传智播客 —高级软件人才实作培训专家！
www.itcast.cn

Mybatis

框架课程

讲师：传智.燕青

1 Mybatis 入门

1.1 单独使用 jdbc 编程问题总结

1.1.1 jdbc 程序

```
Public static void main(String[] args) {
            Connection connection = null;
            PreparedStatement preparedStatement = null;
            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/mybatis?charac
terEncoding=utf-8", "root", "mysql");
                //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) {
```

```
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
            if(preparedStatement!=null){
                try {
                    preparedStatement.close();
                } catch (SQLException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
            if(connection!=null){
                try {
                    connection.close();
                } catch (SQLException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }

        }

    }
```

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。

1.1.2 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)

1.1.3 jdbc 问题总结如下：

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

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

3、 使用 preparedStatement 向占有位符号传参数存在硬编码，因为 sql 语句的 where 条件不一定，可能多也可能少，修改 sql 还要修改代码，系统不易维护。

4、 对结果集解析存在硬编码（查询列名），sql 变化导致解析代码变化，系统不易维护，如果能将数据库记录封装成 pojo 对象解析比较方便。

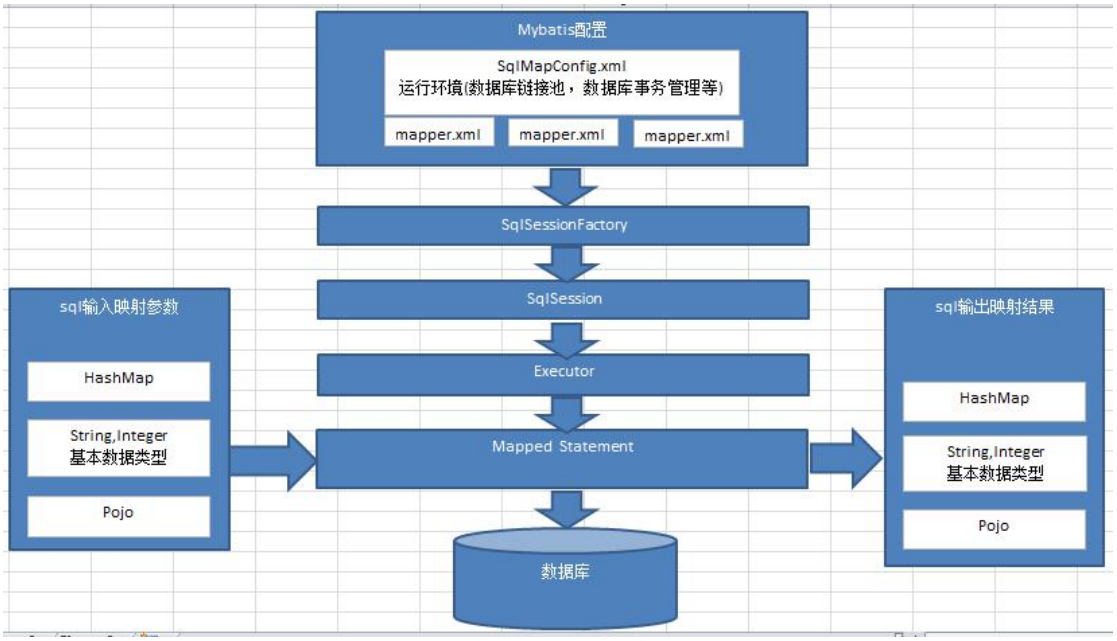
1.2 MyBatis 介绍

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。

1.3 Mybatis 架构



1、 mybatis 配置

SqlMapConfig.xml，此文件作为 mybatis 的全局配置文件，配置了 mybatis 的运行环境等信息。mapper.xml 文件即 sql 映射文件，文件中配置了操作数据库的 sql 语句。此文件需要在 SqlMapConfig.xml 中加载。

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

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

4、 mybatis 底层自定义了 Executor 执行器接口操作数据库，Executor 接口有两个实现，一个是基本执行器、一个是缓存执行器。

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

6、 Mapped Statement 对 sql 执行输入参数进行定义，包括 HashMap、基本类型、pojo，Executor 通过 Mapped Statement 在执行 sql 前将输入的 java 对象映射至 sql 中，输入参数映射就是 jdbc 编程中对 preparedStatement 设置参数。

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

1.4 mybatis 下载

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

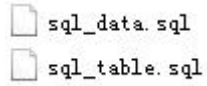
mybatis-3.2.7.jar----mybatis 的核心包

lib----mybatis 的依赖包

mybatis-3.2.7.pdf----mybatis 使用手册

1.5 创建 mysql 数据库

先导入 sql_table.sql，再导入 sql_data.sql 脚本：



如下：

1.6 Mybatis 入门程序

1.6.1 需求

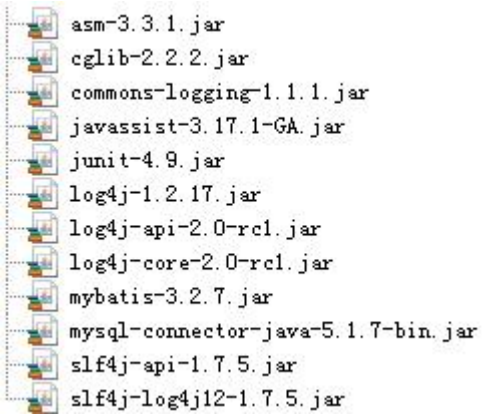
实现以下功能：
根据用户 id 查询一个用户信息
根据用户名称模糊查询用户信息列表
添加用户
更新用户
删除用户

1.6.2 第一步：创建 java 工程

使用 eclipse 创建 java 工程，jdk 使用 1.7.0_72。

1.6.3 第二步：加入 jar 包

加入 mybatis 核心包、依赖包、数据驱动包。



1.6.4 第三步：log4j.properties

在 classpath 下创建 log4j.properties 如下：

```
# 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
```

mybatis 默认使用 log4j 作为输出日志信息。

1.6.5 第四步：SqlMapConfig.xml

在 classpath 下创建 SqlMapConfig.xml，如下：

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

```
value="com.mysql.jdbc.Driver" />
            <property name="url"
value="jdbc:mysql://localhost:3306/mybatis?characterEncoding=u
tf-8" />
            <property name="username" value="root" />
            <property name="password" value="mysql" />
         </dataSource>
      </environment>
   </environments>

</configuration>
```

SqlMapConfig.xml 是 mybatis 核心配置文件，上边文件的配置内容为数据源、事务管理。

1.6.6 第五步：po 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;// 用户姓名
    private String sex;// 性别
    private Date birthday;// 生日
    private String address;// 地址
get/set……
```

1.6.7 第六步：程序编写

1.6.7.1查询

1.6.7.1.1 映射文件：

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">
</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。

在 SqlMapConfig.xml 中添加：

```
<!-- 根据id获取用户信息 -->
    <select id="findUserById" parameterType="int"
resultType="cn.itcast.mybatis.po.User">
        select * from user where id = #{id}
    </select>
    <!-- 自定义条件查询用户列表 -->
    <select id="findUserByUsername" parameterType="java.lang.String"
          resultType="cn.itcast.mybatis.po.User">
       select * from user where username like '%${value}%'
    </select>
```

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。

1.6.7.1.2 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：

```
<mappers>
        <mapper resource="sqlmap/User.xml"/>
</mappers>
```

1.6.7.1.3 测试程序：

```
public class Mybatis_first {

    //会话工厂
    private SqlSessionFactory sqlSessionFactory;

    @Before
```

```
    public void createSqlSessionFactory() throws IOException {
        // 配置文件
        String resource = "SqlMapConfig.xml";
        InputStream inputStream = Resources.getResourceAsStream(resource);

        // 使用SqlSessionFactoryBuilder从xml配置文件中创建
SqlSessionFactory
        sqlSessionFactory = new SqlSessionFactoryBuilder()
                .build(inputStream);

    }

    // 根据 id查询用户信息
    @Test
    public void testFindUserById() {
        // 数据库会话实例
        SqlSession sqlSession = null;
        try {
            // 创建数据库会话实例sqlSession
            sqlSession = sqlSessionFactory.openSession();
            // 查询单个记录，根据用户id查询用户信息
            User user = sqlSession.selectOne("test.findUserById",
10);
            // 输出用户信息
            System.out.println(user);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (sqlSession != null) {
                sqlSession.close();
            }
        }

