目录

一、原生态 jdbc 编程总结

1、Jdbc 程序

```
public static void main(String[] args) {

    // 数据库连接
    Connection connection = null;
    // 预编译的Statement，使用预编译的Statement提高性能
    PreparedStatement preparedStatement = null;
    // 数据结果集
    ResultSet resultSet = null;

    try {
        // 加载数据库驱动
        Class.forName("com.mysql.jdbc.Driver");

        // 通过驱动管理类获取数据库链接
        // connection =
DriverManager.getConnection("jdbc:mysql://localhost:3306/orders","root","root");
        // 可以将上述语句拆分（一般可以从配置文件中读取相关变量）
        String url = "jdbc:mysql://localhost:3306/orders";
        String username = "root";
        String password = "root";
        connection = DriverManager.getConnection(url, username, password);

        // 定义sql语句 ?表示占位符
        String sql = "select * from user where username = ?";
        // 获取预处理statement
        preparedStatement = connection.prepareStatement(sql);
        // 设置参数，第一个参数为sql语句中参数的序号（从1开始），第二个参数为设置的参数值
        preparedStatement.setString(1, "王五");

        // 向数据库发出sql执行查询，查询出结果集
        resultSet = preparedStatement.executeQuery();
        // 遍历查询结果集
        while (resultSet.next()) {
            System.out.println(resultSet.getString("id") + "  " +
resultSet.getString("username"));
        }

    } catch (Exception e) {
        e.printStackTrace();
```

```
    } finally {
        // 释放资源
        if (resultSet != null) {
            try {
                resultSet.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
        if (preparedStatement != null) {
            try {
                preparedStatement.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
        if (connection != null) {
            try {
                connection.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

2、Jdbc 编程步骤总结

上边使用 jdbc 的原始方法（未经封装）实现了查询数据库表记录的操作。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)

3、Jdbc 编程存在的问题

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。

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。

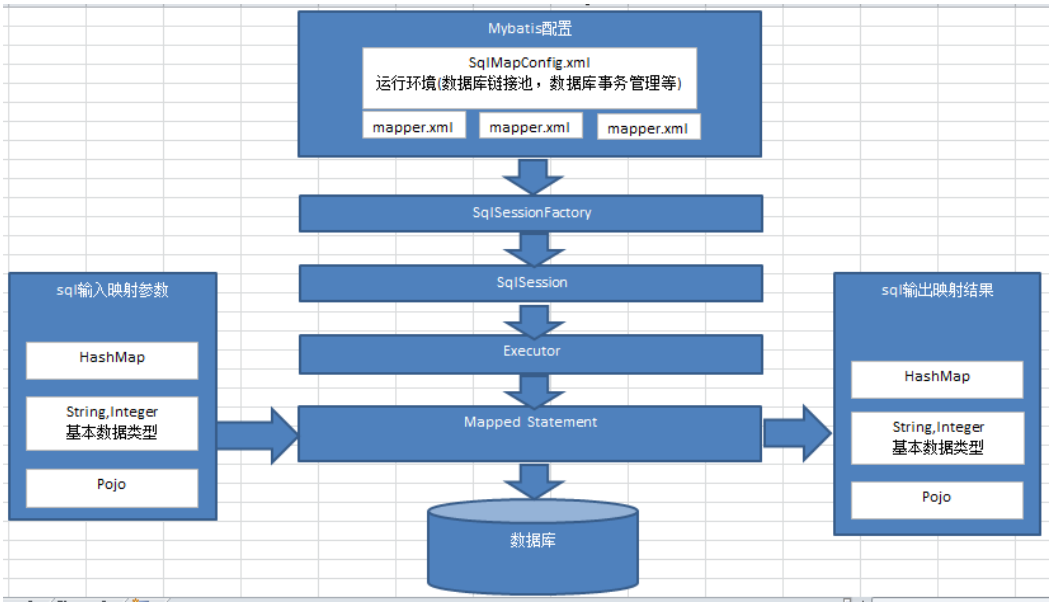
二、Mybatis 简介

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，实质上 Mybatis 对 ibatis 进行一些改进。

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、preparedStatemnt、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。

1、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。

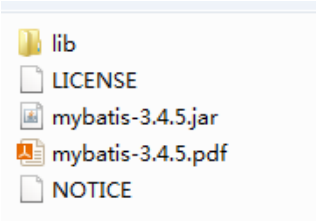
5、Mapped Statement 也是 mybatis 一个底层封装对象，它包装了 mybatis 配置信息及 sql 映射信息等。mapper.xml 文件中一个 sql 对应一个 Mapped Statement 对象，sql 的 id 即是 Mapped statement 的 id。

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。

2、mybatis 下载

mybaits 的代码由 github.com 管理，地址：https://github.com/mybatis/mybatis-3/releases



其中：mybatis-3.4.5.jar----mybatis 的核心包；lib----mybatis 的依赖包；mybatis-3.4.5.pdf----mybatis 使用手册。

三、Mybatis 入门程序

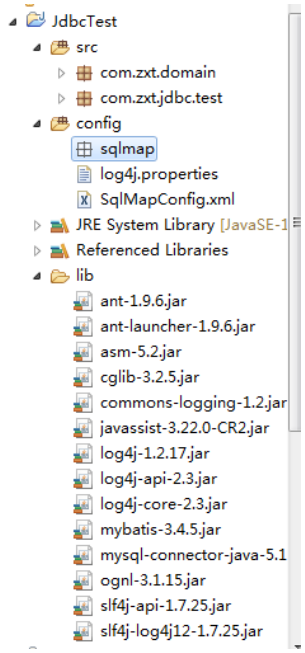
1、需求：实现以下功能：

根据用户 id 查询一个用户信息；

根据用户名称模糊查询用户信息列表；

添加用户、更新用户、删除用户；

2、创建 java 工程，添加开发所需的 jar 包，并且创建相关的配置文件，项目的目录结构如下图所示：



3、mybatis 默认使用 log4j 作为输出日志信息。log4j.properties 的内容如下，在开发阶段需要将日志的级别设置为 debug，方便程序的调试。而在用户使用阶段则需要使用 info 或者 error 级别。

```
# 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
```

4、编写 SqlMapConfig.xml 配置文件，SqlMapConfig.xml 是 mybatis 核心配置文件，进行数据库的事务和连接池配置，这里使用的 environments 标签在 mybatis 与 spring 整合之后将不再使用。

```
<?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，mybatis控制事务-->
            <transactionManager type="JDBC" />
            <!-- 数据库连接池，由mybatis管理-->
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.jdbc.Driver" />
                <property name="url"
value="jdbc:mysql://localhost:3306/orders?characterEncoding=utf-8" />
                <property name="username" value="root" />
                <property name="password" value="root" />
            </dataSource>
        </environment>
    </environments>

</configuration>
```

5、创建 pojo 类：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;
    private Date birthday;
    private String sex;
    private String address;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }
```

6、在 classpath 下的 sqlmap 目录下创建 sql 映射文件 Users.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">
```

```
    <!-- 可以在映射文件中配置很多的sql语句 -->

    <!-- 根据用户id查询用户信息 -->
    <!-- select语句用于查询数据库，其中的id用于标识映射文件中的sql语句，由于会将sql语句封装到mappedStatement对象中，所以也称为Statement的id-->
    <!-- parameterType：指定输入参数的类型，#{}表示占位符，其中的id表示输入的参数，名称可以任意，resultType：指定sql输出结果所映射的Java对象类型，-->
    <select id="findUserById" parameterType="int" resultType="com.zxt.domain.User">
        select * from user where id=#{id}
    </select>

</mapper>
```

namespace：命名空间，用于隔离 sql 语句，后面会讲另一层非常重要的作用。

parameterType:定义输入到 sql 中的映射类型,#{id}表示使用 preparedstatement 设置占位符号并将输入变量 id 传到 sql。

resultType：定义结果映射类型。

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

```
<!-- 加载映射文件 -->
<mappers>
    <mapper resource="sqlmap/User.xml" />
</mappers>
```

8、编写测试代码：

```
private static SqlSessionFactory sqlSessionFactory;

public static void main(String[] args) throws IOException {
    // mybatis配置文件
    String resource = "SqlMapConfig.xml";
    // 得到配置文件流
    InputStream inputStream = Resources.getResourceAsStream(resource);

    // 创建会话工厂
    sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);

    queryUserById();
}

// 根据用户id查询用户信息
public static User queryUserById() {
    // 通过工厂得到SqlSession
```

```
    SqlSession sqlSession = sqlSessionFactory.openSession();

    // 通过SqlSession操作数据库
    // 第一个参数statement：即映射文件中statement的id，该id需要带上命名空间
    // 第二个参数parameter：和映射文件中parameterType类型所匹配的参数
    // 返回则返回与resultType类型匹配的对象
    User user = sqlSession.selectOne("test.findUserById", 1);

    // 关闭SqlSession
    sqlSession.close();

    System.out.println(user);
    return user;
}
```

9、根据用户名模糊查询用户

```
<!-- 根据用户名模糊查询用户 -->
<!-- resultType：返回的是select语句查询返回的单条数据映射的java对象的类型
     ${}表示拼接sql串，表示将接收到的参数内容不加修饰，拼接到sql串中，这种方法容易引起sql注入问题
     ${value}：接收输入参数的内容，如果输入参数是简单类型，则${}里面只能使用value -->
<select id="findUserByName" parameterType="String" resultType="com.zxt.domain.User">
    select * from user where username like '%${value}%'
</select>
```

10、测试代码：

```
public static void main(String[] args) throws IOException {
    // mybatis配置文件
    String resource = "SqlMapConfig.xml";
    // 得到配置文件流
    InputStream inputStream = Resources.getResourceAsStream(resource);

    // 创建会话工厂
    sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);

    queryUserByName();
}

// 根据用户名模糊查询用户信息
public static List<User> queryUserByName() {
    // 通过工厂得到SqlSession
    SqlSession sqlSession = sqlSessionFactory.openSession();

    // 通过SqlSession操作数据库
    List<User> list = sqlSession.selectList("test.findUserByName", "小明");

    // 关闭SqlSession
    sqlSession.close();
```

```
    System.out.println(list);
    return list;
}
```

1、入门程序小结

1、#{}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。拼接方法可能引起 sql 注入问题，不建议使用。**

2、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。

3、selectOne 和 selectList

selectOne 查询一条记录，如果使用 select 语句返回的是多条记录，使用 selectOne 查询则抛出异常。

selectList 可以查询一条或多条记录。一条记录时返回的 list 只有一个元素而已，不会报错。

2、添加用户

1、映射文件配置

```
<!-- 添加用户 -->
<!-- parameterType：指定输入参数是java的pojo类型（包括用户信息）
    #{}中指定pojo的属性名，接收pojo对象的属性值，mybatis通过OGNL获取对象的属性 -->
<insert id="insertUser" parameterType="com.zxt.domain.User">
    insert into user(username, birthday, sex, address) values(#{username}, #{birthday}, #{sex}, #{address})
</insert>
```

2、测试程序：

```java
public static void main(String[] args) throws IOException {
    // mybatis配置文件
    String resource = "SqlMapConfig.xml";
    // 得到配置文件流
    InputStream inputStream = Resources.getResourceAsStream(resource);

    // 创建会话工厂
    sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);

    insertUser();
}

// 添加用户信息
public static void insertUser() {
    // 通过工厂得到SqlSession
    SqlSession sqlSession = sqlSessionFactory.openSession();

    // 创建一个user对象，用于插入数据库
    User user = new User();
    // 由于数据库逐渐设置为自增，所以可以不用设置id的值
    user.setUsername("李小四");
    user.setSex("男");
    user.setBirthday(new Date());
    user.setAddress("西安");

    // 通过SqlSession操作数据库
    sqlSession.insert("test.insertUser", user);

    // 提交事务
    sqlSession.commit();

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

3、在实际的应用中，我们通常还需要知道新插入的用户的主键，而这里直接 `getId()` 无法得到。需要通过配置：

```xml
<!-- 添加用户 -->
<!-- parameterType: 指定输入参数是java的pojo类型（包括用户信息）
    #{}中指定pojo的属性名，接收pojo对象的属性值，mybatis通过OGNL获取对象的属性 -->
<insert id="insertUser" parameterType="com.zxt.domain.User">
    <!-- selectKey将主键返回，需要再返回 -->
    <selectKey keyProperty="id" order="AFTER" resultType="java.lang.Integer">
        select LAST_INSERT_ID()
    </selectKey>
    insert into user(username, birthday, sex, address) values(#{username}, #{birthday}, #{sex}, #{address}
</insert>
```

```
// 获取插入用户的主键
System.out.println(user.getId());
```

添加 selectKey 实现将主键返回。

keyProperty：返回的主键存储在 pojo 中的哪个属性。

order：selectKey 的执行顺序，是相对与 insert 语句来说，由于 mysql 的自增原理执行完 insert 语句之后才将主键生成，所以这里 selectKey 的执行顺序为 after。

resultType：返回的主键是什么类型。

LAST_INSERT_ID()：是 mysql 的函数，返回 auto_increment 自增列新记录 id 值。

注意：LAST_INSERT_ID()函数只能用于自增主键。非自增主键无法通过该方法获取。需要使用 UUID 的值。

使用 mysql 的 uuid()函数获取主键，需要修改表中 id 的字段类型为 String，字段长度为 35。需要先通过 uuid()得到主键，再将主键输入到 sql 语句中。

```
<insert  id="insertUser" parameterType="com.zxt.domain.User">
    <selectKey resultType="java.lang.String" order="BEFORE" keyProperty="id">
        select uuid()
    </selectKey>
    insert into user(id,username,birthday,sex,address)
        values(#{id},#{username},#{birthday},#{sex},#{address})
</insert>
```

注意这里使用的 order 是“BEFORE”。

3、删除用户

1、配置映射文件：

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

2、测试代码：

```
public static void main(String[] args) throws IOException {
    // mybatis配置文件
    String resource = "SqlMapConfig.xml";
    // 得到配置文件流
    InputStream inputStream = Resources.getResourceAsStream(resource);

    // 创建会话工厂
    sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
```

```
    deleteUser();
}

// 删除用户信息
public static void deleteUser() {
    // 通过工厂得到SqlSession
    SqlSession sqlSession = sqlSessionFactory.openSession();

    // 通过SqlSession操作数据库
    sqlSession.insert("test.deleteUser", 26);

    // 提交事务
    sqlSession.commit();

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

4、更新用户

1、配置映射文件

