课堂主题

Mybatis源码专题与手写框架分析与实现

课堂目标

- 掌握sqlsession执行流程的源码分析
- 掌握参数设置源码分析
- 掌握结果集映射源码分析
- 学会如何设计一个框架
- 学会如何手写mybatis框架

知识要点

源码分析篇

接口和对象介绍

SqlSessionFactoryBuilder

```
public class SqlSessionFactoryBuilder {

  public SqlSessionFactory build(Reader reader) {

  public SqlSessionFactory build(Reader reader, String environment) {

  public SqlSessionFactory build(Reader reader, Properties properties) {

  public SqlSessionFactory build(Reader reader, String environment, Properties properties) {
    try {
      XMLConfigBuilder parser = new XMLConfigBuilder(reader, environment, properties);
      return build(parser.parse());
    } catch (Exception e) {
      throw ExceptionFactory.wrapException("Error building SqlSession.", e);
    } finally {
      ErrorContext.instance().reset();
      try {
        reader.close();
      } catch (IOException e) {
        // Intentionally ignore. Prefer previous error.
      }
    }
  }
```

```java
public SqlSessionFactory build(InputStream inputStream) {

public SqlSessionFactory build(InputStream inputStream, String environment) {

public SqlSessionFactory build(InputStream inputStream, Properties properties) {

public SqlSessionFactory build(InputStream inputStream, String environment, Properties properties) {
  try {
    // XMLConfigBuilder:用来解析XML配置文件
    // 使用构建者模式
    XMLConfigBuilder parser = new XMLConfigBuilder(inputStream, environment, properties);
    // parser.parse()：使用XPATH解析XML配置文件，将配置文件封装为Configuration对象
    // 返回DefaultSqlSessionFactory对象，该对象拥有Configuration对象（封装配置文件信息）
    return build(parser.parse());
  } catch (Exception e) {
    throw ExceptionFactory.wrapException("Error building SqlSession.", e);
  } finally {
    ErrorContext.instance().reset();
    try {
      inputStream.close();
    } catch (IOException e) {
      // Intentionally ignore. Prefer previous error.
    }
```