    }

    // 根据用户名称模糊查询用户信息
    @Test
    public void testFindUserByUsername() {
        // 数据库会话实例
        SqlSession sqlSession = null;
        try {
            // 创建数据库会话实例sqlSession
            sqlSession = sqlSessionFactory.openSession();
            // 查询单个记录，根据用户id查询用户信息
```

```
            List<User> list =
sqlSession.selectList("test.findUserByUsername", "张");
            System.out.println(list.size());
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (sqlSession != null) {
                sqlSession.close();
            }
        }

    }
}
```

1.6.7.1.4 #{}和${}

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

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

1.6.7.1.5 parameterType 和 resultType

parameterType：指定输入参数类型，mybatis 通过 ognl 从输入对象中获取参数值拼接在 sql 中。
resultType：指定输出结果类型，mybatis 将 sql 查询结果的一行记录数据映射为 resultType 指定类型的对象。

1.6.7.1.6 selectOne 和 selectList

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

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

org.apache.ibatis.session.defaults.DefaultSqlSession.selectOne(DefaultSqlSession.java:70)

selectList 可以查询一条或多条记录。

1.6.7.2添加

1.6.7.2.1 映射文件：

在 SqlMapConfig.xml 中添加：

```
<!-- 添加用户 -->
    <insert id="insertUser" parameterType="cn.itcast.mybatis.po.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>
```

1.6.7.2.2 测试程序：

```
// 添加用户信息
    @Test
    public void testInsert() {
        // 数据库会话实例
        SqlSession sqlSession = null;
        try {
            // 创建数据库会话实例sqlSession
            sqlSession = sqlSessionFactory.openSession();
            // 添加用户信息
            User user = new User();
            user.setUsername("张小明");
            user.setAddress("河南郑州");
            user.setSex("1");
            user.setPrice(1999.9f);
            sqlSession.insert("test.insertUser", user);
            //提交事务
            sqlSession.commit();
        } catch (Exception e) {
```

```
            e.printStackTrace();
        } finally {
            if (sqlSession != null) {
                sqlSession.close();
            }
        }
    }
```

1.6.7.2.3 mysql 自增主键返回

通过修改 sql 映射文件，可以将 mysql 自增主键返回：

```
<insert id="insertUser" parameterType="cn.itcast.mybatis.po.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>
```

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。

1.6.7.2.4 Mysql 使用 uuid 实现主键

需要增加通过 select uuid()得到 uuid 值

```
<insert    id="insertUser" parameterType="cn.itcast.mybatis.po.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”

1.6.7.2.5 Oracle 使用序列生成主键

首先自定义一个序列且用于生成主键，selectKey 使用如下：

```
<insert    id="insertUser" parameterType="cn.itcast.mybatis.po.User">
<selectKey resultType="java.lang.Integer" order="BEFORE"
keyProperty="id">
SELECT 自定义序列.NEXTVAL FROM DUAL
</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”

1.6.7.3删除

1.6.7.3.1 映射文件：

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

1.6.7.3.2 测试程序：

```
// 根据id删除用户
    @Test
    public void testDelete() {
        // 数据库会话实例
        SqlSession sqlSession = null;
        try {
            // 创建数据库会话实例sqlSession
            sqlSession = sqlSessionFactory.openSession();
            // 删除用户
            sqlSession.delete("test.deleteUserById",18);
            // 提交事务
```

```
            sqlSession.commit();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (sqlSession != null) {
                sqlSession.close();
            }
        }
    }
```

1.6.7.4修改

1.6.7.4.1 映射文件

```
<!-- 更新用户 -->
    <update id="updateUser" parameterType="cn.itcast.mybatis.po.User">
        update user set
username=#{username},birthday=#{birthday},sex=#{sex},address=#{address}
        where id=#{id}
        </update>
```

1.6.7.4.2 测试程序

```
// 更新用户信息
    @Test
    public void testUpdate() {
        // 数据库会话实例
        SqlSession sqlSession = null;
        try {
            // 创建数据库会话实例sqlSession
            sqlSession = sqlSessionFactory.openSession();
            // 添加用户信息
            User user = new User();
            user.setId(16);
            user.setUsername("张小明");
            user.setAddress("河南郑州");
            user.setSex("1");
            user.setPrice(1999.9f);
            sqlSession.update("test.updateUser", user);
```

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

        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (sqlSession != null) {
                sqlSession.close();
            }
        }
    }
```

1.6.8 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。

1.6.9 与 hibernate 不同

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

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

映射文件，工作量大。

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

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。

2 Dao 开发方法

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。

2.1 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

2.2 SqlSession 的使用范围

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。

2.2.1 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。

2.2.2 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。

2.2.3 SqlSession

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。

执行过程如下：

1、 加载数据源等配置信息

Environment environment = configuration.getEnvironment();

2、 创建数据库链接

3、 创建事务对象

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

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

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

结论：

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

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

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

```
    }
```

2.3 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。

2.3.1 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">
<!-- 根据id获取用户信息 -->
    <select id="findUserById" parameterType="int"
resultType="cn.itcast.mybatis.po.User">
        select * from user where id = #{id}
    </select>
<!-- 添加用户 -->
    <insert id="insertUser" parameterType="cn.itcast.mybatis.po.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>
</mapper>
```

2.3.2 Dao 接口

```
Public interface UserDao {
    public User getUserById(int id) throws Exception;
    public void insertUser(User user) throws Exception;
}
```

```
Public class UserDaoImpl implements UserDao {
```

```
    //注入SqlSessionFactory
    public UserDaoImpl(SqlSessionFactory sqlSessionFactory){
        this.setSqlSessionFactory(sqlSessionFactory);
    }

    private SqlSessionFactory sqlSessionFactory;
    @Override
    public User getUserById(int id) throws Exception {
        SqlSession session = sqlSessionFactory.openSession();
        User user = null;
        try {
            //通过sqlsession调用selectOne方法获取一条结果集
            //参数1：指定定义的statement的id,参数2：指定向statement中传递的参
数
            user = session.selectOne("test.findUserById", 1);
            System.out.println(user);

        } finally{
            session.close();
        }
        return user;
    }

    @Override
    Public void insertUser(User user) throws Exception {
        SqlSession sqlSession = sqlSessionFactory.openSession();
        try {
            sqlSession.insert("insertUser", user);
            sqlSession.commit();
        } finally{
            session.close();
        }

    }
}
```

2.3.3 问题

原始 Dao 开发中存在以下问题：

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

2.4 Mapper 动态代理方式

2.4.1 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

2.4.2 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">
<mapper namespace="cn.itcast.mybatis.mapper.UserMapper">
<!-- 根据id获取用户信息 -->
    <select id="findUserById" parameterType="int"
resultType="cn.itcast.mybatis.po.User">
        select * from user where id = #{id}
    </select>
<!-- 自定义条件查询用户列表 -->
    <select id="findUserByUsername" parameterType="java.lang.String"
            resultType="cn.itcast.mybatis.po.User">
       select * from user where username like '%${value}%'
    </select>
<!-- 添加用户 -->
    <insert id="insertUser" parameterType="cn.itcast.mybatis.po.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>

</mapper>
```

2.4.3 Mapper.java(接口文件)

```
/**
 * 用户管理mapper
 */
Public interface UserMapper {
    //根据用户id查询用户信息
    public User findUserById(int id) throws Exception;
    //查询用户列表
    public List<User> findUserByUsername(String username) throws Exception;
    //添加用户信息
    public void insertUser(User user)throws Exception;
}
```