```
<!-- 根据id更新用户 -->
<!-- 需要传入要更新用户的id，以及用户的更新信息
    parameterType: 指定为user对象，包括id和需要更新的信息，必须包含id -->
<update id="updateUser" parameterType="com.zxt.domain.User">
    update user set username=#{username}, birthday=#{birthday}, sex=#{sex}, address=#{address}
    where id=#{id}
</update>
```

2、测试代码：

```
public static void main(String[] args) throws IOException {
    // mybatis配置文件
    String resource = "SqlMapConfig.xml";
    // 得到配置文件流
    InputStream inputStream = Resources.getResourceAsStream(resource);

    // 创建会话工厂
    sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);

    updateUser();
}

// 更新用户信息
public static void updateUser() {
    // 通过工厂得到SqlSession
    SqlSession sqlSession = sqlSessionFactory.openSession();
```

```
    // 创建一个更新user对象
    User user = new User();
    user.setId(22);
    user.setUsername("陈小明");
    user.setSex("男");
    user.setBirthday(new Date());
    user.setAddress("河南郑州");

    // 通过SqlSession操作数据库
    sqlSession.insert("test.updateUser", user);

    // 提交事务
    sqlSession.commit();

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

四、Mybatis 入门小结

1、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。

2、Mybatis 与 hibernate 的区别

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。

总之，按照用户的需求在有限的资源环境下只要能做出维护性、扩展性良好的软件架构都是好架构，所以框架只有适合才是最好。

五、Mybatis 开发基础

1、SqlSession

SqlSession 中封装了对数据库的操作，如：查询、插入、更新、删除等。通过 SqlSessionFactory 创建 SqlSession，而 SqlSessionFactory 是通过 SqlSessionFactoryBuilder 进行创建。

SqlSession 是一个面向用户的接口， sqlSession 中定义了数据库操作，默认使用 DefaultSqlSession 实现类。

2、SqlSessionFactoryBuilder

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。

3、SqlSessionFactory

SqlSessionFactory 是一个接口，接口中定义了 openSession 的不同重载方法，SqlSessionFactory 的最佳使用范围是整个应用运行期间，一旦创建后可以重复使用，**通常以单例模式管理 SqlSessionFactory。**

4、Mybatis 的执行过程

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 = new BatchExecutor(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();
}
```

六、Mybatis 开发 Dao

1、使用 Mybatis 开发 Dao，通常有两个方法，即原始 Dao 开发方法和 Mapper 接口开发方法。

2、需求，将下边的功能实现 Dao：根据用户 id 查询一个用户信息；根据用户名称模糊查询用户信息列表；添加用户信息，删除用户信息等。

1、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。需要向 Dao 中注入 SqlSessionFactory，在具体的方法体内通过 SqlSessionFactory 创建 SqlSession，然后操作数据库。

1.1 映射文件

```
<?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">

<mapper namespace="test">
    <select id="findUserById" parameterType="int" resultType="com.zxt.domain.User">
        select * from user where id=#{id}
    </select>

    <select id="findUserByName" parameterType="String"
resultType="com.zxt.domain.User">
        select * from user where username like '%${value}%'
    </select>

    <insert id="insertUser" parameterType="com.zxt.domain.User">
        <selectKey keyProperty="id" order="AFTER" resultType="java.lang.Integer">
            select LAST_INSERT_ID()
        </selectKey>
    insert into user(username, birthday, sex, address) values(#{username}, #{birthday},
#{sex}, #{address})
    </insert>

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

```
    </delete>

    <update id="updateUser" parameterType="com.zxt.domain.User">
    update user set username=#{username}, birthday=#{birthday}, sex=#{sex},
address=#{address} where id=#{id}
    </update>

</mapper>
```

1.2 Dao 接口

```
public interface UserDao {
    // 根据用户id查询用户信息
    public User selectUserById(int id) throws Exception;

    // 根据用户名模糊查询用户信息
    public List<User> selectUserByName(String name) throws Exception;

    // 添加用户信息
    public void insertUser(User user) throws Exception;

    // 删除用户信息（根据用户id删除）
    public void deleteUser(int id) throws Exception;

    // 更新用户信息
    public void updateUser(User user) throws Exception;
}
```

1.3 Dao 实现类

```
public class UserDaoImpl implements UserDao {

    // 注入SqlSessionFactory对象（与spring整合之后，直接获取bean即可，这里使用构造方法）
    private SqlSessionFactory sqlSessionFactory;
    public UserDaoImpl(SqlSessionFactory sqlSessionFactory) {
        this.sqlSessionFactory = sqlSessionFactory;
    }

    @Override
    public User selectUserById(int id) throws Exception {
        // 通过工厂得到SqlSession
        SqlSession sqlSession = sqlSessionFactory.openSession();

        User user = sqlSession.selectOne("test.findUserById", id);
```

```
        // 关闭SqlSession
        sqlSession.close();

        return user;
    }

    @Override
    public List<User> selectUserByName(String name) throws Exception {
        SqlSession sqlSession = sqlSessionFactory.openSession();

        List<User> list = sqlSession.selectList("test.findUserByName", name);

        sqlSession.close();

        return list;
    }

    @Override
    public void insertUser(User user) throws Exception {
        SqlSession sqlSession = sqlSessionFactory.openSession();

        sqlSession.insert("test.insertUser", user);

        // 提交事务
        sqlSession.commit();
        sqlSession.close();
    }

    @Override
    public void deleteUser(int id) throws Exception {
        SqlSession sqlSession = sqlSessionFactory.openSession();

        sqlSession.insert("test.deleteUser", id);

        // 提交事务
        sqlSession.commit();

        sqlSession.close();
    }

    @Override
    public void updateUser(User user) throws Exception {
        SqlSession sqlSession = sqlSessionFactory.openSession();
```

```
        sqlSession.insert("test.updateUser", user);

        // 提交事务
        sqlSession.commit();

        sqlSession.close();
    }
}
```

1.4 Dao 测试类

```
public class UserDaoImplTest {

    private static SqlSessionFactory sqlSessionFactory;

    private static UserDaoImpl userDaoImpl;

    @Before
    public void setUp() throws Exception {
        // mybatis配置文件
        String resource = "SqlMapConfig.xml";
        // 得到配置文件流
        InputStream inputStream = Resources.getResourceAsStream(resource);

        // 创建会话工厂
        sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);

        // 创建UserDaoImpl的对象
        userDaoImpl = new UserDaoImpl(sqlSessionFactory);
    }

    @Test
    public void testGetUserById() throws Exception {
        // 调用userDaoImpl的方法
        User user = userDaoImpl.selectUserById(10);

        System.out.println(user);
    }

    @Test
    public void testQueryUserByName() throws Exception {
        List<User> list = userDaoImpl.selectUserByName("小明");
```

```
        System.out.println(list);
    }

    @Test
    public void testInsertUser() throws Exception {
        // 创建一个user对象，用于插入数据库
        User user = new User();
        user.setUsername("赵小红");
        user.setSex("女");
        user.setBirthday(new Date());
        user.setAddress("山东菏泽");

        userDaoImpl.insertUser(user);
    }

    @Test
    public void testDeleteUser() throws Exception {
        userDaoImpl.deleteUser(24);;
    }

    @Test
    public void testUpdateUser() throws Exception {
        // 创建一个user对象，用于插入数据库
        User user = new User();
        // 该id是已经存在于数据库的记录的id
        user.setId(25);
        user.setUsername("张小明");
        user.setSex("男");
        user.setBirthday(new Date());
        user.setAddress("河南郑州");

        userDaoImpl.updateUser(user);
    }
}
```

1.5 问题

原始 Dao 开发中存在以下问题：

1、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，事务提交，关闭资源等。

2、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，不利于开发维护。

3、调用 sqlSession 方法传入参数时，由于 sqlSession 方法使用泛型，即使变量类型传入错误，在编译阶段也不报错，不利于程序员开发。

2、Mapper 动态代理方式

2.1 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。例如：

```
<!-- 1、Mapper.xml文件中的namespace与mapper接口的类路径相同。 -->
<mapper namespace="com.zxt.mybatis.mapper.UserMapper">
    <!-- 根据用户id查询用户信息 -->
    <select id="selectUserById" parameterType="int" resultType="com.zxt.domain.User">
        select * from user where id=#{id}
    </select>
```

Mapper类路径

方法名称　　方法输入参数，只有一个　　方法的输出参数

```
public interface UserMapper {
    // 根据用户id查询用户信息
    public User selectUserById(int id) throws Exception;
```

2.2 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 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">

<!-- 1、 Mapper.xml文件中的namespace与mapper接口的类路径相同。 -->
<mapper namespace="com.zxt.mybatis.mapper.UserMapper">
```

```
    <!-- 根据用户id查询用户信息 -->
    <select id="selectUserById" parameterType="int" resultType="com.zxt.domain.User">
        select * from user where id=#{id}
    </select>

    <!-- 根据用户名模糊查询用户 -->
    <select id="selectUserByName" parameterType="String"
resultType="com.zxt.domain.User">
        select * from user where username like '%${value}%'
    </select>

    <!-- 添加用户 -->
    <insert id="insertUser" parameterType="com.zxt.domain.User">
        <!-- selectKey将主键返回，需要再返回 -->
        <selectKey keyProperty="id" order="AFTER" resultType="java.lang.Integer">
            select LAST_INSERT_ID()
        </selectKey>
        insert into user(username, birthday, sex, address) values(#{username},
#{birthday}, #{sex}, #{address})
    </insert>

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

    <!-- 根据id更新用户 -->
    <update id="updateUser" parameterType="com.zxt.domain.User">
        update user set username=#{username}, birthday=#{birthday}, sex=#{sex},
address=#{address} where id=#{id}
    </update>

</mapper>
```

2.3 Mapper.java(接口文件)

```
/**
 *
 * @ClassName: UserDao.java
 *
 * @Description: Mapper接口，用户管理
 * Mapper接口方法名和Mapper.xml中定义的每个statement的id相同;
 * 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;
```

```
 * 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.
 *
 * @author zxt
 *
 * @Date: 2018年1月6日 下午10:40:56
 *
 */
public interface UserMapper {
    // 根据用户id查询用户信息
    public User selectUserById(int id) throws Exception;

    // 根据用户名模糊查询用户信息
    public List<User> selectUserByName(String name) throws Exception;

    // 添加用户信息
    public void insertUser(User user) throws Exception;

    // 删除用户信息（根据用户id删除）
    public void deleteUser(int id) throws Exception;

    // 更新用户信息
    public void updateUser(User user) throws Exception;
}
```

2.4 加载 UserMapper.xml 文件

```
<!-- 加载映射文件 -->
<mappers>
     <mapper resource="sqlmap/User.xml" />

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

2.5 测试

```
public class UserMapperTest {

    private static SqlSessionFactory sqlSessionFactory;

    @Before
    public void setUp() throws Exception {
        // mybatis配置文件
        String resource = "SqlMapConfig.xml";
        // 得到配置文件流
```

```
        InputStream inputStream = Resources.getResourceAsStream(resource);

        // 创建会话工厂
        sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
    }

    @Test
    public void testSelectUserById() throws Exception {
        // 获取Session
        SqlSession sqlSession = sqlSessionFactory.openSession();

        // 通过sqlSession得到Mapper接口的代理对象
        UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

        // 调用UserMapper方法
        User user = userMapper.selectUserById(16);
        System.out.println(user);

        // 关闭Session
        sqlSession.close();
    }

    @Test
    public void testSelectUserByName() throws Exception {
        // 获取Session
        SqlSession sqlSession = sqlSessionFactory.openSession();

        // 通过sqlSession得到Mapper接口的代理对象
        UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

        // 调用UserMapper方法
        List<User> list = userMapper.selectUserByName("小明");
        System.out.println(list);

        // 关闭Session
        sqlSession.close();
    }

    @Test
    public void testInsertUser() throws Exception {
        // 获取Session
        SqlSession sqlSession = sqlSessionFactory.openSession();

        // 通过sqlSession得到Mapper接口的代理对象
```

```
        UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

        // 创建一个user对象，用于插入数据库
        User user = new User();
        user.setUsername("赵小红");
        user.setSex("女");
        user.setBirthday(new Date());
        user.setAddress("山东菏泽");

        // 调用UserMapper方法
        userMapper.insertUser(user);

        // 提交事务
        sqlSession.commit();

        // 关闭Session
        sqlSession.close();
    }

    @Test
    public void testDeleteUser() throws Exception {
        // 获取Session
        SqlSession sqlSession = sqlSessionFactory.openSession();

        // 通过sqlSession得到Mapper接口的代理对象
        UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

        userMapper.deleteUser(39);

        // 提交事务
        sqlSession.commit();

        // 关闭Session
        sqlSession.close();
    }

    @Test
    public void testUpdateUser() throws Exception {
        // 获取Session
        SqlSession sqlSession = sqlSessionFactory.openSession();

        // 通过sqlSession得到Mapper接口的代理对象
        UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
```

```
        // 创建一个user对象，用于插入数据库
        User user = new User();
        user.setId(10);
        user.setUsername("张三");
        user.setSex("男");
        user.setBirthday(new Date());
        user.setAddress("北京市");

        // 调用UserMapper方法
        userMapper.updateUser(user);

        // 提交事务
        sqlSession.commit();

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

2.6 总结

1、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。

2、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（即使输入参数只能有一个），保证 dao 的通用性。

七、SqlMapConfig.xml 全局配置文件

SqlMapConfig.xml 中配置的内容和顺序如下：（SqlMapConfig.xml 中各元素的配置顺序有严格的要求）。

1、properties（属性）

2、settings（全局配置参数）

3、typeAliases（类型别名）

4、typeHandlers（类型处理器）

5、objectFactory（对象工厂）

6、plugins（插件）

7、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）

8、mappers（映射器）

1、properties（属性）

1、需求：将数据库连接信息单独配置在 db.properties 中，只需要在 SqlMapConfig.xml 中加载 db.properties 的配置即可，这样在 SqlMapConfig.xml 中就不需要对数据库的连接信息硬编码了。同时当有大量其他的参数需要配置时，使用单独的 properties 配置文件可以方便管理。

2、在 classpath 下定义 db.properties 文件：

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

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

```
<!-- 加载配置文件 -->
<properties resource="db.properties">
    <!-- 在properties标签里面还可以配置一些属性名和属性值-->
    <!-- <property name="" value=""/> -->
</properties>
```