XMLConfigBuilder

专门用来解析全局配置文件的解析器

XMLMapperBuilder

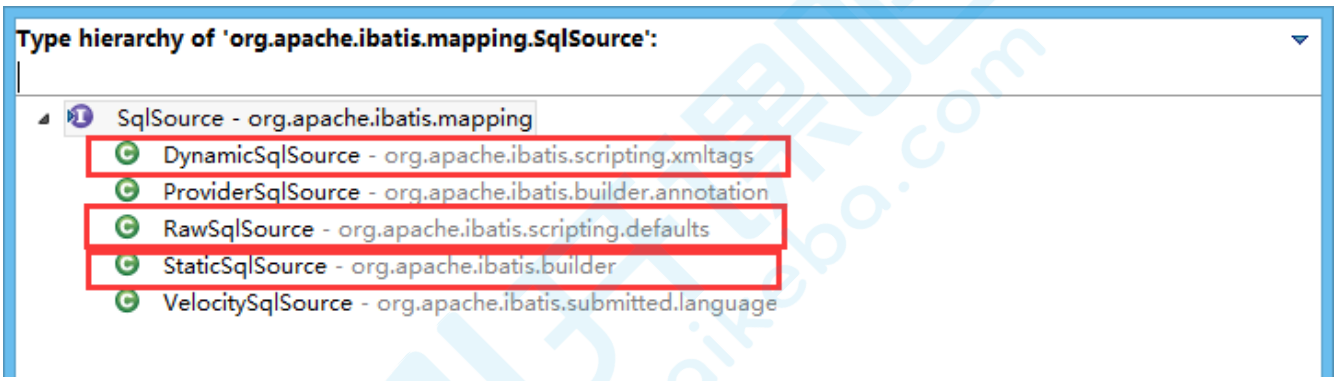
专门用来解析映射文件的解析器

Configuration

MyBatis框架支持开发人员通过配置文件与其进行交流.在配置文件所配置的信息,在框架运行时,会被XMLConfigBuilder解析并存储在一个Configuration对象中.Configuration对象会被作为参数传送给DeFaultSqlSessionFactory.而DeFaultSqlSessionFactory根据Configuration对象信息为Client创建对应特征的SqlSession对象

```java
private void parseConfiguration(XNode root) {
  try {
    //issue #117 read properties first
    propertiesElement(root.evalNode("properties"));
    Properties settings = settingsAsProperties(root.evalNode("settings"));
    loadCustomVfs(settings);
    typeAliasesElement(root.evalNode("typeAliases"));
    pluginElement(root.evalNode("plugins"));
    objectFactoryElement(root.evalNode("objectFactory"));
    objectWrapperFactoryElement(root.evalNode("objectWrapperFactory"));
    reflectorFactoryElement(root.evalNode("reflectorFactory"));
    settingsElement(settings);
    // read it after objectFactory and objectWrapperFactory issue #631
    environmentsElement(root.evalNode("environments"));
    databaseIdProviderElement(root.evalNode("databaseIdProvider"));
    typeHandlerElement(root.evalNode("typeHandlers"));
    mapperElement(root.evalNode("mappers"));
  } catch (Exception e) {
```

SqlSource接口



- DynamicSqlSource：主要是封装动态SQL标签解析之后的SQL语句和带有${}的SQL语句
- RawSqlSource：主要封装带有#{}的SQL语句
- StaticSqlSource：是BoundSql中要存储SQL语句的一个载体，上面两个SqlSource的SQL语句，最终都会存储到该SqlSource实现类中。

