现标准的sql会交给sql引擎进行处理。

SQL引擎：执行标准sql的引擎。

### 1. 代码的可读性和可维护性.

虽然用PreparedStatement来代替Statement会使代码多出几行，但这样的代码无论从可读性还是可维护性上来说，都比直接用Statement的代码高很多档次：

关于PreparedStatement接口，需要重点记住的是：

1. PreparedStatement可以写参数化查询，比Statement能获得更好的性能。
2. 对于PreparedStatement来说，数据库可以使用已经编译过及定义好的执行计划，这种预处理语句查询比普通的查询运行速度更快。
3. PreparedStatement可以阻止常见的SQL注入式攻击。
4. PreparedStatement可以写动态查询语句
5. PreparedStatement与java.sql.Connection对象是关联的，一旦你关闭了connection，PreparedStatement也没法使用了。

6. “?” 叫做占位符。
7. PreparedStatement查询默认返回FORWARD\_ONLY的ResultSet，你只能往一个方向移动结果集的游标。当然你还可以设定为其他类型的值如：“CONCUR\_READ\_ONLY”。
8. 不支持预编译SQL查询的JDBC驱动，在调用connection.prepareStatement(sql)的时候，它不会把SQL查询语句发送给数据库做预处理，而是等到执行查询动作的时候（调用executeQuery()方法时）才把查询语句发送给数据库，这种情况和使用Statement是一样的。
9. 占位符的索引位置从1开始而不是0，如果填入0会导致java.sql.SQLException invalid column index异常。所以如果PreparedStatement有两个占位符，那么第一个参数的索引时1，第二个参数的索引是2。

## 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.

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>1.18.20</version>
</dependency>
```

3. 在实体类上加注解

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class User {

    private int id;
    private String name;
    private String pwd;
}
```

```
1 @Getter and @Setter
2 @FieldNameConstants
3 @ToString
4 @EqualsAndHashCode
5 @AllArgsConstructor, @RequiredArgsConstructor and @NoArgsConstructor
6 @Log, @Log4j, @Log4j2, @Slf4j, @XSlf4j, @CommonsLog, @JBossLog, @Flogger
7 @Data
8 @Builder
9 @Singular
10 @Delegate
11 @Value
12 @Accessors
13 @Wither
14 @SneakyThrows|
```

注解自动生成方法说明：

@Data: 无参构造，get, set, toString, hashCode

@AllArgsConstructor: 有参构造方法

@NoArgsConstructor: 无参构造方法

### Lombok的优缺点

优点：

1. 能通过注解的形式自动生成构造器、getter/setter、equals、hashCode、toString等方法，提高了一定的开发效率
2. 让代码变得简洁，不用过多的去关注相应的方法
3. 属性做修改时，也简化了维护为这些属性所生成的getter/setter方法等

缺点：

1. 不支持多种参数构造器的重载
2. 虽然省去了手动创建getter/setter方法的麻烦，但大大降低了源代码的可读性和完整性，降低了阅读源代码的舒适度

## 总结

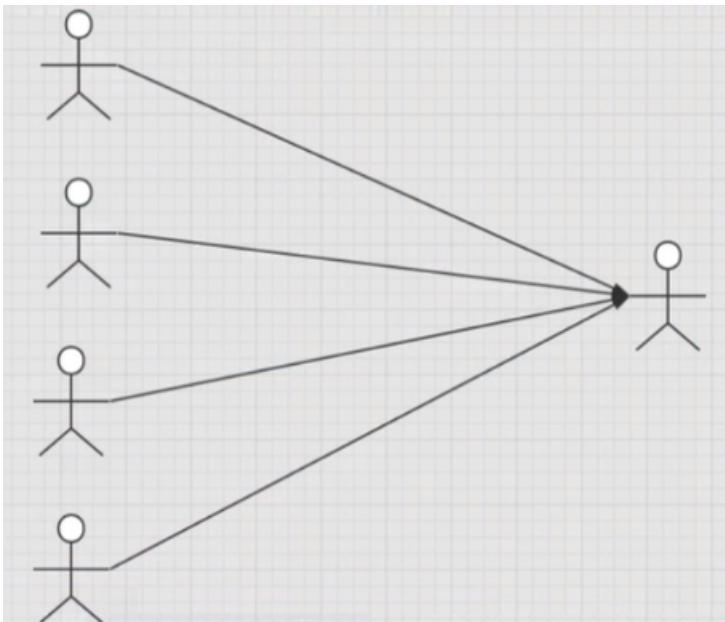
Lombok虽然有很多优点，但Lombok更类似于一种IDE插件，项目也需要依赖相应的jar包。Lombok依赖jar包是因为编译时要用它的注解，为什么说它又类似插件？因为在使用时，eclipse或IntelliJ IDEA都需要安装相应的插件，在编译器编译时通过操作AST（抽象语法树）改变字节码生成，变向的就是说它在改变java语法。

它不像spring的依赖注入或者mybatis的ORM一样是运行时的特性，而是编译时的特性。这里我个人最感觉不爽的地方就是对插件的依赖！因为Lombok只是省去了一些人工生成代码的麻烦，但IDE都有快捷键来协助生成getter/setter等方法，也非常方便。

知乎上有位大神发表过对Lombok的一些看法：

这是一种低级趣味的插件，不建议使用。JAVA发展到今天，各种插件层出不穷，如何甄别各种插件的优劣？能从架构上优化你的设计的，能提高应用程序性能的，实现高度封装可扩展的...，像lombok这种，像这种插件，已经不仅仅是插件了，改变了你如何编写源码，事实上，少去了的代码，你写上去又如何？如果JAVA家族到处充斥这样的东西，那只不过是一堆披着金属颜色的屎，迟早会被其它的语言取代。

