

MyBatis核心源码深度剖析

课程目标：

1. mybatis源码架构说明
2. mybatis核心执行流程梳理
3. mybatis工作机制和实现原理详解
4. mybatis缓存原理分析
5. 手写mybatis框架及其设计模式详解

一、概述

MyBatis 是一款优秀的持久层框架，也是当前最流行的java持久层框架之一，它内部封装了jdbc，使开发者只需要关注sql语句本身，而不需要花费精力去处理加载驱动、创建连接、创建statement等繁杂的过程。采用ORM思想解决了实体和数据库映射的问题，对jdbc进行了封装，屏蔽了jdbc api底层访问细节，使我们不用与jdbc api打交道，就可以完成对数据库的持久化操作。

mybatis通过xml或注解的方式将要执行的各种statement配置起来，并通过java对象和statement中sql的动态参数进行映射生成最终执行的sql语句，最后由mybatis框架执行sql并将结果映射为java对象并返回。

为了更好地学习和理解mybatis背后的设计思路，作为高级开发人员，有必要深入研究了解优秀框架的源码，以便更好的借鉴其思想。同时，框架作为设计模式的主要应用场景，通过研究优秀框架的源码，可以更好的领会设计模式的精髓。

二、MyBatis源码分析导入

2.1 为什么要看MyBatis框架的源码

通过学习开源框架MyBatis的源码，我们可以深入学习到框架的解析思路和底层的实现原理，掌握源码的剖析办法，快速增加查看源码的经验；

- 1) 在使用MyBatis框架进行开发时，如果你对其源码有所了解，可以最大化地减少出故障的可能；
- 2) 学习源码分析的最大好处是可以开阔思维，提升架构设计能力，通过看源码，看别人如何设计，然后领悟到这样设计的好处，理解优秀的代码设计思想；
- 3) 互联网大厂对有经验的开发人员的招聘，对架构思想和底层的理解能力考察方面比较重视，学习完有助于提高自己的竞争力；
- 4) 可以在深入的学习、剖析后，可以对框架进行改造，进而自定义MyBatis框架，提升架构能力。

2.2 如何深入学习MyBatis源码

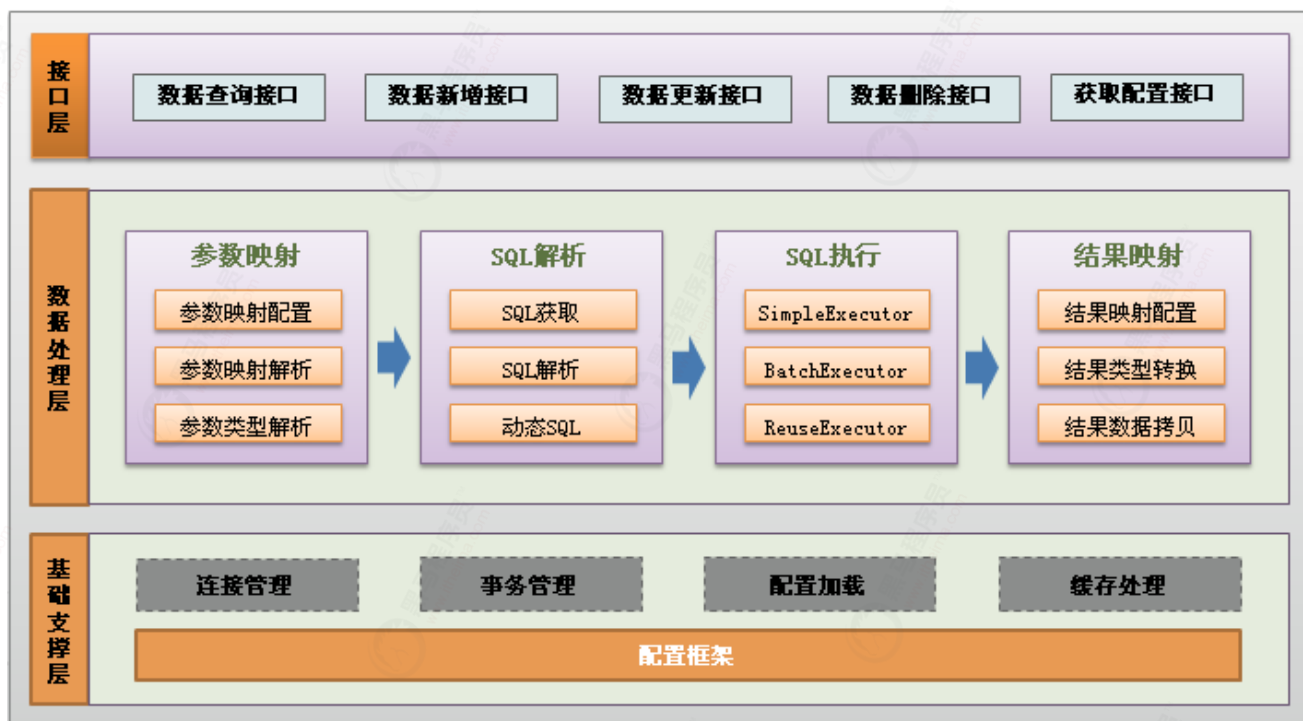
- 1) 查看MyBatis官方文档；
<https://mybatis.org/mybatis-3/zh/index.html>
- 2) 断点跟进源码，参照主线，一步步分析；
- 3) 手动自定义MyBatis框架，加深对框架源码的理解，掌握源码的学习方法，进而提升自身架构能力；

2.3 源码分析的5大原则

- 1) 紧跟入口
- 2) 看图梳理
- 3) 先粗后细
- 4) 精略结合
- 5) 猜想验证

三、MyBatis架构体系深入剖析

3.1 MyBatis的整体架构体系



3.2 MyBatis的工作机制和实现原理

- 1) 接口层
- 2) 数据处理核心层
- 3) 基础支撑层

(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。

(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。

JDBC代码回顾：

```

/**
 * JDBC开发示例代码
 */
public class JDBCTest {

    private static Logger logger = Logger.getLogger(JDBCTest.class);

    public static void main(String[] args) throws Exception {
        // 1、注册驱动
        DriverManager.registerDriver(new com.mysql.jdbc.Driver());
        // 2、建立连接
        Connection con =
DriverManager.getConnection("jdbc:mysql://localhost:3306/mybatis_indepth?
useUnicode=true&characterEncoding=utf-8&serverTimezone=GMT", "root", "root");
        // 3、编写sql, 进行预编译
        String sql = " select * from user;";
        PreparedStatement ps = con.prepareStatement(sql);
        // 4、执行查询, 得到结果集
        ResultSet rs = ps.executeQuery();
        while (rs.next()) {
            int id = rs.getInt("id");
            String name = rs.getString("name");

            logger.info("====> id=" + id + "\tname=" + name);
        }
        //5、关闭事务
        rs.close();
        ps.close();
        con.close();
    }
}

```

mybatis代码:

```

InputStream in;
SqlSessionFactoryBuilder builder;
SqlSessionFactory factory;
SqlSession session;
UserMapper userMapper = null;

@Before
public void init() throws Exception {
    //1.读取配置文件

```

```

    in = Resources.getResourceAsStream("SqlMapConfig.xml");
    //2.创建SqlSessionFactory工厂
    builder = new SqlSessionFactoryBuilder();
    factory = builder.build(in);
    //3.使用工厂生产SqlSession对象
    session = factory.openSession();
    //4.使用SqlSession创建Dao接口的代理对象
    userMapper = session.getMapper(UserMapper.class);
}

@After
public void after() throws IOException {
    //6.释放资源
    session.close();
    in.close();
}

/**
 * 入门案例
 */
@Test
public void testFindById() throws Exception {
    //5.使用代理对象执行方法
    User user = userMapper.findUserById(1);
    System.out.println(user);
}

```

思考：mybatis为我们做了什么？

- mybatis如何获取数据源连接信息？
- mybatis如何获取到需要执行的sql语句？
- mybatis是如何完成sql执行的？
- mybatis如何完成参数映射和结果封装？

1) 接口层

概述：对应 `session` 模块。

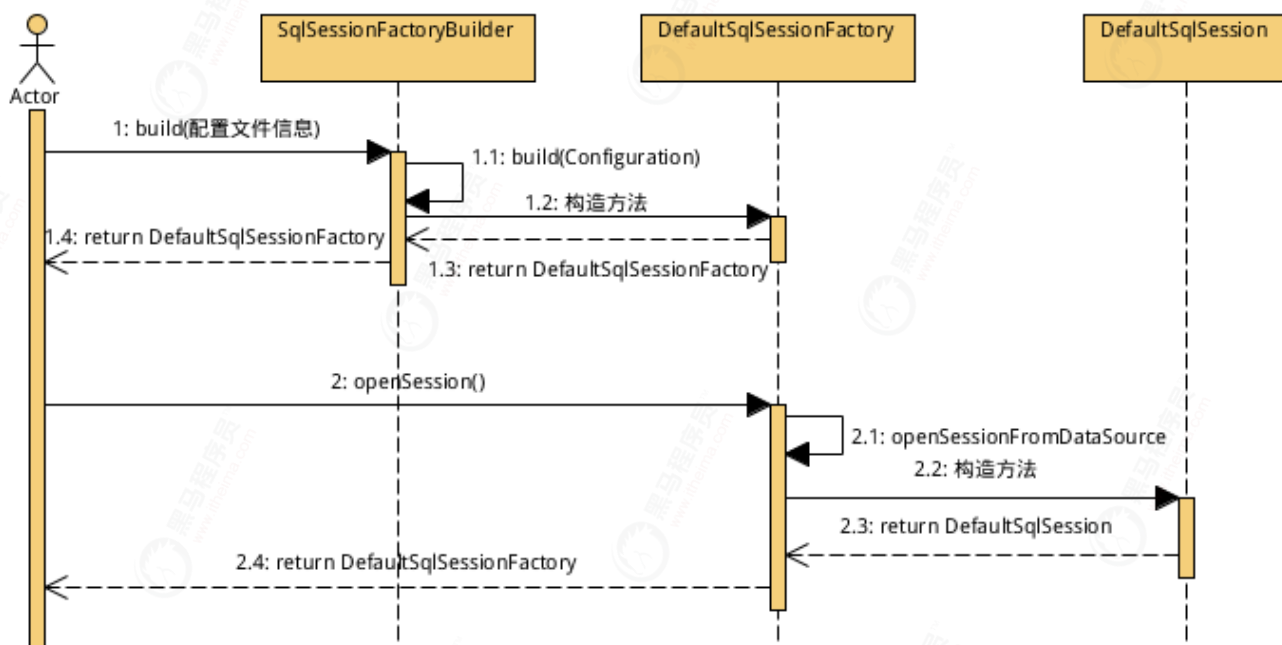
接口层相对简单，其核心是 `SqlSession` 接口，该接口中定义了 MyBatis 暴露给应用程序调用的 API，也就是上层应用与 MyBatis 交互的桥梁。接口层在接收到调用请求时，会调用核心处理层的相应模块来完成具体的数据库操作。

作用：

使用 `SqlSession` 接口和 `Mapper` 接口通知调用哪个 sql 还有关联参数。

可以实现数据的增/删/改/查接口 配置信息维护接口，进行动态的更改配置

1.1) 获取 `SqlSession` 流程分析：



1.2) SqlSession源码分析:

作用：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

2) 数据处理核心层

在核心处理层中，实现了 MyBatis 的核心处理流程。

其中包括 MyBatis 的初始化以及完成一次数据库操作的涉及的全部流程。

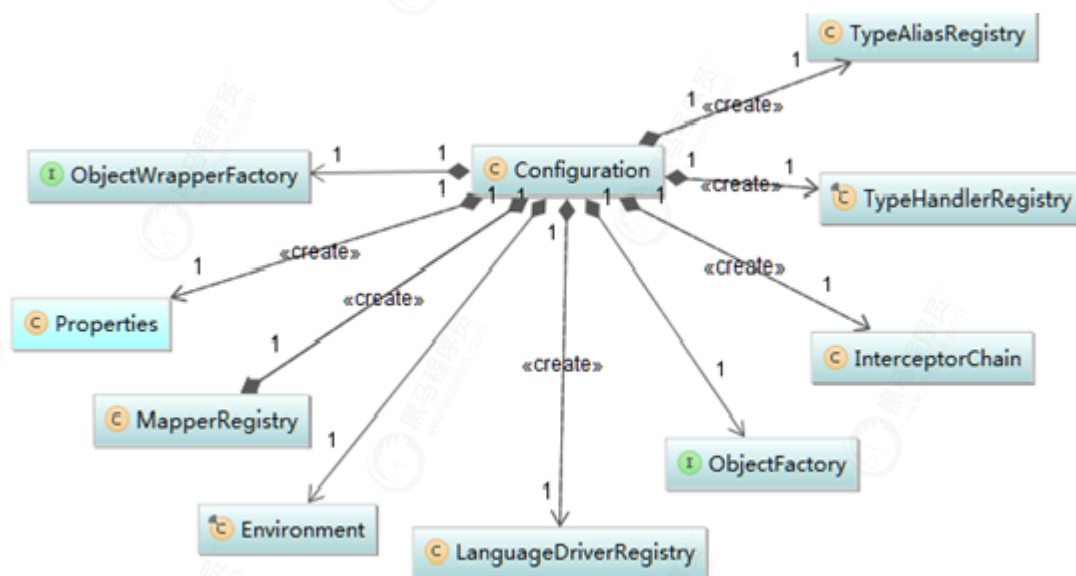
2.1) 配置解析

概述：对应 `builder` 和 `mapping` 模块。前者为配置解析过程，后者主要为 SQL 操作解析后的映射。

在 MyBatis 初始化过程中，会加载 `mybatis-config.xml` 配置文件、映射配置文件以及 Mapper 接口中的注解信息，解析后的配置信息会形成相应的对象并保存到 `Configuration` 对象中。

利用该 Configuration 对象创建 SqlSessionFactory 对象。待 MyBatis 初始化之后，开发人员可以通过初始化得到 SqlSessionFactory 创建 SqlSession 对象并完成数据库操作。

Configuration 概述：是一个所有配置信息的容器对象 实战分析：Configuration 对象涉及到的配置信息分析



简单的理解：MyBatis初始化的过程，就是创建 Configuration对象，加载各种配置信息的过程

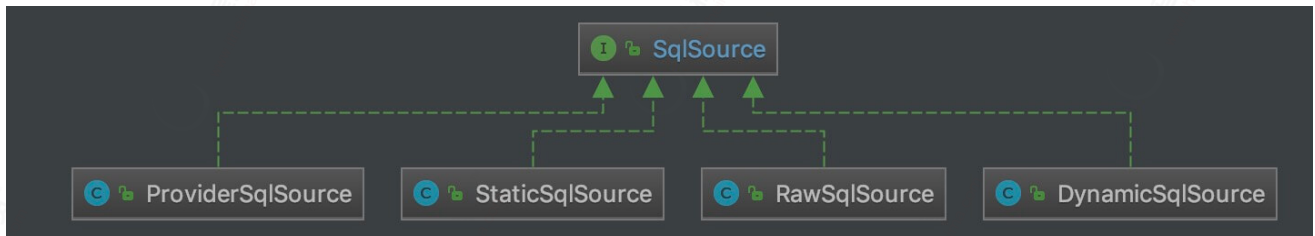
2.2) SQL解析 (SqlSource)

概述： 对应 `scripting` 模块。

MyBatis 中的 `scripting` 模块，会根据用户传入的实参，解析映射文件中定义的动态 SQL 节点，并形成数据库可执行的 SQL 语句。之后会处理 SQL 语句中的占位符，绑定用户传入的实参

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。

实战分析： SqlSource 接口继承体系



各个实现类分析：

RawSqlSource 负责处理静态 SQL 语句，它们最终会把处理后的 SQL 封装 **StaticSqlSource** 进行返回。

StaticSqlSource 处理包含的 SQL 可能含有 “?” 占位符，可以被数据库直接执行。

DynamicSqlSource 负责处理动态 SQL 语句。

ProviderSqlSource 实现 **SqlSource** 接口，基于方法上的 `@ProviderXXX` 注解的 **SqlSource** 实现类。

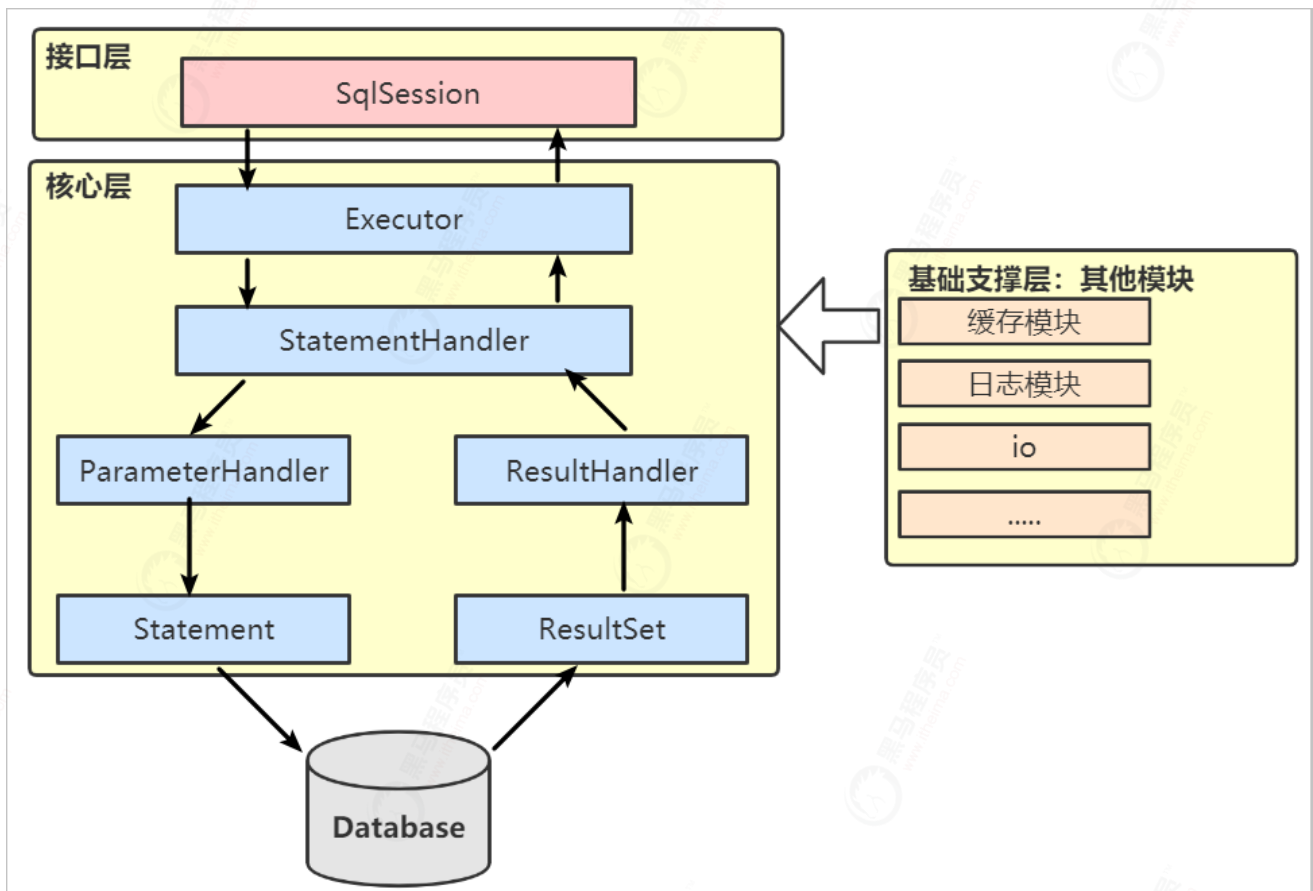
2.3) SQL执行 (Executor)

概述：对应 `executor` 模块

是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。

SQL 语句的执行涉及多个组件，其中比较重要的是 **Executor**、**StatementHandler**、**ParameterHandler** 和 **ResultSetHandler**。

- **Executor** 主要负责维护一级缓存和二级缓存，并提供事务管理的相关操作，它会将数据库相关操作委托给 **StatementHandler**完成。
- **StatementHandler** 首先通过 **ParameterHandler** 完成 SQL 语句的实参绑定，然后通过 `java.sql.Statement` 对象执行 SQL 语句并得到结果集，最后通过 **ResultSetHandler** 完成结果集的映射，得到结果对象并返回。



3) 基础支撑层:

概述: 基础支持层，包含整个 MyBatis 的基础模块，这些模块为核心处理层的功能提供了良好的支撑。

日志: 对应 `logging` 包

概述: Mybatis 提供了详细的日志输出信息，还能够集成多种日志框架，其日志模块的主要功能就是集成第三方日志框架。

设计模式分析

使用的适配器模式分析

缓存机制: 对应 `cache` 包

一级缓存

概述: Session或Statement作用域级别的缓存, 默认是Session, BaseExecutor中根据MappedStatement的Id、SQL、参数值以及rowBound(边界)来构造CacheKey, 并使用BaseExecutor中的localCache来维护此缓存。

