

课堂主题

Mybatis架构分析与源码专题

课堂目标

- 搞明白mybatis框架的架构是啥样的？
- 搞明白mybatis执行流程底层是啥样的？
- 搞明白mybatis的参数是如何类型转换以及设置参数的？
- 搞明白mybatis的结果集是如何类型转换以及结果集映射的？
- 学会手写框架

知识要点

课堂主题

课堂目标

知识要点

架构原理篇

架构图

接口层

数据处理层

支撑层

架构流程图

调用流程图

SqlSession

Executor

StatementHandler

ParameterHandler

ResultSetHandler

TypeHandler

MappedStatement

SqlSource

Configuration

mapper代理执行原理

手写框架篇

JDBC代码

框架设计

源码分析篇

源码阅读方法

源码阅读目的

接口和对象介绍

SqlSessionFactoryBuilder

XMLConfigBuilder

XMLMapperBuilder

Configuration

SqlSource接口

SQLSessionFactory接口

SqlSession接口

Executor接口

StatementHandler接口

ParameterHandler接口

ResultSetHandler接口

源码阅读

加载全局配置文件流程

加载映射文件流程

SqlSource创建流程

获取Mapper代理对象流程

SqlSession执行主流程（自己看）

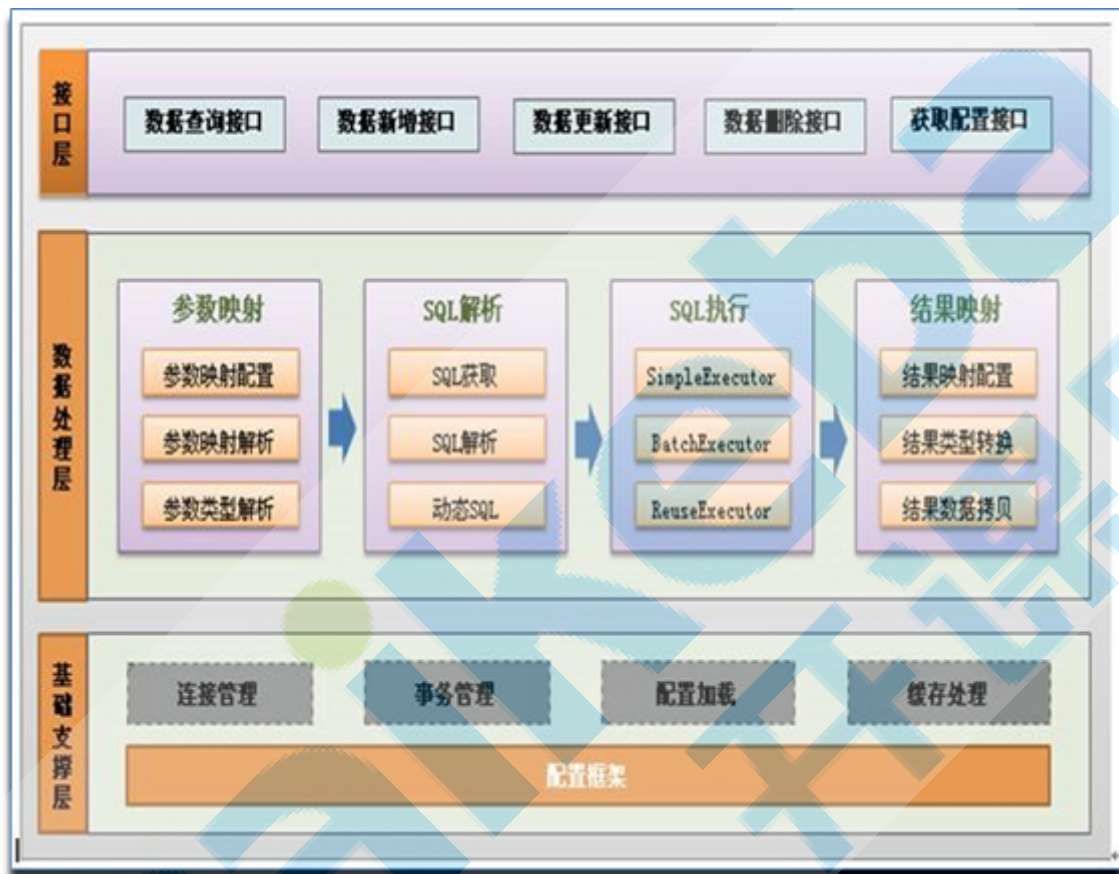
BoundSql获取流程

参数映射流程（自己看）

结果集映射流程（自己看）

架构原理篇

架构图





接口层

接口层是MyBatis提供给开发人员的一套API.主要使用`SqlSession`接口.通过`SqlSession`接口和`Mapper`接口.开发人员,可以通知MyBatis框架调用那一条SQL命令以及SQL命令关联参数.

