# 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 {
```

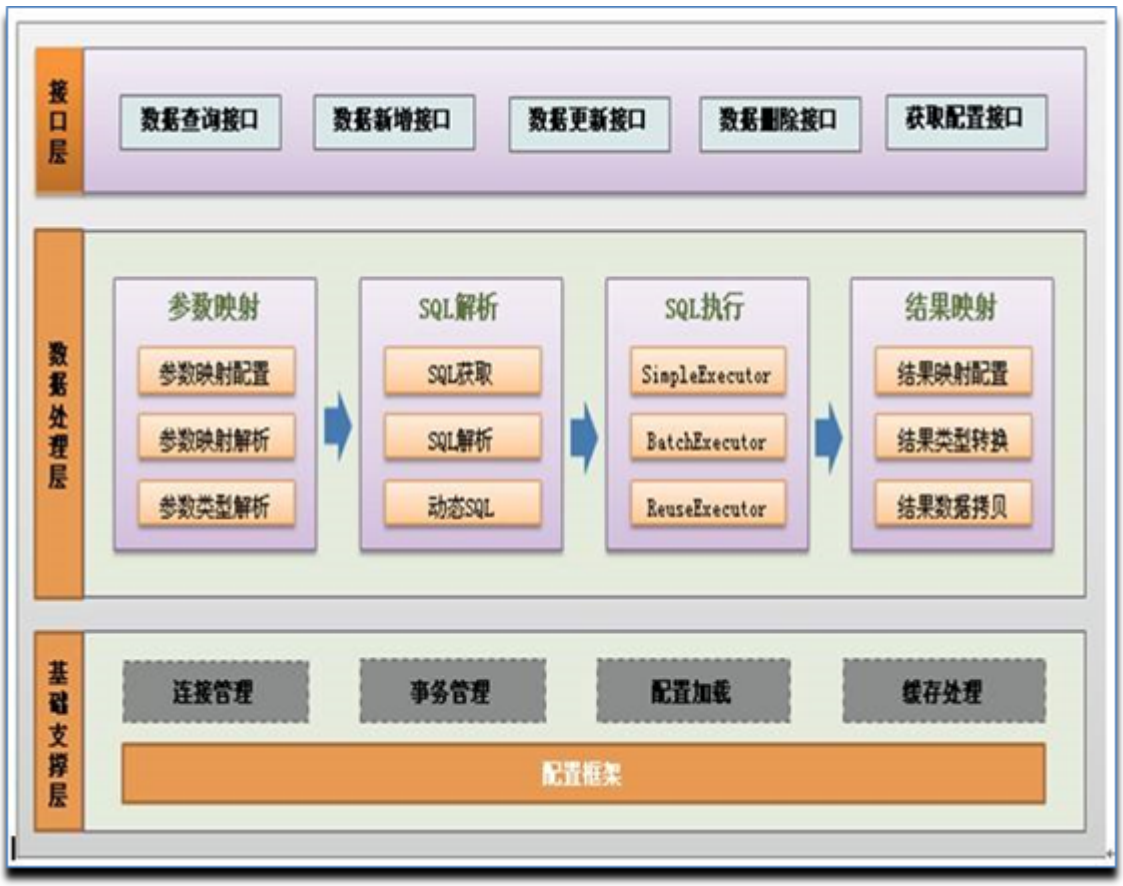
```
                    preparedStatement.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
            if (connection != null) {
                try {
                    connection.close();
                } catch (SQLException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