实战应用场景分析

默认开启的缓存

二级缓存

概述: 全局的二级缓存, 通过CacheExecutor来实现, 其委托TransactionalCacheManager来保存/获取缓存

实战应用场景分析

缓存的效率以及应用场景

注意点: 两级缓存与Mybatis以及整个应用是运行在同一个JVM中的, 共享同一块内存, 如果这两级缓存中的数据量较大, 则可能影响系统中其它功能, 需要缓存大量数据时, 优先考虑使用Redis、Memcache等缓存产品。

数据源/连接池: 对应 `datasource` 包

概述: Mybatis自身提供了相应的数据源实现, 也提供了与第三方数据源集成的接口。

分析: 主要实现类是PooledDataSource, 包含了最大活动连接数、最大空闲连接数、最长取出时间(避免某个线程过度占用)、连接不够时的等待时间。

实战应用: 连接池、检测连接状态等, 选择性能优秀的数据源组件, 对于提供ORM框架以及整个应用的性能都是非常重要的。

事务管理: 对应 `transaction` 包

概述: Mybatis自身对数据库事务进行了抽象, 提供了相应的事务接口和简单实现。

注意点: 一般地, Mybatis与Spring框架集成, 由Spring框架管理事务。

反射: 对应 `reflection` 包

概述: 对Java原生的反射进行了很好的封装, 提供了简易的API, 方便上层调用, 并且对反射操作进行了一系列的优化, 提高了反射操作的性能。

实战应用

- ① 缓存了类的元数据 (MetaClass)
- ② 对象的元数据 (MetaObject)

IO 模块: 对应 `io` 包。

资源加载模块, 主要是对类加载器进行封装, 确定类加载器的使用顺序, 并提供了加载类文件以及其他资源文件的功能。

解析器: 对应 `parsing` 包

解析器模块，主要提供了两个功能：

- 1.对 XPath 进行封装，为 MyBatis 初始化时解析 mybatis-config.xml 配置文件以及映射配置文件提供支持。
- 2.为处理动态 SQL 语句中的占位符提供支持。

3.3 MyBatis的核心配置文件解析原理

在 MyBatis 初始化过程中，会加载 mybatis-config.xml 配置文件、映射配置文件以及 Mapper 接口中的注解信息，解析后的配置信息会形成相应的对象并保存到 Configuration 对象中

1) 解析的目的

概述：通过资源类Resources读入“SqlMapConfig.xml”文件 使用SqlSessionFactoryBuilder类生成我们需要的SqlSessionFactory类。 **目的：**

mybatis解析配置文件最本质的目的是为了获得Configuration对象；

然后，利用该 Configuration 对象创建 SqlSessionFactory对象。待 MyBatis 初始化之后，可以通过初始化得到 SqlSessionFactory 创建 SqlSession 对象并完成数据库操作。

2) XML 解析流程

2.1) 入口

MyBatis 的初始化流程的入口是 SqlSessionFactoryBuilder 的 build 方法：

```
/**
 * 构造 SqlSessionFactory 对象
 *
 * @param reader Reader 对象
 * @param environment 环境
 * @param properties Properties 变量
 * @return SqlSessionFactory 对象
 */
```

```

public SqlSessionFactory build(Reader reader, String environment, Properties
properties) {
    try {
        // <1> 创建 XMLConfigBuilder 对象
        XMLConfigBuilder parser = new XMLConfigBuilder(reader, environment,
properties);
        // <2> 执行 XML 解析
        // <3> 创建 DefaultSqlSessionFactory 对象
        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.
        }
    }
}
}

```

2.2) XMLConfigBuilder

`org.apache.ibatis.builder.xml.XMLConfigBuilder` , 继承 `BaseBuilder` 抽象类, XML 配置构建器;

主要负责解析 `mybatis-config.xml` 配置文件:

```

// 【1.构造设置Properties】
private XMLConfigBuilder(XPathParser parser, String environment, Properties
props) {
    // <1> 创建 Configuration 对象
    super(new Configuration());
    ErrorContext.instance().resource("SQL Mapper Configuration");
    // <2> 设置 Configuration 的 variables 属性
    this.configuration.setVariables(props);
    this.parsed = false;
    this.environment = environment;
    this.parser = parser;
}

// parse 【2. 判断是否解析过】
public Configuration parse() {
    // <1.1> 若已解析, 抛出 BuilderException 异常
    if (parsed) {

```

```

        throw new BuilderException("Each XMLConfigBuilder can only be used
once.");
    }
    // <1.2> 标记已解析
    parsed = true;
    // <2> 解析 XML configuration 节点
    parseConfiguration(parser.evalNode("/configuration"));
    return configuration;
}

```

//parseConfiguration 【3. 解析configuration节点】

```

private void parseConfiguration(XNode root) {
    try {
        //issue #117 read properties first
        // <1> 解析 <properties /> 标签
        propertiesElement(root.evalNode("properties"));
        // <2> 解析 <settings /> 标签
        Properties settings = settingsAsProperties(root.evalNode("settings"));
        // <3> 加载自定义 VFS 实现类
        loadCustomVfs(settings);
        // <4> 解析 <typeAliases /> 标签
        typeAliasesElement(root.evalNode("typeAliases"));
        // <5> 解析 <plugins /> 标签
        pluginElement(root.evalNode("plugins"));
        // <6> 解析 <objectFactory /> 标签
        objectFactoryElement(root.evalNode("objectFactory"));
        // <7> 解析 <objectWrapperFactory /> 标签
        objectWrapperFactoryElement(root.evalNode("objectWrapperFactory"));
        // <8> 解析 <reflectorFactory /> 标签
        reflectorFactoryElement(root.evalNode("reflectorFactory"));
        // <9> 赋值 <settings /> 到 Configuration 属性
        settingsElement(settings);
        // read it after objectFactory and objectWrapperFactory issue #631
        // <10> 解析 <environments /> 标签
        environmentsElement(root.evalNode("environments"));
        // <11> 解析 <databaseIdProvider /> 标签
        databaseIdProviderElement(root.evalNode("databaseIdProvider"));
        // <12> 解析 <typeHandlers /> 标签
        typeHandlerElement(root.evalNode("typeHandlers"));
        // <13> 解析 <mappers /> 标签
        mapperElement(root.evalNode("mappers"));
    } catch (Exception e) {
        throw new BuilderException("Error parsing SQL Mapper Configuration.
Cause: " + e, e);
    }
}

```

new Configuration()

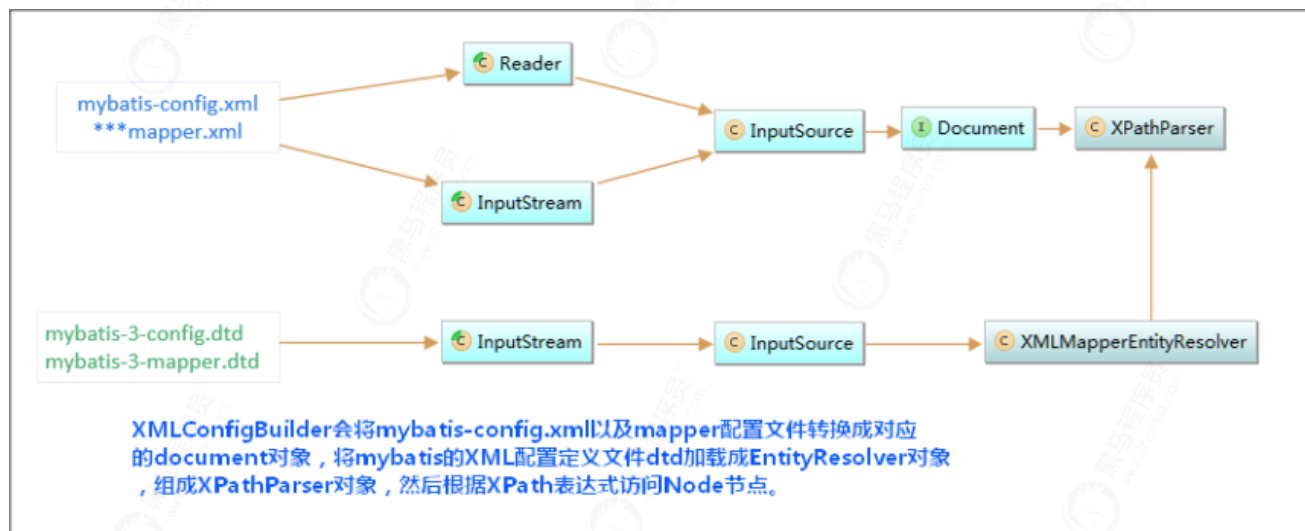
配置文件解析的本质就是获得Configuration对象 很多需要的成员变量需要根据 XML 配置文件解析后来赋值

parser.parse()

该函数就是XML解析的核心 解析全局配置文件，调用parse.evalNode()方法，将指定路径的config配置文件转换为XNode对象 调用parseConfiguration()方法逐步解析配置文件中的各个节点

build(configuration)

该函数用来创建一个具体的SqlSessionFactory对象。创建DefaultSqlSessionFactory对象，并将configuration赋值给相应的成员变量



3) 核心解析逻辑:

`org.apache.ibatis.parsing.XPathParser`，基于Java **XPath** 解析器，用于解析 MyBatis `mybatis-config.xml` 和 `**Mapper.xml` 等 XML 配置文件。属性如下：

```
/**
```

```

    * XML Document 对象
    */
private final Document document;
/**
    * 是否校验
    */
private boolean validation;
/**
    * XML 实体解析器
    */
private EntityResolver entityResolver;
/**
    * 变量 Properties 对象
    */
private Properties variables;
/**
    * Java XPath 对象
    */
private XPath xpath;

/**
    * 构造 XPathParser 对象
    *
    * @param xml XML 文件地址
    * @param validation 是否校验 XML
    * @param variables 变量 Properties 对象
    * @param entityResolver XML 实体解析器
    */
public XPathParser(String xml, boolean validation, Properties variables,
    EntityResolver entityResolver) {
    commonConstructor(validation, variables, entityResolver);
    this.document = createDocument(new InputSource(new StringReader(xml)));
}

/**
    公用的构造方法逻辑
    */
private void commonConstructor(boolean validation, Properties variables,
    EntityResolver entityResolver) {
    this.validation = validation;
    this.entityResolver = entityResolver;
    this.variables = variables;
    // 创建 XPathFactory 对象
    XPathFactory factory = XPathFactory.newInstance();

```

```

        this.xpath = factory.newXPath();
    }

/**
 * 创建 Document 对象 ---->将 XML 文件解析成 Document 对象
 *
 * @param inputSource XML 的 InputSource 对象
 * @return Document 对象
 */
private Document createDocument(InputSource inputSource) {
    // important: this must only be called AFTER common constructor
    try {
        // 1> 创建 DocumentBuilderFactory 对象
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        factory.setValidating(validation); // 设置是否验证 XML

        factory.setNamespaceAware(false);
        factory.setIgnoringComments(true);
        factory.setIgnoringElementContentWhitespace(false);
        factory.setCoalescing(false);
        factory.setExpandEntityReferences(true);

        // 2> 创建 DocumentBuilder 对象
        DocumentBuilder builder = factory.newDocumentBuilder();
        builder.setEntityResolver(entityResolver); // 设置实体解析器
        builder.setErrorHandler(new ErrorHandler() { // 实现都空的

            @Override
            public void error(SAXParseException exception) throws SAXException {
                throw exception;
            }

            @Override
            public void fatalError(SAXParseException exception) throws
SAXException {
                throw exception;
            }

            @Override
            public void warning(SAXParseException exception) throws SAXException
{
            }

        });
        // 3> 解析 XML 文件
        return builder.parse(inputSource);
    }
}

```

```

    } catch (Exception e) {
        throw new BuilderException("Error creating document instance. Cause: "
+ e, e);
    }
}

```

`org.apache.ibatis.builder.xml.XMLMapperEntityResolver`，实现 `EntityResolver` 接口，MyBatis 自定义 `EntityResolver` 实现类，用于加载本地的 `mybatis-3-config.dtd` 和 `mybatis-3-mapper.dtd` 这两个 DTD 文件。代码如下：

```

public class XMLMapperEntityResolver implements EntityResolver {

    private static final String IBATIS_CONFIG_SYSTEM = "ibatis-3-config.dtd";
    private static final String IBATIS_MAPPER_SYSTEM = "ibatis-3-mapper.dtd";
    private static final String MYBATIS_CONFIG_SYSTEM = "mybatis-3-config.dtd";
    private static final String MYBATIS_MAPPER_SYSTEM = "mybatis-3-mapper.dtd";

    /**
     * 本地 mybatis-config.dtd 文件
     */
    private static final String MYBATIS_CONFIG_DTD =
"org/apache/ibatis/builder/xml/mybatis-3-config.dtd";
    /**
     * 本地 mybatis-mapper.dtd 文件
     */
    private static final String MYBATIS_MAPPER_DTD =
"org/apache/ibatis/builder/xml/mybatis-3-mapper.dtd";

    /**
     * Converts a public DTD into a local one
     */
    @Override
    public InputSource resolveEntity(String publicId, String systemId) throws
SAXException {
        try {
            if (systemId != null) {
                String lowerCaseSystemId = systemId.toLowerCase(Locale.ENGLISH);
                // 本地 mybatis-config.dtd 文件
                if (lowerCaseSystemId.contains(MYBATIS_CONFIG_SYSTEM) ||
lowerCaseSystemId.contains(IBATIS_CONFIG_SYSTEM)) {
                    return getInputSource(MYBATIS_CONFIG_DTD, publicId,
systemId);

                    // 本地 mybatis-mapper.dtd 文件

```

```

        } else if (lowerCaseSystemId.contains(MYBATIS_MAPPER_SYSTEM) ||
lowerCaseSystemId.contains(IBATIS_MAPPER_SYSTEM)) {
            return getInputSource(MYBATIS_MAPPER_DTD, publicId,
systemId);
        }
    }
    return null;
} catch (Exception e) {
    throw new SAXException(e.toString());
}
}

private InputSource getInputSource(String path, String publicId, String
systemId) {
    InputSource source = null;
    if (path != null) {
        try {
            // 创建 InputSource 对象
            InputStream in = Resources.getResourceAsStream(path);
            source = new InputSource(in);
            // 设置 publicId、systemId 属性
            source.setPublicId(publicId);
            source.setSystemId(systemId);
        } catch (IOException e) {
            // ignore, null is ok
        }
    }
    return source;
}
}

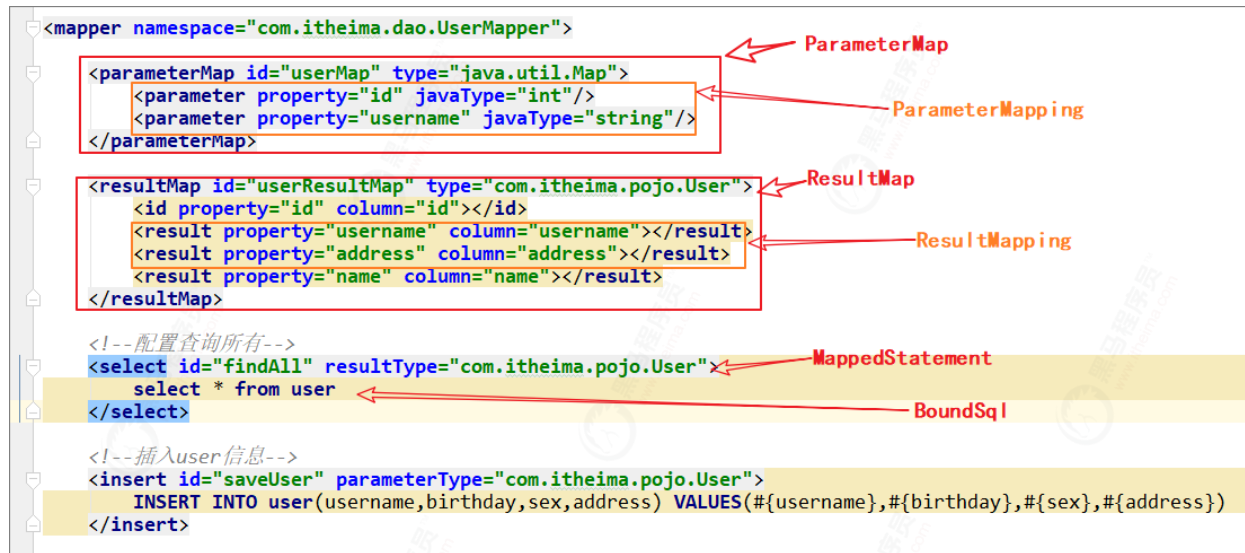
```

3.4 mybatis的mapper映射文件解析原理

其实XMLConfigBuilder在解析核心配置文件中mappers节点时，会进一步解析mapper映射文件。

加载 Mapper 映射配置文件这个步骤的主体是 XMLMapperBuilder

mapper文件示意:



1) XMLMapperBuilder:

概述: 【解析xml文件中的节点】

`org.apache.ibatis.builder.xml.XMLMapperBuilder` , 继承 `BaseBuilder` 抽象类, `Mapper` XML 配置构建器, 主要负责解析 `Mapper` 映射配置文件

配置package, 会遍历该包下所有的类

指定mapper文件的路径resource/url/class

具体解析逻辑通过XMLMapperBuilder类来完成, 解析xml文件中的`<mapper>`节点

`#parse()` 方法, 解析 `Mapper` XML 配置文件。代码如下:

```
public void parse() {
    // <1> 判断当前 Mapper 是否已经加载过
    if (!configuration.isResourceLoaded(resource)) {
        // <2> 解析 `<mapper />` 节点
        configurationElement(parser.evalNode("/mapper"));
        // <3> 标记该 Mapper 已经加载过
        configuration.addLoadedResource(resource);
    }
}
```

```

        // <4> 绑定 Mapper
        bindMapperForNamespace();
    }

    // <5> 解析待定的 <resultMap /> 节点
    parsePendingResultMaps();
    // <6> 解析待定的 <cache-ref /> 节点
    parsePendingCacheRefs();
    // <7> 解析待定的 SQL 语句的节点
    parsePendingStatements();
}

```

`#configurationElement(XNode context)` 方法，解析 `<mapper />` 节点。代码如下：

```

private void configurationElement(XNode context) {
    try {
        // <1> 获得 namespace 属性
        String namespace = context.getStringAttribute("namespace");
        if (namespace == null || namespace.equals("")) {
            throw new BuilderException("Mapper's namespace cannot be empty");
        }
        // <1> 设置 namespace 属性
        builderAssistant.setCurrentNamespace(namespace);
        // <2> 解析 <cache-ref /> 节点
        cacheRefElement(context.evalNode("cache-ref"));
        // <3> 解析 <cache /> 节点
        cacheElement(context.evalNode("cache"));
        // 已废弃！老式风格的参数映射。内联参数是首选，这个元素可能在将来被移除，这里不会记录。
        parameterMapElement(context.evalNodes("/mapper/parameterMap"));
        // <4> 解析 <resultMap /> 节点们
        resultMapElements(context.evalNodes("/mapper/resultMap"));
        // <5> 解析 <sql /> 节点们
        sqlElement(context.evalNodes("/mapper/sql"));
        // <6> 解析 <select /> <insert /> <update /> <delete /> 节点们

        buildStatementFromContext(context.evalNodes("select|insert|update|delete"));
    } catch (Exception e) {
        throw new BuilderException("Error parsing Mapper XML. The XML location is '" + resource + "'. Cause: " + e, e);
    }
}

```

`#cacheRefElement(XNode context)` 方法，解析 `<cache-ref />` 节点。代码如下：

```

private void cacheRefElement(XNode context) {
    if (context != null) {
        // <1> 获得指向的 namespace 名字, 并添加到 configuration 的 cacheRefMap 中
        configuration.addCacheRef(builderAssistant.getCurrentNamespace(),
            context.getStringAttribute("namespace"));
        // <2> 创建 CacheRefResolver 对象, 并执行解析
        CacheRefResolver cacheRefResolver = new
        CacheRefResolver(builderAssistant, context.getStringAttribute("namespace"));
        try {
            cacheRefResolver.resolveCacheRef();
        } catch (IncompleteElementException e) {
            // <3> 解析失败, 添加到 configuration 的 incompleteCacheRefs 中
            configuration.addIncompleteCacheRef(cacheRefResolver);
        }
    }
}