## 10、多对一处理



好比多个学生，对应一个老师

对于学生这边而言，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

```
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, "王祥太");
```

```

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', '小米', '1');
INSERT INTO `student`(`id`, `name`, `tid`) VALUES ('2', '华为', '1');
INSERT INTO `student`(`id`, `name`, `tid`) VALUES ('3', '三星', '1');
INSERT INTO `student`(`id`, `name`, `tid`) VALUES ('4', '苹果', '1');
INSERT INTO `student`(`id`, `name`, `tid`) VALUES ('5', 'oppo', '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
6. 测试查询是否能够成功

Id 和 Result 的属性	
属性	描述
<code>property</code>	映射到列结果的字段或属性。如果用来匹配的 JavaBean 存在给定名字的属性，那么它将会被使用。否则 MyBatis 将会寻找给定名称的字段。无论是哪一种情形，你都可以使用通常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”。
<code>column</code>	数据库中的列名，或者是列的别名。一般情况下，这和传递给 <code>resultSet.getString(columnName)</code> 方法的参数一样。
<code>javaType</code>	一个 Java 类的完全限定名，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 <code>HashMap</code> ，那么你应该明确地指定 <code>javaType</code> 来保证行为与期望的一致。
<code>jdbcType</code>	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，你需要对可能存在空值的列指定这个类型。
<code>typeHandler</code>	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完全限定名，或者是类型别名。
属性	描述
<code>column</code>	数据库中的列名，或者是列的别名。一般情况下，这和传递给 <code>resultSet.getString(columnName)</code> 方法的参数一样。
<code>javaType</code>	一个 Java 类的完全限定名，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 <code>HashMap</code> ，那么你应该明确地指定 <code>javaType</code> 来保证行为与期望的一致。
<code>jdbcType</code>	JDBC 类型，所支持的 JDBC 类型参见这个表格之前的“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，你需要对可能存在空值的列指定这个类型。
<code>typeHandler</code>	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完全限定名，或者是类型别名。
<code>select</code>	用于加载复杂类型属性的映射语句的 ID，它会从 <code>column</code> 属性中指定的列检索数据，作为参数传递给此 <code>select</code> 语句。具体请参考关联元素。
<code>resultMap</code>	结果映射的 ID，可以将嵌套的结果集映射到一个合适的对象树中。它可以作为使用额外 <code>select</code> 语句的替代方案。它可以将多表连接操作的结果映射成一个单一的 <code>ResultSet</code> ，这样的 <code>ResultSet</code> 将会包含重复或部分数据重复的结果集。为了将结果集正确地映射到嵌套的对象树中，MyBatis 允许你“串联”结果映射，以便解决嵌套结果集的问题。想了解更多内容，请参考下面的关联元素。
<code>name</code>	构造方法形参的名字。从 3.4.3 版本开始，通过指定具体的参数名，你可以以任意顺序写出 <code>arg</code> 元素。参看上面的解释。

## 按照查询嵌套处理：

```

<?xml version="1.0" encoding="UTF8" ?>
<!DOCTYPE mapper
    PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="com.xiang.dao.StudentMapper">
    <!--思路：1. 查询所有的学生信息 2. 根据查询出来的学生的tid, 寻找对应的老师 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，我们需要单独处理 对象：association 多对一； 集合：
collection 一对多-->
    <association property="teacher" column="tid" javaType="Teacher"
select="getTeacher"/>
    </resultMap>
    <select id="getTeacher" resultType="Teacher">
        select * from teacher where id = #{id}
    </select>
</mapper>

```

按照结果嵌套处理：

```

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

<resultMap id="StudentTeacher2" type="Student">
    <result property="id" column="sid"/>
    <result property="name" column="sname"/>
    <association property="teacher" javaType="Teacher">
        <result property="name" column="tname"/>
    </association>
</resultMap>

```

回顾Mysql多对一查询方式：

- 子查询
- 联表查询

## 11、一对多处理

比如：一个老师拥有多个学生，对于老师而言，就是一对多的关系

(1) 环境搭建，同上

实体类

@Data

```

public class Student {
    private int id;
    private String name;
    //学生需要关联一个老师
    private int tid;
}

```

@Data

```

public class Teacher {
    private int id;
    private String name;
    //一个老师拥有多个学生，一对多
}

```

```
    private List<Student> students;  
}
```

按照结果嵌套处理

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

按照子查询

```
<select id="getTeacher2" resultMap="TeacherStudent2">  
    select * from mybatis.teacher where id = #{ttid}  
</select>  
<resultMap id="TeacherStudent2" type="Teacher">  
    <collection property="students" javaType="ArrayList" ofType="Student"  
    select="getStudentByTeacherId" column="id"/>  
</resultMap>  
  
<select id="getStudentByTeacherId" resultType="Student">  
    select * from mybatis.student where tid = #{ttid2}  
</select>
```

小结

1. 关联— association (多对一)
2. 集合— collection (一对多)
3. JavaType ofType
  1. javaType用来指定实体类中属性的类型
  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，可以使用日志，建议使用log4j

`resultMap` 元素有很多子元素和一个值得深入探讨的结构。下面是 `resultMap` 元素的概念视图。

### 结果映射 (`resultMap`)

- `constructor` - 用于在实例化类时，注入结果到构造方法中
  - + `idArg` - ID 参数；标记出作为 ID 的结果可以帮助提高整体性能
  - `arg` - 将被注入到构造方法的一个普通结果
- `id` - 一个 ID 结果；标记出作为 ID 的结果可以帮助提高整体性能
- `result` - 注入到字段或 JavaBean 属性的普通结果
- `association` - 一个复杂类型的关联；许多结果将包装成这种类型
  - 嵌套结果映射 - 关联本身可以是一个 `resultMap` 元素，或者从别处引用一个
- `collection` - 一个复杂类型的集合
  - 嵌套结果映射 - 集合本身可以是一个 `resultMap` 元素，或者从别处引用一个
- `discriminator` - 使用结果值来决定使用哪个 `resultMap`
  - `case` - 基于某些值的结果映射
    - 嵌套结果映射 - `case` 本身可以是一个 `resultMap` 元素，因此可以具有相同的结构和元素，或者从别处引用一个

`ResultMap` 的属性列表

面试高频：

Mysql引擎

InnoDB底层原理

索引

索引优化

## 12、动态SQL

所谓动态SQL就是指根据不同的条件生成不同的SQL语句