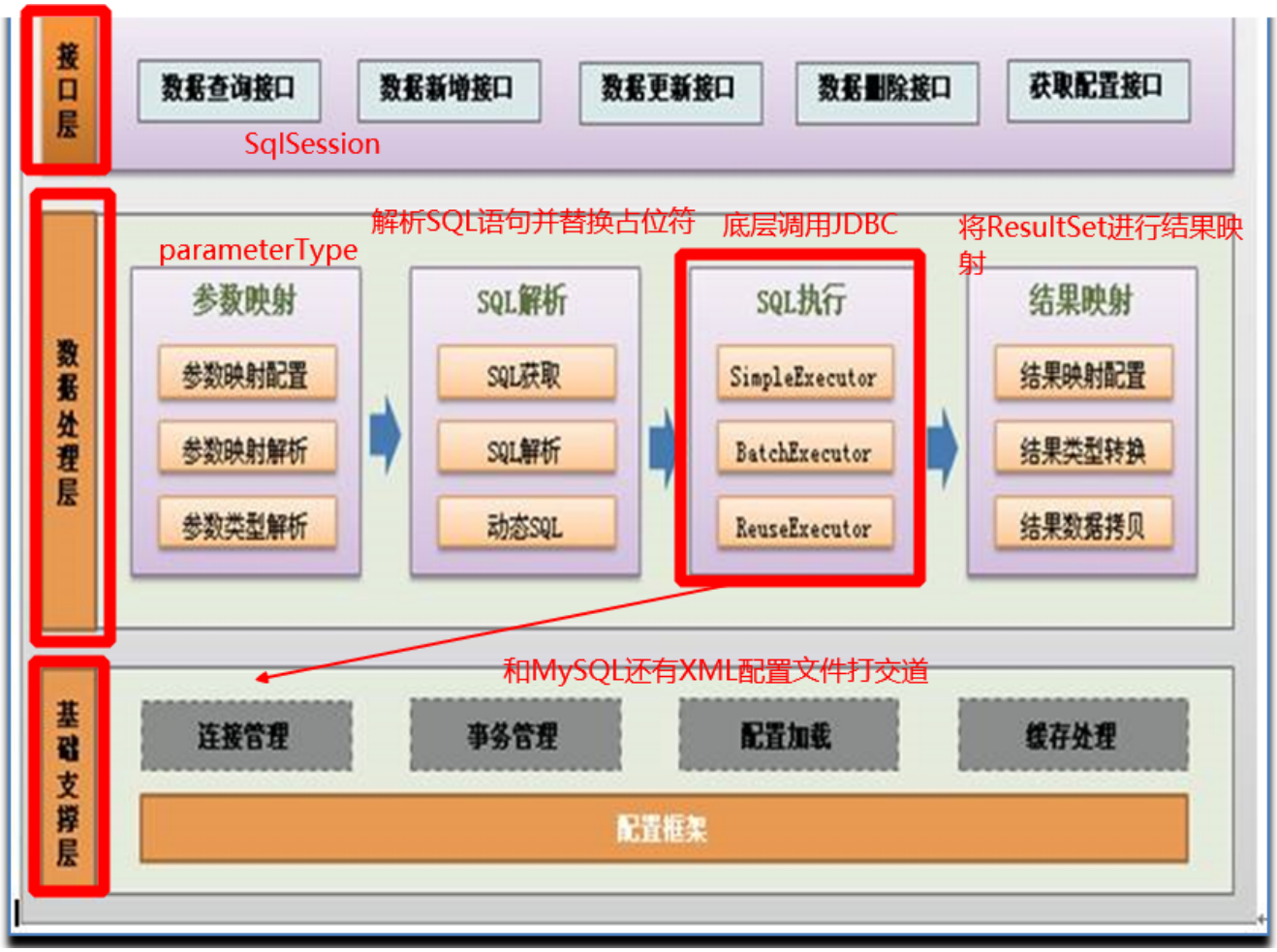
## 框架设计

参考txt格式的笔记

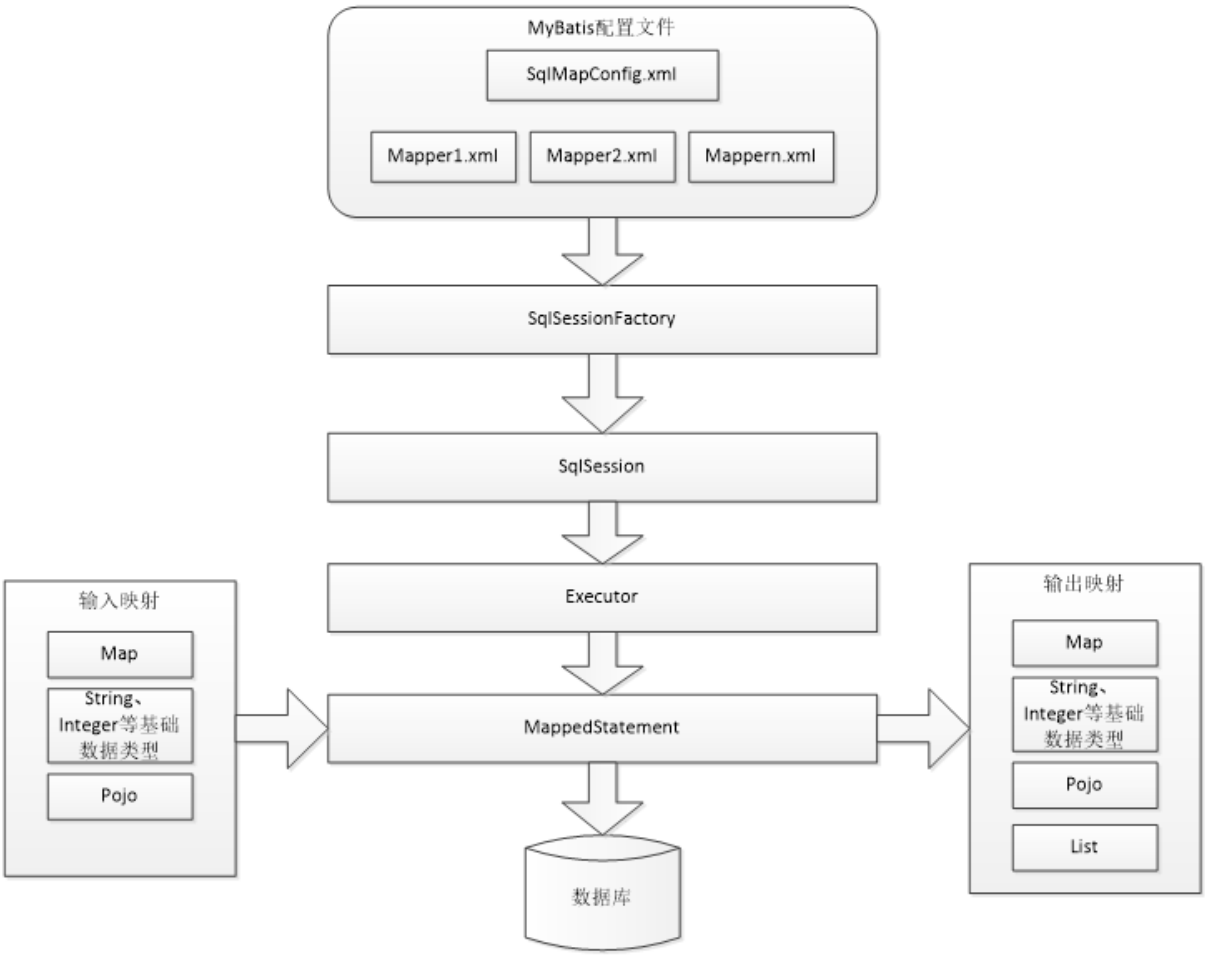
# 架构原理篇

## 架构图




接口层
数据查询接口
数据新增接口
数据更新接口
数据删除接口
获取配置接口
SqlSession
数据处理层
parameterType
解析SQL语句并替换占位符
底层调用JDBC
将ResultSet进行结果映射
参数映射
参数映射配置
参数映射解析
参数类型解析
SQL解析
SQL获取
SQL解析
动态SQL
SQL执行
SimpleExecutor
BatchExecutor
ReuseExecutor
结果映射
结果映射配置
结果类型转换
结果数据拷贝
和MySQL还有XML配置文件打交道
基础支撑层
连接管理
事务管理
配置加载
缓存处理
配置框架

