

Mybatis

第一部分 自定义持久层框架

1.1 分析jdbc操作问题

```
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?
characterEncoding=utf-8", "root", "root");
        // 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, "tom");
        // 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()) {
            int id = resultSet.getInt("id");
            String username = resultSet.getString("username");
            // 封装User
            User user = new User();
            user.setId(id);
            user.setUsername(username);
        }
        System.out.println(user);

    }
} 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();
    }
}
}
```

IDBC问题总结：

- 1、数据库连接创建、释放频繁造成系统资源浪费，从而影响系统性能。
 - 2、Sql语句在代码中硬编码，造成代码不易维护，实际应用中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 问题解决思路

数据库频繁创建连接、释放资源：连接池

sql语句及参数硬编码：配置文件

手动解析封装返回结果集：反射、内省

1.3 自定义框架设计

使用端：

提供核心配置文件

sqlMapConfig.xml: 存放数据源信息, 引入mapper.xml

Mapper.xml: sql语句的配置文件信息

框架端：

1. 读取配置文件

读取完以后以流的形式存在，我们不能讲读取到的配置信息以流的形式存放在内存中，不好操作，可以创建javaBean来存储

- (1) Configuration: 存放数据库基本信息、Map<唯一标识,Mapper> 唯一标识: namespace+"."+id
- (2) MappedStatement: sql语句、statement类型、输入参数java类型、输出参数java类型

2. 解析配置文件

创建SqlSessionFactory类：

方法： SqlSessionFactory build():

第一： 使用dom4j解析配置文件，将解析出来的内容封装到Configuration和MappedStatement中

第二： 创建SqlSessionFactory的实现类DefaultSqlSession

3. 创建SqlSessionFactory:

方法： openSession(): 获取sqlSession接口的实现类实例对象

4. 创建SqlSession接口及实现类: 主要封装CRUD方法

方法： selectList(String StatementId, Object param): 查询所有

selectOne(String StatementId, Object param): 查询单个

close() 释放资源

具体实现： 封装JDBC完成对数据库表的查询操作

设计到的设计模式

构建者模式、工厂模式、代理模式

1.4 自定义框架实现

在使用端项目中创建配置配置文件

创建sqlMapConfig.xml

```
<configuration>
    <!-- 数据库连接信息 -->
    <property name="driverClass" value="com.mysql.jdbc.Driver"></property>
    <property name="jdbcUrl" value="jdbc:mysql:///zdy_mybatis"></property>
    <property name="user" value="root"></property>
    <property name="password" value="root"></property>

    <!-- 引入sql配置信息 -->
    <mapper resource="mapper.xml"></mapper>
</configuration>
```

mapper.xml

```

<mapper namespace="User">
    <select id="selectOne" parameterType="com.lagou.pojo.User"
resultType="com.lagou.pojo.User">
        select * from user where id = #{id} and username =#{username}
    </select>

    <select id="selectList" resultType="com.lagou.pojo.User">
        select * from user
    </select>
</mapper>

```

User实体

```

public class User {
    //主键标识
    private Integer id;
    //用户名
    private String username;

    public Integer getId() {
        return id;
    }

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

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    @Override
    public String toString() {
        return "User{" +
            "id=" + id +
            ", username='" + username + '\'' +
            '}';
    }
}

```

再创建一个Maven子工程并且导入需要用到的依赖坐标

```

<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

```

```

<maven.compiler.encoding>UTF-8</maven.compiler.encoding>
<java.version>1.8</java.version>
<maven.compiler.source>1.8</maven.compiler.source>
<maven.compiler.target>1.8</maven.compiler.target>
</properties>

<dependencies>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.17</version>
    </dependency>
    <dependency>
        <groupId>c3p0</groupId>
        <artifactId>c3p0</artifactId>
        <version>0.9.1.2</version>
    </dependency>
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.12</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.10</version>
    </dependency>
    <dependency>
        <groupId>dom4j</groupId>
        <artifactId>dom4j</artifactId>
        <version>1.6.1</version>
    </dependency>
    <dependency>
        <groupId>jaxen</groupId>
        <artifactId>jaxen</artifactId>
        <version>1.1.6</version>
    </dependency>
</dependencies>

```

Configuration

```

public class Configuration {

    //数据源
    private DataSource dataSource;
    //map集合: key:statementId    value:MappedStatement
    private Map<String, MappedStatement> mappedStatementMap = new
    HashMap<String, MappedStatement>();
}

```

```
public DataSource getDataSource() {
    return dataSource;
}

public void setDataSource(DataSource dataSource) {
    this.dataSource = dataSource;
}

public Map<String, MappedStatement> getMappedStatementMap() {
    return mappedStatementMap;
}

public void setMappedStatementMap(Map<String, MappedStatement>
mappedStatementMap) {
    this.mappedStatementMap = mappedStatementMap;
}
}
```

MappedStatement

```
public class MappedStatement {
    //id
    private Integer id;
    //sql语句
    private String sql;
    //输入参数
    private Class<?> paramterType;
    //输出参数
    private Class<?> resultType;

    public Integer getId() {
        return id;
    }

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

    public String getSql() {
        return sql;
    }

    public void setSql(String sql) {
        this.sql = sql;
    }

    public Class<?> getParamterType() {
        return paramterType;
    }
}
```

```

    }

    public void setParamterType(Class<?> paramterType) {
        this.paramterType = paramterType;
    }

    public Class<?> getResultType() {
        return resultType;
    }

    public void setResultType(Class<?> resultType) {
        this.resultType = resultType;
    }
}

```

Resources

```

public class Resources {
    public static InputStream getResourceAsSteam(String path){
        InputStream resourceAsStream =
Resources.class.getResourceAsStream(path);
        return resourceAsStream;
    }
}

```

SqlSessionFactoryBuilder

```

public class SqlSessionFactoryBuilder {

    private Configuration configuration;

    public SqlSessionFactoryBuilder() {
        this.configuration = new Configuration();
    }

    public SqlSessionFactory build(InputStream inputStream) throws
DocumentException, PropertyVetoException, ClassNotFoundException {
        //1.解析配置文件，封装Configuration
        XMLConfigerBuilder xmlConfigerBuilder = new
XMLConfigerBuilder(configuration);
        Configuration configuration =
xmlConfigerBuilder.parseConfiguration(inputStream);

        //2.创建sqlSessionFactory
        SqlSessionFactory sqlSessionFactory = new
DefaultSqlSessionFactory(configuration);

        return sqlSessionFactory;
    }
}

```

```
    }
}
```

XMLConfigerBuilder

```
public class XMLConfigerBuilder {

    private Configuration configuration;

    public XMLConfigerBuilder(Configuration configuration) {
        this.configuration = new Configuration();
    }

    public Configuration parseConfiguration(InputStream inputStream) throws
DocumentException, PropertyVetoException, ClassNotFoundException {
        Document document = new SAXReader().read(inputStream);
        //<configuration>
        Element rootElement = document.getRootElement();
        List<Element> propertyElements =
rootElement.selectNodes("//property");
        Properties properties = new Properties();
        for (Element propertyElement : propertyElements) {
            String name = propertyElement.attributeValue("name");
            String value = propertyElement.attributeValue("value");
            properties.setProperty(name,value);
        }
        //连接池
        ComboPooledDataSource comboPooledDataSource = new
ComboPooledDataSource();

        comboPooledDataSource.setDriverClass(properties.getProperty("driverClass"));
        comboPooledDataSource.setJdbcUrl(properties.getProperty("jdbcUrl"));
        comboPooledDataSource.setUser(properties.getProperty("username"));
        comboPooledDataSource.setPassword(properties.getProperty("password"));

        //填充configuration
        configuration.setDataSource(comboPooledDataSource);

        //mapper部分
        List<Element> mapperElements = rootElement.selectNodes("//mapper");
        XMLMapperBuilder xmlMapperBuilder = new
XMLMapperBuilder(configuration);
        for (Element mapperElement : mapperElements) {
            String mapperPath = mapperElement.attributeValue("resource");
            InputStream resourceAsStream =
Resources.getResourceAsStream(mapperPath);
            xmlMapperBuilder.parse(resourceAsStream);
        }
    }
}
```

```
    }

    return configuration;
```

```
}
```

XMLMapperBuilder

```
public class XMLMapperBuilder {

    private Configuration configuration;

    public XMLMapperBuilder(Configuration configuration) {
        this.configuration = configuration;
    }

    public void parse(InputStream inputStream) throws DocumentException,
    ClassNotFoundException {
        Document document = new SAXReader().read(inputStream);
        Element rootElement = document.getRootElement();
        String namespace = rootElement.attributeValue("namespace");
        List<Element> select = rootElement.selectNodes("select");

        for (Element element : select) {
            //id的值
            String id = element.attributeValue("id");
            String paramterType = element.attributeValue("paramterType");
            String resultType = element.attributeValue("resultType");
            //输入参数class
            Class<?> paramterTypeClass = getClassType(paramterType);
            //返回结果class
            Class<?> resultTypeClass = getClassType(resultType);
            //statementId
            String key = namespace +"."+id;
            //sql语句
            String textTrim = element.getTextTrim();

            //封装mappedStatement
            MappedStatement mappedStatement = new MappedStatement();
            mappedStatement.setId(id);
            mappedStatement.setParamterType(paramterTypeClass);
            mappedStatement.setResultType(resultTypeClass);
            mappedStatement.setSql(textTrim);
            //填充configuration
            configuration.getMappedStatementMap().put(key,mappedStatement);
        }
    }
}
```

```
    private Class<?> getClassType(String paramterType) throws  
ClassNotFoundException {  
  
        Class<?> aClass = Class.forName(paramterType);  
        return aClass;  
  
    }  
}
```

sqlSessionFactory接口及DefaultSqlSessionFactory实现类

```
public interface SqlSessionFactory {  
  
    public SqlSession openSession();  
}
```

```
public class DefaultSqlSessionFactory implements SqlSessionFactory {  
  
    private Configuration configuration;  
  
    public DefaultSqlSessionFactory(Configuration configuration) {  
        this.configuration = configuration;  
    }  
  
    public SqlSession openSession(){  
        return new DefaultSqlSession(configuration);  
    }  
}
```

sqlSession接口及DefaultSqlSession实现类

```
public interface SqlSession {  
  
    public <E> List<E> selectList(String statementId, Object... param) throws  
Exception;  
    public <T> T selectOne(String statementId, Object... params) throws  
Exception;  
    public void close() throws SQLException;  
}
```

```
public class DefaultSqlSession implements SqlSession {  
  
    private Configuration configuration;  
  
    public DefaultSqlSession(Configuration configuration) {  
        this.configuration = configuration;  
    }
```

```

    }

    //处理器对象
    private Executor simpleExecutor = new SimpleExecutor();

    public <E> List<E> selectList(String statementId, Object... param) throws
Exception{
        MappedStatement mappedStatement =
configuration.getMappedStatementMap().get(statementId);

        List<E> query = simpleExecutor.query(configuration, mappedStatement,
param);

        return query;
    }

    //selectOne中调用selectList
    public <T> T selectOne(String statementId, Object... params) throws
Exception {

        List<Object> objects = selectList(statementId, params);
        if(objects.size() ==1){
            return (T) objects.get(0);
        }else {
            throw new RuntimeException("返回结果过多");
        }
    }

    public void close() throws SQLException {
        simpleExecutor.close();
    }
}

```

Executor

```

public interface Executor {

    <E> List<E> query(Configuration configuration, MappedStatement
mappedStatement, Object[] param) throws Exception;

    void close() throws SQLException;
}

```

SimpleExecutor

```

public class SimpleExecutor implements Executor {

```

```
private Connection connection = null;

public <E> List<E> query(Configuration configuration, MappedStatement mappedStatement, Object[] param) throws SQLException, NoSuchFieldException, IllegalAccessException, InstantiationException, IntrospectionException, InvocationTargetException {

    //获取连接
    connection = configuration.getDataSource().getConnection();

    // select * from user where id = #{id} and username = #{username}
    String sql = mappedStatement.getSql();

    // 对sql进行处理
    BoundSql boundsql = getBoundSql(sql);

    // select * from where id = ? and username = ?
    String finalSql = boundsql.getSqlText();

    //获取传入参数类型
    Class<?> paramterType = mappedStatement.getParamterType();

    //获取预编译preparedStatement对象
    PreparedStatement preparedStatement =
connection.prepareStatement(finalSql);
    List<ParameterMapping> parameterMappingList =
boundsql.getParameterMappingList();

    for (int i = 0; i < parameterMappingList.size(); i++) {
        ParameterMapping parameterMapping = parameterMappingList.get(i);
        String name = parameterMapping.getName();

        //反射
        Field declaredField = paramterType.getDeclaredField(name);
        declaredField.setAccessible(true);
        //参数的值
        Object o = declaredField.get(param[0]);
        //给占位符赋值
        preparedStatement.setObject(i+1,o);
    }

    ResultSet resultSet = preparedStatement.executeQuery();
    Class<?> resultType = mappedStatement.getResultType();
    ArrayList<E> results = new ArrayList<E>();
    while (resultSet.next()){
        ResultSetMetaData metaData = resultSet.getMetaData();
        ...
```

```
E o = (E) resultType.newInstance();
int columnCount = metaData.getColumnCount();
for (int i = 1; i <= columnCount; i++) {
    //属性名
    String columnName = metaData.getColumnName(i);
    //属性值
    Object value = resultSet.getObject(columnName);

    //创建属性描述器，为属性生成读写方法
    PropertyDescriptor propertyDescriptor = new
PropertyDescriptor(columnName, resultType);
    //获取写方法
    Method writeMethod = propertyDescriptor.getWriteMethod();
    //向类中写入值
    writeMethod.invoke(o,value);

}
results.add(o);

}

return results;

}

@Override
public void close() throws SQLException {
    connection.close();
}

private BoundSql getBoundSql(String sql) {
    // 标记处理类：主要是配合通用标记解析器GenericTokenParser类完成对配置文件等的解析工作，其中TokenHandler主要完成处理
    ParameterMappingTokenHandler parameterMappingTokenHandler = new
ParameterMappingTokenHandler();
    //GenericTokenParser：通用的标记解析器，完成了代码片段中的占位符的解析，然后再根据给定的标记处理器（TokenHandler）来进行表达式的处理
    //三个参数：分别为openToken（开始标记）、closeToken（结束标记）、handler（标记处理器）
    GenericTokenParser genericTokenParser = new GenericTokenParser("#
{", "}", parameterMappingTokenHandler);
    String parse = genericTokenParser.parse(sql);

    List<ParameterMapping> parameterMappings =
parameterMappingTokenHandler.getParameterMappings();

    BoundSql boundSql = new BoundSql(parse, parameterMappings);

    return boundSql;
}
```

```
}
```

BoundSql

```
public class BoundSql {  
  
    //解析过后的sql语句  
    private String sqlText;  
  
    //解析出来的参数  
    private List<ParameterMapping> parameterMappingList = new  
ArrayList<ParameterMapping>();  
  
    public BoundSql(String sqlText, List<ParameterMapping>  
parameterMappingList) {  
        this.sqlText = sqlText;  
        this.parameterMappingList = parameterMappingList;  
    }  
  
    public String getSqlText() {  
        return sqlText;  
    }  
  
    public void setSqlText(String sqlText) {  
        this.sqlText = sqlText;  
    }  
  
    public List<ParameterMapping> getParameterMappingList() {  
        return parameterMappingList;  
    }  
  
    public void setParameterMappingList(List<ParameterMapping>  
parameterMappingList) {  
        this.parameterMappingList = parameterMappingList;  
    }  
}
```

1.5 自定义框架优化

通过上述我们的自定义框架，我们解决了JDBC操作数据库带来的一些问题：例如频繁创建释放数据库连接，硬编码，手动封装返回结果集等问题，但是现在我们继续来分析刚刚完成的自定义框架代码，有没有什么问题？

问题如下：

- dao的实现类中存在重复的代码，整个操作的过程模板重复（创建sqlsession，调用sqlsession方法，关闭sqlsession）
- dao的实现类中存在硬编码，调用sqlsession的方法时，参数 statement的id硬编码

解决：使用代理模式来创建接口的代理对象

```

@Test
public void test2() throws Exception{
    InputStream resourceAsStream = Resources.getResourceAsStream( path: "sqlMapConfig.xml");
    SqlSessionFactory build = new SqlSessionFactoryBuilder().build(resourceAsStream);
    SqlSession sqlSession = build.openSession();
    User user = new User();
    user.setId(1);
    user.setUsername("tom");

    //代理对象
    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

    User user1 = userMapper.selectOne(user);

    System.out.println(user1);

    sqlSession.close();

}

```

在sqlSession中添加方法

```

public interface SqlSession {
    public <T> T getMapper(Class<?> mapperClass);

```

实现类

```

@Override
public <T> T getMapper(Class<?> mapperClass) {
    T o = (T) Proxy.newProxyInstance(mapperClass.getClassLoader(), new Class[] {mapperClass}, new InvocationHandler() {
        @Override
        public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
            // selectOne
            String methodName = method.getName();
            // className:namespace
            String className = method.getDeclaringClass().getName();

            //statementid
            String key = className + "." + methodName;

            MappedStatement mappedStatement =
configuration.getMappedStatementMap().get(key);

            Type genericReturnType = method.getGenericReturnType();

            ArrayList arrayList = new ArrayList<> ();
            //判断是否实现泛型类型参数化
            if(genericReturnType instanceof ParameterizedType){
                return selectList(key,args);
            }
        }
    });
}

```

```
        }

        return selectOne(key,args);
    }
});

return o;

}
```

第二部分 Mybatis相关概念

2.1 对象/关系数据库映射(ORM)

ORM全称Object/Relation Mapping：表示对象-关系映射的缩写

ORM完成面向对象的编程语言到关系数据库的映射。当ORM框架完成映射后，程序员既可以利用面向对象程序设计语言的简单易用性，又可以利用关系数据库的技术优势。ORM把关系数据库包装成面向对象的模型。ORM框架是面向对象设计语言与关系数据库发展不同步时的中间解决方案。采用ORM框架后，应用程序不再直接访问底层数据库，而是以面向对象的放松来操作持久化对象，而ORM框架则将这些面向对象的操作转换成底层SQL操作。ORM框架实现的效果：把对持久化对象的保存、修改、删除等操作，转换为对数据库的操作

2.2 Mybatis简介

MyBatis 是一款优秀的基于ORM的半自动轻量级持久层框架，它支持定制化 SQL、存储过程以及高级映射。MyBatis 避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集。MyBatis 可以使用简单的 XML 或注解来配置和映射原生类型、接口和 Java 的 POJO (Plain Old Java Objects, 普通老式 Java 对象) 为数据库中的记录。

2.3 MyBatis历史

1. 原是Apache的一个开源项目iBatis, 2010年6月这个项目由Apache Software Foundation 迁移到了 Google Code, 随着开发团队转投Google Code旗下, iBatis3.x正式更名为MyBatis , 代码于 2013年11月迁移到Github (下载地址见后) 。
2. iBatis一词来源于“internet”和“abatis”的组合，是一个基于Java的持久层框架。

2.4 MyBatis优势

MyBatis是一个半自动化的持久化层框架。对开发人员而言，核心sql还是需要自己优化，sql 和 java 编码分开，功能边界清晰，一个专注业务、一个专注数据。

分析图示如下：



第三部分 Mybatis基本应用

3.1 快速入门

3.1.1 MyBatis开发步骤

MyBatis官网地址：<http://www.mybatis.org/mybatis-3/>

REFERENCE DOCUMENTATION

- Introduction**
- Getting Started
- Configuration XML
- Mapper XML Files
- Dynamic SQL
- Java API
- SQL Builder Class
- Logging
- PROJECT DOCUMENTATION
- Project Information
- CI Management
- Dependencies
- Dependency Information
- Distribution Management
- About
- Issue Management
- Licenses
- Mailing Lists

Introduction

What is MyBatis?

MyBatis is a first class persistence framework with support for custom SQL, stored procedures and advanced mappings. MyBatis eliminates almost all of the JDBC code and manual setting of parameters and retrieval of results. MyBatis can use simple XML or Annotations for configuration and map primitives, Map interfaces and Java POJOs (Plain Old Java Objects) to database records.

Help make this documentation better...

If you find this documentation lacking in any way, or missing documentation for a feature, then the best thing to do is learn about it and then write the documentation yourself!

Sources of this manual are available in xdoc format at [project's Git](#). Fork the repository, update them and send a pull request.

You're the best author of this documentation, people like you have to read it!

Translations

Users can read about MyBatis in following translations:



Do you want to read about MyBatis in your own native language? File an issue providing patches with your mother tongue documentation!