`SqlSession`接口使用方式

```
public void start() throws IOException{
    InputStream is = Resources.getResourceAsStream("myBatis-config.xml");
    SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(is);
    session = factory.openSession();
    session.insert("InsertDept_Sql",dept);
}
```

`Mapper`接口使用方式

```
public void start() throws IOException{
    InputStream is = Resources.getResourceAsStream("myBatis-config.xml");
    SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(is);
    session = factory.openSession();
    session.insert("InsertDept_Sql",dept);
}
```

数据处理层

数据处理层是MyBatis框架内部的核心实现.来完成对映射文件的解析与数据处理:

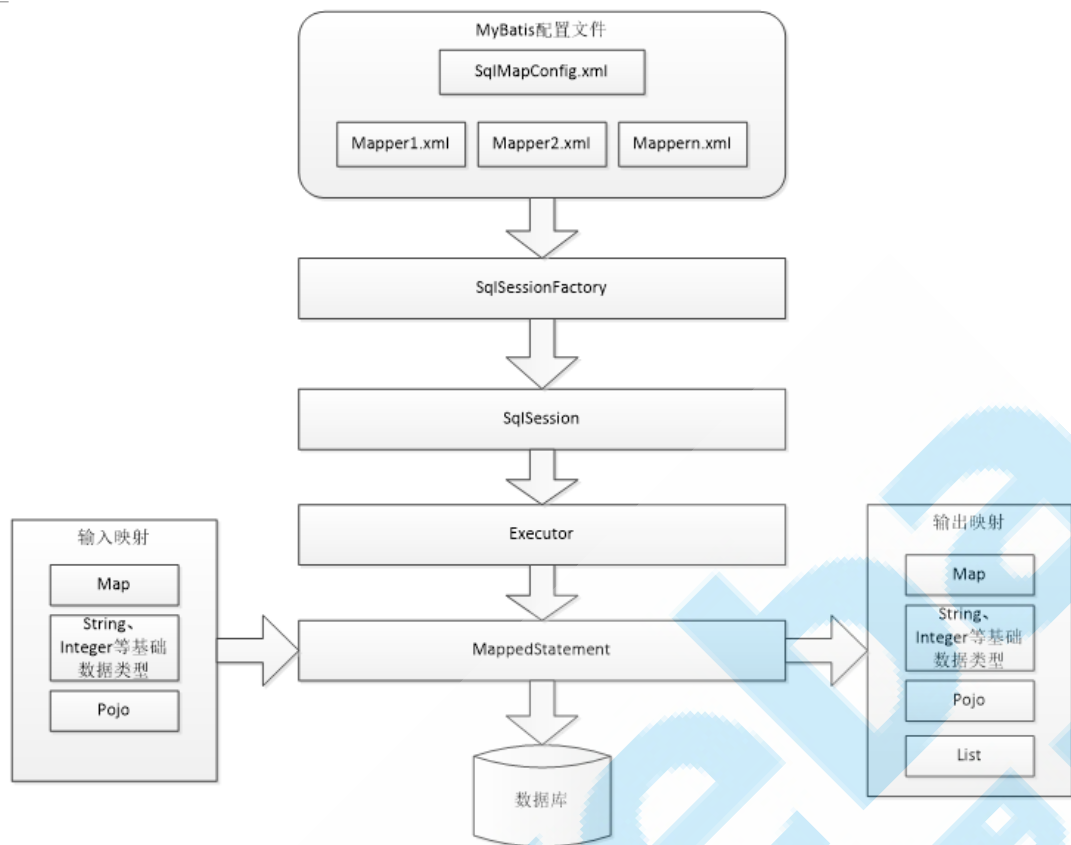
- (1) 参数解析与参数绑定
- (2) SQL解析
- (3) 结果集映射解析与结果集映射处理

支撑层

支撑层用来完成MyBatis与数据库基本连接方式以及SQL命令与配置文件对应.主要负责:

- (1) MyBatis与数据库连接方式管理
- (2) MyBatis对事务管理方式
- (3) 配置文件加载
- (4) MyBatis查询缓存管理

架构流程图



说明:

