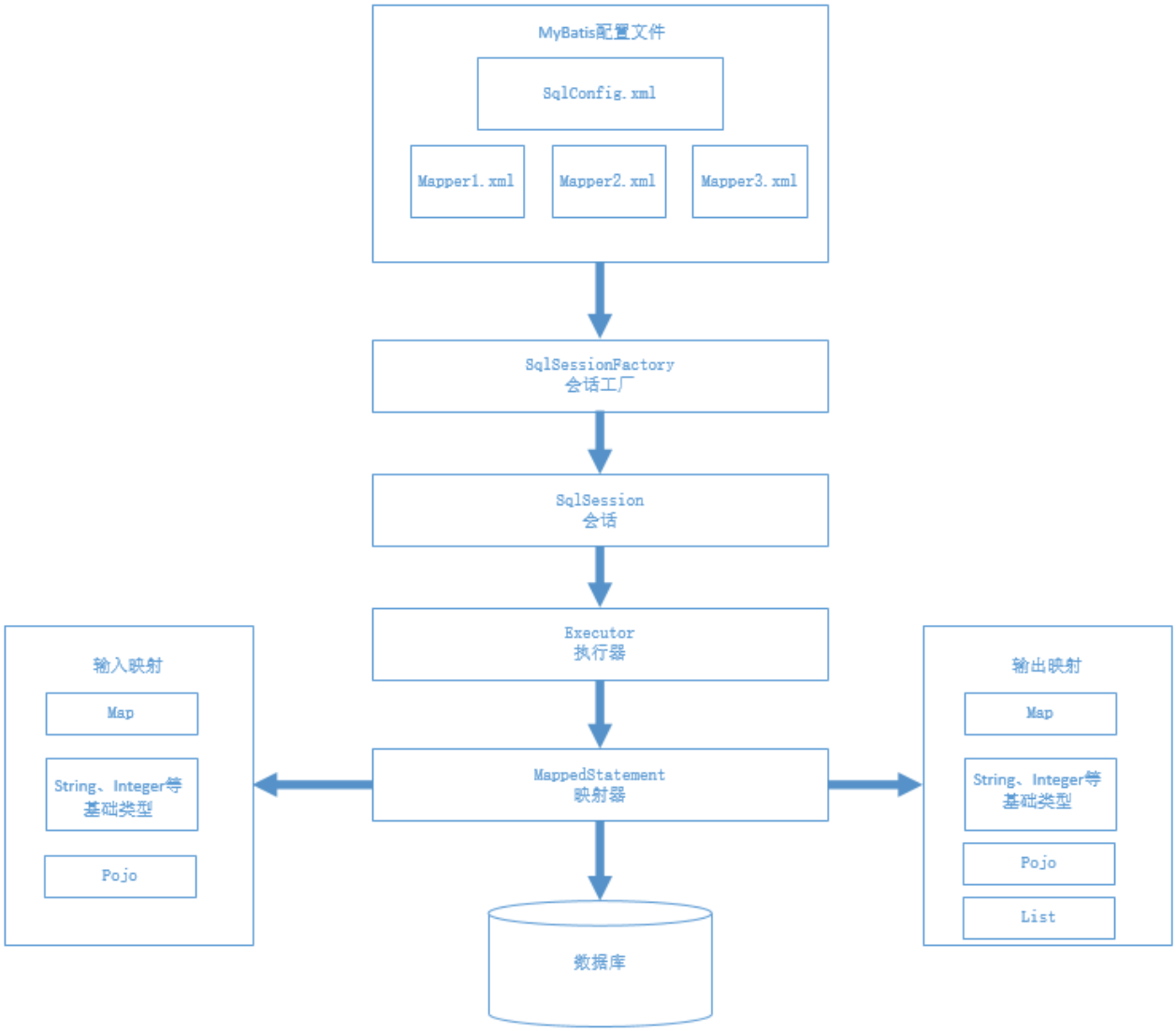
什么是 MyBatis?
解析配置
    配置文件解析过程分析
    解析 properties 节点
通过 sqlSessionFactory 创建 SqlSession
调用执行器执行
    执行器内部执行逻辑
    SimpleExecutor 查询数据库
MapperStatement 的数据库调用
执行过程的总结

什么是 MyBatis?