```
1 动态 SQL 元素和 JSTL 或基于类似 XML 的文本处理器相似。在 MyBatis 之前的版本中，有很多元素需要花时间了解。MyBatis 3 大大精简了元素种类，现在只需学习原来一半的元素便可。MyBatis 采用功能强大的基于 OGNL 的表达式来淘汰其它大部分元素。  
2  
3 if  
4 choose (when, otherwise)  
5 trim (where, set)  
6 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
```

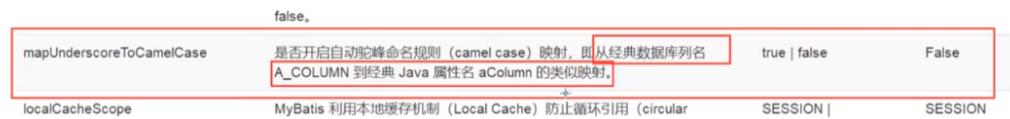
创建一个基础工程

1. 导包lombok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; //java里面用util.Date
    private int views;
}
```

4. 编写实体类对应的Mapper接口和Mapper.xml文件

### 设置表字段自动映射驼峰属性



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

### 新加工具类:

```
package com.xiang.utils;

import org.junit.Test;

import java.util.UUID;
//获得一个随机的id
public class IDUtils {
    public static String getId() {
        return UUID.randomUUID().toString().replaceAll("-", "");
    }

    @Test
    public void test() {
        System.out.println(IDUtils.getId());
        System.out.println(IDUtils.getId());
        System.out.println(IDUtils.getId());
    }
}
```

### 利用测试类插入数据:

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

```

Blog blog = new Blog();
blog.setId(IDUtils.getId());
blog.setTitle("好好学习");
blog.setAuthor("王祥太");
blog.setCreateTime(new Date());
blog.setViews(999);
mapper.addBlog(blog);

blog.setId(IDUtils.getId());
blog.setTitle("天天向上");
mapper.addBlog(blog);

blog.setId(IDUtils.getId());
blog.setTitle("行路难");
mapper.addBlog(blog);

blog.setId(IDUtils.getId());
blog.setTitle("今安在");
mapper.addBlog(blog);

sqlSession.close();
}

```

## (1) IF实现动态SQL

**if**

动态 SQL 通常要做的事情是根据条件包含 where 子句的一部分。比如：

```

<select id="findActiveBlogWithTitleLike"
       resultType="Blog">
    SELECT * FROM BLOG
    WHERE state = 'ACTIVE'
    <if test="title != null">
        AND title like #{title}
    </if>
</select>

```

这条语句提供了一种可选的查找文本功能。如果没有传入“title”，那么所有处于“ACTIVE”状态的BLOG都会返回；[反之若传入了“title”](#)，那么就会对“title”一列进行模糊查找并返回 BLOG 结果（细心的读者可能会发现，“title”参数值是可以包含一些掩码或通配符的）。

如果希望通过“title”和“author”两个参数进行可选搜索该怎么办呢？首先，改变语句的名称让它更具实际意义；然后只要加入另一个条件即可。

```

<select id="findActiveBlogLike"
       resultType="Blog">
    SELECT * FROM BLOG WHERE state = 'ACTIVE'
    <if test="title != null">
        AND title like #{title}
    </if>
    <if test="author != null and author.name != null">
        AND author_name like #{author.name}
    </if>
</select>

```

BlogMapper接口：

```

import com.xiang.pojo.Blog;

import java.util.List;
import java.util.Map;

public interface BlogMapper {
    //插入数据
    int addBlog(Blog blog);
    //按条件查询
    List<Blog> queryBlogByIF(Map map);
}

```

```
}
```

## BlogMapper.xml

```
<?xml version="1.0" encoding="UTF8" ?>
<!DOCTYPE mapper
  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.xiang.dao.BlogMapper">
  <insert id="addBlog" parameterType="Blog">
    insert into mybatis.blog (id, title, author, create_time, views)
    values (#{}id, #{}title, #{}author, #{}createTime, #{}views);
  </insert>

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

`where` 元素只会在至少有一个子元素的条件返回 SQL 子句的情况下才去插入“WHERE”子句，而且，若语句的开头为“AND”或“OR”，`where` 元素也会将它们去除。

## 测试：

```
import com.xiang.dao.BlogMapper;
import com.xiang.pojo.Blog;
import com.xiang.utils.IDUtils;
import com.xiang.utils.MybatisUtils;
import org.apache.ibatis.session.SqlSession;
import org.junit.Test;

import java.util.Date;
import java.util.HashMap;
import java.util.List;

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

        Blog blog = new Blog();
        blog.setId(IDUtils.getId());
        blog.setTitle("好好学习");
        blog.setAuthor("王祥太");
        blog.setCreateTime(new Date());
        blog.setViews(999);
        mapper.addBlog(blog);

        blog.setId(IDUtils.getId());
        blog.setTitle("天天向上");
        mapper.addBlog(blog);

        blog.setId(IDUtils.getId());
        blog.setTitle("行路难");
        mapper.addBlog(blog);

        blog.setId(IDUtils.getId());
```

```

blog.setTitle("今安在");
mapper.addBlog(blog);

    sqlSession.close();
}

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

    HashMap map = new HashMap();
    map.put("title", "行路难");
    map.put("author", "王祥太");