1. mybatis配置文件

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

2. SqlSessionFactory

通过mybatis环境等配置信息构造SqlSessionFactory, 即会话工厂。

3. sqlSession

通过会话工厂创建sqlSession即会话, 程序员通过sqlsession会话接口对数据库进行增删改查操作。

4. Executor执行器

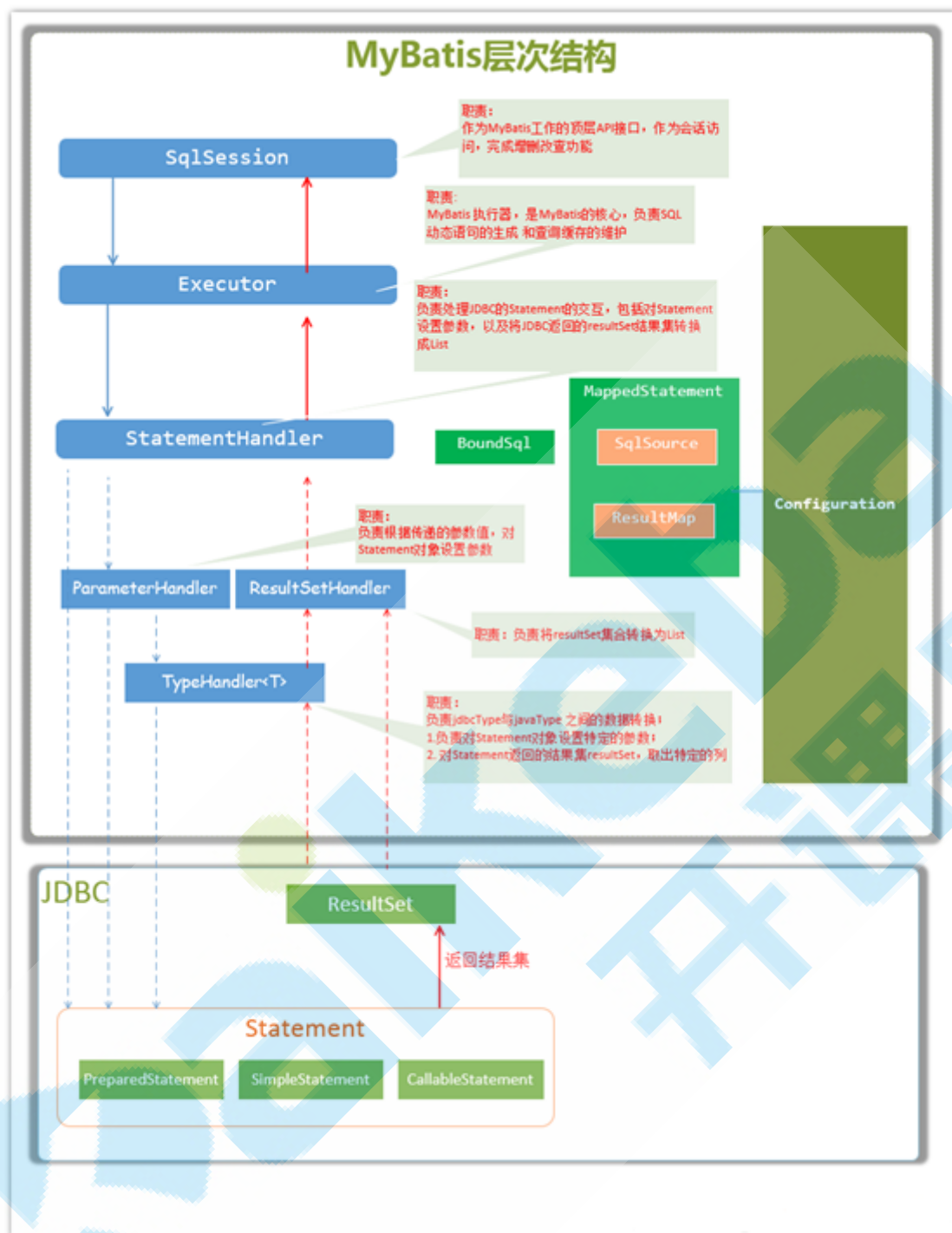
mybatis底层自定义了Executor执行器接口来具体操作数据库, Executor接口有两个实现, 一个是基本执行器 (默认)、一个是缓存执行器, sqlSession底层是通过executor接口操作数据库的。

5. Mapped Statement

它也是mybatis一个底层封装对象，它包装了mybatis配置信息及sql映射信息等。mapper.xml文件中一个**select\insert\update\delete**标签对应一个**Mapped Statement对象**，select\insert\update\delete标签的id即是Mapped statement的id。

- Mapped Statement对sql执行输入参数进行定义，包括HashMap、基本类型、pojo，Executor通过Mapped Statement在执行sql前将输入的java对象映射至sql中，输入参数映射就是jdbc编程中对preparedStatement设置参数。
- Mapped Statement对sql执行输出结果进行定义，包括HashMap、基本类型、pojo，Executor通过Mapped Statement在执行sql后将输出结果映射至java对象中，输出结果映射过程相当于jdbc编程中对结果的解析处理过程。