```
<environments default="development">
    <environment id="development">
        <transactionManager type="JDBC" />
        <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>
```

注意： MyBatis 将按照下面的顺序来加载属性：

1、在 properties 元素体内定义的属性首先被读取。

2、然后会读取 properties 元素中 resource 或 url 加载的属性，它会覆盖已读取的同名属性。

3、最后读取 parameterType 传递的属性，它会覆盖已读取的同名属性。

因此，通过 parameterType 传递的属性具有最高优先级，resource 或 url 加载的属性次之，最低优先级的是 properties 元素体内定义的属性。

建议：不要在 properties 元素体内定义任何属性，将所有的属性值都配置在 properties 配置文件中。此外在 properties 文件中定义的属性名一般需要有一定的特殊性，例如 XXX.XXX.XXX。

2、settings（配置）

Mybatis 框架在运行时可以调整一些运行参数，比如：开启二级缓存，开启延时加载等。mybatis 全局参数通过 settings 来配置，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	

3、typeAliases（类型别名）

1、在 mapper.xml 文件中，定义了很多的 statement，statement 需要 parameterType 指定输入参数的类型，需要 returnType 指定输出参数的类型。

2、在指定输入以及输出参数的全路径时，不方便开发，可以定义一些别名。定义：parameterType 和 returnType 指定类型的别名，方便开发。

3、当输入参数为 String 类型时，我们既可以使用 java.lang.String，也可以使用 String，因为 mybatis 默认使用了一些别名：

别名	映射的类型	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

3.1 自定义别名

在 SqlMapConfig.xml 中配置：

```
<!-- 定义别名 -->
<typeAliases>
    <!-- 单个别名定义 -->
    <!-- <typeAlias alias="user" type="com.zxt.domain.User"/> -->

    <!-- 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.zxt.domain"/>
    <!-- <package name="其它包"/> -->
</typeAliases>
```

4、typeHandlers（类型处理器）

类型处理器用于 java 类型和 jdbc 类型映射，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）。

5、mappers（映射器）

Mapper 配置的几种方法：

（1）<mapper resource=" " />，使用相对于类路径的资源，如：<mapper resource="sqlmap/User.xml" />

（2）<mapper url=" " />，使用完全限定路径，如：<mapper url="file:///E:\J2ee\MybatisDemo\config\sqlmap\User.xml" />

（3）<mapper class=" " />，使用 mapper 接口类路径，如：<mapper class="com.zxt.mybatis.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。

（4）<package name=""/>，注册指定包下的所有 mapper 接口，如：<package name="com.zxt.mybatis.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。（推荐使用）。

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

    <!-- 使用mapper接口类路径加载mapper映射文件 -->
    <!-- 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.zxt.mybatis.mapper.UserMapper"/> -->
    <package name="com.zxt.mybatis.mapper"/>
</mappers>
```

八、Mapper.xml 映射文件

1、输入映射（parameterType（输入类型））

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。

1.1 #{}与${}

#{}实现的是向 prepareStatement 中的预处理语句中设置参数值，sql 语句中#{}表示一个占位符即?。

```
<!-- 根据用户id查询用户信息 -->
<select id="selectUserById" parameterType="int" resultType="user">
        select * from user where id=#{id}
</select>
```

使用占位符#{}可以有效防止 sql 注入，在使用时不需要关心参数值的类型，mybatis 会自动进行 java 类型和 jdbc 类型的转换。#{}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，#{}括号中可以是 value 或其它名称。

${}和#{}不同，通过${}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。使用${}不能防止 sql 注入，但是有时用${}会非常方便，如下的例子：

```
<!-- 根据用户名模糊查询用户 -->
<select id="selectUserByName" parameterType="String" resultType="User">
    select * from user where username like '%${value}%'
</select>
```

如果本例子使用#{}则传入的字符串中必须有%号，而%是人为拼接在参数中，显然有点麻烦，如果采用${}在 sql 中拼接为%的方式则在调用 mapper 接口传递参数就方便很多。

```
// 如果使用#{}占位符号则必须人为在传参数中加%
List<User> list = userMapper.selectUserByName("%管理员%");

//如果使用${}原始符号则不用人为在参数中加%
List<User> list = userMapper.selectUserByName("管理员");
```

再比如 order by 排序，如果将列名通过参数传入 sql，根据传的列名进行排序，应该写为：ORDER BY ${columnName}，如果使用#{}将无法实现此功能。

1.2 传递简单类型

参考上边的例子。

1.3 传递 pojo 对象

Mybatis 使用 ognl 表达式解析对象字段的值，如下例子：

```
<!-- 根据id更新用户 -->
<update id="updateUser" parameterType="com.zxt.domain.User">
    update user set username=#{username}, birthday=#{birthday}, sex=#{sex},
address=#{address} where id=#{id}
</update>
```

上边红色标注的是 user 对象中的字段名称。

1.4 传递 pojo 包装对象

开发中通过 pojo 传递查询条件 ，查询条件是综合的查询条件，不仅包括用户查询条件还包括其它的查询条件（比如将用户购买商品信息也作为查询条件），**这时可以使用包装对象传递输入参数**。

1、定义包装对象：定义包装对象将查询条件(pojo)以类组合的方式包装起来。

```
public class UserQueryVo {

    private UserCustom userCustom;

    public UserCustom getUserCustom() {
        return userCustom;
    }

    public void setUserCustom(UserCustom userCustom) {
        this.userCustom = userCustom;
    }
}
```

其中 UserCustom 是继承自 User 类的扩展类。

2、Mapper.xml 文件配置

```
<!-- 综合查询：根据用户姓名和性别查询 -->
<select id="selectUserList" parameterType="com.zxt.domain.UserQueryVo" resultType="User">
    select * from user where username = #{userCustom.username} and sex = #{userCustom.sex}
</select>
```

说明：mybatis 底层通过 ognl 从 pojo 中获取属性值：#{userCustom.username}，

userCustom 即是传入的包装对象的属性。UserQueryVo 即上边定义的包装对象类型。

3、Mapper.java 接口

```
public interface UserMapper {
    // 综合查询：根据用户姓名和性别查询
    public List<User> selectUserList(UserQueryVo userQueryVo) throws Exception;
```

4、测试代码：

```
@Test
public void testSelectUserList() throws Exception {
    // 获取Session
    SqlSession sqlSession = sqlSessionFactory.openSession();

    // 通过sqlSession得到Mapper接口的代理对象
    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

    // 构造查询条件
    UserQueryVo userQueryVo = new UserQueryVo();
    UserCustom userCustom = new UserCustom();
    userCustom.setUsername("陈小明");
    userCustom.setSex("男");
    userQueryVo.setUserCustom(userCustom);

    // 调用UserMapper方法
    List<User> list = userMapper.selectUserList(userQueryVo);
    System.out.println(list);

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

1.5 传递 HashMap

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

上边红色标注的是 hashmap 的 key。

2、输出映射（resultType（输出类型））

2.1 输出简单类型

1、Mapper.xml 文件配置

```xml
<!-- 查询User表的总用户数 -->
<select id="selectUserCount" resultType="int">
    select count(*) from user
</select>
```

2、Mapper.java 接口

```java
public interface UserMapper {
    // 查询用户表的总用户数
    public int selectUserCount() throws Exception;
```

3、测试代码

```java
@Test
public void testselectUserCount() throws Exception {
    // 获取Session
    SqlSession sqlSession = sqlSessionFactory.openSession();

    // 通过sqlSession得到Mapper接口的代理对象
    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

    // 调用UserMapper方法
    int count = userMapper.selectUserCount();
    System.out.println(count);

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

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。使用 SqlSession 的 selectOne 可查询单条记录。

2.2 输出 pojo 对象

```xml
<!-- 根据用户id查询用户信息 -->
<select id="selectUserById" parameterType="int" resultType="user">
    select * from user where id=#{id}
</select>
```

2.3 输出 pojo 列表

```xml
<!-- 根据用户名模糊查询用户 -->
<select id="selectUserByName" parameterType="String" resultType="User">
    select * from user where username like '%${value}%'
</select>
```

2.4 输出 hashmap

输出 pojo 对象可以改用 hashmap 输出类型，将输出的字段名称作为 map 的 key，value 为字段值。

2.5 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。

使用 resultType 进行输出映射，只有当查询出来的列名和 pojo 中的属性名一致，该列才可以映射成功。只要查询出来的列名和 pojo 中的属性名有一个一致，就会创建 pojo 对象，如果查询出来的列名和 pojo 中的属性名全部不一致，则不会创建 pojo 对象。

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。

3.1 定义 resultMap

例如：select id 编号 ，username 姓名 from `user`，当查询出来的列名和 pojo 的属性名不一致时，需要定义一个 resultMap 对列名和 pojo 属性名之间做一个映射。

```
<!-- 定义resultMap -->
<!-- 将   select id 编号 , username 姓名 from `user`    和 User类中属性做一个映射
    type：resultMap最终映射的java对象类型，可以使用别名， id：对resultMap的唯一标识 -->
<resultMap type="com.zxt.domain.User" id="userResultMap">
    <!-- 最终resultMap对column和 property做一个映射关系
        column：查询出来的列名，property：type指定的pojo类型的属性名-->
    <!-- id：表示查询结果集的唯一标识（一般为主键） -->
    <id column="编号" property="id"/>
```

```
    <result column="姓名" property="username"/>
    <result column="生日" property="birthday"/>
</resultMap>

<select id="selectUserByResultMap" parameterType="com.zxt.domain.UserQueryVo"
resultMap="userResultMap" >
    select id  编号 , username  姓名  , birthday 生日 from user
</select>
```

Mapper.java 接口

```
public interface UserMapper {
    // 使用resultMap查询数据
    public List<User> selectUserByResultMap() throws Exception;
```

测试代码

```
@Test
public void testSelectUserByResultMap() throws Exception {
    // 获取Session
    SqlSession sqlSession = sqlSessionFactory.openSession();

    // 通过sqlSession得到Mapper接口的代理对象
    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

    // 调用UserMapper方法
    List<User> list = userMapper.selectUserByResultMap();
    System.out.println(list);

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

4、动态 sql

4.1 If 标签和 where 标签

```
<!-- 综合查询：根据用户姓名和性别查询 -->
<select id="selectUserList" parameterType="com.zxt.domain.UserQueryVo"
resultType="User">
<!-- select * from user where username = #{userCustom.username} and sex = #{userCustom.sex}
-->

    <!-- 使用动态sql -->
    select * from user
    <where>
        <if test=" userCustom.username != null and userCustom.username != '' ">
```

```
            and username = #{userCustom.username}
        </if>
        <if test=" userCustom.sex != null and userCustom.sex != '' ">
            and sex = #{userCustom.sex}
        </if>
    </where>
</select>
```

此时，会根据查询条件动态生成 sql 语句。其中 if 标签中的 test 用来判断，而<where>标签会自动处理第一个 and。

4.2 sql 片段

将上边的动态 sql 判断代码块抽取出来，组成一个 sql 片段。其它的 statement 就可以直接引用该 sql 片段了，提高了 sql 代码的复用率。

定义 sql 片段

```
<!-- 定义sql片段，id是sql片段的唯一标识 -->
<!-- 经验：sql片段一般基于单表，这样的话这个sql片段才可以重复利用到其他表中，
    并且where标签不要包括进来，因为引用该sql片段的的代码还可能有其他的条件 -->
<sql id="query_user_condition">
    <if test=" userCustom.username != null and userCustom.username != '' ">
        and username = #{userCustom.username}
    </if>
    <if test=" userCustom.sex != null and userCustom.sex != '' ">
        and sex = #{userCustom.sex}
    </if>
</sql>
```

引用 sql 片段

```
<!-- 综合查询：根据用户姓名和性别查询 -->
<select id="selectUserList" parameterType="com.zxt.domain.UserQueryVo"
resultType="User">
    <!-- select * from user where username = #{userCustom.username} and sex =
#{userCustom.sex} -->

    <!-- 使用动态sql -->
    select * from user
    <where>
        <!-- 引用sql的id，如果refid引用的id不再本mapper文件中，则需要加上namespace -->
        <include refid="query_user_condition"></include>

        <!-- 这里可以引用其他的sql -->
    </where>
</select>
```

4.3 foreach 标签

1、现在有如下需求，需要查询 id 属于某个范围的值的用户。Sql 语句实现有如下两种方法：

```
select * from user where username like '%小明%' and sex = '男' and (id = 10 or id = 22 or id = 30)
```

```
select * from user where username like '%小明%' and sex = '男' and id in (10, 22, 30)
```

此时需要向 sql 传递包含多个 id 值的数组或者 List 类型，mybatis 使用 foreach 来解析。

2、为了传递 List 类型数据，需要在查询条件封装的 pojo 中定义 list 属性 ids 存储多个用户 id，并添加 getter/setter 方法。

```
public class UserQueryVo {

    private UserCustom userCustom;

    // 传入多个id
    private List<Integer> ids;

    public List<Integer> getIds() {
        return ids;
    }

    public void setIds(List<Integer> ids) {
        this.ids = ids;
    }
```

3、mapper.xml

```
<!-- 定义sql片段，id是sql片段的唯一标识 -->
<!-- 经验：sql片段一般基于单表，这样的话这个sql片段才可以重复利用到其他表中，
    并且where标签不要包括进来，因为引用该sql片段的的代码还可能有其他的条件 -->
<sql id="query_user_condition">
    <if test=" userCustom.username != null and userCustom.username != '' ">
        and username like '%${userCustom.username}%'
    </if>
    <if test=" userCustom.sex != null and userCustom.sex != '' ">
        and sex = #{userCustom.sex}
    </if>