MyBatis 是一款优秀的持久层框架，它支持自定义 SQL、存储过程以及高级映射。MyBatis 免除了几乎所有的 JDBC 代码以及设置参数和获取结果集的工作。MyBatis 可以通过简单的 XML 或注解来配置和映射原始类型、接口和 Java POJO（Plain Old Java Objects，普通老式 Java 对象）为数据库中的记录。

mybatis 架构图：



从架构图中可以看出，mybatis 的主要步骤为：

1. 解析配置
2. 通过 sqlSessionFactory 创建 sqlSession
3. 调用执行器 executor 执行
4. 最后通过 MapperStatement 进行数据库调用

下面依次分配每个步骤

解析配置

在使用 MyBatis 框架时，通常会进行一定的设置，使其能 更好的满足我们的需求。对于一个框架来说，提供较为丰富的配置文件，也是其灵活性的体现。

配置文件解析过程分析

在使用 MyBatis 时，第一步要做的事情一般是根据配置文件构建 SqlSessionFactory 对象。相关代码大致如下：

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

在上面代码中，我们首先会使用 MyBatis 提供的工具类 Resources 加载配置文件，得到一个输入流。然后再通过 SqlSessionFactoryBuilder 对象的 build 方法构建 SqlSessionFactory 对象。这里的 build 方法是我们分析配置文件解析过程的入口方法。下面我们来看一下这个方法的代码：

```
// -☆- SqlSessionFactoryBuilder
public SqlSessionFactory build(InputStream inputStream){
        // 调用重载方法
        return build(inputStream,null,null);
        }
public SqlSessionFactory build(InputStream inputStream,String
environment,Properties properties){
        try{
        // 创建配置文件解析器
        XMLConfigBuilder parser=
        new XMLConfigBuilder(inputStream,environment,properties);
        // 调用 parse 方法解析配置文件，生成 Configuration 对象
        return build(parser.parse());
        }catch(Exception e){
        throw ExceptionFactory.wrapException("……",e);
        }finally{
        ErrorContext.instance().reset();
        try{
        inputStream.close();
        }catch(IOException e){

        }
        }
        }
public SqlSessionFactory build(Configuration config){
        // 创建 DefaultSqlSessionFactory
        return new DefaultSqlSessionFactory(config);
        }
```

从上面的代码中，我们大致可以猜出 MyBatis 配置文件是通过 XMLConfigBuilder 进行解析的。不过并具体的解析逻辑在内部。继续往下看。这次来看一下 XMLConfigBuilder 的 parse 方法，如下：

```
// -☆- XMLConfigBuilder
public Configuration parse(){
        if(parsed){
        throw new BuilderException("……");
        }
        parsed=true;
        // 解析配置
        parseConfiguration(parser.evalNode("/configuration"));
        return configuration;
        }
```

parse 方法获取了 xml 中的 configuration 节点

```
private void parseConfiguration(XNode root){
        try{
        // 解析 properties 配置
        propertiesElement(root.evalNode("properties"));
        // 解析 settings 配置，并将其转换为 Properties 对象
        Properties settings=
        settingsAsProperties(root.evalNode("settings"));
        // 加载 vfs
        loadCustomVfs(settings);
        // 解析 typeAliases 配置
        typeAliasesElement(root.evalNode("typeAliases"));

        // 解析 plugins 配置
        pluginElement(root.evalNode("plugins"));
        // 解析 objectFactory 配置
        objectFactoryElement(root.evalNode("objectFactory"));
        // 解析 objectWrapperFactory 配置
        objectWrapperFactoryElement(root.evalNode("objectWrapperFactory"));
        // 解析 reflectorFactory 配置
        reflectorFactoryElement(root.evalNode("reflectorFactory"));
        // settings 中的信息设置到 Configuration 对象中
        settingsElement(settings);
        // 解析 environments 配置
        environmentsElement(root.evalNode("environments"));
        // 解析 databaseIdProvider，获取并设置 databaseId 到 Configuration 对象
        databaseIdProviderElement(root.evalNode("databaseIdProvider"));
        // 解析 typeHandlers 配置
        typeHandlerElement(root.evalNode("typeHandlers"));
        // 解析 mappers 配置
        mapperElement(root.evalNode("mappers"));
        }catch(Exception e){
        throw new BuilderException("……");
        }
        }
```

到此，一个完整的配置解析过程就呈现出来了，每个节点的的解析逻辑均封装在了相应的方法中。接下来以 properties 节点解析过程为例

解析 properties 节点

节点的解析工作由 propertiesElement 这个方法完成的，在分析方法的源码前， 我们先来看一下节点是如何配置的。如下：

```

<properties resource="jdbc.properties">
    <property name="jdbc.username" value="coolblog"/>
    <property name="hello" value="world"/>
</properties>
```