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

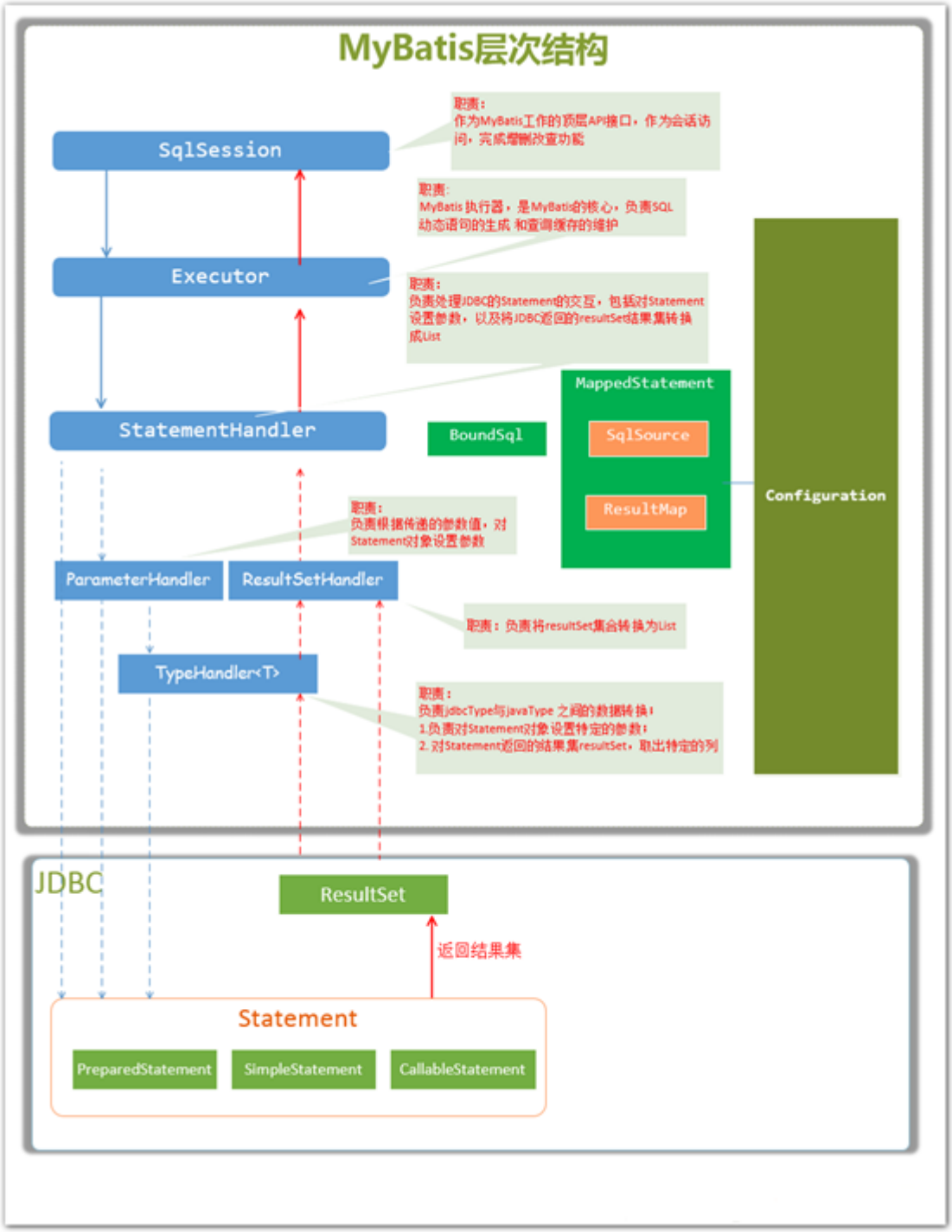
   它也是mybatis一个底层封装对象，它包装了mybatis配置信息及sql映射信息等。mapper.xml文件中一个**select\insert\update\delete标签对应一个Mapped Statement对象**，select\insert\update\delete标签的id即是Mapped statement的id。

