Mybatis 课程教案

版本号：JAVAEE1.0

密　级：受控文档

课

程

标

准

化

2018 年 **07** 月 **X** 日

1.文档属性

文档属性	内容
项目/任务名称：	
项目/任务编号：	
文档名称：	
文档编号：	
文档状态：	
作　　者：	实训部开发组
文档评审通过日期：	
评审负责人签字：	
参考模板：	

2.文档变更过程

版本	修正日期	修正人	描述

目录

1、 ORM 的概念，ORM 到底是什么

● ORM 简介

对象关系映射（Object Relational Mapping，简称 ORM）模式是一种为了解决面向对象与关系数据库存在的互不匹配的现象的技术。简单的说，ORM 是通过使用描述对象和数据库之间映射的元数据，将程序中的对象自动持久化到关系数据库中。

ORM 采用映射元数据来描述对象关系的映射，使得 ORM 中间件能在任何一个应用的业务逻辑层和数据库层之间充当桥梁。Java 典型的 ORM 中间件有:Hibernate,ibatis,speedframework。

ORM 的方法论基于三个核心原则：

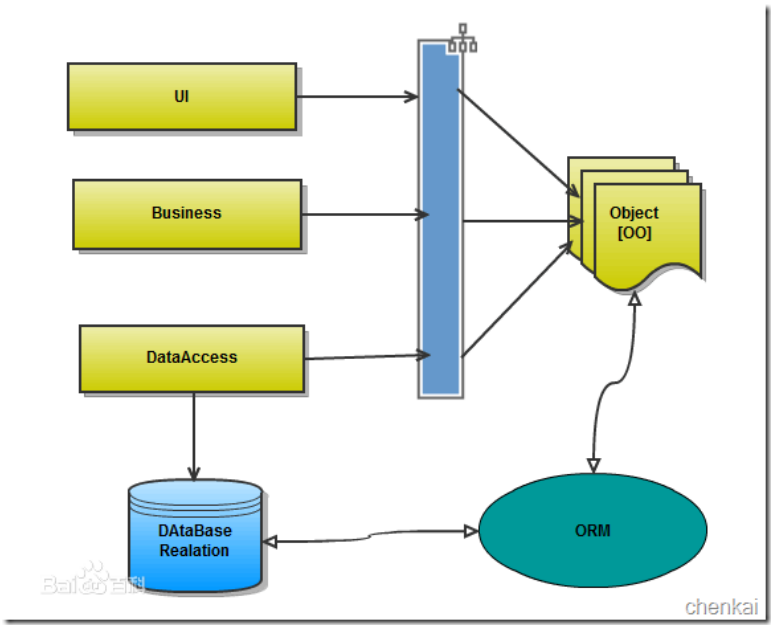
· 简单：以最基本的形式建模数据。

· 传达性：数据库结构被任何人都能理解的语言文档化。

· 精确性：基于数据模型创建正确标准化了的结构。

● ORM 的概念

ORM 解决的主要问题是对象关系的映射。域模型和关系模型分别是建立在概念模型的基础上的。域模型是面向对象的，而关系模型是面向关系的。一般情况下，一个持久化类和一个表对应，类的每个实例对应表中的一条记录，类的每个属性对应表的每个字段。



ORM 技术特点：

1.提高了开发效率。由于 ORM 可以自动对 Entity 对象与数据库中的 Table 进行字段与属性的映射，所以我们实际可能已经不需要一个专用的、庞大的数据访问层。

2.ORM 提供了对数据库的映射，不用 sql 直接编码，能够像操作对象一样从数据库获取数据。

2、 mybatis 简介

2.1. mybatis 是什么

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。

MyBatis 是一个优秀的持久层框架，它对 jdbc 的操作数据库的过程进行封装，使开发者只需要关注 SQL 本身，而不需要花费精力去处理例如注册驱动、创建 connection、创建 statement、手动设置参数、结果集检索等 jdbc 繁杂的过程代码。

Mybatis 通过 xml 或注解的方式将要执行的各种 statement（statement、preparedStatement、CallableStatement）配置起来，并通过 java 对象和 statement 中的 sql 进行映射生成最终执行的 sql 语句，最后由 mybatis 框架执行 sql 并将结果映射成 java 对象并返回。

2.2. 为什么学习 mybatis

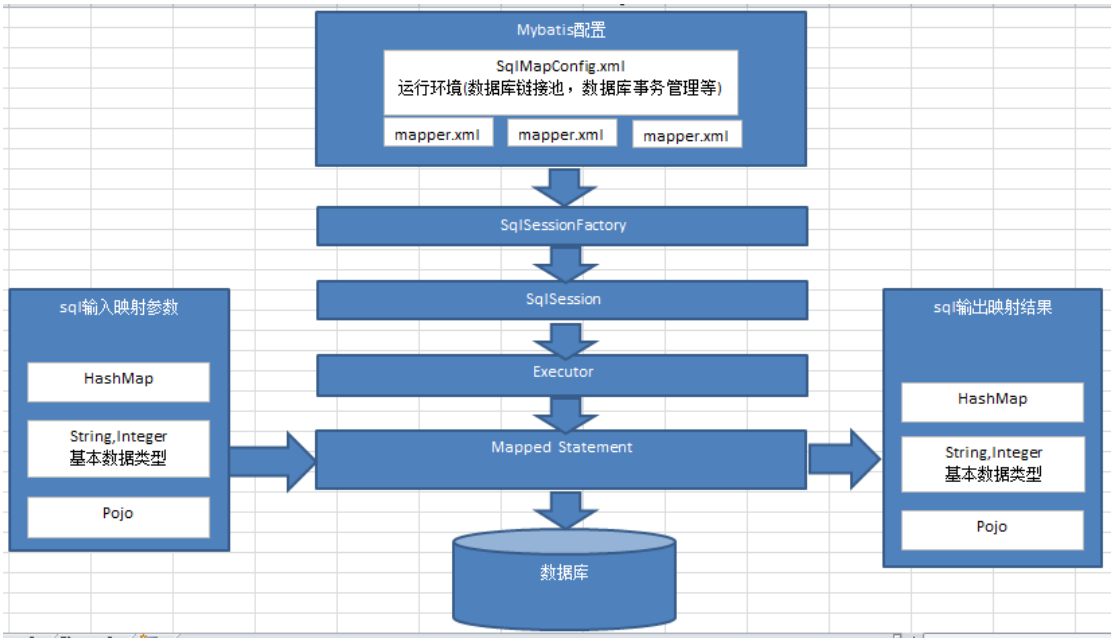
- jdbc 编程步骤:

1、加载数据库驱动
2、创建并获取数据库链接
3、创建 jdbc statement 对象
4、设置 sql 语句
5、设置 sql 语句中的参数(使用 preparedStatement)
6、通过 statement 执行 sql 并获取结果
7、对 sql 执行结果进行解析处理
8、释放资源(resultSet、preparedstatement、connection)

- jdbc 问题总结如下:

1、数据库链接创建、释放频繁造成系统资源浪费从而影响系统性能，如果使用数据库链接池可解决此问题。
2、Sql 语句在代码中硬编码，造成代码不易维护，实际应用 sql 变化的可能较大，sql 变动需要改变 java 代码。
3、使用 preparedStatement 向占有位符号传参数存在硬编码，因为 sql 语句的 where 条件不一定，可能多也可能少，修改 sql 还要修改代码，系统不易维护。
4、对结果集解析存在硬编码（查询列名），sql 变化导致解析代码变化，系统不易维护，如果能将数据库记录封装成 pojo 对象解析比较方便。

2.3. Mybatis 架构



1.mybatis 配置

SqlMapConfig.xml，此文件作为 mybatis 的全局配置文件，配置了 mybatis 的运行环境等信息。

mapper.xml 文件即 sql 映射文件，文件中配置了操作数据库的 sql 语句。此文件需要在 SqlMapConfig.xml 中加载。

2.通过 mybatis 环境等配置信息构造 SqlSessionFactory 即会话工厂

3.由会话工厂创建 sqlSession 即会话，操作数据库需要通过 sqlSession 进行。

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 也是 mybatis 一个底层封装对象，它包装了 mybatis 配置信息及 sql 映射信息等。mapper.xml 文件中一个 sql 对应一个 Mapped Statement 对象，sql 的 id 即是 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 在执行 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。