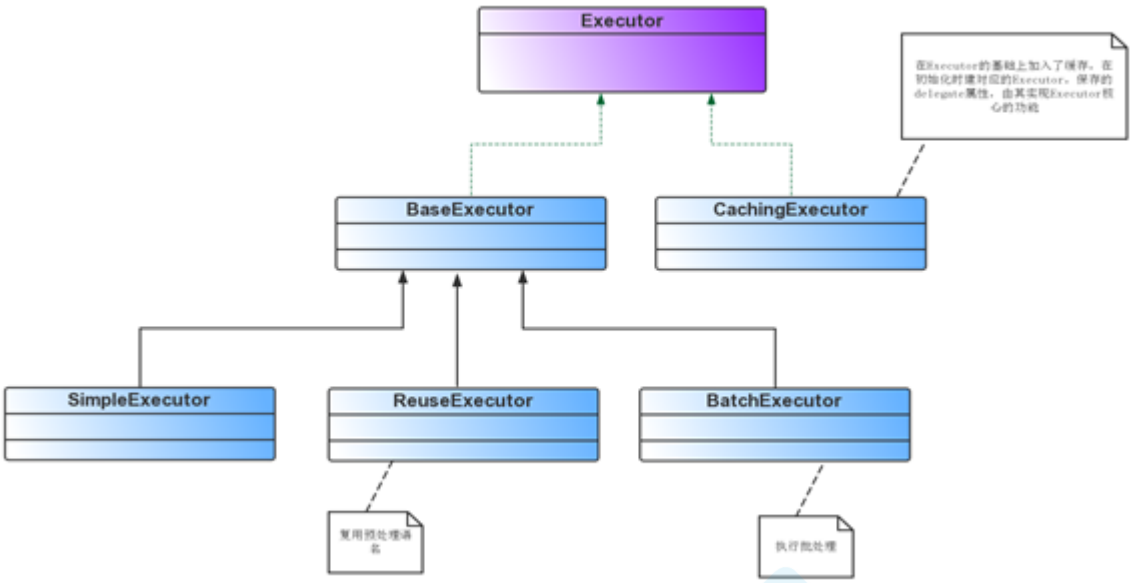
SQLSessionFactory接口

默认实现类是DefaultSQLSessionFactory类

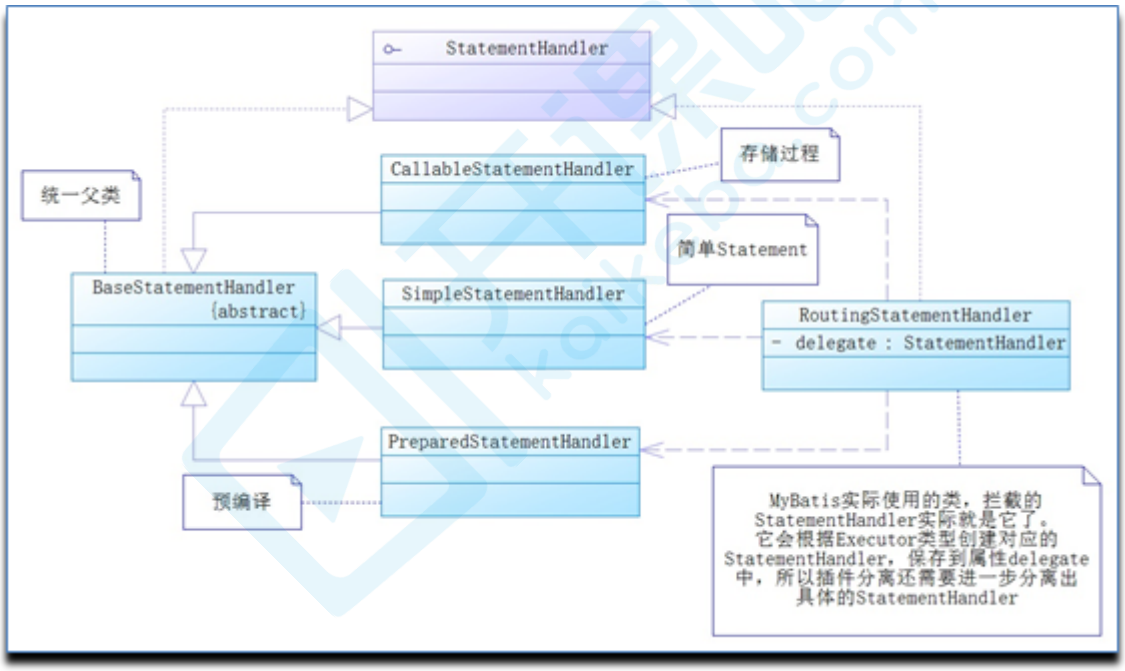
SqlSession接口

默认实现类是DefaultSQLSession类

Executor接口



StatementHandler接口



ParameterHandler接口

ResultSetHandler接口

默认实现类是DefaultResultSetHandler类。

源码阅读

SqlSource创建流程

- 找入口：XMLLanguageDriver#createSqlSource

```
    @Override
    public SqlSource createSqlSource(Configuration configuration, XNode script,
Class<?> parameterType) {
        // 初始化了动态SQL标签处理器
        XMLScriptBuilder builder = new XMLScriptBuilder(configuration, script,
parameterType);
        // 解析动态SQL
        return builder.parseScriptNode();
    }
```