```

#cacheElement(XNode context) 方法, 解析 `cache />` 标签。代码如下:

```

// XMLMapperBuilder.java

private void cacheElement(XNode context) throws Exception {
    if (context != null) {
        // <1> 获得负责存储的 Cache 实现类
        String type = context.getStringAttribute("type", "PERPETUAL");
        Class<? extends Cache> typeClass = typeAliasRegistry.resolveAlias(type);
        // <2> 获得负责过期的 Cache 实现类
        String eviction = context.getStringAttribute("eviction", "LRU");
        Class<? extends Cache> evictionClass =
        typeAliasRegistry.resolveAlias(eviction);
        // <3> 获得 flushInterval、size、readwrite、blocking 属性
        Long flushInterval = context.getLongAttribute("flushInterval");
        Integer size = context.getIntAttribute("size");
        boolean readwrite = !context.getBooleanAttribute("readOnly", false);
        boolean blocking = context.getBooleanAttribute("blocking", false);
        // <4> 获得 Properties 属性
        Properties props = context.getChildrenAsProperties();
        // <5> 创建 Cache 对象
        builderAssistant.useNewCache(typeClass, evictionClass, flushInterval,
            size, readwrite, blocking, props);
    }
}

```

`#resultMapElements(List<XNode> list)` 方法，解析 `<resultMap />` 节点们。代码如下：

```
// 解析 <resultMap /> 节点们
private void resultMapElements(List<XNode> list) throws Exception {
    // 遍历 <resultMap /> 节点们
    for (XNode resultMapNode : list) {
        try {
            // 处理单个 <resultMap /> 节点
            resultMapElement(resultMapNode);
        } catch (IncompleteElementException e) {
            // ignore, it will be retried
        }
    }
}

// 解析 <resultMap /> 节点
private ResultMap resultMapElement(XNode resultMapNode) throws Exception {
    return resultMapElement(resultMapNode, Collections.
        <ResultMapping>emptyList());
}

// 解析 <resultMap /> 节点
private ResultMap resultMapElement(XNode resultMapNode, List<ResultMapping>
    additionalResultMappings) throws Exception {
    ErrorContext.instance().activity("processing " +
        resultMapNode.getValueBasedIdentifier());
    // <1> 获得 id 属性
    String id = resultMapNode.getStringAttribute("id",
        resultMapNode.getValueBasedIdentifier());
    // <1> 获得 type 属性
    String type = resultMapNode.getStringAttribute("type",
        resultMapNode.getStringAttribute("ofType",
            resultMapNode.getStringAttribute("resultType",
                resultMapNode.getStringAttribute("javaType"))));
    // <1> 获得 extends 属性
    String extend = resultMapNode.getStringAttribute("extends");
    // <1> 获得 autoMapping 属性
    Boolean autoMapping = resultMapNode.getBooleanAttribute("autoMapping");
    // <1> 解析 type 对应的类
    Class<?> typeClass = resolveClass(type);
    Discriminator discriminator = null;
    // <2> 创建 ResultMapping 集合
    List<ResultMapping> resultMappings = new ArrayList<>();
    resultMappings.addAll(additionalResultMappings);
    // <2> 遍历 <resultMap /> 的子节点
    List<XNode> resultChildren = resultMapNode.getChildren();
    for (XNode resultChild : resultChildren) {
        // <2.1> 处理 <constructor /> 节点
```

```

        if ("constructor".equals(resultChild.getName())) {
            processConstructorElement(resultChild, typeClass, resultMappings);
            // <2.2> 处理 <discriminator /> 节点
        } else if ("discriminator".equals(resultChild.getName())) {
            discriminator = processDiscriminatorElement(resultChild, typeClass,
resultMappings);
            // <2.3> 处理其它节点
        } else {
            List<ResultFlag> flags = new ArrayList<>();
            if ("id".equals(resultChild.getName())) {
                flags.add(ResultFlag.ID);
            }
            resultMappings.add(buildResultMappingFromContext(resultChild,
typeClass, flags));
        }
    }
    // <3> 创建 ResultMapResolver 对象, 执行解析
    ResultMapResolver resultMapResolver = new
ResultMapResolver(builderAssistant, id, typeClass, extend, discriminator,
resultMappings, autoMapping);
    try {
        return resultMapResolver.resolve();
    } catch (IncompleteElementException e) {
        // <4> 解析失败, 添加到 configuration 中
        configuration.addIncompleteResultMap(resultMapResolver);
        throw e;
    }
}
}

```

#buildStatementFromContext(List<XNode> list) 方法, 解析 <select />、<insert />、<update />、<delete /> 节点们。代码如下:

```

private void buildStatementFromContext(List<XNode> list) {
    if (configuration.getDatabaseId() != null) {
        buildStatementFromContext(list, configuration.getDatabaseId());
    }
    buildStatementFromContext(list, null);
}

private void buildStatementFromContext(List<XNode> list, String
requiredDatabaseId) {
    // <1> 遍历 <select /> <insert /> <update /> <delete /> 节点们
    for (XNode context : list) {
        // <1> 创建 XMLStatementBuilder 对象, 执行解析 =====>
    }
}

```

```

        final XMLStatementBuilder statementParser = new
XMLStatementBuilder(configuration, builderAssistant, context,
requiredDatabaseId);
        try {
            statementParser.parseStatementNode();
        } catch (IncompleteElementException e) {
            // <2> 解析失败, 添加到 configuration 中
            configuration.addIncompleteStatement(statementParser);
        }
    }
}

```

2) XMLStatementBuilder

概述：【解析mapper.xml文件中的crud节点】

`org.apache.ibatis.builder.xml.XMLStatementBuilder` , 继承 `BaseBuilder` 抽象类, Statement XML 配置构建器, 主要负责解析 Statement 配置, 即 `<select />`、`<insert />`、`<update />`、`<delete />` 标签。

开启SQL节点解析, 源码开始解析的入口`parseStatementNode`

`#parseStatementNode()` 方法, 执行 Statement 解析。代码如下:

```

public void parseStatementNode() {
    // <1> 获得 id 属性, 编号。
    String id = context.getStringAttribute("id");
    // <2> 获得 databaseId , 判断 databaseId 是否匹配
    String databaseId = context.getStringAttribute("databaseId");
    if (!databaseIdMatchesCurrent(id, databaseId, this.requiredDatabaseId)) {
        return;
    }

    // <3> 获得各种属性
    Integer fetchSize = context.getIntAttribute("fetchSize");
    Integer timeout = context.getIntAttribute("timeout");
    String parameterMap = context.getStringAttribute("parameterMap");
    String parameterType = context.getStringAttribute("parameterType");
    Class<?> parameterTypeClass = resolveClass(parameterType);
    String resultMap = context.getStringAttribute("resultMap");
    String resultType = context.getStringAttribute("resultType");
    String lang = context.getStringAttribute("lang");

    // <4> 获得 lang 对应的 LanguageDriver 对象
}

```

```

LanguageDriver langDriver = getLanguageDriver(lang);

// <5> 获得 resultType 对应的类
Class<?> resultTypeClass = resolveClass(resultType);
// <6> 获得 resultSet 对应的枚举值
String resultSetType = context.getStringAttribute("resultSetType");
ResultSetType resultSetTypeEnum = resolveResultSetType(resultSetType);
// <7> 获得 statementType 对应的枚举值
StatementType statementType =
StatementType.valueOf(context.getStringAttribute("statementType",
StatementType.PREPARED.toString()));

// <8> 获得 SQL 对应的 sqlCommandType 枚举值
String nodeName = context.getNode().getNodeName();
SqlCommandType sqlCommandType =
SqlCommandType.valueOf(nodeName.toUpperCase(Locale.ENGLISH));
// <9> 获得各种属性
boolean isSelect = sqlCommandType == SqlCommandType.SELECT;
boolean flushCache = context.getBooleanAttribute("flushCache", !isSelect);
boolean useCache = context.getBooleanAttribute("useCache", isSelect);
boolean resultOrdered = context.getBooleanAttribute("resultOrdered", false);

// Include Fragments before parsing
// <10> 创建 XMLIncludeTransformer 对象, 并替换 <include /> 标签相关的内容
XMLIncludeTransformer includeParser = new
XMLIncludeTransformer(configuration, builderAssistant);
includeParser.applyIncludes(context.getNode());

// Parse selectKey after includes and remove them.
// <11> 解析 <selectKey /> 标签
processSelectKeyNodes(id, parameterTypeClass, langDriver);

// Parse the SQL (pre: <selectKey> and <include> were parsed and removed)
// <12> 创建 sqlSource
SqlSource sqlSource = langDriver.createSqlSource(configuration, context,
parameterTypeClass);
// <13> 获得 keyGenerator 对象
String resultSets = context.getStringAttribute("resultSets");
String keyProperty = context.getStringAttribute("keyProperty");
String keyColumn = context.getStringAttribute("keyColumn");
KeyGenerator keyGenerator;
// <13.1> 优先, 从 configuration 中获得 KeyGenerator 对象。如果存在, 意味着是
<selectKey /> 标签配置的
String keyStatementId = id + SelectKeyGenerator.SELECT_KEY_SUFFIX;
keyStatementId = builderAssistant.applyCurrentNamespace(keyStatementId,
true);
if (configuration.hasKeyGenerator(keyStatementId)) {
    keyGenerator = configuration.getKeyGenerator(keyStatementId);
}

```

```

// <13.2> 其次, 根据标签属性的情况, 判断是否使用对应的 Jdbc3KeyGenerator 或者
NoKeyGenerator 对象
} else {
    keyGenerator = context.getBooleanAttribute("useGeneratedKeys", // 优先, 基
于 useGeneratedKeys 属性判断
        configuration.isUseGeneratedKeys() &&
        SqlCommandType.INSERT.equals(sqlCommandType)) // 其次, 基于全局的 useGeneratedKeys
配置 + 是否为插入语句类型
        ? Jdbc3KeyGenerator.INSTANCE : NoKeyGenerator.INSTANCE;
}

// 创建 MappedStatement 对象
builderAssistant.addMappedStatement(id, sqlSource, statementType,
sqlCommandType,
    fetchSize, timeout, parameterMap, parameterTypeClass, resultMap,
resultTypeClass,
    resultSetTypeEnum, flushCache, useCache, resultOrdered,
    keyGenerator, keyProperty, keyColumn, databaseId, langDriver,
resultSets);
}

```

#databaseIdMatchesCurrent(String id, String databaseId, String requiredDatabaseId) 方法, 判断 databaseId 是否匹配。代码如下:

```

private boolean databaseIdMatchesCurrent(String id, String databaseId, String
requiredDatabaseId) {
    // 如果不匹配, 则返回 false
    if (requiredDatabaseId != null) {
        return requiredDatabaseId.equals(databaseId);
    }
    // 如果未设置 requiredDatabaseId , 但是 databaseId 存在, 说明还是不匹配, 则返回
false
    if (databaseId != null) {
        return false;
    }
    // 判断是否已经存在
    id = builderAssistant.applyCurrentNamespace(id, false);
    if (!this.configuration.hasStatement(id, false)) {
        return true;
    }
    MappedStatement previous = this.configuration.getMappedStatement(id, false);
    // 若存在, 则判断原有的sqlFragment是否databaseId 为空。因为, 当前databaseId为空, 这样
两者才能匹配。
    return previous.getDatabaseId() == null;
}

```

```
}
```

3) XMLIncludeTransformer

概述：【解析节点】

`org.apache.ibatis.builder.xml.XMLIncludeTransformer`，XML `<include />` 标签的转换器；

解析节点，该过程会将其替换成节点中定义的SQL片段，并将其中的“\${xxx}”占位符替换为真实的参数

`#applyIncludes(Node source)` 方法，将 `<include />` 标签，替换成引用的 `<sql />`。代码如下：

```
public void applyIncludes(Node source) {
    // <1> 创建 variablesContext，并将 configurationVariables 添加到其中
    Properties variablesContext = new Properties();
    Properties configurationVariables = configuration.getVariables();
    if (configurationVariables != null) {
        variablesContext.putAll(configurationVariables);
    }
    // <2> 处理 <include />
    applyIncludes(source, variablesContext, false);
}
```

`#applyIncludes(Node source, final Properties variablesContext, boolean included)` 方法，使用递归的方式，将 `<include />` 标签，替换成引用的 `<sql />`。代码如下：

```
private void applyIncludes(Node source, final Properties variablesContext,
    boolean included) {
    // <1> 如果是 <include /> 标签
    if (source.getNodeName().equals("include")) {
        // <1.1> 获得 <sql /> 对应的节点
        Node toInclude = findSqlFragment(getStringAttribute(source, "refid"),
            variablesContext);
        // <1.2> 获得包含 <include /> 标签内的属性
        Properties toIncludeContext = getVariablesContext(source,
            variablesContext);
```

的节点

// <1.3> 递归调用 #applyIncludes(...) 方法, 继续替换。注意, 此处是 <sql /> 对应的节点

```
applyIncludes(toInclude, toIncludeContext, true);
if (toInclude.getOwnerDocument() != source.getOwnerDocument()) {
    toInclude = source.getOwnerDocument().importNode(toInclude, true);
}
// <1.4> 将 <include /> 节点替换成 <sql /> 节点
source.getParentNode().replaceChild(toInclude, source);
// <1.4> 将 <sql /> 子节点添加到 <sql /> 节点前面
while (toInclude.hasChildNodes()) {
    toInclude.getParentNode().insertBefore(toInclude.getFirstChild(),
toInclude); // 当子节点添加到其它节点下面后, 这个子节点会不见了, 相当于是“移动操作”
}
// <1.4> 移除 <include /> 标签自身
toInclude.getParentNode().removeChild(toInclude);
// <2> 如果节点类型为 Node.ELEMENT_NODE
} else if (source.getNodeType() == Node.ELEMENT_NODE) {
    // <2.1> 如果在处理 <include /> 标签中, 则替换其上的属性
    if (included && !variablesContext.isEmpty()) {
        // replace variables in attribute values
        NamedNodeMap attributes = source.getAttributes();
        for (int i = 0; i < attributes.getLength(); i++) {
            Node attr = attributes.item(i);
            attr.setNodeValue(PropertyParser.parse(attr.getNodeValue(),
variablesContext));
        }
    }
    // <2.2> 遍历子节点, 递归调用 #applyIncludes(...) 方法, 继续替换
    NodeList children = source.getChildNodes();
    for (int i = 0; i < children.getLength(); i++) {
        applyIncludes(children.item(i), variablesContext, included);
    }
    // <3> 如果在处理 <include /> 标签中, 并且节点类型为 Node.TEXT_NODE, 并且变量非空
    // 则进行变量的替换, 并修改原节点 source
} else if (included && source.getNodeType() == Node.TEXT_NODE
    && !variablesContext.isEmpty()) {
    // replace variables in text node
    source.setNodeValue(PropertyParser.parse(source.getNodeValue(),
variablesContext));
}
}
```

#findSqlFragment(String refid, Properties variables) 方法, 获得对应的 <sql /> 节点。代码如下:

```

private Node findSqlFragment(String refid, Properties variables) {
    // 因为 refid 可能是动态变量, 所以进行替换
    refid = PropertyParser.parse(refid, variables);
    // 获得完整的 refid , 格式为 "${namespace}.${refid}"
    refid = builderAssistant.applyCurrentNamespace(refid, true);
    try {
        // 获得对应的 <sql /> 节点
        XNode nodeToInclude = configuration.getSqlFragments().get(refid);
        // 获得 Node 节点, 进行克隆
        return nodeToInclude.getNode().cloneNode(true);
    } catch (IllegalArgumentException e) {
        throw new IncompleteElementException("Could not find SQL statement to include with refid '" + refid + "'", e);
    }
}

private String getStringAttribute(Node node, String name) {
    return node.getAttributes().getNamedItem(name).getNodeValue();
}

```

4) SqlSourceBuilder创建 SqlSource

`org.apache.ibatis.mapping.SqlSource` , SQL 来源接口。

代表从 Mapper XML 或方法注解上, 读取的一条 SQL 内容。代码如下:

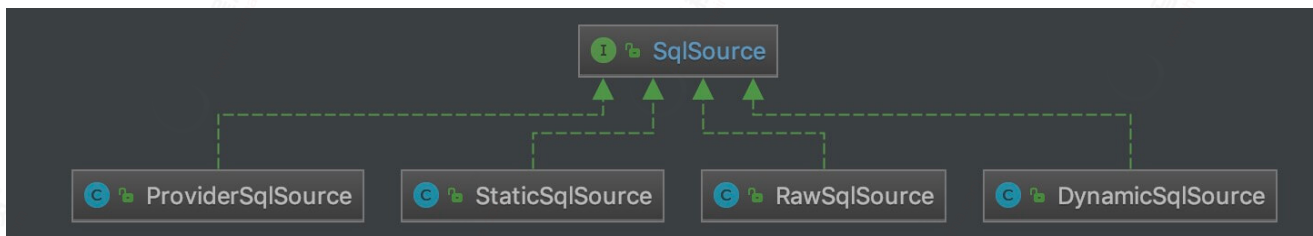
```

public interface SqlSource {

    /**
     * 根据传入的参数对象, 返回 BoundSql 对象
     *
     * @param parameterObject 参数对象
     * @return BoundSql 对象
     */
    BoundSql getBoundSql(Object parameterObject);

}

```



- RawSqlSource 负责处理静态 SQL 语句，它们最终会把处理后的 SQL 封装 StaticSqlSource 进行返回。

使用 `#{}` 表达式，或者不使用任何表达式的情况，所以它是**静态**的，仅需要在构造方法中，直接生成对应的 SQL 。

- StaticSqlSource处理包含的 SQL 可能含有“?” 占位符，可以被数据库直接执行。
- DynamicSqlSource 负责处理动态 SQL 语句。

使用了 OGNL 表达式，或者使用了 `${}` 表达式的 SQL ，所以它是**动态**的，需要在每次执行 `#getBoundSql(Object parameterObject)` 方法，根据参数，生成对应的 SQL 。

- ProviderSqlSource 实现 SqlSource 接口，基于方法上的 `@Providerxxx` 注解的 SqlSource 实现类。

`org.apache.ibatis.builder.SqlSourceBuilder` ， SqlSource 构建器。

负责将 SQL 语句中的 `#{}` 替换成相应的 `?` 占位符，并获取该 `?` 占位符对应的 `ParameterMapping` 对象。

```
/**
 * 执行解析原始 SQL ，成为 SqlSource 对象
 *
 * @param originalSql 原始 SQL
 * @param parameterType 参数类型
 * @return SqlSource 对象
 */
public SqlSource parse(String originalSql, Class<?> parameterType, Map<String,
Object> additionalParameters) {
    // <1> 创建 ParameterMappingTokenHandler 对象
    ParameterMappingTokenHandler handler = new
ParameterMappingTokenHandler(configuration, parameterType,
additionalParameters);
    // <2> 创建 GenericTokenParser 对象
    GenericTokenParser parser = new GenericTokenParser("#{", "}", handler);
    // <3> 执行解析
    String sql = parser.parse(originalSql);
    // <4> 创建 StaticSqlSource 对象
    return new StaticSqlSource(configuration, sql,
handler.getParameterMappings());
}
```

```
/**
    注意: ParameterMappingTokenHandler 是 SqlSourceBuilder 的内部私有静态类。
 */
@Override
public String handleToken(String content) {
    // <1> 构建 ParameterMapping 对象, 并添加到 parameterMappings 中
    parameterMappings.add(buildParameterMapping(content));
    // <2> 返回 ? 占位符
    return "?";
}
```