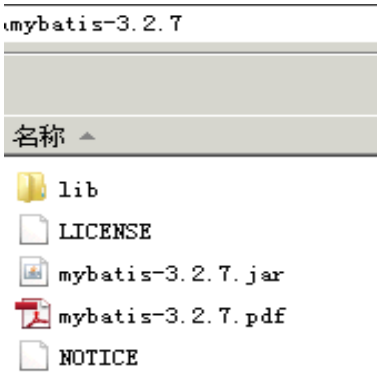
7.Mapped Statement 对 sql 执行输出结果进行定义，包括 HashMap、基本类型、pojo，Executor 通过 Mapped Statement 在执行 sql 后将输出结果映射至 java 对象中，输出结果映射过程相当于 jdbc 编程中对结果的解析处理过程。

3、 mybatis 环境搭建

3.1. mybatis 下载

mybaits 的代码由 github.com 管理

地址：https://github.com/mybatis/mybatis-3/releases



mybatis-3.2.7.jar----mybatis 的核心包

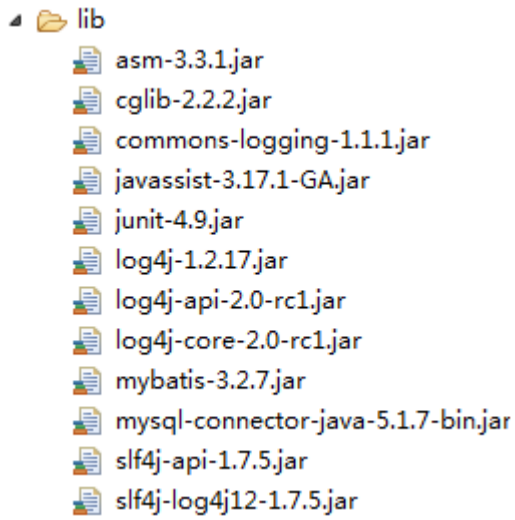
lib----mybatis 的依赖包

mybatis-3.2.7.pdf----mybatis 使用手册

3.2. 创建 java 工程

使用 eclipse 创建 java 工程

3.3. 加入 jar 包



3.4. 配置 log4j.properties

在 src 下创建 log4j.properties 如下(此步骤可以省略，主要是为打印 log 日志，可查看执行的 sql)

mybatis 默认使用 log4j 作为输出日志信息

```
# Global logging configuration
```

```
log4j.rootLogger=DEBUG, stdout
# Console output...
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n
```

3.5. 配置 SqlMapConfig.xml

在 src 下创建创建 mybatis 核心配置文件 SqlMapConfig.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
   <!-- 和spring整合后 environments配置将废除-->
   <environments default="development">
       <environment id="development">
       <!-- 使用jdbc事务管理-->
           <transactionManager type="JDBC" />
       <!-- 数据库连接池-->
           <dataSource type="POOLED">
               <property name="driver" value="com.mysql.jdbc.Driver"
/>
               <property name="url"
value="jdbc:mysql://localhost:3306/mybatis?characterEncoding=utf-8" />
               <property name="username" value="root" />
               <property name="password" value="mysql" />
           </dataSource>
       </environment>
   </environments>
</configuration>
```

4、 Mybatis 入门程序

4.1. 创建 mysql 数据库

创建数据库 mybatis

创建表 user

```
CREATE TABLE `user` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `username` varchar(32) NOT NULL COMMENT '用户名称',
  `birthday` date DEFAULT NULL COMMENT '生日',
  `sex` char(1) DEFAULT NULL COMMENT '性别',
  `address` varchar(256) DEFAULT NULL COMMENT '地址',
```

```
  PRIMARY KEY (`id`)
)
```

4.2. po 类编写

Po 类作为 mybatis 进行 sql 映射使用，po 类通常与数据库表对应，User.java 如下：

```
public class User {
   private int id;
   private String username;// 用户姓名
   private String sex;// 性别
   private Date birthday;// 生日
   private String address;// 地址
  get/set此处省略。
重写toString方法，输出每个属性。
}
```

4.3. 程序编写

4.3.1. 创建 sql 映射文件

在 src 下创建 sql 映射文件 UserMapper.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">
<mapper namespace="test">


</mapper>
```

- **namespace ：命名空间，用于隔离 sql 语句。**

4.3.2. 加载映射文件

mybatis 框架需要加载映射文件，将 UserMapper.xml 添加在 SqlMapConfig.xml，如下：

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

```
</mappers>
```

4.3．3. 根据 id 查询用户信息

在 UserMapper.xml 中添加：

```
<select id="findUserById" parameterType="Integer"
        resultType="com.hpe.po.User">
        select * from user where id = #{id}

</select>
```

parameterType：定义输入到 sql 中的映射类型，

#{id}表示使用占位符并将输入变量 id 传到 sql。

resultType：定义结果映射类型。dddd

4.3．4. 测试

```
public class UserTest {
    private SqlSessionFactory sqlSessionFactory;
    @Before
    public void createSqlSessionFactory() throws Exception {
        String resource = "SqlMapConfig.xml";
        // 通过流的方式将核心配置SqlMapConfig.xml文件读取进来
        InputStream inputStream = Resources.getResourceAsStream(resource);
        // 通过核心配置文件输入流来创建会话工厂
        // 1.创建SqlSessionFactory
        sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);
    }
    @Test
    public void testFindUserById() throws Exception {
        // 怎么执行sql?
        // 通过工厂创建会话
        // 2.创建SqlSession
        SqlSession session = sqlSessionFactory.openSession();
        // 3.使用SqlSession来执行sql语句
        // selectOne:查询一条记录
        // statement:指定statment的id-》哪条sql-》namespace.+id
        User user = session.selectOne("test.findUserById", 2);
```

```
        System.out.println(user);
        // 4.释放资源
        session.close();
    }
}
```

4.4. #{}和${}

#{}表示一个占位符号，通过#{}可以实现 preparedStatement 向占位符中设置值，自动进行 java 类型和 jdbc 类型转换，#{}可以有效防止 sql 注入。 #{}可以接收简单类型值或 pojo 属性值。 如果 parameterType 传输单个简单类型值，#{}括号中可以是 value 或其它名称。

${}表示拼接 sql 串，通过${}可以将 parameterType 传入的内容拼接在 sql 中且不进行 jdbc 类型转换， ${}可以接收简单类型值或 pojo 属性值，如果 parameterType 传输单个简单类型值，${}括号中只能是 value。

4.5. parameterType 和 resultType

parameterType：指定输入参数类型，mybatis 通过 ognl 从输入对象中获取参数值拼接在 sql 中。

resultType：指定输出结果类型，mybatis 将 sql 查询结果的一行记录数据映射为 resultType 指定类型的对象。

4.6. selectOne 和 selectList

selectOne 查询一条记录，如果使用 selectOne 查询多条记录则抛出异常：

```
org.apache.ibatis.exceptions.TooManyResultsException: Expected one result (or null) to be returned by selectOne(), but found: 3
        at org.apache.ibatis.session.defaults.DefaultSqlSession.selectOne(DefaultSqlSession.java:70)
```

selectList 可以查询一条或多条记录。

4.7. 模糊查询（查询名字中带三的用户）

在 UserMapper.xml 中添加：

```
    <!-- 根据名称查询用户（模糊查询） -->
    <!-- string就是 java.lang.String的别名 -->
```

```
    <!-- ${}表示拼接sql串，通过${}可以将parameterType 传入的内容拼接在sql -->
    <select id="findUserByName" parameterType="string"
resultType="com.hpe.po.User">
        select * from user
        where username like '%${value}%'

    </select>
```

测试代码：

```
    @Test
    public void testFindUserByName() throws Exception {
        SqlSession session = sqlSessionFactory.openSession();
        //selectList:查询多条记录
        List<User> users = session.selectList("test.findUserByName","三");
        System.out.println(users);
        session.close();

    }
```