MyBatis开发步骤：

- ①添加MyBatis的坐标
- ②创建user数据表
- ③编写User实体类
- ④编写映射文件UserMapper.xml
- ⑤编写核心文件SqlMapConfig.xml
- ⑥编写测试类

3.1.2 环境搭建

1)导入MyBatis的坐标和其他相关坐标

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.encoding>UTF-8</maven.compiler.encoding>
    <java.version>1.8</java.version>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
</properties>

<!--mybatis坐标-->
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis</artifactId>
    <version>3.4.5</version>
</dependency>
<!--mysql驱动坐标-->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.6</version>
    <scope>runtime</scope>
</dependency>
<!--单元测试坐标-->
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
</dependency>
```

2) 创建user数据表

| 名 | 类型 | 长度 | 小数点 | 允许空值(| |
|----------|---------|----|-----|-------------------------------------|---|
| id | int | 11 | 0 | <input type="checkbox"/> | 1 |
| username | varchar | 50 | 0 | <input checked="" type="checkbox"/> | |
| password | varchar | 50 | 0 | <input checked="" type="checkbox"/> | |

3) 编写User实体

```
public class User {
    private int id;
    private String username;
    private String password;
    //省略get/set方法
}
```

4)编写UserMapper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="userMapper">
  <select id="findAll" resultType="com.lagou.domain.User">
    select * from User
  </select>
</mapper>
```

5) 编写MyBatis核心文件

```
<?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>
  <environments default="development">
    <environment id="development">
      <transactionManager type="JDBC"/>
      <dataSource type="POOLED">
        <property name="driver" value="com.mysql.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql:///test"/>
        <property name="username" value="root"/>
        <property name="password" value="root"/>
      </dataSource>
    </environment>
  </environments>

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

3.1.3 编写测试代码

```

//加载核心配置文件
InputStream resourceAsStream =
Resources.getResourceAsStream("SqlMapConfig.xml");
//获得sqlSession工厂对象
SqlSessionFactory sqlSessionFactory = new
    SqlSessionFactoryBuilder().build(resourceAsStream);
//获得sqlSession对象
SqlSession sqlSession = sqlSessionFactory.openSession();
//执行sql语句
List<User> userList = sqlSession.selectList("userMapper.findAll");
//打印结果
System.out.println(userList);
//释放资源
sqlSession.close();

```

3.1.4 mybatis增删改查操作

MyBatis的插入数据操作

1)编写UserMapper映射文件

```

<mapper namespace="userMapper">
<insert id="add" parameterType="com.lagou.domain.User">
    insert into user values(#{id},#{username},#{password})
</insert>
</mapper>

```

2)编写插入实体User的代码

```

InputStream resourceAsStream =
Resources.getResourceAsStream("SqlMapConfig.xml");
SqlSessionFactory sqlSessionFactory = new
    SqlSessionFactoryBuilder().build(resourceAsStream);
SqlSession sqlSession = sqlSessionFactory.openSession();
int insert = sqlSession.insert("userMapper.add", user);
System.out.println(insert);
//提交事务
sqlSession.commit();
sqlSession.close();

```

3)插入操作注意问题

- 插入语句使用insert标签
- 在映射文件中使用parameterType属性指定要插入的数据类型
- Sql语句中使用#{实体属性名}方式引用实体中的属性值
- 插入操作使用的API是sqlSession.insert("命名空间.id",实体对象);

- 插入操作涉及数据库数据变化，所以要使用sqlSession对象显示的提交事务，即sqlSession.commit()

MyBatis的修改数据操作

1)编写UserMapper映射文件

```
<mapper namespace="userMapper">
    <update id="update" parameterType="com.lagou.domain.User">
        update user set username=#{username},password=#{password} where id=#{id}
    </update>
</mapper>
```

2)编写修改实体User的代码

```
InputStream resourceAsStream =
Resources.getResourceAsStream("SqlMapConfig.xml");
SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(resourceAsStream);
SqlSession sqlSession = sqlSessionFactory.openSession();
int update = sqlSession.update("userMapper.update", user);
System.out.println(update);
sqlSession.commit();
sqlSession.close();
```

3)修改操作注意问题

- 修改语句使用update标签
- 修改操作使用的API是sqlSession.update("命名空间.id",实体对象);

MyBatis的删除数据操作

1)编写UserMapper映射文件

```
<mapper namespace="userMapper">
    <delete id="delete" parameterType="java.lang.Integer">
        delete from user where id=#{id}
    </delete>
</mapper>
```

2)编写删除数据的代码

```

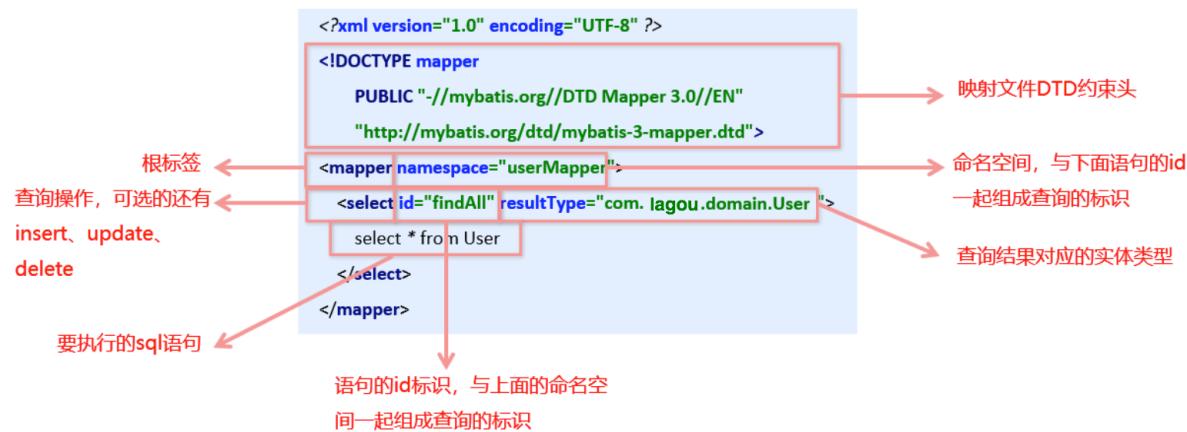
InputStream resourceAsStream =
Resources.getResourceAsStream("SqlMapConfig.xml");
SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(resourceAsStream);
SqlSession sqlSession = sqlSessionFactory.openSession();
int delete = sqlSession.delete("userMapper.delete",3);
System.out.println(delete);
sqlSession.commit();
sqlSession.close();

```

3)删除操作注意问题

- 删除语句使用delete标签
- Sql语句中使用#{任意字符串}方式引用传递的单个参数
- 删除操作使用的API是sqlSession.delete("命名空间.id",Object);

3.1.5 入门映射配置文件分析



3.1.6 入门核心配置文件分析

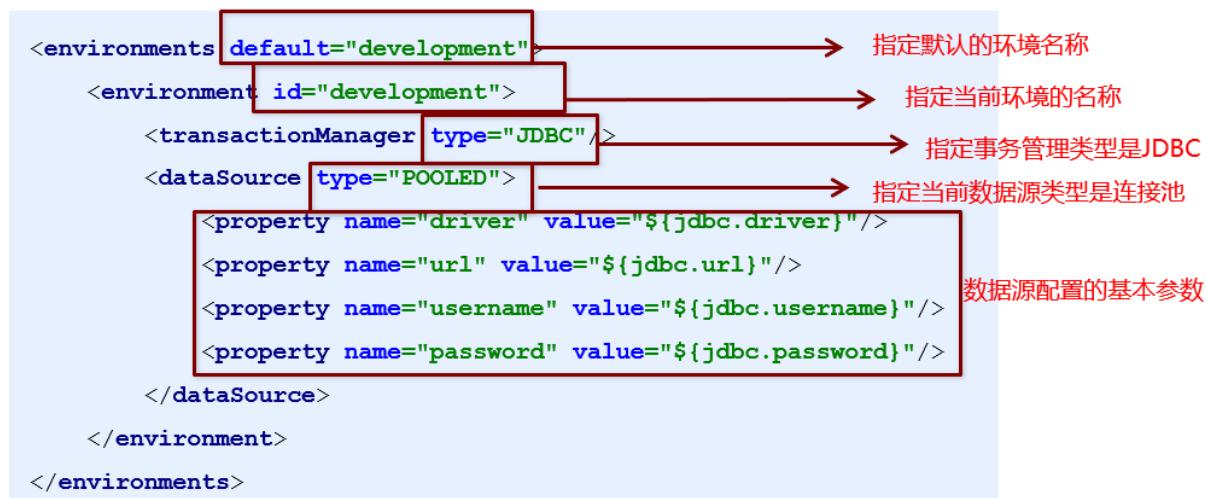
MyBatis核心配置文件层级关系

- configuration 配置
 - properties 属性
 - settings 设置
 - typeAliases 类型别名
 - typeHandlers 类型处理器
 - objectFactory 对象工厂
 - plugins 插件
 - environments 环境
 - environment 环境变量
 - transactionManager 事务管理器
 - dataSource 数据源
 - databaseIdProvider 数据库厂商标识
 - mappers 映射器

MyBatis常用配置解析

1)environments标签

数据库环境的配置，支持多环境配置



其中，事务管理器 (transactionManager) 类型有两种：

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

其中，数据源 (dataSource) 类型有三种：

- UNPOOLED：这个数据源的实现只是每次被请求时打开和关闭连接。
- POOLED：这种数据源的实现利用“池”的概念将 JDBC 连接对象组织起来。
- JNDI：这个数据源的实现是为了能在如 EJB 或应用服务器这类容器中使用，容器可以集中或在外部配置数据源，然后放置一个 JNDI 上下文的引用。

2)mapper标签

该标签的作用是加载映射的，加载方式有如下几种：

- 使用相对于类路径的资源引用，例如：
 - 使用完全限定资源定位符 (URL)，例如：
 - 使用映射器接口实现类的完全限定类名，例如：
- ```
<mapper class="org.mybatis.builbu"
```
- 将包内的映射器接口实现全部注册为映射器，例如：

#### 3.1.7 Mybatis相应API介绍

##### SqlSession工厂构建器SqlSessionFactoryBuilder

常用API： SqlSessionFactory build(InputStream inputStream)

通过加载mybatis的核心文件的输入流的形式构建一个SqlSessionFactory对象

```
String resource = "org/mybatis/builder/mybatis-config.xml";
InputStream inputStream = Resources.getResourceAsStream(resource);
SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
SqlSessionFactory factory = builder.build(inputStream);
```

其中，Resources 工具类，这个类在 org.apache.ibatis.io 包中。Resources 类帮助你从类路径下、文件系统或一个 web URL 中加载资源文件。

### SqlSession工厂对象SqlSessionFactory

SqlSessionFactory 有多个方法创建SqlSession 实例。常用的有如下两个：

| 方法                              | 解释                                                  |
|---------------------------------|-----------------------------------------------------|
| openSession()                   | 会默认开启一个事务，但事务不会自动提交，也就意味着需要手动提交该事务，更新操作数据才会持久化到数据库中 |
| openSession(boolean autoCommit) | 参数为是否自动提交，如果设置为true，那么不需要手动提交事务                     |

### SqlSession会话对象

SqlSession 实例在 MyBatis 中是非常强大的一个类。在这里你会看到所有执行语句、提交或回滚事务和获取映射器实例的方法。

执行语句的方法主要有：

```
<T> T selectOne(String statement, Object parameter)
<E> List<E> selectList(String statement, Object parameter)
int insert(String statement, Object parameter)
int update(String statement, Object parameter)
int delete(String statement, Object parameter)
```

操作事务的方法主要有：

```
void commit()
void rollback()
```

## 3.2 Mybatis的Dao层实现

### 3.2.1 传统开发方式

编写UserDao接口

```
public interface UserDao {
 List<User> findAll() throws IOException;
}
```

编写UserDaoImpl实现

```
public class UserDaoImpl implements UserDao {
 public List<User> findAll() throws IOException {
 InputStream resourceAsStream =
 Resources.getResourceAsStream("SqlMapConfig.xml");
 SqlSessionFactory sqlSessionFactory = new
 SqlSessionFactoryBuilder().build(resourceAsStream);
 SqlSession sqlSession = sqlSessionFactory.openSession();
 List<User> userList = sqlSession.selectList("userMapper.findAll");
 sqlSession.close();
 return userList;
 }
}
```

测试传统方式

```
@Test
public void testTraditionDao() throws IOException {
 UserDao userDao = new UserDaoImpl();
 List<User> all = userDao.findAll();
 System.out.println(all);
}
```

### 3.2.2 代理开发方式

代理开发方式介绍

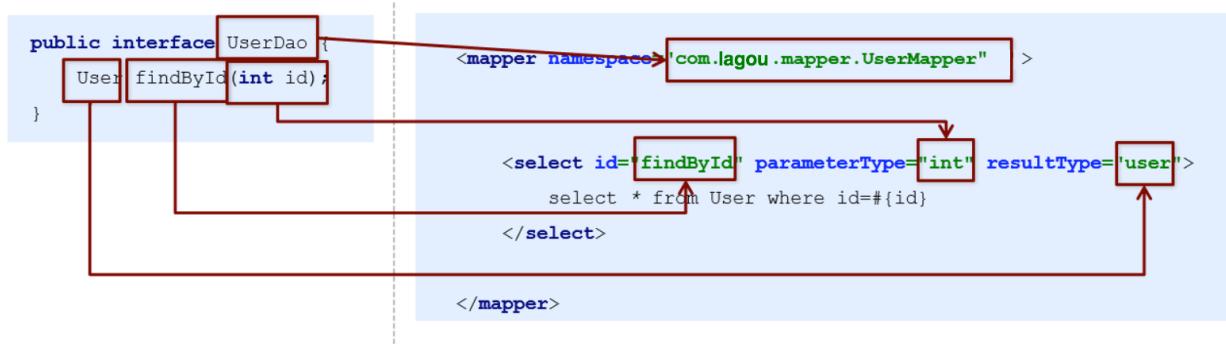
采用 Mybatis 的代理开发方式实现 DAO 层的开发

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接口的全限定名相同
- 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的结果resultType的类型相同

编写UserMapper接口



测试代理方式

```

@Test
public void testProxyDao() throws IOException {
 InputStream resourceAsStream =
 Resources.getResourceAsStream("SqlMapConfig.xml");
 SqlSessionFactory sqlSessionFactory = new
 SqlSessionFactoryBuilder().build(resourceAsStream);
 SqlSession sqlSession = sqlSessionFactory.openSession();
 //获得MyBatis框架生成的UserMapper接口的实现类
 UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
 User user = userMapper.findById(1);
 System.out.println(user);
 sqlSession.close();
}

```

## 第四部分 Mybatis配置文件深入

### 4.1 SqlMapConfig.xml

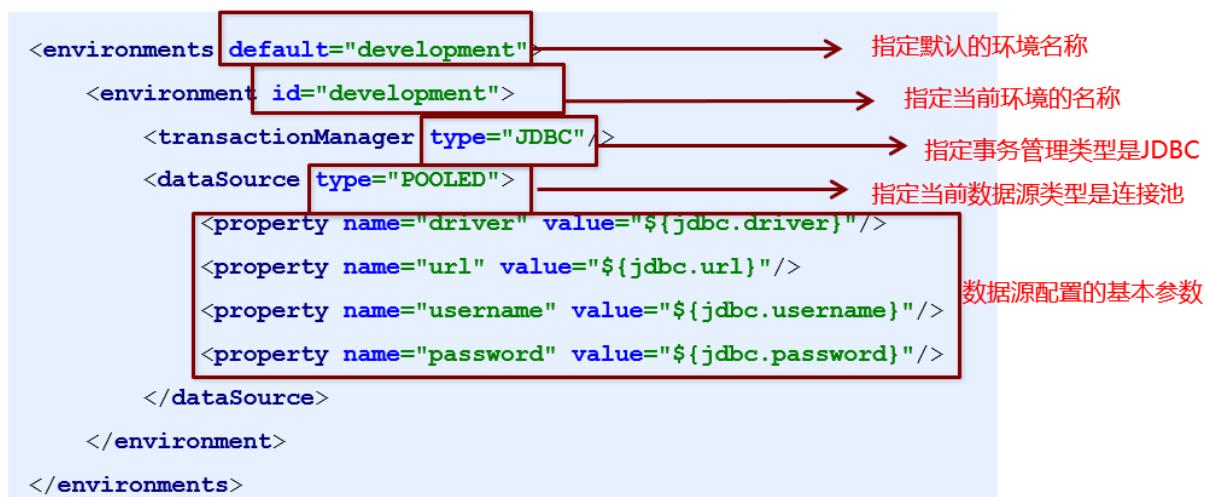
#### 4.1.1 MyBatis核心配置文件层级关系

- configuration 配置
  - properties 属性
  - settings 设置
  - typeAliases 类型别名
  - typeHandlers 类型处理器
  - objectFactory 对象工厂
  - plugins 插件
  - environments 环境
    - environment 环境变量
      - transactionManager 事务管理器
      - dataSource 数据源
  - databaseIdProvider 数据库厂商标识
  - mappers 映射器

#### 4.1.2 MyBatis常用配置解析

##### 1)environments标签

数据库环境的配置，支持多环境配置



其中，事务管理器 (transactionManager) 类型有两种：

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

其中，数据源 (dataSource) 类型有三种：

- UNPOOLED：这个数据源的实现只是每次被请求时打开和关闭连接。
- POOLED：这种数据源的实现利用“池”的概念将 JDBC 连接对象组织起来。
- JNDI：这个数据源的实现是为了能在如 EJB 或应用服务器这类容器中使用，容器可以集中或在外部配置数据源，然后放置一个 JNDI 上下文的引用。

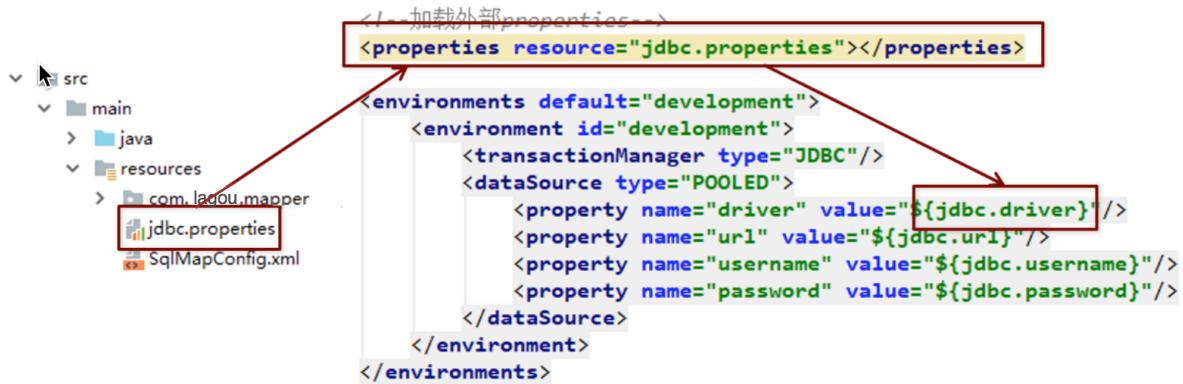
## 2) mapper标签

该标签的作用是加载映射的，加载方式有如下几种：

- 使用相对于类路径的资源引用，例如：
- 使用完全限定资源定位符 (URL)，例如：
- 使用映射器接口实现类的完全限定类名，例如：
- 将包内的映射器接口实现全部注册为映射器，例如：

## 3) Properties标签

习惯将数据源的配置信息单独抽取成一个properties文件，该标签可以加载额外配置的properties文件



#### 4) typeAliases标签

类型别名是为Java 类型设置一个短的名字。原来的类型名称配置如下

```

<select id="findAll" resultType="com.lagou.domain.User">
 select * from User
</select>

```

User全限定名称

配置typeAliases，为com.lagou.domain.User定义别名为user

|                                                                                                                      |                                                                                                 |
|----------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------|
| <pre> &lt;typeAliases&gt;   &lt;typeAlias type="com.lagou.domain.User" alias="user"/&gt; &lt;/typeAliases&gt; </pre> | <pre> &lt;select id="findAll" resultType="user"&gt;   select * from User &lt;/select&gt; </pre> |
|----------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------|

上面我们是自定义的别名， mybatis框架已经为我们设置好的一些常用的类型的别名

| 别名      | 数据类型    |
|---------|---------|
| string  | String  |
| long    | Long    |
| int     | Integer |
| double  | Double  |
| boolean | Boolean |
| ... ... | ... ... |

## 4.2 mapper.xml

### 4.2.1 动态sql语句

动态sql语句概述

Mybatis 的映射文件中，前面我们的 SQL 都是比较简单的，有些时候业务逻辑复杂时，我们的 SQL 是动态变化的，此时在前面的学习中我们的 SQL 就不能满足要求了。

参考的官方文档，描述如下：

## Dynamic SQL

One of the most powerful features of MyBatis has always been its Dynamic SQL capabilities. If you have any experience with JDBC or any similar framework, you understand how painful it is to conditionally concatenate strings of SQL together, making sure not to forget spaces or to omit a comma at the end of a list of columns. Dynamic SQL can be downright painful to deal with.

While working with Dynamic SQL will never be a party, MyBatis certainly improves the situation with a powerful Dynamic SQL language that can be used within any mapped SQL statement.

The Dynamic SQL elements should be familiar to anyone who has used JSTL or any similar XML based text processors. In previous versions of MyBatis, there were a lot of elements to know and understand. MyBatis 3 greatly improves upon this, and now there are less than half of those elements to work with. MyBatis employs powerful OGNL based expressions to eliminate most of the other elements:

- if
- choose (when, otherwise)
- trim (where, set)
- foreach

## 动态 SQL 之

我们根据实体类的不同取值，使用不同的 SQL语句来进行查询。比如在 id如果不为空时可以根据id查询，如果username 不同空时还要加入用户名作为条件。这种情况在我们的多条件组合查询中经常会碰到。

```
<select id="findByCondition" parameterType="user" resultType="user">
 select * from User
 <where>
 <if test="id!=0">
 and id=#{id}
 </if>
 <if test="username!=null">
 and username=#{username}
 </if>
 </where>
</select>
```

当查询条件id和username都存在时，控制台打印的sql语句如下：

```
...
//获得MyBatis框架生成的UserMapper接口的实现类
UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
User condition = new User();
condition.setId(1);
condition.setUsername("lucy");
User user = userMapper.findByCondition(condition);
...
```

```
- Created connection 586084331.
- Setting autocommit to false on JDBC Connection [com.mysql.jdbc.
- ==> Preparing: select * from User WHERE id=? and username=?
- ==> Parameters: 1(Integer), lucy(String)
- <== Total: 1
```

当查询条件只有id存在时，控制台打印的sql语句如下：

```
...
//获得MyBatis框架生成的UserMapper接口的实现类
UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
User condition = new User();
condition.setId(1);
User user = userMapper.findByCondition(condition);
...
```

```
- Setting autocommit to false on JDBC Connection [com.mysql.
- ==> Preparing: select * from User WHERE id=?
- ==> Parameters: 1(Integer)
- <== Total: 1
```

## 动态 SQL 之

循环执行sql的拼接操作，例如：SELECT \* FROM USER WHERE id IN (1,2,5)。

```
<select id="findByIds" parameterType="list" resultType="user">
 select * from User
 <where>
 <foreach collection="array" open="id in(" close=")" item="id"
 separator=", ">>
 #{id}
 </foreach>
 </where>
</select>
```

测试代码片段如下：

```
...
//获得MyBatis框架生成的UserMapper接口的实现类
UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
int[] ids = new int[]{2,5};
List<User> userList = userMapper.findByIds(ids);
System.out.println(userList);
...
```

```
11:21:02,237 DEBUG findByIds:159 - ==> Preparing: select * from User WHERE id in(? , ?)
11:21:02,262 DEBUG findByIds:159 - ==> Parameters: 2(Integer), 5(Integer)
11:21:02,280 DEBUG findByIds:159 - <== Total: 2
[User{id=2, username='tom', password='123'}, User{id=5, username='lucylucy', password='123'}]
```

foreach标签的属性含义如下：

标签用于遍历集合，它的属性：

- collection：代表要遍历的集合元素，注意编写时不要写#{}；
- open：代表语句的开始部分；
- close：代表结束部分；
- item：代表遍历集合的每个元素，生成的变量名；
- separator：代表分隔符；

## SQL片段抽取

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