    <!-- 使用foreach遍历传入的ids -->
    <!-- collection：指定输入对象的集合属性名称， item：每次遍历的中间变量
        open：开始遍历时拼接的sql串， close：结束遍历时拼接的串， separator：遍历的两个对象中间需要拼接的串-->
    <!-- and (id = 10 or id = 22 or id = 30) -->
```

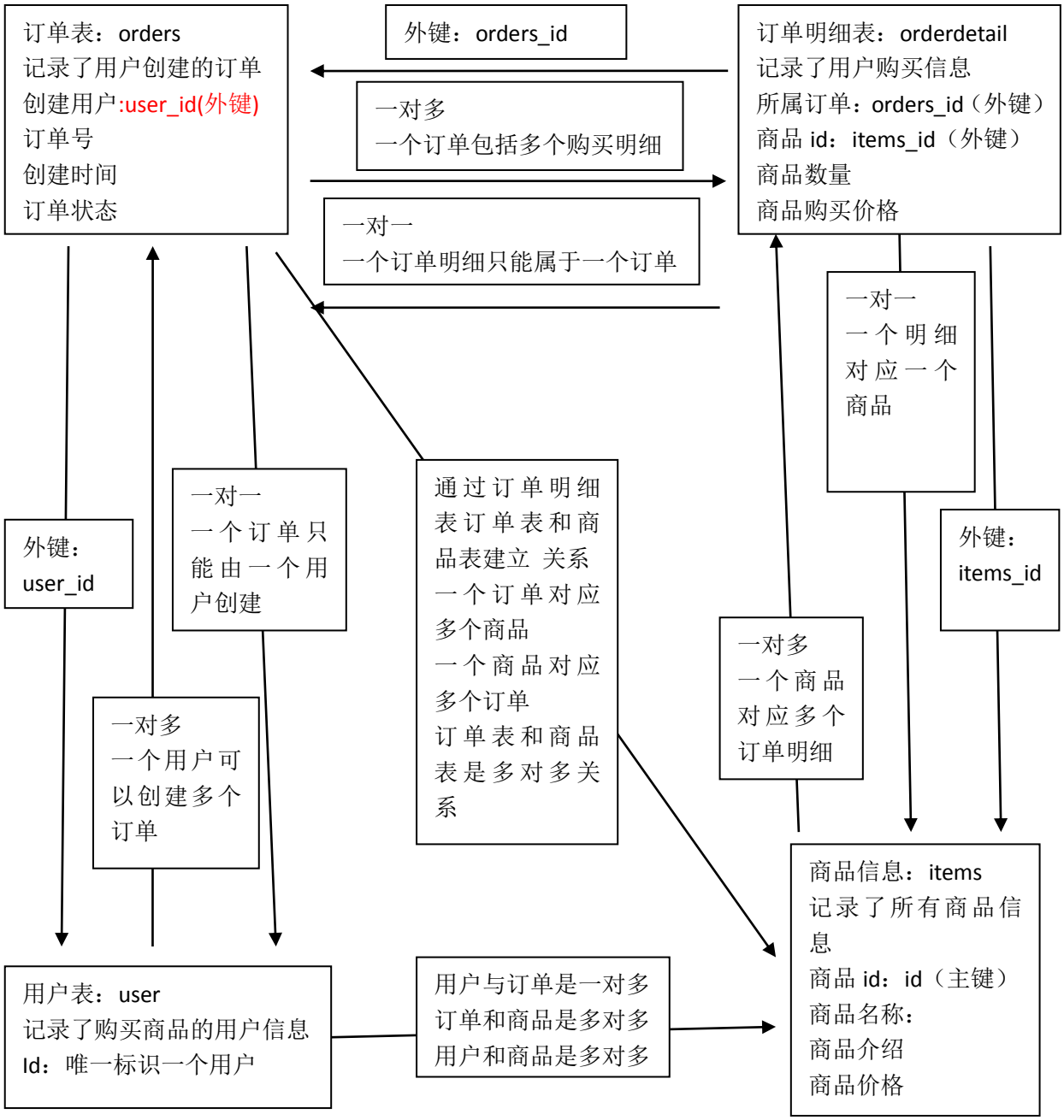
```
    <foreach collection="ids" item="id" open="and ( " close=" )" separator="or">
        id=#{id}
    </foreach>

    <!-- and id in (10, 22, 30) -->
    <foreach collection="ids" item="id" open="and id in ( " close=" )" separator=",">
        #{id}
    </foreach>
</sql>
```

4、测试代码

```
@Test
public void testSelectUserList() throws Exception {
    // 获取Session
    SqlSession sqlSession = sqlSessionFactory.openSession();

    // 通过sqlSession得到Mapper接口的代理对象
    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

    // 构造查询条件
    UserQueryVo userQueryVo = new UserQueryVo();
    UserCustom userCustom = new UserCustom();
    userCustom.setUsername("小明");
    // userCustom.setSex("男");
    List<Integer> ids = new ArrayList<Integer>();
    ids.add(10);
    ids.add(22);
    ids.add(30);
    // ids通过userQueryVo传入到statement中
    userQueryVo.setIds(ids);
    userQueryVo.setUserCustom(userCustom);

    // 调用UserMapper方法
    List<User> list = userMapper.selectUserList(userQueryVo);
    System.out.println(list);

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

九、Mybatis 关联查询

1、商品订单数据模型



简单分析：

用户表 user：记录了购买商品的用户信息

商品表 items：记录了商品信息

订单表 orders：记录了用户所创建的订单（购买商品的订单）

订单明细表 orderdetail：记录了订单详情信息，即购买的商品信息。

表与表之间的关系：

user 和 orders

user-->orders：一个用户可以创建多个订单，一对多

orders-->user：一个订单只能由一个用户创建，一对一

orders 和 orderdetail

orders-->orderdetail：一个订单包括多个订单明细，因为一个订单里面可以购买多个商品，每个商品的购买信息都在 orderdetail 中记录，一对多

orderdetail--> orders：一个订单明细只能包含在一个订单里面，一对一

orderdetail 和 items

orderdetail-->items：一个订单明细只对应一个商品信息，一对一

items-->orderdetail：一个商品可以出现在多个订单明细中，多对一

orders 和 items

orders 和 items 之间可以由 orderdetail 建立关系，一个订单中包括多个订单明细，即多个商品，所以订单到商品是一对多；而一个商品可以包含在多个订单明细，即多个订单中，所以商品到订单也是一对多。总结订单与商品之间是多对多的关系。

user 和 items

一个用户可以购买多个商品，一个商品可由多个用户购买，它们之间也是多对多的关系。

2、一对一查询

场景：查询所有订单信息，关联查询下单用户信息。

注意：因为一个订单信息只会是一个人下的订单，所以从查询订单信息出发关联查询用户信息为一对一查询。如果从用户信息出发查询用户下的订单信息则为一对多查询，因为一个用户可以下多个订单。

```
select orders.*, user.username, user.address
from orders, user
where orders.user_id = user.id
```

2.1 resultType 实现

1、使用 resultType 返回查询结果，首先要定义 pojo 类，pojo 的属性和查询到的列名要完全一致。这里不仅需要查询 orders 的信息，还要查询用户的信息，所以定义 pojo

类继承 orders，并在里面扩展用户的信息。

```
// 通过此类可以映射订单和用户查询的结果，让此类继承字段较多的pojo类
public class OrdersCustom extends Orders {

    // 这里是对订单的信息进行扩展

    // 有选择得将所需要的用户信息扩展进来（这里将所有的用户信息都包含进来）
    private String username;
    private String address;
```

OrdersCustom 类继承 Orders 类后 OrdersCustom 类包括了 Orders 类的所有字段，只需要定义用户的信息字段即可。

2、Mapper.xml

```
<mapper namespace="com.zxt.mybatis.mapper.OrdersMapper">

    <!-- 查询订单信息，关联查询创建该订单的用户信息 -->
    <select id="selectOrdersUsers" resultType="com.zxt.domain.OrdersCustom">
        select orders.*, user.username, user.address
        from orders, user
        where orders.user_id = user.id
    </select>

</mapper>
```

3、Mapper.java 接口

```
public interface OrdersMapper {
    // 查询订单信息，关联查询创建该订单的用户信息
    public List<OrdersCustom> selectOrdersUsers() throws Exception;
}
```

4、测试代码

```
public class OrdersMapperTest {
    private static SqlSessionFactory sqlSessionFactory;

    @Before
    public void setUp() throws Exception {
        // mybatis配置文件
        String resource = "SqlMapConfig.xml";
        // 得到配置文件流
        InputStream inputStream = Resources.getResourceAsStream(resource);

        // 创建会话工厂
        sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
    }
```

```
    @Test
    public void testSelectOrdersUsers() throws Exception {
        // 获取SqlSession
        SqlSession sqlSession = sqlSessionFactory.openSession();

        // 创建OrdersCustom接口的动态代理对象
        OrdersMapper ordersMapper = sqlSession.getMapper(OrdersMapper.class);

        // 调用ordersCustom的方法
        List<OrdersCustom> list = ordersMapper.selectOrdersUsers();
        System.out.println(list);

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

5、总结

定义专门的 po 类作为输出类型，其中定义了 sql 查询结果集所有的字段。此方法较为简单，企业中使用普遍。

2.2 resultMap 实现

1、使用 resultMap 将查询结果中的订单信息映射到 orders 对象中，在 orders 类中添加 User 类型的属性，将关联查询出来的用户信息映射到 orders 对象中的 user 属性中。

需要在 orders 类中添加 user 属性：

```
// 订单表
public class Orders {

    private Integer id;
    private Integer user_id;
    private String number;
    private Date createtime;
    private String note;

    // 用户信息
    private User user;
```

2、Mapper.xml

定义 resultMap

```
<!-- 关联查询订单和用户信息的resultMap -->
<resultMap type="com.zxt.domain.Orders" id="orderUserResultMap">
    <!-- 配置订单信息的映射 -->
```

```
    <!-- id：指定订单信息查询列中的唯一标识，订单信息中的唯一标识 -->
    <id column="id" property="id"/>
    <result column="user_id" property="user_id"/>
    <result column="number" property="number"/>
    <result column="createtime" property="createtime"/>
    <result column="note" property="note"/>

    <!-- 配置关联用户的映射 -->
    <!-- association：用于映射关联查询的单个对象的信息
    property：要将关联查询的信息映射到orders的哪个属性中， javaType：user属性的类型-->
    <association property="user" javaType="com.zxt.domain.User">
        <!-- id：指定用户信息的唯一标识列， javaType：映射到user的那个属性-->
        <id column="user_id" property="id"/>
        <result column="username" property="username"/>
        <result column="address" property="address"/>
    </association>
</resultMap>
```

定义 statement

```
<select id="selectOrdersUsers" resultMap="orderUserResultMap">
    select orders.*, user.username, user.address
    from orders, user
    where orders.user_id = user.id
</select>
```

3、Mapper.java 接口

```
// 使用resultMap实现 查询订单信息，关联查询创建该订单的用户信息
public List<Orders> selectOrdersUsersByResultMap() throws Exception;
```

4、测试代码。

```
@Test
public void testSelectOrdersUsersByResultMap() throws Exception {
    // 获取SqlSession
    SqlSession sqlSession = sqlSessionFactory.openSession();

    // 创建OrdersCustom接口的动态代理对象
    OrdersMapper ordersMapper = sqlSession.getMapper(OrdersMapper.class);

    // 调用ordersCustom的方法
    List<Orders> list = ordersMapper.selectOrdersUsersByResultMap();
    System.out.println(list);

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

2.3 一对一查询总结

resultType：使用 resultType 比较简单，如果查询出来的列名在 pojo 对象中没有对应的属性，需要增加列名对应的属性，即可完成映射。

resultMap：需要单独定义 resultMap，实现较复杂，如果对查询结果有特殊的要求，使用 resultMap 可以完成将关联查询映射到 pojo 的属性中。

resultMap 可以实现延迟加载，而 resultType 无法实现延迟加载。

3、一对多查询

需求：查询所有订单信息及订单下的订单明细信息。订单信息与订单明细为一对多关系。

```
select
    orders.*,
    user.username,
    user.address,
    orderdetail.id orderdetail_id,
    orderdetail.items_id,
    orderdetail.items_num
from
    orders,user,orderdetail
where
    orders.user_id = user.id
    and orders.id = orderdetail.orders_id
```

由于一个订单中包含多个订单明细，所以查询出来的结果中，订单信息会出现重复，如下：

id	user_id	number	createtime
3	30	1000010	2015-02-04 13:22:35
3	30	1000010	2015-02-04 13:22:35
4	30	1000011	2015-02-03 13:22:41
4	30	1000011	2015-02-03 13:22:41

3.1 resultMap 实现

1、若使用 resultType 将结果映射到 pojo 中，则订单信息就会重复，现在要求不让

订单信息重复，则使用 resultMap 实现，在 orders 类中定义一个 orderdetail 的列表属性用来对应一个订单中的订单明细。

```java
// 订单表
public class Orders {

    private Integer id;
    private Integer user_id;
    private String number;
    private Date createtime;
    private String note;

    // 用户信息
    private User user;

    // 订单明细信息
    private List<Orderdetail> orderdetails;
```

2、定义 resultMap：

```xml
<resultMap type="com.zxt.domain.Orders" id="orderAndOrderdetailResultMap">
    <!-- 订单信息 -->
    <id column="id" property="id"/>
    <result column="user_id" property="user_id"/>
    <result column="number" property="number"/>
    <result column="createtime" property="createtime"/>
    <result column="note" property="note"/>

    <!-- 用户信息 -->
    <!-- association：用于映射关联查询的单个对象的信息 -->
    <association property="user" javaType="com.zxt.domain.User">
        <id column="user_id" property="id"/>
        <result column="username" property="username"/>
        <result column="address" property="address"/>
    </association>

    <!-- 订单明细信息 -->
    <!-- collection:将关联查询到的多条记录映射到集合属性中
            ofType：指定映射到集合属性中的pojo类型 -->
    <collection property="orderdetails" ofType="com.zxt.domain.Orderdetail">
        <id column="orderdetail_id" property="id"/>
        <result column="items_id" property="items_id"/>
        <result column="items_num" property="items_num"/>
    </collection>
</resultMap>
```

collection 部分定义了查询订单明细信息。collection：表示关联查询结果集。

property="orderdetails"：关联查询的结果集存储在 com.zxt.domain.Orders 上哪个属性。

ofType=""com.zxt.domain.Orderdetail "：指定关联查询的结果集中的对象类型即 List 中的对象类型。

<id />及<result/>的意义同一对一查询。

3、定义 statement

```
<select id="selectOrderAndOrderdetailByResultMap"
resultMap="orderAndOrderdetailResultMap">
    select orders.*, user.username,   user.address, orderdetail.id orderdetail_id,
           orderdetail.items_id, orderdetail.items_num
    from orders, user, orderdetail
    where orders.user_id = user.id and orders.id = orderdetail.orders_id
</select>
```

4、Mapper.java

```
// 查询订单信息 关联查询该订单的订单明细
public List<Orders> selectOrderAndOrderdetailByResultMap() throws Exception;
```