4.8. 添加（返回自增主键）

在 UserMapper.xml 中添加：

```
<!-- 得到插入记录的主键 keyProperty  -->
    <insert id="addUser" parameterType="com.hpe.po.User" keyProperty="id"
useGeneratedKeys="true">
        insert into user(username,birthday,sex,address)
        values(#{username},#{birthday},#{sex},#{address})

    </insert>
```

测试代码：

```
@Test
    public void testAddUser() throws Exception {
        SqlSession session = sqlSessionFactory.openSession();
        User user = new User(0, "王超", "男", new Date(), "山东日照");
        session.insert("test.addUser", user);
        session.commit();
        System.out.println(user.getId());
        session.close();

    }
```

4.9. 删除

在 UserMapper.xml 中添加：

```
<!-- 根据id删除用户 -->
    <delete id="deleteUserById" parameterType="int">
        delete from user
        where id = #{id}

    </delete>
```

测试代码：

```
@Test
    public void testDeleteUserById() throws Exception {
        SqlSession session = sqlSessionFactory.openSession();
        int delete = session.delete("test.deleteUserById", 8);
        session.commit();
        System.out.println(delete);
        session.close();

    }
```

4.10. 修改

在 UserMapper.xml 中添加：

```
<update id="updateUser" parameterType="com.hpe.po.User">
        update user set
username=#{username},birthday=#{birthday},sex=#{sex},address=#{address}
        where id=#{id}

    </update>
```

测试代码：

```
@Test
    public void testUpdateUser() throws Exception {
        SqlSession session = sqlSessionFactory.openSession();
        User user = new User(3, "立国", "男", new Date(), "山东威海");
        session.update("test.updateUser", user);
        session.commit();
        session.close();

    }
```

5、 Mybatis 解决 jdbc 编程的问题

1、 数据库链接创建、释放频繁造成系统资源浪费从而影响系统性能，如果使用数据库链接池可解决此问题。

解决：在 SqlMapConfig.xml 中配置数据链接池，使用连接池管理数据库链接。

2、 Sql 语句写在代码中造成代码不易维护，实际应用 sql 变化的可能较大，sql 变动需要改变 java 代码。

解决：将 Sql 语句配置在 XXXXmapper.xml 文件中与 java 代码分离。

3、 向 sql 语句传参数麻烦，因为 sql 语句的 where 条件不一定，可能多也可能少，占位符需要和参数一一对应。

解决：Mybatis 自动将 java 对象映射至 sql 语句，通过 statement 中的 parameterType 定义输入参数的类型。

4、 对结果集解析麻烦，sql 变化导致解析代码变化，且解析前需要遍历，如果能将数据库记录封装成 pojo 对象解析比较方便。

解决：Mybatis 自动将 sql 执行结果映射至 java 对象，通过 statement 中的 resultType 定义输出结果的类型。

6、 与 hibernate 不同

Mybatis 和 hibernate 不同，它不完全是一个 ORM 框架，因为 MyBatis 需要程序员自己编写 Sql 语句，不过 mybatis 可以通过 XML 或注解方式灵活配置要运行的 sql 语句，并将 java 对象和 sql 语句映射生成最终执行的 sql，最后将 sql 执行的结果再映射生成 java 对象。

Mybatis 学习门槛低，简单易学，程序员直接编写原生态 sql，可严格控制 sql 执行性能，灵活度高，非常适合对关系数据模型要求不高的软件开发，例如互联网软件、企业运营类软件等，因为这类软件需求变化频繁，一但需求变化要求成果输出迅速。但是灵活的前提是 mybatis 无法做到数据库无关性，如果需要实现支持多种数据库的软件则需要自定义多套 sql 映射文件，工作量大。

Hibernate 对象/关系映射能力强，数据库无关性好，对于关系模型要求高的软件（例如需求固定的定制化软件）如果用 hibernate 开发可以节省很多代码，提高效率。但是 Hibernate 的学习门槛高，要精通门槛更高，而且怎么设计 O/R 映射，在性能和对象模型之间如何权衡，以及怎样用好 Hibernate 需要具有很强的经验和能力才行。

7、 Mybatis 核心对象

两大核心对象：SqlSessionFactory 和 SqlSession

SqlSessionFactoryBuilder 用于创建 SqlSessionFacoty，SqlSessionFacoty 一旦创建完成就不需要 SqlSessionFactoryBuilder 了，因为 SqlSession 是通过 SqlSessionFactory 生产，所以可以将 SqlSessionFactoryBuilder 当成一个工具类使用，最佳使用范围是方法范围即方法体内局部变量。

7.1 SqlSessionFactory

SqlSessionFactory 是一个接口，由 SqlSessionFactoryBuilder 对象创建，接口中定义了 openSession 的不同重载方法。

一旦创建 SqlSessionFactory 类的实例，该实例在应用程序执行期间都存在，根本不需要每一次操作数据库时都重新创建它，所以应用它的最佳方式就是写一个**单例模式**，或使用 Spirng 框架来实现单例模式对 SqlSessionFactory 对象进行的管理。

实现代码如下：

```
// 通过流的方式将核心配置SqlMapConfig.xml文件读取进来
InputStream inputStream = Resources.getResourceAsStream("配置文件文字");
// 通过输入流来创建SqlSessionFactory

sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
```

7.2 SqlSession

SqlSession 是一个面向用户的接口， sqlSession 中定义了数据库操作，如：查询、插入、更新、删除等。默认使用 DefaultSqlSession 实现类。

SqlSession 对象由 SqlSessionFactory 类创建，需要注意，每个线程都应该有自己的 SqlSession 实例。SqlSession 的实例不能共享，它是线程不安全的，所以不能再 Servlet 中生命该对象的一个实例变量。因为 Servlet 是单例的，申明成实例会造成线程安全问题，也绝不能将 SqlSession 实例的对象放在一个类的静态字段甚至是实例字段中。还不可以将 Sqlsession 实例的对象放在任何类型的管理范围中，比如 Servlet 对象中的 Httpsession 会话。在接收到 HTTP 请求时，可以打开一个 SqlSession 对象操作数据库，然后返回响应，就可以关闭它。关闭 SqlSession 很重要，应该确保使用 finally 块来关闭它。

执行过程如下：

1.加载数据源等配置信息

Environment environment = configuration.getEnvironment();

2.创建数据库链接

3.创建事务对象

4.创建 Executor，SqlSession 所有操作都是通过 Executor 完成，mybatis 源码如下：

```
    if (ExecutorType.BATCH == executorType) {
          executor = newBatchExecutor(this, transaction);
        } elseif (ExecutorType.REUSE == executorType) {
          executor = new ReuseExecutor(this, transaction);
        } else {
          executor = new SimpleExecutor(this, transaction);
        }
    if (cacheEnabled) {
          executor = new CachingExecutor(executor,
autoCommit);
        }
```

5.SqlSession 的实现类即 DefaultSqlSession，此对象中对操作数据库实质上用的是 Executor

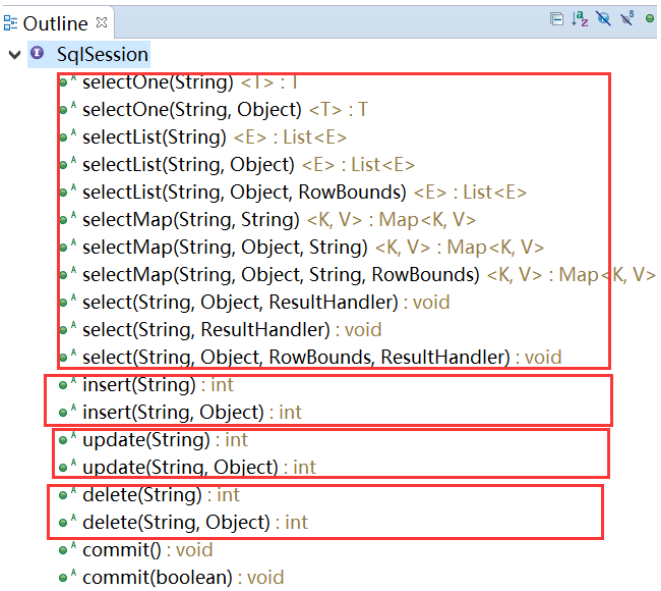
结论：

每个线程都应该有它自己的 SqlSession 实例。SqlSession 的实例不能共享使用，它也是线程不安全的。因此最佳的范围是请求或方法范围。绝对不能将 SqlSession 实例的引用放在一个类的静态字段或实例字段中。