- SqlSource创建流程分析

```
|--XMLLanguageDriver#createSqlSource：创建SqlSource
    |--XMLScriptBuilder#构造方法：初始化动态SQL中的节点处理器集合
    |--XMLScriptBuilder#parseScriptNode:
        |--XMLScriptBuilder#parseDynamicTags：解析select\insert\ update\delete标签中的
SQL语句，最终将解析到的SqlNode封装到MixedSqlNode中的List集合中
        |--DynamicSqlSource#构造方法：如果SQL中包含${}和动态SQL语句，则将SqlNode封装到
DynamicSqlSource
        |--RawSqlSource#构造方法：如果SQL中包含#{}，则将SqlNode封装到RawSqlSource中
            |--ParameterMappingTokenHandler#构造方法
            |--GenericTokenParser#构造方法：指定待分析的openToken和closeToken，并指定处理器
            |--GenericTokenParser#parse：解析SQL语句，处理openToken和closeToken中的内容
                |--ParameterMappingTokenHandler#handleToken：处理token（#{}/${}）
                    |--ParameterMappingTokenHandler#buildParameterMapping：创建
ParameterMapping对象
            |--StaticSqlSource#构造方法：将解析之后的SQL信息，封装到StaticSqlSource

```

- 相关类和接口：

```
    |--XMLLanguageDriver
    |--XMLScriptBuilder
    |--SqlSource
    |--SqlSourceBuilder
```

获取Mapper代理对象流程

- 找入口：DefaultSqlSession#getMapper

```
    @Override
    public <T> T getMapper(Class<T> type) {
        // 从Configuration对象中，根据Mapper接口，获取Mapper代理对象
        return configuration.<T>getMapper(type, this);
    }
```

- 流程分析

```
|--DefaultSqlSession#getMapper：获取Mapper代理对象
    |--Configuration#getMapper：获取Mapper代理对象
        |--MapperRegistry#getMapper：通过代理对象工厂，获取代理对象
            |--MapperProxyFactory#newInstance：调用JDK的动态代理方式，创建Mapper代理

```

SqlSession执行主流程

- 找入口：DefaultSqlSession#selectList()

```
public <E> List<E> selectList(String statement, Object parameter, RowBounds rowBounds) {
        try {
            // 根据传入的statementId，获取MappedStatement对象
            MappedStatement ms = configuration.getMappedStatement(statement);
            // 调用执行器的查询方法
            // 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();
        }
    }
```

- 流程分析

```
|--DefaultSqlSession#selectLIst
    |--CachingExecutor#query
        |--BaseExecutor#query
            |--BaseExecutor#queryFromDatabase
                |--SimpleExecutor#doQuery
                    |--Configuration#newStatementHandler:创建StatementHandler，用来执行MappedStatement对象
                        |--RoutingStatementHandler#构造方法：根据路由规则，设置不同的StatementHandler
                    |--SimpleExecutor#prepareStatement：主要是设置PreparedStatement的参数
                        |--SimpleExecutor#getConnection：获取数据库连接
                        |--PreparedStatementHandler#prepare：创建PreparedStatement对象
                        |--PreparedStatementHandler#parameterize：设置PreparedStatement的参数
                    |--PreparedStatementHandler#query：主要是用来执行SQL语句，及处理结果集
                        |--PreparedStatement#execute：调用JDBC的api执行Statement
                            |--DefaultResultSetHandler#handleResultSets:处理结果集
```

- 相关类和接口：

```
    |--DefaultSqlSession
    |--Executor
        |--CachingExecutor
        |--BaseExecutor
        |--SimpleExecutor
    |--StatementHandler
        |--RoutingStatementHandler
        |--PreparedStatementHandler
    |--ResultSetHandler
        |--DefaultResultSetHandler
```

BoundSql获取流程

- 找入口：MappedStatement#getBoundSql方法

```
public BoundSql getBoundSql(Object parameterObject) {
    // 调用SqlSource获取BoundSql
    BoundSql boundSql = sqlSource.getBoundSql(parameterObject);
    List<ParameterMapping> parameterMappings = boundSql.getParameterMappings();
    if (parameterMappings == null || parameterMappings.isEmpty()) {
      boundSql = new BoundSql(configuration, boundSql.getSql(),
parameterMap.getParameterMappings(), parameterObject);
    }

    // check for nested result maps in parameter mappings (issue #30)
    for (ParameterMapping pm : boundSql.getParameterMappings()) {
      String rmId = pm.getResultMapId();
      if (rmId != null) {
        ResultMap rm = configuration.getResultMap(rmId);
        if (rm != null) {
          hasNestedResultMaps |= rm.hasNestedResultMaps();
        }
      }
    }

    return boundSql;
  }
```