```
<!--抽取sql片段简化编写-->
<sql id="selectUser" select * from User</sql>
<select id="findById" parameterType="int" resultType="user">
 <include refid="selectUser"></include> where id=#{id}
</select>
<select id="findByIds" parameterType="list" resultType="user">
 <include refid="selectUser"></include>
 <where>
 <foreach collection="array" open="id in(" close=")" item="id"
separator=", ">
 #{id}
 </foreach>
 </where>
</select>
```

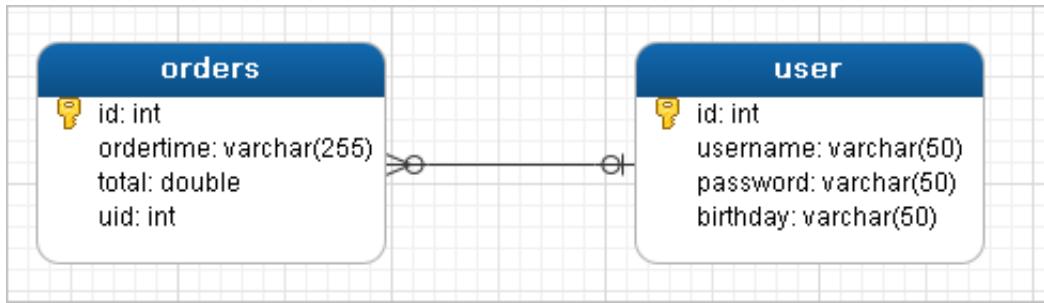
# 第五部分 Mybatis复杂映射开发

## 5.1 一对一查询

### 一对一查询的模型MapperScannerConfigurer

用户表和订单表的关系为，一个用户有多个订单，一个订单只从属于一个用户

一对一查询的需求：查询一个订单，与此同时查询出该订单所属的用户信息



### 一对一查询的语句

对应的sql语句： select \* from orders o,user u where o.uid=u.id;

查询的结果如下：

| 信息 |    |            |       |     |     |          |          |               |
|----|----|------------|-------|-----|-----|----------|----------|---------------|
|    | id | ordertime  | total | uid | id1 | username | password | birthday      |
| ▶  | 1  | 2018-12-12 | 3000  |     | 1   | lucy     | 123      | 1539751863457 |
|    | 2  | 2019-12-12 | 4000  |     | 1   | lucy     | 123      | 1539751863457 |
|    | 3  | 2020-12-12 | 5000  |     | 2   | tom      | 123      | 1539751863457 |

创建Order和User实体

```

public class Order {

 private int id;
 private Date ordertime;
 private double total;

 //代表当前订单从属于哪一个客户
 private User user;
}

public class User {

 private int id;
 private String username;
 private String password;
 private Date birthday;
}

```

创建OrderMapper接口

```

public interface OrderMapper {
 List<Order> findAll();
}

```

配置OrderMapper.xml

```

<mapper namespace="com.lagou.mapper.OrderMapper">
 <resultMap id="orderMap" type="com.lagou.domain.Order">
 <result column="uid" property="user.id"></result>
 <result column="username" property="user.username"></result>
 <result column="password" property="user.password"></result>
 <result column="birthday" property="user.birthday"></result>
 </resultMap>
 <select id="findAll" resultMap="orderMap">
 select * from orders o,user u where o.uid=u.id
 </select>
</mapper>

```

其中还可以配置如下：

```

<resultMap id="orderMap" type="com.lagou.domain.Order">
 <result property="id" column="id"></result>
 <result property="ordertime" column="ordertime"></result>
 <result property="total" column="total"></result>
 <association property="user" javaType="com.lagou.domain.User">
 <result column="uid" property="id"></result>
 <result column="username" property="username"></result>
 <result column="password" property="password"></result>
 <result column="birthday" property="birthday"></result>
 </association>
</resultMap>

```

测试结果

```

OrderMapper mapper = sqlSession.getMapper(OrderMapper.class);
List<Order> all = mapper.findAll();
for(Order order : all){
 System.out.println(order);
}

```

```

09:12:24,650 DEBUG findAll:54 - ==> Preparing: select * from orders o,user u where o.uid=u.id
09:12:24,672 DEBUG findAll:54 - ==> Parameters:
09:12:24,699 DEBUG findAll:54 - <== Total: 3
Order{id=1, ordertime=Wed Dec 12 00:00:00 GMT+08:00 2018, total=3000.0, user=User{id=1, username='lucy'},
Order{id=2, ordertime=Thu Dec 12 00:00:00 GMT+08:00 2019, total=4000.0, user=User{id=1, username='lucy'},
Order{id=3, ordertime=Sat Dec 12 00:00:00 GMT+08:00 2020, total=5000.0, user=User{id=2, username='tom'},
09:12:24,706 DEBUG JdbcTransaction:54 - Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.
09:12:24,706 DEBUG JdbcTransaction:54 - Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@28ac3dc3
09:12:24,706 DEBUG PooledDataSource:54 - Returned connection 682376643 to pool.

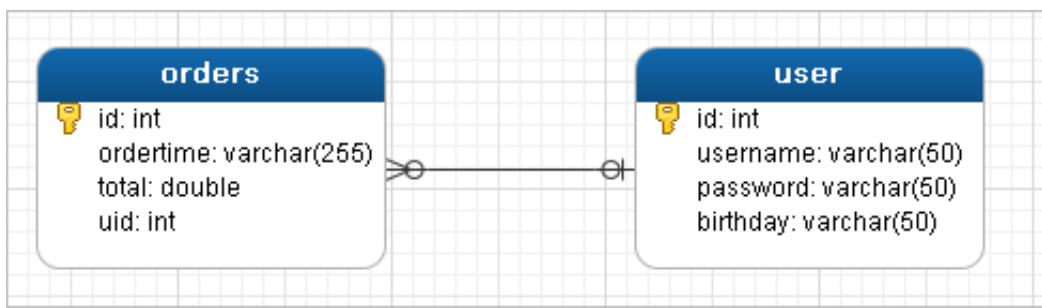
```

## 5.2 一对多查询

### 一对多查询的模型

用户表和订单表的关系为，一个用户有多个订单，一个订单只从属于一个用户

一对多查询的需求：查询所有用户，与此同时查询出该用户具有的订单



### 一对多查询的语句

对应的sql语句： select \*,o.id oid from user u left join orders o on u.id=o.uid;

查询的结果如下：

| 信息 结果1 概况 状态 |    |          |          |            |     |            |       |     |     |
|--------------|----|----------|----------|------------|-----|------------|-------|-----|-----|
|              | id | username | password | birthday   | id1 | ordertime  | total | uid | oid |
| ▶            | 1  | lucy     | 123      | 2018-12-12 | 1   | 2018-12-12 | 3000  | 1   | 1   |
|              | 1  | lucy     | 123      | 2018-12-12 | 2   | 2019-12-12 | 4000  | 1   | 2   |
|              | 2  | tom      | 123      | 2018-12-12 | 3   | 2020-12-12 | 5000  | 2   | 3   |

修改User实体

```

public class Order {

 private int id;
 private Date ordertime;
 private double total;

 //代表当前订单从属于哪一个客户
 private User user;
}

public class User {

 private int id;
 private String username;
 private String password;
 private Date birthday;
 //代表当前用户具备哪些订单
 private List<Order> orderList;
}

```

创建UserMapper接口

```

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

```

## 配置UserMapper.xml

```
<mapper namespace="com.lagou.mapper.UserMapper">
 <resultMap id="userMap" type="com.lagou.domain.User">
 <result column="id" property="id"></result>
 <result column="username" property="username"></result>
 <result column="password" property="password"></result>
 <result column="birthday" property="birthday"></result>
 <collection property="orderList" ofType="com.lagou.domain.Order">
 <result column="oid" property="id"></result>
 <result column="ordertime" property="ordertime"></result>
 <result column="total" property="total"></result>
 </collection>
 </resultMap>
 <select id="findAll" resultMap="userMap">
 select *,o.id oid from user u left join orders o on u.id=o.uid
 </select>
</mapper>
```

## 测试结果

```
UserMapper mapper = sqlSession.getMapper(UserMapper.class);
List<User> all = mapper.findAll();
for(User user : all){
 System.out.println(user.getUsername());
 List<Order> orderList = user.getOrderList();
 for(Order order : orderList){
 System.out.println(order);
 }
 System.out.println("-----");
}
```

```
10:02:27,817 DEBUG findAll:54 - ==> Preparing: select *,o.id oid from user u left join orders o on u.id=o.uid
10:02:27,843 DEBUG findAll:54 - ==> Parameters:
10:02:27,865 DEBUG findAll:54 - <== Total: 4
lucy
Order{id=1, ordertime=Wed Dec 12 00:00:00 GMT+08:00 2018, total=3000.0, user=null}
Order{id=2, ordertime=Thu Dec 12 00:00:00 GMT+08:00 2019, total=4000.0, user=null}

tom
Order{id=3, ordertime=Sat Dec 12 00:00:00 GMT+08:00 2020, total=5000.0, user=null}

haohao

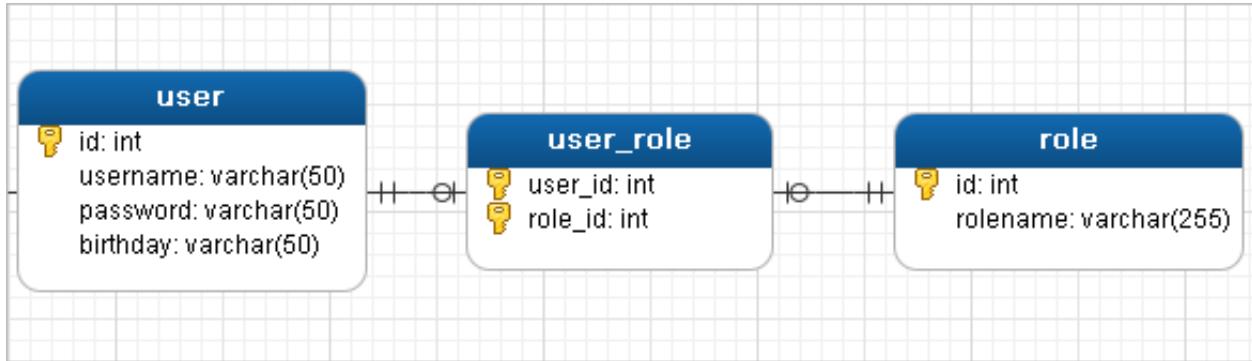
10:02:27,868 DEBUG JdbcTransaction:54 - Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@289d1c02]
10:02:27,869 DEBUG JdbcTransaction:54 - Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@289d1c02]
10:02:27,869 DEBUG PooledDataSource:54 - Returned connection 681384962 to pool.
```

## 5.3 多对多查询

### 多对多查询的模型

用户表和角色表的关系为，一个用户有多个角色，一个角色被多个用户使用

多对多查询的需求：查询用户同时查询出该用户的所有角色



## 多对多查询的语句

对应的sql语句： select u.,r.,r.id rid from user u left join user\_role ur on u.id=ur.user\_id  
inner join role r on ur.role\_id=r.id;

查询的结果如下：

| 信息 |    |          |          |            |     |          |
|----|----|----------|----------|------------|-----|----------|
|    | id | username | password | birthday   | id1 | rolename |
| ▶  | 1  | lucy     | 123      | 2018-12-12 | 1   | CEO      |
|    | 1  | lucy     | 123      | 2018-12-12 | 2   | CFO      |
|    | 2  | tom      | 123      | 2018-12-12 | 2   | CFO      |
|    | 2  | tom      | 123      | 2018-12-12 | 3   | COO      |

## 创建Role实体，修改User实体

```

public class User {
 private int id;
 private String username;
 private String password;
 private Date birthday;
 //代表当前用户具备哪些订单
 private List<Order> orderList;
 //代表当前用户具备哪些角色
 private List<Role> roleList;
}

public class Role {

 private int id;
 private String rolename;

}

```

## 添加UserMapper接口方法

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

## 配置UserMapper.xml

```
<resultMap id="userRoleMap" type="com.lagou.domain.User">
 <result column="id" property="id"></result>
 <result column="username" property="username"></result>
 <result column="password" property="password"></result>
 <result column="birthday" property="birthday"></result>
 <collection property="roleList" ofType="com.lagou.domain.Role">
 <result column="rid" property="id"></result>
 <result column="rolename" property="rolename"></result>
 </collection>
</resultMap>
<select id="findAllUserAndRole" resultMap="userRoleMap">
 select u.*,r.*,r.id rid from user u left join user_role ur on
u.id=ur.user_id
 inner join role r on ur.role_id=r.id
</select>
```

## 测试结果

```
UserMapper mapper = sqlSession.getMapper(UserMapper.class);
List<User> all = mapper.findAllUserAndRole();
for(User user : all){
 System.out.println(user.getUsername());
 List<Role> roleList = user.getRoleList();
 for(Role role : roleList){
 System.out.println(role);
 }
 System.out.println("-----");
}
```

```
10:34:36,884 DEBUG findAllUserAndRole:54 - ==> Preparing: select u.*,r.*,r.id rid from user u left
10:34:36,903 DEBUG findAllUserAndRole:54 - ==> Parameters:
lucy
Role{id=1, rolename='CEO'}
Role{id=2, rolename='CFO'}

tom
Role{id=2, rolename='CFO'}
Role{id=3, rolename='COO'}

10:34:36,937 DEBUG findAllUserAndRole:54 - <== Total: 4
10:34:36,939 DEBUG JdbcTransaction:54 - Resetting autocommit to true on JDBC Connection [com.mysql.]
```

# 第六部分 Mybatis注解开发

## 6.1 MyBatis的常用注解

这几年来注解开发越来越流行，Mybatis也可以使用注解开发方式，这样我们就可以减少编写Mapper映射文件了。我们先围绕一些基本的CRUD来学习，再学习复杂映射多表操作。

@Insert: 实现新增  
@Update: 实现更新  
@Delete: 实现删除  
@Select: 实现查询  
@Result: 实现结果集封装  
@Results: 可以与@Result一起使用，封装多个结果集  
@One: 实现一对结果集封装  
@Many: 实现一对多结果集封装

## 6.2 MyBatis的增删改查

我们完成简单的user表的增删改查的操作

```
private UserMapper userMapper;

@Before
public void before() throws IOException {
 InputStream resourceAsStream =
 Resources.getResourceAsStream("SqlMapConfig.xml");
 SqlSessionFactory sqlSessionFactory = new
 SqlSessionFactoryBuilder().build(resourceAsStream);
 SqlSession sqlSession = sqlSessionFactory.openSession(true);
 userMapper = sqlSession.getMapper(UserMapper.class);
}

@Test
public void testAdd() {
 User user = new User();
 user.setUsername("测试数据");
 user.setPassword("123");
 user.setBirthday(new Date());
 userMapper.add(user);
}

@Test
public void testUpdate() throws IOException {
 User user = new User();
 user.setId(16);
 user.setUsername("测试数据修改");
 user.setPassword("abc");
 user.setBirthday(new Date());
 userMapper.update(user);
}

@Test
public void testDelete() throws IOException {
```

```

 userMapper.delete(16);
 }
 @Test
 public void testFindById() throws IOException {
 User user = userMapper.findById(1);
 System.out.println(user);
 }
 @Test
 public void testfindAll() throws IOException {
 List<User> all = userMapper.findAll();
 for(User user : all){
 System.out.println(user);
 }
 }
}

```

修改MyBatis的核心配置文件，我们使用了注解替代的映射文件，所以我们只需要加载使用了注解的Mapper接口即可

```

<mappers>
 <!--扫描使用注解的类-->
 <mapper class="com.lagou.mapper.UserMapper"></mapper>
</mappers>

```

或者指定扫描包含映射关系的接口所在的包也可以

```

<mappers>
 <!--扫描使用注解的类所在的包-->
 <package name="com.lagou.mapper"></package>
</mappers>

```

## 6.3 MyBatis的注解实现复杂映射开发

实现复杂关系映射之前我们可以在映射文件中通过配置来实现，使用注解开发后，我们可以使用@Results注解，@Result注解，@One注解，@Many注解组合完成复杂关系的配置

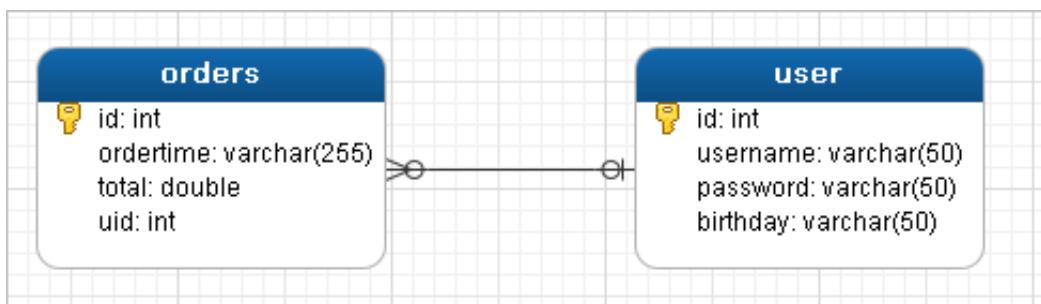
| 注解       | 说明                                                                                                                                                                            |
|----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| @Results | 代替的是标签<resultMap>该注解中可以使用单个@Result注解，也可以使用@Result集合。使用格式：@Results ({@Result () , @Result () }) 或@Results (@Result () )                                                        |
| @Result  | 代替了<id>标签和<result>标签<br>@Result中属性介绍：<br>column: 数据库的列名<br>property: 需要装配的属性名<br>one: 需要使用的@One 注解 (@Result (one=@One) () )<br>many: 需要使用的@Many 注解 (@Result (many=@many) () ) |

## 6.4 一对查询（注解）

## 一对多查询的模型

用户表和订单表的关系为，一个用户有多个订单，一个订单只从属于一个用户

一对多查询的需求：查询一个订单，与此同时查询出该订单所属的用户



## 一对多查询的语句

对应的sql语句：

```
select * from orders;
select * from user where id=查询出订单的uid;
```

查询的结果如下：

| 信息 结果1 概况 状态 |    |            |       |     |     |          |          |               |
|--------------|----|------------|-------|-----|-----|----------|----------|---------------|
|              | id | ordertime  | total | uid | id1 | username | password | birthday      |
| ▶            | 1  | 2018-12-12 | 3000  |     | 1   | lucy     | 123      | 1539751863457 |
|              | 2  | 2019-12-12 | 4000  |     | 1   | lucy     | 123      | 1539751863457 |
|              | 3  | 2020-12-12 | 5000  |     | 2   | tom      | 123      | 1539751863457 |

## 创建Order和User实体

```
public class Order {

 private int id;
 private Date ordertime;
 private double total;

 //代表当前订单从属于哪一个客户
 private User user;
}

public class User {

 private int id;
 private String username;
 private String password;
 private Date birthday;
}
```

## 创建OrderMapper接口

```
public interface OrderMapper {
 List<Order> findAll();
}
```

## 使用注解配置Mapper

```
public interface OrderMapper {
 @Select("select * from orders")
 @Results({
 @Result(id=true,property = "id",column = "id"),
 @Result(property = "ordertime",column = "ordertime"),
 @Result(property = "total",column = "total"),
 @Result(property = "user",column = "uid",
 javaType = User.class,
 one = @One(select =
"com.lagou.mapper.UserMapper.findById"))
 })
 List<Order> findAll();
}
```

```
public interface UserMapper {

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

}
```

## 测试结果

```
@Test
public void testSelectOrderAndUser() {
 List<Order> all = orderMapper.findAll();
 for(Order order : all){
 System.out.println(order);
 }
}
```

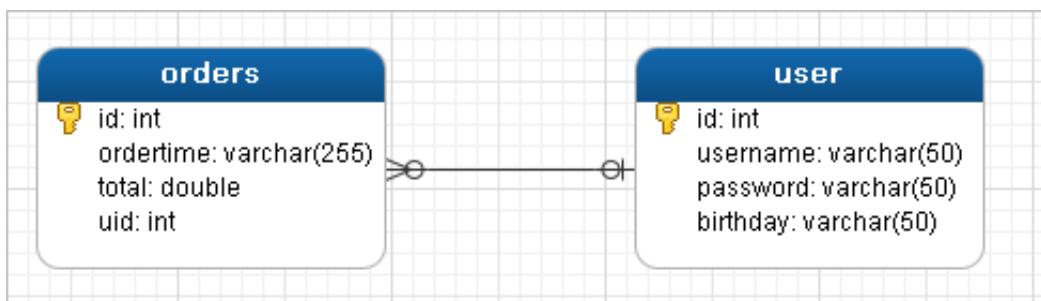
```
12:18:29,699 DEBUG findById:54 - =====> Preparing: select * from user where id=?
12:18:29,699 DEBUG findById:54 - =====> Parameters: 2(Integer)
12:18:29,701 DEBUG findById:54 - <===== Total: 1
12:18:29,701 DEBUG findAll:54 - <== Total: 3
Order{id=1, ordertime=Wed Dec 12 00:00:00 GMT+08:00 2018, total=3000.0, user=User{id=1, username='lucy'},
Order{id=2, ordertime=Thu Dec 12 00:00:00 GMT+08:00 2019, total=4000.0, user=User{id=1, username='lucy'},
Order{id=3, ordertime=Sat Dec 12 00:00:00 GMT+08:00 2020, total=5000.0, user=User{id=2, username='tom'},
```

## 6.5 一对多查询（注解）

## 一对多查询的模型

用户表和订单表的关系为，一个用户有多个订单，一个订单只从属于一个用户

一对多查询的需求：查询所有用户，与此同时查询出每个用户具有的订单



## 一对多查询的语句

对应的sql语句：

```
select * from user;
select * from orders where uid=查询出用户的id;
```

查询的结果如下：

| 信息 | 结果1 | 概况 | 状态 |  |  |  |  |  |  |
|----|-----|----|----|--|--|--|--|--|--|
|    |     |    |    |  |  |  |  |  |  |
|    |     |    |    |  |  |  |  |  |  |
|    |     |    |    |  |  |  |  |  |  |

## 修改User实体

```
public class Order {

 private int id;
 private Date ordertime;
 private double total;

 //代表当前订单从属于哪一个客户
 private User user;
}

public class User {

 private int id;
 private String username;
 private String password;
 private Date birthday;
 //代表当前用户具备哪些订单
 private List<Order> orderList;
```

```
}
```

## 创建UserMapper接口

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

## 使用注解配置Mapper

```
public interface UserMapper {
 @Select("select * from user")
 @Results({
 @Result(id = true,property = "id",column = "id"),
 @Result(property = "username",column = "username"),
 @Result(property = "password",column = "password"),
 @Result(property = "birthday",column = "birthday"),
 @Result(property = "orderList",column = "id",
 javaType = List.class,
 many = @Many(select =
"com.lagou.mapper.OrderMapper.findByUid")))
 })
 List<User> findAllUserAndOrder();
}

public interface OrderMapper {
 @Select("select * from orders where uid=#{uid}")
 List<Order> findByUid(int uid);

}
```

## 测试结果

```
List<User> all = userMapper.findAllUserAndOrder();
for(User user : all){
 System.out.println(user.getUsername());
 List<Order> orderList = user.getOrderList();
 for(Order order : orderList){
 System.out.println(order);
 }
 System.out.println("-----");
}
```

```

14:32:14,813 DEBUG findAllUserAndOrder:54 - ==> Preparing: select * from user
14:32:14,844 DEBUG findAllUserAndOrder:54 - ==> Parameters:
14:32:14,860 DEBUG findByUid:54 - ====> Preparing: select * from orders where uid=? lucy
Order{id=1, ordertime=Wed Dec 12 00:00:00 GMT+08:00 2018, total=3000.0, user=null}
Order{id=2, ordertime=Thu Dec 12 00:00:00 GMT+08:00 2019, total=4000.0, user=null}

tom
Order{id=3, ordertime=Sat Dec 12 00:00:00 GMT+08:00 2020, total=5000.0, user=null}