四、核心执行器executor详解

4.1 jdbc回顾

1) 回顾JDBC执行过程

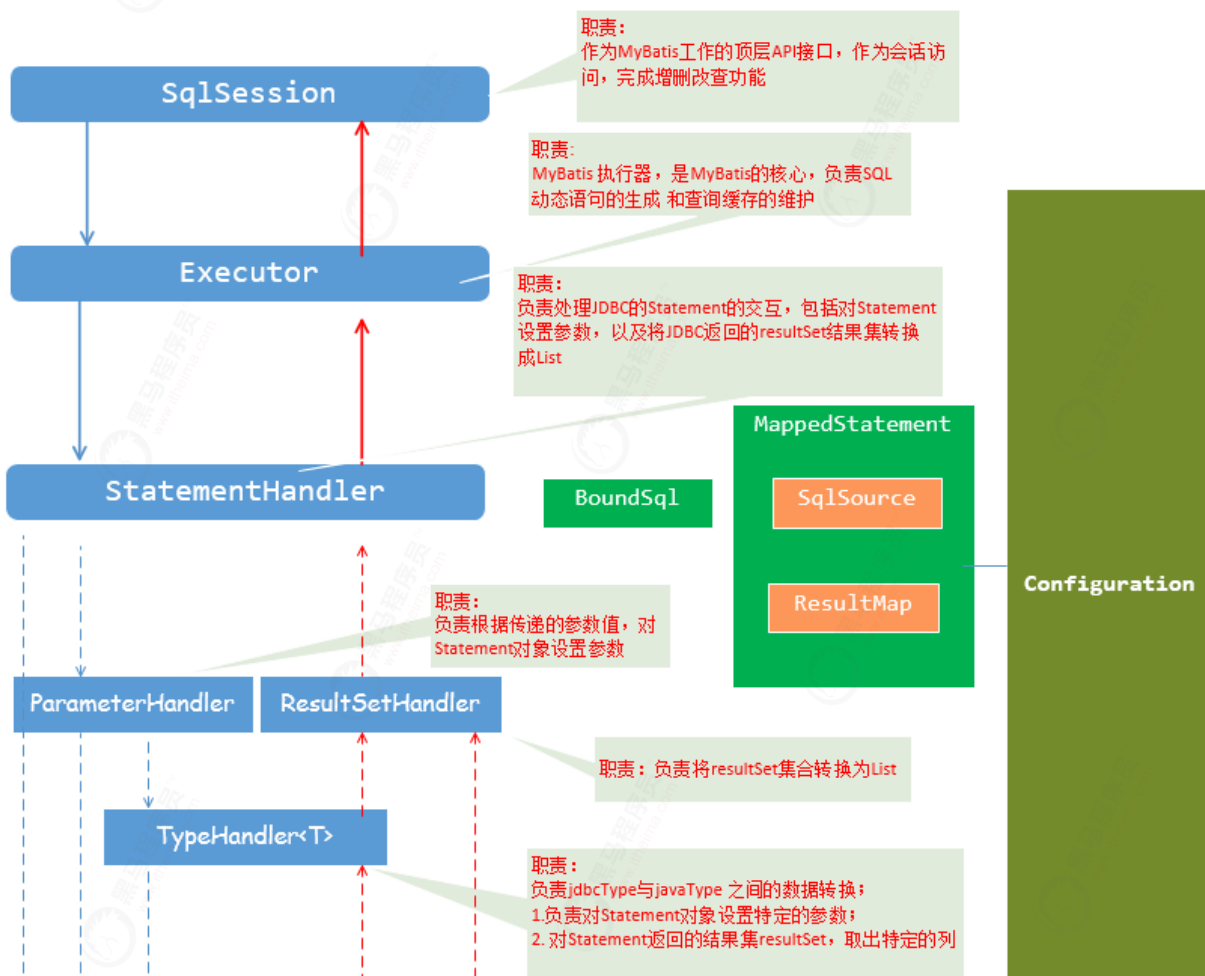
添加jar包 -> 获得连接 -> 预编译SQL -> 执行SQL, 读取结果 -> 关闭事务

```
public static void main(String[] args) throws Exception {
    // 1、注册驱动
    DriverManager.registerDriver(new com.mysql.jdbc.Driver());
    // 2、建立连接
    Connection con =
    DriverManager.getConnection("jdbc:mysql://localhost:3306/test?
    useUnicode=true&characterEncoding=utf-8&serverTimezone=GMT", "root", "root");
    // 3、编写sql, 进行预编译
    String sql = " select * from tb_brand;";
    PreparedStatement ps = con.prepareStatement(sql);
    // 4、执行查询, 得到结果集
    ResultSet rs = ps.executeQuery();
    while (rs.next()) {
        int bid = rs.getInt("bid");
        String bname = rs.getString("bname");
    }
}
```

```
        System.out.println("====> bid=" + bid + "\tbyname=" + bname);
    }
    //5、关闭事务
    rs.close();
    ps.close();
    con.close();
}
```

2) MyBatis对JDBC封装的执行过程

MyBatis层次结构



JDBC

ResultSet

返回结果集

Statement

PreparedStatement

SimpleStatement

CallableStatement

4.2 MyBatis的核心执行组件介绍

在Mybatis中, SqlSession对数据库的操作, 将委托给执行器Executor来完成都将委托给执行器Executor来完成;

Mybatis执行过程, 主要的执行模块是: SqlSession->Executor->StatementHandler->数据库

四个核心组件:

- ① 动态代理 MapperProxy
- ② SQL会话 SqlSession
- ③ 执行器 Executor
- ④ JDBC处理器 StatementHandler

1) SqlSession

SqlSession采用了门面模式方便来让我们使用。他不能跨线程使用, 一级缓存生命周期和它一致;

基本功能: 增删改查基本的操作功能;

辅助功能: 提交、关闭会话;

门面模式: 提供一个统一的门面接口API, 使得系统更加容易使用。

2) Executor

基本功能: 改、查、**维护缓存**;

辅助功能: 提交、关闭执行器、**批处理刷新**;

Executor 主要负责维护一级缓存和二级缓存, 并提供事务管理的相关操作, 它会将数据库相关操作委托给 StatementHandler完成。

3) StatementHandler

经过执行器处理后交给了StatementHandler (声明处理器);

主要作用就是: 参数处理、结果处理;

StatementHandler 首先通过 **ParameterHandler** 完成 SQL 语句的实参绑定, 然后通过 `java.sql.Statement` 对象执行 SQL 语句并得到结果集, 最后通过 **ResultSetHandler** 完成结果集的映射, 得到结果对象并返回。

4.3 Executor执行器分析

1) JDBC中的执行器

JDBC有三种执行器分别是**Statement**（简单执行器）、**PreparedStatement**（预处理执行器）、**CallableStatement**（存储过程执行器）

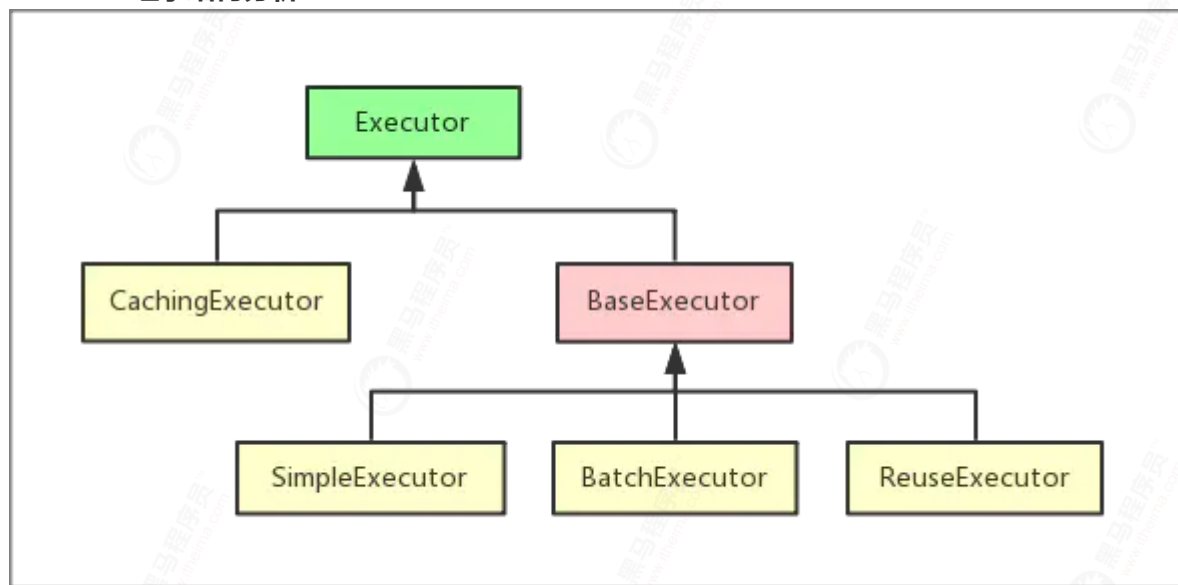
Statement：基本功能：执行静态SQL

PreparedStatement：设置预编译，防止SQL注入

CallableStatement：设置出参、读取参数（用于执行存储过程）

2) Mybatis执行器

Executor继承结构分析



Mybatis给我们提供了三种执行器，分别是：

- **SimpleExecutor**(简单执行器)、
- **ResuseExecutor**(可重用执行器)、
- **BathExecutor**(批处理执行器)

这三个执行器继承了一个**BaseExecutor**（基础执行器），而这个基础执行器实现了Executor接口，其中简单执行器是默认的执行器。

其实还有一种执行器**CachingExecutor**(二级缓存执行器)你开启二级缓存则会实例化它，在BaseExecutor的基础上，实现**二级缓存**功能。（注意: BaseExecutor 的本地缓存，就是一级缓存。）

(1) Executor接口

`org.apache.ibatis.executor.Executor` , 执行器接口。

主要定义了以下内容:

- 读和写操作相关的方法
- 事务相关的方法
- 缓存相关的方法
- 设置延迟加载的方法
- 设置包装的 Executor 对象的方法

```
public interface Executor {

    // 空 ResultHandler 对象的枚举
    ResultHandler NO_RESULT_HANDLER = null;

    // 更新 or 插入 or 删除, 由传入的 MappedStatement 的 SQL 所决定
    int update(MappedStatement ms, Object parameter) throws SQLException;

    // 查询, 带 ResultHandler + CacheKey + BoundSql
    <E> List<E> query(MappedStatement ms, Object parameter, RowBounds rowBounds,
        ResultHandler resultHandler, CacheKey cacheKey, BoundSql boundSql) throws
        SQLException;

    // 查询, 带 ResultHandler
    <E> List<E> query(MappedStatement ms, Object parameter, RowBounds rowBounds,
        ResultHandler resultHandler) throws SQLException;

    // 查询, 返回值为 Cursor
    <E> Cursor<E> queryCursor(MappedStatement ms, Object parameter, RowBounds
        rowBounds) throws SQLException;

    // 刷入批处理语句
    List<BatchResult> flushStatements() throws SQLException;

    // 提交事务
    void commit(boolean required) throws SQLException;
    // 回滚事务
    void rollback(boolean required) throws SQLException;

    // 创建 CacheKey 对象
    CacheKey createCacheKey(MappedStatement ms, Object parameterObject,
        RowBounds rowBounds, BoundSql boundSql);

    // 判断是否缓存
    boolean isCached(MappedStatement ms, CacheKey key);
    // 清除本地缓存
    void clearLocalCache();
}
```

```

// 延迟加载
void deferLoad(MappedStatement ms, MetaObject resultObject, String property,
CacheKey key, Class<?> targetType);

// 获得事务
Transaction getTransaction();
// 关闭事务
void close(boolean forceRollback);
// 判断事务是否关闭
boolean isClosed();

// 设置包装的 Executor 对象
void setExecutorWrapper(Executor executor);

}

```

(2) BaseExecutor (基础执行器)

基础执行器：维护一级缓存，是simple、reuse、batch这三个执行器的父类。主要逻辑是维护缓存，其他实现交给子类。`org.apache.ibatis.executor.BaseExecutor` 实现 `Executor` 接口，提供骨架方法，从而使子类只要实现指定的几个抽象方法即可。

```

/**
 * 事务对象
 */
protected Transaction transaction;

/**
 * 包装的 Executor 对象
 */
protected Executor wrapper;

/**
 * DeferredLoad( 延迟加载 ) 队列
 */
protected ConcurrentLinkedQueue<DeferredLoad> deferredLoads;

/**
 * 本地缓存，即一级缓存
 */
protected PerpetualCache localCache;

/**
 * 本地输出类型的参数的缓存
 */
protected PerpetualCache localOutputParameterCache;
protected Configuration configuration;

```

```

/**
 * 记录嵌套查询的层级
 */
protected int queryStack;
/**
 * 是否关闭
 */
private boolean closed;

// *****

/**
 * clearLocalCache() 方法, 清理一级 (本地) 缓存
 */
@Override
public void clearLocalCache() {
    if (!closed) {
        // 清理 localCache
        localCache.clear();
        // 清理 localOutputParameterCache
        localOutputParameterCache.clear();
    }
}

// createCacheKey(MappedStatement ms, Object parameterObject, RowBounds
rowBounds, BoundSql boundSql) 方法, 创建 CacheKey 对象

// isCached(MappedStatement ms, CacheKey key) 方法, 判断一级缓存是否存在

// query方法
@Override
public <E> List<E> query(MappedStatement ms, Object parameter, RowBounds
rowBounds, ResultHandler resultHandler) throws SQLException {
    // <1> 获得 BoundSql 对象
    BoundSql boundSql = ms.getBoundSql(parameter);
    // <2> 创建 CacheKey 对象
    CacheKey key = createCacheKey(ms, parameter, rowBounds, boundSql);
    // <3> 查询
    return query(ms, parameter, rowBounds, resultHandler, key, boundSql);
}

// update方法

```

```

@Override
public int update(MappedStatement ms, Object parameter) throws SQLException {
    ErrorContext.instance().resource(ms.getResource()).activity("executing an
update").object(ms.getId());
    // <1> 已经关闭, 则抛出 ExecutorException 异常
    if (closed) {
        throw new ExecutorException("Executor was closed.");
    }
    // <2> 清空本地缓存
    clearLocalCache();
    // <3> 执行写操作
    return doUpdate(ms, parameter);
}

```

(3) SimpleExecutor (简单执行器)

- 每次读或写操作都会创建一个新的预处理器 (PrepareStatement) ;
- 每次执行的SQL都会进行一次预编译;
- 执行完成后, 关闭该 Statement 对象。

`org.apache.ibatis.executor.SimpleExecutor` , 继承 `BaseExecutor` 抽象类, 简单的 `Executor` 实现类。

```

// 查询

@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();
        // <1> 创建 StatementHandler 对象
        StatementHandler handler = configuration.newStatementHandler(wrapper,
ms, parameter, rowBounds, resultHandler, boundSql);
        // <2> 初始化 StatementHandler 对象
        stmt = prepareStatement(handler, ms.getStatementLog());
        // <3> 执行 StatementHandler , 进行读操作
        return handler.query(stmt, resultHandler);
    } finally {

```

```

        // <4> 关闭 StatementHandler 对象
        closeStatement(stmt);
    }
}

private Statement prepareStatement(StatementHandler handler, Log statementLog)
throws SQLException {
    Statement stmt;
    // <2.1> 获得 Connection 对象
    Connection connection = getConnection(statementLog);
    // <2.2> 创建 Statement 或 PreparedStatement 对象
    stmt = handler.prepare(connection, transaction.getTimeout());
    // <2.3> 设置 SQL 上的参数, 例如 PreparedStatement 对象上的占位符
    handler.parameterize(stmt);
    return stmt;
}

// 更新
@Override
public int doUpdate(MappedStatement ms, Object parameter) throws SQLException {
    Statement stmt = null;
    try {
        Configuration configuration = ms.getConfiguration();
        // 创建 StatementHandler 对象
        StatementHandler handler = configuration.newStatementHandler(this, ms,
parameter, RowBounds.DEFAULT, null, null);
        // 初始化 StatementHandler 对象
        stmt = prepareStatement(handler, ms.getStatementLog());
        // <3> 执行 StatementHandler, 进行写操作
        return handler.update(stmt);
    } finally {
        // 关闭 StatementHandler 对象
        closeStatement(stmt);
    }
}
}

```

(4) ReuseExecutor (可重用执行器)

- 每次开始读或写操作，以sql作为key，查找Statement对象优先从缓存中获取对应的 Statement 对象。如果不存在，才进行创建。（只要是相同的SQL只会进行一次预处理）
- 执行完成后，不关闭该 Statement 对象，而是放置于Map<String, Statement>内，供下一次使用。
- 其它的，和 SimpleExecutor 是一致的。

原理：

```
//可重用的执行器内部用了一个map，用来缓存SQL语句对应的Statement对象
//key是我们的SQL语句，value是我们的Statement对象
private final Map<String, Statement> statementMap = new HashMap<String,
Statement>();
```

org.apache.ibatis.executor.ReuseExecutor，继承 BaseExecutor 抽象类，可重用的 Executor 实现类。

```
// doQuery
@Override
public <E> List<E> doQuery(MappedStatement ms, Object parameter, RowBounds
rowBounds, ResultHandler resultHandler, BoundSql boundSql) throws SQLException {
    Configuration configuration = ms.getConfiguration();
    // 创建 StatementHandler 对象
    StatementHandler handler = configuration.newStatementHandler(wrapper, ms,
parameter, rowBounds, resultHandler, boundSql);
    // 初始化 StatementHandler 对象
    Statement stmt = prepareStatement(handler, ms.getStatementLog());
    // 执行 StatementHandler，进行读操作
    return handler.query(stmt, resultHandler);
}

// doUpdate
@Override
public int doUpdate(MappedStatement ms, Object parameter) throws SQLException {
    Configuration configuration = ms.getConfiguration();
    // 创建 StatementHandler 对象
    StatementHandler handler = configuration.newStatementHandler(this, ms,
parameter, RowBounds.DEFAULT, null, null);
    // 初始化 StatementHandler 对象
    Statement stmt = prepareStatement(handler, ms.getStatementLog());
    // 执行 StatementHandler，进行写操作
    return handler.update(stmt);
}
```

```

}

private Statement preparedStatement(StatementHandler handler, Log statementLog)
throws SQLException {
    Statement stmt;
    BoundSql boundsSql = handler.getBoundSql();
    String sql = boundsSql.getSql();
    // 存在
    if (hasStatementFor(sql)) {
        // <1.1> 从缓存中获得 Statement 或 PreparedStatement 对象
        stmt = getStatement(sql);
        // <1.2> 设置事务超时时间
        applyTransactionTimeout(stmt);
    } // 不存在
    else {
        // <2.1> 获得 Connection 对象
        Connection connection = getConnection(statementLog);
        // <2.2> 创建 Statement 或 PreparedStatement 对象
        stmt = handler.prepare(connection, transaction.getTimeout());
        // <2.3> 添加到缓存中
        putStatement(sql, stmt);
    }
    // <2> 设置 SQL 上的参数, 例如 PreparedStatement 对象上的占位符
    handler.parameterize(stmt);
    return stmt;
}

// 判断是否存在对应的 Statement 对象
private boolean hasStatementFor(String sql) {
    try {
        return statementMap.keySet().contains(sql) &&
!statementMap.get(sql).getConnection().isClosed();
    } catch (SQLException e) {
        return false;
    }
}
}

```

注意区别



注意:

- ReuseExecutor 考虑到重用性, 但是 Statement 最终还是需要地方关闭。答案就在 `#doFlushStatements(boolean isRollback)` 方法中。而 BaseExecutor 在关闭 `#close()` 方法中, 最终也会调用该方法, 从而完成关闭缓存的 Statement 对象们
- BaseExecutor 在提交或者回滚事务方法中, 最终也会调用该方法, 也能完成关闭缓存的 Statement 对象们。

(5) BatchExecutor (批处理执行器)

- 批处理只对增删改SQL有效 (没有select, JDBC批处理不支持select) ;
- 将所有增删改sql都添加到批处理中 (addBatch()), 等待统一执行 (executeBatch()), 它缓存了多个Statement对象, 每个Statement对象都是addBatch()完毕后, 等待逐一执行 executeBatch()批处理的;
- 执行sql需要满足三个条件才能使用同一个Statement(使用同一个Statement是为了压缩体积、减少SQL预处理)
 - 1.sql相同
 - 2.同一个MappedStatement (sql标签的配置都在这里面)
 - 3.执行的顺序必须是连续的

`org.apache.ibatis.executor.BatchExecutor` , 继承 BaseExecutor 抽象类, 批量执行的 Executor 实现类。

```
public class BatchExecutor extends BaseExecutor {
```

```

/**
 * Statement 数组
 */
private final List<Statement> statementList = new ArrayList<>();
/**
 * BatchResult 数组
 *
 * 每一个 BatchResult 元素, 对应一个 {@link #statementList} 的 Statement 元素
 */
private final List<BatchResult> batchResultList = new ArrayList<>();
/**
 * 当前 SQL
 */
private String currentSql;
/**
 * 当前 MappedStatement 对象
 */
private MappedStatement currentStatement;

//douupdate
@Override
public int douupdate(MappedStatement ms, Object parameterObject) throws
SQLException {
    final Configuration configuration = ms.getConfiguration();
    // <1> 创建 StatementHandler 对象
    final StatementHandler handler = configuration.newStatementHandler(this,
ms, parameterObject, RowBounds.DEFAULT, null, null);
    final BoundSql boundSql = handler.getBoundSql();
    final String sql = boundSql.getSql();
    final Statement stmt;
    // <2> 如果匹配最后一次 currentSql 和 currentStatement , 则聚合到 BatchResult
中
    if (sql.equals(currentSql) && ms.equals(currentStatement)) {
        // <2.1> 获得最后一次的 Statement 对象
        int last = statementList.size() - 1;
        stmt = statementList.get(last);
        // <2.2> 设置事务超时时间
        applyTransactionTimeout(stmt);
        // <2.3> 设置 SQL 上的参数, 例如 PreparedStatement 对象上的占位符
        handler.parameterize(stmt);
        // <2.4> 获得最后一次的 BatchResult 对象, 并添加参数到其中
        BatchResult batchResult = batchResultList.get(last);
        batchResult.addParameterObject(parameterObject);
        // <3> 如果不匹配最后一次 currentSql 和 currentStatement , 则新建 BatchResult
对象
    } else {