5、测试代码

```
@Test
public void testSelectOrderAndOrderdetailByResultMap() throws Exception {
    // 获取SqlSession
    SqlSession sqlSession = sqlSessionFactory.openSession();
    // 创建OrdersCustom接口的动态代理对象
    OrdersMapper ordersMapper = sqlSession.getMapper(OrdersMapper.class);

    // 调用ordersCustom的方法
    List<Orders> list = ordersMapper.selectOrderAndOrderdetailByResultMap();
    System.out.println(list);

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

6、观察发现，orderAndOrderdetailResultMap 中订单信息与用户信息的映射关系与查询订单及关联查询用户信息的 resultMap 的定义一样，所以可以让 orderAndOrderdetailResultMap 继承 orderUserResultMap，这样就可以不用编写重复代码了。

```
<resultMap type="com.zxt.domain.Orders" id="orderAndOrderdetailResultMap"
extends="orderUserResultMap">
    <collection property="orderdetails" ofType="com.zxt.domain.Orderdetail">
        <id column="orderdetail_id" property="id"/>
        <result column="items_id" property="items_id"/>
```

```
        <result column="items_num" property="items_num"/>
    </collection>
</resultMap>
```

7、小结

mybatis 使用 resultMap 的 collection 对关联查询的多条记录映射到一个 list 集合属性中。

若使用 resultType，则需要自己实现去重复。

4、多对多查询

1、需求：查询用户购买的商品信息。需要查询所有用户信息，关联查询订单及订单明细信息，订单明细信息中关联查询商品信息。

```
select user.id user_id, user.username, user.address,
       orders.id orders_id, orders.number,
       orderdetail.id orderdetail_id, orderdetail.items_num,
       items.id items_id, items.name, items.detail, items.price
from user, orders, orderdetail, items
where orders.user_id = user.id
      and orderdetail.orders_id = orders.id
      and items.id = orderdetail.items_id
```

2、映射思想

将用户信息映射到 User 中，在 user 中添加一个 List<Orders> orderslist，将用户创建的订单信息映射到 orderslist；在 Orders 中添加订单明细的属性 List<Orderdetail> orderdetail，将订单明细映射到 orderdetail；在 Orderdetail 中添加 items 属性，将订单明细对应的商品信息映射到 items 中。

3、定义 resultMap

```
<!-- 查询用户信息，关联查询用户购买的商品信息 -->
<resultMap type="com.zxt.domain.User" id="userAndItemsResultMap">
    <!-- 用户信息 -->
    <id column="user_id" property="id"/>
    <result column="username" property="username"/>
    <result column="address" property="address"/>

    <!-- 订单信息：一个用户对应多条订单，使用collection -->
```

```
    <collection property="orderslist" ofType="com.zxt.domain.Orders">
        <id column="orders_id" property="id"/>
        <result column="number" property="number"/>

        <!-- 订单明细信息：一个订单包含多个订单明细，使用collection -->
        <collection property="orderdetails" ofType="com.zxt.domain.Orderdetail">
            <id column="orderdetail_id" property="id"/>
            <result column="items_num" property="items_num"/>

            <!-- 商品信息：一个订单明细只对应一个商品信息，使用association -->
            <association property="items" javaType="com.zxt.domain.Items">
                <id column="items_id" property="id"/>
                <result column="name" property="name"/>
                <result column="detail" property="detail"/>
                <result column="price" property="price"/>
            </association>
        </collection>
    </collection>
</resultMap>

<select id="selectUserAndItemsByResultMap" resultMap="userAndItemsResultMap">
    select user.id user_id, user.username, user.address,
        orders.id orders_id, orders.number,
        orderdetail.id orderdetail_id, orderdetail.items_num,
        items.id items_id, items.name, items.detail, items.price
    from user, orders, orderdetail, items
    where orders.user_id = user.id
        and orderdetail.orders_id = orders.id
        and items.id = orderdetail.items_id
</select>
```

需要关联查询映射的信息是：订单、订单明细、商品信息：

订单：一个用户对应多个订单，使用 collection 映射到用户对象的订单列表属性中；

订单明细：一个订单对应多个明细，使用 collection 映射到订单对象中的明细属性中；

商品信息：一个订单明细对应一个商品，使用 association 映射到订单明细对象的商品属性中。

4、Mapper.java

```
// 查询用户信息 关联查询用户购买的商品明细
public List<User> selectUserAndItemsByResultMap() throws Exception;
```

5、测试代码

```
@Test
public void testSelectUserAndItemsByResultMap() throws Exception {
```

```
    // 获取SqlSession
    SqlSession sqlSession = sqlSessionFactory.openSession();

    // 创建OrdersCustom接口的动态代理对象
    OrdersMapper ordersMapper = sqlSession.getMapper(OrdersMapper.class);

    // 调用ordersCustom的方法
    List<User> list = ordersMapper.selectUserAndItemsByResultMap();
    System.out.println(list);

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

6、小结

一对多是多对多的特例，如下需求：查询用户购买的商品信息，用户和商品的关系是多对多关系。

需求 1：查询字段：用户账号、用户名称、用户性别、商品名称、商品价格(最常见)，企业开发中常见明细列表，用户购买商品明细列表，使用 resultType 将上边查询列映射到 pojo 输出。

需求 2：查询字段：用户账号、用户名称、购买商品数量、商品明细（鼠标移上显示明细），使用 resultMap 将用户购买的商品明细列表映射到 user 对象中。

5、resultMap 小结

resultType

作用：将查询结果按照 sql 列名 pojo 属性名一致性映射到 pojo 中。

场合：常见一些明细记录的展示，比如用户购买商品明细，将关联查询信息全部展示在页面时，此时可直接使用 resultType 将每一条记录映射到 pojo 中，在前端页面遍历 list（list 中是 pojo）即可。

resultMap

使用 association 和 collection 完成一对一和一对多高级映射（对结果有特殊的映射要求）。

association：

作用：将关联查询信息映射到一个 pojo 对象中。

场合：为了方便查询关联信息可以使用 association 将关联订单信息映射为用户对象

的 pojo 属性中，比如：查询订单及关联用户信息。

使用 resultType 无法将查询结果映射到 pojo 对象的 pojo 属性中，根据对结果集查询遍历的需要选择使用 resultType 还是 resultMap。

collection：

作用：将关联查询信息映射到一个 list 集合中。

场合：为了方便查询遍历关联信息可以使用 collection 将关联信息映射到 list 集合中，比如：查询用户权限范围模块及模块下的菜单，可使用 collection 将模块映射到模块 list 中，将菜单列表映射到模块对象的菜单 list 属性中，这样的作的目的也是方便对查询结果集进行遍历查询。

如果使用 resultType 无法将查询结果映射到 list 集合中。

6、延迟加载

1、首先在 mybatis 中默认是不开启延迟加载的，若需要实现延迟加载的功能，需要先进行配置（在 SqlMapConfig.xml 中配置）。

设置项	描述	允许	默认
lazyLoadingEnabled	全局性设置懒加载。如果设为‘false’，则所有相关联的都会被初始化加载。	true false	false
aggressiveLazyLoading	当设置为‘true’的时候，懒加载的对象可能被任何懒属性全部加载。否则，每个属性都按需加载。	true false	true

```
<!-- 全局配置参数，需要时在加载 -->
<settings>
    <!-- 打开延迟加载的开关 -->
    <setting name="lazyLoadingEnabled" value="true"/>
    <!-- 将积极加载改为延迟加载，即按需加载 -->
    <setting name="aggressiveLazyLoading" value="false"/>
</settings>
```

2、什么是延迟加载：resultMap 可以实现高级映射（使用 association、collection 实现一对一及一对多映射），association、collection 具备延迟加载的功能。

需求：查询订单并且关联查询用户的信息，如果先查询订单的信息就可以满足用户的需求，只有当需要用户的信息时才去查询，这里对用户信息的按需查询就是延迟加载。

延迟加载：先从单表（简单）查询，需要时再从关联表查询，大大提高数据库的性能，因为查询单表要比关联表快的多。

3、查询订单关联查询用户信息的延迟加载实例

（1）只查询订单信息，select orders.* from orders.在查询订单的 statement 中使用 association 去延迟加载（执行）关联查询用户的 statement。

```
<select id="selectOrdersUserLazyLoading" resultMap="ordersUserLazyLoadingResultMap">
    select orders.* from orders
</select>
```

（2）关联查询用户的信息，通过上边查询到的订单信息中的 user_id 去关联查询用户信息。

```
<!-- 根据用户id查询用户信息 -->
<select id="selectUserById" parameterType="int" resultType="user">
    select * from user where id=#{id}
</select>
```

上边的 statement 中，先去执行 *selectOrdersUserLazyLoading*，当需要查询用户的时候再去执行 *selectUserById*，且该过程由 resultMap 中的 association 配置来实现。

4、定义 resultMap 和 statement

```
<!-- 延迟加载方式查询订单信息及关联的用户信息 -->
<!-- 延迟加载的resultMap -->
<resultMap type="com.zxt.domain.Orders" id="ordersUserLazyLoadingResultMap">
    <!-- 订单信息的映射 -->
    <id column="id" property="id"/>
    <result column="user_id" property="user_id"/>
    <result column="number" property="number"/>
    <result column="createtime" property="createtime"/>
    <result column="note" property="note"/>

    <!-- 使用association实现关联用户的延迟加载 -->
    <!-- select：指定延迟加载所需要执行的statement的id（即根据用户id（user_id）查询用户的statement）
        即UserMapper.xml里面的根据用户id查询用户信息的statement的id：selectUserById
        column：订单信息中关联用户的关联列，即user_id
    SELECT
        orders.*,
        (select username from user where user.id=orders.user_id) username,
        (select address from user where user.id=orders.user_id) address
    from orders
    -->
    <association property="user" javaType="com.zxt.domain.User"
        select="com.zxt.mybatis.mapper.UserMapper.selectUserById" column="user_id">
    </association>
</resultMap>

<select id="selectOrdersUserLazyLoading" resultMap="ordersUserLazyLoadingResultMap">
```

```
    select orders.* from orders
</select>
```

5、Mapper.java

```
// 查询订单信息，关联查询用户的信息（用户信息延迟加载）
public List<Orders> selectOrdersUserLazyLoading() throws Exception;
```

6、测试代码

测试思路：1、执行上边的 Mapper 方法（selectOrdersUserLazyLoading），内部区调用 OrdersMapper.xml 中的 selectOrdersUserLazyLoading 语句，查询订单的信息（单表）。

2、在程序中去遍历查询出来的 List<Orders>，当我们去调用 Orders 中的 getUser 方法时，开始延迟加载。

3、延迟加载，去调用 UserMapper.xml 里面的 selectUserById 获取用户信息。

```
@Test
// 查询订单信息，关联查询用户的信息（用户信息延迟加载）
public void testSelectOrdersUserLazyLoading() throws Exception {
    // 获取SqlSession
    SqlSession sqlSession = sqlSessionFactory.openSession();

    // 创建OrdersCustom接口的动态代理对象
    OrdersMapper ordersMapper = sqlSession.getMapper(OrdersMapper.class);

    // 查询订单信息
    List<Orders> list = ordersMapper.selectOrdersUserLazyLoading();
    System.out.println(list);
    // 遍历上边的Orders列表，
    for(Orders orders:list) {
        // 调用Orders中的getUser()方法，这里实现了按需加载
        User user = orders.getUser();
        System.out.println(user);
    }

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

7、不使用 mybatis 提供的延迟加载功能是否可以实现延迟加载？

实现方法：针对订单和用户两个表定义两个 mapper 方法。

1、订单查询 mapper 方法

2、根据用户 id 查询用户信息 mapper 方法

默认使用订单查询 mapper 方法只查询订单信息。当需要关联查询用户信息时再调用根据用户 id 查询用户信息 mapper 方法查询用户信息。

6.1 延迟加载小结

作用：当需要查询关联信息时再去数据库查询，默认不去关联查询，提高数据库性能。只有使用 resultMap 支持延迟加载设置。

场合：当只有部分记录需要关联查询其它信息时，此时可按需延迟加载，需要关联查询时再向数据库发出 sql，以提高数据库性能。

当全部需要关联查询信息时，此时不用延迟加载，直接将关联查询信息全部返回即可，可使用 resultType 或 resultMap 完成映射。

十、Mybatis 查询缓存

Mybatis 提供查询缓存，用于减轻数据库压力，提高数据库性能。如下图，是 mybatis 一级缓存和二级缓存的区别图解：

Mybatis 一级缓存的作用域是同一个 SqlSession，在同一个 sqlSession 中两次执行相同的 sql 语句，第一次执行完毕会将数据库中查询的数据写到缓存（内存），第二次会从缓存中获取数据将不再从数据库查询，从而提高查询效率。当一个 sqlSession 结束后该 sqlSession 中的一级缓存也就不存在了。Mybatis 默认开启一级缓存。

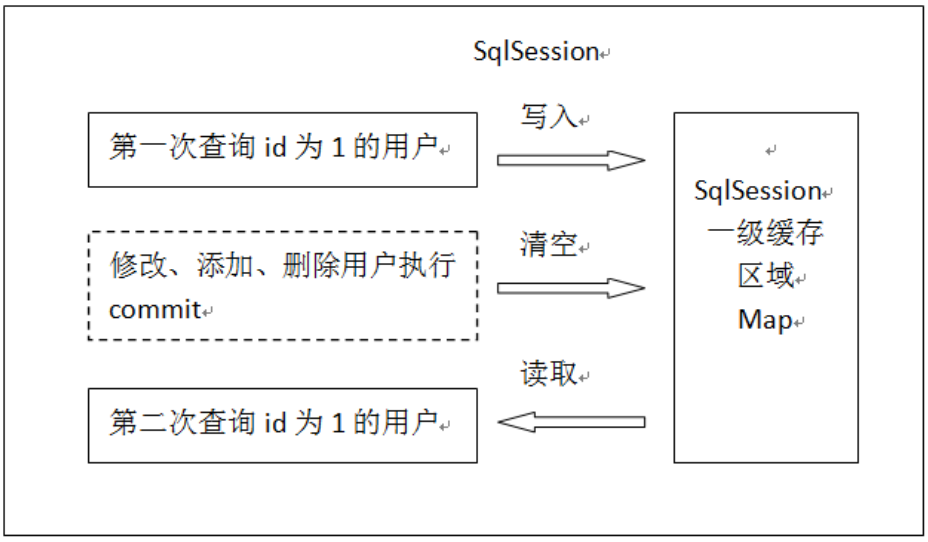
一级缓存属于 SqlSession 级别的缓存。在操作数据库时需要构造 sqlSession 对象，在对象中有一个数据结构（HashMap）用户存储缓存数据，不同 SqlSession 之间的缓存数据区域互不影响。

Mybatis 二级缓存是多个 SqlSession 共享的，其作用域是 mapper 的同一个 namespace，不同的 sqlSession 两次执行相同 namespace 下的 sql 语句且向 sql 中传递参数也相同即最终执行相同的 sql 语句，第一次执行完毕会将数据库中查询的数据写到缓存（内存），第二次会从缓存中获取数据将不再从数据库查询，从而提高查询效率。Mybatis 默认没有开启二级缓存需要在 setting 全局参数中配置开启二级缓存。

1、一级缓存

1.1 原理

下图是根据 id 查询用户的一级缓存图解：



一级缓存区域是根据 SqlSession 为单位划分的。

每次查询会先从缓存区域找，如果找不到从数据库查询，查询到数据将数据写入缓存。Mybatis 内部存储缓存使用一个 HashMap，key 为 hashCode+sqlId+Sql 语句。value 为从查询出来映射生成的 java 对象。

sqlSession 执行 insert、update、delete 等操作 commit 提交后会清空缓存区域。这样做的目的是保证缓存中存储的永远是最新数据，避免脏读。

1.2 测试 1