调用流程图



SqlSession

接收开发人员提供Statement Id 和参数,并返回操作结果

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

维护了一条<select|update|delete|insert>节点的封装

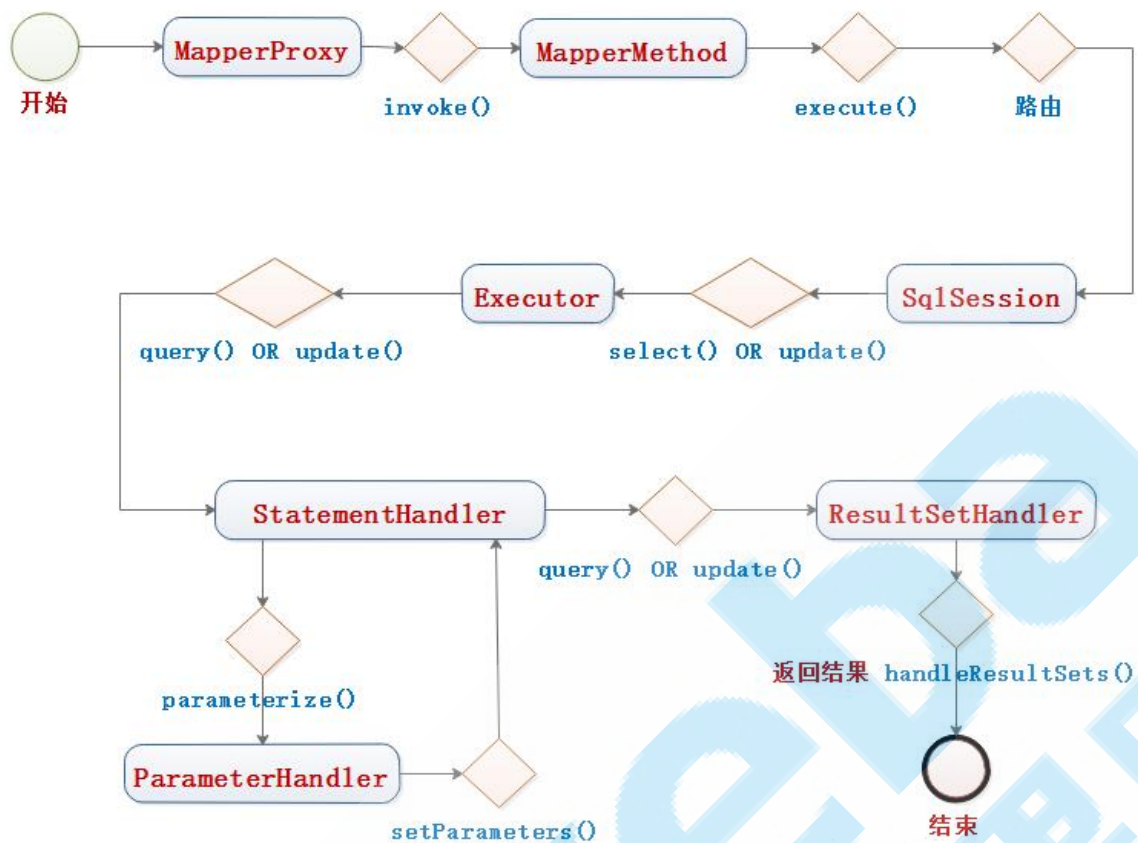
SqlSource

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

Configuration

MyBatis所有的配置信息都维持在Configuration对象之中。

mapper代理执行原理



手写框架篇

JDBC代码

```

1  public class JdbcDemo {
2
3      public static void main(String[] args) {
4          Connection connection = null;
5          PreparedStatement preparedStatement = null;
6          ResultSet rs = null;
7
8          try {
9              // 加载数据库驱动
10             Class.forName("com.mysql.jdbc.Driver");
11
12             // 通过驱动管理类获取数据库链接 connection = DriverManager
13             connection = DriverManager.getConnection(
14                 "jdbc:mysql://localhost:3306/mybatis?characterEncoding=utf-8", "root",
15                 "root");
16
17             // 定义sql语句 ?表示占位符
18         } catch (Exception e) {
19             e.printStackTrace();
20         }
21     }
22 }
  
```

```

16         String sql = "select * from user where username = ?";
17
18         // 获取预处理 statement
19         preparedStatement = connection.prepareStatement(sql);
20
21         // 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
22         preparedStatement.setString(1, "王五");
23
24         // 向数据库发出 sql 执行查询，查询出结果集
25         rs = preparedStatement.executeQuery();
26
27         // 遍历查询结果集
28         while (rs.next()) {
29             System.out.println(rs.getString("id")+"
    "+rs.getString("username"));
30         }
31     } catch (Exception e) {
32         e.printStackTrace();
33     } finally {
34         // 释放资源
35         if (rs != null) {
36             try {
37                 rs.close();
38             } catch (SQLException e) {
39                 e.printStackTrace();
40             }
41         }
42         if (preparedStatement != null) {
43             try {
44                 preparedStatement.close();
45             } catch (SQLException e) {
46                 e.printStackTrace();
47             }
48         }
49         if (connection != null) {
50             try {
51                 connection.close();
52             } catch (SQLException e) {
53                 // TODO Auto-generated catch block e.printStackTrace();
54             }
55         }
56     }
57 }
58 }

```

框架设计

1. 编写自定义的配置文件和映射文件。
2. 使用ClassLoader加载全局配置文件，返回InputStream对象
3. 配置文件加载
 - 全局配置文件加载，将XML信息存储到Configuration对象
 - 使用sax Reader去读取InputStream对象，创建Document对象
 - 使用dom4j+xpath语法去解析Document对象
 - 解析environments标签
 - 封装DataSource对象，放入Configuration对象中
 - 解析mappers标签
 - 映射文件加载，将XML信息封装到MappedStatement对象并放入Map集合中，key为statement的id，value为MappedStatement对象
 - 使用ClassLoader加载全局配置文件，返回InputStream对象
 - 使用sax Reader去读取InputStream对象，创建Document对象
 - 使用dom4j+xpath语法去解析Document对象
 - 解析根标签中的namespace属性和select标签
 - 解析id属性
 - 解析parameterType属性，并处理成Class类型
 - 解析resultType属性，并处理成Class类型
 - 解析statementType属性（便于执行时，是选择Statement还是preparedStatement）
 - 解析SQL语句（#{ }和\${ }）
 - 将#{ }替换为占位符？
 - 存储预编译SQL（已经将#{ }替换为？的SQL语句）
 - 创建List<ParameterMapping>集合，存储SQL参数信息（属性名称、参数类型）
 - 将\${ }替换成真正的参数
 - 使用OGNL表达式获取参数值
 - 替换\${ }的内容
4. sqlSession执行
 - 制定sqlsession接口和api方法
 - 创建SqlSession
 - 创建SqlSession需要先创建SqlSessionFactory，通过SqlSessionFactoryBuilder使用构建者模式来创建，此时需要Configuration对象去创建SqlSessionFactory
 - 执行sqlsession：参数有两个（statementId和参数对象）

- 根据statementId，去Configuration中的MappedStatement集合中查找对应的MappedStatement对象。
- 取出MappedStatement中的SQL信息
- 取出MappedStatement中的statementType，用来创建Statement对象
 - 取出MappedStatement中的Configuration对象，通过Configuration对象，获取DataSource对象，通过DataSource对象，创建Connection，通过Connection创建Statement对象。
- 设置参数
 - 执行preparedStatement.setInt(1,value);
 - 遍历List<ParameterMapping>集合(参数名称、属性类型、顺序)
 - 判断是否是集合类型、8种基本类型、String类型、引用类型
 - 基本类型的话，传过来的参数就是SQL的参数值
 - 引用类型的话，根据参数名称，获取引用类型对应的属性值
 - 调用setXXX方法赋值
- 执行Statement
 - executeQuery方法，获取ResultSet结果集
- 处理结果集
 - 遍历结果集ResultSet
 - 取出ResultSet中的所有列的名称和值和类型，存储到一个集合中
 - 取出MappedStatement中的resultTypeClass，反射进行实例化。
 - 遍历上面的集合，根据集合中的列的名称给属性赋值