接口定义有如下特点：

1、 Mapper 接口方法名和 Mapper.xml 中定义的 statement 的 id 相同

2、 Mapper 接口方法的输入参数类型和 mapper.xml 中定义的 statement 的 parameterType 的类型相同

3、 Mapper 接口方法的输出参数类型和 mapper.xml 中定义的 statement 的 resultType 的类型相同

2.4.4 加载 UserMapper.xml 文件

修改 SqlMapConfig.xml 文件：

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

2.4.5 测试

```
Public class UserMapperTest extends TestCase {

    private SqlSessionFactory sqlSessionFactory;

    protected void setUp() throws Exception {
        //mybatis配置文件
        String resource = "sqlMapConfig.xml";
        InputStream inputStream = Resources.getResourceAsStream(resource);
        //使用SqlSessionFactoryBuilder创建sessionFactory
        sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);
    }


    Public void testFindUserById() throws Exception {
        //获取session
        SqlSession session = sqlSessionFactory.openSession();
        //获取mapper接口的代理对象
        UserMapper userMapper = session.getMapper(UserMapper.class);
        //调用代理对象方法
        User user = userMapper.findUserById(1);
        System.out.println(user);
        //关闭session
        session.close();

    }
    @Test
    public void testFindUserByUsername() throws Exception {
        SqlSession sqlSession = sqlSessionFactory.openSession();
        UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
        List<User> list = userMapper.findUserByUsername("张");
        System.out.println(list.size());

    }
Public void testInsertUser() throws Exception {
        //获取session
        SqlSession session = sqlSessionFactory.openSession();
        //获取mapper接口的代理对象
        UserMapper userMapper = session.getMapper(UserMapper.class);
        //要添加的数据
        User user = new User();
```

```
        user.setUsername("张三");
        user.setBirthday(new Date());
        user.setSex("1");
        user.setAddress("北京市");
        //通过mapper接口添加用户
        userMapper.insertUser(user);
        //提交
        session.commit();
        //关闭session
        session.close();
    }


}
```

2.4.6 总结

◆ selectOne 和 selectList

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

◆ namespace

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

3 SqlMapConfig.xml 配置文件

3.1 配置内容

SqlMapConfig.xml 中配置的内容和顺序如下：

properties（属性）
settings（全局配置参数）
typeAliases（类型别名）
typeHandlers（类型处理器）
objectFactory（对象工厂）
plugins（插件）

environments（环境集合属性对象）
　　environment（环境子属性对象）
　　　　transactionManager（事务管理）
　　　　dataSource（数据源）
mappers（映射器）

3.2 properties（属性）

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

在 classpath 下定义 db.properties 文件，

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

SqlMapConfig.xml 引用如下：

```
<properties resource="db.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：

- 在 properties 元素体内定义的属性首先被读取。
- 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。
- 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。

3.3 settings（配置）

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

详细参见“学习资料/mybatis-settings.xlsx”文件

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,hashCode,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	

3.4 typeAliases（类型别名）

3.4.1 mybatis 支持别名：

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

3.4.2 自定义别名：

在 SqlMapConfig.xml 中配置：

```
<typeAliases>
    <!-- 单个别名定义 -->
    <typeAlias alias="user" type="cn.itcast.mybatis.po.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="cn.itcast.mybatis.po"/>
    <package name="其它包"/>
</typeAliases>
```

3.5 typeHandlers（类型处理器）

类型处理器用于 java 类型和 jdbc 类型映射，如下：

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

mybatis 自带的类型处理器基本上满足日常需求，不需要单独定义。

mybatis 支持类型处理器：

类型处理器	**Java**类型	**JDBC**类型
BooleanTypeHandler	Boolean，boolean	任何兼容的布尔值
ByteTypeHandler	Byte，byte	任何兼容的数字或字节类型
ShortTypeHandler	Short，short	任何兼容的数字或短整型
IntegerTypeHandler	Integer，int	任何兼容的数字和整型
LongTypeHandler	Long，long	任何兼容的数字或长整型
FloatTypeHandler	Float，float	任何兼容的数字或单精度浮点型
DoubleTypeHandler	Double，double	任何兼容的数字或双精度浮点型
BigDecimalTypeHandler	BigDecimal	任何兼容的数字或十进制小数类型
StringTypeHandler	String	CHAR和VARCHAR类型
ClobTypeHandler	String	CLOB和LONGVARCHAR类型
NStringTypeHandler	String	NVARCHAR和NCHAR类型
NClobTypeHandler	String	NCLOB类型
ByteArrayTypeHandler	byte[]	任何兼容的字节流类型
BlobTypeHandler	byte[]	BLOB和LONGVARBINARY类型
DateTypeHandler	Date（java.util）	TIMESTAMP类型
DateOnlyTypeHandler	Date（java.util）	DATE类型
TimeOnlyTypeHandler	Date（java.util）	TIME类型
SqlTimestampTypeHandler	Timestamp（java.sql）	TIMESTAMP类型
SqlDateTypeHandler	Date（java.sql）	DATE类型
SqlTimeTypeHandler	Time（java.sql）	TIME类型
ObjectTypeHandler	任意	其他或未指定类型
EnumTypeHandler	Enumeration类型	VARCHAR-任何兼容的字符串类型，作为代码存储（而不是索引）。

3.6 mappers（映射器）

Mapper 配置的几种方法：

3.6.1 <mapper resource=" " />

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

3.6.2 <mapper url=" " />

使用完全限定路径
如：<mapper url="file:///D:\workspace_spingmvc\mybatis_01\config\sqlmap\User.xml" />

3.6.3 <mapper class=" " />

使用 mapper 接口类路径
如：<mapper class="cn.itcast.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。

3.6.4 <package name=""/>

注册指定包下的所有 mapper 接口
如：<package name="cn.itcast.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。

4 Mapper.xml 映射文件

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。

4.1 parameterType(输入类型)

4.1.1 #{}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id查询用户信息 -->
    <select id="findUserById" 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：

```
<!-- 根据名称模糊查询用户信息 -->
    <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。

4.1.2 传递简单类型

参考上边的例子。

4.1.3 传递 pojo 对象

Mybatis 使用 ognl 表达式解析对象字段的值，如下例子：

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

上边红色标注的是 user 对象中的字段名称。

测试：

```
Public void testFindUserByUser()throws Exception{
        //获取session
        SqlSession session = sqlSessionFactory.openSession();
        //获限mapper接口实例
        UserMapper userMapper = session.getMapper(UserMapper.class);
        //构造查询条件user对象
        User user = new User();
        user.setId(1);
        user.setUsername("管理员");
        //传递user对象查询用户列表
        List<User>list = userMapper.findUserByUser(user);
        //关闭session
        session.close();
    }
```

异常测试：

Sql 中字段名输入错误后测试，username 输入 dusername 测试结果报错：

org.apache.ibatis.exceptions.PersistenceException:
Error querying database. Cause: org.apache.ibatis.reflection.ReflectionException: There is no getter for property named 'dusername' in 'class cn.itcast.mybatis.po.User'
Cause: org.apache.ibatis.reflection.ReflectionException: There is no getter for property named 'dusername' in 'class cn.itcast.mybatis.po.User'

4.1.4 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。

4.1.4.1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 QueryVo {

    private User user;

    //自定义用户扩展类
    private UserCustom userCustom;
```

4.1.4.2mapper.xml 映射文件

```
<!-- 查询用户列表
根据用户名称和用户性别查询用户列表
  -->
  <select id="findUserList" parameterType="queryVo" resultType="user">
     select * from user where username = #{user.username} and sex=#{user.sex}
  </select>

apper>
```

说明：mybatis 底层通过 ognl 从 pojo 中获取属性值：#{user.username}，user 即是传入的包装对象的属性。queryVo 是别名，即上边定义的包装对象类型。

4.1.5 传递 hashmap

Sql 映射文件定义如下：

```
<!-- 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。

测试：

```
Public void testFindUserByHashmap()throws Exception{
       //获取session
       SqlSession session = sqlSessionFactory.openSession();
       //获限mapper接口实例
       UserMapper userMapper = session.getMapper(UserMapper.class);
       //构造查询条件Hashmap对象
       HashMap<String, Object> map = new HashMap<String, Object>();
       map.put("id", 1);
       map.put("username", "管理员");

       //传递Hashmap对象查询用户列表
       List<User>list = userMapper.findUserByHashmap(map);
       //关闭session
       session.close();
   }
```

异常测试：
传递的 map 中的 key 和 sql 中解析的 key 不一致。
测试结果没有报错，只是通过 key 获取值为空。

4.2 resultType(输出类型)

4.2.1 输出简单类型

参考 getnow 输出日期类型，看下边的例子输出整型：

Mapper.xml 文件

```
<!-- 获取用户列表总数 -->
   <select id="findUserCount" parameterType="user" resultType="int">
      select count(1) from user
   </select>
```