上面配置包含了一个 resource 属性，和两个子节点。接下面，我们参照上面的配置，来分析 propertiesElement 方法的逻辑。如下。

```
// -☆- XMLConfigBuilder
private void propertiesElement(XNode context)throws Exception{
        if(context!=null){
    // 解析 propertis 的子节点，并将这些节点内容转换为属性对象 Properties
            Properties defaults=context.getChildrenAsProperties();
    // 获取 propertis 节点中的 resource 和 url 属性值
            String resource=context.getStringAttribute("resource");
            String url=context.getStringAttribute("url");
    // 两者都不用空，则抛出异常
            if(resource!=null&&url!=null){
            throw new BuilderException("...");
            }
            if(resource!=null){
    // 从文件系统中加载并解析属性文件
            defaults.putAll(Resources.getResourceAsProperties(resource));
            }else if(url!=null){
    // 通过 url 加载并解析属性文件
            defaults.putAll(Resources.getUrlAsProperties(url));
            }

            Properties vars=configuration.getVariables();
            if(vars!=null){
            defaults.putAll(vars);
            }
            parser.setVariables(defaults);
    // 将属性值设置到 configuration 中
            configuration.setVariables(defaults);
        }
}

public Properties getChildrenAsProperties(){
        Properties properties=new Properties();
// 获取并遍历子节点
        for(XNode child:getChildren()){
// 获取 property 节点的 name 和 value 属性
        String name=child.getStringAttribute("name");
        String value=child.getStringAttribute("value");
            if(name!=null&&value!=null){
    // 设置属性到属性对象中
                properties.setProperty(name,value);
            }
        }
        return properties;
}

// -☆- XNode
public List<XNode> getChildren() {
        List<XNode> children = new ArrayList<XNode>();
// 获取子节点列表
        NodeList nodeList = node.getChildNodes();
```

```
        if (nodeList != null) {
            for (int i = 0, n = nodeList.getLength(); i < n; i++) {
                Node node = nodeList.item(i);
                if (node.getNodeType() == Node.ELEMENT_NODE) {
        // 将节点对象封装到 XNode 中，并将 XNode 对象放入 children 列表中
                children.add(new XNode(xpathParser, node, variables));
                }
            }
        }
        return children;
}
```

上面就是节点解析过程，不是很复杂。主要包含三个步骤，一是解析节点的子节点，并将解析结果设置到 Properties 对象中。二是从文件系统或通过网络读取属性配置，这取决于节点的 resource 和 url 是否为空。 最后一步则是将包含属性信息的 Properties 对象设置到 XPathParser 和 Configuration 中。 需要注意的是，propertiesElement 方法是先解析节点的子节点内容，然后再 从文件系统或者网络读取属性配置，并将所有的属性及属性值都放入到 defaults 属性对象中。 这会导致同名属性覆盖的问题，也就是从文件系统，或者网络上读取到的属性和属性值会覆 盖掉子节点中同名的属性和及值。假如上面配置中的 jdbc.properties 内容如下：

```
jdbc.driver=com.mysql.cj.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/myblog?...
jdbc.username=root
jdbc.password=1234
```

最终将 xml 中的各种配置进行解析，将各种配置映射为 Configuration 类，再将 Configuration 对象交由 DefaultSqlSessionFactory 对象，创建出 DefaultSqlSessionFactory。

通过 sqlSessionFactory 创建 SqlSession

sqlSessionFactory 的默认实现为 DefaultSqlSessionFactory。那么来看 DefaultSqlSessionFactory 获取 SqlSession 的实现。

```
public class DefaultSqlSessionFactory implements SqlSessionFactory {

    private final Configuration configuration;

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

    @Override
    public SqlSession openSession() {
        return openSessionFromDataSource(configuration.getDefaultExecutorType(),
null, false);
    }

    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);
            final Executor executor = configuration.newExecutor(tx, execType);
            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();
        }
    }

    private TransactionFactory getTransactionFactoryFromEnvironment(Environment environment) {
        if (environment == null || environment.getTransactionFactory() == null) {
            return new ManagedTransactionFactory();
        }
        return environment.getTransactionFactory();
    }
}
```