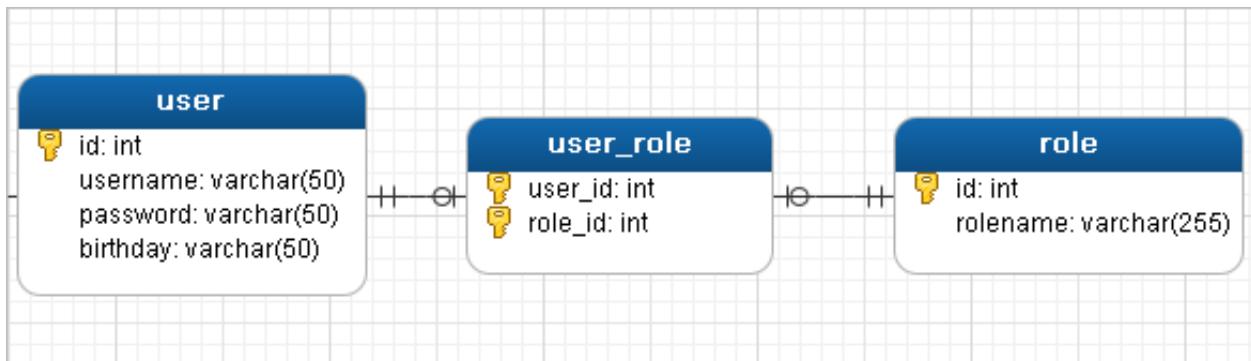
```

## 6.6 多对多查询（注解）

多对多查询的模型

用户表和角色表的关系为，一个用户有多个角色，一个角色被多个用户使用

多对多查询的需求：查询用户同时查询出该用户的所有角色



多对多查询的语句

对应的sql语句：

```

select * from user;

select * from role r,user_role ur where r.id=ur.role_id and ur.user_id=用户的id

```

查询的结果如下：

| 信息 | 结果1      | 概况       | 状态         |     |          |
|----|----------|----------|------------|-----|----------|
| id | username | password | birthday   | id1 | rolename |
| 1  | lucy     | 123      | 2018-12-12 | 1   | CEO      |
|    | lucy     | 123      | 2018-12-12 | 2   | CFO      |
| 2  | tom      | 123      | 2018-12-12 | 2   | CFO      |
|    | tom      | 123      | 2018-12-12 | 3   | COO      |

创建Role实体，修改User实体

```

public class User {
 private int id;
 private String username;
 private String password;
 private Date birthday;

```

```
//代表当前用户具备哪些订单
private List<Order> orderList;
//代表当前用户具备哪些角色
private List<Role> roleList;
}

public class Role {

 private int id;
 private String rolename;

}
```

## 添加UserMapper接口方法

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

## 使用注解配置Mapper

```
public interface UserMapper {
 @Select("select * from user")
 @Results({
 @Result(id = true,property = "id",column = "id"),
 @Result(property = "username",column = "username"),
 @Result(property = "password",column = "password"),
 @Result(property = "birthday",column = "birthday"),
 @Result(property = "roleList",column = "id",
 javaType = List.class,
 many = @Many(select =
"com.lagou.mapper.RoleMapper.findByUid")))
 })
 List<User> findAllUserAndRole();
}

public interface RoleMapper {
 @Select("select * from role r,user_role ur where r.id=ur.role_id and
ur.user_id=#{uid}")
 List<Role> findByUid(int uid);
}
```

## 测试结果

```

UserMapper mapper = sqlSession.getMapper(UserMapper.class);
List<User> all = mapper.findAllUserAndRole();
for(User user : all){
 System.out.println(user.getUsername());
 List<Role> roleList = user.getRoleList();
 for(Role role : roleList){
 System.out.println(role);
 }
 System.out.println("-----");
}

```

```

14:52:12,823 DEBUG findAllUserAndRole:54 - ==> Preparing: select * from user
14:52:12,854 DEBUG findAllUserAndRole:54 - ==> Parameters:
14:52:12,870 DEBUG findByUid:54 - ==> Preparing: select * from role r,user_role ur where r.id=ur.role_id
14:52:12,870 DEBUG findByUid:54 - ==> Parameters: 1(Integer)
14:52:12,870 DEBUG findByUid:54 - <===== Total: 2
14:52:12,870 DEBUG findByUid:54 - ==> Preparing: select * from role r,user_role ur where r.id=ur.role_id
14:52:12,870 DEBUG findByUid:54 - ==> Parameters: 2(Integer)
14:52:12,870 DEBUG findByUid:54 - <===== Total: 2
14:52:12,870 DEBUG findByUid:54 - ==> Preparing: select * from role r,user_role ur where r.id=ur.role_id
14:52:12,870 DEBUG findByUid:54 - ==> Parameters: 5(Integer)
14:52:12,885 DEBUG findByUid:54 - <===== Total: 0
lucy
Role{id=1, rolename='CEO'}
Role{id=2, rolename='CFO'}

tom
Role{id=2, rolename='CFO'}
Role{id=3, rolename='COO'}

```

## 第七部分 Mybatis缓存

缓存就是内存中的数据，常常来自对数据库查询结果的保存，使用缓存，我们可以避免频繁的与数据库进行交互，进而提高响应速度

mybatis也提供了对缓存的支持，分为一级缓存和二级缓存，可以通过下图来理解：



①、一级缓存是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）用于存储缓存数据。不同的sqlSession之间的缓存数据区域（HashMap）是互相不影响的。

②、二级缓存是mapper级别的缓存，多个SqlSession去操作同一个Mapper的sql语句，多个SqlSession可以共用二级缓存，二级缓存是跨SqlSession的

## 7.1 一级缓存

①、我们在一个 sqlSession 中，对 User 表根据id进行两次查询，查看他们发出sql语句的情况。

```
@Test
public void test1(){
 //根据 sqlSessionFactory 产生 session
 SqlSession sqlSession = sessionFactory.openSession();
 UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
 //第一次查询，发出sql语句，并将查询的结果放入缓存中
 User u1 = userMapper.selectUserById(1);
 System.out.println(u1);

 //第二次查询，由于是同一个sqlSession，会在缓存中查找查询结果
 //如果有，则直接从缓存中取出来，不和数据库进行交互
 User u2 = userMapper.selectUserById(1);
 System.out.println(u2);

 sqlSession.close();
}
```

查看控制台打印情况：

```
DEBUG 08-10 22:44:03,114 PooledDataSource forcefully closed/removed all connections. (Log4jImpl.java:46)
DEBUG 08-10 22:44:03,212 Openning JDBC Connection (Log4jImpl.java:46)
DEBUG 08-10 22:44:03,464 Created connection 562266264. (Log4jImpl.java:46)
DEBUG 08-10 22:44:03,467 ooo Using Connection [com.mysql.jdbc.JDBC4Connection@21838098]
DEBUG 08-10 22:44:03,467 ==> Preparing: select * from user where id=? (Log4jImpl.java:46)
DEBUG 08-10 22:44:03,514 ==> Parameters: 1(Integer) (Log4jImpl.java:46)
User [id=1, username=tom, sex=男] 就是在第一次查询时，发出了上面的sql语句，然后第二次查询直接打印用户对象
User [id=1, username=tom, sex=男] 没有发出查询sql语句
DEBUG 08-10 22:44:03,538 Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.
DEBUG 08-10 22:44:03,539 Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@21838098
DEBUG 08-10 22:44:03,539 Returned connection 562266264 to pool. (Log4jImpl.java:46)
```

②、同样是对user表进行两次查询，只不过两次查询之间进行了一次update操作。

```
@Test
public void test2(){
 //根据 sqlSessionFactory 产生 session
 SqlSession sqlSession = sessionFactory.openSession();
 UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
 //第一次查询，发出sql语句，并将查询的结果放入缓存中
 User u1 = userMapper.selectUserById(1);
 System.out.println(u1);

 //第二步进行了一次更新操作，sqlSession.commit()
}
```

```

 u1.setSex("女");
 userMapper.updateUserById(u1);
 sqlSession.commit();

 //第二次查询，由于是同一个sqlSession.commit(),会清空缓存信息
 //则此次查询也会发出 sql 语句
 User u2 = userMapper.selectUserById(1);
 System.out.println(u2);

 sqlSession.close();
 }
}

```

控制台打印情况：

```

DEBUG 08-10 22:58:23,520 ooo Using Connection [com.mysql.jdbc.JDBC4Connection@635fb960] (I)
DEBUG 08-10 22:58:23,520 ==> Preparing: select * from user where id=? (Log4jImpl.java:46)
DEBUG 08-10 22:58:23,522 ==> Parameters: 1(Integer) (Log4jImpl.java:46)
User [id=1, username=tom, sex=男] 第一次查询发出sql语句
DEBUG 08-10 22:58:23,617 ooo Using Connection [com.mysql.jdbc.JDBC4Connection@635fb960] (I)
DEBUG 08-10 22:58:23,617 ==> Preparing: update user set username=? where id=? (Log4jImpl.java:46)
DEBUG 08-10 22:58:23,618 ==> Parameters: tom(String), 1(Integer) (Log4jImpl.java:46)
DEBUG 08-10 22:58:23,619 Committing JDBC Connection [com.mysql.jdbc.JDBC4Connection@635fb960]
DEBUG 08-10 22:58:23,619 ooo Using Connection [com.mysql.jdbc.JDBC4Connection@635fb960] (I)
DEBUG 08-10 22:58:23,620 ==> Preparing: select * from user where id=? (Log4jImpl.java:46)
DEBUG 08-10 22:58:23,623 ==> Parameters: 1(Integer) (Log4jImpl.java:46)
User [id=1, username=tom, sex=男] 由于进行了更新操作，缓存清除了，故第二次查询继续发出sql语句
DEBUG 08-10 22:58:23,628 Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@635fb960]

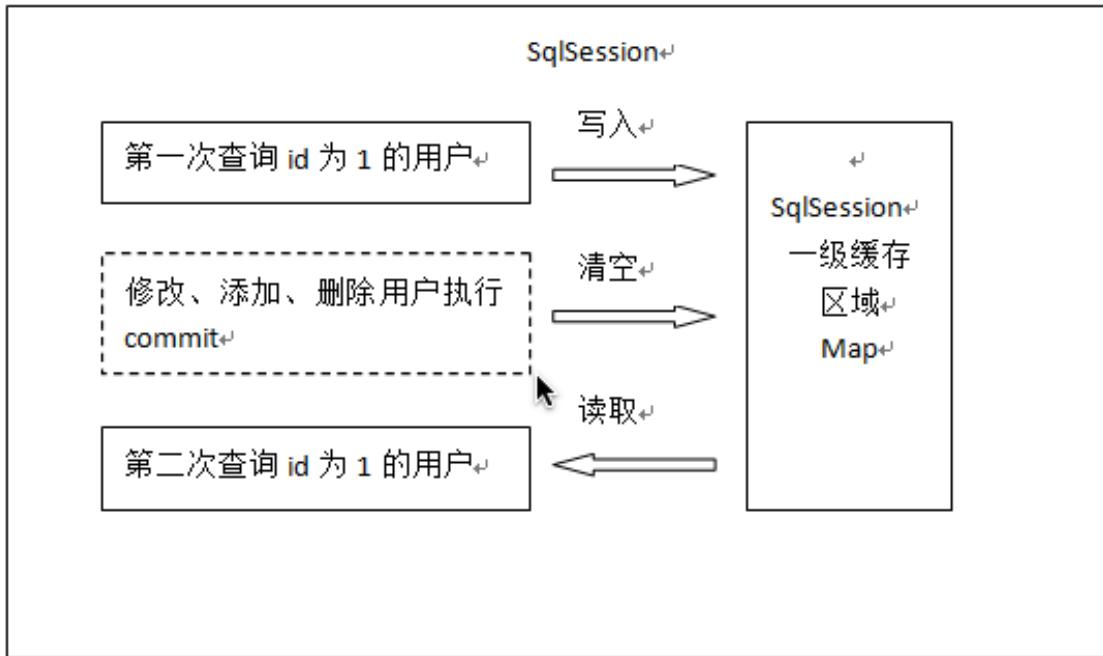
```

### ③、总结

1、第一次发起查询用户id为1的用户信息，先去找缓存中是否有id为1的用户信息，如果没有，从数据库查询用户信息。得到用户信息，将用户信息存储到一级缓存中。

2、如果中间sqlSession去执行commit操作（执行插入、更新、删除），则会清空SqlSession中的一级缓存，这样做的目的为了让缓存中存储的是最新的信息，避免脏读。

3、第二次发起查询用户id为1的用户信息，先去找缓存中是否有id为1的用户信息，缓存中有，直接从缓存中获取用户信息



## 一级缓存原理探究与源码分析

一级缓存到底是什么？一级缓存什么时候被创建、一级缓存的工作流程是怎样的？相信你现在应该会有这几个疑问，那么我们本节就来研究一下一级缓存的本质

大家可以这样想，上面我们一直提到一级缓存，那么提到一级缓存就绕不开 `SqlSession`，所以索性我们就直接从 `SqlSession`，看看有没有创建缓存或者与缓存有关的属性或者方法

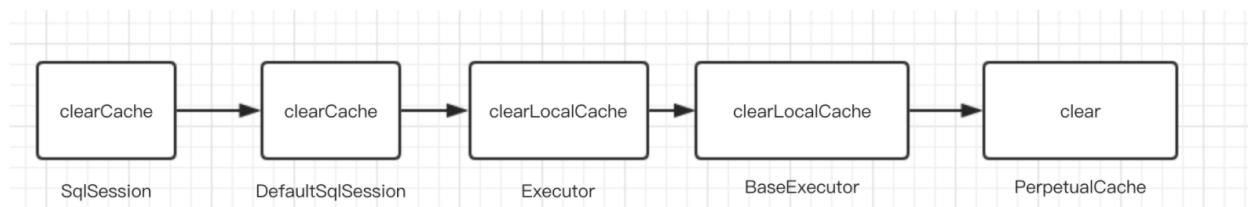
SqISession.java

Inherited members (360)    Anonymous Classes (36)    Lambdas (36)

**SqISession**

- (m) **clearCache(): void**
- (m) **close(): void**
- (m) **commit(): void**
- (m) **commit(boolean): void**
- (m) **delete(String): int**
- (m) **delete(String, Object): int**
- (m) **flushStatements(): List<BatchResult>**
- (m) **getConfiguration(): Configuration**
- (m) **getConnection(): Connection**
- (m) **getMapper(Class<T>): T**
- (m) **insert(String): int**
- (m) **insert(String, Object): int**
- (m) **rollback(): void**
- (m) **rollback(boolean): void**
- (m) **select(String, Object, ResultHandler): void**
- (m) **select(String, Object, RowBounds, ResultHandler): void**
- (m) **select(String, ResultHandler): void**
- (m) **selectCursor(String): Cursor<T>**
- (m) **selectCursor(String, Object): Cursor<T>**
- (m) **selectCursor(String, Object, RowBounds): Cursor<T>**
- (m) **selectList(String): List<E>**
- (m) **selectList(String, Object): List<E>**
- (m) **selectList(String, Object, RowBounds): List<E>**
- (m) **selectMap(String, Object, String): Map<K, V>**
- (m) **selectMap(String, Object, String, RowBounds): Map<K, V>**
- (m) **selectMap(String, String): Map<K, V>**
- (m) **selectOne(String): T**
- (m) **selectOne(String, Object): T**
- (m) **update(String): int**
- (m) **update(String, Object): int**

调研了一圈，发现上述所有方法中，好像只有 `clearCache()` 和缓存沾点关系，那么就直接从这个方法入手吧，分析源码时，我们要看它(此类)是谁，它的父类和子类分别又是谁，对如上关系了解了，你才会对这个类有更深的认识，分析了一圈，你可能会得到如下这个流程图



再深入分析，流程走到 `Perpetualcache` 中的 `clear()` 方法之后，会调用其 `cache.clear()` 方法，那么这个cache 是什么东西呢？点进去发现，`cache` 其实就是 `private Map cache = new HashMap();` 也是一个Map，所以说 `cache.clear()` 其实就是 `map.clear()`，也就是说，缓存其实就是本地存放的一个 map 对象，每一个 `SqISession` 都会存放一个 map 对象的引用，那么这个 `cache` 是何时创建的呢？

你觉得最有可能创建缓存的地方是哪里呢？我觉得是 `Executor`，为什么这么认为？因为 `Executor` 是执行器，用来执行SQL请求，而且清除缓存的方法也在 `Executor` 中执行，所以很可能缓存的创建也很有可能在 `Executor` 中，看了一圈发现 `Executor` 中有一个 `createCacheKey` 方法，这个方法很像是创建缓存的方法啊，跟进去看看，你发现 `createCacheKey` 方法是由 `BaseExecutor` 执行的，代码如下

```

CacheKey cacheKey = new CacheKey();
//MappedStatement的id
// id 就是Sql语句的所在位置 包名 + 类名 + SQL名称
cacheKey.update(ms.getId());
// offset 就是 0
cacheKey.update(rowBounds.getOffset());
// limit 就是 Integer.MAXVALUE
cacheKey.update(rowBounds.getLimit());
// 具体的SQL语句
cacheKey.update(boundSql.getSql());
//后面是update了sql中带的参数
cacheKey.update(value);
...
if (configuration.getEnvironment() != null) {
 // issue #176
 cacheKey.update(configuration.getEnvironment().getId());
}

```

创建缓存key会经过一系列的 update 方法， update 方法由一个 `CacheKey` 这个对象来执行的，这个 update 方法最终由 `updateList` 的 list 来把五个值存进去，对照上面的代码和下面的图示，你应该能理解这五个值都是什么了

```

cacheKey = {CacheKey@1613} "1175668460:2347460:com.mybatis.dao.DeptD
 f multiplier = 37
 f hashCode = 1175668460
 f checksum = 2347460
 f count = 6
 ▼ f updateList = {ArrayList@1687} size = 6
 ▶ 0 = "com.mybatis.dao.DeptDao.findByDeptNo"
 ▶ 1 = {Integer@1690} 0
 ▶ 2 = {Integer@1691} 2147483647
 ▶ 3 = "select * from dept\n \n where deptno = ?"
 ▶ 4 = {Integer@1605} 1
 ▶ 5 = "development"

```

这里需要注意一下最后一个值， `configuration.getEnvironment().getId()` 这是什么，这其实就是在 `mybatis-config.xml` 中的 `<environment>` 标签，见如下。

```

<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>

```

那么我们回归正题，那么创建完缓存之后该用在何处呢？总不会凭空创建一个缓存不使用吧？绝对不会的，经过我们对一级缓存的探究之后，我们发现一级缓存更多是用于查询操作，毕竟一级缓存也叫做查询缓存吧，为什么叫查询缓存我们一会儿说。我们先来看一下这个缓存到底用在哪了，我们跟踪到 query 方法如下：

```

@Override
public <E> List<E> query(MappedStatement ms, Object parameter, RowBounds
rowBounds, ResultHandler resultHandler) throws SQLException {
 BoundSql boundSql = ms.getBoundSql(parameter);
 // 创建缓存
 CacheKey key = createCacheKey(ms, parameter, rowBounds, boundSql);
 return query(ms, parameter, rowBounds, resultHandler, key, boundSql);
}

@SuppressWarnings("unchecked")
@Override
public <E> List<E> query(MappedStatement ms, Object parameter, RowBounds
rowBounds, ResultHandler resultHandler, CacheKey key, BoundSql boundSql)
throws SQLException {
 ...
 list = resultHandler == null ? (List<E>) localCache.getObject(key) : null;
 if (list != null) {
 // 这个主要是处理存储过程用的。
 handleLocallyCachedOutputParameters(ms, key, parameter, boundSql);
 } else {
 list = queryFromDatabase(ms, parameter, rowBounds, resultHandler, key,
 boundSql);
 }
 ...
}

// queryFromDatabase 方法
private <E> List<E> queryFromDatabase(MappedStatement ms, Object parameter,
RowBounds rowBounds, ResultHandler resultHandler, CacheKey key, BoundSql
boundSql) throws SQLException {
 List<E> list;
}

```

```
localCache.putObject(key, EXECUTION_PLACEHOLDER);

try {
 list = doQuery(ms, parameter, rowBounds, resultHandler, boundSql);
} finally {
 localCache.removeObject(key);
}

localCache.putObject(key, list);

if (ms.getStatementType() == StatementType.CALLABLE) {
 localOutputParameterCache.putObject(key, parameter);
}

return list;
}
```

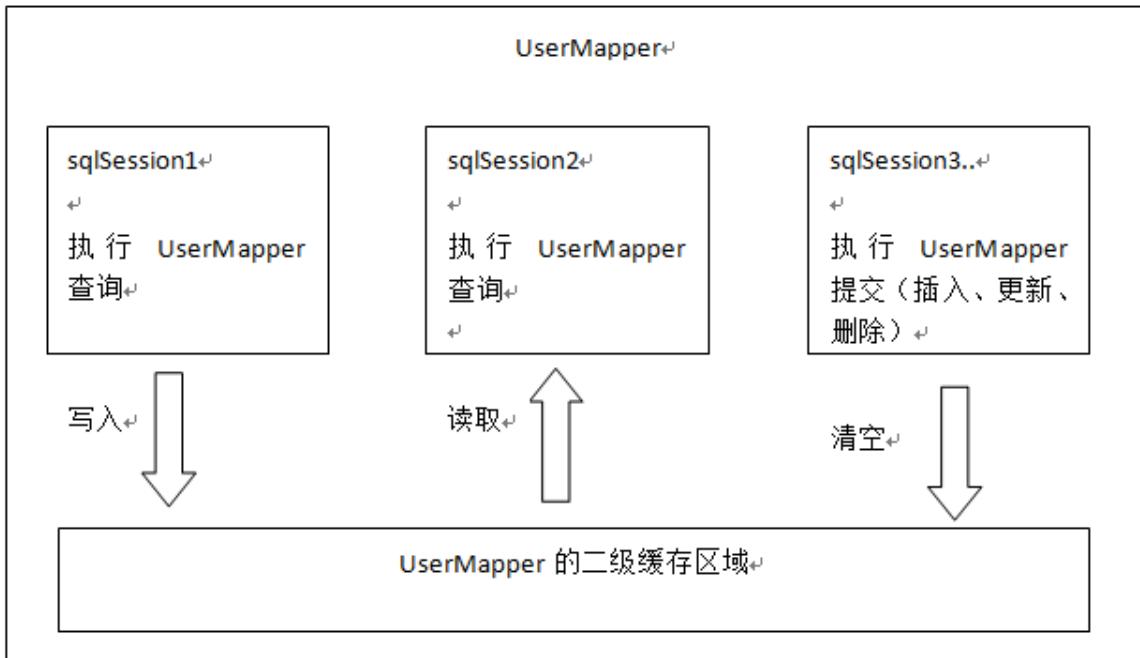
如果查不到的话，就从数据库查，在 `queryFromDatabase` 中，会对 `localcache` 进行写入。`localcache` 对象的 `put` 方法最终交给 `Map` 进行存放

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

@Override
public void putObject(Object key, Object value) {
 cache.put(key, value);
}
```

## 7.2 二级缓存

二级缓存的原理和一级缓存原理一样，第一次查询，会将数据放入缓存中，然后第二次查询则会直接去缓存中取。但是一级缓存是基于 `sqlSession` 的，而二级缓存是基于 `mapper` 文件的 `namespace` 的，也就是说多个 `sqlSession` 可以共享一个 `mapper` 中的二级缓存区域，并且如果两个 `mapper` 的 `namespace` 相同，即使是两个 `mapper`，那么这两个 `mapper` 中执行 SQL 查询到的数据也将存在相同的二级缓存区域中。