Mapper 接口

```
public int findUserCount(User user) throws Exception;
```

调用：

```
Public void testFindUserCount() throws Exception{
        //获取session
        SqlSession session = sqlSessionFactory.openSession();
        //获取mapper接口实例
        UserMapper userMapper = session.getMapper(UserMapper.class);

        User user = new User();
        user.setUsername("管理员");

        //传递Hashmap对象查询用户列表
        int count = userMapper.findUserCount(user);

        //关闭session
        session.close();
    }
```

总结：
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 session 的 selectOne 可查询单条记录。

4.2.2 输出 pojo 对象

参考 findUserById 的定义：
Mapper.xml

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

Mapper 接口：

```
public User findUserById(int id) throws Exception;
```

测试：

```
Public void testFindUserById() throws Exception {
        //获取session
        SqlSession session = sqlSessionFactory.openSession();
        //获限mapper接口实例
        UserMapper userMapper = session.getMapper(UserMapper.class);
```

```
        //通过mapper接口调用statement
        User user = userMapper.findUserById(1);
        System.out.println(user);
        //关闭session
        session.close();
    }
```

使用 session 调用 selectOne 查询单条记录。

4.2.3 输出 pojo 列表

参考 selectUserByName 的定义：

Mapper.xml

```
<!-- 根据名称模糊查询用户信息 -->
    <select id="findUserByUsername" parameterType="string"
resultType="user">
        select * from user where username like '%${value}%'
    </select>
```

Mapper 接口：

```
public List<User> findUserByUsername(String username) throws Exception;
```

测试：

```
Public void testFindUserByUsername()throws Exception{
        //获取session
        SqlSession session = sqlSessionFactory.openSession();
        //获限mapper接口实例
        UserMapper userMapper = session.getMapper(UserMapper.class);
        //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.findUserByUsername("管理员");
        //关闭session
        session.close();
    }
```

使用 session 的 selectList 方法获取 pojo 列表。

4.2.4 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。

4.2.5 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。

4.3 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。

4.3.1 Mapper.xml 定义

```
<!-- 查询用户列表 根据用户名称和用户性别查询用户列表 -->
<select id="findUserListResultMap" parameterType="queryVo" resultMap="userListResultMap">
    select id id_,username username_,birthday birthday_ from user

    <!-- where自动将第一个and去掉 -->
    <where>
        <!--
        refid: 指定 sql片段的id，如果要引用其它命名空间的sql片段，需要前边加namespace

         -->
        <include refid="query_user_where"/>
    </where>
</select>
```

使用 resultMap 指定上边定义的 personmap。

4.3.2 定义 resultMap

由于上边的 mapper.xml 中 sql 查询列和 Users.java 类属性不一致，需要定义 resultMap：userListResultMap 将 sql 查询列和 Users.java 类属性对应起来

```
<!-- 定义resultMap，将用户查询的字段和user这个pojo的属性名作一个对应关系 -->
<!--
type:最终映射的java对象。
id: resultMap的唯一标识
 -->
<resultMap type="user" id="userListResultMap">
    <!-- id标签: 查询结果集的唯 一标识 列(主键或唯 一标识 )
    column: sql查询字段名(列名)
    property: pojo的属性名

    result标签: 普通列
     -->

    <id column="id_" property="id"/>
    <result column="username_" property="username"/>
    <result column="birthday_" property="birthday"/>


</resultMap>
```

<id />：此属性表示查询结果集的唯一标识，非常重要。如果是多个字段为复合唯一约束则定义多个<id />。
Property：表示 person 类的属性。
Column：表示 sql 查询出来的字段名。
Column 和 property 放在一块儿表示将 sql 查询出来的字段映射到指定的 pojo 类属性上。

<result />：普通结果，即 pojo 的属性。

4.3.3 Mapper 接口定义

```
public List<User> findUserListResultMap() throws Exception;
```

4.4 动态 sql(重点)

通过 mybatis 提供的各种标签方法实现动态拼接 sql。

4.4.1 If

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

注意要做不等于空字符串校验。

4.4.2 Where

上边的 sql 也可以改为：

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

<where />可以自动处理第一个 and。

4.4.3 foreach

向 sql 传递数组或 List，mybatis 使用 foreach 解析，如下：

4.4.3.1通过 pojo 传递 list

◆ 需求

传入多个 id 查询用户信息，用下边两个 sql 实现：

SELECT * FROM USERS WHERE username LIKE '%张%' AND (id =10 OR id =89 OR id=16)

SELECT * FROM USERS WHERE username LIKE '%张%' id IN (10,89,16)

◆ 在 pojo 中定义 list 属性 ids 存储多个用户 id，并添加 getter/setter 方法

```
public class QueryVo {

    private User user;

    //自定义用户扩展类
    private UserCustom userCustom;

    //传递多个用户id
    private List<Integer> ids;
```

◆ mapper.xml

```
<if test="ids!=null and ids.size>0">
          <foreach collection="ids" open=" and id in(" close=")" item="id"
separator="," >
              #{id}
          </foreach>
</if>
```

◆ 测试代码：

```
List<Integer> ids = new ArrayList<Integer>();
       ids.add(1);//查询id为1的用户
       ids.add(10); //查询id为10的用户
       queryVo.setIds(ids);
       List<User> list = userMapper.findUserList(queryVo);
```

4.4.3.2传递单个 List

传递 List 类型在编写 mapper.xml 没有区别，唯一不同的是只有一个 List 参数时它的参数名为 list。

如下：

◆ Mapper.xml

```
<select id="selectUserByList" parameterType="java.util.List"
resultType="user">
        select * from user
        <where>
        <!-- 传递List，List中是pojo -->
        <if test="list!=null">
        <foreach collection="list" item="item" open="and id
in("separator=","close=")">
            #{item.id}
        </foreach>
        </if>
        </where>
    </select>
```

◆ Mapper 接口

```
public List<User> selectUserByList(List userlist) throws Exception;
```

◆ 测试：

```
Public void testselectUserByList()throws Exception{
        //获取session
        SqlSession session = sqlSessionFactory.openSession();
        //获限mapper接口实例
        UserMapper userMapper = session.getMapper(UserMapper.class);
        //构造查询条件List
        List<User> userlist = new ArrayList<User>();
        User user = new User();
        user.setId(1);
        userlist.add(user);
        user = new User();
        user.setId(2);
        userlist.add(user);
        //传递userlist列表查询用户列表
        List<User>list = userMapper.selectUserByList(userlist);
        //关闭session
```

```
        session.close();
    }
```

4.4.3.3传递单个数组（数组中是 pojo）：

请阅读文档学习。

◆ Mapper.xml

```
<!-- 传递数组综合查询用户信息 -->
    <select id="selectUserByArray" parameterType="Object[]"
resultType="user">
        select * from user
        <where>
        <!-- 传递数组 -->
        <if test="array!=null">
        <foreach collection="array" index="index" item="item" open="and id
in("separator=","close=")">
            #{item.id}
        </foreach>
        </if>
        </where>
    </select>
```

sql 只接收一个数组参数，这时 sql 解析参数的名称 mybatis 固定为 array，如果数组是通过一个 pojo 传递到 sql 则参数的名称为 pojo 中的属性名。

index：为数组的下标。

item：为数组每个元素的名称，名称随意定义

open：循环开始

close：循环结束

separator：中间分隔输出

◆ Mapper 接口：

```
public List<User> selectUserByArray(Object[] userlist) throws Exception;
```

◆ 测试：

```
Public void testselectUserByArray()throws Exception{
        //获取session
        SqlSession session = sqlSessionFactory.openSession();
        //获限mapper接口实例
        UserMapper userMapper = session.getMapper(UserMapper.class);
```

```
        //构造查询条件List
        Object[] userlist = new Object[2];
        User user = new User();
        user.setId(1);
        userlist[0]=user;
        user = new User();
        user.setId(2);
        userlist[1]=user;
        //传递user对象查询用户列表
        List<User>list = userMapper.selectUserByArray(userlist);
        //关闭session
        session.close();
    }
```

4.4.3.4传递单个数组（数组中是字符串类型）：

请阅读文档学习。

◆ Mapper.xml

```
<!-- 传递数组综合查询用户信息 -->
    <select id="selectUserByArray" parameterType="Object[]"
resultType="user">
        select * from user
        <where>
        <!-- 传递数组 -->
        <if test="array!=null">
        <foreach collection="array"index="index"item="item"open="and id
in("separator=","close=")">
            #{item}
        </foreach>
        </if>
        </where>
    </select>
```