- 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




## 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

### 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

# 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

```java
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

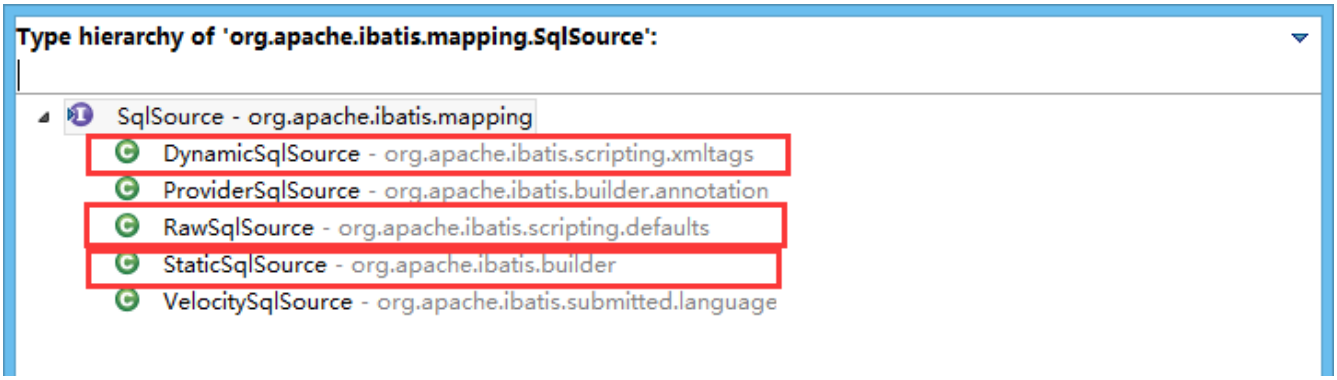
专门用来解析映射文件的解析器

## 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