打开一个 SqlSession；使用完毕就要关闭它。通常把这个关闭操作放到 finally 块中以确保每次都能执行关闭。如下：

```
SqlSession session = sqlSessionFactory.openSession();
try {
      // do work
} finally {
     session.close();
}
```

SqlSession 封装的方法如下：



8、 SqlMapConfig.xml 配置文件

8.1 配置内容

SqlMapConfig.xml 中配置的内容和顺序如下：

- properties（属性）
- settings（全局配置参数）
- typeAliases（类型别名）
- typeHandlers（类型处理器）
- objectFactory（对象工厂）
- plugins（插件）
- environments（环境集合属性对象）
- environment（环境子属性对象）
  - transactionManager（事务管理）
  - dataSource（数据源）
- mappers（映射器）

8.2 properties（属性）

SqlMapConfig.xml 可以引用 java 属性文件中的配置信息如下：

<properties>是一个配置属性的元素，该元素用于将内部的配置外在化，即通过外部的配置来动态的代替内部定义的属性。例如数据库的连接等属性。

1.在 classpath 下定义 db.properties 文件，

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/mybatis?characterEncoding=utf-8
jdbc.username=root
jdbc.password=123456
```

2.SqlMapConfig.xml 引用<properties>如下：

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

3.修改数据库链接信息

```
<dataSource type="POOLED">
        <property name="driver" value="${jdbc.driver}" />
        <property name="url" value="${jdbc.url}" />
        <property name="username" value="${jdbc.username}" />
        <property name="password" value="${jdbc.password}" />
</dataSource>
```

8.3 **settings（配置）

mybatis 全局配置参数，全局参数将会影响 mybatis 的运行行为。

Setting(设置)	Description（描述）	Valid Values(验证值组)	Default(默认值)
cacheEnabled	在全局范围内启用或禁用缓存配置任何映射器在此配置下。	true | false	TRUE
lazyLoadingEnabled	在全局范围内启用或禁用延迟加载。禁用时，所有协会将热加载。	true | false	TRUE
aggressiveLazyLoading	启用时，有延迟加载属性的对象将被完全加载后调用懒惰的任何属性。否则，每一个属性是按需加载。	true | false	TRUE
multipleResultSetsEnabled	允许或不允许从一个单独的语句（需要兼容的驱动程序）要返回多个结果集。	true | false	TRUE
useColumnLabel	使用列标签，而不是列名。在这方面，不同的驱动有不同的行为。参考驱动文档或测试两种方法来决定你的驱动程序的行为如何。	true | false	TRUE
useGeneratedKeys	允许JDBC支持生成的密钥。兼容的驱动程序是必需的。此设置强制生成的键被使用，如果设置为true，一些驱动会不兼容性，但仍然可以工作。	true | false	FALSE

autoMappingBehavior	指定MyBatis的应如何自动映射列到字段/属性。NONE自动映射。 PARTIAL只会自动映射结果没有嵌套结果映射定义里面。 FULL会自动映射的结果映射任何复杂的（包含嵌套或其他）。	NONE, PARTIAL, FULL	PARTIAL
defaultExecutorType	配置默认执行人。SIMPLE执行人确实没有什么特别的。 REUSE执行器重用准备好的语句。 BATCH执行器重用语句和批处理更新。	SIMPLE REUSE BATCH	SIMPLE
defaultStatementTimeout	设置驱动程序等待一个数据库响应的秒数。	Any positive integer	Not Set (null)
safeRowBoundsEnabled	允许使用嵌套的语句RowBounds。	true | false	FALSE
mapUnderscoreToCamelCase	从经典的数据库列名A_COLUMN启用自动映射到骆驼标识的经典的Java属性名aColumn。	true | false	FALSE
localCacheScope	MyBatis的使用本地缓存，以防止循环引用，并加快反复嵌套查询。默认情况下（SESSION）会话期间执行的所有查询缓存。如果localCacheScope=STATMENT本地会话将被用于语句的执行，只是没有将数据共享之间的两个不同的调用相同的SqlSession。	SESSION | STATEMENT	SESSION
dbcTypeForNull	指定为空值时，没有特定的JDBC类型的参数的JDBC类型。有些驱动需要指定列的JDBC类型，但其他像NULL，VARCHAR或OTHER的工作与通用值。	JdbcType enumeration. Most common are: NULL, VARCHAR and OTHER	OTHER

lazyLoadTriggerMethods	指定触发延迟加载的对象的方法。	A method name list separated by commas	equals,clone,hash Code,toString
defaultScriptingLanguage	指定所使用的语言默认为动态SQL生成。	A type alias or fully qualified class name.	org.apache.ibatis.scripting.xmltags.XMLDynamicLanguageDriver
callSettersOnNulls	指定如果setter方法或地图的put方法时，将调用检索到的值是null。它是有用的，当你依靠Map.keySet（）或null初始化。注意原语（如整型，布尔等）不会被设置为null。	true | false	FALSE
logPrefix	指定的前缀字串，MyBatis将会增加记录器的名称。	Any String	Not set
logImpl	指定MyBatis的日志实现使用。如果此设置是不存在的记录的实施将自动查找。	SLF4J | LOG4J | LOG4J2 | JDK_LOGGING | COMMONS_LOGGING | STDOUT_LOGGING |	Not set
proxyFactory	指定代理工具，MyBatis将会使用创建懒加载能力的对象。	CGLIB | JAVASSIST	

**以上介绍的配置内容大多数不需要开发人员去配置它，了解即可。

8.4 typeAliases（类型别名）

<typeAliases>元素用于为配置文件中的 java 类型配置一个简短的名字，即设置别名。其使用的意义在于减少全限定类名的冗余。

在 SqlMapConfig.xml 中配置：

```
<typeAliases>
    <!-- 别名定义 -->
    <typeAlias alias="user" type="com.hpe.po.User"/>
</typeAliases>
```

- type：指定需要被定义别名的类的全限定名。
- alias：自定义的名字。

上述配置后，可在工程的 UserMapper 中的把 resultType 或者 parameterType 中使用 user 别名来代替原有的 com.hpe.po.User。

如果 POJO 类过多，还可以通过自动扫描包的形式自定义别名，示例如下：