如果数组中是简单类型则写为#{item}，不用再通过 ognl 获取对象属性值了。

◆ Mapper 接口：

```
public List<User> selectUserByArray(Object[] userlist) throws Exception;
```

◆ 测试：

```
Public void testselectUserByArray()throws Exception{
```

```
        //获取session
        SqlSession session = sqlSessionFactory.openSession();
        //获限mapper接口实例
        UserMapper userMapper = session.getMapper(UserMapper.class);
        //构造查询条件List
        Object[] userlist = new Object[2];
        userlist[0]=”1”;
        userlist[1]=”2”;
        //传递user对象查询用户列表
        List<User>list = userMapper.selectUserByArray(userlist);
        //关闭session
        session.close();
    }
```

4.4.4 Sql 片段

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

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

◆ 将 where 条件抽取出来：

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

```
        and username like '%${username}%'
    </if>
</sql>
```

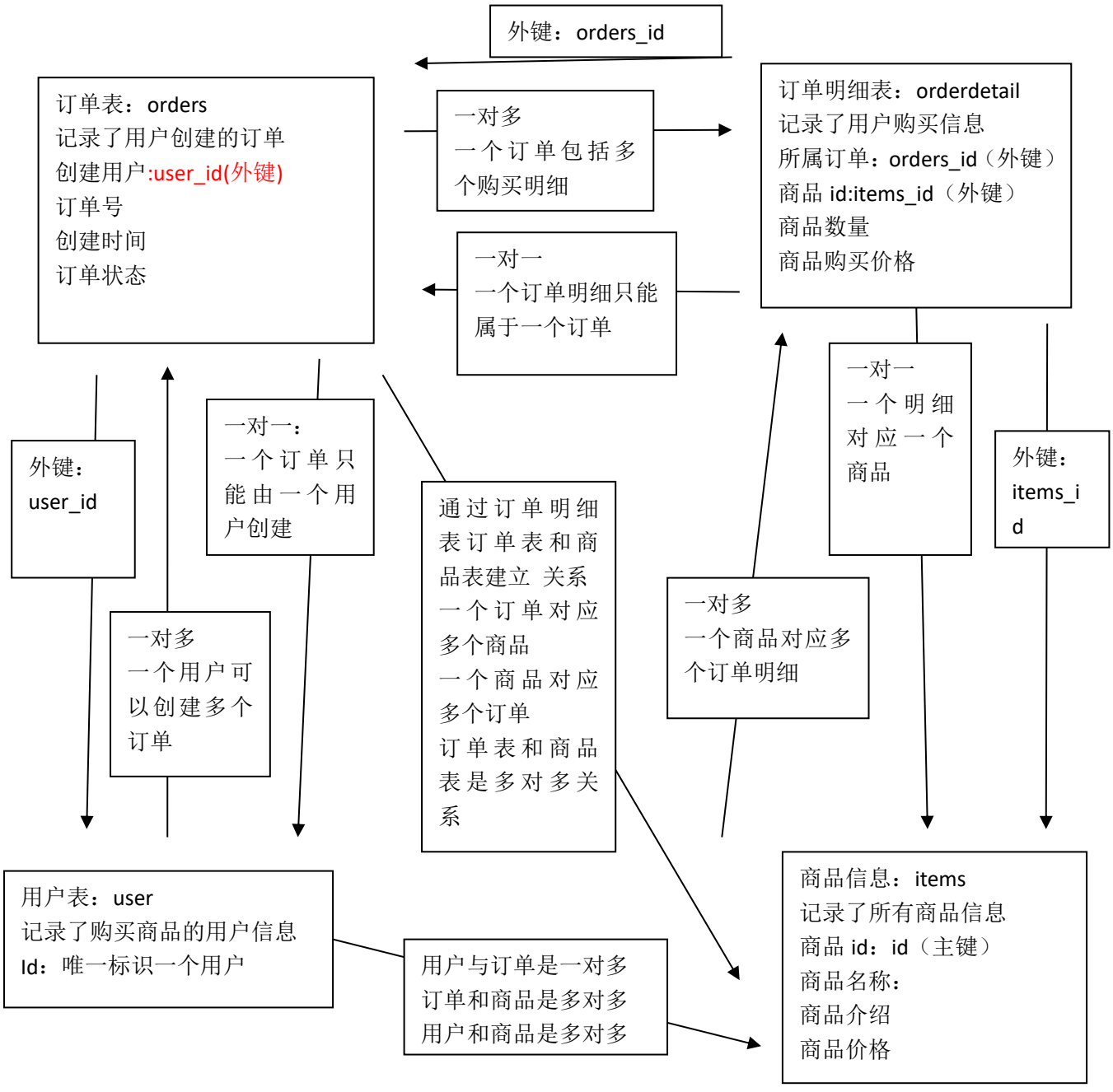
◆ 使用 include 引用：

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

注意：如果引用其它 mapper.xml 的 sql 片段，则在引用时需要加上 namespace，如下：
<include refid="namespace.sql 片段"/>

5 关联查询

5.1 商品订单数据模型



5.2 一对一查询

案例：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。

5.2.1 方法一：

使用 resultType，定义订单信息 po 类，此 po 类中包括了订单信息和用户信息：

5.2.1.1Sql 语句：

```
SELECT
   orders.*,
   user.username,
   userss.address
FROM
   orders,
   user
WHERE orders.user_id = user.id
```

5.2.1.2定义 po 类

Po 类中应该包括上边 sql 查询出来的所有字段，如下：

```
public class OrdersCustom extends Orders {

    private String username;// 用户名称
    private String address;// 用户地址
get/set。。。。
```

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。

5.2.1.3Mapper.xml

```
<!-- 查询所有订单信息 -->
    <select id="findOrdersList"
resultType="cn.itcast.mybatis.po.OrdersCustom">
    SELECT
    orders.*,
    user.username,
    user.address
    FROM
    orders,user
    WHERE orders.user_id = user.id
    </select>
```

5.2.1.4Mapper 接口：

```
public List<OrdersCustom> findOrdersList() throws Exception;
```

5.2.1.5测试：

```
Public void testfindOrdersList()throws Exception{
        //获取session
        SqlSession session = sqlSessionFactory.openSession();
        //获限mapper接口实例
        UserMapper userMapper = session.getMapper(UserMapper.class);
        //查询订单信息
        List<OrdersCustom> list = userMapper.findOrdersList();
        System.out.println(list);
        //关闭session
        session.close();
    }
```

5.2.1.6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。

5.2.2 方法二：

使用 resultMap，定义专门的 resultMap 用于映射一对一查询结果。

5.2.2.1Sql 语句：

```
SELECT
   orders.*,
   user.username,
   user.address
FROM
   orders,
   user
WHERE orders.user_id = user.id
```

5.2.2.2定义 po 类

在 Orders 类中加入 User 属性，user 属性中用于存储关联查询的用户信息，因为订单关联查询用户是一对一关系，所以这里使用单个 User 对象存储关联查询的用户信息。

5.2.2.3Mapper.xml

```
<select id="findOrdersListResultMap" resultMap="userordermap">
    SELECT
    orders.*,
    user.username,
    user.address
    FROM
    orders,user
    WHERE orders.user_id = user.id
    </select>
```

这里 resultMap 指定 *userordermap*。

5.2.2.4定义 resultMap

需要关联查询映射的是用户信息，使用 association 将用户信息映射到订单对象的用户属性中。

```
<!-- 订单信息resultmap -->
    <resultMap type="cn.itcast.mybatis.po.Orders" id="userordermap">
    <!-- 这里的id，是mybatis在进行一对一查询时将user字段映射为user对象时要使用，
必须写 -->
<id property="id" column="id"/>
<result property="user_id" column="user_id"/>
<result property="number" column="number"/>
<association property="user" javaType="cn.itcast.mybatis.po.User">
<!-- 这里的id为user的id，如果写上表示给user的id属性赋值 -->
<id property="id" column="user_id"/>
<result property="username" column="username"/>
<result property="address" column="address"/>
</association>
</resultMap>
```

association：表示进行关联查询单条记录

property：表示关联查询的结果存储在 cn.itcast.mybatis.po.Orders 的 user 属性中

javaType：表示关联查询的结果类型

<id property="id" column="user_id"/>：查询结果的 user_id 列对应关联对象的 id 属性，这里是<id />表示 user_id 是关联查询对象的唯一标识。