可以看到 SqlSessionFactory 中创建 SqlSession 的方法，就是获取 configuration 中的 Environment、TransactionFactory、Executor。将这三个对象交由 DefaultSqlSession 进行创建，并返回。

调用执行器执行

在上一节中，获取到了 SqlSession，而 mybatis 执行的过程中最主要的是调用 Executor 进行执行，那我们来看 SqlSession 是如何调用 Executor。

上一节已经确定了 SqlSession 的默认实现类是 DefaultSqlSession，那么来看 DefaultSqlSession 的实现。

```

public class DefaultSqlSession implements SqlSession {

    private final Configuration configuration;
    private final Executor executor;

    private final boolean autoCommit;
    private boolean dirty;
    private List<Cursor<?>> cursorList;

    public DefaultSqlSession(Configuration configuration, Executor executor, boolean autoCommit) {
        this.configuration = configuration;
        this.executor = executor;
        this.dirty = false;
        this.autoCommit = autoCommit;
```

```
16      }
17
18      @Override
19      public <T> T selectOne(String statement) {
20          return this.selectOne(statement, null);
21      }
22
23
24      @Override
25      public <T> T selectOne(String statement, Object parameter) {
26          // Popular vote was to return null on 0 results and throw exception on too
    many.
27          List<T> list = this.selectList(statement, parameter);
28          if (list.size() == 1) {
29              return list.get(0);
30          } else if (list.size() > 1) {
31              throw new TooManyResultsException("Expected one result (or null) to be
    returned by selectOne(), but found: " + list.size());
32          } else {
33              return null;
34          }
35      }
36
37
38      @Override
39      public <E> List<E> selectList(String statement, Object parameter) {
40          return this.selectList(statement, parameter, RowBounds.DEFAULT);
41      }
42
43
44      @Override
45      public <E> List<E> selectList(String statement, Object parameter, RowBounds
    rowBounds) {
46          return selectList(statement, parameter, rowBounds,
    Executor.NO_RESULT_HANDLER);
47      }
48
49
50      private <E> List<E> selectList(String statement, Object parameter, RowBounds
    rowBounds, ResultHandler handler) {
51          try {
52              MappedStatement ms = configuration.getMappedStatement(statement);
53              dirty |= ms.isDirtySelect();
54              return executor.query(ms, wrapCollection(parameter), rowBounds,
    handler);
55          } catch (Exception e) {
56              throw ExceptionFactory.wrapException("Error querying database.  Cause:
    " + e, e);
57          } finally {
58              ErrorContext.instance().reset();
59          }
60      }
61  }
```

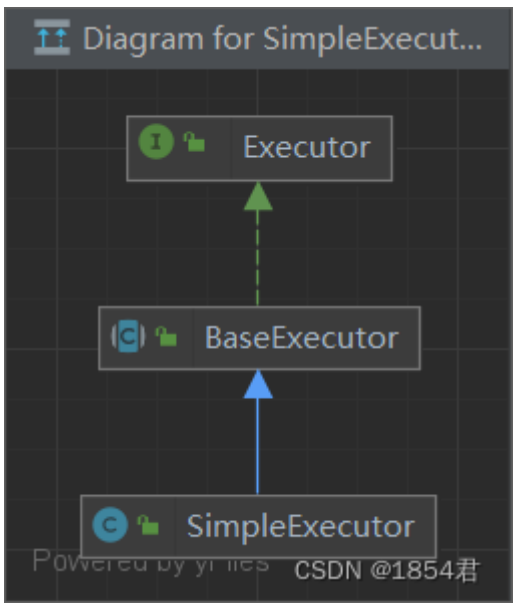
以 SqlSession 的 selectOne 为例，SqlSession 方法经过了好几个方法的重载，最终是调用了 selectList 方法。而 selectList 方法的主要逻辑为调用执行器的 query 方法：

```
    private <E> List<E> selectList(String statement, Object parameter, RowBounds rowBounds, ResultHandler handler) {
        try {
            // 通过 mapper.xml 中的方法id 从 configuration 获取解析后的对应 MappedStatement 对象。
            MappedStatement ms = configuration.getMappedStatement(statement);
            // 判断是否是脏读
            dirty |= ms.isDirtySelect();
            // 调用执行器执行
            return executor.query(ms, wrapCollection(parameter), rowBounds, handler);
        } catch (Exception e) {
            throw ExceptionFactory.wrapException("Error querying database.  Cause: " + e, e);
        } finally {
            ErrorContext.instance().reset();
        }
    }
```