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

```

## choose、when、otherwise实现动态SQL:

### choose, when, otherwise

有时我们不想应用到所有的条件语句，而只想从中择其一项。针对这种情况，MyBatis 提供了 choose 元素，它有点像 Java 中的 switch 语句。

还是上面的例子，但是这次变为提供了“title”就按“title”查找，提供了“author”就按“author”查找的情形，若两者都没有提供，就返回所有符合条件的 BLOG（实际情况可能是由管理员按一定策略选出 BLOG 列表，而不是返回大量无意义的随机结果）。

```

<select id="findActiveBlogLike"
       resultType="Blog">
    SELECT * FROM BLOG WHERE state = 'ACTIVE'
    <choose>
        <when test="title != null">
            AND title like #{title}
        </when>
        <when test="author != null and author.name != null">
            AND author_name like #{author.name}
        </when>
        <otherwise>
            AND featured = 1
        </otherwise>
    </choose>
</select>

```

这会导致查询失败。如果仅仅第二个条件匹配又会怎样？这条 SQL 最终会是这样：

```

SELECT * FROM BLOG
WHERE
AND title like 'someTitle'

```

这个查询也会失败。这个问题不能简单地用条件句式来解决，如果你也曾经被迫这样写过，那么你很可能从此以后都不会再写出这种语句了。

MyBatis 有一个简单的处理，这在 90% 的情况下都会有用。而在不能使用的地方，你可以自定义处理方式来令其正常工作。一处简单的修改就能达到目的：

```

<select id="findActiveBlogLike"
       resultType="Blog">
    SELECT * FROM BLOG
    <where>
        <if test="state != null">
            state = #{state}
        </if>
        <if test="title != null">
            AND title like #{title}
        </if>
        <if test="author != null and author.name != null">
            AND author_name like #{author.name}
        </if>
    </where>
</select>

```

## BlogMapper接口

```
List<Blog> queryBlogByChoose(Map map);
```

BlogMapper.xml:

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

测试:

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

    HashMap map = new HashMap();
    map.put("title", "行路难");
    map.put("author", "王祥太");
    map.put("views", 999);

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

## trim ( where , set )

`prefixOverrides` 属性会忽略通过管道分隔的文本序列（注意此例中的空格也是必要的）。它的作用是移除所有指定在 `prefixOverrides` 属性中的内容，并且插入 `prefix` 属性中指定的内容。

类似的用于动态更新语句的解决方案叫做 `set`。`set` 元素可以用于动态包含需要更新的列，而舍去其它的。比如：

```
<update id="updateAuthorIfNecessary">
    update Author
    <set>
        <if test="username != null">username=#{username},</if>
        <if test="password != null">password=#{password},</if>
        <if test="email != null">email=#{email},</if>
        <if test="bio != null">bio=#{bio}</if>
    </set>
    where id=#{id}
</update>
```

这里，`set` 元素会动态前置 `SET` 关键字，同时也会删掉无关的逗号，因为用了条件语句之后很可能就会在生成的 SQL 语句的后面留下这些逗号。（译者注：因为用的是`if`元素，若最后一个`if`没有匹配上而前面的匹配上，SQL语句的最后就会有一个逗号遗留）

若你对 `set` 元素等价的自定义 `trim` 元素的代码感兴趣，那这就是它的真面目：

这里，`set` 元素会动态前置 `SET` 关键字，同时也会删掉无关的逗号，因为用了条件语句之后很可能就会在生成的 SQL 语句的后面留下这些逗号。（译者注：因为用的是`if`元素，若最后一个`if`没有匹配上而前面的匹配上，SQL语句的最后就会有一个逗号遗留）

若你对 `set` 元素等价的自定义 `trim` 元素的代码感兴趣，那这就是它的真面目：

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

注意这里我们删去的是后缀值，同时添加了前缀值。

```
set实现动态SQL  
BlogMapper接口  
//更新博客  
int updateBlog(Map map);  
BlogMapper.xml:
```

```
<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>
```

测试：

```
@Test  
public void updateBlog() {  
    SqlSession sqlSession = MybatisUtils.getSqlSession();  
    BlogMapper mapper = sqlSession.getMapper(BlogMapper.class);  
  
    //  
    HashMap map = new HashMap();  
    map.put("title", "行路难2");  
    map.put("author", "王祥太3号");  
    map.put("id", "3d5266d98e1145f0898eb89fef0a0ce");  
    mapper.updateBlog(map);  
    sqlSession.close();  
}
```

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

If where set choose when

## SQL片段

有的时候，我们可能会将一些公共的部分抽取出来，方便复用

### 1. 使用sql标签抽取公共部分

```
<!--将重复用到的部分，抽取成sql片段，需要用到的地方可以直接引用这个片段-->  
<sql id="if-title-author">  
    <if test="title != null">  
        title = #{title}  
    </if>  
    <if test="author != null">  
        and author = #{author}  
    </if>  
</sql>
```

## 2. 在需要使用的地方使用include标签引用即可

```
<select id="queryBlogByIF" parameterType="map" resultType="Blog">
    select * from mybatis.blog
    <where>
        <include refid="if-title-author"></include>
        <!-- <if test="title != null">
            title = #{title}
        </if>
        <if test="author != null">
            and author=#{author}
        </if>-->
    </where>
</select>
```

注意事项：

- 最好基于单表来定义SQL片段
- 不要存在where标签

## foreach

### foreach

动态 SQL 的另外一个常用的操作需求是对一个集合进行遍历，通常是在构建 IN 条件语句的时候。比如：