源码分析篇

源码阅读方法

1. 找主线。
2. 找入口。
3. 记笔记。（类名#方法名（数据成员变量））
4. 参考其他人的源码阅读经验。

源码阅读目的

- 通过阅读源码，提升对设计模式的理解，提升编程能力
- 通过阅读源码，可以找到问题根源，用来解决问题
- 应付面试

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.  
            }  
        }  
    }  
}
```



```

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

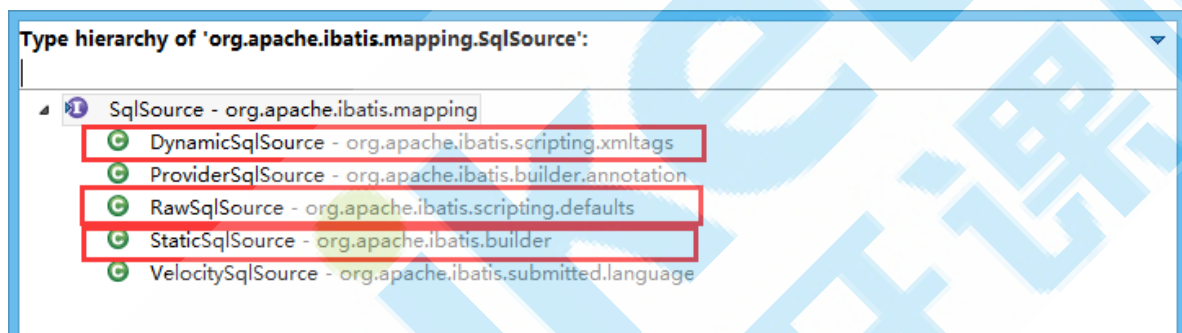
专门用来解析映射文件的解析器

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

```
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。

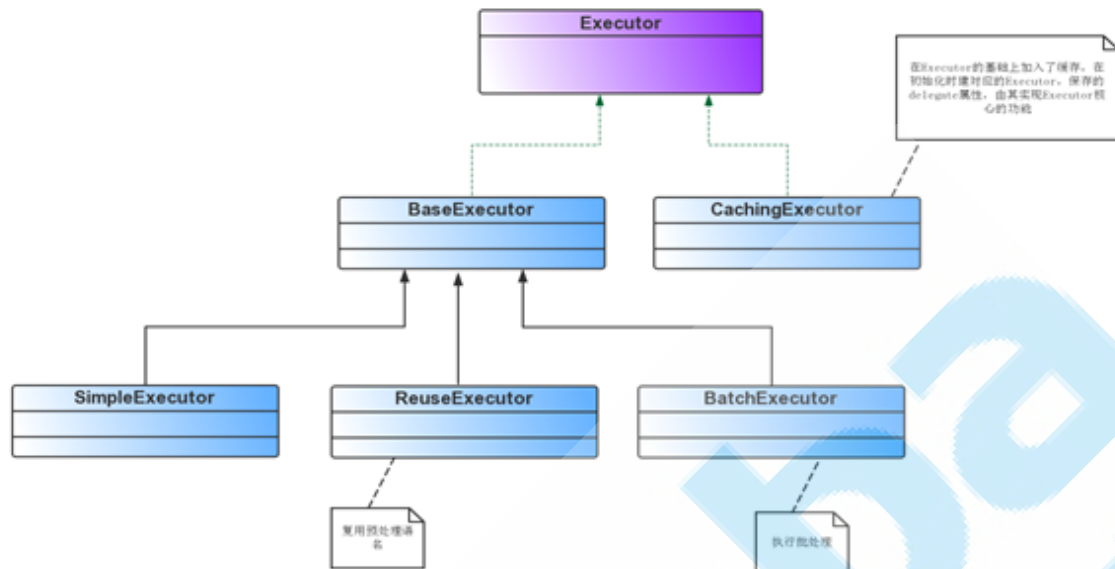
SQLSessionFactory接口

默认实现类是DefaultSQLSessionFactory类

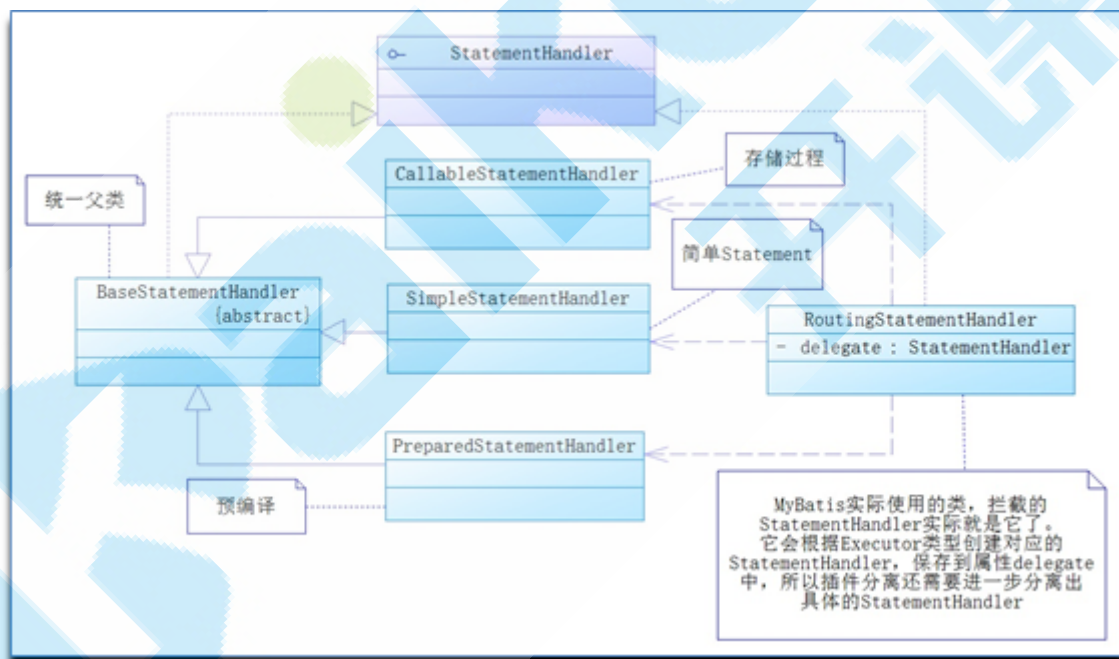
SqlSession接口

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

加载全局配置文件流程

- 找入口：SqlSessionFactoryBuilder#build方法