```
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。

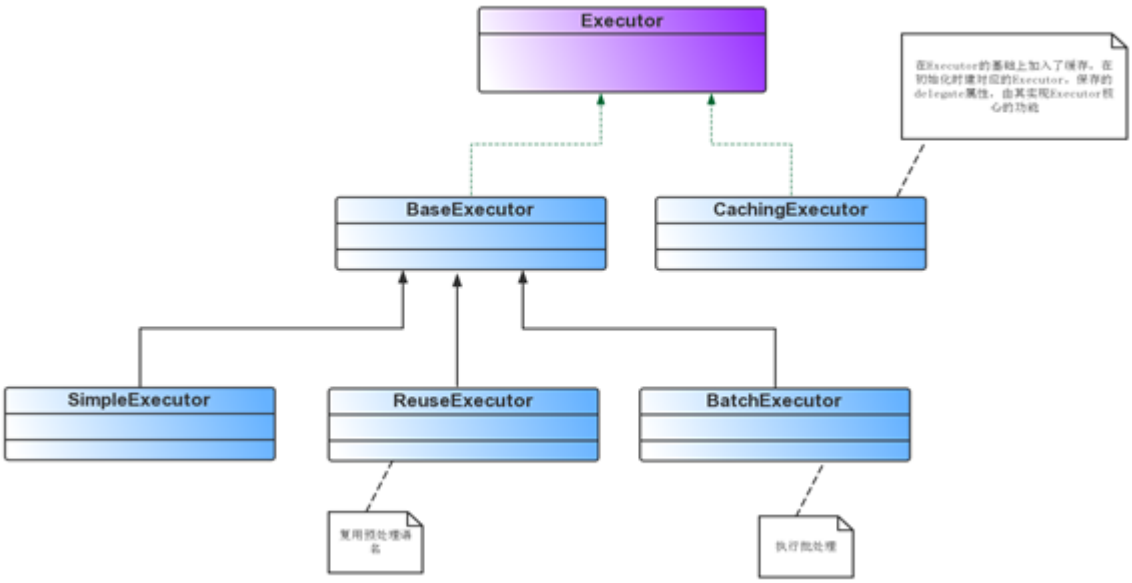
## SQLSessionFactory接口

默认实现类是DefaultSQLSessionFactory类

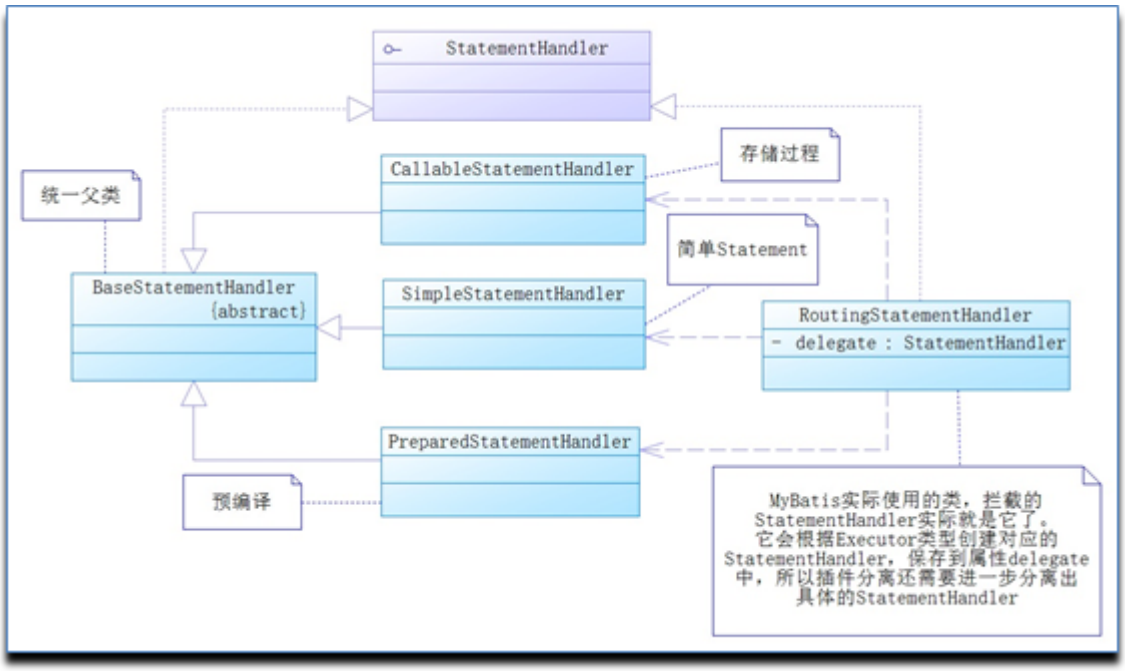
## SqlSession接口

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

```
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.
      }
    }
  }
```

- 流程分析

```
SqlSessionFactoryBuilder#build  ：用于构建SqlSessionFactory对象
    |--XMLConfigBuilder#构造参数：用来解析全局文件文件的解析器
        |--XPathParser#构造参数：用来使用XPath语法解析XML的解析器
            |--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）
        |--Configuration#构造参数：创建Configuration对象，同时初始化内置类的别名
    |--XMLConfigBuilder#parse():全局配置文件的解析器
        |--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

        |--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
    |--SqlSessionFactoryBuilder#build：创建SqlSessionFactory接口的默认实现类DefaultSqlSessionFactory（Configuration对象）