可以看到 SqlSession 的执行，其实默认就是通过 executor 进行调用。

执行器内部执行逻辑

执行器的默认实现类为：org.apache.ibatis.executor.SimpleExecutor。首先看他的类图：



可以看到 SimpleExecutor 继承了 BaseExecutor 而 BaseExecutor 实现了 Executor。SimpleExecutor 间接的实现了 Executor 接口，而 query 的主要逻辑在 BaseExecutor 中，下面先看 BaseExecutor 的逻辑。

```
public abstract class BaseExecutor implements Executor {

    @Override
```

```
    public <E> List<E> query(MappedStatement ms, Object parameter, RowBounds rowBounds, ResultHandler resultHandler) throws SQLException {
        // 将参数将由 MappedStatement 获取 xml 中的 sql 并进行参数替换
        BoundSql boundSql = ms.getBoundSql(parameter);
        // 创建缓存 key，作为一级缓存的 key。
        CacheKey key = createCacheKey(ms, parameter, rowBounds, boundSql);
        // 调用重载方法继续执行
        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 {
        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 {
                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;
    }


```

```
    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;
    }


    protected abstract <E> List<E> doQuery(MappedStatement ms, Object parameter,
RowBounds rowBounds, ResultHandler resultHandler, BoundSql boundSql)
            throws SQLException;
}
```

在 BaseExecutor 方法中，调用的逻辑较多，并且还调用了抽象方法，但他的主要逻辑：

1. 获取 BoundSql、获取 CacheKey
2. 调用重载 query 方法
3. 对当前条件是否符合要求的条件校验
4. 尝试使用 cacheKey 从缓存中获取结果。如果结果存在则返回，不存在继续执行
5. 结果不存在则调用 queryFromDatabase 方法，通过方法名可以看出这是与数据库开始打交道的方法。
6. queryFromDatabase 的主要逻辑是调用了 doQuery 方法。doQuery 是抽象方法，在他的子类中，即本节开头所说的 SimpleExecutor。

SimpleExecutor 查询数据库

省略 SimpleExecutor 其他代码，只关注主要逻辑。在 doQuery 中最主要的就是创建 StatementHandler，并通过 StatementHandler 解析出 jdbc 中的 Statement 对象，并调用 StatementHandler 的 query 方法，将 jdbc 的 Statement 对象交由其执行。这里的 StatementHandler 就是我们所说的 MapperStatement。

```
public class SimpleExecutor extends BaseExecutor {


    @Override
    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();
            StatementHandler handler = configuration.newStatementHandler(wrapper,
ms, parameter, rowBounds, resultHandler, boundSql);
            stmt = prepareStatement(handler, ms.getStatementLog());
            return handler.query(stmt, resultHandler);
```

```
        } finally {
            closeStatement(stmt);
        }
    }


    private Statement prepareStatement(StatementHandler handler, Log statementLog)
throws SQLException {
        Statement stmt;
        Connection connection = getConnection(statementLog);
        stmt = handler.prepare(connection, transaction.getTimeout());
        handler.parameterize(stmt);
        return stmt;
    }

}
```

MapperStatement 的数据库调用

MapperStatement 的调用在 StatementHandler 中，而 StatementHandler 的默认实现是 SimpleStatementHandler 中，我们来看他的 query 方法是如何实现。

```
public class SimpleStatementHandler extends BaseStatementHandler {

    @Override
    public <E> List<E> query(Statement statement, ResultHandler resultHandler) throws
SQLException {
        String sql = boundSql.getSql();
        statement.execute(sql);
        return resultSetHandler.handleResultSets(statement);
    }
}
```

可以看到 query 方法其实最主要的逻辑就是调用 jdbc 的 Statement 对象进行 sql 的执行，并将执行的结果交由 ResultHandler 进行处理，最终将结果返回。

执行过程的总结

整个执行过程可以由开头的结论进行总结。并且在整个探究的过程中可以发现，mybatis 并没有越过 jdbc 直接进行数据库的调用，而是将 jdbc 进行了封装，并提供了另一套方便、优雅的使用方式交由使用者进行使用，在实现该使用方式时，mybatis 创造出了整个流程，并由几个关键类进行支撑，如：SqlSessionFactory、Executor 等。

1. 解析配置
2. 通过 sqlSessionFactory 创建 sqlSession
3. 调用执行器 executor 执行
4. 最后通过 MapperStatement 进行数据库调用