```
1 public SqlSessionFactory build(InputStream inputStream, String environment,
2   Properties properties) {
3     try {
4       // XMLConfigBuilder:用来解析XML配置文件
5       // 使用构建者模式
6       XMLConfigBuilder parser = new XMLConfigBuilder(inputStream,
7     environment, properties);
8       // parser.parse(): 使用XPath解析XML配置文件，将配置文件封装为Configuration对象
9       // 返回DefaultSqlSessionFactory对象，该对象拥有Configuration对象（封装配置文件信息）
10      return build(parser.parse());
11    } catch (Exception e) {
12      throw ExceptionFactory.wrapException("Error building SqlSession.",
13    e);
14    } finally {
15      ErrorContext.instance().reset();
16      try {
17        inputStream.close();
18      } catch (IOException e) {
19        // Intentionally ignore. Prefer previous error.
20      }
21    }
22  }
```

- 流程分析

```
1 SqlSessionFactoryBuilder#build : 用于构建SqlSessionFactory对象
2   |--XMLConfigBuilder#构造参数: 用来解析全局文件文件的解析器
3   |--XPathParser#构造参数: 用来使用XPath语法解析XML的解析器
4   |--XPathParser#createDocument(): 解析全局配置文件，封装为Document对象（封装一些子节点，使用XPath语法解析获取）
5   |--Configuration#构造参数: 创建Configuration对象，同时初始化内置类的别名
6   |--XMLConfigBuilder#parse(): 全局配置文件的解析器
7   |--XPathParser#evalNode(xpath语法): XPath解析器，专门用来通过Xpath语法解析XML返回XNode节点
8
9   |--XMLConfigBuilder#parseConfiguration(XNode): 从全局配置文件根节点开始解析，加载的信息设置到Configuration对象中
10  |--SqlSessionFactoryBuilder#build: 创建SqlSessionFactory接口的默认实现类DefaultSqlSessionFactory（Configuration对象）
11
```

- 总结:

```
1      1.SqlSessionFactoryBuilder创建SqlSessionFactory时，需要传入一个Configuration
      对象。
2      2.XMLConfigBuilder对象会去实例化Configuration。
3      3.XMLConfigBuilder对象会去初始化Configuration对象。
4      通过XPathParser去解析全局配置文件，形成Document对象
5      通过XPathParser去获取指定节点的XNode对象。
6      解析Xnode对象的信息，然后封装到Configuration对象中
```

- 相关类和接口:

```
1      |--SqlSessionFactoryBuilder
2      |--XMLConfigBuilder
3      |--XPathParser
4      |--Configuration
```

加载映射文件流程

- 找入口: XMLConfigBuilder#mapperElement方法

```
1      /**
2      * 解析<mappers>标签
3      * @param parent mappers标签对应的XNode对象
4      * @throws Exception
5      */
6      private void mapperElement(XNode parent) throws Exception {
7          if (parent != null) {
8              // 获取<mappers>标签的子标签
9              for (XNode child : parent.getChildren()) {
10                 // <package>子标签
11                 if ("package".equals(child.getName())) {
12                     // 获取mapper接口和mapper映射文件对应的package包名
13                     String mapperPackage = child.getStringAttribute("name");
14                     // 将包下所有的mapper接口以及它的代理对象存储到一个Map集合中，key为mapper
15                     // 接口类型，value为代理对象工厂
16                     configuration.addMappers(mapperPackage);
17                 } else { // <mapper>子标签
18                     // 获取<mapper>子标签的resource属性
19                     String resource = child.getStringAttribute("resource");
20                     // 获取<mapper>子标签的url属性
21                     String url = child.getStringAttribute("url");
22                     // 获取<mapper>子标签的class属性
```



```

22         String mapperClass = child.getStringAttribute("class");
23         // 按照 resource ---> url ---> class的优先级去解析取<mapper>子标签，它们
    们是互斥的
24         if (resource != null && url == null && mapperClass == null) {
25             ErrorContext.instance().resource(resource);
26             InputStream inputStream =
Resources.getResourceAsStream(resource);
27             // 专门用来解析mapper映射文件
28             XMLMapperBuilder mapperParser = new
XMLMapperBuilder(inputStream, configuration, resource,
configuration.getSqlFragments());
29             // 通过XMLMapperBuilder解析mapper映射文件
30             mapperParser.parse();
31         } else if (resource == null && url != null && mapperClass ==
null) {
32             ErrorContext.instance().resource(url);
33             InputStream inputStream = Resources.getUrlAsStream(url);
34             XMLMapperBuilder mapperParser = new
XMLMapperBuilder(inputStream, configuration, url,
configuration.getSqlFragments());
35             // 通过XMLMapperBuilder解析mapper映射文件
36             mapperParser.parse();
37         } else if (resource == null && url == null && mapperClass !=
null) {
38             Class<?> mapperInterface = Resources.forName(mapperClass);
39             // 将指定mapper接口以及它的代理对象存储到一个Map集合中，key为mapper接口
    类型，value为代理对象工厂
40             configuration.addMapper(mapperInterface);
41         } else {
42             throw new BuilderException("A mapper element may only specify a
url, resource or class, but not more than one.");
43         }
44     }
45 }
46 }
47 }

```

- 流程分析