```
<select id="selectPostIn" resultType="domain.blog.Post">
    SELECT *
    FROM POST P
    WHERE ID in
        <foreach item="item" index="index" collection="list"
            open="(" separator="," close=")"
            item="#{item}"
        </foreach>
    </select>
```

(1, 2,3,4,5)

foreach 元素的功能非常强大，它允许你指定一个集合，声明可以在元素体内使用的集合项 (item) 和索引 (index) 变量。它也允许你指定开头与结尾的字符串以及在迭代结果之间放置分隔符。这个元素是很智能的，因此它不会偶然地附加多余的分隔符。

**注意** 你可以将任何可迭代对象（如 List、Set 等）、Map 对象或者数组对象传递给 foreach 作为集合参数。当使用可迭代对象或者数组时，index 是当前迭代的次数，item 的值是本次迭代获取的元素。当使用 Map 对象（或者 Map.Entry 对象的集合）时，index 是键，item 是值。

至此我们已经完成了涉及 XML 配置文件和 XML 映射文件的讨论。下一章将详细探讨 Java API，这样就能提高已创建的映射文件的利用效率。

BlogMapper 接口：

```
//查询多个title的博客
List<Blog> queryBlogByForeach(Map map);
```

BlogMapper.xml :

```
<!--我们传递一个万能的map，这个map中可以存在一个集合-->
<select id="queryBlogByForeach" parameterType="map" resultType="blog">
    select * from mybatis.blog
    <where>
        <foreach collection="titles" item="title" open="(" separator="or" close=")">
            title = #{title}
        </foreach>
    </where>
</select>
```

测试：

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

    HashMap map = new HashMap();
    ArrayList<String> titles = new ArrayList<>();
```

```
titles.add("好好学习");
titles.add("天天向上");

map.put("titles", titles);
List<Blog> blogs = mapper.queryBlogByForeach(map);

for (Blog blog : blogs) {
    System.out.println(blog);
}
sqlSession.close();
}
```

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，在对应的去修改成为我们动态SQL实现通用即可

## 13、缓存

### (1) 简介

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，直接从缓存中查询，从而提高查询效率，解决了高并发系统的性能问题

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

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

### (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、二级缓存

默认情况下，只有一级缓存开启，（SqlSession级别的LRU缓存，也称为本地缓存）。

二级缓存需要手动开启和配置，他基于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Cache接口来自定义二级缓存。

LRU：最近最少使用

一级缓存：一次数据库连接完成的缓存。

这个更高级的配置创建了一个 FIFO 缓存，每隔 60 秒刷新，最多可以存储结果对象或列表的 512 个引用，而且返回的对象被认为是只读的，因此对它们进行修改可能会在不同线程中的调用者产生冲突。

可用的清除策略有：

- 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们。
- 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。

默认的清除策略是 LRU。

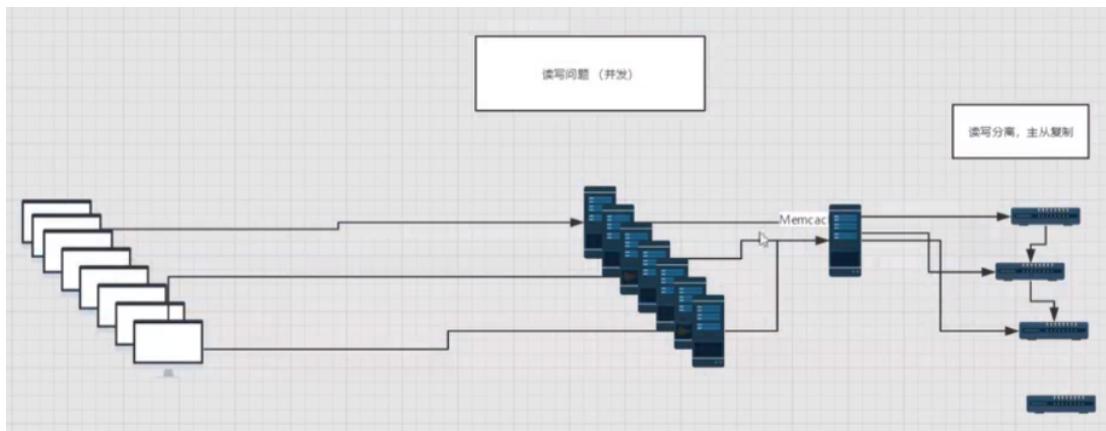
flushInterval (刷新间隔) 属性可以被设置为任意的正整数，设置的值应该是一个以毫秒为单位的合理时间量。默认情况是不设置，也就是没有刷新间隔，缓存仅仅会在调用语句时刷新。

size (引用数目) 属性可以被设置为任意正整数，要注意欲缓存对象的大小和运行环境中可用的内存资源。默认值是 1024。

readOnly (只读) 属性可以被设置为 true 或 false。只读的缓存会给所有调用者返回缓存对象的相同实例。因此这些对象不能被修改。这就提供了可观的性能提升。而可读写的缓存会（通过序列化）返回缓存对象的拷贝。速度上会慢一些，但是更安全，因此默认值是 false。

**提示** 二级缓存是事务性的。这意味着，当 SqlSession 完成并提交时，或是完成并回滚，但没有执行 flushCache=true 的 insert/delete/update 语句时，缓存会获得更新。

## 主从复制



### (3) 一级缓存

测试缓存步骤：

1. 启开日志
2. 测试在一个Session中查询两次相同记录
3. 查看日志输出

```
PooledDataSource forcefully closed/removed all connections.  
PooledDataSource forcefully closed/removed all connections.  
PooledDataSource forcefully closed/removed all connections.  
Opening JDBC Connection  
Created connection 811760110.  
==> Preparing: select * from user where id = ?  
==> Parameters: 1(Integer)  
<== Columns: id, name, pwd  
<== Row: 1, 狂神, 123456  
<== Total: 1  
User(id=1, name=狂神, pwd=123456)  
=====  
User(id=1, name=狂神, pwd=123456)  
true  
Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@306279ee]  
Returned connection 811760110 to pool.  
Process finished with exit code 0
```

缓存失效的情况：

## 缓存

MyBatis 内置了一个强大的事务性查询缓存机制，它可以非常方便地配置和定制。为了使它更加强大而且易于配置，我们对 MyBatis 3 中的缓存实现进行了许多改进。

默认情况下，只启用了本地的会话缓存，它仅仅对一个会话中的数据进行缓存。要启用全局的二级缓存，只需要在你的 SQL 映射文件中添加一行：

```
<cache/>
```