- 流程分析

```
|--DynamicSqlSource#getBoundSql
    |--SqlSourceBuilder#parse：解析SQL语句中的#{},并将对应的参数信息封装到ParameterMapping对象集合中，然后封装到StaticSqlSource中
        |--ParameterMappingTokenHandler#构造方法
        |--GenericTokenParser#构造方法：指定待分析的openToken和closeToken，并指定处理器
        |--GenericTokenParser#parse：解析SQL语句，处理openToken和closeToken中的内容
            |--ParameterMappingTokenHandler#handleToken：处理token（#{}/${}）
                |--ParameterMappingTokenHandler#buildParameterMapping：创建ParameterMapping对象
        |--StaticSqlSource#构造方法：将解析之后的SQL信息，封装到StaticSqlSource

|--RawSqlSource#getBoundSql
    |--StaticSqlSource#getBoundSql
        |--BoundSql#构造方法：将解析后的sql信息、参数映射信息、入参对象组合到BoundSql对象中
```

参数映射流程

- 找入口：PreparedStatementHandler#parameterize方法

```
  public void parameterize(Statement statement) throws SQLException {
    // 通过ParameterHandler处理参数
    parameterHandler.setParameters((PreparedStatement) statement);
  }
```

- 流程分析

```
|--PreparedStatementHandler#parameterize：设置PreparedStatement的参数
    |--DefaultParameterHandler#setParameters：设置参数
        |--BaseTypeHandler#setParameter：
            |--xxxTypeHandler#setNonNullParameter:调用PreparedStatement的setxxx方法
```

结果集映射流程

- 找入口：DefaultResultSetHandler#handleResultSets方法

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

        final List<Object> multipleResults = new ArrayList<>();

        int resultSetCount = 0;
        // 获取第一个结果集，并放到ResultSet装饰类
        ResultSetWrapper rsw = getFirstResultSet(stmt);

        List<ResultMap> resultMaps = mappedStatement.getResultMaps();
```

```
        int resultMapCount = resultMaps.size();
        validateResultMapsCount(rsw, resultMapCount);
        while (rsw != null && resultMapCount > resultSetCount) {
            ResultMap resultMap = resultMaps.get(resultSetCount);
            // 处理结果集
            handleResultSet(rsw, resultMap, multipleResults, null);
            rsw = getNextResultSet(stmt);
            cleanUpAfterHandlingResultSet();
            resultSetCount++;
        }

        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++;
            }
        }

        return collapseSingleResultList(multipleResults);
    }
```

```
|--DefaultResultSetHandler#handleResultSets
    |--DefaultResultSetHandler#handleResultSet
        |--DefaultResultSetHandler#handleRowValues
            |--DefaultResultSetHandler#handleRowValuesForSimpleResultMap
                |--DefaultResultSetHandler#getRowValue
                    |--DefaultResultSetHandler#createResultObject：创建映射结果对象
                    |--DefaultResultSetHandler#applyAutomaticMappings
                    |--DefaultResultSetHandler#applyPropertyMappings
```

相关类和接口：

```
    |--DefaultResultSetHandler
```

手写框架篇

JDBC代码

```java
public class JdbcDemo {

    public static void main(String[] args) {
        Connection connection = null;
        PreparedStatement preparedStatement = null;
        ResultSet rs = 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
            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
            preparedStatement.setString(1, "王五");

            // 向数据库发出 sql 执行查询，查询出结果集
            rs = preparedStatement.executeQuery();