```
<typeAliases>
    <!-- 批量别名定义，扫描整个包下的类，别名为类名（首字母大写或小写都可以）-->
    <package name="com.hpe.po"/>
</typeAliases>
```

- mybatis 支持别名：

别名	映射的类型
_byte	byte
_long	long
_short	short
_int	int
_integer	int
_double	double
_float	float
_boolean	boolean
string	String
byte	Byte
long	Long
short	Short
int	Integer
integer	Integer
double	Double
float	Float

boolean	Boolean
date	Date
decimal	BigDecimal
bigdecimal	BigDecimal

8.5 **typeHandlers（类型处理器）

typeHandlers 的作用就是将预处理语句中传入的参数从 javaTpye(java 类型)转换成 jdbcType(jdbc 类型)，或者从数据库去除结果时将 jdbcType 转换成 javaTpye。

类型处理器用于 java 类型和 jdbc 类型映射，如下：

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

mybatis 自带的类型处理器基本上满足日常需求，不需要单独定义。

mybatis 支持类型处理器：

类型处理器	**Java**类型	**JDBC**类型
BooleanTypeHandler	Boolean，boolean	任何兼容的布尔值
ByteTypeHandler	Byte，byte	任何兼容的数字或字节类型
ShortTypeHandler	Short，short	任何兼容的数字或短整型
IntegerTypeHandler	Integer，int	任何兼容的数字和整型
LongTypeHandler	Long，long	任何兼容的数字或长整型
FloatTypeHandler	Float，float	任何兼容的数字或单精度浮点型

DoubleTypeHandler	Double，double	任何兼容的数字或双精度浮点型
BigDecimalTypeHandler	BigDecimal	任何兼容的数字或十进制小数类型
StringTypeHandler	String	CHAR和VARCHAR类型
ClobTypeHandler	String	CLOB和LONGVARCHAR类型
NStringTypeHandler	String	NVARCHAR和NCHAR类型
NClobTypeHandler	String	NCLOB类型
ByteArrayTypeHandler	byte[]	任何兼容的字节流类型
BlobTypeHandler	byte[]	BLOB和LONGVARBINARY类型
DateTypeHandler	Date（java.util）	TIMESTAMP类型
DateOnlyTypeHandler	Date（java.util）	DATE类型
TimeOnlyTypeHandler	Date（java.util）	TIME类型
SqlTimestampTypeHandler	Timestamp（java.sql）	TIMESTAMP类型
SqlDateTypeHandler	Date（java.sql）	DATE类型
SqlTimeTypeHandler	Time（java.sql）	TIME类型
ObjectTypeHandler	任意	其他或未指定类型
EnumTypeHandler	Enumeration类型	VARCHAR-任何兼容的字符串类型，作为代码存储（而不是索引）。

**以上介绍的配置内容大多数不需要开发人员去配置它，了解即可。

8.6 environments 环境配置

```
<!--默认的环境为 id=mysql的 -->
<environments default="mysql">
    <!-- 配置一个id 为mysql的数据库环境 -->
    <environment id="mysql">
        <!-- 使用jdbc事务管理 -->
        <transactionManager type="JDBC" />
        <!-- 数据库连接池 -->
        <dataSource type="POOLED">
            <property name="driver" value="${jdbc.driver}" />
            <property name="url" value="${jdbc.url}" />
            <property name="username" value="${jdbc.username}" />
            <property name="password" value="${jdbc.password}" />
        </dataSource>
    </environment>

</environments>
```

8.7 mappers（映射器）

<mappers>元素用于指定 Mybatis 映射文件的位置，配置的几种方法：

- <mapper resource=" " />

  使用相对于类路径的资源

  如：<mapper resource="com/hpe/mapper/UserMapper.xml" />

- <mapper url=" " />

  使用完全限定路径

  如：<mapper url="file:/// D:\Program Files\workSpace\MyBatisWorkSpace\mybatis_02_02\src\com\hpe\mapper\UserMapper.xml" />

- <mapper class=" " />

  使用 mapper 接口类路径

  如：<mapper class="cn.hpe.mapper.UserMapper"/>

注意：此种方法要求 mapper 接口名称和 mapper 映射文件名称相同，且放在同一个目录中。

- <package name=""/>

  注册指定包下的所有 mapper 接口

  如：<package name="cn.hpe.mapper"/>

  注意：此种方法要求 mapper 接口名称和 mapper 映射文件名称相同，且放在同一个目录中。

9、 原始 Dao 开发方式

原始 Dao 开发方法需要程序员编写 Dao 接口和 Dao 实现类。

9.1. 映射文件

原 User.mapper.xml

9.2. Dao 接口

```
public interface IUserDao {
 //根据id查询用户信息
 User findUserById(int id);
 //根据名称查询用户（模糊查询）
 List<User> findUserByName(String username);
 //根据id删除用户
 int deleteUserById(int id);
 //根据id修改用户
 int updateUser(User user);
 //添加用户返回主键
 int addUser(User user);
}
```

9.3. 实现类

```
public class UserDaoImpl implements IUserDao {
 private SqlSessionFactory factory;
```

```
public UserDaoImpl(SqlSessionFactory factory) {
    this.factory = factory;
}
@Override
public User findUserById(int id) {
    SqlSession session = factory.openSession();
    User user = null;
    try {
        user = session.selectOne("test.findUserById", id);
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        session.close();
    }
    return user;
}
@Override
public List<User> findUserByName(String username) {
    SqlSession session = factory.openSession();
    List<User> list = null;
    try {
        list = session.selectList("test.findUserByName", username);
    } catch (Exception e) {
        e.printStackTrace();
    }
    finally {
        session.close();
    }
    return list;
}
```

```
    @Override
    public int deleteUserById(int id) {
        SqlSession session = factory.openSession();
        int row=0;
        try {
            row = session.delete("test.deleteUserById", id);
            session.commit();
        } catch (Exception e) {
            e.printStackTrace();
        }
        finally {
            session.close();
        }
        return row;
    }
    @Override
    public int updateUser(User user) {
        SqlSession session = factory.openSession();
        int row=0;
        try {
            row = session.update("test.updateUser", user);
            session.commit();
        } catch (Exception e) {
            e.printStackTrace();
        }finally {
            session.close();
        }
        return row;
    }
    @Override
```

```
    public int addUser(User user) {
        SqlSession session = factory.openSession();
        try {
            session.insert("test.addUser", user);
            session.commit();
        } catch (Exception e) {
            e.printStackTrace();
        }finally {
            session.close();
        }
        return user.getId();
    }
}
```

9.4. 测试类

```
public class UserDaoImplTest {
    private SqlSessionFactory factory;
    private IUserDao userDao;
    @Before
    public void setUp() throws Exception {
        factory = new
SqlSessionFactoryBuilder().build(Resources.getResourceAsStream("SqlMapConfig.xml"))
;
        userDao = new UserDaoImpl(factory);
    }
    @Test
    public void testFindUserById() {
        User user = userDao.findUserById(1);
        System.out.println(user);
    }
    @Test
```

```
    public void testFindUserByName() {
        List<User> list = userDao.findUserByName("三");
        System.out.println(list);
    }
    @Test
    public void testDeleteUserById() {
        int result = userDao.deleteUserById(5);
        System.out.println(result);
    }
    @Test
    public void testUpdateUser() {
        int result = userDao.updateUser(new User(1, "张三丰", "中", new Date(), "北京"));
        System.out.println(result);
    }
    @Test
    public void testAddUser() {
        int result = userDao.addUser(new User(0, "明明", "女", new Date(), "昌平"));
        System.out.println(result);
    }
   }
```

9.5. 问题

原始 Dao 开发中存在以下问题：

- Dao 方法体存在重复代码：通过 SqlSessionFactory 创建 SqlSession，调用 SqlSession 的数据库操作方法
- 调用 `sqlSession` 的数据库操作方法需要指定 `statement` 的 `id`，这里存在硬编码，不得于开发维护。

10、 Mapper 动态代理方式

10.1. 实现原理

Mapper 接口开发方法只需要程序员编写 Mapper 接口（相当于 Dao 接口），由 Mybatis 框架根据接口定义创建接口的动态代理对象，代理对象的方法体同上边 Dao 接口实现类方法。

Mapper 接口开发需要遵循以下规范：

1、 Mapper.xml 文件中的 namespace 与 mapper 接口的类路径相同。
2、 Mapper 接口方法名和 Mapper.xml 中定义的每个 statement 的 id 相同
3、 Mapper 接口方法的输入参数类型和 mapper.xml 中定义的每个 sql 的 parameterType 的类型相同
4、 Mapper 接口方法的输出参数类型和 mapper.xml 中定义的每个 sql 的 resultType 的类型相同

10.2. Mapper.xml（映射文件）

定义 mapper 映射文件 UserMapper.xml（内容同 Users.xml），需要修改 namespace 的值为 UserMapper 接口路径。将 UserMapper.xml 放在 classpath 下 com.hpe.mapper 目录 下。

```
<?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">

<!--定义mapper映射文件UserMapper.xml，需要修改namespace的值为 UserMapper接口路径 -
->

<mapper namespace="com.hpe.mapper.UserMapper">

******

</mapper>
```

10.2. Mapper.java（接口文件）

在 UserMapper.xml 的同目录下，新建接口 UserMapper.java