```

- 总结：

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

- 相关类和接口：

```
|--SqlSessionFactoryBuilder
|--XMLConfigBuilder
|--XPathParser
|--Configuration
```

## 加载映射文件流程

- 找入口：XMLConfigBuilder#mapperElement方法

```java
/**
  * 解析<mappers>标签
  * @param parent  mappers标签对应的XNode对象
  * @throws Exception
  */
 private void mapperElement(XNode parent) throws Exception {
   if (parent != null) {
     // 获取<mappers>标签的子标签
     for (XNode child : parent.getChildren()) {
       // <package>子标签
       if ("package".equals(child.getName())) {
         // 获取mapper接口和mapper映射文件对应的package包名
         String mapperPackage = child.getStringAttribute("name");
         // 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接口类型，value为代理对象工厂
         configuration.addMappers(mapperPackage);
       } else {// <mapper>子标签
         // 获取<mapper>子标签的resource属性
         String resource = child.getStringAttribute("resource");
         // 获取<mapper>子标签的url属性
         String url = child.getStringAttribute("url");
         // 获取<mapper>子标签的class属性
         String mapperClass = child.getStringAttribute("class");
         // 按照 resource ---> url ---> class的优先级去解析取<mapper>子标签，它们是互斥的
         if (resource != null && url == null && mapperClass == null) {
           ErrorContext.instance().resource(resource);
           InputStream inputStream = Resources.getResourceAsStream(resource);
           // 专门用来解析mapper映射文件
           XMLMapperBuilder mapperParser = new XMLMapperBuilder(inputStream, configuration, resource, configuration.getSqlFragments());
           // 通过XMLMapperBuilder解析mapper映射文件
           mapperParser.parse();
```

```
          } else if (resource == null && url != null && mapperClass == null) {
            ErrorContext.instance().resource(url);
            InputStream inputStream = Resources.getUrlAsStream(url);
            XMLMapperBuilder mapperParser = new XMLMapperBuilder(inputStream,
configuration, url, configuration.getSqlFragments());
            // 通过XMLMapperBuilder解析mapper映射文件
            mapperParser.parse();
          } else if (resource == null && url == null && mapperClass != null) {
            Class<?> mapperInterface = Resources.classForName(mapperClass);
            // 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
            configuration.addMapper(mapperInterface);
          } else {
            throw new BuilderException("A mapper element may only specify a url,
resource or class, but not more than one.");
          }
        }
      }
    }
  }
```

- 流程分析

```
XMLConfigBuilder#mapperElement:解析全局配置文件中的<mappers>标签
    |--XMLMapperBuilder#构造方法：专门用来解析映射文件的
        |--XPathParser#构造方法：
            |--XPathParser#createDocument()：创建Mapper映射文件对应的Document对象
            |--MapperBuilderAssistant#构造方法：用于构建MappedStatement对象的
    |--XMLMapperBuilder#parse()：
        |--XMLMapperBuilder#configurationElement：专门用来解析mapper映射文件
            |--XMLMapperBuilder#buildStatementFromContext：用来创建MappedStatement对象的
                |--XMLMapperBuilder#buildStatementFromContext
                    |--XMLStatementBuilder#构造方法：专门用来解析MappedStatement
                    |--XMLStatementBuilder#parseStatementNode:
                        |--MapperBuilderAssistant#addMappedStatement:创建
MappedStatement对象
                            |--MappedStatement.Builder#构造方法
                            |--MappedStatement#build方法：创建MappedStatement对象，并存储
到Configuration对象中

```

- 相关类和接口：

```
    |--XMLConfigBuilder
    |--XMLMapperBuilder
    |--XPathParser
    |--MapperBuilderAssistant
    |--XMLStatementBuilder
    |--MappedStatement
```

## 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（自己看）

- 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（自己看）

- 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

```