            // 遍历查询结果集
            while (rs.next()) {
                System.out.println(rs.getString("id")+" "+rs.getString("username"));
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            // 释放资源
            if (rs != null) {
                try {
                    rs.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) {
                    // TODO Auto-generated catch block e.printStackTrace();
                }
            }
        }
    }
}
```

框架设计

1. 编写两类配置文件（一类是配置数据源信息、一类是编写SQL语句的）
2. 通过Classloader去加载，返回InputStream对象
3. 加载配置文件
    1. 加载全局配置文件-----XMLConfigBuilder
        1. 通过Sax解析，读取InputStream对象，形成Document对象
        2. 通过dom4j + xpath完成dom处理，最终将数据封装到Configuration对象中
            1. 解析environments标签
                1. 创建DataSource对象，封装到Configuration中
            2. 解析mappers标签
                1. 将映射文件中的内容（MappedStatement对象）封装到Configuration对象中
    2. 加载映射文件---XMLMapperBuilder
        1. select标签
            1. id属性
            2. parameterType属性（要转成Class类对象）
            3. resultType属性（要转成Class类对象）
            4. SQL语句（将sql语句中的#{}进行处理）
                1. 处理器#{}信息-----替换为?，并且将对应的参数信息，封装到一个ParameterMapping对象中，便于执行阶段，为该?(占位符)，设置参数

4. sqlsession执行流程
    1. 先要指定sqlsession接口的api方法（主要是CRUD方法）
    2. sqlsession的创建
        1. 需要先创建SqlSessionFactory，SQLSessionFactory需要通过SQLSessionFactoryBuilder类去读取Configuration对象后，去创建SQLSessionFactory对象。
    3. sqlsession执行
        1. 先读取MappedStatement对象（存有要执行的SQL信息）
        2. 通过Configuration对象，获取DataSource对象，然后创建Connection
        3. 读取MappedStatement对象中的statementType（如果为空，则默认是Prepared）

1. 获取SQL语句---通过MappedStatement对象中获取了SqlSource对象，然后获取getBoundSql
2. 通过Connection去按照statementType指定的类型去创建Statement对象
3. 预编译SQL语句

4. 参数设置
    1. 遍历parameterMapping集合(使用 i下标进行遍历)
        1. 取出每个ParameterMapping对象中的参数名称、参数类型
        2. 通过参数类型判断应该去调用preparedStatement.set什么类型的方法
            1. SqlSession接口传入的参数只有一个，如果该参数类型是简单类型的话，直接将参数赋值即可。如果该参数是对象类型的话，那么需要通过反射去对象中 获取指定的属性值。
5. 执行statement，并返回ResultSet
6. 映射ResultSet（column和property属性默认是一致的）
    1. ResultSet中的每一行，可以获取该行所有的列的名称和列的类型还有列的值
    2. 通过MappedStatement获取ResultType的值，通过反射创建该对象
    3. 如何通过反射给一个对象赋值（要考虑类型判断）
        1. 直接通过暴力方式，给私有属性set值
        2. 一种是通过反射获取指定方法名称的方法对象，通过反射调用赋值
        3. 可以使用内省技术，获取一个对象的指定属性的set方法，然后赋值

扩展点

- 构建者模式
- 适配模式
- 抽象模板方法模式
- 工厂模式
- 代理模式
- 委托模式

总结

作业

作业完成时间

周一上午10点之前

作业产出物

每个同学手写框架项目

作业检查方式

会找出10位同学的项目作业，在群里让助教老师点评一二

必须完成的作业

目前只关注功能实现，先不关注设计思想

- 手写mybatis框架中的参数映射流程
- 手写mybatis框架中的结果集映射流程

附加作业

- 在我手写的框架基础之上，【添加StatementHandler及它的子类】，去创建Statement、完成Statement的参数设置、执行Statement
- 在我手写的框架基础知识，【添加TypeHandler，及它的子类】，去适配不同的类型，然后执行preparedStatement.setXXX方法进行赋值。

下节预告

spring架构分析和源码分析