如何使用二级缓存：

### ①、开启二级缓存

和一级缓存默认开启不一样，二级缓存需要我们手动开启

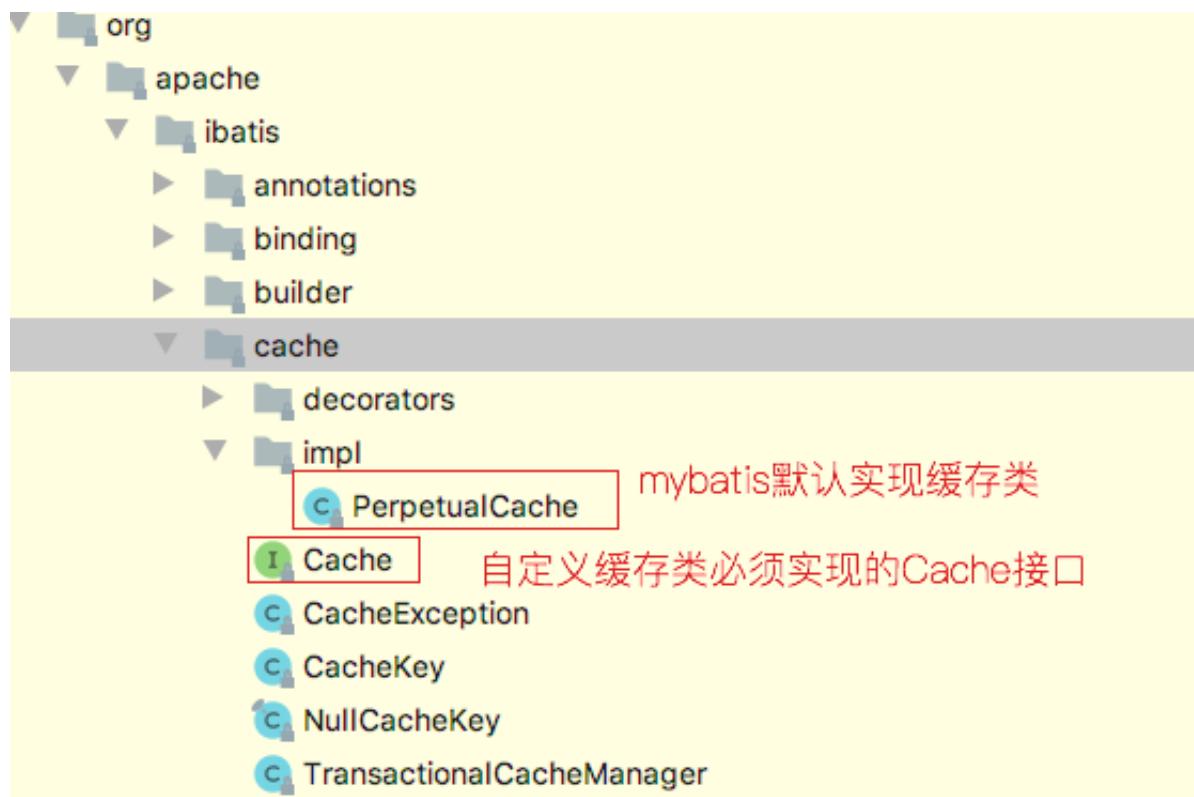
首先在全局配置文件 sqlMapConfig.xml 文件中加入如下代码：

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

其次在 UserMapper.xml 文件中开启缓存

```
<!-- 开启二级缓存 -->
<cache></cache>
```

我们可以看到 mapper.xml 文件中就这么一个空标签，其实这里可以配置 PerpetualCache 这个类是 mybatis 默认实现缓存功能的类。我们不写 type 就使用 mybatis 默认的缓存，也可以去实现 Cache 接口来自定义缓存。



```
public class PerpetualCache implements Cache {
 private final String id;
 private Map<Object, Object> cache = new HashMap();

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

我们可以看到二级缓存底层还是HashMap结构

## ②、po类实现Serializable序列化接口

```
public class User implements Serializable{
 //用户ID
 private int id;
 //用户名
 private String username;
 //用户性别
 private String sex;
```

开启了二级缓存后，还需要将要缓存的pojo实现Serializable接口，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多种多样，不一定只存在内存中，有可能存在硬盘中，如果我们要再取这个缓存的话，就需要反序列化了。所以mybatis中的pojo都去实现Serializable接口

## ③、测试

## 一、测试二级缓存和sqlSession 无关

```
@Test
public void testTwoCache(){
 //根据 sqlSessionFactory 产生 session
 SqlSession sqlSession1 = sessionFactory.openSession();
 SqlSession sqlSession2 = sessionFactory.openSession();

 UserMapper userMapper1 = sqlSession1.getMapper(UserMapper.class);
 UserMapper userMapper2 = sqlSession2.getMapper(UserMapper.class);
 //第一次查询，发出sql语句，并将查询的结果放入缓存中
 User u1 = userMapper1.selectUserById(1);
 System.out.println(u1);
 sqlSession1.close(); //第一次查询完后关闭sqlSession

 //第二次查询，即使sqlSession1已经关闭了，这次查询依然不发出sql语句
 User u2 = userMapper2.selectUserById(1);
 System.out.println(u2);
 sqlSession2.close();
```

可以看出上面两个不同的sqlSession,第一个关闭了， 第二次查询依然不发出sql查询语句

## 二、测试执行 commit() 操作，二级缓存数据清空

```
@Test
public void testTwoCache(){
 //根据 sqlSessionFactory 产生 session
 SqlSession sqlSession1 = sessionFactory.openSession();
 SqlSession sqlSession2 = sessionFactory.openSession();
 SqlSession sqlSession3 = sessionFactory.openSession();

 String statement = "com.lagou.pojo.UserMapper.selectUserById" ;
 UserMapper userMapper1 = sqlSession1.getMapper(UserMapper.class);
 UserMapper userMapper2 = sqlSession2.getMapper(UserMapper.class);
 UserMapper userMapper3 = sqlSession2.getMapper(UserMapper.class);
 //第一次查询，发出sql语句，并将查询的结果放入缓存中
 User u1 = userMapper1.selectUserById(1);
 System.out.println(u1);
 sqlSession1.close(); //第一次查询完后关闭sqlSession

 //执行更新操作，commit()
 u1.setUsername("aaa");
 userMapper3.updateUserById(u1);
 sqlSession3.commit();

 //第二次查询，由于上次更新操作，缓存数据已经清空（防止数据脏读），这里必须再次发出sql语句
 User u2 = userMapper2.selectUserById(1);
 System.out.println(u2);
```

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

查看控制台情况：

```
DEBUG 08-10 23:44:14,151 Cache hit ratio [com.ye.twocache.usermodel]: 0.0 (Log4jImpl.java:46)
DEBUG 08-10 23:44:14,136 Opening JDBC Connection (Log4jImpl.java:46)
DEBUG 08-10 23:44:14,339 Created connection 892915569. (Log4jImpl.java:46)
DEBUG 08-10 23:44:14,343 ooo Using Connection [com.mysql.jdbc.JDBC4Connection@3538cf71] (Log4jI
DEBUG 08-10 23:44:14,344 ==> Preparing: select * from user where id=? (Log4jImpl.java:46)
DEBUG 08-10 23:44:14,391 ==> Parameters: 1(Integer) (Log4jImpl.java:46)
Jser [id=1, username=tom, sex=?] 第一次查询，缓存中没有数据，故向数据库发出 sql 语句，并将数据放入二级缓存
DEBUG 08-10 23:44:14,422 Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Co
DEBUG 08-10 23:44:14,422 Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@3538cf71] (Log
DEBUG 08-10 23:44:14,422 Returned connection 892915569 to pool. (Log4jImpl.java:46)
DEBUG 08-10 23:44:14,423 Openning JDBC Connection (Log4jImpl.java:46) 执行更新操作 commit(), 会清空二级缓存
DEBUG 08-10 23:44:14,423 Checked out connection 892915569 from pool. (Log4jImpl.java:46)
DEBUG 08-10 23:44:14,424 Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Con
DEBUG 08-10 23:44:14,425 ooo Using Connection [com.mysql.jdbc.JDBC4Connection@3538cf71] (Log4jI
DEBUG 08-10 23:44:14,425 ==> Preparing: update user set username=? where id=? (Log4jImpl.java
DEBUG 08-10 23:44:14,425 ==> Parameters: aaa(String), 1(Integer) (Log4jImpl.java:46)
DEBUG 08-10 23:44:14,426 ooo Using Connection [com.mysql.jdbc.JDBC4Connection@3538cf71] (Log4jI
DEBUG 08-10 23:44:14,426 ==> Preparing: select * from user where id=? (Log4jImpl.java:46)
DEBUG 08-10 23:44:14,426 ==> Parameters: 1(Integer) (Log4jImpl.java:46)
Jser [id=1, username=aaa, sex=?] 第二次查询，由于上面的更新操作，缓存已经清空，故这里会发出sql语句
DEBUG 08-10 23:44:14,427 Rolling back JDBC Connection [com.mysql.jdbc.JDBC4Connection@3538cf71]
DEBUG 08-10 23:44:14,612 Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Co
DEBUG 08-10 23:44:14,613 Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@3538cf71] (Log
DEBUG 08-10 23:44:14,614 Returned connection 892915569 to pool. (Log4jImpl.java:46)
```

#### ④、useCache和flushCache

mybatis中还可以配置userCache和flushCache等配置项，userCache是用来设置是否禁用二级缓存的，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="selectUserById" useCache="false"
resultType="com.lagou.pojo.User" parameterType="int">
 select * from user where id=#{id}
</select>
```

这种情况是针对每次查询都需要最新的数据sql，要设置成useCache=false，禁用二级缓存，直接从数据库中获取。

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。

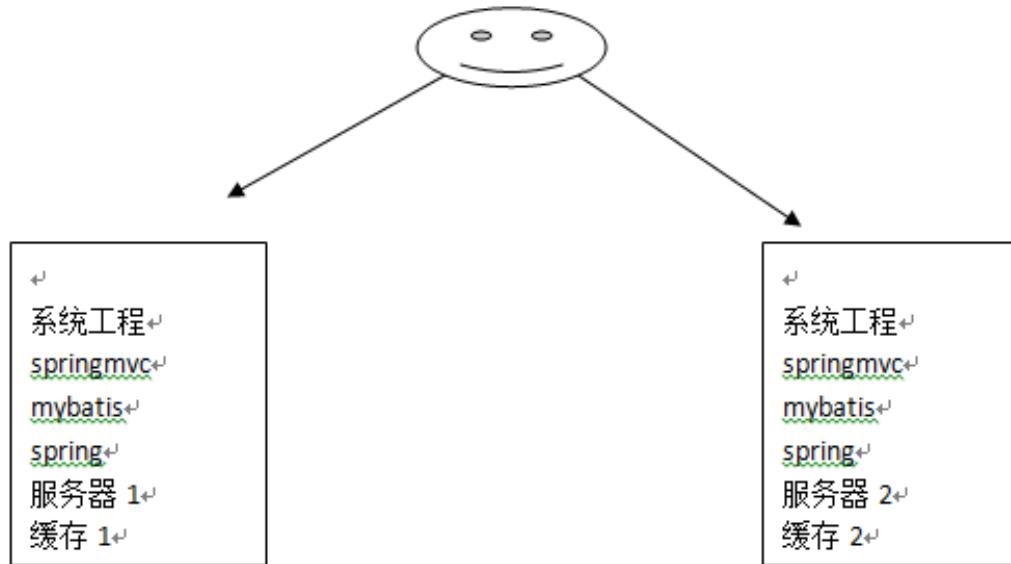
设置statement配置中的flushCache="true" 属性，默认情况下为true，即刷新缓存，如果改成false则不会刷新。使用缓存时如果手动修改数据库表中的查询数据会出现脏读。

```
<select id="selectUserById" flushCache="true" useCache="false"
resultType="com.lagou.pojo.User" parameterType="int">
 select * from user where id=#{id}
</select>
```

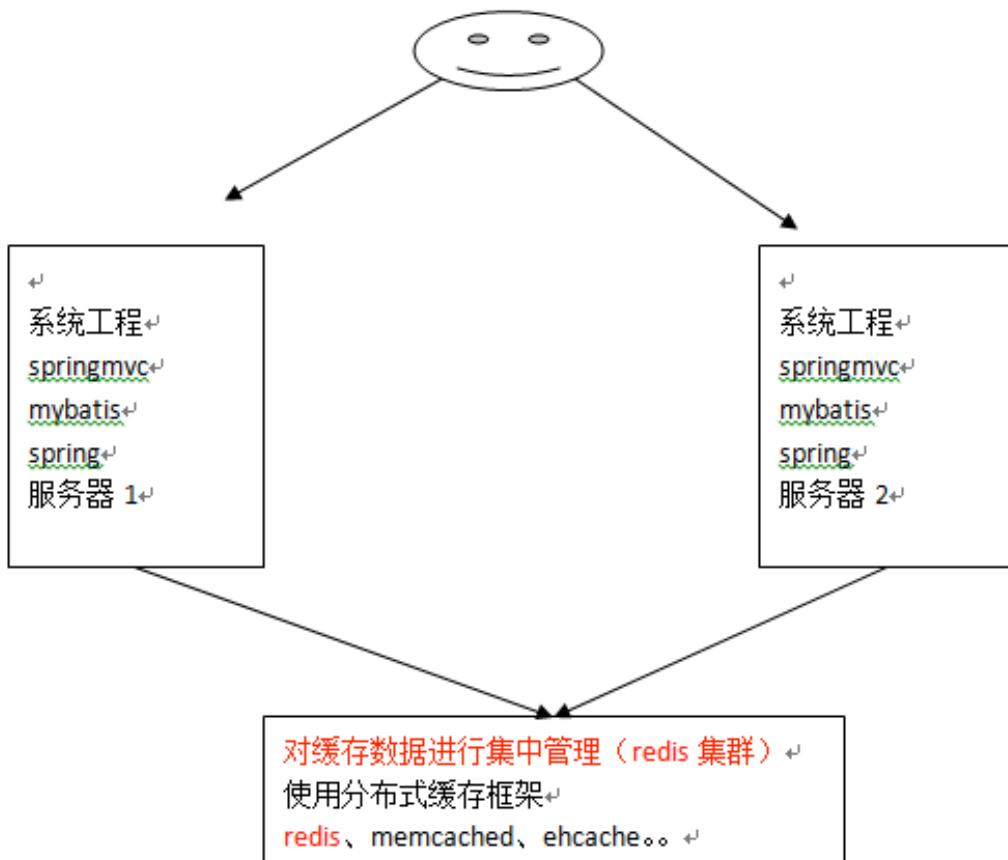
一般下执行完commit操作都需要刷新缓存，flushCache=true表示刷新缓存，这样可以避免数据库脏读。所以我们不用设置，默认即可

## 7.3 二级缓存整合redis

上面我们介绍了mybatis自带的二级缓存，但是这个缓存是单服务器工作，无法实现分布式缓存。那么什么是分布式缓存呢？假设现在有两个服务器1和2，用户访问的时候访问了1服务器，查询后的缓存就会放在1服务器上，假设现在有个用户访问的是2服务器，那么他在2服务器上就无法获取刚刚那个缓存，如下图所示：



为了解决这个问题，就得找一个分布式的缓存，专门用来存储缓存数据的，这样不同的服务器要缓存数据都往它那里存，取缓存数据也从它那里取，如下图所示：



如上图所示，在几个不同的服务器之间，我们使用第三方缓存框架，将缓存都放在这个第三方框架中，然后无论有多少台服务器，我们都能从缓存中获取数据。

## 这里我们介绍mybatis与redis的整合。

刚刚提到过， mybatis提供了一个cache接口，如果要实现自己的缓存逻辑，实现cache接口开发即可。mybatis本身默认实现了一个，但是这个缓存的实现无法实现分布式缓存，所以我们要自己来实现。redis分布式缓存就可以， mybatis提供了一个针对cache接口的redis实现类,该类存在mybatis-redis包中

实现：

### 1.pom文件

```
<dependency>
 <groupId>org.mybatis.caches</groupId>
 <artifactId>mybatis-redis</artifactId>
 <version>1.0.0-beta2</version>
</dependency>
```

### 2.配置文件

#### Mapper.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="com.lagou.mapper.IUserMapper">

<cache type="org.mybatis.caches.redis.RedisCache" />

<select id="findAll" resultType="com.lagou.pojo.User" useCache="true">
 select * from user
</select>
```

### 3.redis.properties

```
redis.host=localhost
redis.port=6379
redis.connectionTimeout=5000
redis.password=
redis.database=0
```

### 4.测试

```
@Test
public void SecondLevelCache(){
 SqlSession sqlSession1 = sqlSessionFactory.openSession();
 SqlSession sqlSession2 = sqlSessionFactory.openSession();
 SqlSession sqlSession3 = sqlSessionFactory.openSession();

 IUserMapper mapper1 = sqlSession1.getMapper(IUserMapper.class);
```

```

IUserMapper mapper2 = sqlSession2.getMapper(IUserMapper.class);
IUserMapper mapper3 = sqlSession3.getMapper(IUserMapper.class);

User user1 = mapper1.findUserById(1);
sqlSession1.close(); //清空一级缓存

User user = new User();
user.setId(1);
user.setUsername("lisi");
mapper3.updateUser(user);
sqlSession3.commit();

User user2 = mapper2.findUserById(1);

System.out.println(user1==user2);

}

```

### 源码分析：

RedisCache和大家普遍实现Mybatis的缓存方案大同小异，无非是实现Cache接口，并使用jedis操作缓存；不过该项目在设计细节上有一些区别；

```

public final class RedisCache implements Cache {
 public RedisCache(final String id) {
 if (id == null) {
 throw new IllegalArgumentException("Cache instances require an
ID");
 }
 this.id = id;
 RedisConfig redisConfig =
RedisConfigurationBuilder.getInstance().parseConfiguration();
 pool = new JedisPool(redisConfig, redisConfig.getHost(),
redisConfig.getPort(),
redisConfig.getConnectionTimeout(),
redisConfig.getSoTimeout(), redisConfig.getPassword(),
redisConfig.getDatabase(), redisConfig.getClientName());
 }
}

```

RedisCache在mybatis启动的时候，由MyBatis的CacheBuilder创建，创建的方式很简单，就是调用RedisCache的带有String参数的构造方法，即RedisCache(String id)；而在RedisCache的构造方法中，调用了RedisConfigurationBuilder来创建RedisConfig对象，并使用RedisConfig来创建JedisPool。

RedisConfig类继承了JedisPoolConfig，并提供了host,port等属性的包装，简单看一下RedisConfig的属性： public class RedisConfig extends JedisPoolConfig {

```
private String host = Protocol.DEFAULT_HOST;
private int port = Protocol.DEFAULT_PORT;
private int connectionTimeout = Protocol.DEFAULT_TIMEOUT;
private int soTimeout = Protocol.DEFAULT_TIMEOUT;
private String password;
private int database = Protocol.DEFAULT_DATABASE;
private String clientName;
```

RedisConfig对象是由RedisConfigurationBuilder创建的，简单看下这个类的主要方法：

```
public RedisConfig parseConfiguration(ClassLoader classLoader) {
 Properties config = new Properties();
 InputStream input =
 classLoader.getResourceAsStream(redisPropertiesFilename);
 if (input != null) {
 try {
 config.load(input);
 } catch (IOException e) {
 throw new RuntimeException(
 "An error occurred while reading classpath property '" +
 + redisPropertiesFilename
 + "', see nested exceptions", e);
 } finally {
 try {
 input.close();
 } catch (IOException e) {
 // close quietly
 }
 }
 }

 RedisConfig jedisConfig = new RedisConfig();
 setConfigProperties(config, jedisConfig);
 return jedisConfig;
}
```

核心的方法就是parseConfiguration方法，该方法从classpath中读取一个redis.properties文件：

```
host=localhost
port=6379
connectionTimeout=5000
soTimeout=5000
password=
database=0
clientName=
```

并将该配置文件中的内容设置到RedisConfig对象中，并返回；接下来，就是RedisCache使用RedisConfig类创建完成JedisPool；在RedisCache中实现了一个简单的模板方法，用来操作Redis：

```
private Object execute(RedisCallback callback) {
 Jedis jedis = pool.getResource();
 try {
 return callback.doWithRedis(jedis);
 } finally {
 jedis.close();
 }
}
```

模板接口为RedisCallback，这个接口中就只需要实现了一个doWithRedis方法而已：

```
public interface RedisCallback {
 Object doWithRedis(Jedis jedis);
}
```

接下来看看Cache中最重要的两个方法：putObject和getObject，通过这两个方法来查看mybatis-redis储存数据的格式：

```
@Override
public void putObject(final Object key, final Object value) {
 execute(new RedisCallback() {
 @Override
 public Object doWithRedis(Jedis jedis) {
 jedis.hset(id.toString().getBytes(), key.toString().getBytes(),
 SerializeUtil.serialize(value));
 return null;
 }
 });
}