```

1 XMLConfigBuilder#mapperElement:解析全局配置文件中的<mappers>标签
2     |--XMLMapperBuilder#构造方法: 专门用来解析映射文件的
3     |--XPathParser#构造方法:
4         |--XPathParser#createDocument(): 创建Mapper映射文件对应的Document对
    象
5         |--MapperBuilderAssistant#构造方法: 用于构建MappedStatement对象的
6     |--XMLMapperBuilder#parse():
7         |--XMLMapperBuilder#configurationElement: 专门用来解析mapper映射文件

```



```

8      |--XMLMapperBuilder#buildStatementFromContext: 用来创建
MappedStatement对象的
9      |--XMLMapperBuilder#buildStatementFromContext
10     |--XMLStatementBuilder#构造方法: 专门用来解析
MappedStatement
11     |--XMLStatementBuilder#parseStatementNode:
12     |--MapperBuilderAssistant#addMappedStatement:创建
MappedStatement对象
13     |--MappedStatement.Builder#构造方法
14     |--MappedStatement#build方法: 创建MappedStatement
对象, 并存储到Configuration对象中
15

```

- 相关类和接口:

```

1      |--XMLConfigBuilder
2      |--XMLMapperBuilder
3      |--XPathParser
4      |--MapperBuilderAssistant
5      |--XMLStatementBuilder
6      |--MappedStatement

```

SqlSource创建流程

- 找入口: XMLLanguageDriver#createSqlSource

```

1      @Override
2      public SqlSource createSqlSource(Configuration configuration, XNode
script, Class<?> parameterType) {
3          // 初始化了动态SQL标签处理器
4          XMLScriptBuilder builder = new XMLScriptBuilder(configuration,
script, parameterType);
5          // 解析动态SQL
6          return builder.parseScriptNode();
7      }

```

- SqlSource创建流程分析

```

1  |--XMLLanguageDriver#createSqlSource: 创建SqlSource
2      |--XMLScriptBuilder#构造方法: 初始化动态SQL中的节点处理器集合
3      |--XMLScriptBuilder#parseScriptNode:
4          |--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
5          |--DynamicSqlSource#构造方法: 如果SQL中包含${}和动态SQL语句, 则将SqlNode封装到DynamicSqlSource
6          |--RawSqlSource#构造方法: 如果SQL中包含#{}, 则将SqlNode封装到RawSqlSource
中
7              |--ParameterMappingTokenHandler#构造方法
8              |--GenericTokenParser#构造方法: 指定待分析的openToken和closeToken,
并指定处理器
9              |--GenericTokenParser#parse: 解析SQL语句, 处理openToken和
closeToken中的内容
10         |--ParameterMappingTokenHandler#handleToken: 处理token (#{
}/${})
11         |--ParameterMappingTokenHandler#buildParameterMapping:
创建ParameterMapping对象
12         |--StaticSqlSource#构造方法: 将解析之后的SQL信息, 封装到
StaticSqlSource
13

```

- 相关类和接口:

```

1  |--XMLLanguageDriver
2  |--XMLScriptBuilder
3  |--SqlSource
4  |--SqlSourceBuilder

```

获取Mapper代理对象流程

- 找入口: DefaultSqlSession#getMapper

```

1      @Override
2      public <T> T getMapper(Class<T> type) {
3          // 从Configuration对象中, 根据Mapper接口, 获取Mapper代理对象
4          return configuration.<T>getMapper(type, this);
5      }

```

- 流程分析

```

1 | --DefaultSqlSession#getMapper: 获取Mapper代理对象
2 |   |--Configuration#getMapper: 获取Mapper代理对象
3 |     |--MapperRegistry#getMapper: 通过代理对象工厂, 获取代理对象
4 |       |--MapperProxyFactory#newInstance: 调用JDK的动态代理方式, 创建Mapper
   代理
5

```

SqlSession执行主流程（自己看）

- 找入口: DefaultSqlSession#selectList()

```

1 public <E> List<E> selectList(String statement, Object parameter, RowBounds
   rowBounds) {
2     try {
3         // 根据传入的statementId, 获取MappedStatement对象
4         MappedStatement ms =
configuration.getMappedStatement(statement);
5         // 调用执行器的查询方法
6         // RowBounds是用来逻辑分页
7         // wrapCollection(parameter) 是用来装饰集合或者数组参数
8         return executor.query(ms, wrapCollection(parameter), rowBounds,
Executor.NO_RESULT_HANDLER);
9     } catch (Exception e) {
10        throw ExceptionFactory.wrapException("Error querying database.
Cause: " + e, e);
11    } finally {
12        ErrorContext.instance().reset();
13    }
14 }

```

- 流程分析

```

1 | --DefaultSqlSession#selectList
2 |   |--CachingExecutor#query
3 |     |--BaseExecutor#query
4 |       |--BaseExecutor#queryFromDatabase
5 |         |--SimpleExecutor#doQuery
6 |           |--Configuration#newStatementHandler: 创建
StatementHandler, 用来执行MappedStatement对象
7 |             |--RoutingStatementHandler#构造方法: 根据路由规则, 设置
不同的StatementHandler
8 |               |--SimpleExecutor#prepareStatement: 主要是设置
PreparedStatement的参数

```