基本上就是这样。这个简单语句的效果如下：

- 映射语句文件中的所有 select 语句的结果将会被缓存。
- 映射语句文件中的所有 insert、update 和 delete 语句会刷新缓存。
- 缓存会使用最近最少使用算法（LRU, Least Recently Used）算法来清除不需要的缓存。
- 缓存不会定时进行刷新（也就是说，没有刷新间隔）。
- 缓存会保存列表或对象（无论查询方法返回哪种）的 1024 个引用。
- 缓存会被视为读/写缓存，这意味着获取到的对象并不是共享的，可以安全地被调用者修改，而不干扰其他调用者或线程所做的潜在修改。

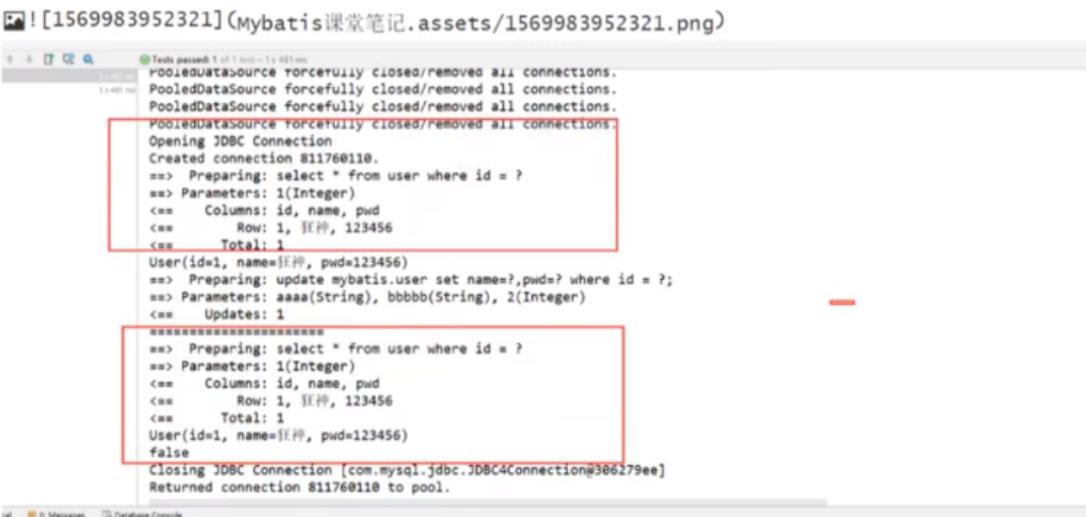
**提示** 缓存只作用于 cache 标签所在的映射文件中的语句。如果你混合使用 Java API 和 XML 映射文件，在共用接口中的语句将不会被默认缓存。你需要使用 @CacheNamespaceRef 注解指定缓存作用域。

这些属性可以通过 cache 元素的属性来修改。比如：

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

这个更高级的配置创建了一个 FIFO 缓存，每隔 60 秒刷新，最多可以存储结果对象或列表的 512 个引用，而且返回的对象被认为是只读的，因此对它们进行修改可能会在不同线程中的调用者产生冲突。

1. 查询不同的记录。
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。

 ! [1569983952321] (Mybatis课堂笔记.assets/1569983952321.png)

3. 查询不同Mapper.xml（不仅是一级缓存，二级缓存都不在了）。

4) 手动清理缓存。sqlSession.clearCache(); //手动清理缓存

```

PooledDataSource forcefully closed/removed all connections.
Opening JDBC Connection
Created connection 811760110.
==> Preparing: select * from user where id = ?
==> Parameters: 1(Integer)
<==    Columns: id, name, pwd
<==        Row: 1, 狂神, 123456
<==    Total: 1
User(id=1, name=狂神, pwd=123456)
=====
==> Preparing: select * from user where id = ?
==> Parameters: 1(Integer)
<==    Columns: id, name, pwd
<==        Row: 1, 狂神, 123456
<==    Total: 1
User(id=1, name=狂神, pwd=123456)
false
Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@306279ee]
Returned connection 811760110 to pool.

```

```

public class MyTest {
    @Test
    public void queryUserById() {
        SqlSession sqlSession = MybatisUtils.getSqlSession();
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);

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

        // mapper.updateUser(new User(2, "xxx", "www"));
        sqlSession.clearCache(); //手动清理缓存

        System.out.println("=====");
        User user2 = mapper.queryUserById(1);
        System.out.println(user2);
        System.out.println(user == user2);
        sqlSession.close();
    }
}

```

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

一级缓存相当于一个Map

**提示** 缓存只作用于 `cache` 标签所在的映射文件中的语句。如果你混合使用 Java API 和 XML 映射文件，在共用接口中的语句将不会被默认缓存。你需要使用 `@CacheNamespaceRef` 注解指定缓存作用域。

这些属性可以通过 `cache` 元素的属性来修改。比如：

```

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

```

这个更高级的配置创建了一个 FIFO 缓存，每隔 60 秒刷新，最多可以存储结果对象或列表的 512 个引用，而且返回的对象被认为是只读的，因此对它们进行修改可能会在不同线程中的调用者产生冲突。

可用的清除策略有：

- **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们。
- **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。

默认的清除策略是 LRU。

### 3. 二级缓存

- 二级缓存也叫全局缓存，一级缓存的作用域太低了，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。

步骤：

1) 开启全局缓存：

设置名	描述	有效值	默认值
cacheEnabled	全局地开启或关闭配置文件中的所有映射器已经配置的任何缓存。	true   false	true

<!--显示地开启全局缓存（二级缓存）-->

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

2) 在要使用二级缓存的Mapper.xml中开启

<!--开启二级缓存-->

```
<cache/>
```

也可以自定义参数：

<!--在当前Mapper.xml中使用二级缓存，  
使用FIFO缓存策略，  
没隔60秒刷新缓存，  
最多可以存储结果或列表的512个引用，  
只读-->