@Override
public Object getObject(final Object key) {
 return execute(new RedisCallback() {
 @Override
 public Object doWithRedis(Jedis jedis) {
 return SerializeUtil.deserialize(jedis.hget(id.toString().getBytes(),
 key.toString().getBytes()));
 }
 });
}
```

可以很清楚的看到，mybatis-redis在存储数据的时候，是使用的hash结构，把cache的id作为这个hash的key（cache的id在mybatis中就是mapper的namespace）；这个mapper中的查询缓存数据作为hash的field，需要缓存的内容直接使用SerializeUtil存储，SerializeUtil和其他的序列化类差不多，负责对象的序列化和反序列化；

# 第八部分 Mybatis插件

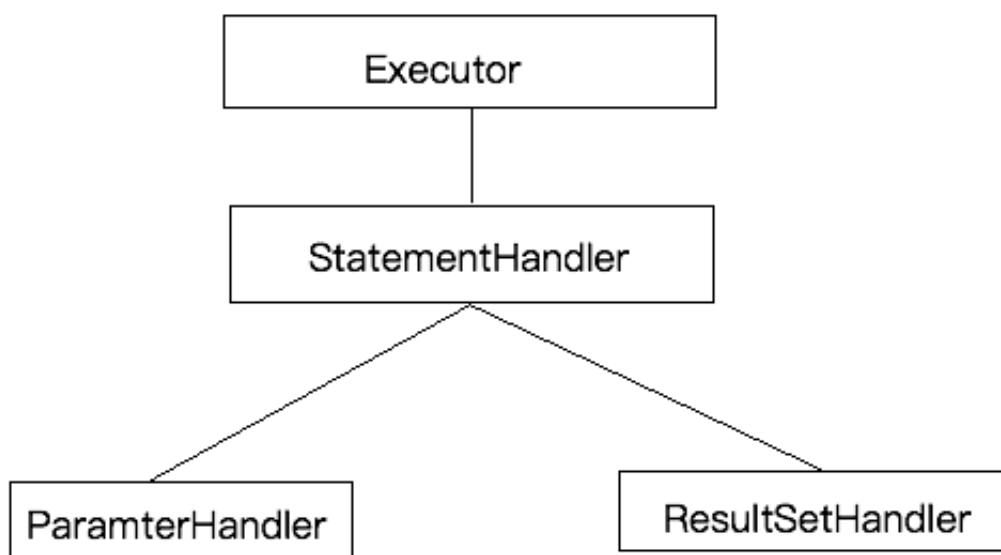
## 1. 插件简介

一般情况下，开源框架都会提供插件或其他形式的拓展点，供开发者自行拓展。这样好处是显而易见的，一是增加了框架的灵活性。二是开发者可以结合实际需求，对框架进行拓展，使其能够更好的工作。以 MyBatis 为例，我们可基于 MyBatis 插件机制实现分页、分表，监控等功能。由于插件和业务无关，业务也无法感知插件的存在。因此可以无感植入插件，在无形中增强功能

## 2. Mybatis插件介绍

Mybatis作为一个应用广泛的优秀的ORM开源框架，这个框架具有强大的灵活性，在四大组件

(Executor、StatementHandler、ParameterHandler、ResultSetHandler) 处提供了简单易用的插件扩展机制。Mybatis对持久层的操作就是借助于四大核心对象。MyBatis支持用插件对四大核心对象进行拦截，对mybatis来说插件就是拦截器，用来增强核心对象的功能，增强功能本质上是借助于底层的动态代理实现的，换句话说，MyBatis中的四大对象都是代理对象



MyBatis 所允许拦截的方法如下：

- 执行器Executor (update、query、commit、rollback等方法)；
- SQL语法构建器StatementHandler (prepare、parameterize、batch、update、query等方法)；
- 参数处理器ParameterHandler (getParameterObject、setParameters方法)；
- 结果集处理器ResultSetHandler (handleResultSets、handleOutputParameters等方法)；

## 3. Mybatis插件原理

在四大对象创建的时候

- 1、每个创建出来的对象不是直接返回的，而是interceptorChain.pluginAll(parameterHandler);
- 2、获取到所有的Interceptor（拦截器）（插件需要实现的接口）；调用 interceptor.plugin(target);返回target包装后的对象

- 3、插件机制，我们可以使用插件为目标对象创建一个代理对象；AOP（面向切面）我们的插件可以为四大对象创建出代理对象，代理对象就可以拦截到四大对象的每一个执行；

## 拦截

插件具体是如何拦截并附加额外的功能的呢？以ParameterHandler来说

```
public ParameterHandler newParameterHandler(MappedStatement mappedStatement,
Object object, BoundSql sql, InterceptorChain interceptorChain){
 ParameterHandler parameterHandler =
 mappedStatement.getLang().createParameterHandler(mappedStatement, object, sql);
 parameterHandler = (ParameterHandler) interceptorChain.pluginAll(parameterHandler);
 return parameterHandler;
}
public Object pluginAll(Object target) {
 for (Interceptor interceptor : interceptors) {
 target = interceptor.plugin(target);
 }
 return target;
}
```

interceptorChain 保存了所有的拦截器(interceptors)，是mybatis初始化的时候创建的。调用拦截器链中的拦截器依次的对目标进行拦截或增强。interceptor.plugin(target)中的target就可以理解为mybatis中的四大对象。返回的target是被重重代理后的对象

如果我们想要拦截 Executor 的 query 方法，那么可以这样定义插件：

```
@Intercepts({
 @Signature(
 type = Executor.class,
 method = "query",
 args ={MappedStatement.class, Object.class, RowBounds.class, ResultHandler.class}
)
})
public class ExamplePlugin implements Interceptor {
 // 省略逻辑
}
```

除此之外，我们还需将插件配置到sqlMapConfig.xml中。

```
<plugins>
 <plugin interceptor="com.lagou.plugin.ExamplePlugin">
 </plugin>
</plugins>
```

这样 MyBatis 在启动时可以加载插件，并保存插件实例到相关对象（InterceptorChain，拦截器链）中。待准备工作做完后，MyBatis 处于就绪状态。我们在执行 SQL 时，需要先通过 DefaultSqlSessionFactory 创建 SqlSession。Executor 实例会在创建 SqlSession 的过程中被创建，Executor 实例创建完毕后，MyBatis 会通过 JDK 动态代理为实例生成代理类。这样，插件逻辑即可在 Executor 相关方法被调用前执行。

以上就是 MyBatis 插件机制的基本原理

## 4. 自定义插件

### 4.1 插件接口

Mybatis插件接口-Interceptor

- Intercept方法，插件的核心方法
- plugin方法，生成target的代理对象
- setProperties方法，传递插件所需参数

### 4.2 自定义插件

设计实现一个自定义插件

```
@Intercepts({//注意看这个大花括号，也就这说这里可以定义多个@Signature对多个地方拦截，都用
这个拦截器
 @Signature(type = StatementHandler.class,//这是指拦截哪个接口
 method = "prepare",//这个接口内的哪个方法名，不要拼错了
 args = { Connection.class, Integer.class}),//这是拦截的方法的入
参，按顺序写到这，不要多也不要少，如果方法重载，可是要通过方法名和入参来确定唯一的
})
public class MyPlugin implements Interceptor {

 private final Logger logger = LoggerFactory.getLogger(this.getClass());

 // // 这里是每次执行操作的时候，都会进行这个拦截器的方法内
 @Override
 public Object intercept(Invocation invocation) throws Throwable {
 //增强逻辑
 System.out.println("对方法进行了增强....");
 return invocation.proceed(); //执行原方法
 }
 /**
 * // 主要是为了把这个拦截器生成一个代理放到拦截器链中
 * @Description 包装目标对象 为目标对象创建代理对象
 * @Param target为要拦截的对象
 * @Return 代理对象
 */
 @Override
 public Object plugin(Object target) {
 System.out.println("将要包装的目标对象: "+target);
 }
}
```

```
 return Plugin.wrap(target,this);
 }

 /** 获取配置文件的属性 */
 // 插件初始化的时候调用，也只调用一次，插件配置的属性从这里设置进来
 @Override
 public void setProperties(Properties properties) {
 System.out.println("插件配置的初始化参数: "+properties);
 }
}
```

## sqlMapConfig.xml

```
<plugins>
 <plugin interceptor="com.lagou.plugin.MySqlPagingPlugin">
 <!--配置参数-->
 <property name="name" value="Bob"/>
 </plugin>
</plugins>
```

## mapper接口

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

## mapper.xml

```
<mapper namespace="com.lagou.mapper.UserMapper">
 <select id="selectUser" resultType="com.lagou.pojo.User">
 SELECT
 id,username
 FROM
 user
 </select>
</mapper>
```

## 测试类

```
public class PluginTest {

 @Test
 public void test() throws IOException {
 InputStream resourceAsStream =
Resources.getResourceAsStream("sqlMapConfig.xml");
 SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(resourceAsStream);
```

```

SqlSession sqlSession = sqlSessionFactory.openSession();
UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

List<User> byPaging = userMapper.selectUser();
for (User user : byPaging) {
 System.out.println(user);
}
}
}

```

## 5. 源码分析

### 执行插件逻辑

Plugin 实现了 InvocationHandler 接口，因此它的 invoke 方法会拦截所有的方法调用。invoke 方法会对所拦截的方法进行检测，以决定是否执行插件逻辑。该方法的逻辑如下：

```

// -☆- Plugin
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
 try {
 /*
 * 获取被拦截方法列表，比如：
 * signatureMap.get(Executor.class)，可能返回 [query, update, commit]
 */
 Set<Method> methods = signatureMap.get(method.getDeclaringClass());
 // 检测方法列表是否包含被拦截的方法
 if (methods != null && methods.contains(method)) {
 // 执行插件逻辑
 return interceptor.intercept(new Invocation(target, method, args));
 }
 // 执行被拦截的方法
 return method.invoke(target, args);
 } catch (Exception e) {
 throw ExceptionUtil.unwrapThrowable(e);
 }
}

```

invoke 方法的代码比较少，逻辑不难理解。首先，invoke 方法会检测被拦截方法是否配置在插件的 @Signature 注解中，若是，则执行插件逻辑，否则执行被拦截方法。插件逻辑封装在 intercept 中，该方法的参数类型为 Invocation。Invocation 主要用于存储目标类，方法以及方法参数列表。下面简单看一下该类的定义

```

public class Invocation {

 private final Object target;
 private final Method method;
 private final Object[] args;

 public Invocation(Object target, Method method, Object[] args) {
 this.target = target;
 this.method = method;
 this.args = args;
 }

 // 省略部分代码

 public Object proceed() throws InvocationTargetException, IllegalAccessException {
 // 调用被拦截的方法
 return method.invoke(target, args);
 }
}

```

关于插件的执行逻辑就分析结束

## 6 pageHelper分页插件

MyBatis可以使用第三方的插件来对功能进行扩展，分页助手PageHelper是将分页的复杂操作进行封装，使用简单的方式即可获得分页的相关数据

开发步骤：

- ①导入通用PageHelper坐标
- ②在mybatis核心配置文件中配置PageHelper插件
- ③测试分页数据获取

### ①导入通用PageHelper坐标

```

<!-- 分页助手 -->
<dependency>
 <groupId>com.github.pagehelper</groupId>
 <artifactId>pagehelper</artifactId>
 <version>3.7.5</version>
</dependency>
<dependency>
 <groupId>com.github.jsqlparser</groupId>
 <artifactId>jsqlparser</artifactId>
 <version>0.9.1</version>
</dependency>

```

## ②在mybatis核心配置文件中配置PageHelper插件

```
<!-- 注意：分页助手的插件 配置在通用Mapper之前 -->
<plugin interceptor="com.github.pagehelper.PageHelper">
 <!-- 指定方言 -->
 <property name="dialect" value="mysql"/>
</plugin>
```

## ③测试分页代码实现

```
@Test
public void testPageHelper(){
 //设置分页参数
 PageHelper.startPage(1,2);

 List<User> select = userMapper2.select(null);
 for(User user : select){
 System.out.println(user);
 }
}
```

## 获得分页相关的其他参数

```
//其他分页的数据
PageInfo<User> pageInfo = new PageInfo<User>(select);
System.out.println("总条数: "+pageInfo.getTotal());
System.out.println("总页数: "+pageInfo.getPages());
System.out.println("当前页: "+pageInfo.getPageNum());
System.out.println("每页显示长度: "+pageInfo.getPageSize());
System.out.println("是否第一页: "+pageInfo.isIsFirstPage());
System.out.println("是否最后一页: "+pageInfo.isIsLastPage());
```

# 7 通用Mapper

## 什么是通用Mapper

通用Mapper就是为了解决单表增删改查，基于Mybatis的插件机制。开发人员不需要编写SQL，不需要在DAO中增加方法，只要写好实体类，就能支持相应的增删改查方法

## 如何使用

1.首先在maven项目，在pom.xml中引入mapper的依赖

```
<dependency>
 <groupId>tk.mybatis</groupId>
 <artifactId>mapper</artifactId>
 <version>3.1.2</version>
</dependency>
```

## 2.Mybatis配置文件中完成配置

```
<plugins>
 <!-- 分页插件 :如果有分页插件，要排在通用mapper之前
 <plugin interceptor="com.github.pagehelper.PageHelper">
 <property name="dialect" value="mysql"/>
 </plugin>
-->
 <plugin
interceptor="tk.mybatis.mapper.mapperhelper.MapperInterceptor">
 <!--通用Mapper接口，多个通用接口用逗号隔开-->
 <property name="mappers"
value="tk.mybatis.mapper.common.Mapper" />
 </plugin>
</plugins>
```

## 3.实体类设置主键

```
@Table(name = "t_user")
public class User {
 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Integer id;
 private String username;
}
```

## 4.定义通用mapper

```
import com.lagou.domain.User;
import tk.mybatis.mapper.common.Mapper;

public interface UserMapper extends Mapper<User> {
}
```

## 5.测试

```
public class UserTest {

 @Test
 public void test1() throws IOException {
```

```

 InputStream resourceAsStream = Resources.getResourceAsStream("mybatis-
mapper-config.xml");
 SqlSessionFactory build = new
SqlSessionFactoryBuilder().build(resourceAsStream);
 SqlSession sqlSession = build.openSession();
 UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
 User user = new User();
 user.setId(4);
 // (1) mapper基础接口
 //select接口
 User user1 = userMapper.selectOne(user); //根据实体中的属性进行查询, 只能有
一个返回值
 List<User> users = userMapper.select(null); //查询全部结果
 userMapper.selectByPrimaryKey(1); //根据主键字段进行查询, 方法参数必须包含完
整的主键属性, 查询条件使用等号
 userMapper.selectCount(user); //根据实体中的属性查询总数, 查询条件使用等号
 // insert 接口
 int insert = userMapper.insert(user); //保存一个实体, null值也会保存, 不会使
用数据库默认值
 int i = userMapper.insertSelective(user); //保存实体, null的属性不会保存,
会使用数据库默认值
 // update 接口
 int i1 = userMapper.updateByPrimaryKey(user); //根据主键更新实体全部字段,
null值会被更新
 // delete接口
 int delete = userMapper.delete(user); //根据实体属性作为条件进行删除, 查询条件
使用等号
 userMapper.deleteByPrimaryKey(1); //根据主键字段进行删除, 方法参数必须包含完
整的主键属性

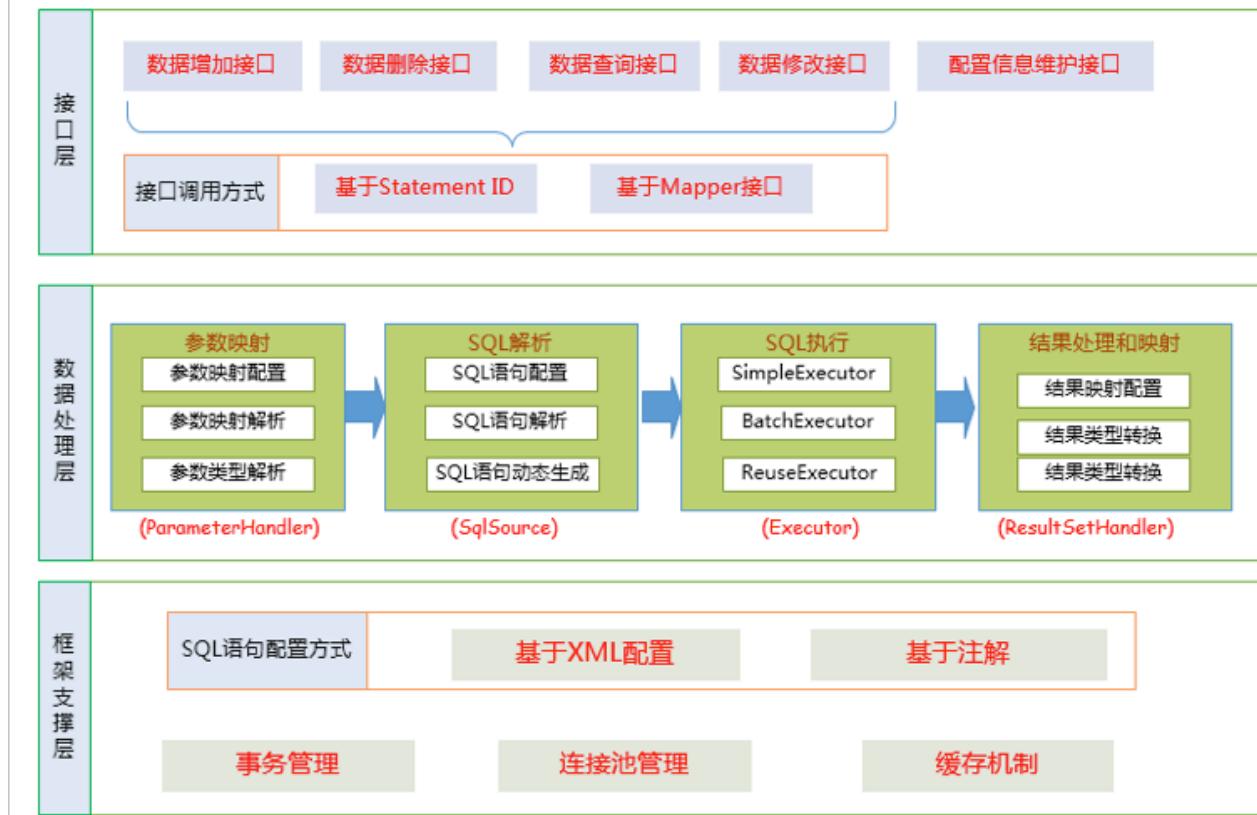
 // (2) example方法
 Example example = new Example(User.class);
 example.createCriteria().andEqualTo("id",1);
 example.createCriteria().andLike("val", "1");
 //自定义查询
 List<User> users1 = userMapper.selectByExample(example);
 }

}

```

## 第九部分 Mybatis架构原理

### 9.1 架构设计



我们把Mybatis的功能架构分为三层：

(1) API接口层：提供给外部使用的接口API，开发人员通过这些本地API来操纵数据库。接口层一接收到调用请求就会调用数据处理层来完成具体的数据处理。

MyBatis和数据库的交互有两种方式：

a. 使用传统的MyBatis提供的API；

b. 使用Mapper代理的方式

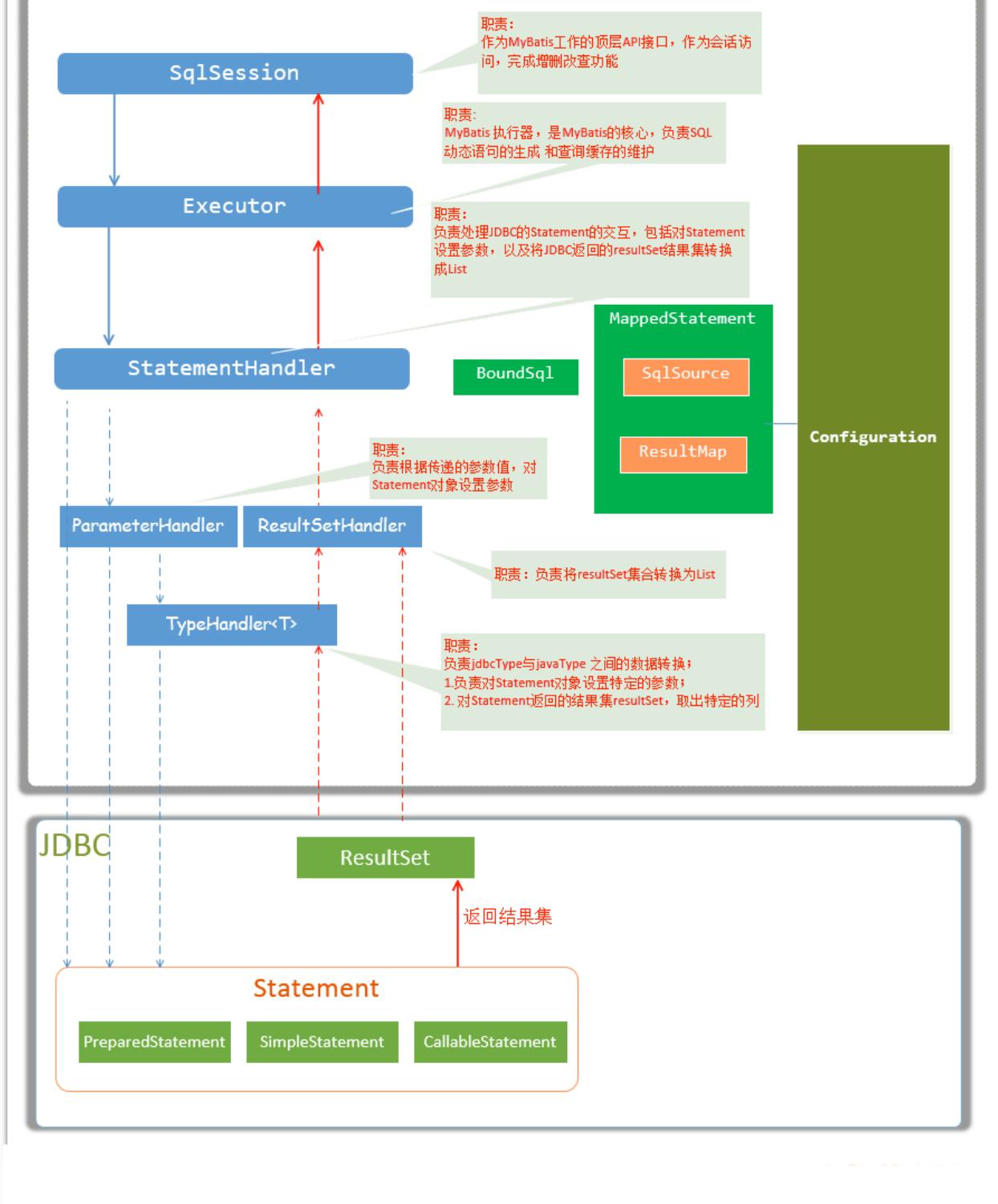
(2) 数据处理层：负责具体的SQL查找、SQL解析、SQL执行和执行结果映射处理等。它主要的目的是根据调用的请求完成一次数据库操作。

(3) 基础支撑层：负责最基础的功能支撑，包括连接管理、事务管理、配置加载和缓存处理，这些都是共用的东西，将他们抽取出来作为最基础的组件。为上层的数据处理层提供最基础的支撑

## 9.2 主要构件及其相互关系

| 构件               | 描述                                                                      |
|------------------|-------------------------------------------------------------------------|
| SqlSession       | 作为MyBatis工作的主要顶层API，表示和数据库交互的会话，完成必要数据库增删改查功能                           |
| Executor         | MyBatis执行器，是MyBatis 调度的核心，负责SQL语句的生成和查询缓存的维护                            |
| StatementHandler | 封装了JDBC Statement操作，负责对JDBC statement 的操作，如设置参数、将Statement结果集转换成List集合。 |
| ParameterHandler | 负责对用户传递的参数转换成JDBC Statement 所需要的参数，                                     |
| ResultSetHandler | 负责将JDBC返回的ResultSet结果集对象转换成List类型的集合；                                   |
| TypeHandler      | 负责java数据类型和jdbc数据类型之间的映射和转换                                             |
| MappedStatement  | MappedStatement维护了一条<select update delete insert>节点的封装                  |
| SqlSource        | 负责根据用户传递的parameterObject，动态地生成SQL语句，将信息封装到BoundSql对象中，并返回               |
| BoundSql         | 表示动态生成的SQL语句以及相应的参数信息                                                   |

# MyBatis层次结构



## 9.3 总体流程

### (1) 加载配置并初始化

触发条件：加载配置文件

配置来源于两个地方，一个是配置文件(主配置文件conf.xml,mapper文件\*.xml)，一个是java代码中的注解，将主配置文件内容解析封装到Configuration,将sql的配置信息加载成为一个mappedstatement对象，存储在内存之中

## (2)接收调用请求

触发条件：调用Mybatis提供的API

传入参数：为SQL的ID和传入参数对象

处理过程：将请求传递给下层的请求处理层进行处理。

## (3)处理操作请求

触发条件：API接口层传递请求过来

传入参数：为SQL的ID和传入参数对象

处理过程：

(A)根据SQL的ID查找对应的MappedStatement对象。

(B)根据传入参数对象解析MappedStatement对象，得到最终要执行的SQL和执行传入参数。

(C)获取数据库连接，根据得到的最终SQL语句和执行传入参数到数据库执行，并得到执行结果。

(D)根据MappedStatement对象中的结果映射配置对得到的执行结果进行转换处理，并得到最终的处理结果。

(E)释放连接资源。

## (4)返回处理结果

将最终的处理结果返回。

# 第十部分 Mybatis源码剖析

## 10.1 传统方式源码剖析：

源码剖析-初始化