```
public interface UserMapper {

 // 根据id查询用户信息

 User findUserById(int id);
```

```
    // 根据名称查询用户（模糊查询）
    List<User> findUserByName(String username);
    // 根据id删除用户
    int deleteUserById(int id);
    // 根据id修改用户
    boolean updateUser(User user);
    // 添加用户返回主键
    int addUser(User user);
}
```

接口定义有如下特点：

1、 Mapper 接口方法名和 Mapper.xml 中定义的 statement 的 id 相同
2、 Mapper 接口方法的输入参数类型和 mapper.xml 中定义的 statement 的 parameterType 的类型相同
3、 Mapper 接口方法的输出参数类型和 mapper.xml 中定义的 statement 的 resultType 的类型相同

10.3. 加载 UserMapper.xml 文件

修改 SqlMapConfig.xml 文件：

```
<!-- 加载映射文件 -->
<mappers>
    <mapper resource="com/hpe/mapper/UserMapper.xml" />
</mappers>
```

- 加载 **UserMapper.xml** 文件

<!-- 加载映射文件 -->

<mappers>

<!-- 加载映射文件，使用相对于类路径的资源**-->**

<!-- <mapper resource="com/hpe/mapper/UserMapper.xml"/> -->

<!--使用 mapper 接口类路径**-->**

注意：此种方法要求 **mapper** 接口名称和 **mapper** 映射文件名称相同，且放在同一个目录中。

<!-- <mapper class="com.hpe.mapper.UserMapper"/> -->

<!-- 加载包内的文件（**mapper** 多了注册起来会很麻烦，直接加载包提高编码效率）**-->**

注意：此种方法要求 **mapper** 接口名称和 **mapper** 映射文件名称相同，且放在同一个目录中。

<package name=*"com.hpe.mapper"* />

</mappers>

10. 4. 测试

```
public class UserMapperTest {

    private SqlSessionFactory factory;

    @Before

    public void setUp() throws Exception {

        factory = new
SqlSessionFactoryBuilder().build(Resources.getResourceAsStream("SqlMapConfig.xml"))
;

    }

    @Test

    public void testFindUserById() {

        // 获取session

        SqlSession session = factory.openSession();

        // 获取mapper接口的代理对象

        UserMapper mapper = session.getMapper(UserMapper.class);

        // 调用代理对象方法

        User user = mapper.findUserById(1);

        // 关闭session

        session.close();
```

```
        System.out.println(user);
    }
    @Test
    public void testFindUserByName() {
        // 获取session
        SqlSession session = factory.openSession();
        // 获取mapper接口的代理对象
        UserMapper mapper = session.getMapper(UserMapper.class);
        // 调用代理对象方法
        List<User> list = mapper.findUserByName("三");
        // 关闭session
        session.close();
        System.out.println(list);
    }
    @Test
    public void testDeleteUserById() {
        // 获取session
        SqlSession session = factory.openSession();
        // 获取mapper接口的代理对象
        UserMapper mapper = session.getMapper(UserMapper.class);
        // 调用代理对象方法
        int row = mapper.deleteUserById(8);
        session.commit();
        // 关闭session
        session.close();
        System.out.println(row);
    }
    @Test
    public void testUpdateUser() {
        // 获取session
```

```
        SqlSession session = factory.openSession();
        // 获取mapper接口的代理对象
        UserMapper mapper = session.getMapper(UserMapper.class);
        // 调用代理对象方法
        boolean result = mapper.updateUser(new User(7, "小勇", "女", new Date(), "上
海"));
        session.commit();
        // 关闭session
        session.close();
        System.out.println(result);
    }
    @Test
    public void testAddUser() {
        // 获取session
        SqlSession session = factory.openSession();
        // 获取mapper接口的代理对象
        UserMapper mapper = session.getMapper(UserMapper.class);
        User user = new User(2, "小辉", "男", new Date(), "山东烟台");
        // 调用代理对象方法
        int result = mapper.addUser(user);
        session.commit();
        // 关闭session
        session.close();
        System.out.println(result);
        System.out.println(user.getId());
    }
   }
```

10.5. 总结

◆ selectOne 和 selectList

动态代理对象调用 sqlSession.selectOne()和 sqlSession.selectList()是根据 mapper 接口方法的返回值决定，如果返回 list 则调用 selectList 方法，如果返回单个对象则调用 selectOne 方法。

◆ namespace

mybatis 官方推荐使用 mapper 代理方法开发 mapper 接口，程序员不用编写 mapper 接口实现类，使用 mapper 代理方法时，输入参数可以使用 pojo 包装对象或 map 对象，保证 dao 的通用性。

11、 Mapper.xml 详解

Mapper.xml 映射文件中定义了操作数据库的 sql，每个 sql 是一个 statement，映射文件是 mybatis 的核心。

11.1. parameterType(输入类型)

- **传递简单类型**

  int,string,long 等

- **传递 pojo 对象**

  Mybatis 使用 ognl 表达式解析对象字段的值，如下例子：

```
    <!—传递pojo对象综合查询用户信息 -->
        <select id="findUserByUser" parameterType="user"
resultType="user">
          select * from user where id=#{id} and username like
'%${username}%'
        </select>
```

- **传递 map**

接口定义一个传递 map 参数的查询方法。

```
// 根据姓名和性别查询
List<User> findUserByMap(Map<String, Object> map);
```

Sql 映射文件定义如下：

```
<!-- 传递map综合查询用户信息 -->
    <select id="findUserByMap" parameterType="map" resultType="user">
       select * from user where id=#{id} and username like '%${username}%'
    </select>
```

*上边红色标注的是 hashmap 的 key。

测试：

```
private SqlSession session;
    @Before
    public void init() throws Exception {
        SqlSessionFactory factory = new SqlSessionFactoryBuilder()
                .build(Resources.getResourceAsStream("SqlMapConfig.xml"));
        session = factory.openSession();
    }
    @Test
    public void testmap() {
        UserMapper userMapper = session.getMapper(UserMapper.class);
        // 构造查询条件Hashmap对象
        Map<String, Object> map = new HashMap<String, Object>();
        map.put("id", 1);
        map.put("username", "张三");
        // 传递Map对象查询用户列表
        List<User> list = userMapper.findUserByMap(map);
        // 关闭session
        session.close();
    }
```

异常测试：

传递的 map 中的 key 和 sql 中解析的 key 不一致。

测试结果没有报错，只是通过 key 获取值为空。

- **传递多参数**

接口定义一个传递多个参数的查询方法。

```
//多个参数

List<User> findUserByParams(String name,String sex);
```

Sql 映射文件定义如下：

```
<!-- 传递多个参数查询用户信息 -->
    <select id="findUserByParams" resultType="user">
       select * from user where  username = #{0} and sex = #{1}

    </select>
```

*多参数那么就不使用 parameterType， 改用#｛index｝是第几个就用第几个的索引，索引从 0 开始

测试：

```
@Test
    public void testparams() {
        UserMapper userMapper = session.getMapper(UserMapper.class);
        // 传递多参数查询用户列表
        List<User> list = userMapper.findUserByParams("王超", "男");
        System.err.println(list.size());
        // 关闭session
        session.close();

    }
```

还有一种方式是使用基于注解来传递多参数。

接口的多参数查询方法

```
// 多个参数

List<User> findUserByParams2(@Param("name")String name, @Param("sex")String sex);
```

Sql 映射文件定义如下：

```
<!-- 传递多个参数查询用户信息 ,基于注解-->
    <select id="findUserByParams2" resultType="user">
       select * from user where  username = #{name} and sex = #{sex}

    </select>
```

测试：

```
@Test
```

```
    public void testparams2() {
        UserMapper userMapper = session.getMapper(UserMapper.class);
        // 传递多参数查询用户列表
        List<User> list = userMapper.findUserByParams2("王超", "男");
        System.err.println(list.size());
        // 关闭session
        session.close();

    }
```

**其中方法名和 ID 一致，#{}中的参数名与方法中的参数名一致， 这里采用的是@Param 这个参数，实际上@Param 这个最后会被 Mabatis 封装为 map 类型的。

11.2. resultType(输出类型)

- **输出简单类型**

接口的多参数查询方法