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

#### 4. 测试：

```
import com.xiang.dao.UserMapper;
import com.xiang.pojo.User;
import com.xiang.utils.MybatisUtils;
import org.apache.ibatis.session.SqlSession;
import org.junit.Test;

public class MyTest {
    @Test
    public void queryUserById() {

        SqlSession sqlSession = MybatisUtils.getSqlSession();
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
        User user = mapper.queryUserById(1);
        System.out.println(user);
        sqlSession.close();

        SqlSession sqlSession2 = MybatisUtils.getSqlSession();
        UserMapper mapper2 = sqlSession2.getMapper(UserMapper.class);
        User user1 = mapper2.queryUserById(1);
        System.out.println(user1);
        System.out.println(user == user1);
        sqlSession2.close();

    }
}
```

问题：

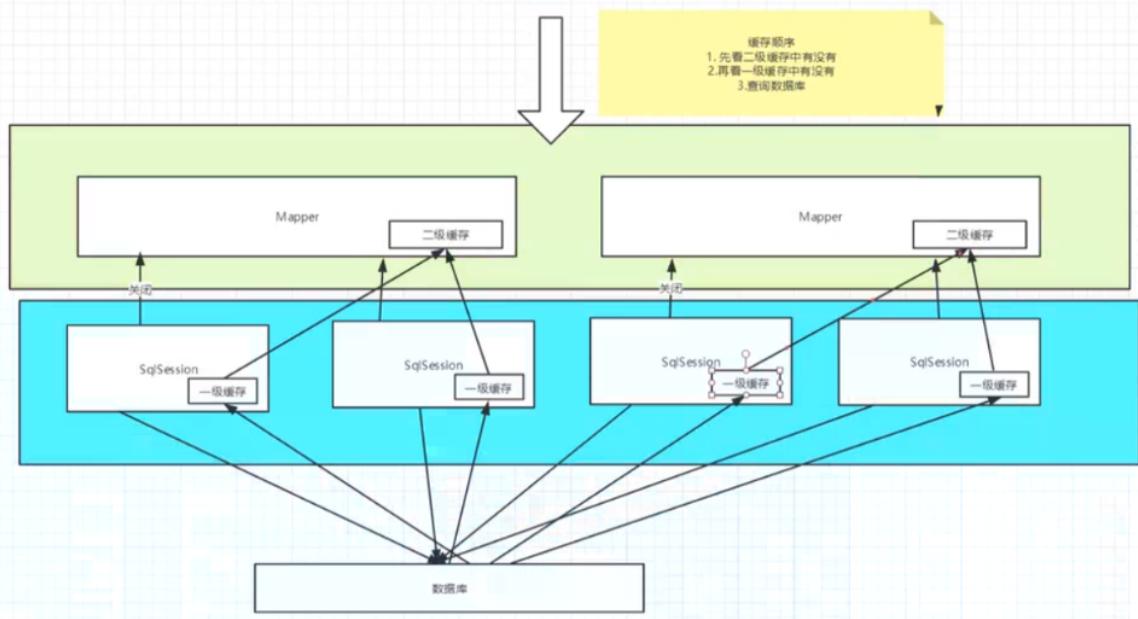
1. 直接使用<cache/>开启缓存，未设置缓存策略时，需要序列化实体类。否则会出现：

```
Cause: java.io.NotSerializableException: com.xiang.pojo.User 异常
PooledDataSource forcefully closed/removed all connections.
Cache Hit Ratio [com.xiang.dao.UserMapper]: 0.0
Opening JDBC Connection
Created connection 71016405.
==> Preparing: select * from user where id = ?
==> Parameters: 1(Integer)
<== Columns: id, name, pwd
<== Row: 1, 张三, 123
<== Total: 1
User(id=1, name=张三, pwd=123)
Closing JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@43b9fd5]
Returned connection 71016405 to pool.
Cache Hit Ratio [com.xiang.dao.UserMapper]: 0.5
User(id=1, name=张三, pwd=123)
false
```

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，再次执行同一个Mapper的会话时，会去二级缓存查找。