```
@Test
// 一级缓存测试
public void testCache1() throws Exception {
    // 获取Session
    SqlSession sqlSession = sqlSessionFactory.openSession();
    // 通过sqlSession得到Mapper接口的代理对象
    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

    // 第一次发起请求查询用户id为10的用户
    User user1 = userMapper.selectUserById(10);
    System.out.println(user1);

    // 第二次发起请求查询用户id为10的用户
    User user2 = userMapper.selectUserById(10);
    System.out.println(user2);

    sqlSession.close();
}
```

可以看出两次查询相同 id 的用户，只向数据库发送了一次 sql 请求。

```
DEBUG [main] - Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connectic
DEBUG [main] - ==>  Preparing: select * from user where id=?
DEBUG [main] - ==> Parameters: 10(Integer)
DEBUG [main] - <==      Total: 1
User [id=10, username=张三, birthday=Sun Jan 07 00:00:00 CST 2018, sex=男, address=北京市]
User [id=10, username=张三, birthday=Sun Jan 07 00:00:00 CST 2018, sex=男, address=北京市]
```

1.3 测试 2

```java
@Test
// 一级缓存测试
public void testCache1() throws Exception {
    // 获取Session
    SqlSession sqlSession = sqlSessionFactory.openSession();

    // 通过sqlSession得到Mapper接口的代理对象
    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

    // 第一次发起请求查询用户id为10的用户
    User user1 = userMapper.selectUserById(10);
    System.out.println(user1);

    // sqlSession执行insert、update、delete等操作commit提交后会清空缓存区域。
    userMapper.updateUser(user1);
    sqlSession.commit();

    // 第二次发起请求查询用户id为10的用户
    User user2 = userMapper.selectUserById(10);
    System.out.println(user2);

    sqlSession.close();
}
```

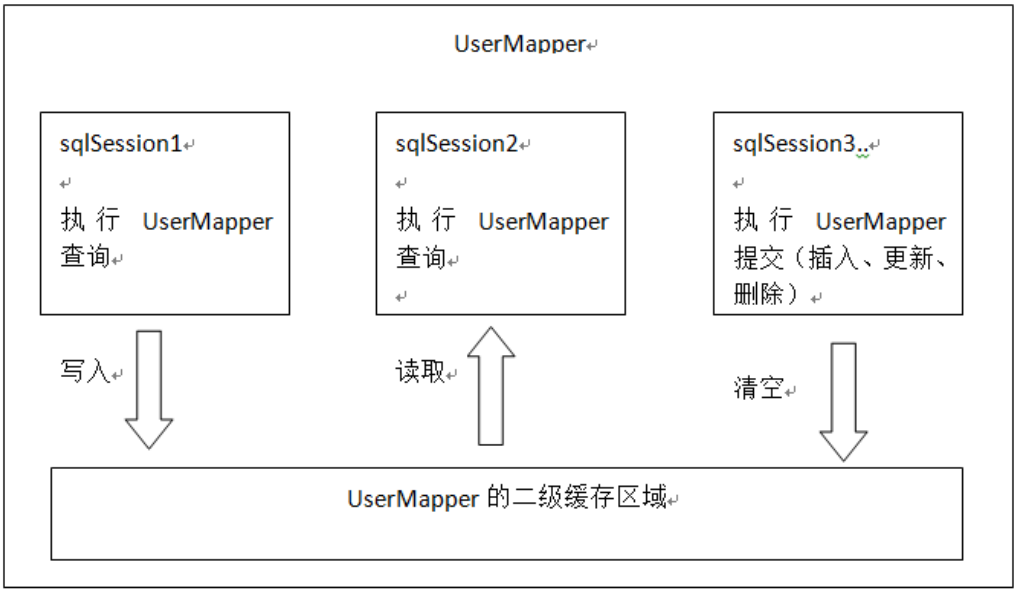
执行 update 操作之后，缓存数据被清空，再次发起查询请求时，无法从一级缓存中找到数据，需要再次向数据库发送 sql 请求。

```
DEBUG [main] - Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@24a3
DEBUG [main] - ==>  Preparing: select * from user where id=?
DEBUG [main] - ==> Parameters: 10(Integer)
DEBUG [main] - <==      Total: 1
User [id=10, username=张三, birthday=Sun Jan 07 00:00:00 CST 2018, sex=男, address=北京市]
DEBUG [main] - ==>  Preparing: update user set username=?, birthday=?, sex=?, address=? where id=?
DEBUG [main] - ==> Parameters: 张三(String), 2018-01-07 00:00:00.0(Timestamp), 男(String), 北京市(Stri
DEBUG [main] - <==    Updates: 1
DEBUG [main] - Committing JDBC Connection [com.mysql.jdbc.JDBC4Connection@24a35978]
DEBUG [main] - ==>  Preparing: select * from user where id=?
DEBUG [main] - ==> Parameters: 10(Integer)
DEBUG [main] - <==      Total: 1
User [id=10, username=张三, birthday=Sun Jan 07 00:00:00 CST 2018, sex=男, address=北京市]
```

2、二级缓存

2.1 原理

下图是多个 sqlSession 请求 UserMapper 的二级缓存图解。



二级缓存区域是根据 mapper 的 namespace 划分的，相同 namespace 的 mapper 查询数据放在同一个区域，如果使用 mapper 代理方法每个 mapper 的 namespace 都不同，此时可以理解为二级缓存区域是根据 mapper 划分。

每次查询会先从缓存区域找，如果找不到从数据库查询，查询到数据将数据写入缓存。Mybatis 内部存储缓存使用一个 HashMap，key 为 hashCode+sqlId+Sql 语句。value 为从查询出来映射生成的 java 对象

sqlSession 执行 insert、update、delete 等操作 commit 提交后会清空缓存区域。

2.2 开启二级缓存

在核心配置文件 SqlMapConfig.xml 中加入：<setting name="cacheEnabled" value="true"/>

	描述	允许值	默认值
cacheEnabled	对在此配置文件下的所有 cache 进行全局性开/关设置。	true false	true

```
<!-- 开启二级缓存 -->
<setting name="cacheEnabled" value="true"/>
```

此外，还需要在你的 Mapper 映射文件中添加一行：<cache />，表示此 mapper 开启

二级缓存。

```
<!-- 开启本mapper的namespace的二级缓存 -->
<cache />
```

2.3 实现序列化

二级缓存需要查询结果映射的 pojo 对象实现 java.io.Serializable 接口实现序列化和反序列化操作，注意如果存在父类、成员 pojo 都需要实现序列化接口。

```
public class Orders implements Serializable

public class User implements Serializable

....
```

```
public class User implements Serializable {
    private static final long serialVersionUID = 1L;
```

为了将缓存数据取出，执行反序列化操作，因为二级缓存数据存储介质可能是多种多样的，并一定就是在内存中。

2.4 测试 1

```
@Test
// 二级缓存测试
public void testCache2() throws Exception {
    // 获取Session
    SqlSession sqlSession1 = sqlSessionFactory.openSession();
    SqlSession sqlSession2 = sqlSessionFactory.openSession();

    // 通过sqlSession得到Mapper接口的代理对象
    UserMapper userMapper1 = sqlSession1.getMapper(UserMapper.class);
    UserMapper userMapper2 = sqlSession2.getMapper(UserMapper.class);

    // 第一次发起请求查询用户id为10的用户
    User user1 = userMapper1.selectUserById(10);
    System.out.println(user1);

    // 关闭sqlSession1，这是因为只有当sqlSession1关闭了，数据才写入到二级缓存中
    sqlSession1.close();

    // 第二次发起请求查询用户id为10的用户
    User user2 = userMapper2.selectUserById(10);
    System.out.println(user2);
```

```
    sqlSession2.close();
}
```

从结果日志可以看出，当有查询请求时，首先从缓存数据中查找，若没有再发送 sql 请求给数据库。第二个 userMapper2 查询同 userMapper1 查询的用户一样时，直接从缓存数据中返回。

```
DEBUG [main] - Cache Hit Ratio [com.zxt.mybatis.mapper.UserMapper]: 0.0
DEBUG [main] - Opening JDBC Connection
DEBUG [main] - Created connection 1873859565.
DEBUG [main] - Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection
DEBUG [main] - ==>  Preparing: select * from user where id=?
DEBUG [main] - ==> Parameters: 10(Integer)
DEBUG [main] - <==      Total: 1
User [id=10, username=张三, birthday=Sun Jan 07 00:00:00 CST 2018, sex=男, address=北京市]
DEBUG [main] - Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connectio
DEBUG [main] - Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@6fb0d3ed]
DEBUG [main] - Returned connection 1873859565 to pool.
DEBUG [main] - Cache Hit Ratio [com.zxt.mybatis.mapper.UserMapper]: 0.5
User [id=10, username=张三, birthday=Sun Jan 07 00:00:00 CST 2018, sex=男, address=北京市]
```

缓存命中率，一开始是0

2.5 测试 2

```
@Test
// 二级缓存测试
public void testCache2() throws Exception {
    // 获取Session
    SqlSession sqlSession1 = sqlSessionFactory.openSession();
    SqlSession sqlSession2 = sqlSessionFactory.openSession();
    SqlSession sqlSession3 = sqlSessionFactory.openSession();

    // 通过sqlSession得到Mapper接口的代理对象
    UserMapper userMapper1 = sqlSession1.getMapper(UserMapper.class);
    UserMapper userMapper2 = sqlSession2.getMapper(UserMapper.class);
    UserMapper userMapper3 = sqlSession3.getMapper(UserMapper.class);

    // 第一次发起请求查询用户id为10的用户
    User user1 = userMapper1.selectUserById(10);
    System.out.println(user1);

    // 关闭sqlSession1，这是因为只有当sqlSession1关闭了，数据才写入到二级缓存中
    sqlSession1.close();

    User user = userMapper3.selectUserById(10);
    user.setUsername("张三三");
    userMapper3.updateUser(user);
    // 执行提交操作，将会清空 UserMaper下的二级缓存
    sqlSession3.commit();
    sqlSession3.close();
```

```
    // 第二次发起请求查询用户id为10的用户
    User user2 = userMapper2.selectUserById(10);
    System.out.println(user2);

    sqlSession2.close();
}
```

```
DEBUG [main] - Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@6fb0
DEBUG [main] - ==>  Preparing: select * from user where id=?
DEBUG [main] - ==> Parameters: 10(Integer)
DEBUG [main] - <==      Total: 1
User [id=10, username=张三, birthday=Sun Jan 07 00:00:00 CST 2018, sex=男, address=北京市]
DEBUG [main] - Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@6fb
DEBUG [main] - Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@6fb0d3ed]
DEBUG [main] - Returned connection 1873859565 to pool.
DEBUG [main] - Cache Hit Ratio [com.zxt.mybatis.mapper.UserMapper]: 0.5
DEBUG [main] - Opening JDBC Connection
DEBUG [main] - Checked out connection 1873859565 from pool.
DEBUG [main] - Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@6fb0
DEBUG [main] - ==>  Preparing: update user set username=?, birthday=?, sex=?, address=? where id=?
DEBUG [main] - ==> Parameters: 张三三(String), 2018-01-07 00:00:00.0(Timestamp), 男(String), 北京市(Str
DEBUG [main] - <==    Updates: 1
DEBUG [main] - Committing JDBC Connection [com.mysql.jdbc.JDBC4Connection@6fb0d3ed]
DEBUG [main] - Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@6fb
DEBUG [main] - Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@6fb0d3ed]
DEBUG [main] - Returned connection 1873859565 to pool.
DEBUG [main] - Cache Hit Ratio [com.zxt.mybatis.mapper.UserMapper]: 0.3333333333333333
DEBUG [main] - Opening JDBC Connection
DEBUG [main] - Checked out connection 1873859565 from pool.
DEBUG [main] - Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@6fb0
DEBUG [main] - ==>  Preparing: select * from user where id=?
DEBUG [main] - ==> Parameters: 10(Integer)
DEBUG [main] - <==      Total: 1
User [id=10, username=张三三, birthday=Sun Jan 07 00:00:00 CST 2018, sex=男, address=北京市]
```

2.6 禁用二级缓存

在 statement 中设置 useCache=false 可以禁用当前 select 语句的二级缓存，即每次查询都会发出 sql 去查询，默认情况是 true，即该 sql 使用二级缓存。

```
<select    id="selectOrderListResultMap"    resultMap="ordersUserMap"
useCache="false">
```

2.7 刷新缓存

在 mapper 的同一个 namespace 中，如果有其它 insert、update、delete 操作数据后需要刷新缓存，如果不执行刷新缓存会出现脏读。

设置 statement 配置中的 flushCache="true" 属性，默认情况下为 true 即刷新缓存，如果改成 false 则不会刷新。使用缓存时如果手动修改数据库表中的查询数据会出现脏读。

如下：