```

```

// <3.1> 获得 Connection
Connection connection = getConnection(ms.getStatementLog());
// <3.2> 创建 Statement 或 PreparedStatement 对象
stmt = handler.prepare(connection, transaction.getTimeout());
// <3.3> 设置 SQL 上的参数, 例如 PreparedStatement 对象上的占位符
handler.parameterize(stmt);    //fix Issues 322
// <3.4> 重新设置 currentSql 和 currentStatement
currentSql = sql;
currentStatement = ms;
// <3.5> 添加 Statement 到 statementList 中
statementList.add(stmt);
// <3.6> 创建 BatchResult 对象, 并添加到 batchResultList 中
batchResultList.add(new BatchResult(ms, sql, parameterObject));
}
// <4> 批处理
handler.batch(stmt);
return BATCH_UPDATE_RETURN_VALUE;
}

```

```

//doQuery
//和 SimpleExecutor 的该方法, 逻辑差不多。区别在发生查询之前, 先调用
#flushStatements() 方法, 刷入批处理语句。
@Override
public <E> List<E> doQuery(MappedStatement ms, Object parameterObject,
RowBounds rowBounds, ResultHandler resultHandler, BoundSql boundSql)
throws SQLException {
Statement stmt = null;
try {
// *****刷入批处理语句
flushStatements();

Configuration configuration = ms.getConfiguration();
// 创建 StatementHandler 对象
StatementHandler handler =
configuration.newStatementHandler(wrapper, ms, parameterObject, rowBounds,
resultHandler, boundSql);
// 获得 Connection 对象
Connection connection = getConnection(ms.getStatementLog());
// 创建 Statement 或 PreparedStatement 对象
stmt = handler.prepare(connection, transaction.getTimeout());
// 设置 SQL 上的参数, 例如 PreparedStatement 对象上的占位符
handler.parameterize(stmt);
// 执行 StatementHandler , 进行读操作
return handler.query(stmt, resultHandler);
} finally {
// 关闭 StatementHandler 对象

```

```
        closeStatement(stmt);  
    }  
}  
  
}
```

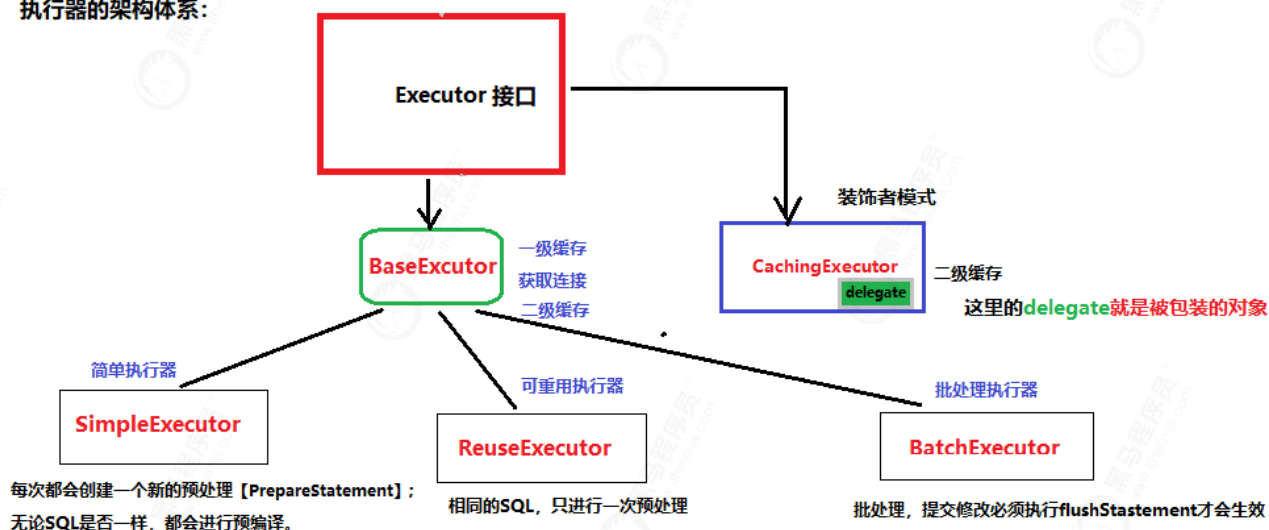
- doUpdate代码上 `if (sql.equals(currentSql) && ms.equals(currentStatement))` 可以看出上一个添加的是否是这个sql (`currentSql`) 并且是同一个MappedStatement `currentStatement` (映射语句);
- 满足条件就将参数放到当前这个BatchResult对象中的参数。属性是 (`parameterObject`)
【`batchResult.addParameterObject(parameterObject);`】
- 不满足条件则获取一个Statement实例 再实例化BatchResult对象。最后放到list中去, 再给 `current`和MappedStatement `currentStatement`赋值
- 批处理提交必须执行flushStatements才会生效(会将一个个的Statement提交)可以减少与数据库交互次数

(6) CachingExecutor (二级缓存执行器)

CachingExecutor二级缓存执行器, 属于缓存章节内容, 在后面缓存章节详细讲解;

3) Executor的创建:

执行器的架构体系:



在上面的学习中，我们已经理解了各种 Executor 的实现代码。

那么，Executor 对象究竟在 MyBatis 中，是如何被创建的呢？

其实 Configuration 类中，提供 `newExecutor` 方法，代码如下：

```
/**
 * 创建 Executor 对象 -----在创建SqlSession执行
 *
 * @param transaction 事务对象
 * @param executorType 执行器类型
 * @return Executor 对象
 */
public Executor newExecutor(Transaction transaction, ExecutorType executorType)
{
    // <1> 获得执行器类型
    //可以通过在 mybatis-config.xml 配置 <setting name="defaultExecutorType"
    value="" />
    executorType = executorType == null ? defaultExecutorType : executorType; //
    使用默认
    executorType = executorType == null ? ExecutorType.SIMPLE : executorType; //
    判断默认
    // <2> 3个分支3种执行器BatchExecutor/ReuseExecutor/SimpleExecutor: 默认为
    SimpleExecutor 对象
    Executor executor;
    if (ExecutorType.BATCH == executorType) {
        executor = new BatchExecutor(this, transaction);
    } else if (ExecutorType.REUSE == executorType) {
        executor = new ReuseExecutor(this, transaction);
    } else {
        executor = new SimpleExecutor(this, transaction);
    }
    // <3> 如果开启缓存, 创建 CachingExecutor 对象, 装饰者模式进行包装
    if (cacheEnabled) {
```

```
        executor = new CachingExecutor(executor);
    }
    // <4> 应用插件
    executor = (Executor) interceptorChain.pluginAll(executor);
    return executor;
}
```

小结:

1. 我们可以在 `ExecutorType` 中看到，枚举了三种类型：SIMPLE、REUSE、BATCH
2. 如果没有指定类型，默认为 SimpleExecutor 对象
3. 如果开启缓存，创建 CachingExecutor 对象，进行包装

五、MyBatis进阶之缓存原理

Mybatis 提供了缓存策略，通过缓存策略来减少数据库的查询次数，从而提高性能。

Mybatis的缓存分为：一级缓存和二级缓存

5.1 一级缓存

1) 概述:

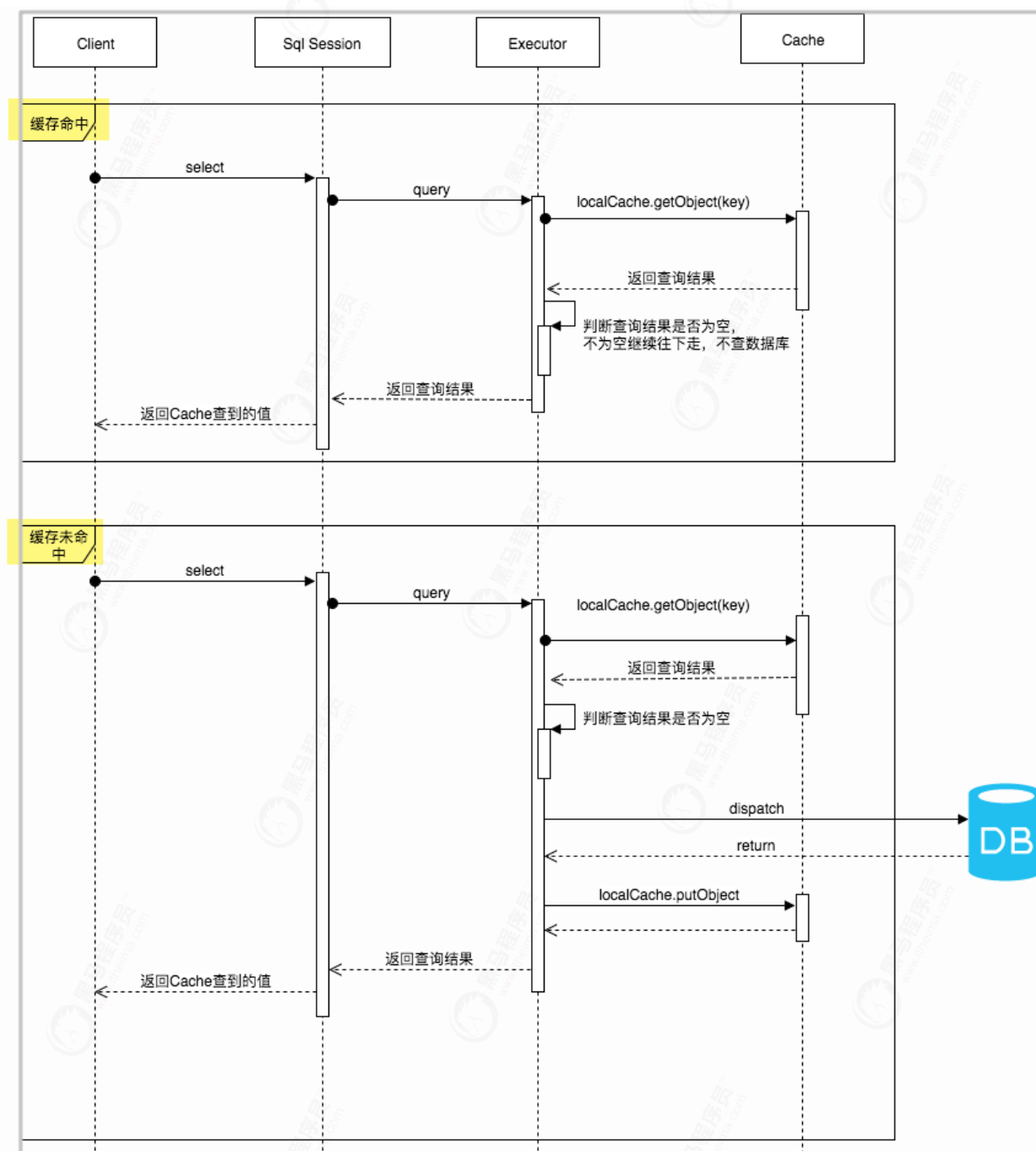
当我们使用MyBatis开启一次和数据库的会话，MyBatis会创建出一个SqlSession对象表示一次数据库会话，建立一个简单的缓存，将每次查询到的结果缓存起来，当下次查询的时候，如果判断先前有个完全一样的查询，会直接从缓存中直接将结果取出，返回给用户，不需要再进行一次数据库查询了。

说明：对于会话（Session）级别的数据缓存，我们称之为一级数据缓存，简称一级缓存。

特点:

- 一级缓存是默认开启的
- 一级缓存是基于SqlSession生命周期的

2) 一级缓存示意图:



SqlSession -- DefaultSqlSession

Executor -- BaseExecutor -- "PerpetualCache localCache"

3) 测试一级缓存:

- 测试1: 测试多次查询同一条数据

```
@Test
public void testLocalCache01() {
    User user = userMapper.findUserById(101);
    System.out.println("第一次查询:"+user);
    User user2 = userMapper.findUserById(101);
    System.out.println("第二次查询:"+user2);
}
```

查询结果查看:

调用了两次dao查询方法,但只执行了一条sql语句,说明第二次查询获得的对象是从缓存中取的,并未执行sql语句

```
DEBUG com.itheima.dao.UserMapper.findUserById:137 - ==> Preparing: select * from user where id = ?
DEBUG com.itheima.dao.UserMapper.findUserById:137 - ==> Parameters: 101(Integer)
DEBUG com.itheima.dao.UserMapper.findUserById:137 - <== Total: 1
第一次查询:User(id=101, username=itcast, address=beijing, name=传智播客)
第二次查询:User(id=101, username=itcast, address=beijing, name=传智播客)
```

第二次查询未真正查询数据库

- 测试2: 测试修改后再查询:

```
/**
 * 在两次查询中间 执行 更新操作:
 */
@Test
public void testLocalCache02() {
    User user = userMapper.findUserById(101);
    System.out.println("-----第一次查询:"+user);

    userMapper.saveUser(new User(null, "bxg", "bj", "博学谷"));
    System.out.println("-----更新了数据-----");

    User user2 = userMapper.findUserById(101);
    System.out.println("-----第二次查询:"+user2);
}
```

查询结果查看:

在一次数据库会话中,如果对数据库发生了修改操作,在修改操作后执行的相同查询,查询了数据库,一级缓存失效。

```
DEBUG com.itheima.dao.UserMapper.findUserById:137 - ==> Preparing: select * from user where id = ?
DEBUG com.itheima.dao.UserMapper.findUserById:137 - ==> Parameters: 101(Integer)
DEBUG com.itheima.dao.UserMapper.findUserById:137 - <==      Total: 1
-----第一次查询:User(id=101, username=itcast, address=beijing, name=传智播客)
DEBUG com.itheima.dao.UserMapper.saveUser:137 - ==> Preparing: INSERT INTO user(username,address,name) VALUES(?,?,?)
DEBUG com.itheima.dao.UserMapper.saveUser:137 - ==> Parameters: bxc(String), bj(String), 博学谷(String)
DEBUG com.itheima.dao.UserMapper.saveUser:137 - <==      Updates: 1
-----更新了数据-----
DEBUG com.itheima.dao.UserMapper.findUserById:137 - ==> Preparing: select * from user where id = ?
DEBUG com.itheima.dao.UserMapper.findUserById:137 - ==> Parameters: 101(Integer)
DEBUG com.itheima.dao.UserMapper.findUserById:137 - <==      Total: 1
-----第二次查询:User(id=101, username=itcast, address=beijing, name=传智播客)
```

4) 小结:

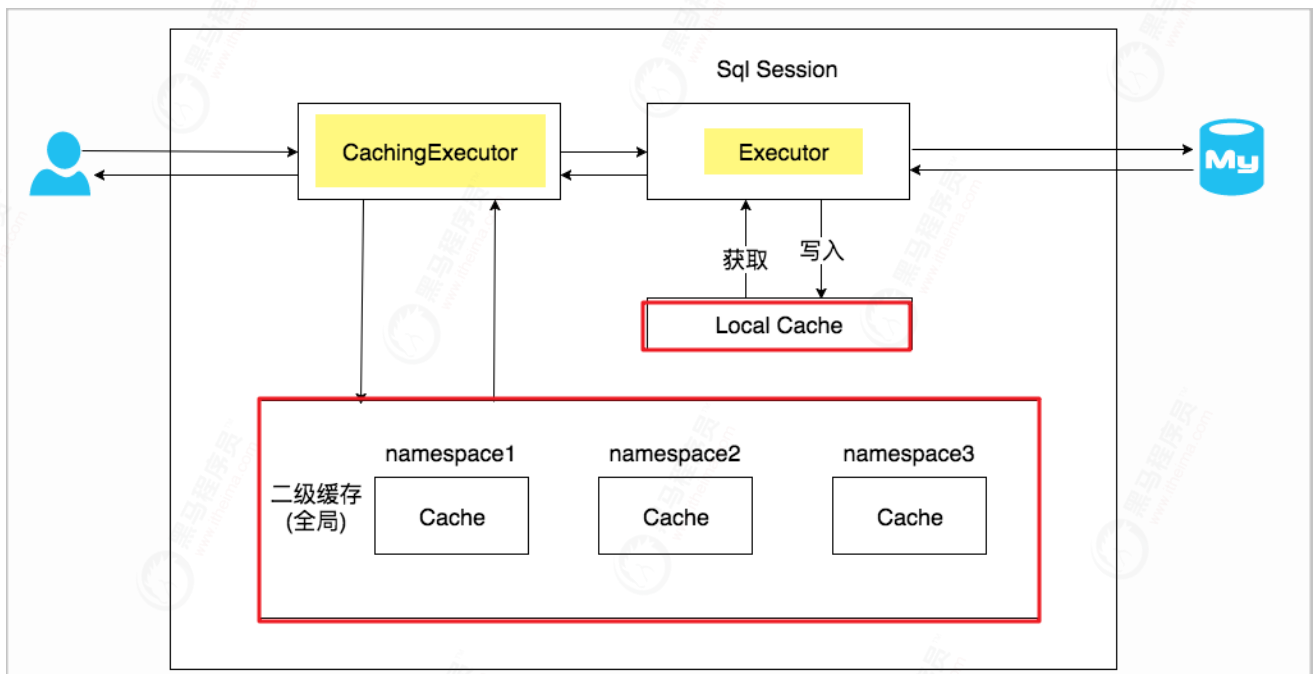
一级缓存生命周期 当会话结束时, SqlSession对象及其内部的Executor对象还有PerpetualCache对象也一并释放掉 如果SqlSession调用了close()方法, 会释放掉一级缓存PerpetualCache对象, 一级缓存将不可用 如果SqlSession调用了clearCache(), 会清空PerpetualCache对象中的数据, 但是该对象仍可使用 SqlSession中执行了任何一个update操作(update()、delete()、insert()), 都会清空PerpetualCache对象的数据, 但是该对象可以继续使用

5.2 二级缓存

1) 概述:

- 一级缓存中, 其最大的共享范围就是一个 SqlSession 内部;
- 如果多个 SqlSession 之间需要共享缓存, 则需要使用到**二级缓存**。
- 开启二级缓存后, 会使用 CachingExecutor 装饰 Executor, 进入一级缓存的查询流程前, 先在 CachingExecutor 进行二级缓存的查询。

2) 二级缓存示意图:



解读

开启二级缓存后，执行器Executor会被CachingExecutor包装一层。（装饰器模式）二级缓存是手动开启的，作用域为sessionfactory 二级缓存则是全局级别的，不同的session共用同一个二级缓存维护二级缓存，只有在提交事务之后二级缓存才会保存（查询的时候先走二级缓存再走一级缓存【开启二级缓存的条件下】）

它采用的是**装饰器模式**它里面有个属性是 Executor delegate;（实例化哪个自己配置的，默认是 SimpleExecutor，使用有参构造给属性赋值）这里存的就是你三种执行器中的一种。

装饰者模式：在不改变原有类和继承的情况下，通过**包装原对象**去扩展一个新的功能。

3) 开启二级缓存配置：

// MyBatis的配置文件中二级缓存：默认已开启

```
<settings>
    <!--默认是true protected boolean cacheEnabled = true;-->
    <setting name = "cacheEnabled" value = "true" />
</settings>
```

//mapper映射XML中配置cache或者 cache-ref: cache标签用于声明这个namespace使用二级缓存，并且可以自定义配置。

```
<cache/>
```

4) CachingExecutor剖析:

`org.apache.ibatis.executor.CachingExecutor`，实现 `Executor` 接口，支持二级缓存的 `Executor` 的实现类。

- 先从缓存中获取查询结果，存在就返回;
- 不存在，再委托给 `Executor delegate` 去数据库取;
- `delegate` 可以是 `SimpleExecutor`、`ReuseExecutor`、`BatchExecutor`。

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

//query
@Override
public <E> List<E> query(MappedStatement ms, Object parameterObject, RowBounds
rowBounds, ResultHandler resultHandler, CacheKey key, BoundSql boundSql)
    throws SQLException {
    // <1> 获得 Cache 对象，即当前 MappedStatement 对象的二级缓存。
    Cache cache = ms.getCache();
    if (cache != null) {
        // <2.1> 如果需要清空缓存，则进行清空
        // 注意： 注意，此时清空的仅仅，当前事务中查询数据产生的缓存。而真正的清空，在事务的提
        交时。
```

```

flushCacheIfRequired(ms);
if (ms.isUseCache() && resultHandler == null) { // <2.2>
    // 暂时忽略, 存储过程相关
    ensureNoOutParams(ms, boundSql);
    @SuppressWarnings("unchecked")
    // <2.3> 从二级缓存中, 获取结果
    List<E> list = (List<E>) tcm.getObject(cache, key);
    if (list == null) {
        // <2.4.1> 如果不存在, 则从数据库中查询
        list = delegate.query(ms, parameterObject, rowBounds,
resultHandler, key, boundSql);
        // <2.4.2> 缓存结果到二级缓存中
        tcm.putObject(cache, key, list);
    }
    // <2.5> 如果存在, 则直接返回结果
    return list;
}
}
// <3> 不使用缓存, 则从数据库中查询
return delegate.query(ms, parameterObject, rowBounds, resultHandler, key,
boundSql);
}