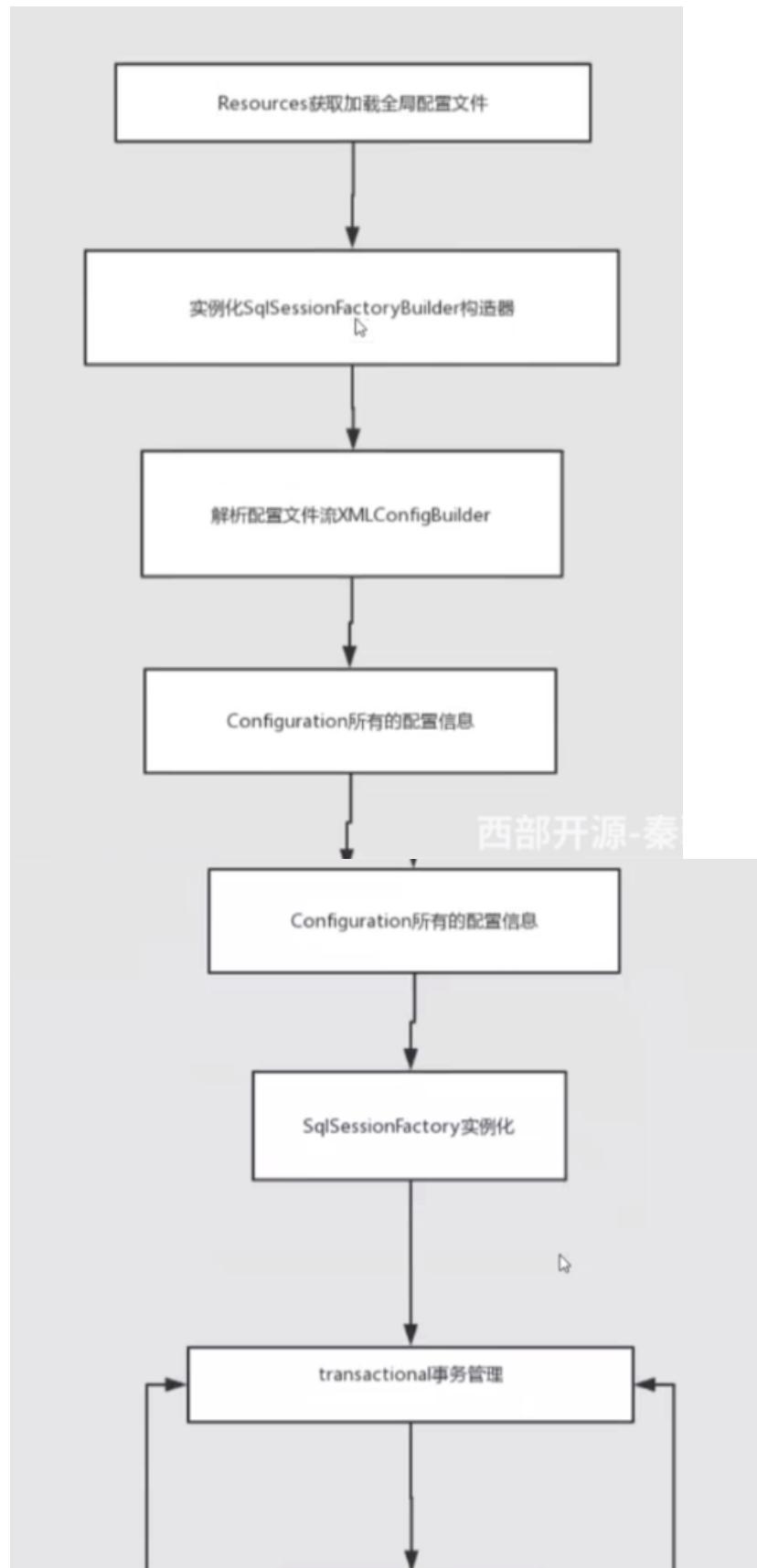
## (5) 缓存原理

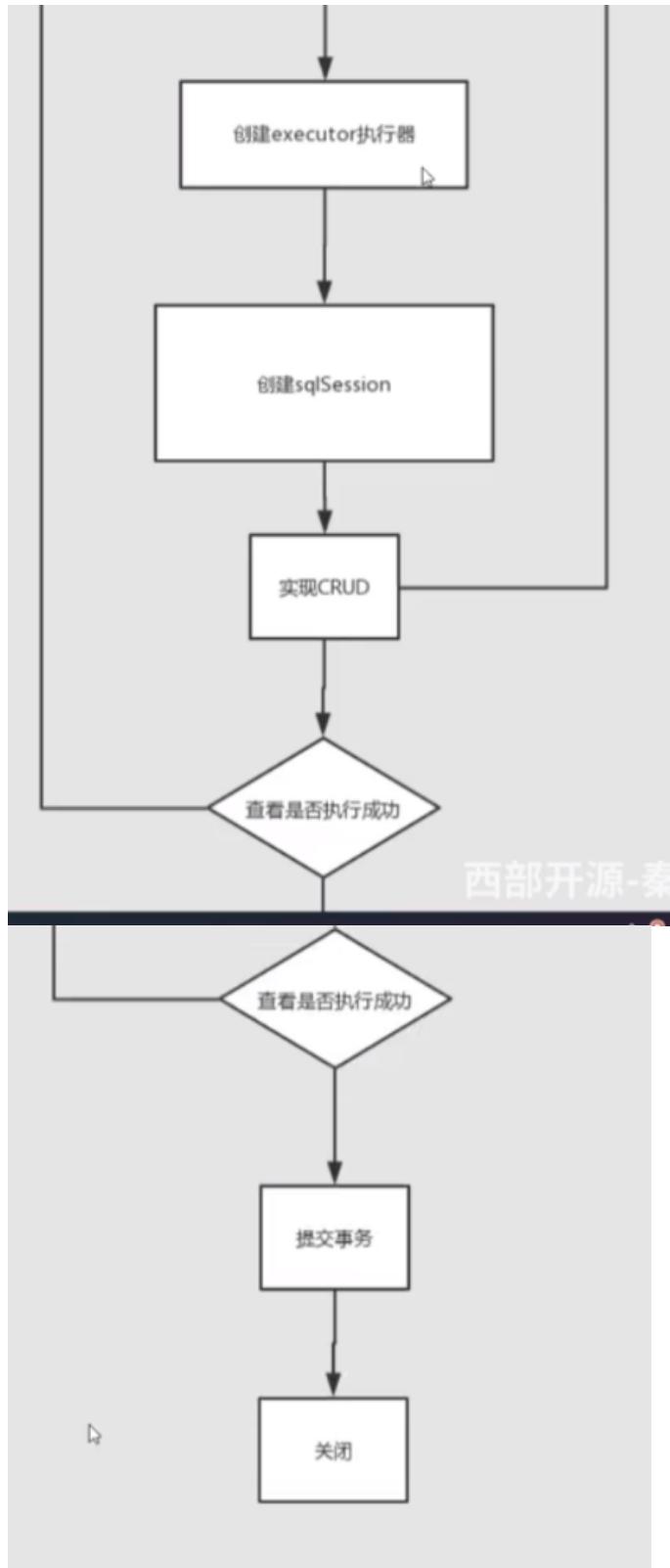


缓存只是为了提高查询的效率

## (6) 自定义缓存—ehcache

EhCache是一个纯Java的进程内缓存框架，具有快速、精干等特定，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  
在程序中使用ehcache，先要导包，mybatis-ehcache  
目前大多使用Redis数据库来做缓存





西部开源-秦