```
<insert      id="insertUser"      parameterType="con.zxt.domain.User"
flushCache="true">
```

2.8 Mybatis Cache 参数

flushInterval（刷新间隔）可以被设置为任意的正整数，而且它们代表一个合理的毫秒形式的时间段。默认情况是不设置，也就是没有刷新间隔，缓存仅仅调用语句时刷新。size（引用数目）可以被设置为任意正整数，要记住你缓存的对象数目和你运行环境的可用内存资源数目。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。这提供了很重要的性能优势。可读写的缓存会返回缓存对象的拷贝（通过序列化）。这会慢一些，但是安全，因此默认是 false。

如下例子：<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，并每隔 60 秒刷新，存数结果对象或列表的 512 个引用，而且返回的对象被认为是只读的，因此在不同线程中的调用者之间修改它们会导致冲突。可用的收回策略有，默认的是 LRU：

1、LRU – 最近最少使用的：移除最长时间不被使用的对象。

2、FIFO – 先进先出：按对象进入缓存的顺序来移除它们。

3、SOFT – 软引用：移除基于垃圾回收器状态和软引用规则的对象。

4、WEAK – 弱引用：更积极地移除基于垃圾收集器状态和弱引用规则的对象。

3、mybatis 整合 ehcache

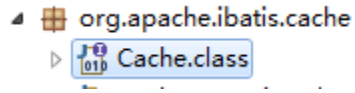
ehCache 是一个纯 Java 的进程内缓存框架，是一种广泛使用的开源 Java **分布式缓存**，具有快速、精干等特点，是 Hibernate 中默认的 CacheProvider。

不使用分布式缓存，缓存数据在各自的服务器中存储，不方便系统开发，所以要使用分布式缓存对缓存数据进行集中式管理。

而 mybatis 本身无法实现分布式缓存，需要和其他的分布式缓存进行整合。整合方法：mybatis 提供了一个 Cache 接口，如果要实现自己的缓存逻辑，只需要实现 Cache 接口即可。

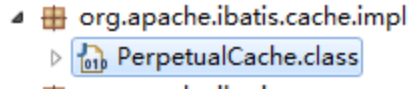
3.1 mybatis 整合 ehcache 原理

mybatis 提供二级缓存 Cache 接口，如下：




```java
public interface Cache {

  /**
   * @return The identifier of this cache
   */
  String getId();

  /**
   * @param key Can be any object but usually it is a {@link CacheKey}
   * @param value The result of a select.
   */
  void putObject(Object key, Object value);

  /**
   * @param key The key
   * @return The object stored in the cache.
   */
  Object getObject(Object key);
```

它的默认实现类：

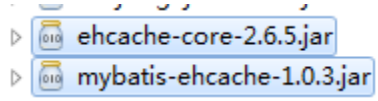



```java
public class PerpetualCache implements Cache {

  private final String id;

  private Map<Object, Object> cache = new HashMap<Object, Object>();

  public PerpetualCache(String id) {
    this.id = id;
  }

  @Override
  public String getId() {
    return id;
  }
```

通过实现 Cache 接口可以实现 mybatis 缓存数据通过其它缓存数据库整合，mybatis 的特长是 sql 操作，缓存数据的管理不是 mybatis 的特长，为了提高缓存的性能将 mybatis 和第三方的缓存数据库整合，比如 ehcache、memcache、redis 等。

3.2 mybatis 整合 ehcache 方法

1、引入 jar 包



2、配置 cache 中 type 为 ehcache 实现 cache 接口的类型。

```
<!-- 开启本mapper的namespace的二级缓存 -->
<!-- type: 指定Cache接口的实现类的类型，mybatis默认使用PerpetualCache
     要和ehcache整合，需要配置type为ehcache实现的cache接口的类型 -->
<cache type="org.mybatis.caches.ehcache.EhcacheCache"/>
```

3、classpath 下添加配置文件：ehcache.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="../config/ehcache.xsd">
    <!-- <diskStore path="F:\develop\ehcache" /> -->
    <defaultCache
        maxElementsInMemory="1000"
        maxElementsOnDisk="10000000"
        eternal="false"
        overflowToDisk="false"
        timeToIdleSeconds="120"
        timeToLiveSeconds="120"
        diskExpiryThreadIntervalSeconds="120"
        memoryStoreEvictionPolicy="LRU">
    </defaultCache>
</ehcache>
```

属性说明：

diskStore：指定数据在磁盘中的存储位置。

defaultCache：当借助 CacheManager.add("demoCache")创建 Cache 时，EhCache 便会采用<defalutCache/>指定的的管理策略

以下属性是必须的：

maxElementsInMemory - 在内存中缓存的 element 的最大数目

maxElementsOnDisk - 在磁盘上缓存的 element 的最大数目，若是 0 表示无穷大

eternal - 设定缓存的 elements 是否永远不过期。如果为 true，则缓存的数据始终有效，如果为 false 那么还要根据 timeToIdleSeconds，timeToLiveSeconds 判断

overflowToDisk - 设定当内存缓存溢出的时候是否将过期的 element 缓存到磁盘上

以下属性是可选的：

timeToIdleSeconds - 当缓存在 EhCache 中的数据前后两次访问的时间超过 timeToIdleSeconds 的属性取值时，这些数据便会删除，默认值是 0,也就是可闲置时间无穷大

timeToLiveSeconds - 缓存 element 的有效生命期，默认是 0.,也就是 element 存

活时间无穷大

diskSpoolBufferSizeMB 这个参数设置 DiskStore(磁盘缓存)的缓存区大小.默认是 30MB.每个 Cache 都应该有自己的一个缓冲区.

diskPersistent - 在 VM 重启的时候是否启用磁盘保存 EhCache 中的数据，默认是 false。

diskExpiryThreadIntervalSeconds - 磁盘缓存的清理线程运行间隔，默认是 120 秒。每个 120s，相应的线程会进行一次 EhCache 中数据的清理工作

memoryStoreEvictionPolicy - 当内存缓存达到最大，有新的 element 加入的时候，移除缓存中 element 的策略。默认是 LRU（最近最少使用），可选的有 LFU（最不常使用）和 FIFO（先进先出）。

4、还可以根据需求调整如下参数：

```
<cache type="org.mybatis.caches.ehcache.EhcacheCache" >
    <property name="timeToIdleSeconds" value="3600"/>
    <property name="timeToLiveSeconds" value="3600"/>
    <!-- 同ehcache参数maxElementsInMemory -->
    <property name="maxEntriesLocalHeap" value="1000"/>
    <!-- 同ehcache参数maxElementsOnDisk -->
    <property name="maxEntriesLocalDisk" value="10000000"/>
    <property name="memoryStoreEvictionPolicy" value="LRU"/>
</cache>
```

4、二级缓存总结

4.1 二级缓存应用场景

对于访问多的查询请求且用户对查询结果实时性要求不高，此时可采用 mybatis 二级缓存技术降低数据库访问量，提高访问速度，业务场景比如：耗时较高的统计分析 sql、电话账单查询 sql 等。

实现方法如下：通过设置刷新间隔时间，由 mybatis 每隔一段时间自动清空缓存，根据数据变化频率设置缓存刷新间隔 flushInterval，比如设置为 30 分钟、60 分钟、24 小时等，根据需求而定。

4.2 二级缓存局限性

mybatis 二级缓存对细粒度的数据级别的缓存实现不好，比如如下需求：对商品信息进行缓存，由于商品信息查询访问量大，但是要求用户每次都能查询最新的商品信息，此时

如果使用 mybatis 的二级缓存就无法实现当一个商品变化时只刷新该商品的缓存信息而不刷新其它商品的信息，因为 mybaits 的二级缓存区域以 mapper 为单位划分，当一个商品信息变化会将所有商品信息的缓存数据全部清空。解决此类问题需要在业务层根据需求对数据有针对性缓存。

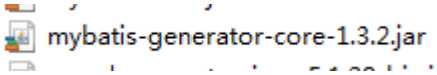
十一、Mybatis 逆向工程

1、什么是逆向工程

开发 mybatis 项目，程序员需要自己编写 sql 代码，mybatis 官方提供逆向工程，可以针对单表自动生成 mybatis 执行所需要的代码（mapper.java、mapper.xml、po）。

在实际的企业开发中常用的方式是：由数据库表生成 java 代码。

2、逆向工程所需 jar 包



3、mapper 生成配置文件

在项目目录中添加 generatorConfig.xml 配置文件，在里面配置 mapper 生成的详细信息，注意改下几点：

1、添加要生成的数据库表；

2、po 文件所在包路径；

3、mapper 文件所在包路径。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE generatorConfiguration
  PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"
  "http://mybatis.org/dtd/mybatis-generator-config_1_0.dtd">

<generatorConfiguration>
    <context id="testTables" targetRuntime="MyBatis3">
        <commentGenerator>
            <!-- 是否去除自动生成的注释 true：是 ： false:否 -->
            <property name="suppressAllComments" value="true" />
        </commentGenerator>

        <!--数据库连接的信息：驱动类、连接地址、用户名、密码 -->
        <jdbcConnection driverClass="com.mysql.jdbc.Driver"
        connectionURL="jdbc:mysql://localhost:3306/orders?characterEncoding=utf-8"
            userId="root" password="root">
        </jdbcConnection>
```

```
        <!-- 默认false，把JDBC DECIMAL 和 NUMERIC 类型解析为 Integer，为 true时把JDBC
DECIMAL 和 NUMERIC 类型解析为java.math.BigDecimal -->
        <javaTypeResolver>
            <property name="forceBigDecimals" value="false" />
        </javaTypeResolver>

        <!-- targetProject:生成PO类的位置 -->
        <javaModelGenerator targetPackage="com.zxt.ssm.po" targetProject=".\src">
            <!-- enableSubPackages:是否让schema作为包的后缀 -->
            <property name="enableSubPackages" value="false" />
            <!-- 从数据库返回的值被清理前后的空格 -->
            <property name="trimStrings" value="true" />
        </javaModelGenerator>

       <!-- targetProject:mapper映射文件生成的位置 -->
        <sqlMapGenerator targetPackage="com.zxt.ssm.mapper" targetProject=".\src">
            <!-- enableSubPackages:是否让schema作为包的后缀 -->
            <property name="enableSubPackages" value="false" />
        </sqlMapGenerator>

        <!-- targetPackage：mapper接口生成的位置 -->
        <javaClientGenerator type="XMLMAPPER" targetPackage="com.zxt.ssm.mapper"
targetProject=".\src">
            <!-- enableSubPackages:是否让schema作为包的后缀 -->
            <property name="enableSubPackages" value="false" />
        </javaClientGenerator>

        <!-- 指定数据库表 -->
        <table tableName="items"></table>
        <table tableName="orders"></table>
        <table tableName="orderdetail"></table>
        <table tableName="user"></table>
    </context>

</generatorConfiguration>
```

4、使用 java 类生成 mapper 文件

```
public class GeneratorSqlmap {
    public void generator() throws Exception {

        List<String> warnings = new ArrayList<String>();
        boolean overwrite = true;
        // 指定 逆向工程配置文件
```

```
        File configFile = new File("generatorConfig.xml");
        ConfigurationParser cp = new ConfigurationParser(warnings);
        Configuration config = cp.parseConfiguration(configFile);
        DefaultShellCallback callback = new DefaultShellCallback(overwrite);
        MyBatisGenerator myBatisGenerator = new MyBatisGenerator(config, callback,
warnings);
        myBatisGenerator.generate(null);
    }

    public static void main(String[] args) throws Exception {
        try {
            GeneratorSqlmap generatorSqlmap = new GeneratorSqlmap();
            generatorSqlmap.generator();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

5、生成的源代码的使用方式

拷贝生成的 mapper 文件到工程中指定的目录中，Mapper.xml 的文件拷贝至 mapper 目录内，Mapper.java 的文件拷贝至 mapper 目录内；

注意：mapper xml 文件和 mapper.java 文件在一个目录内且文件名相同。

6、Mapper 接口测试

学会使用 mapper 自动生成的增、删、改、查方法。

```
public interface UserMapper {
    int countByExample(UserExample example);
     // 删除符合条件的记录
    int deleteByExample(UserExample example);
     // 根据主键删除
    int deleteByPrimaryKey(Integer id);
     // 插入对象所有字段
    int insert(User record);
     // 插入对象不为空的字段
    int insertSelective(User record);
     // 自定义查询条件查询结果集
    List<User> selectByExample(UserExample example);
     // 根据主键查询
    User selectByPrimaryKey(Integer id);
```

```
    // 根据主键将对象中不为空的值更新至数据库
    int updateByExampleSelective(@Param("record") User record, @Param("example")
UserExample example);
    // 更新
    int updateByExample(@Param("record") User record, @Param("example") UserExample
example);
    // 根据主键将对象中不为空的值更新至数据库
    int updateByPrimaryKeySelective(User record);
    // 根据主键将对象中不为空的值更新至数据库
    int updateByPrimaryKey(User record);
}
```

测试代码：

```
public class UserMapperTest {
    private ApplicationContext applicationContext;
    private UserMapper userMapper;

    @Before
    // 在setUp方法里面得到spring的容器
    public void setUp() throws Exception {
        applicationContext = new
ClassPathXmlApplicationContext("classpath:spring/applicationContext.xml");

        userMapper = (UserMapper) applicationContext.getBean("userMapper");
    }

    @Test
    public void testCountByExample() {
        fail("Not yet implemented");
    }

    @Test
    public void testDeleteByExample() {
        fail("Not yet implemented");
    }

    @Test
    public void testDeleteByPrimaryKey() {
        userMapper.deleteByPrimaryKey(43);
    }

    @Test
    public void testInsert() {
        // 构造要插入的对象
        User user = new User();
```

```
        user.setUsername("关云长");
        user.setSex("男");
        user.setBirthday(new Date());
        user.setAddress("湖北荆州");

        userMapper.insert(user);
    }

    @Test
    public void testInsertSelective() {
        fail("Not yet implemented");
    }

    @Test
    // 自定义条件查询
    public void testSelectByExample() {
        UserExample userExample = new UserExample();

        // 通过Criteria构造查询条件
        UserExample.Criteria criteria = userExample.createCriteria();
        criteria.andUsernameEqualTo("陈小明");

        // 可能返回多条记录
        List<User> list = userMapper.selectByExample(userExample);

        System.out.println(list);
    }

    @Test
    public void testSelectByPrimaryKey() {
        User user = userMapper.selectByPrimaryKey(10);
        System.out.println(user);
    }

    @Test
    public void testUpdateByExampleSelective() {
        fail("Not yet implemented");
    }

    @Test
    public void testUpdateByExample() {
        fail("Not yet implemented");
    }
```

```
    @Test
    // updateByPrimaryKeySelective  只更新参数中不为空的字段
    public void testUpdateByPrimaryKeySelective() {
        // 该方法更新不需要先查询，只需要更新的用户的id在原数据表中存在即可
        User user = new User();
        user.setId(10);
        user.setUsername("张三三");

        userMapper.updateByPrimaryKeySelective(user);
    }

    @Test
    // updateByPrimaryKey 对所有字段都更新
    public void testUpdateByPrimaryKey() {
        // 先查询数据，再更新
        User user = userMapper.selectByPrimaryKey(10);
        // 将user姓名改成  张三
        user.setUsername("张三");
        userMapper.updateByPrimaryKey(user);
    }
}
```