// update
@Override
public int update(MappedStatement ms, Object parameterObject) throws
SQLException {
    // 如果需要清空缓存, 则进行清空
    flushCacheIfRequired(ms);
    // 执行 delegate 对应的方法
    return delegate.update(ms, parameterObject);
}

```

解析:

1. 获得 Cache 对象, 即当前 MappedStatement 对象的**二级缓存**
2. 如果**没有** Cache 对象, 说明该 MappedStatement 对象, 未设置**二级缓存**, 则调用 `delegate` 属性的 `#query(...)` 方法, 直接从数据库中查询。
3. 如果**有** Cache 对象, 说明该 MappedStatement 对象, 有设置**二级缓存**:

5) 缓存执行器流程分析:

```
CachingExecutor.java x
91
92 @Override
93 public <E> List<E> query(MappedStatement ms, Object parameterObject, RowBounds rowBounds, ResultHandler resultHandler)
94     throws SQLException {
95     Cache cache = ms.getCache();
96     if (cache != null) { 开始走二级缓存的逻辑
97         flushCacheIfRequired(ms); 判断要不要清空二级缓存
98         if (ms.isUseCache() && resultHandler == null) {
99             ensureNoOutParams(ms, boundSql);
100             /unchecked/
101             List<E> list = (List<E>) tcm.getObject(cache, key); 通过缓存事务管理器来获取缓存
102             if (list == null) { 将查询操作交给下一个执行器来执行【查询数据库】
103                 list = delegate.<E> query(ms, parameterObject, rowBounds, resultHandler, key, boundSql);
104                 tcm.putObject(cache, key, list); // issue #578 and #116
105             } 再将执行后的结果交给二级缓存来处理
106             return list;
107         }
108     } 如果缓存中没有,就直接调用下一个执行器来进行查询
109     return delegate.<E> query(ms, parameterObject, rowBounds, resultHandler, key, boundSql);
110 }
```

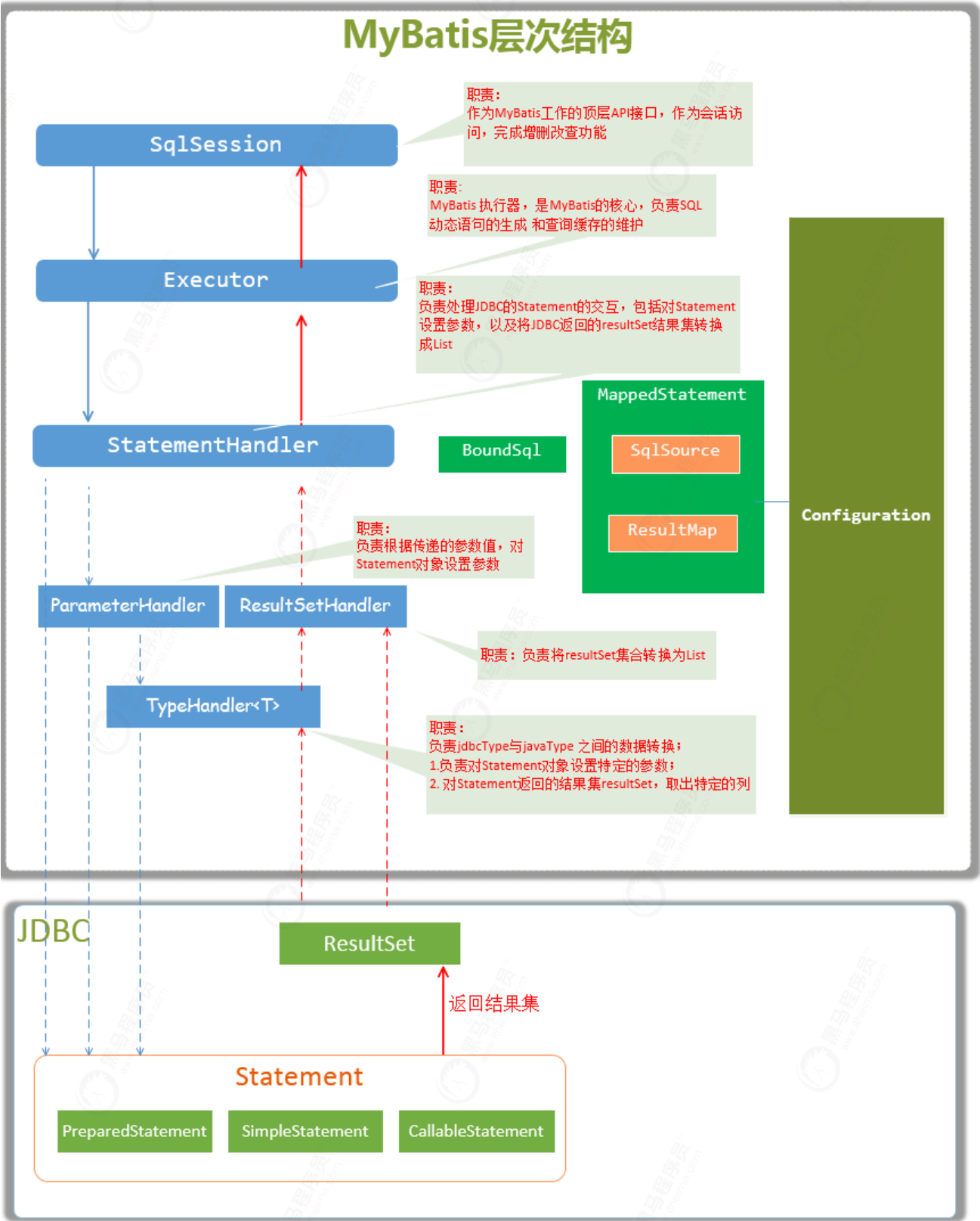
6) 小结:

- 在优化系统性能时,优化数据库性能是非常重要的一个环节,而添加缓存则是优化数据库时最有效的手段之一。
- 正确、合理地使用缓存可以将一部分数据库请求拦截在缓存这一层。
- MyBatis 中提供的一级缓存和二级缓存,而这两级缓存都是依赖于基础支持层中的缓存模块实现的。这里需要注意的是,在分布式环境下,由于默认的MyBatis Cache实现都是基于本地的,占用系统资源,并且有一定安全性问题,建议直接使用Redis、Memcached等分布式缓存可能成本更低,安全性也更高。

六、MyBatis源码之SQL执行过程

6.1 SQL语句的执行过程介绍

MyBatis核心执行组件:



6.2 SQL执行的入口分析

1) 为Mapper接口创建代理对象

```
// 方式1:
User user = session.selectOne("com.itheima.dao.UserMapper.findUserById", 101);

// 方式2:
UserMapper mapper = session.getMapper(UserMapper.class);
List<User> userList = mapper.findAll();
```

2) 执行代理逻辑

- 方式1入口分析:

session是DefaultSqlSession类型的，因为sqlSessionFactory默认生成的SqlSession是DefaultSqlSession类型。

selectOne()会调用selectList()。

```
// DefaultSqlSession类
public <E> List<E> selectList(String statement, Object parameter, RowBounds
rowBounds) {
    try {
        MappedStatement ms = configuration.getMappedStatement(statement);
        // CURD操作是交给Excetor去处理的
        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();
    }
}
```

- 方式2入口分析:

获取代理对象:

```
//DefaultSqlSession类 =====>
@Override
public <T> T getMapper(Class<T> type) {
    return configuration.getMapper(type, this);
}
```

```

}

// Configuration类 =====>
public <T> T getMapper(Class<T> type, SqlSession sqlSession) {
    return mapperRegistry.getMapper(type, sqlSession);
}

//MapperRegistry ----> apperProxyFactory.newInstance =====>
public <T> T getMapper(Class<T> type, SqlSession sqlSession) {
    //从缓存中获取该Mapper接口的代理工厂对象
    final MapperProxyFactory<T> mapperProxyFactory = (MapperProxyFactory<T>)
knownMappers.get(type);
    //如果该Mapper接口没有注册过，则抛异常
    if (mapperProxyFactory == null) {
        throw new BindingException("Type " + type + " is not known to the
MapperRegistry.");
    }
    try {
        //【使用代理工厂创建Mapper接口的代理对象】
        return mapperProxyFactory.newInstance(sqlSession);
    } catch (Exception e) {
        throw new BindingException("Error getting mapper instance. Cause: " + e,
e);
    }
}

//MapperProxyFactory ---->此时生成代理对象 =====>
protected T newInstance(MapperProxy<T> mapperProxy) {
    //Mybatis底层是调用JDK的Proxy类来创建代理实例
    return (T) Proxy.newProxyInstance(mapperInterface.getClassLoader(), new
Class[] { mapperInterface }, mapperProxy);
}

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

```

代理对象执行逻辑:

```

//MapperProxy =====>
/**代理对象执行的方法，代理以后，所有Mapper的方法调用时，都会调用这个invoke方法*/

```

```

public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
    try {
        if (Object.class.equals(method.getDeclaringClass())) {
            //如果是Object方法，则调用方法本身
            return method.invoke(this, args);
        } else {
            //调用接口方法：根据被调用接口的Method对象，从缓存中获取MapperMethodInvoker对象
            //apper接口中的每一个方法都对应一个MapperMethodInvoker对象，而MapperMethodInvoker
            对象里面的MapperMethod保存着对应的SQL信息和返回类型以完成SQL调用 ...
            return cachedInvoker(method).invoke(proxy, method, args, sqlSession);
        }
    } catch (Throwable t) {
        throw ExceptionUtil.unwrapThrowable(t);
    }
}

/**
    获取缓存中MapperMethodInvoker，如果没有则创建一个，而MapperMethodInvoker内部封装这一
    个MethodHandler
    */
private MapperMethodInvoker cachedInvoker(Method method) throws Throwable {
    try {
        return methodCache.computeIfAbsent(method, m -> {
            if (m.isDefault()) {
                //如果调用接口的是默认方法 (default方法)
                try {
                    if (privateLookupInMethod == null) {
                        return new
DefaultMethodInvoker(getMethodHandleJava8(method));
                    } else {
                        return new
DefaultMethodInvoker(getMethodHandleJava9(method));
                    }
                } catch (IllegalAccessException | InstantiationException |
InvocationTargetException
                    | NoSuchMethodException e) {
                    throw new RuntimeException(e);
                }
            } else {
                //如果调用的普通方法 (非default方法)，则创建一个PlainMethodInvoker并放
                入缓存，其中MapperMethod保存对应接口方法的SQL以及入参和出参的数据类型等信息
                return new PlainMethodInvoker(new MapperMethod(mapperInterface,
method, sqlSession.getConfiguration()));
            }
        });
    } catch (RuntimeException re) {
        Throwable cause = re.getCause();
    }
}

```

```

        throw cause == null ? re : cause;
    }
}

// MapperProxy内部类: PlainMethodInvoker =====>
// 当cacheInvoker返回了PlainMethodInvoker实例之后,紧接着调用了这个实例的
PlainMethodInvoker:invoke方法
@Override
public Object invoke(Object proxy, Method method, Object[] args, SqlSession
sqlSession) throws Throwable {
    //Mybatis实现接口方法的核心: MapperMethod::execute方法:
    return mapperMethod.execute(sqlSession, args);
}

// MapperMethod =====>
public Object execute(SqlSession sqlSession, Object[] args) {
    Object result;
    switch (command.getType()) {
        case INSERT: {
            // 将args进行解析,如果是多个参数则,则根据@param注解指定名称将参数转换为Map,
            // 如果是封装实体则不转换
            Object param = method.convertArgsToSqlCommandParam(args);
            result = rowCountResult(sqlSession.insert(command.getName(),
param));
            break;
        }
        case UPDATE: {
            Object param = method.convertArgsToSqlCommandParam(args);
            result = rowCountResult(sqlSession.update(command.getName(),
param));
            break;
        }
        case DELETE: {
            Object param = method.convertArgsToSqlCommandParam(args);
            result = rowCountResult(sqlSession.delete(command.getName(),
param));
            break;
        }
        case SELECT:
            //查询操作
            if (method.returnsVoid() && method.hasResultHandler()) {
                executeWithResultHandler(sqlSession, args);
                result = null;
            } else if (method.returnsMany()) {
                result = executeForMany(sqlSession, args);
            } else if (method.returnsMap()) {

```

```

        result = executeForMap(sqlSession, args);
    } else if (method.returnsCursor()) {
        result = executeForCursor(sqlSession, args);
    } else {
        //解析参数, 因为SqlSession::selectOne方法参数只能传入一个, 但是我们
        //Mapper中可能传入多个参数,
        //有可能是通过@param注解指定参数名, 所以这里需要将Mapper接口方法中的多个参
        //数转化为一个ParamMap,
        //也就是说如果是传入的单个封装实体, 那么直接返回出来; 如果传入的是多个参数,
        //实际上都转换成了Map
        Object param = method.convertArgsToSqlCommandParam(args);
        //可以看到动态代理最后还是使用SqlSession操作数据库的
        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());
    }
    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() + ").");
    }
    return result;
}

```

// 此时我们发现: 回到了sqlSession中

```

private <E> Object executeForMany(SqlSession sqlSession, Object[] args) {
    List<E> result;
    Object param = method.convertArgsToSqlCommandParam(args);
    if (method.hasRowBounds()) {
        RowBounds rowBounds = method.extractRowBounds(args);
        result = sqlSession.selectList(command.getName(), param, rowBounds);
    } else {
        result = sqlSession.selectList(command.getName(), param);
    }
}

```

```
// ...
return result;
}
```

```
public Object execute(SqlSession sqlSession, Object[] args) {
    Object result;
    switch (command.getType()) {
        case INSERT: {
            Object param = method.convertArgsToSqlCommandParam(args);
            result = rowCountResult(sqlSession.insert(command.getName(), param));
            break;
        }
        case UPDATE: {
            Object param = method.convertArgsToSqlCommandParam(args);
            result = rowCountResult(sqlSession.update(command.getName(), param));
            break;
        }
        case DELETE: {
            Object param = method.convertArgsToSqlCommandParam(args);
            result = rowCountResult(sqlSession.delete(command.getName(), param));
            break;
        }
        case SELECT:
            if (method.returnsVoid() && method.hasResultHandler()) {
                executeWithResultHandler(sqlSession, args);
                result = null;
            } else if (method.returnsMany()) {
                result = executeForMany(sqlSession, args);
            } else if (method.returnsMap()) {
                result = executeForMap(sqlSession, args);
            } else if (method.returnsCursor()) {
                result = executeForCursor(sqlSession, args);
            } else {
                Object param = method.convertArgsToSqlCommandParam(args);
                result = sqlSession.selectOne(command.getName(), param);
                if (method.returnsOptional()
                    && (result == null || !method.getReturnType().equals(result.getClass()))) {
                    result = Optional.ofNullable(result);
                }
            }
    }
}
```

【最后getMapper获得代理对象后，回到sqlsession中，执行同样的逻辑】

```
private <E> Object executeForMany(SqlSession sqlSession, Object[] args) {
    List<E> result;
    Object param = method.convertArgsToSqlCommandParam(args);
    if (method.hasRowBounds()) {
        RowBounds rowBounds = method.extractRowBounds(args);
        result = sqlSession.selectList(command.getName(), param, rowBounds);
    } else {
        result = sqlSession.selectList(command.getName(), param);
    }
}
```

6.3 查询语句的执行过程分析

1) selectOne方法分析

```
// DefaultSqlSession类 =====>

// selectOne
@Override
public <T> T selectOne(String statement, Object parameter) {
    // //selectOne()会调用selectList().
    List<T> list = this.selectList(statement, parameter);
    if (list.size() == 1) {
        return list.get(0);
    } else if (list.size() > 1) {
        throw new TooManyResultsException("Expected one result (or null) to be returned by selectOne(), but found: " + list.size());
    } else {
        return null;
    }
}
```

```

        return null;
    }
}

// selectList
public <E> List<E> selectList(String statement, Object parameter, RowBounds
rowBounds) {
    try {
        MappedStatement ms = configuration.getMappedStatement(statement);
        // CURD操作是交给Excetor去处理的
        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();
    }
}

```

2) sql获取

```

// CachingExecutor =====>
public <E> List<E> query(MappedStatement ms, Object parameterObject, RowBounds
rowBounds, ResultHandler resultHandler) throws SQLException {
    // 获取绑定的sql命令, 比如"SELECT * FROM xxx"
    BoundSql boundSql = ms.getBoundSql(parameterObject);
    CacheKey key = createCacheKey(ms, parameterObject, rowBounds, boundSql);
    return query(ms, parameterObject, rowBounds, resultHandler, key, boundSql);
}

@Override
public <E> List<E> query(MappedStatement ms, Object parameterObject, RowBounds
rowBounds, ResultHandler resultHandler, CacheKey key, BoundSql boundSql)
throws SQLException {
    Cache cache = ms.getCache();
    if (cache != null) {
        flushCacheIfRequired(ms);
        if (ms.isUseCache() && resultHandler == null) {
            ensureNoOutParams(ms, boundSql);
            @SuppressWarnings("unchecked")
            List<E> list = (List<E>) tcm.getObject(cache, key);
            if (list == null) {
                list = delegate.query(ms, parameterObject, rowBounds,
resultHandler, key, boundSql);
                tcm.putObject(cache, key, list); // issue #578 and #116
            }
        }
    }
    return delegate.query(ms, parameterObject, rowBounds, resultHandler, key, boundSql);
}

```

```

    }
    return list;
}
}
return delegate.query(ms, parameterObject, rowBounds, resultHandler, key,
boundSql);
}

```

//真正执行query操作的是SimpleExecutor代理来完成的，SimpleExecutor的父类BaseExecutor的query方法中：

```

// BaseExecutor类:SimpleExecutor的父类 =====>
@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++;
        //localCache是一级缓存，如果找不到就调用queryFromDatabase从数据库中查找
        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();
        }
        deferredLoads.clear();
        if (configuration.getLocalCacheScope() == LocalCacheScope.STATEMENT) {
            clearLocalCache();
        }
    }
}

```

```

    }
    return list;
}

//第一次, 没有缓存, 所以会调用queryFromDatabase方法来执行查询。
private <E> List<E> queryFromDatabase(...) throws SQLException {
    List<E> list;
    localCache.putObject(key, EXECUTION_PLACEHOLDER);
    try {
        // 查询
        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;
}

```

```

// SimpleExecutor类 =====>
public <E> List<E> doQuery(...) throws SQLException {
    Statement stmt = null;
    try {
        Configuration configuration = ms.getConfiguration();
        StatementHandler handler = configuration.newStatementHandler(...);
        // 1: SQL查询参数的设置
        stmt = prepareStatement(handler, ms.getStatementLog());
        // StatementHandler封装了Statement
        // 2: SQL查询操作和结果集的封装
        return handler.<E>query(stmt);
    } finally {
        closeStatement(stmt);
    }
}

```

3) 参数设置:

```

// SimplyExecutor类 =====>
// 【1】 参数设置: prepareStatement
private Statement prepareStatement(StatementHandler handler, Log statementLog)
throws SQLException {

```

```

Statement stmt;
// 通过getConnection方法来获取一个Connection,
Connection connection = getConnection(statementLog);
// 调用prepare方法来获取一个Statement
stmt = handler.prepare(connection, transaction.getTimeout());

// 设置SQL查询中的参数值 ***
handler.parameterize(stmt);
return stmt;
}

// RoutingStatementHandler =====>
// PreparedStatementHandler =====>
@Override
public void parameterize(Statement statement) throws SQLException {
    parameterHandler.setParameters((PreparedStatement) statement);
}

// DefaultParameterHandler =====> 此时参数设置成功
@Override
public void setParameters(PreparedStatement ps) {
    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)) {
                    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);
                }
                TypeHandler typeHandler = parameterMapping.getTypeHandler();
                JdbcType jdbcType = parameterMapping.getJdbcType();
                if (value == null && jdbcType == null) {
                    jdbcType = configuration.getJdbcTypeForNull();
                }
            }
        }
    }
}

```

```

        try {
            typeHandler.setParameter(ps, i + 1, value, jdbcType);
        } catch (TypeException | SQLException e) {
            throw new TypeException("Could not set parameters for
mapping.....");
        }
    }
}
}
}
}
}

```

4) SQL执行和结果集的封装

```

// RoutingStatementHandler =====>
@Override
public <E> List<E> query(Statement statement) throws SQLException {
    return delegate.<E>query(statement);
}

// PreparedStatementHandler =====>
@Override
public <E> List<E> query(Statement statement, ResultHandler resultHandler)
throws SQLException {
    // 这里就到了熟悉的PreparedStatement了
    PreparedStatement ps = (PreparedStatement) statement;
    // 执行SQL查询操作
    ps.execute();
    // 结果交给ResultHandler来处理
    return resultSetHandler.<E> handleResultSets(ps);
}