<result property="username" column="username"/>：查询结果的 username 列对应关联对象的 username 属性。

5.2.2.5Mapper 接口：

```
public List<Orders> findOrdersListResultMap() throws Exception;
```

5.2.2.6测试：

```
Public void testfindOrdersListResultMap()throws Exception{
        //获取session
        SqlSession session = sqlSessionFactory.openSession();
        //获限mapper接口实例
        UserMapper userMapper = session.getMapper(UserMapper.class);
        //查询订单信息
        List<Orders> list = userMapper.findOrdersList2();
        System.out.println(list);
```

```
        //关闭session
        session.close();
    }
```

5.2.2.7小结：

使用 association 完成关联查询，将关联查询信息映射到 pojo 对象中。

5.3 一对多查询

案例：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。

使用 resultMap 实现如下：

5.3.1 Sql 语句：

```
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
```

5.3.2 定义 po 类

在 Orders 类中加入 User 属性。
在 Orders 类中加入 List<Orderdetail> orderdetails 属性

5.3.3 Mapper.xml

```
<select id="findOrdersDetailList" resultMap="userorderdetailmap">
    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>
```

5.3.4 定义 resultMap

```
<!-- 订单信息resultmap -->
<resultMap type="cn.itcast.mybatis.po.Orders" id="userorderdetailmap">
<id property="id"column="id"/>
<result property="user_id" column="user_id"/>
<result property="number" column="number"/>
<association property="user" javaType="cn.itcast.mybatis.po.User">
<id property="id" column="user_id"/>
<result property="username" column="username"/>
<result property="address" column="address"/>
</association>
<collection property="orderdetails"
ofType="cn.itcast.mybatis.po.Orderdetail">
    <id property="id" column="orderdetail_id"/>
    <result property="items_id" column="items_id"/>
    <result property="items_num" column="items_num"/>
</collection>
</resultMap>
```

黄色部分和上边一对一查询订单及用户信息定义的 resultMap 相同，
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 cn.itcast.mybatis.po.Orders 上哪个属性。
ofType="cn.itcast.mybatis.po.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。

5.3.4.1resultMap 使用继承

上边定义的 resultMap 中黄色部分和一对一查询订单信息的 resultMap 相同，这里使用继承可以不再填写重复的内容，如下：

```
<resultMap type="cn.itcast.mybatis.po.Orders"
id="userorderdetailmap" extends="userordermap">
<collection property="orderdetails"
ofType="cn.itcast.mybatis.po.Orderdetail">
   <id property="id" column="orderdetail_id"/>
   <result property="items_id" column="items_id"/>
   <result property="items_num" column="items_num"/>
</collection>
</resultMap>
```

使用 extends 继承订单信息 userordermap。

5.3.5 Mapper 接口：

```
public List<Orders>findOrdersDetailList () throws Exception;
```

5.3.6 测试：

```
Public void testfindOrdersDetailList()throws Exception{
       //获取session
       SqlSession session = sqlSessionFactory.openSession();
       //获限mapper接口实例
       UserMapper userMapper = session.getMapper(UserMapper.class);
       //查询订单信息
```

```
        List<Orders> list = userMapper.findOrdersDetailList();
        System.out.println(list);
        //关闭session
        session.close();
    }
```

5.3.7 小结

使用 collection 完成关联查询，将关联查询信息映射到集合对象。

5.4 多对多查询

5.4.1 查询用户购买的商品信息

5.4.1.1需求

查询用户购买的商品信息。

5.4.1.2sql

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
    orders.*,
    USER .username,
    USER .address,
    orderdetail.id orderdetail_id,
    orderdetail.items_id,
    orderdetail.items_num,
    items.name items_name,
    items.detail items_detail
FROM
    orders,
    USER,
    orderdetail,
    items
WHERE
```

orders.user_id = USER .id
AND orders.id = orderdetail.orders_id
AND orderdetail.items_id = items.id

5.4.1.3po 定义

在 *User* 中添加 List<Orders> orders 属性，在 Orders 类中加入 List<Orderdetail> orderdetails 属性

5.4.1.4resultMap

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。

```
<!-- 一对多查询
    查询用户信息、关联查询订单、订单明细信息、商品信息
     -->
     <resultMap type="cn.itcast.mybatis.po.User"
id="userOrderListResultMap">
       <id column="user_id" property="id"/>
       <result column="username" property="username"/>
       <collection property="orders"
ofType="cn.itcast.mybatis.po.Orders">
         <id  column="id" property="id"/>
         <result property="number" column="number"/>
           <collection property="orderdetails"
ofType="cn.itcast.mybatis.po.Orderdetail">
               <id  column="orderdetail_id" property="id"/>
               <result property="ordersId" column="id"/>
               <result property="itemsId" column="items_id"/>
               <result property="itemsNum" column="items_num"/>
               <association property="items"
javaType="cn.itcast.mybatis.po.Items">
                    <id column="items_id" property="id"/>
                    <result column="items_name" property="name"/>
                    <result column="items_detail" property="detail"/>
               </association>
           </collection>
      </collection>
```

```
    </resultMap>
```

5.4.2 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。

5.5 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。

5.6 延迟加载

需要查询关联信息时，使用 mybatis 延迟加载特性可有效的减少数据库压力，首次查询只查询主要信息，关联信息等用户获取时再加载。

5.6.1 打开延迟加载开关

在 mybatis 核心配置文件中配置：

lazyLoadingEnabled、aggressiveLazyLoading

设置项	描述	允许值	默认值
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

```
<settings>
	<setting name="lazyLoadingEnabled" value="true"/>
	<setting name="aggressiveLazyLoading" value="false"/>
</settings>
```

5.6.2 一对一查询延迟加载

5.6.2.1 需求

查询订单信息，关联查询用户信息。

默认只查询订单信息，当需要查询用户信息时再去查询用户信息。

5.6.2.2Sql 语句：

```
SELECT
   orders.*
FROM
   orders
```

5.6.2.3定义 po 类

在 Orders 类中加入 User 属性。

5.6.2.4Mapper.xml

```
<select id="findOrdersList3" resultMap="userordermap2">
    SELECT
    orders.*
    FROM
    orders
</select>
```

5.6.2.5定义 resultMap

```
<!-- 订单信息resultmap -->
<resultMap type="cn.itcast.mybatis.po.Orders" id="userordermap2">
<id property="id" column="id"/>
<result property="user_id" column="user_id"/>
<result property="number" column="number"/>
<association property="user" javaType="cn.itcast.mybatis.po.User"
select="findUserById" column="user_id"/>
</resultMap>
```

association:
select="findUserById"：指定关联查询 sql 为 findUserById

column="user_id"：关联查询时将 users_id 列的值传入 findUserById
最后将关联查询结果映射至 cn.itcast.mybatis.po.User。

5.6.2.6Mapper 接口：