```
// 简单输出类型

int findUserCount();
```

Sql 映射文件定义如下：

```
<!-- 获取用户列表总数 -->
    <select id="findUserCount" resultType="int">
       select count(*) from user

    </select>
```

测试：

```
@Test
    public void testcount() {
        // 获取mapper接口的代理对象
        UserMapper userMapper = session.getMapper(UserMapper.class);
        // 调用代理对象方法
        int result = userMapper.findUserCount();
        // 关闭session
        session.close();
        System.out.println(result);

    }
```

**输出简单类型必须查询出来的结果集有一条记录，最终将第一个字段的值转换为输

出类型。

- **输出 pojo 对象**
- **输出 pojo 列表**
- **resultType 总结**

输出 pojo 对象和输出 pojo 列表在 sql 中定义的 resultType 是一样的。

返回单个 pojo 对象要保证 sql 查询出来的结果集为单条，内部使用 session.selectOne 方法调用，mapper 接口使用 pojo 对象作为方法返回值。

返回 pojo 列表表示查询出来的结果集可能为多条，内部使用 session.selectList 方法，mapper 接口使用 List<pojo>对象作为方法返回值。

11.3. resultMap

resultType 可以指定 pojo 将查询结果映射为 pojo，但需要 pojo 的属性名和 sql 查询的列名一致方可映射成功。

如果 sql 查询字段名和 pojo 的属性名不一致，可以通过 resultMap 将字段名和属性名作一个对应关系 ，resultMap 实质上还需要将查询结果映射到 pojo 对象中。

resultMap 可以实现将查询结果映射为复杂类型的 pojo，比如在查询结果映射对象中包括 pojo 和 list 实现一对一查询和一对多查询。

- **案例一：解决数据库表字段名和 pojo 的属性名称不一致的问题**

  第一步：新建 car 表，sql 如下：

```
CREATE TABLE `car` (
  `carId` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(32) DEFAULT NULL,
  `userId` int(11) DEFAULT NULL,
  PRIMARY KEY (`carId`)
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8;
```

  第二步：新建实体类 com.hpe.po.Car

```
public class Car {
    private int carId;//汽车id
    private int userId;//属于哪个人
```

```
    private String carName;//汽车名称

    get*** set***
```

注意：汽车名称表中的字段为 name，实体类中的名称为 carName，二者不同.

第三步：查询所有汽车--接口方法

```
//查询所有汽车

List<Car> selectCars();
```

第四步：sql 映射文件：

```
<select id="selectCars" resultType="com.hpe.po.Car">
        select * from car

    </select>
```

第五步：测试

```
@Test
    public void testRelutMap() {
        // 获取mapper接口的代理对象
        UserMapper userMapper = session.getMapper(UserMapper.class);
        // 调用代理对象方法
        List<Car> result = userMapper.selectCars();
        // 关闭session
        session.close();
        System.out.println(result);

    }
```

结果输出为[Car [carId=1, userId=1, carName=null]]，carName 没有查询出值。

解决方法：

第六步：修改 sql 映射文件，新增 reslutMap

```
<resultMap type="com.hpe.po.Car" id="CarResultMap">
        <id column="carId" property="carId" />
        <result column="name" property="carName" />
        <result column="userId" property="userId" />

</resultMap>

***

<select id="selectCars" resultMap="CarResultMap">
        select * from car

</select>
```

完美解决。

resultMap 元素详解：

1. <id />：此属性表示查询结果集的唯一标识，非常重要。如果是多个字段为复合唯一约束则定义多个<id />。
2. Property：表示对应实体类的属性。
3. Column：表示 sql 查询出来的字段名。即数据库表的字段名
4. Column 和 property 放在一块儿表示将 sql 查询出来的字段映射到指定的 pojo 类属性上。
5. <result />：普通结果

- **案例二：解决映射为复杂类型的 pojo 的问题**

现在需要做这么一个功能，查询所有用户，包括把用户拥有的汽车信息也查询出来？

Sql 语句：

SELECT * from `user` u LEFT JOIN car c ON u.id = c.userId

面向对象的思想，需要把查询结果封装为一个对象，则需要新建实体类，步骤如下：

第一步：新建实体类 com.hpe.vo.UserVo

```
public class UserVo extends User{

    private List<Car> cars;

   get***  set***
```

因为一个人拥有 0~N 个车，故一对多的关系

第二步：新建查询接口方法：

```
//查询某人有某车

List<UserVo> selectCarsByUserId();
```

第三步：sql 映射文件：

```
<!-- 查询某人有某车 -->
    <select id="selectCarsByUserId" resultType="com.hpe.vo.UserVo">
        SELECT * from `user` u LEFT JOIN car c ON u.id = c.userId

    </select>
```

第四步：测试

```
@Test
    public void testRelutMap2() {
```

```
        // 获取mapper接口的代理对象
        UserMapper userMapper = session.getMapper(UserMapper.class);
        // 调用代理对象方法
        List<UserVo> result = userMapper.selectCarsByUserId();
        // 关闭session
        session.close();
        System.out.println(result);

    }
```

发现查询出来的结果是 car 信息都为 null，没有值，和 sql 查询出结果不一致，如下：

```
[UserVo [cars=null, getId()=1, getUsername()=张三, getSex()=男,
```

解决方法：

第五步：修改 sql 映射文件：

```
<resultMap type="com.hpe.vo.UserVo" id="UserResultMap">
        <id column="id" property="id" />
        <result column="username" property="username" />
        <result column="birthday" property="birthday" />
        <result column="sex" property="sex" />
        <result column="address" property="address" />
        <collection property="cars" ofType="com.hpe.po.Car">
            <id column="carId" property="carId" />
            <result column="name" property="carName" />
            <result column="userId" property="userId" />
        </collection>

</resultMap>

***

<select id="selectCarsByUserId" resultMap="UserResultMap" >
        SELECT * from `user` u LEFT JOIN car c ON u.id = c.userId

</select>
```

```
<!--column不做限制，可以为任意表的字段，而property须为type 定义的pojo属性-->
<resultMap id="唯一的标识" type="映射的pojo对象">
  <id column="表的主键字段，或者可以为查询语句中的别名字段" jdbcType="字段类型" property="映射pojo对象的主键属性" />
  <result column="表的一个字段（可以为任意表的一个字段）" jdbcType="字段类型" property="映射到pojo对象的一个属性（须
一个属性）"/>
  <association property="pojo的一个对象属性" javaType="pojo关联的pojo对象">
    <id column="关联pojo对象对应表的主键字段" jdbcType="字段类型" property="关联pojo对象的主席属性"/>
    <result  column="任意表的字段" jdbcType="字段类型" property="关联pojo对象的属性"/>
  </association>
  <!-- 集合中的property须为oftype定义的pojo对象的属性-->
  <collection property="pojo的集合属性" ofType="集合中的pojo对象">
    <id column="集合中pojo对象对应的表的主键字段" jdbcType="字段类型" property="集合中pojo对象的主键属性" />
    <result column="可以为任意表的字段" jdbcType="字段类型" property="集合中的pojo对象的属性" />
  </collection>
</resultMap>
```

12、 动态 sql（重点）

通过 mybatis 提供的各种标签方法实现动态拼接 sql。

- <if>

```
<!-- 传递 pojo 综合查询用户信息 -->
<select id="findUserByUser" parameterType="user" resultType="user">
    select * from user
    where 1=1
    <if test="id!=null and id!=0">
        and id=#{id}
    </if>
    <if test="username!=null and username!=''">
        and username like '%${username}%'
    </if>
</select>
```

注意要做不等于空字符串校验。

- <where>

上边的 sql 也可以改为：

```
<select id="findUserByUser" parameterType="user" resultType="user">
        select * from user
        <where>
        <if test="id!=null and id!=0">
        and id=#{id}
        </if>
        <if test="username!=null and username!=''">
        and username like '%${username}%'
        </if>
        </where>