// DefaultResultSetHandler类 (封装返回值, 将查询结果封装成Object对象)
@Override
public List<Object> handleResultSets(Statement stmt) throws SQLException {
    ErrorContext.instance().activity("handling
results").object(mappedStatement.getId());

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

    int resultSetCount = 0;
    ResultSetWrapper rsw = getFirstResultSet(stmt);

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

```

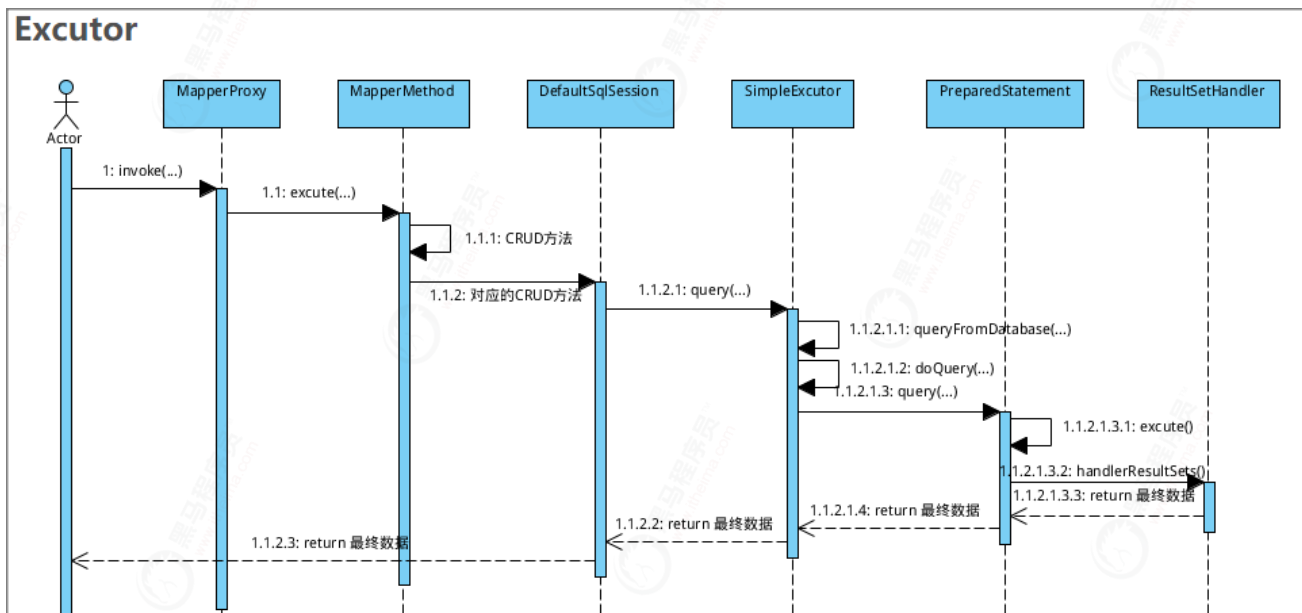
```

int resultMapCount = resultMaps.size();
validateResultMapsCount(rsw, resultMapCount);
while (rsw != null && resultMapCount > resultSetCount) {
    ResultMap resultMap = resultMaps.get(resultSetCount);
    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);
}

```



6.4 更新语句的执行过程分析

1. Executor 的 update 方法分析

- insert、update 和 delete 操作都会清空一二级缓存

2. doUpdate 方法

3. PreparedStatementHandler 的 update 方法

- 默认是创建PreparedStatementHandler，然后执行prepareStatement方法。
- 执行结果为受影响行数
- 执行更新语句的SQL

1) sqlSession增删改方法分析:

```

// DefaultSqlSession =====>

@Override
public int insert(...) {
    return update(statement, parameter);
}

@Override
public int update(String statement) {
    return update(statement, null);
}

@Override

```

```

public int delete(...) {
    return update(...);
}

// insert 、delete操作是通过调用update语句进行的相关逻辑
@Override
public int update(String statement, Object parameter) {
    try {
        dirty = true;
        MappedStatement ms = configuration.getMappedStatement(statement);
        // 增删改 最终底层都是 update
        return executor.update(ms, wrapCollection(parameter));
    } catch (Exception e) {
        throw ExceptionFactory.wrapException("Error updating database. Cause: " +
e, e);
    } finally {
        ErrorContext.instance().reset();
    }
}

```

2) sql获取

```

// CachingExecutor =====>
@Override
public int update(MappedStatement ms, Object parameterObject) throws
SQLException {
    // 执行增删改, 清除缓存
    flushCacheIfRequired(ms);
    // 跳转BaseExecutor
    return delegate.update(ms, parameterObject);
}

// BaseExecutor =====>
@Override
public int update(MappedStatement ms, Object parameter) throws SQLException {
    ErrorContext.instance().resource(ms.getResource()).activity("executing an
update").object(ms.getId());
    if (closed) {
        throw new ExecutorException("Executor was closed.");
    }
}

```

```

    }
    // 清除 LocalCache 一级缓存
    clearLocalCache();
    //执行 doUpdate
    return doUpdate(ms, parameter);
}

// SimpleExecutor =====>
// doUpdate
@Override
public int doUpdate(MappedStatement ms, Object parameter) throws SQLException {
    Statement stmt = null;
    try {
        Configuration configuration = ms.getConfiguration();
        StatementHandler handler = configuration.newStatementHandler(...);
        // 【1】.获取statement,并进行参数映射
        stmt = prepareStatement(handler, ms.getStatementLog());
        // 【2】.handler.update()方法执行具体sql指令
        return handler.update(stmt);
    } finally {
        closeStatement(stmt);
    }
}
}

```

3) 参数设置:

```

// SimplyExecutor类 =====>
// 【1】 prepareStatement
private Statement prepareStatement(StatementHandler handler, Log statementLog)
throws SQLException {
    Statement stmt;
    Connection connection = getConnection(statementLog);
    // 使用connection对象信息创建statement, 并将超时时间绑定
    stmt = handler.prepare(connection, transaction.getTimeout());
    // parameterize方法设置sql执行时候需要的参数
    handler.parameterize(stmt);
    return stmt;
}

// RoutingStatementHandler =====>
// PreparedStatementHandler =====>

```

```

@Override
public void parameterize(Statement statement) throws SQLException {
    parameterHandler.setParameters((PreparedStatement) statement);
}

// DefaultParameterHandler =====> 此时参数设置成功
@Override
public void setParameters(PreparedStatement ps) {
    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)) {
                    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);
                }
                TypeHandler typeHandler = parameterMapping.getTypeHandler();
                JdbcType jdbcType = parameterMapping.getJdbcType();
                if (value == null && jdbcType == null) {
                    jdbcType = configuration.getJdbcTypeForNull();
                }
                try {
                    typeHandler.setParameter(ps, i + 1, value, jdbcType);
                } catch (TypeException | SQLException e) {
                    throw new TypeException("Could not set parameters for
mapping.....");
                }
            }
        }
    }
}

```

4) SQL执行

```
// RoutingStatementHandler =====>
@Override
public int update(Statement statement) throws SQLException {
    return delegate.update(statement);
}

// PreparedStatementHandler =====>
@Override
public int update(Statement statement) throws SQLException {
    // 这里就是底层JDBC的PreparedStatement 操作了
    PreparedStatement ps = (PreparedStatement) statement;
    // 执行SQL增删改操作
    ps.execute();
    // 获取影响的行数
    int rows = ps.getUpdateCount();
    Object parameterObject = boundSql.getParameterObject();
    KeyGenerator keyGenerator = mappedStatement.getKeyGenerator();
    keyGenerator.processAfter(executor, mappedStatement, ps, parameterObject);
    // 返回影响的行数
    return rows;
}
```

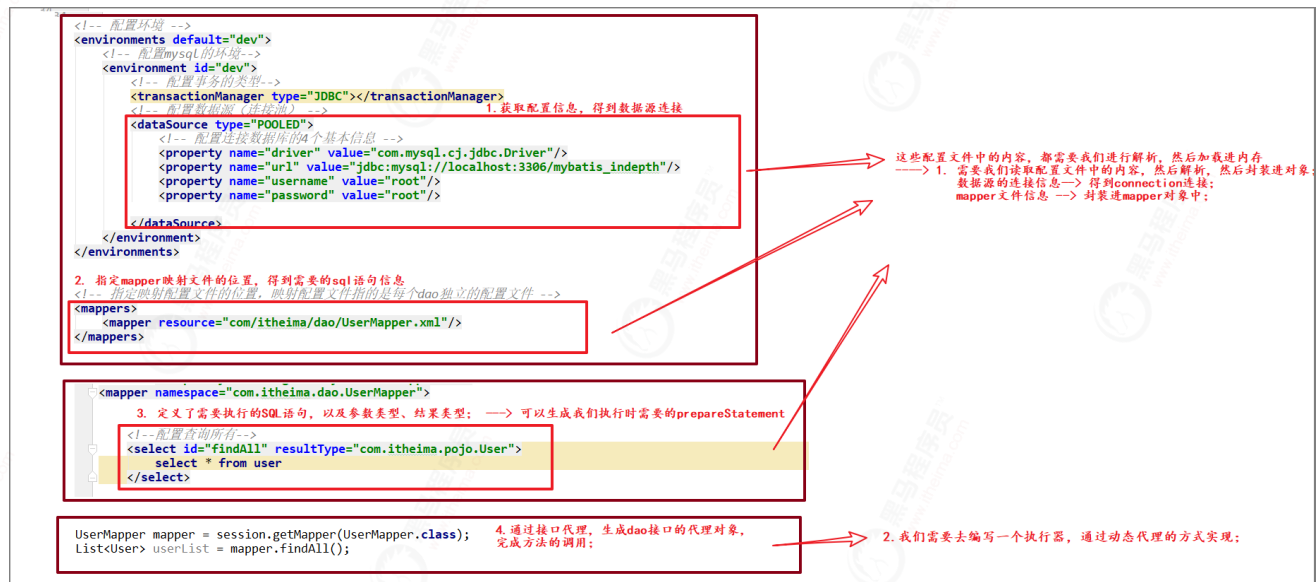
小结:

mybatis执行SQL的流程都是:

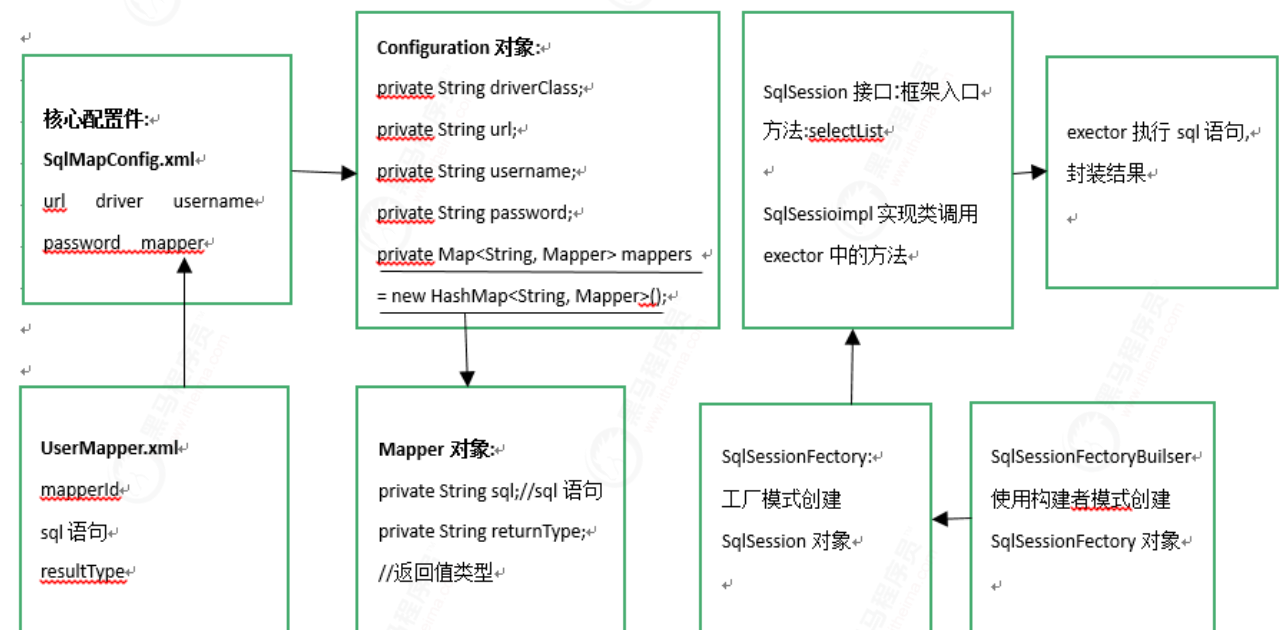
- 1.根据statement字符串从configuration中获取对应的mappedStatement;
- 2.根据获取的mappedStatement创建相应的Statement实例;
- 3.根据传入的参数对statement实例进行参数设置;
- 4.执行statement并执行后置操作;

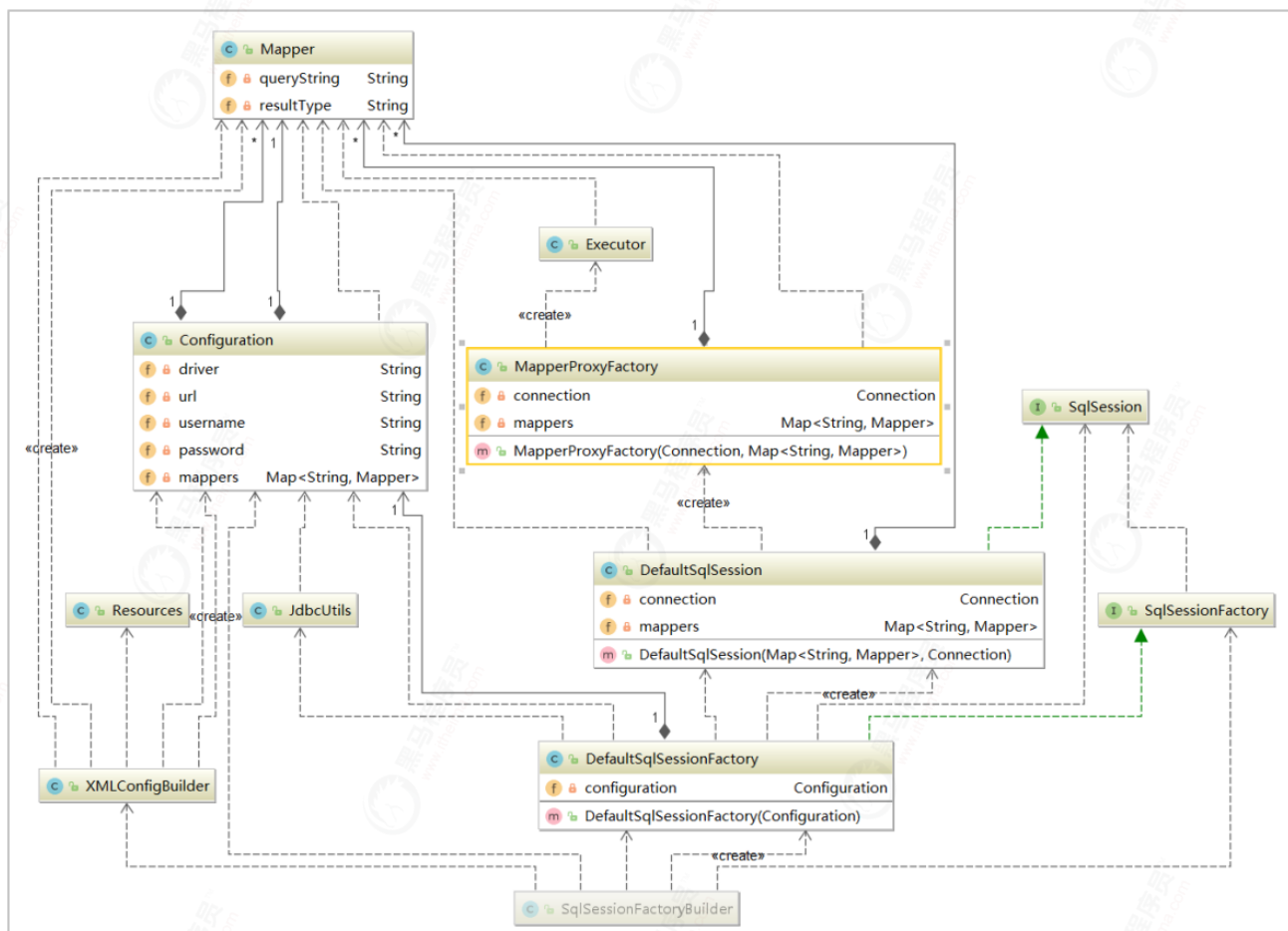
七、MyBatis进阶之自定义MyBatis框架

7.1 自定义MyBatis框架流程分析



7.2 自定义框架原理介绍





7.3 准备工作:

1) 创建maven工程

Groupid:com.itheima
 ArtifactId:custom-mybatis
 Packing:jar

2) 添加pom依赖:

```

<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.15</version>
</dependency>
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.12</version>
</dependency>

```

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</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>
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <version>1.18.12</version>
</dependency>
```

7.4 自定义MyBatis框架的实现步骤

需要定义的类的分析：

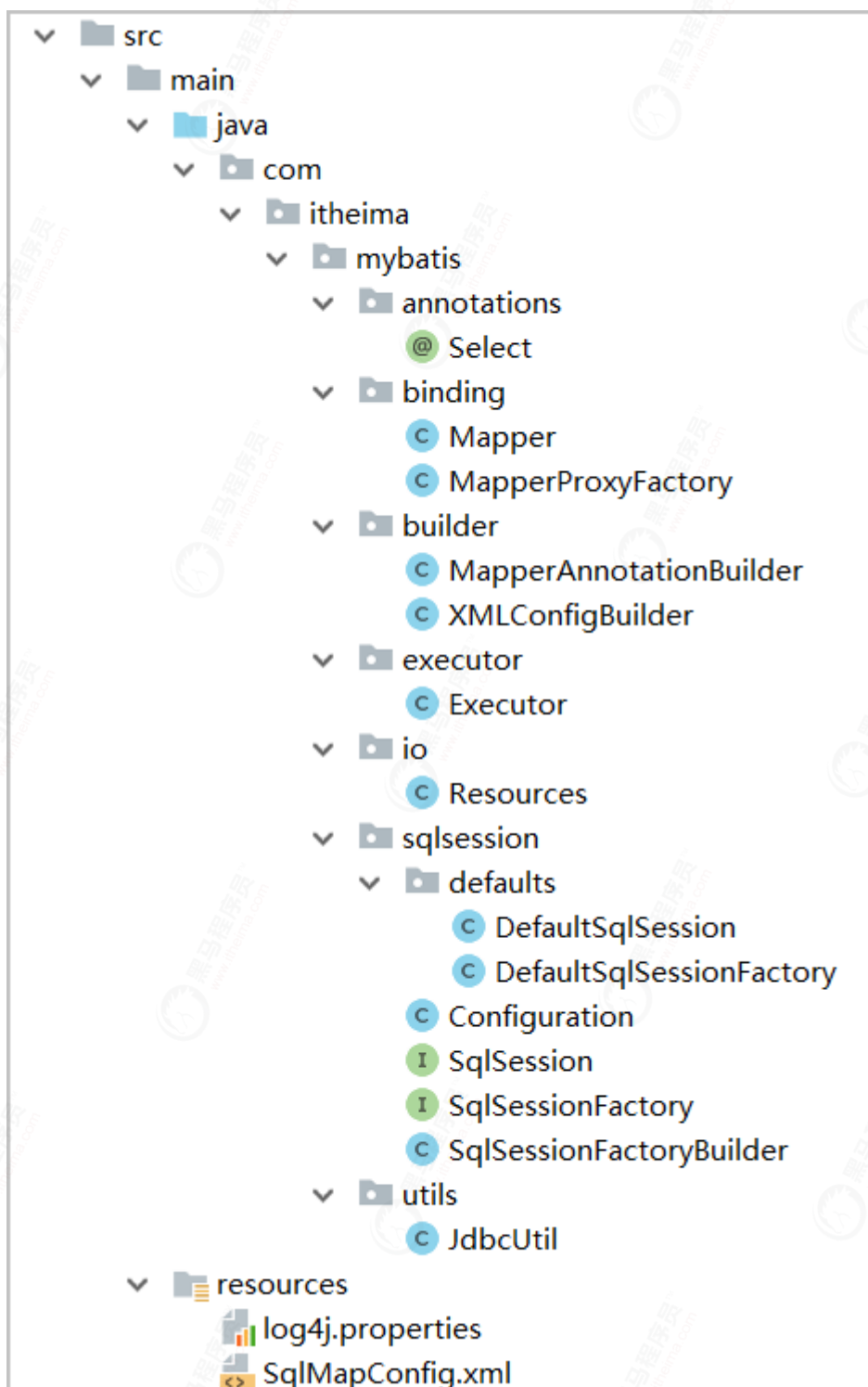
1. 加载类加载目录下的资源文件 Resources.java
2. 映射文件解析后信息类 Mapper.java
3. 保存配置文件的配置信息 Configuration.java
4. 配置文件解析的工具类 XMLConfigBuilder.java
5. SQL语句的执行对象 Executor.java
6. 获取数据源的工具类 JdbcUtil.java
7. SqlSession 接口及实现类
8. SqlSessionFactory 接口及实现类
9. SqlSessionFactory 的构建者类 SqlSessionFactoryBuilder.java
10. 动态代理的处理类 MapperProxyFactory.java