```
public List<Orders> findOrdersList3() throws Exception;
```

5.6.2.7测试：

```
Public void testfindOrdersList3()throws Exception{
        //获取session
        SqlSession session = sqlSessionFactory.openSession();
        //获限mapper接口实例
        UserMapper userMapper = session.getMapper(UserMapper.class);
        //查询订单信息
        List<Orders> list = userMapper.findOrdersList3();
        System.out.println(list);
        //开始加载，通过orders.getUser方法进行加载
        for(Orders orders:list){
            System.out.println(orders.getUser());
        }
        //关闭session
        session.close();
    }
```

5.6.2.8延迟加载的思考

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。

5.6.3 一对多延迟加载

一对多延迟加载的方法同一对一延迟加载，在 collection 标签中配置 select 内容。
本部分内容自学。

5.6.4 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：

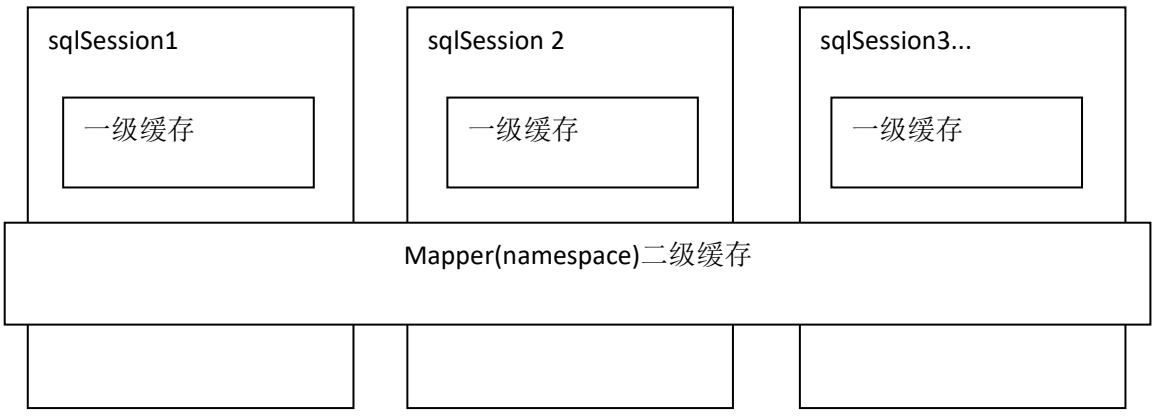
当只有部分记录需要关联查询其它信息时，此时可按需延迟加载，需要关联查询时再向数据库发出 sql，以提高数据库性能。

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。

6 查询缓存

6.1 mybatis 缓存介绍

如下图，是 mybatis 一级缓存和二级缓存的区别图解：



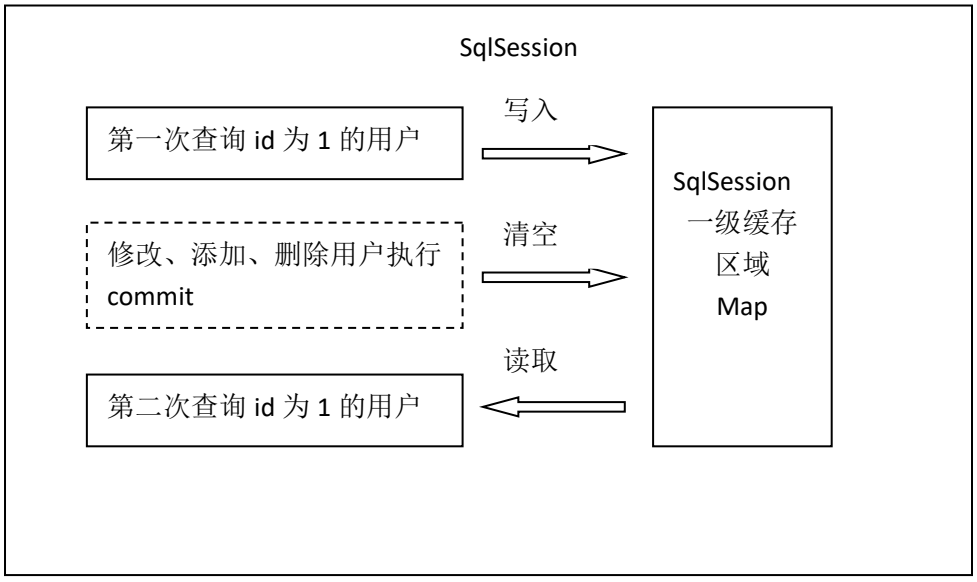
Mybatis 一级缓存的作用域是同一个 SqlSession，在同一个 sqlSession 中两次执行相同的 sql 语句，第一次执行完毕会将数据库中查询的数据写到缓存（内存），第二次会从缓存中获取数据将不再从数据库查询，从而提高查询效率。当一个 sqlSession 结束后该 sqlSession 中的一级缓存也就不存在了。Mybatis 默认开启一级缓存。

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。

6.2 一级缓存

6.2.1 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。

6.2.2 测试 1

```
    //获取session
        SqlSession session = sqlSessionFactory.openSession();
        //获限mapper接口实例
        UserMapper userMapper = session.getMapper(UserMapper.class);
        //第一次查询
        User user1 = userMapper.findUserById(1);
        System.out.println(user1);
        //第二次查询，由于是同一个session则不再向数据发出语句直接从缓存取出
        User user2 = userMapper.findUserById(1);
        System.out.println(user2);
        //关闭session
        session.close();
```

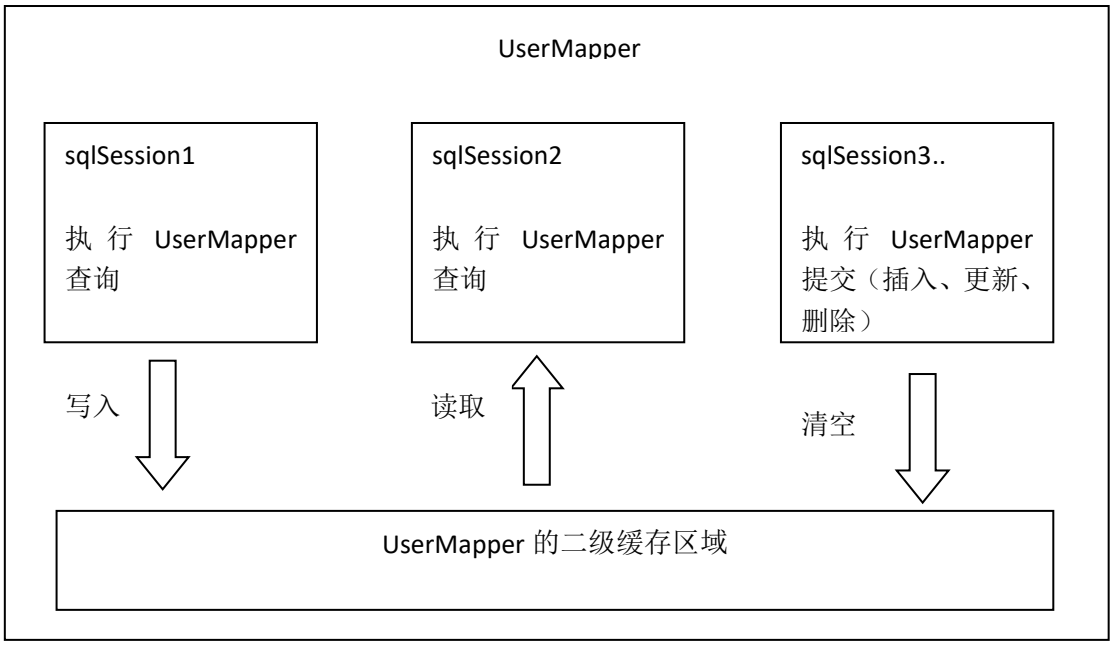
6.2.3 测试 2

```
//获取session
        SqlSession session = sqlSessionFactory.openSession();
        //获限mapper接口实例
        UserMapper userMapper = session.getMapper(UserMapper.class);
        //第一次查询
        User user1 = userMapper.findUserById(1);
        System.out.println(user1);
        //在同一个session执行更新
        User user_update = new User();
        user_update.setId(1);
        user_update.setUsername("李奎");
        userMapper.updateUser(user_update);
        session.commit();
        //第二次查询，虽然是同一个session但是由于执行了更新操作session的缓存被清空，这里重新发出sql操作
        User user2 = userMapper.findUserById(1);
        System.out.println(user2);
```

6.3 二级缓存

6.3.1 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。

6.3.2 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

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。

6.3.3 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

....

6.3.4 测试

```
//获取session1
        SqlSession session1 = sqlSessionFactory.openSession();
        UserMapper userMapper = session1.getMapper(UserMapper.class);
        //使用session1执行第一次查询
        User user1 = userMapper.findUserById(1);
        System.out.println(user1);
        //关闭session1
        session1.close();
        //获取session2
        SqlSession session2 = sqlSessionFactory.openSession();
        UserMapper userMapper2 = session2.getMapper(UserMapper.class);
        //使用session2执行第二次查询，由于开启了二级缓存这里从缓存中获取数据不再向数据库发出sql
        User user2 = userMapper2.findUserById(1);
        System.out.println(user2);
        //关闭session2
        session2.close();
```

6.3.5 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="findOrderListResultMap" resultMap="ordersUserMap" useCache="false">

6.3.6 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="cn.itcast.mybatis.po.User" flushCache="true">

6.3.7 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:

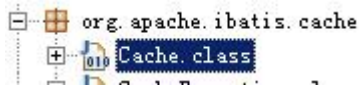
1. LRU – 最近最少使用的:移除最长时间不被使用的对象。
2. FIFO – 先进先出:按对象进入缓存的顺序来移除它们。
3. SOFT – 软引用:移除基于垃圾回收器状态和软引用规则的对象。
4. WEAK – 弱引用:更积极地移除基于垃圾收集器状态和弱引用规则的对象。

6.3.8 mybatis 整合 ehcache

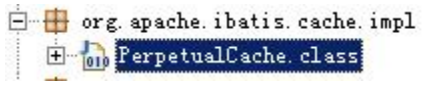
EhCache 是一个纯 Java 的进程内缓存框架，是一种广泛使用的开源 Java 分布式缓存，具有快速、精干等特点，是 Hibernate 中默认的 CacheProvider。

6.3.8.1mybatis 整合 ehcache 原理

mybatis 提供二级缓存 Cache 接口，如下：

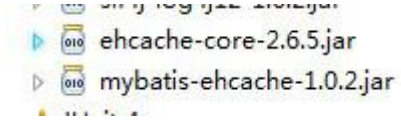


它的默认实现类：



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。

6.3.8.2第一步：引入缓存的依赖包



maven 坐标：