```
InputStream inputStream = Resources.getResourceAsStream("mybatis-config.xml");
//这一行代码正是初始化工作的开始。
SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(inputStream);
```

进入源码分析：

```
// 1.我们最初调用的build
public SqlSessionFactory build(InputStream inputStream) {
 //调用了重载方法
 return build(inputStream, null, null);
}

// 2.调用的重载方法
public SqlSessionFactory build(InputStream inputStream, String environment,
Properties properties) {
 try {
 // XMLConfigBuilder是专门解析mybatis的配置文件的类
```

```

XMLConfigBuilder parser = new XMLConfigBuilder(inputStream, environment,
properties);
//这里又调用了一个重载方法。parser.parse()的返回值是Configuration对象
return build(parser.parse());
} catch (Exception e) {
throw ExceptionFactory.wrapException("Error building SqlSession.", e)
}

```

MyBatis在初始化的时候，会将MyBatis的配置信息全部加载到内存中，使用org.apache.ibatis.session.Configuration实例来维护

下面进入对配置文件解析部分：

首先对Configuration对象进行介绍：

Configuration对象的结构和xml配置文件的对象几乎相同。

回顾一下xml中的配置标签有哪些：

properties（属性），settings（设置），typeAliases（类型别名），typeHandlers（类型处理器），objectFactory（对象工厂），mappers（映射器）等

Configuration也有对应的对象属性来封装它们

也就是说，初始化配置文件信息的本质就是创建Configuration对象，将解析的xml数据封装到Configuration内部属性中

```

/**
 * 解析 XML 成 Configuration 对象。
 */
public Configuration parse() {
 // 若已解析，抛出 BuilderException 异常
 if (parsed) {
 throw new BuilderException("Each XMLConfigBuilder can only be used
once.");
 }
 // 标记已解析
 parsed = true;
 // 解析 XML configuration 节点
 parseConfiguration(parser.evalNode("/configuration"));
 return configuration;
}

/**
 * 解析 XML
 */
private void parseConfiguration(XNode root) {
 try {
 //issue #117 read properties first
 // 解析 <properties /> 标签
 propertiesElement(root.evalNode("properties"));
 // 解析 <settings /> 标签
 }
}

```

```

Properties settings =
settingsAsProperties(root.evalNode("settings"));
 // 加载自定义的 VFS 实现类
 loadCustomVfs(settings);
 // 解析 <typeAliases /> 标签
 typeAliasesElement(root.evalNode("typeAliases"));
 // 解析 <plugins /> 标签
 pluginElement(root.evalNode("plugins"));
 // 解析 <objectFactory /> 标签
 objectFactoryElement(root.evalNode("objectFactory"));
 // 解析 <objectWrapperFactory /> 标签

 objectWrapperFactoryElement(root.evalNode("objectWrapperFactory"));
 // 解析 <reflectorFactory /> 标签
 reflectorFactoryElement(root.evalNode("reflectorFactory"));
 // 赋值 <settings /> 到 Configuration 属性
 settingsElement(settings);
 // read it after objectFactory and objectWrapperFactory issue #631
 // 解析 <environments /> 标签
 environmentsElement(root.evalNode("environments"));
 // 解析 <databaseIdProvider /> 标签
 databaseIdProviderElement(root.evalNode("databaseIdProvider"));
 // 解析 <typeHandlers /> 标签
 typeHandlerElement(root.evalNode("typeHandlers"));
 // 解析 <mappers /> 标签
 mapperElement(root.evalNode("mappers"));
} catch (Exception e) {
 throw new BuilderException("Error parsing SQL Mapper
Configuration. Cause: " + e, e);
}
}
}

```

介绍一下MappedStatement：

作用： MappedStatement与Mapper配置文件中的一个select/update/insert/delete节点相对应。 mapper中配置的标签都被封装到了此对象中，主要用途是描述一条SQL语句。 初始化过程：回顾刚刚开始介绍的加载配置文件的过程中，会对mybatis-config.xml中的各个标签都进行解析，其中有 mappers 标签用来引入mapper.xml文件或者配置mapper接口的目录。

```

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

```

这样的一个select标签会在初始化配置文件时被解析封装成一个MappedStatement对象，然后存储在 Configuration对象的mappedStatements属性中， mappedStatements 是一个HashMap，存储时key = 全限定类名 + 方法名， value = 对应的MappedStatement对象。

- 在configuration中对应的属性为

```
Map<String, MappedStatement> mappedStatements = new StrictMap<MappedStatement>("Mapped Statements collection")
```

- 在XMLConfigBuilder中的处理：

```
private void parseConfiguration(XNode root) {
 try {
 // 省略其他标签的处理
 mapperElement(root.evalNode("mappers"));
 } catch (Exception e) {
 throw new BuilderException("Error parsing SQL Mapper Configuration.
Cause: " + e, e);
 }
}
```

到此对xml配置文件的解析就结束了，回到步骤 2. 中调用的重载build方法

```
// 5. 调用的重载方法
public SqlSessionFactory build(Configuration config) {
 // 创建了DefaultSqlSessionFactory对象，传入Configuration对象。
 return new DefaultSqlSessionFactory(config);
}
```

## 源码剖析-执行SQL流程

先简单介绍**SqlSession**：

SqlSession是一个接口，它有两个实现类：DefaultSqlSession（默认）和SqlSessionManager（弃用，不做介绍）

SqlSession是MyBatis中用于和数据库交互的顶层类，通常将它与ThreadLocal绑定，一个会话使用一个SqlSession，并且在使用完毕后需要close。

```
public class DefaultSqlSession implements SqlSession {

 private final Configuration configuration;
 private final Executor executor;
```

SqlSession中的两个最重要的参数，configuration与初始化时的相同，Executor为执行器

## Executor：

Executor也是一个接口，他有三个常用的实现类：  
BatchExecutor（重用语句并执行批量更新）  
ReuseExecutor（重用预处理语句prepared statements）  
SimpleExecutor（普通的执行器，默认）

继续分析，初始化完毕后，我们就要执行SQL了

```
SqlSession sqlSession = factory.openSession();
List<User> list =
sqlSession.selectList("com.lagou.mapper.UserMapper.getUserByName");
```

获得sqlSession

```
//6. 进入openSession方法。
public SqlSession openSession() {
 //getDefaultExecutorType()传递的是SimpleExecutor
 return openSessionFromDataSource(configuration.getDefaultExecutorType(),
null, false);
}

//7. 进入openSessionFromDataSource。
//ExecutorType 为Executor的类型, TransactionIsolationLevel为事务隔离级别,
autoCommit是否开启事务
//openSession的多个重载方法可以指定获得的SeqSession的Executor类型和事务的处理
private SqlSession openSessionFromDataSource(ExecutorType execType,
TransactionIsolationLevel level, boolean autoCommit) {
 Transaction tx = null;
 try {
 final Environment environment = configuration.getEnvironment();
 final TransactionFactory transactionFactory =
getTransactionFactoryFromEnvironment(environment);
 tx = transactionFactory.newTransaction(environment.getDataSource(),
level, autoCommit);
 //根据参数创建指定类型的Executor
 final Executor executor = configuration.newExecutor(tx, execType);
 //返回的是DefaultSqlSession
 return new DefaultSqlSession(configuration, executor, autoCommit);
 } catch (Exception e) {
 closeTransaction(tx); // may have fetched a connection so lets call
close()
 }
}
```

执行sqlsession中的api

```
//8.进入selectList方法, 多个重载方法。
public <E> List<E> selectList(String statement) {
 return this.selectList(statement, null);
}
public <E> List<E> selectList(String statement, Object parameter) {
 return this.selectList(statement, parameter, RowBounds.DEFAULT);
}

public <E> List<E> selectList(String statement, Object parameter, RowBounds
rowBounds) {
 try {
```

```

//根据传入的全限定名+方法名从映射的Map中取出MappedStatement对象
MappedStatement ms = configuration.getMappedStatement(statement);
//调用Executor中的方法处理
//RowBounds是用来逻辑分页
//wrapCollection(parameter)是用来装饰集合或者数组参数
return executor.query(ms, wrapCollection(parameter), rowBounds,
Executor.NO_RESULT_HANDLER);
} catch (Exception e) {
throw ExceptionFactory.wrapException("Error querying database. Cause: "
+ e, e);
} finally {
ErrorContext.instance().reset();
}

```

## 源码剖析-executor

继续源码中的步骤，进入 executor.query()

```

//此方法在SimpleExecutor的父类BaseExecutor中实现
public <E> List<E> query(MappedStatement ms, Object parameter, RowBounds
rowBounds, ResultHandler resultHandler) throws SQLException {
//根据传入的参数动态获得SQL语句，最后返回用BoundSql对象表示
BoundSql boundSql = ms.getBoundSql(parameter);
//为本次查询创建缓存的key
CacheKey key = createCacheKey(ms, parameter, rowBounds, boundSql);
return query(ms, parameter, rowBounds, resultHandler, key, boundSql);
}

//进入query的重载方法中
public <E> List<E> query(MappedStatement ms, Object parameter, RowBounds
rowBounds, ResultHandler resultHandler, CacheKey key, BoundSql boundSql)
throws SQLException {
ErrorContext.instance().resource(ms.getResource()).activity("executing a
query").object(ms.getId());
if (closed) {
throw new ExecutorException("Executor was closed.");
}
if (queryStack == 0 && ms.isFlushCacheRequired()) {
clearLocalCache();
}
List<E> list;
try {
queryStack++;
list = resultHandler == null ? (List<E>) localCache.getObject(key) :
null;
if (list != null) {
handleLocallyCachedOutputParameters(ms, key, parameter, boundSql);
}
}
}

```

```

 } else {
 // 如果缓存中没有本次查找的值，那么从数据库中查询
 list = queryFromDatabase(ms, parameter, rowBounds, resultHandler, key,
 boundSql);
 }
} finally {
 queryStack--;
}
if (queryStack == 0) {
 for (DeferredLoad deferredLoad : deferredLoads) {
 deferredLoad.load();
 }
 // issue #601
 deferredLoads.clear();
 if (configuration.getLocalCacheScope() == LocalCacheScope.STATEMENT) {
 // issue #482
 clearLocalCache();
 }
}
return list;
}

//从数据库查询
private <E> List<E> queryFromDatabase(MappedStatement ms, Object parameter,
RowBounds rowBounds, ResultHandler resultHandler, CacheKey key, BoundSql
boundSql) throws SQLException {
 List<E> list;
 localCache.putObject(key, EXECUTION_PLACEHOLDER);
 try {
 // 查询的方法
 list = doQuery(ms, parameter, rowBounds, resultHandler, boundSql);
 } finally {
 localCache.removeObject(key);
 }
 // 将查询结果放入缓存
 localCache.putObject(key, list);
 if (ms.getStatementType() == StatementType.CALLABLE) {
 localOutputParameterCache.putObject(key, parameter);
 }
 return list;
}

// SimpleExecutor中实现父类的doQuery抽象方法
public <E> List<E> doQuery(MappedStatement ms, Object parameter, RowBounds
rowBounds, ResultHandler resultHandler, BoundSql boundSql) throws SQLException
{
 Statement stmt = null;
 try {
 Configuration configuration = ms.getConfiguration();

```

```

// 传入参数创建StatementHandler对象来执行查询
StatementHandler handler = configuration.newStatementHandler(wrapper,
ms, parameter, rowBounds, resultHandler, boundSql);
// 创建jdbc中的statement对象
stmt = prepareStatement(handler, ms.getStatementLog());
// StatementHandler进行处理
return handler.query(stmt, resultHandler);
} finally {
closeStatement(stmt);
}
}

// 创建Statement的方法
private Statement prepareStatement(StatementHandler handler, Log statementLog)
throws SQLException {
Statement stmt;
//条代码中的getConnection方法经过重重调用最后会调用openConnection方法，从连接池中获得连接。
Connection connection = getConnection(statementLog);
stmt = handler.prepare(connection, transaction.getTimeout());
handler.parameterize(stmt);
return stmt;
}
//从连接池获得连接的方法
protected void openConnection() throws SQLException {
if (log.isDebugEnabled()) {
log.debug("Opening JDBC Connection");
}
//从连接池获得连接
connection = dataSource.getConnection();
if (level != null) {
connection.setTransactionIsolation(level.getLevel());
}
}

```

上述的Executor.query()方法几经转折，最后会创建一个StatementHandler对象，然后将必要的参数传递给

StatementHandler，使用StatementHandler来完成对数据库的查询，最终返回List结果集。

从上面的代码中我们可以看出，Executor的功能和作用是：

- (1、根据传递的参数，完成SQL语句的动态解析，生成BoundSql对象，供StatementHandler使用；
- (2、为查询创建缓存，以提高性能
- (3、创建JDBC的Statement连接对象，传递给\*StatementHandler\*对象，返回List查询结果。

## 源码剖析-StatementHandler

StatementHandler对象主要完成两个工作：

- 对于JDBC的PreparedStatement类型的对象，创建的过程中，我们使用的是SQL语句字符串会包

含若干个? 占位符，我们其后再对占位符进行设值。StatementHandler通过parameterize(statement)方法对Statement进行设值；

- StatementHandler通过List query(Statement statement, ResultHandler resultHandler)方法来完成执行Statement，和将Statement对象返回的resultSet封装成List；

进入到StatementHandler的parameterize(statement) 方法的实现：

```
public void parameterize(Statement statement) throws SQLException {
 // 使用ParameterHandler对象来完成对Statement的设值
 parameterHandler.setParameters((PreparedStatement) statement);
}
```

```
/*
 *
 * ParameterHandler类的setParameters(PreparedStatement ps) 实现
 * 对某一个Statement进行设置参数
 */

public void setParameters(PreparedStatement ps) throws SQLException {
 ErrorContext.instance().activity("setting
parameters").object(mappedStatement.getParameterMap().getId());
 List<ParameterMapping> parameterMappings = boundSql.getParameterMappings();

 if (parameterMappings != null) {
 for (int i = 0; i < parameterMappings.size(); i++) {
 ParameterMapping parameterMapping = parameterMappings.get(i);
 if (parameterMapping.getMode() != ParameterMode.OUT) {
 Object value;
 String propertyName = parameterMapping.getProperty();
 if (boundSql.hasAdditionalParameter(propertyName)) { // issue #448 ask
first for additional params
 value = boundSql.getAdditionalParameter(propertyName);
 } else if (parameterObject == null) {
 value = null;
 } else if
(typeHandlerRegistry.hasTypeHandler(parameterObject.getClass())) {
 value = parameterObject;
 } else {
 MetaObject metaObject =
configuration.newMetaObject(parameterObject);
 value = metaObject.getValue(propertyName);
 }
 }
 }
 }
 // 每一个Mapping都有一个TypeHandler，根据TypeHandler来对preparedStatement进
行设置参数
 TypeHandler typeHandler = parameterMapping.getTypeHandler();
 JdbcType jdbcType = parameterMapping.getJdbcType();
 if (value == null && jdbcType == null) jdbcType =
configuration.getJdbcTypeForNull();
}
```

```
// 设置参数
 typeHandler.setParameter(ps, i + 1, value, jdbcType);
 }
}
}
```

从上述的代码可以看到,StatementHandler 的parameterize(Statement) 方法调用了 ParameterHandler的setParameters(statement) 方法,

ParameterHandler的setParameters(Statement)方法负责根据我们输入的参数，对statement对象的？占位符处进行赋值。

进入到StatementHandler 的List query(Statement statement, ResultHandler resultHandler)方法的实现：

```
public <E> List<E> query(Statement statement, ResultHandler resultHandler)
throws SQLException {
// 1.调用preparedStatemnt。execute()方法, 然后将resultSet交给resultSetHandler处理

PreparedStatement ps = (PreparedStatement) statement;
ps.execute();
//2. 使用ResultHandler来处理ResultSet
return resultSetHandler.<E> handleResultSets(ps);
}
```

从上述代码我们可以看出，StatementHandler 的List query(Statement statement, ResultHandler resultHandler)方法的实现，是调用了ResultSetHandler的handleResultSets(Statement) 方法。ResultSetHandler的handleResultSets(Statement) 方法会将Statement语句执行后生成的resultSet 结果集转换成List 结果集

```
public List<Object> handleResultSet(Statement stmt) throws SQLException {
 ErrorContext.instance().activity("handling
results").object(mappedStatement.getId());
}

// 多 ResultSet 的结果集合，每个 ResultSet 对应一个 Object 对象。而实际上，每
个 Object 是 List<Object> 对象。
// 在不考虑存储过程的多 ResultSet 的情况，普通的查询，实际就一个 ResultSet，也
就是说，multipleResults 最多就一个元素。
final List<Object> multipleResults = new ArrayList<>();

int resultSetCount = 0;
// 获得首个 ResultSet 对象，并封装成 ResultSetWrapper 对象
ResultSetWrapper rsw = getFirstResultSet(stmt);

// 获得 resultMap 数组
// 在不考虑存储过程的多 ResultSet 的情况，普通的查询，实际就一个 ResultSet，也
就是说，resultMaps 就一个元素。
List<ResultMap> resultMaps = mappedStatement.getResultMaps();
```

```

int resultMapCount = resultMaps.size();
validateResultMapsCount(rsw, resultMapCount); // 校验
while (rsw != null && resultMapCount > resultSetCount) {
 // 获得 ResultMap 对象
 ResultMap resultMap = resultMaps.get(resultSetCount);
 // 处理 ResultSet , 将结果添加到 multipleResults 中
 handleResultSet(rsw, resultMap, multipleResults, null);
 // 获得下一个 ResultSet 对象, 并封装成 ResultSetWrapper 对象
 rsw = getNextResultSet(stmt);
 // 清理
 cleanUpAfterHandlingResultSet();
 // resultSetCount ++
 resultSetCount++;
}

// 因为 `mappedStatement.resultSets` 只在存储过程中使用, 本系列暂时不考虑, 忽略即可
String[] resultSets = mappedStatement.getResultSets();
if (resultSets != null) {
 while (rsw != null && resultSetCount < resultSets.length) {
 ResultMapping parentMapping =
nextResultMaps.get(resultSets[resultSetCount]);
 if (parentMapping != null) {
 String nestedResultMapId =
parentMapping.getNestedResultMapId();
 ResultMap resultMap =
configuration.getResultMap(nestedResultMapId);
 handleResultSet(rsw, resultMap, null, parentMapping);
 }
 rsw = getNextResultSet(stmt);
 cleanUpAfterHandlingResultSet();
 resultSetCount++;
 }
}

// 如果是 multipleResults 单元素, 则取首元素返回
return collapseSingleResultList(multipleResults);
}

```

## 10.2 Mapper代理方式:

回顾下写法:

```

public static void main(String[] args) {
 //前三步都相同
 InputStream inputStream =
 Resources.getResourceAsStream("sqlMapConfig.xml");
 SqlSessionFactory factory = new
 SqlSessionFactoryBuilder().build(inputStream);
 SqlSession sqlSession = factory.openSession();

 //这里不再调用SqlSession 的api, 而是获得了接口对象, 调用接口中的方法。
 UserMapper mapper = sqlSession.getMapper(UserMapper.class);
 List<User> list = mapper.getUserByName("tom");
}

```

思考一个问题，通常的Mapper接口我们都没有实现的方法却可以使用，是什么呢？答案很简单 **动态代理**

开始之前介绍一下MyBatis初始化时对接口的处理：MapperRegistry是Configuration中的一个属性，它内部维护一个HashMap用于存放mapper接口的工厂类，每个接口对应一个工厂类。mappers中可以配置接口的包路径，或者某个具体的接口类。

```

<mappers>
 <mapper class="com.lagou.mapper.UserMapper" />
 <package name="com.lagou.mapper" />
</mappers>

```

- 当解析mappers标签时，它会判断解析到的是mapper配置文件时，会再将对应配置文件中的增删改查标签一封装成MappedStatement对象，存入mappedStatements中。（上文介绍了）当判断解析到接口时，会  
建此接口对应的MapperProxyFactory对象，存入HashMap中，key = 接口的字节码对象，value = 此接口对应的MapperProxyFactory对象。

### 源码剖析-getMapper()

进入sqlSession.getMapper(UserMapper.class)中

```

//DefaultSqlSession中的getMapper
public <T> T getMapper(Class<T> type) {
 return configuration.<T>getMapper(type, this);
}

//configuration中的给getMapper
public <T> T getMapper(Class<T> type, SqlSession sqlSession) {
 return mapperRegistry.getMapper(type, sqlSession);
}

//MapperRegistry中的getMapper
public <T> T getMapper(Class<T> type, SqlSession sqlSession) {
 //从MapperRegistry中的HashMap中拿MapperProxyFactory
}

```

```

 final MapperProxyFactory<T> mapperProxyFactory = (MapperProxyFactory<T>)
knownMappers.get(type);
 if (mapperProxyFactory == null) {
 throw new BindingException("Type " + type + " is not known to the
MapperRegistry.");
 }
 try {
 // 通过动态代理工厂生成示例。
 return mapperProxyFactory.newInstance(sqlSession);
 } catch (Exception e) {
 throw new BindingException("Error getting mapper instance. Cause: " + e,
e);
 }
}

//MapperProxyFactory类中的newInstance方法
public T newInstance(SqlSession sqlSession) {
 // 创建了JDK动态代理的Handler类
 final MapperProxy<T> mapperProxy = new MapperProxy<>(sqlSession,
mapperInterface, methodCache);
 // 调用了重载方法
 return newInstance(mapperProxy);
}

//MapperProxy类，实现了InvocationHandler接口
public class MapperProxy<T> implements InvocationHandler, Serializable {

 //省略部分源码

 private final SqlSession sqlSession;
 private final Class<T> mapperInterface;
 private final Map<Method, MapperMethod> methodCache;

 // 构造，传入了SqlSession，说明每个session中的代理对象的不同的！
 public MapperProxy(SqlSession sqlSession, Class<T> mapperInterface,
Map<Method, MapperMethod> methodCache) {
 this.sqlSession = sqlSession;
 this.mapperInterface = mapperInterface;
 this.methodCache = methodCache;
 }

 //省略部分源码
}

```

## 源码剖析-`invoke()`

在动态代理返回了示例后，我们就可以直接调用mapper类中的方法了，但代理对象调用方法，执行是在MapperProxy中的`invoke`方法中

```

public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
 try {
 // 如果是 Object 定义的方法, 直接调用
 if (Object.class.equals(method.getDeclaringClass())) {
 return method.invoke(this, args);