两大步：

1-4：是我们前面提到的 解析xml,封装对象；

5-10：是得到sqlsession，获取代理对象，执行sql的核心代码；

包结构参考:



1) 定义加载资源类 `Resources.java`

```

/**
 * 加载资源的工具类
 */
public class Resources {

    /**
     * 加载类加载目录下的资源
     * @param path
     * @return
     */
    public static InputStream getResourceAsStream(String path){
        return Resources.class.getClassLoader().getResourceAsStream(path);
    }
}

```

2) 定义mybatis映射配置信息类 Mapper.java

```

/**
 * 映射的配置信息类
 */
@Data
public class Mapper {

    private String queryString;// SQL语句

    private String resultType;// 实体类全限定名

}

```

3) 定义mybatis核心配置信息类 Configuration.java

```

/**
 * mybatis配置的实体类
 * 用来保存mybatis.xml解析过程中的节点属性
 */
@Data
public class Configuration {

    private String driver;        //jdbc驱动类
    private String url;          //jdbc连接字符串
    private String username;     //数据库连接用户名
    private String password;     //数据库连接密码

    private Map<String, Mapper> mappers = new HashMap<String, Mapper>();//映射文件

    public void setMappers(Map<String, Mapper> mappers) {

```

```
        this.mappers.putAll(mappers); //此处需要使用追加的方式
    }

}
```

4) 配置文件解析的工具类 XMLConfigBuilder

```
/**
 * 用于解析配置文件:
 *     sqlmapconfig.xml
 *     xxxMapper.xml
 */
public class XMLConfigBuilder {

    //定义封装连接信息的配置对象 (mybatis的配置对象)
    private static Configuration cfg = new Configuration();

    /**
     * 解析主配置文件, 把里面的内容填充到DefaultSqlSession所需要的地方
     * 使用的技术: dom4j+xpath
     */
    public static Configuration loadConfiguration(InputStream config){
        try{

            //1. 获取SAXReader对象
            SAXReader reader = new SAXReader();
            //2. 根据字节输入流获取Document对象
            Document document = reader.read(config);
            //3. 获取根节点
            Element root = document.getRootElement();

            //4. 解析数据源信息
            loadDataSourceElement(root);

            //5. 加载mapper
            loadMapperElement(root);

            //6. 返回Configuration
            return cfg;

        }catch(Exception e){
            throw new RuntimeException(e);
        }finally{
            try {
                config.close();
            }catch(Exception e){
                e.printStackTrace();
            }
        }
    }
}
```

```

    }

}

/**
 * 解析数据源信息
 */
public static void loadDataSourceElement(Element root) {
    //1.使用xpath中选择指定节点的方式，获取所有property节点
    List<Element> propertyElements = root.selectNodes("//property");
    //2.遍历节点
    for(Element propertyElement : propertyElements){
        //判断节点是连接数据库的哪部分信息
        //取出name属性的值
        String name = propertyElement.attributeValue("name");
        if("driver".equals(name)){
            //表示驱动
            //获取property标签value属性的值
            String driver = propertyElement.attributeValue("value");
            cfg.setDriver(driver);
        }
        if("url".equals(name)){
            //表示连接字符串
            //获取property标签value属性的值
            String url = propertyElement.attributeValue("value");
            cfg.setUrl(url);
        }
        if("username".equals(name)){
            //表示用户名
            //获取property标签value属性的值
            String username = propertyElement.attributeValue("value");
            cfg.setUsername(username);
        }
        if("password".equals(name)){
            //表示密码
            //获取property标签value属性的值
            String password = propertyElement.attributeValue("value");
            cfg.setPassword(password);
        }
    }
}

/**
 * 加载mapper
 */
public static void loadMapperElement(Element root) throws Exception {
    //取出mappers中的所有mapper标签，判断他们使用了resource还是class属性

```

```

List<Element> mapperElements = root.selectNodes("//mappers/mapper");
//遍历集合
for(Element mapperElement : mapperElements){
    //判断mapperElement使用的是哪个属性
    Attribute attribute = mapperElement.attribute("resource");
    if(attribute != null){
        System.out.println("-----XML方式mapper-----");
        //表示有resource属性, 用的是XML
        //取出属性的值
        String mapperPath = attribute.getValue();//获取属性的
值"com/itheima/dao/IUserDao.xml"
        //把映射配置文件的内容获取出来, 封装成一个map
        Map<String,Mapper> mappers =
loadMapperConfiguration(mapperPath);
        //给configuration中的mappers赋值
        cfg.setMappers(mappers);
    }
}

/**
 * 根据传入的参数, 解析XML, 并且封装到Map中
 * @param mapperPath    映射配置文件的位置
 * @return    map中包含了获取的唯一标识 (key是由dao的全限定类名和方法名组成)
 *           以及执行所需的必要信息 (value是一个Mapper对象, 里面存放的是执行的SQL语句和
要封装的实体类全限定类名)
 */
private static Map<String,Mapper> loadMapperConfiguration(String
mapperPath)throws IOException {
    InputStream in = null;
    try{
        //定义返回值对象
        Map<String,Mapper> mappers = new HashMap<String,Mapper>();
        //1.根据路径获取字节输入流
        in = Resources.getResourceAsStream(mapperPath);
        //2.根据字节输入流获取Document对象
        SAXReader reader = new SAXReader();
        Document document = reader.read(in);
        //3.获取根节点
        Element root = document.getRootElement();
        //4.获取根节点的namespace属性取值
        String namespace = root.attributeValue("namespace");//是组成map中key的
部分

        //5.获取所有的select节点
        List<Element> selectElements = root.selectNodes("//select");
        //6.遍历select节点集合

```

```

        for(Element selectElement : selectElements){
            //取出id属性的值      组成map中key的部分
            String id = selectElement.attributeValue("id");
            //取出resultType属性的值 组成map中value的部分
            String resultType = selectElement.attributeValue("resultType");
            //取出文本内容      组成map中value的部分
            String queryString = selectElement.getText();
            //创建key
            String key = namespace+"."+id;
            //创建value
            Mapper mapper = new Mapper();
            mapper.setQueryString(queryString);
            mapper.setResultType(resultType);
            //把key和value存入mappers中
            mappers.put(key,mapper);
        }
        return mappers;
    }catch(Exception e){
        throw new RuntimeException(e);
    }finally{
        in.close();
    }
}
}

```

5) SQL语句的执行对象 Executor.java

```

/**
 * 负责执行SQL语句，并且封装结果集
 */
public class Executor {

    /**
     * 根据mapper查询多条数据
     * @param mapper
     * @param conn
     * @param <E>
     * @return
     */
    public <E> List<E> selectList(Mapper mapper, Connection conn) {
        PreparedStatement pstmt = null;
        ResultSet rs = null;
        try {
            //1.取出mapper中的数据
            String queryString = mapper.getQueryString();//select * from user
            String resultType = mapper.getResultType();//com.itheima.domain.User

```

```

        Class domainClass = Class.forName(resultType);
        //2.获取PreparedStatement对象
        pstmt = conn.prepareStatement(queryString);
        //3.执行SQL语句, 获取结果集
        rs = pstmt.executeQuery();
        //4.封装结果集
        List<E> list = new ArrayList<E>(); //定义返回值
        while (rs.next()) {
            //将结果集中的数据封装到实体中
            E obj = extraction(rs, domainClass);
            //把赋好值的对象加入到集合中
            list.add(obj);
        }
        return list;
    } catch (Exception e) {
        throw new RuntimeException(e);
    } finally {
        release(conn, pstmt, rs);
    }
}

```

```

/**

```

```

 * 根据mapper查询数据, 查询一条数据, 如果结果集中有多条数据, 返回第一条数据封装的实体对象
 * @param mapper
 * @param conn
 * @param <E>
 * @return
 */

```

```

public <E> E selectOne(Mapper mapper, Connection conn) {
    PreparedStatement pstmt = null;
    ResultSet rs = null;
    try {
        //1.取出mapper中的数据
        String queryString = mapper.getQueryString(); //select * from user
        where id = ?
        String resultType = mapper.getResultType(); //com.itheima.domain.User
        Class domainClass = Class.forName(resultType);
        //2.获取PreparedStatement对象
        pstmt = conn.prepareStatement(queryString);
        //3.执行SQL语句, 获取结果集
        rs = pstmt.executeQuery();
        //4.封装结果集
        if (rs.next()) {
            //将结果集中的数据封装到实体中
            E obj = extraction(rs, domainClass);
            return obj;
        }
        return null;
    }
}

```

```

    } catch (Exception e) {
        throw new RuntimeException(e);
    } finally {
        release(conn, pstmt, rs);
    }
}

/**
 * 将结果集中的一条数据封装到实体对象中
 * @param rs
 * @param domainClass
 * @param <E>
 * @return
 * @throws Exception
 */
private <E> E extraction(ResultSet rs, Class domainClass) throws Exception{
    //实例化要封装的实体类对象
    E obj = (E) domainClass.newInstance();
    //取出结果集的元信息: ResultSetMetaData
    ResultSetMetaData rsmd = rs.getMetaData();
    //取出总列数
    int columnCount = rsmd.getColumnCount();
    //遍历总列数
    for (int i = 1; i <= columnCount; i++) {
        //获取每列的名称, 列名的序号是从1开始的
        String columnName = rsmd.getColumnName(i);
        //根据得到列名, 获取每列的值
        Object columnValue = rs.getObject(columnName);
        //给obj赋值: 使用Java内省机制 (借助PropertyDescriptor实现属性的封装)
        PropertyDescriptor pd = new PropertyDescriptor(columnName,
domainClass); //要求: 实体类的属性和数据库表的列名保持一种
        //获取它的写入方法
        Method writeMethod = pd.getWriteMethod();
        //把获取的列的值, 给对象赋值
        writeMethod.invoke(obj, columnValue);
    }

    return obj;
}

private void release(Connection conn, PreparedStatement pstmt, ResultSet rs)
{
    if (rs != null) {
        try {
            rs.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

```

    }

    if (pstmt != null) {
        try {
            pstmt.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    if (conn != null) {
        try {
            conn.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
}

```

6) 编写获取数据源的工具类 JdbcUtil

```

/**
 * 用于创建数据源的工具类
 */
public class JdbcUtil {

    /**
     * 用于获取一个连接
     * @param cfg
     * @return
     */
    public static Connection getConnection(Configuration cfg){
        try {
            Class.forName(cfg.getDriver());
            return DriverManager.getConnection(cfg.getUrl(), cfg.getUsername(),
            cfg.getPassword());
        } catch (Exception e){
            throw new RuntimeException(e);
        }
    }
}

```

7) 定义SqlSession的接口及实现类

接口 SqlSession.java

```
public interface SqlSession {

    /**
     * 获取mapper代理对象
     * @param clzz
     * @param <T>
     * @return
     */
    <T> T getMapper(Class<T> clzz) ;

    /**
     * 释放资源
     */
    void close();

}
```

实现类 DefaultSqlSession.java

```
public class DefaultSqlSession implements SqlSession {

    private Connection connection;
    private Map<String, Mapper> mappers;

    /**
     * 创建SqlSession对象时,初始化执行器对象
     * @param mappers
     * @param connection
     */
    public DefaultSqlSession(Map<String, Mapper> mappers, Connection connection)
    {
        this.mappers = mappers ;
        this.connection = connection ;
    }

    /**
     * 获取持久层对象的代理对象
     *
     * @param clzz
     * @param <T>
     * @return
     */
}
```

```

    */
    public <T> T getMapper(Class<T> clzz) {
        return (T) Proxy.newProxyInstance(clzz.getClassLoader(), new Class[]
{clzz}, new MapperProxyFactory(connection,mappers));
    }

    @Override
    public void close() {
        if(connection!=null){
            try {
                connection.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}

```

8) 编写动态代理的处理类 MapperProxyFactory.java

```

/**
 * 创建代理对象的InvocationHandler的实现类
 */
public class MapperProxyFactory implements InvocationHandler {

    private Connection connection;
    private Map<String, Mapper> mappers;

    public MapperProxyFactory(Connection connection, Map<String, Mapper>
mappers) {
        this.connection = connection ;
        this.mappers = mappers ;
    }

    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        //1. 获取需要执行的sql语句
        //1.1 获取执行的方法名称----需要根据方法名称找到对应的需要执行的sql语句
        String methodName = method.getName();
        //1.2 获取方法所在类的全路径名称
        String className = method.getDeclaringClass().getName();
        //1.3 获取需要执行的statement
        String statment = className+"."+methodName;
        //1.4 获取返回值类型
        Class<?> type = method.getReturnType();
    }
}

```

```

//1.5 根据查询结果调用查询方法
if(type == List.class){
    return new Executor().selectList(mappers.get(statement),connection);
}
if(type != List.class){
    return new Executor().selectOne(mappers.get(statement),connection);
}

return method.invoke(new Object(),args) ;
}
}

```

9) 创建SqlSessionFactory的接口及实现类

接口 SqlSessionFactory.java

```

/**
 * SqlSessionFactory
 */
public interface SqlSessionFactory {

    /**
     * 创建session
     * @return
     */
    public SqlSession openSession();
}

```

实现类 DefaultSqlSessionFactory.java

```

/**
 * SqlSessionFactory的实现类
 */
public class DefaultSqlSessionFactory implements SqlSessionFactory {

    /**
     * 配置文件的输入流
     */
    private Configuration configuration;

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

    /**
     * 创建sqlSession
     * @return
     */
}

```

```

    public SqlSession openSession() {
        DefaultSqlSession sqlSession = new
DefaultSqlSession(configuration.getMappers(),
jdbcutil.getConnection(configuration));
        return sqlSession;
    }
}

```

10) 定义SqlSessionFactory的构建者类

```

/**
 * sqlSession工厂的构建者
 */
public class SqlSessionFactoryBuilder {

    /**
     * 构建session工厂
     * @param is
     * @return
     */
    public SqlSessionFactory build(InputStream is){
        Configuration configuration = XMLConfigBuilder.loadConfiguration(is);
        DefaultSqlSessionFactory factory = new
DefaultSqlSessionFactory(configuration);
        return factory;
    }
}

```

11) 把自定义的mybatis打成jar: install

12) 测试自定义框架

创建maven测试工程

引入依赖

打成jar文件的自定义MyBatis框架

mysql驱动

junit测试

引入配置文件:自定义mybatis一样,将resource中的文件复制一份

创建User实体类

需求:

之前导入mybatis依赖能运行的项目, 去掉mybatis, 引入我们自定义的这个jar的依赖, 如果能够成功运行, 就代表自定义框架是OK的!

测试代码:

```
/**
 * 测试代码
 */
public static void main(String[] args) throws Exception {
    //1.读取配置文件
    InputStream in = Resources.getResourceAsStream("SqlMapConfig.xml");
    //2.创建SqlSessionFactory工厂
    SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
    SqlSessionFactory factory = builder.build(in);
    //3.使用工厂生产SqlSession对象
    SqlSession session = factory.openSession();
    //4.使用SqlSession创建Dao接口的代理对象
    IUserDao userDao = session.getMapper(IUserDao.class);
    //5.使用代理对象执行方法
    List<User> users = userDao.findAll();
    for(User user : users){
        System.out.println(user);
    }
    //6.释放资源
    session.close();
    in.close();
}
```

7.5 补充: 注解方式实现自定义MyBatis框架

1) 修改xml解析类 XMLConfigBuilder:

添加注解解析逻辑:

```
/**
 * 4. 加载mapper
 * 添加解析注解方式mapper代码
 */
```

```

public static void loadMapperElement(Element root) throws Exception {
    //取出mappers中的所有mapper标签, 判断他们使用了resource还是class属性
    List<Element> mapperElements = root.selectNodes("//mappers/mapper");
    //遍历集合
    for(Element mapperElement : mapperElements){
        //判断mapperElement使用的是哪个属性
        Attribute attribute = mapperElement.attribute("resource");
        if(attribute != null){
            System.out.println("-----XML方式mapper-----");
            //表示有resource属性, 用的是XML
            //取出属性的值
            String mapperPath = attribute.getValue();//获取属性的
值"com/itheima/dao/IUserDao.xml"
            //把映射配置文件的内容获取出来, 封装成一个map
            Map<String,Mapper> mappers =
loadMapperConfiguration(mapperPath);
            //给configuration中的mappers赋值
            cfg.setMappers(mappers);

        }else{
            // *****添加解析注解方式mapper代码*****
            System.out.println("-----注解方式mapper-----");
            //表示没有resource属性, 用的是注解
            //获取class属性的值
            String daoClassPath = mapperElement.attributeValue("class");
            //根据daoClassPath获取封装的必要信息
            Map<String,Mapper> mappers =
MapperAnnotationBuilder.loadMapperAnnotation(daoClassPath);
            //给configuration中的mappers赋值
            cfg.setMappers(mappers);
        }
    }
}

```

2) 编写mapper注解解析 MapperAnnotationBuilder

```

/**
 * 用于解析注解方式的mapper:
 *     xxxMapper.java
 */
public class MapperAnnotationBuilder {

    /**
     * 根据传入的参数, 得到dao中所有被select注解标注的方法。

```

```

* 根据方法名称和类名，以及方法上注解value属性的值，组成Mapper的必要信息
* @param daoClassPath
* @return
*/
public static Map<String,Mapper> loadMapperAnnotation(String
daoClassPath)throws Exception{
    //定义返回值对象
    Map<String,Mapper> mappers = new HashMap<String, Mapper>();

    //1.得到dao接口的字节码对象
    Class daoClass = Class.forName(daoClassPath);
    //2.得到dao接口中的方法数组
    Method[] methods = daoClass.getMethods();
    //3.遍历Method数组
    for(Method method : methods){
        //取出每一个方法，判断是否有select注解
        boolean isAnnotated = method.isAnnotationPresent(Select.class);
        if(isAnnotated){
            //创建Mapper对象
            Mapper mapper = new Mapper();
            //取出注解的value属性值
            Select selectAnno = method.getAnnotation(Select.class);
            String queryString = selectAnno.value();
            mapper.setQueryString(queryString);
            //获取当前方法的返回值，还要求必须带有泛型信息
            Type type = method.getGenericReturnType();//List<User>
            //判断type是不是参数化的类型
            if(type instanceof ParameterizedType){
                //强转
                ParameterizedType ptype = (ParameterizedType)type;
                //得到参数化类型中的实际类型参数
                Type[] types = ptype.getActualTypeArguments();
                //取出第一个
                Class domainClass = (Class)types[0];
                //获取domainClass的类名
                String resultType = domainClass.getName();
                //给Mapper赋值
                mapper.setResultType(resultType);
            }
            //组装key的信息
            //获取方法的名称
            String methodName = method.getName();
            String className = method.getDeclaringClass().getName();
            String key = className+"."+methodName;
            //给map赋值
            mappers.put(key,mapper);
        }
    }
}

```

```
        return mappers;
    }
}
```

3) 自定义@Select 注解

```
/**
 * 查询的注解
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Select {

    /**
     * 配置SQL语句的
     * @return
     */
    String value();
}
```

4) 测试:

- 修改持久层接口

```
/**
 * 用户的持久层接口
 */
public interface IUserDao {

    /**
     * 查询所有操作
     * @return
     */
    @Select("select * from user")
    List<User> findAll();
}
```

- 修改 SqlMapConfig.xml

```
<!-- 指定映射配置文件的位置，映射配置文件指的是每个dao独立的配置文件 -->
<mappers>
    <!--<mapper resource="com/itheima/dao/IUserDao.xml"/>-->
    <mapper class="com.itheima.dao.IUserDao"/>
</mappers>
```

- 测试代码：

```
/**
 * 测试代码
 */
public static void main(String[] args) throws Exception {
    //1.读取配置文件
    InputStream in = Resources.getResourceAsStream("SqlMapConfig.xml");
    //2.创建SqlSessionFactory工厂
    SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
    SqlSessionFactory factory = builder.build(in);
    //3.使用工厂生产SqlSession对象
    SqlSession session = factory.openSession();
    //4.使用SqlSession创建Dao接口的代理对象
    IUserDao userDao = session.getMapper(IUserDao.class);
    //5.使用代理对象执行方法
    List<User> users = userDao.findAll();
    for(User user : users){
        System.out.println(user);
    }
    //6.释放资源
    session.close();
    in.close();
}
```