```
<dependency>
          <groupId>org.mybatis.caches</groupId>
          <artifactId>mybatis-ehcache</artifactId>
          <version>1.0.2</version>
       </dependency>
```

6.3.8.3第二步：引入缓存配置文件

classpath 下添加：ehcache.xml
内容如下：

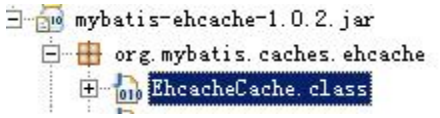
```
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="../config/ehcache.xsd">
    <diskStore path="F:\develop\ehcache" />
    <defaultCache
```

```
        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）

6.3.8.4第三步：开启 ehcache 缓存

EhcacheCache 是 ehcache 对 Cache 接口的实现：



修改 mapper.xml 文件，在 cache 中指定 EhcacheCache。

```
<cache type="org.mybatis.caches.ehcache.EhcacheCache"/>
```

根据需求调整缓存参数：

```
<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>
```

6.3.9 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。

6.3.10局限性

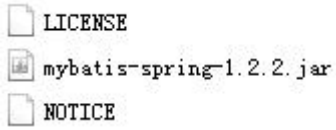
mybatis 二级缓存对细粒度的数据级别的缓存实现不好，比如如下需求：对商品信息进行缓存，由于商品信息查询访问量大，但是要求用户每次都能查询最新的商品信息，此时如果使用 mybatis 的二级缓存就无法实现当一个商品变化时只刷新该商品的缓存信息而不刷新其它商品的信息，因为 mybaits 的二级缓存区域以 mapper 为单位划分，当一个商品信息变化会将所有商品信息的缓存数据全部清空。解决此类问题需要在业务层根据需求对数据有针对性缓存。

7 与 spring 整合

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

7.1 mybatis 与 spring 整合 jar

mybatis 官方提供与 mybatis 与 spring 整合 jar 包：



还包括其它 jar：
spring3.2.0
mybatis3.2.7
dbcp 连接池
数据库驱动

参考：

7.2 Mybatis 配置文件

在 classpath 下创建 *mybatis/SqlMapConfig.xml*

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

<!—使用自动扫描器时，mapper.xml文件如果和mapper.java接口在一个目录则此处不用定义mappers -->
<mappers>
<package name="cn.itcast.mybatis.mapper" />
</mappers>
</configuration>
```

7.3 Spring 配置文件：

在 classpath 下创建 applicationContext.xml，定义数据库链接池、SqlSessionFactory。

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:context="http://www.springframework.org/schema/contex
t"
    xmlns:aop="http://www.springframework.org/schema/aop"
xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/b
eans

    http://www.springframework.org/schema/beans/spring-beans-3.
2.xsd
        http://www.springframework.org/schema/mvc

    http://www.springframework.org/schema/mvc/spring-mvc-3.2.xs
d
        http://www.springframework.org/schema/context

    http://www.springframework.org/schema/context/spring-contex
t-3.2.xsd
        http://www.springframework.org/schema/aop

    http://www.springframework.org/schema/aop/spring-aop-3.2.xs
d
        http://www.springframework.org/schema/tx

    http://www.springframework.org/schema/tx/spring-tx-3.2.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 -->
    <!-- 让spring管理sqlsessionfactory 使用mybatis和spring整合包
中的 -->
    <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>
```

注意：在定义 sqlSessionFactory 时指定数据源 dataSource 和 mybatis 的配置文件。

7.4 Mapper 编写的三种方法

7.4.1 Dao 接口实现类继承 SqlSessionDaoSupport

使用此种方法即原始 dao 开发方法，需要编写 dao 接口，dao 接口实现类、映射文件。

1、 在 sqlMapConfig.xml 中配置映射文件的位置

```
<mappers>
  <mapper resource="mapper.xml 文件的地址" />
  <mapper resource="mapper.xml 文件的地址" />
</mappers>
```

2、 定义 dao 接口

3、 dao 接口实现类集成 SqlSessionDaoSupport

dao 接口实现类方法中可以 this.getSqlSession()进行数据增删改查。

4、 spring 配置

```
<bean id=" "class="mapper接口的实现">
```

```
    <property name="sqlSessionFactory"
ref="sqlSessionFactory"></property>
    </bean>
```

7.4.2 使用 org.mybatis.spring.mapper.MapperFactoryBean

此方法即 mapper 接口开发方法，只需定义 mapper 接口，不用编写 mapper 接口实现类。每个 mapper 接口都需要在 spring 配置文件中定义。

1、 在 sqlMapConfig.xml 中配置 mapper.xml 的位置
如果 mapper.xml 和 mappre 接口的名称相同且在同一个目录，这里可以不用配置

```
<mappers>
  <mapper resource="mapper.xml 文件的地址" />
  <mapper resource="mapper.xml 文件的地址" />
</mappers>
```

2、 定义 mapper 接口

3、 Spring 中定义

```
<bean id="" class="org.mybatis.spring.mapper.MapperFactoryBean">
    <property name="mapperInterface" value="mapper接口地址"/>
    <property name="sqlSessionFactory" ref="sqlSessionFactory"/>
    </bean>
```

7.4.3 使用 mapper 扫描器

此方法即 mapper 接口开发方法，只需定义 mapper 接口，不用编写 mapper 接口实现类。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 接口生成代代理对象。

1、 mapper.xml 文件编写，

2、 定义 mapper 接口

注意 mapper.xml 的文件名和 mapper 的接口名称保持一致，且放在同一个目录

3、 配置 mapper 扫描器

```
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
 <property name="basePackage" value="mapper接口包地址"></property>
```

```
<property name="sqlSessionFactoryBeanName" value="sqlSessionFactory"/>
    </bean>
```

basePackage：扫描包路径，中间可以用逗号或分号分隔定义多个包

4、 使用扫描器后从 spring 容器中获取 mapper 的实现对象

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

8 Mybatis 逆向工程

使用官方网站的 mapper 自动生成工具 mybatis-generator-core-1.3.2 来生成 po 类和 mapper 映射文件。

8.1 第一步：mapper 生成配置文件：

在 generatorConfig.xml 中配置 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

配置文件如下：
详见 generatorSqlmapCustom 工程

8.2 第二步：使用 java 类生成 mapper 文件：

```
Public void generator() throws Exception{
      List<String> warnings = new ArrayList<String>();
      boolean overwrite = true;
      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();
      }
}
```

8.3 第三步：拷贝生成的 mapper 文件到工程中指定的目录中

8.3.1 Mapper.xml

Mapper.xml 的文件拷贝至 mapper 目录内

8.3.2 Mapper.java

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。

8.3.3 第四步 Mapper 接口测试

学会使用 mapper 自动生成的增、删、改、查方法。

```
//删除符合条件的记录
int deleteByExample(UserExample example);
```

```
//根据主键删除
int deleteByPrimaryKey(String 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
UserselectByPrimaryKey(String id);
//根据主键将对象中不为空的值更新至数据库
int updateByPrimaryKeySelective(User record);
//根据主键将对象中所有字段的值更新至数据库
int updateByPrimaryKey(User record);
```

8.4 逆向工程注意事项

8.4.1 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 文件不是内容而是直接覆盖没有此问题。

8.4.2 Table schema 问题

下边是关于针对 oracle 数据库表生成代码的 schema 问题：

Schma 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
```