7、逆向工程注意事项

Mapper 文件内容不覆盖而是追加

XXXMapper.xml 文件已经存在时，如果进行重新生成则 mapper.xml 文件内容不被覆盖而是进行内容追加，结果导致 mybatis 解析失败。

解决方法：删除原来已经生成的 mapper xml 文件再进行生成。Mybatis 自动生成的 po 及 mapper.java 文件，不是追加而是直接覆盖没有此问题。

Table schema 问题

下边是针对 oracle 数据库表生成代码的 schema 问题：

Schema 即数据库模式，oracle 中一个用户对应一个 schema，可以理解为用户就是 schema。

当 Oralce 数据库存在多个 schema 可以访问相同的表名时，使用 mybatis 生成该表的 mapper.xml 将会出现 mapper.xml 内容重复的问题，结果导致 mybatis 解析错误。

解决方法：在 table 中填写 schema，如下：

<table schema="XXXX" tableName=" " >

XXXX 即为一个 schema 的名称，生成后将 mapper.xml 的 schema 前缀批量去掉，如果不去掉当 oracle 用户变更了 sql 语句将查询失败。快捷操作方式：mapper.xml 文件中批量替换：“from XXXX.”为空。

Oracle 查询对象的 schema 可从 dba_objects 中查询，如下：

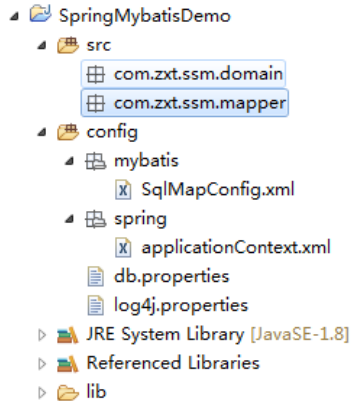
```
select * from dba_objects
```

十二、Spring 与 Mybatis 的整合开发

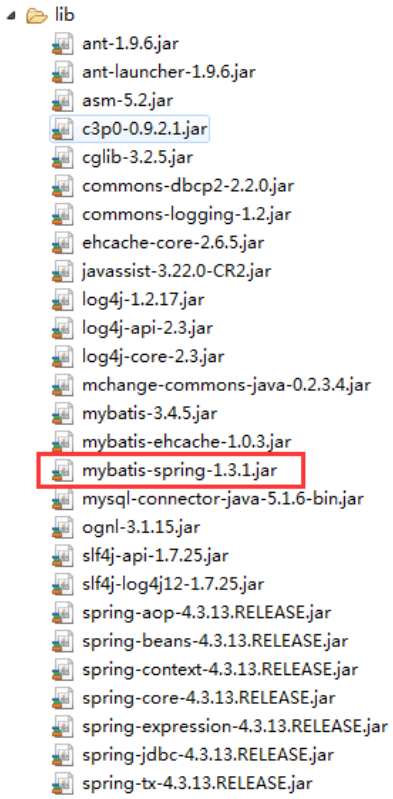
实现 mybatis 与 spring 进行整合，通过 spring 管理 SqlSessionFactory、mapper 接口。

1、整合环境

1、创建一个 java 项目，其大致的结构如下所示：



2、项目所需要的 jar 包，包括 mybatis、spring 以及 mybatis 与 spring 整合的 jar 包，早期 mybatis 与 spring 整合的整合包由 spring 官方提供，后来由 mybatis 提供。



2、SqlSessionFactory

在 applicationContext.xml 中配置 SqlSessionFactory 以及数据源，SqlSessionFactory 在 mybatis 与 spring 整合的包中。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx.xsd">
     <!-- 加载配置文件 -->
     <context:property-placeholder location="classpath:db.properties"/>

     <!-- 数据库连接池 -->
     <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
destroy-method="close">
         <property name="driverClassName" value="${jdbc.driver}"/>
         <property name="url" value="${jdbc.url}"/>
         <property name="username" value="${jdbc.username}"/>
         <property name="password" value="${jdbc.password}"/>
         <property name="maxActive" value="10"/>
         <property name="maxIdle" value="5"/>
     </bean>

     <!-- mapper配置 -->
     <!-- SqlSessionFactory -->
     <!-- 让spring管理sqlsessionfactory 使用mybatis和spring整合包中的sqlSession工厂 -->
     <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
         <!-- 数据库连接池 -->
         <property name="dataSource" ref="dataSource" />
         <!-- 加载mybatis的全局配置文件 -->
         <property name="configLocation" value="classpath:mybatis/SqlMapConfig.xml" />
     </bean>

</beans>
```

3、Mybatis 配置文件

Mybatis 与 spring 整合之后，环境配置信息都由 spring 容器管理，而 mybatis 中主要配置的就是加载 mybatis 的配置文件 mapper.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>

    <!-- 加载映射文件 -->
    <mappers>
        <mapper resource="sqlmap/User.xml" />

        <!-- 使用mapper接口类路径加载mapper映射文件 -->
        <!-- 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.zxt.ssm.mapper"/>
    </mappers>

</configuration>
```

4、Dao 的开发

4.1 User.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">
    <!-- 可以在映射文件中配置很多的sql语句 -->
    <select id="findUserById" parameterType="int"
resultType="com.zxt.ssm.domain.User">
        select * from user where id=#{id}
    </select>

    <select id="findUserByName" parameterType="String"
resultType="com.zxt.ssm.domain.User">
        select * from user where username like '%${value}%'
    </select>

    <insert id="insertUser" parameterType="com.zxt.ssm.domain.User">
        <!-- selectKey将主键返回，需要再返回 -->
```

```
        <selectKey keyProperty="id" order="AFTER" resultType="java.lang.Integer">
            select LAST_INSERT_ID()
        </selectKey>
        insert into user(username, birthday, sex, address) values(#{username},
#{birthday}, #{sex}, #{address})
    </insert>

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

    <!-- 根据id更新用户 -->
    <update id="updateUser" parameterType="com.zxt.ssm.domain.User">
        update user set username=#{username}, birthday=#{birthday}, sex=#{sex},
address=#{address}
        where id=#{id}
    </update>

</mapper>
```

4.2 Dao 接口

```
public interface UserDao {
    // 根据用户id查询用户信息
    public User selectUserById(int id) throws Exception;

    // 根据用户名模糊查询用户信息
    public List<User> selectUserByName(String name) throws Exception;

    // 添加用户信息
    public void insertUser(User user) throws Exception;

    // 删除用户信息（根据用户id删除）
    public void deleteUser(int id) throws Exception;

    // 更新用户信息
    public void updateUser(User user) throws Exception;
}
```

4.3 Dao 实现类

Dao 的实现类中要生成 SqlSession，因此需要注入 SqlSessionFactory 对象，在使用 spring 之后，可以让 Dao 的实现类继承 SqlSessionDaoSupport，因为该类中包含有 SqlSessionFactory 的变量和 set 方法，我们在使用的时候，只需要在 spring 中对

UserDao 的实现类中配置该变量的值即可。

```
// SqlSessionDaoSupport类中有SqlSessionFactory变量，及其set方法
public class UserDaoImpl extends SqlSessionDaoSupport implements UserDao {

    @Override
    public User selectUserById(int id) throws Exception {
        SqlSession sqlSession = this.getSqlSession();

        User user = sqlSession.selectOne("test.findUserById", id);

        // 关闭SqlSession的操作也不需要了，因为资源都由spring容器管理了
        return user;
    }

    @Override
    public List<User> selectUserByName(String name) throws Exception {
        SqlSession sqlSession = this.getSqlSession();

        List<User> list = sqlSession.selectList("test.findUserByName", name);

        return list;
    }

    @Override
    public void insertUser(User user) throws Exception {
        SqlSession sqlSession = this.getSqlSession();

        sqlSession.insert("test.insertUser", user);

        // 提交事务
        sqlSession.commit();
    }

    @Override
    public void deleteUser(int id) throws Exception {
        SqlSession sqlSession = this.getSqlSession();

        sqlSession.insert("test.deleteUser", id);

        // 提交事务
        sqlSession.commit();
    }

    @Override
```

```
    public void updateUser(User user) throws Exception {
        SqlSession sqlSession = this.getSqlSession();

        sqlSession.insert("test.updateUser", user);

        // 提交事务
        sqlSession.commit();
    }
}
```

4.4 Spring 配置

```
<!-- 原始dao的配置 -->
<bean id="userDao" class="com.zxt.ssm.dao.UserDaoImpl">
    <property name="sqlSessionFactory" ref="sqlSessionFactory"></property>
</bean>
```

4.5 测试类编写

```
public class UserDaoImplTest {
    private ApplicationContext applicationContext;
    private UserDao userDao;

    @Before
    // 在setUp方法里面得到spring的容器
    public void setUp() throws Exception {
        applicationContext = new
ClassPathXmlApplicationContext("classpath:spring/applicationContext.xml");

        userDao = (UserDao) applicationContext.getBean("userDao");
    }

    @Test
    public void testSelectUserById() throws Exception {
        User user = userDao.selectUserById(10);
        System.out.println(user);
    }

    @Test
    public void testSelectUserByName() throws Exception {
        List<User> list = userDao.selectUserByName("张");
        System.out.println(list);
    }

    @Test
    public void testInsertUser() throws Exception {
```

```
        // 创建一个user对象，用于插入数据库
        User user = new User();
        user.setUsername("柏小青");
        user.setSex("女");
        user.setBirthday(new Date());
        user.setAddress("甘肃兰州");

        userDao.insertUser(user);
    }

    @Test
    public void testDeleteUser() throws Exception {
        userDao.deleteUser(39);
    }

    @Test
    public void testUpdateUser() throws Exception {
        // 创建一个user对象，用于插入数据库
        User user = new User();
        // 该id是已经存在于数据库的记录的id
        user.setId(32);
        user.setUsername("李小四");
        user.setSex("男");
        user.setBirthday(new Date());
        user.setAddress("陕西西安");

        userDao.updateUser(user);
    }
}
```

5、Mapper 开发方法

1、UserMapper.xml 同上，内容不变。

2、UserMapper.java 接口

```
public interface UserMapper {
    // 根据用户id查询用户信息
    public User selectUserById(int id) throws Exception;

    // 根据用户名模糊查询用户信息
    public List<User> selectUserByName(String name) throws Exception;

    // 添加用户信息
    public void insertUser(User user) throws Exception;
```

```
    // 删除用户信息（根据用户id删除）
    public void deleteUser(int id) throws Exception;

    // 更新用户信息
    public void updateUser(User user) throws Exception;
}
```

3、Mapper 接口不需要编写实现类，因此需要在 spring 容器中配置 Mapper 接口的代理类实现。

```
<!-- mapper配置 -->
<!-- MapperFactoryBean：根据接口生成代理对象 -->
<bean id="userMapper" class="org.mybatis.spring.mapper.MapperFactoryBean">
    <!-- 根据接口生成代理对象，所以得指定接口（通过mapperInterface） -->
    <property name="mapperInterface"
value="com.zxt.ssm.mapper.UserMapper"></property>
    <!-- 同样需要配置sqlSessionFactory -->
    <property name="sqlSessionFactory" ref="sqlSessionFactory"></property>
</bean>
```

4、测试代码

```
public class UserMapperTest {
    private ApplicationContext applicationContext;
    private UserMapper userMapper;
    @Before
    // 在setUp方法里面得到spring的容器
    public void setUp() throws Exception {
        applicationContext = new
ClassPathXmlApplicationContext("classpath:spring/applicationContext.xml");

        userMapper = (UserMapper) applicationContext.getBean("userMapper");
    }

    @Test
    public void testSelectUserById() throws Exception {
        User user = userMapper.selectUserById(10);
        System.out.println(user);
    }
    @Test
    public void testSelectUserByName() throws Exception {
        List<User> list = userMapper.selectUserByName("小");
        System.out.println(list);
    }

    @Test
```

```
    public void testInsertUser() throws Exception {
        // 创建一个user对象，用于插入数据库
        User user = new User();
        user.setUsername("赵之龙");
        user.setSex("男");
        user.setBirthday(new Date());
        user.setAddress("天津市");
        userMapper.insertUser(user);
    }

    @Test
    public void testDeleteUser() throws Exception {
        userMapper.deleteUser(41);
    }

    @Test
    public void testUpdateUser() throws Exception {
        // 创建一个user对象，用于插入数据库
        User user = new User();
        // 该id是已经存在于数据库的记录的id
        user.setId(40);
        user.setUsername("赵子龙");
        user.setSex("男");
        user.setBirthday(new Date());
        user.setAddress("天津市");

        userMapper.updateUser(user);
    }
}
```

5、上述方法有一个缺点，那就是每编写一个 Mapper 都需要进行配置，因此可以使用 Mapper 的扫描器，只需要在 spring 配置文件中定义一个 mapper 扫描器，自动扫描包中的 mapper 接口生成代理对象。

```
<!-- Mapper批量扫描，在Mapper的包中扫描出mapper接口，自动创建代理对象，并且在spring容器中注册 -->
<!-- 扫描之后，包下的Mapper的代理类的bean对象的id和Mapper接口名一样，首字母小写 -->
<!-- 这里同样需要：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。 -->
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
    <!-- 指定扫描的包名，如果有过个包，不能用通配符，可以用逗号或分号分隔定义多个包 -->
    <property name="basePackage" value="com.zxt.ssm.mapper"></property>
    <!-- 这里配置SqlSessionFactory需要使用sqlSessionFactoryBeanName而不是sqlSessionFactory -->
    <!-- 由于使用sqlSessionFactory之后，扫描会先于上面配置文件的加载，这样会出现错误 -->
    <property name="sqlSessionFactoryBeanName" value="sqlSessionFactory"></property>
```

```
</bean>
```

如果将 mapper.xml 和 mapper 接口的名称保持一致且放在一个目录，则不用在 sqlMapConfig.xml 中进行配置。