    </select>
```

<where >可以自动处理多余的 AND 和 OR 。

- <set>

在进行 update 操作时，在实际应用中，大多数情况下是需要更新一个或者某几个字段，如果每一条数据的全部属性都更新那么效率是非常差的，这个时候就需要一个只更新需要更新的字段的功能，<set>就可以完成这个工作。

```
<update id="updateUser" parameterType="com.hpe.po.User" >
        update user
        <set>
            <if test="username!=null and username!=''">
                username = #{username},
            </if>
            <if test="address!=null and address!=''">
                address = #{address},
            </if>
        </set>
        where id=#{id}

    </update>
```

**<set>会动态的前置 SET 关键字，同时也会消除 SQL 语句中的对于的的逗号。

注意，使用<set>和<if>元素组合进行 update 语句动态 SQL 组装时，如果<set>原始内包含的的内容都为空，则会出现 SQL 语法错误。

- <foreach>

<foreach>元素用于数组和集合的循环遍历。

根据查询一个 list 中的 id 的集合查询用户信息?

第一步：接口方法

```
// 根据ids查询用户信息 foreach

List<User> selectUserByIds(List<Integer> ids);
```

第二步：sql 映射文件：

```
<!-- 根据ids查询用户信息 foreach -->
    <select id="selectUserByIds" parameterType="list" resultType="user">
        select * from user where id in
        <foreach collection="list" item="id" index="index" open="("
            separator="," close=")">
            #{id}
        </foreach>

    </select>
```

第三步：测试

```
@Test
    public void testforeach() {
        // 获取mapper接口的代理对象
        UserMapper userMapper = session.getMapper(UserMapper.class);
        List<Integer> ids = new ArrayList<>();
        ids.add(1);
        ids.add(11);
        // 调用代理对象方法
        List<User> result = userMapper.selectUserByIds(ids);
        // 关闭session
        session.close();
        System.out.println(result);

    }
```

collection:配置的是传递过来的参数类型，它可以是一个 array（数组）、list（list 集合）、map 集合的键，或者是 POJO 包装类中数组或集合类型的属性名。

item：循环中当前的元素

index：当前元素在集合的位置下标

open：循环开始

close：循环结束

separator：中间分隔符

**在使用<foreach>时最关键也最容易出错的就是就是就是 collection 属性，该属性是必须指定的，主要有以下 3 中情况：

1、如果传入的是单个参数类型是一个数组或者 List 的时候，collection 属性值可以分别为 array 和 list。

2、如果传入的参数是多个的时候，就需要吧他们封装成一个 map 了，这个时候的 collection 属性值是 map。遍历 map 的时候 index 表示的就是 map 的 key，item 就是 map 的值。

3、如果传入的参数是 POJO 包装类的时候，collection 属性值就是该包装类中需要进行遍历的数组或者集合的属性名。

● Sql 片段

Sql 中可将重复的 sql 提取出来，使用时用 include 引用即可，最终达到 sql 重用的目的，如下：

```
<!-- 传递 pojo 综合查询用户信息 -->
<select id="findUserByUser" parameterType="user" resultType="user">
    select * from user
    where 1=1
    <if test="id!=null and id!=0">
        and id=#{id}
    </if>
    <if test="username!=null and username!=''">
        and username like '%${username}%'
    </if>
</select>
```

◆ 将 where 条件抽取出来：

```
<sql id="query_user_where">
    <if test="id!=null and id!=0">
        and id=#{id}
```

```
        </if>
        <if test="username!=null and username!=""">
            and username like '%${username}%'
        </if>
    </sql>
```

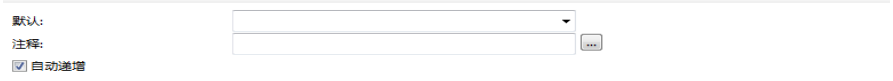
◆ 使用 include 引用:

```
    <select id="findUserByUser" parameterType="user" resultType="user">
        select * from user
        <where>
            <include refid="query_user_where"></include>
        </where>
    </select>
```

13、 mybatis 与 servlet 整合

mybatis 主要用于 web 项目，我们把之前的项目改成一个用 mybatis 和 servlet 的简单登录的 web 项目。

- 创建 user 表



- 创建 web 项目
- 添加 jar 包

- 创建数据库连接的配置文件

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/mybatis_home
jdbc.username=root
jdbc.password=123456
```

- 创建 log4j.properties

```
# Global logging configuration
log4j.rootLogger=DEBUG, stdout
# Console output...
log4j.appender.stdout=org.apache.log4j.ConsoleAppe
log4j.appender.stdout.layout=org.apache.log4j.Patt
log4j.appender.stdout.layout.ConversionPattern=%5p
```

- 创建 SqlMapConfig 配置文件

配置数据库连接池

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

<environments default="development">

    <environment id="development">

        <!-- 使用jdbc事务管理 -->
        <transactionManager type="JDBC" />
        <!-- 数据库连接池 -->
        <dataSource type="POOLED">
            <property name="driver" value="${jdbc.driver}" />
            <property name="url"
                value="${jdbc.url}" />
            <property name="username" value="${jdbc.username}" />
            <property name="password" value="${jdbc.password}" />
        </dataSource>
    </environment>
</environments>
```

- 创建实体类并在 sqlMapConfig 中声明别名

```java
public class User {

    private int id;
    private String username;
    private String password;
    private String realName;
    private String sex;
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
```

```xml
<typeAliases>
    <package name="com.hpe.po" />
</typeAliases>
```

- 创建 UserMapper 接口

```java
public interface UserMapper {

    // 根据用户名密码查询
    User selectUserByNameAndPwd(User user);


}
```

- 创建 UserMapper.xml 文件

```xml
<mapper namespace="com.hpe.mapper.UserMapper">
    <!-- 根据用户名密码查询 -->
    <select id="selectUserByNameAndPwd" parameterType="user" resultType="user">
        select * from user
        where username=#{username} and password=#{password}
    </select>
</mapper>
```

- SqlMapConfig 中配置 UserMapper.xml

```
<mappers>
    <package name="com.hpe.mapper"/>
</mappers>
```

- 创建工具类

```
public class MybatisUtil {

    private static SqlSessionFactory factory;

    static{
        try {
            factory = new SqlSessionFactoryBuilder().build(Resources.getR
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static SqlSessionFactory getSqlSessionFactory()
    {
        return factory;
    }

}
```

- 创建 UserService 接口

```
public interface UserService {

    User login(User user);

}
```

- 创建 UserService 接口的实现类

```
public class UserServiceImpl implements UserService {

    private SqlSessionFactory factory = MybatisUtil.getSqlSessionFactory();

    @Override
    public User login(User user) {

        SqlSession session = factory.openSession();
        UserMapper mapper = session.getMapper(UserMapper.class);
        User result = mapper.selectUserByNameAndPwd(user);
        session.close();
        return result;

    }

}
```

- 创建登录界面

```
<form action="userServlet?action=login" method="post">

    username:<input type="text" name="username"><br>
    password:<input type="text" name="password"><br>
    <input type="submit" value="提交">

</form>
${msg }
```

- 创建 servlet

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    request.setCharacterEncoding("utf-8");
    response.setContentType("text/html;charset=utf-8");
    String action = request.getParameter("action");
    try {
        Method method = getClass().getDeclaredMethod(action, HttpServletRequest.class, HttpS
        method.invoke(this, request, response);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

```
protected void login(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

    String username = request.getParameter("username");
    String password = request.getParameter("password");
    User user = new User();
    user.setUsername(username);
    user.setPassword(password);
    SqlSession session = factory.openSession();
    UserMapper mapper = session.getMapper(UserMapper.class);
    User currentUset = mapper.selectUserByNameAndPwd(user);
    session.close();
```

```
        if (currentUset != null) {
            HttpSession httpSession = request.getSession();
            httpSession.setAttribute("user", currentUset);
            request.setAttribute("msg", "<script>alert('登录成功')</script>");
            request.getRequestDispatcher("success.jsp").forward(request, response);
        }
        else
        {
            request.setAttribute("msg", "<script>alert('登录失败')</script>");
            request.getRequestDispatcher("login.jsp").forward(request, response);

        }

    }
```