 } else if (isDefaultMethod(method)) {
 return invokeDefaultMethod(proxy, method, args);
 }
 } catch (Throwable t) {
 throw ExceptionUtil.unwrapThrowable(t);
 }
 // 获得 MapperMethod 对象
 final MapperMethod mapperMethod = cachedMapperMethod(method);
 // 重点在这: MapperMethod最终调用了执行的方法
 return mapperMethod.execute(sqlSession, args);
}

```

进入execute方法:

```

public Object execute(SqlSession sqlSession, Object[] args) {
 Object result;
 //判断mapper中的方法类型, 最终调用的还是SqlSession中的方法
 switch (command.getType()) {
 case INSERT: {
 // 转换参数
 Object param = method.convertArgsToSqlCommandParam(args);
 // 执行 INSERT 操作
 // 转换 rowCount
 result = rowCountResult(sqlSession.insert(command.getName(),
param));
 break;
 }
 case UPDATE: {
 // 转换参数
 Object param = method.convertArgsToSqlCommandParam(args);
 // 转换 rowCount
 result = rowCountResult(sqlSession.update(command.getName(),
param));
 break;
 }
 case DELETE: {
 // 转换参数
 Object param = method.convertArgsToSqlCommandParam(args);
 // 转换 rowCount
 result = rowCountResult(sqlSession.delete(command.getName(),
param));
 break;
 }
 }
}

```

```
 break;
 }

 case SELECT:
 // 无返回，并且有 ResultHandler 方法参数，则将查询的结果，提交给
 ResultHandler 进行处理
 if (method.returnsVoid() && method.hasResultHandler()) {
 executeWithResultHandler(sqlSession, args);
 result = null;
 }
 // 执行查询，返回列表
 } else if (method.returnsMany()) {
 result = executeForMany(sqlSession, args);
 }
 // 执行查询，返回 Map
 } else if (method.returnsMap()) {
 result = executeForMap(sqlSession, args);
 }
 // 执行查询，返回 Cursor
 } else if (method.returnsCursor()) {
 result = executeForCursor(sqlSession, args);
 }
 // 执行查询，返回单个对象
 } else {
 // 转换参数
 Object param = method.convertArgsToSqlCommandParam(args);
 // 查询单条
 result = sqlSession.selectOne(command.getName(), param);
 if (method>ReturnsOptional() &&
 (result == null ||
 !method.getReturnType().equals(result.getClass())))
 {
 result = Optional.ofNullable(result);
 }
 }
 break;
 case FLUSH:
 result = sqlSession.flushStatements();
 break;
 default:
 throw new BindingException("Unknown execution method for: " +
command.getName());
 }
 // 返回结果为 null，并且返回类型为基本类型，则抛出 BindingException 异常
 if (result == null && method.getReturnType().isPrimitive() &&
!method.returnsVoid())
 {
 throw new BindingException("Mapper method '" + command.getName()
+ " attempted to return null from a method with a primitive
return type (" + method.getReturnType() + ").");
 }
 // 返回结果
 return result;
}
```

# 第十一部分 设计模式

虽然我们都知道有3类23种设计模式，但是大多停留在概念层面，Mybatis源码中使用了大量的设计模式，观察设计模式在其中的应用，能够更深入的理解设计模式

Mybatis至少用到了以下的设计模式的使用：

| 模式        | mybatis体现                                                                                            |
|-----------|------------------------------------------------------------------------------------------------------|
| Builder模式 | 例如SqlSessionFactoryBuilder、Environment;                                                              |
| 工厂方法模式    | 例如SqlSessionFactory、TransactionFactory、LogFactory                                                    |
| 单例模式      | 例如ErrorContext和LogFactory;                                                                           |
| 代理模式      | Mybatis实现的核心，比如MapperProxy、ConnectionLogger，用的jdk的动态代理还有executor.loader包使用了cglib或者javassist达到延迟加载的效果 |
| 组合模式      | 例如SqlNode和各个子类ChooseSqlNode等；                                                                        |
| 模板方法模式    | 例如BaseExecutor和SimpleExecutor，还有BaseTypeHandler和所有的子类例如IntegerTypeHandler；                           |
| 适配器模式     | 例如Log的Mybatis接口和它对jdbc、log4j等各种日志框架的适配实现；                                                            |
| 装饰者模式     | 例如Cache包中的cache.decorators子包中等各个装饰者的实现；                                                              |
| 迭代器模式     | 例如迭代器模式PropertyTokenizer；                                                                            |

接下来对Builder构建者模式、工厂模式、代理模式进行解读，先介绍模式自身的知识，然后解读在Mybatis中怎样应用了该模式。

## 11.1 Builder构建者模式

Builder模式的定义是“将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。”，它属于创建类模式，一般来说，如果一个对象的构建比较复杂，超出了构造函数所能包含的范围，就可以使用工厂模式和Builder模式，相对于工厂模式会产出一个完整的产品，Builder应用于更加复杂的对象的构建，甚至只会构建产品的一个部分，直白来说，就是使用多个简单的对象一步一步构建成为一个复杂的对象

例子：使用构建者设计模式来生产computer

主要步骤：

1、将需要构建的目标类分成多个部件（电脑可以分为主机、显示器、键盘、音箱等部件）；

- 2、创建构建类；
- 3、依次创建部件；
- 4、将部件组装成目标对象

## 1. 定义

computer

```
public class Computer {
 private String displayer;
 private String mainUnit;
 private String mouse;
 private String keyboard;

 public String getDisplayer() {
 return displayer;
 }

 public void setDisplayer(String displayer) {
 this.displayer = displayer;
 }

 public String getMainUnit() {
 return mainUnit;
 }

 public void setMainUnit(String mainUnit) {
 this.mainUnit = mainUnit;
 }

 public String getMouse() {
 return mouse;
 }

 public void setMouse(String mouse) {
 this.mouse = mouse;
 }

 public String getKeyboard() {
 return keyboard;
 }

 public void setKeyboard(String keyboard) {
 this.keyboard = keyboard;
 }

 @Override
 public String toString() {
```

```
 return "Computer{" +
 "displayer='" + displaye + '\'' +
 ", mainUnit='" + mainUnit + '\'' +
 ", mouse='" + mouse + '\'' +
 ", keyboard='" + keyboard + '\'' +
 '}';
 }
```

## ComputerBuilder

```
public static class ComputerBuilder{
 private ComputerBuilder target =new ComputerBuilder();

 public Builder installDisplayer(String displaye){
 target.setDisplaye(displaye);
 return this;
 }

 public Builder installMainUnit(String mainUnit){
 target.setMainUnit(mainUnit);
 return this;
 }

 public Builder installMouse(String mouse){
 target.setMouse(mouse);
 return this;
 }

 public Builder installKeyboard(String keyboard){
 target.setKeyboard(keyboard);
 return this;
 }

 public ComputerBuilder build(){
 return target;
 }
}
```

## 2.调用

```

public static void main(String[] args) {
 ComputerBuilder computerBuilder = new ComputerBuilder();

 computerBuilder.installDisplayer("显示器");
 computerBuilder.installMainUnit("主机");
 computerBuilder.installKeybord("键盘");
 computerBuilder.installMouse("鼠标");
 Computer computer = computerBuilder.Builder();

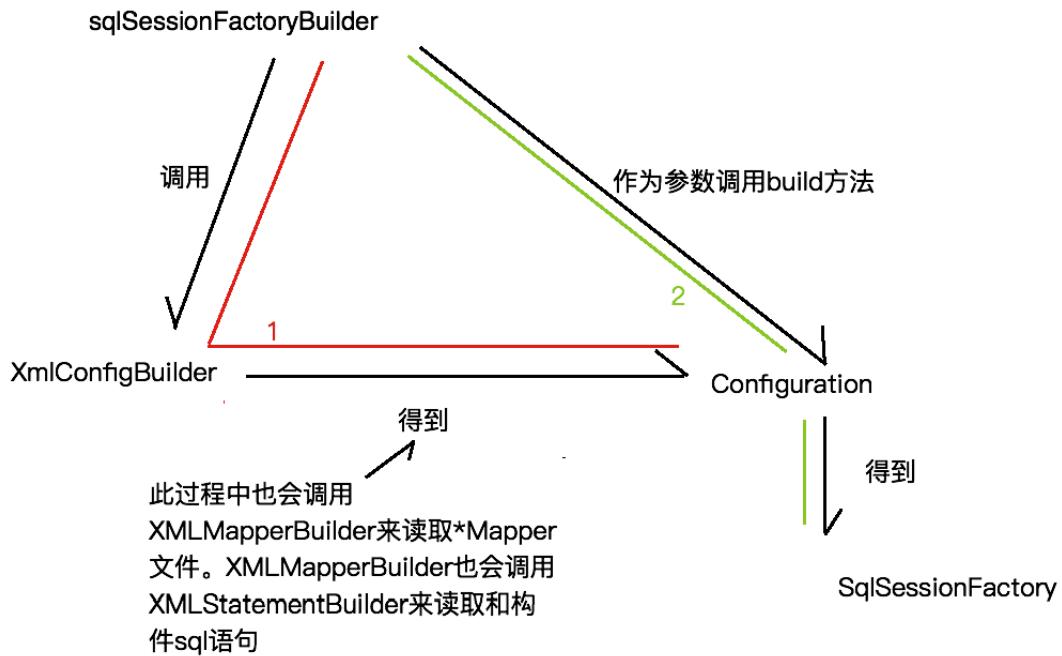
 System.out.println(computer);
}

```

## Mybatis中的体现

SqlSessionFactory的构建过程：

Mybatis的初始化工作非常复杂，不是只用一个构造函数就能搞定的。所以使用了建造者模式，使用了大量的Builder，进行分层构造，核心对象Configuration使用了XmlConfigBuilder来进行构造



在Mybatis环境的初始化过程中，`SqlSessionFactoryBuilder`会调用`XMLConfigBuilder`读取所有的`MybatisMapConfig.xml`和所有的`*Mapper.xml`文件，构建Mybatis运行的核心对象`Configuration`对象，然后将该`Configuration`对象作为参数构建一个`SqlSessionFactory`对象。

```

private void parseConfiguration(XNode root) {
 try {
 //issue #117 read properties first
 // 解析 <properties /> 标签
 propertiesElement(root.evalNode("properties"));
 // 解析 <settings /> 标签
 Properties settings = settingsAsProperties(root.evalNode("settings"));
 // 加载自定义的 VFS 实现类
 loadCustomVfs(settings);
 // 解析 <typeAliases /> 标签
 typeAliasesElement(root.evalNode("typeAliases"));
 // 解析 <plugins /> 标签
 pluginElement(root.evalNode("plugins"));
 // 解析 <objectFactory /> 标签
 objectFactoryElement(root.evalNode("objectFactory"));
 // 解析 <objectWrapperFactory /> 标签
 objectWrapperFactoryElement(root.evalNode("objectWrapperFactory"));
 // 解析 <reflectorFactory /> 标签
 reflectorFactoryElement(root.evalNode("reflectorFactory"));
 // 赋值 <settings /> 到 Configuration 属性
 settingsElement(settings);
 // read it after objectFactory and objectWrapperFactory issue #631
 // 解析 <environments /> 标签
 environmentsElement(root.evalNode("environments"));
 // 解析 <databaseIdProvider /> 标签
 databaseIdProviderElement(root.evalNode("databaseIdProvider"));
 }
}

```

其中XMLConfigBuilder在构建Configuration对象时，也会调用XMLMapperBuilder用于读取\*Mapper文件，而XMLMapperBuilder会使用XMLStatementBuilder来读取和build所有的SQL语句。

```

// 解析 <mappers /> 标签
mapperElement(root.evalNode("mappers"));

```

在这个过程中，有一个相似的特点，就是这些Builder会读取文件或者配置，然后做大量的XpathParser解析、配置或语法的解析、反射生成对象、存入结果缓存等步骤，这么多的工作都不是一个构造函数所能包括的，因此大量采用了Builder模式来解决

|                                                                                                                                                                              |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <span style="color: #0070C0;">C</span> <span style="color: #0070C0;">m</span> <span style="color: #0070C0;">SqlSessionFactoryBuilder</span>                                  |
| <span style="color: #E91E63;">m</span> <span style="color: #0070C0;">m</span> <span style="color: #0070C0;">build(Reader): SqlSessionFactory</span>                          |
| <span style="color: #E91E63;">m</span> <span style="color: #0070C0;">m</span> <span style="color: #0070C0;">build(Reader, String): SqlSessionFactory</span>                  |
| <span style="color: #E91E63;">m</span> <span style="color: #0070C0;">m</span> <span style="color: #0070C0;">build(Reader, Properties): SqlSessionFactory</span>              |
| <span style="color: #E91E63;">m</span> <span style="color: #0070C0;">m</span> <span style="color: #0070C0;">build(Reader, String, Properties): SqlSessionFactory</span>      |
| <span style="color: #E91E63;">m</span> <span style="color: #0070C0;">m</span> <span style="color: #0070C0;">build(InputStream): SqlSessionFactory</span>                     |
| <span style="color: #E91E63;">m</span> <span style="color: #0070C0;">m</span> <span style="color: #0070C0;">build(InputStream, String): SqlSessionFactory</span>             |
| <span style="color: #E91E63;">m</span> <span style="color: #0070C0;">m</span> <span style="color: #0070C0;">build(InputStream, Properties): SqlSessionFactory</span>         |
| <span style="color: #E91E63;">m</span> <span style="color: #0070C0;">m</span> <span style="color: #0070C0;">build(InputStream, String, Properties): SqlSessionFactory</span> |
| <span style="color: #E91E63;">m</span> <span style="color: #0070C0;">m</span> <span style="color: #0070C0;">build(Configuration): SqlSessionFactory</span>                   |

SqlSessionFactoryBuilder类根据不同的输入参数来构建SqlSessionFactory这个工厂对象

## 11.2 工厂模式

在Mybatis中比如SqlSessionFactory使用的是工厂模式，该工厂没有那么复杂的逻辑，是一个简单工厂模式。

简单工厂模式(Simple Factory Pattern): 又称为静态工厂方法(Static Factory Method)模式，它属于创建型模式。

在简单工厂模式中，可以根据参数的不同返回不同类的实例。简单工厂模式专门定义一个类来负责创建其他类的实例，被创建的实例通常都具有共同的父类

例子：生产电脑

假设有一个电脑的代工生产商，它目前已经可以代工生产联想电脑了，随着业务的拓展，这个代工生产商还要生产惠普的电脑，我们就需要用一个单独的类来专门生产电脑，这就用到了简单工厂模式。下面我们来实现简单工厂模式：

### 1. 创建抽象产品类

我们创建一个电脑的抽象产品类，他有一个抽象方法用于启动电脑：

```
public abstract class Computer {
 /**
 * 产品的抽象方法，由具体的产品类去实现
 */
 public abstract void start();
}
```

### 2. 创建具体产品类

接着我们创建各个品牌的电脑，他们都继承了他们的父类Computer，并实现了父类的start方法：

```
public class LenovoComputer extends Computer{
 @Override
 public void start() {
 System.out.println("联想电脑启动");
 }
}
```

```
public class HpComputer extends Computer{
 @Override
 public void start() {
 System.out.println("惠普电脑启动");
 }
}
```

### 3. 创建工厂类

接下来创建一个工厂类，它提供了一个静态方法createComputer用来生产电脑。你只需要传入你想生产的电脑的品牌，它就会实例化相应品牌的电脑对象

```
public class ComputerFactory {
 public static Computer createComputer(String type){
```

```

Computer mComputer=null;
switch (type) {
 case "lenovo":
 mComputer=new LenovoComputer();
 break;
 case "hp":
 mComputer=new HpComputer();
 break;

}
return mComputer;
}
}

```

客户端调用工厂类

客户端调用工厂类，传入“hp”生产出惠普电脑并调用该电脑对象的start方法：

```

public class CreatComputer {
 public static void main(String[] args){
 ComputerFactory.createComputer("hp").start();
 }
}

```

**Mybatis体现：**

Mybatis中执行Sql语句、获取Mappers、管理事务的核心接口SqlSession的创建过程使用到了工厂模式。

有一个SqlSessionFactory来负责SqlSession的创建

```

C DefaultSqlSessionFactory
m DefaultSqlSessionFactory(Configuration)
m openSession(): SqlSession ↑SqlSessionFactory
m openSession(boolean): SqlSession ↑SqlSessionFactory
m openSession(ExecutorType): SqlSession ↑SqlSessionFactory
m openSession(TransactionIsolationLevel): SqlSession ↑SqlSession
m openSession(ExecutorType, TransactionIsolationLevel): SqlSession ↑SqlSession
m openSession(ExecutorType, boolean): SqlSession ↑SqlSession
m openSession(Connection): SqlSession ↑SqlSessionFactory
m openSession(ExecutorType, Connection): SqlSession ↑SqlSession
m getConfiguration(): Configuration ↑SqlSessionFactory

```

SqlSessionFactory

可以看到，该Factory的 `openSession ()` 方法重载了很多个，分别支持 `autoCommit`、`Executor`、`Transaction` 等参数的输入，来构建核心的 `SqlSession` 对象。

在 `DefaultSqlSessionFactory` 的默认工厂实现里，有一个方法可以看出工厂怎么产出一个产品：

```

private SqlSession openSessionFromDataSource(ExecutorType execType,
TransactionIsolationLevel level, boolean autoCommit) {
 Transaction tx = null;
 try {
 final Environment environment = configuration.getEnvironment();
 final TransactionFactory transactionFactory =
getTransactionFactoryFromEnvironment(environment);
 tx = transactionFactory.newTransaction(environment.getDataSource(), level,
autoCommit);
 //根据参数创建制定类型的Executor
 final Executor executor = configuration.newExecutor(tx, execType);
 //返回的是 DefaultSqlSession
 return new DefaultSqlSession(configuration, executor, autoCommit);
 } catch (Exception e) {
 closeTransaction(tx); // may have fetched a connection so lets call
close()
 throw ExceptionFactory.wrapException("Error opening session. Cause: " +
e, e);
 } finally {
 ErrorContext.instance().reset();
 }
}

```

这是一个openSession调用的底层方法，该方法先从configuration读取对应的环境配置，然后初始化TransactionFactory获得一个Transaction对象，然后通过Transaction获取一个Executor对象，最后通过configuration、Executor、是否autoCommit三个参数构建了SqlSession

### 11.3 代理模式

代理模式(Proxy Pattern)：给某一个对象提供一个代理，并由代理对象控制对原对象的引用。代理模式的英文叫做Proxy，它是一种对象结构型模式，代理模式分为静态代理和动态代理，我们来介绍动态代理

举例：

创建一个抽象类，Person接口，使其拥有一个没有返回值的doSomething方法。

```

/**
 * 抽象类人
 */
public interface Person {
 void doSomething();
}

```

创建一个名为Bob的Person接口的实现类，使其实现doSomething方法

```
/**
 * 创建一个名为Bob的人的实现类
 */
public class Bob implements Person {
 public void doSomething() {
 System.out.println("Bob doing something!");
 }
}
```

(3) 创建JDK动态代理类，使其实现InvocationHandler接口。拥有一个名为target的变量，并创建getTarget获取代理对象方法

```
/**
 * JDK动态代理
 * 需实现InvocationHandler接口
 */
public class JDKDynamicProxy implements InvocationHandler {

 // 被代理的对象
 Person target;

 // JDKDynamicProxy构造函数
 public JDKDynamicProxy(Person person) {
 this.target = person;
 }

 // 获取代理对象
 public Person getTarget() {
 return (Person)
Proxy.newProxyInstance(target.getClass().getClassLoader(),
target.getClass().getInterfaces(), this);
 }

 // 动态代理invoke方法
 public Person invoke(Object proxy, Method method, Object[] args) throws
Throwable {
 // 被代理方法前执行
 System.out.println("JDKDynamicProxy do something before!");
 // 执行被代理的方法
 Person result = (Person) method.invoke(target, args);
 // 被代理方法后执行
 System.out.println("JDKDynamicProxy do something after!");
 return result;
 }
}
```

创建JDK动态代理测试类JDKDynamicTest

```

/**
 * JDK动态代理测试
 */
public class JDKDynamicTest {

 public static void main(String[] args) {

 System.out.println("不使用代理类,调用doSomething方法。");
 // 不使用代理类
 Person person = new Bob();
 // 调用doSomething方法
 person.doSomething();

 System.out.println("----- 分割线 -----");
 System.out.println("-----");

 System.out.println("使用代理类,调用doSomething方法。");
 // 获取代理类
 Person proxyPerson = new JDKDynamicProxy(new Bob()).getTarget();
 // 调用doSomething方法
 proxyPerson.doSomething();

 }
}

```

### Mybatis中实现：

代理模式可以认为是Mybatis的核心使用的模式，正是由于这个模式，我们只需要编写 `Mapper.java` 接口，不需要实现，由Mybatis后台帮我们完成具体SQL的执行。

当我们使用Configuration的getMapper方法时，会调用mapperRegistry.getMapper方法，而该方法又会调用mapperProxyFactory.newInstance(sqlSession)来生成一个具体的代理：

```

public class MapperProxyFactory<T> {

 private final Class<T> mapperInterface;
 private final Map<Method, MapperMethod> methodCache = new
 ConcurrentHashMap<Method, MapperMethod>();

 public MapperProxyFactory(Class<T> mapperInterface) {
 this.mapperInterface = mapperInterface;
 }

 public Class<T> getMapperInterface() {
 return mapperInterface;
 }

 public Map<Method, MapperMethod> getMethodCache() {
 return methodCache;
 }
}

```

```

 }

 @SuppressWarnings("unchecked")
 protected T newInstance(MapperProxy<T> mapperProxy) {
 return (T) Proxy.newProxyInstance(mapperInterface.getClassLoader(), new
Class[] { mapperInterface },
 mapperProxy);
 }

 public T newInstance(SqlSession sqlSession) {
 final MapperProxy<T> mapperProxy = new MapperProxy<T>(sqlSession,
mapperInterface, methodCache);
 return newInstance(mapperProxy);
 }
}

```

在这里，先通过T newInstance(SqlSession sqlSession)方法会得到一个MapperProxy对象，然后调用T newInstance(MapperProxy mapperProxy)生成代理对象然后返回。而查看MapperProxy的代码，可以看到如下内容：

```

public class MapperProxy<T> implements InvocationHandler, Serializable {

 @Override
 public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
 try {
 if (Object.class.equals(method.getDeclaringClass())) {
 return method.invoke(this, args);
 } else if (isDefaultMethod(method)) {
 return invokeDefaultMethod(proxy, method, args);
 }
 } catch (Throwable t) {
 throw ExceptionUtil.unwrapThrowable(t);
 }
 final MapperMethod mapperMethod = cachedMapperMethod(method);
 return mapperMethod.execute(sqlSession, args);
 }
}

```

非常典型的，该MapperProxy类实现了InvocationHandler接口，并且实现了该接口的invoke方法。通过这种方式，我们只需要编写Mapper.java接口类，当真正执行一个Mapper接口的时候，就会转发给MapperProxy.invoke方法，而该方法则会调用后续的sqlSession.createExecutor.execute.prepareStatement等一系列方法，完成SQL的执行和返回。