```

9           |--SimpleExecutor#getConnection: 获取数据库连接
10          |--PreparedStatementHandler#prepare: 创建
PreparedStatement对象
11          |--PreparedStatementHandler#parameterize: 设置
PreparedStatement的参数
12          |--PreparedStatementHandler#query: 主要是用来执行SQL语句,
及处理结果集
13          |--PreparedStatement#execute: 调用JDBC的api执行
Statement
14          |--DefaultResultSetHandler#handleResultSets:处理
结果集

```

- 相关类和接口:

```

1      |--DefaultSqlSession
2      |--Executor
3          |--CachingExecutor
4          |--BaseExecutor
5          |--SimpleExecutor
6      |--StatementHandler
7          |--RoutingStatementHandler
8          |--PreparedStatementHandler
9      |--ResultSetHandler
10         |--DefaultResultSetHandler

```

BoundSql获取流程

- 找入口: MappedStatement#getBoundSql方法

```

1  public BoundSql getBoundSql(Object parameterObject) {
2      // 调用SqlSource获取BoundSql
3      BoundSql boundSql = sqlSource.getBoundSql(parameterObject);
4      List<ParameterMapping> parameterMappings =
boundSql.getParameterMappings();
5      if (parameterMappings == null || parameterMappings.isEmpty()) {
6          boundSql = new BoundSql(configuration, boundSql.getSql(),
parameterMap.getParameterMappings(), parameterObject);
7      }
8
9      // check for nested result maps in parameter mappings (issue #30)
10     for (ParameterMapping pm : boundSql.getParameterMappings()) {
11         String rmId = pm.getResultMapId();
12         if (rmId != null) {
13             ResultMap rm = configuration.getResultMap(rmId);
14             if (rm != null) {
15                 hasNestedResultMaps |= rm.hasNestedResultMaps();

```

```

16         }
17     }
18 }
19
20     return boundSql;
21 }

```

- 流程分析

```

1  |--DynamicSqlSource#getBoundSql
2      |--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
3          |--ParameterMappingTokenHandler#构造方法
4          |--GenericTokenParser#构造方法: 指定待分析的openToken和closeToken,并指定
   处理器
5          |--GenericTokenParser#parse: 解析SQL语句,处理openToken和closeToken中的
   内容
6              |--ParameterMappingTokenHandler#handleToken: 处理token (#{}/${})
7              |--ParameterMappingTokenHandler#buildParameterMapping: 创建
   ParameterMapping对象
8          |--StaticSqlSource#构造方法: 将解析之后的SQL信息,封装到StaticSqlSource
9
10 |--RawSqlSource#getBoundSql
11     |--StaticSqlSource#getBoundSql
12         |--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
13

```

参数映射流程（自己看）

- 找入口: PreparedStatementHandler#parameterize方法

```

1     public void parameterize(Statement statement) throws SQLException {
2         // 通过ParameterHandler处理参数
3         parameterHandler.setParameters((PreparedStatement) statement);
4     }

```

- 流程分析

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

结果集映射流程（自己看）

- 找入口：DefaultResultSetHandler#handleResultSets方法

```
1 public List<Object> handleResultSets(Statement stmt) throws SQLException {
2     ErrorContext.instance().activity("handling
  results").object(mappedStatement.getId());
3
4     final List<Object> multipleResults = new ArrayList<>();
5
6     int resultSetCount = 0;
7     // 获取第一个结果集，并放到ResultSet装饰类
8     ResultSetWrapper rsw = getFirstResultSet(stmt);
9
10    List<ResultMap> resultMaps = mappedStatement.getResultMaps();
11    int resultMapCount = resultMaps.size();
12    validateResultMapsCount(rsw, resultMapCount);
13    while (rsw != null && resultMapCount > resultSetCount) {
14        ResultMap resultMap = resultMaps.get(resultSetCount);
15        // 处理结果集
16        handleResultSet(rsw, resultMap, multipleResults, null);
17        rsw = getNextResultSet(stmt);
18        cleanUpAfterHandlingResultSet();
19        resultSetCount++;
20    }
21
22    String[] resultSets = mappedStatement.getResultSets();
23    if (resultSets != null) {
24        while (rsw != null && resultSetCount < resultSets.length) {
25            ResultMapping parentMapping =
  nextResultMaps.get(resultSets[resultSetCount]);
26            if (parentMapping != null) {
27                String nestedResultMapId =
  parentMapping.getNestedResultMapId();
28                ResultMap resultMap =
  configuration.getResultMap(nestedResultMapId);
29                handleResultSet(rsw, resultMap, null, parentMapping);
30            }

```



```

31         rsw = getNextResultSet(stmt);
32         cleanUpAfterHandlingResultSet();
33         resultSetCount++;
34     }
35 }
36
37     return collapseSingleResultList(multipleResults);
38 }

```

```

1  |--DefaultResultSetHandler#handleResultSets
2      |--DefaultResultSetHandler#handleResultSet
3          |--DefaultResultSetHandler#handleRowValues
4              |--DefaultResultSetHandler#handleRowValuesForSimpleResultMap
5                  |--DefaultResultSetHandler#getRowValue
6                      |--DefaultResultSetHandler#createResultObject: 创建映射结果
对象
7                      |--DefaultResultSetHandler#applyAutomaticMappings
8                      |--DefaultResultSetHandler#applyPropertyMappings

```

相关类和接口：

```

1  |--DefaultResultSetHandler

```